Co to jest SI?

"The exciting new effort to make computers think ... *machines with minds*, in the full and literal sense" (Haugeland, 1985) "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning ..." (Bellman, 1978)	"The study of mental faculties through the use of computational models" (Charniak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act" (Winston, 1992)
"The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil, 1990) "The study of how to make computers do things at which, at the moment, people are better" (Rich and Knight, 1991)	"A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes" (Schalkoff, 1990) "The branch of computer science that is concerned with the automation of intelligent behavior" (Luger and Stubblefield, 1993)

1. Systemy symulujące ludzkie myślenie.
2. Systemy symulujące racjonalne myślenie.
3. Systemy symulujące ludzkie działanie.
4. Systemy symulujące racjonalne działanie.

- Systemy symulujące ludzkie myślenie: jak ludzie myślą (psychologia poznawcza).
- Systemy symulujące ludzkie działanie: jak ludzie działają (język naturalny, reprezentacja wiedzy, automatyczne wnioskowanie i uczenie się, rozpoznawanie obiektów).
- Systemy symulujące racjonalne myślenie: jak powinno się racjonalnie myśleć (logika).
- Systemy symulujące racjonalne działanie: jak powinno się racjonalnie działać (logika, teoria podejmowania decyzji, działanie w warunkach niepełnej informacji).

Zalety rozważania SI jako konstruowanie systemów działających racjonalnie

- Pojęcie działania jest ogólniejsze od pojęcia myślenia.
- Pojęcie "racjonalny" jest lepiej określone niż pojęcie "ludzki".

Racjonalnie działający obiekt = AGENT

Podstawy SI

- Filozofia.
- Matematyka (twierdzenie Gödla).
- Psychologia.
- Informatyka (złożoność obliczeniowa, problemy NP-zupełne).
- Lingwistyka.

Historia SI

1. Początki (1943-1956)

   (a) W. McCulloch & W. Pitts (1943).
   Pierwszy model sieci neuronowej do symulacji działania mózgu.

   (b) A. Turing (1950).
   Artykuł: Computing Machinery and Intelligence. *Mind*, 59.

   (c) A. Newell & H. Simon (1952).
   Logic Theorist – pierwszy program dowodzący twierdzenia.

   (d) J. McCarthy (1956).
   Termin "sztuczna inteligencja".

2. Wczesny entuzjazm (1952-1969)

   (a) A. Samuel (1952).
   Samouczący się program grający w warcaby.

   (b) J. McCarthy (1958). Język LISP. "Advise Taker" – pierwszy program wykorzystujący bazę wiedzy o świecie.

   (c) A. Newell & H. Simon (1961). Program "GPS" (General Problem Solver). Pierwszy program rozwiązujący problemy przy użyciu technik typowych dla ludzi.

   (d) J. Robinson (1965) Metoda rezolucji.

   (e) C. Green (1969). Podstawy programowania w logice.

3. Czas refleksji (1966-1970)

   (a) Systemy symulujące inteligentne działanie wymagają dużej wiedzy o świecie.

   (b) Posiadanie algorytmu nie zawsze gwarantuje znalezienia rozwiązania. (Złożoność obliczeniowa.)

4. Systemy z wiedzą o świecie (1969-1979)

   (a) Pierwsze systemy doradcze. DENDRAL (analiza chemiczna), MYCIN (infekcje bakteryjne).

   (b) System konwersacyjny LUNAR. Odpowiadał na pytania dotyczące próbek znalezionych przez stację Apollo.

   (c) Nowe formalizmy reprezentacji wiedzy: ramy i sieci semantyczne.

5. Przemysł SI (1980-1988)

   (a) Pierwszy komercyjny system doradczy, R1 (McDermott, 1982). Konfigurował komputery firmy DEC. Szacuje się, że przynosił zysk rzędu 40 mln $ rocznie.

   (b) Japoński projekt "komputery piątej generacji".

6. Ostatnie lata (1980 –)

   (a) Wnioskowanie niemonotoniczne.

   (b) Wnioskowanie probabilistyczne.

   (c) Użycie sieci neuronowych i algorytmów genetycznych do automatyzacji uczenia się.

   (d) Formalizacja jednocześnie działających agentów.

7. Stan dzisiejszy

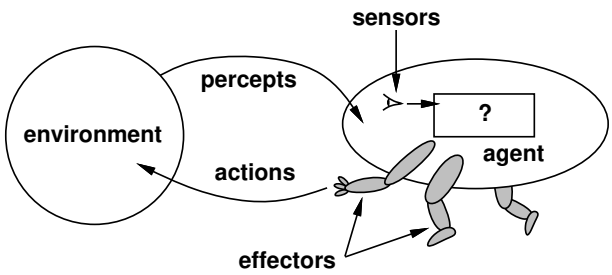
   (a) DEEP BLUE – program szachowy o współczynniku FIDE 2700.

   (b) PEGASUS – system analizy i syntezy mowy.

   (c) MARVEL – system do obróbki danych otrzymywanych ze stacji kosmicznych.

   (d) PROMETHEUS – system wspomagający pracę kierowcy.

RACJONALNY AGENT



RACJONALNOŚĆ $\neq$ NIEOMYLNOŚĆ !

- Miara skuteczności (ang. performance measure): jak skuteczny jest agent.
- Ciąg obserwacji (ang. percept sequence): wszystkie dotychczasowe obserwacje agenta.
- Wiedza o środowisku.
- Repertuar działań (akcji).

Idealnie racjonalny agent: Idealnie racjonalny agent powinien wykonać działanie prowadzące do maksymalizacji miary skuteczności, w oparciu o dany ciąg obserwacji i wiedzę o środowisku, w którym działa.

Jak wyspecyfikować racjonalnego agenta?

Podając odwzorowanie ze zbioru ciągów obserwacji w zbiór dopuszczalnych działań.

Uwaga: Powyższe odwzorowanie nie musi być podane w postaci tablicy.

Przykład: Agent obliczający pierwiastek kwadratowy z liczby dodatniej = prosty program.

Autonomiczność: wykorzystywanie obserwacji przy wyborze odpowiedniego działania.

Struktura racjonalnego agenta

AGENT = ARCHITEKTURA + PROGRAM

Przykłady agentów

Agent Type	Percepts	Actions	Goals	Environment
Medical diagnosis system	Symptoms, findings, patient's answers	Questions, tests, treatments	Healthy patient, minimize costs	Patient, hospital
Satellite image analysis system	Pixels of varying intensity, color	Print a categorization of scene	Correct categorization	Images from orbiting satellite
Part-picking robot	Pixels of varying intensity	Pick up parts and sort into bins	Place parts in correct bins	Conveyor belt with parts
Refinery controller	Temperature, pressure readings	Open, close valves; adjust temperature	Maximize purity, yield, safety	Refinery
Interactive English tutor	Typed words	Print exercises, suggestions, corrections	Maximize student's score on test	Set of students

Szkielet programu symulującego agenta:

```
function SKELETON-AGENT(percept) returns action
  static: memory, the agent's memory of the world

  memory ← UPDATE-MEMORY(memory, percept)
  action ← CHOOSE-BEST-ACTION(memory)
  memory ← UPDATE-MEMORY(memory, action)
  return action
```

Program symulujący agenta nie powinien sprowadzać sie do przeglądania tablicy:

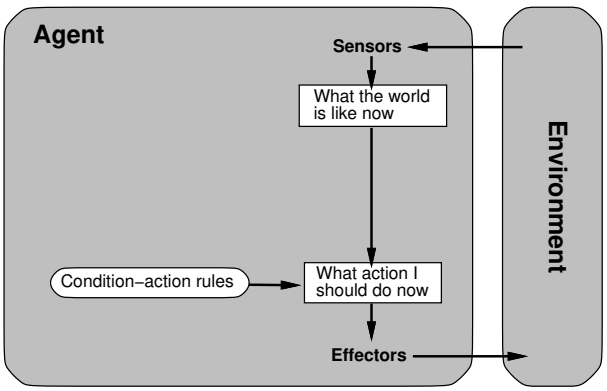
```
function TABLE-DRIVEN-AGENT(percept) returns action
  static: percepts, a sequence, initially empty
          table, a table, indexed by percept sequences, initially fully specified

  append percept to the end of percepts
  action ← LOOKUP(percepts, table)
  return action
```

Reguła produkcyjna:

if światło czerwone **then** stój.

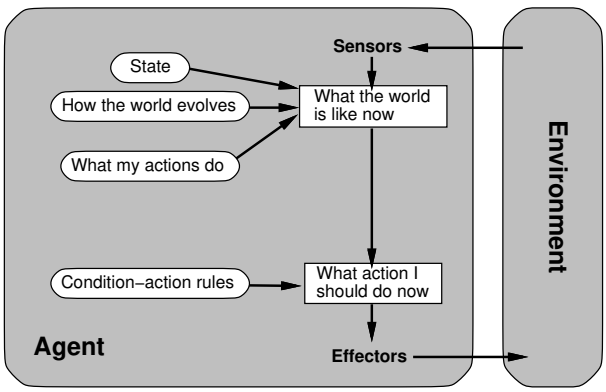
Rodzaje agentów

1. Prosty agent reaktywny: Reaguje wyłącznie na bieżącą obserwację. Nie przechowuje aktualnego stanu świata.



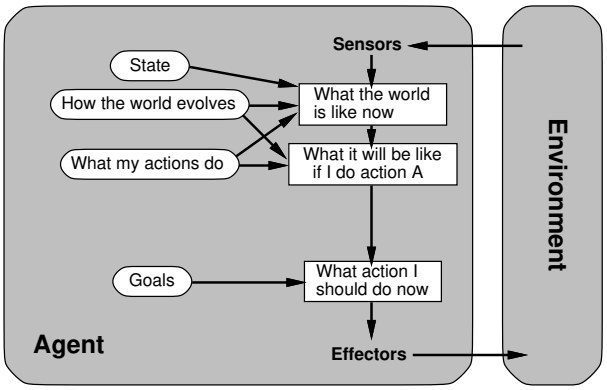

```
function SIMPLE-REFLEX-AGENT(percept) returns action
  static: rules, a set of condition-action rules

  state ← INTERPRET-INPUT(percept)
  rule ← RULE-MATCH(state, rules)
  action ← RULE-ACTION[rule]
  return action
```


2. Agent reaktywny: Reaguje na bieżącą obserwację w oparciu o aktualny stan świata.

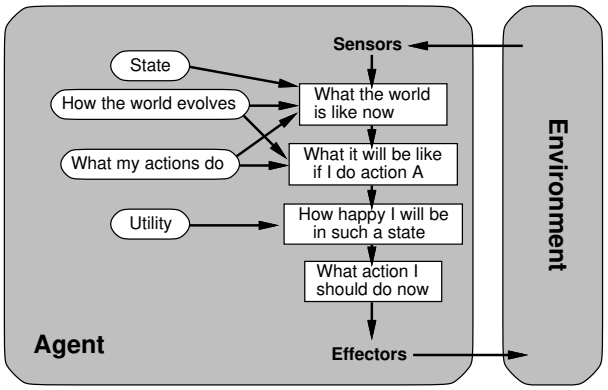



```
function REFLEX-AGENT-WITH-STATE(percept) returns action
  static: state, a description of the current world state
          rules, a set of condition-action rules

  state ← UPDATE-STATE(state, percept)
  rule ← RULE-MATCH(state, rules)
  action ← RULE-ACTION[rule]
  state ← UPDATE-STATE(state, action)
  return action
```

3. Agent realizujący cel: konstruuje plan w celu osiągnięcia pewnego celu.



4. Agent optymalny: realizuje pewien cel w sposób optymalny.



Własności środowisk

- Dostępne — niedostępne (accessible — inaccessible).
- Deterministyczne — niedeterministyczne (deterministic — nondeterministic).
- Modularne — niemodularne (episodic — nonepisodic).
- Statyczne — dynamiczne — semidynamiczne (static — dynamic — semidynamic).
- Dyskretne — ciągłe (discrete — continuous).

Przykłady środowisk

Environment	Accessible	Deterministic	Episodic	Static	Discrete
Chess with a clock	Yes	Yes	No	Semi	Yes
Chess without a clock	Yes	Yes	No	Yes	Yes
Poker	No	No	No	Yes	Yes
Backgammon	Yes	No	No	Yes	Yes
Taxi driving	No	No	No	No	No
Medical diagnosis system	No	No	No	No	No
Image-analysis system	Yes	Yes	Yes	Semi	No
Part-picking robot	No	No	Yes	No	No
Refinery controller	No	No	No	No	No
Interactive English tutor	No	No	No	No	Yes

Symulator środowiska

```
procedure RUN-ENVIRONMENT(state, UPDATE-FN, agents, termination)
  inputs: state, the initial state of the environment
          UPDATE-FN, function to modify the environment
          agents, a set of agents
          termination, a predicate to test when we are done

  repeat
    for each agent in agents do
      PERCEPT[agent] ← GET-PERCEPT(agent, state)
    end
    for each agent in agents do
      ACTION[agent] ← PROGRAM[agent](PERCEPT[agent])
    end
    state ← UPDATE-FN(actions, agents, state)
  until termination(state)
```

Rozwiązywanie problemów

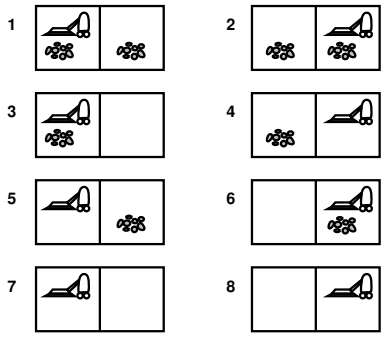
Podstawowe fazy:

1. Sformułowanie celu.
2. Sformułowanie problemu.
3. Znalezienie rozwiązania.
4. Wykonanie.

```
function SIMPLE-PROBLEM-SOLVING-AGENT(p) returns an action
  inputs: p, a percept
  static: s, an action sequence, initially empty
          state, some description of the current world state
          g, a goal, initially null
          problem, a problem formulation

  state ← UPDATE-STATE(state, p)
  if s is empty then
      g ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, g)
      s ← SEARCH(problem)
  action ← RECOMMENDATION(s, state)
  s ← REMAINDER(s, state)
  return action
```


Rodzaje problemów

Scenariusz ("vacuum world"):



Cel: Wyczyścić oba pokoje. ({7,8})

Akcje: $Left, Right, Suck$.

1. **Problemy z pełną informacją:** agent wie w jakim stanie się znajduje i zna efekty wszystkich akcji.
   Przykład: Agent wie, że znajduje się w stanie 5. Jeśli zna efekty akcji, to również wie, że sekwencja $Left, Suck$ prowadzi do stanu końcowego.

2. **Problemy z niepełną informacją o stanie początkowym.**
   Przykład: Agent nie zna stanu początkowego. Tym niemniej, jeśli zna efekty akcji, wie, że ciąg $Right, Suck, Left, Suck$ zawsze prowadzi do stanu końcowego.

3. **Problemy z niepełną informacją zarówno o stanie początkowym, jak i efektach akcji.**
   Przykład: Zakładamy, że $Suck$ czyści brudny pokój, ale rozsypuje kurz, jeśli pokój jest czysty. Agent wie, że znajduje się w stanie 1 lub 3. Potrafi stwierdzić, czy dany pokój jest czysty czy nie, tylko jeśli się w tym pokoju znajduje. Nie ma ciągu akcji gwarantującego stan końcowy. W celu rozwiązania zadania należy najpierw wykonać akcje $Suck$, $Right$, sprawdzić czy pokój jest brudny i, jeśli tak, wykonać akcję $Suck$.

Definiowanie problemów

1. **Problemy z pełną informacją o stanie początkowym.** Stan początkowy, zbiór akcji (operatorów), funkcja celu (pozwala stwierdzić, czy dany stan jest stanem końcowym) i funkcja kosztu (mierzy efektywność rozwiązania).

2. **Problemy z niepełną informacją o stanie początkowym.** Zbiór stanów początkowych, zbiór akcji, funkcja celu i funkcja kosztu.

Przestrzeń stanów problemu: Zbiór wszystkich stanów osiągalnych ze zbioru stanów początkowych w wyniku wykonywania wszystkich możliwych ciągów akcji.

Struktura danych opisująca problem:

datatype PROBLEM
components:
$INITIAL-STATE(S)$
$OPERATORS$
$GOAL-TEST$
$PATH-COST-FUNCTION$

Zasada abstrakcji: Problem powinien być opisany na właściwym poziomie abstrakcji. Należy unikać zarówno zbyt szczegółowego, jak i zbyt ogólnego opisu.

Przykłady problemów

Problem ośmiu hetmanów

Cel: Ustawić 8 hetmanów tak, aby się nie szachowały.
Funkcja kosztu: 0.

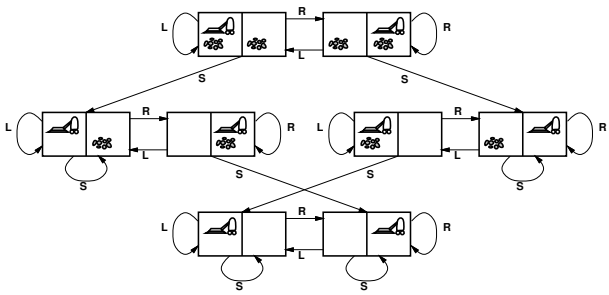
Stany: Dowolny układ od 0 do 8 hetmanów na szachownicy
Akcje: Połóż hetmana na dowolne pole.
Przy tej specyfikacji mamy 64^8 możliwych sekwencji akcji.

Stany: Dowolny układ 8 hetmanów, każdy w innej kolumnie.
Akcje: Przesuń szachowanego hetmana na inne pole w tej samej kolumnie.
Ten opis znacznie redukuje liczbę kombinacji.

Problem z odkurzaczem

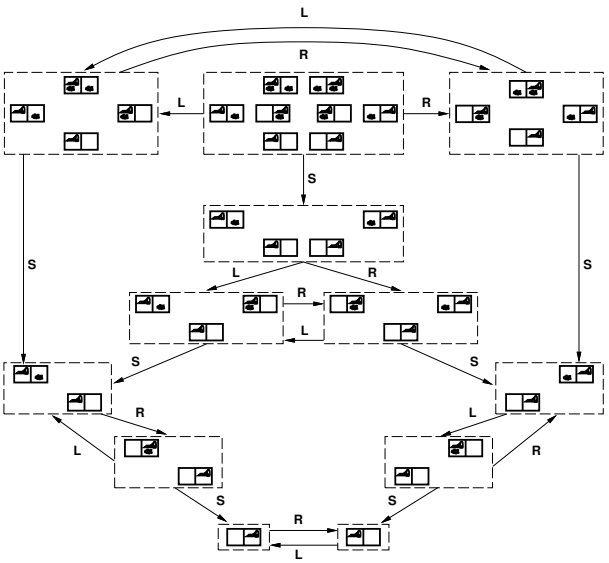
1. Przy założeniu pełnej informacji:

Stany: Jeden z ośmiu stanów występujących na rysunku poniżej.
Akcje: $Left, Right, Suck$.
Cel: Oba pokoje czyste.
Koszt: Liczba wykonanych akcji.
Przestrzeń stanów:



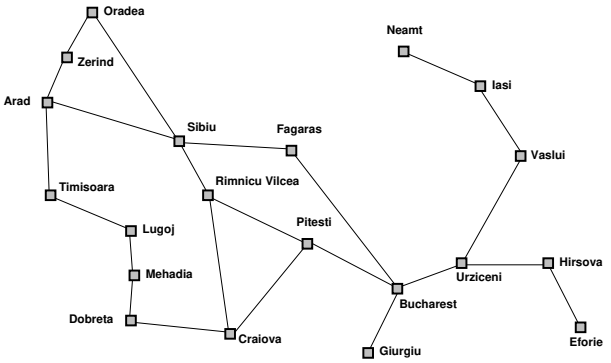
2. Niepełna informacja o stanie początkowym:

Stany: Podzbiór stanów 1 – 8.
Akcje, cel, koszt: jak wyżej.
Przestrzeń stanów:



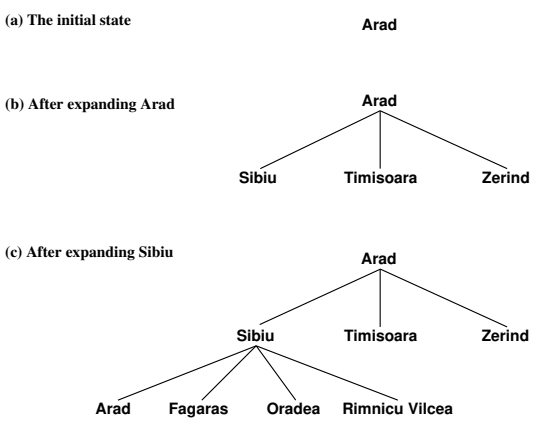
Przeszukiwanie przestrzeni stanów

Problem: Znaleźć drogę z Aradu do Bukaresztu.



Stany: Jedno z 20 miast.
Akcje: Przejazd z miasta do miasta.
Cel: Bukareszt.
Stan początkowy: Arad.
Koszt: Liczba miast przez które jedziemy.

Przeszukiwanie przestrzeni stanów sprowadza się do budowy *drzewa przeszukiwań*. Korzeniem drzewa jest stan początkowy.




```
function GENERAL-SEARCH( problem, strategy) returns a solution, or failure
   initialize the search tree using the initial state of problem
   loop do
      if there are no candidates for expansion then return failure
      choose a leaf node for expansion according to strategy
      if the node contains a goal state then return the corresponding solution
      else expand the node and add the resulting nodes to the search tree
   end
```


Struktury danych do konstruowania drzew przeszukiwań

Struktura wierzchołka drzewa:

- Stan, który reprezentuje.
- Wierzchołek, będący przodkiem.
- Akcja, której wykonanie spowodowało konstrukcję wierzchołka.
- Długość drogi od korzenia do wierzchołka.
- Koszt drogi od korzenia do wierzchołka.

datatype node
components: state, parent-node, operator
depth, path-cost.

Zbiór wierzchołków drzewa przeszukiwań wygodnie jest reprezentować w formie kolejki.

Operacje na kolejkach:

- MAKE-QUEUE(l) – tworzy kolejkę z elementów zbioru l.
- EMPTY ?(q) – sprawdza, czy kolejka q jest pusta.
- REMOVE-FRONT(q) – usuwa pierwszy element z kolejki q i zwraca go.
- QUEUING-FN(l, q) – dodaje elementy zbioru l do kolejki q.

```
function GENERAL-SEARCH( problem, QUEUING-FN) returns a solution, or failure

  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  loop do
     if nodes is empty then return failure
     node ← REMOVE-FRONT(nodes)
     if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
     nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
  end
```


Strategie przeszukiwania

Kryteria oceny strategii:

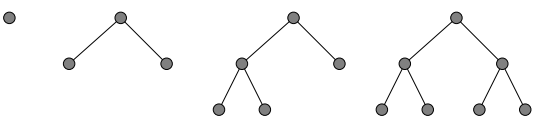
- **Pełność** – czy zawsze znajduje rozwiązanie, o ile istnieje.
- **Złożoność czasowa**.
- **Złożoność pamięciowa**.
- **Optymalność** – czy znalezione rozwiązanie jest rozwiązaniem optymalnym.

Istnieją dwa typy strategii: **strategie ogólne** i **strategie heurystyczne**.

Ogólne strategie przeszukiwania

Przeszukiwanie wszerz (breadth-first)

Wierzchołki o głębokości d konstruowane są zawsze przed wierzchołkami o głębokości $d + 1$.



Przeszukiwanie wszerz można zrealizować ogólną procedurą przeszukiwania, definiując funkcję QUEUING-FN(l, q) jako wstawienie elementów zbioru l na koniec kolejki q. ($enqueue-at-end$).

function BREADTH-FIRST-SEARCH($problem$)
return GENERAL-SEARCH($problem, enqueue-at-end$).

Zalety przeszukiwania wszerz:

- Pełność.
- Optymalne rozwiązanie, o ile funkcja kosztu zwiększa się wraz z długością drogi.

Wady przeszukiwania wszerz:

- Bardzo duża złożoność pamięciowa. Jeśli współczynnik rozgałęzienia drzewa wynosi b, a rozwiązanie problemu wymaga drogi o długości d, to maksymalna liczba skonstruowanych wierzchołków wynosi

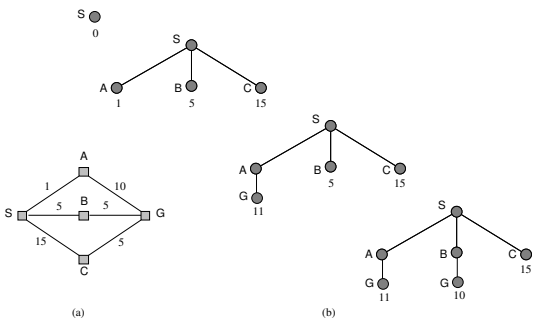
$$1 + b + b^2 + \cdots + b^d.$$

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

Strategia jednolitego kosztu (Uniform cost search)

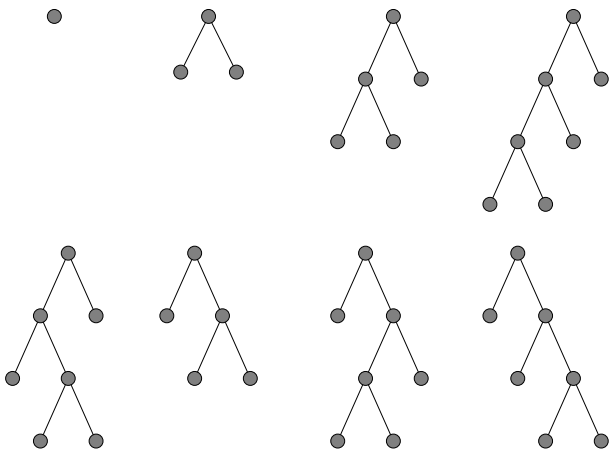
Znajduje optymalne rozwiązanie, o ile koszt każdej dopuszczalnej akcji jest dodatni. Polega na rozszerzaniu drogi o najmniejszym koszcie.

Problem najkrótszej drogi z S do G:



Przeszukiwanie w głąb (depth-first search)

Rozszerza zawsze wierzchołek o największej głębokości.



Przeszukiwanie w głąb można zrealizować ogólną procedurą przeszukiwania, definiując funkcję QUEUING-FN(l, q) jako wstawienie elementów zbioru l na początek kolejki q. ($enqueue - at - front$).

function BREADTH-FIRST-SEARCH($problem$)
return GENERAL-SEARCH($problem, enqueue{-}at{-}front$).

Zalety: Bardzo mała złożoność pamięciowa.

Wady: Strategia niepełna.

Ograniczone przeszukiwanie w głąb (Depth-limited search)

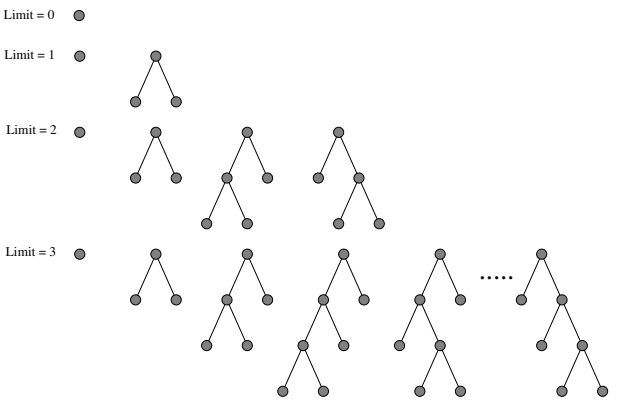
Wersja klasycznego przeszukiwania w głąb. Zakłada się maksymalną długość ścieżki w drzewie przeszukiwania. Przykładowo, w problemie znalezienia drogi z Aradu do Bukaresztu można przyjąć, że maksymalna długość ścieżki wynosi 19 (na mapie znajduje się 20 miast).

Ograniczone przeszukiwanie w głąb nie zawsze znajduje rozwiązanie optymalne.

Można je stosować tylko do specjalnej klasy problemów. Jest bardzo efektywne.

Iteracyjne przeszukiwanie w głąb (Iterative deepening search)

Wersja ograniczonego przeszukiwania w głąb. Można ją stosować do zadań, dla których nie potrafimy podać maksymalnej długości ścieżki w drzewie przeszukiwania. Technicznie, metoda sprowadza się do iteracyjnego wykonywania metody ograniczonego przeszukiwania w głąb (dla długości 1, 2, 3, . . .).




```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution sequence
    inputs: problem, a problem

    for depth ← 0 to ∞ do
      if DEPTH-LIMITED-SEARCH(problem, depth) succeeds then return its result
    end
    return failure
```

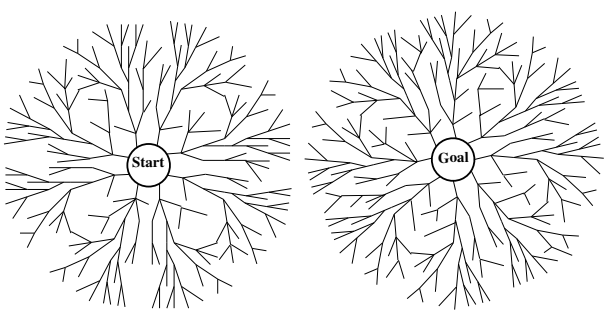
b – współczynnik rozgałęzienia.
d – długość najkrótszego rozwiązania.

- Złożoność czasowa = $O(b^d)$.
- Złożoność pamięciowa = $O(b * d)$.

Iteracyjne przeszukiwanie w głąb jest najlepszą metodą przeszukiwania dla problemów o dużej przestrzeni stanów i nieznanym ograniczeniu długości rozwiązania.

Przeszukiwanie dwukierunkowe (Bidirectional search)

Pomysł: Idziemy do przodu począwszy od stanu początkowego i do tyłu począwszy od stanu końcowego. Kończymy kiedy obie drogi spotkają się.



Dlaczego to ma sens? Załóżmy, że istnieje rozwiązanie o długości d. Rozwiązanie to znajdziemy po wykonaniu $O(b^{d/2})$ kroków.
Dla $b = 10$ i $d = 6$, strategia wszerz wymaga wygenerowania 1 111 111 wierzchołków, podczas gdy strategia dwukierunkowa tylko 2 222.

Problemy

- Co to znaczy "iść od stanu końcowego w tył"? Dla niektórych problemów znalezienie poprzedników może być bardzo trudne.
- Co zrobić, jeśli istnieje wiele stanów końcowych, które nie są explicite dane?
- Musimy umieć efektywnie stwierdzić, że obie drogi przeszukiwania spotkały się.

Analiza

W tabeli poniżej:

d – długość rozwiązania.

b – współczynnik rozgałęzienia.

m – długość drzewa przeszukiwań.

l – ograniczenie na długość ścieżki.

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

Unikanie powtarzających się stanów

- Nie wybieraj nigdy akcji, która nie zmienia stanu.
- Unikaj cykli w ścieżce. Wybierając akcję, sprawdź czy nie prowadzi ona do stanu, który już występuje w ścieżce.
- Nie generuj stanu, który był kiedykolwiek wcześniej wygenerowany (być może w innej ścieżce). To niestety wymaga przechowywania wszystkich dotychczas wygenerowanych stanów.

Przeszukiwanie z warunkami ograniczającymi (Constraint satisfaction search)

Problem z warunkami ograniczającymi

- Stany zdefiniowane są przez wartości pewnego zbioru zmiennych, $x_1, \ldots, x_n$, każda zmienna x_i z określonej dziedziny D_i.
- Stany końcowe są to stany spełniające pewien warunek, α, określony na zmiennych $x_1, \ldots, x_n$.
- Znalezienie rozwiązania polega na znalezieniu wartości zmiennych $x_1, \ldots, x_n$, spełniających warunek α.

Problem 8 hetmanów

Zmienne: $x_1, \ldots, x_8$ – pozycje hetmanów.

Dziedziną każdej zmiennej jest para $\langle i, j \rangle$, gdzie $1 \leq i, j \leq 8$.

Warunek: Żadna para hetmanów nie znajduje się w tym samym rzędzie, kolumnie ani na tej samej przekątnej.

Jak stosować ogólny algorytm przeszukiwania do problemów z warunkami ograniczającymi?

- W stanie początkowym zmienne nie mają określonych wartości.
- Każda akcja nadaje wartość jednej ze zmiennych.
- Stanem końcowym jest każdy stan spełniający warunki ograniczające.

Uwagi

- Maksymalna długość rozwiązania to liczba zmiennych. Możemy zatem stosować strategię ograniczonego przeszukiwania w głąb.

- Wartość przypisana zmiennej nie może się zmienić. Zatem warto sprawdzać warunek po każdej akcji. Jeśli można określić, że jest fałszywy, należy wykonać *nawrót* (*backtracking*).

- W celu zwiększenia efektywności można dodatkowo użyć metody *sprawdzania w przód* (*forward checking*): nadając wartość pewnej zmiennej usuwamy z dziedziny każdej nieokreślonej jeszcze zmiennej te wartości, które są sprzeczne z wartościami zmiennych już określonych. Jeśli w wyniku usunięcia któraś z dziedzin stanie się pusta, wykonujemy nawrót.

Strategie heurystyczne

Strategia "najpierw najlepszy" (Best-first search)

Pomysł: Wprowadzamy funkcję, EVAL-FN, porządkującą liniowo zbiór stanów od "najlepszego" do "najgorszego". Do dalszego rozszerzenia wybierany jest "najlepszy" węzeł spośród dotychczas skonstruowanych.

function BEST-FIRST-SEARCH(*problem*, EVAL-FN) **returns** a solution sequence
  inputs: *problem*, a problem
    Eval-Fn, an evaluation function

  Queueing-Fn ← a function that orders nodes by EVAL-FN
  return GENERAL-SEARCH(*problem*, *Queueing-Fn*)

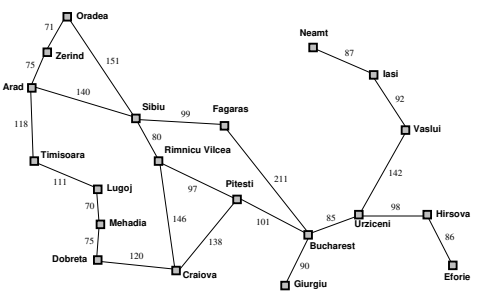
Problem: Jak określić funkcję EVAL-FN?

Przeszukiwanie zachłanne (greedy search)

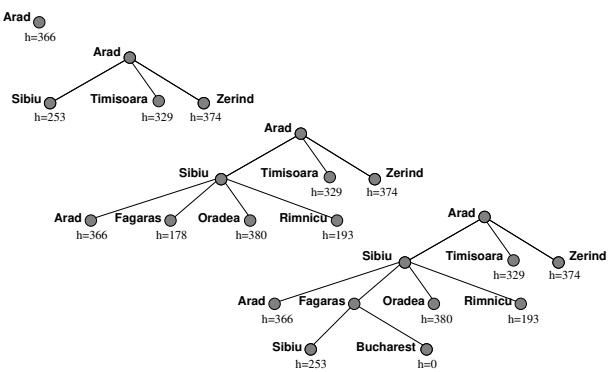
Stany porządkujemy wprowadzając *funkcję heurystyczną* h:

$h(n)$ = szacunkowy koszt optymalnej drogi ze stanu n do celu.

function GREEDY-SEARCH($problem$)
return BEST-FIRST-SEARCH($problem, h$).



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



Własności:

- Przeszukiwanie zachłanne nie gwarantuje optymalnego rozwiązania: znaleziona droga jest dłuższa od drogi przez Rimnicu, Vilcea i Pitesci.

- Przeszukiwanie zachłanne jest niepełne, ponieważ dopuszcza nieskończone rozszerzanie ścieżki, na której nie ma rozwiązania.

- Nawet dla problemów o skończonej przestrzeni stanów możemy otrzymać nieskończoną ścieżkę, o ile nie unikamy powtarzających się stanów: znaleźć drogę pomiędzy Iasi i Fagaras.

- Pesymistyczna złożoność (czasowa i pamięciowa) dla przeszukiwania zachłannego wynosi $O(b^m)$, gdzie m jest długością drzewa przeszukiwań.

Przeszukiwanie A^* (A^* search)

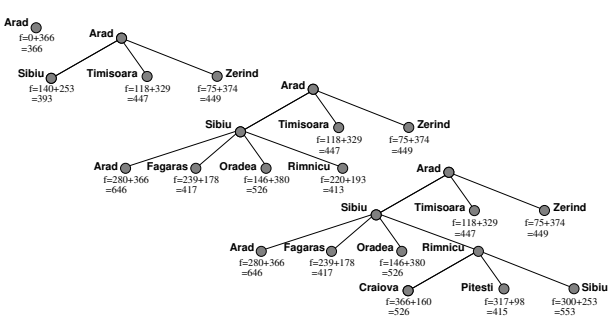
Łączy przeszukiwanie zachłanne i strategię jednolitego kosztu. Z każdym skonstruowanym dotąd stanem n wiążemy wartość $g(n)$, definiującą rzeczywisty koszt ścieżki od stanu początkowego do stanu n. Jak poprzednio wprowadzamy funkcję heurystyczną $h(n)$. Jako funkcji porządkującej stany używamy funkcji $f(n) = g(n) + h(n)$. Wartość $f(n)$ można zatem interpretować jako *szacunkowy koszt optymalnego rozwiązania zawierającego stan n.*

Funkcję heurystyczną h nazywamy *dopuszczalną* jeśli dla każdego stanu n, $h(n) \leq h^*(n)$, gdzie $h^*(n)$ oznacza rzeczywisty koszt ścieżki od stanu n do optymalnego stanu końcowego.

Przeszukiwanie A^* – przeszukiwanie "najpierw najlepszy" z funkcją porządkującą $f = g + h$, gdzie h jest heurystyką dopuszczalną.

function A^*-SEARCH($problem$)
return BEST-FIRST-SEARCH($problem, g + h$).

W zadaniu znalezienia drogi z Aradu do Bukaresztu stosujemy funkcję h taką, że $h(n)$ oznacza rzeczywistą odległość od miasta n do Bukaresztu. Ponieważ odległość nigdy nie przekracza drogi, mamy gwarancję, że h jest funkcją dopuszczalną.



Niech $f = g + h$ będzie funkcją porządkującą dla przeszukiwania A^*. Funkcja f jest *monotoniczna* jeśli dla dowolnego węzła n w drzewie przeszukiwań, $f(n) \geq f(przodek(n))$.

- Przeszukiwanie A^* jest optymalne i pełne, pod warunkiem, że funkcja porządkująca jest monotoniczna.

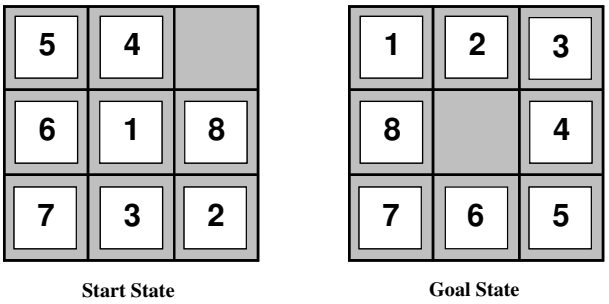
- Jeśli funkcja f nie jest monotoniczna, można zastąpić ją funkcją

$$f'(n) = max(f(n), f(przodek(n))).$$

- Przeszukiwanie A^* jest znacznie efektywniejsze od przeszukiwania wg strategii jednolitego kosztu. Niestety jego koszt jest również wykładniczy.

Funkcje heurystyczne

Problem:



- Typowe rozwiązanie: $\approx$ 20 kroków.
- Współczynnik rozgałęzienia: $\approx$ 3.
- Brutalne przeszukiwanie: do $3.5*10^9$ stanów.
- Przy unikaniu powtórzeń: do 9! = 362880 stanów.

Jak zdefiniować funkcję heurystyczną?

- h_1 = suma kostek leżących na złej pozycji. W przykładzie: h_1(stan początkowy) = 7. Heurystyka dopuszczalna: musimy wykonać przynajmniej tyle kroków, ile kostek leży na złej pozycji.

- h_2 = suma odległości kostek od pozycji docelowej. W przykładzie: h_2(stan początkowy) = 18. Heurystyka również dopuszczalna.

W celu oszacowania jakości funkcji heurystycznej wprowadzamy *efektywny współczynnik rozgałęzienia*, b^*.

Jeśli strategia A^* konstruuje drzewo składające się z N węzłów, a głębokość rozwiązania wynosi d, to b^* jest rozwiązaniem równania

$$N = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d.$$

Przykład: Jeśli $N = 52$, $d = 5$, to $b^* = 1.91$.

Uwaga: Zwykle efektywny współczynnik rozgałęzienia jest w miarę stabilny dla dużej klasy instancji problemu. Dlatego przetestowanie heurystyki na niewielkiej próbce instancji pozwala na ocenę jej jakości. Efektywny współczynnik rozgałęzienia dla dobrej funkcji heurystycznej powinien być niewiele większy niż 1.

Porównanie metody iteracyjnego przeszukiwania w głąb i strategii A^* z h_1 i h_2. Dane średnie dla 100 instancji problemu, dla różnych długości rozwiązania:

	Search Cost			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Uwaga: Zawsze lepiej jest używać funkcji heurystycznej o dużych wartościach, pod warunkiem, że nie przekracza ona funkcji kosztu rzeczywistego.

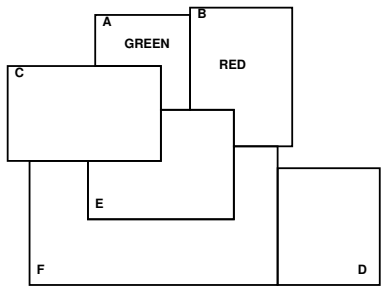
Jak znaleźć dobrą funkcję heurystyczną?

1. Uprościć problem. Często koszt optymalnego rozwiązania uproszczonego problemu jest dobrą heurystyką oryginalnego problemu. Zauważmy: gdybyśmy mogli dowolnie przenosić kostki, to h_1 wyznaczałoby liczbę kroków optymalnego rozwiązania. Podobnie, gdybyśmy mogli dowolnie przesuwać kostki o jedną pozycję, to h_2 wyznaczałoby liczbę kroków optymalnego rozwiązania.

2. Jeśli mamy kilka heurystyk, $h_1, \ldots h_k$, i żadna zdecydowanie nie dominuje, to dobrym pomysłem jest wziąć heurystykę h określoną

$$h(n) = max(h_1(n), \ldots, h_k(n)).$$

Heurystyki dla problemów z warunkami ograniczającymi



- *Heurystyka najbardziej ograniczonej zmiennej* - w każdym kroku wybieramy zmienną o minimalnej liczbie dopuszczalnych wartości.

- *Heurystyka najmniej ograniczającej wartości* – wybranej zmiennej przypisujemy wartość, która eliminuje minimalną liczbę dopuszczalnych wartości innych zmiennych.

Algorytmy iteracyjnego ulepszania (Iterative improvement algorithms)

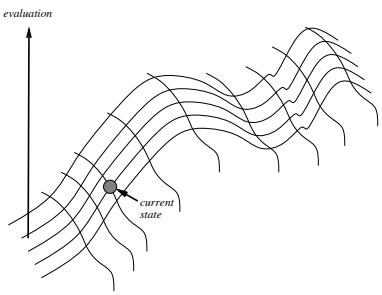
- Dotyczą problemów, dla których nieistotna jest droga uzyskania rozwiązania (np. problem 8 hetmanów).

- Pomysł: Startujemy od pewnego stanu i staramy się go "poprawić".

- W celu rozróżniania "jakości" stanów wprowadzamy funkcję heurystyczną VALUE.

- Algorytmy iteracyjnego ulepszania są bardzo efektywne (liniowe), ale rzadko znajdują od razu dobre rozwiązanie.

Metoda wspinaczki (hill-climbing search)




```
function HILL-CLIMBING( problem) returns a solution state
  inputs: problem, a problem
  static: current, a node
          next, a node

  current ← MAKE-NODE(INITIAL-STATE[problem])
  loop do
     next ← a highest-valued successor of current
     if VALUE[next] < VALUE[current] then return current
     current ← next
  end
```

Wady:

- Lokalne maksima.
- Plateaux.

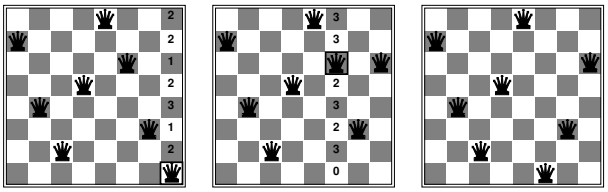
Symulowane wyżarzanie (Simulated annealing)

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  static: current, a node
          next, a node
          T, a "temperature" controlling the probability of downward steps
  current ← MAKE-NODE(INITIAL-STATE[problem])
  for t ← 1 to ∞ do
      T ← schedule[t]
      if T=0 then return current
      next ← a randomly selected successor of current
      ΔE ← VALUE[next] − VALUE[current]
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(ΔE/T)
```

- Algorytm opisuje proces wyżarzania się materiału do temperatury 0. Funkcja VALUE odpowiada całkowitej energii atomów materiału. T odpowiada temperaturze. Lista $schedule$ określa stopień obniżania się temperatury. Zmiany stanów odpowiadają losowym fluktuacjom materiału na skutek obniżania się temperatury.
- Algorytm okazał się bardzo skuteczny dla problemów optymalizacji.

Heurystyka minimalizacji konfliktów dla problemów z warunkami ograniczającymi

Stosowana przy wyborze nowej wartości dla wybranej zmiennej. Wybiera wartość, która minimalizuje liczbę konfliktów z innymi zmiennymi. Jest niezwykle skuteczna. Pozwala rozwiązać problem 1 000 000 hetmanów w granicach 50 kroków!



Gry

Gry stanowią idealną dziedzinę dla SI, ponieważ:

- Wymagają inteligencji.
- Posiadają jasne reguły.
- Stany opisujące przebieg gry są łatwo reprezentowalne w komputerze.

Problem wyboru optymalnej strategii gry przypomina problem przeszukiwania, ale jest bardziej skomplikowany, ponieważ:

- Należy uwzględniać ruchy przeciwnika.
- Nietrywialne gry mają ogromną przestrzeń stanów. Szachy: współczynnik rozgałęzienia około 35; około 10^{40} poprawnych pozycji.
- Programy grające muszą działać w czasie rzeczywistym.

Optymalna strategia dla gier dwuosobowych

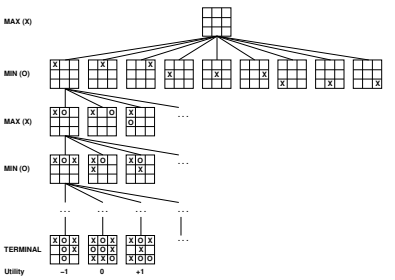
Grają MIN i MAX; MAX rozpoczyna.

Gra jako problem przeszukiwania:

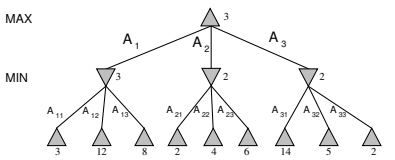
- **Stan początkowy:** zawiera pozycję planszy i informację, kto rozpoczyna grę.
- Zbiór **akcji:** zbiór poprawnych ruchów.
- **Test końcowy:** procedura stwierdzająca, czy nastąpił koniec gry.
- **Funkcja wypłaty (payoff function):** numeryczna wartość partii dla obu graczy.

Drzewo gry

Częściowe drzewo gry dla gry "kółko i krzyżyk:



Drzewo gry dla trywialnej gry, w której każdy z graczy wykonuje jeden ruch:



Algorytm MINIMAX

Znajduje optymalną strategię dla gracza MAX.

1. Wygeneruj drzewo gry. Przypisz wartość funkcji wypłaty liściom drzewa.
2. Użyj wartości przypisanych liściom do określenia wartości wierzchołków będących ich przodkami. Kontynuuj, aż przypiszesz wartości wszystkim potomkom korzenia.
3. Wybierz ruch prowadzący do potomka korzenia o największej wartości.

```
function MINIMAX-DECISION(game) returns an operator

  for each op in OPERATORS[game] do
      VALUE[op] ← MINIMAX-VALUE(APPLY(op, game), game)
  end
  return the op with the highest VALUE[op]

function MINIMAX-VALUE(state, game) returns a utility value

  if TERMINAL-TEST[game](state) then
      return UTILITY[game](state)
  else if MAX is to move in state then
      return the highest MINIMAX-VALUE of SUCCESSORS(state)
  else
      return the lowest MINIMAX-VALUE of SUCCESSORS(state)
```


Funkcje oceniające (evaluation functions)

Funkcja oceniająca zwraca oszacowanie wartości danego wierzchołka drzewa gry (pozycji).

Funkcja oceniająca ma kluczowy wpływ na jakość programu grającego.

- Funkcja oceniająca powinna zgadzać się z funkcją wypłaty dla pozycji końcowych.
- Funkcja oceniająca powinna być obliczalna w rozsądnym czasie.

Bardzo prosta funkcja oceniająca dla gry w szachy:

$$w_1 * f_1 + w_2 * f_2 + w_3 * f_3 + w_4 * f_4$$

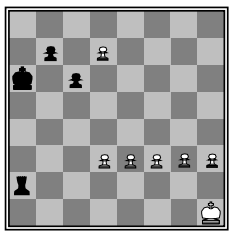
gdzie w_i jest numeryczną wartością przypisaną figurom (pion = 1, goniec i skoczek = 3, wieża = 5, hetman = 9), natomiast f_i oznacza liczbę figur danego rodzaju w danej pozycji.

Programy grające przeszukują fragment drzewa gry, posługując się funkcją oceniającą. Na ogół stosuje się strategię iteracyjnego przeglądania w głąb.

Problemy:

- Problem przerwania przeszukiwania: przeszukiwania nie można przerwać w dowolnym wierzchołku drzewa gry.

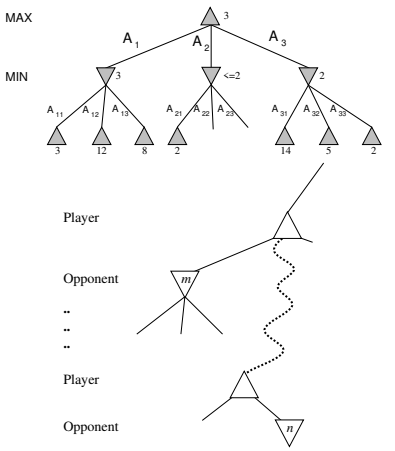
- Problem horyzontu:



Black to move

Odcięcie alfa-beta (Alpha-beta prunning)

Podobne do algorytmu minimax, ale odcina gałęzie drzewa, które nie mogą mieć wpływu na wynik.



Poniżej: α oznacza wartość najlepszego (względem funkcji oceniającej) dotychczas znalezionego węzła dla gracza MAX, β oznacza wartość najlepszego dotychczas znalezionego węzła dla gracza MIN. Funkcja CUTOFF-TEST sprawdza, czy doszliśmy do maksymalnej dopuszczalnej głębokości.

function MAX-VALUE(*state, game,* α, β) **returns** the minimax value of *state*
  inputs: *state*, current state in game
    game, game description
    α, the best score for MAX along the path to *state*
    β, the best score for MIN along the path to *state*

  if CUTOFF-TEST(*state*) **then return** EVAL(*state*)
  for each *s* **in** SUCCESSORS(*state*) **do**
    $\alpha \leftarrow$ MAX(α, MIN-VALUE(*s, game,* α, β))
    if $\alpha \geq \beta$ **then return** β
  end
  return α

function MIN-VALUE(*state, game,* α, β) **returns** the minimax value of *state*

  if CUTOFF-TEST(*state*) **then return** EVAL(*state*)
  for each *s* **in** SUCCESSORS(*state*) **do**
    $\beta \leftarrow$ MIN(β, MAX-VALUE(*s, game,* α, β))
    if $\beta \leq \alpha$ **then return** α
  end
  return β

Złożoność algorytmów minimax i alfa-beta

Ponieważ oba algorytmy realizują przeszukiwanie w głąb, istotna jest tylko ich złożoność czasowa.

Poniżej d oznacza dopuszczalną głębokość, natomiast b współczynnik rozgałęzienia.

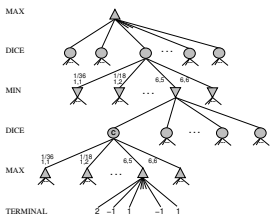
Mini-max: $O(b^d)$.

Alfa-beta:

- Przypadek najlepszy: $O(b^{d/2})$.
- Średnio: $O(b^{3d/4})$.

Gry z elementem losowości

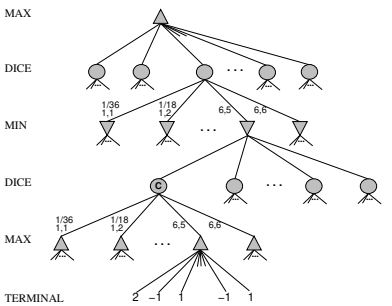
Drzewo gry dla gry backgammon:



Nie możemy obliczyć wartości węzła C. Możemy tylko obliczyć jego *wartość oczekiwaną*. Niech d_i będzie możliwym wynikiem rzutu kośćmi, $P(d_i)$ prawdopodobieństwem tego rzutu. Niech $S(\mathrm{C}, d_i)$ oznacza zbiór wszystkich poprawnych pozycji, do których można dojść z C wyrzucając d_i. Wówczas wartość oczekiwana węzła C wynosi

$$\sum_i [P(d_i) * max_{s \in S(\mathrm{C}, d_i)}(wyplata(s))].$$

Uwaga1: Funkcja oceniająca dla gier z elementem losowości musi być liniową transformacją przybliżonego prawdopodobieństwa wygrania z tej pozycji. Dla procedury minimax nie ma znaczenia, czy wartościami węzłów są 1, 2, 3, 4 czy 1, 20, 30, 400. Dla procedury **expectminimax** (wersja procedury minimax dla gier z elementem losowości) może mieć:

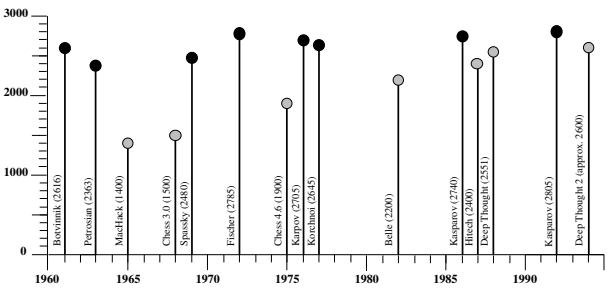


Uwaga2: Złożoność procedury expectminimax: $O(b^d * n^d)$, gdzie n jest liczbą możliwych wyników rzutu kostką (kostkami).

Obecny stan programów grających

1. Szachy

Najlepszy istniejący program — Deep Blue. Wykorzystuje bardzo potężną architekturę równoległą. Analizuje 100 - 200 miliardów pozycji na ruch. Dochodzi do głębokości 14.



2. Warcaby

Najlepszy program – Chinook. Pierwszy program, który startował w 1993 roku w otwartych mistrzostwach świata. Przegrał w finale 18.5 – 21.5. W roku 1994 został oficjalnym mistrzem świata.

3. Otello

Programy grające w Otello są znacznie lepsze od ludzi.

4. Backgammon

Najlepszy program mieści się w pierwsze trójce na świecie.

5. Go

Nie ma dobrych programów (b ≈ 360).

Agent wnioskujący

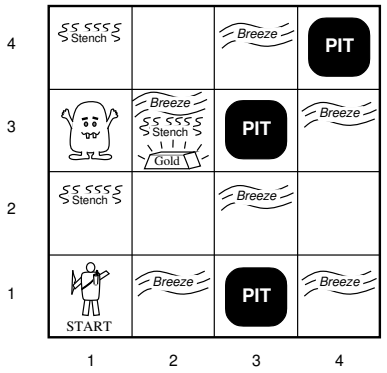
Baza wiedzy (knowledge base): Zbiór faktów o świecie. Fakty te, zwane **zdaniami** lub **formułami**, zapisane są w **języku reprezentacji wiedzy**. Agent komunikuje się z bazą wiedzy (KB), używając operacji TELL i ASK:

- TELL: Agent → KB.
- ASK: KB → Agent.

Ogólny model agenta posługującego się bazą wiedzy:

```
function KB-AGENT(percept) returns an action
  static: KB, a knowledge base
          t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  action ← ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action
```

Świat Wumpusa (Wumpus world)



Obserwacje, akcje i cele agenta:

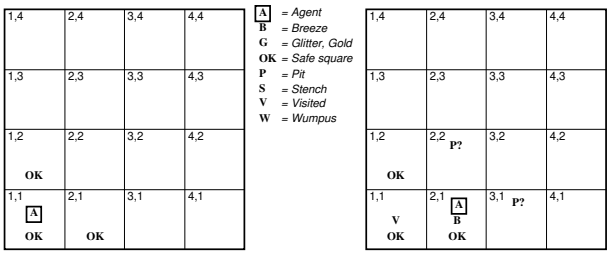
- W kwadracie gdzie mieszka Wumpus oraz w kwadratach ściśle przylegających agent czuje smród (stench).

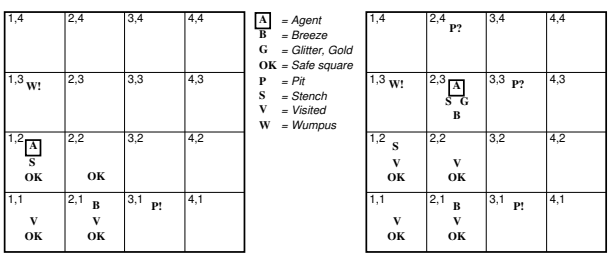
- W kwadratach ściśle przylegających do jamy (pit), agent czuje wiatr (breeze).
- W kwadracie, gdzie znajduje się złoto, agent obserwuje błysk (glitter).
- Jeśli agent wejdzie na ścianę, czuje uderzenie (bump).
- Kiedy Wumpus zostaje zabity, w całej jaskini rozlega się wycie (scream).
- Obserwacje agenta reprezentujemy jako listę złożoną z 5 elementów. Przykładowo: $[Stench, Breeze, Glitter, None, None]$. Agent nie potrafi rozpoznać kwadratu, w którym się znajduje.
- Akcje agenta: $Go-Forward$, $Turn-Left$, $Turn-Right$, $Grab$ (podnosi obiekt znajdujący się w tym samym kwadracie co agent),

$Release$ (pozwala agentowi pozbyć się posiadanego przedmiotu), $Shoot$ (wypuszcza strzałę w kierunku, w którym agent patrzy; strzała albo zabija Wumpusa, albo odbija się od zewnętrznej ściany jaskini; agent ma tylko jedną strzałę), $Climb$ (pozwala agentowi opuścić jaskinię, o ile znajduje się on w kwadracie startowym).

- Agent ginie jeśli wejdzie do kwadratu, w którym znajduje się jama lub Wumpus.
- Celem agenta jest znalezienie złota i wydostanie się z jaskini. Dokładniej: 1000 punktów za wydostanie się z jaskini ze złotem, -1 punkt kary za każdą wykonaną akcję i -10000 punktów kary za zginięcie.

Wnioskowanie w świecie Wumpusa





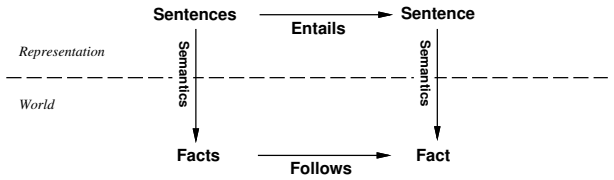
Reprezentacja, wnioskowanie i logika

Reprezentacja wiedzy (knowledge representation: Dziedzina SI zajmująca się metodami reprezentowania wiedzy o świecie.

Język reprezentacji wiedzy tworzą:

- **Syntaktyka** — opisuje budowę zdań.
- **Semantyka** — opisuje związek pomiędzy zdaniami i odpowiadającymi im faktami zachodzącymi w świecie.

Wnioskowanie: proces wyprowadzania nowych zdań ze zdań przyjętych jako prawdziwe (tzn. reprezentujących prawdziwe fakty). Poprawne wnioskowanie powinno zapewniać prawdziwość wyprowadzonych zdań.



Model dla zbioru zdań X: każdy świat w którym prawdziwe są wszystkie zdania ze zbioru X.

Wynikanie (entailment): Zdanie A wynika ze zbioru zdań X, $X \models A$, jeśli A jest prawdziwe w każdym modelu dla X.

Wyprowadzalność (derivability): Zdanie A jest wyprowadzalne ze zbioru zdań X przy użyciu procedury dowodzenia Π, $X \vdash_\Pi A$, wtw gdy Π znajduje dowód zdania A ze zdań zbioru X.

Poprawność procedury dowodzenia: Procedura dowodzenia Π jest poprawna wtw gdy dla każdego zbioru zdań X i każdego zdania A, $X \vdash_\Pi A$ pociąga $X \models A$.

Pełność procedury dowodzenia: Procedura dowodzenia Π jest pełna wtw gdy dla każdego zbioru zdań X i każdego zdania A, $X \models A$ pociąga $X \vdash_\Pi A$.

Logika: Język reprezentacji wiedzy (syntaktyka, semantyka) oraz procedura dowodzenia.

Language	Ontological Commitment (What exists in the world)	Epistemological Commitment (What an agent believes about facts)
Propositional logic	facts	true/false/unknown
First-order logic	facts, objects, relations	true/false/unknown
Temporal logic	facts, objects, relations, times	true/false/unknown
Probability theory	facts	degree of belief 0…1
Fuzzy logic	degree of truth	degree of belief 0…1

Rachunek zdań

Alfabet: stałe logiczne ($True$ i $False$), symbole zdaniowe (P, Q, ...), spójniki logiczne ($\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ i $\neg$) oraz nawiasy okrągłe.

Zdania (formuły):

- $True$, $False$ i symbole zdaniowe są formułami. (Formuły atomowe)
- Jeśli A i B są formułami, to $(A \wedge B)$, $(A \vee B)$, $(A \Rightarrow B)$ i $(A \Leftrightarrow B)$ są formułami.
- Jeśli A jest formułą, to $(\neg A)$ jest formułą.

Literał: Formuła atomowa lub negacja formuły atomowej.

Semantyka rachunku zdań

Interpretacja — Funkcja przyporządkowująca symbolom zdaniowym elementy zbioru $\{True, False\}$.

Wartość formuły w interpretacji:

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
False	*False*	*True*	*False*	*False*	*True*	*True*
False	*True*	*True*	*False*	*True*	*True*	*False*
True	*False*	*False*	*False*	*True*	*False*	*False*
True	*True*	*False*	*True*	*True*	*True*	*True*

Tautologia: Formuła prawdziwa w każdej interpretacji.

Rachunek zdań jest **rozstrzygalny**, tzn. istnieje algorytm, który dla dowolnej formuły A stwierdza, czy A jest tautologią.

P	H	$P \vee H$	$(P \vee H) \wedge \neg H$	$((P \vee H) \wedge \neg H) \Rightarrow P$
False	*False*	*False*	*False*	*True*
False	*True*	*True*	*False*	*True*
True	*False*	*True*	*True*	*True*
True	*True*	*True*	*False*	*True*

Model formuły A: interpretacja, w której A jest prawdziwa.

Formuła A jest **spełnialna** wtw gdy posiada model. W przeciwnym przypadku A jest niespełnialna.

Uwaga: A jest tautologią wtw gdy $\neg A$ jest niespełnialna.

Twierdzenie o dedukcji:

$X \cup \{A\} \models B$ wtw gdy $X \models A \Rightarrow B$. ■

Dla rachunku zdań istnieje pełna i poprawna procedura dowodzenia.

Reguła wnioskowania: $\frac{\alpha_1, \ldots \alpha_n}{\beta}$.

Reguła jest **poprawna**, jeśli β jest prawdziwa w każdej interpretacji, w której prawdziwe są $\alpha_1, \ldots \alpha_n$.

Przykłady poprawnych reguł wnioskowania

Reguła **odrywania** (modus ponens): $\frac{\alpha, \quad \alpha \Rightarrow \beta}{\beta}$.

Reguła **eliminacji koniunkcji**: $\frac{\alpha_1 \wedge \cdots \wedge \alpha_n}{\alpha_i}$

Reguła **wprowadzenia koniunkcji**: $\frac{\alpha_1, \ldots, \alpha_n}{\alpha_1 \wedge \cdots \wedge \alpha_n}$

Reguła **rezolucji**: $\frac{\alpha \vee \beta, \quad \neg\beta \vee \gamma}{\alpha \vee \gamma}$.

Poprawność reguły rezolucji ($\frac{\alpha \vee \beta, \quad \neg\beta \vee \gamma}{\alpha \vee \gamma}$):

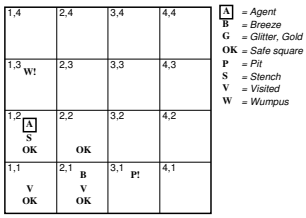
α	β	γ	$\alpha \vee \beta$	$\neg\beta \vee \gamma$	$\alpha \vee \gamma$
False	*False*	*False*	*False*	*True*	*False*
False	*False*	*True*	*False*	*True*	*True*
False	*True*	*False*	*True*	*False*	*False*
False	*True*	*True*	*True*	*True*	*True*
True	*False*	*False*	*True*	*True*	*True*
True	*False*	*True*	*True*	*True*	*True*
True	*True*	*False*	*True*	*False*	*True*
True	*True*	*True*	*True*	*True*	*True*

Monotoniczność: Jeśli formuła A wynika ze zbioru formuł X, to A wynika również z każdego nadzbioru zbioru X.

Złożoność rachunku zdań: Problem stwierdzenia, czy formuła jest tautologią jest NP-zupełny.

Jak sformalizować świat Wumpusa w rachunku zdań?

Wiedza agenta po wykonaniu trzeciego ruchu. Bieżąca lista obserwacji: [$Stench$, $None$, $None$, $None$, $None$].



Obserwacje:

$$\neg S_{1,1},\ \neg B_{1,1},\ \neg S_{2,1},\ B_{2,1},\ S_{1,2},\ \neg B_{1,2}.$$

Wiedza o środowisku:

$$\begin{aligned}
\neg S_{1,1} &\Rightarrow \neg W_{1,1} \wedge \neg W_{1,2} \wedge \neg W_{2,1} && (1)\\
\neg S_{2,1} &\Rightarrow \neg W_{1,1} \wedge \neg W_{2,1} \wedge \neg W_{2,2} \wedge \neg W_{3,1} && (2)\\
\neg S_{1,2} &\Rightarrow \neg W_{1,1} \wedge \neg W_{1,2} \wedge \neg W_{2,2} \wedge \neg W_{1,3} && (3)\\
S_{1,2} &\Rightarrow W_{1,3} \vee W_{1,2} \vee W_{2,2} \vee W_{1,1} && (4)
\end{aligned}$$

KB – koniunkcja obserwacji i formuł (1) - (4).

Chcemy pokazać, że z KB wynika $W_{1,3}$.
W tym celu wystarczy pokazać, że formuła

$$KB \Rightarrow W_{1,3} \qquad (5)$$

jest tautologią. Metoda zero-jedynkowa jest niepraktyczna (4096 wierszy). Łatwo wykazać, że (5) jest tautologią, posługując się regułą odrywania, eliminacji koniunkcji i rezolucji.

Uwaga: Rachunek zdań nie jest wygodny do formalizacji świata Wumpusa (zbyt wiele symboli zdaniowych, co czyni formalizację nieefektywną).

Jak reprezentować akcje?

$$A_{1,1} \wedge East_A \wedge W_{2,1} \Rightarrow \neg Forward.$$

Jeśli używamy rachunku zdań, nie możemy zadać bazie wiedzy pytania "Jaką akcję wybrać?" Musimy zadać serię pytań, każde dotyczące innej akcji: "Czy mam iść naprzód", Czy mam się odwrócić w prawo?", itp.

Model agenta posługującego się bazą wiedzy w rachunku zdań:

```
function PROPOSITIONAL-KB-AGENT(percept) returns an action
  static: KB, a knowledge base
          t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  for each action in the list of possible actions do
      if ASK(KB, MAKE-ACTION-QUERY(t, action)) then
          t ← t + 1
          return action
  end
```


Świat Wumpusa zmienia się wraz z upływem czasu. Jak reprezentować te zmiany?

Jedyną metodą jest odpowiednie indeksowanie stałych zdaniowych:

$$\begin{aligned}
A^0_{1,1} \wedge East^0_A \wedge W^0_{2,1} &\Rightarrow \neg Forward^0\\
A^1_{1,1} \wedge East^1_A \wedge W^1_{2,1} &\Rightarrow \neg Forward^1\\
A^2_{1,1} \wedge East^2_A \wedge W^2_{2,1} &\Rightarrow \neg Forward^2\\
&\vdots\\
A^0_{1,1} \wedge North^0_A \wedge W^0_{1,2} &\Rightarrow \neg Forward^0\\
A^1_{1,1} \wedge North^1_A \wedge W^1_{1,2} &\Rightarrow \neg Forward^1\\
A^2_{1,1} \wedge North^2_A \wedge W^2_{1,2} &\Rightarrow \neg Forward^2\\
&\vdots
\end{aligned}$$

Jeśli przyjmiemy, że agent może wykonać 100 akcji w trakcie gry, to potrzebujemy 6400 formuł na wyrażenie prostego faktu, że nie należy iść do kwadratu, w którym znajduje się Wumpus.

Logika pierwszego rzędu

Dopuszcza obiekty oraz funkcje i relacje określone na tych obiektach.

Rozszerzamy alfabet:

- Zmienne indywiduowe: $x, y, x', y_1, \ldots$
- Stałe indywiduowe: A, Jan, $\ldots$
- Symbole funkcji: f, $ojciec, \ldots$
- Symbole predykatów: P, $Brat, \ldots$
- Kwantyfikatory: $\forall$, $\exists$.
- Predykat równości: $=$.

Termy: Wskazują na obiekty.

- Zmienna (stała) indywiduowa jest termem.
- Jeśli f jest symbolem funkcji n-argumentowej i $t_1, \ldots, t_n$ są termami, to $f(t_1, \ldots, t_n)$ jest termem.

Term zamknięty (ground term) – term, w którym nie występują zmienne.

Formuły atomowe: Formuły atomowe rachunku zdań oraz

- Jeśli P jest symbolem relacji n-argumentowej i $t_1, \ldots, t_n$ są termami, to $P(t_1, \ldots, t_n)$ jest formułą atomową.
- Jeśli t_1 i t_2 są termami, to $(t_1 = t_2)$ jest formułą atomową.

Formuły:

- Formuła atomowa jest formułą.
- Jeśli A i B są formułami, to $(A \wedge B)$, $(A \vee B)$, $(A \Rightarrow B)$ i $(A \Leftrightarrow B)$ są formułami.
- Jeśli A jest formułą, x jest zmienną indywiduową, to $(\neg A)$, $(\forall x A)$ i $(\exists x A)$ są formułami.

Semantyka

Interpretacja: Para $\langle D, m \rangle$, gdzie

- D – niepusty zbiór, **dziedzina** interpretacji.
- m – funkcja, która stałym przyporządkowuje elementy dziedziny, n-argumentowym symbolom funkcyjnym, funkcje z D^n w D, symbolom zdaniowym, wartości ze zbioru $\{True, False\}$, oraz n-argumentowym symbolom relacyjnym, n-argumentowe relacje określone na D^n.

Wartościowanie zmiennych w interpretacji: funkcja ze zbioru zmiennych indywiduowych w dziedzinę interpretacji.

$V_a^M(t)$ - wartość termu t w interpretacji $M = \langle D, m \rangle$, względem wartościowania a:

- $V_a^M(x) = a(x)$, gdzie x jest zmienną.
- $V_a^M(C) = m(C)$, gdzie C jest stałą.
- $V_a^M(f(t_1, \ldots, t_n)) = m(f)(V_a^M(t_1), \ldots V_a^M(t_n))$.

$a(x \leftarrow d)$ — wartościowanie identyczne z a, dla wszystkich zmiennych poza x. Zmiennej x przypisana jest wartość d.

$V_a^M(A)$ - wartość formuły A w interpretacji M, względem wartościowania a:

- $V_a^M(Q) = m(Q)$, dla stałej zdaniowej C.
- $V_a^M(P(t_1, \ldots, t_n)) = m(P)(V_a^M(t_1), \ldots V_a^M(t_n))$.
- $V_a^M(t_1 = t_2) = (V_a^M(t_1) = V_a^M(t_2))$.

- $V_a^M(B \wedge C) = V_a^M(B) \wedge V_a^M(C)$.
- $V_a^M(B \vee C) = V_a^M(B) \vee V_a^M(C)$.
- $V_a^M(B \Rightarrow C) = V_a^M(B) \Rightarrow V_a^M(C)$.
- $V_a^M(B \Leftrightarrow C) = V_a^M(B) \Leftrightarrow V_a^M(C)$.
- $V_a^M(\forall x B) = min_{d \in D}[V_{a(x \leftarrow d)}^M(B)]$.
- $V_a^M(\exists x B) = max_{d \in D}[V_{a(x \leftarrow d)}^M(B)]$.

Formuła A jest **spełnialna** jeśli ma wartość $True$ w pewnej interpretacji M, względem pewnego wartościowania a.

Formuła A jest **prawdziwa** w interpretacji M jeśli ma wartość $True$ w tej interpretacji, względem każdego wartościowania a. M nazywa się wtedy **modelem** dla A.

Formuła A jest **tautologią** jeśli jest prawdziwa w każdej interpretacji.

W logice pierwszego rzędu zachodzi twierdzenie o dedukcji.

Problem stwierdzenia, czy dana formuła logiki pierwszego rzędu jest tautologią jest problemem nierozstrzygalnym. Jest on natomiast **półrozstrzygalny**: Istnieje algorytm, który dla zadanej formuły A stwierdza, że A jest tautologią, pod warunkiem, że tak istotnie jest.

Teoria: Zbiór formuł. Teoria jest **niesprzeczna** jeśli posiada model, tzn. istnieje interpretacja, w której prawdziwe są wszystkie formuły teorii.

Istnieje poprawna i pełna procedura dowodzenia dla logiki pierwszego rzędu.

Komunikacja z bazą wiedzy

TELL$(KB, Brat(Jan, Piotr))$

ASK$(KB, Lubi(Anna, Jan))$.

Pytania typu "kto", "gdzie", "kiedy".
Przykładowo: Kto jest bratem Piotra?

ASK$(KB, \exists x\ Brat(x, Piotr))$.

Oczekiwana odpowiedź:

$\{x/Jan\}, \{x/Stefan\}$.

Model agenta posługującego się bazą wiedzy w logice pierwszego rzędu:

function KB-AGENT(*percept*) **returns** an *action*
  static: *KB*, a knowledge base
    t, a counter, initially 0, indicating time

  TELL(*KB*, MAKE-PERCEPT-SENTENCE(*percept*, *t*))
  action ← ASK(*KB*, MAKE-ACTION-QUERY(*t*))
  TELL(*KB*, MAKE-ACTION-SENTENCE(*action*, *t*))
  t ← *t* + 1
  return *action*

Świat Wumpusa w logice pierwszego rzędu

Typowa obserwacja:

$Percept([Stench, Breeze, Glitter, None, None], 5)$.

Akcje: $Turn(Right)$, $Turn(Left)$, $Forward$, $Grab$, $Shoot$, $Release$, $Climb$.

Funkcja MAKE-ACTION-QUERY konstruuje pytanie typu

$\exists a\ Action(a, 5)$.

Prawidłowa odpowiedź: $\{a/Grab\}$.

Prosty agent reaktywny

$$\forall s, b, u, c, t\ Percept([s, b, Glitter, u, c], t) \Rightarrow Action(Grab, t).$$

Lepiej:

$$\forall s, b, u, c, t\ Percept([s, b, Glitter, u, c], t) \Rightarrow AtGold(t)$$

$$\forall t\ AtGold(t) \Rightarrow Action(Grab, t).$$

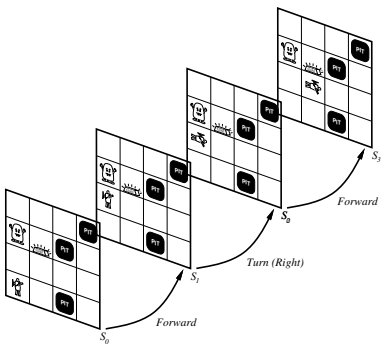
Prosty agent reaktywny nie poradzi sobie ze światem Wumpusa: Nigdy nie będzie wiedział, jak wrócić do kwadratu startowego w celu wykonania akcji $Climb$; często się zapętli.

Agent reaktywny

Agent reaktywny reaguje na bieżącą obserwację w oparciu o aktualny stan świata. Baza wiedzy takiego agenta musi więc mieć charakter dynamiczny.

Rachunek sytuacyjny (situation calculus, Hayes, McCarthy 1969) – system oparty na logice pierwszego rzędu do formalizacji dynamicznie zmieniającego się świata.

Świat = ciąg **sytuacji**, każda opisuje stan świata w pewnym momencie czasu. Nowa sytuacja powstaje z bieżącej sytuacji w wyniku wykonania akcji.



Każdy symbol reprezentujący relację zmieniającą się w czasie, posiada dodatkowy argument, odpowiadający sytuacji:

$$At(Agent, [1, 1], S_0) \wedge At(Agent, [1, 2], S_1).$$

$Result(action, situation)$ – sytuacja będąca wynikiem wykonania akcji $action$ w sytuacji $situation$.

$$S_1 = Result(Forward, S_0).$$

Akcje definiuje się podając ich efekty.

$$Portable(Gold).$$

$$\forall s\ AtGold(s) \Rightarrow Present(Gold, s).$$

$$\forall x, s\ Present(x, s) \wedge Portable(x) \Rightarrow Holding(x, Result(Grab, s)).$$

$$\forall x, s\ \neg Holding(x, Result(Release, s)).$$

Potrzebne są jeszcze **aksjomaty tła (frame axioms)**: (1) Jeśli agent posiada przedmiot i go nie wyrzuca, to posiada ten przedmiot w następnym stanie. (2) Jeśli agent nie posiada przedmiotu i go nie podnosi, to nie posiada go w następnym stanie.

$$\forall a, x, s\ Holding(x, s) \wedge (a \neq Release) \Rightarrow Holding(x, Result(a, s)).$$

$$\forall a, x, s\ \neg Holding(x, s) \wedge [a \neq Grab \vee \neg(Present(x, s) \wedge Portable(x))] \Rightarrow \neg Holding(x, Result(a, s)).$$

Aksjomaty definiujące efekty akcji można połączyć z aksjomatami tła, otrzymując

$$\forall a, x, s\ Holding(x, Result(a, s)) \Leftrightarrow [a = Grab \wedge Present(x, s) \wedge Portable(x)) \vee (Holding(x, s) \wedge a \neq Release)].$$

Powyższy akjomat nazywamy **aksjomatem następnika stanu (successor-state axiom)**.

Liczba aksjomatów następnika stanu = liczba symboli predykatów zmieniających się w czasie.

Położenie agenta

W sytuacji początkowej: $At(Agent, [1, 1], S_0)$.

Co agent powinien wiedzieć o swojej pozycji?

1. Kierunek, w którym patrzy (0 – wschód, 90 – północ, 180 – zachód, 270 – południe).

$Orientation(Agent, S_0) = 90$.

2. "Mapa" terenu. Mapę określa funkcja $LocationToward$, w skrócie LT, która dla danej pozycji i kierunku zwraca pozycję znajdującą się o jedną kratkę do przodu względem kierunku.

$\forall x, y\ LT([x, y], 0) = [x + 1, y]$.

$\forall x, y\ LT([x, y], 90) = [x, y + 1]$.

$\forall x, y\ LT([x, y], 180) = [x - 1, y]$.

$\forall x, y\ LT([x, y], 270) = [x, y - 1]$.

Posługując się funkcją LT, możemy określić kwadrat znajdujący się bezpośrednio przed agentem p:

$$\forall p, l, s\ At(p, l, s) \Rightarrow [LocationAhead(p, s) = LT(l, Orientation(p, s))].$$

Możemy też określić "ścisłe przyleganie":

$$\forall l_1, l_2\ Adjacent(l_1, l_2) \Leftrightarrow \exists d\ l_1 = LT(l_2, d).$$

3. "Geografia" terenu.

$$\forall x, y\ Wall([x, y]) \Leftrightarrow (x = 0 \vee x = 5 \vee y = 0 \vee y = 5).$$

4. Związek pomiędzy akcjami i położeniem agenta:

$$\forall a, l, p, s\ At(p, l, Result(a, s)) \Leftrightarrow [(a = Forward \wedge l = LocationAhead(p, s) \wedge \neg Wall(l)) \vee (At(p, l, s) \wedge a \neq Forward)].$$

Własności świata

Akjomaty obserwacyjne: Opisują własności świata na podstawie obserwacji.

$$\forall l, s\ At(Agent, l, s) \wedge Stench(s) \Rightarrow Smelly(l).$$

$$\forall l_1, s\ Smelly(l_1) \Rightarrow [\exists l_2\ At(Wumpus, l_2, s) \wedge (l_2 = l_1 \vee Adjacent(l_1, l_2))].$$

$$\forall x, y, g, u, c, s\ Percept([None, None, g, u, c], s) \wedge At(Agent, x, s) \wedge Adjacent(x, y) \Rightarrow OK(y).$$

Aksjomaty przyczynowe: Opisują, w jaki sposób własności świata wpływają na obserwacje.

$$\forall l_1, l_2, s\ At(Wumpus, l_1, s) \wedge Adjacent(l_1, l_2) \Rightarrow Smelly(l_2).$$

$$\forall l, s\ \neg At(Wumpus, l, s) \wedge \neg Pit(l) \Leftrightarrow OK(l).$$

Wybór akcji

Klasyfikujemy akcje:

- $Great$ – Wzięcie złota i wydostanie się z jaskini ze złotem.
- $Good$ – Wejście do bezpiecznego kwadratu, który nie był jeszcze odwiedzany.
- $Medium$ – Wejście do bezpiecznego kwadratu, który był już odwiedzany.
- $Risky$ – Wejście do kwadratu, w którym może być Wumpus lub jama.
- $Deadly$ – Wejście do kwadratu, w którym na pewno jest Wumpus lub jama.

.

Jak sformalizować wybór akcji? Wprowadzamy dwuargumentowy symbol predykatowy $Action$. $Action(a, s)$ interpretujemy: w stanie s wybierz akcję a.

$$\forall a, s\ Great(a, s) \Rightarrow Action(a, s).$$

$$\forall a, s\ Good(a, s) \wedge [\neg\exists b\ Great(b, s)] \Rightarrow Action(a, s).$$

$$\forall a, s\ Medium(a, s) \wedge [\neg\exists b\ Great(b, s) \vee Good(b, s)] \Rightarrow Action(a, s).$$

$$\forall a, s\ Risky(a, s) \wedge [\neg\exists b\ Great(b, s) \vee Good(b, s) \vee Medium(b, s)] \Rightarrow Action(a, s).$$

Agent realizujący cel

Agent reaktywny potrafi rozsądnie poruszać się po jaskini. W momencie znalezienia złota celem agenta jest jak najszybszy powrót do kwadratu startowego i wydostanie się z jaskini.

$$\forall s\ Holding(Gold, s) \Rightarrow GoalLocation([1, 1], s).$$

Agent powinien znaleźć sekwencję akcji (plan) realizującą ten cel. Istnieją dwie techniki rozwiązania tego zadania:

- Przeszukiwanie
- Planowanie

Problemy związane z formalizowaniem dynamicznie zmieniającego się świata

- **Problem tła** (frame problem) – Jak efektywnie reprezentować inercję? Połączenie aksjomatów opisujących efekty akcji i aksjomatów tła w aksjomaty następnika stanu nie rozwiązuje problemu dla akcji niedeterministycznych.
- **Problem kwalifikacji** (qualification problem) – Działając w rzeczywistym świecie, prawie nigdy nie wiemy, czy akcja się powiedzie. Jak sobie z tym poradzić formalizując dynamicznie zmieniający się świat?
- **Problem ramifikacji** (ramification problem) – Jak sformalizować uboczne efekty akcji?

Inżynieria wiedzy (Knowledge engineering)

Zajmuje się metodologią konstruowania bazy wiedzy.

Inżynieria oprogramowania:

- Języki programowania.
- Programy.
- Kompilatory.
- Realizacja programów.

Inżynieria wiedzy:

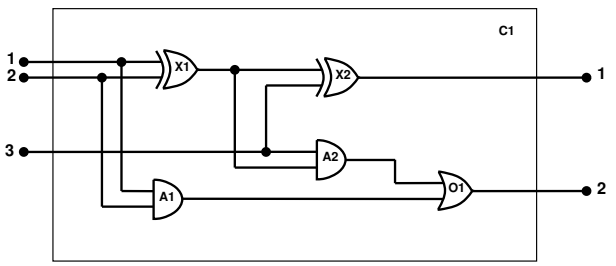
- Języki reprezentacji wiedzy.
- Bazy wiedzy.
- Procedury dowodzenia.
- Dowodzenie.

Metodologia konstruowania bazy wiedzy

1. Analiza dziedziny.
2. Wybór alfabetu.
3. Opis ogólnej wiedzy o dziedzinie.
4. Opis problemu.
5. Testowanie.

Uwaga: Nie należy oczekiwać, że skonstruowana baza wiedzy będzie od razu poprawna i pełna.

Przykład: prosty sumator



1. Analiza problemu.

- Ważne: bramki, wejście-wyjście, połączenia.
- Nieważne: przewody, wielkość, kolor.

2. Alfabet

- **Stałe**. Bramki: $X_1, X_2, A_1, A_2, O1$. Rodzaje bramek: XOR, OR, AND, NOT. Możliwe sygnały: On i Off.
- **Symbole funkcyjne**. $Type$ (zwraca rodzaj bramki), In i Out (zwracają wejścia i wyjścia (*terminale*) bramek, np. $In(1, X_1)$), $Signal$ (zwraca sygnał terminala).
- **Symbole predykatowe**. $Connected$ (zachodzi, jeśli dwa terminale są połączone).

Uwaga: To nie jest jedyny możliwy alfabet, który może być użyty do poprawnej formalizacji prostego sumatora.

3. Opis ogólnej wiedzy o dziedzinie

(Poniżej C jest skrótem od $Connected$.)

1. $\forall t_1, t_2\ C(t_1, t_2) \Rightarrow Signal(t_1) = Signal(t_2)$.
2. $\forall t\ Signal(t) = On \vee Signal(t) = Off$.
3. $On \neq Off$.
4. $\forall t_1, t_2\ C(t_1, t_2) \Leftrightarrow C(t_2, t_1)$.
5. $\forall g\ Type(g) = OR \Rightarrow$
   $[Signal(Out(1, g)) = On \Leftrightarrow$
   $\exists n\ Signal(In(n, g)) = On]$.
6. $\forall g\ Type(g) = AND \Rightarrow$
   $[Signal(Out(1, g)) = Off \Leftrightarrow$
   $\exists n\ Signal(In(n, g)) = Off]$.
7. $\forall g\ Type(g) = XOR \Rightarrow$
   $[Signal(Out(1, g)) = On \Leftrightarrow$
   $Signal(In(1, g)) \neq Signal(In(2, g))]$.
8. $\forall g\ Type(g) = NOT \Rightarrow$
   $[Signal(Out(1, g)) \neq Signal(In(1, g))]$.

4. Opis problemu

$$
\begin{aligned}
&Type(X_1) = XOR\\
&Type(X_2) = XOR\\
&Type(A_1) = AND\\
&Type(A_2) = AND\\
&Type(O_1) = OR\\
&C(Out(1, X_1), In(1, X_2))\\
&C(In(1, C_1), In(1, X_1))\\
&\vdots\\
&C(Out(1, O_1), Out(2, C_1))\\
&C(In(3, C_1), In(1, A_2)).
\end{aligned}
$$

5. Testowanie

Pytanie:

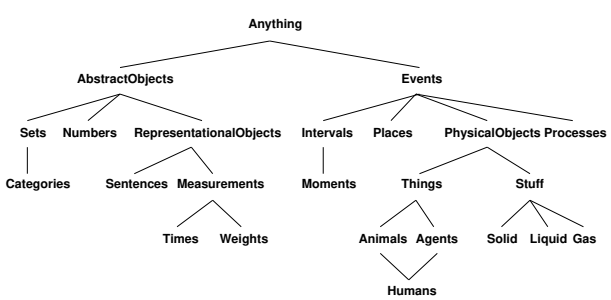
$$
\begin{aligned}
\exists i_1, i_2, i_3\ &(Signal(In(1, C_1)) = i_1)\\
&\wedge (Signal(In(2, C_1)) = i_2)\\
&\wedge (Signal(In(3, C_1)) = i_3)\\
&\wedge (Signal(Out(1, C_1)) = Off)\\
&\wedge (Signal(Out(2, C_1)) = On).
\end{aligned}
$$

Poprawna odpowiedź:

$$
\begin{aligned}
&(i_1 = On \wedge i_2 = On \wedge i_3 = Off) \vee\\
&(i_1 = On \wedge i_2 = Off \wedge i_3 = On) \vee\\
&(i_1 = Off \wedge i_2 = On \wedge i_3 = On).
\end{aligned}
$$

Ontologia ogólna

Dotyczy ogólnej formalizacji fundamentalnych dziedzin wiedzy, tzn. dziedzin, które występują w wielu konkretnych problemach: kategorie, miary, czas, procesy, itp.



Kategorie

Zbiory podobnych obiektów.

Jak reprezentować kategorie?

1. Przy użyciu jednoargumentowych predykatów: $Tomato(a)$.

2. Stosując technikę **reifikacji** – predykat lub funkcja traktowane są jako stała języka: $Tomatoes$ (stała, interpretowana jako obiekt reprezentujący zbiór wszystkich pomidorów).

$Tomato_{12} \in Tomatoes.$
$\forall x\ x \in Tomatoes \Rightarrow x \in Fruits.$
$\forall x\ x \in Tomatoes \Rightarrow Red(x) \wedge Round(x).$

Miary

1. Miary ilościowe

$Length(L_1) = Inches(1.5).$
$\forall l\ Centimeters(2.54 * l) = Inches(l).$
$\forall t\ Centigrade(t) = Fahrenheit(32 + 1.8 * t).$
$Mass(Tomato_{12}) = Kilograms(0.16).$
$\forall d\ d \in Days \Rightarrow (Duration(d) = Hours(24)).$
$\forall b\ \ b \in DollarBills \Rightarrow$
$(CashValue(b) = \$(1.00)).$

2. Miary jakościowe

$\forall e_1, e_2\ \ e_1 \in Exercises\ \wedge e_2 \in Exercises \wedge$
$Wrote(e_1, Norvig) \wedge Wrote(e_2, Russel) \Rightarrow$
$Difficulty(e_1) > Difficulty(e_2).$

Obiekty złożone

$PartOf(Bucharest, Romania).$
$PartOf(Romania, EasternEurope).$
$PartOf(EasternEurope, Europe).$
$\forall x, y, z\ \ PartOf(x, y) \wedge PartOf(y, z) \Rightarrow$
$PartOf(x, z).$

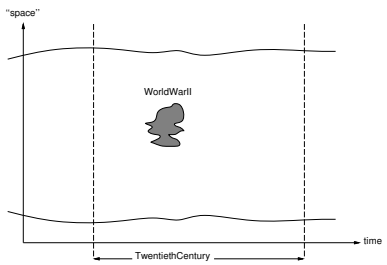
$\forall a\ \ Biped(a) \Rightarrow$
$\exists l_1, l_2, b\ \ Leg(l_1) \wedge Leg(l_2) \wedge Body(b) \wedge$
$PartOf(l_1, a) \wedge PartOf(l_2, a) \wedge PartOf(b, a) \wedge$
$Attached(l_1, b) \wedge Attached(l_2, b) \wedge l_1 \neq l_2 \wedge$
$\forall l_3\ Leg(l_3) \wedge PartOf(l_3, a) \Rightarrow (l_3 = l_1 \vee l_3 = l_2).$

"Obiekt" składający się z trzech jabłek:

$BunchOf(\{Apple_1, Apple_2, Apple_3\}).$

Zdarzenia (events)

Rachunek zdarzeń (Event calculus) – ciągła wersja rachunku sytuacyjnego.



Zdarzenie — coś, co odbywa się w czasie i przestrzeni.

Przedział (interval) — specjalne zdarzenie, posiadające tylko wymiar czasowy. Zawiera, jako podzdarzenia, wszystkie zdarzenia, które wystąpiły w tym przedziale czasu.

$SubEvent(BattleOfBritain, WorldWarII)$.

$SubEvent(WorldWarII, TwentiethCentury)$.

Zdarzenia możemy grupować w kategorie:

$$\exists j\ j \in Journeys \wedge Origin(NewYork, j) \wedge Destination(Delhi, j) \wedge Traveller(Shankar, j) \wedge SubEvent(j, Yesterday).$$

Kategoriom zdarzeń możemy przypisywać nazwy. Np. $Go(x, o, d)$ – x podróżuje z o do d:

$$\forall e, x, o, d\ e \in Go(x, o, d) \Leftrightarrow [e \in Journeys \wedge Traveller(x, e) \wedge Origin(o, e) \wedge Destination(d, e)].$$

Używamy notacji $E(c, i)$, aby wyrazić fakt, że zdarzenie kategorii c jest podzdarzeniem zdarzenia (lub przedziału) i:

$$\forall c, i\ E(c, i) \Leftrightarrow \exists e\ e \in c \wedge SubEvent(e, i).$$

Zatem:

$$E(Go(Shankar, NewYork, Delhi), Yesterday).$$

Miejsca

$In(NewYork, USA)$

$$\forall x, l\ Location(x) = l \Leftrightarrow At(x, l) \wedge \forall l_2\ At(x, l_2) \Rightarrow In(l, l_2).$$

Procesy

Proces – zdarzenie kategorii c, którego każde podzdarzenie należy również do kategorii c.

$E(Flying(Shankar), Yesterday)$.

Używamy notacji $T(c, i)$ na wyrażenie faktu, że zdarzenie c odbyło się dokładnie w przedziale czasowym i.

$T(Working(Stuart), TodayLunchHour)$.

Użyteczna notacja

Nie możemy napisać

$T(At(A_1, Loc_1) \wedge At(A_2, Loc_2), i)$.

Wprowadzamy symbol funkcyjny And zdefiniowany aksjomatem

$$\forall p, q, e\ T(And(p, q), e) \Leftrightarrow T(p, e) \wedge T(q, e).$$

Możemy napisać:

$T(And(At(A_1, Loc_1), At(A_2, Loc_2)), i)$.

Wygodnie jest używać notacji infiksowej i normalnego spójnika $\wedge$. Definiujemy zatem:

$$T(p \wedge q, e) \Leftrightarrow T(And(p, q), e).$$

Używając wyrażenia postaci $T(p \wedge q, e)$, musimy zawsze pamiętać, że $p \wedge q$ jest termem (wskazującym na kategorię zdarzeń) a nie formułą.

Czas i akcje

Podstawowym pojęciem jest **przedział (odcinek) czasu**. Z przedziałem związana jest **długość**. Przedziały o długości 0 nazywamy **momentami czasu**.

$\forall i\ i \in Intervals \Rightarrow$
$[i \in Moments \Leftrightarrow Duration(i) = 0]$.

$\forall i\ Interval(i) \Rightarrow [Duration(i) =$
$(Time(End(i)) - Time(Start(i)))]$.

Możliwe położenia dwóch odcinków czasu:

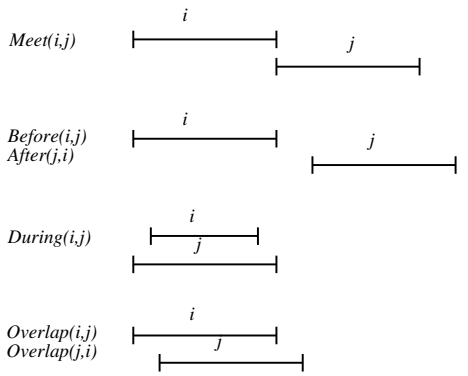
$\forall i, j\ Meet(i, j) \Leftrightarrow$
$Time(End(i)) = Time(Start(j))$.

$\forall i, j\ Before(i, j) \Leftrightarrow$
$Time(End(i)) < Time(Start(j))$.

$\forall i, j\ After(j, i) \Leftrightarrow Before(i, j)$.

$\forall i, j\ During(i, j) \Leftrightarrow$
$Time(Start(j)) \leq Time(Start(i)) \wedge$
$Time(End(i)) \leq Time(End(j))$.

$\forall i, j\ Overlap(i, j) \Leftrightarrow$
$\exists k\ During(k, i) \wedge During(k, j)$.



Relacje pomiędzy odcinkami czasu wykorzystywane są przy definiowaniu akcji:

$\forall x, y, i_0\ T(Engaged(x, y), i_0) \Rightarrow$
$\exists i_1\ [Meet(i_0, i_1) \vee After(i_1, i_0)] \wedge$
$T(Marry(x, y) \vee BreakEngagement(x, y), i_1)$.

Uwaga: $Marry(x, y) \vee BreakEngagement(x, y)$ należy traktować jako term, wskazujący na kategorię zdarzeń.

$\forall x, y, i_0\ T(Marry(x, y), i_0) \Rightarrow$
$\exists i_1\ T(Spouse(x, y), i_1) \wedge Meet(i_0, i_1)$.

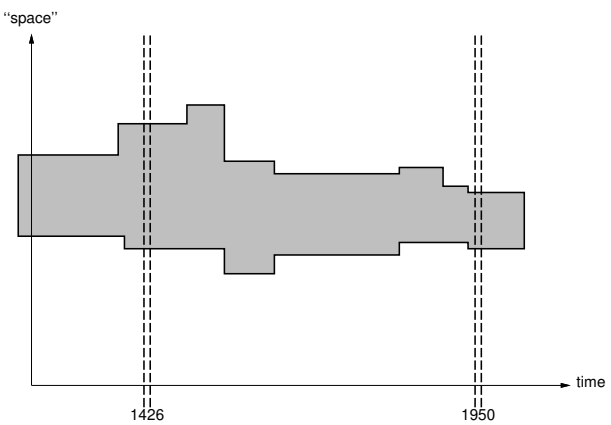
$\forall x, a, b, i_0\ \exists i_1\ T(Go(x, a, b), i_0) \Rightarrow$
$T(In(x, b), i_1) \wedge Meet(i_0, i_1)$.

Fluenty (Fluents)

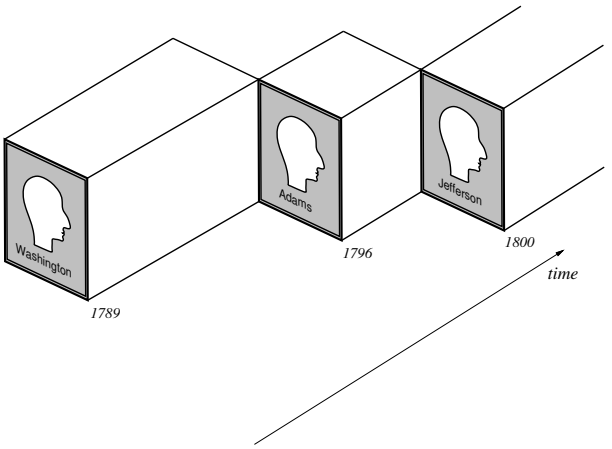
Są to obiekty zmieniające się w czasie.

$T(Area(Poland, SqMiles(233000)), AD1426)$.

$T(Area(Poland, SqMiles(117000)), AD1950)$.



$President(USA)$ – obiekt wskazujący na różne osoby w różnych momentach czasu.



Możemy napisać:

$$T(Male(President(USA)), 19thCentury).$$

Sądy (Propositional attitudes)

Opisują relacje pomiędzy agentem i zdaniem: "a wie, że p", "a wierzy, że p", "a chce aby, p", itp.

Możemy napisać:

$$Believes(Agent, Flies(Superman))$$

pod warunkiem, że $Flies(Superman)$ traktujemy jako term.

Problem: Logika klasyczna jest **ekstensjonalna** – tzn. wartość wyrażenia jest funkcją wartości podwyrażeń tego wyrażenia. W konsekwencji, z faktu $A = B$ wynika

$$Knows(Agent, A) \Leftrightarrow Knows(Agent, B).$$

Jak sobie z tym poradzić?

Użyć logiki modalnej, która jest **intensjonalna**, lub traktować termy reprezentujące zdania jako napisy.

Procedury dowodzenia w logice pierwszego rzędu

Podstawienie: Zbiór postaci

$$\{x_1/t_1, \ldots x_n/t_n\} \qquad (1)$$

gdzie $x_1, \ldots, x_n$ są różnymi zmiennymi, natomiast $t_1, \ldots t_n$ są termami. Dopuszczamy podstawienie **puste**, oznaczane ϵ.

Niech θ będzie podstawieniem postaci (1) i niech α będzie wyrażeniem (tzn. formułą lub termem). Piszemy $SUBST(\theta, \alpha)$ na oznaczenie wyrażenia powstałego z α w wyniku jednoczesnego zastąpienia wszystkich wolnych wystąpień zmiennych $x_1, \ldots, x_n$ termami $t_1, \ldots t_n$. Na przykład, jeśli $\theta = \{x/Sam, y/Pam\}$, to

$$SUBST(\theta, Likes(x, y)) = Likes(Sam, Pam).$$

Literał: Formuła atomowa lub negacja formuły atomowej.

Zdanie: Formuła bez zmiennych wolnych.

Reguła: Zdanie postaci

$$\forall \vec{x}\ (L_1 \wedge \cdots \wedge L_{n-1} \Rightarrow L_n)$$

gdzie $L_1, \ldots, L_n$ $(n > 1)$ są literałami różnymi od $True$ i $False$.

Zakładamy, że każda formuła bazy wiedzy jest albo regułą, albo formułą postaci $\forall \vec{x}\ L$, gdzie L jest literałem.

Powyższe założenie pozwala opuścić kwantyfikatory (uniwersalne).

Uogólniona reguła odrywania:

$$\frac{L'_1, \ldots, L'_{n-1},\ (L_1 \wedge \cdots \wedge L_{n-1} \Rightarrow L_n)}{SUBST(\theta, L_n)}$$

gdzie $L'_1, \ldots, L'_{n-1}$, $L_1, \ldots L_n$ są literałami, natomiast θ jest podstawieniem takim, że dla każdego $i \in \{1, \ldots, n-1\}$, $SUBST(\theta, L'_i) = SUBST(\theta, L_i)$.

Złożeniem podstawień θ_1, θ_2 jest podstawienie, oznaczane $COMPOSE(\theta_1, \theta_2)$, którego efekt sprowadza się do kolejnego wykonania θ_1 i θ_2. Zatem:

$$SUBST(COMPOSE(\theta_1, \theta_2), E) = SUBST(\theta_2, SUBST(\theta_1, E)).$$

Podstawienie θ jest **unifikatorem** wyrażeń $E_1, \ldots, E_n$ jeśli $SUBST(\theta, E_1) = SUBST(\theta, E_2) = \cdots = SUBST(\theta, E_n)$.

Podstawienie $\{x/A\}$ jest unifikatorem wyrażeń $P(x), P(a)$. Wyrażenia $P(x), Q(b)$ nie są unifikowalne.

Podstawienie θ jest **najogólniejszym** unifikatorem wyrażeń $E_1, \ldots, E_n$, jeśli dla każdego unifikatora tych wyrażeń, γ, istnieje podstawienie λ takie, że $\gamma = COMPOSE(\theta, \lambda)$.

Unifikowalne wyrażenia posiadają dokładnie jeden najogólniejszy unifikator.

Wyrażenia $P(John, x)$, $P(y, z)$ posiadają nieskończenie wiele unifikatorów:

$$\{y/John, x/z\},\ \{y/John, x/z, w/Freda\},$$
$$\{y/John, x/John, z/John\} \ldots.$$

Najogólniejszy unifikator $-$ $\{y/John, x/z\}$.

Problem unifikacji: Stwierdzić, czy dane wyrażenia $E_1, \ldots, E_n$ są unifikowalne. Jeśli tak, znaleźć ich najogólniejszy unifikator.

Problem unifikacji jest rozstrzygalny.

Zakładamy, że mamy funkcję $UNIFY$, która dla zadanych wyrażeń $E_1, \ldots, E_n$, zwraca ich najogólniejszy unifikator θ lub specjalną stałą $fail$, jeśli unifikator nie istnieje.

Wariant (renaming) — Formuła A jest wariantem formuły B, jeśli A można otrzymać z B, zastępując niektóre zmienne, zmiennymi nie występującymi w B.

$Likes(x, John)$ jest wariantem $Likes(y, John)$. $Likes(x, x)$ nie jest wariantem $Likes(x, y)$.

Formuły naszej bazy wiedzy są takiej postaci, że każda może być zastąpiona swoim wariantem. (Powstała w ten sposób nowa baza wiedzy będzie równoważna starej.)

Dlaczego należy przemianowywać zmienne?

$Knows(x, Elizabeth)$.

$Knows(John, x) \Rightarrow Hates(John, x)$.

Powinniśmy, stosując ogólną regułę odrywania, wywnioskować $Hates(John, Elizabeth)$. Niestety jest to niemożliwe: formuły $Knows(John, x)$ i $Knows(x, Elizabeth)$ nie są unifikowalne. Jeśli przemianujemy zmienne w jednej z formuł, np. formułę $Knows(x, Elizabeth)$ zastąpimy przez $Knows(y, Elizabeth)$, będziemy mogli wyprowadzić $Hates(John, Elizabeth)$.

Możemy założyć: Przesłanki uogólnionej reguły odrywania nie zawierają wspólnych zmiennych.

Procedury dowodzenia oparte na uogólnionej regule odrywania

Metoda progresywna (Forward-chaining).

Stosujemy, kiedy nowy fakt p zostaje dodany do bazy wiedzy KB. Procedura znajduje nowe fakty, które wynikają z $KB \cup \{p\}$.

```
procedure FORWARD-CHAIN(KB, p)

  if there is a sentence in KB that is a renaming of p then return
  Add p to KB
  for each (p1 ∧ ... ∧ pn ⇒ q) in KB such that for some i, UNIFY(pi, p) = θ succeeds do
      FIND-AND-INFER(KB, [p1, ..., pi−1, pi+1, ..., pn], q, θ)
  end

procedure FIND-AND-INFER(KB, premises, conclusion, θ)

  if premises = [ ] then
      FORWARD-CHAIN(KB, SUBST(θ, conclusion))
  else for each p' in KB such that UNIFY(p', SUBST(θ, FIRST(premises))) = θ2 do
      FIND-AND-INFER(KB, REST(premises), conclusion, COMPOSE(θ, θ2))
  end
```


Metoda regresywna (Backward-chaining).

Stosujemy, kiedy chcemy otrzymać odpowiedź na pytanie zadane bazie wiedzy.

```
function BACK-CHAIN(KB, q) returns a set of substitutions

  BACK-CHAIN-LIST(KB, [q], {})

function BACK-CHAIN-LIST(KB, qlist, θ) returns a set of substitutions
  inputs: KB, a knowledge base
          qlist, a list of conjuncts forming a query (θ already applied)
          θ, the current substitution
  static: answers, a set of substitutions, initially empty

  if qlist is empty then return {θ}
  q ← FIRST(qlist)
    for each q'i in KB such that θi ← UNIFY(q, q'i) succeeds do
      Add COMPOSE(θ, θi) to answers
    end
    for each sentence (p1 ∧ ... ∧ pn ⇒ q'i) in KB such that θi ← UNIFY(q, q'i) succeeds do
      answers ← BACK-CHAIN-LIST(KB, SUBST(θi, [p1 ... pn]), COMPOSE(θ, θi)) ∪ answers
    end
  return the union of BACK-CHAIN-LIST(KB, REST(qlist), θ) for each θ ∈ answers
```


Przykład użycia metody regresywnej

$American(x) \wedge Weapon(y) \wedge Nation(z) \wedge$
$Hostile(z) \wedge Sells(x, z, y) \Rightarrow Criminal(x).$

$Owns(Nono, M1).$

$Missile(M1).$

$Owns(Nono, x) \wedge Missile(x) \Rightarrow$
$Sells(West, Nono, x).$

$Missile(x) \Rightarrow Weapon(x).$

$Enemy(x, America) \Rightarrow Hostile(x).$

$American(West).$

$Nation(Nono).$

$Enemy(Nono, America).$

$Nation(America).$

Metody wnioskowania oparte na uogólnionej regule odrywania są niepełne.

$\forall x\ P(x) \Rightarrow Q(x).$

$\forall x\ \neg P(x) \Rightarrow R(x).$

$\forall x\ Q(x) \Rightarrow S(x).$

$\forall x\ R(x) \Rightarrow S(x).$

$T(a).$

Z powyższych formuł wynika $S(A)$. ($S(A)$ jest prawdziwa jeśli $Q(A)$ lub $R(A)$ jest prawdziwa, a jedna z tych formuł musi być prawdziwa, ponieważ $P(A)$ jest prawdziwa lub $\neg P(A)$ jest prawdziwa.) Niestety formuły $S(A)$ nie można wyprowadzić ani metodą progresywną ani regresywną.

Metoda rezolucji (Resolution method)

Metoda rezolucji (Robinson, 1965) jest pełną (i stosunkowo efektywną) procedurą dowodzenia twierdzeń w logice pierwszego rzędu.

Problem dowodzenia twierdzeń: Dla danej bazy wiedzy KB i danej formuły α, stwierdzić, czy $KB \models \alpha$. (Problem półrozstrzygalny.)

Problem dowodzenia twierdzeń jest równoważny **problemowi spełnialności**: Dla danej bazy wiedzy KB i danej formuły α, stwierdzić, czy zbiór $KB \cup \{\neg\alpha\}$ jest spełnialny.

Ponieważ bazy wiedzy są skończone, $KB \cup \{\neg\alpha\}$ można zawsze traktować jako pojedynczą formułę.

Metoda rezolucji, dla danej formuły β, próbuje stwierdzić, czy β jest spełnialna.

Metodę rezolucji stosuje się dla formuł w postaci **klauzulowej**.

Klauzula – formuła postaci

$$l_1 \vee \cdots \vee l_n \quad (n \geq 0) \tag{6}$$

gdzie $l_1, \ldots, l_n$ są literałami. Jeśli $n = 0$, klauzulę (6) nazywamy **kluzulą pustą** i oznaczamy symbolem $\bot$.

Postać kluzulowa:

$$\forall\, \overline{x_1}\, C_1 \wedge \cdots \wedge \forall\, \overline{x_m}\, C_m \quad (m \geq 1)$$

gdzie $C_1, \ldots, C_n$ są klauzulami, natomiast $\overline{x_i}$ jest listą wszystkich zmiennych występujących w C_i.

Twierdzenie Istnieje algorytm, który dla dowolnej formuły A znajduje formułę B taką, że: (1) B jest w postaci klauzulowej. (2) A jest spełnialna wtw gdy B jest spełnialna. (A i B nie muszą być równoważne.)

Sprowadzanie formuły A do postaci klauzulowej

1. Niech $\overline{x}$ będzie listą wszystkich zmiennych wolnych występujących w A. Zastępujemy A formułą $\exists\, \overline{x}\, A$.

2. Eliminujemy wszystkie redundantne kwantyfikatory, tzn. każdy kwantyfikator postaci $\forall x$ lub $\exists x$ taki, że zmienna x nie występuje w obrębie działania tego kwantyfikatora.

3. Przemianowujemy zmienne tak, aby wszystkie kwantyfikowane zmienne były różne.

4. Eliminujemy "$\Rightarrow$" i "$\Leftrightarrow$", zastępując $B \Rightarrow C$ przez $\neg B \vee C$ i $B \Leftrightarrow C$ przez $((\neg B \vee C) \wedge (\neg C \vee B))$.

5. Przesuwamy "$\neg$" w prawo, zastępując $\neg\forall x B$ przez $\exists x\ \neg B$, $\neg\exists x\ B$ przez $\forall x\ \neg B$, $\neg(B \vee C)$ przez $\neg B \wedge \neg C$, $\neg(B \wedge C)$ przez $\neg B \vee \neg C$, $\neg\neg B$ przez B, tak długo aż wszystkie symbole negacji występują bezpośrednio przed formułami atomowymi.

6. (Krok opcjonalny. Poniżej Q oznacza $\forall$ lub $\exists$; $\Diamond$ oznacza $\wedge$ lub $\vee$.) Przesuwamy kwantyfikatory w prawo, zastępując $Qx(B \Diamond C)$ przez $B \Diamond (QxC)$, jeśli x nie jest wolna w B, i przez $(QxB) \Diamond C$, jeśli x nie jest wolna w C.

7. Skolemizacja (Usunięcie kwantyfikatorów egzystencjalnych). Powtarzamy poniższy proces, aż wszystkie kwantyfikatory $\exists$ zostaną usunięte.

Bierzemy dowolną podformułę postaci $\exists y\ B(y)$. Niech $x_1, \ldots, x_n$ $(n \geq 0)$ będzie listą wszystkich zmiennych wolnych występujących w $\exists y\ B(y)$, które są uniwersalnie kwantyfikowane w pewnej nadformule formuły $\exists y\ B(y)$. Jeśli $n = 0$, zastępujemy $\exists y\ B(y)$ przez $B(A)$, gdzie A jest **nową** stałą indywiduową. Jeśli $n > 0$, zastępujemy $\exists y\ B(y)$ przez $B(F(x_1, \ldots, x_n))$, gdzie F jest **nowym** symbolem funkcji n-argumentowej.

8. Usuwamy wszystkie kwantyfikatory uniwersalne.

9. Rozdzielamy $\wedge$ względem $\vee$, zastępując $(B \wedge C) \vee D$ przez $(B \vee D) \wedge (C \vee D)$ oraz $B \vee (C \wedge D)$ przez $(B \vee C) \wedge (B \vee D)$.

10. Eliminujemy stałe $True$ i $False$. Aktualna formuła jest koniunkcją klauzul. Usuwamy każdą klauzulę zawierającą literał $\neg False$ lub $True$. Z pozostałych klauzul usuwamy każdy literał $\neg True$ i $False$.

11. Dodajemy kwantyfikatory uniwersalne. Aktualna formuła jest postaci $C_1 \wedge \cdots \wedge C_k$, gdzie $C_1, \ldots, C_k$ są klauzulami. Zastępujemy ją formułą

$$\forall\, \overline{x_1}\, C_1 \wedge \cdots \wedge \forall\, \overline{x_k}\, C_k$$

gdzie $\overline{x_i}$ jest listą wszystkich zmiennych występujących w C_i.

Uwaga: Kroki 1 i 7 nie zachowują równoważności. (Zachowują spełnialność.)

Formułę w postaci kluzulowej

$$\forall\, \overline{x_1}\, C_1 \wedge \cdots \wedge \forall\, \overline{x_m}\, C_m$$

możemy zapisać jako zbiór klauzul, tzn. w postaci $\{C_1, \ldots, C_m\}$. Możemy też zawsze opuścić kwantyfikatory.

Literał dodatni – formuła atomowa.

Literał ujemny – negacja formuły atomowej.

$| l |$ oznacza formułę atomową literału l.

Piszemy $l_1, \ldots, l_n \in C$, aby wyrazić fakt, że literały $l_1, \ldots, l_n$ występują w kluzuli C. Jeśli $l_1, \ldots, l_n \in C$, to $C - \{l_1, \ldots, l_n\}$ oznacza klauzulę otrzymaną z C w wyniku usunięcia $l_1, \ldots, l_n$.

Dla wyrażenia E i podstawienia θ, piszemy $E\theta$ jako skrót zapisu $SUBST(\theta, E)$.

Reguła rezolucji (Resolution rule)

Definicja 1 Niech C_1 i C_2 będą klauzulami bez wspólnych zmiennych. Niech $l_1, \ldots, l_n \in C_1$ $(n > 0)$ i $l'_1, \ldots, l'_k \in C_2$ $(k > 0)$. Załóżmy ponadto, że wszystkie literały $l_1, \ldots, l_n$ są dodatnie i wszystkie literały $l'_1, \ldots, l'_k$ są ujemne, lub odwrotnie. Jeśli θ jest najogólniejszym unifikatorem zbioru $\{| l_1 |, \ldots, | l_n |, | l'_1 |, \ldots, | l'_k |\}$, to kluzulę

$$(C_1\theta - \{l_1\theta, \ldots, l_n\theta\}) \vee (C_2\theta - \{l'_1\theta, \ldots, l'_k\theta\})$$

nazywamy **binarną rezolwentą** klauzul C_1 i C_2.

Przykład Niech

$C_1 = P(x) \vee P(y) \vee Q(B)$.
$C_2 = R(A) \vee \neg P(F(A))$.

Klauzula $Q(B) \vee R(A)$ jest binarną rezolwentą klauzul C_1 i C_2. $(l_1 = P(x), l_2 = P(y), l'_1 = \neg P(F(A))$; $\theta = \{x/F(A), y/F(A)\}$.)

Definicja 2 **Rezolwentą** klauzul C_1 i C_2, oznaczaną $RES(C_1, C_2)$, nazywamy dowolną binarną rezolwentę wariantu klauzuli C_1 i wariantu klauzuli C_2.

Definicja 3 **Regułą rezolucji** nazywamy regułę wnioskowania

$$\frac{C_1, C_2}{RES(C_1, C_2)}.$$

Dowód rezolucyjny ze zbioru klauzul CL:
Ciąg klauzul $C_1, \ldots, C_n$ taki, że $C_n = \perp$ i dla każdego $1 \leq i \leq n$, $C_i \in CL$ lub C_i jest rezolwentą C_j i C_k, gdzie $j, k < i$.

Twierdzenie Niech CL będzie zbiorem klauzul nie zawierających symbolu "=". CL jest niespełnialny wtw gdy istnieje dowód rezolucyjny z CL.

Aksjomaty równości

CL – zbiór klauzul. Przez $EQ(CL)$ oznaczamy **zbiór aksjomatów równości dla** CL. $EQ(CL)$ zawiera następujące klauzule:

$x = x$.

$x \neq y \vee y = x$.

$x \neq y \vee y \neq z \vee x = z$.

$x_1 \neq y_1 \vee \cdots \vee x_n \neq y_n \vee$
$\neg P(x_1, \ldots, x_n) \vee P(y_1, \ldots, y_n)$,
dla każdego n-argumentowego ($n > 0$) symbolu predykatowego P występującego w CL.

$x_1 \neq y_1 \vee \cdots \vee x_n \neq y_n \vee$
$(F(x_1, \ldots, x_n) = F(y_1, \ldots, y_n))$,
dla każdego n-argumentowego ($n > 0$) symbolu funkcyjnego F występującego w CL.

Twierdzenie Niech CL będzie zbiorem klauzul z równością. CL jest niespełnialny wtw gdy istnieje dowód rezolucyjny z $CL \cup EQ(CL)$.

Twierdzenie Niech CL będzie zbiorem klauzul nie zawierających symbolu "=". CL jest niespełnialny wtw gdy istnieje liniowy dowód rezolucyjny z CL.

Twierdzenie Niech CL będzie zbiorem klauzul z równością. CL jest niespełnialny wtw gdy istnieje liniowy dowód rezolucyjny ze zbioru $CL \cup EQ(CL)$.

Klauzulę C_0 w liniowym dowodzie rezolucyjnym nazywamy **klauzulą główną**.

Twierdzenie Niech CL będzie niespełnialnym zbiorem klauzul i niech $CL' \subseteq CL$. Jeśli zbiór $CL - CL'$ jest spełnialny, to istnieje liniowy dowód rezolucyjny z CL (ewentualnie z $CL \cup EQ(CL)$) z klauzulą główną $C \in CL'$.

Logiczne systemy wnioskowania

- Udowadniacze twierdzeń (theorem provers) – AURA, OTTER, SCAN.
- Języki programowania w logice (logic programming languages) – PROLOG, LIFE, GÖDEL.
- Systemy produkcyjne (production systems) – OPS-5, CLIPS, SOAR .
- Ramy (frames) – OWL, KODIAK, FRAIL.
- Sieci semantyczne (semantic nets) – CG, NETL, SNEPS

Programowanie w logice

Klauzula Horna – klauzula zawierająca co najwyżej jeden literał dodatni.

Każda klauzula Horna jest jednej z następujących postaci:

$\neg P_1 \vee \cdots \vee \neg P_n \vee Q$	$(n > 0)$	(1)
Q		(2)
$\neg P_1 \vee \cdots \vee \neg P_n$	$(n > 0)$	(3)
$\bot$		(4)

W programowaniu w logice klauzule (1)-(4) zapisujemy:

$Q \leftarrow P_1, \ldots P_n.$

$Q \leftarrow .$

$\leftarrow P_1, \ldots, P_n.$

$\bot.$

Klauzule Horna posiadają odpowiednią interpretację proceduralną.

- Klauzula postaci $Q \leftarrow P_1, \ldots P_n$ traktowana jest jako procedura o nagłówku Q i treści $P_1, \ldots P_n$. Wywołanie procedury Q sprowadza się do kolejnego wywołania procedur $P_1, \ldots P_n$.
- Klauzula postaci Q traktowana jest jako procedura o pustej treści.
- Klauzulę postaci $\leftarrow P_1, \ldots, P_n,$ nazywaną **klauzulą celu**, można traktować jako dane programu.
- Klauzula pusta traktowana jest jako instrukcja *stop*.

Program – uporządkowany ciąg procedur.

Dane – klauzula celu.

Realizacja – (uproszczona) rezolucja liniowa.

Programowanie w logice jest programowaniem **deklaratywnym**: program jest logiczną specyfikacją problemu. Niestety języki programowania w logice nie są w pełni deklaratywne: w celu zwiększenia efektywności języki te dopuszczają wiele mechanizmów pozalogicznych.

Języki programowania w logice posiadają duży zbiór wbudowanych predykatów związanych z arytmetyką i wejściem/wyjściem. Literały odpowiadające tym predykatom realizowane są wykonaniem odpowiedniego kodu, a nie rezolucją.

Systemy produkcyjne

Realizują wnioskowanie progresywne, używając bardzo ograniczonego języka. Są używane w wielu systemach doradczych (ekspertowych).

System produkcyjny składa się z **pamięci roboczej**, zawierającej zdania atomowe, i zbioru **reguł**. Reguła jest wyrażeniem postaci

$$P_1 \wedge \cdots \wedge P_n \Rightarrow act_1 \wedge \cdots act_m$$

gdzie $P_1, \ldots, P_n$ są formułami atomowymi, natomiast $act_1, \ldots, act_m$ są działaniami, które należy wykonać, jeśli wszystkie P_i znajdują się w pamięci roboczej. W takim przypadku regułę nazywamy **realizowalną**. Dopuszczalne akcje, to dodanie (usunięcie) elementu do (z) pamięci roboczej.

Praca systemu produkcyjnego składa się z ciągu **cykli**. Każdy cykl składa się z trzech faz: (1) Fazy dopasowywania (znalezienie realizowalnych reguł) (2) Fazy wyboru (wybór reguł do wykonania i ustalenie ich kolejności). (3) Fazy wykonania (wykonanie wybranych reguł).

Faza dopasowywania

Wystarczy unifikacja. Ale naiwne dopasowywanie jest bardzo nieefektywne. Jeśli pamięć robocza zawiera *w* elementów, system używa *r* reguł, każda reguła zawiera *n* formuł atomowych i celem systemu jest zrealizowanie *c* cykli, to algorytm naiwnego dopasowywania musi wykonać *wrnc* unifikacji. Dla *w =100*, *r=200*, *n=5* i *c=1000*, należy wykonać 100 000000 unifikacji!

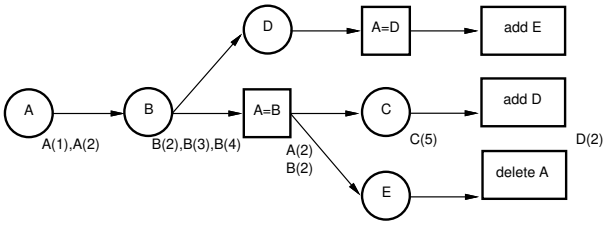
Algorytm dopasowywania "rete"

Pamięć robocza:

$$\{A(1), A(2), B(2), B(3), B(4), C(5)\}$$

Reguły:

$A(x) \wedge B(x) \wedge C(y) \Rightarrow$ add $D(x)$.
$A(x) \wedge B(y) \wedge D(x) \Rightarrow$ add $E(x)$.
$A(x) \wedge B(x) \wedge E(z) \Rightarrow$ delete $A(x)$.



Faza wyboru

Niektóre systemy produkcyjne wykonują wszystkie realizowalne działania. Inne systemy wykonują tylko niektóre. Kilka możliwych strategii wyboru:

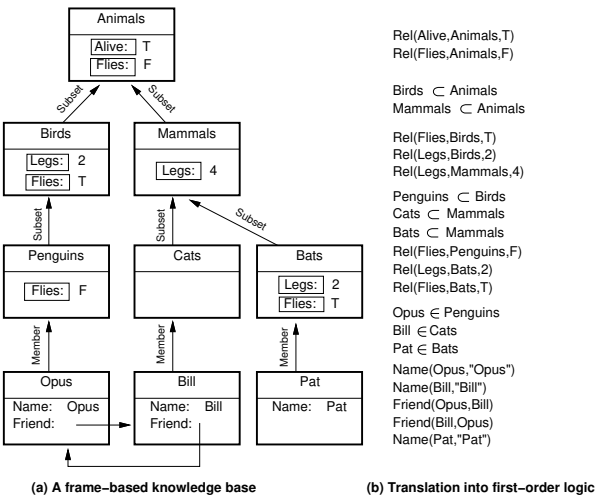
- Preferujemy reguły, które odnoszą się do ostatnio utworzonych elementów pamięci roboczej.
- Preferujemy reguły bardziej "specyficzne". Z poniższych reguł należy wybrać drugą:

  $Mammal(x) \Rightarrow$ add $Legs(x, 4)$.
  $Mammal(x) \wedge Human(x) \Rightarrow$ add $Legs(x, 2)$.

- Preferujemy działania, którym nadano wyższy priorytet. Z poniższych reguł powinniśmy wybrać drugą:

  $P(x) \Rightarrow Action(Dust(x))$
  $R(x) \Rightarrow Action(Evacuate)$.

Ramy i sieci semantyczne



Logiczna reprezentacja powyższej sieci oparta jest na technice reifikacji. Unikamy przez to sprzeczności, ale tracimy na ekspresyjności. Należy dodać dodatkowe aksjomaty. Poniżej: $Val(R, x, y)$ – formuła $Rel(R, x, y)$ znajduje się w bazie wiedzy; $Holds(R, x, y)$ – x i y znajdują się w relacji R.

$$\forall r, x, b\ Holds(r, x, b) \equiv [Val(r, x, b) \vee \exists p\ x \in p \wedge Rel(r, p, b) \wedge \neg InterveningRel(x, p, r)].$$

$$\forall x, p, r\ InterveningRel(x, p, r) \equiv \exists i\ Intervening(x, i, p) \wedge \exists b' Rel(r, i, b').$$

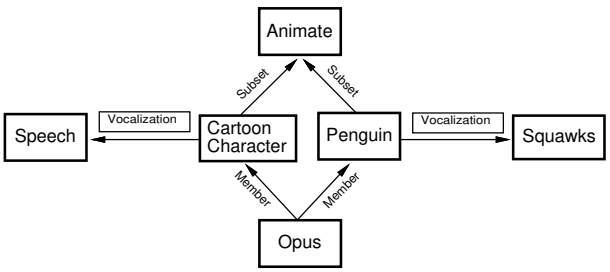
$$\forall x, i, p\ Intervening(x, i, p) \equiv (x \in i) \wedge (i \subset p).$$

$$\forall r, a, b\ Rel(r, a, b) \equiv [r, a, b] \in \{[Alive, Animal, T], \ldots\}.$$

$$\forall r, a, b\ Val(r, a, b) \equiv [r, a, b] \in \{[Friend, Opus, Bill], \ldots\}.$$

Rozważana sieć jest **niemonotoniczna**: dodanie nowej przesłanki może unieważnić dotychczas wyprowadzalne twierdzenia. Np. dodanie $Rel(Legs, Cats, 3)$ unieważnia wyprowadzalny poprzednio fakt, że $Bill$ ma cztery nogi.

Niektóre sieci semantyczne dopuszczają **wielokrotne dziedziczenie**: obiekt może należeć do kilku klas i dziedziczyć ich własności. Może to prowadzić do konfliktów:



Planowanie

Prosty agent planujący:

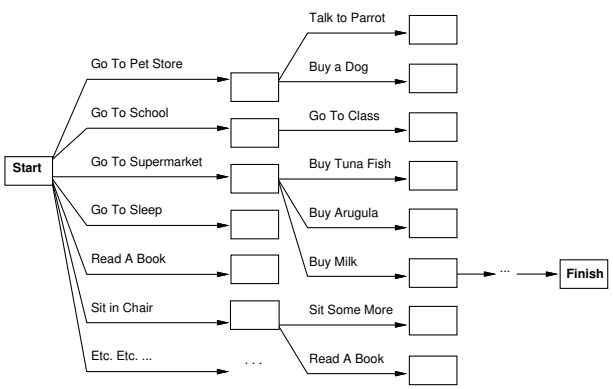
function SIMPLE-PLANNING-AGENT(*percept*) **returns** an *action*
static: *KB*, a knowledge base (includes action descriptions)
p, a plan, initially *NoPlan*
t, a counter, initially 0, indicating time
local variables: *G*, a goal
current, a current state description

TELL(*KB*, MAKE-PERCEPT-SENTENCE(*percept*, *t*))
current ← STATE-DESCRIPTION(*KB*, *t*)
if *p* = *NoPlan* **then**
G ← ASK(*KB*, MAKE-GOAL-QUERY(*t*))
p ← IDEAL-PLANNER(*current*, *G*, *KB*)
if *p* = *NoPlan* **or** *p* is empty **then** *action* ← *NoOp*
else
action ← FIRST(*p*)
p ← REST(*p*)
TELL(*KB*, MAKE-ACTION-SENTENCE(*action*, *t*))
t ← *t* + 1
return *action*

Każde zadanie znalezienia planu można teoretycznie rozwiązać posługując się jedną z metod przeszukiwania przestrzeni stanów. Jest to jednak często niewykonalne w praktyce, ze względu na skomplikowany opis stanów i wysoki współczynnik rozgałęzienia.

Celem agenta jest skonstruowanie planu, gwarantującego posiadanie szklanki mleka, kilku bananów i wiertarki.

Mały fragment przestrzeni stanów dla tego zadania:



Systemy planujące są znacznie bardziej wyrafinowane od systemów realizujących przeszukiwanie przestrzeni stanów.

Podstawowe cechy systemów planujących

- Wykorzystanie języka formalnego (na ogół logiki) do opisu przestrzeni stanów, celu i akcji. (Zwiększenie ekspresji.)
- Konstruowanie planu metodą zstępującą. Zaczynamy od planu przybliżonego, który później ulepszamy. (Zwiększenie efektywności.)
- Wykorzystanie metody "dziel i zwyciężaj" przy konstruowaniu planu. Typowe systemy planujące dzielą cel na podcele, które próbują kolejno zrealizować. (Zwiększenie efektywności.)

Planowanie w rachunku sytuacyjnym

Stan początkowy:

$At(Home, S_0) \wedge \neg Have(Milk, S_0) \wedge$
$\neg Have(Bananas, S_0) \wedge \neg Have(Drill, S_0).$

Cel:

$\exists s\ At(Home, s) \wedge Have(Milk, s) \wedge$
$Have(Bananas, s) \wedge Have(Drill, s).$

Akcje:

$\forall a, s\ Have(Milk, Result(a, s)) \Leftrightarrow$
$[a = Buy(Milk) \wedge At(Supermarket, s)] \vee$
$[Have(Milk(s) \wedge a \neq Drop(Milk)].$

Jeśli do opisu sytuacji początkowej i opisu akcji dodamy zaprzeczenie formuły celu i użyjemy rezolucji, otrzymamy odpowiedź w rodzaju:

$\{s/Result(Go(Home), Result(Buy(Drill),$
$Result(Go(HardwareStore),$
$Result(Buy(Banana), Result(Buy(Milk),$
$Result(Go(SuperMarket), S_0))))))\}.$

Rachunek sytuacyjny jest bardzo eleganckim i ekspresyjnym formalizmem planującym. Niestety, z praktycznego punktu widzenia, jest nieefektywny.

Planowanie efektywne wymaga:

(1) Użycia podzbioru logiki pierwszego rzędu jako języka formalizacji.

(2) Użycia wyspecjalizowanych algorytmów do planowania zamiast udowadniacza twierdzeń ogólnego przeznaczenia typu metody rezolucji.

Większość systemów planujących wywodzi się z systemu STRIPS (STanford Research Institute Problem Solver, 1970).

Reprezentacja stanów, celu i akcji w systemach typu STRIPS

Stany reprezentowane są jako koniunkcja literałów nie zawierających zmiennych:

$At(Home) \wedge \neg Have(Milk) \wedge \cdots$

Cel jest dowolną koniunkcją literałów:

$At(home) \wedge Have(Milk) \wedge \cdots$

Formuła celu może zawierać zmienne:

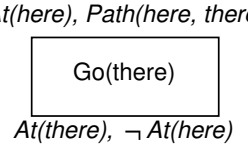
$At(x) \wedge Sells(x, Milk).$

Zakłada się, że zmienne występujące w formule celu są egzystencjalnie kwantyfikowane.

Opis akcji

- **Nazwa akcji**.
- **Warunek wstępny (precondition)** – koniunkcja literałów dodatnich, które muszą zachodzić, aby akcja była wykonywalna.
- **Efekty akcji** – Koniunkcja literałów opisujących zmiany stanu w wyniku wykonania akcji.

$Op(ACTION : Go(there),$
$PRECOND : At(here) \wedge Path(here, there),$
$EFFECT : At(there) \wedge \neg At(here)).$



Uwaga: Powyższy opis definiuje całą klasę akcji.

Stany będziemy często zapisywali jako zbiory literałów.

Akcja jest **stosowalna** w stanie *s* jeśli wszystkie warunki wstępne tej akcji są zawarte w *s*. W wyniku wykonania akcji A w stanie *s* otrzymujemy stan, który: (1) Zawiera wszystkie literały występujące w koniunkcji efektów akcji. (2) Zawiera wszystkie literały występujące w *s*, za wyjątkiem tych, których negacje występują w koniunkcji efektów akcji.

Jeśli stan *s* zawiera

$$At(Home), Path(Home, Supermarket), \ldots$$

to akcja $Go(Supermarket)$ jest stosowalna i w wyniku wykonania tej akcji otrzymamy:

$$\neg At(Home), At(Supermarket), \\ Path(Home, Supermarket), \ldots$$

System planujący może być oparty na przeszukiwaniu przestrzeni stanów lub przeszukiwaniu **przestrzeni planów**. To drugie rozwiązanie jest efektywniejsze.

Jak reprezentować plany?

Cel: $RightShoeOn \wedge LeftShoeOn$.

Stan początkowy: $True$.

Akcje:

Op(ACTION: $RightShoe$,
PRECOND: $RightSockOn$,
EFFECT: $RightShoeOn$).

Op(ACTION: $LeftShoe$,
PRECOND: $LeftSockOn$,
EFFECT: $LeftShoeOn$).

Op(ACTION: $RightSock$,
PRECOND: $True$,
EFFECT: $RightSockOn$).

Op(ACTION: $LeftSock$,
PRECOND: $True$,
EFFECT: $LeftSockOn$).

Plan częściowy— częściowo uporządkowany zbiór schematów akcji.

Plan liniowy — liniowo uporządkowany zbiór schematów akcji.

Plan ukonkretniony (częściowy, liniowy) — częściowo (liniowo) uporządkowany zbiór akcji.

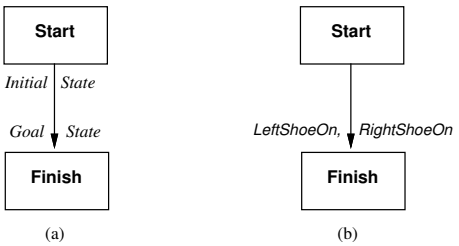
Linearyzacja planu częściowego P — każdy plan liniowy zawierający dokładnie akcje (schematy akcji) planu P, zgodny z częściowym porządkiem określonym na P.

Plan reprezentujemy jako strukturę danych zawierającą:

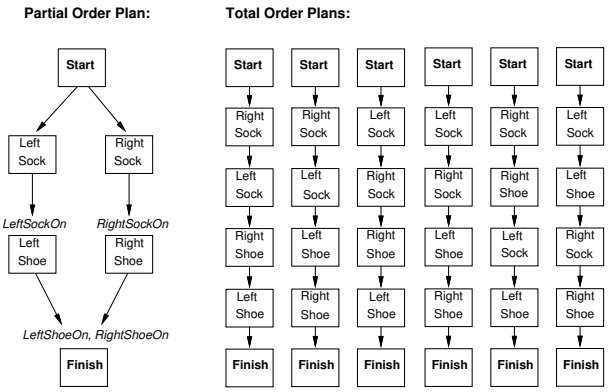
- Zbiór **operatorów (plan steps)**: akcje i (lub) schematy akcji.
- Zbiór **węzłów porządkujących (ordering constraints)**: wyrażenia postaci $S_i \prec S_j$, gdzie S_i, S_j są operatorami.
- Zbiór **ograniczeń zmiennych (variable constraints)**: wyrażenia postaci $v = x$, gdzie v jest zmienną, natomiast x jest inną zmienną lub stałą.
- Zbiór **powiązań przyczynowych (causal links)**: wyrażenia postaci $S_i \xrightarrow{c} S_j$, interpretowane: operator S_i zapewnia warunek wstępny c dla operatora S_j.

Plan początkowy: składa się z pary operatorów, $Start$ i $Finish$, powiązanych węzłem $Start \prec Finish$. Operator $Start$ nie posiada warunków wstępnych. Jego efektem działania jest realizacja stanu początkowego. Warunkiem wstępnym operatora $Finish$ jest formuła celu. Jego efektem działania jest akcja pusta. Plan początkowy nazywany jest też **planem minimalnym**.

Poniżej: (a) Ogólna struktura planu początkowego. (b) Plan początkowy dla problemu zakładania butów.



Poniżej: częściowy plan (będący rozwiązaniem problemu zakładania butów) i jego sześć linearyzacji.



Co to jest rozwiązanie problemu planowania (plan końcowy)?

Plan końcowy powinien być ukonkretniony.

Dopuszczamy natomiast plany częściowe jako rozwiązanie zadania planowania, ponieważ:

1. Plan częściowy może posiadać wiele linearyzacji. Nie widać powodu, dlaczego system planujący miałby wybrać jedną z nich jako rozwiązanie.

2. Plan częściowy może być wykonany w trybie równoległym.

3. Plan częściowy można w naturalny sposób wykorzystać jako "podplan" przy rozwiązywaniu większego zadania.

Plan pełny – plan, w którym warunek wstępny każdego operatora jest zapewniony w wyniku realizacji innego operatora. Dokładniej, dla dowolnego warunku wstępnego c dowolnego operatora S_i, istnieje operator S_j taki, że:

1. $S_j \prec S_i$.
2. $c \in \text{EFFECTS}(S_j)$.
3. Plan nie zawiera operatora S_k takiego, że $\neg c \in \text{EFFECTS}(S_k)$ i $S_j \prec S_k \prec S_i$.

Plan poprawny – plan, w którym zarówno zbiór węzłów porządkujących, jak i zbiór ograniczeń zmiennych są niesprzeczne.

Każdy plan pełny i poprawny jest planem końcowym, czyli rozwiązaniem zadania planowania.

Konstruowanie planu

Ogólny pomysł: Zaczynamy od planu początkowego. W kolejnym kroku rozszerzamy plan, zapewniając jeden z warunków wstępnych istniejącego już operatora (albo poprzez dodanie nowego operatora, albo poprzez dodanie połączenia przyczynowego biegnącego od wcześniej dodanego operatora). Powtarzamy tak długo, aż otrzymamy plan końcowy lub kolejny krok jest niewykonalny. W drugim przypadku, jeśli możliwe, robimy nawrót.

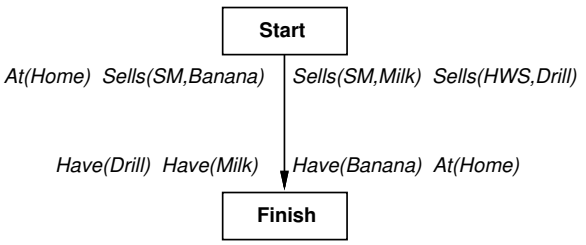
Op(ACTION: $Start$,
EFFECT: $At(Home) \wedge Sells(HWS, Drill) \wedge Sells(SM, Milk) \wedge Sells(SM, Banana)$).

Op(ACTION: $Finish$,
PRECOND: $Have(Drill) \wedge Have(Milk) \wedge Have(Banana) \wedge At(Home)$).

Op(ACTION: $Go(there)$,
PRECOND: $At(here)$,
EFFECT: $At(there) \wedge \neg At(here)$).

Op(ACTION: $Buy(x)$,
PRECOND: $At(store) \wedge Sells(store, x)$,
EFFECT: $Have(x)$).

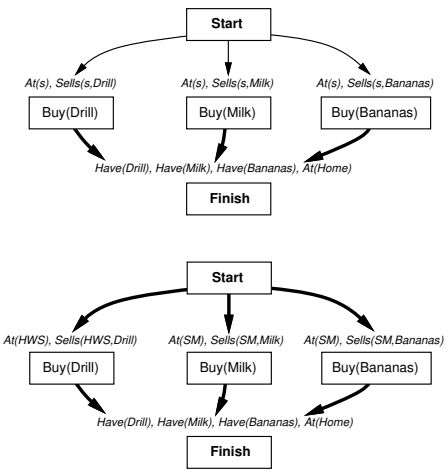
Plan początkowy:



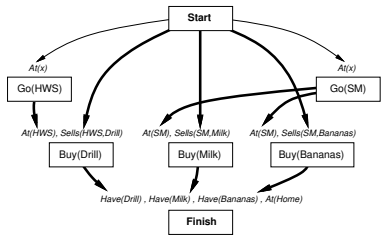
Poniżej: (1) Cienkie strzałki wskazują na porządek pomiędzy operatorami. (2) Pogrubione strzałki reprezentują połączenia przyczynowe. (System planujący "chroni" wszystkie połączenia przyczynowe.)

Góra: Plan zapewniający trzy spośród czterech warunków wstępnych operatora $Finish$.

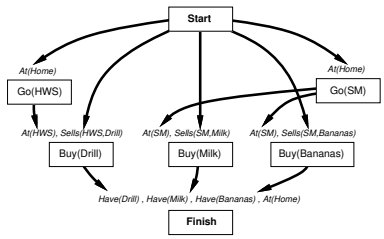
Dół: Plan zapewniający dodatkowo warunki wstępne $Sells$ w operatorach Buy.



Plan zapewniający warunki wstępne At w operatorach Buy:

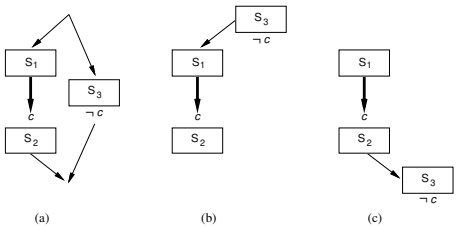


Plan zapewniający warunki wstępne At w operatorach $Go(HWS)$ i $Go(SM)$ (potrzebny nawrót):



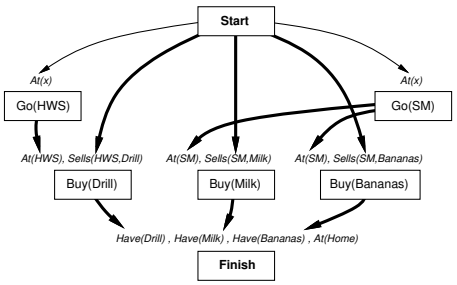
W jaki sposób system planujący potrafi stwierdzić, że konieczny jest nawrót?

Połączenia przyczynowe są chronione: system zapewnia, że **groźby (threats)**, czyli operatory, które mogą zniszczyć zapewniony wcześniej warunek, są umieszczane przed (promocja) lub po (degradacja) operatorze zapewniającym ten warunek.



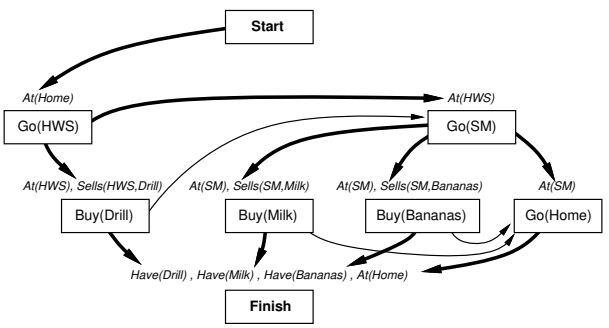
W naszym przypadku ani promocja, ani degradacja nie rozwiązują problemu. Dlatego system planujący wykonuje nawrót.

Po nawrocie:

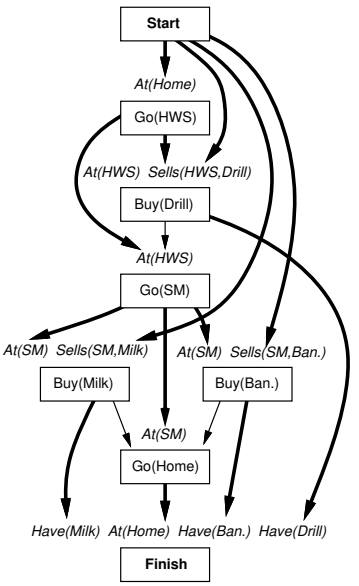


Zapewniamy warunek wstępny $At(x)$ operatora $Go(SM)$, dodając nowe połączenie postaci $Go(HWS) \stackrel{At(HWS)}{\rightarrow} Go(SM)$. Niestety krok ten wprowadza nową groźbę: agent może pójść ze sklepu z narzędziami do supermarketu, nie kupując przedtem wiertarki. Groźbę tę łatwo usunać, używając metody degradacji. Musimy jeszcze osiągnąć warunek wstępny $At(Home)$ operatora $Finish$. Wystarczy dodać operator $Go(Home)$ i zapewnić, że wystąpi on po operatorach $Buy(Milk)$ i $Buy(Banana)$.

Rozwiązanie:



W czytelniejszej formie:



Algorytm POP
(Partial-Order Planning Algorithm)

function POP(*initial, goal, operators*) **returns** *plan*

  plan ← MAKE-MINIMAL-PLAN(*initial, goal*)
  loop do
    if SOLUTION?(*plan*) **then return** *plan*
    S_{need}, c ← SELECT-SUBGOAL(*plan*)
    CHOOSE-OPERATOR(*plan, operators*, S_{need}, c)
    RESOLVE-THREATS(*plan*)
  end

---

function SELECT-SUBGOAL(*plan*) **returns** S_{need}, c

  pick a plan step S_{need} from STEPS(*plan*)
    with a precondition c that has not been achieved
  return S_{need}, c

---

procedure CHOOSE-OPERATOR(*plan, operators*, S_{need}, c)

  choose a step S_{add} from *operators* or STEPS(*plan*) that has c as an effect
  if there is no such step **then fail**
  add the causal link $S_{add} \xrightarrow{c} S_{need}$ to LINKS(*plan*)
  add the ordering constraint $S_{add} \prec S_{need}$ to ORDERINGS(*plan*)
  if S_{add} is a newly added step from *operators* **then**
    add S_{add} to STEPS(*plan*)
    add *Start* $\prec S_{add} \prec$ *Finish* to ORDERINGS(*plan*)

---

procedure RESOLVE-THREATS(*plan*)

  for each S_{threat} that threatens a link $S_i \xrightarrow{c} S_j$ in LINKS(*plan*) **do**
    choose either
      Promotion: Add $S_{threat} \prec S_i$ to ORDERINGS(*plan*)
      Demotion: Add $S_j \prec S_{threat}$ to ORDERINGS(*plan*)
    if not CONSISTENT(*plan*) **then fail**
  end

Uwagi:

- Algorytm POP jest algorytmem niedeterministycznym: instrukcja **choose** wybiera jedną z dopuszczalnych możliwości. Jeśli wszystkie dopuszczalne wybory zostały już dokonane, algorytm kończy działanie, nie znajdując rozwiązania.

- Instrukcja **fail** oznacza nawrót do ostatnio wykonywanej instrukcji **choose**.

- Algorytm POP jest algorytmem regresywnym: zaczyna od formuły celu i w każdym kroku próbuje osiągnąć jeden z warunków wstępnych wprowadzonego wcześniej operatora.

- Wybór operatora S_{need} i warunku c w procedurze SELECT-SUBGOAL nie prowadzi nigdy do nawrotu (zauważmy, że procedura nie używa instrukcji **choose**). Powód: jeśli chcemy znaleźć rozwiązanie, musimy zapewnić wszystkie warunki wstępne wszystkich operatorów.

- Algorytm POP jest poprawny i pełny: każdy znaleziony plan jest rozwiązaniem problemu; jeśli istnieje rozwiązanie problemu, zostanie ono znalezione.

- Algorytm POP, w wersji podanej powyżej, nie jest do końca wyspecyfikowany: pomija ograniczenia zmiennych.

Częściowo ukonkretnione operatory

Czy procedura RESOLVE-THREATS powinna traktować operator, którego efektem jest $\neg At(x)$ jako groźbę dla warunku $At(Home)$? Jest to groźba **potencjalna**.

Możliwy sposób rozwiązania problemu gróźb potencjalnych:

Ignorujemy każdą groźbę potencjalną do momentu, kiedy staje się ona groźbą rzeczywistą. Tzn. procedura RESOLVE-THREATS nie rozważa operatora, którego efektem jest $\neg At(x)$ jako groźby dla $At(Home)$. Ale: jeśli ograniczenie zmiennych $x = Home$ zostanie dodane do planu, powyższa potencjalna groźba staje się groźbą rzeczywistą i musi zostać rozwiązana. (Przy użyciu metody promocji lub degradacji.)

Zmodyfikowane procedury
CHOOSE-OPERATOR i RESOLVE-THREATS

procedure CHOOSE-OPERATOR(*plan*, *operators*, S_{needs}, *c*)

choose a step S_{add} from *operators* or STEPS(*plan*) that has c_{add} as an effect
such that *u* = UNIFY(c, c_{add}, BINDINGS(*plan*))
if there is no such step
then fail
add *u* to BINDINGS(*plan*)
add $S_{add} \xrightarrow{c} S_{need}$ to LINKS(*plan*)
add $S_{add} \prec S_{need}$ to ORDERINGS(*plan*)
if S_{add} is a newly added step from *operators* **then**
add S_{add} to STEPS(*plan*)
add *Start* $\prec S_{add} \prec$ *Finish* to ORDERINGS(*plan*)

---

procedure RESOLVE-THREATS(*plan*)

for each $S_i \xrightarrow{c} S_j$ **in** LINKS(*plan*) **do**
for each S_{threat} **in** STEPS(*plan*) **do**
for each c' **in** EFFECT(S_{threat}) **do**
if SUBST(BINDINGS(*plan*), *c*) = SUBST(BINDINGS(*plan*), $\neg c'$) **then**
choose either
Promotion: Add $S_{threat} \prec S_i$ to ORDERINGS(*plan*)
Demotion: Add $S_j \prec S_{threat}$ to ORDERINGS(*plan*)
if not CONSISTENT(*plan*)
then fail
end
end
end

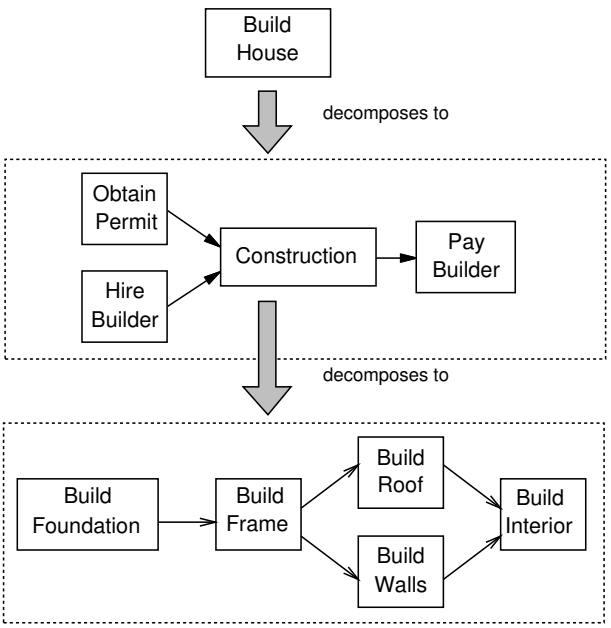
Planowanie hierarchiczne (Hierarchical planning)

Planowanie odpowiada konstruowaniu programu. Planowanie hierarchiczne odpowiada konstruowaniu programu metodą zstępującą.

Plan [$Go(Supermarket)$, $Buy(Milk)$, $Go(home)$] jest dobry dla człowieka, ale nie dla robota. Z drugiej strony, posługując się algorytmem POP, nie jesteśmy w stanie skonstruować planu [$Forward(1m), TurnLeft(90deg), \ldots$].

Praktyczne systemy planujące oparte są na metodzie **hierarchicznej dekompozycji**: Startujemy, posługując się operatorami wysokiego poziomu (operatory abstrakcyjne). W kolejnych krokach każdy abstrakcyjny operator zostaje zastąpiony swoją **implementacją**, tzn. planem, który go realizuje i który używa operatorów niższego poziomu. Kontynuujemy ten proces tak długo, aż otrzymany plan zawiera tylko operatory **podstawowe (prymitywne)**.

Hierarchiczna dekompozycja:



Rozszerzenie języka

Dzielimy zbiór dopuszczalnych operatorów na operatory podstawowe i abstrakcyjne.

Dodajemy zbiór **metod dekompozycyjnych**. Każda metoda jest wyrażeniem postaci $Decompose(o, p)$, gdzie *o* jest operatorem abstrakcyjnym, natomiast *p* jest planem.

$Decompose(Construction,$
$Plan($STEPS: $\{S_1 : Build(Foundation),$
$S_2 : Build(Frame), S_3 : Build(Roof),$
$S_4 : Build(Walls), S_5 : Build(Interior)\}$
ORDERINGS: $\{S_1 \prec S_2 \prec S_3 \prec S_5,$
$S_2 \prec S_4 \prec S_5\},$
BINDINGS: $\{\},$
LINKS: $\{S_1 \xrightarrow{Foundation} S_2, S_2 \xrightarrow{Frame} S_3,$
$S_2 \xrightarrow{Frame} S_4, S_3 \xrightarrow{Roof} S_5, S_4 \xrightarrow{Walls} S_5\})).$

Plan *p* poprawnie implementuje operator *o* jeśli jest pełnym i poprawnym rozwiązaniem problemu znalezienia planu z warunkami wstępnymi i formułą celu operatora *o*. Dokładniej:

- *p* jest poprawny, tzn. zarówno zbiór węzłów porządkujących, jak i zbiór ograniczeń zmiennych są niesprzeczne.
- Każdy efekt operatora *o* musi być zapewniony przez pewien operator planu *p* (i nie może być "unieważniony" przez inny, późniejszy operator planu *p*).
- Każdy warunek wstępny każdego operatora planu *p* musi być osiągnięty w wyniku realizacji pewnego operatora planu *p* lub być jednym z warunków wstępnych operatora *o*.

Algorytm HD-POP (Hierarchical decomposition POP)

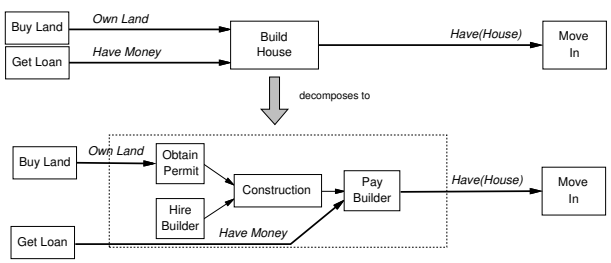
```
function HD-POP( plan, operators, methods) returns plan
  inputs: plan, an abstract plan with start and goal steps (and possibly other steps)

  loop do
    if SOLUTION?( plan) then return plan
    S_need, c ← SELECT-SUB-GOAL( plan)
    CHOOSE-OPERATOR( plan, operators, S_need, c)
    S_nonprim ← SELECT-NONPRIMITIVE( plan)
    CHOOSE-DECOMPOSITION( plan, methods, S_nonprim)
    RESOLVE-THREATS( plan)
  end
```

Procedury SELECT-SUBGOAL, CHOOSE-OPERATOR i RESOLVE-THREATS zdefiniowane są jak w przypadku algorytmu POP. Procedura SOLUTION? musi dodatkowo sprawdzić, czy każdy operator planu jest operatorem podstawowym. Funkcja SELECT-NONPRIMITIVE wybiera dowolny operator występujący w planie spośród operatorów abstrakcyjnych. Procedura CHOOSE-DECOMPOSITION wybiera metodę dekompozycyjną dla operatora $S_{nonprim}$ i odpowiednio modyfikuje aktualny plan.

Załóżmy, że wybrana została metoda $Decompose(S_{nonprim}, p)$. Aktualny plan, P, zostaje zmodyfikowany następująco.

- Do zbioru operatorów planu P dodajemy wszystkie operatory planu *p*; usuwamy operator $S_{nonprim}$.
- Do zbioru ograniczeń zmiennych planu P dodajemy ograniczenia zmiennych planu *p*. Jeśli operacja ta prowadzi do sprzeczności, robimy nawrót.
- Zastępujemy każdy węzeł porządkujący postaci $S_a \prec S_{nonprim}$ zbiorem węzłów $\{S_a \prec S_{m_1}, \ldots, S_a \prec S_{m_i}\}$, gdzie $S_{m_1}, \ldots, S_{m_i}$ są maksymalnymi (względem relacji $\prec$) operatorami planu *p*. Każdy węzeł porządkujący postaci $S_{nonprim} \prec S_a$ zastępujemy zbiorem $\{S_{m_1} \prec S_a, \ldots, S_{m_j} \prec S_a\}$, gdzie $S_{m_1}, \ldots, S_{m_j}$ są minimalnymi operatorami planu *p*. Następnie wywołujemy procedurę RESOLVE-THREADS w celu ewentualnego dodania nowych węzłów.

- Każde powiązanie przyczynowe występujące w P, postaci $S_i \xrightarrow{c} S_{nonprim}$, zastępujemy zbiorem powiązań postaci $S_i \xrightarrow{c} S_m$, gdzie S_m jest operatorem w *p* z warunkiem wstępnym *c* i *c* nie jest warunkiem wstępnym żadnego wcześniejszego operatora w *p*. Podobnie, każde powiązanie postaci $S_{nonprim} \xrightarrow{c} S_j$, występujące w P, zastępujemy zbiorem powiązań postaci $S_m \xrightarrow{c} S_j$, gdzie S_m jest operatorem w *p* z efektem *c* i *c* nie jest efektem żadnego późniejszego operatora w *p*.



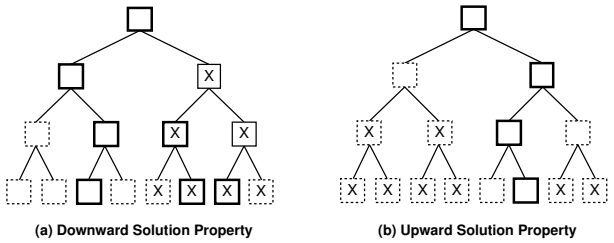
Rozwiązanie abstrakcyjne: pełny i poprawny plan zawierający przynajmniej jeden operator abstrakcyjny.

Rozwiązanie: pełny i poprawny plan zawierający wyłącznie operatory podstawowe.

Plan P nazywamy **abstrakcją** planu *p*, jeśli *p* można otrzymać z P stosując metodę dekompozycji.

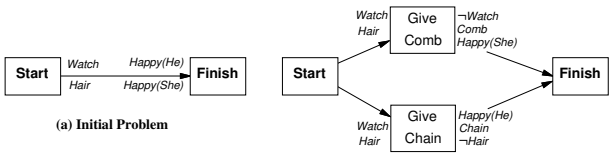
Własność zstępujących rozwiązań: Każde rozwiązanie abstrakcyjne jest abstrakcją pewnego rozwiązania.

Własność wstępujących rozwiązań: Żaden sprzeczny plan abstrakcyjny nie jest abstrakcją rozwiązania.



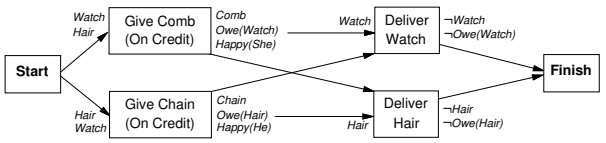
(a) Downward Solution Property (b) Upward Solution Property

Przykład, że własność wstępujących rozwiązań nie zawsze zachodzi:



(a) Initial Problem

(b) Abstract Inconsistent Plan



(c) Decomposition of (b) into a Consistent Solution

Niech $M = Decompose(o, p)$ będzie metodą dekompozycyjną. Operator *o'* występujący w *p* nazywamy operatorem **głównym** metody M, jeśli każdy warunek wstępny operatora *o* jest warunkiem wstępnym operatora *o'* i każdy warunek końcowy operatora *o* jest warunkiem końcowym operatora *o'*.

Jeśli każda metoda dekompozycyjna posiada dokładnie jeden operator główny, to zagwarantowana jest własność wstępujących rozwiązań.

Planowanie warunkowe (Conditional planning)

Op(ACTION: $Remove(x)$,
PRECOND: $On(x)$,
EFFECT: $Off(x) \wedge ClearHub(x) \wedge \neg On(x)$).

Op(ACTION: $PutOn(x)$,
PRECOND: $Off(x) \wedge ClearHub(x)$,
EFFECT: $On(x) \wedge \neg ClearHub(x) \wedge \neg Off(x)$).

Op(ACTION: $Inflate(x)$,
PRECOND: $Intact(x) \wedge Flat(x)$,
EFFECT: $Inflated(x) \wedge \neg Flat(x)$).

Stan początkowy:

$On(Tire1) \wedge Flat(Tire1) \wedge Inflated(Spare)$.

Formuła celu:

$On(x) \wedge Inflated(x)$.

Systemy planujące omawiane wcześniej skonstruują plan $[Remove(Tire1), PutOn(Spare)]$.

Lepszy plan:

if $Intact(Tire1)[Inflate(Tire1)]$
else $[Remove(Tire1, PutOn(Spare)]$.

Uwaga: Realizując plan warunkowy, agent musi obserwować świat, w którym działa.

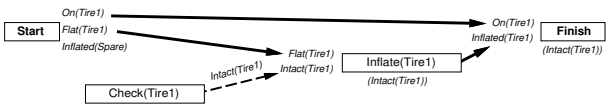
Op(ACTION: $CheckTire(x)$,
PRECOND: $Tire(x)$,
EFFECT: $KnowsWhether(\text{“}Intact(x)\text{”})$.

Konstrukcja planu warunkowego zakłada, że z każdym operatorem planu związany jest **kontekst** – zbiór warunków, które muszą być spełnione, aby operator był wykonalny.

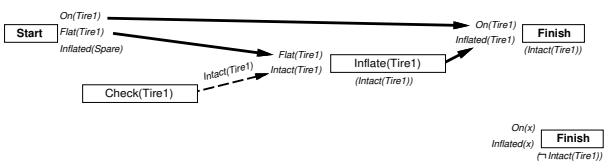
Zaczynamy od planu początkowego: (Z operatorem $Finish$ związany jest kontekst $True$.)

Warunek $On(x)$ zapewniamy dodając połączenie biegnące od operatora $Start$ (przy podstawieniu $\{x/Tire1\}$). Warunek $Inflated(Tire1)$ gwarantuje operator $Inflate(Tire1)$:

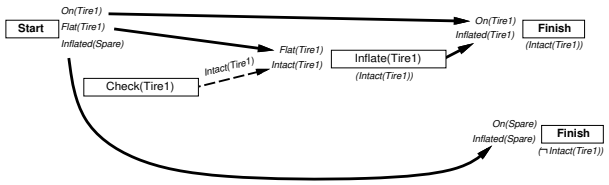
Nie mamy operatora, który zapewnia warunek $Intact(Tire1)$. Wprowadzamy **połączenie warunkowe**:



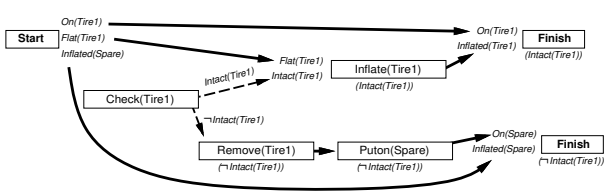
Musimy teraz skonstruować plan dla przypadku, kiedy warunek $Intact(Tire)$ nie zachodzi. Duplikujemy operator $Finish$, dołączając kontekst $\neg Intact(Tire1)$.



Warunku $Inflated(x)$ nie możemy zapewnić wprowadzając operator $Inflate(Tire1)$: warunek wstępny tego operatora, $Intact(Tire1)$, jest sprzeczny z kontekstem. Warunek $Inflated(x)$ zapewniamy dodając połączenie biegnące od operatora $Start$, przy podstawieniu $\{x/Spare\}$.



Dodajemy operatory $PutOn(Spare)$ oraz $Remove(Tire1)$, a następnie rozwiązujemy groźbę, jaką dla połączenia $Start \overset{On(Tire1)}{\longrightarrow} Finish$ stanowi operator $Remove(Tire1)$. Ten sposób rozwiązywania gróźb nazywa się metodą **warunkowania**. Ostateczny plan:



Algorytm CPOP (Conditional POP)

function CPOP(*initial, goals, operators*) **returns** *plan*

plan ← MAKE-PLAN(*initial, goals*)
loop do
  Termination:
    if there are no unsatisfied preconditions
      and the contexts of the finish steps are exhaustive
      then return *plan*
  Alternative context generation:
    if the plans for existing finish steps are complete and have contexts $C_1 \ldots C_n$ **then**
      add a new finish step with a context $\neg(C_1 \vee \ldots \vee C_n)$
      this becomes the *current context*
  Subgoal selection and addition:
    find a plan step S_{need} with an open precondition c
  Action selection:
    choose a step S_{add} from *operators* or STEPS(*plan*) that adds c or
        knowledge of c and has a context compatible with the current context
    if there is no such step
      then fail
    add $S_{add} \xrightarrow{c} S_{need}$ to LINKS(*plan*)
    add $S_{add} \prec S_{need}$ to ORDERINGS(*plan*)
    if S_{add} is a newly added step **then**
      add S_{add} to STEPS(*plan*)
      add $Start \prec S_{add} \prec Finish$ to ORDERINGS(*plan*)
  Threat resolution:
    for each step S_{threat} that potentially threatens any causal link $S_i \xrightarrow{c} S_j$
        with a compatible context **do**
      choose one of
        Promotion: Add $S_{threat} \prec S_i$ to ORDERINGS(*plan*)
        Demotion: Add $S_j \prec S_{threat}$ to ORDERINGS(*plan*)
        Conditioning:
          find a conditional step S_{cond} possibly before both S_{threat} and S_j, where
            1. the context of S_{cond} is compatible with the contexts of S_{threat} and S_j;
            2. the step has outcomes consistent with S_{threat} and S_j, respectively
          add conditioning links for the outcomes from S_{cond} to S_{threat} and S_j
          augment and propagate the contexts of S_{threat} and S_j
      if no choice is consistent
          then fail
    end
end

Planowanie w zmieniającym się świecie

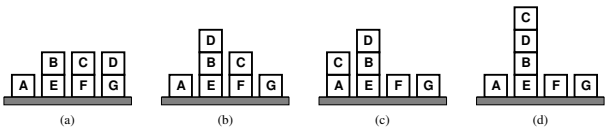
$Move(x, y)$ – przenieś klocek x na klocek y:

Op(ACTION: $Move(x, y)$,
PRECOND: $Clear(x) \wedge Clear(y) \wedge On(x, z)$,
EFFECT: $On(x, y) \wedge Clear(z) \wedge \neg Clear(y) \wedge \neg On(x, z)$).

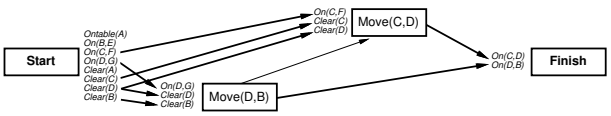
Formuła celu:

$On(C, D) \wedge On(D, B)$.

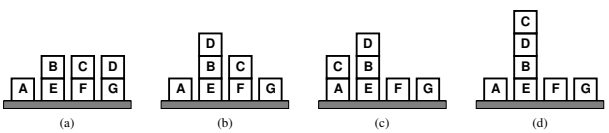
Stan początkowy: poniżej (a).



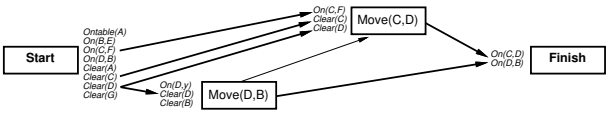
Agent A konstruuje plan:



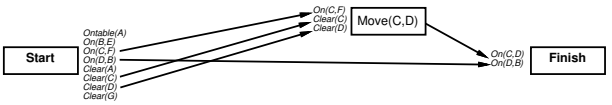
W trakcie konstrukcji powyższego planu, agent B przenosi klocek D na B. Aktualny stan świata: poniżej (b).



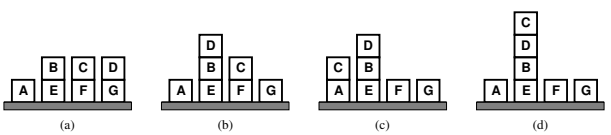
Agent A rekonstruuje plan. Rozpoczyna od usunięcia połączeń $Start \xrightarrow{On(D,G)} Move(D, B)$ i $Start \xrightarrow{Clear(B)} Move(D, B)$:



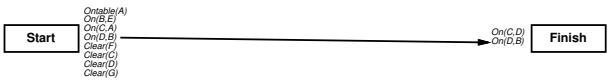
Operator $Move(D, B)$ jest teraz redundantny, ponieważ warunek $On(D, B)$ zagwarantowany jest w stanie początkowym. Plan końcowy:



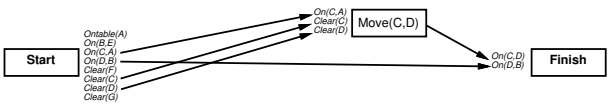
Agent A rozpoczyna realizację planu. Niestety, przez pomyłkę wykonuje akcję $Move(C, A)$. Aktualny stan świata: poniżej (c).



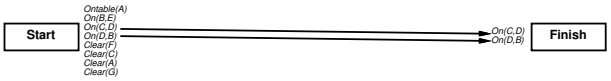
Agent A ponownie rekonstruuje plan. Rozpoczyna od planu:



Plan końcowy:



Agent A poprawnie realizuje plan:



Szkic algorytmu

function SITUATED-PLANNING-AGENT(*percept*) **returns** an *action*
static: *KB*, a knowledge base (includes action descriptions)
p, a plan, initially *NoPlan*
t, a counter, initially 0, indicating time
G, a goal

TELL(*KB*, MAKE-PERCEPT-SENTENCE(*percept*, *t*))
current ← STATE-DESCRIPTION(*KB*, *t*)
EFFECTS(START(*p*)) ← *current*
if *p* = *NoPlan* **then**
G ← ASK(*KB*, MAKE-GOAL-QUERY(*t*))
p ← MAKE-PLAN(*current*, *G*, *KB*)
action ← *NoOp* (the default)

Termination:
if there are no open preconditions and *p* has no steps other than START and FINISH **then**
p ← *NoPlan* and skip remaining steps

Resolving standard flaws:
resolve any open condition by adding a causal link from any existing
possibly prior step or a new step
resolve potential threats by promotion or demotion

Remove unsupported causal links:
if there is a causal link START $\xrightarrow{c}$ *S* protecting a proposition *c*
that no longer holds in START **then**
remove the link and any associated bindings

Extend causal links back to earliest possible step:
if there is a causal link $S_j \xrightarrow{c} S_k$ such that
another step S_i exists with $S_i \prec S_j$ and the link $S_i \xrightarrow{c} S_k$ is safe **then**
replace $S_j \xrightarrow{c} S_k$ with $S_i \xrightarrow{c} S_k$

Remove redundant actions:
remove any step *S* that supplies no causal links

Execute actions when ready for execution:
if a step *S* in the plan other than FINISH satisfies the following:
(a) all preconditions satisfied by START;
(b) no other steps necessarily between START and *S*; and
(c) *S* does not threaten any causal link in *p* **then**
add ordering constraints to force all other steps after *S*
remove *S* from *p*, and all causal links to and from *S*
action ← the action in *S*

TELL(*KB*, MAKE-ACTION-SENTENCE(*action*, *t*))
t ← *t* + 1
return *action*