University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Michał Zawalski

Solving Complex Decision-Making Problems with Structured Reasoning and Planning

PhD thesis
in Computer Science

Supervisor:
dr hab. Piotr Miłoś
Institute of Mathematics Polish Academy of Sciences

Warsaw, July 2025

Author's Declaration:
I hereby declare that this dissertation is my own work.

Date: ________________ **Signature:** ____________________________

mgr Michał Zawalski

Supervisor's Declaration:
The dissertation is ready to be reviewed.

Date: ________________ **Signature:** ____________________________

dr hab. Piotr Miłoś

Abstract

Reinforcement Learning (RL) has achieved remarkable success in Artificial Intelligence (AI), powering breakthroughs in robotics, gaming, and real-world decision-making. However, despite these advancements, RL continues to struggle with fundamental challenges such as low sample efficiency, limited generalization, and poor interpretability. In contrast, humans excel at robustly solving complex problems by recognizing abstract structures and reasoning through them efficiently. Inspired by these capabilities, my research explores ways to address the core limitations of RL by reflecting structured reasoning and planning. I focus on improving generalization and interpretability of trained policies by developing models that explicitly reason about solutions, going beyond brittle pattern recognition. Additionally, I design methods that enable RL agents to operate on high-level concepts, facilitating structured decision-making and more efficient long-horizon reasoning.

The challenge of policy generalization is particularly pronounced in the field of robotics. Therefore, in that domain, we show how to teach robotic policies to think critically and reason through tasks with our Embodied Chain-of-Thought (ECoT) approach. Policies trained with ECoT not only learn more efficiently, but, more importantly, generalize beyond the training distribution by abstracting task structures from complex visual observations. By explicitly reasoning through tasks, these policies become more interpretable and easier to control.

Recognizing the abstract structure of a problem is of little value without efficient tools to control it. Inspired by how humans operate on high-level concepts, we propose Subgoal Search (kSubS), a novel hierarchical search algorithm that effectively solves complex reasoning tasks. The human ability to adapt to the complexity of the problem at hand inspired us to further develop Adaptive Subgoal Search (AdaSubS) that adapts to the complexity of the problem by dynamically generating subgoals. Our extensive experimental and theoretical analysis highlights unique advantages of subgoal-based methods, including the ability to transform multimodal training signal into a learning advantage and robustness to low-quality data.

My research takes a step toward bridging the gap between natural intelligence and autonomous learning systems by drawing inspiration from human-like reasoning. On the practical side, it also provides state-of-the-art solvers for challenging domains, advancing both the theoretical foundations and the applicability of RL.

Keywords

Reinforcement Learning, Imitation Learning, Hierarchical Search, Robotics, Planning, Chain-of-Thought, Explainable AI

Tytuł pracy w języku polskim

Rozwiązywanie Złożonych Problemów Decyzyjnych Poprzez Strukturalne Rozumowanie i Planowanie

Streszczenie

Uczenie przez wzmocnienie (Reinforcement Learning, RL) osiągnęło imponujące sukcesy w dziedzinie sztucznej inteligencji, umożliwiając przełomy w robotyce, grach komputerowych oraz podejmowaniu decyzji w rzeczywistych scenariuszach. Pomimo tych osiągnięć, RL nadal zmaga się z podstawowymi wyzwaniami, takimi jak niska efektywność treningu, ograniczona zdolność uogólniania oraz słaba interpretowalność. Tymczasem ludzie potrafią skutecznie rozwiązywać złożone problemy, rozpoznając abstrakcyjne struktury i sprawnie rozumując za ich pomocą.

Zainspirowany tymi ludzkimi zdolnościami, w swojej pracy badawczej poszukuję sposobów na przezwyciężenie kluczowych ograniczeń RL poprzez odzwierciedlenie strukturalnego rozumowania i planowania. Skupiam się na poprawie uogólniania i interpretowalności wytrenowanych polityk, rozwijając modele, które w sposób jawny rozumują nad rozwiązaniami, wychodząc poza niestabilne rozpoznawanie wzorców. Projektuję także metody, które umożliwiają agentom RL operowanie na wysokopoziomowych krokach, wspierając strukturalne podejmowanie decyzji oraz efektywne rozumowanie w długim horyzoncie czasowym.

Problem uogólniania polityki jest szczególnie widoczny w robotyce. Dlatego w tej dziedzinie pokazujemy, jak nauczyć roboty krytycznego myślenia i rozumowania przy użyciu podejścia Embodied Chain-of-Thought (ECoT). Polityki trenowane z ECoT nie tylko uczą się wydajniej, ale co ważniejsze – uogólniają poza rozkład danych treningowych, dostrzegając strukturę zadania w złożonych obserwacjach wizualnych. Poprzez jawne rozumowanie nad zadaniami, polityki te stają się bardziej interpretowalne i łatwiejsze do kontrolowania.

Samo rozpoznanie abstrakcyjnej struktury problemu jest mało użyteczne bez skutecznych narzędzi do jej kontrolowania. Inspirując się tym, jak ludzie operują na pojęciach wysokiego poziomu, proponujemy Subgoal Search (kSubS) – nowy hierarchiczny algorytm przeszukiwania, który skutecznie rozwiązuje złożone zadania wymagające rozumowania. Zdolność człowieka do adaptacji względem złożoności danego problemu zainspirowała nas do opracowania Adaptive Subgoal Search (AdaSubS) – adaptacyjnej wersji, która dynamicznie generuje podcele w zależności od złożoności zadania. Nasza szeroko zakrojona analiza eksperymentalna i teoretyczna ukazuje unikalne zalety metod opartych na podcelach, w tym zdolność do wykorzystywania niejednoznacznego sygnału treningowego oraz odporność na dane niskiej jakości.

Moje badania stanowią krok w kierunku zbliżenia naturalnej inteligencji i autonomicznych systemów uczących się, czerpiąc inspirację z ludzkiego rozumowania. W praktyce, dostarczają także najnowocześniejszych rozwiązań dla wymagających dziedzin, poszerzając zarówno podstawy teoretyczne, jak i możliwości stosowania RL.

Contents

List of publications 7

1. Introduction 11

2. Background 15
- 2.1. Reinforcement Learning 15
- 2.2. Planning 16
- 2.3. Robotics 17

3. Subgoal Search 19
- 3.1. Overview 19
- 3.2. Components of Adaptive Subgoal Search 19
- 3.3. Experiments 20
- 3.4. Adaptive planners 21
- 3.5. What matters in subgoal search? 23
  - 3.5.1. Subgoal methods are robust to diverse sources of data 23
  - 3.5.2. Subgoal methods are value noise filters 24
- 3.6. Summary 25

4. Embodied Chain-of-Thought Reasoning 27
- 4.1. Overview 27
- 4.2. Reasoning steps 27
- 4.3. Experiments 29
  - 4.3.1. Performance 29
  - 4.3.2. Diagnosing Policy Failures Through Inspecting Reasoning Chains . 29
  - 4.3.3. Does ECoT capability transfer to other robots? 30
- 4.4. Summary 30

5. Conclusions 33
- 5.1. Future directions 33
- 5.2. Acknowledgments 34

Bibliography 35

A. Complete publications 47

List of publications

P1 **Off-Policy Correction For Multi-Agent Reinforcement Learning**
Authors: Michał Zawalski, Błażej Osiński, Henryk Michalewski, Piotr Miłoś
Published at: Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems (CORE A*, ministry points: 200, cited: 3)
My contributions:

- **Design & Methodology Development:** Participated in the development of the experimental methodology, including the choice of evaluation and ablation study setups.
- **Literature Review:** Conducted an in-depth review of existing multi-agent learning methods and their applicability to the setting under consideration.
- **Experimental Implementation:** Led the implementation of experiments, hyperparameter tuning, and various design choice explorations.
- **Results Analysis & Interpretation:** Analyzed results, identified key findings, and provided insights into the algorithm's scalability.
- **Manuscript Preparation:** Wrote the majority of the manuscript, including the methods and results sections.
- **Dissemination:** Presented the work through a poster session at the AAMAS conference.

Contribution Estimate: 60%

P2 **Subgoal Search for Complex Reasoning Tasks**
Authors: Konrad Czechowski*, Tomasz Odrzygóźdź*, Marek Zbysiński, Michał Zawalski, Krzysztof Olejnik, Yuhuai Wu, Łukasz Kuciński, Piotr Miłoś
Published at: Advances in Neural Information Processing Systems 2021 (CORE A*, ministry points: 200, cited: 29)
My contributions:

- **Research Questions & Conceptualization:** Contributed to identifying the Rubik's Cube as a case study for evaluating Subgoal Search in combinatorial reasoning tasks.
- **Experimental Implementation:** Implemented the subgoal search algorithm on the Rubik's Cube, trained the models, and tuned hyperparameters.

- **Design & Methodology Development:** Explored key design choices and conducted ablation studies to understand their impact on performance.
- **Results Analysis & Interpretation:** Analyzed experimental results and summarized findings related to the Rubik's Cube in the manuscript.

Contribution Estimate: 15%

P3 **Fast and Precise: Adjusting Planning Horizon with Adaptive Subgoal Search**
Authors: Michał Zawalski*, Michał Tyrolski*, Konrad Czechowski*, Damian Stachura, Piotr Piękos, Tomasz Odrzygóźdź, Yuhuai Wu, Łukasz Kuciński, Piotr Miłoś
Published at: International Conference on Learning Representations 2023 as Notable Top 5%, (CORE A*, ministry points: 200, cited: 11)
My contributions:

- **Research Questions & Conceptualization:** Formulated the primary research question on how to adapt planning horizons dynamically in subgoal search algorithms.
- **Algorithm Design & Development:** Invented the core adaptive subgoal search algorithm and experimented with alternative adaptive techniques.
- **Experimental Implementation:** Led the design and execution of most experiments, also including parameter tuning and ablation studies.
- **Results Analysis & Interpretation:** Analyzed experimental outcomes and formulated conclusions.
- **Manuscript Preparation:** Wrote the majority of the manuscript, including the methods and results sections.
- **Dissemination:** Delivered both oral and poster presentations at ICLR.

Contribution Estimate: 50%

P4 **Robotic Control via Embodied Chain-of-Thought Reasoning**
Authors: Michał Zawalski*, William Chen*, Karl Pertsch, Oier Mees, Chelsea Finn, Sergey Levine
Published at: Conference on Robot Learning 2024 (unranked[1], Impact Factor: 4.79, cited: 94)
My contributions:

- **Research Questions & Conceptualization:** Co-developed the core idea of using embodied chain-of-thought reasoning for robotic control.
- **Experimental Implementation:** Built the proof-of-concept experiments and the pipelines for extracting embodied features.

[1]Although CoRL and TMLR are not yet ranked by popular services due to their relatively short history, they are widely recognized as highly influential, as evidenced by the numerous top researchers who choose to publish there.

- **Design & Methodology Development:** Designed and refined the experimental framework, including key design choices.
- **Model Training & Evaluation:** Managed model training and conducted approximately half of the robotic evaluations.
- **Results Analysis & Interpretation:** Interpreted experimental outcomes and provided insights into the method's effectiveness.
- **Manuscript Preparation:** Contributed to the manuscript writing, particularly in the methods and results sections.

Contribution Estimate: 40%

P5 **What Matters in Hierarchical Search for Combinatorial Reasoning Problems?**
Authors: Michał Zawalski, Gracjan Góral, Michał Tyrolski, Emilia Wiśnios, Franciszek Budrowski, Marek Cygan, Łukasz Kuciński, Piotr Miłoś
Submitted to: Transactions on Machine Learning Research (unranked[1], Impact Factor: 4.09)
My contributions:

- **Leadership & Coordination:** Leaded the research efforts and coordinated collaboration among team members.
- **Research Questions & Conceptualization:** Contributed to identifying key properties affecting hierarchical search performance.
- **Theoretical Foundations:** Worked on the theoretical aspects supporting the main claims of the paper.
- **Experimental Implementation:** Designed and executed experiments to evaluate the influence of identified properties.
- **Results Analysis & Interpretation:** Analyzed experimental results and formulated conclusions.
- **Manuscript Preparation:** Wrote the majority of the manuscript, including both theoretical and experimental findings.

Contribution Estimate: 40%

Chapter 1

Introduction

Imagine a chess grandmaster planning ten moves ahead, a child learning to ride a bicycle after a few falls, or a cook adjusting recipe ingredients based on taste – examples of humans learning through experience and reasoning about consequences. For decades, artificial intelligence researchers have pursued systems capable of matching this natural, adaptive intelligence. Early AI systems operated through rigid, predefined instructions and logical rules [Newell and Simon, 1956, McCarthy, 1959], creating brittle agents that failed when confronted with novel situations. Recently, deep learning has revolutionized how machines perceive and interact with the world. Drawing inspiration from neural structures, these systems process information through layered networks that extract increasingly abstract patterns from raw data, enabling breakthroughs in visual perception [Krizhevsky et al., 2012, He et al., 2016], image and video generation [Goodfellow et al., 2014, Ho et al., 2020, Liu et al., 2024], language modeling [Vaswani et al., 2017, Devlin et al., 2019, Brown et al., 2020, OpenAI, 2023], and complex decision-making [Mnih et al., 2015, Silver et al., 2018b, Berner et al., 2019, Hafner et al., 2023] that seemed impossible just a decade ago.

Among recent advancements, Reinforcement Learning (RL) has emerged as a particularly powerful paradigm for developing adaptive decision-making systems. The core principle of RL – learning optimal behaviors through trial-and-error interactions with an environment – mirrors how humans acquire skills through experience rather than explicit instruction. This approach has led to remarkable achievements: defeating world champions in chess [Silver et al., 2018a] and Go [Silver et al., 2016b], mastering video games [Mnih et al., 2015, Vinyals et al., 2019], and enabling robots to perform dexterous manipulation [OpenAI et al., 2019, Kim et al., 2024b] and navigate complex environments autonomously [Chiang et al., 2019, Shah et al., 2021, Shah et al., 2022]. Furthermore, RL has begun transforming healthcare treatment protocols [Yu et al., 2023], financial trading strategies [Zhang et al., 2019], and industrial control systems [Schoettler et al., 2020], demonstrating potential to reshape decision-making across a wide variety of domains.

Despite these impressive advances, current RL approaches face fundamental limitations that restrict their broader applicability. Modern systems require millions of trial-and-error interactions to learn even relatively simple tasks, while humans can often master similar challenges with much less experience. Furthermore, RL agents often fail to transfer learned behaviors to unseen tasks or changing environments. Furthermore, decisions made by RL systems frequently lack interpretability, offering little insight into the reasoning process. These challenges motivate the central question driving my research: *Can we overcome the fundamental limitations of RL by drawing inspiration from human reasoning?* While this question encompasses a wide research landscape beyond the scope of any single dissertation, my work explores one promising direction: *How can we develop RL systems that efficiently recognize abstract problem structures and conduct interpretable, structured reasoning?*

My dissertation addresses these question by developing novel frameworks that integrate structured reasoning mechanisms into reinforcement learning. Rather than viewing RL and human-inspired reasoning as separate approaches, I demonstrate that their integration creates systems that combine the flexibility of learning-based methods with the efficiency and generalizability of structured reasoning. In summary, my works make three main contributions to the field:

1. We developed Embodied Chain-of-Thought (ECoT), a novel approach enabling robotic policies to reason explicitly through tasks, improving generalization beyond training distributions and enhancing interpretability.

2. We created Subgoal Search (kSubS), a hierarchical search algorithm operating on high-level concepts that effectively solves even complex reasoning tasks by breaking them into manageable subgoals.

3. We designed Adaptive Subgoal Search (AdaSubS), a dynamic planning algorithm that adapts to the complexity of the problem by controlling the range of subgoals.

Together, these contributions represent concrete steps toward reinforcement learning systems that think, reason, and plan more like humans – a capability that paves the way for the next revolution in artificial intelligence.

Embodied Chain-of-Thought Reasoning In [P4], we introduce Embodied Chain-of-Thought Reasoning (ECoT), a novel approach to training robotic policies that addresses a fundamental challenge in robotics: generalization beyond the training distribution. Traditional behavioral cloning maps raw visual inputs directly to actions, creating a brittle association that often fails when encountering novel scenarios. In contrast, ECoT trains agents to *think before they act* by decomposing decision-making into a structured reasoning process. The agent first analyzes the scene, identifies relevant objects and relationships, formulates a plan, and only then executes appropriate actions. This explicit reasoning allows the agent to internalize not merely the surface-level behaviors but the underlying reasoning processes of experts, enabling more robust and adaptable decision-making.

We build a scalable pipeline that extracts embodied reasoning patterns from existing datasets without requiring additional data collection. Our experimental results demonstrate that ECoT significantly outperforms state-of-the-art methods, particularly in out-of-distribution scenarios, confirming that structured reasoning enhances generalization more effectively than simply scaling data volume. Beyond performance improvements, ECoT advances interpretability and safety – by making the agent's decision process transparent, it allows human operators to understand why an agent chose particular actions and even guide its reasoning through natural language interventions.

Subgoal Search Unlike classical search algorithms that typically operate at a low level of abstraction, humans plan by identifying and reasoning over high-level steps that emphasize key decision points. Inspired by this cognitive strategy, in [P2] we introduce Subgoal Search (kSubS), a planning framework that leverages *subgoals* – high-level steps that span multiple actions at once. By guiding the search process through these subgoals, kSubS enables deeper and more structured planning, even under constrained computational resources. Building on this idea, we developed Adaptive Subgoal Search (AdaSubS) [P3], which improves the flexibility of the Subgoal Search framework by dynamically adjusting the subgoal distances based on environmental complexity. In challenging areas requiring detailed reasoning, AdaSubS generates short subgoals for careful navigation. In simpler parts of the problem, it uses distant subgoals to advance faster.

Through extensive experimental evaluation, we demonstrate that both kSubS and AdaSubS successfully solve tasks with high combinatorial complexity. Our analysis reveals several unique advantages of hierarchical search, including robustness to ambiguous demonstrations and resilience against severe value approximation errors [P5]. Additionally, we reveal subtle evaluation pitfalls that can lead to misleading conclusions.

Chapter 2

Background

2.1. Reinforcement Learning

At its core, RL is a framework in which an agent learns by interacting with an environment to maximize cumulative rewards. Formally, this interaction is modeled as a Markov Decision Process (MDP), defined by a state space, action space, transition dynamics, reward function, and discount factor. The agent iteratively explores the environment, updating its policy based on observed rewards and transitions, aiming to optimize long-term outcomes.

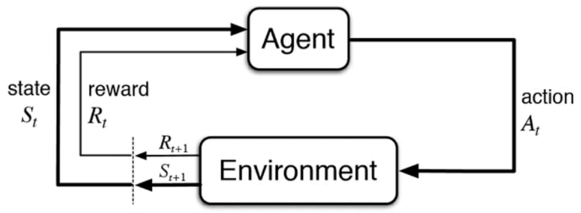


Figure 2.1: The framework of Reinforcement Learning. Iteratively, the agent observes a state of the environment, computes the next action, and executes it, receiving a reward.

Although RL has led to groundbreaking successes, fundamental challenges persist, limiting its broader applicability. Among these challenges, sample efficiency – the ability to learn from a limited number of interactions – remains a critical bottleneck. Traditional RL methods often require vast amounts of trial-and-error learning, making them impractical for real-world applications where data collection is expensive or unsafe. Additionally, RL policies frequently struggle with generalization, failing to adapt to conditions beyond the training distribution. Finally, the lack of interpretability in learned policies hinders trust and deployability in safety-critical environments.

Significant progress has been made in addressing sample efficiency by leveraging offline datasets and imitation learning [Collaboration et al., 2024]. Offline RL enables agents to learn from previously collected data without direct environment interaction, while imitation learning allows agents to mimic expert demonstrations, significantly reducing the need for trial-and-error exploration. These approaches have led to impressive results, such as robotic policies capable of learning complex skills from human demonstrations [Kim et al., 2024b, Walke et al., 2023b, Chi et al., 2023], agents reaching superhuman performance in game playing by imitating experts [Silver et al., 2016a, Berner et al., 2019], or autonomous driving cars [Osiński et al., 2020]. Despite these advancements, efficiently learning structured, generalizable, and interpretable policies remains a key challenge.

2.2. Planning

Planning plays a crucial role in decision-making, enabling agents to anticipate future outcomes before acting [Hamrick et al., 2021]. In RL, planning methods improve efficiency by leveraging models of the environment to simulate future states, reducing the reliance on direct interactions. Compared to model-free approaches, which rely solely on experience, RL with planning accelerates learning and improves generalization by guiding exploration more effectively.

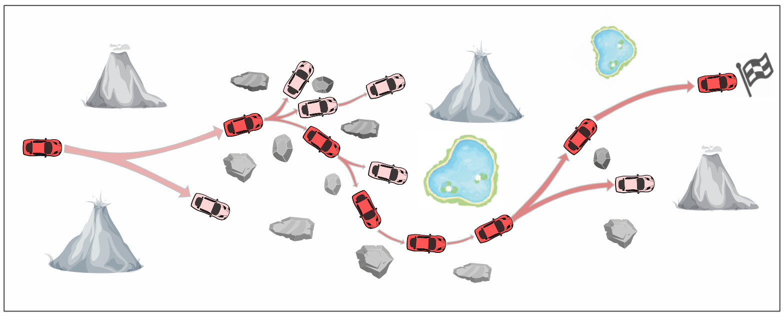


Figure 2.2: An illustrative example of planning. The agent predicts the consequences of its actions before executing them.

Classical planning algorithms that have historically been used to guide decision-making in structured environments, remain essential even in contemporary applications. Greedy BestFS [Cormen et al., 2009] prioritizes the most promising nodes based on heuristic estimates but can be misled by inaccurate heuristics. A* [Russell and Norvig, 2020] improves on this by combining heuristic estimates with actual costs, ensuring optimal solutions when heuristics are admissible. MCTS [Coulom, 2006], a probabilistic approach, balances exploration and exploitation, making it effective for large and complex domains. Recently MCTS gained popularity driven by huge successes of AlphaZero and related algorithms [Silver et al., 2018b, Fawzi et al., 2022, Jumper et al., 2021]. While early AI systems relied on handcrafted heuristics to evaluate search trees, recent advancements have demonstrated that learned heuristics can outperform manually designed ones [Silver et al., 2016a].

2.3. Robotics

One of the most exciting frontiers of RL research lies in robotics, where autonomous agents must interact with the physical world to perform complex tasks. Robotics research spans a wide range of applications, including manipulation [Kim et al., 2024a], locomotion [Li et al., 2025], autonomous navigation [Shah et al., 2022], and human-robot interaction [Christen et al., 2023]. Recent advancements have enabled robots to achieve impressive feats, such as dexterous manipulation and locomotion of humanoids [Radosavovic et al., 2024].

Despite this progress, robotics presents unique challenges that make RL difficult to apply in practice. One major hurdle is data availability, as collecting high-quality training data for real-world robotic systems is costly and time-consuming [Popov et al., 2017, Walke et al., 2023a, Collaboration et al., 2023]. Furthermore, the discrepancy between simulated training environments and real-world deployment, known as the sim-to-real gap, poses significant challenges, often requiring extensive domain adaptation techniques [Peng et al., 2018, Loquercio et al., 2019, Akkaya et al., 2019]. Additionally, robotic systems must generalize across diverse environments and tasks while ensuring safety and interpretability, particularly in human-centric settings.

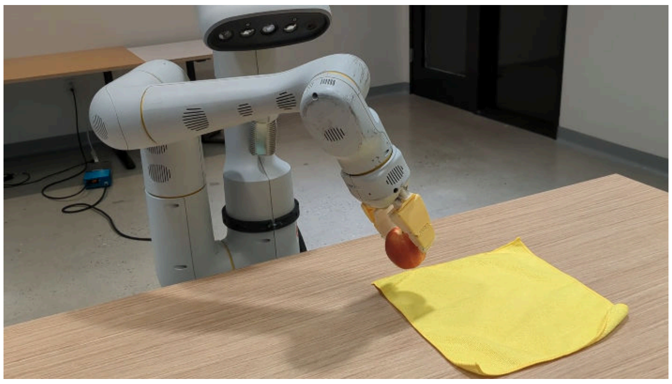


Figure 2.3: Google Robot manipulating an apple.

Given these challenges, there is a strong need for sample-efficient, interpretable, and generalizable RL methods that can enable robots to reason and plan effectively. Hence, in this dissertation, I explore how human-like reasoning and planning can be incorporated to build more capable and efficient robotic policies.

Chapter 3

Subgoal Search

3.1. Overview

Complex problems become more tractable when decomposed into smaller subproblems – a principle central to human reasoning [Hollerman et al., 2000]. Consider how humans navigate: rather than planning every step from start to finish, we identify visible landmarks, reach them sequentially, and progressively work toward our destination. Following this principle, Subgoal Search (kSubS), introduces a structured reasoning framework that mirrors this human strategy by guiding exploration through intermediate objectives [Czechowski et al., 2021]. Unlike classical planning methods that struggle with long-horizon tasks, kSubS enables deeper, more efficient search within computational constraints while reducing approximation errors. This is achieved through a trained subgoal generator that iteratively builds the search tree by identifying promising subgoals at a fixed distance k.

However, many problems contain both difficult bottlenecks requiring precise planning and trivial sections that can be skipped with longer subgoals. For instance, navigating a winding road with a car demands precise, short-term decisions, whereas on a straight highway, planning can extend further ahead. Building on this insight, *Adaptive Subgoal Search* (AdaSubS) improves upon kSubS by introducing a set of subgoal generators trained for different distances [Zawalski et al., 2023]. At each step, the planner dynamically selects the appropriate generator, adjusting the planning horizon based on search progress.

We evaluate kSubS and AdaSubS in Sokoban, Rubik's Cube, and the inequality theorem prover INT [Wu et al., 2021], which are known to be NP-hard [Demaine et al., 2018, Culberson, 1997]. Both methods significantly outperform low-level baselines, with AdaSubS achieving state-of-the-art results on INT.

3.2. Components of Adaptive Subgoal Search

AdaSubS utilizes the following components: *subgoal generators*, *verifier*, *conditional low-level policy* (CLLP), and *value function*. These components are implemented using

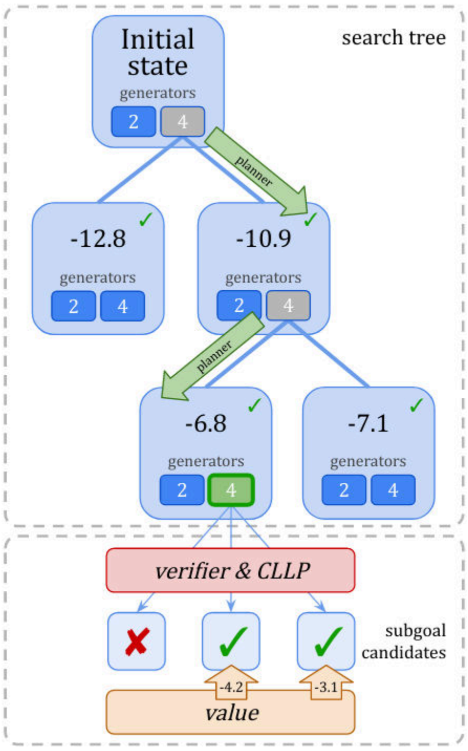


Figure 3.1: An example iteration of the search performed by AdaSubS.

Algorithm 1 Adaptive Subgoal Search

```
Requires:   $C_1$                          max number of nodes
            $V$                            value function network
            $\rho_{k_0}, \ldots, \rho_{k_m}$   subgoal generators
            SOLVED                         predicate of solution

function SOLVE($s_0$)
    $T \leftarrow \emptyset$    ▷ priority queue with lexicographic order
    parents ← {}
    for $k$ in $k_0, \ldots, k_m$ do
        T.PUSH((($k, V(s_0)$), $s_0$))
    seen.ADD($s_0$)                              ▷ seen is a set
    while $0 <$ LEN(T) and LEN(seen) $< C_1$ do
        ($k$, _), s ← T.EXTRACT_MAX()
        subgoals ← $\rho_k$(s)
        for s′ in subgoals do
            if s′ in seen then continue
            if not IS_VALID(s, s′) then
                continue
            seen.ADD(s′)
            parents[s′] ← $s$
            for $k$ in $k_0, \ldots, k_m$ do
                T.PUSH((($k, V(s')$), s′))
            if SOLVED(s′) then
                return LL_PATH($s'$, parents)
                                          ▷ get the low-level path
    return False
```

trained neural networks. To solve a task, AdaSubS iteratively builds a tree of subgoals reachable from the initial state until the target state is reached or the search budget is depleted. In each iteration, it chooses a node in the tree that is subsequently expanded by one of the generators. The chosen generator creates a few subgoal candidates, i.e., states expected to be a few steps closer to the goal than the current node. For each of them, we use the verifier and CLLP to check whether they are valid and reachable within a few steps. For the correct subgoals, we compute their value function, place them in the search tree, and the next iteration follows. The complete pipeline of Adaptive Subgoal Search is summarized by Algorithm 1, and illustrated in Figure 3.1.

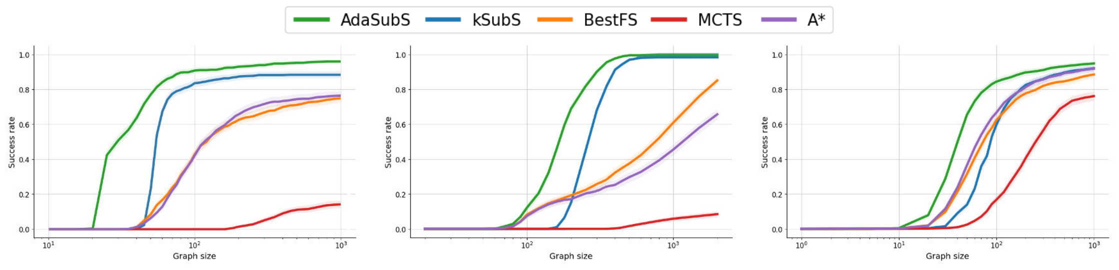
3.3. Experiments

We empirically demonstrate the efficiency of Adaptive Subgoal Search on three complex reasoning domains: Sokoban, Rubik's Cube, and the inequality proving benchmark INT [Wu et al., 2021].

As the performance metric, we use the *success rate*, defined as the fraction of solved problem instances. The computational budget is defined as the *graph size*, i.e., the number of nodes visited during the search and evaluated with a neural network (subgoal generator, value function, verifier, or conditional low-level policy). That includes both the high-level subgoals, and the low-level intermediate states visited by CLLP.

As our baselines, we use classical low-level search algorithms: BestFS, A*, and MCTS. Whenever possible, all tested methods share exactly the same components. In particular, all methods use the same value function network for a fair comparison.

As shown in Figure 3.2, while both subgoal methods outperform classical search baselines, AdaSubS excel in all tested domains. Not only they can solve the hardest instances for which baselines fail, but also they require considerably smaller search budget when solving simpler instances.



(a) Solving INT – inequalities. Components are trained on randomly generated proofs of length 25.

(b) Solving the Rubik's cube. Components are trained on various cube solvers.

(c) Solving Sokoban. Components are trained on trajectories obtained from tree search.

Figure 3.2: Subgoal methods outperform low-level tree search algorithms, usually by a wide margin. The results were measured on a fixed set of 1000 problems for each domain. Shaded areas indicate 95% confidence intervals.

We further tested the generalization properties of subgoal methods in the INT environment. Specifically, we trained the components on proofs of length 15 and evaluated on increasingly longer proofs without any retraining. Figure 3.3 shows that while kSubS can successfully learn to solve the in-distribution instances, its quality decreases when applied to OOD problems. On the other hand, AdaSubS is much more resilient to the distribution shift, as it retains as much as 50% of its initial performance, even when evaluated on twice longer proofs. Hence, the ability of adaptively controlling the planning horizon turns out to be essential in OOD generalization.

3.4. Adaptive planners

There are various ways to implement adaptivity in the framework of Subgoal Search, but not all work equally well. Specifically, we designed and tested the following methods: *MixSubS*, *Iterative mixing*, *Strongest-first*, *Longest-first*. Each method uses a set of n generators $\rho_{k_1}, \ldots, \rho_{k_n}$ trained to produce subgoals on different distances $k_1 < \ldots < k_n$. A detailed description of the methods with pseudocodes can be found in [Zawalski et al., 2023].

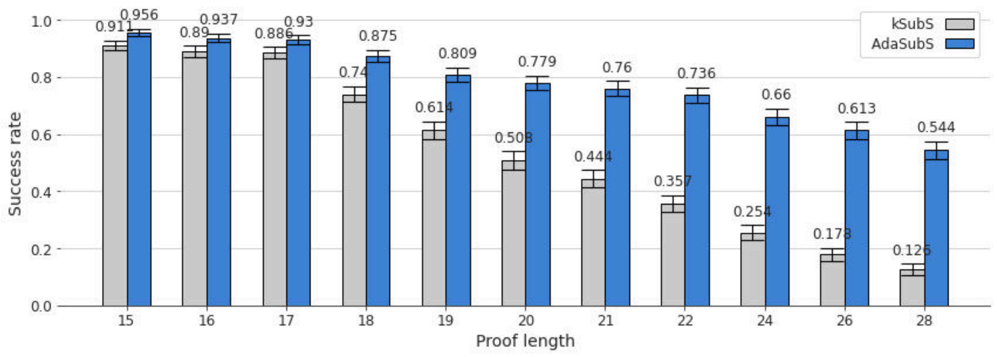


Figure 3.3: Out-of-distribution performance of AdaSubS and kSubS in INT. Both methods were trained on proofs of length 15. Error bars correspond to 95% confidence intervals.

INT		Small budget		Large budget	
		with verifier	without	with verifier	without
kSubS	$k = 4$	2.2%	0.1%	82.4%	83.0%
	$k = 3$	4.0%	0.2%	89.6%	90.7%
	$k = 2$	2.1%	0.5%	89.8%	91.7%
	$k = 1$	0.0%	0.0%	34.7%	46.0%
MixSubS	$k = [4, 3, 2]$	0.0%	0.0%	94.6%	95.0%
	$k = [3, 2, 1]$	0.0%	0.0%	92.2%	92.9%
	$k = [3, 2]$	17.0%	14.8%	92.2%	93.5%
Iterative mixing	iterations $= [1, 1, 1]$	32.0%	30.1%	87.0%	88.6%
	iterations $= [10, 1, 1]$	43.0%	44.8%	95.1%	96.0%
	iterations $= [4, 2, 1]$	54.0%	52.1%	93.6%	95.5%
Strongest-first		39.5%	40.8%	88.5%	89.8%
Longest-first		59.0%	51.5%	95.7%	95.5%

Rubik (with verifier)		
	Small budget	Large budget
kSubS	28.8%	98.6%
MixSubS	49.1%	99.2%
Iterative mixing	50.6%	99.1%
Strongest-first	33.4%	99.0%
Longest-first	58.0%	99.2%

Sokoban (small budget)		
	with verifier	without
kSubS	26.0%	4.7%
MixSubS	52.7%	37.7%
Iterative mixing	64.5%	52.6%
Strongest-first	54.6%	41.9%
Longest-first	72.2%	63.4%

Table 3.1: *(left)* Results for the INT. For each case, unless stated otherwise, the distances of subgoal generators are $k = [3, 2, 1]$. *(right)* Shortened results for Rubik and Sokoban; for complete results and configurations, see [Zawalski et al., 2023]. The results were obtained on 1000 problems each, which yields $\pm 3\%$ Bernoulli 95% confidence intervals.

Already the simple *MixSubS* works better than the non-adaptive baselines. In particular, it can outperform the maximum of performances of kSubS for each k. *Iterative mixing* is able to exhibit strong performance; however, it needs tedious schedule tuning for each domain. *Strongest-first* and *Longest-first* implement bias towards longer distances. When *Strongest-first* encounters an area with overoptimistic value estimates, it wastes a lot of compute to examine it with all subgoal distances. On the other hand, *Longest-first* first explores other areas before using shorter subgoals and thus is able to avoid this problem. The observed effects occur robustly across our test scenarios.

3.5. What matters in subgoal search?

The results discussed above demonstrate the effectiveness of the subgoal planning. In [Zawalski et al., 2024b], we further established a solid foundation for the domain of hierarchical search and combinatorial reasoning by extensively investigating how the choice of environments and training data impacts the performance of subgoal-based methods.

We identified key attributes that contribute to the advantages of hierarchical search: *high value approximation errors*, *training data collected from diverse sources*, *complex action spaces*, and *the presence of dead ends in the environment.* Furthermore, we proposed a consistent evaluation methodology to facilitate meaningful comparisons between methods. Our study provides a deeper understanding of when subgoal methods should be favored over low-level approaches.

Below we highlight selected key findings. Extended discussion of all the properties can be found in [Zawalski et al., 2024b].

3.5.1. Subgoal methods are robust to diverse sources of data

Achieving superhuman performance in complex tasks often involves large-scale datasets of demonstrations obtained from agents with varying skill levels and strategies [Silver et al., 2016a]. By training models on data collected from a variety of solvers and testing them in the Rubik's Cube and N-Puzzle environments, we show that the variability in training data has a significant impact on the performance of search algorithms.

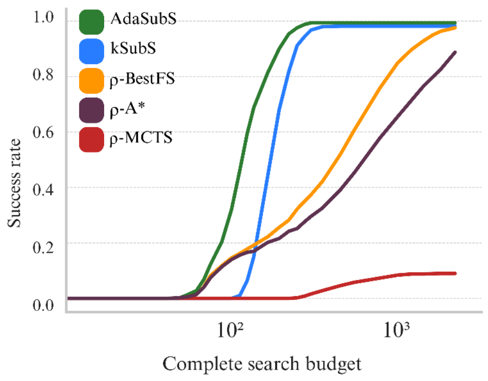


Figure 3.4: Solving the Rubik's Cube. Components are trained on data from 4 different solvers.

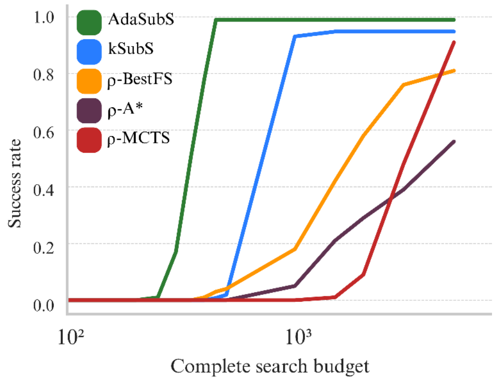


Figure 3.5: Solving the N-Puzzle. Components are trained on data from 2 different solvers.

As shown in Figures 3.4-3.5, subgoal methods consistently outperform low-level methods by a wide margin. However, when the training dataset is limited to a single source of demonstrations – whether the demonstrations are long and structured or short and direct – this performance gap disappears (see Figures 3.6-3.8). Notably, subgoal methods, particularly AdaSubS, maintain stable performance across all training setups, while low-level methods are highly sensitive to the characteristics of the training data.

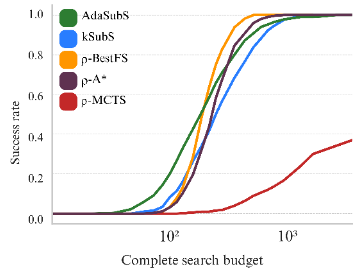


Figure 3.6: Solving the Rubik's Cube. Components are trained on reversed random shuffles.

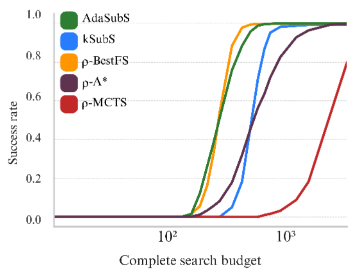


Figure 3.7: Solving the Rubik's Cube. Components are trained on the *Beginner* algorithmic solver.

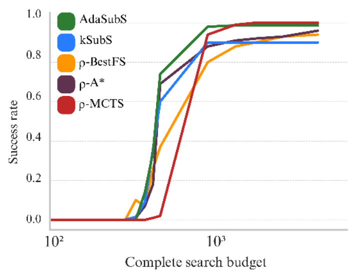


Figure 3.8: Solving N-Puzzle. Components are trained on an algorithmic solver.

To explain those results, we found that value functions trained on diverse data often fail to assign consistently low values to the initial states of tasks. When demonstrations differ significantly in their length or execution style, the value function learns this variation, leading to inconsistent value predictions. Hierarchical methods can overcome this issue by relying on subgoals. Subgoals enable the agent to make long steps toward the solution, effectively bypassing regions of the state space where the value function is inconsistent or noisy, as it does not need to assess every small step along the way (this property is further studied in Section 3.5.2). In contrast, low-level methods operate on a finer, step-by-step level, executing small, atomic actions. This makes them more sensitive to the variability in the value function because they must evaluate each intermediate state on the way.

3.5.2. Subgoal methods are value noise filters

We found that the classical search algorithms are highly sensitive to the quality of the value function. To show that in a controlled setting, we added Gaussian noise to the value estimates and observed how different noise levels impacted the success rate of solving tasks.

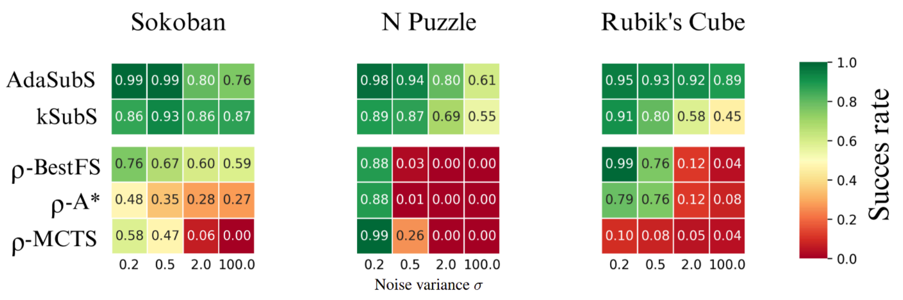


Figure 3.9: Success rate of low-level and subgoal methods as the approximation errors of the value function increase. $\sigma = 100$ results in completely random value estimates.

While BestFS is able to solve nearly all instances under ideal conditions, its performance significantly declines as value function errors increase, even to 0% (see Figure 3.9). A* and MCTS behave similarly. In contrast, the subgoal methods show remarkable resilience. Particularly AdaSubS, which maintains nearly unchanged success rate, despite high value errors.

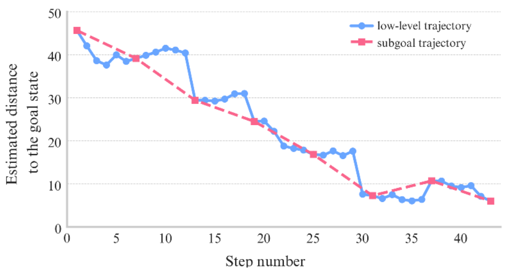


Figure 3.10: Value estimates along a solving trajectory generated by BestFS. Even small approximation errors cause non-decreasing values, slowing down the search. In contrast, the subgoal path mitigates these errors, leading to mostly monotonic values along the trajectory.

These results align with our findings in Section 3.5.1, where using diverse training data naturally introduced value estimation errors. As observed in [Zawalski et al., 2023], the search process of subgoal methods is guided by subgoal generators, which reduces reliance on the value function. Subgoal generators and the conditional policies connecting subgoals are not directly influenced by the value approximation errors. The value function is used only in high-level nodes, which represent only a fraction of the search tree.

3.6. Summary

Subgoal Search and Adaptive Subgoal Search offer a new perspective on planning. By decomposing the tasks into smaller subproblems, they outperform classical search algorithms in complex reasoning tasks while maintaining simplicity. By adapting the planning horizon to the complexity of the problem, AdaSubS achieved state-of-the-art results on INT and scales favorably to OOD instances. Additionally, by an extensive experimental validation we identified the key properties of the environments that contribute to the advantage of subgoal methods. We believe that those contributions will streamline further research in this growing area.

Chapter 4

Embodied Chain-of-Thought Reasoning

4.1. Overview

Vision-language-action (VLA) models—fine-tuned vision-language models (VLMs) adapted to robotic control—have emerged as a powerful way to leverage the broad knowledge encoded in foundation models trained on Internet-scale data [Bommasani et al., 2022]. While these models typically learn direct mappings from observations to actions, bypassing explicit reasoning, they have achieved state-of-the-art results across diverse tasks and robot embodiments [Brohan et al., 2023,Embodiment Collaboration et al., 2024,Kim et al., 2024a]. However, recent advances in chain-of-thought (CoT) prompting [Wei et al., 2023] have shown that explicit reasoning steps can significantly enhance performance on complex tasks, and are now standard in language modeling.

Inspired by this, we introduce Embodied Chain-of-Thought Reasoning (ECoT) for VLA policies [Zawalski et al., 2024a]. Unlike conventional VLAs, ECoT agents perform structured, multi-step textual reasoning before selecting each action. Unlike standard CoT approaches in language models, ECoT interleaves abstract subgoal reasoning with grounded, multimodal perception, mirroring how humans combine high-level planning with contextual understanding. To enable the relatively weak LLM backbones of open-source VLAs to perform such reasoning effectively, we design a scalable pipeline for synthetically generating embodied CoT training data for large robot datasets.

4.2. Reasoning steps

Our goals when designing the steps of our embodied chain-of-thought reasoning chains are twofold: encourage the model to (A) reason through the required high-level steps of the task at hand and determine which step needs to be executed next, and (B) increasingly ground this reasoning in lower-level features of the scene and robot state before predicting the robot action.

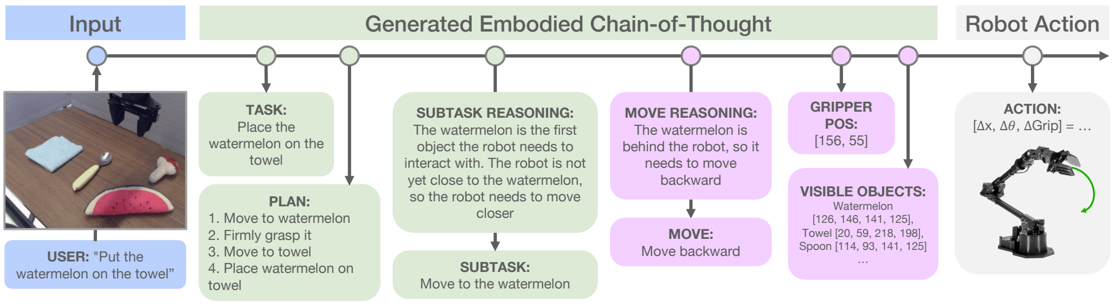

Input
Generated Embodied Chain-of-Thought
Robot Action
USER: "Put the watermelon on the towel"
TASK: Place the watermelon on the towel
PLAN: 1. Move to watermelon 2. Firmly grasp it 3. Move to towel 4. Place watermelon on towel
SUBTASK REASONING: The watermelon is the first object the robot needs to interact with. The robot is not yet close to the watermelon, so the robot needs to move closer
SUBTASK: Move to the watermelon
MOVE REASONING: The watermelon is behind the robot, so it needs to move backward
MOVE: Move backward
GRIPPER POS: [156, 55]
VISIBLE OBJECTS: Watermelon [126, 146, 141, 125], Towel [20, 59, 218, 198], Spoon [114, 93, 141, 125] ...
ACTION: [Δx, Δθ, ΔGrip] = ...

Figure 4.1: Steps of our embodied chain-of-thought reasoning. We interleave several intermediate reasoning steps into the mapping from inputs to robot actions. **Green**: "standard" linguistic chain-of-thought steps that break a given instruction into the required sub-tasks. **Purple**: *Embodied* chain-of-thought steps that require grounding the policy's reasoning in the scene and robot state. Our experiments show that these grounded reasoning steps are key to improving policy performance with chain-of-thought reasoning.

We visualize the ECoT reasoning steps that we train the VLA to perform for an example task in Figure 4.1. From left to right, the model is trained to first rephrase the task instruction (**TASK**) and predict a high-level plan of steps for achieving the instructed task (**PLAN**). Next, it reasons through which of the sub-tasks should be executed at the present step (**SUBTASK**), a task which requires understanding the current state of the scene and robot. Then, the model predicts an even lower-level language command like "move left" or "move up" (**MOVE**) that is closely related to the low-level actions the robot needs to execute. Finally, we ask the model to predict precise, spatially grounded features that describe the scene and thus force the model to pay close attention to all elements of the input image– specifically, the pixel position of the robot end effector (**GRIPPER**) and the names and bounding box pixel coordinates of all objects in the scene (**OBJECTS**).

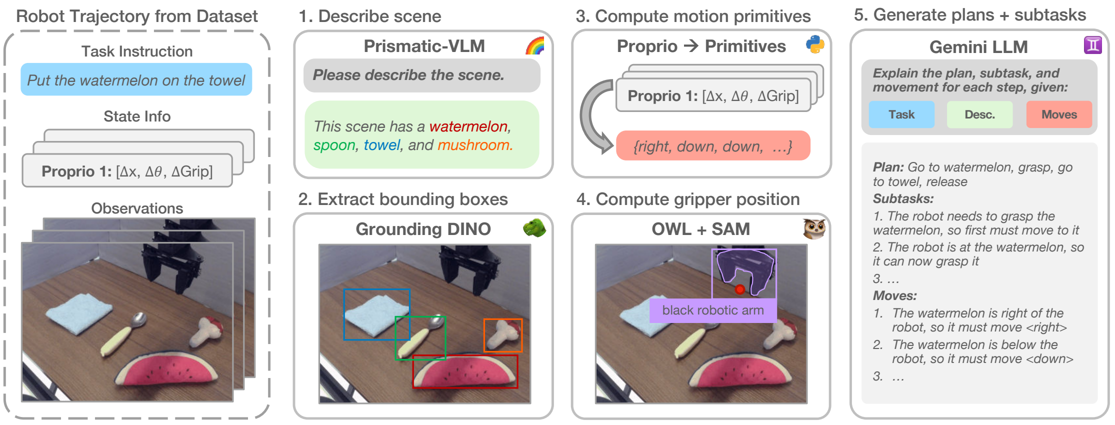

Robot Trajectory from Dataset
Task Instruction
Put the watermelon on the towel
State Info
Proprio 1: [Δx, Δθ, ΔGrip]
Observations
1. Describe scene
Prismatic-VLM
Please describe the scene.
This scene has a watermelon, spoon, towel, and mushroom.
2. Extract bounding boxes
Grounding DINO
3. Compute motion primitives
Proprio → Primitives
Proprio 1: [Δx, Δθ, ΔGrip]
{right, down, down, ...}
4. Compute gripper position
OWL + SAM
black robotic arm
5. Generate plans + subtasks
Gemini LLM
Explain the plan, subtask, and movement for each step, given:
Task
Desc.
Moves
Plan: Go to watermelon, grasp, go to towel, release
Subtasks:
1. The robot needs to grasp the watermelon, so first must move to it
2. The robot is at the watermelon, so it can now grasp it
3. ...
Moves:
1. The watermelon is right of the robot, so it must move <right>
2. The watermelon is below the robot, so it must move <down>
3. ...

Figure 4.2: Our pipeline for generating synthetic embodied chain-of-thought data at scale for a given robot dataset. We use a Prismatic VLM [Karamcheti et al., 2024] to create a scene description (**1**), and Grounding Dino [Liu et al., 2023] to detect bounding boxes for all objects (**2**). We then compute templated motion primitives from the low-level robot states (**3**) and the robot gripper position using OWLv2 [Minderer et al., 2024] and SAM [Kirillov et al., 2023] (**4**). Finally, all information is passed to a large Gemini language model [Gemini Team, 2024] to create the synthetic reasoning chain (**5**).

4.3. Experiments

4.3.1. Performance

Type	Task	Algorithm (ID View)					Algorithm (OOD View)				
		Octo	**OpenVLA**	**RT-2-X**	**Naive CoT**	**ECoT (Ours)**	**Octo**	**OpenVLA**	**RT-2-X**	**Naive CoT**	**ECoT (Ours)**
ID	Put mushroom in pot	29%	88%	94%	71%	**100%**	35%	59%	**76%**	**76%**	65%
	Put spoon on towel	60%	**90%**	80%	60%	80%	20%	**80%**	**80%**	60%	**80%**
	Put carrot on plate	70%	80%	90%	90%	**100%**	40%	90%	90%	**100%**	90%
	Wipe [plate / pan] with towel	13%	**50%**	38%	38%	**50%**	0%	50%	0%	13%	**63%**
Spatial Relations	Put mushroom in [left / right / middle] container	0%	22%	17%	22%	**33%**	0%	17%	22%	55%	**67%**
	Put purple object in [left / right / middle] container	0%	28%	17%	50%	**56%**	0%	22%	11%	**55%**	39%
	Put [right / left] object on middle object	0%	13%	0%	50%	**63%**	0%	25%	25%	50%	**63%**
OOD Objects	Pick up [screwdriver / hammer / measuring tape / detergent / watermelon]	30%	20%	**80%**	50%	50%	30%	20%	**80%**	50%	50%
	Move mushroom to [measuring tape / detergent]	0%	10%	70%	20%	**100%**	10%	0%	**90%**	40%	**90%**
	Put mushroom in tall cup	0%	**80%**	0%	70%	30%	10%	20%	0%	20%	**30%**
	Place watermelon on towel	20%	30%	60%	60%	**70%**	50%	10%	**90%**	30%	40%
OOD Instructions	Pick up any object that is not [yellow / a duck / a sponge / a towel]	50%	33%	**58%**	50%	42%	17%	17%	**67%**	25%	**67%**
	Put the edible object in the bowl	13%	25%	13%	25%	**88%**	0%	13%	25%	25%	**100%**
	Put the object used for [eating / drinking] on towel	25%	38%	38%	38%	**75%**	13%	0%	25%	38%	**75%**
Aggregate		21% ± 3.3%	44% ± 3.9%	47 ± 4.0%	48 ± 4.0%	**66% ± 3.8%**	16% ± 2.9%	30% ± 3.6%	48 ± 4.0%	48% ± 4.0%	**64 ± 3.9%**

Table 4.1: **Comparison of success rates for OpenVLA, RT-2-X, and ECoT** across two scenes (one with in-distribution camera view and one with out-of-distribution). Mean ± one StdErr. On aggregate, our ECoT policy achieves the highest success rate, improving absolute success rate by 45%, 22%, 19%, and 18% over Octo, OpenVLA, RT-2-X, and naïve CoT respectively in the in-distribution view setting and 48%, 34%, 16%, and 16% in the out-of-distribution view setting.

We report performance of all approaches on our evaluation set in Table 4.1. We see that while OpenVLA achieves high performance on in-distribution tasks, it struggles on the hard generalization cases we test. RT-2-X performs better than vanilla OpenVLA, potentially due to the larger robot pre-training dataset (note again that OpenVLA and our approach are *only* trained on the Bridge dataset) and the fact that it *co-trains* the policy with Internet-scale vision-language data *and* robot data, while all other approaches only use robot data during fine-tuning.

Importantly, we find that our ECoT policy substantially outperforms the OpenVLA policy across all generalization evaluations. This is notable, since both policies are based on the exact same VLM base model and use the same robot data for fine-tuning. The only difference is in the use of CoT reasoning by our approach. Curiously, our ECoT model even surpasses the performance of RT-2-X in the tested tasks, even though RT-2-X is trained on 10 additional robot datasets and uses a network that is 7x larger (55B vs. 7B). Finally, the results in Table 4.1 show that including *embodied* reasoning about visual inputs and the low-level robot state significantly boosts performance over the "Naïve CoT" ablation of our approach, which only reasons about high-level linguistic features like sub-task plans.

4.3.2. Diagnosing Policy Failures Through Inspecting Reasoning Chains

In addition to improving performance, chain-of-thought reasoning provides a tool for users and researchers to better understand the decisions the policy takes. By inspecting and visualizing the model's reasoning steps, we can discover potential mistakes in the reasoning chain that led to policy failure downstream.

Additionally, training a policy to reason through a task step-by-step in natural language provides a powerful mechanism for humans to interact with the policy and *correct* its behavior. Instead of needing involved teleoperation equipment to provide direct robot action feedback like in DAgger approaches [Kelly et al., 2019], humans can now simply correct the policy's behavior by modifying its reasoning chains via natural language feedback. Prior work introduced carefully designed policy architectures and explicitly trained policies to support such correction via language [Sharma et al., 2022, Shi et al., 2024]. Here, similar capabilities emerge naturally by training VLA policies to perform chain-of-thought reasoning.

Experiments show that our ECoT policy can make effective use of the human language intervention, increasing its success rate by 48% on our most challenging evaluation tasks. In contrast, we evaluate the vanilla, non-CoT OpenVLA policy and RT-2-X in the same way, providing each with a single human language correction per rollout, but find that neither of them can benefit from the human intervention to the same degree.

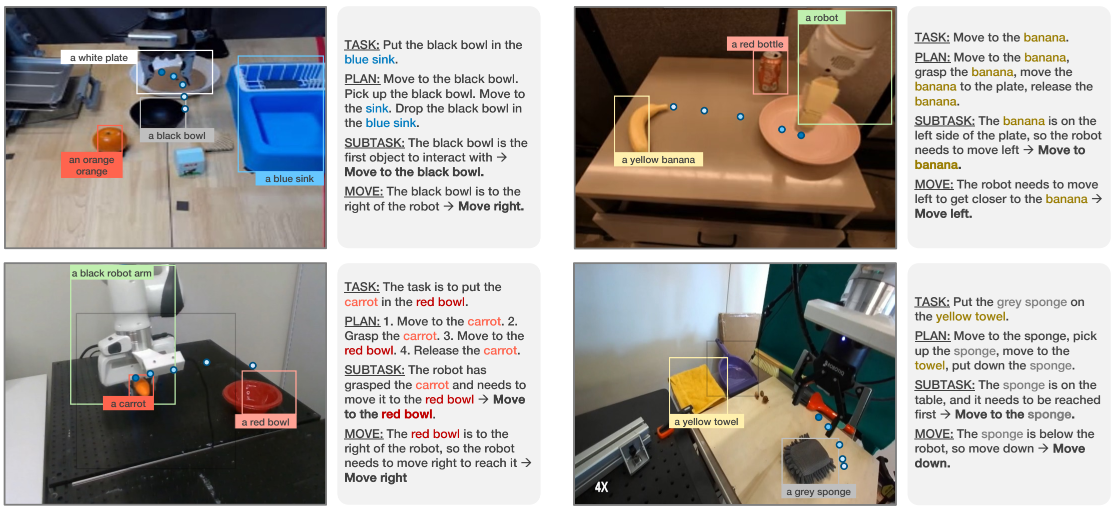
4.3.3. Does ECoT capability transfer to other robots?

We test whether fine-tuning a *generalist* VLA policy with ECoT data can transfer ECoT reasoning between robot embodiments. Concretely, we use the official checkpoint of the OpenVLA-7B model [Kim et al., 2024a], which was trained on a mix of 27 robot datasets. We continue training the released checkpoint on this mix, but replace the original BridgeData V2 dataset with our generated ECoT dataset. As a result, approximately 13% of the training data is ECoT data.

We make two key findings. First, fine-tuning a pre-trained VLA to perform ECoT reasoning is substantially faster than training an ECoT VLA from the base VLM. Secondly, we find that the fine-tuned model can perform ECoT reasoning on other robot embodiments than it has been trained for. It correctly recognizes robot grippers, objects and their positions, and predicting future gripper movements, despite the large differences in robot appearance, scene layout and camera setup. We hypothesize that the VLM pre-training enables the model to generalize the concepts of robot end-effector position and movement, and object identity and positions between robots and scenes.

4.4. Summary

We demonstrated that training VLA policies to perform chain-of-thought reasoning can substantially increase their performance without the need to collect additional robot training data. Instead of simply applying the CoT recipe from language modeling, our experiments underline the importance of adding reasoning steps that are strongly grounded in the scene and robot state, and involve for example object bounding boxes, the robot's end-effector, or low-level robot movements. Importantly, this approach not only improve the performance, but also allow inspecting the *inner thinking* of the model and straightforward interactive correction of failures.



TASK: Put the black bowl in the blue sink.

PLAN: Move to the black bowl. Pick up the black bowl. Move to the sink. Drop the black bowl in the blue sink.

SUBTASK: The black bowl is the first object to interact with → **Move to the black bowl.**

MOVE: The black bowl is to the right of the robot → **Move right.**

TASK: Move to the banana.

PLAN: Move to the banana, grasp the banana, move the banana to the plate, release the banana.

SUBTASK: The banana is on the left side of the plate, so the robot needs to move left → **Move to banana.**

MOVE: The robot needs to move left to get closer to the banana → **Move left.**

TASK: The task is to put the carrot in the red bowl.

PLAN: 1. Move to the carrot. 2. Grasp the carrot. 3. Move to the red bowl. 4. Release the carrot.

SUBTASK: The robot has grasped the carrot and needs to move it to the red bowl → **Move to the red bowl.**

MOVE: The red bowl is to the right of the robot, so the robot needs to move right to reach it → **Move right**

TASK: Put the grey sponge on the yellow towel.

PLAN: Move to the sponge, pick up the sponge, move to the towel, put down the sponge.

SUBTASK: The sponge is on the table, and it needs to be reached first → **Move to the sponge.**

MOVE: The sponge is below the robot, so move down → **Move down.**

Figure 4.3: Our OXE fine-tuned ECoT model can generate reasonings for non-WidowX robots too, despite never having seen reasoning annotations for said embodiment.

Chapter 5

Conclusions

My research addressed a fundamental challenge in reinforcement learning: enabling agents to reason and act with human-like adaptability, efficiency, and interpretability. While deep RL has achieved remarkable successes, its limitations in generalization, data efficiency, and transparency highlight the need for more structured approaches.

To this end, we introduced three contributions that integrate structured reasoning and planning into RL. Embodied Chain-of-Thought (ECoT) enables agents to explicitly plan before acting, improving generalization and interpretability in robotic tasks. Subgoal Search (kSubS) introduces a hierarchical planning framework that successfully decomposes complex problems into manageable parts. Adaptive Subgoal Search (AdaSubS) extends this by adjusting planning granularity based on task complexity, striking a balance between precision and efficiency.

Together, these methods demonstrate that combining learning with structured reasoning enhances robustness, performance, and human-alignment. By planning through subgoals and reasoning chains, agents gain the ability to generalize beyond training data and provide interpretable decision traces.

While challenges remain in scaling and automating structured representations, this work offers concrete steps toward RL systems that do more than react – they reason. This marks progress toward the broader goal of building AI that thinks and adapts more like humans.

5.1. Future directions

While this dissertation presents significant advancements, several open challenges remain, providing opportunities for future research:

- **Enhancing Subgoal Search with better subgoal discovery.** Exploring alternative approaches to generating subgoals and leveraging the latest architectures, such as diffusion models, have the potential to further improve the effectiveness of the framework.

- **Fine-tuning with online interactions.** While Subgoal Search can successfully leverage offline data for training, extending it with a self-improvement loop is another promising direction.

- **Subgoal Search on latent representations.** Subgoal Search cannot be used in environments with visual observations as long as it operates on full states rather than representations.

- **Applying Subgoal Search to robotics.** We demonstrated the effectiveness of Subgoal Search in solving complex reasoning tasks. Following the human-like planning intuition, it would be exciting to see Subgoal Search guiding robots in solving intricate manipulation tasks.

- **Adaptive Embodied Chain-of-Thought.** A lot of computations can be saved by deciding when to reason carefully and when thinking is not required.

- **Exploring human-robot interaction.** Since using the ECoT approach leads to interactive policies, it may be used as an advanced and easy control interface, replacing scripting or teleoperation.

5.2. Acknowledgments

I would like to thank all the amazing people who made my PhD journey both possible and enjoyable.

First and foremost, I am deeply grateful to my supervisor, Piotr Miłoś, for his mentorship and constant support since my Master's studies. His guidance, insightful discussions, and encouragement – whether during the research process, paper writing, or struggling with peer reviews – have been invaluable. I am also thankful to him for introducing me to Sergey Levine and for countless other ways of support along the way.

My sincere thanks to Łukasz Kuciński, whose support has been constant since my Master's as well, and whose help throughout my PhD I truly appreciate.

I would also like to thank Sergey Levine for the opportunity to join his lab at Berkeley, and for his generous support during my time there.

My special thanks go to Konrad Czechowski, who supervised my work on Subgoal Search, and with whom I shared many hours of inspiring meetings and conversations. I'm also particularly grateful to Karl Pertsch for his day-to-day mentorship during my time at Berkeley, for introducing me to robotics, and for his constant support.

I am fortunate to have collaborated with a group of exceptional co-authors, to whom I owe a great deal: Franciszek Budrowski, William Chen, Marek Cygan, Konrad Czechowski, Chelsea Finn, Gracjan Góral, Łukasz Kuciński, Sergey Levine, Oier Mees, Henryk Michalewski, Piotr Miłoś, Tomasz Odrzygóźdź, Krzysztof Olejnik, Błażej Osiński, Karl Pertsch, Piotr Piękos, Damian Stachura, Michał Tyrolski, Emilia Wiśnios, Yuhuai Wu, and Marek Zbysiński.

I'm also thankful to everyone I worked with at the Warsaw RL Lab, IDEAS, and BAIR. The opportunity to exchange ideas, work together, and share everyday conversations has been both intellectually and personally enriching. Those informal chats often sparked some of the most valuable ideas, and I'm grateful for each one.

Finally, I want to thank all the countless people who have supported me in big and small ways over the years: through spontaneous discussions, thoughtful critiques, interesting lectures and seminars, and countless everyday kindnesses. Your impact on my work has been profound.

Bibliography

[Akkaya et al., 2019] Akkaya, I., Andrychowicz, M., Chociej, M., Litwin, M., McGrew, B., Petron, A., Paino, A., Plappert, M., Powell, G., Ribas, R., et al. (2019). Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113.*

[Berner et al., 2019] Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J., Petrov, M., de Oliveira Pinto, H. P., Raiman, J., Salimans, T., Schlatter, J., Schneider, J., Sidor, S., Sutskever, I., Tang, J., Wolski, F., and Zhang, S. (2019). Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680.

[Bommasani et al., 2022] Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S., Bernstein, M. S., Bohg, J., Bosselut, A., Brunskill, E., Brynjolfsson, E., Buch, S., Card, D., Castellon, R., Chatterji, N., Chen, A., Creel, K., Davis, J. Q., Demszky, D., Donahue, C., Doumbouya, M., Durmus, E., Ermon, S., Etchemendy, J., Ethayarajh, K., Fei-Fei, L., Finn, C., Gale, T., Gillespie, L., Goel, K., Goodman, N., Grossman, S., Guha, N., Hashimoto, T., Henderson, P., Hewitt, J., Ho, D. E., Hong, J., Hsu, K., Huang, J., Icard, T., Jain, S., Jurafsky, D., Kalluri, P., Karamcheti, S., Keeling, G., Khani, F., Khattab, O., Koh, P. W., Krass, M., Krishna, R., Kuditipudi, R., Kumar, A., Ladhak, F., Lee, M., Lee, T., Leskovec, J., Levent, I., Li, X. L., Li, X., Ma, T., Malik, A., Manning, C. D., Mirchandani, S., Mitchell, E., Munyikwa, Z., Nair, S., Narayan, A., Narayanan, D., Newman, B., Nie, A., Niebles, J. C., Nilforoshan, H., Nyarko, J., Ogut, G., Orr, L., Papadimitriou, I., Park, J. S., Piech, C., Portelance, E., Potts, C., Raghunathan, A., Reich, R., Ren, H., Rong, F., Roohani, Y., Ruiz, C., Ryan, J., Ré, C., Sadigh, D., Sagawa, S., Santhanam, K., Shih, A., Srinivasan, K., Tamkin, A., Taori, R., Thomas, A. W., Tramèr, F., Wang, R. E., Wang, W., Wu, B., Wu, J., Wu, Y., Xie, S. M., Yasunaga, M., You, J., Zaharia, M., Zhang, M., Zhang, T., Zhang, X., Zhang, Y., Zheng, L., Zhou, K., and Liang, P. (2022). On the opportunities and risks of foundation models.

[Brohan et al., 2023] Brohan, A., Brown, N., Carbajal, J., Chebotar, Y., Chen, X., Choromanski, K., Ding, T., Driess, D., Dubey, A., Finn, C., Florence, P., Fu, C., Arenas, M. G., Gopalakrishnan, K., Han, K., Hausman, K., Herzog, A., Hsu, J., Ichter, B., Irpan, A., Joshi, N., Julian, R., Kalashnikov, D., Kuang, Y., Leal, I., Lee, L., Lee, T.-W. E., Levine, S., Lu, Y., Michalewski, H., Mordatch, I., Pertsch, K., Rao, K., Reymann, K., Ryoo, M., Salazar, G., Sanketi, P., Sermanet, P., Singh, J., Singh, A., Soricut, R., Tran, H., Vanhoucke, V., Vuong, Q., Wahid, A., Welker, S., Wohlhart, P., Wu, J., Xia, F., Xiao, T., Xu, P., Xu, S., Yu, T., and Zitkovich, B. (2023). Rt-2: Vision-language-action models transfer web knowledge to robotic control.

[Brown et al., 2020] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H., editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual.*

[Chi et al., 2023] Chi, C., Feng, S., Du, Y., Xu, Z., Cousineau, E., Burchfiel, B., and Song, S. (2023). Diffusion policy: Visuomotor policy learning via action diffusion. In Bekris, K. E., Hauser, K., Herbert, S. L., and Yu, J., editors, *Robotics: Science and Systems XIX, Daegu, Republic of Korea, July 10-14, 2023.*

[Chiang et al., 2019] Chiang, H. L., Faust, A., Fiser, M., and Francis, A. G. (2019). Learning navigation behaviors end-to-end with autorl. *IEEE Robotics Autom. Lett.*, 4(2):2007–2014.

[Christen et al., 2023] Christen, S. J., Yang, W., Pérez-D'Arpino, C., Hilliges, O., Fox, D., and Chao, Y. (2023). Learning human-to-robot handovers from point clouds. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2023, Vancouver, BC, Canada, June 17-24, 2023*, pages 9654–9664. IEEE.

[Collaboration et al., 2023] Collaboration, O. X., Padalkar, A., Pooley, A., Jain, A., Bewley, A., Herzog, A., Irpan, A., Khazatsky, A., Raj, A., Singh, A., Brohan, A., Raffin, A., Wahid, A., Burgess-Limerick, B., Kim, B., Schölkopf, B., Ichter, B., Lu, C., Xu, C., Finn, C., Xu, C., Chi, C., Huang, C., Chan, C., Pan, C., Fu, C., Devin, C., Driess, D., Pathak, D., Shah, D., Büchler, D., Kalashnikov, D., Sadigh, D., Johns, E., Ceola, F., Xia, F., Stulp, F., Zhou, G., Sukhatme, G. S., Salhotra, G., Yan, G., Schiavi, G., Kahn, G., Su, H., Fang, H., Shi, H., Amor, H. B., Christensen, H. I., Furuta, H., Walke, H., Fang, H., Mordatch, I., Radosavovic, I., and et al. (2023). Open x-embodiment: Robotic learning datasets and RT-X models. *CoRR*, abs/2310.08864.

[Collaboration et al., 2024] Collaboration, O. X.-E., O'Neill, A., Rehman, A., Maddukuri, A., Gupta, A., Padalkar, A., Lee, A., Pooley, A., Gupta, A., Mandlekar, A., Jain, A., Tung, A., Bewley, A., Herzog, A., Irpan, A., Khazatsky, A., Rai, A., Gupta, A., Wang, A. E., Singh, A., Garg, A., Kembhavi, A., Xie, A., Brohan, A., Raffin, A., Sharma, A., Yavary, A., Jain, A., Balakrishna, A., Wahid, A., Burgess-Limerick, B., Kim, B., Schölkopf, B., Wulfe, B., Ichter, B., Lu, C., Xu, C., Le, C., Finn, C., Wang, C., Xu, C., Chi, C., Huang, C., Chan, C., Agia, C., Pan, C., Fu, C., Devin, C., Xu, D., Morton, D., Driess, D., Chen, D., Pathak, D., Shah, D., Büchler, D., Jayaraman, D., Kalashnikov, D., Sadigh, D., Johns, E., Foster, E. P., Liu, F., Ceola, F., Xia, F., Zhao, F., Stulp, F., Zhou, G., Sukhatme, G. S., Salhotra, G., Yan, G., Feng, G., Schiavi, G., Berseth, G., Kahn, G., Wang, G., Su, H., Fang, H., Shi, H., Bao, H., Amor, H. B., Christensen, H. I., Furuta, H., Walke, H., Fang, H., Ha, H., Mordatch, I., Radosavovic, I., Leal, I., Liang, J., Abou-Chakra, J., Kim, J., Drake, J., Peters, J., Schneider, J., Hsu, J., Bohg, J., Bingham, J., Wu, J., Gao, J., Hu, J., Wu, J., Wu, J., Sun, J., Luo, J., Gu, J., Tan, J., Oh, J., Wu, J., Lu, J., Yang, J., Malik, J., Silvério, J., Hejna, J., Booher, J., Tompson, J., Yang, J., Salvador, J., Lim, J. J., Han, J., Wang, K., Rao, K., Pertsch, K., Hausman, K., Go, K., Gopalakrishnan, K., Goldberg, K., Byrne, K., Oslund, K., Kawaharazuka, K., Black, K., Lin, K., Zhang, K., Ehsani, K., Lekkala, K., Ellis, K., Rana, K., Srinivasan, K., Fang, K., Singh, K. P., Zeng, K., Hatch, K., Hsu, K., Itti, L., Chen, L. Y., Pinto, L., Fei-Fei, L., Tan, L., Fan, L. J., Ott, L., Lee, L., Weihs, L., Chen, M., Lepert, M., Memmel, M., Tomizuka, M., Itkina, M., Castro, M. G., Spero, M., Du, M., Ahn, M., Yip, M. C., Zhang, M., Ding, M., Heo, M., Srirama, M. K., Sharma, M., Kim, M. J., Kanazawa, N., Hansen, N., Heess, N., Joshi, N. J., Sünderhauf, N., Liu, N., Palo, N. D., Shafiullah, N. M. M., Mees, O., Kroemer, O., Bastani, O., Sanketi, P. R., Miller, P. T., Yin, P., Wohlhart, P., Xu, P., Fagan, P. D., Mitrano, P., Sermanet, P., Abbeel, P., Sundaresan, P., Chen, Q., Vuong, Q., Rafailov, R., Tian, R., Doshi, R., Martín-Martín, R., Baijal, R., Scalise, R., Hendrix, R., Lin, R., Qian, R., Zhang, R., Mendonca, R., Shah, R., Hoque, R., Julian, R., Bustamante, S., Kirmani, S., Levine, S., Lin, S., Moore, S., Bahl, S., Dass, S., Sonawani, S. D., Song, S., Xu, S., Haldar, S., Karamcheti, S., Adebola, S., Guist, S., Nasiriany, S., Schaal, S., Welker, S., Tian, S., Ramamoorthy, S., Dasari, S., Belkhale, S., Park, S., Nair, S., Mirchandani, S., Osa, T., Gupta, T., Harada, T., Matsushima, T., Xiao, T., Kollar, T., Yu, T., Ding, T., Davchev, T., Zhao, T. Z., Armstrong, T., Darrell, T., Chung, T., Jain, V., Vanhoucke, V., Zhan, W., Zhou, W., Burgard, W., Chen, X., Wang, X., Zhu, X., Geng, X., Liu, X., Xu, L., Li, X., Lu, Y., Ma, Y. J., Kim, Y., Chebotar, Y., Zhou, Y., Zhu, Y., Wu, Y., Xu, Y., Wang, Y., Bisk, Y., Cho, Y., Lee, Y., Cui, Y., Cao, Y., Wu, Y., Tang, Y., Zhu, Y., Zhang, Y., Jiang, Y., Li, Y., Li, Y., Iwasawa, Y., Matsuo, Y., Ma, Z., Xu, Z., Cui, Z. J., Zhang, Z., and Lin, Z. (2024). Open x-embodiment: Robotic learning datasets and RT-X models : Open x-embodiment collaboration. In *IEEE International Conference on Robotics and Automation, ICRA 2024, Yokohama, Japan, May 13-17, 2024*, pages 6892–6903. IEEE.

[Cormen et al., 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms.* MIT press.

[Coulom, 2006] Coulom, R. (2006). Efficient selectivity and backup operators in monte-carlo tree search. In van den Herik, H. J., Ciancarini, P., and Donkers, H. H. L. M., editors, *Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. Springer.

[Culberson, 1997] Culberson, J. C. (1997). Sokoban is pspace-complete.

[Czechowski et al., 2021] Czechowski, K., Odrzygózdz, T., Zbysinski, M., Zawalski, M., Olejnik, K., Wu, Y., Kucinski, L., and Milos, P. (2021). Subgoal search for complex reasoning tasks. In Ranzato, M., Beygelzimer, A., Dauphin, Y. N., Liang, P., and Vaughan, J. W., editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 624–638.

[Demaine et al., 2018] Demaine, E. D., Eisenstat, S., and Rudoy, M. (2018). Solving the rubik's cube optimally is np-complete. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Devlin et al., 2019] Devlin, J., Chang, M., Lee, K., and Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding. In Burstein, J., Doran, C., and Solorio, T., editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics.

[Embodiment Collaboration et al., 2024] Embodiment Collaboration, O'Neill, A., Rehman, A., Maddukuri, A., Gupta, A., Padalkar, A., Lee, A., Pooley, A., Gupta, A., Mandlekar, A., Jain, A., Tung, A., Bewley, A., Herzog, A., Irpan, A., Khazatsky, A., Rai, A., Gupta, A., Wang, A., Kolobov, A., Singh, A., Garg, A., Kembhavi, A., Xie, A., Brohan, A., Raffin, A., Sharma, A., Yavary, A., Jain, A., Balakrishna, A., Wahid, A., Burgess-Limerick, B., Kim, B., Schölkopf, B., Wulfe, B., Ichter, B., Lu, C., Xu, C., Le, C., Finn, C., Wang, C., Xu, C., Chi, C., Huang, C., Chan, C., Agia, C., Pan, C., Fu, C., Devin, C., Xu, D., Morton, D., Driess, D., Chen, D., Pathak, D., Shah, D., Büchler, D., Jayaraman, D., Kalashnikov, D., Sadigh, D., Johns, E., Foster, E., Liu, F., Ceola, F., Xia, F., Zhao, F., Frujeri, F. V., Stulp, F., Zhou, G., Sukhatme, G. S., Salhotra, G., Yan, G., Feng, G., Schiavi, G., Berseth, G., Kahn, G., Wang, G., Su, H., Fang, H.-S., Shi, H., Bao, H., Amor, H. B., Christensen, H. I., Furuta, H., Walke, H., Fang, H., Ha, H., Mordatch, I., Radosavovic, I., Leal, I., Liang, J., Abou-Chakra, J., Kim, J., Drake, J., Peters, J., Schneider, J., Hsu, J., Bohg, J., Bingham, J., Wu, J., Gao, J., Hu, J., Wu, J., Wu, J., Sun, J., Luo, J., Gu, J., Tan, J., Oh, J., Wu, J., Lu, J., Yang, J., Malik, J., Silvério, J., Hejna, J., Booher, J., Tompson, J., Yang, J., Salvador, J., Lim, J. J., Han, J., Wang, K., Rao, K., Pertsch, K., Hausman, K., Go, K., Gopalakrishnan, K., Goldberg, K., Byrne, K., Oslund, K., Kawaharazuka, K., Black, K., Lin, K., Zhang, K., Ehsani, K., Lekkala, K., Ellis, K., Rana, K., Srinivasan, K., Fang, K., Singh, K. P., Zeng, K.-H., Hatch, K., Hsu, K., Itti, L., Chen, L. Y., Pinto, L., Fei-Fei, L., Tan, L., Fan, L. J., Ott, L., Lee, L., Weihs, L., Chen, M., Lepert, M., Memmel, M., Tomizuka, M., Itkina, M., Castro, M. G., Spero, M., Du, M., Ahn, M., Yip, M. C., Zhang, M., Ding, M., Heo, M., Srirama, M. K., Sharma, M., Kim, M. J., Kanazawa, N., Hansen, N., Heess, N., Joshi, N. J., Suenderhauf, N., Liu, N., Palo, N. D., Shafiullah, N. M. M., Mees, O., Kroemer, O., Bastani, O., Sanketi, P. R., Miller, P. T., Yin, P., Wohlhart, P., Xu, P., Fagan, P. D., Mitrano, P., Sermanet, P., Abbeel, P., Sundaresan, P., Chen, Q., Vuong, Q., Rafailov, R., Tian, R., Doshi, R., Mart'in-Mart'in, R., Baijal, R., Scalise, R., Hendrix, R., Lin, R., Qian, R., Zhang, R., Mendonca, R., Shah, R., Hoque, R., Julian, R., Bustamante, S., Kirmani, S., Levine, S., Lin, S., Moore, S., Bahl, S., Dass, S., Sonawani, S., Song, S., Xu, S., Haldar, S., Karamcheti, S., Adebola, S., Guist, S., Nasiriany, S., Schaal, S., Welker, S., Tian, S., Ramamoorthy, S., Dasari, S., Belkhale, S., Park, S., Nair, S., Mirchandani, S., Osa, T., Gupta, T., Harada, T., Matsushima, T., Xiao, T., Kollar, T., Yu, T., Ding, T., Davchev, T., Zhao, T. Z., Armstrong, T., Darrell, T., Chung, T., Jain, V., Vanhoucke, V., Zhan, W., Zhou, W., Burgard, W., Chen, X., Chen, X., Wang, X., Zhu, X., Geng, X., Liu, X., Liangwei, X., Li, X., Pang, Y., Lu, Y., Ma, Y. J., Kim, Y., Chebotar, Y., Zhou, Y., Zhu, Y., Wu, Y., Xu, Y., Wang, Y., Bisk, Y., Cho, Y., Lee, Y., Cui, Y., Cao, Y., Wu, Y.-H., Tang, Y., Zhu, Y., Zhang, Y., Jiang, Y., Li, Y., Li, Y., Iwasawa, Y., Matsuo, Y., Ma, Z., Xu, Z., Cui, Z. J., Zhang, Z., Fu, Z., and Lin, Z. (2024). Open x-embodiment: Robotic learning datasets and rt-x models.

[Fawzi et al., 2022] Fawzi, A., Balog, M., Huang, A., Hubert, T., Romera-Paredes, B., Barekatain, M., Novikov, A., Ruiz, F. J. R., Schrittwieser, J., Swirszcz, G., Silver, D., Hassabis, D., and Kohli, P. (2022). Discovering faster matrix multiplication algorithms with reinforcement learning. *Nat.*, 610(7930):47–53.

[Gemini Team, 2024] Gemini Team (2024). Gemini: A family of highly capable multimodal models.

[Goodfellow et al., 2014] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. C., and Bengio, Y. (2014). Generative adversarial networks. *CoRR*, abs/1406.2661.

[Hafner et al., 2023] Hafner, D., Pasukonis, J., Ba, J., and Lillicrap, T. P. (2023). Mastering diverse domains through world models. *CoRR*, abs/2301.04104.

[Hamrick et al., 2021] Hamrick, J. B., Friesen, A. L., Behbahani, F., Guez, A., Viola, F., Witherspoon, S., Anthony, T., Buesing, L. H., Velickovic, P., and Weber, T. (2021). On the role of planning in model-based deep reinforcement learning. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

[He et al., 2016] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society.

[Ho et al., 2020] Ho, J., Jain, A., and Abbeel, P. (2020). Denoising diffusion probabilistic models. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H., editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

[Hollerman et al., 2000] Hollerman, J. R., Tremblay, L., and Schultz, W. (2000). Involvement of basal ganglia and orbitofrontal cortex in goal-directed behavior. *Progress in brain research*, 126:193–215.

[Jumper et al., 2021] Jumper, J. M., Evans, R., Pritzel, A., Green, T., Figurnov, M., Ronneberger, O., Tunyasuvunakool, K., Bates, R., Zídek, A., Potapenko, A., Bridgland, A., Meyer, C., Kohl, S. A. A., Ballard, A., Cowie, A., Romera-Paredes, B., Nikolov, S., Jain, R., Adler, J., Back, T., Petersen, S., Reiman, D. A., Clancy, E., Zielinski, M., Steinegger, M., Pacholska, M., Berghammer, T., Bodenstein, S., Silver, D., Vinyals, O., Senior, A. W., Kavukcuoglu, K., Kohli, P., and Hassabis, D. (2021). Highly accurate protein structure prediction with alphafold. *Nature*, 596:583–589.

[Karamcheti et al., 2024] Karamcheti, S., Nair, S., Balakrishna, A., Liang, P., Kollar, T., and Sadigh, D. (2024). Prismatic vlms: Investigating the design space of visually-conditioned language models.

[Kelly et al., 2019] Kelly, M., Sidrane, C., Driggs-Campbell, K., and Kochenderfer, M. J. (2019). Hg-dagger: Interactive imitation learning with human experts.

[Kim et al., 2024a] Kim, M., Pertsch, K., Karamcheti, S., Xiao, T., Balakrishna, A., Nair, S., Rafailov, R., Foster, E., Sanketi, P., Vuong, Q., Kollar, T., Burchfiel, B., Tedrake, R., Sadigh, D., Levine, S., Liang, P., and Finn, C. (2024a). Openvla: An open-source vision-language-action model.

[Kim et al., 2024b] Kim, M. J., Pertsch, K., Karamcheti, S., Xiao, T., Balakrishna, A., Nair, S., Rafailov, R., Foster, E. P., Sanketi, P. R., Vuong, Q., Kollar, T., Burchfiel, B., Tedrake, R., Sadigh, D., Levine, S., Liang, P., and Finn, C. (2024b). Openvla: An open-source vision-language-action model. In Agrawal, P., Kroemer, O., and Burgard, W., editors, *Conference on Robot Learning, 6-9 November 2024, Munich, Germany*, volume 270 of *Proceedings of Machine Learning Research*, pages 2679–2713. PMLR.

[Kirillov et al., 2023] Kirillov, A., Mintun, E., Ravi, N., Mao, H., Rolland, C., Gustafson, L., Xiao, T., Whitehead, S., Berg, A. C., Lo, W.-Y., et al. (2023). Segment anything. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4015–4026.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Bartlett, P. L., Pereira, F. C. N., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pages 1106–1114.

[Li et al., 2025] Li, Z., Peng, X. B., Abbeel, P., Levine, S., Berseth, G., and Sreenath, K. (2025). Reinforcement learning for versatile, dynamic, and robust bipedal locomotion control. *Int. J. Robotics Res.*, 44(5):840–888.

[Liu et al., 2023] Liu, S., Zeng, Z., Ren, T., Li, F., Zhang, H., Yang, J., Li, C., Yang, J., Su, H., Zhu, J., and Zhang, L. (2023). Grounding dino: Marrying dino with grounded pre-training for open-set object detection.

[Liu et al., 2024] Liu, Y., Zhang, K., Li, Y., Yan, Z., Gao, C., Chen, R., Yuan, Z., Huang, Y., Sun, H., Gao, J., He, L., and Sun, L. (2024). Sora: A review on background, technology, limitations, and opportunities of large vision models. *CoRR*, abs/2402.17177.

[Loquercio et al., 2019] Loquercio, A., Kaufmann, E., Ranftl, R., Dosovitskiy, A., Koltun, V., and Scaramuzza, D. (2019). Deep drone racing: From simulation to reality with domain randomization. *IEEE Transactions on Robotics*, 36(1):1–14.

[McCarthy, 1959] McCarthy, J. (1959). Programs with common sense.

[Minderer et al., 2024] Minderer, M., Gritsenko, A., and Houlsby, N. (2024). Scaling open-vocabulary object detection. *Advances in Neural Information Processing Systems*, 36.

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

[Newell and Simon, 1956] Newell, A. and Simon, H. A. (1956). The logic theory machine-a complex information processing system. *IRE Trans. Inf. Theory*, 2(3):61–79.

[OpenAI, 2023] OpenAI (2023). GPT-4 technical report. *CoRR*, abs/2303.08774.

[OpenAI et al., 2019] OpenAI, Akkaya, I., Andrychowicz, M., Chociej, M., Litwin, M., McGrew, B., Petron, A., Paino, A., Plappert, M., Powell, G., Ribas, R., Schneider, J., Tezak, N., Tworek, J., Welinder, P., Weng, L., Yuan, Q., Zaremba, W., and Zhang, L. (2019). Solving rubik's cube with a robot hand. *CoRR*, abs/1910.07113.

[Osiński et al., 2020] Osiński, B., Jakubowski, A., Zięcina, P., Miłoś, P., Galias, C., Homoceanu, S., and Michalewski, H. (2020). Simulation-based reinforcement learning for real-world autonomous driving. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6411–6418. IEEE.

[Peng et al., 2018] Peng, X. B., Andrychowicz, M., Zaremba, W., and Abbeel, P. (2018). Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE International Conference on Robotics and Automation, ICRA 2018, Brisbane, Australia, May 21-25, 2018*, pages 1–8. IEEE.

[Popov et al., 2017] Popov, I., Heess, N., Lillicrap, T. P., Hafner, R., Barth-Maron, G., Vecerík, M., Lampe, T., Tassa, Y., Erez, T., and Riedmiller, M. A. (2017). Data-efficient deep reinforcement learning for dexterous manipulation. *CoRR*, abs/1704.03073.

[Radosavovic et al., 2024] Radosavovic, I., Zhang, B., Shi, B., Rajasegaran, J., Kamat, S., Darrell, T., Sreenath, K., and Malik, J. (2024). Humanoid locomotion as next token prediction. In Globersons, A., Mackey, L., Belgrave, D., Fan, A., Paquet, U., Tomczak, J. M., and Zhang, C., editors, *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

[Russell and Norvig, 2020] Russell, S. and Norvig, P. (2020). *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson.

[Schoettler et al., 2020] Schoettler, G., Nair, A., Luo, J., Bahl, S., Ojea, J. A., Solowjow, E., and Levine, S. (2020). Deep reinforcement learning for industrial insertion tasks with visual inputs and natural rewards. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2020, Las Vegas, NV, USA, October 24, 2020 - January 24, 2021*, pages 5548–5555. IEEE.

[Shah et al., 2021] Shah, D., Eysenbach, B., Kahn, G., Rhinehart, N., and Levine, S. (2021). Ving: Learning open-world navigation with visual goals. In *IEEE International Conference on Robotics and Automation, ICRA 2021, Xi'an, China, May 30 - June 5, 2021*, pages 13215–13222. IEEE.

[Shah et al., 2022] Shah, D., Osinski, B., Ichter, B., and Levine, S. (2022). Lm-nav: Robotic navigation with large pre-trained models of language, vision, and action. In Liu, K., Kulic, D., and Ichnowski, J., editors, *Conference on Robot Learning, CoRL 2022, 14-18 December 2022, Auckland, New Zealand*, volume 205 of *Proceedings of Machine Learning Research*, pages 492–504. PMLR.

[Sharma et al., 2022] Sharma, P., Sundaralingam, B., Blukis, V., Paxton, C., Hermans, T., Torralba, A., Andreas, J., and Fox, D. (2022). Correcting robot plans with natural language feedback.

[Shi et al., 2024] Shi, L. X., Hu, Z., Zhao, T. Z., Sharma, A., Pertsch, K., Luo, J., Levine, S., and Finn, C. (2024). Yell at your robot: Improving on-the-fly from language corrections.

[Silver et al., 2016a] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T. P., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016a). Mastering the game of go with deep neural networks and tree search. *Nat.*, 529(7587):484–489.

[Silver et al., 2016b] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016b). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489.

[Silver et al., 2018a] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. (2018a). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 1144:1140–1144.

[Silver et al., 2018b] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. (2018b). A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. In Guyon, I., von Luxburg, U., Bengio, S., Wallach, H. M., Fergus, R., Vishwanathan, S. V. N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008.

[Vinyals et al., 2019] Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J. P., Jaderberg, M., Vezhnevets, A. S., Leblond, R., Pohlen, T., Dalibard, V., Budden, D., Sulsky, Y., Molloy, J., Paine, T. L., Gulcehre, C., Wang, Z., Pfaff, T., Wu, Y., Ring, R., Yogatama, D., Wünsch, D., McKinney, K., Smith, O., Schaul, T., Lillicrap, T., Kavukcuoglu, K., Hassabis, D., Apps, C., and Silver, D. (2019). Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354.

[Walke et al., 2023a] Walke, H., Black, K., Lee, A., Kim, M. J., Du, M., Zheng, C., Zhao, T., Hansen-Estruch, P., Vuong, Q., He, A., Myers, V., Fang, K., Finn, C., and Levine, S. (2023a). Bridgedata v2: A dataset for robot learning at scale. In *Conference on Robot Learning (CoRL)*.

[Walke et al., 2023b] Walke, H. R., Black, K., Zhao, T. Z., Vuong, Q., Zheng, C., Hansen-Estruch, P., He, A. W., Myers, V., Kim, M. J., Du, M., Lee, A., Fang, K., Finn, C., and Levine, S. (2023b). Bridgedata V2: A dataset for robot learning at scale. In Tan, J., Toussaint, M., and Darvish, K., editors, *Conference on Robot Learning, CoRL 2023, 6-9 November 2023, Atlanta, GA, USA*, volume 229 of *Proceedings of Machine Learning Research*, pages 1723–1736. PMLR.

[Wei et al., 2023] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., and Zhou, D. (2023). Chain-of-thought prompting elicits reasoning in large language models.

[Wu et al., 2021] Wu, Y., Jiang, A. Q., Ba, J., and Grosse, R. B. (2021). INT: an inequality benchmark for evaluating generalization in theorem proving. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

[Yu et al., 2023] Yu, C., Liu, J., Nemati, S., and Yin, G. (2023). Reinforcement learning in healthcare: A survey. *ACM Comput. Surv.*, 55(2):5:1–5:36.

[Zawalski et al., 2024a] Zawalski, M., Chen, W., Pertsch, K., Mees, O., Finn, C., and Levine, S. (2024a). Robotic control via embodied chain-of-thought reasoning. *CoRR*, abs/2407.08693.

[Zawalski et al., 2024b] Zawalski, M., Góral, G., Tyrolski, M., Wisnios, E., Budrowski, F., Kucinski, L., and Milos, P. (2024b). What matters in hierarchical search for combinatorial reasoning problems? *CoRR*, abs/2406.03361.

[Zawalski et al., 2023] Zawalski, M., Tyrolski, M., Czechowski, K., Odrzygózdz, T., Stachura, D., Piekos, P., Wu, Y., Kucinski, L., and Milos, P. (2023). Fast and precise: Adjusting planning horizon with adaptive subgoal search. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

[Zhang et al., 2019] Zhang, Z., Zohren, S., and Roberts, S. J. (2019). Deep reinforcement learning for trading. *CoRR*, abs/1911.10107.

Appendix A

Complete publications

Off-Policy Correction For Multi-Agent Reinforcement Learning

Michał Zawalski
University of Warsaw
Warsaw, Poland
m.zawalski@uw.edu.pl

Błażej Osiński
University of Warsaw
Warsaw, Poland
blazej.osinski@gmail.com

Henryk Michalewski
Google Research
henrykm@google.com

Piotr Miłoś
Polish Academy of Sciences
Warsaw, Poland
pmilos@impan.pl

ABSTRACT

Multi-agent reinforcement learning (MARL) provides a framework for problems involving multiple interacting agents. Despite apparent similarity to the single-agent case, multi-agent problems are often harder to train and analyze theoretically. In this work, we propose MA-Trace, a new on-policy actor-critic algorithm, which extends V-Trace to the MARL setting. The key advantage of our algorithm is its high scalability in a multi-worker setting. To this end, MA-Trace utilizes importance sampling as an off-policy correction method, which allows distributing the computations with no impact on the quality of training. Furthermore, our algorithm is theoretically grounded – we prove a fixed-point theorem that guarantees convergence. We evaluate the algorithm extensively on the StarCraft Multi-Agent Challenge, a standard benchmark for multi-agent algorithms. MA-Trace achieves high performance on all its tasks and exceeds state-of-the-art results on some of them.

KEYWORDS

Reinforcement Learning, V-Trace, Importance Sampling, Scalability

1 INTRODUCTION

Reinforcement learning has witnessed impressive development in recent years. Famously, superhuman performance has been achieved in games Go [30], StarCraft II [36], Dota 2 [4] and other applications. These successes are the result of rapid algorithmic development. Research in directions like trust-region optimization [29], principle of maximum entropy [13], importance sampling [8], distributional RL [3] or bridging the sim-to-real gap [2] are among these which brought significant progress. Multi-agent reinforcement learning (MARL), a framework for problems involving multiple interacting agents, is similar to the standard, single-agent setting. However, it is inherently harder. The challenges are both theoretical (e.g., partial observability and lack of the Markov property) and practical (MARL algorithms often suffer from inferior stability and scalability).

In this work, we take a step towards amending this situation. We propose MA-Trace, a new on-policy actor-critic algorithm, which adheres to the centralized training and decentralized execution paradigm [10, 19, 25]. The key component of MA-Trace is the usage of importance sampling. This mechanism, based on V-Trace [8], provides off-policy correction for training data. As we demonstrate empirically, it allows distributing the computations efficiently in a multi-worker setup. Another advantage of MA-Trace is the fact that it is theoretically grounded. We provide a fixed-point theorem that guarantees convergence.

The on-policy algorithms directly optimize the objective; thus, they tend to be more stable and robust to hyperparameter choices than off-policy methods [1, 33, 35]. However, it is often impractical to train an on-policy algorithm in the distributed setting. When data collection is performed using many workers, the communication latency, asynchronicity, and other factors make the behavioral policies lag behind the target one. This results in a shift of the collected data towards off-policy distribution, which hurts the training quality. V-Trace reduces this shift by utilizing importance weights, thus permitting highly scalable training. Following that scheme, MA-Trace can be distributed to many workers to vastly reduce the wall-time of training with no negative impact on the results.

We evaluate MA-Trace on StarCraft Multi-Agent Challenge [27] – a standard benchmark for multi-agent algorithms. Our approach achieves competitive performance on all tasks and exceeds state-of-the-art results on some of them. Additionally, we provide a comprehensive set of ablations to quantify the influence of each component on the final results. We confirm that importance sampling is a key factor for MA-Trace's performance and show that our algorithm scales favorably with the number of actor workers. Additionally, we provide a few quite surprising findings, e.g. that an observation-based critic network performs better than a state-based.

For the description of key ideas and videos, visit our webpage: https://sites.google.com/view/ma-trace/main-page. The code used for our experiments is available at https://github.com/awarelab/seed_rl.

Our main contributions are the following:

(1) We introduce MA-Trace – a simple, scalable and effective multi-agent reinforcement learning algorithm with theoretical guarantees.
(2) We confirm that the training of MA-Trace can be easily distributed on multiple workers with nearly perfect speed-up and no negative impact on the quality.
(3) We provide extensive experimental validation of the MA-Trace algorithm in StarCraft Multi-Agent Challenge, including ablations with regard to importance sampling, centralization of learning, scaling and sharing of parameters.

2 RELATED WORK

For a general overview of multi-agent reinforcement learning (MARL) we refer to [5, 15]. Unsurprisingly, the development of MARL methods is closely coupled with the algorithmic progress in RL. A simple approach to multi-agent learning was proposed by Tan [34]: the IQL algorithm uses independent Q learners for each agent, with improvements proposed in [11, 17, 24].

MA-Trace adheres to the centralized training and decentralized execution (CTDE) paradigm. CTDE [9, 16, 23] is based on using the centralized information during training. During execution, the agents act using only their respective observations. Following this scheme, [9] introduces the RIAL and DIAL algorithms in the context of Q-learning. CTDE is particularly easy to implement with actor-critic algorithms; the centralized information is imputed only to the critic network (which is not used during the execution). COMA [10] is an example of such an algorithm; additionally, it uses a counterfactual baseline to deal with multi-agent credit assignment explicitly.

Another approach to take advantage of the multi-agent structure is the *value decomposition* method. VDN [32] propose a linear decomposition of the collective Q function into agent-local Q functions. Following this idea, [25] introduced QMIX, which learns a complex state-dependent decomposition by using monotonic mixing hypernetworks. Extensions of QMIX include MAVEN [21], COMIX [6], SMIX(λ) [39], and QTRAN [31] that can represent even general non-monotonic factorizations.

MA-Trace is based on V-Trace [8], a distributed single-agent algorithm. The idea of extending RL algorithms to the multi-agent setting has been successfully executed multiple times. Lowe et al. [19] propose a multi-agent actor-critic algorithm MADDPG, which is based on the DDPG algorithm [18]. Yu et al. [40] introduce also MASAC, extending SAC [14], and MATD3 building on top of TD3 [12]. Recently [41] showed that MAPPO, a multi-agent version of PPO [29], achieves surprisingly strong results in the most popular benchmarks, comparable with off-policy methods.

Espeholt et al. [8] propose the V-Trace algorithm to address the problem that in distributed (e.g. multi-node) training the policy used to generate experience is likely to lag behind the policy used for learning. Munos et al. [22] considered earlier a similar off-policy corrections for the target of the Q-function. These corrections are intended to focus on samples generated by behavioral policies close to the target one. Leaky V-Trace, a more general version of the V-Trace correction, was considered by Zahavy et al. [42]. Vinyals et al. [37] adapt V-Trace importance corrections to large action space to train grandmaster level StarCraft II agents. These corrections are refinements of the concept of importance sampling; see [33, Sections 5.5, 12.9] for a broader discussion.

Blending all these concepts, DOP [38] utilizes value decomposition and importance sampling to successfully train decentralized agents with policy gradients on off-policy samples. This is substantially different from our work since in MA-Trace we use importance weights to enable efficient multi-node training. DOP does not consider distributing the computations, the objective optimized by that algorithm requires providing on-policy samples, which is impossible to satisfy in a highly distributed setting.

3 BACKGROUND

Multi-agent reinforcement learning task is formalized by *decentralized partially observable Markov decision processes* (Dec-POMDP) [23]. A Dec-POMDP is defined as a tuple $(\mathcal{N}, \mathcal{S}, \mathcal{A}, P, r, \mathcal{Z}, O, \gamma, \rho_0)$. $\mathcal{N}$ is the set of agents $\{1, \ldots, n\}$, $\mathcal{S}$ is the state space, $\mathcal{A}$ is the set of actions available to agents, P is the transition kernel, r is the reward function, $\mathcal{Z}$ is the space of collective observations, O is the set of observation functions $\{O_1, \ldots, O_n\}$, γ is the discount factor and ρ_0 is the initial state distribution. At state $s \in \mathcal{S}$, the agents select actions $a_i \sim \pi_i(\cdot|O_i(s))$, where π_i are their respective polices. Fix $a := (a_1, \ldots, a_n)$. The agents receive rewards according to the reward functions $r_i = r_i(s, a)$ and the system evolves to the next step generated by $P(s, a)$ (might be stochastic). In the so-called fully cooperative setting, assumed in this work, the rewards are equal, i.e. $r_1 = \ldots = r_n$.

The agents learn a joint policy

$$\pi(a|o) = \prod_{i=1}^{n} \pi_i(a_k|o_k) \tag{1}$$

with the aim to maximize the expected discounted return

$$J(\pi) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \right].$$

The expected discounted return obtained by policy π starting from state $s \in \mathcal{S}$ is called the value function

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s \right], \tag{2}$$

4 MA-TRACE ALGORITHM

4.1 Overview of the algorithm

In this work, we introduce a multi-agent actor-critic algorithm based on V-trace: **MA-Trace**, see Algorithm 1. It follows the paradigm of centralized training, decentralized execution. Each agent compute its action taking its local observation as input. On the other hand, the critic network operates only during training, so it does not need to obey decentralization requirements. Furthermore, it can utilize any kind of additional information. We study two versions of MA-Trace: with the critic $V : \mathcal{S} \to \mathbb{R}$ taking as inputs full states, and with the critic $V : \mathcal{Z} \to \mathbb{R}$ taking as input the joint observation of all agents, denoted respectively as MA-Trace (full) and MA-Trace (obs). MA-Trace (full) requires collecting states s_t in line 6 of Algorithm 1 and using them as input to V_ϕ in lines 10 and 14.

The value function V^π corresponding to policy π can be, for example, obtained by repeated application of the Bellman operator. This requires on-policy data. The central innovation of V-Trace in the single-player setting and MA-Trace in the multi-player setting is to allow for slightly off-policy data by utilizing importance sampling. To this end, we use the V-Trace-inspired policy evaluation operator $\mathcal{R}$, defined as

$$\mathcal{R}V(s) := V(s) + \mathbb{E}_\mu \left[\sum_{t=0}^{+\infty} \gamma^t (c_0 \cdots c_{t-1}) \rho_t (r_t + \gamma V(s_{t+1}) - V(s_t)) | s_0 = s \right], \tag{3}$$

where $c_t = c(s_t, a_t), \rho_t := \rho(s_t, a_t)$ are importance sampling corrections and $c : \mathcal{S} \times \mathcal{A} \to \mathbb{R}_+, \rho : \mathcal{S} \times \mathcal{A} \to \mathbb{R}_+$ are measurable functions. In our algorithms we specialize to

$$c_t := \min\left(\bar{c}, \frac{\pi(a_t|s_t)}{\mu(a_t|s_t)}\right), \quad \rho_t := \min\left(\bar{\rho}, \frac{\pi(a_t|s_t)}{\mu(a_t|s_t)}\right), \tag{4}$$

where μ is a policy that collected the data and $\bar{c}, \bar{\rho}$ are hyperparameters (usually set to 1.0). Intuitively speaking, c_t controls the speed of training and ρ_t balances the learned value function between V^{π} and V^{μ}. These parameters are further discussed in Corollary 3. The operator $\mathcal{R}$ leads to n-step Monte-Carlo target v_t given the state s_t

$$v_t := V(s_t) + \sum_{u=t}^{t+n+1} \gamma^{u-t} \left(\prod_{i=t}^{u-1} c_i\right) \rho_u \big(r_u + \gamma V(s_{u+1}) - V(s_u)\big). \tag{5}$$

It is a random variable; the clipping with the min function in (4) is instrumental to reducing its variance and thus making it applicable in learning. A key advantage of MA-Trace is a significant reduction in the wall-time due to distributed data collection (see line 6 in Algorithm 1). From the algorithmic standpoint, the major problem to address is that the policy used for collection $\pi_{\theta'}$ might be outdated due to communication overheads. This is successfully achieved with the importance correction mechanisms described above.

We use the communication model proposed in [7, Figure 1, Figure 3]. It consists of a single learner and actor workers. The actors are simple loops around the environment, generating observations (and rewards) transmitted to the learner. The learner makes inferences (and sends back actions); moreover, it handles trajectory accumulation and training.

Algorithm 1 MA-Trace

```
Require:  $d$  density of training
          $\alpha$  learning rate
1:  for $k$ in $0, \dots, n-1$ do
2:      $\omega_k \leftarrow$ random actor parameters
3:  $\phi \leftarrow$ random critic parameters
4:  for epoch in $0, 1, 2, \dots$ do
5:      $\mathcal{D} \leftarrow \emptyset$
6:      add trajectories $\{\tau_i\}$ sampled with $\pi_{\theta'}$ to $\mathcal{D}$
7:                      ▷ collected by multiple workers possibly remote.
8:      for $i$ in $0, \dots, d-1$ do
9:          sample $s_t \sim \mathcal{D}$
10:         $\phi \leftarrow \phi - \alpha \nabla_\phi \left[\|v_t - V_\phi(s_t)\|_2^2\right]$      ▷ $v_t$ is calculated
    according to (5)
11:     for $i$ in $0, \dots, d-1$ do
12:         sample $s_t \sim \mathcal{D}$
13:         for $k$ in $1, \dots, n$ do
14:             $g_k \leftarrow \rho_t \nabla_\omega \log(\pi_\omega(a_{t,k}|s_{t,k}))(r_{t,k} + \gamma V_\phi^k(s_{t+1}) -$
    $V_\phi^k(s_t))$
15:                                 ▷ $\rho_t$ is calculated according to (4)
16:             $\omega_k \leftarrow \theta_k + \alpha g_k$
```

4.2 Theoretical analysis of MA-Trace

The operator, $\mathcal{R}$ enjoys the fixed point property. We present a proof of the following Theorem in Appendix A.

THEOREM 1. *Let c, ρ be such that for any $s \in \mathcal{S}, a \in \mathcal{A}$*

$$\rho(s,a) - c(s,a)\,\mathbb{E}_{a' \sim \mu(\cdot|s')}\left[\rho(s', a')\right] \geq 0, \tag{6}$$

where s' is the state obtained from s after issuing action a. Assume also that $\mathbb{E}_\mu \rho_0 \geq \beta \in (0, 1]$. Then the operator $\mathcal{R}$ is a C_∞ contraction with a unique fixed point $V^{\tilde{\pi}}$ which is a value function of a policy $\tilde{\pi}$ given by

$$\tilde{\pi}(a|s) := \frac{\rho(s,a)\mu(a|s)}{\sum_{b \in \mathcal{A}} \rho(s,b)\mu(b|s)}. \tag{7}$$

The contraction constant is smaller than $1 - (1-\gamma)\beta < 1$.

REMARK 2. *The theorem is an extended version of [8, Theorem 1]. First, we assume the vectorized statement, which is natural for the multi-agent setting. Second, the condition (6) admits more general importance sampling weights. We also fix a mathematical inaccuracy present in the original proof of [8, Theorem 1], see Remark 6.*

Now we can easily show that the result follows for the importance weights used in our work.

COROLLARY 3. *Let c_t, ρ_t be importance sampling weights (4) and $0 \leq \bar{c} \leq \bar{\rho}$. Assume also that $\mathbb{E}_\mu \rho_0 \geq \beta \in (0, 1]$. Then the operator $\mathcal{R}$ is a C_∞ contraction with a unique fixed point $V^{\tilde{\pi}}$ which is a value function of a policy $\tilde{\pi}$ given by*

$$\tilde{\pi}(a|s) := \frac{\min\left(\bar{\rho}\mu(a|s), \pi(a|s)\right)}{\sum_{b \in \mathcal{A}} \min\left(\bar{\rho}\mu(b|s), \pi(b|s)\right)}.$$

The contraction constant is smaller than $1 - (1-\gamma)\beta < 1$.

PROOF. It is easy to check that $c(s,a) = \min\left(\bar{c}, \frac{\pi(a|s)}{\mu(a|s)}\right), \rho(s,a) = \min\left(\bar{\rho}, \frac{\pi(a|s)}{\mu(a|s)}\right)$ yield the importance sampling weights (4). Moreover, for any $s' \in \mathcal{S}$

$$\begin{aligned}\mathbb{E}_{a' \sim \mu(\cdot|s')}\left[\rho(s', a')\right] &= \mathbb{E}_{a' \sim \mu(\cdot|s')} \min\left(\bar{\rho}, \frac{\pi(a'|s')}{\mu(a'|s')}\right) \\ &\leq \mathbb{E}_{a \sim \mu(\cdot|s')}\left(\frac{\pi(a'|s')}{\mu(a'|s')}\right) \\ &= 1\end{aligned}$$

and therefore (6) holds whenever $0 \leq \bar{c} \leq \bar{\rho}$. Now the result follows by Theorem 1. □

Observe that when $\bar{\rho}$ is infinite, the fixed point $V^{\tilde{\pi}}$ corresponds to the target policy π. On the other hand, when $\bar{\rho}$ tends to 0, $V^{\tilde{\pi}}$ gets close to the value of the behavioral policy μ. In the general case, when $\bar{\rho}$ is finite but positive, the fixed point is the value function of a policy located somewhere between π and μ. However, $\tilde{\pi}$ does not depend on c_t – these weights affect the speed of convergence only [8].

REMARK 4. *There is a theoretical difference between MA-Trace (full) and MA-Trace (obs), which perhaps is subtle in some cases. The Markov property is a key element required in the proof of Corollary 3. While it is by definition true for MA-Trace (state) it might fail for MA-Trace (obs) - if the concatenated observations do not provide a sufficient statistic of s_t.*

For the actor network we use policy gradient updates. Here we also need importance sampling to correct for using the off-policy behavioral policy μ. We recall the factorization (1); analogously we denote joint decentralized parameterized policy $\pi_\omega(a_1, \dots, a_K|s) = \prod_{k=1}^{K} \pi_\omega(a_i|s_i)$. The policy gradient theorem suggests the ascent in the direction:

$$g := \mathbb{E}_{a_t \sim \pi_\omega} \left[\nabla_\omega \log \pi_\omega(a_t|s_t) A^{\pi_\omega}(s_t, a_t) \right],$$

where $a_t = (a_{t,1}, \dots, a_{t,K})$ and A^{π_ω} is some advantage estimator. For off-policy data collected with μ we have

$$g \approx \mathbb{E}_{a_t \sim \mu} \left[\rho_t \nabla_\omega \log \pi_\omega(a_t|s_t) A^{\pi_\omega}(s_t, a_t) \right],$$

where the equality holds if $\bar{c} = +\infty$ in (4). This formula leads to the practical Monte-Carlo estimator used in line 14 of Algorithm 1:

$$\rho_t (\nabla_{\omega_i} \log(\pi_{\omega_i}(a_{t,i}|s_{t,i})))(r_t + \gamma V(s_{t+1}) - V(s_t)). \qquad (8)$$

5 EXPERIMENTS

5.1 Environment

We evaluate MA-Trace on StarCraft Multi-Agent Challenge (SMAC) [28] (version 4.10), which is a standard benchmark for multi-agent algorithms, based on a popular real-time strategy game StarCraft II. It provides 14 micromanagement tasks of varying difficulty and structure.

The aim is to win a battle against a built-in AI by using your team of agents. In easier tasks, often rudimentary coordination is enough. However, harder tasks involve engaging a stronger enemy (e.g., having more units), which requires inventing smart techniques and tricks. Each unit has a limited line of sight, which makes the environment partially observable. We provide more details in Appendix F.

5.2 Main result

In Table 1 and Figure 1, we present the results of the main version of our algorithm – MA-Trace (obs), in which the critic uses stacked observations of agents as described in Appendix F. MA-Trace (obs) reaches competitive results and in some cases exceeds the current state-of-the-art. We compare with a selection of the state-of-the-art algorithms on SMAC following [41] and [26]. We also demonstarte scalability, which lets MA-Trace can reach good performance even using short training (wall time).

Task	MA-Trace (our)	MAPPO	IPPO	QMIX	COMA	IQL
2s3z	99 [99.5; 99.7]	100	100	95	43	75
3s5z	97 [96.6; 98.6]	100	100	88	1	10
1c3s5z	100 [99.8; 100]	100	100	96	31	21
5m_vs_6m	78 [74.3; 78.4]	89	87	75	1	49
10m_vs_11m	96 [86.2; 97.7]	97	93	95	7	34
27m_vs_30m	99 [99; 100]	94	69	39	0	0
3s5z_vs_3s6z	87 [81.6; 92.1]	84	83	83	0	0
MMM2	98 [98; 98.8]	90	87	87	0	0
2s_vs_1sc	99 [99; 99.6]	100	100	97	98	100
3s_vs_5z	0 [0; 0]	100	100	98	0	45
6h_vs_8z	85 [71; 88.8]	88	84	9	0	0
bane_vs_bane	100 [100; 100]	100	100	100	64	99
2c_vs_64zg	98 [98; 98.5]	100	98	92	0	7
corridor	91 [88.6; 96.1]	100	98	84	0	0

Table 1: Median win rate of MA-Trace (obs) compared with other algorithms. In *3s_vs_5z*, our agent discovers that keeping the opponents alive leads to higher rewards than killing them. This strategy, however, yields a low win rate. See Appendix F.1 for a detailed study.

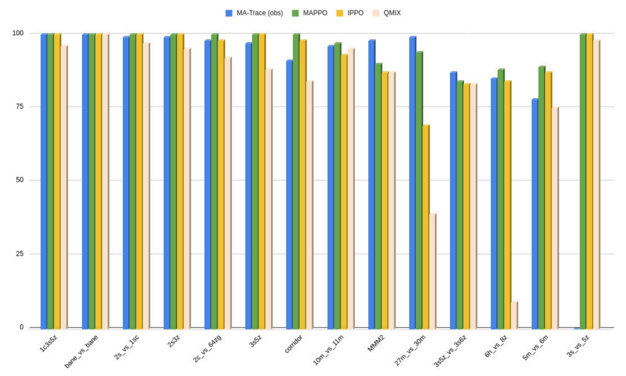


Figure 1: MA-Trace compared with state-of-the-arts algorithms on SMAC.

5.3 Training details

We use standard feed-forward networks for the actor and critic networks with two hidden layers of 64 neurons and ReLU activations. The critic network of MA-Trace (obs) takes stacked observations of agents as input, while MA-Trace (full) utilizes the full state provided by SMAC. DecMA-Trace have a critic using single-agent observations. See details in Appendix C.

For each reported version of MA-Trace, we have searched for the best hyperparameters to ensure a fair comparison. The values of all hyperparameters are listed in Appendix D.2.

To estimate the performance of MA-Trace we run training for 3 days or until convergence. We report the median win rate of 10 runs (with different random seeds) along with the interquartile range. Training curves for all the tasks can be found in Appendix G.

5.4 Ablations

Below we present a comprehensive list of ablations to evaluate the design choices of our algorithm. In each case, we present training curves for tasks, which best illustrate our claims. For the complete training results and more details, we refer to Appendix E.

Advantage of using importance sampling. Using the importance weights is the key algorithmic innovation of MA-Trace (and V-Trace), responsible for the strong performance we report. Indeed, already for 30 actor workers, using the weights is essential. Otherwise, the algorithm is unstable and suffers from poor asymptotic performance. See Figure 2 and Appendix E.2.

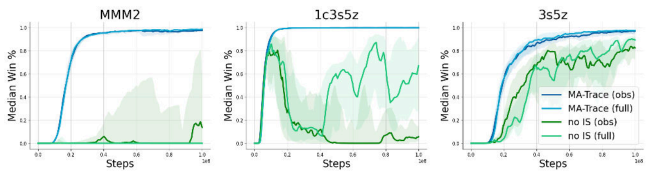


Figure 2: MA-Trace using 30 distributed workers with and without importance sampling (no IS).

Training scaling. The importance sampling enables V-Trace to be truly scalable in multi-node setups. MA-Trace enjoys the same property. Importantly, we do not observe any degradation in the training performance when trained in the multi-node setup. See Figure 3 and Appendix E.3.

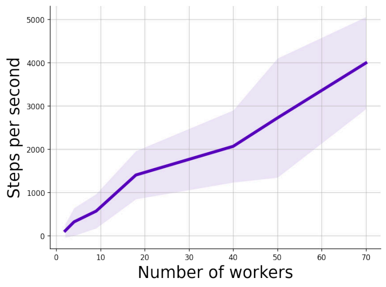


Figure 3: Speed of MA-Trace training with respect to the number of distributed workers, with standard deviation shaded. The speed is measured as the average number of steps processed per second.

Input for the critic network. We found that MA-Trace (full) performs slightly worse than MA-Trace (obs). Usually the differences are small. However, in two harder tasks, *corridor* and *6h_vs_8z*, MA-Trace (full) learns much slower and often fails. This is perhaps surprising, as the full state contains additional information (e.g., about invisible opponents). To deepen the analysis, we ran MA-Trace (obs+full), which uses both the observations and full state as the critic input. This improves the results, though they are still slightly inferior to MA-Trace (obs); see Figure 4 and Appendix G. A more detailed discussion of this topic can be found in Appendix E.1.

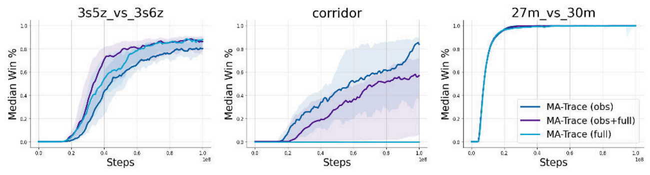


Figure 4: Comparison of using the full state MA-Trace (full) and aggregated agents' observations MA-Trace (obs) and both.

Centralized vs decentralized. As noted by [20], centralized training in some cases may suffer from higher variance. Therefore we compared MA-Trace with its decentralized version (i.e. having indpendent critics for each agent). The latter typically obtains weaker results and is less stable. See Figure 5 and details in Appendix B.

Sharing actors networks. We follow a common approach of sharing the policy network between agents. In some works, e.g., [26], to preserve individuality, the observations are enriched with agent ID. This might be beneficial if agents should be assigned different roles within the team. However, we find these benefits rather minor and opt for input provided by the environment (i.e., without ID). See Figure 7 and details in Appendix C.2.

One can also use separate networks for each agent. We check that MA-Trace works considerably worse in such a case. In rare

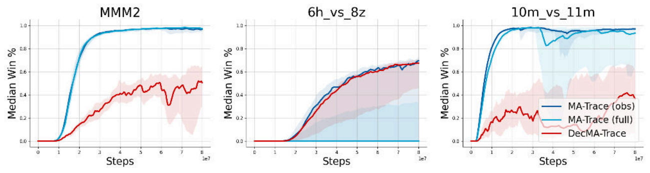


Figure 5: Performance of MA-Trace during centralized and decentralized training.

cases, using separate networks is advantageous, but only in the easiest tasks, e.g., *3s5z*. See Figure 6 and details in Appendix C.1.

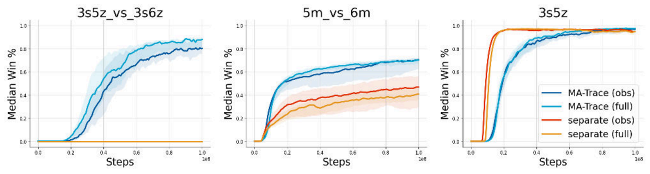


Figure 6: Performance of shared (standard MA-Trace) and separate agents' networks.

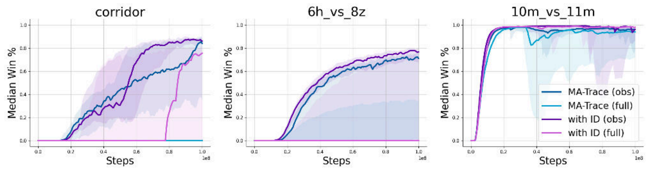


Figure 7: The impact of enriching observation with agent ID.

6 LIMITATIONS AND FURTHER WORK

We show that MA-Trace successfully solves SMAC tasks. Further benchmarking is needed to underpin its quality. This includes testing on more environments, both fully cooperative (like SMAC) and competitive. The latter might require further algorithmic developments.

The importance sampling weights successfully reduce distributional shifts arising in distributed training. An interesting question is whether they can also reduce the shifts introduced by the nonstationarity of the multi-agent environment.

MA-Trace exhibits lower sample efficiency than the other methods we used for comparisons. This, at least partially, can be explained by its on-policy nature. Adapting the importance correction to accommodate more off-policy data would be an important achievement.

7 CONCLUSIONS

In our work, we introduced MA-Trace, a new multi-agent reinforcement learning algorithm. We evaluated it on 14 scenarios constituting the StarCraft Multi-Agent Challenge and confirmed its strong performance. We also included ablations regarding importance sampling, centralization of learning, scaling, and sharing of parameters.

Thanks to the use of importance weights, MA-Trace is highly scalable. Furthermore, the convergence properties of our algorithm highlighted by Theorem 1 show that it has not only experimental but also mathematical grounding.

ACKNOWLEDGMENTS

We thank Konrad Staniszewski for discussions and experiments in the prequel of this project.

The work of Michał Zawalski was supported by the Polish National Science Center (NCN) grant UMO-2019/35/O/ST6/03464. The work of Henryk Michalewski was supported by the Polish National Science Center grant UMO-2018/29/B/ST6/02959. The work of Piotr Miłoś was supported by the NCN grant UMO-2017/26/E/ST6/00622. This research was supported by the PL-Grid Infrastructure. Some experiments were performed using the Entropy cluster funded by NVIDIA, Intel, the Polish National Science Center grant UMO-2017/26/E/ST6/00622 and ERC Starting Grant TOTAL. Our experiments were managed using https://neptune.ai. We would like to thank the Neptune team for providing access to the team version and technical support.

REFERENCES

[1] Josh Achiam. 2018. Spinning Up in Deep RL. https://spinningup.openai.com.

[2] Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Józefowicz, Bob McGrew, Jakub W. Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor, Josh Tobin, Peter Welinder, Lilian Weng, and Wojciech Zaremba. 2020. Learning dexterous in-hand manipulation. *Int. J. Robotics Res.* 39, 1 (2020). https://doi.org/10.1177/0278364919887447

[3] Marc G. Bellemare, Will Dabney, and Rémi Munos. 2017. A Distributional Perspective on Reinforcement Learning. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017 (Proceedings of Machine Learning Research, Vol. 70)*, Doina Precup and Yee Whye Teh (Eds.). PMLR, 449–458. http://proceedings.mlr.press/v70/bellemare17a.html

[4] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. 2019. Dota 2 with Large Scale Deep Reinforcement Learning. *CoRR* abs/1912.06680 (2019). arXiv:1912.06680 http://arxiv.org/abs/1912.06680

[5] Lucian Busoniu, Robert Babuska, and Bart De Schutter. 2008. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 38, 2 (2008), 156–172.

[6] Christian Schroeder de Witt, Bei Peng, Pierre-Alexandre Kamienny, Philip Torr, Wendelin Böhmer, and Shimon Whiteson. 2020. Deep multi-agent reinforcement learning for decentralized continuous cooperative control. *arXiv preprint arXiv:2003.06709* (2020).

[7] Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. 2020. SEED RL: Scalable and Efficient Deep-RL with Accelerated Central Inference. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=rkgvXlrKwH

[8] Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. 2018. IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018 (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer G. Dy and Andreas Krause (Eds.). PMLR, 1406–1415. http://proceedings.mlr.press/v80/espeholt18a.html

[9] Jakob N. Foerster, Yannis M. Assael, Nando de Freitas, and Shimon Whiteson. 2016. Learning to Communicate with Deep Multi-Agent Reinforcement Learning. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett (Eds.). 2137–2145. https://proceedings.neurips.cc/paper/2016/hash/c7635bfd99248a2cdef8249ef7bfbef4-Abstract.html

[10] Jakob N. Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. 2018. Counterfactual Multi-Agent Policy Gradients. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). AAAI Press, 2974–2982. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17193

[11] Jakob N. Foerster, Nantas Nardelli, Gregory Farquhar, Triantafyllos Afouras, Philip H. S. Torr, Pushmeet Kohli, and Shimon Whiteson. 2017. Stabilising Experience Replay for Deep Multi-Agent Reinforcement Learning. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017 (Proceedings of Machine Learning Research, Vol. 70)*, Doina Precup and Yee Whye Teh (Eds.). PMLR, 1146–1155. http://proceedings.mlr.press/v70/foerster17b.html

[12] Scott Fujimoto, Herke van Hoof, and David Meger. 2018. Addressing Function Approximation Error in Actor-Critic Methods. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018 (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer G. Dy and Andreas Krause (Eds.). PMLR, 1582–1591. http://proceedings.mlr.press/v80/fujimoto18a.html

[13] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018 (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer G. Dy and Andreas Krause (Eds.). PMLR, 1856–1865. http://proceedings.mlr.press/v80/haarnoja18b.html

[14] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*. PMLR, 1861–1870.

[15] Pablo Hernandez-Leal, Bilal Kartal, and Matthew E. Taylor. 2020. A Very Condensed Survey and Critique of Multiagent Deep Reinforcement Learning. In *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems, AAMAS '20, Auckland, New Zealand, May 9-13, 2020*, Amal El Fallah Seghrouchni, Gita Sukthankar, Bo An, and Neil Yorke-Smith (Eds.). International Foundation for Autonomous Agents and Multiagent Systems, 2146–2148. https://dl.acm.org/doi/abs/10.5555/3398761.3399105

[16] Landon Kraemer and Bikramjit Banerjee. 2016. Multi-agent reinforcement learning as a rehearsal for decentralized planning. *Neurocomputing* 190 (2016), 82–94.

[17] Martin Lauer and Martin A. Riedmiller. 2000. An Algorithm for Distributed Reinforcement Learning in Cooperative Multi-Agent Systems. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29 - July 2, 2000*, Pat Langley (Ed.). Morgan Kaufmann, 535–542.

[18] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2016. Continuous control with deep reinforcement learning. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1509.02971

[19] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. 2017. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 6379–6390. https://proceedings.neurips.cc/paper/2017/hash/68a9750337a418a86fe06c1991a1d64c-Abstract.html

[20] Xueguang Lyu, Yuchen Xiao, Brett Daley, and Christopher Amato. 2021. Contrasting Centralized and Decentralized Critics in Multi-Agent Reinforcement Learning. *CoRR* arXiv/2102.04402 (2021). arXiv:2102.04402 https://arxiv.org/abs/2102.04402

[21] Anuj Mahajan, Tabish Rashid, Mikayel Samvelyan, and Shimon Whiteson. 2019. MAVEN: Multi-Agent Variational Exploration. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 7611–7622. https://proceedings.neurips.cc/paper/2019/hash/f816dc0acface7498e10496222e9db10-Abstract.html

[22] Rémi Munos, Tom Stepleton, Anna Harutyunyan, and Marc G. Bellemare. 2016. Safe and Efficient Off-Policy Reinforcement Learning. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett (Eds.). 1046–1054. https://proceedings.neurips.cc/paper/2016/hash/c3992e9a68c5ae12bd18488bc579b30d-Abstract.html

[23] Frans A. Oliehoek and Christopher Amato. 2016. *A Concise Introduction to Decentralized POMDPs*. Springer. https://doi.org/10.1007/978-3-319-28929-8

[24] Shayegan Omidshafiei, Jason Pazis, Christopher Amato, Jonathan P How, and John Vian. 2017. Deep decentralized multi-task multi-agent reinforcement learning under partial observability. In *International Conference on Machine Learning*. PMLR, 2681–2690.

[25] Tabish Rashid, Mikayel Samvelyan, Christian Schröder de Witt, Gregory Farquhar, Jakob N. Foerster, and Shimon Whiteson. 2018. QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018 (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer G. Dy and Andreas Krause (Eds.). PMLR, 4292–4301. http://proceedings.mlr.press/v80/rashid18a.html

[26] Tabish Rashid, Mikayel Samvelyan, Christian Schröder de Witt, Gregory Farquhar, Jakob N. Foerster, and Shimon Whiteson. 2020. Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning. *CoRR* abs/2003.08839 (2020). arXiv:2003.08839 https://arxiv.org/abs/2003.08839

[27] Mikayel Samvelyan, Tabish Rashid, Christian Schröder de Witt, Gregory Farquhar, Nantas Nardelli, Tim G. J. Rudner, Chia-Man Hung, Philip H. S. Torr, Jakob N. Foerster, and Shimon Whiteson. 2019. The StarCraft Multi-Agent Challenge. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '19, Montreal, QC, Canada, May 13-17, 2019*, Edith Elkind, Manuela Veloso, Noa Agmon, and Matthew E. Taylor (Eds.). International Foundation for Autonomous Agents and Multiagent Systems, 2186–2188. http://dl.acm.org/citation.cfm?id=3332052

[28] Mikayel Samvelyan, Tabish Rashid, Christian Schroeder de Witt, Gregory Farquhar, Nantas Nardelli, Tim G. J. Rudner, Chia-Man Hung, Philiph H. S. Torr, Jakob Foerster, and Shimon Whiteson. 2019. The StarCraft Multi-Agent Challenge. *CoRR* abs/1902.04043 (2019).

[29] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *CoRR* abs/1707.06347 (2017). arXiv:1707.06347 http://arxiv.org/abs/1707.06347

[30] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529 (01 2016), 484–489. https://doi.org/10.1038/nature16961

[31] Kyunghwan Son, Daewoo Kim, Wan Ju Kang, David Hostallero, and Yung Yi. 2019. QTRAN: Learning to Factorize with Transformation for Cooperative Multi-Agent Reinforcement Learning. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 5887–5896. http://proceedings.mlr.press/v97/son19a.html

[32] Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinicius Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z. Leibo, Karl Tuyls, and Thore Graepel. 2017. Value-Decomposition Networks For Cooperative Multi-Agent Learning. arXiv:1706.05296 [cs.AI]

[33] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[34] Ming Tan. 1993. Multi-Agent Reinforcement Learning: Independent versus Cooperative Agents. In *Machine Learning, Proceedings of the Tenth International Conference, University of Massachusetts, Amherst, MA, USA, June 27-29, 1993*, Paul E. Utgoff (Ed.). Morgan Kaufmann, 330–337. https://doi.org/10.1016/b978-1-55860-307-3.50049-6

[35] John N. Tsitsiklis and Benjamin Van Roy. 1997. An analysis of temporal-difference learning with function approximation. *IEEE Trans. Autom. Control.* 42, 5 (1997), 674–690. https://doi.org/10.1109/9.580874

[36] Oriol Vinyals, I. Babuschkin, Wojciech Czarnecki, Michaël Mathieu, Andrew Dudzik, J. Chung, D. Choi, Richard Powell, Timo Ewalds, P. Georgiev, Junhyuk Oh, Dan Horgan, M. Kroiss, Ivo Danihelka, Aja Huang, L. Sifre, Trevor Cai, J. Agapiou, Max Jaderberg, A. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, D. Budden, Yury Sulsky, James Molloy, T. Paine, Caglar Gulcehre, Ziyu Wang, T. Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver. 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* (2019), 1–5.

[37] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* 575, 7782 (2019), 350–354.

[38] Yihan Wang, Beining Han, Tonghan Wang, Heng Dong, and Chongjie Zhang. 2020. Off-Policy Multi-Agent Decomposed Policy Gradients. *CoRR* abs/2007.12322 (2020). arXiv:2007.12322 https://arxiv.org/abs/2007.12322

[39] Chao Wen, Xinghu Yao, Yuhui Wang, and Xiaoyang Tan. 2020. SMIX(λ): Enhancing Centralized Value Functions for Cooperative Multi-Agent Reinforcement Learning. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 7301–7308. https://aaai.org/ojs/index.php/AAAI/article/view/6223

[40] Chao Yu, Akash Velu, Eugene Vinitsky, Yu Wang, Alexandre Bayen, and Yi Wu. 2020. Benchmarking Multi-agent Deep Reinforcement Learning Algorithms. Workshop in Conference on Neural Information Processing Systems 2020. https://www.researchgate.net/publication/349943157_Benchmarking_Multi-agent_Deep_Reinforcement_Learning_Algorithms

[41] Chao Yu, Akash Velu, Eugene Vinitsky, Yu Wang, Alexandre M. Bayen, and Yi Wu. 2021. The Surprising Effectiveness of MAPPO in Cooperative, Multi-Agent Games. *CoRR* abs/2103.01955 (2021). arXiv:2103.01955 https://arxiv.org/abs/2103.01955

[42] Tom Zahavy, Zhongwen Xu, Vivek Veeriah, Matteo Hessel, Junhyuk Oh, Hado van Hasselt, David Silver, and Satinder Singh. 2020. A Self-Tuning Actor-Critic Algorithm. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/f02208a057804ee16ac72ff4d3cec53b-Abstract.html

A THE PROOF OF THE FIXED POINT THEOREM

Let C_∞^K be the space of functions $\mathcal{S} \mapsto \mathbb{R}^K$ endowed with the infinity norm $\|f\|_{C_\infty^K} = \sup_{s\in\mathcal{S}} |f(s)|_\infty$. C_∞^K is Banach if $\mathcal{S}$ is finite or we additionally assume that functions are continuous (this encompasses the case when $\mathcal{S}$ is discrete).

For an easy reference we recall the statement of Theorem 1. Let $c : \mathcal{S}\times\mathcal{A} \to \mathbb{R}_+$, $\rho : \mathcal{S}\times\mathcal{A} \to \mathbb{R}_+$ be measurable functions and set $c_t = c(s_t, a_t)$, $\rho_t = \rho(s_t, a_t)$.

Define the operator $\mathcal{R} : C_\infty^K \mapsto C_\infty^K$ by

$$\begin{aligned}\mathcal{R}V(s) &:= V(s)+\\ &+\mathbb{E}_\mu\left[\sum_{t=0}^{+\infty}\gamma^t (c_0\cdots c_{t-1})\rho_t(r_t+\gamma V(s_{t+1})-V(s_t))|s_0=s\right].\end{aligned} \tag{9}$$

Theorem 5. *Let c, ρ be such that for any $s \in \mathcal{S}, a \in \mathcal{A}$*

$$\alpha(s,a) := \rho(s,a) - c(s,a)\,\mathbb{E}_{a'\sim\mu(\cdot|s')}\left[\rho(s',a')\right] \geq 0, \tag{10}$$

where s' is the state obtained from s after issuing action a. Assume also that $\mathbb{E}_\mu \rho_0 \geq \beta \in (0,1]$. Then the operator $\mathcal{R}$ is C_∞^K contraction with a unique fixed point $V^{\tilde\pi}$ which is a value function of a policy $\tilde\pi$ given by

$$\tilde\pi(a|s) := \frac{\rho(s,a)\mu(a|s)}{\sum_{b\in\mathcal{A}}\rho(s,b)\mu(b|s)}. \tag{11}$$

The contraction constant η is smaller than $1-(1-\gamma)\beta < 1$.

Remark 6. *We note that the proof [8, Theorem 1] has some glitches. A careful examination reveals that the analysis of the first displayed equation on page 12 in [8] is not correct. We fix it by making introducing filtration and analyzing the measurability of explicitly and argue that such an approach makes the proof more clear*

Proof. Let $\tilde c_t := \prod_{u=0}^t c_u$ with the convention $\tilde c_t = 1$ for $t \leq 0$. We use the same convention for ρ_s. It is convenient to express $\mathcal{R}$ in the following form

$$\begin{aligned}\mathcal{R}V(s) =&(1-\mathbb{E}_\mu\rho_0)V(s)+\\ &+\mathbb{E}_\mu\left[\sum_{t=0}^{+\infty}\gamma^t\tilde c_{t-1}\left(\rho_t r_t+\gamma[\rho_t - c_t\rho_{t+1}]V(s_{t+1})\right)\right].\end{aligned}$$

Fix $V_1, V_2 \in C_\infty^K$ and put $C_\infty^K \ni \Delta V := V_1 - V_2$. For $s \in \mathcal{S}$ we write

$$\begin{aligned}\mathcal{R}V_1(s) - \mathcal{R}V_2(s) =&(1-\mathbb{E}_\mu\rho_0)\Delta V(s)+\\ &+\mathbb{E}_\mu\left[\sum_{t=0}^{+\infty}\gamma^{t+1}\tilde c_{t-1}[\rho_t - c_t\rho_{t+1}]\Delta V(s_{t+1})\right].\end{aligned}$$

We define filtration $\{\mathcal{F}_t\}_{t=0}^{+\infty}$ by

$$\mathcal{F}_t := \sigma((s_0, a_0, \ldots, a_{t-1}, s_t)).$$

We rewrite in terms of conditional expectation

$$\begin{aligned}a :=&\mathbb{E}_\mu\left[\tilde c_{t-1}[\rho_t - c_t\rho_{t+1}]\Delta V(s_{t+1})\right]\\ =&\mathbb{E}_\mu\left(\mathbb{E}_\mu\left[\tilde c_{t-1}[\rho_t - c_t\rho_{t+1}]\Delta V(s_{t+1})|\mathcal{F}_{t+1}\right]\right).\end{aligned}$$

Clearly, all terms except for ρ_{t+1} are $\mathcal{F}_{t+1}$ measurable, thus

$$a = \mathbb{E}_\mu\left[\tilde c_{t-1}[\rho_t - c_t\,\mathbb{E}_\mu(\rho_{t+1}|\mathcal{F}_{t+1})]\Delta V(s_{t+1})\right].$$

Recall (10) and observe that $\rho_t - c_t\,\mathbb{E}_\mu(\rho_{t+1}|\mathcal{F}_{t+1}) = \alpha(s_t, a_t) =: \alpha_{t+1}$. Let us also put by convention $\alpha_0 = 1 - \mathbb{E}_\mu\rho_0$. Thus shifting indices we get

$$\mathcal{R}V_1(s) - \mathcal{R}V_2(s) = \mathbb{E}_\mu\left[\sum_{t=0}^{+\infty}\gamma^t\tilde c_{t-2}\alpha_t\Delta V(s_t)\right].$$

By our assumptions we have $\tilde c_t, \alpha_t \geq 0$ and thus we get

$$|\mathcal{R}V_1(s) - \mathcal{R}V_2(s)| \leq \|V_1 - V_2\|_{C_\infty^K}\,\mathbb{E}_\mu\left[\sum_{t=0}^{+\infty}\gamma^t\tilde c_{t-2}\alpha_t\right].$$

We are left with rather straightforward calculations (assume by convention that $\rho_{-1} = 1$).

$$\begin{aligned}\mathbb{E}_\mu\left[\sum_{t=0}^{+\infty}\gamma^t\tilde c_{t-2}\alpha_t\right] &= \mathbb{E}_\mu\left[\sum_{t=0}^{+\infty}\gamma^t\tilde c_{t-2}\rho_{t-1}\right] - \mathbb{E}_\mu\left[\sum_{t=0}^{+\infty}\gamma^t\tilde c_{t-1}\,\mathbb{E}_\mu(\rho_t|\mathcal{F}_t)\right]\\ &= \mathbb{E}_\mu\left[\sum_{t=0}^{+\infty}\gamma^t\tilde c_{t-2}\rho_{t-1}\right] - \mathbb{E}_\mu\left[\sum_{t=0}^{+\infty}\gamma^t\tilde c_{t-1}\rho_t\right]\\ &= 1+\gamma\,\mathbb{E}_\mu\left[\sum_{t=0}^{+\infty}\gamma^t\tilde c_{t-1}\rho_t\right] - \mathbb{E}_\mu\left[\sum_{t=0}^{+\infty}\gamma^t\tilde c_{t-1}\rho_t\right]\\ &= 1+(\gamma-1)\,\mathbb{E}_\mu\left[\sum_{t=0}^{+\infty}\gamma^t\tilde c_{t-1}\rho_t\right] \leq 1+(\gamma-1)\beta.\end{aligned}$$

The last inequality follows by dropping all summands except $t = 0$.

Now, we are to determine the unique fixed point. Recall (11), we calculate

$$\begin{aligned}&\mathbb{E}_\mu\left[\rho_t\left(r_t+\gamma V^{\tilde\pi}(s_{t+1}) - V^{\tilde\pi}(s_t)\right)|s_t\right]\\ &= c\,\mathbb{E}_{\tilde\pi}\left[\rho_t\left(r_t+\gamma V^{\tilde\pi}(s_{t+1}) - V^{\tilde\pi}(s_t)\right)|s_t\right]\\ &= 0,\end{aligned}$$

where $c = \sum_{b\in\mathcal{A}}\rho(s,b)\mu(b|s)$ is the normalizing constant and the second equality hold by the Bellman equation. ☐

B CENTRALIZED TRAINING AND DECENTRALIZED EXECUTION

In a multi-agent problem, the agents take actions based on their local observations s_i, according to their policies π_i. **Decentralized training** is a simple approach of learning π_i, using any (single-agent) RL algorithm separately for each agent. To better utilize the structure of a multi-agent problem, a paradigm of **centralized training and decentralized execution** was introduced. The key idea is to centralize the learning process of all the agents – use shared knowledge provided by observations and any other information available during training to more effectively optimize the decentralized policies. It is now considered a leading paradigm and is usually chosen off-the-shelf.

Recently the authors of [20] suggested that this should be reinvestigated. Sharing the knowledge in centralized training is indeed beneficial, but the paradigm of independent training also has advantages. The multi-agent problems lack the strong theoretical guarantees associated with the standard case. For example, if we train all the agents simultaneously, the environment changes during training from every agent's perspective. Such non-stationarity can destabilize standard algorithms. According to [20], decentralized learning partially mitigates this issue – for every agent, a separate value network is trained; thus it averages much of the stochasticity in the environment, producing more stable estimates. On the other hand, because of the limited information, these values are less accurate. Therefore choosing between centralized and decentralized training is a tradeoff.

To address these issues, we compared MA-Trace with its decentralized version, DecMA-Trace. However, in the StarCraft Multi-Agent Challenge, the decentralized version learns much slower and fails to reach good performance; see Figure 8.

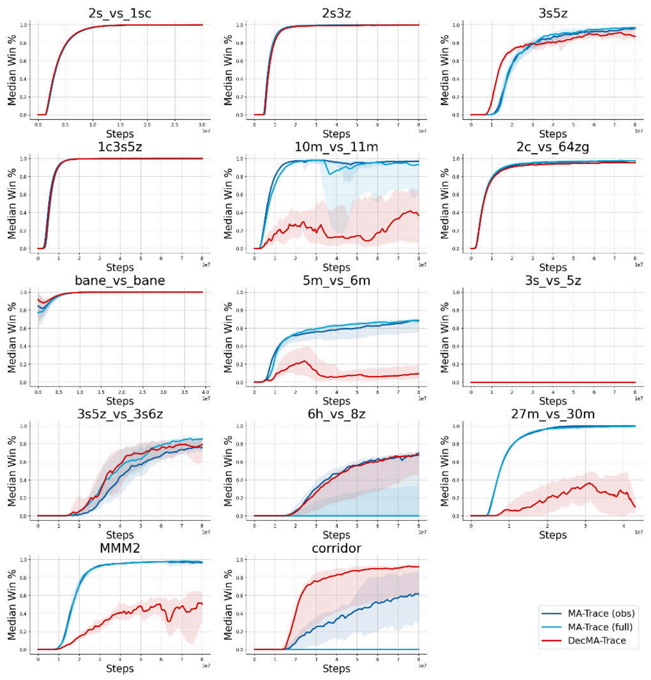


Figure 8: Comparison of MA-Trace and DecMA-Trace on the StarCraft Multi-Agent Challenge tasks.

C NETWORKS' ARCHITECTURES

In our experiments, we use feed-forward networks with two hidden layers of 64 neurons and ReLU activations (without normalization). This is smaller than the networks used in benchmark [40] and [26], but we found this sufficient to obtain good results. In this work, we focus on algorithmic aspects rather than tuning architectures. However, for completeness, we include a discussion of some most popular design choices.

C.1 Sharing parameters

Our default (and most effective) scheme uses a single shared network for the actors and a separate one for the critic. We experimented with sharing feature extractors between the agents and the critic, which performed worse. We also checked the performance when training separate networks for all the components. Interestingly, with this approach, the learning is much faster on the easy tasks, such as *3s5z* or *2c_vs_64zg*, but completely fails on the hardest ones; see Figure 9.

C.2 ID experiments

Sharing the agents' networks, as described in Section C.1, can lead to poor results if agents are not homogenous (for example, the behavior of shooting units should differ from melee units). This might be circumvented by adding units' characteristics (which is a default in SMAC) or enriching the observations with one-hot encoded ID. In Figure 10 we present the results of experiments in which we use these two mechanisms. We observe some improvements, which are, however, relatively minor.

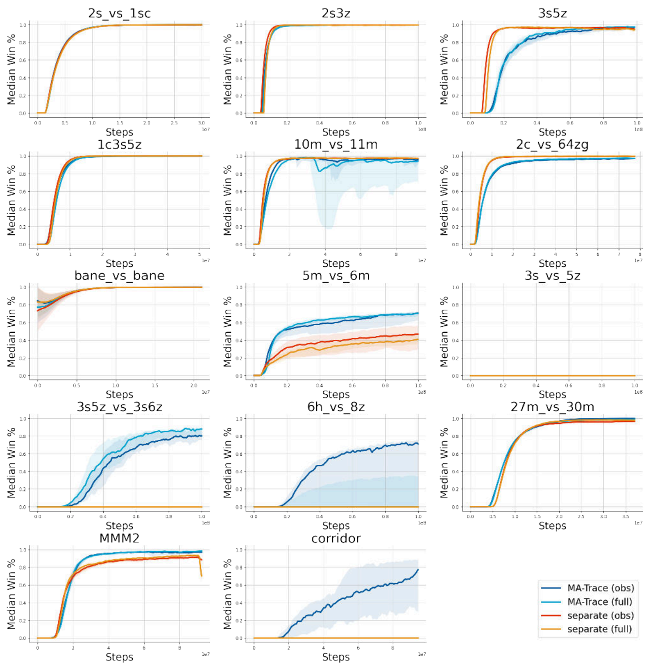


Figure 9: Comparison of standard MA-Trace (with a shared actor network) and its ablation with separate networks.

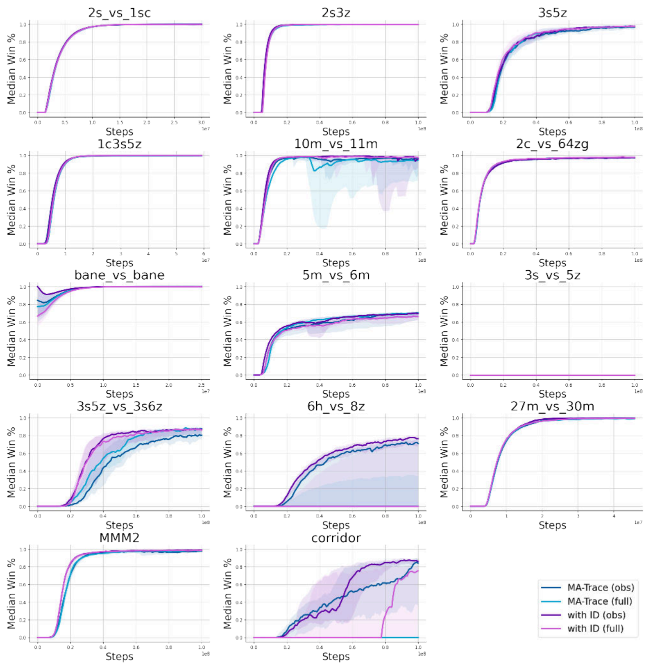


Figure 10: Comparison of MA-Trace conditioned on observation with and without (default) agents' IDs.

C.3 Agents with memory

The units in StarCraft Multi-agent Challenge have limited sight range, thus the environment is partially observable. A common approach to mitigate such an issue, is to give the agents memory, e.g. by using recurrent networks or frame stacking. The authors of [26] show that using recurrent policy is required to achieve good results in the hardest tasks. However, our experiments show that memoryless policies can be successfully trained to solve all the tasks (possibly except *3s_vs_5z*, see Section F.1).

To implement a simple memory it is enough to pass a few previous observations concatenated ("framestack"). We experimented with both observation-based and state-based critics. Figures 11 and 12 show the progress of training memory-equipped agents. One can observe hardly any difference in most tasks. On the easy task *2c_vs_64zg*, using memory leads to faster training. What is particularly interesting, on the *corridor*, task the stacked versions learn much faster. This may be related to the strategy learned by MA-Trace, in which at some point a group of units has to hide and wait (see Section F.2 for detailed description). Execution of this strategy is probably easier when using at least short memory. However, on some other tasks, such as *5m_vs_6m*, training with memory is much less stable and effective.

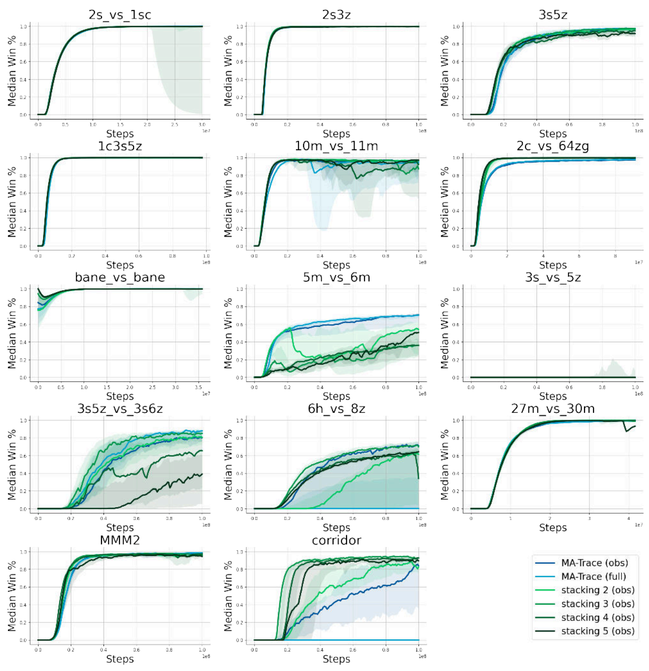


Figure 11: Comparison of MA-Trace with growing number of stacked frames. The critic networks in ablations are conditioned on aggregated observations.

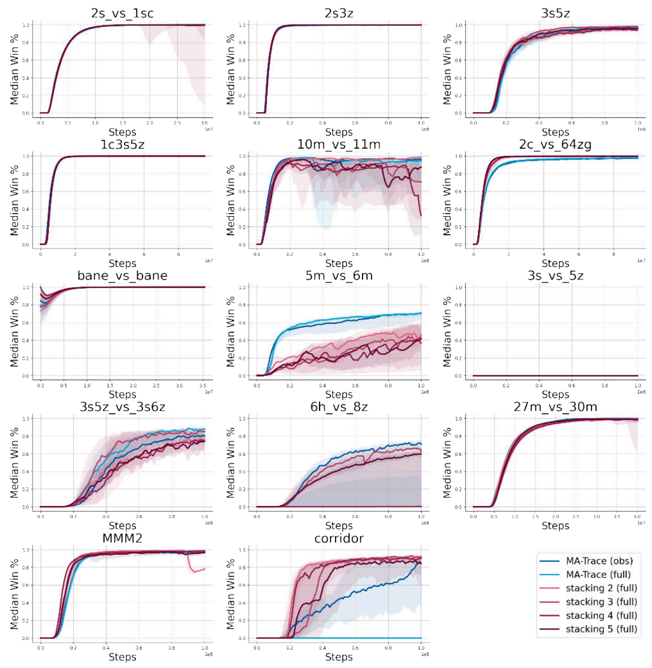


Figure 12: Comparison of MA-Trace with growing number of stacked frames. The critic networks in ablations are conditioned on environment state.

D TRAINING DETAILS

D.1 SEED RL

We base our code on SEED RL [7], which is an open-source framework for distributed learning, licensed under the Apache License, Version 2.0. This library provides an implementation of V-Trace for single-agent environments.

D.2 Hyperparameters

In our experiments the parameters crucial to the final performance were: *learning rate* and *entropy cost*. For other hyperparameters, we use the default values provided by SEED. The most relevant, common for all the experiments, are listed in Table 2.

hyperparameter	value
batch size	32
optimizer	Adam
gamma	0.99
ρ	1.0
c	1.0
λ	1.0
initial entropy cost	1.0
target entropy	10^{-5}

Table 2: Default parameters for our experiments.

We searched for the best learning rate in the set $[3.5 \cdot 10^{-3}, 2.5 \cdot 10^{-3}, 1.5 \cdot 10^{-3}, 10^{-3}, 7 \cdot 10^{-4}, 5 \cdot 10^{-4}, 3.5 \cdot 10^{-4}]$. We experimented with adding learning rate schedules with warmup and decay, though no such scheme appeared to be beneficial.

Another hyperparameter that has a strong influence on the final results is *entropy cost*. By default, SEED sets it to a constant value of $2.5 \cdot 10^{-4}$ and allows to use annealing. We found that aggressive exploration at the beginning is crucial to reach good results. We annealed *entropy cost* from 1 towards 10^{-5}. The speed of adjustment was tuned in the set $[10, 5, 2.5, 1, 0.5]$.

In Table 3, we show the best parameters for our experiments, found by the above grid-search. Unlike in some other works, we found no advantage of using gradient clipping; thus, we leave the gradients not clipped.

hyperparameter	MA-Trace (obs)	MA-Trace (full)
learning rate	10^{-3}	$7 \cdot 10^{-4}$
entropy adjustment	10	10

Table 3: Specific parameters for our experiments.

D.3 Infrastructure used

The typical configuration of a computational node used in our experiments was: the Intel Xeon E5-2697 2.60GHz processor with 128GB memory. On a single node, we ran one experiment with 30 workers. A typical experiment was run for about 20h. For the final evaluation, we extended the training to 3 days, which is usually equivalent to about $3 \cdot 10^8$ environment steps. We did not use GPUs; we found that with the relatively small size of the network it offers only slight wall-time improvement while generating substantial additional costs.

E ABLATIONS

E.1 Critic comparison

During centralized training, the critic network in MA-Trace algorithm uses any information available. A natural choice is to aggregate the observations available to the agents, and we denote this version as MA-Trace (obs). This might not be a sufficient statistic of the (Markov) state of the environment. SMAC provides additional access to such a state description, which we use in MA-Trace (full). Intuitively, using complete information should be advantageous.

As shown in Figure 16, on most tasks, both the versions do not differ much. On *3s5z_vs_3s6z* there is a small advantage on the full-state side. However, MA-Trace (full) shows little progress on *6h_vs_8z* and *corridor*, as opposed to MA-Trace (obs). Therefore we consider MA-Trace (obs) to be our default version.

Such behavior is a bit counter-intuitive. We speculate that some information available in the agents' observation is not easily accessible (computable) for the full state. To verify this, we compared the two versions with another, MA-Trace (obs+full), which uses both the aggregated observations and the full state. As we can see in Figure 16, it trains similarly to MA-Trace (obs), without significant advantage on any task. Moreover, on the hardest tasks, the training progresses a bit slower. We leave a precise explanation of this behavior as future work.

E.2 Importance sampling ablation

We claim that the strong performance of MA-Trace is due to using importance weights. To verify this statement, we compared MA-Trace with its ablation without importance weights. The training curves for all the tasks are shown in Figure 13. Without the corrections, the training is unstable and reaches good performance only on the easiest tasks. The difference gets even larger when using more compute workers. One notable exception is the *corridor* task, in which the ablated version trains faster than MA-Trace, though eventually both reach similar results.

E.3 Scaling experiments

MA-Trace utilizes importance sampling to reduce the shifts arising naturally in distributed training (when behavioral policies lag behind the target policy). This allows our algorithm to be highly scalable. Our experiments confirm that MA-Trace scales almost linearly on at least 70 workers.

As shown in Section E.2, the importance weights indeed enable stable training, even when distributed to many workers. What is equally important, our experiments show that scaling does not affect the learned policies significantly.

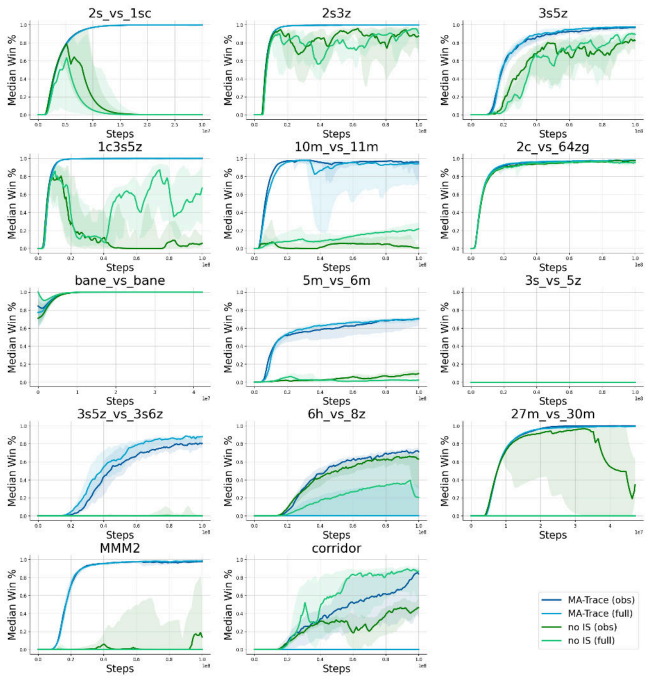


Figure 13: Comparison of MA-Trace with and without importance sampling.

F STARCRAFT MULTI-AGENT CHALLENGE

To evaluate MA-Trace, we use StarCraft Multi-Agent Challenge (SMAC) [28]. It is a standard benchmark for multi-agent algorithms, used e.g. by [10, 26, 40] and many others. The challenge is based on a popular real-time strategy game StarCraft II and consists of 14 micromanagement tasks. Each task is a small arena in which two teams, controlled by the player by the built-in AI, fight against each other. The goal in every task is to defeat (kill) all the enemy units.

Units belong to one of the three races: Protoss, Terran or Zerg. Additionally, they are divided on a number of classes with unique characteristic, such as speed, shooting range, fire power etc. In each turn they can move or attack an enemy in their shooting range. A unit is considered defeated if its health drops to 0. A defeated unit remain inactive.

SMAC provides a variety of different tasks. In the easier tasks, the opponents control the same forces. Therefore to win such a game, it is enough to coordinate slightly better than the built-in AI. In the harder tasks, however, the computer starts with a stronger squad. This can be a minor advantage, such as in the task *10 marines vs 11 marines*, or quite a big difference. In particularly hard scenarios, such as *corridor*, it is unreasonable for the player to engage in an open fight, so it is essential to develop a long-term strategy to obtain a positional advantage.

For the hardest tasks, the authors of the challenge consider the so-called microtricks sufficient to win consistently. However, it appears that MA-Trace in some cases develop unexpected strategies, see e.g. Section F.2. What is particularly interesting, our algorithm manages to learn some techniques associated with professional human players, such as focusing fire, withdrawing low-health units, hit-and-run, sacrificing a unit to deceive the opponent and others.

Units in the game have limited sight range, which makes the environment partially observable. The observations received by individual agents contain information about all the visible units (including themselves) – their health, energy, position, class, and other relevant features. All the units beyond the sight range are marked as dead (i.e., defeated and invisible units are not distinguished). It is possible that the aggregated observations do not provide full information, for there can be enemy units hidden beyond the sight range of any ally. Therefore, to facilitate centralized training, SMAC provides additional access to the full state of the environment.

Learning from binary reward (win/lose) is prohibitively hard in most tasks. Therefore SMAC provides dense rewards to enable training. The team receives additional points for damage dealt and defeating a unit. This scheme is arguably natural and leads to successful training. However, in some cases, it might reinforce undesired behaviors, see, e.g., Section F.1.

F.1 *3s_vs_5z* task

MA-Trace masters all the tasks except *3s_vs_5z*, see Table 1. In that scenario, we control 3 Stalkers fighting against 5 Zealots. Stalkers can attack the enemy from a distance; however, they are no match for Zealots in close combat. A strategy to gain an advantage is to shoot the enemies while they are away and flee when they get close.

All the units in this task are Protoss, so they all have protective shields. The shields absorb some amount of damage, until they are down. However, as opposed to regular health, the shields regenerate slowly with time. As dealing damage yields rewards, it might be beneficial to keep enemy alive infinitely. Apparently, MA-Trace learns to this strategy.

Figure 14 shows an example of the winning rate and episode rewards. As we can see, after a short training, our algorithm wins almost every time. Further, it discovers that the reward scheme can be exploited – at some point, the mean return increases fast, while the actual win rate decreases and becomes unstable.

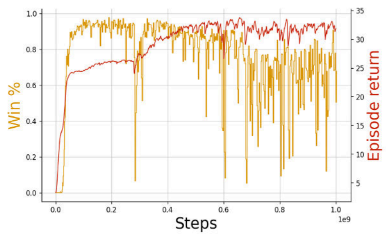


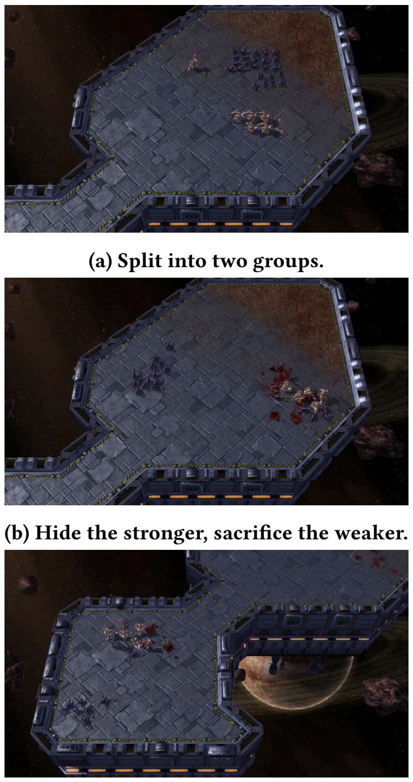
Figure 14: The winning rate and episode rewards in a training for the task *3s_vs_5z*.

F.2 *Corridor* task

Another *super-hard* scenario (according to [26]) is *corridor*. In this task, we control a team of 6 Zealots against 24 enemy Zerglings. Though Zealots are far more powerful in combat, they are outnumbered; thus, the open fight is clearly an unreasonable strategy. However, the fighting arena contains a narrow passage. The authors of SMAC suggest that a winning strategy is to gather the forces in that passage, where the number of enemy units should lose its importance (possibly inspired by the Battle of Thermopylae).

However, MA-Trace develops an alternative interesting strategy. Firstly, our forces split into two groups. One (strong) hides in the corner, where it easily defeats a few enemies, while the other (one or two units) attracts majority of the enemies to the other side and sacrifice itself. After defeating the second group, the enemies pass to the far side of the arena and wait, unaware of the hidden group. Then the strong group attack them from behind and defeat the Zerglings one by one.

The strategy is outlined in Figure 15.



(a) Split into two groups.

(b) Hide the stronger, sacrifice the weaker.

(c) Attack from behind.

Figure 15: The consecutive parts of strategy executed by MA-Trace on the *corridor* task. The yellow soldiers are our units, while the blue creatures are enemy units.

G FULL TRAINING DATA

In this section, we provide the main practical results of our work – complete training data for all standard versions of MA-Trace on all the SMAC tasks. They were trained for three days or until convergence. We report the median win rates and interquartile ranges.

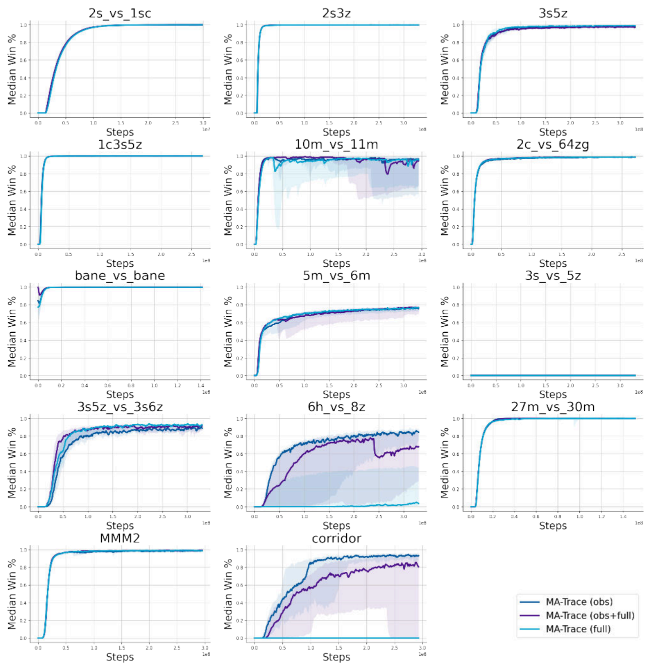


Figure 16: Training curves of main MA-Trace versions on all the tasks available in StarCraft Multi-Agent Challenge.

Subgoal Search For Complex Reasoning Tasks

Konrad Czechowski[*]
University of Warsaw
k.czechowski@mimuw.edu.pl

Tomasz Odrzygóźdź[*]
University of Warsaw
tomaszo@impan.pl

Marek Zbysiński
University of Warsaw
m.zbysinski@students.mimuw.edu.pl

Michał Zawalski
University of Warsaw
m.zawalski@uw.edu.pl

Krzysztof Olejnik
University of Warsaw
k.olejnik3@student.uw.edu.pl

Yuhuai Wu
University of Toronto,
Vector Institute
ywu@cs.toronto.edu

Łukasz Kuciński
Polish Academy of Sciences
lkucinski@impan.pl

Piotr Miłoś
Polish Academy of Sciences,
University of Oxford,
deepsense.ai
pmilos@impan.pl

Abstract

Humans excel in solving complex reasoning tasks through a mental process of moving from one idea to a related one. Inspired by this, we propose Subgoal Search (kSubS) method. Its key component is a learned subgoal generator that produces a diversity of subgoals that are both achievable and closer to the solution. Using subgoals reduces the search space and induces a high-level search graph suitable for efficient planning. In this paper, we implement kSubS using a transformer-based subgoal module coupled with the classical best-first search framework. We show that a simple approach of generating k-th step ahead subgoals is surprisingly efficient on three challenging domains: two popular puzzle games, Sokoban and the Rubik's Cube, and an inequality proving benchmark INT. kSubS achieves strong results including state-of-the-art on INT within a modest computational budget.

1 Introduction

Reasoning is often regarded as a defining property of advanced intelligence [39, 18]. When confronted with a complicated task, humans' thinking process often moves from one idea to a related idea, and the progress is made through milestones, or *subgoals*, rather than through atomic actions that are necessary to transition between subgoals [15]. During this process, thinking about one subgoal can lead to a possibly diverse set of subsequent subgoals that are conceptually reachable and make a promising step towards the problem's solution. This intuitive introspection is backed by neuroscience evidence [20], and in this work, we present an algorithm that mimics this process. Our approach couples a deep learning generative subgoal modeling with classical search algorithms to allow for successful planning with subgoals. We showcase the efficiency of our method on the following complex reasoning tasks: two popular puzzle games Sokoban and the Rubik's Cube, and an inequality theorem proving benchmark INT [54], achieving the state-of-the-art results in INT and competitive results for the remaining two.

The deep learning revolution has brought spectacular advancements in pattern recognition techniques and models. Given the hard nature of reasoning problems, these are natural candidates to provide

[*]equal contribution

search heuristics [4]. Indeed, such a blend can produce impressive results [43, 44, 36, 1]. These approaches seek solutions using elementary actions. Others, e.g. [29, 33, 23], utilize variational subgoals generators to deal with long-horizon visual tasks. We show that these ideas can be pushed further to provide algorithms capable of dealing with combinatorial complexity.

We present Subgoal Search (kSubS) method and give its practical implementations: MCTS-kSubS and BF-kSubS. kSubS consists of the following four components: planner, subgoal generator, a low-level policy, and a value function. The planner is used to search over the graph induced by the subgoal generator and is guided by the value function. The role of the low-level policy is to prune the search tree as well as to transition between subgoals. In this paper, we assume that the generator predicts subgoals that are k step ahead (towards the solution) from the current state, and to emphasize this we henceforth add k to the method's abbreviation. MCTS-kSubS and BF-kSubS differ in the choice of the search engine: the former uses Monte-Carlo Tree Search (MCTS), while the latter is backed by Best-First Search (BestFS). We provide two sets of implementations for the generator, the low-level policy, and the value functions. The first one uses transformer architecture [48] for each component, while the second utilizes a convolutional network for the generator and the value function, and the classical breadth-first search for the low-level policy. This lets us showcase the versatility and effectiveness of the approach.

The subgoal generator lies at the very heart of Subgoal Search, being an implementation of reasoning with high-level ideas. To be useful in a broad spectrum of contexts, the generator should be implemented as a learnable (generative) model. As a result, it is expected to be imperfect and (sometimes) generate incorrect predictions, which may turn the search procedure invalid. Can we thus make planning with learned subgoals work? In this paper, we answer this question affirmatively: we show that the autoregressive framework of transformer-based neural network architecture [49] leads to superior results in challenging domains.

We train the transformer with the objective to predict the k-th step ahead. The main advantages of this subgoal objective are simplicity and empirical efficiency. We used expert data to generate labels for supervised training. When offline datasets are available, which is the case for the environments considered in this paper[2], such an approach allows for stable and efficient optimization with high-quality gradients. Consequently, this method is often taken when dealing with complex domains (see e.g. [41, 51]) or when only an offline expert is available[3]. Furthermore, we found evidence of out-of-distribution generalization.

Finally, we formulate the following hypothesis aiming to shed some light on why kSubS is successful: we speculate that subgoal generation may alleviate errors in the value function estimation. Planning methods based on learning, including kSubS, typically use imperfect value function-based information to guide the search. While traditional low-level search methods are susceptible to local noise, subgoal generation allows for evaluations of the value functions at temporally distant subgoals, which improves the signal-to-noise ratio and allows a "leap over" the noise.

To sum up, our contributions are:

1. We propose Subgoal Search method with two implementations: MCTS-kSubS, BF-kSubS. We demonstrate that our approach requires a relatively little search or, equivalently, is able to handle bigger problems. We also observe evidence of out-of-distribution generalization.
2. We provide evidence that a transformer-based autoregressive model learned with a simple supervised objective to predict states k-th step ahead is an effective tool to generate valid and diverse subgoals.
3. We show in our experiments that using subgoal planning help to might mitigate the negative influence of value function errors on planning.

We provide the code of our method and experiment settings at `https://github.com/subgoal-search/subgoal-search`, and a dedicated website `https://sites.google.com/view/subgoal-search`.

[2]The dataset for INT or Sokoban can be easily generated or are publicly available. For the Rubik's Cube, we use random data or simple heuristic (random data are often sufficient for robotic tasks and navigation.)

[3]For example, the INT engine can easily generate multiple proves of random statements, but *cannot* prove a given theorem.

2 Related work

In classical AI, reasoning is often achieved by *search* ([39]). Search rarely can be exhaustive, and a large body of algorithms and heuristics has been developed over the years, [39, Section 3.5]. It is hypothesized that progress can be achieved by combining search with learning [4]. Among notable successful examples of this approach are Alpha Zero [42], or solving Rubik's cube using a learned heuristic function to guide the A* algorithm (see [1]).

An eminent early example of using goal-directed reasoning is the PARADISE algorithm ([52]). In deep learning literature, [24] was perhaps the first work implementing subgoal planning. This was followed by a large body of work on planning with subgoals in the latent space for visual tasks ([23, 30, 33, 29, 21, 9, 34]) or landmark-based navigation methods ([40, 26, 14, 45, 55]).

The tasks considered in the aforementioned studies are often quite forgiving when it comes to small errors in the subgoal generation. This can be contrasted with complex reasoning domains, in which even a small variation of a state may drastically change its meaning or render it invalid. Thus, neural networks may struggle to generate semantically valid states ([4, Section 6.1]).

Assuming that this problem was solved, a generated subgoal still remains to be assessed. The exact evaluation may, in general, require exhaustive search or access to an oracle (in which case the original problem is essentially solved). Consequently, it is unlikely that a simple planner (e.g., one unrolling independent sequences of subgoals [9]) will either produce an informative outcome, could be easily improved using only local changes via gradient descent [30], or cross-entropy method (CEM) [29, 33, 34]. Existing approaches based, which are based on more powerful subgoal search methods, have their limitations, on the other hand. [13] is perhaps the closest to our method and uses MCTS to search the subgoal-induced graph. However, it uses a predefined (not learned) predicate function as a subgoal generator, limiting applicability to the problems with available high-quality heuristics. Learned subgoals are used in [31], a method of hierarchical planning. That said, the subgoal space needs to be relatively small for this method to work (or crafted manually to reduce cardinality). To the best of our knowledge, our algorithm is the first domain agnostic hierarchical planner for combinatorially complex domains.

More generally, concepts related to goals and subgoals percolated to reinforcement learning early on, leading, among others, to prominent ideas like hindsight [22], hierarchical learning [47, 7] or the Horde architecture [46]. Recently, with the advent of *deep* reinforcement learning, these ideas have been resurfacing and scaled up to deliver their initial promises. For example, [50] implements ideas of [7] and a very successful hindsight technique [2] is already considered to be a core RL method. Further, a recent paper [35] utilizes a maximum entropy objective to select achievable goals in hindsight training. Apart from the aforementioned hierarchical planning, the idea to use neural networks for subgoal generation was used to improve model-free reinforcement learning agents [11, 6] and imitation learning algorithms [32].

3 Method

Our method, Subgoal Search (kSubS), is designed for problems, which can be formulated as a search over a graph with a known transition model. Formally, let $G = (\mathcal{S}, \mathcal{E})$ be a directed graph and $\tilde{\mathcal{S}} \subset \mathcal{S}$ be the set of success states. We assume that, during the solving process, the algorithm can, for a given node g, determine the edges starting at g and check if $g \in \tilde{S}$.

Subgoal Search consists of four components: planner, subgoal generator, low-level policy, and a value function. The planner, coupled with a value function, is used to search over the graph induced by the subgoal generator. Namely, for each selected subgoal, the generator allows for sampling the candidates for the next subgoals. Only these reachable by the low-level policy are used. The procedure continues until the solution is found or the computational budget is exhausted. Our method searches over a graph $\tilde{G} = (\mathcal{S}, \mathcal{E}_s)$, with the edges $\mathcal{E}_s$ given by the subgoal generator. Provided a reasonable generator, paths to $\tilde{\mathcal{S}}$ are shorter in $\tilde{G}$ than in G and thus easier to find.

In this paper we provide BestFS- and MCTS- backed implementations kSubS. Algorithm 1 presents BF-kSubS; see Algorithm 4 in Appendix A.1 for MCTS-kSubS.

For INT and Rubik's Cube, we use transformer models (see Appendix B.1) in all components other than the planner. For Sokoban, we use convolutional networks (see Appendix B.2). While transformers could also be used in Sokoban, we show that a simplified setup already achieves strong results. This showcases that Subgoal Search is general enough to work with different design choices. In Section 4.2 we describe datasets used train these neural models.

Algorithm 1 Best-First Subgoal Search (BF-kSubS)

```
Require:  $C_1$      max number of nodes
          $V$        value function network
          SOLVED     predicate of solution
function SOLVE(s_0)
    T.PUSH(($V$(s_0), s_0))           ▷ T is priority queue
    paths[s_0] ← []                   ▷ paths is dict of lists
    seen.ADD(s_0)                     ▷ seen is set
    while $0 <$ LEN(T) and LEN(seen) $< C_1$ do
        _, s ← T.EXTRACT_MAX()
        subgoals ← SUB_GENERATE(s)
                                      ▷ see Algorithm 3
        for s' in subgoals do
            if s' in seen then continue
            seen.ADD(s')
            actions ← GET_PATH(s, s')
                                      ▷ see Alg 2 or Alg 9
            if actions.EMPTY() then
                continue
            T.PUSH(($V$(s'), s'))
            paths[s'] ← paths[s] + actions
            if SOLVED(s') then
                return paths[s']
    return False
```

Algorithm 2 Low-level conditional policy

```
Require:  $C_2$   steps limit
          $\pi$   low-level conditional
                  policy network
          $M$     model of the environment
function GET_PATH(s_0, subgoal)
    step ← 0
    s ← s_0
    action_path ← []
    while step $< C_2$ do
        action ← $\pi$.PREDICT(s, subgoal)
        action_path.APPEND(action)
        s ← $M$.NEXT_STATE(s, action)
        if s = subgoal then
            return action_path
        step ← step + 1
    return []
```

Subgoal generator Formally, it is a mapping $\rho\colon \mathcal{S} \to \mathbf{P}(\mathcal{S})$, where $\mathbf{P}(\mathcal{S})$ is a family of probability distributions over the environment's state space $\mathcal{S}$. More precisely, let us define a trajectory as a sequence of state-action pairs $(s_0, a_0), (s_1, a_1), \dots, (s_n, a_n)$, with $(s_i, a_i) \in \mathcal{S} \times \mathcal{A}$, where $\mathcal{A}$ stands for the action space and n is the trajectory's length. We will say that the generator predicts k-*step ahead* subgoals, if at any state s_ℓ it aims to predict $s_{\min(\ell+k,n)}$. We show, perhaps surprisingly, that this simple objective is an efficient way to improve planning, even for small values of k, i.e. $k \in \{2, 3, 4, 5\}$.

Operationally, the subgoal generator takes as input an element of the **s** $\in \mathcal{S}$ and returns *subgoals*, a set of new candidate states expected to be closer to the solution and is implemented by Algorithm 3. It works as follows: first, we generate C_3 subgoal candidates with SUB_NET_GENERATE. Then, we prune this set to obtain a total probability greater than C_4.

For INT and Rubik's Cube, we represent states as sequences modeled with a transformer. Following the practice routinely used in language modeling, [48], SUB_NET_GENERATE employs beam search to generate a set of high likelihood outputs.[4] With the same goal in mind, for Sokoban, we use another method described Appendix C.1. SUB_NET_GENERATE uses the subgoal generator network, which is trained in a supervised way on the expert data, with training examples being: s_ℓ (input) and $s_{\min(\ell+k,n)}$ (label), see Appendix D for details.

Low-level conditional policy Formally, it is a mapping $\pi\colon \mathcal{S} \times \mathcal{S} \to \mathcal{A}^*$. It is used to generate a sequence of actions on how to reach a subgoal starting from a given initial state. Operationally, it may return an empty sequence if the subgoal cannot be reached within C_2 steps, see Algorithm 2. This is used as a pruning mechanism for the *subgoals* set in Algorithm 1.

[4]In language modeling, typically, only one beam search output is used. In our case, however, we utilize all of them, which turns out to be a diverse set of subgoals.

In INT and Rubik's Cube, we use Algorithm 2, which utilizes low-level policy network π. Similarly to the subgoal generator, it is trained using expert data in a supervised fashion, i.e. for a pair $(s_\ell, s_{\min(\ell+i,n)})$, with $i \leq k$, its objective is to predict a_ℓ.

When the branching factor is small, the low-level policy can be realized by a simple breadth-first search mechanism, see Algorithm 9 in Appendix, which we illustrate on Sokoban.

Algorithm 3 Subgoal generator

```
Require:  $C_3$   number of subgoals
          $C_4$   target probability
          $\rho$   subgoal generator network
  function SUB_GENERATE(s)
      subgoals $\leftarrow \emptyset$
      states, probs $\leftarrow$ SUB_NET_GENERATE($\rho$, s; $C_3$)
                    $\triangleright$ (states, probs) is sorted wrt probs
      total_p $\leftarrow 0$
      for state, p $\in$ (states, probs) do
          if total_p $> C_4$ then break
          subgoals.ADD(state)
          total_p $\leftarrow$ total_p + p
      return subgoals
```

Value function Formally, it is a mapping $V\colon \mathcal{S} \to \mathbb{R}$, that assigns to each state a value related to its distance to the solution, and it is used to guide the search (see Algorithm 1 and Algorithm 4). For its training, we use expert data. For each state s_ℓ the training target is negated distance to the goal: $\ell - n$, where n denotes the end step of a trajectory that s_ℓ belongs to.

Planner This is the engine that we use to search the subgoal-induced graph. In this paper, we use BestFS (Algorithm 1) and MCTS (Algorithm 4 in Appendix A.1). The former is a classic planning method, which maintains a priority queue of states waiting to be explored, and greedily (with respect to their value) selects elements from it (see, e.g., [39]). MCTS is a search method that iteratively and explicitly builds a search tree, using (and updating) the collected node statistics (see, e.g., [5]). In this paper, we use an AlphaZero-like [41] algorithm for single-player games.

We note that the subgoal generator can be combined with virtually any search algorithm and can benefit from an additional structure. For example, for domains providing a factored state representation, the width-based methods [25, 12] would likely be stronger search mechanisms.

4 Experiments

In this section, we empirically demonstrate the efficiency of MCTS-kSubS and BF-kSubS. In particular, we show that they vastly outperform their standard ("non-subgoal") counterparts. As a testing ground, we consider three challenging domains: Sokoban, Rubik's Cube, and INT. All of them require non-trivial reasoning. The Rubik's Cube is a well-known 3-D combination puzzle. Sokoban is a complex video puzzle game known to be NP-hard and thus challenging for planning methods. INT [54] is a recent theorem proving benchmark.

4.1 Training protocol and baselines

Our experiments consist of three stages. First, we collect domain-specific expert data, see Section 4.2. Secondly, we train the subgoal generator, low-level conditional policy, and value function networks using the data and targets described in Section 3. For more details see Appendix D. Eventually, we evaluate the planning performance of MCTS-kSubS and BF-kSubS, details of which are presented below. In the second step, we use supervised learning, which makes our setup stable with respect to network initialization, see details in Appendix D.1.3.

As baselines, we use BestFS and MCTS (being a single-player implementation of AlphaZero). In INT and Rubik's Cube, both the algorithms utilize policy networks (trained with behavioral cloning, on the same dataset, which we used to train kSubS). Note that distribution over actions induces a distribution over states; thus the policy network can be regarded as a subgoal generator for $k = 1$. More details about the baselines can be found in Appendix I.

4.2 Search Domains and Datasets

Sokoban is a single-player complex game in which a player controls an agent whose goal is to place boxes on target locations solely by pushing them; without crossing any obstacles or walls. Sokoban has recently been used to test the boundaries in RL [16, 28]. Sokoban is known to be hard [10], mainly due to its combinatorial complexity and the existence of irreversible states. Deciding if a given Sokoban board is solvable is an NP-hard problem [8].

We collect the expert dataset consisting of all successful trajectories occurring during the training of an MCTS agent (using an implementation of [28]). These are suboptimal, especially in the early phases of the training or for harder boards. For both expert training and kSubS evaluation, we generate Sokoban boards following the approach of [37].

Rubik's Cube is a classical 3-dimensional puzzle. It is considered challenging due to the fact that the search space has more than 4.3×10^{18} configurations. Similarly to Sokoban, Rubik's Cube has been recently used as a testing ground for RL methods [1].

To collect the expert dataset, we generate random paths of length 30 starting from the solved cube and take them backward. These backward solutions are highly sub-optimal (optimal solutions are proven to be shorter than 26 [38]).

INT: Inequality Benchmark for Theorem Proving. INT provides a generator of inequalities, which produces mathematical formulas along with their proofs, see [54, Section 3.3]. Proofs are represented as sequences of consecutively applied mathematical axioms (there are $K = 18$ axioms in total). An action in INT is a tuple containing an axiom and its input entities. The action space in this problem can be very large, reaching up to 10^6 elements, which significantly complicates planning.

The INT generator constructs paths by randomly applying axioms starting with a trivial statement. Such a path taken backward constitutes the proof of its final statement (not guaranteed to be optimal). The proof length, denoted by L, is an important hyperparameter regulating the difficulty – we use $5, 10, 15$.

For more details on datasets see Appendix C.

	INT	Sokoban	Rubik
k	3	4	4
C_1	400	5000	1500
C_2	4	4	7
C_3	4	4	3
C_4	1	0.98	1

Table 1: BF-kSubS hyperparameters.

4.3 Main results

In this section, we present our most important finding: Subgoal Search enables for more efficient search and consequently scales up to problems with higher difficulty. Specifically, MCTS-kSubS and BF-kSubS perform significantly better than the respective methods not using subgoals, including state-of-the-art on INT.

In Figure 1, we present the performance of Subgoal Search. We measure the *success rate* as a function of the *search budget*. The success rate is measured on 1000 instances of a given problem (which results in confidence intervals within ± 0.03). For BF-kSubS the search budget is referred to as *graph size* and includes the number of nodes visited by Algorithm 1. For INT and Rubik's Cube, we include both the subgoal generated by `SUB_GENERATE` and the nodes visited by `GET_PATH` (as they induce a significant computational cost stemming from using low-level policy π in Algorithm 2). For Sokoban, we use Algorithm 9 to realize `GET_PATH`, as it has a negligible cost (less than 1% of the total runtime of Algorithm 1), we do not include these nodes into graph size.

For MCTS, we report *MCTS passes*, which is a common metric for MCTS, see details in Appendix A.1.

Below we discuss the results separately for each domain. We provide examples of solutions and generated subgoals in Appendix H.

INT The difficulty of the problems in INT increases fast with the proof length L and the number of accessible axioms. W used $K = 18$; all of available axioms. We observe, that BF-kSubS scales to proofs of length $L = 10$ and $L = 15$, which are significantly harder than $L = 5$ considered in [54], see Table 2. The same holds for MCTS-kSubS, see Appendix A.2.

We check also that MCTS-kSubS vastly outperforms the baseline - AlphaZero algorithm, see Figure 1 (top, left). An MCTS-based agent was also evaluated in [54]. Its implementation uses graph neural

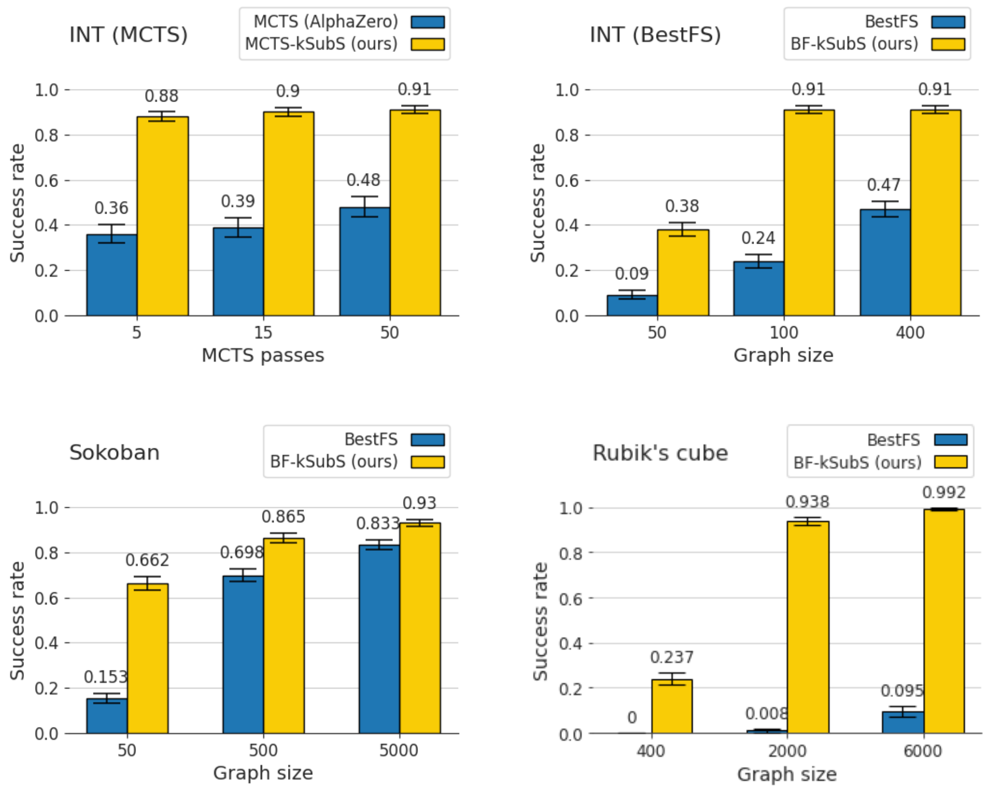


Figure 1: The performance of Subgoal Search. (top, left) comparison on INT (with the proof length 15) to AlphaZero. (top, right) BF-kSubS consistently achieves high performance even for small computational budgets. (bottom, left) similarly on Sokoban (board size 12x12 with 4 boxes) the advantage of BF-kSubS is clearly visible for small budget. (bottom, right) BestFS fails to solve Rubik's Cube, while BF-kSubS can achieve near-perfect performance.

	Proof length	5		10		15	
	Method	BestFS	BF-kSubS (ours)	BestFS	BF-kSubS (ours)	BestFS	BF-kSubS (ours)
Graph size	50	0.82	**0.99**	0.47	**0.97**	0.09	**0.38**
	100	0.89	**0.99**	0.64	**0.99**	0.24	**0.91**
	200	0.92	**0.99**	0.67	**0.99**	0.35	**0.91**
	400	0.93	**0.99**	0.72	**0.99**	0.47	**0.91**

Table 2: INT success rates for various proof lengths and graphs sizes.

networks architectures and achieves 92% success rate for $L = 5$. Our transformed-based baseline is stronger - it solves over 99% instances on the same problem set.

Sokoban Using BF-kSubS allows for significantly higher success rates rates within the same computational budget, see Table 3. Our solution scales well to the board size as big as 20×20; note that 10×10 boards are typically used in deep RL research [16, 37]. Importantly, we observe that already for a small computational budget (graph size 1000) BF-kSubS obtains higher success rates than the expert we used to create the dataset (these are 78%, 67%, 60% for the respective board sizes).

We also tested how the quality of BF-kSubS depends on the size of the training dataset for Sokoban, the results can be found in Appendix F.

Rubik's Cube BF-kSubS solves nearly 100% of cubes, BestFS solve less than 10%, see Figure 1 (bottom, right). This is perhaps the most striking example of the advantage of using a subgoal generator instead of low-level actions. We present possible explanation in Appendix K.

	Board size	12 x 12		16 x 16		20 x 20	
	Method	BestFS	BF-kSubS (ours)	BestFS	BF-kSubS (ours)	BestFS	BF-kSubS (ours)
Graph size	50	0.15	**0.66**	0.04	**0.42**	0.02	**0.46**
	100	0.46	**0.79**	0.23	**0.62**	0.10	**0.55**
	1000	0.75	**0.89**	0.61	**0.79**	0.47	**0.70**
	5000	0.83	**0.93**	0.69	**0.85**	0.58	**0.77**

Table 3: Sokoban success rates for various board sizes (each with 4 boxes).

Out-of-distribution (OOD) generalization OOD generalization is considered to be the crucial ability to make progress in hard combinatorial optimization problems [4] and automated theorem proving [54]. The INT inequality generator has been specifically designed to benchmark this phenomenon. We check that Subgoal Search trained on proofs on length 10 generalizes favourably to longer problems, see Figure 4. Following [54], we speculate that search is a computational mechanism that delivers OOD generalization.

It might be hard to compare computational budgets between various algorithms and their versions. In Appendix E we measure that BF-kSubS and MCTS-kSubS offer very practical benefits, sometimes as much as 7$\times$ faster execution.

4.4 Analysis of k (subgoal distance) parameter

The subgoals are trained to predict states k steps ahead of the current one. Higher k should make planning easier as the search graph is smaller. However, as k increases, the quality of the generator may drop, and thus the overall effect is uncertain. Similarly, the task of the low-level conditional policy becomes more difficult as k increases. The optimal value of k is 3 and 4 for INT and Rubik's Cube, respectively. In these environments, increasing k further degrades performance. In Sokoban, we observe monotonic improvement up to $k = 10$. This is perhaps because low-level conditional policy (Algorithm 9, based on breadth-first search) never fails to fill the path from a state to the valid subgoal. The running cost of Algorithm 9 quickly becomes unacceptable (recall that for $k = 4$, which we used in the main experiment, it has still a negligible cost - below $< 1\%$ of the total runtime).

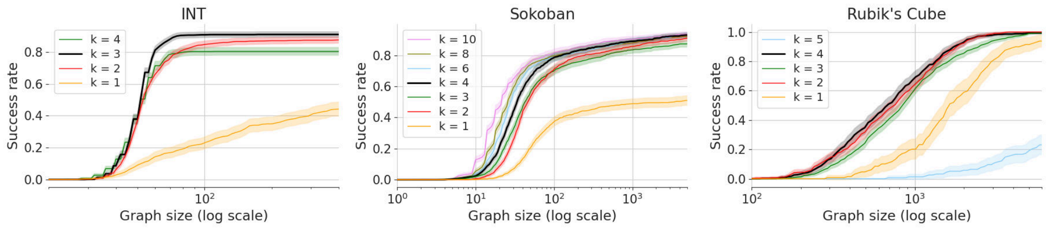


Figure 2: BF-kSubS success rates for different values of k. Black curves represent the values of k used in the main experiments (that is $k = 4$ for Rubik's Cube and Sokoban and $k = 3$ for INT).

4.5 Quality of subgoals

The learned subgoal generator is likely to be imperfect (especially in hard problems). We study this on 10×10 boards of Sokoban, which are small enough to calculate the true distance $dist$ to the solution using the Dijkstra algorithm. In Figure 3, we study $\Delta := dist_{s_1} - dist_{s_2}$, where s_1 is a sampled state and s_2 is a subgoal generated from s_1. Ideally, the histogram should concentrate on $k = 4$ used in training. We see that in slightly more than 65% of cases subgoals lead to an improvement.

The low-level conditional policy in Algorithm 2 provides additional verification of generated states. We check that in INT and Rubik's Cube, about 50% of generated subgoals can be reached by this policy (the rest is discarded).

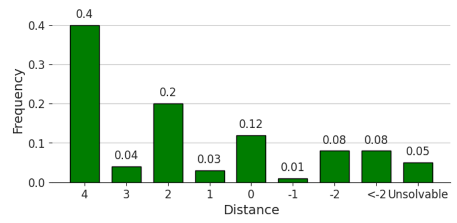


Figure 3: Histogram of Δ. Note that 17% of subgoals increases the distance. Additional, 5% leads to unsolvable "dead states" present in Sokoban.

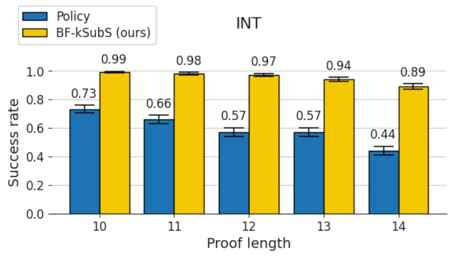


Figure 4: Out-of-distribution generalization to longer proofs. We compare with the behavioral cloning agent (Policy) studied in [54].

4.6 Value errors

There might be various explanations for the success of our method. One of them is that Subgoal Search better handles errors of learned value functions. In this section, we discuss this using a synthetic grid world example and performing statistical analysis on the real value function trained to approximate the distance to the solution (as described in Section 3).

Grid world example Consider a grid world with the state space $S = \{1, \ldots, n\}^m$, with $(0, \ldots, 0)$ being the initial state and $(n, \ldots, n)$ the goal state. A pair of states is connected by an edge if they are at distance 1 from each other. Let:

- Synthetic value function: the negative distance to the goal plus i.i.d Gaussian noise $\mathcal{N}(0, \sigma^2)$.
- Synthetic SUB_GENERATE (instead of Algorithm 3): Let $B_k(s)$ be the states within distance k from s. We return $C_3 - 1$ states sampled uniformly from $B_k(s)$ and a one "good subgoal" being a state in $B_k(s)$ with the minimal distance to the solution.
- Node expansion in BestFS (baseline): implemented as SUB_GENERATE above with $k = 1$.

In this setup, one easily sees that the probability that the good subgoal will have the highest value estimation among the generated states grows with k. Consequently, kSubS can handle higher levels of noise than the baseline BestFS, see Table 4.

σ	BestFS	BF-kSubS
3	0.999	1
10	0.142	1
20	0.006	0.983

Table 4: Success rates on the grid world ($m = 6, n = 10$), depending on the value function noise scale. We use the search budget of 500 nodes and $k = 4$ for kSubS.

Value monotonicity Imagine a solution path from the starting state to the goal state. Due to the errors, the value estimates on the path may not be monotonic. This is an undesirable property, which is likely to hinder search and make finding the path harder. Now consider the subpath consisting of consecutive states spaced k actions apart, as can be constructed by Subgoal Search. For this version, the value estimates are more likely to be monotonic and easier to find. To illustrate this, we measure monotonicity on solution paths found by our algorithm for INT. The probability that value decreases when moving k steps ahead drops from 0.32 when $k = 1$ to mere 0.02 for $k = 4$ (see Table 7 in Appendix).

Overoptimism Alternatively, one can consider that erroneously positive values misguide a search method (a phenomenon known as over-optimism [19]). To illustrate this, consider $\mathcal{S}_3(s)$, the set of all states having the same distance to the solution as s and within distance 3 from s. Intuitively, $\mathcal{S}_3(s)$ contains similar states with respect to the difficulty of solving. In Sokoban, the standard deviation of value function prediction for $\mathcal{S}_3(s)$ is equal to 2.43 (averaged over different s on Sokoban boards). This is high when compared to the average increase of value for moving one step closer to the solution, which is only 1.34. Consequently, it is likely that $\mathcal{S}_3(s)$ contains a suboptimal state, e.g., having a higher value than the best immediate neighbors of s (which by properties of the game will be closer to solution in Sokoban). Indeed, we measure that the probability of such an event is 64%. However, it drops significantly to 29% if one considers states closer by 4 steps (say given by a subgoal generator).

5 Limitations and future work

In this section, we list some limitations of our work and suggest further research directions.

Reliance on expert data In this version, we use expert data to train learnable models. As kSubS improves the performance, we speculate that training akin to AlphaZero can be used, i.e. in a planner-learner loop without any outside knowledge.

Optimality and completeness kSubS searches over a reduced state space, which might produce suboptimal solutions or even fail to find them. This is arguably unavoidable if we seek an efficient method for complex problems.

Subgoals definition We use simple k-step ahead subgoals, which is perhaps not always optimal. Our method can be coupled with other subgoal paradigms. Unsupervised detection of landmarks (see e.g. [55]) seems an attractive future research direction.

More environments In future work, we plan to test kSubS on more environments to understand its strengths and weaknesses better. In this work, we generate subgoals in the state space, which might be limiting for tasks with high dimensional input (e.g., visual).

Reliance on a model of the environment We use a perfect model of the environment, which is a common practice for some environments, e.g., INT. Extending kSubS to use learned (imperfect) models is an important future research direction.

Determinism Our method requires the environment to be deterministic.

OOD generalization A promising future direction is to investigate and leverage the out-of-distribution generalization delivered by our method and compare to (somewhat contradictory) findings of [17, 54].

Classical planning methods For many search problems, the state space can be represented in factored fashion (or such representation can be learned [3]). In such cases, the search can be greatly improved with width-based methods [25, 12]. It is an interesting research direction to combine kSubS with such methods.

6 Conclusions

We propose Subgoal Search, a search algorithm based on subgoal generator. We present two practical implementations MCTS-kSubS and BF-kSubS meant to be effective in complex domains requiring reasoning. We confirm that indeed our implementations excel in Sokoban, Rubik's Cube, and inequality benchmark INT. Interestingly, a simple k step ahead mechanism of generating subgoals backed up by transformer-based architectures performs surprisingly well. This evidence, let us hypothesize, that our methods (and related) can be further scaled up to even harder reasoning tasks.

Acknowledgments and Disclosure of Funding

The work of Konrad Czechowski, Tomasz Odrzygóźdź and Piotr Miłoś was supported by the Polish National Science Center grant UMO-2017/26/E/ST6/00622. We gratefully acknowledge Polish high-performance computing infrastructure PLGrid (HPC Centers: ACK Cyfronet AGH, PCSS) for providing computer facilities and support within computational grant no. PLG/2020/013619. Our experiments were managed using `https://neptune.ai`. We would like to thank the Neptune team for providing us access to the team version and technical support.

References

[1] Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. Solving the rubik's cube with deep reinforcement learning and search. Nature Machine Intelligence, 1(8):356–363, 2019.

[2] Marcin Andrychowicz, Dwight Crow, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In Advances in Neural Information Processing Systems 30: Annual Conference on Neural

Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA, pages 5048–5058, 2017.

[3] Masataro Asai and Alex Fukunaga. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. In Thirty-Second AAAI Conference on Artificial Intelligence, 2018.

[4] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. European Journal of Operational Research, 2020.

[5] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in games, 4(1):1–43, 2012.

[6] Elliot Chane-Sane, Cordelia Schmid, and Ivan Laptev. Goal-conditioned reinforcement learning with imagined subgoals. In International Conference on Machine Learning, pages 1430–1440. PMLR, 2021.

[7] Peter Dayan and Geoffrey E. Hinton. Feudal reinforcement learning. In Stephen Jose Hanson, Jack D. Cowan, and C. Lee Giles, editors, Advances in Neural Information Processing Systems 5, [NIPS Conference, Denver, Colorado, USA, November 30 - December 3, 1992], pages 271–278. Morgan Kaufmann, 1992.

[8] Dorit Dor and Uri Zwick. Sokoban and other motion planning problems. Computational Geometry, 13(4):215–228, 1999.

[9] Kuan Fang, Yuke Zhu, Animesh Garg, Silvio Savarese, and Li Fei-Fei. Dynamics learning with cascaded variational inference for multi-step manipulation. arXiv preprint arXiv:1910.13395, 2019.

[10] Alan Fern, Roni Khardon, and Prasad Tadepalli. The first learning track of the international planning competition. Machine Learning, 84(1-2):81–107, 2011.

[11] Carlos Florensa, David Held, Xinyang Geng, and Pieter Abbeel. Automatic goal generation for reinforcement learning agents. In International conference on machine learning, pages 1515–1528. PMLR, 2018.

[12] Guillem Frances, Miquel Ramírez Jávega, Nir Lipovetzky, and Hector Geffner. Purely declarative action descriptions are overrated: Classical planning with simulators. In IJCAI 2017. Twenty-Sixth International Joint Conference on Artificial Intelligence; 2017 Aug 19-25; Melbourne, Australia.[California]: IJCAI; 2017. p. 4294-301. International Joint Conferences on Artificial Intelligence Organization (IJCAI), 2017.

[13] Thomas Gabor, Jan Peter, Thomy Phan, Christian Meyer, and Claudia Linnhoff-Popien. Subgoal-based temporal abstraction in Monte-Carlo Tree Search. In IJCAI, pages 5562–5568, 2019.

[14] Wei Gao, David Hsu, Wee Sun Lee, Shengmei Shen, and Karthikk Subramanian. Intention-net: Integrating planning and deep learning for goal-directed autonomous navigation. In 1st Annual Conference on Robot Learning, CoRL 2017, Mountain View, California, USA, November 13-15, 2017, Proceedings, volume 78 of Proceedings of Machine Learning Research, pages 185–194. PMLR, 2017.

[15] Timothy Gowers. The importance of mathematics. Springer-Verlag, 2000.

[16] Arthur Guez, Mehdi Mirza, Karol Gregor, Rishabh Kabra, Sébastien Racanière, Théophane Weber, David Raposo, Adam Santoro, Laurent Orseau, Tom Eccles, et al. An investigation of model-free planning. In International Conference on Machine Learning, pages 2464–2473. PMLR, 2019.

[17] Jessica B. Hamrick, Abram L. Friesen, Feryal Behbahani, Arthur Guez, Fabio Viola, Sims Witherspoon, Thomas Anthony, Lars Holger Buesing, Petar Velickovic, and Theophane Weber. On the role of planning in model-based deep reinforcement learning. In 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021. OpenReview.net, 2021.

[18] Demis Hassabis, Dharshan Kumaran, Christopher Summerfield, and Matthew Botvinick. Neuroscience-inspired artificial intelligence. Neuron, 95(2):245–258, 2017.

[19] Hado Hasselt. Double q-learning. Advances in neural information processing systems, 23:2613–2621, 2010.

[20] Jeffrey R Hollerman, Leon Tremblay, and Wolfram Schultz. Involvement of basal ganglia and orbitofrontal cortex in goal-directed behavior. Progress in brain research, 126:193–215, 2000.

[21] Dinesh Jayaraman, Frederik Ebert, Alexei A. Efros, and Sergey Levine. Time-agnostic prediction: Predicting predictable video frames. In 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net, 2019.

[22] Leslie Pack Kaelbling. Learning to achieve goals. In Ruzena Bajcsy, editor, Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambéry, France, August 28 - September 3, 1993, pages 1094–1099. Morgan Kaufmann, 1993.

[23] Taesup Kim, Sungjin Ahn, and Yoshua Bengio. Variational temporal abstraction. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada, pages 11566–11575, 2019.

[24] Thanard Kurutach, Aviv Tamar, Ge Yang, Stuart J. Russell, and Pieter Abbeel. Learning plannable representations with causal infogan. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada, pages 8747–8758, 2018.

[25] Nir Lipovetzky and Hector Geffner. Width and serialization of classical planning problems. In ECAI 2012, pages 540–545. IOS Press, 2012.

[26] Kara Liu, Thanard Kurutach, Christine Tung, Pieter Abbeel, and Aviv Tamar. Hallucinative topological memory for zero-shot visual planning. In Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event, volume 119 of Proceedings of Machine Learning Research, pages 6259–6270. PMLR, 2020.

[27] Yinhan Liu, Jiatao Gu, Naman Goyal, Xian Li, Sergey Edunov, Marjan Ghazvininejad, Mike Lewis, and Luke Zettlemoyer. Multilingual denoising pre-training for neural machine translation. CoRR, abs/2001.08210, 2020.

[28] Piotr Miłoś, Łukasz Kuciński, Konrad Czechowski, Piotr Kozakowski, and Maciek Klimek. Uncertainty-sensitive learning and planning with ensembles. arXiv preprint arXiv:1912.09996, 2019.

[29] Suraj Nair and Chelsea Finn. Hierarchical foresight: Self-supervised learning of long-horizon tasks via visual subgoal generation. In 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020. OpenReview.net, 2020.

[30] Soroush Nasiriany, Vitchyr Pong, Steven Lin, and Sergey Levine. Planning with goal-conditioned policies. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada, pages 14814–14825, 2019.

[31] Giambattista Parascandolo, Lars Buesing, Josh Merel, Leonard Hasenclever, John Aslanides, Jessica B Hamrick, Nicolas Heess, Alexander Neitz, and Theophane Weber. Divide-and-conquer monte carlo tree search for goal-directed planning. arXiv preprint arXiv:2004.11410, 2020.

[32] Sujoy Paul, Jeroen Vanbaar, and Amit Roy-Chowdhury. Learning from trajectories via subgoal discovery. Advances in Neural Information Processing Systems, 32:8411–8421, 2019.

[33] Karl Pertsch, Oleh Rybkin, Frederik Ebert, Shenghao Zhou, Dinesh Jayaraman, Chelsea Finn, and Sergey Levine. Long-horizon visual planning with goal-conditioned hierarchical predictors. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual, 2020.

[34] Karl Pertsch, Oleh Rybkin, Jingyun Yang, Shenghao Zhou, Konstantinos G. Derpanis, Kostas Daniilidis, Joseph J. Lim, and Andrew Jaegle. Keyframing the future: Keyframe discovery for visual prediction and planning. In Alexandre M. Bayen, Ali Jadbabaie, George J. Pappas, Pablo A. Parrilo, Benjamin Recht, Claire J. Tomlin, and Melanie N. Zeilinger, editors, Proceedings of the 2nd Annual Conference on Learning for Dynamics and Control, L4DC 2020, Online Event, Berkeley, CA, USA, 11-12 June 2020, volume 120 of Proceedings of Machine Learning Research, pages 969–979. PMLR, 2020.

[35] Silviu Pitis, Harris Chan, Stephen Zhao, Bradly C. Stadie, and Jimmy Ba. Maximum entropy gain exploration for long horizon multi-goal reinforcement learning. In Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event, volume 119 of Proceedings of Machine Learning Research, pages 7750–7761. PMLR, 2020.

[36] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. arXiv preprint arXiv:2009.03393, 2020.

[37] Sébastien Racanière, Theophane Weber, David P. Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adrià Puigdomènech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, Razvan Pascanu, Peter Battaglia, Demis Hassabis, David Silver, and Daan Wierstra. Imagination-augmented agents for deep reinforcement learning. In NIPS, 2017.

[38] Tomas Rokicki and M Davidson. God's number is 26 in the quarter-turn metric, 2014.

[39] Stuart Russell and Peter Norvig. Artificial intelligence: A modern approach. ed. 3. 2010.

[40] Nikolay Savinov, Alexey Dosovitskiy, and Vladlen Koltun. Semi-parametric topological memory for navigation. In 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings. OpenReview.net, 2018.

[41] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. nature, 529(7587):484–489, 2016.

[42] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. Science, 1144:1140–1144, 2018.

[43] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. ArXiv, abs/1712.01815, 2017.

[44] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George Van Den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. Nature, 2017.

[45] Gregory J. Stein, Christopher Bradley, and Nicholas Roy. Learning over subgoals for efficient navigation of structured, unknown environments. In 2nd Annual Conference on Robot Learning, CoRL 2018, Zürich, Switzerland, 29-31 October 2018, Proceedings, volume 87 of Proceedings of Machine Learning Research, pages 213–222. PMLR, 2018.

[46] Richard S. Sutton, Joseph Modayil, Michael Delp, Thomas Degris, Patrick M. Pilarski, Adam White, and Doina Precup. Horde: a scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In Liz Sonenberg, Peter Stone, Kagan Tumer, and Pinar Yolum, editors, 10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011), Taipei, Taiwan, May 2-6, 2011, Volume 1-3, pages 761–768. IFAAMAS, 2011.

[47] Richard S. Sutton, Doina Precup, and Satinder P. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. Artif. Intell., 112(1-2):181–211, 1999.

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman

Garnett, editors, Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA, pages 5998–6008, 2017.

[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA, pages 5998–6008, 2017.

[50] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Manfred Otto Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. ArXiv, abs/1703.01161, 2017.

[51] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. Nature, 575(7782):350–354, Nov 2019.

[52] David Wilkins. Using patterns and plans in chess. Artif. Intell., 14(2):165–203, 1980.

[53] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. Huggingface's transformers: State-of-the-art natural language processing. CoRR, abs/1910.03771, 2019.

[54] Yuhuai Wu, Albert Jiang, Jimmy Ba, and Roger Grosse. Int: An inequality benchmark for evaluating generalization in theorem proving. arXiv preprint arXiv:2007.02924, 2020.

[55] Lunjun Zhang, Ge Yang, and Bradly C. Stadie. World model as a graph: Learning latent landmarks for planning. CoRR, abs/2011.12491, 2020.

A MCTS

A.1 MCTS-kSubS algorithm

In Algorithm 4 we present a general MCTS solver based on AlphaZero. Solver repeatedly queries the planner for a list of actions and executes them one by one. Baseline planner returns only a single action at a time, whereas MCTS-kSubS gives around k actions – to reach the desired subgoal (number of actions depends on a subgoal distance, which not always equals k in practice).

MCTS-kSubS operates on a high-level subgoal graph: nodes are subgoals proposed by the generator (see Algorithm 3) and edges – lists of actions informing how to move from one subgoal to another (computed by the low-level conditional policy in Algorithm 2). The graph structure is represented by $tree$ variable. For every subgoal, it keeps up to C_3 best nearby subgoals (according to generator scores) along with a mentioned list of actions and sum of rewards to obtain while moving from the parent to the child subgoal.

Most of MCTS implementation is shared between MCTS-kSubS and AlphaZero baseline, as we can treat the behavioral-cloning policy as a subgoal generator with $k = 1$. All the differences between MCTS-kSubS and the baseline are encapsulated in GEN_CHILDREN function (Algorithms 5 and 6). To generate children subgoals MCTS-kSubS runs subgoal generator and low-level conditional policy, whereas the baseline uses behavioral cloning policy for that purpose.

Algorithm 4 MCTS solver (common for AlphaZero baseline and MCTS-kSubS)

```
Require:  $L_a$          action limit
          $L_p$          planner calls limit
          $P$            planning passes
          $\gamma$       discount factor
          $c_{puct}$     exploration weight
          $\tau$         sampling temperature
          $V$            value function
          $env$          environment
          $M$            environment model
Use:      $tree$         tree structure
          $N(s,i)$       visit count
          $W(s,i)$       total child-value
          $Q(s,i)$       mean child-value
          $\pi_e$        exploration policy
# Initialize $N, W, Q$ to zero
function SOLVER
    $s \leftarrow env$.RESET()
    solution $\leftarrow$ []              ▷ List of actions
    for $1 \dots L_p$ do
        actions $\leftarrow$ PLANNER(s)
        for $a$ in actions do
            $s', r \leftarrow env$.STEP($a$)
            solution.APPEND($a$)
            $s \leftarrow s'$
        if solution.LENGTH() $> L_a$ then
            return None
        if $env$.SOLVED() then
            return solution
    return None
function PLANNER(state)
    for $1 \dots P$ do
        path, leaf $\leftarrow$ SELECT(state)
        EXPAND(leaf)
        UPDATE(path, leaf)
    return CHOOSE_ACTIONS(state)

function SELECT(state)
    $s \leftarrow$ state
    path $\leftarrow$ []
    while $s$ belongs to $tree$ do
        $i \leftarrow$ SELECT_CHILD($s$)
        $s', r,$ actions $\leftarrow tree[s][i]$
        path.APPEND(($s, i, r$))
        $s \leftarrow s'$
    return path, $s$
function EXPAND(leaf)
    children, probs $\leftarrow$ GEN_CHILDREN(leaf)
    $tree$[leaf] $\leftarrow$ children
    $\pi$(leaf, $\cdot$) $\leftarrow$ probs
    for $i \leftarrow 1$ to children.LENGTH() do
        $s', r,$ actions $\leftarrow tree$[leaf][$i$]
        $W$(leaf, $i$) $\leftarrow r + \gamma * V(s')$
        $N$(leaf, $i$) $\leftarrow 1$
        $Q$(leaf, $i$) $\leftarrow W$(leaf, $i$)
function UPDATE(path, leaf)
    quality $\leftarrow V$(leaf)
    for $s, i, r \leftarrow$ reversed(path) do
        quality $\leftarrow r + \gamma *$ quality
        $W(s,i) \leftarrow W(s,i) +$ quality
        $N(s,i) \leftarrow N(s,i) + 1$
        $Q(s,i) \leftarrow \frac{W(s,i)}{N(s,i)}$
function SELECT_CHILD($s$)
    $U(s,i) \leftarrow \sqrt{\sum_{i'} N(s,i')}/(1 + N(s,i))$
    $i \leftarrow argmax_i(Q(s,i) + c_{puct}\pi_e(s,i)U(s,i))$
    return $i$
function CHOOSE_ACTIONS($s$)
    $i \sim softmax\big(\frac{1}{\tau}\log N(s,\cdot)\big)$
    $s', r,$ actions $\leftarrow tree[s][i]$
    return actions
```

Algorithm 5 GEN_CHILDREN for MCTS-kSubS

For functions GET_PATH and SUB_GENERATE see Algorithms 2 and 3.

```
function GEN_CHILDREN(state)
    $s \leftarrow$ state
    children $\leftarrow$ [ ]
    probs $\leftarrow$ [ ]
    for subgoal, prob $\leftarrow$ SUB_GENERATE($s$) do
        actions $\leftarrow$ GET_PATH($s$, subgoal)
        if actions.EMPTY( ) then continue
        $r \leftarrow M$.REWARD_SUM($s$, actions)
        children.APPEND((subgoal, $r$, actions))
        probs.APPEND(prob)
    return children, probs
```

Algorithm 6 GEN_CHILDREN for AlphaZero

```
Require: $\pi_b$    behavioral cloning policy
function GEN_CHILDREN(state)
    $s \leftarrow$ state
    children $\leftarrow$ [ ]
    probs $\leftarrow$ [ ]
    for $a$, prob $\leftarrow \pi_b$.GEN_ACTIONS($s$) do
        $s', r \leftarrow M$.NEXT_STATE_REWARD($s, a$)
        children.APPEND(($s', r, [a]$))
        probs.APPEND(prob)
    return children, probs
```

Variables $tree, N, W, Q, \pi_e$ are reused across subsequent planner invocations within a single solver run. We limit the number of planner calls L_p for better control over the computational budget for MCTS-kSubS. For MCTS pseudocode we assume a slightly modified version of SUB_GENERATE function (defined originally in Algorithm 3). We presume that the function along with subgoals returns also their respective probabilities – as MCTS needs them to guide exploration.

A.2 Detailed results of MCTS-kSubS

We evaluate MCTS-based approaches on INT proofs of length 15. We set $c_{puct} = \tau = 1$ and $\gamma = 0.99$. We tuned P on MCTS (AlphaZero) baseline and we run three variants with $P \in \{5, 15, 50\}$. We run MCTS-kSubS ($k = 3$) with the same set of parameters and with kSubS-specific parameters fixed to $C_2 = C_3 = 4$ (in order to match the setup for corresponding INT BF-kSubS experiments).

We limit the maximum number of actions to $L_a = 24$ for both methods. Having the same number of planning passes P, during a single call MCTS-kSubS visits k-times more new states than the baseline (because of states visited by the low-level conditional policy). Therefore, to ensure a similar computational budget, we limit the number of planner calls to $L_p = 8$ for MCTS-kSubS and to $L_p = 24$ for the baseline – so the number of states visited over the course of a single solver run is similar for both methods.

Top-left part of Figure 1 illustrates results of MCTS experiments. For every number of planning passes P, MCTS-kSubS has significantly higher success rate than the corresponding baseline experiment. The highest difference is 0.52 for $P = 5$ (0.88 for MCTS-kSubS, 0.36 for the baseline) and slightly decreases with increasing number of passes to still impressive 0.43 for $P = 50$ (0.91 for MCTS-kSubS, 0.48 for the baseline). Comparing MCTS-kSubS for $P = 5$ with the baseline for $P = 50$, shows advantage of our method still by a significant margin of 0.40, despite having 10 times smaller computational budget.

MCTS-kSubS performed better also in terms of run time. For every tested P it was at least twice as fast as the corresponding baseline experiment.

High effectiveness of MCTS-kSubS, in terms of both search success rate as well as run time, shows that our kSubS method is not specific to BestFS planner, but potentially allows to boost a wide range of other planners.

B Architectures and hyperparameters

B.1 Transformer

For INT and Rubik we use mBART [27] – one of the state-of-the-art sequence-to-sequence transformer architectures. To speed up training and inference we use its lightweight version. We reduced the dimensionality of the model, so the number of learned parameters decreased from the original 680M to 45M. The set of our hyperparameters matches the values proposed in [48]: we used 6 layers of encoder and 6 layers of decoder; we adjusted model's dimension to 512 and number of attention heads to 8; the inner-layer of position-wise fully connected networks had dimensionality 2048. The difference in our model's size compared to 65M parameters reported in [48] results from vocabulary size. For our problems, it is enough to have 10-70 distinct tokens, whereas natural language models require a much larger vocabulary (tens of thousands of tokens).

For inference we used number of beams equal to 16 on INT and 32 on Rubik's Cube.

B.2 Sokoban

In Sokoban, we use three neural network architectures: for generating subgoals, for assigning value and one for baseline policy.

We took the value function network architecture from [28]. For the subgoal generator network we used the same convolutional architecture as in [28], with two exceptions. First, instead of predicting single regression target we predicted distribution over $d \times d \times 7 + 1$ classes. Secondly, we added batch norm layers between convolutional layers to speed up training. To make the comparison between BestFS and BF-kSubS fair, we also evaluated the training of expert from [28] with additional batch norm layers, but it turned out to actually hurt the performance of the expert. The architecture for baseline policy was the same as in [28] with only one modification: it predicts one of our actions instead of a single number.

C Data processing and datasets

C.1 Sokoban

Dataset. We collected expert datasets using an RL agent (MCTS-based) from [28]. Precisely, we trained 3 agents on Sokoban boards of different sizes (12×12, 16×16 and 20×20, all with four boxes). During the training process, we collected all successful trajectories, in the form of sequences of consecutive states. The number of trajectories in our datasets were: 154000 for 12×12 boards, 45000 for 16×16 boards and 21500 for 20×20 boards. The difference comes from the fact that the larger boards take more time to solve, hence fewer data is collected in the same time span.

Subgoal generation. For a given `state` the generation of subgoals is depicted in Algorithm 7. We maintain a queue of modified states (MS). Iteratively we take a MS from queue, concatenate it with `state` and pass through subgoal generator network (`subgoal_net`.SORTED_PREDICTIONS). This produces a probability distribution over candidates for further modifications of given MS. We take the most probable candidates, apply each of them to MS, and add the new modified states to the queue. If among the best subgoal generator network predictions there is a special "valid subgoal" token (encoded with $d \times d \times 7 + 1$), we put MS to subgoal candidates list (`subgoals_and_probs`). During this process, each MS is assigned the probability, which is a product of probabilities of modifications, leading to this MS. When the queue is empty, we take subgoal candidates and choose the ones with the highest probability such that the target probability (C_4) is reached (similar to Algorithm 3). The generation of subgoals for a given state is illustrated in Figure 5.

This process is designed to be computationally efficient. The majority of subgoals differ from the input by only several pixels, which leads to short paths of point-wise modifications. Note, that we do not utilize any Sokoban-specific assumptions.

Datapoints for training. Preparing data points for the training of the generator is described in Algorithm 8. For each trajectory in the dataset, we choose randomly 10% of state pairs for the training (we do not use all states from a trajectory in order to reduce the correlation in the data).

Low-level conditional policy. In Algorithm 9 we describe the BFS-based algorithm that verifies subgoals in Sokoban.

Performance of RL agent. We observed that each of the three RL agents we used (for 12×12, 16×16 and 20×20 boards), had a significantly lower success rate than our method's counterparts (that learns from these agents). For 12×12 boards it could solve around 78% of problems, for 16×16 boards it dropped to 67% and for 20×20 it was only 60%.

Algorithm 7 Sokoban subgoal generator

```
Require:  d              dimension of a board
          internal_cl    a number between 0 and 1
          subgoal_net    CNN returning distribution over modifications.
function GENERATE_SUBGOALS(state)
    subgoals_and_probs ← []
    q ← Queue()                                                    ▷ FIFO queue
    q.INSERT((state, 1))
    while not q not empty do
        modified_state, parent_prob ← q.POP( )
        network_input ← CONCATENATE(state, modified_state)
        predictions, probs ← subgoal_net.SORTED_PREDICTIONS(network_input)
        total_p ← 0
        for prediction, p ∈ (predictions, probs) do
            if total_p ≥ internal_cl then break
            total_p ← total_p + p
            if prediction = $d \times d \times 7 + 1$ then
                subgoals_and_probs.ADD((modified_state, parent_prob × p))
            else
                new_modified_state ← APPLY_CHANGE(modified_state, prediction)
                q.INSERT((new_modified_state, parent_prob × p))
        subgoals ← SORT_BY_PROBABILITY(subgoals)
    total_p ← 0
    for subgoal, p ∈ subgoals_and_probs do
        if total_p > $C_4$ then break
        subgoals.ADD(state)
        total_p ← total_p + p
    return subgoals
function APPLY_CHANGE(state, modification)
    ▷ modification is an integer in range $[1, d \times d \times 7]$ encoding which pixel of state
    ▷ to change (and to which value).
    row ← $\frac{\text{modification}}{d \times 7}$                              ▷ Integer division
    column ← $\frac{\text{modification} - \text{row} \times d \times 7}{7}$          ▷ Integer division
    depth ← modification − row × d × 7 − column × 7
    modified_state ← state
    SET_TO_ZEROES(modified_state[row, column])
    modified_state[row, column, depth] ← 1
    return modified_state
```

Algorithm 8 Sokoban generate network inputs and targets

```
Require:   d   dimension of a board
function GENERATE_INPUTS_AND_TARGETS(state, subgoal)
    inputs ← []                                                        ▷ empty list
    targets ← []                                                       ▷ empty list
    modified_state ← state
    input ← CONCATENATE(state, modified_state)
    inputs.APPEND(input)
    target_class_num ← 0
    for i ∈ 1 . . . d do
        for j ∈ 1 . . . d do
            for c ∈ 1 . . . 7 do
                target_class_num ← target_class_num + 1
                if subgoal[i, j, c] = 1 AND modified_state[i, j, c] = 0 then
                    targets.APPEND(target_class_num)
                    ▷ Numpy notation, replace pixel values on position (i, j) with values
                    ▷ from subgoal
                    modified_state[i, j, :] ← subgoal[i, j, :]
                    input ← CONCATENATE(state, modified_state)
                    inputs.APPEND(input)
    ▷ Last target: no more changes to the modified_state are needed (class enumerated
    ▷ with $d \times d \times 7 + 1$)
    targets.APPEND($d \times d \times 7 + 1$)
    return inputs, targets
```

Algorithm 9 BFS low-level conditional policy

```
Require:   $k$           limit of steps
           $M$           model of the Sokoban environment
Use:       bfs_queue     BFS queue;
                         stores pairs of a state and the action path to it (from the root).
# Initialize bfs_queue to empty
function GET_PATH($s_0$, subgoal)
    step ← 0
    bfs_queue.ADD(($s_0$, []))
    while bfs_queue not empty do
        s, action_path ← bfs_queue.POP( )
        for action ∈ action_space do
            s ← $M$.NEXT_STATE(s, action)
            action_path.APPEND(action)
            if s = subgoal then
                return action_path
            if LEN(action_path) < $k$ then
                bfs_queue.ADD((s, action_path))
    return []
```

C.2 INT

State representation. A state in INT consists of objectives to prove and ground truth assumptions, which are logic statements that are assumed to hold and can be used in proving. Each objective, as well as each ground truth assumption, is a mathematical statement. In our setup, as in the original paper [54], there is always only one objective to prove, but there may be a varying number of ground truth statements.

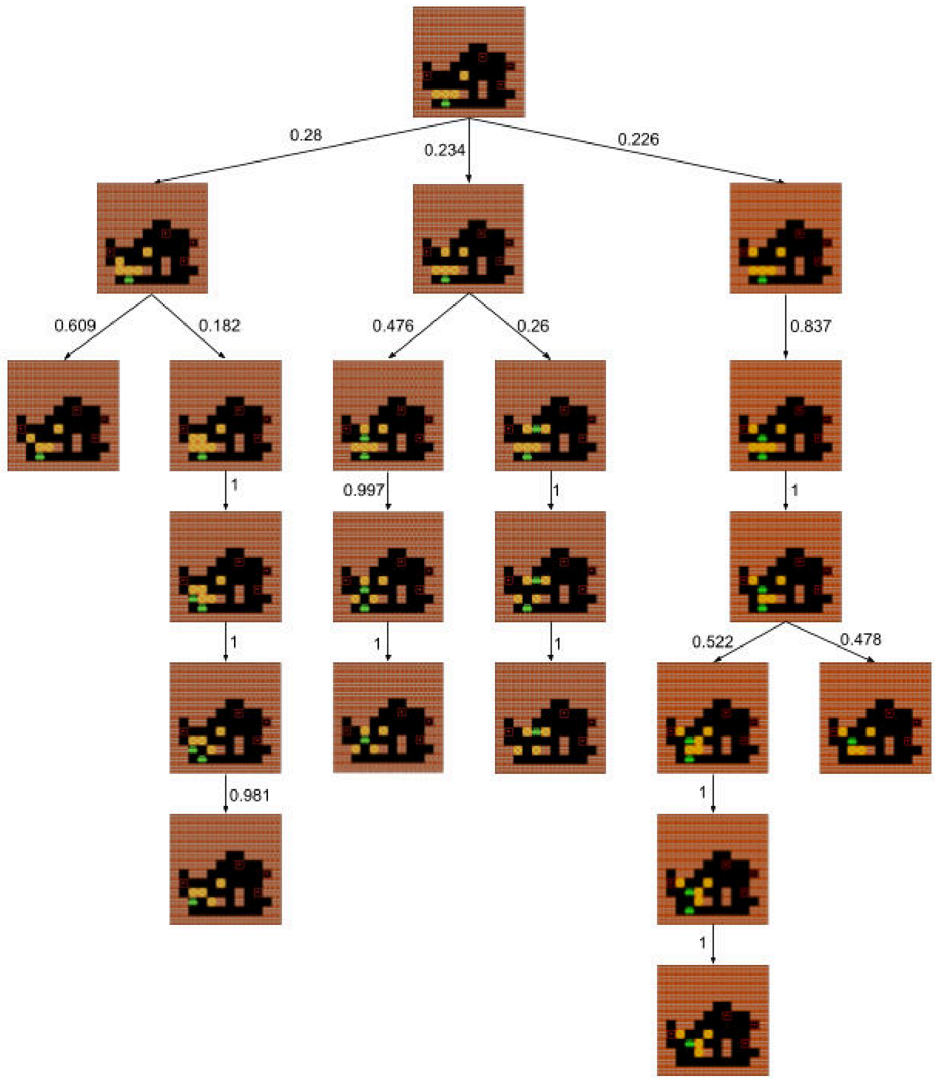


Figure 5: A detailed view of subgoal generation for Sokoban. Arrow represent probabilities of a given modification. Final subgoals are located in the leaves.

Each mathematical statement can be converted to a string by a method `logic_statement_to_seq_string` in INT library. In our code, we represent the full state as a string in the following form (all symbols are tokens, not logical operations):

$$\#[\text{objective}]\&[\text{1st ground truth}]\&[\text{2nd ground truth}]\&\ldots\ \&[k-\text{th ground truth}]\$$$

For example, the state representation could look like this:

$$\#(((b+b)*((b+b)*(b+b)))*((((b+b)+f)*(b+b))*(b+b))) \geq 0\&(b+f) = b\&(b+f) \geq 0\$$$

Action representation. An action in INT consists of a chosen axiom (one from the set of ordered field axioms, see [54, Appendix C]) and a sequence of entities onto which the axiom will be applied. An entity is an algebraic expression that is a part of the larger statement. For example, $(a+b)$ is an entity, which is a part of $(a+b)\cdot c = (1+f)$. In our code, we represent entities by indicating the symbol of their mathematical operation, or if the entity is atomic (a single variable), by indicating the variable itself. More precisely, directly after the symbol of operation, we add a special character '$\sim$'. For example, we indicate $(a+b)$ inside $(a+b)\cdot c$ in the following way: $(a+\sim b)\cdot c = (1+f)$. Typically, in a logic statement there may be several entities that have the same text form, but are located in different positions, for example $(a+b)$ appears twice in $(a+b)\cdot(a+b)=(1+0)$. Our way of encoding actions unambiguously identifies every entity. If the action has more than one input, we use more different indicators.

Low-level conditional policy input representation. Low-level conditional policy takes as an input two states: s and s', where s is the initial state and s' is the subgoal. The input is constructed in the following way: first, we represent both s and s' as strings and then we find the difference (character delta) between these strings using the function ndiff from difflib[5] Python library. We observed that using the character delta, instead of the concatenation of s and s', significantly improved the performance.

C.3 Rubik's Cube

State representation The state of the Rubik's Cube is determined by the arrangement of 54 colored labels on its faces. Therefore, to represent the observations we simply put the labels in a fixed order. An example state is as follows:

$$?byywygrygobbrboorgwbowryoogywbggywrrroyrogyowwbrwwbbgg\$,$$

where the tokens *b, g, o, r, w, y* stand for *blue, green, orange, red, white*, and *yellow*. The consecutive blocks of 9 tokens correspond to consecutive faces of the cube. Observe, that not every permutation of colors is valid. For example, the tokens on positions 5, 14, 23, 32, 41, and 50 correspond to centers of faces, thus they are fixed. There are more such constraints, but they are irrelevant to the pipeline itself.

Action representation In our experiments we use quarter turns, i.e. an action corresponds to rotating a face by $90°$, either clockwise or counterclockwise. Since the action space contains only 12 elements, we use unique tokens to represent each of them.

Low-level conditional policy input representation. The conditional policy takes two states s and s', which correspond to the current state and the state to be reached. To represent such pairs, on every position we put a token corresponding to a pair of colors – one located on that position in s and the other in s'. Since there are only 6 distinct colors on the Rubik's Cube, this requires using only 36 tokens.

D Training details

D.1 INT and Rubik's Cube

D.1.1 Transformer training

For transformer training and inference we used HuggingFace's Transformers library [53]. We did not use any pretrained checkpoints from HuggingFace model hub. We took mBART model class instead – and trained it from scratch in a supervised way using HuggingFace's training pipeline. We generated (or loaded from disk) a fresh dataset for every epoch. Training batch was of size 32. For regularization, we set $dropout = 0.1$, but we did not use label smoothing (as opposed to [48]).

For the Adam optimizer we set $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. Learning rate schedule followed the formula:

$$lr = peak\_lr * \min\left(\frac{step\_num}{warmup\_steps}, \sqrt{\frac{warmup\_steps}{step\_num}}\right),$$

where $peak\_lr = 3\cdot10^{-4}$ and $warmup\_steps = 4000$.

The schedule curve matches the one proposed in [48], but they use $peak\_lr \approx 7\cdot10^{-4}$.

D.1.2 Sequence generation

We used beam search with the number of beams set to 16 for INT and to 32 for Rubik's Cube. The number of returned sequences varied from 1 to 4 depending on the network type.

[5]https://docs.python.org/3/library/difflib.html

We set softmax temperature to 1 by default. For Rubik's Cube subgoal generator we tuned this parameter and the value of 0.5 performed best. We conducted a corresponding experiment for the baseline policy, but the temperature did not impact results in this case, as the policy outputs only a single token. For INT we did not tune the temperature, so we kept the value of 1.

D.1.3 Random seeds

Due to use of the supervised learning, we observed little variance with respect to the random initialization. We tested this for the subgoal generator on proofs of length 10 and for $k = 3$. Namely, we trained 5 models of the subgoal generator, starting from different initializations. The success rate barely varied, as they stayed in the interval $[0.990, 0.992]$. In the other experiments, we used a single seed.

D.2 Sokoban

For training of the convolutional networks in Sokoban we set the learning rate to 10^{-4} and the number of epochs to 200.

E Wall-time for kSubS

As indicated in Table 2, kSubS builds smaller search graphs. This has the practical advantage of making fewer neural network calls and consequently a substantially better wall-time.

The gains might be as high as 7 times due to costly sequential calls of transformer networks, see Table 5.

Proof length	5		10		15	
Method	BestFS	BF-kSubS (ours)	BestFS	BF-kSubS (ours)	BestFS	BF-kSubS (ours)
Total wall-time	4h 12m	**3h 44m**	29h 22m	**5h 55m**	69h 15m	**9h 22m**
Avg. generator calls	NA	**3.04**	NA	**3.89**	NA	**6.23**
Avg. value calls	23.34	**4.01**	112.35	**4.80**	159.46	**6.59**
Avg. policy calls	22.41	**8.43**	112.21	**13.09**	161.02	**20.29**

Table 5: Resources consumption for INT. We present evaluation on 1000 proofs and split into calls of subgoal generator network (used only in the subgoal search), value network and policy network (we report an average number of calls for a single proof).

F Training dataset size analysis

We tested how the success rate of BF-kSubS on 12x12 Sokoban boards depends on the size of the training set. The full dataset consists of 125k trajectories. We trained subgoal generator and value network on subsets consisting of 0.5, 0.25, 0.05 and 0.01 of all trajectories. The results are presented in Table 6.

Fraction of the dataset	1	0.5	0.25	0.05	0.01
Success rate	0.93	0.86	0.84	0.48	0.14

Table 6: Sokoban success rates for different training set sizes.

G Value errors

G.1 INT analysis

Due to the size of state spaces, it is impossible to search over the entire space of INT formulas to find the shortest proofs. We instead analyze value estimations along proofs generated by INT engine. The monotonicity analysis in Section 4.6 was performed using 100 such proofs of length 10. The probabilities of value decrease for different step lengths l are presented in Table 7.

G.2 Sokoban Analysis

Here, we present details related to the Sokoban analysis from Section 4.6. We sampled 500 Sokoban boards (with dimension $(12, 12)$). For each board, we calculated a full state-action graph and minimal distance from each state to the solution (this was needed to compute $S(s)$ sets later on). Since Sokoban graphs can be very large we set the limit on the graph size to 200000, which left us with 119 boards. Next, for each board, we took the sequence of states from the shortest solving path from the initial state (let us call this dataset as *shortest-paths* - SP). For each pair of consecutive states in SP, we calculated the difference of the value estimation, and averaged them, which gave us a mean one-step improvement of 1.34. We calculated this metric for 5 value function networks trained with different initialization, obtaining mean one-step improvement between $[1.23, 1.41]$.

l	Value decrease prob.
1	0.316
2	0.217
3	0.080
4	0.020

Table 7

To calculate the standard deviation of value function estimates for $S(s)$ we took SP, and limit it to states s such that $|S(s)| \geq 5$ (lets denote it as SP5). We calculated standard deviation for each

$s \in SP5$ separately. This gave us a mean deviation of 2.43. (between $[2.24, 2.86]$ for 5 value networks trained with different initialization) The same set SP5 was used to calculate probabilities related to overoptimistic errors on Sokoban described at the end of Section 4.6.

To calculate the above statistics we used the value function trained with supervised learning to approximate the distance to the solution. We observe that similar problems arise also when using value function trained with reinforcement learning [28]. In such setup, mean variance of value function estimates for $S(s)$ is 0.84, when one step improvement equals to 0.33 . Probability that there is a state in $S(s)$ with value higher than best immediate neighbour of s is 86% and it drops to 38%, if one considers states closer by 4 steps.

H Example subgoals

H.1 Example subgoals sets

In this section, we present some example outcomes of the subgoal generator for INT and Sokoban.

H.1.1 INT

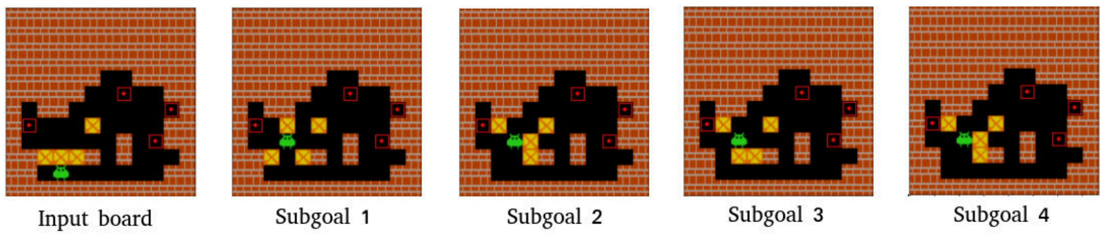
In (1) and (2) we provide two examples of applying the subgoal generator (trained on proofs of length 5) to the given states in INT. The number of subgoals varies since not all of the outputs generated by the network could be reached by the conditional low-level policy.

$$\begin{aligned}
\text{Input state: } & (((b\cdot b)+((b+(b+f))\cdot b))+(f+f))\geq((((b+(b+b))\cdot b)+0)+c) \\
\text{Ground truth 1: } & (b+f)=b \\
\text{Ground truth 2: } & (f+f)\geq c \\
\text{Subgoal 1: } & ((b\cdot b)+((b+(b+f))\cdot b))=(((b+(b+b))\cdot b)+0) \\
\text{Subgoal 2: } & (((b+(b+f))\cdot b)+(b\cdot b))=(((b+(b+b))\cdot b)+0) \\
\text{Subgoal 3: } & ((b\cdot b)+((b+(f+b))\cdot b))=(((b+(b+b))\cdot b)+0)
\end{aligned} \tag{1}$$

$$\begin{aligned}
\text{Input state: } & ((((b\cdot(\tfrac{1}{b}))+a)^2)+(c+(\tfrac{1}{b})))=((((((\tfrac{1}{b})\cdot b)+a)\cdot(a+1))+c)+(\tfrac{1}{b})) \\
\text{Subgoal 1: } & ((((b\cdot(\tfrac{1}{b}))+a)^2)+(c+(\tfrac{1}{b})))=((((b\cdot(\tfrac{1}{b}))+a)\cdot(a+1))+(c+(\tfrac{1}{b}))) \\
\text{Subgoal 2: } & ((((b\cdot(\tfrac{1}{b}))+a)\cdot((b\cdot(\tfrac{1}{b}))+a))+(c+(\tfrac{1}{b})))=((((b\cdot(\tfrac{1}{b}))+a)\cdot(a+1))+(c+(\tfrac{1}{b})))
\end{aligned} \tag{2}$$

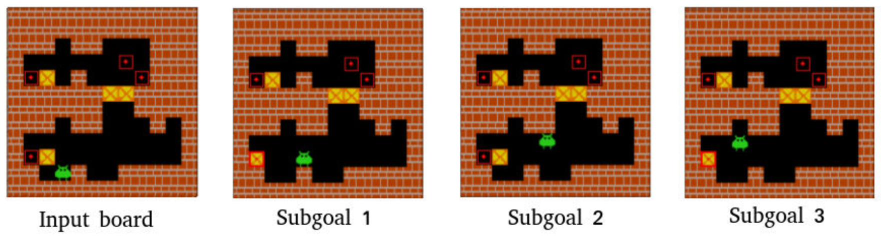
H.1.2 Sokoban

Here we present two examples of the outcomes of the subgoal generator trained for 12×12 boards:



H.2 Example solutions with subgoals

In this section, we present several examples of solutions obtained with our method. For simplicity, we only show the subgoal states on which the successful trajectories were constructed. In our setup, the last subgoal is always a solution.



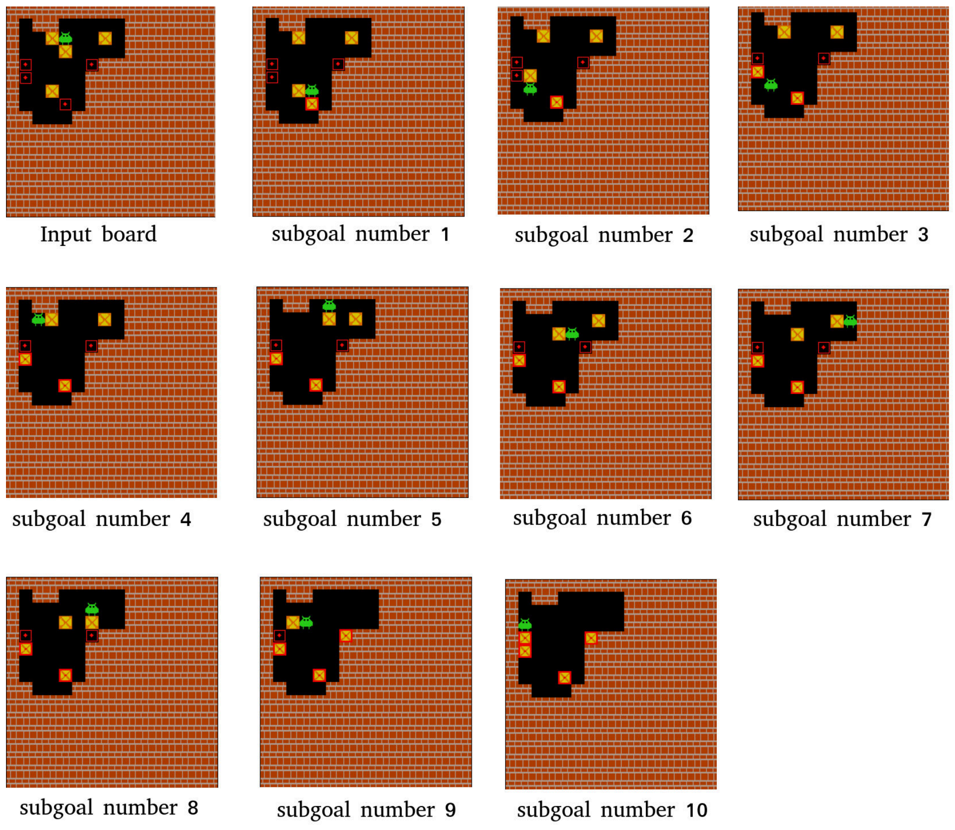
H.2.1 INT

An example solution of INT problem of length 5:

$$\begin{aligned}
\text{Problem: } & (((0\cdot((a+0)+(-(a\cdot 1))))\cdot(\tfrac{1}{(0^2)}))+((a+0)+(0^2)))\geq((((0^2)+(1+(a+0)))+b)+(-((a+0)+f))) \\
\text{1st subgoal: } & (((0\cdot((a+0)+(-(a\cdot 1))))\cdot(\tfrac{1}{(0^2)}))+((a+0)+(0^2)))=((0^2)+(1+(a+0))) \\
\text{2nd subgoal: } & (((0\cdot((0+a)+(-(a\cdot 1))))\cdot(\tfrac{1}{(0^2)}))+((a+0)+(0^2)))=(1+((a+0)+(0^2))) \\
\text{3rd subgoal: } & (0+a)=(a\cdot 1) \\
\text{4th subgoal: } & a=a
\end{aligned} \tag{3}$$

H.2.2 Sokoban

An example solution of Sokoban board:



I Baselines

Our first baseline is the low-level policy trained with behavioral cloning from the expert data trained to solve the problem (contrary to the low-level conditional policy P, which aims to achieve subgoals). Such policy was used [54]. We verified that our behavioral cloning policy reproduces the results from [54] for proofs of lengths 5.

MCTS. As a baseline for MCT-kSubS we used an AlphaZero-based MCTS planner described in Appendix A.1.

BestFS. The baseline for BF-kSubS is a low-level planning. We substitute SUB_GENERATE with a function returning adjacent states indicated by the most probable actions of behavioral cloning policy, see Algorithm 10.

Algorithm 10 Low-level generator

```
Require:  $C_3$   number of states to produce
          $NB$    number of beams in sampling
          $\pi_b$ behavioral cloning policy
          $M$     model of the environment
function SUB_GENERATE(s)
    actions ← BEAM_SEARCH($\pi_b$, s; $C_3$, $NB$)
    subgoals ← []
    for action ∈ actions do
        $s \leftarrow M$.NEXT_STATE(s, action)
        subgoals.APPEND(s)
    return subgoals
```

J Simple planners

An appropriate search mechanism is an important design element of our method. To show this, we evaluate an alternative, simpler procedure used by e.g. [9] for subgoal-based planning. It works by sampling independent sequences of subgoals and selects the best one. This method solved none of 1000 Rubik's Cube instances despite using the same subgoal-generator as BF-kSubS (which has a success rate of 0.999 with a comparable computational budget).

K Investigation of baseline-BestFS on Rubik's Cube

To obtain the training datasets on the Rubik's Cube environment, we generated random paths starting from a solved cube and stored them in the reversed order. These backward solutions are highly sub-optimal: for example, the states obtained by 10 random moves are usually in a distance of about 6 - 7 steps from the final state and the gap gets much larger for a higher number of moves. This means that on collected trajectories only some of the actions indeed lead to the solution and the majority of them only introduce noise.

We observed that the baseline is much weaker than BF-kSubS even for $k = 1$, despite the effort on tuning it. We extended the training of behavioral cloning policy and did a grid-search over parameters to further improve its success rate. We managed to reach no more than 10% success rate for the best version. To provide a meaningful evaluation of the baseline, we also trained it on very short trajectories consisting of 10 random moves. Such a curriculum allowed the behavioral cloning policy to learn, however still suffered from randomness in data (for states located far from the solution).

Interestingly, BF-kSubS for $k = 1$ turned out to perform much better than the baseline. Full understanding of this phenomenon requires additional research, however we hypothesize that in our setup learning to predict states is an easier task that predicting an action. A potential reasons are that: states prediction provides a denser learning signal and Transformers perform better when dealing with sequences (then with predicting a single token).

Note that both kSubS and baseline policy learn from fully off-policy data. The problem of solving the Rubik's Cube is challenging, thus learning from noisy off-policy trajectories can be simply too hard for standard algorithms.

L Technical details

L.1 Infrastructure used

We had 2 types of computational nodes at our disposal, depending on whether a job required GPU or not. GPU tasks used a single Nvidia V100 32GB card (mostly for transformer training) and Nvidia

RTX 2080Ti 11GB (for evaluation) with 4 CPU cores and 16GB RAM. The typical configuration of CPU job was the Intel Xeon E5-2697 2.60GHz processor (28 cores) with 128GB memory.

Each transformer for INT was trained on a single GPU for 3 days (irrespective of proof length and the network type). INT evaluation experiments used a single GPU for a time period varying from several hours to 3 days – with baselines being the bottleneck.

Transformers for Rubik's Cube required more training – every network was trained for 6 days on a single GPU. Because of relatively short sequences representing the cube's state, we were able to run evaluations without GPU. We used 20 CPU cores with 20GB memory, while still fitting in a 3-day run time.

We trained and evaluated Sokoban on CPU only mostly because of rather small neural network sizes. Training time varied from 2 to 3 days, whereas evaluation took only an hour.

FAST AND PRECISE: ADJUSTING PLANNING HORIZON WITH ADAPTIVE SUBGOAL SEARCH

Michał Zawalski *
University of Warsaw
m.zawalski@uw.edu.pl

Michał Tyrolski *
University of Warsaw
michal.tyrolski@gmail.com

Konrad Czechowski *
University of Warsaw
k.czechowski@mimuw.edu.pl

Tomasz Odrzygóźdź
IDEAS NCBR
tomaszo@impan.pl

Damian Stachura
Jagiellonian University
damian.stachura1@gmail.com

Piotr Piękos
KAUST†
piotrpiekos@gmail.com

Yuhuai Wu
Google Research
& Stanford University
yuhuai@google.com

Łukasz Kuciński
Polish Academy of Sciences
lkucinski@impan.pl

Piotr Miłoś
Ideas NCBR,
Polish Academy of Sciences,
deepsense.ai
pmilos@impan.pl

ABSTRACT

Complex reasoning problems contain states that vary in the computational cost required to determine the right action plan. To take advantage of this property, we propose Adaptive Subgoal Search (AdaSubS), a search method that adaptively adjusts the planning horizon. To this end, AdaSubS generates diverse sets of subgoals at different distances. A verification mechanism is employed to filter out unreachable subgoals swiftly, making it possible to focus on feasible further subgoals. In this way, AdaSubS benefits from the efficiency of planning with longer-term subgoals and the fine control with shorter-term ones, and thus scales well to difficult planning problems. We show that AdaSubS significantly surpasses hierarchical planning algorithms on three complex reasoning tasks: Sokoban, the Rubik's Cube, and the inequality-proving benchmark INT.

1 INTRODUCTION

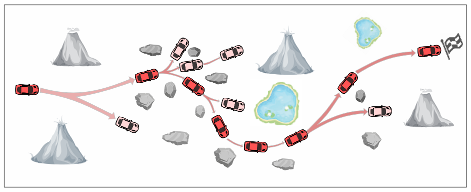
When solving hard problems, people often try to decompose them into smaller parts that are typically easier to complete (Hollerman et al., 2000). Similarly, *subgoal search methods* aim to solve complex tasks by considering intermediate subgoals leading towards the main goal. Besides their intuitive appeal, such approaches offer many practical advantages. Most notably, they enable deeper search within a smaller computational budget and reduce the negative impact of approximation errors. *Subgoal search* methods powered by deep learning have shown promising results for continuous control tasks, such as robotic arm manipulation (Nair & Finn, 2020; Jayaraman et al., 2019; Fang et al., 2019) and navigation (Kim et al., 2019; Savinov et al., 2018). Recently, Czechowski et al. (2021) showed that the usage of a subgoal generator can significantly improve search efficiency on discrete domains with high combinatorial complexity.

This paper uses Czechowski et al. (2021) as a starting point and pushes forward, building upon the following observation: many complex reasoning problems contain states that vary in complexity, measured by the computational cost required to determine the right action plan. To illustrate this, imagine driving a car. When traversing a narrow, winding street, it is crucial to focus on the closest events: the next turn, the next car to avoid, etc. However, after entering a straight, empty street, it

*equal contribution; Published as a conference paper at ICLR 2023, notable-top-5%.
†Work done while at the University of Warsaw

is enough to think about reaching its far end. This suggests that careful balancing of the subgoal distance is desirable: this involves selecting longer-term subgoals, if possible, to advance faster towards the goal, and choosing shorter-term subgoals to power through the harder states. Hence, the question arises whether it is possible and, if so, how to incorporate this adaptive subgoal generation procedure into subgoal search methods. In this paper, we answer this question affirmatively.

We propose a novel planning algorithm *Adaptive Subgoal Search* (AdaSubS), which adaptively chooses from subgoals with different horizons. Our method benefits both from the efficiency of planning with longer-term subgoals and from the reliability of shorter-term ones. AdaSubS *prioritizes further distances*, retracting to shorter ranges only when stuck. Additionally, we introduce a *verifier network*, which assesses whether the proposed subgoal is valid and reachable. The verifier makes it possible to efficiently discard faulty subgoals, which are common and more costly to detect in longer horizons. AdaSubS is a data-driven algorithm whose key components are implemented as learnable deep models. In most cases, we use general-purpose transformer architectures to model subgoal generators and the verifier networks. We train those models on offline data.


An illustrative example of adaptive planning. The planner may choose long-distance subgoals in the easier areas (e.g. the left most part) and use short distances in the hard areas (e.g. middle part).

We show the effectiveness of AdaSubS in three challenging domains: Sokoban, Rubik's Cube, and the inequality theorem prover INT (Wu et al., 2021). AdaSubS significantly surpasses hierarchical planning algorithms and sets a new state-of-the-art on INT.

Our main contributions are:

1. We propose Adaptive Subgoal Search (AdaSubS), a new algorithm that adjusts the planning horizon to take into account the varying complexity of the state space.
2. We present a comprehensive study of adaptive methods, showing that they outperform similar algorithms without adaptation. Amongst these, AdaSubS is the best choice across environments and planning budgets.
3. We also observe a strong indication of out-of-distribution generalization. AdaSubS trained on the proof of length 15 in INT (longest considered in the literature so far) retains more than 50% of its performance when the proof length is increased two-fold.

The code of our method is available at https://github.com/AdaptiveSubgoalSearch/adaptive_subs.

2 Related work

The combination of planning algorithms with deep learning is an active area of research. It provided impressive results e.g., in automated theorem proving (Polu & Sutskever, 2020), chess and Go (Silver et al., 2017), Atari benchmark (Schrittwieser et al., 2019), and video compression (Mandhane et al., 2022).

In the field of hierarchical planning, the majority of deep-learning-based methods have focused on visual domains (Kim et al., 2019; Pertsch et al., 2020a; Jayaraman et al., 2019; Fang et al., 2019) or on landmark-based navigation methods (Liu et al., 2020a; Gao et al., 2017; Zhang et al., 2020) . This body of work often relies on variational autoencoders for the compression of visual observations and uses planning mechanisms suitable for continuous control settings.

There exist many approaches to hierarchical planning utilizing different temporal distances. Kim et al. (2019) and Pertsch et al. (2020b) use hierarchical variational models to learn the temporal structure of tasks by reconstructing the visual state sequences. Pertsch et al. (2020a); Parascandolo et al. (2020); Jurgenson et al. (2020) recursively construct a plan by generating subgoals in the middle between the existing ones. Allen et al. (2021) generate macro-actions that help to speed-up the search. This differs from our work, as we use learning to generate subgoals (as opposed to action sequences) and the process is agnostic with respect to the size of the action space. These works have been shown to work on domains with limited combinatorial complexity.

Recently, Czechowski et al. (2021) has shown how combinatorially complex domains can be treated with a hierarchical planning method. Their approach shares similarities with our Adaptive Subgoal Search; however, it cannot address variable environment complexity. By using the mechanism of adaptive selection of the subgoal generation distance and verifier, we successfully tackle this problem, confirmed by significant performance gains. Our verifier is based on ideas similar to Cobbe et al. (2021); Kurutach et al. (2018); Ahn et al. (2022).

Our approach relates to established search algorithms (Cormen et al., 2009; Russell & Norvig, 2010), such as Best First Search or A*. Adaptivity techniques in the classical setup are discussed in Fickert (2022). Koenig & Likhachev (2006) propose an adaptive mechanism to improve A* by updating the goal-distance heuristic with local search. AdaSubS instead uses a fixed heuristic and adapts to the local complexity by alternating between subgoal distances. Domain-independent PDDL-based planners (McDermott et al., 1998) do not use training and attempt to solve problems in a zero-shot manner. Thus, they indicate a lower bound on performance. On the other hand, there are domain-specific methods (Korf, 1997; Büchner et al., 2022; Muppasani et al., 2022). Due to their focus, they indicate an upper bound.

AdaSubS relates to multi-queue methods (see Richter & Westphal (2010); Helmert (2006)), which alternate between multiple heuristics. Some of our planners, e.g., IterativeMixing or Longest-first, can be viewed through the lens of this approach in the sense that we could keep the priority queues for each generator separate (but with the same heuristic being a value function) and have an alternation mechanism between them. The key difference lies in the expansion phase: we expand subgoals instead of children and only the generator associated with the currently selected queue is used. [1]

Different instances of Sokoban, Rubik's Cube, and INT can be viewed as tasks with varying degrees of difficulty. Consequently, AdaSubS benefits from sharing data between these instances in a manner typical to multitask learning (Caruana, 1998). In particular, we use a goal-conditioned policy which is trained similarly as in Andrychowicz et al. (2020) or Kaelbling (1993). Additionally, the out-of-distribution generalization of AdaSubS hints at strong meta-learning capabilities of the method (Yu et al., 2020; Duan et al., 2016; Wang et al., 2016; Hessel et al., 2019).

3 Method

For this work, we propose Adaptive Subgoal Search (AdaSubS), a subgoal-based search algorithm designed to solve tasks that can be formulated as a search over a graph with a known transition model. AdaSubS is the best choice stemming from a careful study of methods based on the principle of mixing subgoal distances; see Section 4.4 for their definitions and empirical comparisons.

AdaSubS (see Algorithm 1) utilizes the following key components: *subgoal generators*, *verifier*, *conditional low-level policy* (CLLP), and *value function*. These components are implemented using trained neural networks (see Appendix B). To solve a task, AdaSubS iteratively builds a tree of subgoals reachable from the initial state until the target state is reached or the search budget is depleted. In each iteration, it chooses a node in the tree that is expanded by one of the generators. The chosen generator creates a few subgoal candidates, i.e., states expected to be a few steps closer to the target than the current node. For each of them, we use the verifier and CLLP to check whether they are valid and reachable within a few steps. For the correct subgoals, we compute their value function, place them in the search tree, and the next iteration follows (see Figure 1).

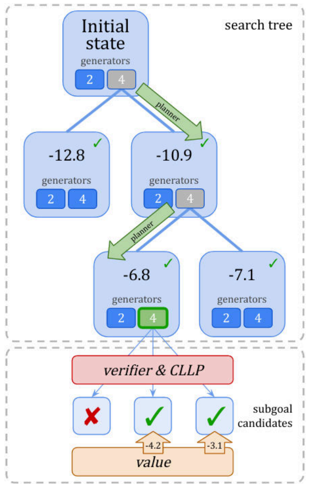


Figure 1: An example iteration of the search performed by AdaSubS.

AdaSubS follows the general structure of Best-First Search (BestFS); thus, the key design decision is how we prioritize nodes to expand and choose generators to produce subgoals. We defer the answer to these questions after providing details of the algorithm components (see also Appendix G and the flowchart there).

[1]For search engines using multi-queues, see Fast downward `https://www.fast-downward.org/`; LAPKT `https://github.com/LAPKT-dev/LAPKT-public`. For PDDL generators, see `https://github.com/AI-Planning/pddl-generators`

Subgoal generators. The subgoal generator, or more precisely the k-subgoal generator, takes a state as input and returns a diverse set of new candidate states expected to be k step closer to the solution. The key trade-off, which AdaSubS needs to address, is that further subgoals, i.e., those for higher values of k, advance faster towards the target but are also increasingly harder to generate and verify. We typically use a few (e.g., 3) generators with a list of k chosen basing on experiments, namely for INT [3, 2, 1], Rubik [4, 3, 2] and Sokoban [8, 4, 2]. Note that this is the only component of AdaSubS that outputs a set of predictions.

Conditional low-level policy (CLLP). CLLP returns a path of low-level actions between two states (see Algorithm 2). CLLP calls iteratively a conditional low-level policy network (PN). PN takes as input the current and target states and returns an action. It is possible that CLLP is not able to reach the target, in which case an empty sequence is returned. The role of CLLP is two-fold: it serves as a mechanism allowing AdaSubS to transition between subgoals, and together with the verifier network, it is used in the subgoal verification algorithm (see Algorithm 3).

Verifier. The verifier network is used in the verification algorithm (see Algorithm 3), to answer the following binary classification question: given a starting state and a goal state, is it possible to reach the latter from the former using conditional low-level policy? Computationally, the evaluation of the verifier network is faster than CLLP. However, since the verifier is a binary classifier, we expect two types of error to occur: accept invalid subgoals or reject valid subgoals. The verification algorithm accepts a subgoal if the verifier network values are above a certain threshold (likewise, they are rejected if the value is below another threshold), see $\texttt{t}_{\texttt{lo}}$ and $\texttt{t}_{\texttt{hi}}$ in Algorithm 3. In the remaining case, the algorithm falls back on CLLP to decide whether to keep or discard a given subgoal.

Value function. The value function is a neural network that estimates the negative distance between the current and goal states. The planner uses this information to select the next node to expand.

Adaptive Subgoal Search. The particular way in which AdaSubS chooses nodes to expand and a generator to produce subgoals (see highlighted lines in Algorithm 1) implements an adaptive mechanism that adjusts the planning horizon. The key difficulty to tackle here is that further subgoals, despite being capable of advancing the search faster, are more likely to be faulty. Nevertheless, we assume an optimistic approach prioritizing the longer distances (e.g., higher k). If the search using the long steps gets stuck, the planner retracts and expands the most promising, high-value nodes with closer, more conservative subgoals. The verifier network helps in mitigating the risks of this strategy, as it allows for the efficient rejection of faulty subgoals. This way, by traversing easier parts using fast long-distance subgoals and conservative ones in harder parts, AdaSubS adapts to the variable complexity of the environment.

Algorithm 1 presents a simple implementation of this approach. The nodes in the search tree are placed in a max-priority queue T with keys, being the pairs (k, v) of the next subgoal distance and its estimated value, sorted in lexicographical order. In this way, Algorithm 1 uses the highest k possible, searching with the BestFS strategy over values. If for a given k, all generated subgoals are invalid (faulty or unreachable), Algorithm 1 will expand for shorter distances. If successful, we go back to generating with the highest value of k. After reaching the target states, AdaSubS reconstructs the path of subgoals and fills it with low-level actions; see function LL_PATH in Algorithm 4.

With a slight modification, AdaSubS can be guaranteed to find a solution to any given problem, provided there is a large enough computational budget. See Appendix G.1 for details.

3.1 Training objectives

The components of AdaSubS are trained using a dataset of offline trajectories of subsequent states and actions: $(s_0, a_0), \ldots, (s_{n-1}, a_{n-1}), s_n$. We do not assume that they are perfect; for some of our environments, even randomly generated trajectories may turn out to be sufficient. Details on how the data is collected for each domain can be found in Section 4.1 and Appendix B.

Provided with such data, we train the k-generators to map s_i onto s_{i+k}. The value function is trained to map s_i onto $(i - n)$. CLLP is trained to map (s_i, s_{i+d}) onto a_i for every $d \leq k_{\text{max}}$ (k_{max} is the maximal distance of the subgoal generators used).

AdaSubS still works, albeit much worse if we disable the verifier network (e.g., by setting $\texttt{t}_{\texttt{hi}} = 1$ and $\texttt{t}_{\texttt{lo}} = 0$ in Algorithm 3). However, it is a useful setup to gather a dataset of subgoals and their reachability verified by CLLP. This dataset is used to train the verifier network, see Appendix D.

Algorithm 1 Adaptive Subgoal Search

```
Requires:   $C_1$                          max number of nodes
            $V$                            value function network
            $\rho_{k_0}, \dots, \rho_{k_m}$  subgoal generators
            SOLVED                         predicate of solution

function SOLVE(s_0)
    T ← ∅        ▷ priority queue with lexicographic order
    parents ← {}
    for $k$ in $k_0, \dots, k_m$ do
        T.PUSH((($k$, $V$(s_0)), s_0))
    seen.ADD(s_0)                                ▷ seen is a set
    while $0 <$ LEN(T) and LEN(seen) $< C_1$ do
        ($k$, _), s ← T.EXTRACT_MAX()
        subgoals ← $\rho_k$(s)
        for s' in subgoals do
            if s' in seen then continue
            if not IS_VALID(s, s') then
                continue
            seen.ADD(s')
            parents[s'] ← $s$
            for $k$ in $k_0, \dots, k_m$ do
                T.PUSH((($k$, $V$(s')), s'))
            if SOLVED(s') then
                return LL_PATH($s'$, parents)
                                ▷ get low-level path, see Alg. 4
    return False
```

Algorithm 2 Conditional low-level policy

```
Requires:   $C_2$    steps limit
            $\pi$    conditional low-level
                     policy network
            $M$      model of the environment

function GET_PATH(s_0, subgoal)
    step ← 0, s ← s_0
    action_path ← []
    while step $< C_2$ do
        action ← $\pi$.PREDICT(s, subgoal)
        action_path.APPEND(action)
        s ← $M$.NEXT_STATE(s, action)
        if s = subgoal then
            return action_path               ▷ success
        step ← step + 1
    return []                    ▷ subgoal is unreachable
```

Algorithm 3 Verification algorithm

```
Requires:   $v$       verifier network
            t_hi      upper threshold
            t_lo      lower threshold

function IS_VALID(s, s')
    if $v$(s, s') > t_hi then return True
    else if $v$(s, s') < t_lo then return False
    return GET_PATH(s, s') ≠ []
```

For INT and Rubik's Cube, we use transformer models for all the key components. For the Sokoban, we utilize convolutional networks, for details see Appendix B.

4 Experiments

We empirically demonstrate the efficiency of Adaptive Subgoal Search on three complex reasoning domains: Sokoban, Rubik's Cube, and the inequality proving benchmark INT (Wu et al., 2021). We demonstrate that AdaSubS is the best choice in a family of adaptive methods. Interestingly, even weaker methods in this class also outperform non-adaptive baselines. Finally, we show that AdaSubS has strong out-of-distribution generalization properties on INT.

As the performance metric, we use the success rate, defined as the fraction of solved problem instances. The computational budget is defined as the graph size, i.e., the number of nodes visited during the search and evaluated with a neural network (subgoal generator, value function, verifier, or conditional low-level policy). In Appendix C we provide details concerning the number of neural network calls, wall-time evaluations and memory usage.

4.1 Experimental domains and datasets

Sokoban is a puzzle in which the goal is to push boxes on target locations. It is a popular testing ground for classical planning methods (Lipovetzky & Geffner, 2012), and deep-learning approaches (Guez et al., 2019; Miłoś et al., 2019). Sokoban is considered to be hard (Fern et al., 2011) due to its combinatorial complexity. Finding a solution for a given Sokoban board is an NP-hard problem. In our experiments we used 12×12 Sokoban boards with four boxes.

Rubik's Cube is a famous 3D puzzle with over 4.3×10^{19} possible configurations (Korf, 1997). Recently Agostinelli et al. (2019); Czechowski et al. (2021) have developed methods for solving Rubik's Cube using neural networks.

INT is a benchmark for automated theorem proving proposed by (Wu et al., 2021). It consists of a generator of mathematical inequalities and a tool for proof verification. An action (proof step) in INT is a string containing an axiom and a specification of its input entities, making the action space effectively infinite and thus challenging search algorithms.

To collect offline trajectories datasets for Rubik's Cube, we generate random paths of length 20 starting from the solved cube and take them in reversed order. For INT we use the generator provided by Wu et al. (2021). For Sokoban, we use the expert data generated by a reinforcement learning agent (Miłoś et al., 2019). Detailed information is contained in Appendix D.

4.2 Protocol and baselines

Our protocol consists of three stages. In the first one, an offline dataset is prepared; see Section 4.1 and Appendix D. Secondly, we use this dataset to train the learnable components of AdaSubS: the family of subgoal generators, verifier network, and value network, see Section 3.1. Evaluation is the final step in which the algorithm's performance is verified. We measure its *success rate* using 1000 instances of a problem for each domain.

As baselines, we use BestFS and kSubS, both with the same models' checkpoints as AdaSubS. The former is a well-known class of search algorithms (including A^*), which, among others, performs strongly on problems with high combinatorial complexity (Pearl, 1984), achieves state-of-the-art results in theorem proving (Polu & Sutskever, 2020), and strong results on Rubik's Cube (Agostinelli et al., 2019; Czechowski et al., 2021). BestFS baseline selects actions with a trained policy network.

kSubS is the first general learned hierarchical planning algorithm proven to work on complex reasoning domains (Czechowski et al., 2021) (called BF-kSubS there), attaining good results on Sokoban and Rubik's Cube, and INT. kSubS can be view as a non-adaptive version of AdaSubS realized by a suitable hyperparameters choice: a single subgoal generator and inactive verifier ($\mathtt{t_{lo}} = 0$,$\mathtt{t_{hi}} = 1$).

For details on the hyperparameter choice for our method and the baselines, see Appendix F. For a more detailed description of the baselines, see Appendix E.

4.3 Main results: in- and out- of distribution performance

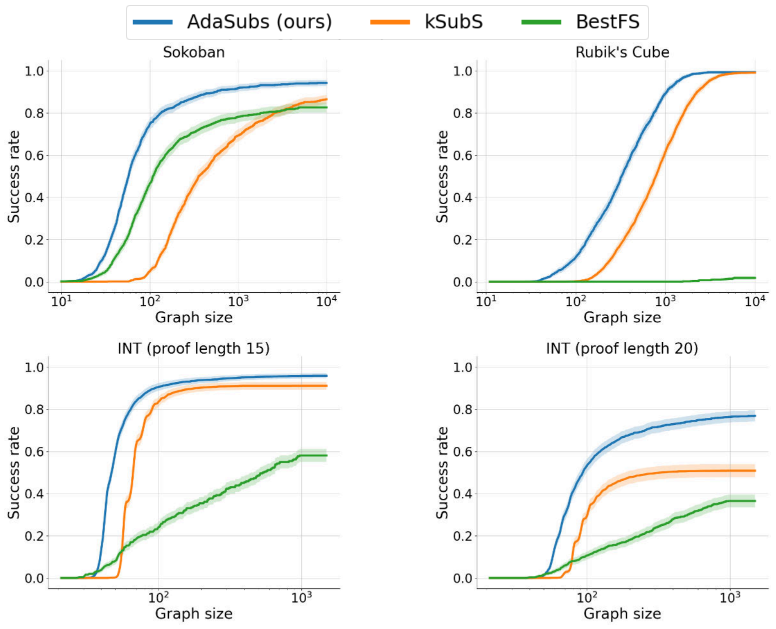


Figure 2: Success rates of AdaSubS, kSubS, and BestFS expressed in terms of graph size. The figure in the bottom right shows the out-of-distribution performance of methods evaluated on INT with proof length 20 but trained on length 15. The remaining figures present in-distribution performance. The results were measured on a fixed set of 1000 problems for each domain. Shaded areas indicate 95% confidence intervals.

AdaSubS shows strong in- and out- of distribution performance. The results for the former regime are presented in Figure 2, which shows that AdaSubS is able to make use of search capacity in the most effective manner, practically dominating other methods across the graph size spectrum. Taking a closer look at the low computational budgets, one can observe that AdaSubS achieves significantly positive success rates while the competing methods struggle. Perhaps the most striking difference is observed for INT, where at the budget of 50 nodes AdaSubS achieves around 60% success rate, while kSubS has a success rate close to zero and BestFS does not exceed 10%. This is particularly impressive since the budget of 50 nodes is only slightly larger than three times the considered proof length. To summarize, AdaSubS performs well in low computational regimes, which can be helpful in systems that need to solve search problems under compute or memory constraints.

At the far end of the computational budget spectrum, AdaSubS still performs the best, achieving above 90% performance in each environment ($\sim$ 95% for INT, 100% for Rubik's Cube, and 93% for Sokoban). Importantly, when success rates are high, and consequently the absolute differences between methods' results seem to be low, it is instructive to think about failure rates. For instance, in the case of INT (the proof length 15), the failure rate of kSubS is 9%, almost twice the failure rate of AdaSubS. For more results on low and high budgets, see Tables 10-12 in Appendix H.4.

For the out-of-distribution analysis, we used INT, an environment designed to study this phenomenon. We investigate how methods trained on the proofs of length 15 perform on problems with longer proofs (see Figure 3). The length 15 is the longest considered in the literature (Czechowski et al., 2021). However, we go much further, studying proof lengths up to 28. AdaSubS retains more than 50% of its performance, suffering a relatively slow decay of 3.5% (on average) per one step of the proof. This stands in stark contrast to kSubS, which already loses half of its performance at length 21. AdaSubS not only outperforms kSubS at each difficulty level but also achieves the most significant advantage in the hardest problems. Additionally, we provide a full profile of the success rate with respect to the graph size for proof length 20; see the bottom right-hand corner of Figure 2. AdaSubS performs much better than the baselines, with the biggest advantage for large budgets. Results for the other environments, namely Sokoban and Rubik, are included in Appendix K.

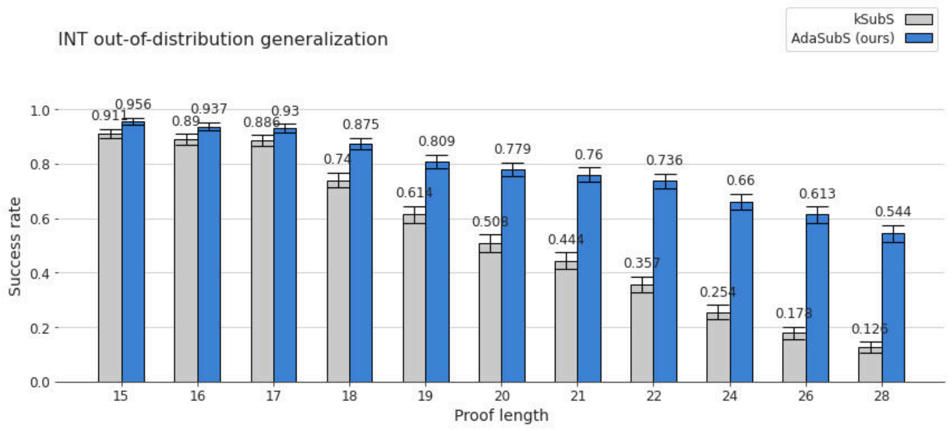


Figure 3: Out-of-distribution performance of AdaSubS and kSubS for long proofs in INT with budget of 5000 nodes. Both methods were trained on proofs of length 15. Error bars correspond to 95% confidence intervals.

The performance boost of AdaSubS over the baselines stems from two components: the verifier and the adaptativeness of subgoal selection. The former makes it possible to assign a bigger fraction of a computational budget on search by recovering a part of it from CLLP. This, in principle, could already provide significant gain when using the method. However, as shown in Table 1 and Tables 10-12 in Appendix H.4, the verifier helps, but only to a limited degree. Consequently, the majority of the improvement stems from the algorithmic novelty offered by the adaptive procedure. The adaptivity mechanism in AdaSubS creates this interesting dynamic that incentives the method to be optimistic about choosing subgoal distances while providing a safety net in case this optimism fails. How it works in practice can clearly be seen in Sokoban, where AdaSubS uses 8-subgoals 91.8% of the time, 4-subgoals 7.4% of the time, and 2-subgoals the remaining 0.8% of the time[2].

As a final note, Figure 2 can be used to infer the computational budget required for achieving a certain success rate. Additionally, the ratio of success rate to graph size can measure the efficiency

[2]Additionally, the 8-generator, the 4-generator, and the 2-generator generate subgoals that are on average 6.9, 3.9, and 2.0 steps away, respectively. For AdaSubS parameters, see Table 7 in Appendix F.

of the chosen budget, while the derivative of the success rate with respect to graph size provides the marginal utility of the increase in the budget.

4.4 Developing adaptive search methods

In this section, we present a comprehensive empirical study of four adaptive methods. This enables us to draw two main conclusions: adaptive search methods outperform non-adaptive ones, and *Longest-first* (on which AdaSubS is based) is the most efficient adaptive procedure. We present full results for INT, the hardest environment, and shortened results for Rubik and Sokoban, see Table 1. The complete set of numerical results with extended discussion can be found in Appendix H.

The four adaptive methods presented in this section are implemented using the search method[3]. Their adaptivity mechanism is defined by setting the way the subgoals are generated and the order in which the states are processed[4]. This happens in two distinguished lines in Algorithm 1 and changing them determines how the search prioritizes various distances.

INT					
		Small budget (50 nodes)		Large budget (1000 nodes)	
		with verifier	without	with verifier	without
BestFS		-	1.7%	-	36.7%
kSubS	$k = 4$	2.2%	0.1%	82.4%	83.0%
	$k = 3$	4.0%	0.2%	89.6%	90.7%
	$k = 2$	2.1%	0.5%	89.8%	91.7%
	$k = 1$	0.0%	0.0%	34.7%	46.0%
MixSubS	$k = [4, 3, 2]$	0.0%	0.0%	94.6%	95.0%
	$k = [3, 2, 1]$	0.0%	0.0%	92.2%	92.9%
	$k = [3, 2]$	17.0%	14.8%	92.2%	93.5%
Iterative mixing	iterations $= [1, 1, 1]$	32.0%	30.1%	87.0%	88.6%
	iterations $= [10, 1, 1]$	43.0%	44.8%	95.1%	96.0%
	iterations $= [4, 2, 1]$	54.0%	52.1%	93.6%	95.5%
Strongest-first		39.5%	40.8%	88.5%	89.8%
Longest-first		59.0%	51.5%	95.7%	95.5%

Rubik (with verifier)		
	400 nodes	6000 nodes
BestFS	0.0%	1.8%
kSubS	28.8%	98.6%
MixSubS	49.1%	99.2%
Iterative mixing	50.6%	99.1%
Strongest-first	33.4%	99.0%
Longest-first	58.0%	99.2%

Sokoban (small budget, 100 nodes)		
	with verifier	without
BestFS	-	45.9%
kSubS	26.0%	4.7%
MixSubS	52.7%	37.7%
Iterative mixing	64.5%	52.6%
Strongest-first	54.6%	41.9%
Longest-first	72.2%	63.4%

Table 1: *(left)* Results for the INT. For each case, unless stated otherwise, the distances of subgoal generators are $k = [3, 2, 1]$. *(right)* Shortened results for Rubik and Sokoban, for complete results see Table 11 and Table 12. The results were obtained on 1000 problems each, which yields $\pm 3\%$ Bernoulli 95% confidence intervals.

Specifically, we designed and tested the following methods: *MixSubS*, *Iterative mixing*, *Strongest-first*, *Longest-first*. Each method uses a set of n generators $\rho_{k_1}, \ldots, \rho_{k_n}$ trained to produce subgoals on different distances $k_1 < \ldots < k_n$ (recall Section 3.1 for training details). A more detailed description of the methods (and pseudocodes) can be found in Appendix H.

- *MixSubS* is the simplest approach, in which for each processed state we generate one subgoal from each generator ρ_{k_i} ($\texttt{subgoals} \leftarrow \cup_{j=1}^{n} \rho_{k_j}(\texttt{s})$). In each iteration, *MixSubS* chooses a state with the highest value estimation $V(s)$ to process.
- *Iterative Mixing* is similar to *MixSubS* and enables for advanced schedules of generators to be used. In the consecutive iterations, the i-th generator is used to expand l_i nodes before switching to the next generator. This allows us to flexibly prioritize the better generators, but at the cost of tuning additional hyperparameters $l_1, \ldots, l_n$. For these reasons, it is not practical, but useful as a reference point.
- *Strongest-first* uses one generator at a time ($\texttt{subgoals} \leftarrow \rho_{k_\ell}(\texttt{s})$), where k_ℓ is the longest distance not previously used in $\texttt{s}$. In each iteration, *Strongest-first* chooses a state with the highest value estimation $V(s)$ to process.
- *Longest-first* prioritizes long subgoals over the whole search procedure. Only if the queue does not contain any nodes with the highest k, it uses subgoals of lower distances. The nodes are processed in the order of their value estimation $V(s)$.

[3]Adaptivity may also be implemented using the subgoal generator. We considered various approaches in this category, however, they did not perform better than the non-adaptive baseline kSubS, see Appendix H.3. We speculate that assessing the state difficulty is a hard learning problem that is easier to handle via search.

[4]For similar considerations in classical planning, see multi-heuristic best-first search (Helmert, 2006).

The high-level empirical conclusion is that the performance of methods is roughly ordered as follows: *Longest-first* > *Iterative mixing* > *MixSubS* > *Strongest-first* > *kSubS* > *BestFS*.

In more detail, already the simple *MixSubS* works better than the non-adaptive baselines. In particular, it can outperform the maximum of performances of kSubS for each k. This is in line with the intuition that our mixing mechanism can elicit benefits of various distances while avoiding their drawbacks. We conjecture that whenever a single generator begins to struggle, the search advances with the help of another generator, allowing for stable progress. *Iterative mixing* is able to exhibit strong performance; however, it needs tedious schedule tuning for each domain.

Strongest-first and *Longest-first* implement bias towards longer distances. Even though they are quite similar, they display a large performance difference. We speculate that when *Strongest-first* encounters an area with falsely large value estimates, it wastes a lot of compute to examine it with all subgoal distances. On the other hand, *Longest-first* first explores other areas before using shorter subgoals and thus is able to avoid this problem. We stress that these effects are far from being obvious; however, they occur robustly across our test scenarios.

The verifier is beneficial on small budgets, especially when long subgoals are used. For large budgets, gains diminish. However, a properly tuned verifier never decreases the results significantly.

5 Limitations and future work

Determinism, access to model dynamics Our main focus is combinatorially complex domains. There are many applications of interest in which we can assume access to underlying dynamics and determinism (for example Automated Theorem Proving). Nevertheless, it is an interesting future direction to adjust our method to stochastic domains and learned models.

Reliance on the offline data In our experiments, we need offline datasets of successful trajectories. We leave for future work developing an algorithm based on the expert iteration paradigm.

Path optimization The goal of our algorithm is to find any path leading to the solution. In many real-world problems, it is also important to find a short path or one with a high reward (see Appendix I for optimality measures for Sokoban).

Completeness and memory constraints Our algorithm is not guaranteed to find a solution. We found it is not problematic in practice. If such a property is required, it can be assured by a simple change, see Appendix G.1. However, since AdaSubS keeps a list of visited nodes (see Algorithm 1) and caches some computations (see Algorithm 4), it requires memory proportional to the search budget, which can grow up to the size of the state space when seeking completeness.

Adversarial configuration The performance of AdaSubS can deteriorate when the ability to train its components is reduced and the environment is hard. For example, 'walk-the-line' environment (with states either lying on a unique solving trajectory or leading to deadstates) and no training data.

Combine with recursive search methods In some domains, one can generate useful subgoals for long distances and recursively split the problem (Pertsch et al., 2020a; Parascandolo et al., 2020; Jurgenson et al., 2020). It would be interesting to propose an algorithm that automatically detects when such an approach is possible and combine two ways (our and recursive) of generating subgoals.

6 Conclusions

We study planning methods that adapt to the local complexity of a solved problem. We concentrate on the adaptive selection of the subgoal distance realized by mixing various subgoal generators. We prove that methods based on this principle outperform non-adaptive counterparts They can tackle complex reasoning tasks as demonstrated on Sokoban, the Rubik's Cube, and INT. Our main algorithm, AdaSubS, is the best of the tested choices across all environments and search budgets. Interestingly, AdaSubS exhibits high out-of-distribution generalization capability, retaining much of its performance for instances of harder problems on INT than it was trained for.

7 Acknowledgments and Disclosure of Funding

The work of Michał Zawalski and Piotr Miłoś was supported by the Polish National Science Center grant 2021/43/O/ST6/02478. This research was supported by the PL-Grid infrastructure. Our experiments were managed using `https://neptune.ai`. We would like to thant the Neptune team for providing us access to the team version and technical support.

References

Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. Solving the rubik's cube with deep reinforcement learning and search. Nature Machine Intelligence, 1(8):356–363, 2019.

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil J Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, and Mengyuan Yan. Do as i can, not as i say: Grounding language in robotic affordances, 2022. URL `https://arxiv.org/abs/2204.01691`.

Cameron Allen, Michael Katz, Tim Klinger, George Konidaris, Matthew Riemer, and Gerald Tesauro. Efficient black-box planning using macro-actions with focused effects. IJCAI-21, 2021.

Marcin Andrychowicz, Anton Raichuk, Piotr Stanczyk, Manu Orsini, Sertan Girgin, Raphaël Marinier, Léonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, Sylvain Gelly, and Olivier Bachem. What matters in on-policy reinforcement learning? A large-scale empirical study. CoRR, abs/2006.05990, 2020. URL `https://arxiv.org/abs/2006.05990`.

Clemens Büchner, Patrick Ferber, Jendrik Seipp, and Malte Helmert. A comparison of abstraction heuristics for rubik's cube. In ICAPS 2022 Workshop on Heuristics and Search for Domain-independent Planning, 2022.

Rich Caruana. Multitask learning. Springer, 1998.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. arXiv preprint arXiv:2110.14168, 2021.

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. Introduction to algorithms. MIT press, 2009.

Konrad Czechowski, Tomasz Odrzygóźdź, Marek Zbysiński, Michał Zawalski, Krzysztof Olejnik, Yuhuai Wu, Łukasz Kuciński, and Piotr Miłoś. Subgoal search for complex reasoning tasks. Advances in Neural Information Processing Systems, 34:624–638, 2021.

Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. RL^2: Fast reinforcement learning via slow reinforcement learning. arXiv preprint arXiv:1611.02779, 2016.

Kuan Fang, Yuke Zhu, Animesh Garg, Silvio Savarese, and Li Fei-Fei. Dynamics learning with cascaded variational inference for multi-step manipulation. arXiv preprint arXiv:1910.13395, 2019.

Alan Fern, Roni Khardon, and Prasad Tadepalli. The first learning track of the international planning competition. Machine Learning, 84(1-2):81–107, 2011.

Maximilian Fickert. Adaptive search techniques in ai planning and heuristic search. 2022.

Wei Gao, David Hsu, Wee Sun Lee, Shengmei Shen, and Karthikk Subramanian. Intention-net: Integrating planning and deep learning for goal-directed autonomous navigation. In 1st Annual Conference on Robot Learning, CoRL 2017, Mountain View, California, USA, November 13-15, 2017, Proceedings, volume 78 of Proceedings of Machine Learning Research, pp. 185–194. PMLR, 2017. URL `http://proceedings.mlr.press/v78/gao17a.html`.

Arthur Guez, Mehdi Mirza, Karol Gregor, Rishabh Kabra, Sébastien Racanière, Théophane Weber, David Raposo, Adam Santoro, Laurent Orseau, Tom Eccles, et al. An investigation of model-free planning. In International Conference on Machine Learning, pp. 2464–2473. PMLR, 2019.

Malte Helmert. The fast downward planning system. Journal of Artificial Intelligence Research, 26: 191–246, 2006.

Matteo Hessel, Hubert Soyer, Lasse Espeholt, Wojciech Czarnecki, Simon Schmitt, and Hado van Hasselt. Multi-task deep reinforcement learning with popart. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 33, pp. 3796–3803, 2019.

Jeffrey R Hollerman, Leon Tremblay, and Wolfram Schultz. Involvement of basal ganglia and orbitofrontal cortex in goal-directed behavior. Progress in brain research, 126:193–215, 2000.

Dinesh Jayaraman, Frederik Ebert, Alexei A. Efros, and Sergey Levine. Time-agnostic prediction: Predicting predictable video frames. In 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net, 2019. URL `https://openreview.net/forum?id=SyzVb3CcFX`.

Tom Jurgenson, Or Avner, Edward Groshev, and Aviv Tamar. Sub-goal trees a framework for goal-based reinforcement learning. In International Conference on Machine Learning, pp. 5020–5030. PMLR, 2020.

Leslie Pack Kaelbling. Learning to achieve goals. In Ruzena Bajcsy (ed.), Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambéry, France, August 28 - September 3, 1993, pp. 1094–1099. Morgan Kaufmann, 1993.

Taesup Kim, Sungjin Ahn, and Yoshua Bengio. Variational temporal abstraction. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (eds.), Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada, pp. 11566–11575, 2019. URL `https://proceedings.neurips.cc/paper/2019/hash/b5d3ad899f70013367f24e0b1fa75944-Abstract.html`.

Sven Koenig and Maxim Likhachev. Real-time adaptive a. In Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems, pp. 281–288, 2006.

Richard E Korf. Finding optimal solutions to rubik's cube using pattern databases. In AAAI/IAAI, pp. 700–705, 1997.

Thanard Kurutach, Aviv Tamar, Ge Yang, Stuart J. Russell, and Pieter Abbeel. Learning plannable representations with causal infogan. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (eds.), Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada, pp. 8747–8758, 2018. URL `https://proceedings.neurips.cc/paper/2018/hash/08aac6ac98e59e523995c161e57875f5-Abstract.html`.

Nir Lipovetzky and Hector Geffner. Width and serialization of classical planning problems. In ECAI 2012, pp. 540–545. IOS Press, 2012.

Kara Liu, Thanard Kurutach, Christine Tung, Pieter Abbeel, and Aviv Tamar. Hallucinative topological memory for zero-shot visual planning. In Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event, volume 119 of Proceedings of Machine Learning Research, pp. 6259–6270. PMLR, 2020a. URL `http://proceedings.mlr.press/v119/liu20h.html`.

Yinhan Liu, Jiatao Gu, Naman Goyal, Xian Li, Sergey Edunov, Marjan Ghazvininejad, Mike Lewis, and Luke Zettlemoyer. Multilingual denoising pre-training for neural machine translation. CoRR, abs/2001.08210, 2020b. URL `https://arxiv.org/abs/2001.08210`.

Amol Mandhane, Anton Zhernov, Maribeth Rauh, Chenjie Gu, Miaosen Wang, Flora Xue, Wendy Shang, Derek Pang, Rene Claus, Ching-Han Chiang, et al. Muzero with self-competition for rate control in vp9 video compression. arXiv preprint arXiv:2202.06626, 2022.

Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL - The Planning Domain Definition Language, 1998.

Piotr Miłoś, Łukasz Kuciński, Konrad Czechowski, Piotr Kozakowski, and Maciek Klimek. Uncertainty-sensitive learning and planning with ensembles. arXiv preprint arXiv:1912.09996, 2019.

Bharath Muppasani, Vishal Pallagani, Kausik Lakkaraju, Biplav Srivastava, and Forest Agostinelli. Solving the rubik's cube with a pddl planner. 2022.

Suraj Nair and Chelsea Finn. Hierarchical foresight: Self-supervised learning of long-horizon tasks via visual subgoal generation. In 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020. OpenReview.net, 2020. URL `https://openreview.net/forum?id=H1gzR2VKDH`.

Giambattista Parascandolo, Lars Buesing, Josh Merel, Leonard Hasenclever, John Aslanides, Jessica B Hamrick, Nicolas Heess, Alexander Neitz, and Theophane Weber. Divide-and-conquer monte carlo tree search for goal-directed planning. arXiv preprint arXiv:2004.11410, 2020.

Judea Pearl. Heuristics: intelligent search strategies for computer problem solving. Addison-Wesley Longman Publishing Co., Inc., 1984.

Karl Pertsch, Oleh Rybkin, Frederik Ebert, Shenghao Zhou, Dinesh Jayaraman, Chelsea Finn, and Sergey Levine. Long-horizon visual planning with goal-conditioned hierarchical predictors. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (eds.), Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual, 2020a. URL `https://proceedings.neurips.cc/paper/2020/hash/c8d3a760ebab631565f8509d84b3b3f1-Abstract.html`.

Karl Pertsch, Oleh Rybkin, Jingyun Yang, Shenghao Zhou, Konstantinos G. Derpanis, Kostas Daniilidis, Joseph J. Lim, and Andrew Jaegle. Keyframing the future: Keyframe discovery for visual prediction and planning. In Alexandre M. Bayen, Ali Jadbabaie, George J. Pappas, Pablo A. Parrilo, Benjamin Recht, Claire J. Tomlin, and Melanie N. Zeilinger (eds.), Proceedings of the 2nd Annual Conference on Learning for Dynamics and Control, L4DC 2020, Online Event, Berkeley, CA, USA, 11-12 June 2020, volume 120 of Proceedings of Machine Learning Research, pp. 969–979. PMLR, 2020b. URL `http://proceedings.mlr.press/v120/pertsch20a.html`.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. arXiv preprint arXiv:2009.03393, 2020.

Silvia Richter and Matthias Westphal. The lama planner: Guiding cost-based anytime planning with landmarks. Journal of Artificial Intelligence Research, 39:127–177, 2010.

Stuart Russell and Peter Norvig. Artificial intelligence: A modern approach. ed. 3. 2010.

Nikolay Savinov, Alexey Dosovitskiy, and Vladlen Koltun. Semi-parametric topological memory for navigation. In 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings. OpenReview.net, 2018. URL `https://openreview.net/forum?id=SygwwGbRW`.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy P. Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. ArXiv, abs/1911.08265, 2019.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. ArXiv, abs/1712.01815, 2017.

Jane X Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn. arXiv preprint arXiv:1611.05763, 2016.

Yuhuai Wu, Albert Q. Jiang, Jimmy Ba, and Roger Baker Grosse. INT: an inequality benchmark for evaluating generalization in theorem proving. In 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021. OpenReview.net, 2021. URL `https://openreview.net/forum?id=O6LPudowNQm`.

Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Karol Hausman, Chelsea Finn, and Sergey Levine. Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning. In Conference on robot learning, pp. 1094–1100. PMLR, 2020.

Lunjun Zhang, Ge Yang, and Bradly C. Stadie. World model as a graph: Learning latent landmarks for planning. CoRR, abs/2011.12491, 2020. URL `https://arxiv.org/abs/2011.12491`.

A LOW-LEVEL PATH FUNCTION

The low-level path function (see LL_PATH, Algorithm 4) computes a path from the starting state to the goal state in the environment using low-level actions. However, it not only responsible for returning the path but also for checking false positive errors of the verifier. Specifically, the verifier can accept an unreachable state in Algorithm 3 and then wrongly include it in the solution path. Thus, LL_PATH has to construct a low-level path and confirm that every step on the way is achievable.

Algorithm 4 Low-level path

```
function LL_PATH(s, parents)
    ▷ parents is the dictionary of parent nodes in the subgoal tree. (S,C) $\in$ parents means that C is a subgoal for state S
    path $\leftarrow$ []
    while s in parents.KEYS() do
        subgoal_path $\leftarrow$ GET_PATH(parents[s], s)                    ▷ see Algorithm 2.
                                ▷ In practice, to reduce the number of neural network calls, we cache
                                ▷ the results of the GET_PATH calls in Alg 1 and reterive them here.
        if subgoal_path = [] then return False                            ▷ mistake of the verifier
        path $\leftarrow$ concatenate(subgoal_path, path)
        s $\leftarrow$ parents[s]
    return path
```

B TRAINING DETAILS

B.1 ARCHITECTURES

INT and Rubik's cube. All components of AdaSubS utilize the same architecture. Specifically, we used mBart, a transformer from the HuggingFace library (see (Liu et al., 2020b)). To make training of the model and the inference faster we reduced the number of parameters: we used 45M learned parameters instead of 680M in the original implementation. We used 6 layers of encoder and 6 layers of decoder. The dimension of the model was set to 512 and the number of attention heads to 8. We adjusted the size of the inner layer of position-wise fully connected to 2048. During the inference, we used beam search with width 16 for INT and width 32 for Rubik's Cube. Our implementation of the model follows (Czechowski et al., 2021, Appendix B.1)

Sokoban. We used four convolutional neural networks: the subgoal generator, conditional low-level policy, value, and the verifier. They all share the same architecture with a different last layer, depending on the type of output. Each model had 7 convolutional layers with kernel size (3,3) and 64 channels. Conditional low-level policy and verifier need two Sokoban boards as an input, so for these networks we concatenate them (across the last, depth dimension) and we treat two boards as one tensor. For the value function on top of a stack of convolutional layers there is a fully connected layer with 150 outputs representing 150 distances to the goal or. CLLP has analogous final layers with the one exception that there are only two classes: determining whether it is possible to reach a subgoal by CLLP or not. The network used for generatiing subgoals returns two outputs: distribution over possible modifications of a given state, and prediction whether a modified state is a good subgoal. The first output is obtained with a fully connected layer, the second with global average pooling followed by a fully connected layer. Generation of a single subgoal is realised as a sequence of calls to this network. We start from a given state and iteratively apply modifications with high probability assigned by the first head of the network, until the second head predict that no more iterations are needed. (see also Appendix G.1)

B.2 TRAINING PIPELINE

To ensure a fair comparison with (Czechowski et al., 2021) we followed their settings for a training pipeline.

INT and Rubik's Cube. To train the models we used the training pipeline from the HuggingFace library (Liu et al., 2020b). We trained our models from scratch without using any pretrained checkpoints. The size of the training batch was 32, the dropout was set to 0.1, and there was no label smoothing. We used the Adam optimizer with the following parameters: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$. We applied the warm-up learning schedule with 4000 warm-up steps and a peak learning rate of $3 \cdot 10^{-4}$. For inference in INT, we used temperature 1 and for Rubik's Cube to 0.5 (the optimal value was chosen experimentally).

Sokoban. To train of all networks we used a supervised setting with learning rate 10^{-4} and trained for 200 epochs. We used the Adam optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-7}$.

B.3 DATASETS

For dataset used to train all the network see Appendix D.

C COMPUTATIONAL BUDGET ANALYSIS

The default metric of the graph size that we use for comparisons counts all the states visited during the search, both high-level subgoals and intermediate states passed by the CLLP. It is a good estimate of the number of steps the algorithm takes to solve the given problem. For completeness, in this section, we analyze the total number of calls to every learned component of the pipeline for AdaSubS and the baseline kSubS.

Since all of the main components are deep neural networks, their evaluation time dominates the computational budget. Tables 3, 2 and 4 present the average number of calls to each component in 1000 test episodes, fixed for all the methods. That indicates which component consumes the largest part of the computational budget. The results are presented for different numbers of beams (see Appendix G.1) used for sampling predictions from the subgoal generators, the only component that outputs a set of predictions. The default number of beams was 16 for Sokoban and INT, and 32 for the Rubik's Cube (see Appendix F for the complete list of the parameters).

As the tables show, not only does AdaSubS solve more problems within smaller search graphs but also calls each component fewer times, which results in faster inference.

In the Rubik's Cube, the calls to the generators dominate the computations. However, when using smaller beams, this number can be significantly reduced while preserving the high success rate. In all the environments, AdaSubS is less sensitive to reducing the number of beams than kSubS in terms of performance. This is the case since in AdaSubS every single generator creates fewer subgoal candidates (see Tables 7-9), and thus it does not require a wide beam search. Therefore, by reducing the number of beams, AdaSubS can provide strong results within a much shorter time.

In Rubik's Cube and Sokoban, using the verifier in AdaSubS significantly reduces the number of calls to the low-level policy. However, in INT this is not the case. In most cases when kSubS fails to find a solution, at some point it cannot create any valid subgoal, and thus the search ends early. AdaSubS does not suffer from this issue, since it uses more generators. Thus, it counts the calls even from hard instances that require much larger graphs.

As shown in Table 5, if we count the calls only for the tasks solved by both methods, AdaSubS provides an advantage. Therefore, AdaSubS indeed provides better results within a smaller computational budget compared to kSubS.

Environment	Rubik's Cube					
Variant	kSubS (32 beams)	kSubS (4 beams)	AdaSubS (32 beams)	AdaSubS (8 beams)	AdaSubS (4 beams)	AdaSubS (2 beams)
Success rate	98.8	97.1	99.2	99.2	99	98.5
Generator calls	6085	852	8872	2205	1244	680
Verifier calls	0	0	277	275	311	340
Policy calls	1330	1526	352	350	395	446
Value calls	259	285	163	162	181	197
Total calls	7675	2664	9666	2994	2133	1665
Wall-time	86.3 sec	49.9 sec	96.1 sec	49.2 sec	47.3 sec	37.3 sec

Table 2: Comparison of the average number of calls to the generator, verifier, policy, and value networks for different numbers of beams (width of beam search in subgoal generation)) and the average wall time. Results were obtained using fixed 1000 instances of Rubik's Cube

C.1 MEMORY USAGE

AdaSubS keeps track of the search tree composed of high-level nodes. Thus, the amount of required memory grows linearly with the search budget. However, if we use longer subgoals, the tree is sparse because we do not store the nodes visited by the low-level policy.

Note that the BestFS baseline, which uses only low-level steps, usually requires much larger memory because it must record every step. In practice, when evaluating the BestFS baseline in the INT environment, we often had problems with experiments crashing because of exceeding the memory limit on the machine. We never observed similar issues when running AdaSubS.

Environment	Sokoban					
Variant	kSubS (16 beams)	kSubS (8 beams)	kSubS (4 beams)	AdaSubS (16 beams)	AdaSubS (8 beams)	AdaSubS (4 beams)
Success rate	84.4	84.6	82.3	94	94	94.1
Generator calls	2500	1281	746	3389	1692	848
Verifier calls	0	0	0	211	211	212
Policy calls	4576	4693	5554	248	247	247
Value calls	183	187	216	82	82	82
Total calls	7260	6161	6301	3931	2233	1391
Wall-time	48.7 sec	36.6 sec	33.6	54.4	39.3 sec	30.6 sec

Table 3: Comparison of the average number of calls to generator, verifier, policy, and value networks for different number of beams (width of beam search in subgoal generation)) and the average wall-time. Results were obtained using fixed 1000 instances of Sokoban

Environment	INT				
Variant	kSubS (16 beams)	kSubS (4 beams)	AdaSubS (16 beams)	AdaSubS (8 beams)	AdaSubS (4 beams)
Success rate	90.7	89.7	96	96	95.3
Generator calls	107	26.0	362	166	76.3
Verifier calls	0	0	67.9	62.2	57.2
Policy calls	378	366	801	738	659
Value calls	6.9	6.6	14.0	13.0	11.9
Total calls	492	399	1245	974	805
Wall-time	15.3 sec	12.1 sec	43.8 sec	31.9 sec	31.1 sec

Table 4: Comparison of the average number of calls to generator, verifier, policy, and value networks for different number of beams (width of beam search in subgoal generation)) and the average wall-time. Results were obtained using fixed 1000 instances of INT problems.

Environment	INT	
Variant	kSubS	AdaSubS
Generator calls	93.4	102
Verifier calls	0	19
Policy calls	328	300
Value calls	6.2	5.6
Total calls	428	427

Table 5: Comparison of the average number of calls to generator, verifier, policy, and value networks for problems solved by both methods for INT environment.

D Datasets and data processing

Sokoban. To collect offline data for Sokoban we used an MCTS-based RL agent from (Miłoś et al., 2019). In effect, the dataset consisted of all successful trajectories obtained by the agent: 154000 trajectories for 12x12 boards with four boxes. We use 15% of states from each trajectory to create the training dataset $\mathcal{D}$. We performed the split of dataset $\mathcal{D}$ into two parts of equal size: $\mathcal{D}_1$ and $\mathcal{D}_2$. The former was used to train the subgoal generators and conditional low-level policy, while the latter was used to train the verifier network. This split mitigates the possibility of the verifier's over-optimism concerning the probability of achieving subgoals by CLLP.

INT. We represent both states and actions as strings. For states, we used an internal INT tool for such representation. For actions, we concatenate one token representing the axiom and the arguments for this axiom (tokenized mathematical expressions) following (Czechowski et al., 2021).

To generate the dataset of successful trajectories we used the configurable generator of inequalities from the INT benchmark (see (Wu et al., 2021)). We adjusted it to construct trajectories of length 15 with all available axioms. The dataset used for our experiments consisted of $2 \cdot 10^6$ trajectories.

Rubik's Cube. To construct a single successful trajectory we performed 20 random permutations on an initially solved Rubik's Cube and took the reverse of this sequence, replacing each move with its reverse. Using this procedure we collected 10^7 trajectories. Such solutions are usually sub-optimal, since random moves are not guaranteed to increase the distance from the solution. They can even make loops in the trajectories.

D.1 Dataset for verifier

The verifier answers the question of whether a given subgoal is reachable by the CLLP. Thus, the dataset for training this component cannot be simply extracted from the offline trajectories.

To get the training samples for the Rubik's Cube and INT, we run AdaSubS without the verifier. In other words, we set $\texttt{t}_{\texttt{hi}} = 1$ and $\texttt{t}_{\texttt{lo}} = 0$, which essentially means that the reachability of all the subgoal candidates is checked solely by CLLP. During the searching, the generators create subgoal candidates, which are then verified by CLLP. Therefore, after working on some problem instances, we obtain a reach dataset of valid and not valid subgoals, marked by CLLP.

For the experiments in Sokoban, the limitation of the size of the offline dataset is an important factor for the final performance. Therefore, to ensure a fair comparison of AdaSubS with baselines, we do not generate additional solutions. Instead, we split the dataset as described above into $\mathcal{D}_1$ and $\mathcal{D}_2$ and used only $\mathcal{D}_2$ to generate data for the verifier. From every trajectory in $\mathcal{D}_2$, we sample some root states. For every such state, we use the subgoal generators to predict subgoal candidates. Then, CLLP checks the validity of each of them and we include them in the verifier training dataset.

After collection, it is essential to balance the dataset. Easy instances with short solutions provide fewer datapoints than hard tasks that require a deep search. Thus, it may happen that a substantial fraction of data collected this way comes from a single instance, reducing the diversity of the dataset. We have observed such issues, particularly in the INT environment. To prevent this, during the collection of the data for INT, we limit the datapoints that can be collected from a single problem instance to at most 100. This way, we collected about $5 \cdot 10^6$ training samples for the verifier for each domain.

E Baselines

As baselines, we use BestFS and BF-kSubS.

BestFS is a well-known class of search algorithms (including A^*), which, among others, performs well on problems with high combinatorial complexity (Pearl, 1984), achieves state-of-the-art results in theorem proving (Polu & Sutskever, 2020), and strong results on Rubik's Cube (Agostinelli et al., 2019; Czechowski et al., 2021).

Similarly to AdaSubS, BestFS iteratively expands the graph of visited states by choosing nodes with the highest value and adding its children to the priority queue. However, instead of using children from the subgoal tree, it uses direct neighbors in the environment space. In other words, we use a single policy network to generate neighbor subgoals in the distance of 1 action from a given node and treat it as a new subgoal. One can implement BestFS by replacing the call to a subgoal generator ρ_k in Algorithm 1 with ρ_{BFS}.

The implementation of ρ_{BFS} differs slightly between environments. In Rubik's Cube, ρ_{BFS} for each action estimates the probability that it leads to the solution. In every iteration, we take the top 3 predictions. In Sokoban, we use the same training objective, but instead of taking a fixed number of actions, we take the smallest subset of actions with estimated probabilities summing to at least 98%. For INT, we use beam search to generate high-probability actions. This is necessary since in INT the actions are represented as sequences.

Algorithm 5 BestFS

```
Requires:   $V$            value function network
            $\rho_{BFS}$   policy
            SOLVED         predicate of solution
function SOLVE($s_0$)
    T ← ∅                                                    ▷ priority queue
    parents ← {}
    T.PUSH(($V(s_0)$, $s_0$))
    seen.ADD($s_0$)                                          ▷ seen is a set
    while $0 <$ LEN(T) and LEN(seen) $< C_1$ do
        _, s ← T.EXTRACT_MAX()
        actions ← $\rho_{BFS}$(s)
        for a in actions do
            s′ ← ENV_STEP(s, a)
            if s′ in seen then
                continue
            if not IS_VALID(s, s′) then
                continue
            seen.ADD(s′)
            parents[s′] ← $s$
            T.PUSH(($V(s')$, s′))
            if SOLVED(s′) then
                return LL_PATH($s'$, parents)
                                              ▷ get low-level path, see Alg. 4
    return False
```

BF-kSubS is the first, according to our knowledge, general learned hierarchical planning algorithm shown to work on complex reasoning domains (Czechowski et al., 2021), attaining strong results on Sokoban and Rubik's Cube and state-of-the-art results on INT. BF-kSubS is a special case of AdaSubS with the following hyperparameters choice: a single subgoal generator and inactive verifier (with $t_{lo} = 0$ and $t_{hi} = 1$) in Algorithm 3).

F Hyperparameters

Environment	Sokoban	Rubik's Cube	INT
learning rate	10^{-4}	$3 \cdot 10^{-4}$	$3 \cdot 10^{-4}$
learning rate warmup steps	-	4000	4000
batch size	32	32	32
kernel size	[3, 3]	-	-
weight decay	10^{-4}	-	-
dropout	-	0.1	0.1

Table 6: Hyperparameters used for training.

Environment	Sokoban		
Method	kSubS	MixSubS	AdaSubS (ours)
number of subgoals	4	1	1
number of beams	16	16	16
beam search temperature	1	1	1
k-generators	8	[8, 4, 2]	[8, 4, 2]
number of steps to check (C_2)	10	[10, 6, 4]	[10, 6, 4]
max steps in solution check	-	18	18
max nodes in search tree (C_1)	5000	5000	5000
acceptance threshold of verifier (t_{hi})	-	0.99	0.99
rejection threshold of verifier (t_{lo})	-	0.1	0.1

Table 7: Hyperparameters used for evaluation in the Sokoban environment.

Environment	the Rubik's Cube		
Method	kSubS	MixSubS	AdaSubS (ours)
number of subgoals	3	1	1
number of beams	32	32	32
beam search temperature	0.5	0.5	0.5
k-generators	4	[4, 3]	[4, 3, 2]
number of steps to check (C_2)	4	[4, 3]	[4, 3, 2]
max steps in solution check	-	4	4
max nodes in search tree (C_1)	5000	5000	5000
acceptance threshold of verifier (t_{hi})	-	0.995	0.995
rejection threshold of verifier (t_{lo})	-	0.0005	0.0005

Table 8: Hyperparameters used for evaluation in the Rubik's Cube environment.

Most of the hyperparameters, both for training and evaluation, are either default or have little impact on the performance of the algorithms. The values are in line with (Czechowski et al., 2021), which ensures fair comparison.

The parameter C_1 (see Algorithm 1) controls the number of high-level nodes in the search tree. It is lower than the actual graph size that we use for comparisons since it counts neither the intermediate states visited by CLLP nor the subgoals that turned out to be invalid. That hyperparameter was chosen to enable all the algorithms evaluated to reach the graph size values used for comparison in Figure 2 and others in Section 4.

F.1 Tuning the hyperparameters

The most important training hyperparameter that has to be tuned is the learning rate. To do this, we compared the prediction accuracy for ten training runs corresponding to ten learning rates in

Environment	INT		
Method	kSubS	MixSubS	AdaSubS (ours)
number of subgoals	4	2	3
number of beams	16	16	16
beam search temperature	1	1	1
k-generators	3	[3, 2, 1]	[3, 2, 1]
number of steps to check (C_2)	3	[3, 2, 1]	[3, 2, 1]
max steps in solution check	-	5	5
max nodes in search tree (C_1)	400	400	400
acceptance threshold of verifier (t_{hi})	-	1	1
rejection threshold of verifier (t_{lo})	-	0.1	0.1

Table 9: Hyperparameters used for evaluation in the INT environment.

the range $[10^{-5}; 10^{-3}]$. We shared the training hyperparameters across all the components for each environment, as they share the same network architecture.

In evaluation, the most important hyperparameter of AdaSubS is the set of k-generators. To select the set for Sokoban, we started with small values of k (e.g., 1 or 2) and increased k multiplicatively, doubling it as long as the new set of generators performed better. We used a similar procedure in Rubik's Cube and INT, but due to the bounds on the optimal path length (at most 26 and 15, respectively), we increased k additively (incrementing k by one). This way, we have chosen $k = [8, 4, 2]$ for Sokoban, $k = [4, 3, 2]$ for Rubik's Cube, and $k = [3, 2, 1]$ for INT. When evaluating a set of generators, we also need to set the number of subgoals each generator should output. Thus, each time we tried three different values ($\{1, 2, 3\}$ for Rubik's Cube and Sokoban, $\{2, 3, 4\}$ for INT) and used the one that resulted in the highest success rate. We ran the evaluation pipeline 15 times to tune that parameter. The other parameters have rather minor impact on the results, and thus we did not tune them extensively.

For kSubS, we took the values of k used in (Czechowski et al., 2021) for Rubik's Cube and INT. For Sokoban, we use $k = 8$ since it outperforms $k = 4$ proposed in this work.

In the case of the verifier thresholds, we want high precision and high recall (see discussion in Section 4.5). While "loose" thresholds increase the processing speed, "tight" thresholds usually result in higher solved rates. To solve this trade-off, we propose to set the thresholds giving roughly 99% recall for the lower threshold and 99% precision for the upper. After estimating the parameters of the verifiers, it turned out that in the case of Sokoban, Rubik's Cube, and INT, the thresholds should be in the intervals $[0; 0.1]$ and $[0.9; 1]$. The final values were determined by running a grid search over five values in each interval. That required running the evaluation about 25 times. Note that since the verifier is used to increase the processing speed, it is enough to tune the thresholds only for the final version of the pipeline.

We train INT and Rubik's Cube components on a single GPU for about three days. The components for Sokoban are trained on CPUs for about a day. The evaluations in all the environments were also performed on CPU nodes. See Appendix J for the details of the actual infrastructure used in this project. Therefore, to properly tune the parameters, requires ten training runs on GPU and 40 evaluations on CPU nodes for each environment.

G Components of AdaSubS

G.1 Subgoal generators

The main purpose of the subgoal generator is to propose subgoal candidates in every iteration of the planner loop. That is, given the current state s of the environment it should output other states that are a few steps closer to goal g.

We train a k-generator by extracting training data from successful trajectories. Let $s_0, s_1, \ldots, s_n$ be a trajectory that leads to the goal s_n. For every state s_i we train the k-generator network to output the state s_{i+k}, exactly k steps ahead. Provided with a dataset of trajectories, this is a supervised objective. Clearly, a state that is k steps ahead does not need to be exactly k steps closer to the solution, especially if the trajectories include noise or exploration. However, it is guaranteed to be at most k steps closer, which enables reliable limits to be set for checking reachability.

In a simple approach, k is a hyperparameter that needs to be fixed, as proposed by (Czechowski et al., 2021). However, this is a strong limitation if the environment exhibits a variable complexity problem. Therefore, AdaSubS instead uses a set of generators, trained for different values of k. This way, the planner can adjust the expansion to match the local complexity of the problem. Additionally, training a set of generators can be easily parallelized.

For our generators, we use the transformer architecture. The input state is encoded as a sequence of tokens, as described in (Czechowski et al., 2021, Appendix C). The network produces another sequence of tokens on the output, which is then decoded to a subgoal state. The output sequence is optimized for the highest joint probability with beam search: the consecutive tokens are sampled iteratively and a fixed number of locally best sequences passes to the next iteration. This way, the generator enables sampling of a diverse set of subgoals by adjusting the beam width and sampling temperature. The exact number of the subgoals that the generators output are given in Appendix F.

As noted in Section 3.1, for the Sokoban environment instead of transformers we use simple convolutional networks. In this domain, the subgoal is created by a sequence of changes to the input state. The generator network is trained to predict the probability of changing every pixel. Then, the subgoals are obtained as a sequence of modifications that maximize joint probability. For simplicity, in AdaSubS we use beam search for all domains, including Sokoban.

During the inference, the generators output a limited set of subgoals to explore. Thus, if the generators are not trained well enough, even with an infinite computational budget, the search may fail to find a solution to the given problem, even if one exists. However, with a slight modification, AdaSubS can be guaranteed to find a solution to any given problem (or correctly report that the solution does not exist). We achieve that by adding an exhaustive single-step policy as the last generator. It would populate an empty queue with all children of the highest valued but not yet expanded node. Note that such a modification never decreases the score since the dummy generator is only used when a search is about to fail. In practice, this type of modification is not necessary to obtain strong results.

G.2 Conditional low-level policy (CLLP)

If we want to add a subgoal candidate to our search tree, we need to check whether it is reachable from the current state. This can be done using CLLP – a mapping that given a state and a subgoal produces a sequence of actions that connects those configurations, or claims that there is no such configuration. Specifically, the policy network, given the state and subgoal, iteratively selects the best action and executes it until the subgoal is reached or a threshold number of steps is exceeded, as shown in Algorithm 2.

CLLP is trained to imitate the policy that collected the training data. For every pair of states s_i, s_j that are located at most d steps from each other, it is trained to predict action a_i, taken in state s_i. Such action may not be optimal but usually it leads closer to s_j. The threshold d controls the range of the policy, as it is trained to connect states that are at most d steps away. Thus, it is essential to set the hyperparameter d to a value that is greater than the distances of all the generators used.

It is essential that the policy can successfully reach the correct subgoals, as it is a necessary condition for adding them to the search tree. The training metrics show that in all our environments the policy can reach more than 95% of correct subgoals. This percentage is even higher for short distances.

G.3 Verifier

To check whether a k-subgoal is reachable with the conditional policy, we need to call it up to k times. If we decide to use generators with long horizons, it becomes a significant computational cost. To mitigate this issue, we use the verifier that estimates the validity of a subgoal candidate in a single call. During the search, the generated subgoal candidates are evaluated by the verifier. For each of them, it estimates whether they are valid and outputs its confidence. If the returned confidence exceeds a fixed threshold, we do not run the costly check with the conditional policy. We perform such a check only in case the verifier is uncertain (see Algorithm 3).

At the end of the search, when a solving trajectory is found, we need to find the paths between all the pairs of consecutive subgoals that were omitted due to the verifier (see Algorithm 4). Since the length of the final trajectory is usually much smaller than the search tree, that final check requires much less computations.

It should be noted that the verifier estimates validity with respect to the conditional policy that is used. In case a valid subgoal is generated but the policy cannot reach it for some reason, it cannot be used to build the search tree anyway, for no solution that uses it can be generated in the final phase. Thus, the verifier should be trained to predict whether the CLLP that is used can reach the subgoal, rather than whether it is reachable by an optimal policy.

To train the verifier, we run our pipeline on some problem instances. All the subgoals created by the generators are validated with CLLP. This way, eventually we obtain a dataset of reachable and unreachable subgoal candidates. We train the verifier network to fit that data. Unlike the other components, training the verifier does not require access to any trajectories, only to a number of problem instances.

G.4 Value function.

The value function $V : \mathcal{S} \to \mathbb{R}$ estimates the negative distance between the current state s and the goal state g. During the search, this information is used to select the most promising nodes to expand. For every trajectory $s_0, \dots, s_n$ in the dataset it is trained to output $i - n$ given s_i. We opted for a simple training objective but any value function can be used in the algorithm.

G.5 Quality of AdaSubS components

The effectiveness of AdaSubS depends on the quality of its trainable components: below we present an analysis of generators and verifiers.

Generator: k trade-off. The quality of generators deteriorates when k is increased. In Sokoban, nearly 90% of subgoals created with the 4-generator are valid (according to CLLP). However at the same time for the 16-generator, this figure drops to 50%. Similarly, in Rubik's Cube, about 82% of subgoals proposed by the 4-generator are valid, while the 3-generator has over 99% accuracy. On the other hand, longer distances are beneficial to the search as they make it possible to build sparser search trees and achieve a solution faster. It turns out that the optimal choice of k depends on the search budget. It pays to be optimistic (i.e., choose long distances) for small budgets, while more prudent choices have the upper hand when more compute is available. Crucially, AdaSubS with multiple generators successfully resolves the trade-off, outperforming every single generator, see Figure 4.

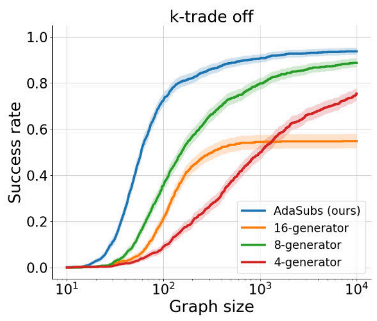
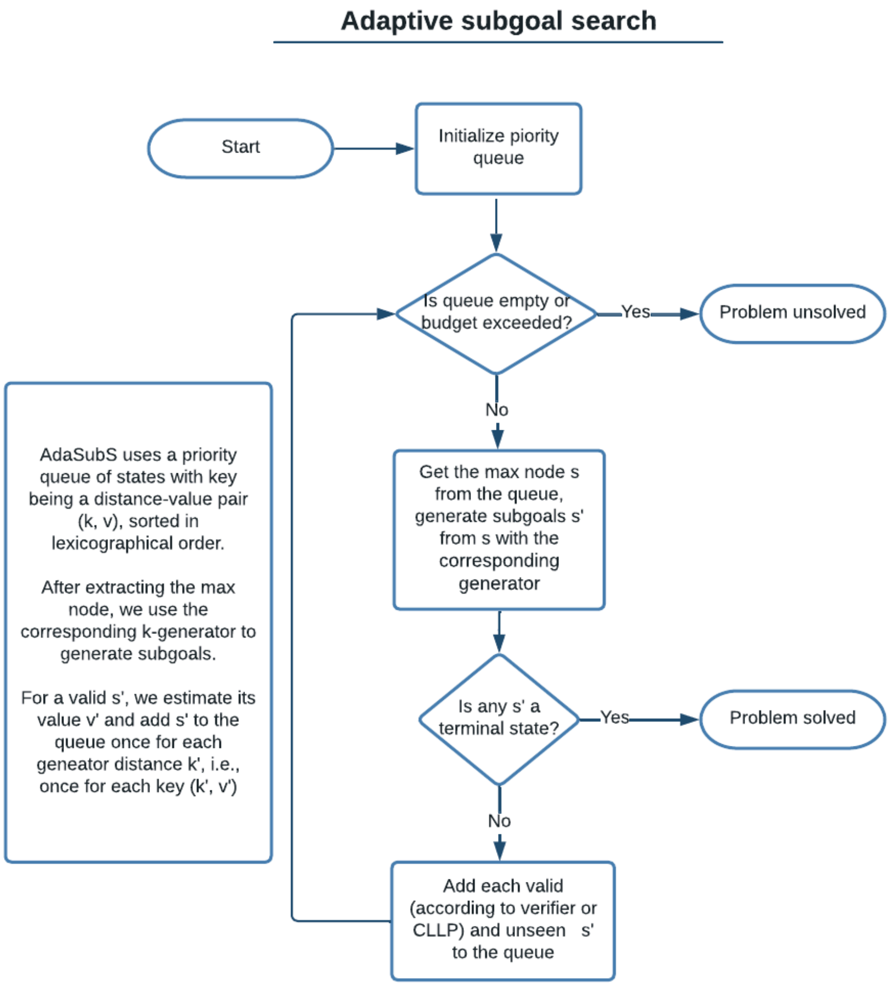


Figure 4: Comparison of success rates for different subgoal generators for Sokoban. AdaSubS-k describes using a single generator with distance k.

Verifier: precision and recall For each subgoal, the verifier outputs a classification probability p, which is used to accept the subgoal (if $p > t_{hi}$), reject the subgoal (if $p < t_{lo}$) or to pass to the further verification (if $p \in [t_{lo}, t_{hi}]$) by CLLP. For the acceptance task, we require high precision, as one false positive can lead to the failure of the whole search procedure. This leads to a high value for the corresponding parameter t_{hi} (e.g. for Sokoban $t_{hi} = 0.99$, which corresponds to a precision rate of 97%). For the rejection task, we do not wish to reject true positives. In other words, we aim for high recall. In this case, errors usually increase the search cost; the corresponding parameter t_{lo} is set to $t_{lo} = 0.1$ for Sokoban (this gives the recall of 99%). Additionally, for the verifier to be useful, we need to avoid passing subgoals to the further verification $p \in [t_{lo}, t_{hi}]$. It turns out that for the thresholds selected, the verifier is able to assess 70% states without the assistance of CLLP.

G.6 Adaptive Subgoal Search flowchart


Adaptive subgoal search
Start
Initialize piority queue
Is queue empty or budget exceeded?
Yes
Problem unsolved
No
Get the max node s from the queue, generate subgoals s' from s with the corresponding generator
Is any s' a terminal state?
Yes
Problem solved
No
Add each valid (according to verifier or CLLP) and unseen s' to the queue
AdaSubS uses a priority queue of states with key being a distance-value pair (k, v), sorted in lexicographical order.
After extracting the max node, we use the corresponding k-generator to generate subgoals.
For a valid s', we estimate its value v' and add s' to the queue once for each geneator distance k', i.e., once for each key (k', v')

H Developing adaptive search methods

There are many natural ways to incorporate adaptivity into the subgoal search pipeline. We experimented with several designs to find one that gives strong results in any domain. Here we provide a detailed description of all the variants tested and the numerical results of their evaluation in our environments. Their implementations can be found in Section H.2.

An adaptive algorithm should adjust the complexity of the proposed subgoals to the local complexity of the environment in the neighbourhood of the processed state. This can be realized using the following two approaches:

- Use an adaptive planner that provides a list of k-generators, the most promising node in every step selects and a generator to expand it.
- Use an adaptive subgoal generator that instead of proposing fixed-distance subgoals learns to automatically adjust the distance.

H.1 Adaptive planners

When implementing adaptivity with the planner, we need to specify a list of k-generators $\rho_{k_0}, \ldots, \rho_{k_m}$. In every iteration, the algorithm will select a node to expand and generators from the list that will create the new subgoals. This way, it can directly control the complexity of the subgoals and adapt to the current state and progress of the search.

MixSubS. Given a list of trained k-generators, a simple approach is to call all of them each time a node is expanded. In every iteration, we choose the node with the highest value in the tree and add subgoals proposed by each generator ρ_{k_0} to ρ_{k_m}. See Algorithm 6 for the implementation.

Observe that in the easy areas of the environment the search will progress fast, since the furthest subgoal will most likely have the highest value, so it will be chosen as the next node to expand. On the other hand, in the hard parts the shortest generators are more likely to provide subgoals that advance towards the target at least a step.

This method already achieve superior results compared to single generators, both on small and large budget. In the Rubik environment, it even reaches 100% solved cubes. MixSubS provides the advantage of planning with different horizons, but at the same time, it produces many unnecessary nodes in the easy areas, while taking only long steps is sufficient to solve the task. Additionally, one may want to prioritize the generators that perform better, which cannot be done with this method.

Iterative mixing. In this approach, we specify a number of iterations l_i for each generator ρ_{k_i}. We use ρ_{k_0} to expand the highest-valued nodes in the first l_0 iterations. Then, we use ρ_{k_1} to expand the best nodes in the following l_1 iterations and the procedure follows for the consecutive generators. After finishing with the last one, we start again from the beginning. See Algorithm 7 for the implementation.

This algorithm offers the flexibility of specifying the exact number of iterations for each generator, which forms an explicit prioritization. It can resemble some of the listed algorithms for carefully chosen l_i. However, tuning the number of iterations requires much more effort than the other parameter-free algorithms do. Therefore, we experimented with another two mixing approaches that select the generator automatically in every iteration.

Strongest-first. Another natural implementation of the planner is to choose the node with the highest value and expand it with the longest generator that was not used there yet. See Algorithm 8 for the implementation. While this greedy approach maintain clear advantage over single generators, it is outperformed by most of the mixing methods, even the simple mixes. We hypothesize that this method is more sensitive to the errors of the value function – if the search enters an area that the value function estimates too optimistically, it spends too much time trying to exploit it.

Longest-first (used by AdaSubS). This method in every iteration selects the longest generator that has at least one node to expand and highest-valued node for that generator in the queue. This way, it explicitly prioritizes using the longest generators and turns to the shorter only when the search is stuck. See Algorithm 9 for the implementation. As shown in the tables below, this method outperforms all other designs, in all the environments and within all budget constraints. It prioritizes the better generators, but does not require any additional hyperparameters to by specified. Therefore, we consider it the best mixing algorithm and use in AdaSubS as the default planner.

H.2 Adaptive planners implementations

In this section we provide the implementations of the planners. The lines highlighted in blue indicate the differences with the AdaSubS code. All the methods require specifying the list of generators $\rho_{k_0}, \dots, \rho_{k_m}$. The Iterative mixing planner additionally requires a list of iterations $l_0, \dots, l_m$.

Algorithm 6 MixSubS

```
function SOLVE($s_0$)
    $T \leftarrow \emptyset$                                   ▷ priority queue
    parents ← {}
    T.PUSH(($V(s_0), s_0$))
    seen.ADD($s_0$)
    while $0 <$ LEN(T) and LEN(seen) $< C_1$ do
        $\_, s \leftarrow$ T.EXTRACT_MAX()
        subgoals ← $\{\rho_{k_1}(s), \dots, \rho_{k_m}(s)\}$
        for $s'$ in subgoals do
            if $s'$ in seen then continue
            if not IS_VALID($s, s'$) then
                continue
            seen.ADD($s'$)
            parents[$s'$] ← $s$
            T.PUSH(($V(s'), s'$))
            if SOLVED($s'$) then
                return LL_PATH($s'$, parents)
    return False
```

Algorithm 7 Iterative mixing

```
function SOLVE($s_0$)
    $T_{k_i} \leftarrow \emptyset$                        ▷ $m + 1$ priority queues
    parents ← {}
    for $k$ in $k_0, \dots, k_m$ do
        $T_k$.PUSH(($V(s_0), s_0$))
    seen.ADD($s_0$)
    cnt ← 0                                     ▷ Iterations counter
    id ← 0                                      ▷ Current generator id
    while $0 <$ LEN(T) and LEN(seen) $< C_1$ do
        if cnt $= l_{id}$ or LEN($T_{k_{id}}$) $= 0$ then
            id ← (id + 1)%($m + 1$), cnt ← 0
        cnt ← cnt + 1
        $\_, s \leftarrow T_{k_{id}}$.EXTRACT_MAX()
        subgoals ← $\rho_{k_{id}}(s)$
        for $s'$ in subgoals do
            if $s'$ in seen then continue
            if not IS_VALID($s, s'$) then
                continue
            seen.ADD($s'$)
            parents[$s'$] ← $s$
            for $k$ in $k_0, \dots, k_m$ do
                $T_k$.PUSH(($V(s'), s'$))
        if SOLVED($s'$) then
            return LL_PATH($s'$, parents)
    return False
```

Algorithm 8 Strongest-first

```
function SOLVE($s_0$)
    $T \leftarrow \emptyset$    ▷ priority queue with lexicographic order
    parents ← {}
    for $k$ in $k_0, \dots, k_m$ do
        T.PUSH((($V(s_0), k$), $s_0$))
    seen.ADD($s_0$)
    while $0 <$ LEN(T) and LEN(seen) $< C_1$ do
        ($\_, k$), $s$ ← T.EXTRACT_MAX()
        subgoals ← $\rho_k(s)$
        for $s'$ in subgoals do
            if $s'$ in seen then continue
            if not IS_VALID($s, s'$) then
                continue
            seen.ADD($s'$)
            parents[$s'$] ← $s$
            for $k$ in $k_0, \dots, k_m$ do
                T.PUSH((($V(s'), k$), $s'$))
            if SOLVED($s'$) then
                return LL_PATH($s'$, parents)
    return False
```

Algorithm 9 Longest-first

```
function SOLVE($s_0$)
    $T \leftarrow \emptyset$    ▷ priority queue with lexicographic order
    parents ← {}
    for $k$ in $k_0, \dots, k_m$ do
        T.PUSH((($k, V(s_0)$), $s_0$))
    seen.ADD($s_0$)
    while $0 <$ LEN(T) and LEN(seen) $< C_1$ do
        ($k, \_$), $s$ ← T.EXTRACT_MAX()
        subgoals ← $\rho_k(s)$
        for $s'$ in subgoals do
            if $s'$ in seen then continue
            if not IS_VALID($s, s'$) then
                continue
            seen.ADD($s'$)
            parents[$s'$] ← $s$
            for $k$ in $k_0, \dots, k_m$ do
                T.PUSH((($k, V(s')$), $s'$))
            if SOLVED($s'$) then
                return LL_PATH($s'$, parents)
    return False
```

H.3 Adaptive generators

A k-generator is trained to propose subgoals that should be exactly k steps ahead. However, instead of matching a fixed distance, it can opt for long subgoals when the next steps are clear and short when difficulties appear, or both if it is not certain.

Implementing this idea requires changing the training of the generator. Given a training trajectory, for each state s_i we need to select the target state $s_{t(i)}$ that should be the output of the generator. We tested a few methods that select this target.

Longest-reachable We use the low-level conditional policy to estimate the local complexity around s_i. Specifically, we choose $s_{t(i)}$ to be the furthest state on the trajectory such that it is reachable from s_i with the CLLP and so do all its predecessors. In other words, we check whether CLLP starting in s_i can reach s_{i+1}, s_{i+2}, etc. When we find the first state s_j that is not reachable, we set $t(i)$ to be $j - 1$.

Intuitively, this approach makes the generator learn to output subgoals as distant as possible, but still reachable for CLLP. However, this way the targets are selected on the borderline of reachability, which may lead to too hard subgoals in some cases.

Sampling-reachable To make target state selection more robust, we modify reachability verification. Instead of greedily following the best action determined by CLLP probabilities, in every step we sample the action. This way, we are more likely to take suboptimal actions, so the selected target should be reachable with higher confidence.

Secondary-reachable Another method of making more robust selection is to follow the action with the lowest probability that exceeds a fixed threshold, e.g. 25%. Intuitively, we follow the action that CLLP considers to be good, but is less certain than in the case of the highest-ranked. Therefore, a subgoal reached in this way should be reachable with even higher confidence when following the greedy actions.

Our experiments show that the adaptive generators trained according to those designs perform well in the environments we consider. For instance, all the methods nearly reach a 90% solve rate on Sokoban. However, none of them provide better results than the kSubS baseline. Therefore in this work we focus on planner-based adaptivity and leave tuning the adaptive generators pipeline for future work.

H.4 Benchmarking results

Tables 10-12 show the numerical results achieved by the adaptive planners described in section H.1, compared to baselines: BestFS and kSubS. For some of the methods a few variants are provided. In each table, the longest-first, strongest-first and iterative mixing methods use the same set of generators: $[3, 2, 1]$ for INT, $[4, 3, 2]$ for Rubik, and $[8, 4, 2]$ for Sokoban. Our main algorithm, Adaptive Subgoal Search, uses the longest-first planner and the verifier network.

INT					
		Small budget (50 nodes)		Large budget (1000 nodes)	
		with verifier	without	with verifier	without
BestFS		-	1.7%	-	36.7%
kSubS	k=4	2.2%	0.1%	82.4%	83.0%
	k=3	4.0%	0.2%	89.6%	90.7%
	k=2	2.1%	0.5%	89.8%	91.7%
	k=1	0.0%	0.0%	34.7%	46.0%
MixSubS	k=[4,3,2]	0.0%	0.0%	94.6%	95.0%
	k=[3,2,1]	0.0%	0.0%	92.2%	92.9%
	k=[3,2]	17.0%	14.8%	92.2%	93.5%
Iterative mixing	iterations=[1,1,1]	32.0%	30.1%	87.0%	88.6%
	iterations=[10,1,1]	43.0%	44.8%	95.1%	96.0%
	iterations=[4,2,1]	54.0%	52.1%	93.6%	95.5%
Strongest-first		39.5%	40.8%	88.5%	89.8%
Longest-first (AdaSubS)		59.0%	51.5%	95.7%	95.5%

Table 10: INT benchmark

Rubik					
		Small budget (400 nodes)		Large budget (6000 nodes)	
		with verifier	without	with verifier	without
BestFS		-	0.0%	-	1.8%
kSubS	k=4	28.8%	24.5%	98.6%	98.8%
	k=3	19.3%	18.6%	95.6%	95.4%
	k=2	8.2%	4.5%	99.0%	95.8%
	k=1	0.5%	0.5%	76.5%	76.5%
MixSubS	k=[4,3,2]	29.1%	20.9%	99.1%	100.0%
	k=[4,3]	49.1%	45.1%	99.2%	100.0%
Iterative mixing	iterations=[1,1,1]	33.5%	23.0%	99.2%	100.0%
	iterations=[10,1,1]	50.6%	43.6%	99.1%	99.9%
	iterations=[4,2,1]	48.4%	41.2%	99.2%	100.0%
Strongest-first		33.4%	27.1%	99.0%	99.9%
Longest-first (AdaSubS)		58.0%	52.4%	99.2%	100.0%

Table 11: Rubik benchmark

Sokoban					
		Small budget (100 nodes)		Large budget (5000 nodes)	
		with verifier	without	with verifier	without
BestFS		-	45.9%	-	82.6%
kSubS	k=16	13.7%	5.1%	60.5%	63.5%
	k=8	26.0%	4.7%	85.6%	84.4%
	k=4	8.2%	2.6%	68.1%	65.5%
	k=2	1.4%	0.7%	40.0%	38.3%
MixSubS	k=[8,4,2]	52.7%	37.7%	91.7%	90.2%
	k=[16,8,4]	55.6%	44.9%	89.1%	89.0%
Iterative mixing	iterations=[1,1,1]	52.7%	37.7%	91.7%	90.2%
	iterations=[10,1,1]	68.3%	58.6%	92.5%	92.1%
	iterations=[4,2,1]	64.5%	52.6%	93.5%	93.2%
Strongest-first		54.6%	41.9%	92.0%	90.8%
Longest-first (AdaSubS)		72.2%	63.4%	93.4%	93.6%

Table 12: Sokoban benchmark

I Optimality of solutions for Sokoban

We did an additional experiment for Sokoban: we used simple BFS to find the optimal path, and we compared its length with the outcome of AdaSubS. The results are as follows (tested on 600 Sokoban boards):

- For 7% of boards, AdaSubS found the optimal solution.
- The average difference between the length of the AdaSubS solution and the optimal solution is 15 steps.
- On average, the AdaSubS solution is 38% longer than the optimal one.

J Infrastructure used

We performed experiments using two types of hardware: with and without access to GPUs. In the former, we used nodes equipped with a single Nvidia V100 32GB card or Nvidia RTX 2080Ti 11GB card. Each such node had 4 CPU cores and 168GB of RAM. In the latter, we used nodes equipped with Intel Xeon E5-2697 2.60GHz CPU with 28 cores and 128GB RAM.

Each transformer model was trained on a single GPU node for 3 days. Sokoban models were trained on CPU nodes (due to the small size of the models).

K Out-of-distribution analysis

In Rubik's Cube, we trained the generators on very short trajectories, obtained by applying only 10 random moves to the ordered cube (that covers about $7 \cdot 10^9$ configurations, compared to $4 \cdot 10^{19}$ in total). Even with such limited data, AdaSubS succeeded in solving 99% of the fully scrambled cubes we tested.

In Sokoban, we check that AdaSubS trained on boards with 4 boxes can tackle instances with 5, 6, and 7 boxes. Specifically, it solves 87%, 82%, and 74% of cases, respectively, while kSubS only solves 74%, 67%, and 56%, respectively. The results are presented in Figure 5. Note that exactly like in the case of INT, AdaSubS consistently has half the failure rate of that of kSubS on OOD instances.

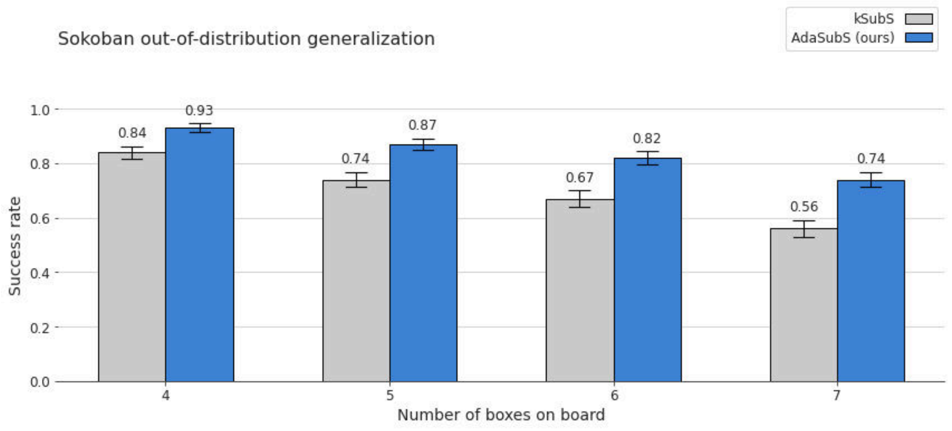


Figure 5: Out-of-distribution performance of AdaSubS and kSubS for Sokoban boards with more boxes, with budget of 5000 nodes. Both methods were trained on instances with 4 boxes. Error bars correspond to 95% confidence intervals.

Robotic Control via Embodied Chain-of-Thought Reasoning

Michał Zawalski[*] University of Warsaw, U.C. Berkeley
William Chen[*] U.C. Berkeley
Karl Pertsch U.C. Berkeley, Stanford University
Oier Mees U.C. Berkeley
Chelsea Finn Stanford University
Sergey Levine U.C. Berkeley

Abstract: A key limitation of learned robot control policies is their inability to generalize outside their training data. Recent works on vision-language-action models (VLAs) have shown that the use of large, internet pre-trained vision-language models as the backbone of learned robot policies can substantially improve their robustness and generalization ability. Yet, one of the most exciting capabilities of large vision-language models in other domains is their ability to reason iteratively through complex problems. Can that same capability be brought into robotics to allow policies to improve performance by reasoning about a given task before acting? Naive use of "chain-of-thought" (CoT) style prompting is significantly less effective with standard VLAs because of the relatively simple training examples that are available to them. Additionally, the purely-semantic reasoning about sub-tasks common to regular CoT is insufficient for robot policies that need to ground their reasoning in sensory observations and the robot state. To this end, we introduce Embodied Chain-of-Thought Reasoning (ECoT) for VLAs, in which we train VLAs to perform multiple steps of reasoning about plans, sub-tasks, motions, and visually grounded features like object bounding boxes and end effector positions, before predicting the robot action. We design a scalable pipeline for generating synthetic training data for ECoT on large robot datasets. We demonstrate that ECoT increases the absolute success rate of OpenVLA, the current strongest open-source VLA policy, by 28% across challenging generalization tasks without any additional robot training data. Additionally, ECoT makes it easier for humans to interpret a policy's failures and correct its behavior interactively using natural language. Finally, we show that our model learns to transfer ECoT reasonings to unseen embodiments and tasks.

Keywords: Vision-Language-Action Models, Embodied Chain-of-Thought Reasoning

1 Introduction

Robotic policies trained end-to-end for outputting raw low-level actions in response to sensory observations provide a powerful and appealing learning-based approach to robotic control, obviating the need for complex sensing and control stacks, and processing complex observations into dexterous low-level controls [1, 2, 3]. However, this kind of "reactive" control is not necessarily the best choice in settings that demand broad generalization, such as novel scenes or interactions with unfamiliar objects. In such situations, a person might need to think more carefully – if they were asked to put fruit on one plate and vegetables on another, they might first try to figure out which objects are fruits and which are vegetables, rather than simply performing a learned skill from "muscle memory." In the same way, we would like our robotic policies to both perform well-practiced end-to-end control, and to "reason through" novel situations before grounding their commands into actions. Such reasoning

[*]Equal contribution, correspondence at `m.zawalski@uw.edu.pl` and `verityw@berkeley.edu`.

might include identifying and locating task relevant objects, producing a plan to accomplish a task, and translating sub-tasks and observations into movements.

Vision-language-action models (VLAs) – pre-trained vision-language models (VLMs) fine-tuned to produce robot actions – have gained popularity as an approach for leveraging the diversity of Internet data captured within large foundation models [4] in a simple and scalable policy learning recipe. Despite achieving state-of-the-art performance across a wide range of tasks and robot embodiments [5, 6, 7], VLAs typically learn a direct mapping from observations to actions without any intermediate reasoning. However, there have been many recent works exploring how language models (which serve as the backbone of VLAs) can be prompted to textually "think step-by-step" about a given task. Such chain-of-thought reasoning (CoT) [8] significantly improves their performance on complex reasoning tasks and is now de-facto a standard practice in language modeling [9].

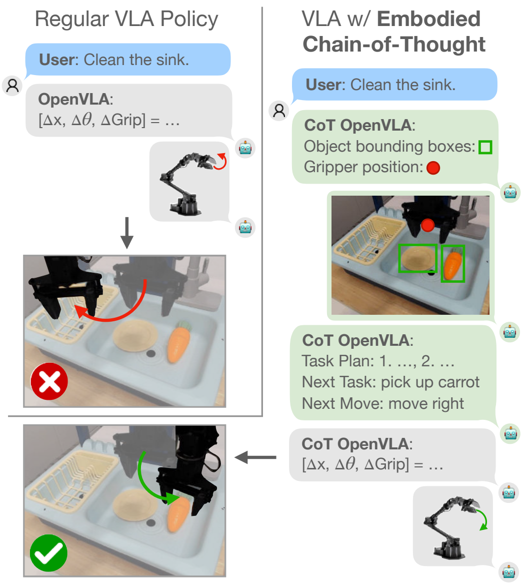


Figure 1: We propose embodied chain-of-thought reasoning for vision-language-action models (VLAs): prior VLAs directly predict the next robot action given the task (**left**), we instead train VLA policies to think "step-by-step" (**right**). Crucially, reasoning through low-level visual and embodied features like object bounding boxes and gripper positions in addition to purely textual CoT elements like sub-task plans, forces the policy to "think carefully" *and* "look carefully" before acting. Embodied CoT reasoning increases the absolute success rate of state-of-the-art OpenVLA policies [7] by 28% in challenging generalization tasks.

We thus hypothesize that we can similarly boost VLA performance by training them to *textually reason* about their plan, environment, and motions, thereby allowing them to produce more accurate and robust robot actions. However, simply applying the CoT techniques from language modeling to the robotics domain faces several challenges. For one, current VLAs build on relatively small, open-source VLMs that cannot match closed models in their ability to perform meaningful reasoning when simply prompted to think step-by-step [8]. Additionally, the most common CoT reasoning in language models, breaking tasks into sub-tasks, albeit helpful, is insufficient for reasoning about robotic tasks. The VLA policy needs to ground its plans and reasoning in its *observations* of the environment and robot state. Only then can the reasoning direct the agent's attention toward fine-grained spatial or semantic perceptual features that are key for solving robot manipulation tasks. Put simply, we need VLAs to not only "think carefully", but also "look carefully."

To this end, we introduce **Embodied Chain-of-Thought Reasoning (ECoT)** for VLA policies. In contrast to prior VLAs, embodied chain-of-thought policies perform multiple steps of textual reasoning before predicting the next robot action (see Fig. 1, right). In contrast to existing CoT reasoning approaches for language models, they interleave semantic-level reasoning about sub-tasks with "embodied" reasoning tasks that require the policy to pay attention to its multi-modal inputs, from predicting bounding boxes of objects in the scene to reasoning about low-level movement primitives that need to be executed based on the current robot state. To enable the relatively weak LLM backbones of open-source VLAs to perform such reasoning effectively, we design a scalable pipeline for synthetically generating embodied CoT training data for large robot datasets. Concretely, we use powerful pre-trained open-vocabulary object detectors and large language models to generate the reasoning supervision for our policies.

Our experiments show that by training state-of-the-art VLAs to perform multiple steps of reasoning before action prediction, we can substantially boost their ability to perform challenging generalization tasks. Our approach increases the absolute success rate of OpenVLA [7], the current best-performing open-source VLA policy, by 28% across a suite of robot manipulation tasks that involve generalization to new objects, scenes, viewpoints, and instructions without any additional robot training data. Beyond raw performance improvements, our experiments show that training VLAs with embodied CoT makes policy failures more interpretable and allows humans to easily correct policy behavior by modifying faulty reasoning chains via natural language feedback.

2 Related Work

Scaling robot learning. A long-standing goal of robot learning is to train policies that can generalize to a wide range of unstructured real-world environments. Towards this goal, recent works have explored training "generalist robot policies" [10, 11, 12, 13, 14, 15, 16, 17] on diverse robot datasets [18, 19, 10, 20, 21, 22, 23, 13, 14, 24, 15, 25, 6]. As a result of their diverse robot training datasets, many of these policies can be prompted in natural language to solve various manipulation tasks, and some generalist policies can even control multiple robot embodiments [16, 26, 6]. Importantly, these works demonstrate that training robot policies on large and diverse datasets is a promising approach towards improving policy robustness and generalization ability.

Vision-language models for robot generalization. In a push towards generalization far beyond what is observed in robot datasets, the recent development of strong, open-source vision-language models that learn visuo-linguistic representations [27, 28], generate images from text [29], or generate text in response to images and prompts [30, 31, 32, 33, 34] have resulted in a large number of works that explore the integration of such models into robot learning pipelines, e.g., to generate goals [35], to provide reward signals [36, 37, 38], or to learn visual state representations [39, 40, 41]. Since collection of the aforementioned large-scale robot datasets is challenging, using models pre-trained on Internet-scale data is an appealing alternate path towards robust robot policies that can act in a variety of unstructured real-world environments. Most relevant to our work are recent approaches for integrating pre-trained vision-language models into learned robot policies. While some works use strong structural priors in their policies to enable this integration [42, 43, 44], vision-language-action models (VLAs) have recently been proposed as a simple yet scalable alternative [5, 6, 7], achieving state-of-the-art performance for generalist robot policies [7] and showing impressive levels of generalization to new objects and scenes. However, existing VLAs do not sufficiently leverage some of the most appealing properties of the underlying language and vision-language models, specifically their ability to *reason* through the steps required to solve a given task.

Reasoning for language and control. Such step-by-step reasoning is a key ingredient for the ability of large language models (LLMs) to solve a wide range of complex tasks. Prompting LLMs (directly [45] or with in-context examples [8]) to "think step-by-step" about the problem before formulating an answer can significantly improve their performance, with such chain-of-thought reasoning techniques becoming standard practice in language modeling and (vision-)language model training [9, 46]. A number of works have explored similar techniques in the context of high-level task planning for robotics [47, 48, 49, 50, 51, 52, 53]. These approaches use pre-trained or fine-tuned LLMs to decompose tasks into high-level sub-tasks, but rely on pre-trained low-level policies to execute them. However, we argue that (1) careful reasoning can be beneficial for both high-level sub-task reasoning *and* during low-level control and (2) all such levels of reasoning should be strongly grounded in visual observations of the scene and the agent's state. Thus, in contrast to these prior works and language-only CoT, our approach trains a VLA policy to autoregressively generate CoTs (for high- and low-level reasoning) and actions given input instructions and observations, ensuring that both are firmly grounded in the agent's environment. We empirically confirm that such a formulation is critical to effectively leveraging (V)LM reasoning capabilities for control.

3 Preliminaries: Vision-Language-Action Models

Our work leverages VLAs as the backbone for our embodied chain-of-thought policies. VLAs use a simple policy learning recipe: starting from a pre-trained vision-language model, they directly finetune the model to autoregressively predict the next robot

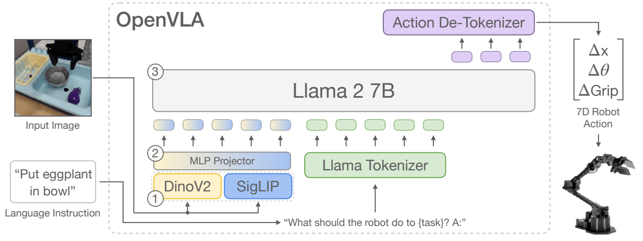


Figure 2: The OpenVLA model. Reproduced with permission from Kim et al. [7].

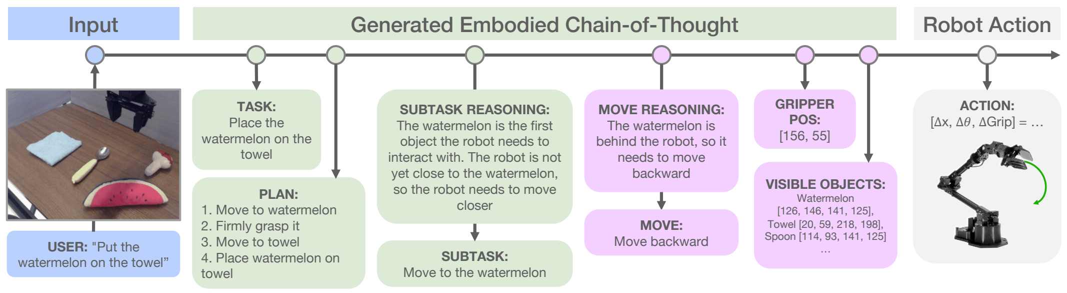

Input
Generated Embodied Chain-of-Thought
Robot Action
USER: "Put the watermelon on the towel"
TASK: Place the watermelon on the towel
PLAN: 1. Move to watermelon 2. Firmly grasp it 3. Move to towel 4. Place watermelon on towel
SUBTASK REASONING: The watermelon is the first object the robot needs to interact with. The robot is not yet close to the watermelon, so the robot needs to move closer
SUBTASK: Move to the watermelon
MOVE REASONING: The watermelon is behind the robot, so it needs to move backward
MOVE: Move backward
GRIPPER POS: [156, 55]
VISIBLE OBJECTS: Watermelon [126, 146, 141, 125], Towel [20, 59, 218, 198], Spoon [114, 93, 141, 125] ...
ACTION: [Δx, Δθ, ΔGrip] = ...

Figure 3: Steps of our embodied chain-of-thought reasoning. We interleave several intermediate reasoning steps into the mapping from inputs to robot actions. **Green**: "standard" linguistic chain-of-thought steps that break a given instruction into the required sub-tasks. **Purple**: *Embodied* chain-of-thought steps that require grounding the policy's reasoning in the scene and robot state. Our experiments show that these grounded reasoning steps are key to improving policy performance with chain-of-thought reasoning.

action a given the current image observation I and task instruction T. To enable this, the continuous robot actions are typically converted to discrete action tokens $\mathcal{T}_a$ in the vocabulary of the vision-language model via a per-dimension action discretization scheme that assigns each continuous value to one of 256 bins [5, 7].

In this work, we use the recently released OpenVLA model [7] (see Fig. 2), since it achieves state-of-the-art performance and is fully open-source. The model builds on the Prismatic VLM [34] and consists of a fused visual encoder that combines pre-trained SigLIP [54] and/or DinoV2 [55] features and a Llama 2 7B [56] LLM backbone. During training, input images are encoded into visual token embeddings using the pre-trained vision encoders, the task instruction is mapped to task tokens using Llama 2's text tokenizer, and the model is trained to map these inputs to the target action tokens. Next, we will discuss how we can improve upon this conventional VLA training recipe by enabling the VLA to reason through the task at hand before deciding which action to take.

4 Embodied Chain-of-Thought Reasoning for Visuomotor Policies

In this section, we discuss our approach for training VLAs to perform embodied chain-of-thought reasoning about plans, sub-tasks, motions, and visual features before predicting the next robot action (see Fig. 1). Unlike many proprietary large language models, the relatively small LLM backbones used in current VLAs struggle to perform involved reasoning when simply prompted to think step-by-step [8]. Instead, we propose to explicitly train VLA models to perform embodied CoT reasoning. Concretely, we label data from existing robot datasets post-hoc with reasoning chains filled with features extracted from various pre-trained models and use the resulting dataset of observation-reasoning-action tuples for training. In practice, we ensure that all elements of the generated reasoning data can be represented as strings, such that we can use the Llama 2 text tokenizer to translate them into reasoning tokens. Then, we can simply train the VLA to autoregressively predict these tokens, directly followed by action tokens.

While this approach is conceptually simple, its implementation requires answering multiple key questions: (1) Which reasoning steps are suitable for guiding policies in solving embodied robot manipulation tasks (Fig. 3)? (2) How can we generate training data for these reasoning steps at scale on existing robot datasets (Section 4.2)? Another practical consideration arises *after* training, while using ECoT policies for robot control: carefully reasoning through each action can significantly slow down policy inference. We discuss solutions to these problems in the following sections.

4.1 Designing Embodied Chain-of-Thought Reasoning Steps

Our goals when designing the steps of our embodied chain-of-thought reasoning chains are twofold: encourage the model to (A) reason through the required high-level steps of the task at hand and determine which step needs to be executed next, and (B) increasingly ground this reasoning in lower-level features of the scene and robot state before predicting the robot action.

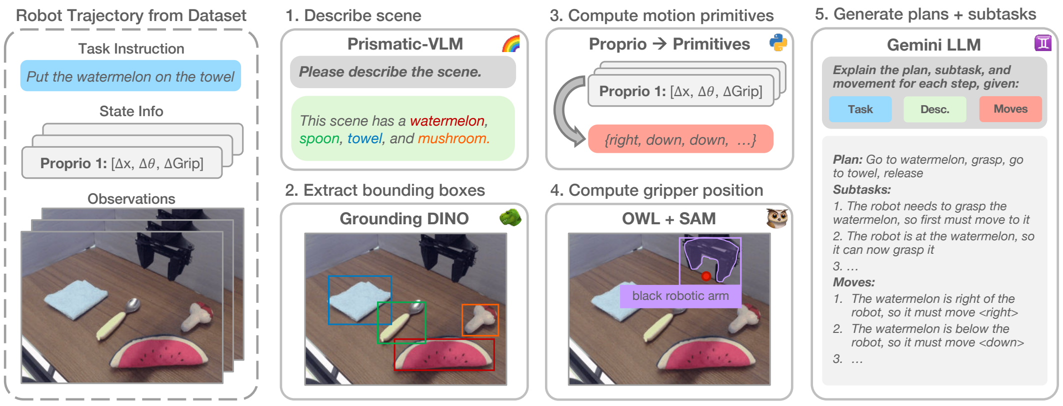

Robot Trajectory from Dataset
Task Instruction
Put the watermelon on the towel
State Info
Proprio 1: [Δx, Δθ, ΔGrip]
Observations
1. Describe scene
Prismatic-VLM
Please describe the scene.
This scene has a watermelon, spoon, towel, and mushroom.
2. Extract bounding boxes
Grounding DINO
3. Compute motion primitives
Proprio → Primitives
Proprio 1: [Δx, Δθ, ΔGrip]
{right, down, down, ...}
4. Compute gripper position
OWL + SAM
black robotic arm
5. Generate plans + subtasks
Gemini LLM
Explain the plan, subtask, and movement for each step, given:
Task
Desc.
Moves
Plan: Go to watermelon, grasp, go to towel, release
Subtasks:
1. The robot needs to grasp the watermelon, so first must move to it
2. The robot is at the watermelon, so it can now grasp it
3. ...
Moves:
1. The watermelon is right of the robot, so it must move <right>
2. The watermelon is below the robot, so it must move <down>
3. ...

Figure 4: Our pipeline for generating synthetic embodied chain-of-thought data at scale for a given robot dataset. We use a Prismatic VLM [34] to create a scene description **(1)**, and Grounding Dino [28] to detect bounding boxes for all objects **(2)**. We then compute templated motion primitives from the low-level robot states **(3)** and the robot gripper position using OWLv2 [57] and SAM [58] **(4)**. Finally, all information is passed to a large Gemini language model [59] to create the synthetic reasoning chain **(5)**.

We visualize the ECoT reasoning steps that we train the VLA to perform for an example task in Fig. 3. From left to right, the model is trained to first rephrase the task instruction (**TASK**) and predict a high-level plan of steps for achieving the instructed task (**PLAN**). Next, it reasons through which of the sub-tasks should be executed at the present step (**SUBTASK**), a task which requires understanding the current state of the scene and robot. Then, the model predicts an even lower-level language command like "move left" or "move up" (**MOVE**) that is closely related to the low-level actions the robot needs to execute. Finally, we ask the model to predict precise, spatially grounded features that describe the scene and thus force the model to pay close attention to all elements of the input image– specifically, the pixel position of the robot end effector (**GRIPPER**) and the names and bounding box pixel coordinates of all objects in the scene (**OBJECTS**).

While we believe that our choice of reasoning tasks and their order is well-aligned with our intuition for a sensible step-by-step solution to the task, we by no means exhaustively explored all possible reasoning tasks. Testing alternative tasks and task orderings, and finding ways to automatically determine sensible reasoning chains are important directions for future work.

4.2 Generating Embodied Chain-of-Thought Data at Scale

The gold standard for obtaining high-quality reasoning chains are direct human annotations. However, this approach is impractical for large robot learning datasets [6], which consist of millions of individual transitions. Thus, we instead propose to leverage pre-trained vision and/or language foundation models to automatically generate ECoT training data, akin to synthetic data generation in NLP [60].

We provide an overview of our data generation pipeline in Fig. 4. For a given image-instruction pair, we first prompt a Prismatic-7B VLM [34] to generate a detailed description of the scene. We then concatenate the original instruction and this generated description and input the resulting string into Grounding DINO [28], an open-vocabulary object detector. It detects all relevant object instances and their bounding boxes and associates them with the corresponding language snippets from the input text. We filter the predictions based on the provided confidence score, only keeping detections with a box- and text-confidence larger than 0.3 and 0.2 respectively to use for the **OBJECT** features. See Appendix A for example detections.

Next, we generate the per-step low-level action primitives in **MOVE** (e.g., "move left", "move up"). Following Belkhale et al. [61], we use the robot proprioception to determine the movement direction for the next 4 time steps (assuming a fixed camera), and translate this into one of 729 templated movement primitives (see Appendix B for a list of all primitives). We use OWLv2 [57] and SAM [58] to detect 2D end effector positions in the training images (**GRIPPER**) paired with 3D positions extracted from the robot state to fit a robust estimate of the projection matrix using RANSAC [62]. We then use the 2D projections of the robot end-effector position for our training. This process is repeated for each trajectory independently, eliminating the need to assume fixed camera parameters.

To generate the final reasoning chain, we feed each episode's task instruction, scene description, and per-step movement primitives into Gemini 1.0 [59] and prompt it to produce both a high-level plan of sub-tasks in accordance with the task instruction and observed movement primitives and the current sub-task for each step. We also ask it to briefly explain the primitive movement and chosen sub-task in each step, which we include in the ECoT training data. We run our data generation pipeline on the complete Bridge v2 dataset [13], with more than 2.5M transitions, over the course of 7 days.

4.3 Efficient Chain-of-Thought Inference for Robot Policies

Inference speed is a key challenge for ECoT policies. The additional reasoning tokens the model needs to predict can significantly reduce the achievable control frequency by increasing the number of tokens to be predicted per timestep from 7 for OpenVLA to 350 for ECoT. We explore a simple solution for speeding up inference: we keep parts of the reasoning chain like the high-level plan or the current sub-task fixed for multiple steps. Crucially, *encoding* previously predicted tokens is much faster for Transformer-based policies like OpenVLA than *generating* it. We compare two such strategies: (1) *synchronous* execution, where we predict the high-level reasoning every N steps, and (2) *asynchronous* execution, in which one ECoT policy instance continually updates the high-level reasoning chains, while a second policy instance uses the most recent reasoning chain to predict low-level reasoning steps and robot actions. We report the trade-off between performance and inference speed for all approaches in Section 5.5. Note that these runtime improvements are orthogonal to widely used approaches for improving throughput of large language and vision-language models, like optimized computation kernels [63] and speculative decoding [64], which we leave for future works.

5 Experiments

In this section, we investigate the effectiveness of ECoT for robot control across a range of challenging manipulation tasks. We answer the following questions: (1) Does embodied chain-of-thought reasoning improve the performance of VLA policies (Section 5.2)? (2) Does embodied chain-of-thought reasoning make it easier to interpret and correct policy failures (Sections 5.3 and 5.4)? (3) How can we optimize the runtime efficiency of policies with embodied CoT reasoning (Section 5.5)?

5.1 Experimental Setup

Robot setup and training data. We perform evaluations with the 6-DoF WidowX robot arm from the Bridge V2 paper [13], a commonly used setup for evaluating generalizable robot policies [16, 7]. Given a single 3rd person camera and natural language instruction, the policy predicts end-effector velocity actions to control the robot. Walke et al. [13] provide a large and diverse teleoperated dataset of 60k demonstrations. We apply our pipeline for synthetic generation of chain-of-thought data (Section 4.2) on this dataset to obtain our training dataset.

Evaluation tasks. We design a suite of challenging evaluation tasks that focus on testing the policies' generalization ability along multiple axes: processing spatial relations, interacting with unseen objects, and performing unseen instructions. All policies are evaluated on the same real-world setups to control for camera angle, lighting, and background. We perform 314 total trials per approach.

Comparisons. We compare our policy (**ECoT**) to state-of-the-art VLA policies, namely **OpenVLA** [7], the same model our approach is built upon, but trained *without* chain-of-thought reasoning, and **RT-2-X** [6], a 55B parameter closed VLA policy. To ensure fair comparison, we train the OpenVLA policy on the same dataset we use for training our approach, the Bridge V2 data [13]. For RT-2-X we cannot control the data distribution in the same way since the model is closed. The RT-2-X policy we compare to was trained on Bridge V2 data *and additional datasets from the Open X-Embodiment dataset* [6]. Thus, it has access to *more* training data than our approach. We also compare against **Octo** [16], which is also trained on that dataset, but was not fine-tuned from a VLM (i.e., it is not a VLA). Finally, we compare to **Naïve CoT**, a version of our model that only uses *non-embodied* CoT reasoning about sub-tasks akin to conventional CoT reasoning in language modeling (see Fig. 3). This comparison will test the importance of using *embodied* reasoning for VLA policies.

5.2 Embodied Chain-of-Thought Reasoning Improves Policy Generalization

We report performance of all approaches on our evaluation set in Table 1. We see that while OpenVLA achieves high performance on in-distribution tasks, it struggles on the hard generalization cases we

Table 1: **Comparison of success rates for OpenVLA, RT-2-X, and ECoT** across two scenes (one with in-distribution camera view and one with out-of-distribution). Mean ± one StdErr. On aggregate, our ECoT policy achieves the highest success rate, improving absolute success rate by 45%, 22%, 19%, and 18% over Octo, OpenVLA, RT-2-X, and naïve CoT respectively in the in-distribution view setting and 48%, 34%, 16%, and 16% in the out-of-distribution view setting.

Type	**Task**	**Algorithm (ID View)**					**Algorithm (OOD View)**				
		Octo	**OpenVLA**	**RT-2-X**	**Naive CoT**	**ECoT (Ours)**	**Octo**	**OpenVLA**	**RT-2-X**	**Naive CoT**	**ECoT (Ours)**
ID	Put mushroom in pot	29%	88%	94%	71%	**100%**	35%	59%	**76%**	**76%**	65%
	Put spoon on towel	60%	**90%**	80%	60%	80%	20%	**80%**	**80%**	60%	**80%**
	Put carrot on plate	70%	80%	90%	90%	**100%**	40%	90%	90%	**100%**	90%
	Wipe [plate / pan] with towel	13%	**50%**	38%	38%	**50%**	0%	50%	0%	13%	**63%**
Spatial Relations	Put mushroom in [left / right / middle] container	0%	22%	17%	22%	**33%**	0%	17%	22%	55%	**67%**
	Put purple object in [left / right / middle] container	0%	28%	17%	50%	**56%**	0%	22%	11%	**55%**	39%
	Put [right / left] object on middle object	0%	13%	0%	50%	**63%**	0%	25%	25%	50%	**63%**
OOD Objects	Pick up [screwdriver / hammer / measuring tape / detergent / watermelon]	30%	20%	**80%**	50%	50%	30%	20%	**80%**	50%	50%
	Move mushroom to [measuring tape / detergent]	0%	10%	70%	20%	**100%**	10%	0%	**90%**	40%	**90%**
	Put mushroom in tall cup	0%	**80%**	0%	70%	30%	10%	20%	0%	20%	**30%**
	Place watermelon on towel	20%	30%	60%	60%	**70%**	50%	10%	**90%**	30%	40%
OOD Instructions	Pick up any object that is not [yellow / a duck / a sponge / a towel]	50%	33%	**58%**	50%	42%	17%	17%	**67%**	25%	**67%**
	Put the edible object in the bowl	13%	25%	13%	25%	**88%**	0%	13%	25%	25%	**100%**
	Put the object used for [eating / drinking] on towel	25%	38%	38%	38%	**75%**	13%	0%	25%	38%	**75%**
Aggregate		21% ± 3.3%	44% ± 3.9%	47 ± 4.0%	48 ± 4.0%	**66% ± 3.8%**	16% ± 2.9%	30% ± 3.6%	48 ± 4.0%	48% ± 4.0%	**64 ± 3.9%**

test. RT-2-X performs better than vanilla OpenVLA, potentially due to the larger robot pre-training dataset (note again that OpenVLA and our approach are *only* trained on the Bridge dataset) and the fact that it *co-trains* the policy with Internet-scale vision-language data *and* robot data, while all other approaches only use robot data during fine-tuning.

Importantly, we find that our ECoT policy substantially outperforms the OpenVLA policy across all generalization evaluations. This is notable, since both policies are based on the exact same VLM base model and use the same robot data for fine-tuning. The only difference is in the use of CoT reasoning by our approach. Curiously, our ECoT model even surpasses the performance of RT-2-X in the tested tasks, even though RT-2-X is trained on 10 additional robot datasets and uses a networks that is 7x larger (55B vs. 7B). Finally, the results in Table 1 show that including *embodied* reasoning about visual inputs and the low-level robot state significantly boosts performance over the "Naïve CoT" ablation of our approach, which only reasons about high-level linguistic features like sub-task plans.

We visualize qualitative examples of our model's reasoning in Fig. 5. The left two examples show that the model successfully breaks down the task into a sequence of sub-tasks and then crucially *grounds* those sub-tasks in the scene by predicting the relevant bounding boxes and gripper position of the robot, before deciding on the next move and concrete low-level robot action. We visualize more chain-of-thought examples from our evaluation tasks in Appendix, Fig. 8.

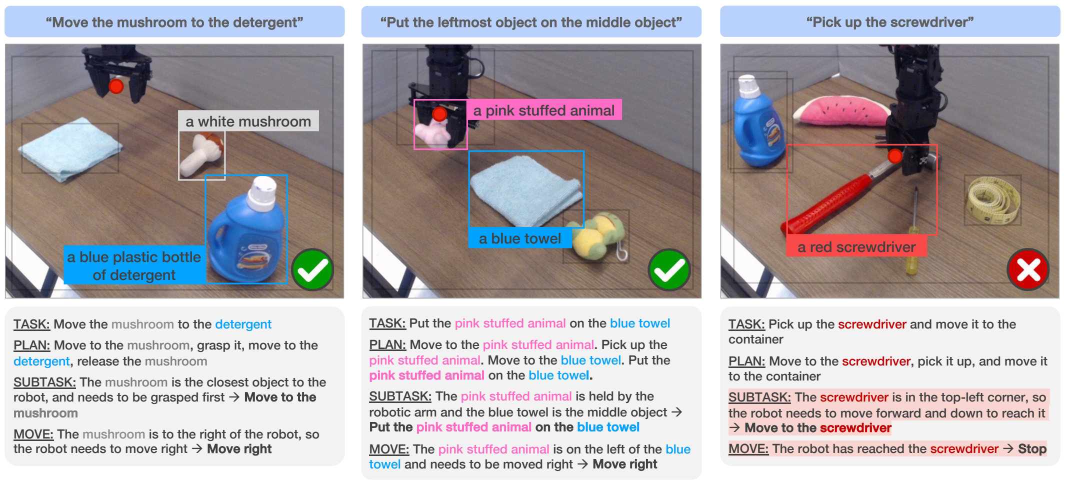
5.3 Diagnosing Policy Failures Through Inspecting Reasoning Chains

In the previous section we showed that training VLA policies to reason through a given task step-by-step can significantly improve their performance on challenging generalization tasks. In addition to improving performance, such chain-of-thought reasoning provides a tool for users and researchers to better understand the decisions the policy takes. By inspecting and visualizing the model's reasoning steps, we can discover potential mistakes in the reasoning chain that led to policy failure downstream.

We give an example of this in Fig. 5 (right): the ECoT policy failed to solve the task *pick up the screwdriver*. Inspecting the reasoning chain, we can see that the hammer is incorrectly identified as a screwdriver, causing the robot to reach for that instead. It should be noted that such inspection of reasoning chains is not a "bullet-proof" approach for interpreting the failures of an end-to-end trained policy: the model *could* predict a particular plan and then still deviate from it when choosing the final action. However, in practice we find that reasoning chains often correlate strongly with the executed actions. We provide more examples for diagnosing policy failures via its reasoning chains in Fig. 8.

5.4 Chain-of-Thought Reasoning Enables Interactive Policy Correction

Intuitively, training a policy to reason through a task step-by-step in natural language provides a powerful mechanism for humans to interact with the policy and *correct* its behavior. Instead of needing involved teleoperation equipment to provide direct robot action feedback like in DAgger approaches [65], humans can now simply correct the policy's behavior by modifying its reasoning


“Move the mushroom to the detergent”
a white mushroom
a blue plastic bottle of detergent
“Put the leftmost object on the middle object”
a pink stuffed animal
a blue towel
“Pick up the screwdriver”
a red screwdriver

TASK: Move the mushroom to the detergent

PLAN: Move to the mushroom, grasp it, move to the detergent, release the mushroom

SUBTASK: The mushroom is the closest object to the robot, and needs to be grasped first → **Move to the mushroom**

MOVE: The mushroom is to the right of the robot, so the robot needs to move right → **Move right**

TASK: Put the pink stuffed animal on the blue towel

PLAN: Move to the pink stuffed animal. Pick up the pink stuffed animal. Move to the blue towel. Put the **pink stuffed animal on the blue towel.**

SUBTASK: The pink stuffed animal is held by the robotic arm and the blue towel is the middle object → **Put the pink stuffed animal on the blue towel**

MOVE: The pink stuffed animal is on the left of the blue towel and needs to be moved right → **Move right**

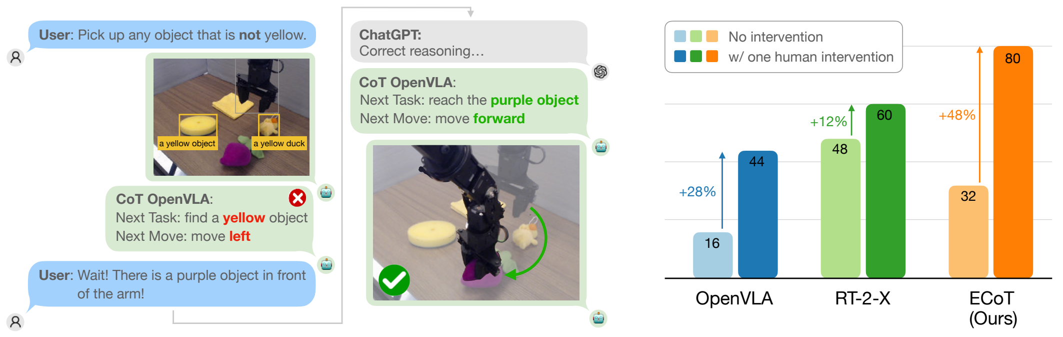
TASK: Pick up the screwdriver and move it to the container

PLAN: Move to the screwdriver, pick it up, and move it to the container

SUBTASK: The screwdriver is in the top-left corner, so the robot needs to move forward and down to reach it → **Move to the screwdriver**

MOVE: The robot has reached the screwdriver → **Stop**

Figure 5: **Qualitative ECoT predictions from our model for two successful trajectories (left, middle) and one failure (right).** Irrelevant bounding boxes are greyed out for readability. Left: high-level reasoning and low-level object segmentations are correct, leading to a successful rollout. Middle: the command is correctly rephrased to refer to specific objects (i.e., “the leftmost object” is identified as the pink toy). Right: the hammer is incorrectly identified as a screwdriver, causing the robot to take inappropriate actions.



User: Pick up any object that is **not** yellow.

CoT OpenVLA:
Next Task: find a **yellow** object
Next Move: move **left**

User: Wait! There is a purple object in front of the arm!

ChatGPT:
Correct reasoning…

CoT OpenVLA:
Next Task: reach the **purple object**
Next Move: move **forward**

Figure 6: Embodied chain-of-thought training enables interactive human policy correction in natural language. **Left**: given a human intervention in natural language, we use ChatGPT to correct our model’s reasoning chains. **Right**: our embodied chain-of-thought policy can benefit from a human language intervention most, increasing success rate by 48% on our most challenging evaluation tasks.

chains via natural language feedback. Prior work introduced carefully designed policy architectures and explicitly trained policies to support such correction via language [66, 67]. Here, we test whether similar capabilities emerge naturally by training VLA policies to perform chain-of-thought reasoning.

To test whether chain-of-thought policies enable human correction purely via language feedback, we rerun evaluations of our ECoT policy on the most challenging tasks from Table 1 (put mushroom in cup, pick up out-of-distribution object, and pick up non-yellow object), in which our policy *without human intervention* achieved an average success rate of only 32%. We visualize our approach for human intervention in Fig. 6: we allow a human operator to interrupt policy execution *once* over the course of the episode and provide natural language feedback (e.g. “no, the screwdriver is in the back right corner”, “release the mushroom now!”, or “the cup is tall”). Then, we use ChatGPT to adapt our model’s reasoning chain based on the language feedback, prompting it to produce a corrected reasoning chain (see Fig. 11 for the exact prompt used). Finally, we feed this corrected chain back into our policy and continue execution, holding the corrected reasoning chain fixed for 5 steps.

The results in Fig. 6 (right) show that our ECoT policy can make effective use of the human language intervention, increasing its success rate by 48% on our most challenging evaluation tasks. In contrast, we evaluate the vanilla, non-CoT OpenVLA policy and RT-2-X in the same way, providing each with a single human language correction per rollout, but find that neither of them can benefit from the human intervention to the same degree (for both we also use ChatGPT to incorporate the intervention into the original task instruction to allow for fair comparison).

5.5 Efficient Chain-of-Thought Inference

We compare the performance of approaches for accelerating CoT policy inference (see Section 4.3) to naïvely running CoT inference at every step of execution in Table 2. We also report the speed-up that is achieved by both proposed approaches from Section 4.3 compared to naïve execution. Both approaches achieve inference speed improvements while at least matching performance, with asynchronous execution achieving the largest speed-up at the cost of doubling the compute required at inference time (since two policy instances are running in parallel). We use the 5-step freeze approach for the main results presented in Table 1, since it provides the best performance-speed tradeoff. We used a small task subset (put mushroom in pot, move mushroom to detergent or measuring tape, and put the left/right object on the middle).

Table 2: Performance of accelerated CoT inference approaches and their relative speed-ups over the naïve approach, averaged across 3 tasks (25 trials total).

	Success	Speed-Up
Naïve	63%	–
5-Step	72%	+24%
Async	65%	+40%

5.6 Exploring Alternative Embodied Chain-of-Thought Design Choices

We explore some alternative design choices for ECoT, changing the reasoning step orderings, reasoning features, and training data mixes. We first train a model that (1) predicts both the current and next 4 gripper positions and (2) predicts the bounding boxes right after generating the plan. Notably, the latter means that the **bounding boxes are frozen** (due to the 5-step freeze approach detailed in Section 5.5) – while this means that moving the objects may make the frozen bounding boxes out of date, it also speeds up inference significantly (since the bounding box reasoning step is generated at every step for our base policy and often consists of many tokens). Next, we repeat this experiment, but also **co-train** on the base Prismatic VLM's vision-language pre-training data (where each batch is 75% ECoT data and 25% vision-language data), as this may enable better retention of vision-language modeling capabilities and knowledge. Finally, we take an OpenVLA model pre-trained on the full OXE dataset and **fine-tune** it on the the same dataset, but replace the Bridge V2 data with our ECoT equivalent. This allows us to see (1) if fine-tuning on a more diverse set of demonstrations will aid with more general reasoning and (2) if reasoning capabilities can emerge on other embodiments (despite only the Bridge data being labeled with reasoning chains).

For evaluation, we consider a large subset of the tasks in Table 1 in the in-distribution view setting only, totaling 106 trials per model. All results are presented in Table 3. While our base policy still performs the best, all variants still outperform all non-reasoning baseline generalist policies. We also find that the OXE fine-tuned model can generate plausible reasonings for observations from other robot embodiments, despite not having been trained on any reasoning data other than the Bridge V2 WidowX synthetic reasoning chains. We present more detailed qualitative results on these ECoT variants in Appendix D.

Table 3: **Success rate of ECoT trained with various design choices**, as evaluated on a large subset of trials on the harder out-of-distribution view setting. While our base policy performs the best on aggregate (69%), the other approaches achieve higher performance on certain tasks. All policies in this table outperform OpenVLA, RT-2-X, and Octo's performances on the same trial subset (29%, 46%, and 14% aggregate success rates respectively).

Task	**Base ECoT**	**Frozen Bbox ECoT**	**Co-trained ECoT**	**Fine-tuned ECoT**
Put mushroom in pot	57%	86%	86%	86%
Put spoon on towel	80%	60%	40%	80%
Put carrot on plate	83%	67%	100%	100%
Wipe [plate / pan] with towel	75%	50%	25%	25%
Put mushroom in [left / right / middle] container	89%	55%	11%	44%
Put purple object in [left / right / middle] container	44%	67%	44%	67%
Put [right / left] object on middle object	63%	75%	75%	75%
Pick up [screwdriver / hammer / measuring tape / detergent / watermelon]	50%	60%	80%	70%
Move mushroom to [measuring tape / detergent]	90%	90%	100%	60%
Put mushroom in tall cup	20%	20%	20%	40%
Place watermelon on towel	60%	0%	20%	40%
Pick up any object that is not [yellow / a duck / a sponge / a towel]	67%	58%	42%	17%
Put the edible object in the bowl	100%	75%	50%	38%
Put the object used for [eating / drinking] on towel	75%	38%	50%	25%
Aggregate	69%	60%	56%	54%

6 Discussion and Limitations

In this work, we demonstrated that training VLA policies to perform chain-of-thought reasoning can substantially increase their performance without the need to collect additional robot training data. Instead of simply applying the CoT recipe from language modeling, our experiments underline the importance of adding reasoning steps that are strongly grounded in the scene and robot state, and involve for example object bounding boxes, the robot's end-effector, or low-level robot movements.

While our results are encouraging, our approach has several limitations. First, our model does not adapt the *structure* of its reasoning chains to the task at hand; it always performs *all* steps of reasoning in the fixed order we chose. A more effective strategy may involve executing only a subset of reasoning steps based on the robot and scene state, and future work can explore directly optimizing the model to pick the best reasoning steps. Additionally, scaling the ECoT training to a larger subset of the OXE dataset [6] will allow the resulting policies to be applicable to more downstream tasks. Last, the speed of execution for ECoT policies is still limiting: while our runtime optimizations in Section 4.3 improve the achievable control frequencies, exploring other approaches for improving LLM throughput [63] can unlock CoT reasoning for higher-frequency control tasks.

Acknowledgements

We would like to thank Sidd Karamcheti for his help with using the Prismatic VLM and OpenVLA codebases. We also thank Google DeepMind for providing Alpha API access to the RT-2-X model. This research was partly supported by ONR under N00014-20-1-2383, NSF IIS-2150826 and IIS-2246811 and the AI Institute. Michał Zawalski was funded by the Polish National Agency for Academic Exchange (PPN/STA/2021/1/00079/U/00001).

References

[1] A. Agarwal, A. Kumar, J. Malik, and D. Pathak. Legged locomotion in challenging terrains using egocentric vision, 2022.

[2] T. Z. Zhao, V. Kumar, S. Levine, and C. Finn. Learning fine-grained bimanual manipulation with low-cost hardware, 2023.

[3] Z. Fu, T. Z. Zhao, and C. Finn. Mobile aloha: Learning bimanual mobile manipulation with low-cost whole-body teleoperation. *arXiv preprint arXiv:2401.02117*, 2024.

[4] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill, E. Brynjolfsson, S. Buch, D. Card, R. Castellon, N. Chatterji, A. Chen, K. Creel, J. Q. Davis, D. Demszky, C. Donahue, M. Doumbouya, E. Durmus, S. Ermon, J. Etchemendy, K. Ethayarajh, L. Fei-Fei, C. Finn, T. Gale, L. Gillespie, K. Goel, N. Goodman, S. Grossman, N. Guha, T. Hashimoto, P. Henderson, J. Hewitt, D. E. Ho, J. Hong, K. Hsu, J. Huang, T. Icard, S. Jain, D. Jurafsky, P. Kalluri, S. Karamcheti, G. Keeling, F. Khani, O. Khattab, P. W. Koh, M. Krass, R. Krishna, R. Kuditipudi, A. Kumar, F. Ladhak, M. Lee, T. Lee, J. Leskovec, I. Levent, X. L. Li, X. Li, T. Ma, A. Malik, C. D. Manning, S. Mirchandani, E. Mitchell, Z. Munyikwa, S. Nair, A. Narayan, D. Narayanan, B. Newman, A. Nie, J. C. Niebles, H. Nilforoshan, J. Nyarko, G. Ogut, L. Orr, I. Papadimitriou, J. S. Park, C. Piech, E. Portelance, C. Potts, A. Raghunathan, R. Reich, H. Ren, F. Rong, Y. Roohani, C. Ruiz, J. Ryan, C. Ré, D. Sadigh, S. Sagawa, K. Santhanam, A. Shih, K. Srinivasan, A. Tamkin, R. Taori, A. W. Thomas, F. Tramèr, R. E. Wang, W. Wang, B. Wu, J. Wu, Y. Wu, S. M. Xie, M. Yasunaga, J. You, M. Zaharia, M. Zhang, T. Zhang, X. Zhang, Y. Zhang, L. Zheng, K. Zhou, and P. Liang. On the opportunities and risks of foundation models, 2022.

[5] A. Brohan, N. Brown, J. Carbajal, Y. Chebotar, X. Chen, K. Choromanski, T. Ding, D. Driess, A. Dubey, C. Finn, P. Florence, C. Fu, M. G. Arenas, K. Gopalakrishnan, K. Han, K. Hausman, A. Herzog, J. Hsu, B. Ichter, A. Irpan, N. Joshi, R. Julian, D. Kalashnikov, Y. Kuang, I. Leal, L. Lee, T.-W. E. Lee, S. Levine, Y. Lu, H. Michalewski, I. Mordatch, K. Pertsch, K. Rao, K. Reymann, M. Ryoo, G. Salazar, P. Sanketi, P. Sermanet, J. Singh, A. Singh, R. Soricut, H. Tran, V. Vanhoucke, Q. Vuong, A. Wahid, S. Welker, P. Wohlhart, J. Wu, F. Xia, T. Xiao, P. Xu, S. Xu, T. Yu, and B. Zitkovich. Rt-2: Vision-language-action models transfer web knowledge to robotic control, 2023.

[6] Embodiment Collaboration, A. O'Neill, A. Rehman, A. Maddukuri, A. Gupta, A. Padalkar, A. Lee, A. Pooley, A. Gupta, A. Mandlekar, A. Jain, A. Tung, A. Bewley, A. Herzog, A. Irpan, A. Khazatsky, A. Rai, A. Gupta, A. Wang, A. Kolobov, A. Singh, A. Garg, A. Kembhavi, A. Xie, A. Brohan, A. Raffin, A. Sharma, A. Yavary, A. Jain, A. Balakrishna, A. Wahid, B. Burgess-Limerick, B. Kim, B. Schölkopf, B. Wulfe, B. Ichter, C. Lu, C. Xu, C. Le, C. Finn, C. Wang, C. Xu, C. Chi, C. Huang, C. Chan, C. Agia, C. Pan, C. Fu, C. Devin, D. Xu, D. Morton, D. Driess, D. Chen, D. Pathak, D. Shah, D. Büchler, D. Jayaraman, D. Kalashnikov, D. Sadigh, E. Johns, E. Foster, F. Liu, F. Ceola, F. Xia, F. Zhao, F. V. Frujeri, F. Stulp, G. Zhou, G. S. Sukhatme, G. Salhotra, G. Yan, G. Feng, G. Schiavi, G. Berseth, G. Kahn, G. Wang, H. Su, H.-S. Fang, H. Shi, H. Bao, H. B. Amor, H. I. Christensen, H. Furuta, H. Walke, H. Fang, H. Ha, I. Mordatch, I. Radosavovic, I. Leal, J. Liang, J. Abou-Chakra, J. Kim, J. Drake, J. Peters, J. Schneider, J. Hsu, J. Bohg, J. Bingham, J. Wu, J. Gao, J. Hu, J. Wu, J. Wu, J. Sun, J. Luo, J. Gu, J. Tan, J. Oh, J. Wu, J. Lu, J. Yang, J. Malik, J. Silvério, J. Hejna, J. Booher, J. Tompson, J. Yang, J. Salvador, J. J. Lim, J. Han, K. Wang, K. Rao, K. Pertsch, K. Hausman, K. Go, K. Gopalakrishnan, K. Goldberg, K. Byrne, K. Oslund, K. Kawaharazuka, K. Black, K. Lin, K. Zhang, K. Ehsani, K. Lekkala, K. Ellis, K. Rana, K. Srinivasan, K. Fang, K. P. Singh, K.-H. Zeng, K. Hatch, K. Hsu, L. Itti, L. Y. Chen, L. Pinto, L. Fei-Fei, L. Tan, L. J. Fan, L. Ott, L. Lee, L. Weihs, M. Chen, M. Lepert, M. Memmel, M. Tomizuka, M. Itkina, M. G. Castro, M. Spero, M. Du, M. Ahn, M. C. Yip, M. Zhang, M. Ding, M. Heo, M. K. Srirama, M. Sharma, M. J. Kim, N. Kanazawa, N. Hansen, N. Heess, N. J. Joshi, N. Suenderhauf, N. Liu, N. D. Palo, N. M. M. Shafiullah, O. Mees, O. Kroemer, O. Bastani, P. R. Sanketi,

P. T. Miller, P. Yin, P. Wohlhart, P. Xu, P. D. Fagan, P. Mitrano, P. Sermanet, P. Abbeel, P. Sundaresan, Q. Chen, Q. Vuong, R. Rafailov, R. Tian, R. Doshi, R. Mart'in-Mart'in, R. Baijal, R. Scalise, R. Hendrix, R. Lin, R. Qian, R. Zhang, R. Mendonca, R. Shah, R. Hoque, R. Julian, S. Bustamante, S. Kirmani, S. Levine, S. Lin, S. Moore, S. Bahl, S. Dass, S. Sonawani, S. Song, S. Xu, S. Haldar, S. Karamcheti, S. Adebola, S. Guist, S. Nasiriany, S. Schaal, S. Welker, S. Tian, S. Ramamoorthy, S. Dasari, S. Belkhale, S. Park, S. Nair, S. Mirchandani, T. Osa, T. Gupta, T. Harada, T. Matsushima, T. Xiao, T. Kollar, T. Yu, T. Ding, T. Davchev, T. Z. Zhao, T. Armstrong, T. Darrell, T. Chung, V. Jain, V. Vanhoucke, W. Zhan, W. Zhou, W. Burgard, X. Chen, X. Chen, X. Wang, X. Zhu, X. Geng, X. Liu, X. Liangwei, X. Li, Y. Pang, Y. Lu, Y. J. Ma, Y. Kim, Y. Chebotar, Y. Zhou, Y. Zhu, Y. Wu, Y. Xu, Y. Wang, Y. Bisk, Y. Cho, Y. Lee, Y. Cui, Y. Cao, Y.-H. Wu, Y. Tang, Y. Zhu, Y. Zhang, Y. Jiang, Y. Li, Y. Li, Y. Iwasawa, Y. Matsuo, Z. Ma, Z. Xu, Z. J. Cui, Z. Zhang, Z. Fu, and Z. Lin. Open x-embodiment: Robotic learning datasets and rt-x models, 2024.

[7] M. Kim, K. Pertsch, S. Karamcheti, T. Xiao, A. Balakrishna, S. Nair, R. Rafailov, E. Foster, P. Sanketi, Q. Vuong, T. Kollar, B. Burchfiel, R. Tedrake, D. Sadigh, S. Levine, P. Liang, and C. Finn. Openvla: An open-source vision-language-action model. 2024.

[8] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[9] H. W. Chung, L. Hou, S. Longpre, B. Zoph, Y. Tay, W. Fedus, Y. Li, X. Wang, M. Dehghani, S. Brahma, A. Webson, S. S. Gu, Z. Dai, M. Suzgun, X. Chen, A. Chowdhery, A. Castro-Ros, M. Pellat, K. Robinson, D. Valter, S. Narang, G. Mishra, A. Yu, V. Zhao, Y. Huang, A. Dai, H. Yu, S. Petrov, E. H. Chi, J. Dean, J. Devlin, A. Roberts, D. Zhou, Q. V. Le, and J. Wei. Scaling instruction-finetuned language models, 2022.

[10] D. Kalashnikov, A. Irpan, P. Pastor, J. Ibarz, A. Herzog, E. Jang, D. Quillen, E. Holly, M. Kalakrishnan, V. Vanhoucke, and S. Levine. Qt-opt: Scalable deep reinforcement learning for vision-based robotic manipulation, 2018.

[11] D. Kalashnikov, J. Varley, Y. Chebotar, B. Swanson, R. Jonschkowski, C. Finn, S. Levine, and K. Hausman. Mt-opt: Continuous multi-task robotic reinforcement learning at scale, 2021.

[12] F. Ebert, Y. Yang, K. Schmeckpeper, B. Bucher, G. Georgakis, K. Daniilidis, C. Finn, and S. Levine. Bridge data: Boosting generalization of robotic skills with cross-domain datasets, 2021.

[13] H. Walke, K. Black, A. Lee, M. J. Kim, M. Du, C. Zheng, T. Zhao, P. Hansen-Estruch, Q. Vuong, A. He, V. Myers, K. Fang, C. Finn, and S. Levine. Bridgedata v2: A dataset for robot learning at scale. In *Conference on Robot Learning (CoRL)*, 2023.

[14] A. Brohan, N. Brown, J. Carbajal, Y. Chebotar, J. Dabis, C. Finn, K. Gopalakrishnan, K. Hausman, A. Herzog, J. Hsu, J. Ibarz, B. Ichter, A. Irpan, T. Jackson, S. Jesmonth, N. J. Joshi, R. Julian, D. Kalashnikov, Y. Kuang, I. Leal, K.-H. Lee, S. Levine, Y. Lu, U. Malla, D. Manjunath, I. Mordatch, O. Nachum, C. Parada, J. Peralta, E. Perez, K. Pertsch, J. Quiambao, K. Rao, M. Ryoo, G. Salazar, P. Sanketi, K. Sayed, J. Singh, S. Sontakke, A. Stone, C. Tan, H. Tran, V. Vanhoucke, S. Vega, Q. Vuong, F. Xia, T. Xiao, P. Xu, S. Xu, T. Yu, and B. Zitkovich. Rt-1: Robotics transformer for real-world control at scale, 2023.

[15] H. Bharadhwaj, J. Vakil, M. Sharma, A. Gupta, S. Tulsiani, and V. Kumar. Roboagent: Generalization and efficiency in robot manipulation via semantic augmentations and action chunking, 2023.

[16] Octo Model Team, D. Ghosh, H. Walke, K. Pertsch, K. Black, O. Mees, S. Dasari, J. Hejna, C. Xu, J. Luo, T. Kreiman, Y. Tan, L. Y. Chen, P. Sanketi, Q. Vuong, T. Xiao, D. Sadigh, C. Finn, and S. Levine. Octo: An open-source generalist robot policy. In *Proceedings of Robotics: Science and Systems*, Delft, Netherlands, 2024.

[17] D. Shah, A. Sridhar, N. Dashora, K. Stachowicz, K. Black, N. Hirose, and S. Levine. Vint: A foundation model for visual navigation, 2023.

[18] L. Pinto and A. Gupta. Supersizing self-supervision: Learning to grasp from 50k tries and 700 robot hours, 2015.

[19] A. Mandlekar, Y. Zhu, A. Garg, J. Booher, M. Spero, A. Tung, J. Gao, J. Emmons, A. Gupta, E. Orbay, S. Savarese, and L. Fei-Fei. Roboturk: A crowdsourcing platform for robotic skill learning through imitation, 2018.

[20] A. Gupta, A. Murali, D. Gandhi, and L. Pinto. Robot learning in homes: Improving generalization and reducing dataset bias, 2018.

[21] S. Dasari, F. Ebert, S. Tian, S. Nair, B. Bucher, K. Schmeckpeper, S. Singh, S. Levine, and C. Finn. Robonet: Large-scale multi-robot learning, 2020.

[22] S. Cabi, S. G. Colmenarejo, A. Novikov, K. Konyushkova, S. Reed, R. Jeong, K. Zolna, Y. Aytar, D. Budden, M. Vecerik, O. Sushkov, D. Barker, J. Scholz, M. Denil, N. de Freitas, and Z. Wang. Scaling data-driven robotics with reward sketching and batch reinforcement learning, 2020.

[23] E. Jang, A. Irpan, M. Khansari, D. Kappler, F. Ebert, C. Lynch, S. Levine, and C. Finn. Bc-z: Zero-shot task generalization with robotic imitation learning. In *Conference on Robot Learning*, pages 991–1002. PMLR, 2022.

[24] H.-S. Fang, H. Fang, Z. Tang, J. Liu, J. Wang, H. Zhu, and C. Lu. Rh20t: A robotic dataset for learning diverse skills in one-shot. In *RSS 2023 Workshop on Learning for Task and Motion Planning*, 2023.

[25] A. Khazatsky, K. Pertsch, S. Nair, A. Balakrishna, S. Dasari, S. Karamcheti, S. Nasiriany, M. K. Srirama, L. Y. Chen, K. Ellis, P. D. Fagan, J. Hejna, M. Itkina, M. Lepert, Y. J. Ma, P. T. Miller, J. Wu, S. Belkhale, S. Dass, H. Ha, A. Jain, A. Lee, Y. Lee, M. Memmel, S. Park, I. Radosavovic, K. Wang, A. Zhan, K. Black, C. Chi, K. B. Hatch, S. Lin, J. Lu, J. Mercat, A. Rehman, P. R. Sanketi, A. Sharma, C. Simpson, Q. Vuong, H. R. Walke, B. Wulfe, T. Xiao, J. H. Yang, A. Yavary, T. Z. Zhao, C. Agia, R. Baijal, M. G. Castro, D. Chen, Q. Chen, T. Chung, J. Drake, E. P. Foster, J. Gao, D. A. Herrera, M. Heo, K. Hsu, J. Hu, D. Jackson, C. Le, Y. Li, K. Lin, R. Lin, Z. Ma, A. Maddukuri, S. Mirchandani, D. Morton, T. Nguyen, A. O'Neill, R. Scalise, D. Seale, V. Son, S. Tian, E. Tran, A. E. Wang, Y. Wu, A. Xie, J. Yang, P. Yin, Y. Zhang, O. Bastani, G. Berseth, J. Bohg, K. Goldberg, A. Gupta, A. Gupta, D. Jayaraman, J. J. Lim, J. Malik, R. Martín-Martín, S. Ramamoorthy, D. Sadigh, S. Song, J. Wu, M. C. Yip, Y. Zhu, T. Kollar, S. Levine, and C. Finn. Droid: A large-scale in-the-wild robot manipulation dataset. 2024.

[26] K. Bousmalis, G. Vezzani, D. Rao, C. Devin, A. X. Lee, M. Bauza, T. Davchev, Y. Zhou, A. Gupta, A. Raju, A. Laurens, C. Fantacci, V. Dalibard, M. Zambelli, M. Martins, R. Pevceviciute, M. Blokzijl, M. Denil, N. Batchelor, T. Lampe, E. Parisotto, K. Żołna, S. Reed, S. G. Colmenarejo, J. Scholz, A. Abdolmaleki, O. Groth, J.-B. Regli, O. Sushkov, T. Rothörl, J. E. Chen, Y. Aytar, D. Barker, J. Ortiz, M. Riedmiller, J. T. Springenberg, R. Hadsell, F. Nori, and N. Heess. Robocat: A self-improving generalist agent for robotic manipulation, 2023.

[27] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever. Learning transferable visual models from natural language supervision, 2021.

[28] S. Liu, Z. Zeng, T. Ren, F. Li, H. Zhang, J. Yang, C. Li, J. Yang, H. Su, J. Zhu, and L. Zhang. Grounding dino: Marrying dino with grounded pre-training for open-set object detection, 2023.

[29] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer. High-resolution image synthesis with latent diffusion models, 2022.

[30] H. Liu, C. Li, Q. Wu, and Y. J. Lee. Visual instruction tuning, 2023.

[31] J. Li, D. Li, C. Xiong, and S. Hoi. Blip: Bootstrapping language-image pre-training for unified vision-language understanding and generation, 2022.

[32] J. Li, D. Li, S. Savarese, and S. Hoi. Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models, 2023.

[33] W. Dai, J. Li, D. Li, A. M. H. Tiong, J. Zhao, W. Wang, B. Li, P. Fung, and S. Hoi. Instructblip: Towards general-purpose vision-language models with instruction tuning, 2023.

[34] S. Karamcheti, S. Nair, A. Balakrishna, P. Liang, T. Kollar, and D. Sadigh. Prismatic vlms: Investigating the design space of visually-conditioned language models, 2024.

[35] K. Black, M. Nakamoto, P. Atreya, H. Walke, C. Finn, A. Kumar, and S. Levine. Zero-shot robotic manipulation with pretrained image-editing diffusion models, 2023.

[36] Y. Du, K. Konyushkova, M. Denil, A. Raju, J. Landon, F. Hill, N. de Freitas, and S. Cabi. Vision-language models as success detectors, 2023.

[37] S. A. Sontakke, J. Zhang, S. M. R. Arnold, K. Pertsch, E. Bıyık, D. Sadigh, C. Finn, and L. Itti. Roboclip: One demonstration is enough to learn robot policies, 2023.

[38] Y. J. Ma, W. Liang, V. Som, V. Kumar, A. Zhang, O. Bastani, and D. Jayaraman. Liv: Language-image representations and rewards for robotic control, 2023.

[39] S. Nair, A. Rajeswaran, V. Kumar, C. Finn, and A. Gupta. R3m: A universal visual representation for robot manipulation, 2022.

[40] S. Karamcheti, S. Nair, A. S. Chen, T. Kollar, C. Finn, D. Sadigh, and P. Liang. Language-driven representation learning for robotics, 2023.

[41] W. Chen, O. Mees, A. Kumar, and S. Levine. Vision-language models provide promptable representations for reinforcement learning, 2024.

[42] A. Stone, T. Xiao, Y. Lu, K. Gopalakrishnan, K.-H. Lee, Q. Vuong, P. Wohlhart, S. Kirmani, B. Zitkovich, F. Xia, C. Finn, and K. Hausman. Open-world object manipulation using pre-trained vision-language models. In *arXiv preprint*, 2023.

[43] M. Shridhar, L. Manuelli, and D. Fox. Perceiver-actor: A multi-task transformer for robotic manipulation. In *Proceedings of the 6th Conference on Robot Learning (CoRL)*, 2022.

[44] F. Liu, K. Fang, P. Abbeel, and S. Levine. Moka: Open-vocabulary robotic manipulation through mark-based visual prompting, 2024.

[45] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa. Large language models are zero-shot reasoners, 2023.

[46] P. Lu, B. Peng, H. Cheng, M. Galley, K.-W. Chang, Y. N. Wu, S.-C. Zhu, and J. Gao. Chameleon: Plug-and-play compositional reasoning with large language models, 2023.

[47] W. Huang, F. Xia, T. Xiao, H. Chan, J. Liang, P. Florence, A. Zeng, J. Tompson, I. Mordatch, Y. Chebotar, P. Sermanet, N. Brown, T. Jackson, L. Luu, S. Levine, K. Hausman, and B. Ichter. Inner monologue: Embodied reasoning through planning with language models, 2022.

[48] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng. Code as policies: Language model programs for embodied control, 2023.

[49] A. Zeng, M. Attarian, B. Ichter, K. Choromanski, A. Wong, S. Welker, F. Tombari, A. Purohit, M. Ryoo, V. Sindhwani, J. Lee, V. Vanhoucke, and P. Florence. Socratic models: Composing zero-shot multimodal reasoning with language, 2022.

[50] O. Mees, J. Borja-Diaz, and W. Burgard. Grounding language with visual affordances over unstructured data. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, London, UK, 2023.

[51] P. Sharma, A. Torralba, and J. Andreas. Skill induction and planning with latent language, 2022.

[52] I. Singh, V. Blukis, A. Mousavian, A. Goyal, D. Xu, J. Tremblay, D. Fox, J. Thomason, and A. Garg. Progprompt: Generating situated robot task plans using large language models, 2022.

[53] H. Ha, P. Florence, and S. Song. Scaling up and distilling down: Language-guided robot skill acquisition, 2023.

[54] X. Zhai, B. Mustafa, A. Kolesnikov, and L. Beyer. Sigmoid loss for language image pre-training, 2023.

[55] M. Oquab, T. Darcet, T. Moutakanni, H. Vo, M. Szafraniec, V. Khalidov, P. Fernandez, D. Haziza, F. Massa, A. El-Nouby, M. Assran, N. Ballas, W. Galuba, R. Howes, P.-Y. Huang, S.-W. Li, I. Misra, M. Rabbat, V. Sharma, G. Synnaeve, H. Xu, H. Jegou, J. Mairal, P. Labatut, A. Joulin, and P. Bojanowski. Dinov2: Learning robust visual features without supervision, 2024.

[56] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[57] M. Minderer, A. Gritsenko, and N. Houlsby. Scaling open-vocabulary object detection. *Advances in Neural Information Processing Systems*, 36, 2024.

[58] A. Kirillov, E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo, et al. Segment anything. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4015–4026, 2023.

[59] Gemini Team. Gemini: A family of highly capable multimodal models, 2024.

[60] S. Mukherjee, A. Mitra, G. Jawahar, S. Agarwal, H. Palangi, and A. Awadallah. Orca: Progressive learning from complex explanation traces of gpt-4, 2023.

[61] S. Belkhale, T. Ding, T. Xiao, P. Sermanet, Q. Vuong, J. Tompson, Y. Chebotar, D. Dwibedi, and D. Sadigh. Rt-h: Action hierarchies using language, 2024.

[62] M. A. Fischler and R. C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6): 381–395, 1981.

[63] NVIDIA. Tensorrt-llm. https://github.com/NVIDIA/TensorRT-LLM?tab=readme-ov-file, 2024.

[64] Y. Leviathan, M. Kalman, and Y. Matias. Fast inference from transformers via speculative decoding, 2023.

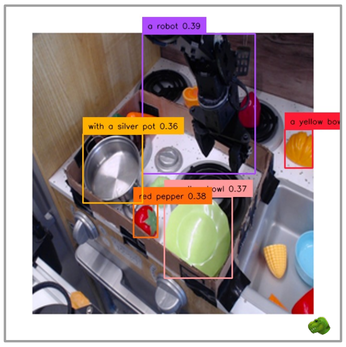
[65] M. Kelly, C. Sidrane, K. Driggs-Campbell, and M. J. Kochenderfer. Hg-dagger: Interactive imitation learning with human experts, 2019.

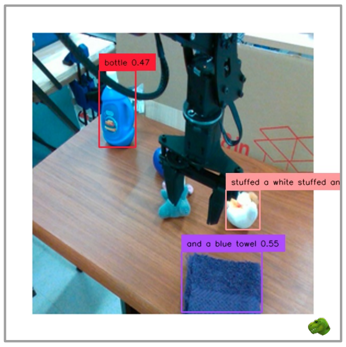
[66] P. Sharma, B. Sundaralingam, V. Blukis, C. Paxton, T. Hermans, A. Torralba, J. Andreas, and D. Fox. Correcting robot plans with natural language feedback, 2022.

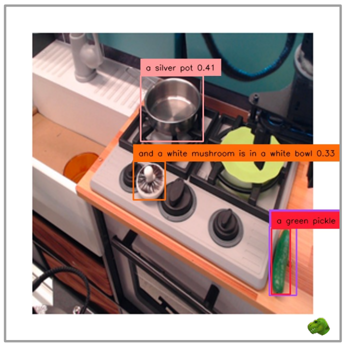
[67] L. X. Shi, Z. Hu, T. Z. Zhao, A. Sharma, K. Pertsch, J. Luo, S. Levine, and C. Finn. Yell at your robot: Improving on-the-fly from language corrections, 2024.

A Grounding DINO Detections and Prismatic Descriptions

We provide example scene descriptions provided by Prismatic VLM and bounding boxes provided by Grounding DINO in Fig. 7.



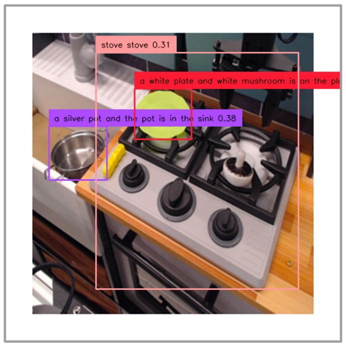


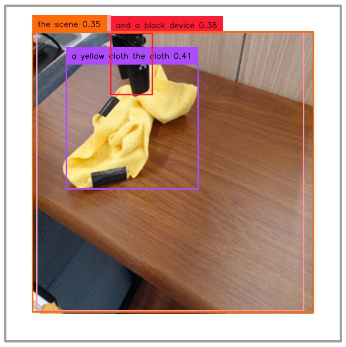


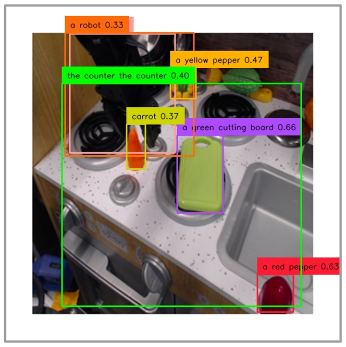
A robot is holding a carrot above a green plate, with a silver pot, red pepper, and a yellow bowl on the counter

A wooden table holds a blue bottle, a grey stuffed animal, a white stuffed animal, and a blue towel, with a box and a black arm behind them

A green pickle is on the counter, a silver pot is on the stove burner, and a white mushroom is in a white bowl. The stove burner is below the pot, and the pickle is near the counter


stove stove 0.31
a white plate and white mushroom is on the pl
a silver pot and the pot is in the sink 0.38



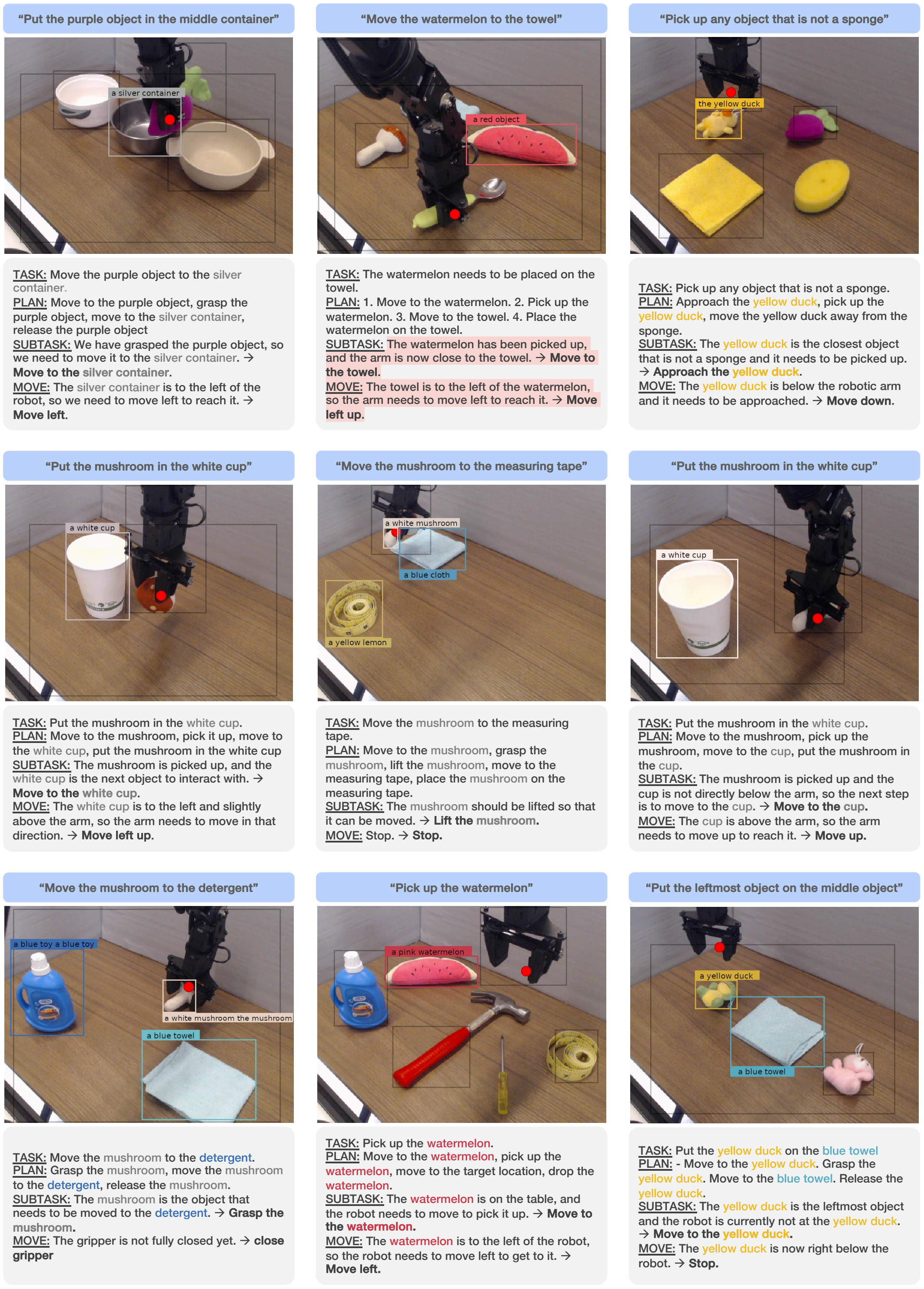


A silver pot is in the sink, a white plate is on the stove, and a white mushroom is on the plate. The sink is to the left of the stove, and the pot is in the sink

The scene has a table with a blue chair, a yellow cloth on the table, and a black device. The cloth is to be moved from the left side of the table to the center.

A robot is putting a carrot on a cutting board, which is on the counter. There is a green cutting board, a red pepper, a yellow pepper, and a black pot nearby. The counter is white with silver knobs and a sink.

Figure 7: Examples of captions of observations from the Bridge dataset as generated by our Prismatic VLM, as well as associated bounding boxes generated by Grounding DINO.


"Put the purple object in the middle container"
a silver container
"Move the watermelon to the towel"
a red object
"Pick up any object that is not a sponge"
the yellow duck
"Put the mushroom in the white cup"
a white cup
"Move the mushroom to the measuring tape"
a white mushroom
a blue cloth
a yellow lemon
"Put the mushroom in the white cup"
a white cup
"Move the mushroom to the detergent"
a blue toy a blue toy
a white mushroom the mushroom
a blue towel
"Pick up the watermelon"
a pink watermelon
"Put the leftmost object on the middle object"
a yellow duck
a blue towel

"Put the purple object in the middle container"

TASK: Move the purple object to the silver container.
PLAN: Move to the purple object, grasp the purple object, move to the silver container, release the purple object
SUBTASK: We have grasped the purple object, so we need to move it to the silver container. → **Move to the silver container.**
MOVE: The silver container is to the left of the robot, so we need to move left to reach it. → **Move left.**

"Move the watermelon to the towel"

TASK: The watermelon needs to be placed on the towel.
PLAN: 1. Move to the watermelon. 2. Pick up the watermelon. 3. Move to the towel. 4. Place the watermelon on the towel.
SUBTASK: The watermelon has been picked up, and the arm is now close to the towel. → **Move to the towel.**
MOVE: The towel is to the left of the watermelon, so the arm needs to move left to reach it. → **Move left up.**

"Pick up any object that is not a sponge"

TASK: Pick up any object that is not a sponge.
PLAN: Approach the yellow duck, pick up the yellow duck, move the yellow duck away from the sponge.
SUBTASK: The yellow duck is the closest object that is not a sponge and it needs to be picked up. → **Approach the yellow duck.**
MOVE: The yellow duck is below the robotic arm and it needs to be approached. → **Move down.**

"Put the mushroom in the white cup"

TASK: Put the mushroom in the white cup.
PLAN: Move to the mushroom, pick it up, move to the white cup, put the mushroom in the white cup
SUBTASK: The mushroom is picked up, and the white cup is the next object to interact with. → **Move to the white cup.**
MOVE: The white cup is to the left and slightly above the arm, so the arm needs to move in that direction. → **Move left up.**

"Move the mushroom to the measuring tape"

TASK: Move the mushroom to the measuring tape.
PLAN: Move to the mushroom, grasp the mushroom, lift the mushroom, move to the measuring tape, place the mushroom on the measuring tape.
SUBTASK: The mushroom should be lifted so that it can be moved. → **Lift the mushroom.**
MOVE: Stop. → **Stop.**

"Put the mushroom in the white cup"

TASK: Put the mushroom in the white cup.
PLAN: Move to the mushroom, pick up the mushroom, move to the cup, put the mushroom in the cup.
SUBTASK: The mushroom is picked up and the cup is not directly below the arm, so the next step is to move to the cup. → **Move to the cup.**
MOVE: The cup is above the arm, so the arm needs to move up to reach it. → **Move up.**

"Move the mushroom to the detergent"

TASK: Move the mushroom to the detergent.
PLAN: Grasp the mushroom, move the mushroom to the detergent, release the mushroom.
SUBTASK: The mushroom is the object that needs to be moved to the detergent. → **Grasp the mushroom.**
MOVE: The gripper is not fully closed yet. → **close gripper**

"Pick up the watermelon"

TASK: Pick up the watermelon.
PLAN: Move to the watermelon, pick up the watermelon, move to the target location, drop the watermelon.
SUBTASK: The watermelon is on the table, and the robot needs to move to pick it up. → **Move to the watermelon.**
MOVE: The watermelon is to the left of the robot, so the robot needs to move left to get to it. → **Move left.**

"Put the leftmost object on the middle object"

TASK: Put the yellow duck on the blue towel
PLAN: - Move to the yellow duck. Grasp the yellow duck. Move to the blue towel. Release the yellow duck.
SUBTASK: The yellow duck is the leftmost object and the robot is currently not at the yellow duck. → **Move to the yellow duck.**
MOVE: The yellow duck is now right below the robot. → **Stop.**

Figure 8: More qualitative examples of successful and failed chain-of-thought reasonings.

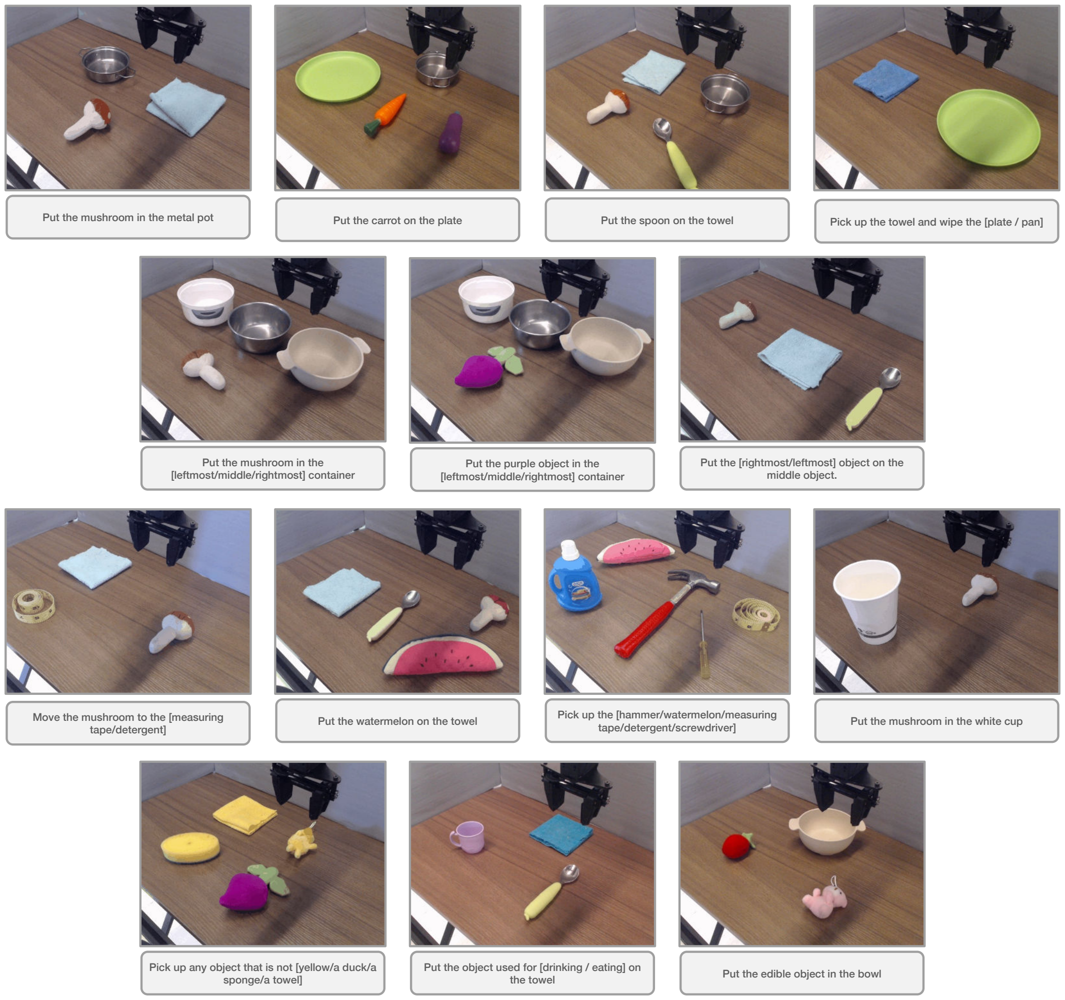

Put the mushroom in the metal pot
Put the carrot on the plate
Put the spoon on the towel
Pick up the towel and wipe the [plate / pan]
Put the mushroom in the [leftmost/middle/rightmost] container
Put the purple object in the [leftmost/middle/rightmost] container
Put the [rightmost/leftmost] object on the middle object.
Move the mushroom to the [measuring tape/detergent]
Put the watermelon on the towel
Pick up the [hammer/watermelon/measuring tape/detergent/screwdriver]
Put the mushroom in the white cup
Pick up any object that is not [yellow/a duck/a sponge/a towel]
Put the object used for [drinking / eating] on the towel
Put the edible object in the bowl

Figure 9: Example starting scenes and associated prompt for all task types.

B List of Movement Primitives

To classify a movement, we take the difference between the current state of the robot and its position four steps ahead. Based on the axes where the difference exceeds a threshold of 0.03, we assign it a label of the following form:

```
move [forward/backward] [left/right] [up/down], tilt [up/down], rotate
        [clockwise/counterclockwise], [close/open] gripper
```

Whenever the movement in a certain axis is below the threshold, we omit its block for simplicity. For instance, if the robot is just moving left, the label is `move left`. If no movement is detected, the label is `stop`.

While technically it results in $3^6 = 729$ possible labels, only 54 are used in more than 0.1% of cases:

1. `stop` (26.9%)
2. `close gripper` (10.8%)
3. `open gripper` (7.2%)
4. `move down` (6.8%)
5. `move left` (6.6%)
6. `move right` (6.1%)
7. `move up` (5.7%)
8. `move forward` (3.0%)
9. `move backward` (2.4%)
10. `move up, open gripper` (2.1%)
11. `move forward right` (1.1%)
12. `move up, close gripper` (1.0%)
13. `move backward left` (1.0%)
14. `move forward left` (0.9%)
15. `move left down` (0.8%)
16. `move down, close gripper` (0.8%)
17. `move right down` (0.8%)
18. `move left up` (0.8%)
19. `move right up` (0.8%)
20. `move right, rotate clockwise` (0.8%)
21. `move left, rotate counterclockwise` (0.8%)
22. `move backward right` (0.8%)
23. `rotate counterclockwise` (0.7%)
24. `move down, open gripper` (0.7%)
25. `rotate clockwise` (0.7%)
26. `move forward down` (0.7%)
27. `move up, rotate clockwise` (0.5%)
28. `move up, rotate counterclockwise` (0.5%)
29. `move backward up` (0.5%)
30. `move left, rotate clockwise` (0.3%)
31. `move backward down` (0.3%)
32. `move right, open gripper` (0.3%)
33. `move forward up` (0.3%)
34. `move left, open gripper` (0.3%)
35. `move right, rotate counterclockwise` (0.3%)
36. `move backward, open gripper` (0.2%)
37. `move down, rotate clockwise` (0.2%)
38. `move down, rotate counterclockwise` (0.2%)
39. `move forward, rotate counterclockwise` (0.2%)
40. `move forward, rotate clockwise` (0.2%)
41. `move forward, open gripper` (0.2%)
42. `move right, close gripper` (0.2%)
43. `move backward, rotate clockwise` (0.2%)
44. `move backward, rotate counterclockwise` (0.2%)
45. `move left, close gripper` (0.2%)
46. `move backward right, rotate clockwise` (0.1%)
47. `move backward left, rotate counterclockwise` (0.1%)
48. `move right up, open gripper` (0.1%)
49. `move right up, close gripper` (0.1%)
50. `move backward, close gripper` (0.1%)
51. `rotate clockwise, close gripper` (0.1%)
52. `rotate counterclockwise, close gripper` (0.1%)
53. `move left up, open gripper` (0.1%)
54. `move forward right, rotate clockwise` (0.1%)

C Prompts

We now provide all the prompts used for data generation and policy language conditioning.

For using generating scene descriptions with Prismatic (step 1 in Fig. 4), we use the prompt: “Briefly describe the things in this scene and their spatial relations to each other.” We prepend “The robot task is: [*TASK*].” if the given demonstration trajectory contains a corresponding task instruction (where we ensure that said instruction contains at least one space character to remove noisy instructions).

We provide the prompt for Gemini data labeling (step 5 in Fig. 4) in Fig. 10.

The prompts used for our language-conditioned policies are provided in Fig. 9, along with example starting scenes for the associated tasks. For the OpenVLA-based policies, said prompts are inserted into the template provided by the original authors [7]: “A chat between a curious user and an artificial intelligence assistant. The assistant gives helpful, detailed, and polite answers to the user’s questions. USER: What action should the robot take to [*PROMPT*]? ASSISTANT:”. The agent then generates reasoning text (if trained to do so) and an action.

We provide the prompt used for human interventions with ChatGPT in Fig. 11.

```
Annotate the training trajectory with reasoning

## Specification of the experimental setup

You’re an expert reinforcement learning researcher.  You’ve trained an optimal policy for controlling a robotic arm.  The
robot successfully completed a task specified by the instruction:  "unfold the cloth from top right to bottom left".  For
that purpose, the robotic arm executed a sequence of actions.  Consecutive states that were visited can be characterized by
the following features:

‘‘‘python
trajectory_features = {
    0: "stop"
    1: "stop"
    2: "move␣forward␣left"
    3: "move␣forward␣down"
    4: "move␣forward␣down"
    ...
    36: "stop"
}
‘‘‘

Each entry in that dictionary corresponds to a single step on the trajectory and describes the move that is about to be
executed.

## Scene description

The robot is operating in the following environment.  A black and red toy stove with a yellow banana in a silver pot, a blue
toy brush, and a purple towel on the counter, surrounded by white tiled walls and a grey sink.

## Your objective

I want you to annotate the given trajectory with reasoning.  That is, for each step, I need to know not only which action
should be chosen, but importantly what reasoning justifies that action choice.  I want you to be descriptive and include all
the relevant information available.  The reasoning should include the task to complete, the remaining high-level steps, the
high-level movements that should be executed and why they are required, the premises that allow inferring the direction of
each move, including the locations of relevant objects, possible obstacles or difficulties to avoid, and any other relevant
justification.

### Begin by describing the task

Start by giving an overview of the task.  Make it more comprehensive than the simple instruction.  Include the activity,
the objects the robotic arm interacts with, and their relative locations in the environment.  Then, describe the high-level
movements that were most likely executed, based on the task that was completed and the primitive movements that were
executed.  Then, for each high-level movement write the interval of steps that movement consists of.  Also, for each
high-level movement write a justification for why it should be executed.  Write an answer for this part using markdown and
natural language.  Be descriptive and highlight all the relevant details, but ensure that your description is consistent
with the trajectory that was executed, specified by the features listed above in the ‘trajectory_features‘ dictionary.

### List the reasonings for each step

Finally, for each step describe the reasoning that allows to determine the correct action.  For each step describe the
remaining part of the objective, the current progress, the objects that are still relevant for determining the plan,
and the plan for the next steps, based on the available features.  Start the reasoning from a high level and gradually
add finer features.  I need you to be descriptive and very precise.  Ensure that the reasoning is consistent with the
task and the executed trajectory.  Write the answer for this part as a Python-executable dictionary.  For every step in
the initial trajectory there should be exactly one separate item of the form <step id>:<reasoning>.  Do not group the
answers.  The final dictionary should have exactly the same set of integer keys as the dictionary of features provided in
the ‘trajectory_features‘ dictionary above.  The reasoning should be a single string that describes the reasoning in natural
language and includes all the required features.

Each reasoning string should have the following form:
- Describe the full task that remains to be completed (but only describe what remains), and place it inside a tag <task>.
- Describe the complete high-level plan for completing the remaining task (the list of remaining high-level steps), and place
it inside a tag <plan>.
- Describe the high-level step that should be executed now (chosen from the list of high-level steps), and place it inside a
tag <subtask>.
- Describe why the chosen high-level step should be executed now, which features of the current environment influence that
decision, and how it should be done.  Place it within a tag <subtask_reason>.
- Describe the current primitive movement of the arm that needs to be executed, and place it inside a tag <move>.
- Describe why the chosen movement should be executed now and which features of the current environment influence that
decision.  Place it inside a tag <move_reason>.

## Task summary

Here is a breakdown of what needs to be done:

- Describe the task.
- Describe the high-level movements that were executed, based on the completed task and the listed features.
- Describe the plan for the solution that allowed the robot to complete the task successfully.
- For each step on the trajectory, describe the reasoning that leads to determining the correct action.  The reasoning should
be descriptive and precise.  You should provide exactly one reasoning string for each step on the trajectory specified by
‘trajectory_features‘.
- At the very end of the response, write a single label FINISHED to indicate that the answer is complete.
```

Figure 10: Prompt used for Gemini to generate plans, subtasks, and movement labels.

```
# Objective

You're an expert reinforcement learning researcher.  You've trained a policy for controlling a robotic arm.  The policy
computes the correct action based on a reasoning that leads to it, which includes the task that remains to be completed,
the plan for completing that task, and the subtask that currently needs to be done.  I want you to prepare such a reasoning,
based on a feedback from a user of that robot.

The reasoning must have the following elements:
- TASK: the task that remains to be done.
- PLAN: a list of high-level steps that need to be executed.
- SUBTASK REASONING: reasoning that determines the current subtask.
- SUBTASK REASONING: reasoning that determines the current subtask.
- SUBTASK: the current subtask that should be executed.
- MOVE REASONING: reasoning that determines the current move.
- MOVE: the current move that should be executed

Write the answer as a python string.  It will be used as an additional input for the policy, so keep the format exactly as
described.

# Examples

Given the task "Put the tomato inside the pot on the left burner" and feedback "you are too low, move up", the reasoning
should be "TASK: Put the tomato inside the pot on the left burner.  PLAN: Go to the tomato, grasp it, transport it to the
stove, position it in the pot.  SUBTASK REASONING: The tomato is grasped.  The tomato is already near the pot, but below its
edge.  SUBTASK: Position the tomato in the pot.  MOVE REASONING: The pot is above current position.  Move the arm up.  MOVE:
Move up."

Given the task "place the silver lid on the silver pot on the upper right of the table" and feedback "move to the pot", the
reasoning should be "TASK: The lid needs to be placed on a silver pot on the upper right part of the scene.  PLAN: First
move to the lid, then grip it, then move to the pot, then place the lid on the pot.  SUBTASK REASONING: The lid is gripped,
so it should be moved to the pot.  SUBTASK: Move to the pot."

Given the task "move the fork to the bottom left side of the counter" and feedback "move down to grasp the fork", the
reasoning should be "TASK: Pick up the fork and move it to the bottom left side of the counter.  PLAN: 1.  Move to the fork.
2.  Pick up the fork.  3.  Move to the bottom left side of the counter.  4.  Put down the fork.  SUBTASK REASONING: The fork
is the first object that needs to be reached.  SUBTASK: 1.  Move to the fork.  MOVE REASONING: The fork is still downward
from the current position of the arm, so the arm continues to move that direction.  MOVE: move down"

Given the task "remove the cylinder from the green cube and place it on top of the red cube" and feedback "close", the
reasoning should be "TASK: Remove the cylinder from the green cube and place it on top of the red cube.  PLAN: Approach the
green cube, close the gripper around the cylinder, move the cylinder towards the red cube, open the gripper to place the
cylinder on top of the red cube.  SUBTASK REASONING: The arm is now in contact with the cylinder, so it should close the
gripper to grab it.  SUBTASK: Close the gripper MOVE REASONING: The cylinder has already been reached and the gripper is
closing.  MOVE: Close gripper"

Given the task "pick up the towel" and feedback "go right", the reasoning should be "TASK: Pick up the towel.  PLAN: Move to
the towel, Grasp the towel, Pick up the towel SUBTASK REASONING: The arm should reach the towel first.  SUBTASK: Move to the
towel.  MOVE REASONING: The towel is to the right, so the arm should move right.  MOVE: move right"

# The current task

The policy generated the following reasoning:  "TASK: Put the mushroom in the white cup.  PLAN: Move to the mushroom, pick
it up, move to the cup, put the mushroom in the cup.  SUBTASK REASONING: The mushroom is picked up, and the cup is the next
object to interact with.  SUBTASK: Move to the cup.  MOVE REASONING: The cup is positioned below the arm.  MOVE: Move down.
GRIPPER POSITION: [111, 61] VISIBLE OBJECTS: a white cup [124, 25, 176, 113], a wooden table [13, 21, 241, 248], a wooden
table [10, 21, 249, 250]"

Given the task "put the mushroom in the white cup" and feedback "no, the cup is actually in front of the gripper", what
should be the reasoning?
```

Figure 11: Prompt used for ChatGPT during human intervention experiments

D Analysis of Embodied Chain-of-Thought Variants

We present some qualitative analyses of our ECoT variants (frozen bounding box, vision-language co-trained, and OXE fine-tuned).

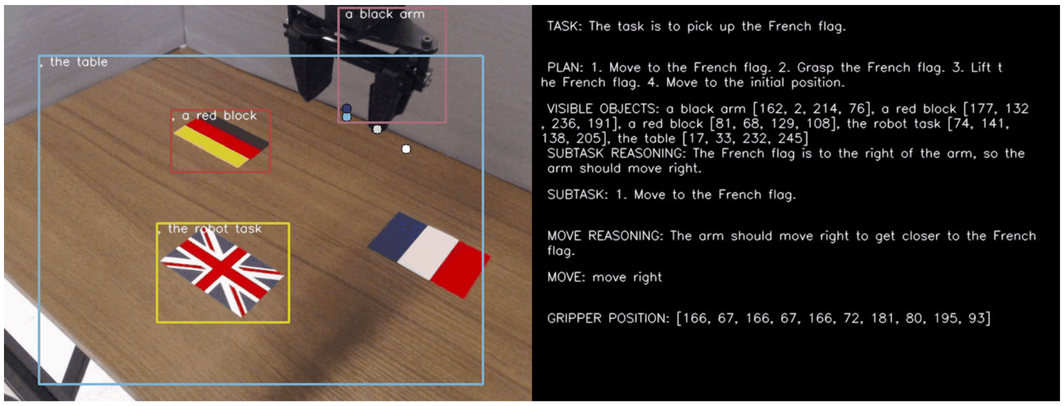
All approaches are slightly less performant than our base ECoT policy. However, they also all nonetheless outperform equivalent OpenVLA policies.

Vision-language co-training. We expected vision-language co-training to allow the VLA to keep even more of its vision-language modeling capabilities and knowledge. We find that, while this is indeed the case, it is not sufficient to leverage this for easier human-robot interaction experiments (e.g., performing human interventions without the aid of an external "pure" vision-language model). We also test semantic generalization tasks similar to those presented by Brohan et al. [5]. Specifically, we tested whether the co-trained model could successfully recognize concepts not present in the training data, such as people, countries, animals, letters, or landmarks. Our findings indicate that co-training mildly improves the overall quality of reasoning and allows the model to consistently recognize celebrities, such as Taylor Swift or Donald Trump. However, in other tasks, co-training offers little to no benefit. We attribute this to the model's ability to retain most of its prior knowledge through the generation of diverse reasonings, even if those concepts are not directly expressed in the data. Some of the reasonings are illustrated in Figure 12.

Fine-tuning for other embodiments. For the OXE fine-tuned variant, we find that the model can perform embodied chain-of-thought reasoning on the WidowX station after $20,000$ gradient steps, approximately one fifth of the updates needed for training an ECoT VLA from a base Prismatic VLM (due to the fine-tuned variant starting with an OpenVLA, which already can map observations to actions). In preliminary evaluations on a small subset of tasks, we find that this is even the case after just $2,500$ steps, though there is no significant performance difference between said checkpoint and the base OXE-pre-trained OpenVLA.

Additionally, when presented with observations and tasks from other robot embodiments, we find that this fine-tuned model can generate plausible reasonings as well (including visually-grounded features like the end effector position, which is naturally highly dependent on the robot embodiment). *This is an emergent generalization, given that we only trained the model on the Bridge V2 WidowX reasoning data*, with all other OXE demonstrations lacking such reasoning annotations. See Figure 13 for qualitative examples. While we leave analysis of this generalization to future works, the ability for the model to produce reasonings for embodiments that it was not trained on offers a promising channel for scaling up robot reasoning, e.g., by bootstrapping good reasonings for downstream training.

wc: TODO: Add SIMPLER expts if they exist

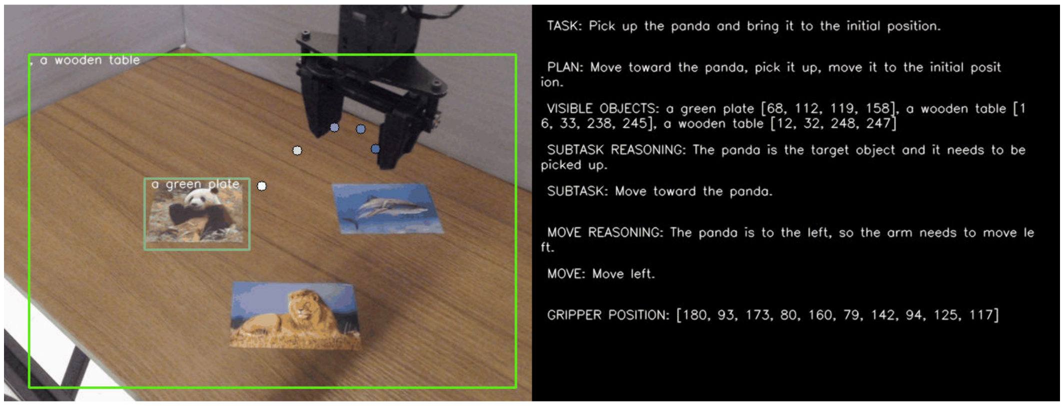


```
TASK: The task is to pick up the French flag.

PLAN: 1. Move to the French flag. 2. Grasp the French flag. 3. Lift t
he French flag. 4. Move to the initial position.
VISIBLE OBJECTS: a black arm [162, 2, 214, 76], a red block [177, 132
, 236, 191], a red block [81, 68, 129, 108], the robot task [74, 141,
138, 205], the table [17, 33, 232, 245]
SUBTASK REASONING: The French flag is to the right of the arm, so the
arm should move right.
SUBTASK: 1. Move to the French flag.

MOVE REASONING: The arm should move right to get closer to the French
flag.
MOVE: move right

GRIPPER POSITION: [166, 67, 166, 67, 166, 72, 181, 80, 195, 93]
```

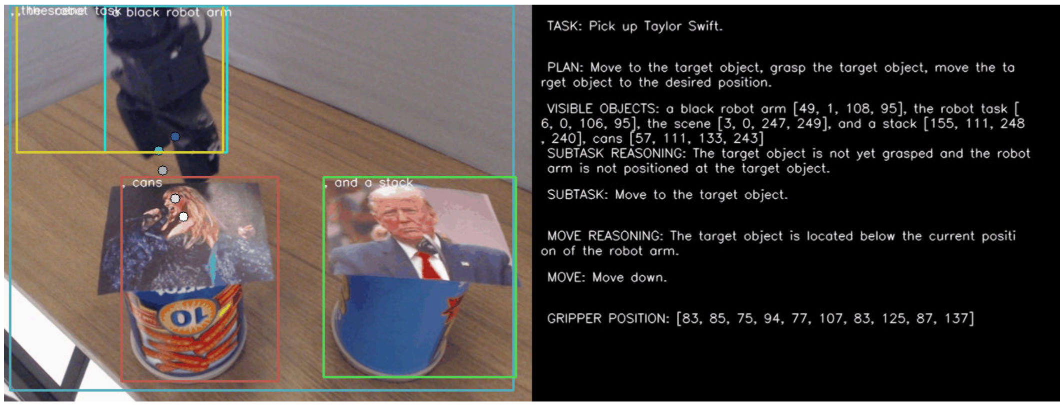


```
TASK: Pick up the panda and bring it to the initial position.

PLAN: Move toward the panda, pick it up, move it to the initial posit
ion.
VISIBLE OBJECTS: a green plate [68, 112, 119, 158], a wooden table [1
6, 33, 238, 245], a wooden table [12, 32, 248, 247]
SUBTASK REASONING: The panda is the target object and it needs to be
picked up.
SUBTASK: Move toward the panda.

MOVE REASONING: The panda is to the left, so the arm needs to move le
ft.
MOVE: Move left.

GRIPPER POSITION: [180, 93, 173, 80, 160, 79, 142, 94, 125, 117]
```



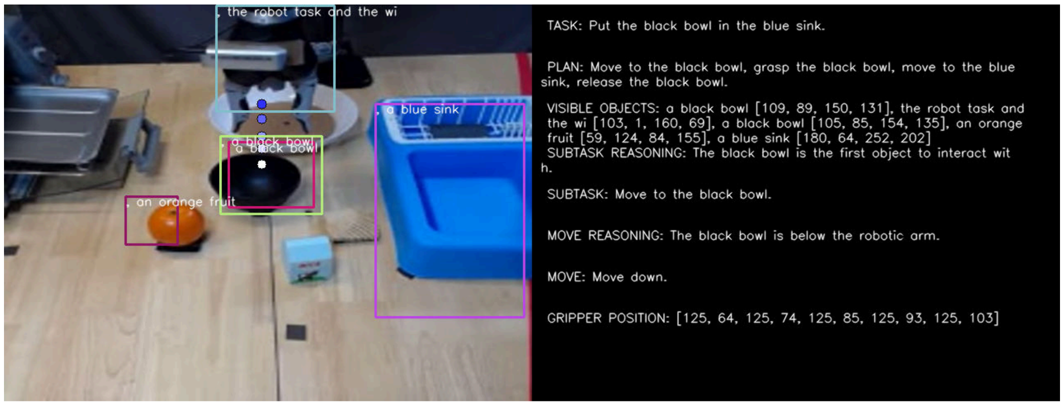
```
TASK: Pick up Taylor Swift.

PLAN: Move to the target object, grasp the target object, move the ta
rget object to the desired position.
VISIBLE OBJECTS: a black robot arm [49, 1, 108, 95], the robot task [
6, 0, 106, 95], the scene [3, 0, 247, 249], and a stack [155, 111, 248
, 240], cans [57, 111, 133, 243]
SUBTASK REASONING: The target object is not yet grasped and the robot
arm is not positioned at the target object.
SUBTASK: Move to the target object.

MOVE REASONING: The target object is located below the current positi
on of the robot arm.
MOVE: Move down.

GRIPPER POSITION: [83, 85, 75, 94, 77, 107, 83, 125, 87, 137]
```

Figure 12: Example reasonings generated for out-of-distribution concepts, such as flags, animals, or celebrities.



TASK: Put the black bowl in the blue sink.

PLAN: Move to the black bowl, grasp the black bowl, move to the blue sink, release the black bowl.

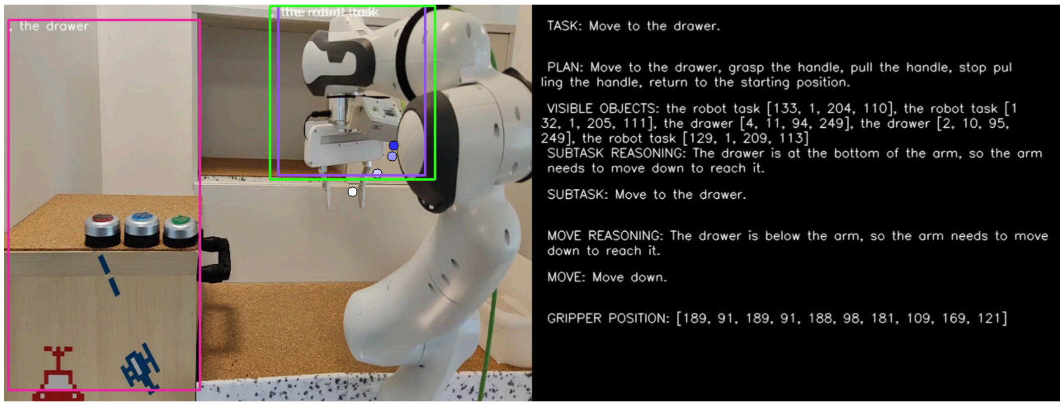
VISIBLE OBJECTS: a black bowl [109, 89, 150, 131], the robot task and the wi [103, 1, 160, 69], a black bowl [105, 85, 154, 135], an orange fruit [59, 124, 84, 155], a blue sink [180, 64, 252, 202]
SUBTASK REASONING: The black bowl is the first object to interact wit h.

SUBTASK: Move to the black bowl.

MOVE REASONING: The black bowl is below the robotic arm.

MOVE: Move down.

GRIPPER POSITION: [125, 64, 125, 74, 125, 85, 125, 93, 125, 103]



TASK: Move to the drawer.

PLAN: Move to the drawer, grasp the handle, pull the handle, stop pul ling the handle, return to the starting position.

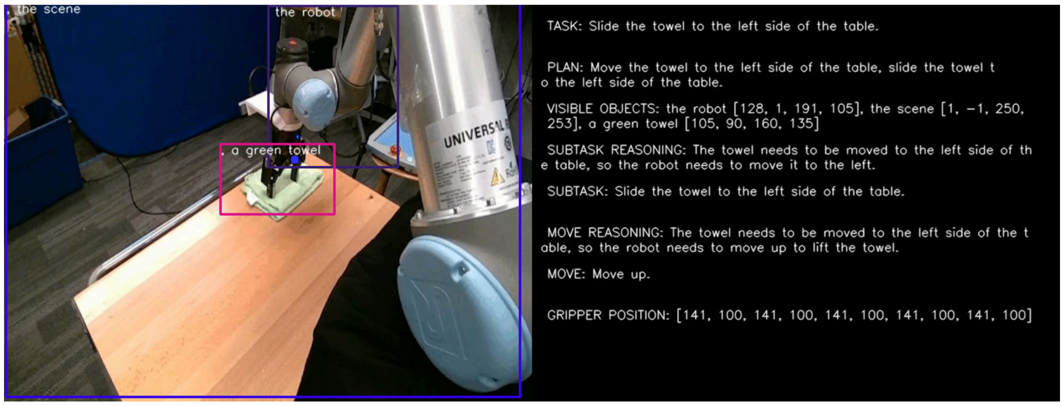
VISIBLE OBJECTS: the robot task [133, 1, 204, 110], the robot task [1 32, 1, 205, 111], the drawer [4, 11, 94, 249], the drawer [2, 10, 95, 249], the robot task [129, 1, 209, 113]
SUBTASK REASONING: The drawer is at the bottom of the arm, so the arm needs to move down to reach it.

SUBTASK: Move to the drawer.

MOVE REASONING: The drawer is below the arm, so the arm needs to move down to reach it.

MOVE: Move down.

GRIPPER POSITION: [189, 91, 189, 91, 188, 98, 181, 109, 169, 121]



TASK: Slide the towel to the left side of the table.

PLAN: Move the towel to the left side of the table, slide the towel t o the left side of the table.

VISIBLE OBJECTS: the robot [128, 1, 191, 105], the scene [1, −1, 250, 253], a green towel [105, 90, 160, 135]

SUBTASK REASONING: The towel needs to be moved to the left side of th e table, so the robot needs to move it to the left.

SUBTASK: Slide the towel to the left side of the table.

MOVE REASONING: The towel needs to be moved to the left side of the t able, so the robot needs to move up to lift the towel.

MOVE: Move up.

GRIPPER POSITION: [141, 100, 141, 100, 141, 100, 141, 100, 141, 100]

Figure 13: Example reasonings generated for out-of-distribution embodiments.

What Matters in Hierarchical Search for Solving Combinatorial Problems?

Anonymous authors
Paper under double-blind review

Abstract

Combinatorial problems, particularly the notorious NP-hard tasks, remain a significant challenge for AI research. A common approach to addressing them combines search with heuristics learned from demonstrations. Recently, hierarchical planning has emerged as a powerful framework in this context, enabling agents to decompose complex problems into manageable subgoals. However, the foundations of this approach, particularly the behavior and limitations of learned heuristics, remain underexplored. Our goal is to advance research in this area and establish a solid conceptual and empirical foundation. Specifically, we identify the following key characteristics, whose presence favors the choice of hierarchical search methods: *hard-to-learn value functions, complex action spaces, presence of dead ends in the environment,* or *training data collected from diverse sources.* Through in-depth empirical analysis, we establish that hierarchical search methods consistently outperform standard search methods across these dimensions, and we formulate insights for future research. On the practical side, we also propose a set of evaluation guidelines to enable meaningful comparisons between methods and reassess the state-of-the-art algorithms.

1 Introduction

The ability to solve discrete tasks that require sophisticated reasoning, particularly those involving NP-hard problems, is essential for advancing AI (Bengio et al., 2021). These include complex problems like theorem proving (Wu et al., 2021; Trinh et al., 2024), constraint satisfaction problem (Achiam et al., 2017), molecule alignment (Needleman & Wunsch, 1970; Smith & Waterman, 1981), social network analysis (Kipf & Welling, 2017), or navigation (LaValle, 2006; Choset et al., 2005). Even driving a car, which typically involves continuous control of steering and speed, requires high-level discrete decision-making, e.g., when to overtake, when to change lanes, or how to navigate through traffic (Kiran et al., 2022).

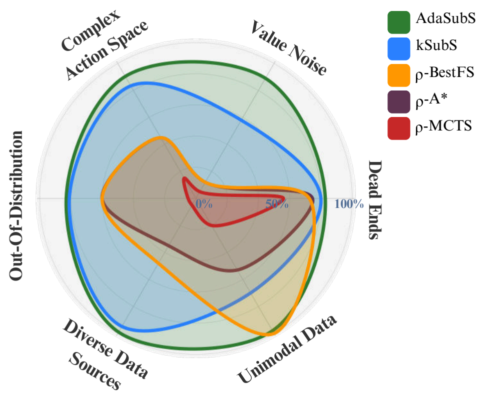


Figure 1: Schematic performance comparison of hierarchical methods (AdaSubS, kSubS) and low-level methods (ρ-BestFS, ρ-A*, ρ-MCTS) across six dimensions: *handling data collected from diverse sources*, *learning from clean unimodal demonstrations*, *avoiding dead ends*, *performance under high-value approximation errors*, *handling complex action space*, and *generalizing to out-of-distribution instances.*

Addressing such tasks, known as combinatorial problems, requires efficient planning strategies due to the vast and complex search spaces involved (Bruck & Goodman, 1987). A widely adopted solution is to learn heuristics from demonstrations, even suboptimal, via imitation learning – a flexible approach that combined with effective planning methods has become a standard paradigm. Recently, hierarchical search emerged as a powerful framework in this context, inspired by how humans plan their actions (Hull, 1932; Fishbach & Dhar, 2005; Kool & Botvinick, 2014). This general-purpose method breaks down a problem into manageable subproblems, or subgoals, making the overall task more tractable. Hierarchical search has been successfully applied to a variety of combinatorial tasks, as evidenced by methods like Subgoal Search (kSubS) (Czechowski et al., 2021), and further advanced by Adaptive Subgoal Search (AdaSubS) (Zawalski et al., 2023), Hierarchical Imitation Planning with Search (HIPS) (Kujanpää et al., 2023a), or HIPS-ε (Kujanpää et al., 2023b).

Our goal is to advance research in hierarchical planning and establish a solid conceptual and empirical foundation. We choose *kSubS* and *AdaSubS* as representative examples of simple yet effective hierarchical methods. These approaches introduce a single layer of subgoals over low-level search algorithms, enabling us to isolate and measure the benefits of hierarchical decomposition in a controlled manner. Through extensive empirical analysis, we identify four key properties of environments and training data whose presence favors the use of hierarchical search methods: *hard-to-learn value functions, complex action spaces, presence of dead ends in the environment*, or *data collected from diverse sources*. We further analyze these findings from a general perspective and prove theorems that explain the most unexpected results. Our findings offer a clearer understanding of when hierarchical approaches should be preferred over low-level methods.

In summary, our contributions are as follows:

- We conduct a detailed empirical comparison of hierarchical and low-level search methods, both guided by learned heuristics, across a range of combinatorial tasks, revealing consistent trends in performance and robustness.
- We provide a theoretical analysis that explains key empirical findings and extends to a broader class of hierarchical methods, offering insights that generalize beyond the specific algorithms evaluated.
- We identify key tasks and data characteristics that favor hierarchical approaches and propose evaluation guidelines to support more consistent and transparent benchmarking in future research.

2 Related Work

Solving Decision-Making Problems Decision-making problems are often framed as Markov Decision Processes (MDPs) (Sutton et al., 1999), which can be solved using Reinforcement Learning (RL) algorithms like PPO (Schulman et al., 2017) or DQN (Mnih et al., 2015). These methods learn policies through interaction with the environment. An alternative to learning from trial and error is Imitation Learning (IL), training models directly from offline demonstrations. The availability of large-scale datasets (Walke et al., 2023; Collaboration et al., 2023; Grauman et al., 2022; Dosovitskiy et al., 2017), make it applicable to the most complex domains like robotics (Mandlekar et al., 2018; Edmonds et al., 2017; Kim et al., 2024), autonomous driving (Kelly et al., 2019; Li et al., 2022; Zhang & Cho, 2017), and physics-based control (Kim et al., 2020; Fickinger et al., 2022). Key foundational methods such as Behavioral Cloning (BC) (Sutton & Barto, 1998), Inverse Reinforcement Learning (IRL) (Baker et al., 2009), or DAgger (Ross et al., 2011) have been instrumental in advancing IL for complex environments where direct exploration is less practical. In this work, we use IL to train components for the search methods, such as the policy and value function, which is a widely adopted approach (Czechowski et al., 2021; Zawalski et al., 2023; Takano, 2023).

Subgoal Methods Hierarchical Reinforcement Learning methods tackle complex decision-making tasks by breaking them into subgoals. HIRO (Nachum et al., 2018) reuses past data by goal relabeling. HAC (Levy et al., 2019) builds a multi-layer hierarchy of policies trained with hindsight. Hierarchical Diffuser (Chen et al., 2024) learns to predict future states with diffusion models. Graph-based methods, such as SoRB (Eysenbach et al., 2019) or DHRL (Lee et al., 2022) build a high-level graph of states, which then allow for efficient shortest path finding. GCP (Pertsch et al., 2020) learns to predict middle states between two given observations. Algorithms such as HPG (Ghavamzadeh & Mahadevan, 2003) or H-DDPG (Yang et al., 2018) extend the classical RL algorithms to the hierarchical setting.

In the area of combinatorial problems, there has been growing interest in applying HRL techniques. kSubS (Czechowski et al., 2021) introduces a hierarchical search algorithm that iteratively generates subgoals to construct a search tree. Building on this, AdaSubS (Zawalski et al., 2023) incorporates multiple subgoal generators, each trained to predict subgoals at different distances from the target, allowing for dynamic adaptation of the planning horizon based on problem complexity. HIPS (Kujanpää et al., 2023a) and HIPS-ε (Kujanpää et al., 2023b) perform search using subgoals generated by VQ-VAE models (van den Oord et al., 2017).

Low-level Search Algorithms Traditional search algorithms like Best-First Search (BestFS), A* (Cormen et al., 2009; Russell & Norvig, 2009), and Monte Carlo Tree Search (MCTS) (Veness et al., 2009; James et al., 2017) have long been the foundation for solving complex decision-making problems. Recent advancements have improved these methods by integrating neural network-based heuristics, improving their efficiency in large search spaces (Silver et al., 2018; Yonetani et al., 2021). A variant of ρ-BestFS used in (Czechowski et al., 2021; Zawalski et al., 2023), leverage heuristics learned through behavioral cloning to guide search. More recent algorithms, like PHS (Orseau & Lelis, 2021) or LevinTS (Orseau et al., 2023), combine policy-driven and value-based approaches, offering both theoretical guarantees and strong empirical performance. Additionally, PDDL planners (Haslum et al., 2019) solve decision-making problems by using predefined action models and goals, with domain-independent planners offering broad applicability, while domain-specific ones achieve higher performance in specialized tasks.

Empirical Studies on Algorithmic Performance Our work aligns with recent empirical studies that investigate the conditions under which various algorithmic approaches excel. For instance, Andrychowicz et al. (2020) investigate how specific design choices influence the performance of PPO, while other research compares offline reinforcement learning with behavioral cloning (Kumar et al., 2022) or explores design choices for language-conditioned robotic imitation learning (Mees et al., 2022). In this paper, we focus on hierarchical search in combinatorial problems, specifically studying the conditions where hierarchical methods outperform low-level planners. To the best of our knowledge, this is the first systematic study of the relationship between hierarchical and low-level search in this context.

3 Combinatorial Environments

Our study targets solving combinatorial environments – domains with discrete, compact state representations corresponding to exponentially large configuration spaces, which makes them highly challenging to solve. This class includes several NP-hard problems, such as the Traveling Salesman Problem (Applegate et al., 2006), the Rubik's Cube (Singmaster, 1981), Sokoban (Culberson, 1997), or solving non-linear inequalities (Sahni, 1974). In our study, we specifically focus on goal-reaching tasks. To efficiently solve combinatorial problems an algorithm should have the following desirable properties:

In combinatorial environments, each problem instance is typically entirely distinct from others, making it unrealistic to assume that offline data provides comprehensive state space coverage. This is especially critical in problems like the Rubik's Cube, where even with a vast training dataset, any new state will be entirely different from those previously encountered. Some approaches rely on sufficient state space coverage, but in many combinatorial problems, this assumption is impractical.

1. **Learning from offline data.** Since combinatorial environments are characterized by a large space of possible configurations, learning without priors or handcrafted dense rewards is infeasible due to the challenge of exploration[1]. To address this, a canonical solution is to leverage offline data, even suboptimal. Other possible approaches, such as clever reward shaping, usually require significant domain knowledge.

2. **Combinatorial space abstraction.** In combinatorial environments, each problem instance is typically entirely distinct from others. Hence, it is unrealistic to assume a comprehensive state space

[1]For instance, we tested PPO (Schulman et al., 2017) on the Rubik's Cube, but, unsurprisingly, it failed to make any progress due to never reaching the goal in the haystack of 4.3×10^{19} states, hence never observing a positive reward.

coverage by training data or repeated visits to nearby states, an assumption that some approaches implicitly rely on.

3. **Search.** Methods that don't use search and follow a single action trajectory are inherently limited by computational complexity, since they can perform only a constant number of operations before choosing an action. Solving NP-hard problems within a fixed computation budget is computationally infeasible (Bruck & Goodman, 1987).

Many hierarchical methods have not been designed for combinatorial problems, so they fail to meet the listed conditions and cannot be expected to be efficient in these applications. For instance, Chen et al. (2024); Yang et al. (2018) require continuous state or action space, Ghavamzadeh & Mahadevan (2003) learns only from online interactions, Eysenbach et al. (2019); Huang et al. (2019); Lee et al. (2022) assume a good coverage of the whole state space, and Nachum et al. (2018); Levy et al. (2019) do not use planning to determine actions.

4 Subgoal Methods

Subgoal methods, or hierarchical methods, are a family of algorithms designed to solve complex decision-making tasks by breaking down the overall objective into smaller, more manageable subgoals (Sutton et al., 1999). Instead of searching for a sequence of low-level actions that directly lead from the initial state to the goal, the agent first identifies high-level intermediate targets – subgoals – that guide the trajectory toward the final goal. The use of subgoals is widely considered as a method that scales better to longer horizons (Chen et al., 2024; Lee et al., 2022), mitigates errors in value approximations (Czechowski et al., 2021), and reduces overall complexity by decomposing the problem into smaller subproblems (Sutton et al., 1999; Zawalski et al., 2023). The process of searching involves the following components:

- **Subgoal generator** that, given a state within the search tree, outputs a set of subgoals. For instance, a subgoal may be a future state (Czechowski et al., 2021; Zawalski et al., 2023) or a class of desired outcomes (Jiang et al., 2019; Panov & Skrynnik, 2018). See Figure 16 for example subgoals. The subgoal generator can be implemented using models such as transformers with beam search (Czechowski et al., 2021; Zawalski et al., 2023), VQ-VAE (Kujanpää et al., 2023a), or other generative architectures. The generator is used by the planner to construct a search tree of subgoals.
- **Low-level policy** that determines a path of low-level actions between subgoals. For instance, it may be a trained goal-reaching policy (Czechowski et al., 2021; Zawalski et al., 2023), a local search (Czechowski et al., 2021; Kujanpää et al., 2023a), or a stored path from previous episodes (Eysenbach et al., 2019; Lee et al., 2022).
- **Planner** that determines the order in which subgoals are generated. Standard planning algorithms like BestFS (Czechowski et al., 2021), PHS (Kujanpää et al., 2023a), or their modified forms (Zawalski et al., 2023), are typically used.
- **Value function** that estimates the distance between the given state and the goal state. The planner uses this information to select the next node to expand with the subgoal generator. In some works it is also called *heuristic value*. In our study, we focus on value functions learned from demonstrations, but in general, values learned through RL or even scripted heuristics can be used in search.

In our experiments, we use kSubS (Czechowski et al., 2021) and AdaSubS (Zawalski et al., 2023) as subgoal methods well-suited for combinatorial problems, as they satisfy the conditions formulated in Section 3. We also experimented with HIPS and HIPS-ε (Kujanpää et al., 2023a;b), but these methods generally fail to solve the problems within a reasonable computational budget. Therefore, their results are omitted from the main text and discussed in Appendix I.

We compare the performance of the selected subgoal approaches against three popular low-level methods: BestFS, A*, and MCTS. To ensure a fair comparison and improve efficiency, we augment these algorithms by using a trained policy to select the top actions before each node expansion. We refer to them as ρ-BestFS, ρ-A*, and ρ-MCTS. A detailed description, analysis, and pseudocode for each of these algorithms can be found in Appendix F. See also Appendix H for diagrams explaining different search methods.

4.1 Training Components

In our experiments, the models for both subgoal methods and low-level searches were trained using imitation learning, following standard practice (Nair et al., 2018; Czechowski et al., 2021). Specifically, we collected a dataset of approximately 500 000 trajectories for each environment. Trajectories are sequences of consecutive states and actions leading to the goal state. We used various methods of dataset collection, like hand-crafted algorithms, trained policies, reversed random shuffles, and others, which let us to study the influence of training data characteristics on the performance of search methods.

To ensure a fair comparison, all methods shared common components whenever applicable (e.g., each method uses the same value function). This allows us to focus on the differences between the search algorithms, rather than heuristic biases. No additional heuristics were used, ensuring that performance differences arise solely from the algorithmic approaches.

More details on training the components, including specific objectives, are provided in Appendix D.

4.2 Performance Metrics

Our primary performance metric is the *success rate*, defined as the percentage of problem instances solved within a given *complete search budget*. The complete search budget is the total number of visited states in the search tree. In particular, for subgoal methods, the budget includes both the generated subgoals and the states visited by the low-level policy used to connect these subgoals.

By accounting for the total number of visited states, this metric provides a unified and fair comparison of search efficiency across different methods. We argue that reporting only the number of visited subgoal nodes would unfairly favor subgoal methods (see Appendix I for details).

While *wall-clock time* could serve as an alternative budget metric, it suffers from high variance and is sensitive to hardware, making it difficult to reproduce results. Nevertheless, we report wall-clock measurements to validate their correlation with the complete search budget. Detailed discussion of that metric is provided in Appendix G.

5 Analysis

We investigate how environmental properties and training data influence the performance of hierarchical methods compared to low-level search approaches in combinatorial tasks. While previous works (Czechowski et al., 2021; Zawalski et al., 2023; Kujanpää et al., 2023a;b) show a considerable advantage of hierarchical methods, our experiments reveal that this advantage is not consistent across all scenarios (see Figures 4 or 5 for specific examples). Specifically, we answer the following research questions:

Q1. Is hierarchical search more effective than low-level search for solving combinatorial problems?

Q2. What environmental properties and characteristics of the training data amplify performance differences? When hierarchical search should be preferred over low-level search?

Q3. What pitfalls should be avoided when interpreting experimental results?

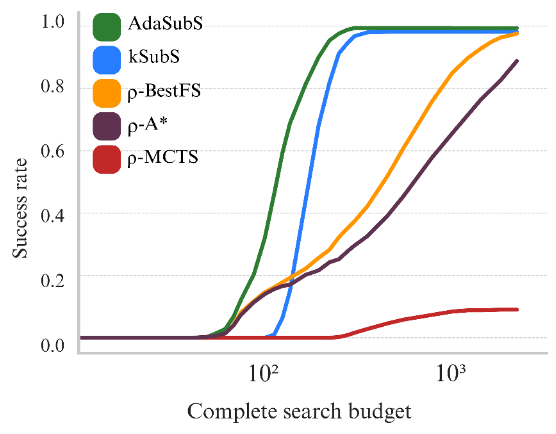
To address these questions, we conducted a wide range of experiments comparing subgoal and low-level search algorithms across a variety of combinatorial tasks. Below, in each subsection we summarize the key findings that reveal the most significant factors affecting performance, followed by a brief discussion. For each finding, we link it to the relevant research questions. The extended analysis of these factors can be found in Appendix B.

We present our findings using the *Rubik's Cube, Sokoban, N-Puzzle*, and *Inequality Theorem Proving* (INT) (Wu et al., 2021) environments[2]. These classical benchmarks are widely used in planning research (McAleer et al., 2019; Czechowski et al., 2021) and are known to be NP-hard (Demaine et al., 2018; Culberson, 1997; Ratner & Warmuth, 1986). Since different algorithms exhibit significant performance variations depending on the problem structure, we evaluate them in a range of environments to ensure the robustness of our findings. Detailed descriptions of these environments can be found in Appendix A.

All methods in our study were trained using imitation learning. In particular, all algorithms share the same value function, as stated in Section 4.1. To ensure fair comparisons, we measured complete search budgets, in contrast to counting only high-level search nodes, to avoid giving any unfair advantage to subgoal methods, as discussed in Section 4.2 (which contributes to the research question Q3). We tuned hyperparameters of each method separately for each experiment to ensure optimal performance.

5.1 Subgoal Methods are Robust to Diverse Sources of Data

Achieving superhuman performance in complex tasks often involves large-scale datasets of demonstrations obtained from agents with varying skill levels and strategies (Silver et al., 2016). By training models on data collected from a variety of solvers and testing them in the Rubik's Cube and N-Puzzle environments, we show that the variability in training data has a significant impact on the performance of search algorithms. Our training data included algorithmic solvers, computational solvers, and random shuffles, as detailed in Appendix B.1.



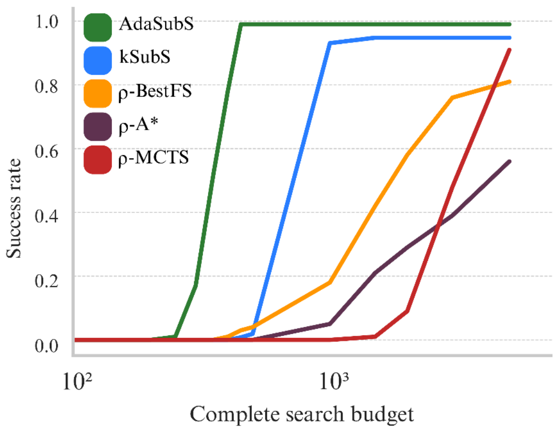


Figure 2: Solving the Rubik's Cube. Components are trained on data from 4 different solvers.

Figure 3: Solving the N-Puzzle. Components are trained on data from 2 different solvers.

As shown in Figures 2-3, subgoal methods consistently outperform low-level methods by a wide margin (Q1). However, when the training dataset is limited to a single source of demonstrations – whether the demonstrations are long and structured or short and direct – this performance gap disappears (see Figures 4-6). Notably, subgoal methods, particularly AdaSubS, maintain stable performance across all training setups, while low-level methods are highly sensitive to the characteristics of the training data.

To explain those results, we found that value functions trained on diverse data often fail to assign consistently low values to the initial states of tasks. When demonstrations differ significantly in their length or execution style, the value function learns this variation, leading to inconsistent value predictions. Hierarchical methods can overcome this issue by relying on subgoals. Subgoals enable the agent to make long steps toward the solution, effectively bypassing regions of the state space where the value function is inconsistent or noisy, as it does not need to assess every small step along the way (this property is further studied in Section 5.2). In contrast, low-level methods operate on a finer, step-by-step level, executing small, atomic actions.

[2]We note that classical environments have domain-specific solvers that achieve high performance by relying on expert knowledge. However, our goal is to compare general-purpose search methods that require no domain knowledge.

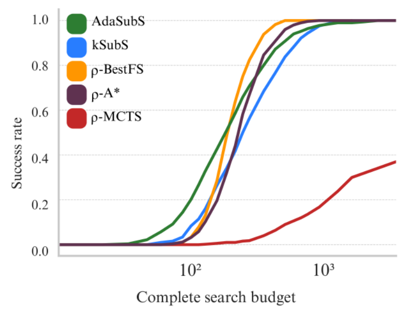


Figure 4: Solving the Rubik's Cube. Components are trained on reversed random shuffles.

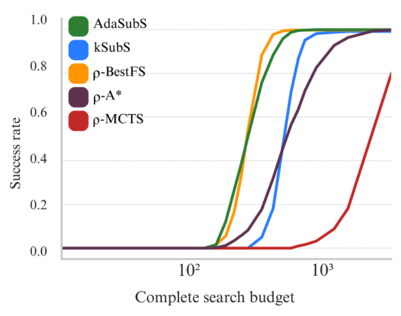


Figure 5: Solving the Rubik's Cube. Components are trained on the *Beginner* algorithmic solver.

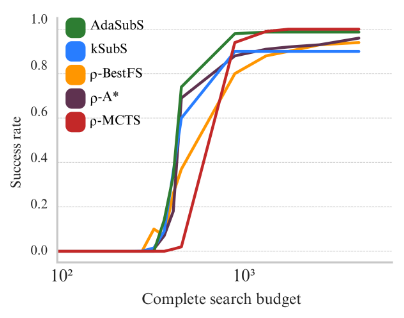


Figure 6: Solving N-Puzzle. Components are trained on an algorithmic solver.

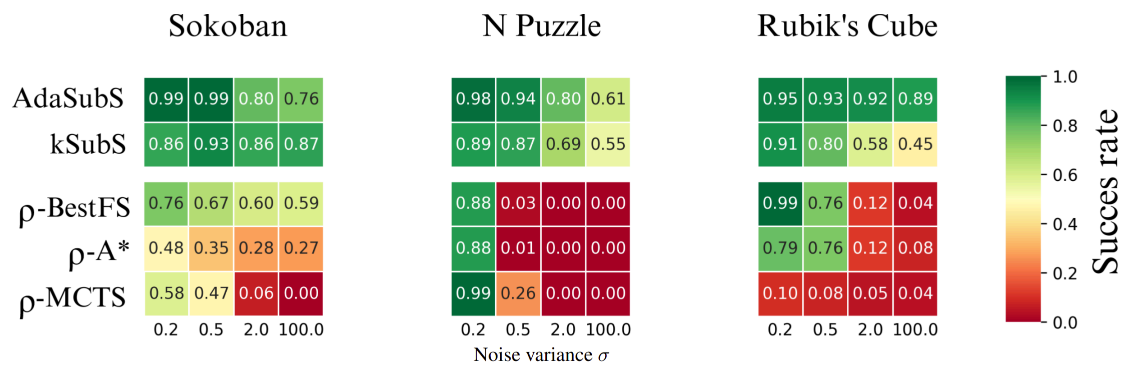
This makes them more sensitive to the variability in the value function because they must evaluate each intermediate state on the way.

More detailed analysis of the experiments involving diverse data sources is provided in Appendix B.1.

Takeaway *Subgoal methods successfully leverage diverse demonstrations (Q2), while low-level search performs better when trained on homogeneous trajectories (Q2).*

5.2 Subgoal Methods are Value Noise Filters

We found that the classical search algorithms are highly sensitive to the quality of the value function. To show this in a controlled setting, we added Gaussian noise to the value estimates and observed how different noise levels impacted the success rate of solving tasks.



Sokoban	0.2	0.5	2.0	100.0
AdaSubS	0.99	0.99	0.80	0.76
kSubS	0.86	0.93	0.86	0.87
ρ-BestFS	0.76	0.67	0.60	0.59
ρ-A*	0.48	0.35	0.28	0.27
ρ-MCTS	0.58	0.47	0.06	0.00

N Puzzle	0.2	0.5	2.0	100.0
AdaSubS	0.98	0.94	0.80	0.61
kSubS	0.89	0.87	0.69	0.55
ρ-BestFS	0.88	0.03	0.00	0.00
ρ-A*	0.88	0.01	0.00	0.00
ρ-MCTS	0.99	0.26	0.00	0.00

Rubik's Cube	0.2	0.5	2.0	100.0
AdaSubS	0.95	0.93	0.92	0.89
kSubS	0.91	0.80	0.58	0.45
ρ-BestFS	0.99	0.76	0.12	0.04
ρ-A*	0.79	0.76	0.12	0.08
ρ-MCTS	0.10	0.08	0.05	0.04

Figure 7: Success rate of low-level and subgoal methods as the approximation errors of the value function increase. $\sigma = 100$ results in completely random value estimates.

While ρ-BestFS is able to solve nearly all instances under ideal conditions, its performance significantly declines as value function errors increase, even to 0% (see Figure 7). ρ-A* and ρ-MCTS behave similarly. In contrast, the subgoal methods show remarkable resilience. Particularly AdaSubS, which maintains nearly unchanged success rate, despite high value errors (Q2).

These results align with our findings in Section 5.1, where using diverse training data naturally introduced value estimation errors. As observed by Zawalski et al. (2023), the search process of subgoal methods is guided by subgoal generators, which reduces reliance on the value function. Subgoal generators and the conditional policies connecting subgoals are not directly influenced by the value approximation errors. The value function is used only in high-level nodes, which represent only a fraction of the search tree.

In hierarchical methods, the distance between high-level nodes spans multiple steps, increasing the likelihood that value estimates for subsequent high-level nodes along the solution path will be monotonic (see Figure 8 for an illustrative example), which makes planning more efficient. This supports the claim by Czechowski et al. (2021) that subgoals effectively mitigate the impact of value noise. To further ground that result, we prove the following theorem:

Theorem 1 (Search advancement formula)**.** *Let $g_k : S \to \mathcal{P}(S)$ be a stochastic k-subgoal generator that, given a state $s \in S$ samples a set of b subgoals $\{s_i\}$ such that the distances $d(s_i, s)$ are independent, uniformly distributed in the interval $[-k; k]$. Let $V : S \to \mathbb{R}$ be a value function with approximation error uniformly distributed in the interval $[-\sigma; \sigma]$.*

Then, after n iterations of search, the expected total progress toward the goal is:

$$\mathbb{E}_{Adv} = \frac{nb}{4\sigma k} \int_{-k}^{k} x \left(\int_{-\sigma}^{\sigma} \tilde{u}(x+h)^{b-1} \mathrm{d}h \right) \mathrm{d}x, \tag{1}$$

where $\tilde{u}(x)$ is CDF of the sum of two uniform variables $U(-k,k)+U(-\sigma,\sigma)$. Additionally, if we approximate that sum as $U(-k-\sigma, k+\sigma)$, we get

$$\mathbb{E}_{Adv} \approx \frac{n\left((k+\sigma)^b(bk^2+bk\sigma-2k\sigma-2\sigma^2)+\sigma^b(2k\sigma+bk\sigma+2\sigma^2)-k^b(bk^2)\right)}{(b+1)(b+2)k\sigma(k+\sigma)^{b-1}} \tag{2}$$

Proof. See Appendix K for the proof. □

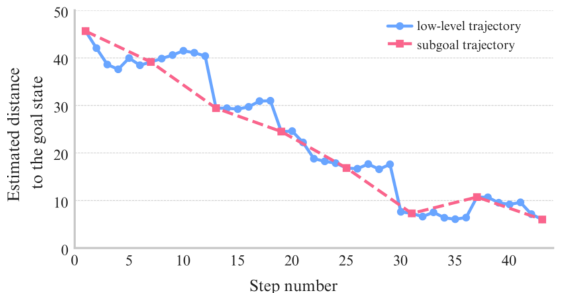


Figure 8: Value estimates along a solving trajectory generated by ρ-BestFS. Even small approximation errors cause non-decreasing values, slowing down the search. In contrast, the subgoal path mitigates these errors, leading to mostly monotonic values along the trajectory.

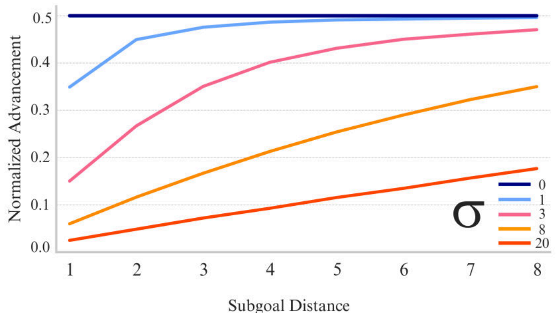


Figure 9: Normalized advancement $\mathbb{E}_{Adv}/k$ for a single search iteration, according to Theorem 1. The value for each subgoal is divided by its length to represent the advancement per atomic action for easier comparison.

Theorem 1 quantifies the expected progress of the search at each step, with Equation 1 giving an exact formula and Equation 2 providing a useful approximation. To compare subgoal methods with low-level methods in theory, under different levels of value approximation error, we model low-level search by setting $k = 1$, which represents a single action. Figure 9 shows the expected search progress with a branching factor of $b = 3$, normalized by the number of actions leading to a subgoal.

When value estimates are perfect (i.e., $\sigma = 0$), both subgoal and low-level searches perform similarly. However, as value approximation errors increase, subgoal methods become significantly more resilient. At high noise levels ($\sigma = 20$), single-step searches make very little progress, advancing only 0.025 per action. In contrast, subgoals of length 8 achieve much greater progress – 1.4 for the entire subgoal, which is 0.175 per action. This 7-fold increase in theoretical efficiency explains why subgoal methods outperform low-level methods in our experiments.

Further analysis of these experiments can be found in Appendix B.2.

Takeaway *Subgoal methods successfully handle value approximation errors. Thus, they should be used when estimating the value is hard, for instance, when learning from diverse and suboptimal demonstrations (Q2).*

5.3 Subgoal Methods Handle Complex Action Spaces

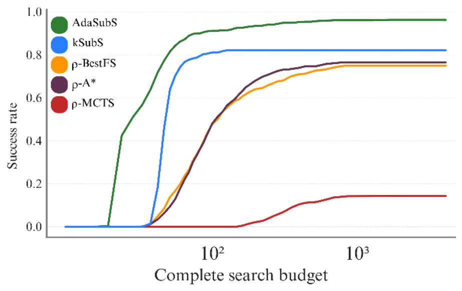


Figure 10: Solving INT. Components are trained on randomly generated proofs.

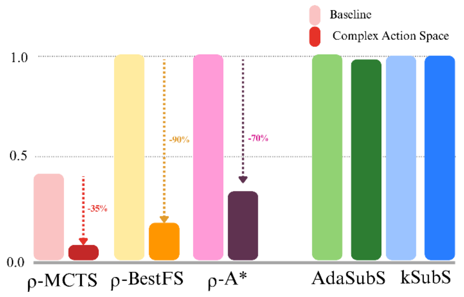


Figure 11: Solving the Rubik's cube with expanded action space, compared with the standard setup. Components are trained on reverse random shuffles.

In environments with large action spaces, search methods often struggle due to the exponential increase in the number of choices (Sutton & Barto, 1998). As shown in Figure 10, subgoal methods demonstrate a clear advantage over low-level search methods in the INT environment (Wu et al., 2021), a benchmark on proving mathematical inequalities (Q1). The INT environment is particularly challenging because of its highly complex observation and action spaces, making it the most difficult benchmark among those used in (Czechowski et al., 2021; Zawalski et al., 2023; Kujanpää et al., 2023a;b).

Given a complex action space, each node expansion in low-level methods involves executing many similar actions, limiting their ability to efficiently search through the space. In contrast, subgoal methods compute actions only to connect subgoals, which is a much simpler task. This targeted approach reduces the negative impact of a large action space, allowing subgoal methods to maintain their efficiency even as the action space grows (Q2).

To confirm this explanation, we conducted experiments on a modified version of the Rubik's Cube, where the action space was artificially inflated by giving the agent access to 100 copies of each action. As shown in Figure 11, this simple modification drastically reduces the success rates of all low-level methods, even below 35%. In contrast, subgoal methods remain largely unaffected, performing similarly to the standard setup. We can explain that result with the following theorem:

Theorem 2 (Densification of the action space)**.** *Fix any state s from the state space S. Let $f : A \to [0,1]$ be the action distribution induced by the data-collecting policy for the state s. Assume that f is continuous and has a unique maximum.*

For clarity, assume $A = [0,1]$. Consider a sequence of increasingly dense discrete action spaces $A_n := \{i/n\}_{i=0}^{n} \subset A$. Let $\rho_n : S \times A_n \to [0,1]$ be a family of policies that learn the distribution $f|_{A_n}$ over actions, with uniform approximation error $U(-E, E)$, where $E \in \mathbb{R}_+$. Let r_n be the range of the top K actions according to the probabilities estimated by ρ_n. Then

$$\lim_{n\to\infty} \mathbb{E}[r_n] = 0.$$

Proof. See Appendix L □

Intuitively, this theorem states that as the action space become more dense and complex, the actions sampled for search become increasingly less diverse, which strongly impedes successful planning. Note that this analysis is strictly more general than the last experiment, where we simply copied the available actions. Further analysis of the experiments involving large action spaces is provided in Appendix B.3.

Takeaway *When facing a problem with a complex action space, subgoal methods should outperform low-level search (Q2).*

5.4 Subgoal Methods Avoid Dead Ends

Once an agent encounters a dead end, reaching the goal becomes impossible, leading to wasted computational effort. Our results, presented in Figure 5.4, show that subgoal methods tend to enter dead ends less often than low-level methods. Using longer subgoals improves the ability to bypass those areas.

Among low-level methods, ρ-A* performs the best at minimizing dead ends rate, as its node selection regularizes values by depth in the search tree, preventing it from over-committing to dead ends. However, even ρ-A* is outperformed by subgoal methods, which rely on greedy value estimates and subgoals.

Deciding whether a state is a dead end can be NP-hard. Hence, it is much harder for the value function to penalize dead ends compared to the policy, which only ranks the available actions and does not have to identify dead ends (Feng et al., 2022). Furthermore, demonstrations used for imitation learning lead to the goal state, hence they contain no dead ends. Therefore the value function trained this way is never directly instructed to penalize dead ends. At the same time, during training of the policy the actions leading to dead ends are never reinforced. Our experiments show that hierarchical search relies much less on the value guidance compared to low-level search (Section 5.2), which further supports our conclusions. For a more detailed analysis, see Appendix B.4.

Search algorithm	**Dead ends rate**
ρ-MCTS	22.0%
ρ-BestFS	18.5%
ρ-A*	13.7%
kSubS (4 steps)	12.7%
kSubS (8 steps)	10.0%
AdaSubS	8.86%

Figure 12: Fraction of dead ends encountered during search between hierarchical and low-level methods in Sokoban.

Takeaway *Subgoal Methods Are More Effective at Avoiding Dead Ends Compared to Low-Level Search (Q2).*

5.5 Subgoal Methods Generalize Out-Of-Distribution

Planners that can generalize to out-of-distribution (OOD) instances are essential for robust decision-making (Kirk et al., 2023; Shen et al., 2021). We tested two types of generalization in the Sokoban environment: by significantly changing the layout of the board and by using extremely difficult boards from the DeepMind dataset (Guez et al., 2018) (see Figure 13 for examples).

In both cases, subgoal methods show better performance than low-level methods, with the gap increasing as the distribution shift become more visible (see Figures 14-15). However, we found that kSubS, when using twice longer subgoals, collapses in OOD evaluations, despite outperforming ρ-BestFS and other low-level methods on in-distribution tasks. As the subgoal distance increases, predicting the distant future becomes more challenging, making it less likely for the generated subgoals to be valid and reachable, especially in OOD tasks. In contrast, low-level methods avoid this issue, as selecting an action from a limited set always results in a valid move. Thus, while subgoal methods can be effective in OOD scenarios, excessively long subgoals can degrade performance (Q2).

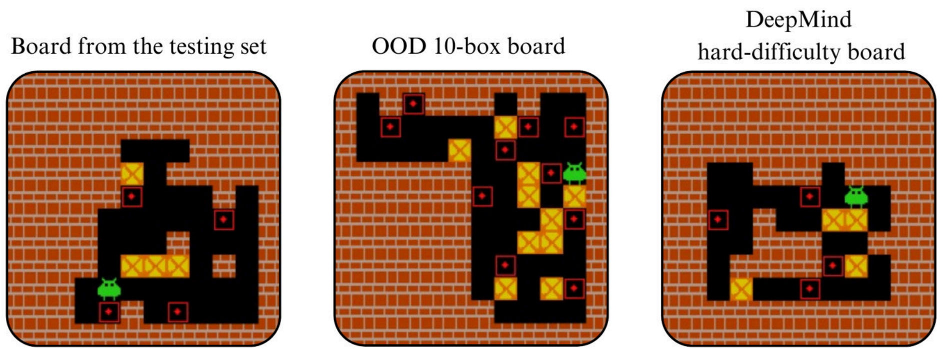


Figure 13: Examples of Sokoban boards used in OOD experiments

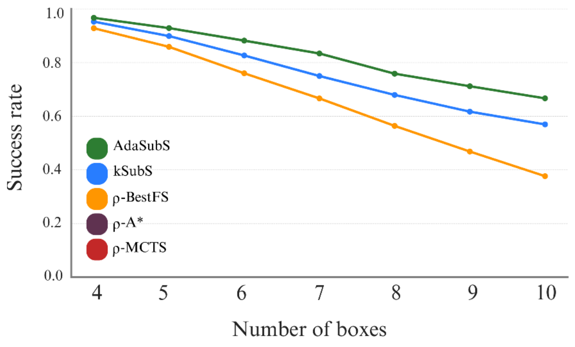


Figure 14: Averaged OOD results on Sokoban boards with OOD layouts. These instances were generated by systematically varying all parameters of the instance generator.

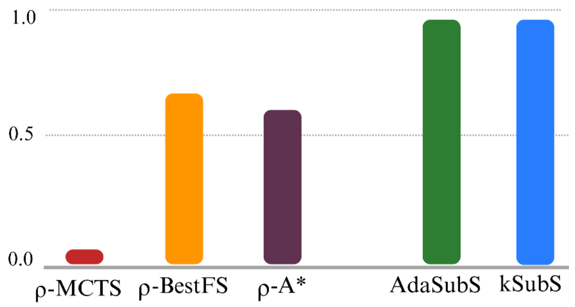


Figure 15: Performance on DeepMind extra hard boards.

When evaluated on extremely challenging instances (see Figure 1) introduced by (Guez et al., 2018), all methods required a significantly higher search budget but maintained the same performance order as in the previous experiment (Q1). Solving these instances requires more advanced strategies than those learned during training. Subgoal methods are better equipped to handle this increased complexity because selecting subgoals is closely related to choosing a broader strategy because of their longer horizon. In contrast, low-level methods must assess each individual action, which limits their ability to foresee the long-term consequences of their choices.

Takeaway *Subgoal methods can scale better than low-level methods on OOD instances, provided the subgoals are not too long (Q2).*

6 Discussion of the Results and Future Directions

We identified several features that facilitate the performance of subgoal methods; however, this list is not exhaustive. Since our study is primarily empirical, it is difficult to make truly universal claims. This highlights the need for further research, including the analysis of additional subgoal-based and low-level algorithms, as well as a broader range of environments. Although most of our conclusions were confirmed across multiple settings, extending the evaluation to more domains would further strengthen their validity.

Empirical results suggest general trends, but Theorems 1-2 offer theoretical support for key findings. These theorems, which apply to the general class of hierarchical methods described in Section 4, reinforce the broader relevance of our results. Additional findings from our study may also inspire further theoretical investigation.

We advocate for the use of a complete search budget as a more meaningful metric than alternatives such as the number of high-level nodes or wall-clock time. Nevertheless, for completeness, we report runtime comparisons in Appendix G. Developing a flexible, low-variance, and reproducible evaluation framework based on wall-clock time remains an important direction for future work.

Our study has broader implications for other complex domains. For example, advancements in robotics often face significant challenges due to limited data, leading many methods to rely on collective datasets like Open X-Embodiment (Collaboration et al., 2023). As shown in our experiments, hierarchical search methods benefit substantially from training on diverse expert data (Section 5.1). Furthermore, the data bottleneck increases the need for the models to generalize to out-of-distribution scenes and tasks, which is also an advantage of hierarchical methods (Section 5.5). Finally, an essential aspect of robotics involves preventing the robot from becoming stuck or losing the manipulated object, events that can be seen as dead-end scenarios (Section 5.4). Successful applications of hierarchical methods in robotics include models such as SuSIE (Black et al., 2024) and HIQL (Park et al., 2023).

Additionally, our experiments indicate that hierarchical methods scale well in long-horizon tasks, as evidenced by their performance in the N-Puzzle and the Rubik's Cube (using Beginner demonstrations), where the average sequence of steps often exceeds 200. Interestingly, while low-level methods can still perform well in these scenarios, we observed that they tend to be much more sensitive to hyperparameter tuning.

It is important to note that we do not claim hierarchical methods are universally superior to low-level approaches in all complex domains. Instead, the properties highlighted in our analysis suggest cases where they should be considered.

7 Conclusions

We conducted a thorough comparison of hierarchical and low-level search methods for combinatorial tasks. Our experiments provides empirical and some theoretical evidence that hierarchical approaches should be preferred in environments where value estimation is challenging and learned estimates face significant uncertainty, particularly when learning from diverse suboptimal data. Furthermore, subgoal methods demonstrate better scalability in complex action spaces and are more effective at avoiding dead ends than low-level methods. Thus, in environments characterized by those properties, it is advisable to consider subgoal methods as an alternative to low-level search.

While we use Subgoal Search as a representative hierarchical method in our experiments, our analysis is framed from the broader perspective of hierarchical methods, as introduced in Section 4. Notably, Theorems 1 and 2, as well as the general properties illustrated in Figures 8 and 11, apply to a wide class of subgoal methods, not just to the specific implementations used for our experiments.

Based on our results, we propose guidelines for future research in this area. According to our experiments, the best-performing low-level search was usually ρ-BestFS with a confidence threshold (see Appendix F). Although it is rather sensitive to the threshold value, which has to be optimized for each domain separately, we advocate using this simple method as a standard baseline for further research in hierarchical search. Our guidelines are comprehensively discussed in Appendix J.

Additionally, we identified easy-to-overlook mistakes in reporting the results that may lead to misleading conclusions. Most importantly, the reported *complete search budget* of hierarchical methods must include all the visited states and not only the high-level nodes as used in some prior works.

8 Reproducibility Statement

The code used to run all our experiments is available at `https://github.com/subgoalsearchmatters/what-matters-in-hierarchical-search`. We also link there datasets used for training our models. Hence, all our results are fully reproducible.

References

Joshua Achiam, David Held, Aviv Tamar, and Pieter Abbeel. Constrained policy optimization. In Doina Precup and Yee Whye Teh (eds.), *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pp. 22–31. PMLR, 2017. URL `http://proceedings.mlr.press/v70/achiam17a.html`.

Marcin Andrychowicz, Anton Raichuk, Piotr Stanczyk, Manu Orsini, Sertan Girgin, Raphaël Marinier, Léonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, Sylvain Gelly, and Olivier Bachem. What matters in on-policy reinforcement learning? A large-scale empirical study. *CoRR*, abs/2006.05990, 2020. URL `https://arxiv.org/abs/2006.05990`.

David L Applegate, Robert E Bixby, Václav Chvátal, and William J Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

Chris L Baker, Rebecca Saxe, and Joshua B Tenenbaum. Action understanding as inverse planning. *Cognition*, 113(3):329–349, 2009.

Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, 2021.

Kevin Black, Mitsuhiko Nakamoto, Pranav Atreya, Homer Rich Walke, Chelsea Finn, Aviral Kumar, and Sergey Levine. Zero-shot robotic manipulation with pre-trained image-editing diffusion models. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL `https://openreview.net/forum?id=c0chJTSbci`.

Jehoshua Bruck and Joseph W. Goodman. On the power of neural networks for solving hard problems. In Dana Z. Anderson (ed.), *Neural Information Processing Systems, Denver, Colorado, USA, 1987*, pp. 137–143. American Institue of Physics, 1987. URL `http://papers.nips.cc/paper/70-on-the-power-of-neural-networks-for-solving-hard-problems`.

Robert Brunetto and Otakar Trunda. Deep heuristic-learning in the rubik's cube domain: An experimental evaluation. In Jaroslava Hlavácová (ed.), *Proceedings of the 17th Conference on Information Technologies - Applications and Theory (ITAT 2017), Martinské hole, Slovakia, September 22-26, 2017*, volume 1885 of *CEUR Workshop Proceedings*, pp. 57–64. CEUR-WS.org, 2017. URL `https://ceur-ws.org/Vol-1885/57.pdf`.

Murray Campbell, A. Joseph Hoane Jr., and Feng-Hsiung Hsu. Deep blue. *Artif. Intell.*, 134(1-2):57–83, 2002. doi: 10.1016/S0004-3702(01)00129-1. URL `https://doi.org/10.1016/S0004-3702(01)00129-1`.

Chang Chen, Fei Deng, Kenji Kawaguchi, Çaglar Gülçehre, and Sungjin Ahn. Simple hierarchical planning with diffusion. *CoRR*, abs/2401.02644, 2024. doi: 10.48550/ARXIV.2401.02644. URL `https://doi.org/10.48550/arXiv.2401.02644`.

Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Michael Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan (eds.), *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pp. 15084–15097, 2021. URL `https://proceedings.neurips.cc/paper/2021/hash/7f489f642a0ddb10272b5c31057f0663-Abstract.html`.

Howie Choset, Kevin M. Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia E. Kavraki, and Sebastian Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, Cambridge, MA, 2005. ISBN 978-0-262-03327-5.

Open X-Embodiment Collaboration, Abby O'Neill, Abdul Rehman, Abhiram Maddukuri, Abhishek Gupta, Abhishek Padalkar, Abraham Lee, Acorn Pooley, Agrim Gupta, Ajay Mandlekar, Ajinkya Jain, Albert Tung, Alex Bewley, Alex Herzog, Alex Irpan, Alexander Khazatsky, Anant Rai, Anchit Gupta, Andrew

Wang, Anikait Singh, Animesh Garg, Aniruddha Kembhavi, Annie Xie, Anthony Brohan, Antonin Raffin, Archit Sharma, Arefeh Yavary, Arhan Jain, Ashwin Balakrishna, Ayzaan Wahid, Ben Burgess-Limerick, Beomjoon Kim, Bernhard Schölkopf, Blake Wulfe, Brian Ichter, Cewu Lu, Charles Xu, Charlotte Le, Chelsea Finn, Chen Wang, Chenfeng Xu, Cheng Chi, Chenguang Huang, Christine Chan, Christopher Agia, Chuer Pan, Chuyuan Fu, Coline Devin, Danfei Xu, Daniel Morton, Danny Driess, Daphne Chen, Deepak Pathak, Dhruv Shah, Dieter Büchler, Dinesh Jayaraman, Dmitry Kalashnikov, Dorsa Sadigh, Edward Johns, Ethan Foster, Fangchen Liu, Federico Ceola, Fei Xia, Feiyu Zhao, Freek Stulp, Gaoyue Zhou, Gaurav S. Sukhatme, Gautam Salhotra, Ge Yan, Gilbert Feng, Giulio Schiavi, Glen Berseth, Gregory Kahn, Guanzhi Wang, Hao Su, Hao-Shu Fang, Haochen Shi, Henghui Bao, Heni Ben Amor, Henrik I Christensen, Hiroki Furuta, Homer Walke, Hongjie Fang, Huy Ha, Igor Mordatch, Ilija Radosavovic, Isabel Leal, Jacky Liang, Jad Abou-Chakra, Jaehyung Kim, Jaimyn Drake, Jan Peters, Jan Schneider, Jasmine Hsu, Jeannette Bohg, Jeffrey Bingham, Jeffrey Wu, Jensen Gao, Jiaheng Hu, Jiajun Wu, Jialin Wu, Jiankai Sun, Jianlan Luo, Jiayuan Gu, Jie Tan, Jihoon Oh, Jimmy Wu, Jingpei Lu, Jingyun Yang, Jitendra Malik, João Silvério, Joey Hejna, Jonathan Booher, Jonathan Tompson, Jonathan Yang, Jordi Salvador, Joseph J. Lim, Junhyek Han, Kaiyuan Wang, Kanishka Rao, Karl Pertsch, Karol Hausman, Keegan Go, Keerthana Gopalakrishnan, Ken Goldberg, Kendra Byrne, Kenneth Oslund, Kento Kawaharazuka, Kevin Black, Kevin Lin, Kevin Zhang, Kiana Ehsani, Kiran Lekkala, Kirsty Ellis, Krishan Rana, Krishnan Srinivasan, Kuan Fang, Kunal Pratap Singh, Kuo-Hao Zeng, Kyle Hatch, Kyle Hsu, Laurent Itti, Lawrence Yunliang Chen, Lerrel Pinto, Li Fei-Fei, Liam Tan, Linxi "Jim" Fan, Lionel Ott, Lisa Lee, Luca Weihs, Magnum Chen, Marion Lepert, Marius Memmel, Masayoshi Tomizuka, Masha Itkina, Mateo Guaman Castro, Max Spero, Maximilian Du, Michael Ahn, Michael C. Yip, Mingtong Zhang, Mingyu Ding, Minho Heo, Mohan Kumar Srirama, Mohit Sharma, Moo Jin Kim, Naoaki Kanazawa, Nicklas Hansen, Nicolas Heess, Nikhil J Joshi, Niko Suenderhauf, Ning Liu, Norman Di Palo, Nur Muhammad Mahi Shafiullah, Oier Mees, Oliver Kroemer, Osbert Bastani, Pannag R Sanketi, Patrick "Tree" Miller, Patrick Yin, Paul Wohlhart, Peng Xu, Peter David Fagan, Peter Mitrano, Pierre Sermanet, Pieter Abbeel, Priya Sundaresan, Qiuyu Chen, Quan Vuong, Rafael Rafailov, Ran Tian, Ria Doshi, Roberto Mart'in-Mart'in, Rohan Baijal, Rosario Scalise, Rose Hendrix, Roy Lin, Runjia Qian, Ruohan Zhang, Russell Mendonca, Rutav Shah, Ryan Hoque, Ryan Julian, Samuel Bustamante, Sean Kirmani, Sergey Levine, Shan Lin, Sherry Moore, Shikhar Bahl, Shivin Dass, Shubham Sonawani, Shuran Song, Sichun Xu, Siddhant Haldar, Siddharth Karamcheti, Simeon Adebola, Simon Guist, Soroush Nasiriany, Stefan Schaal, Stefan Welker, Stephen Tian, Subramanian Ramamoorthy, Sudeep Dasari, Suneel Belkhale, Sungjae Park, Suraj Nair, Suvir Mirchandani, Takayuki Osa, Tanmay Gupta, Tatsuya Harada, Tatsuya Matsushima, Ted Xiao, Thomas Kollar, Tianhe Yu, Tianli Ding, Todor Davchev, Tony Z. Zhao, Travis Armstrong, Trevor Darrell, Trinity Chung, Vidhi Jain, Vincent Vanhoucke, Wei Zhan, Wenxuan Zhou, Wolfram Burgard, Xi Chen, Xiaolong Wang, Xinghao Zhu, Xinyang Geng, Xiyuan Liu, Xu Liangwei, Xuanlin Li, Yao Lu, Yecheng Jason Ma, Yejin Kim, Yevgen Chebotar, Yifan Zhou, Yifeng Zhu, Yilin Wu, Ying Xu, Yixuan Wang, Yonatan Bisk, Yoonyoung Cho, Youngwoon Lee, Yuchen Cui, Yue Cao, Yueh-Hua Wu, Yujin Tang, Yuke Zhu, Yunchu Zhang, Yunfan Jiang, Yunshuang Li, Yunzhu Li, Yusuke Iwasawa, Yutaka Matsuo, Zehan Ma, Zhuo Xu, Zichen Jeff Cui, Zichen Zhang, and Zipeng Lin. Open X-Embodiment: Robotic learning datasets and RT-X models. `https://arxiv.org/abs/2310.08864`, 2023.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009. ISBN 0262033844.

Joseph C. Culberson. Sokoban is pspace-complete. 1997. URL `https://api.semanticscholar.org/CorpusID:61114368`.

Konrad Czechowski, Tomasz Odrzygózdz, Marek Zbysinski, Michal Zawalski, Krzysztof Olejnik, Yuhuai Wu, Lukasz Kucinski, and Piotr Milos. Subgoal search for complex reasoning tasks. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan (eds.), *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pp. 624–638, 2021. URL `https://proceedings.neurips.cc/paper/2021/hash/05d8cccb5f47e5072f0a05b5f514941a-Abstract.html`.

Erik D. Demaine, Sarah Eisenstat, and Mikhail Rudoy. Solving the rubik's cube optimally is np-complete. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi: 10.4230/LIPICS.STACS.2018.24. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.STACS.2018.24.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio (eds.), *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423.

Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pp. 1–16, 2017.

Gabriel Dulac-Arnold, Richard Evans, Peter Sunehag, and Ben Coppin. Reinforcement learning in large discrete action spaces. *CoRR*, abs/1512.07679, 2015. URL http://arxiv.org/abs/1512.07679.

Mark Edmonds, Feng Gao, Xu Xie, Hangxin Liu, Siyuan Qi, Yixin Zhu, Brandon Rothrock, and Song-Chun Zhu. Feeling the force: Integrating force and pose for fluent discovery through imitation learning to open medicine bottles. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3530–3537, 2017. doi: 10.1109/IROS.2017.8206196.

Ben Eysenbach, Russ R Salakhutdinov, and Sergey Levine. Search on the replay buffer: Bridging planning and reinforcement learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/5c48ff18e0a47baaf81d8b8ea51eec92-Paper.pdf.

Mehdi Fatemi, Taylor W. Killian, Jayakumar Subramanian, and Marzyeh Ghassemi. Medical dead-ends and learning to identify high-risk states and treatments. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan (eds.), *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pp. 4856–4870, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/26405399c51ad7b13b504e74eb7c696c-Abstract.html.

Dieqiao Feng, Carla P Gomes, and Bart Selman. Left heavy tails and the effectiveness of the policy and value networks in DNN-based best-first search for sokoban planning. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=b6to5kfFhQh.

Arnaud Fickinger, Samuel Cohen, Stuart Russell, and Brandon Amos. Cross-domain imitation learning via optimal transport. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=xP3cPq2hQC.

Ayelet Fishbach and Ravi Dhar. Goals as excuses or guides: The liberating effect of perceived goal progress on choice. *Journal of Consumer Research*, 32(3):370–377, 2005.

Justin Fu, Aviral Kumar, Ofir Nachum, George Tucker, and Sergey Levine. D4RL: datasets for deep data-driven reinforcement learning. *CoRR*, abs/2004.07219, 2020. URL https://arxiv.org/abs/2004.07219.

Mohammad Ghavamzadeh and Sridhar Mahadevan. Hierarchical policy gradient algorithms. In Tom Fawcett and Nina Mishra (eds.), *Machine Learning, Proceedings of the Twentieth International Conference (ICML 2003), August 21-24, 2003, Washington, DC, USA*, pp. 226–233. AAAI Press, 2003. URL http://www.aaai.org/Library/ICML/2003/icml03-032.php.

Kristen Grauman, Andrew Westbury, Eugene Byrne, Zachary Chavis, Antonino Furnari, Rohit Girdhar, Jackson Hamburger, Hao Jiang, Miao Liu, Xingyu Liu, Miguel Martin, Tushar Nagarajan, Ilija Radosavovic, Santhosh Kumar Ramakrishnan, Fiona Ryan, Jayant Sharma, Michael Wray, Mengmeng Xu, Eric Zhongcong Xu, Chen Zhao, Siddhant Bansal, Dhruv Batra, Vincent Cartillier, Sean Crane, Tien Do, Morrie Doulaty, Akshay Erapalli, Christoph Feichtenhofer, Adriano Fragomeni, Qichen Fu, Abrham Gebreselasie, Cristina Gonzalez, James Hillis, Xuhua Huang, Yifei Huang, Wenqi Jia, Weslie Khoo, Jachym Kolar, Satwik Kottur, Anurag Kumar, Federico Landini, Chao Li, Yanghao Li, Zhenqiang Li, Karttikeya Mangalam, Raghava Modhugu, Jonathan Munro, Tullie Murrell, Takumi Nishiyasu, Will Price, Paola Ruiz Puentes, Merey Ramazanova, Leda Sari, Kiran Somasundaram, Audrey Southerland, Yusuke Sugano, Ruijie Tao, Minh Vo, Yuchen Wang, Xindi Wu, Takuma Yagi, Ziwei Zhao, Yunyi Zhu, Pablo Arbelaez, David Crandall, Dima Damen, Giovanni Maria Farinella, Christian Fuegen, Bernard Ghanem, Vamsi Krishna Ithapu, C. V. Jawahar, Hanbyul Joo, Kris Kitani, Haizhou Li, Richard Newcombe, Aude Oliva, Hyun Soo Park, James M. Rehg, Yoichi Sato, Jianbo Shi, Mike Zheng Shou, Antonio Torralba, Lorenzo Torresani, Mingfei Yan, and Jitendra Malik. Ego4d: Around the world in 3,000 hours of egocentric video, 2022.

Arthur Guez, Mehdi Mirza, Karol Gregor, Rishabh Kabra, Sebastien Racaniere, Theophane Weber, David Raposo, Adam Santoro, Laurent Orseau, Tom Eccles, Greg Wayne, David Silver, Timothy Lillicrap, and Victor Valdes. An investigation of model-free planning: boxoban levels. https://github.com/deepmind/boxoban-levels/, 2018.

Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. *An Introduction to the Planning Domain Definition Language*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2019. ISBN 978-3-031-00456-8. doi: 10.2200/S00900ED2V01Y201902AIM042. URL `https://doi.org/10.2200/S00900ED2V01Y201902AIM042`.

Zhiao Huang, Fangchen Liu, and Hao Su. Mapping state space using landmarks for universal goal reaching. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 1940–1950, 2019. URL `https://proceedings.neurips.cc/paper/2019/hash/3b712de48137572f3849aabd5666a4e3-Abstract.html`.

Clark L. Hull. The goal gradient hypothesis and maze learning. *Psychological Review*, 39(1):25–43, 1932.

Steven James, George Konidaris, and Benjamin Rosman. An analysis of monte carlo tree search. *Proceedings of the AAAI Conference on Artificial Intelligence*, 31(1), Feb. 2017. doi: 10.1609/aaai.v31i1.11028. URL `https://ojs.aaai.org/index.php/AAAI/article/view/11028`.

Yiding Jiang, Shixiang Gu, Kevin Murphy, and Chelsea Finn. Language as an abstraction for hierarchical deep reinforcement learning. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 9414–9426, 2019. URL `https://proceedings.neurips.cc/paper/2019/hash/0af787945872196b42c9f73ead2565c8-Abstract.html`.

Michael Kelly, Chelsea Sidrane, Katherine Driggs-Campbell, and Mykel J. Kochenderfer. Hg-dagger: Interactive imitation learning with human experts. In *2019 International Conference on Robotics and Automation (ICRA)*, pp. 8077–8083, 2019. doi: 10.1109/ICRA.2019.8793698.

Kuno Kim, Yihong Gu, Jiaming Song, Shengjia Zhao, and Stefano Ermon. Domain adaptive imitation learning. In Hal Daumé III and Aarti Singh (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 5286–5295. PMLR, 13–18 Jul 2020. URL `https://proceedings.mlr.press/v119/kim20c.html`.

Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan Foster, Grace Lam, Pannag Sanketi, Quan Vuong, Thomas Kollar, Benjamin Burchfiel, Russ Tedrake, Dorsa Sadigh, Sergey Levine, Percy Liang, and Chelsea Finn. Openvla: An open-source vision-language-action model. *arXiv preprint arXiv:2406.09246*, 2024.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL `https://openreview.net/forum?id=SJU4ayYgl`.

B. Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A. Al Sallab, Senthil Kumar Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Trans. Intell. Transp. Syst.*, 23(6):4909–4926, 2022. doi: 10.1109/TITS.2021.3054625. URL `https://doi.org/10.1109/TITS.2021.3054625`.

Robert Kirk, Amy Zhang, Edward Grefenstette, and Tim Rocktäschel. A survey of zero-shot generalisation in deep reinforcement learning. *J. Artif. Intell. Res.*, 76:201–264, 2023. doi: 10.1613/JAIR.1.14174. URL `https://doi.org/10.1613/jair.1.14174`.

Wouter Kool and Matthew Botvinick. A labor/leisure tradeoff in cognitive control. *Journal of Experimental Psychology: General*, 143(1):131–141, 2014.

Kalle Kujanpää, Joni Pajarinen, and Alexander Ilin. Hierarchical imitation learning with vector quantized models. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (eds.), *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pp. 17896–17919. PMLR, 2023a. URL `https://proceedings.mlr.press/v202/kujanpaa23a.html`.

Kalle Kujanpää, Joni Pajarinen, and Alexander Ilin. Hybrid search for efficient planning with completeness guarantees. *CoRR*, abs/2310.12819, 2023b. doi: 10.48550/ARXIV.2310.12819. URL `https://doi.org/10.48550/arXiv.2310.12819`.

Aviral Kumar, Joey Hong, Anikait Singh, and Sergey Levine. When should we prefer offline reinforcement learning over behavioral cloning? *CoRR*, abs/2204.05618, 2022. doi: 10.48550/ARXIV.2204.05618. URL `https://doi.org/10.48550/arXiv.2204.05618`.

Steven M LaValle. *Planning algorithms*. Cambridge university press, 2006.

Seungjae Lee, Jigang Kim, Inkyu Jang, and H. Jin Kim. DHRL: A graph-based approach for long-horizon and sparse hierarchical reinforcement learning. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022. URL `http://papers.nips.cc/paper_files/paper/2022/hash/58b286aea34a91a3d33e58af0586fa40-Abstract-Conference.html`.

Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *CoRR*, abs/2005.01643, 2020. URL `https://arxiv.org/abs/2005.01643`.

Andrew Levy, George Dimitri Konidaris, Robert Platt Jr., and Kate Saenko. Learning multi-level hierarchies with hindsight. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL `https://openreview.net/forum?id=ryzECoAcY7`.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault (eds.), *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 7871–7880, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.703. URL `https://aclanthology.org/2020.acl-main.703`.

Quanyi Li, Zhenghao Peng, and Bolei Zhou. Efficient learning of safe driving policy via human-ai copilot optimization. In *International Conference on Learning Representations*, 2022. URL `https://openreview.net/forum?id=0cgU-BZp2ky`.

Ajay Mandlekar, Yuke Zhu, Animesh Garg, Jonathan Booher, Max Spero, Albert Tung, Julian Gao, John Emmons, Anchit Gupta, Emre Orbay, Silvio Savarese, and Li Fei-Fei. ROBOTURK: A crowdsourcing platform for robotic skill learning through imitation. In *2nd Annual Conference on Robot Learning, CoRL 2018, Zürich, Switzerland, 29-31 October 2018, Proceedings*, volume 87 of *Proceedings of Machine Learning Research*, pp. 879–893. PMLR, 2018. URL `http://proceedings.mlr.press/v87/mandlekar18a.html`.

Stephen McAleer, Forest Agostinelli, Alexander Shmakov, and Pierre Baldi. Solving the rubik's cube with approximate policy iteration. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL `https://openreview.net/forum?id=Hyfn2jCcKm`.

Oier Mees, Lukás Hermann, and Wolfram Burgard. What matters in language conditioned robotic imitation learning over unstructured data. *IEEE Robotics Autom. Lett.*, 7(4):11205–11212, 2022. doi: 10.1109/LRA.2022.3196123. URL `https://doi.org/10.1109/LRA.2022.3196123`.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

Ofir Nachum, Shixiang Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pp. 3307–3317, 2018. URL `https://proceedings.neurips.cc/paper/2018/hash/e6384711491713d29bc63fc5eeb5ba4f-Abstract.html`.

Ashvin Nair, Bob McGrew, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Overcoming exploration in reinforcement learning with demonstrations. In *2018 IEEE International Conference on Robotics and Automation, ICRA 2018, Brisbane, Australia, May 21-25, 2018*, pp. 6292–6299. IEEE, 2018. doi: 10.1109/ICRA.2018.8463162. URL `https://doi.org/10.1109/ICRA.2018.8463162`.

Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

Laurent Orseau and Levi H. S. Lelis. Policy-guided heuristic search with guarantees. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pp. 12382–12390. AAAI Press, 2021. doi: 10.1609/AAAI.V35I14.17469. URL `https://doi.org/10.1609/aaai.v35i14.17469`.

Laurent Orseau, Marcus Hutter, and Levi H. S. Lelis. Levin tree search with context models. In Edith Elkind (ed.), *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI-23*, pp. 5622–5630. International Joint Conferences on Artificial Intelligence Organization, 8 2023. doi: 10.24963/ijcai.2023/624. URL `https://doi.org/10.24963/ijcai.2023/624`. Main Track.

Aleksandr I. Panov and Aleksey Skrynnik. Automatic formation of the structure of abstract machines in hierarchical reinforcement learning with state clustering. *CoRR*, abs/1806.05292, 2018. URL `http://arxiv.org/abs/1806.05292`.

Seohong Park, Dibya Ghosh, Benjamin Eysenbach, and Sergey Levine. HIQL: offline goal-conditioned RL with latent states as actions. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL `http://papers.nips.cc/paper_files/paper/2023/hash/6d7c4a0727e089ed6cdd3151cbe8d8ba-Abstract-Conference.html`.

Karl Pertsch, Oleh Rybkin, Frederik Ebert, Shenghao Zhou, Dinesh Jayaraman, Chelsea Finn, and Sergey Levine. Long-horizon visual planning with goal-conditioned hierarchical predictors. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/c8d3a760ebab631565f8509d84b3b3f1-Abstract.html.

Daniel Ratner and Manfred K. Warmuth. Finding a shortest solution for the N × N extension of the 15-puzzle is intractable. In Tom Kehler (ed.), *Proceedings of the 5th National Conference on Artificial Intelligence. Philadelphia, PA, USA, August 11-15, 1986. Volume 1: Science*, pp. 168–172. Morgan Kaufmann, 1986. URL http://www.aaai.org/Library/AAAI/1986/aaai86-027.php.

Stéphane Ross, Geoffrey J. Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In Geoffrey J. Gordon, David B. Dunson, and Miroslav Dudík (eds.), *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011*, volume 15 of *JMLR Proceedings*, pp. 627–635. JMLR.org, 2011. URL http://proceedings.mlr.press/v15/ross11a/ross11a.pdf.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, USA, 3rd edition, 2009. ISBN 0136042597.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition).* Pearson, 2020. ISBN 9780134610993. URL http://aima.cs.berkeley.edu/.

Sartaj Sahni. Computationally related problems. *SIAM J. Comput.*, 3(4):262–279, 1974. doi: 10.1137/0203021. URL https://doi.org/10.1137/0203021.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

Zheyan Shen, Jiashuo Liu, Yue He, Xingxuan Zhang, Renzhe Xu, Han Yu, and Peng Cui. Towards out-of-distribution generalization: A survey. *CoRR*, abs/2108.13624, 2021. URL https://arxiv.org/abs/2108.13624.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nat.*, 529(7587):484–489, 2016. doi: 10.1038/NATURE16961. URL https://doi.org/10.1038/nature16961.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018. doi: 10.1126/science.aar6404. URL https://www.science.org/doi/abs/10.1126/science.aar6404.

David Singmaster. *Notes on Rubik's Magic Cube.* Enslow Publishers, 1981.

Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

Pei Sun, Henrik Kretzschmar, Xerxes Dotiwalla, Aurelien Chouard, Vijaysai Patnaik, Paul Tsui, James Guo, Yin Zhou, Yuning Chai, Benjamin Caine, Vijay Vasudevan, Wei Han, Jiquan Ngiam, Hang Zhao, Aleksei Timofeev, Scott Ettinger, Maxim Krivokon, Amy Gao, Aditya Joshi, Yu Zhang, Jonathon Shlens, Zhifeng Chen, and Dragomir Anguelov. Scalability in perception for autonomous driving: Waymo open dataset. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pp. 2443–2451. Computer Vision Foundation / IEEE, 2020. doi: 10.1109/CVPR42600.2020.00252. URL https://openaccess.thecvf.com/content_CVPR_2020/html/Sun_Scalability_in_Perception_for_Autonomous_Driving_Waymo_Open_Dataset_CVPR_2020_paper.html.

Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction.* Adaptive computation and machine learning. MIT Press, 1998. ISBN 978-0-262-19398-6. URL `https://www.worldcat.org/oclc/37293240`.

Richard S. Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artif. Intell.*, 112(1-2):181–211, 1999. doi: 10.1016/S0004-3702(99)00052-1. URL `https://doi.org/10.1016/S0004-3702(99)00052-1`.

Kyo Takano. Self-supervision is all you need for solving rubik's cube. *Trans. Mach. Learn. Res.*, 2023, 2023. URL `https://openreview.net/forum?id=bnBeNFB27b`.

Trieu Trinh, Yuhuai Wu, Quoc Le, He He, and Thang Luong. Solving olympiad geometry without human demonstrations. *Nature*, 2024. doi: 10.1038/s41586-023-06747-5.

Aäron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural discrete representation learning. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 6306–6315, 2017. URL `https://proceedings.neurips.cc/paper/2017/hash/7a98af17e63a0ac09ce2e96d03992fbc-Abstract.html`.

Joel Veness, David Silver, Alan Blair, and William Uther. Bootstrapping from game tree search. In Y. Bengio, D. Schuurmans, J. Lafferty, C. Williams, and A. Culotta (eds.), *Advances in Neural Information Processing Systems*, volume 22. Curran Associates, Inc., 2009. URL `https://proceedings.neurips.cc/paper_files/paper/2009/file/389bc7bb1e1c2a5e7e147703232a88f6-Paper.pdf`.

Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom Le Paine, Çaglar Gülçehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft II using multi-agent reinforcement learning. *Nat.*, 575(7782):350–354, 2019. doi: 10.1038/S41586-019-1724-Z. URL `https://doi.org/10.1038/s41586-019-1724-z`.

Homer Walke, Kevin Black, Abraham Lee, Moo Jin Kim, Max Du, Chongyi Zheng, Tony Zhao, Philippe Hansen-Estruch, Quan Vuong, Andre He, Vivek Myers, Kuan Fang, Chelsea Finn, and Sergey Levine. Bridgedata v2: A dataset for robot learning at scale. In *Conference on Robot Learning (CoRL)*, 2023.

Yuhuai Wu, Albert Jiang, Jimmy Ba, and Roger Baker Grosse. {INT}: An inequality benchmark for evaluating generalization in theorem proving. In *International Conference on Learning Representations*, 2021. URL `https://openreview.net/forum?id=O6LPudowNQm`.

Zhaoyang Yang, Kathryn E. Merrick, Lianwen Jin, and Hussein A. Abbass. Hierarchical deep reinforcement learning for continuous action control. *IEEE Trans. Neural Networks Learn. Syst.*, 29(11):5174–5184, 2018. doi: 10.1109/TNNLS.2018.2805379. URL `https://doi.org/10.1109/TNNLS.2018.2805379`.

Ryo Yonetani, Tatsunori Taniai, Mohammadamin Barekatain, Mai Nishimura, and Asako Kanezaki. Path planning using neural a* search. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pp. 12029–12039. PMLR, 2021. URL `http://proceedings.mlr.press/v139/yonetani21a.html`.

Michal Zawalski, Michal Tyrolski, Konrad Czechowski, Tomasz Odrzygózdz, Damian Stachura, Piotr Piekos, Yuhuai Wu, Lukasz Kucinski, and Piotr Milos. Fast and precise: Adjusting planning horizon with adaptive subgoal search. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023.* OpenReview.net, 2023. URL `https://openreview.net/pdf?id=7JsGYvjE88d`.

Jiakai Zhang and Kyunghyun Cho. Query-efficient imitation learning for end-to-end simulated driving. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, pp. 2891–2897. AAAI Press, 2017.

Appendix

Table of Contents

A Environments **23**

B Key Factors For Hierarchical Search **25**
B.1 Learning from diverse data sources . . . 25
B.2 Value Approximation Errors . . . 26
B.3 Complex Action Spaces . . . 29
B.4 Dead Ends . . . 30

C Network Architectures & Training Details **32**

D Offline Pretraining **34**
D.1 Components . . . 34
D.2 Supervised Objectives . . . 34

E Offline Pretraining: Trajectories **35**
E.1 Rubik's Cube . . . 35
E.2 INT . . . 35
E.3 N-Puzzle . . . 35
E.4 Sokoban . . . 35

F Algorithms **36**
F.1 Best-First Search . . . 36
F.2 Monte Carlo Tree Search . . . 38
F.3 A* Search . . . 40
F.4 kSubS And AdaSubS . . . 42
F.5 HIPS And HIPS-ε . . . 43

G Wall Times **44**

H Hierarchical Search **45**

I Further Discussion On HIPS Results **46**

J Common Pitfalls In Hierarchical Search evaluations **47**
J.1 Complete Search Budget . . . 47
J.2 Baselines . . . 47
J.3 Code Quality . . . 48

K Proof Of The Search Advancement formula **50**

L Proof Of The Densification Of The Action Space Theorem **51**

M Comparison with DeepCubeA **54**

A Environments

Sokoban Sokoban is a classic puzzle game where the objective is to push boxes onto target locations within a confined space. It is a popular testing ground for classical planning methods and deep-learning approaches due to its combinatorial complexity and difficulty in finding solutions. Recognized as a PSPACE-hard problem, Sokoban is used to evaluate different computational strategies. Our experiments use 12 $\times$ 12 Sokoban boards with four boxes to assess the performance of our proposed models. An illustrative example of a simple Sokoban search tree with a solving path is shown in Figure 16.

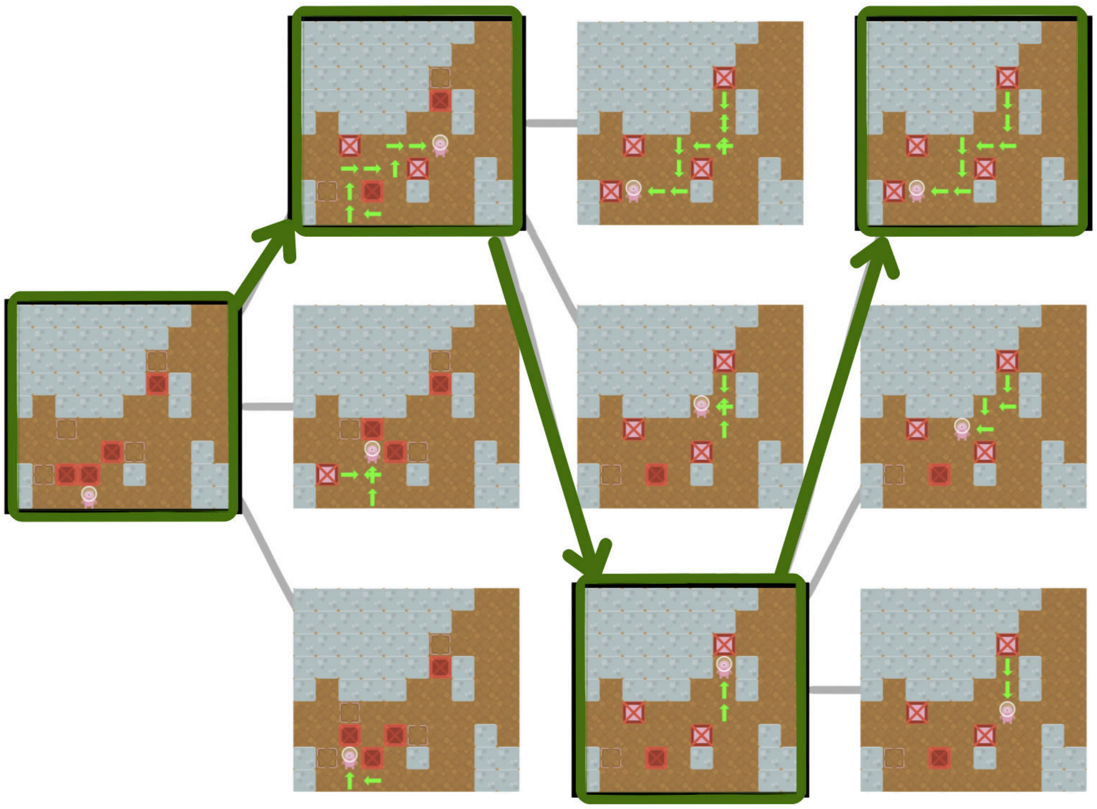


Figure 16: Hierarchical Search applied to solving Sokoban. This tree, depicted in figures, employs bolded green arrows to highlight selected subgoals within a hierarchical search framework earmarked for subsequent exploration. The illustration demonstrates that these intermediate goals exhibit variability in terms of both their spatial distance and the methodology by which a planning algorithm may leverage them.

Rubik's Cube The Rubik's Cube, a renowned 3D puzzle, has over 4.3×10^{19} possible configurations, highlighting the huge search space and the computational challenge it poses. Recent advancements in solving the Rubik's Cube with neural networks underscore the potential of deep learning methods in navigating complex, high-dimensional puzzles. For the exact representation of the Rubik's Cube state, see Figure 17.

N-Puzzle The N-Puzzle, a classic sliding puzzle game, comes in various sizes, including the 3x3 (8-puzzle), 4x4 (15-puzzle), and 5x5 (24-puzzle). The goal is to rearrange a frame of numbered square tiles into a specific pattern, a task that tests the algorithm's ability to plan and execute a sequence of moves efficiently. Figure 18 shows a visualization of a trajectory in 24-puzzle.

INT INT (INequality Theorem proving) is an automated theorem-proving benchmark for high school algebraic inequality proofs. (Wu et al., 2021) provides a generator of mathematical inequalities and a proof verification tool. Each action in INT maps to a proof step, which specifies a chosen axiom and its input

wbrwyggwwoboybygbryrorroboygrbggbggbwybrooogrywrowywwy	s_0	Initial State
wbrwyggggobwybwgbgooyrroyrbrrbwgbygbwybrooogroyrowywwy	s_1	One Action (= single rotation)
wbywyoggbobwybwgbgoorrryyryywrggrbbbgybgoorgroyoowrwww	s_2	
gyowyoggbwbwwbwwbgoorrryyryywwggbbbyboryogggroyoowrbrr	s_3	
. . .		
yyyyyyyyybbbbbbrrrrrrrrrgggggggggooooooooobbbwwwwwwwww	s_{n-1}	
yyyyyyyyybbbbbbbbbrrrrrrrrrgggggggggooooooooowwwwwwwww	s_n	Solving State

Figure 17: Example trajectory of Rubik starting from initial state s_0 leading to the final solution s_n.

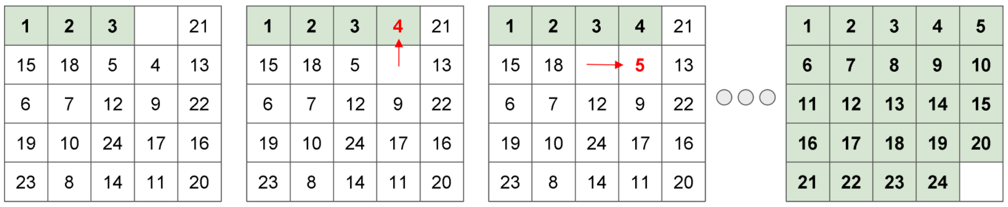


Figure 18: Example trajectory of n-puzzle starting from initial state s_0 leading to the final solution s_n. Red arrows indicate low-level actions.

entities - which makes action space very high-dimensional, enabling up to a million valid actions at a step. This large action space makes INT a desirable but challenging environment for expanding HRL paradigms to vast action spaces.

We used 25-step proofs for this paper, representing an uplift from 15 considered in (Czechowski et al., 2021; Zawalski et al., 2023) (the latter used longer proofs, but only for evaluating 15-trained models). Each step is an application of an axiom to an axiom-specific number of entities (entities are bracketed or bracketable parts of the theorem's goal).

Example Theorems for INT environment

Theorem 1 Premises: $((c+c)+d) \geq a$;
$(d+e) \geq 0$;
$((c+c)+f) \geq (0+a)$;
$(b+g) \geq 0$;

Goal: $((((((c+c)+(c+c)) \cdot 4c)+((c+c)+d))+(d+e))+((c+c)+f))+(b+g))$
$\geq ((((0+a)+0)+(0+a))+0)$

Theorem 2 Goal: $(((0+b)+c)+a) \geq (0+(0+(b+(c+a))))$

Theorem 3 Premises: $(a+d) \geq 0$;
$(a+e) \geq (c \cdot c)$;
$(e+f) \geq 0$;
$(c+g) \geq 0$;
$(c+h) \geq (c+g)$;
$(c+i) \geq 0$;

Goal: $(((((((c \cdot c) \cdot (a+d))+(a+e)) \cdot (e+f)) \cdot (c+g))+(c+h)) \cdot (c+i))$
$\geq ((((((0 \cdot (a+d))+(c \cdot c)) \cdot (e+f)) \cdot (c+g))+(c+g)) \cdot (c+i))$

Figure 19: A comprehensive representation of theorems pertaining to goal achievement in mathematical expressions, showcasing the logical structure and underlying premises leading to the formulated goals.

B Key Factors For Hierarchical Search

According to our experiments, the attributes pivotal for leveraging the advantages of high-level search include:

- learning from diverse data sources,
- hard-to-learn value function,
- complex action space,
- presence of dead ends

In Section 5, we show our main experiments that support our findings. In this appendix, we present an extended analysis of each property.

B.1 Learning from diverse data sources

Achieving superhuman performance in complex tasks, as demonstrated by AlphaGo Silver et al. (2016), often involves large-scale datasets of demonstrations obtained from agents with varying skill levels and strategies. However, this diversity introduces challenges such as inconsistencies in demonstrations and variations in quality (Fu et al., 2020; Chen et al., 2021; Levine et al., 2020). Widely used datasets like D4RL (Fu et al., 2020), Open X-Embodiment (Collaboration et al., 2023), or Waymo Open Dataset (Sun et al., 2020) reflect this diversity, highlighting the need to address these challenges effectively. We want to answer the question whether such setting is handled better by high-level or low-level search algorithms.

Experiment setup For this analysis, we focus on the Rubik's cube environment. We collected a dataset of 500 000 trajectories, computed with four different solvers for the Rubik's cube:

- Beginner – the simplest human-oriented solving algorithm. It aims to order the cube layer by layer with a few primitive tactics. Because of that the solutions are structured, but also very long (typically between 150 and 200 moves).
- CFOP – an algorithm designed for speedcubers. It is based on the same principle as Beginner, but employs many advanced tactics that make the solutions faster (typically about 100 moves).
- Kociemba – a computational solver that finds near-optimal solutions (usually between 20 and 40 moves) in short time. It is heavily optimized based on the algebraic properties of the Rubik's cube.
- Random – solutions obtained by scrambling an ordered cube with random moves and reversing the trajectory.

Figure 30 shows example solutions generated with each solver. Clearly, the algorithmic solvers (Beginner and CFOP) generate much longer solutions that the other methods. They are also more structured, as they are based on building patterns. The computational solver Kociemba on the other hand go directly towards the solution because its moves are carefully optimized to ensure maximal advantage. Because of that, this dataset represent a truly diverse set of demonstrations.

Results As shown in Figure 2, the subgoal methods outperform the low-level methods by a wide margin. While ρ-BestFS is comparable on small budgets, it struggles with solving most of the instances. Also, it should be noted that the performance of the subgoal methods changes only slightly compared to training on a single Random solver (Figure 4) while the low-level searches are heavily affected.

Learned values To find the sources of that outcome, we checked the values learned by the heuristic function. Because of the diversity introduced by combining the experts, we should expect that the estimates are subject to high uncertainty and possibly high variance.

Figure 20 shows the distribution of the learned heuristic for random fully shuffled cubes. Although most instances can be solved optimally within 20-26 moves, the estimates range from 14 to 90 steps. Furthermore,

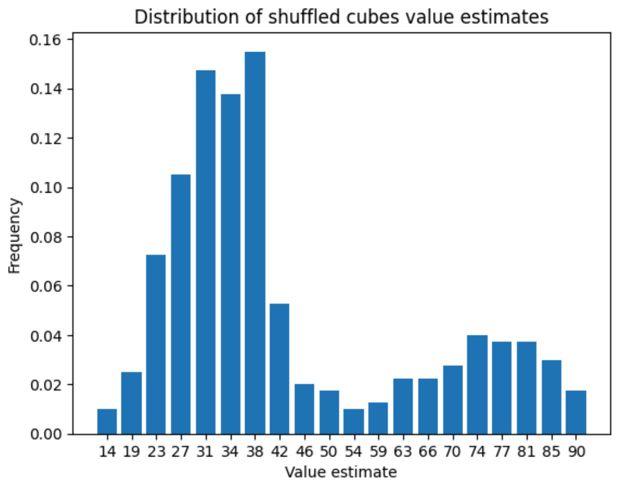


Figure 20: Value distribution for fully scrambled cubes, learned on data coming from diverse experts. The values are rescaled so that the x-axis represent the estimated number of steps to the solution. The values represent the mean of each interval.

the distribution is clearly bimodal – one mode correspond to a typical length of Kociemba solution, the other to CFOP.

Furthermore, Figure 25 shows the distribution of value estimates throughout the solutions for each solver. We observe that for the algorithmic solvers the initial distance is considerably underestimated. After about 20% moves the value network recognizes the pattern of layers built by the solvers and expect a long solution by assigning values close to 100. On the other hand, the values learned for the states visited by the computational solvers start as overestimated, but steadily decrease towards 0.

While it is a reasonable strategy for the value to fit to the provided dataset, it creates a challenge for the search. If a search algorithm aims to imitate Beginner or CFOP, it has to reach the layer pattern, characteristic of those solvers. However, the random states tend to have very low distance estimate, compared to the initial layer patterns. Because of that, for tens of steps the heuristic estimates would be actually increasing, making the reached states less and less probable to expand.

In practice, the low-level searches usually fail to cross this gap. On the other hand, the high-level methods are partially guided by the subgoal generators that ignore the values. The value gap that spans across about 30 steps can be crossed with as few as 5 subgoals of length 6. Because of that both kSubS and AdaSubS can successfully leverage the schematic algorithmic solutions.

To finally confirm that conclusion, we must answer the question whether the performance of low-level searches would increase if they could leverage the algorithmic solutions as well. For that purpose, we trained the components for each method using data only from the Beginner solver. This way we remove the challenge of noisy initial values. As shown in Figure 5, the low-level searches indeed perform much better. BestFS even matches the performance of AdaSubS. That confirms our observation that low-level searchas fail to utilize multimodal data because they rely too much on the value function and seek monotonic slopes.

At the same time we observe that since BestFS and AdaSubS show nearly identical performance on Beginner solutions, it is questionable that hierarchical methods handle long-horizon tasks better, which is a common belief (Nachum et al., 2018; Eysenbach et al., 2019; Chen et al., 2024).

B.2 Value Approximation Errors

In many practical scenarios, value function estimates are based on either limited data samples or hand-crafted heuristics (Campbell et al., 2002; Mnih et al., 2015; Walke et al., 2023). This often leads to high

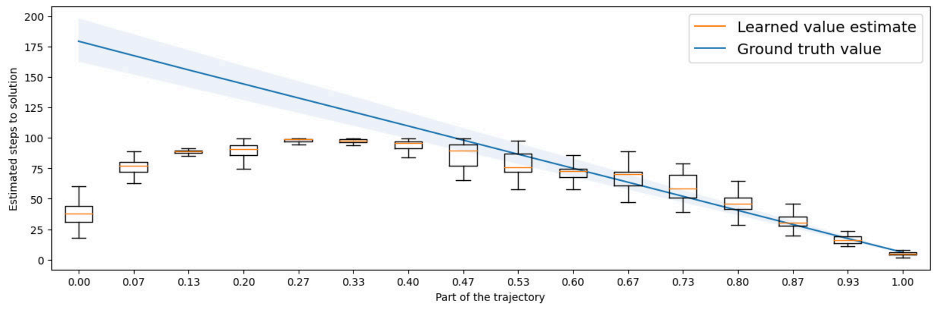


Figure 21: Beginner solver

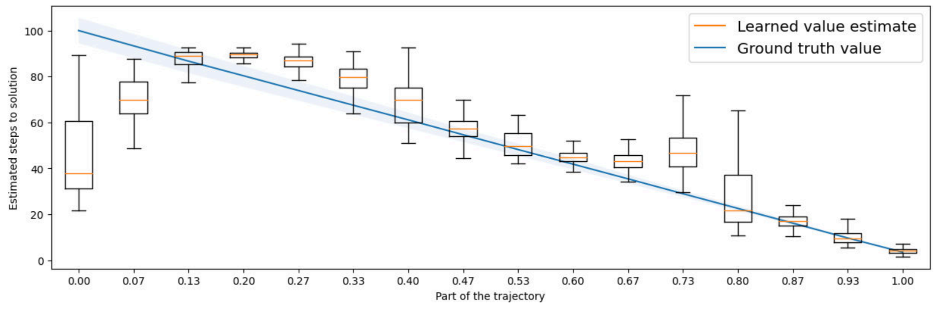


Figure 22: CFOP solver

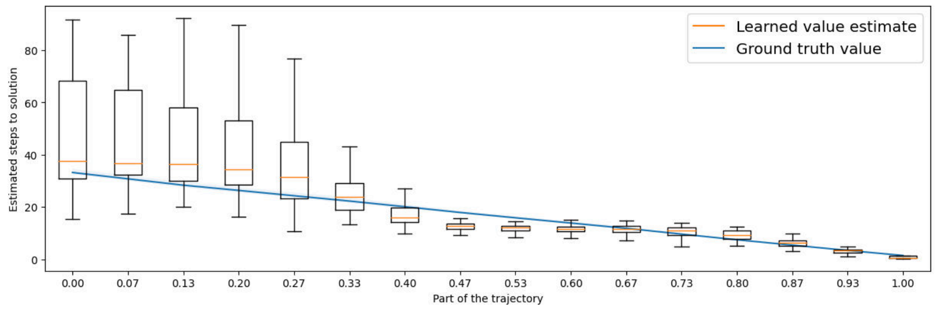


Figure 23: Kociemba solver

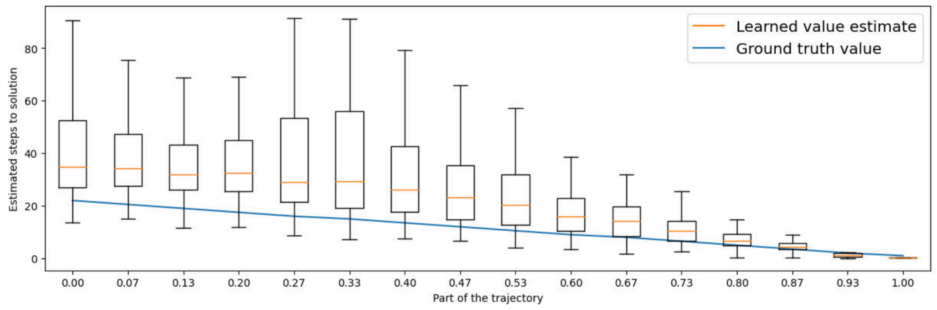


Figure 24: Reversed random 20-move trajectories

Figure 25: The learned value estimates distribution for various solvers. For each plot 100 episodes were solved using the respective solver. The boxes represent the distribution of value estimates for the consecutive points of the solution. The x-axis denotes the relative part of the trajectory (i.e., 0.5 denotes the middle point in each trajectory, regardless of its length). The blue line indicates the true number of steps to the solution.

approximation errors. If search algorithms rely too heavily on these imperfect estimates, they can make poor decisions, especially in large and complex environments where accurate value estimates are even harder to obtain (Collaboration et al., 2023; Vinyals et al., 2019).

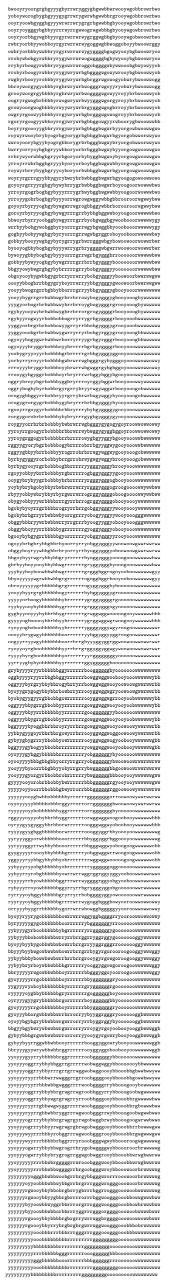


Figure 26: Beginner

Figure 27: CFOP

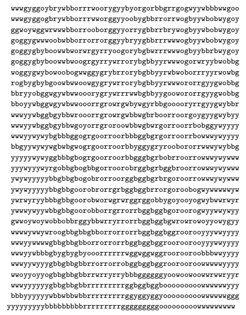


Figure 28: Kociemba

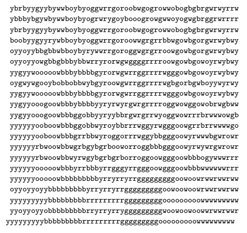


Figure 29: Random

Figure 30: Example solutions computed by each solver. Because the algorithmic solvers typically require over 100 steps, we use a tiny font to display it.

Section B.1 hints that when value estimates are subject to high uncertainty, subgoal methods should outperform low-level searches. To confirm that intuition, we run an experiment in a Rubik's cube, N-Puzzle, and Sokoban environments (Section 5.2). During inference, we add additional noise to the value estimates. That is, whenever a node is added to the search tree and its value estimate equals $\hat{v}$, we add it with the value of $\hat{v} + \mathcal{N}(0, \sigma)$ instead.

Figure 7 shows that as the amount of noise increases, each low-level method gets less and less efficient. On the extreme, when using fully random values ($\sigma = 100$), they struggle to solve any instance.

On the other hand, subgoal methods are much more resilient to noise in the value. Adaptive Subgoal Search is nearly not affected by the presence of noise. kSubS is able to retain as much as $40\% - 90\%$ success rate, even with completely random values.

Observe that the search performed by low-level methods is guided mainly by the value function. Hence, if the computed estimates are subject to high variance, low-level search struggles to make any progress. On the other hand, the subgoal search is guided both by the value function and the subgoal generator. Both the subgoal generator and the conditional policy that connects subgoals do not depend on the values. Hence, the value function is used only in the high-level nodes, which is only a fraction of the search tree.

An extreme case of that behavior is demonstrated by Adaptive Subgoal Search. Because in our configuration each generator outputs a single subgoal, the value is nearly not used at all for search. Only when the search is stuck, the secondary generators select the highest-ranked node to expand, which in this case is simply a random node of the tree. To summarize, given random value estimates, AdaSubS reduces to the following strategy:

1. Start from the root node,
2. Move from the current node to the subgoal until possible,
3. If the search is stuck, expand a random node in the search tree with a secondary generator and return to (2).

The experiments show that this simple strategy is surprisingly competitive to the greedy best-first approach, even without noise. Interestingly, it could be implemented in low-level search as well. We leave that promising experiment for future work.

B.3 Complex Action Spaces

In environments with large action spaces, search methods often struggle due to the exponential increase in the number of choices at each decision point (Sutton & Barto, 1998). This complexity makes it difficult to efficiently identify optimal actions, slowing down decision-making and exploration (Dulac-Arnold et al., 2015; Silver et al., 2016).

The primary difference between low-level methods and subgoal methods is that the former predicts the next action, and the latter – the next state. In many environments, the action space is as simple as a few bits, allowing for iterating over all possible actions, and sampling them. At the same time, states may be considerably larger, up to the extreme of image observations. However, in some environments, the action space is comparable to the state space, or even more complex. A classic example is the AntMaze environment, in which actions are 8-dimensional, while the goal space is only 2-dimensional (Fu et al., 2020).

Among the combinatorial reasoning environments we consider, INT has the most complex action space. In INT, actions correspond to proof steps and are represented as the chosen axiom, specification of its input entities, and the required premises (Wu et al., 2021). Thus, the complexity of the action is at least comparable to the states. Moreover, solving the INT inequalities is based on constant simplification of the given expression, so the state is getting even smaller with each step.

Our experiments, shown in Figure 10, clearly confirm the advantage of using subgoal methods in the INT environment. To further verify the source of that advantage, we conducted another experiment, in a modified Rubik's cube environment. Recall that the experiment presented in Section 5.1 shows that subgoals offer no significant advantage in the *original* Rubik's cube (with a single data source). Now, we want to check whether the outcome would be different if the action space were more complex. For that purpose, we extended the action space 100 times. That is, the new action space consists of 1200 possible moves to choose from – 100 copies of each original action.

As shown in Figure 11, the subgoal methods are barely affected by the change, while the low-level searches are unable to exceed 20% success rate. That result confirms our proposition that when facing a complex action space, hierarchical methods offer considerably better performance.

According to our analysis, the primary issue with low-level searches in the augmented Rubik's cube is the lack of diversity of visited states. When for each state there are hundreds of actions that lead to a similar outcome, they are rated similarly by the policy. Hence, all the top actions essentially lead to the same outcome, which strongly limits the branching factor and trivializes the search trees. On the other hand, subgoal methods are not affected because subgoal generation does not depend on the action space. The conditional policy that connects the generated subgoals does not build a search tree, but always follows the single best action. Because of that, subgoal methods maintain their performance, even though the action space is much more complex.

It is also important to note that even though some state spaces may seem complex, the underlying manifold of possible configurations is in fact low-dimensional. For instance, we use 12x12 Sokoban boards, where each square is encoded as one-hot of 7 possible items, so technically the state space is 1008-dimensional, while there are only 4 actions. However, in practice the subgoal is defined by the positions of agent and boxes, which is at most 10-dimensional, hence rather simple to generate.

B.4 Dead Ends

Dead-end states present a major challenge in decision-making and planning tasks. Once an agent encounters a dead end, reaching the goal becomes impossible, leading to wasted computational effort as the algorithm may continue exploring parts of the search space that do not contribute to solving the problem (Russell & Norvig, 2020). Failing to identify dead-ends may even lead to unsafe behavior (Fatemi et al., 2021; Sutton & Barto, 1998). At the same time, identifying dead-ends is NP-complete in many environments.

Specifically, a dead-end state s is one from which there exists no feasible sequence of actions that leads to the goal state. Figure 31 shows an illustrative example of a dead-end state.

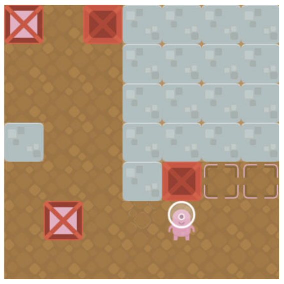


Figure 31: An example dead-end in Sokoban – a box that is pushed to the corner cannot be moved anymore, so the objective is not possible to achieve.

B.4.1 Examples Of Dead-Ends In kSubS vs. BestFS

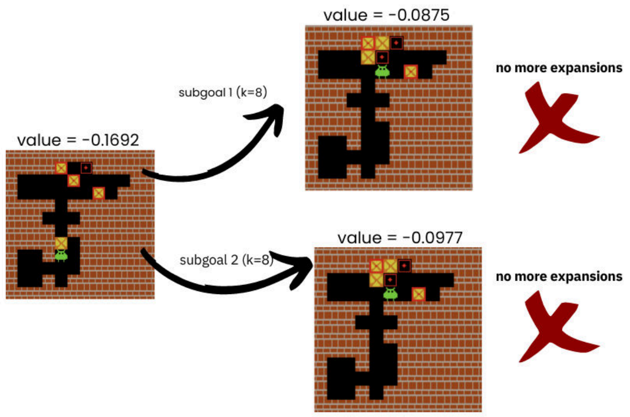


Figure 32: We illustrate a scenario where the kSubS algorithm encounters dead-ends, hindering the search process. The figure shows a case where the algorithm generates two subgoals at an expected distance (k=8), but both lead to dead-ends, wasting a portion of the search budget (18 nodes). As a result, the kSubS algorithm backtracks from this subtree and continues searching elsewhere within the tree.

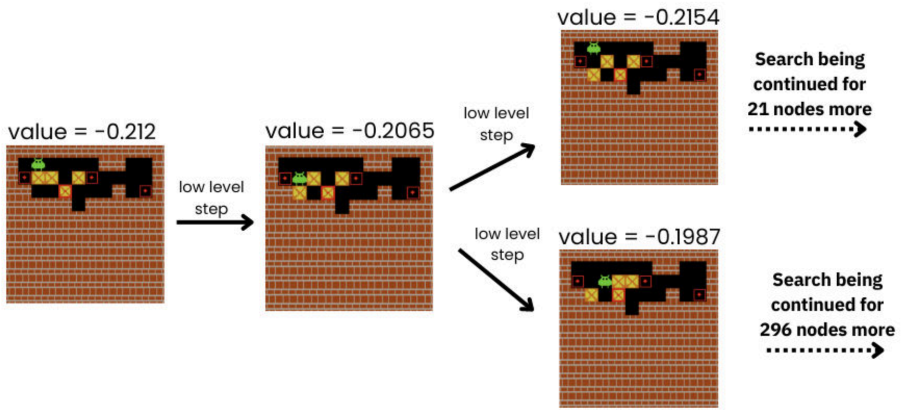


Figure 33: The figure shows BestFS expanding two nodes from a dead-end. This resulted in the exploration of over 300 additional nodes from that state, ultimately failing to find a solution within the given search budget.

In this subsection, we present examples of how each method handles dead-end situations during the search process.

For this presentation, we analyzed 128 search trees initiated from identical starting boards for both algorithms. The kSubS algorithm encountered dead-ends in 3 instances. To resolve these, it navigated through 13 high-level nodes and 105 low-level nodes within the corresponding subtrees. In contrast, the BestFS algorithm encountered dead-ends in 18 instances, requiring the traversal of 4431 nodes. Note that BestFS does not distinguish between high-level and low-level nodes in its search.

Examples of dead-end handling are shown in Figure 32 for kSubS and Figure 33 for BestFS. Observe that in the case showed in Figure 32 expanding the parent node resulted in adding two more dead-ends to the search tree. Because they have higher values, they were immediately expanded. However, the subgoal generator understood that the only way to reach solution is to make an invalid transition of releasing the blocked box. Such subgoals cannot be achieved by the conditional policy, hence no more subgoal was created in that branch. On the other hand, low-level search is unable to propose invalid transitions, so it stays in dead-end until the value estimates are higher than for other branches.

C Network Architectures & Training Details

Environment	Hyperparameter	Generator	CLLP	Value	Policy
INT	learning rate	0.0001	0.0001	0.0003	0.0001
	learning rate scheduling	linear	linear	linear	linear
	warmup steps	4000	4000	2000	4000
	batch size	32	32	128	32
	weight decay	1e-05	1e-05	1e-05	1e-05
	dropout	0.1	0.1	0	0.1
Rubik's Cube	learning rate	0.0001	0.0005	3e-7	0.0001
	learning rate scheduling	linear	linear	linear	linear
	warmup steps	5000	50000	50000	1000
	batch size	512	5000	5000	2048
	weight decay	0.0001	0.001	0.00001	0.0001
	dropout	0.1	0	0	0
Sokoban	learning rate	0.00001	0.0001	0.0001	0.0001
	learning rate scheduling	linear	linear	linear	linear
	warmup steps	2500	1000	1000	1000
	batch size	512	2048	2048	2048
	weight decay	0.0001	0.0001	0.0001	0.000001
	dropout	0	0.1	0	0
N-Puzzle	learning rate	0.0001	0.0001	0.0001	0.0001
	learning rate scheduling	linear	linear	linear	linear
	warmup steps	5000	2000	2000	2000
	batch size	4096	4096	512	4096
	weight decay	0.00001	0.00001	0.00001	0.0001
	dropout	0.1	0	0	0

Table 1: Training-related hyperparameter values

We used BART (Lewis et al., 2020) and BERT (Devlin et al., 2019) architectures from HuggingFace Transformers for all components. Subgoal generators and INT's policies (CLLP and baseline policy) use BART. The remaining policies and value functions use BERT. Following the practice in (Zawalski et al., 2023), we've reduced model size parameters, as detailed in Table 2.

INT As states in INT are complex objects, we prefer to use their string representations and avoid mapping arbitrarily generated strings into complex states. Requisite modifications to the component definition are best illustrated analogously to D.1. A generator is redefined as follows:

$$\mathcal{G}_{\text{int}} : \underbrace{\mathcal{S}}_{\text{state to expand}} \rightarrow \underbrace{P(\mathcal{T})}_{\text{set of } \textit{proposed} \text{ subgoals (in string format)}}$$

and conditional level policy:

$$\mathcal{P}_{\text{int}} : \underbrace{\mathcal{S}}_{\text{current state}} \times \underbrace{\mathcal{T}}_{\text{subgoal } \textit{representation}} \rightarrow \underbrace{\mathcal{A}}_{\text{action}}$$

Sokoban Unlike prior work (Zawalski et al., 2023; Czechowski et al., 2021), which used convolutional networks for all components, we work on tokenized representations of Sokoban boards and use BERT/BART architectures instead. This modification did not adversely impact our ability to replicate AdaSubS and kSubS results.

Training pipeline We trained our models from scratch using the HuggingFace Transformer pipeline. Detailed training parameters, which varied across environments, can be found in 1.

Infrastructure For training, we used a single NVIDIA A100 40GB GPU node, and each component's training took up to 48 hours. Because we used pre-trained trajectories, we did not need to use more than one core during training. We ran an evaluation using 24-core CPU jobs on Xeon Platinum 8268 nodes with 192GB of memory.

Environment	Hyperparameter	Generator	CLLP	Value	Policy
	d model	512	512	-	512
	decoder layers	6	6	-	6
INT	intermediate size	-	-	256	-
	encoder attention heads	8	8	-	8
	hidden size	-	-	128	-
	num hidden layers	-	-	2	-
	decoder ffn dim	2048	2048	-	2048
	encoder ffn dim	2048	2048	-	2048
	encoder layers	6	6	-	6
	decoder attention heads	8	8	-	8
	d model	256	-	-	-
	decoder layers	3	-	-	-
Sokoban	intermediate size	-	512	128	512
	encoder attention heads	4	-	-	-
	hidden size	-	512	128	512
	num hidden layers	-	6	1	6
	encoder ffn dim	2048	-	-	-
	decoder ffn dim	1024	-	-	-
	encoder layers	3	-	-	-
	decoder attention heads	4	-	-	-
	d model	64	-	-	-
	decoder layers	3	-	-	-
N-Puzzle	intermediate size	-	128	128	256
	encoder attention heads	4	-	-	-
	hidden size	-	128	128	256
	num hidden layers	-	2	1	3
	encoder ffn dim	64	-	-	-
	decoder ffn dim	64	-	-	-
	encoder layers	3	-	-	-
	decoder attention heads	4	-	-	-
	d model	256	-	-	-
	decoder layers	3	-	-	-
Rubik's Cube	intermediate size	-	512	128	512
	encoder attention heads	4	-	-	-
	hidden size	-	512	128	512
	num hidden layers	-	2	1	6
	encoder ffn dim	2048	-	-	-
	decoder ffn dim	1024	-	-	-
	encoder layers	3	-	-	-
	decoder attention heads	4	-	-	-

Table 2: Model-related hyperparameter values

D Offline Pretraining

Models are pretrained using an offline imitation learning approach. Specifically, given a set of solution trajectories $\{(s_0, s_1, \ldots, s_{n_i})\}_{i=1}^N$ produced by an expert $\mathcal{M}$, or multiple experts $\{\mathcal{M}_j\}_{j=1}^M$ in cases where offline trajectories are collected from multiple experts, the objective is to learn from these trajectories. It is important to note that these trajectories are not required to be optimal; they may include loops or numerous redundant actions. Description of all components can be found in section $D.1$ and supervised training objectives in section $D.2$.

D.1 Components

During the pretraining phase, models undergo an offline imitation learning process. Specifically, they are trained on a set of solution trajectories $\{(s_0, s_1, \ldots, s_{n_i})\}_{i=1}^N$, which are collected to facilitate the learning of decision-making strategies.

Generator The generator component is responsible for generating subgoal propositions upon receiving a state. These propositions are designed to facilitate progress toward the solution by suggesting intermediate steps that direct the search process more efficiently.

$$\mathcal{G}: \underbrace{\mathcal{S}}_{\text{state to expand}} \rightarrow \underbrace{P(\mathcal{S})}_{\text{set of subgoal propositions}}$$

Conditional Low-Level Policy The Conditional Low-Level Policy (CLLP) plays a crucial role in node expansion by evaluating each subgoal proposition. For a given current state and a subgoal, the CLLP recommends actions that lead toward achieving the subgoal. A path from the current node to the subgoal is constructed through the iterative execution of these actions. Subgoals reached within a predefined number of steps, k, are incorporated into the graph, while those that are not are discarded.

$$\mathcal{P}: \underbrace{\mathcal{S}}_{\text{current state}} \times \underbrace{\mathcal{S}}_{\text{subgoal state}} \rightarrow \underbrace{\mathcal{A}}_{\text{action}}$$

Value The value function estimates the distance from a current state to the final solution. This estimation is used to guide the selection and expansion of nodes, influencing the overall search strategy.

$$\mathcal{V}: \underbrace{\mathcal{S}}_{\text{state to evaluate}} \rightarrow \underbrace{\mathbb{R}}_{\text{value of the state}}$$

Behavioral Cloning Policy The policy Π_{BC} is a decision-making function that maps the current state to an action. It encapsulates the strategy derived from the learning process, guiding the agent's actions towards achieving the final goal.

$$\Pi_{\text{BC}}: \underbrace{\mathcal{S}}_{\text{current state}} \rightarrow \underbrace{\mathcal{A}}_{\text{action}}$$

D.2 Supervised Objectives

Each expert trajectory is defined as a sequence of states and corresponding actions $(s_0, a_0), \ldots, (s_{n-1}, a_{n-1}), s_n$ that delineate a path to a solution. The training methodology leverages this data through several key self-supervised imitation mappings:

- A k-subgoal generator that maps a state s_i to a future state s_{i+k}, simulating the achievement of intermediate goals.
- A value function that estimates the remaining steps to the solution by mapping state s_i to a numerical value $(i - n)$, representing the estimated distance from the goal.
- A policy that maps each state-action pair (s_i, s_{i+d}), with $d \leq k$, to the corresponding action a_i, thereby guiding the decision-making process towards the solution.

E Offline Pretraining: Trajectories

E.1 Rubik's Cube

E.1.1 Random

To construct a random successful trajectory, we performed 20 random permutations on an initially solved Rubik's Cube and took the reverse of this sequence, replacing each move with its reverse. Such solutions are usually sub-optimal since random moves are not guaranteed to increase the distance from the solution. They can even make loops in the trajectories. However, a cube scrambled with 20 moves is usually close to a random state, so such trajectories give a decent space coverage.

E.1.2 Beginner, CFOP

Beginner and *CFOP* are algorithms commonly used by humans. They solve the cube by ordering the stickers layer by layer. Because of that, the solutions are highly structured and long – usually between 100 and 200 moves. Both algorithms are composed of several subroutines that help building the consecutive layers. Thus, the structure of such solutions highly resembles the subgoal search.

E.1.3 Kociemba

The *Kociemba two-stage solver* leverages the algebraic structure of the Rubik's Cube. In the first stage, its goal is to enter a specific subgroup. Since that subgroup is much smaller than the whole space, completing the solution may be done efficiently. *Kociemba* finds reasonably short solutions (usually between 20 and 40 moves) and works reasonably fast.

E.1.4 Size Of Datasets

For training the components on a dataset collected by a single solver, we generate 100 000 trajectories. For the experiment with diverse experts, each solver generates 25 000 trajectories for a total of 100 000.

E.2 INT

Trajectories are constructed from sequences of axiom applications, similarly to (Zawalski et al., 2023), who followed (Wu et al., 2021). A set of up to 15 (out of 18) axioms is first selected, and then a random axiom order is set and validated. Finally, a proof is converted to a relevant trajectory. Approximately 500,000 trajectories were generated for model pre-training.

We capped the number of axioms at 15 because some pairs of axioms (eg. terminal axioms) cannot be in one trajectory.

E.3 N-Puzzle

To collect data for N-puzzles, we utilized an algorithm that initially arranges block number 1, followed by block number 2, and so forth, as depicted in Figure 18. The training set comprises approximately 10, 000 trajectories.

E.4 Sokoban

To collect trajectories for Sokoban, we used a trained MCTS agent that gathered approximately 100, 000 trajectories.

F Algorithms

F.1 Best-First Search

Overview Best-First Search greedily prioritizes node expansions with the highest heuristic estimates, aiming for paths that likely lead to the goal. While not ensuring optimality, BestFS provides a simple yet efficient strategy for navigating complex search spaces. The high-level pseudocode for BestFS is outlined in Algorithm 1, and the detailed pseudocode is presented in Algorithm 2.

Algorithm 1 Pseudocode for Best-First Search

```
while has nodes to expand do
    Take node $N$ with the highest value
    Select children $n_i$ of $N$
    Compute values $v_i$ for the children
    Add $(n_i, v_i)$ to the search tree
end while
```

Heuristic In our implementation, we adhere to the Best-First Search principle by utilizing the learned value function, a common practice in the planning domain (Brunetto & Trunda, 2017; Czechowski et al., 2021; Zawalski et al., 2023; Kujanpää et al., 2023a). It should be noted that in each of our experiments, all the compared algorithms use the same value function network. This way we ensure that the differences come solely from the algorithmic part.

Selecting children When expanding a node during search, the standard BestFS algorithm adds all its children. However, in our implementation, we aimed to reduce the search tree size by selecting only the most promising children. We achieve this by sorting the children according to their probability distribution predicted by the policy network. For choosing the final subset of children, we employ two approaches. In the simpler variant, we always select the top k actions. In the second variant, we add top children until their cumulative probability exceeds a fixed threshold t_{conf}.

This pruning does not adversely affect the standard algorithm, as nodes are still chosen based on their heuristic values, while the threshold sets a practical limit on the search space. Our results demonstrate that BestFS tends to perform much better with a confidence threshold (Figure 34). However, its performance is highly sensitive to this threshold as it balances exploration and exploitation, illustrating the impact of different confidence thresholds on success rates.

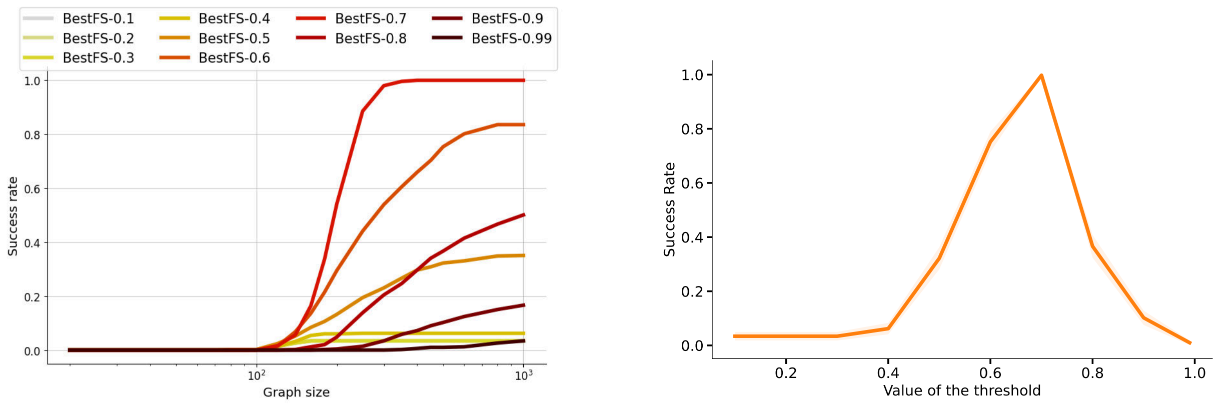


Figure 34: Comparison of success rates for the BestFS algorithm on the Rubik's Cube with various confidence threshold values. BestFS-X represents the BestFS algorithm with the confidence threshold set to X. *Left:* The plot displays the achieved success rate relative to the graph size. *Right:* The plot illustrates the success rate for a budget of 500 nodes.

Completeness In the Rubik's Cube environment with random trajectories, the subgoal methods solve more instances than BestFS given a low search budget, but with more resources, BestFS takes the lead (see Figure 4). Also, in other experiments, we may observe that BestFS typically requires higher computational budget to solve the simplest instances, but its performance increases considerably with more resources.

That behavior is related to the fact that the search trees built by hierarchical methods are much sparser because the branching occurs only in the high-level nodes. On the other hand, the low-level algorithms can cover a higher fraction of the space. On the extreme, if we used all the available actions for every expansion, the low-level search would be *guaranteed* to find a solution if one exists. Our mechanism of selecting the actions removes that guarantee. However, at the same time, it drastically improves performance (compare BestFS-0.7 with BestFS-0.99 which is complete), which makes it a much better choice for our study.

We note that the high-level algorithms could be made complete, as proposed in (Kujanpää et al., 2023b; Zawalski et al., 2023). However, to maximize the efficiency we choose to keep the tested algorithms in their original form. The ability to search with sparse trees not only lets the methods advance fast, but also withdraw quickly if the branch does not lead to the solution (is a dead end).

Hyperparameters To identify the most suitable solving parameters, we used grid search. Initially we grid over coarse values (namely 0.1, 0.2, 0.3, 0.4,0.5, 0.6, 0.7, 0.8, 0.9, and 0.99). Then we check finer values (with precision of 0.05) around the best-performing threshold. The best-performing thresholds range from 0.6 to 0.85, depending on the environment and the components that are used.

For determining the best number of top actions k for the simpler variant, we simply check every possible number of actions. Usually selecting 2 actions is by far the best choice.

Details regarding hyperparameters of the networks are listed in Appendix D.1.

Algorithm 2 Complete pseudocode for Best-First Search

```
Require:
  value function network $V$,
  policy $\rho_{BFS}$
  predicate of solution SOLVED

  function SEARCH($s_0$)
  $T \leftarrow \emptyset$ {priority queue}
  $T$.PUSH(($V(s_0), s_0$))
  $parents \leftarrow \{\}$
  $seen$.ADD($s_0$) {$seen$ is a set}
  while $0 <$ LEN($T$) and LEN($seen$) $< max\_budget$ do
    $\_, s \leftarrow T$.EXTRACTMAX() {select node with the highest value}
    $actions \leftarrow \rho_{BFS}(s)$
    for $a$ in $actions$ do
      $s' \leftarrow$ ENVSTEP($s, a$)
      if $s'$ in $seen$ then
        continue
      end if
      $seen$.ADD($s'$)
      $parents[s'] \leftarrow s$
      $T$.PUSH(($V(s'), s'$))
      if SOLVED($s'$) then
        {solution found}
        return EXTRACTLOWLEVELTRAJECTORY($s'$, $parents$)
      end if
    end for
  end while
  return False {solution not found}
```

F.2 Monte Carlo Tree Search

Overview Our Monte Carlo Tree Search (MCTS) solver, designed for a single-player setting, is based on the AlphaZero framework (Silver et al., 2018). The high-level workflow of MCTS is illustrated in Figure 35, and detailed pseudocode is provided in Algorithm 3.

The algorithm's operation consists of four primary stages:

- **Selection**: The most promising node is selected using Polynomial Upper Confidence Trees (PUCT), augmented with an exploration weight to strike a balance between exploiting known strategies and investigating new pathways.
- **Expansion**: The selected node is expanded, generating new child nodes that correspond to prospective future actions. This expansion widens the search tree and enables the exploration of various outcomes.
- **Simulation**: Following the AlphaZero approach (Silver et al., 2018), policy and value networks replace traditional simulations. The policy network suggests favorable moves, while the value network predicts their probability of success, directing the algorithm towards beneficial trajectories.
- **Backpropagation**: The insights derived from the networks are used to update node values, improving future decision-making.

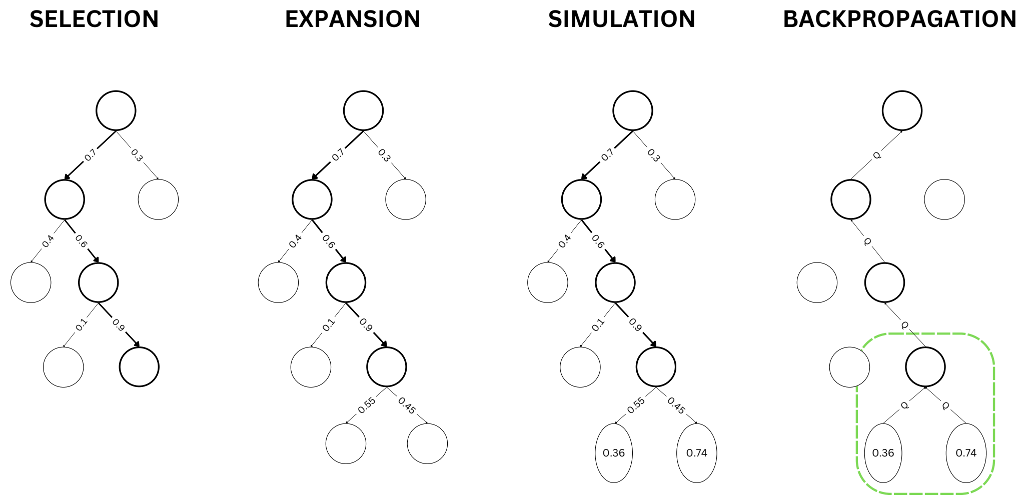


Figure 35: Schematic diagram of the MCTS algorithm in our implementation. Arrows show policy network probabilities and node values are valued network predictions. Q values, calculated via PUCT, integrate these with exploration-exploitation balance.

Hyperparameters In the MCTS algorithm, the parameters were set as follows: sampling temperatures were chosen from [0, 0.5, 1]. The number of steps varied between 200 and 1000, and the number of simulations ranged from 5 to 300. The discount factor and exploration weight were consistently set at 1.

Algorithm 3 MCTS Solver

```
Require:
  Number of simulations: $N_s$
  Discount factor: $\gamma$
  Exploration weight: $c_{\text{puct}}$
  Sampling temperature: $\tau$
  Value function: $V$
  Environment model: $M$
  Initial state: $initial\_state$ from env

  function SEARCH(($initial\_state$))
  $root \leftarrow initial\_state$
  $iteration \leftarrow 0$
  while $iteration < N_s$ do
    $node \leftarrow root$
    while $node$ is not a leaf do
      $node \leftarrow$ selectChild($node$), according to PUCT formula
    end while
    $leaf \leftarrow node$
    Expand the leaf using the environment model $M$, policy $\pi$, value function $V$, and discount factor $\gamma$
    Backpropagate results through the path to update $N, W, Q$
    $iteration \leftarrow iteration + 1$
  end while
  $best\_child \leftarrow$ Sample child of the $root$ according to $\tau$ and $N$
  return action leading to $best\_child$
```

F.3 A* Search

Overview Like Best-First Search, A* prioritizes the exploration of promising nodes. However, A* strategically guides its search by incorporating both the actual cost to reach a node and a heuristic estimate of the remaining distance to the goal. This way it balances the greedy exploitation and conservative exploration. The high-level pseudocode for A* is outlined in Algorithm 4, and the detailed pseudocode is presented in Algorithm 5.

Algorithm 4 Pseudocode for A*

```
while has nodes to expand do
    Take node $N$ with the highest value
    Select children $n_i$ of $N$
    Compute values $v_i$ for the children
    Compute depth $d_i$ for the children
    Add $(n_i, \lambda d_i + v_i)$ to the search tree
end while
```

Heuristic A* guidance is achieved through the following cost function:

$$f(node) = \lambda g(node) + h(node)$$

where:

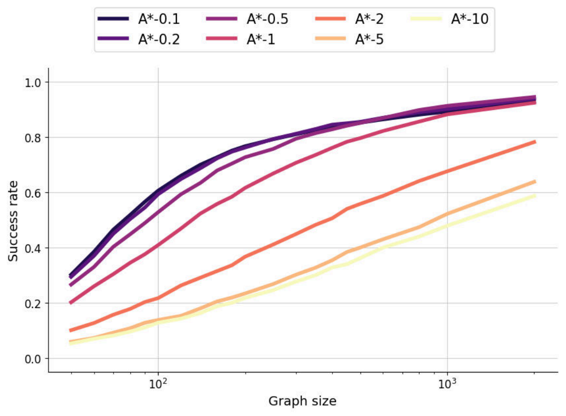
- $g(node)$: The cost to reach *node* from the start state, in our case its depth in the search tree.
- $h(node)$: A heuristic estimate of the cost from *node* to the goal state.
- λ: A scaling factor balancing the influence of actual cost and heuristic estimate.

For heuristic h, we used a value network, like for BestFS (see Appendix F.1). If the heuristic used for A* is *admissible*, i.e. it never overestimates the cost of reaching the goal, A* is guaranteed to find an optimal solution. For instance, if we used $h(node) \equiv 0$, A* would reduce to the Dijkstra algorithm. The heuristic that we learn is not guaranteed to be admissible. Firstly, it estimates the distance according to the demonstrations, which is always an upper bound for the optimal distance. Secondly, the approximation errors introduce additional uncertainty. However, our main focus is on finding any solution, not necessarily an optimal one.

Selecting children During the search, A* maintains a priority queue of nodes to be explored. Similarly to BetsFS (Appendix F.1) for reducing the search tree size, we select the most promising children. At each iteration, the node with the lowest $f(node)$ value is selected for expansion. The algorithm proceeds until the goal state is reached or the computational budget is exceeded.

Hyperparameters The key parameter for A* is the cost weight λ. On the extreme, setting $\lambda = 0$ reduces A* to greedy BestFS, while setting $\lambda = \infty$ makes it equivalent to Breadth-First Search. By tuning that parameter, we control the trade-off between exploration and exploitation of the search.

To tune the depth parameter for our experiments, we grided over values $[0.1, 0.2, 0.5, 1, 2, 5, 10]$. However, usually the best choice was to keep the cost weight low (0.1 or 0.2, see Figure 36). While conservative search allows A* avoid more dead-ends than BestFS (see Figure 5.4), usually greedy steps lead to finding the solution much faster.



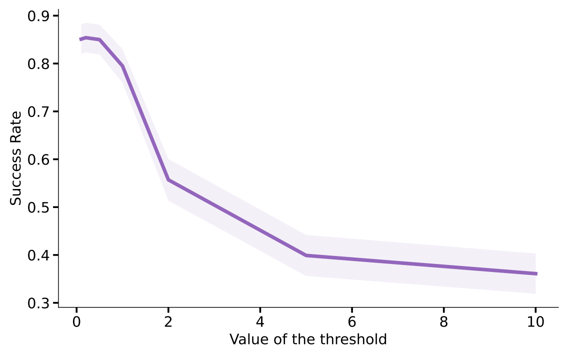


Figure 36: Figures presented above illustrate the impact of depth cost scaling on the overall success rate of the A* algorithm on Sokoban, employing a confidence threshold of 0.85. In most experiments, the smaller the depth scaling factor is, the better is the final success rate. The left figure shows the success rate curves for different choices of cost weight λ, while the right plot compares those variants for a fixed budget of 500 computation nodes.

Algorithm 5 Complete pseudocode for A^* Search

```
Require:
  value function network $V$
  policy $\rho_{BFS}$
  predicate of solution SOLVED
  depth scaling factor $\lambda$

  function SEARCH($s_0$)
  $T \leftarrow \emptyset$ {priority queue}
  $T$.PUSH($(V(s_0), s_0)$)
  $parents \leftarrow \{\}$
  $seen$.ADD($s_0$) {$seen$ is a set}
  while $0 <$ LEN($T$) and LEN($seen$) $< max\_budget$ do
    $\_, s \leftarrow T$.EXTRACTMAX() {select node with the highest value}
    $actions \leftarrow \rho_{BFS}(s)$
    for $a$ in $actions$ do
      $s' \leftarrow$ ENVSTEP($s, a$)
      if $s'$ in $seen$ then
        continue
      end if
      $seen$.ADD($s'$)
      $parents[s'] \leftarrow s$
      $T$.PUSH($(V(s') - \lambda \cdot depth(s'), s')$)
      if SOLVED($s'$) then
        {solution found}
        return EXTRACTLOWLEVELTRAJECTORY($s'$, $parents$)
      end if
    end for
  end while
  return False {solution not found}
```

F.4 kSubS And AdaSubS

Overview AdaSubS is a hierarchical search algorithm designed to solve combinatorial problems by operating on high-level nodes, which represent multiple steps rather than single actions. It employs multiple generators $\mathcal{G}_{k_1}, \mathcal{G}_{k_2}, \ldots, \mathcal{G}_{k_m}$ to generate subsequent subgoals, a value function $\mathcal{V}$ to estimate the distance from a given state to the solution, and a conditional low-level policy $\mathcal{P}$ to execute a series of actions leading from one subgoal to the next. kSubS is a special case of AdaSubS, where only a single generator is used. These methods are introduced and studied in (Czechowski et al., 2021; Zawalski et al., 2023).

Stages The method begins by adding m initial nodes (one per each generator) to a priority queue, where each initial node i is assigned a priority $(k_i, \mathcal{V}(s_0))$. Here, k_i is the length of the generator used during the node's expansion, and $\mathcal{V}(s_0)$ estimates the distance (in low-level actions) between s_0 and the solution. The following steps are repeated until a solution is found or the budget is exhausted:

- **Selection for expansion**: The node $((k, \mathcal{V}(s), s)$ with the highest priority is extracted from the queue. This priority structure ensures that the algorithm prioritizes expanding the longest subgoals whenever possible.
- **Generating subgoals**: The current state s is passed to the selected generator $\mathcal{G}_k$, which produces multiple subgoal propositions represented as states $s_1^*, s_2^*, \ldots, s_p^*$.
- **Verifying reachability**: Since $\mathcal{G}_k$ can produce invalid or unreachable subgoals, each proposed subgoal must be verified. The conditional low-level policy $\mathcal{P}$ begins an iterative process, taking single steps from s towards the proposed subgoal s_j^*. If s_j^* is reached within k steps, the subgoal is accepted, and new high-level nodes $\{\big((k_i, \mathcal{V}(s_j^*)), s_j^*\big)\}_{i\in\{1\ldots m\}}$ are added to the priority queue as potential future subgoals to expand.

For a graphical overview of how AdaSubS works, see Appendix H.

Algorithm 6 Complete pseudocode for Adaptive Subgoal Search

```
Require:
  $C_1$ max number of nodes,
  $V$ value function network,
  $\rho_{k_0}, \ldots, \rho_{k_m}$ subgoal generators,
  SOLVED predicate of solution

  function SOLVE($(s_0)$)
  $T \leftarrow \emptyset$ {priority queue with lexicographic order}
  $parents \leftarrow \{\}$
  for $k$ in $k_0, \ldots, k_m$ do
    $T.push((k, V(s_0)), s_0)$
  end for
  $seen.add(s_0)$ {$seen$ is a set}
  while $0 < \text{len}(T)$ and $\text{len}(seen) < C_1$ do
    $(k, \_), s \leftarrow T.extract\_max()$
    $subgoals \leftarrow \rho_k(s)$
    for $s'$ in $subgoals$ do
      if $s'$ not in $seen$ then
        if IS_VALID(s, s') then
          $seen.add(s')$
          $parents[s'] \leftarrow s$
          for $k$ in $k_0, \ldots, k_m$ do
            $T.push((k, V(s')), s')$
          end for
          if SOLVED(s') then
            return EXTRACTLOWLEVELTRAJECTORY(s', parents)
          end if
        end if
      end if
    end for
  end while
  return False
```

F.5 HIPS And HIPS-ε

Here we show a pseudocode for HIPS and HIPS-ε methods. For details see Alg. 7

Algorithm 7 Complete pseudocode for HIPS with BestFS-PHS* and VQ-VAE

```
Require:
  $C_1$ max number of nodes,
  $VAE$ Variational Autoencoder for subgoal generation,
  SOLVED predicate of solution,
  $\epsilon$ exploration parameter for balancing,
  $V$ value function for PHS* cost estimation

  function EXTENDED_HIPS_SOLVE(($s_0$))
  Initialize search data structures, including priority queues.
  $seen.add(s_0)$ {Track seen states}
  while search conditions are met do
    Use PHS* search strategy to select a state $s$.
    Generate subgoals $subgoals \leftarrow VAE(s)$.
    for each $s'$ in $subgoals$ do
      if $s'$ not seen and is valid then
        Evaluate $s'$ using $V$ for PHS* cost.
        Update priority queue based on PHS* cost.
        if SOLVED($s'$) then
          return Construct solution path.
        end if
      end if
    end for
  end while
  return False {Solution not found}
```

G Wall Times

In our experiments, we focus on measuring the search budget in terms of the number of visited states before finding the solution. However, it is also important to consider the total running time for completeness.

Subgoal methods introduce computational overhead. However, we note that each low-level method calls policy and value function once in every visited state, and similarly, subgoal methods also call policy and value once in every visited state. The only additional computation in subgoal methods comes from invoking the subgoal generator, which occurs in a fraction of the nodes explored. In each experiment, all methods share exactly the same heuristic function and use policies of equal size. As a result, the Complete Search Budget metric should be closely aligned with computational cost.

We opted to focus on a budget metric that is hardware-independent, reproducible, and widely applicable, ensuring that our results can serve as a reference point for future research. The Complete Search Budget answers the question "How many states must be explored before finding a solution?" rather than "How long does it take to find a solution?". These are slightly different questions, but both are relevant when assessing planner quality.

We acknowledge that we did not optimize the implementation for runtime efficiency, instead opting for the architectures used by (Czechowski et al., 2021) and (Zawalski et al., 2023) rather than optimizing computational complexity. Additionally, measuring wall-clock time introduces confounding factors, such as a bug in Hugging Face's beam search implementation that prevents decoding parallelization, introducing bias against subgoal methods.

For completeness, we report wall-clock times of each method in Table 3.

	ρ-BestFS	ρ-A*	ρ-MCTS	kSubS	AdaSubS
Rubik	26	26	153	214	96
INT	1997	1985	-	1444	1999
Sokoban	34	36	59	125	123
NPuzzle	27	32	29	40	39

Table 3: Comparison of evaluation time of search algorithms. The values express the total time of solving 500 instances, in minutes.

While all methods perform with similar runtime in the INT environment, subgoal methods generally require more time during evaluation in most other experiments. However, even the largest observed differences in evaluation time mildly affect the main conclusions. For example, the robustness of subgoal methods to value noise remains evident.

H Hierarchical Search

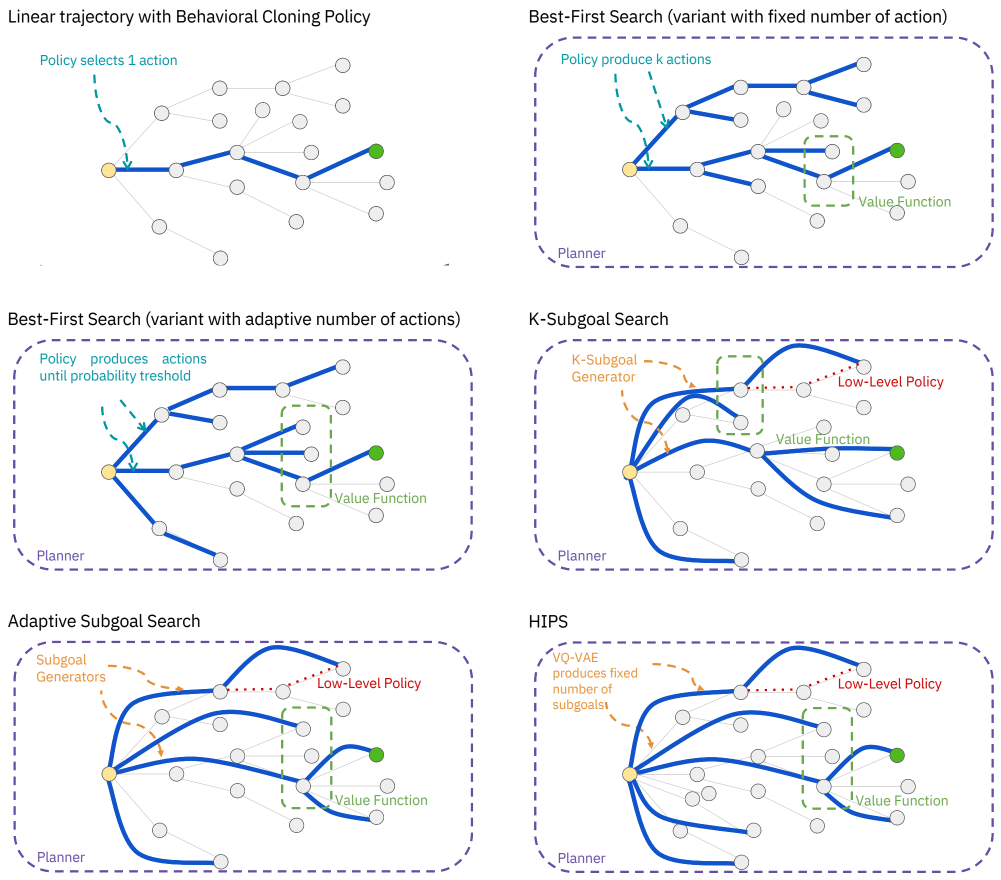


Figure 37: Overview of the search methods under consideration, accompanied by illustrative examples depicted in various plots for each method. Specifically, straight blue lines are utilized to represent low-level actions that occur within the search space. In contrast, long skip connections are used to symbolize subgoals within the search process.

I Further Discussion On HIPS Results

HIPS and HIPS-ε (Kujanpää et al., 2023a;b) are recent hierarchical search algorithms proposing to generate subgoals with variational autoencoders. We attempted to use HIPS and HIPS-ε in greedy and prior-informed variations, and for all HIPS methods, the cost of inference was prohibitively high.

To compare these methods, we used A*-generated data from HIPS papers, in contrast to all other experiments (which use data generated by us).

Our evaluation, illustrated in 38, shows that HIPS uses 100x more low-level nodes in search than comparable subgoal search methods and baselines - despite relatively similar subgoal efficiency as calculated in relevant papers. These findings informed our decision not to evaluate HIPS in the rest of the paper.

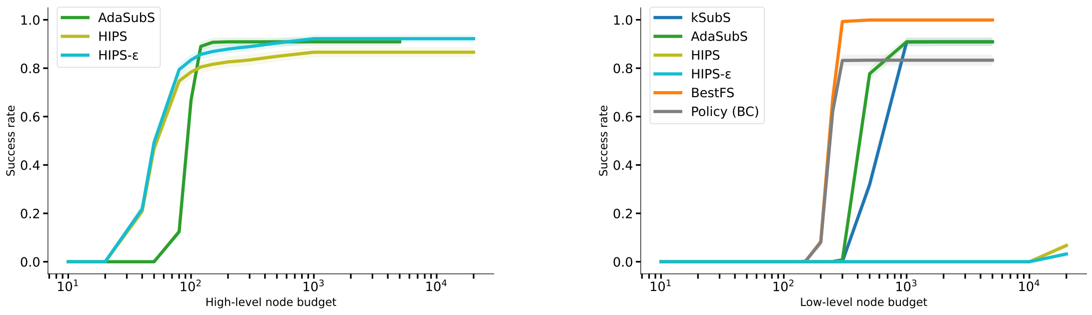


Figure 38: A comparison of high-level and low-level node budgets for considered methods: HIPS, subgoal search methods, and baselines on N-Puzzle. The low-level node budget represents the number of all states that have ever been visited during the search. The bimodal distribution indicates that HIPS methods use disproportionately (over 100x) more low-level nodes than comparable subgoal search methods and baselines. This directly translates to prohibitively slow solving time.

J Common Pitfalls In Hierarchical Search evaluations

In this study, one of our primary goals is to identify common but often overlooked pitfalls in evaluating hierarchical search methods, which can lead to misleading conclusions. Based on our findings, we propose a set of guidelines that help ensure meaningful and consistent comparisons across different methods. We observed that the nature of hierarchical search makes it easy, whether intentionally or not, to present results in a way that favors certain methods, often without readers being aware. In this section, we present key insights on this issue, with an emphasis on the following evaluation guidelines:

- Report results using a *complete search budget*.
- Include ρ-BestFS with a confidence threshold as a baseline.
- Ensure careful tuning of the confidence threshold.
- Use up-to-date code for running experiments.

J.1 Complete Search Budget

We define the performance metric in terms of *success rate*, which is the percentage of problem instances solved within a specified *complete search budget*. This budget refers to the total number of states visited during the search process. For hierarchical methods, this includes both the subgoals generated and the states visited by the low-level policies connecting those subgoals.

Reporting the *complete search budget* is crucial, as opposed to the *sparse search budget*, which counts only the high-level nodes in the search tree. As discussed in Appendix I, Kujanpää et al. (2023a) rely on the sparse search budget for their evaluations. This creates a misleading impression that HIPS outperforms low-level baselines, while in reality, it requires significantly more computational effort to solve the same problems.

To illustrate this issue, consider a simple environment where an agent must navigate a 100x100 empty room to reach a goal on the opposite side. In this case, a hierarchical method may require only a single subgoal – directly corresponding to the goal state – while a low-level method, even if following the optimal path, would require at least 100 steps. A sparse search budget would misleadingly indicate that the hierarchical method solves the task in one step, while the low-level approach requires 100 steps, implying a 100x higher cost. However, both methods traverse the same path, making this comparison inaccurate. Using the *complete search budget*, both methods would be assigned the same cost, providing a much more meaningful comparison.

This issue arises in practical settings as well. Figure 40 compares subgoal methods and low-level BestFS on the Sokoban environment. The dashed line represents the same runs but evaluated with the sparse search budget instead of the complete search budget. For BestFS, both budget measures are equivalent. The figure clearly demonstrates that while kSubS and ρ-BestFS visit a similar number of states to solve an instance, the sparse search budget falsely amplifies the difference between the two methods.

J.2 Baselines

A common evaluation practice in hierarchical search studies is to compare hierarchical methods against the search algorithm used as the planner (Czechowski et al., 2021; Zawalski et al., 2023; Kujanpää et al., 2023a;b). While this is generally a good approach, it is critical to ensure that baseline methods are properly tuned to allow for fair comparisons.

Our study shows that the most effective low-level method is ρ-BestFS with a confidence threshold. This simple greedy search often performs significantly better than other low-level methods and, in some cases, is competitive with subgoal methods. However, if we were to follow prior works such as (Czechowski et al., 2021; Zawalski et al., 2023) and restrict our comparisons to variants of BestFS that select a fixed number of actions in each node expansion, without employing a confidence threshold (see Appendix F.1 for detailed definitions and analysis), we would artificially widen the gap between BestFS and subgoal methods. As noted in Appendix F.1, the performance of ρ-BestFS is highly sensitive to the confidence threshold, and

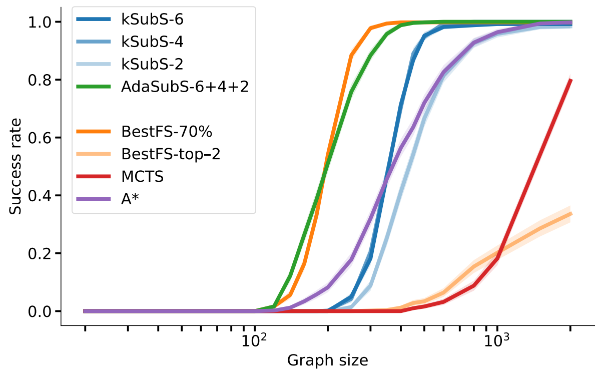


Figure 39: Solving the Rubik's Cube. The light orange line represents the best-performing variant of BestFS that selects a fixed number of actions for each expansion. The solid orange line represents BestFS with actions confidence threshold, which is much more efficient.

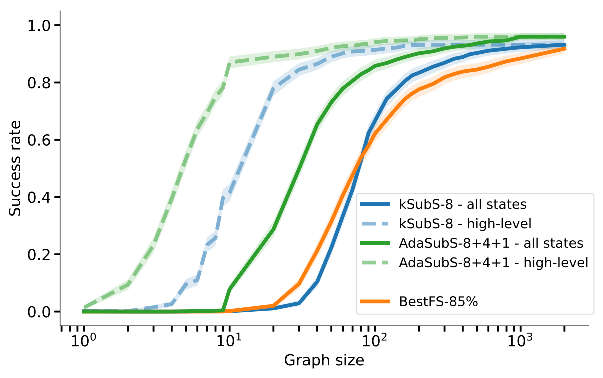


Figure 40: Solving Sokoban. Solid lines correspond to using *complete search budget* as the search tree size metric. Dashed lines correspond to the same runs, but using *sparse search budget* as the search tree size metric. For BestFS, both methods are equivalent.

proper tuning is essential. Nevertheless, we advocate for using ρ-BestFS with a confidence threshold as a standard baseline in evaluations of hierarchical methods.

J.3 Code Quality

While our results generally align with the findings of (Czechowski et al., 2021; Zawalski et al., 2023), we observed some notable differences. Most strikingly, when components were trained on reverse random shuffles of the Rubik's Cube, our models demonstrated significantly better performance. In particular, (Zawalski et al., 2023) reports that both kSubS and AdaSubS substantially outperform ρ-BestFS. However, in our experiments, these methods perform similarly, with only minor differences between them (see Figure 41).

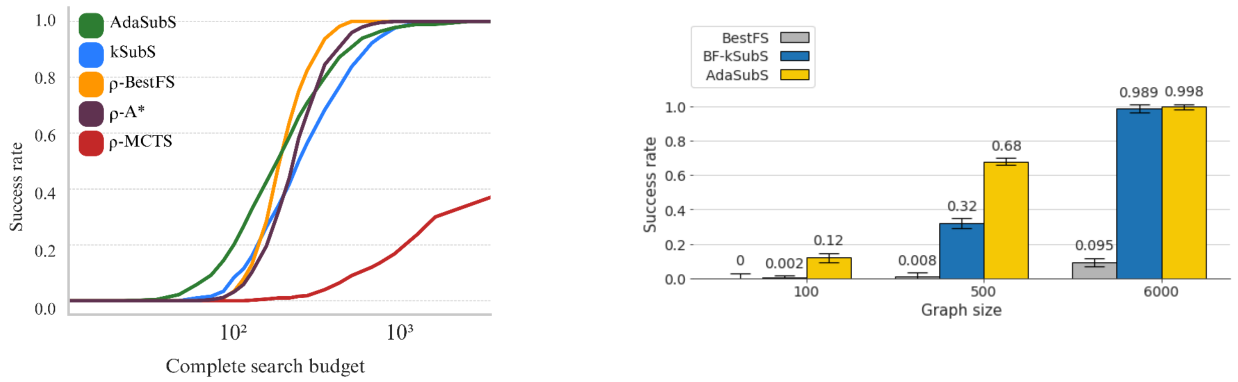


Figure 41: Solving the Rubik's Cube. Components are trained on reverse random shuffles. The left chart present our results, while the right presents results of the same experiment from (Zawalski et al., 2023).

For this study, we re-implemented all algorithms from scratch, using up-to-date libraries and carefully tuning hyperparameters. Our experiments revealed that low-level methods are highly sensitive to the quality of the value function, whereas subgoal-based methods are more resilient (Section 5.2). We hypothesize that the discrepancy in performance compared to (Czechowski et al., 2021; Zawalski et al., 2023) may stem from insufficient training of the value function in their implementation, leading to the observed performance gap.

Using the original implementations of kSubS and AdaSubS, which is a common practice, would replicate the same limitation. This shows the importance of re-implementing algorithms independently and carefully tuning their components, ensuring that evaluations are not biased by potential shortcomings in the original implementations.

K Proof Of The Search Advancement formula

Theorem 3 (Search advancement formula, complete statement)**.** *Let $g_k : S \to \mathcal{P}(S)$ be a stochastic k-subgoal generator that, given a state $s \in S$ samples a set of b subgoals $\{s_i\}$ such that the distances $d(s_i, s)$ are independent, uniformly distributed in the interval $[-k; k]$. Let $V : S \to \mathbb{R}$ be a value function with approximation error uniformly distributed in the interval $[-\sigma; \sigma]$.*

Then, after n iterations of search, the expected total progress toward the goal is:

$$\mathbb{E}_{Adv} = \frac{nb}{4\sigma k} \int_{-k}^{k} x \left(\int_{-\sigma}^{\sigma} \tilde{u}(x+h)^{b-1} \mathrm{d}h \right) \mathrm{d}x, \tag{3}$$

where $\tilde{u}(x)$ is CDF of the sum of two uniform variables $U(-k,k) + U(-\sigma,\sigma)$. Additionally, if we approximate that sum as $U(-k-\sigma, k+\sigma)$, we get

$$\mathbb{E}_{Adv} \approx \frac{n\left((k+\sigma)^b(bk^2 + bk\sigma - 2k\sigma - 2\sigma^2) + \sigma^b(2k\sigma + bk\sigma + 2\sigma^2) - k^b(bk^2)\right)}{(b+1)(b+2)k\sigma(k+\sigma)^{b-1}} \tag{4}$$

Proof. Let $A_1, \ldots, A_b$ be independent and identically distributed (i.i.d.) random variables sampled from $U(-k,k)$, and let $B_1, \ldots, B_b$ be i.i.d. random variables sampled from $U(-\sigma,\sigma)$. Denote the CDF of the sum $A_i + B_i$ as $\tilde{u}(x)$, and its corresponding probability density function (PDF) as $p(x) = \tilde{u}'(x)$. Let $I = \arg\max_i(A_i + B_i)$.

We now define the cumulative likelihood of selecting the largest sum among the subgoals:

$$CLS(x) = \mathbb{P}\left(\forall_{1 \le i \le b}\, A_i + B_i < x\right).$$

Since the A_i's and B_i's are independent, it follows that $CLS(x) = \tilde{u}(x)^b$, which represents the cumulative distribution of the largest sum $A_i + B_i$. Differentiating this expression gives the PDF of the largest sum:

$$PLS(x) = CLS'(x) = b \cdot \tilde{u}(x)^{b-1} \cdot p(x).$$

Now, consider the event that $A_I = x$, which is equivalent to the event that the maximum $\max_i(A_i + B_i) = x + h$ for some $h \in [-\sigma, \sigma]$ and $B_I = h$. Given that $\max_i(A_i + B_i) = x + h$, there are $p(x+h) \cdot 4\sigma k$ possible values of B_I, since $A_I \in [-k,k]$ and $B_I \in [-\sigma,\sigma]$. Therefore, the PDF of this variable is

$$q(x) = \int_{-\sigma}^{\sigma} \frac{PLS(x+h)}{p(x+h) \cdot 4\sigma k} \,\mathrm{d}h = \int_{-\sigma}^{\sigma} \frac{b \cdot \tilde{u}(x+h)^{b-1}}{4\sigma k} \,\mathrm{d}h.$$

Thus, the expected value of A_I, which represents the progress in each step, is given by

$$\mathbb{E}[A_I] = \int_{-k}^{k} x q(x) \,\mathrm{d}x = \frac{b}{4\sigma k} \int_{-k}^{k} x \left(\int_{-\sigma}^{\sigma} \tilde{u}(x+h)^{b-1} \,\mathrm{d}h \right) \mathrm{d}x.$$

If we model the search process as advancing to the best subgoal in each iteration, the total expected progress after n iterations is

$$\mathbb{E}_{Adv} = n\mathbb{E}[A_I] = \frac{nb}{4\sigma k} \int_{-k}^{k} x \left(\int_{-\sigma}^{\sigma} \tilde{u}(x+h)^{b-1} \,\mathrm{d}h \right) \mathrm{d}x.$$

Finally, by approximating the PDF $p(x) \approx \frac{1}{2k+2\sigma} \mathbb{1}_{[-k-\sigma, k+\sigma]}$, and substituting this approximation into the previous expression, we arrive at the closed-form approximation:

$$\mathbb{E}_{Adv} \approx \frac{n\left((k+\sigma)^b(bk^2 + bk\sigma - 2k\sigma - 2\sigma^2) + \sigma^b(2k\sigma + bk\sigma + 2\sigma^2) - k^b(bk^2)\right)}{(b+1)(b+2)k\sigma(k+\sigma)^{b-1}}.$$

□

L Proof Of The Densification Of The Action Space Theorem

In Section 5.3, we showed experimentally that both in the mathematical INT environment and Rubik's Cube with multiplied action space the advantage of subgoal methods is significant. We attributed those benefits to the ability of subgoal methods to use states as actions and the reduced diversity in low-level search. And indeed, we can prove in general that as the action space gets more complex, the diversity of top actions drops.

To give an illustrative example, in the Rubik's Cube experiment, to model the increasingly complex action space, for an arbitrary state we can view the training data as a ground-truth density function f over an interval $[0, 1]$, that is split evenly between the actions (i.e. into 12 intervals of length 1/12). Then, we can define arbitrarily dense action spaces A_n consisting of n points distributed evenly in the domain. For instance, A_{12} corresponds to the standard Rubik's Cube action space, while A_{1200} corresponds to the variant multiplied 100 times. Our theorem confirms that the actions selected by the policy gets less diverse as the complexity of the action space increases, up to the extreme of converging to a single point as n approaches infinity. In practice, it is even more general, since the data-driven action distribution f may also model smooth interpolation between actions.

While this is rather intuitive when the learned distributions are perfect, it may seem that approximation errors, induced both by the limited training data and the policy network can actually improve diversity. We show that the result holds even in presence of arbitrarily large approximation errors, which is a bit counter-intuitive.

Formally, the theorem is as follows:

Theorem 4 (Densification of the action space)**.** *Fix any state s from the state space S. Let $f : A \to [0, 1]$ be the action distribution induced by the data-collecting policy for the state s. Assume that f is continuous and has a unique maximum. For clarity, assume $A = [0, 1]$.*

Consider a sequence of increasingly dense discrete action spaces $A_n := \{i/n\}_{i=0}^{n} \subset A$. Let $\rho_n : S \times A_n \to [0, 1]$ be a family of policies that learn the distribution $f|_{A_n}$ over actions, with uniform approximation error $U(-E, E)$, where $E \in \mathbb{R}_+$. Let r_n be the range of the top K actions according to the probabilities estimated by ρ_n. Then

$$\lim_{n\to\infty} \mathbb{E}[r_n] = 0.$$

Intuitively, this theorem states that as the action space become more dense and complex, the actions sampled for search become increasingly less diverse, which strongly impedes successful planning. Note that this analysis is strictly more general than the experiment in Section 5.3 with the Rubik's Cube environment, where we simply copied the available actions. Here we model the complexity by adding dense intermediate actions, which leads to a similar conclusion.

While we assume a one-dimensional action domain for clarity, it is straightforward to generalize the proof to cover arbitrarily high-dimensional action spaces.

Firstly, we shall prove the following key lemma.

Lemma 1. *Let $f : [0, 1] \to \mathbb{R}$ be a continuous function with a unique maximum. Let $\{a_n\}$ be a partition of the interval $[0, 1]$ into n uniformly spaced points, i.e., $a_{n,i} = \frac{i}{n}$ for $i = 0, 1, \dots, n$. Define $e_{n,i}$ as i.i.d. samples from a uniform distribution $U(-E, E)$. For a fixed n, let $r_n \in \mathbb{R}$ denote the smallest interval length such that the points in $\{a_n\}$ corresponding to the top K values of $f(a_{n,i}) + e_{n,i}$ are contained within this interval. Then*

$$\lim_{n\to\infty} \mathbb{E}[r_n] = 0.$$

Proof. Define $p_{n,i,k}$ as the probability that $f(a_{n,i}) + e_{n,i}$ is the k-th highest value among all points in $\{a_n\}$. Let m be the unique point such that $f(m)$ is maximal. Without loss of generality, we may assume that $m = 0$.

Let $d_{n,k}$ denote the expected distance of the k-th highest point from 0, expressed as

$$d_{n,k} := \sum_{i=0}^{n} p_{n,i,k} a_{n,i}.$$

For sufficiently large n, it holds that $r_n \leq d_{n,1} + \ldots + d_{n,K} \leq K d_{n,K}$. Thus, it suffices to prove that $\lim_{n\to\infty} d_{n,K} = 0$.

Fix $\alpha \in (0,1)$ such that $f(a_{n,\alpha n}) \geq f(a_{n,\alpha' n})$ for each $\alpha' > \alpha$. Since f is continuous and $m = 0$ is the unique maximum of f, there exist such α arbitrarily close to 0. Let $q_{n,\alpha}$ be the probability that $f(a_{n,\alpha n}) + e_{n,\alpha n}$ is among the top K values. Since m is a unique maximum, there exists $0 < \beta < \alpha$ such that $f(a_{n,\beta n}) > f(a_{n,\alpha n})$. Therefore, if at least K points $a_{n,i}$ with $i/n < \beta$ satisfy $e_{n,i} > E - (f(a_{n,\beta n}) - f(a_{n,\alpha n}))$, then $f(a_{n,\alpha n}) + e_{n,\alpha n}$ cannot be among the top K. The probability of this event is a strict upper bound on $q_{n,\alpha}$.

The events $e_{n,i} > E - (f(a_{n,\beta n}) - f(a_{n,\alpha n}))$ are pairwise independent, each occurring with probability

$$c := \frac{f(a_{n,\beta n}) - f(a_{n,\alpha n})}{2E} > 0.$$

For sufficiently large n, the probability that at most K of the βn trials succeed is bounded by

$$1 - K\binom{\beta n}{K}(1-c)^{\beta n}.$$

Using the asymptotic behavior of binomial coefficients and exponential terms, it follows that

$$\lim_{n\to\infty} n^2 q_{n,\alpha} = 0, \tag{5}$$

with convergence that is exponential.

Using the definition of $d_{n,K}$, decompose it as

$$d_{n,K} = \sum_{i=0}^{n} p_{n,i,K} a_{n,i} = \sum_{i=0}^{\alpha n} p_{n,i,K} a_{n,i} + \sum_{i=\alpha n}^{n} p_{n,i,K} a_{n,i}.$$

For $i \geq \alpha n$, since we know that $f(a_{n,\alpha n}) \geq f(a_{n,\alpha' n})$ for each $\alpha' > \alpha$, we can bound $p_{n,i,K}$ by $p_{n,\alpha n,K}$ for sufficiently large n. Therefore

$$\sum_{i=\alpha n}^{n} p_{n,i,K} a_{n,i} \leq (1-\alpha) n p_{n,\alpha n,K}.$$

Since $p_{n,\alpha n,K} \leq q_{n,\alpha}$, it follows that

$$(1-\alpha) n^2 p_{n,\alpha n,K} \leq (1-\alpha) n^2 q_{n,\alpha}.$$

According to Equation 5, this term converges to 0.

For $i \leq \alpha n$, observe that $a_{n,i} < \alpha$ and the probabilities $p_{n,i,K}$ sum to at most 1. Thus

$$\sum_{i=0}^{\alpha n} p_{n,i,K} a_{n,i} \leq \alpha.$$

Combining these bounds, we have

$$\lim_{n\to\infty} d_{n,K} \leq \alpha.$$

Since $\alpha > 0$ was an arbitrarily small constant, it follows that $\lim_{n\to\infty} d_{n,K} = 0$.

By the relation $r_n \leq K d_{n,K}$ and the fact that $\lim_{n\to\infty} d_{n,K} = 0$, we conclude that

$$\lim_{n\to\infty} \mathbb{E}[r_n] = 0.$$

□

Now, Theorem 4 is a straightforward implication of Lemma 1, applied to the sequence of policies ρ_n and increasingly dense action spaces A_n.

M Comparison with DeepCubeA

In contrast to the general-purpose search methods and pre-defined heuristics examined in our main study, DeepCubeA (McAleer et al., 2019) takes a different approach: it learns a value function and heuristic directly through deep reinforcement learning. This allowed DeepCubeA to successfully solve the Rubik's Cube without relying on human-provided knowledge. To provide a more complete picture of the performance landscape, and to understand the relative strengths of learned versus pre-defined heuristics, we include a comparison with DeepCubeA.

DeepCubeA employs Iterative Deepening A* (IDA*) as its core search algorithm. IDA* is a variant of A* that performs a series of depth-first searches with increasing cost thresholds. In each iteration, it explores nodes in a depth-first manner, but only up to a maximum cost defined by *f(node) = g(node) + h(node)*, where *g(node)* is the path cost (depth) and *h(node)* is the heuristic estimate of the remaining cost. If a solution is not found within the current threshold, the threshold is increased, and the search restarts. This process continues until a solution is found or a resource limit is reached.

While IDA* guarantees finding an optimal solution (given an admissible heuristic), it can revisit the same nodes multiple times across iterations, leading to redundant computations. A*, as described in Section 5, maintains an open list of all explored nodes, avoiding this redundancy. Because A* explores all nodes up to a given cost before expanding nodes with higher costs, and given that we are primarily concerned with finding any solution rather than necessarily the optimal solution, A* provides a more efficient exploration strategy for our analysis, and effectively majorizes the behavior of IDA*.

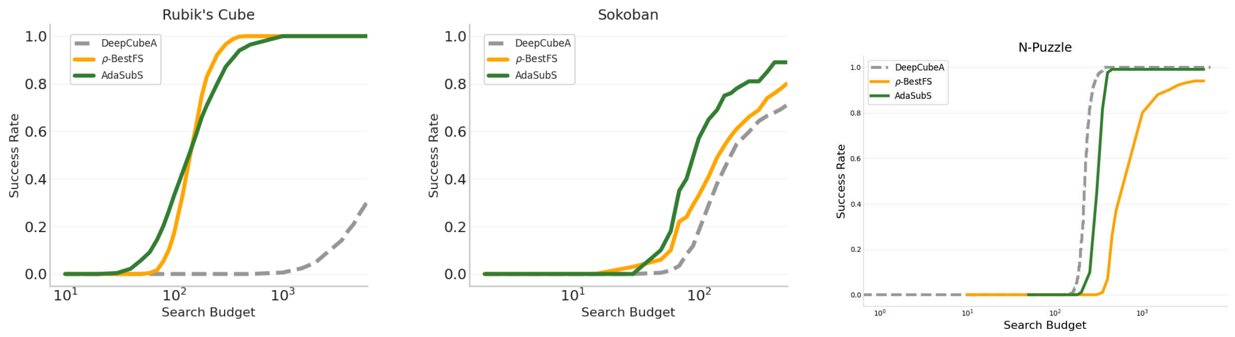


Figure 42: Comparison with DeepCubeA

Figure 42 presents a comparison of methods used in our study (hierarchical AdaSubS and low-level ρ-BestFS) with DeepCubeA – a well-established algorithm that solved the Rubik's Cube with deep learning and tree search, without human knowledge. The plots show evaluation in Rubik's Cube (left), Sokoban (middle), and N-Puzzle (right). The performance of DeepCubeA is weaker or on-par with the methods that we analyze in the paper.

The takeaway from this comparison is twofold. Firstly, performance of our implementations is competitive with well-established general-purpose solvers. Secondly, it is hard to understand the relation between search algorithms if they use different heuristics for solving. Hence, we stress that in each experiment presented in the main paper, all methods share the same value function to ensure a fair comparison.