Sztuczna Inteligencja (SI)

Wykład dla III roku ZSI

Na podstawie: Stuart Russel & Peter Norvig *Artificial Intelligence: A Modern Approach*, Prentice Hall, 1995.

http://www.cs.berkeley.edu/~russell/aima

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 konstruowania systemów działających racjonalnie

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

Racjonalnie działający obiekt = AGENT

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

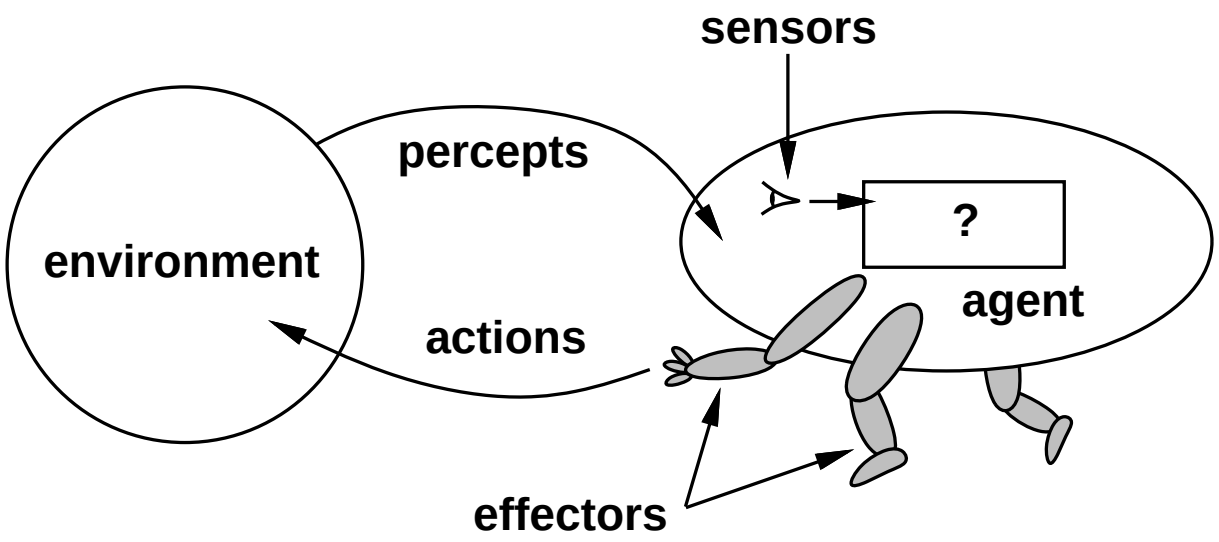
   (a) DEEP BLUE – program szachowy o współczynniku FIDE 2700.

   (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 ≠ NIEOMYLNOŚĆ !

Racjonalność zależy od 4 elementów:

- Miara skuteczności (ang. performance measure): definiuje stopień skuteczności agenta - jak często agent osiąga cel.
- 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

$$\boxed{\text{AGENT} = \text{ARCHITEKTURA} + \text{PROGRAM}}$$

SI jako program agenta

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
```

Każde wywołanie powoduje aktualizację pamięci agenta o nową obserwację, wybór najlepszej akcji oraz aktualizację pamięci o efekt wybranej akcji. Pamięć zostaje zachowana do następnego wywołania.

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 $\leftarrow$ LOOKUP(*percepts*, *table*)
  return *action*

Wady takiego podejścia:

1. Rozmiar tablicy (na ogół bardzo duży).
2. Czas tworzenia.
3. Brak autonomii. Niespodziewana zmiana w środowisku może spowodować całkowite zagubienie się agenta lub konieczność budowania tablicy od nowa.
4. W przypadku agenta uczącego się, nauka poprawnych wartości dla wszystkich wierszy tablicy będzie trwać w nieskończoność.

Rodzaje agentów

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

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

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

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

Rozwiązywanie problemów

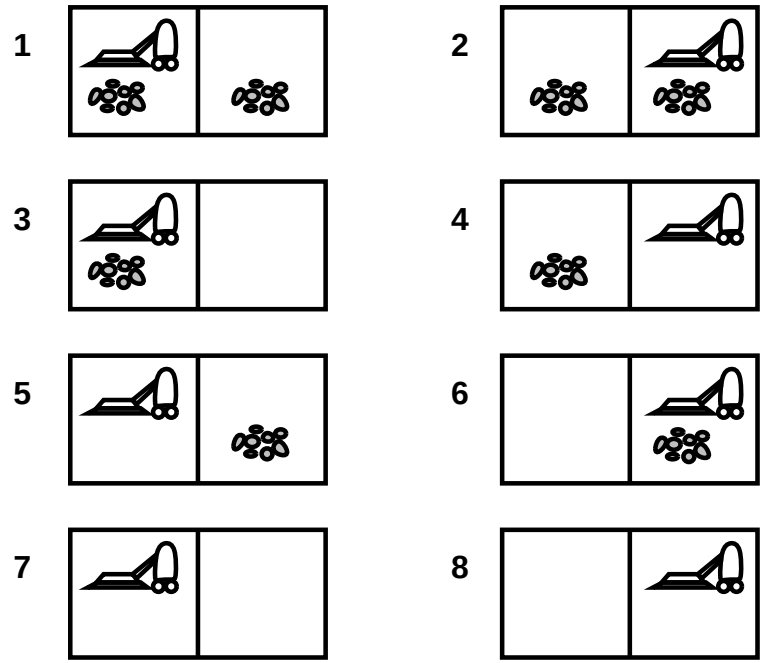
Podstawowe fazy:

1. Sformułowanie celu - zbiór stanów świata spełniających cel.
2. Sformułowanie problemu - wybór rozważanych akcji i stanów.
3. Znalezienie rozwiązania - ciąg akcji do wykonania.
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

Przykład: 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 $Right, Suck$ 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 / niepełną informacją o stanie początkowym.**

   - Stan początkowy / zbiór stanów początkowych,
   - zbiór akcji (operatorów),
   - funkcja celu (pozwala stwierdzić, czy dany stan jest stanem końcowym),
   - funkcja kosztu (mierzy efektywność rozwiązania).

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

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:

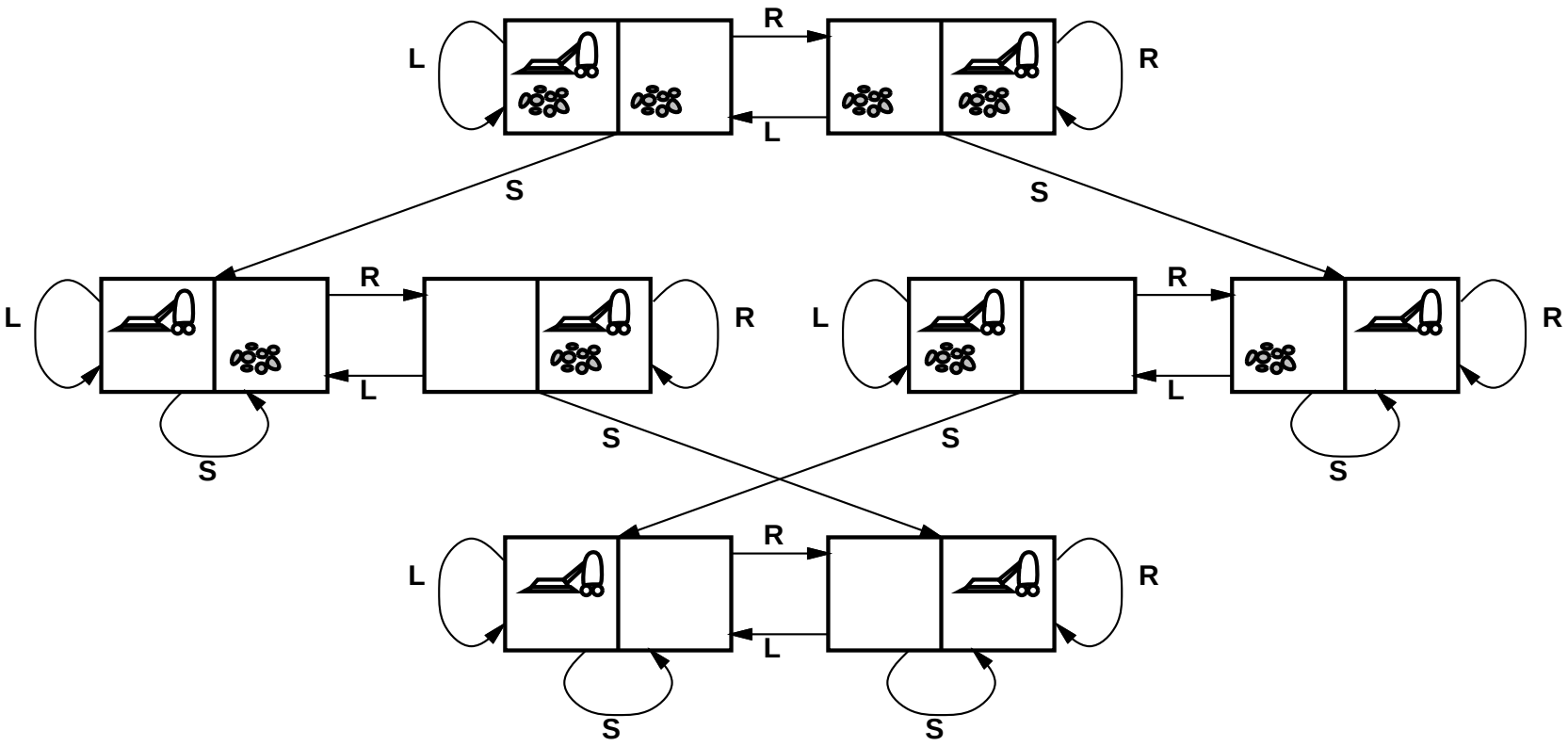
Stany: Jeden z ośmiu stanów występujących na rysunku poniżej.

Akcje: $Left, Right, Suck$.

Cel: Oba pokoje czyste.

Koszt: Liczba wykonanych akcji.

Przestrzeń stanów:

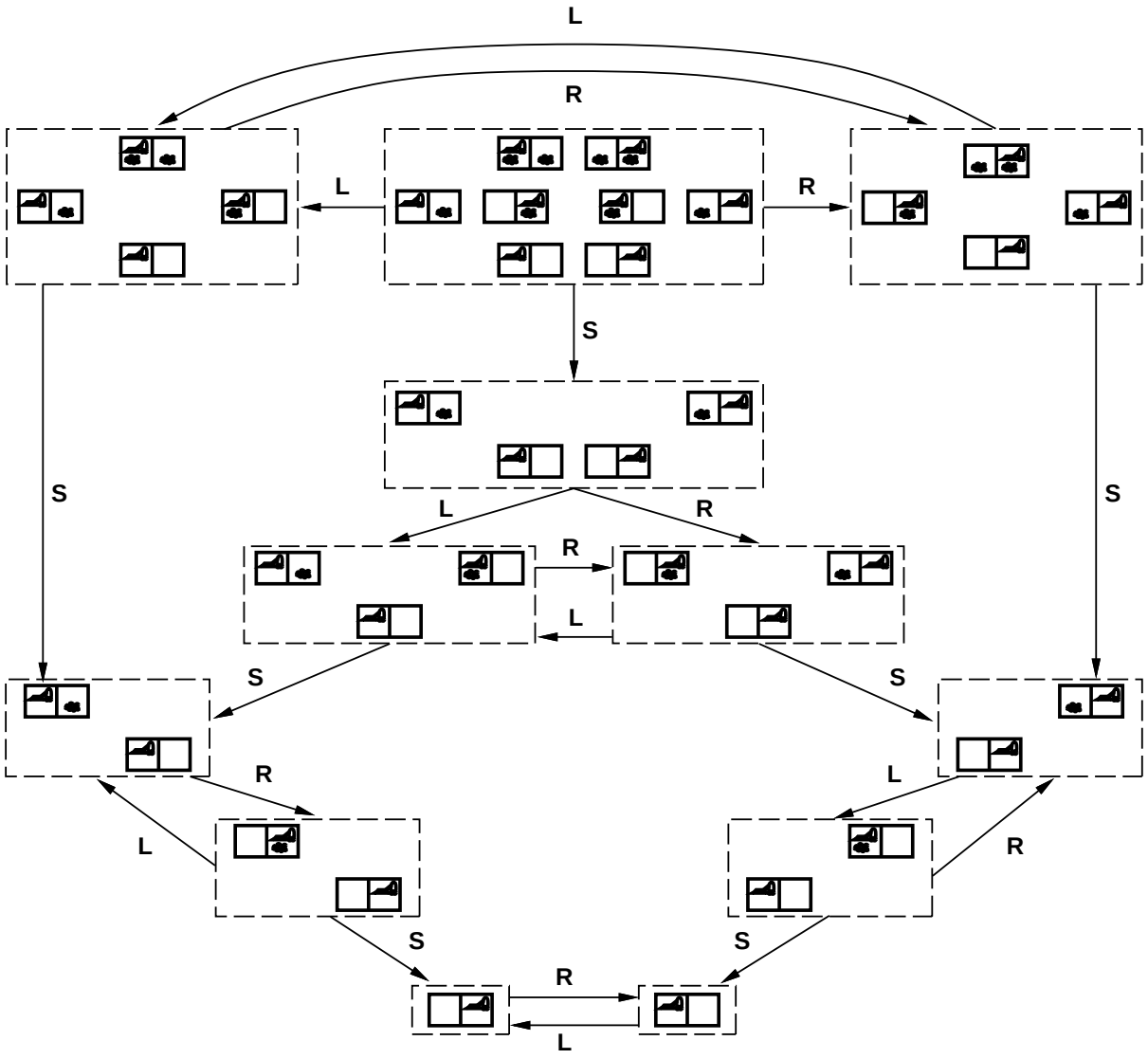


2. Niepełna informacja o stanie początkowym:

Stany: Podzbiór stanów $1 - 8$.

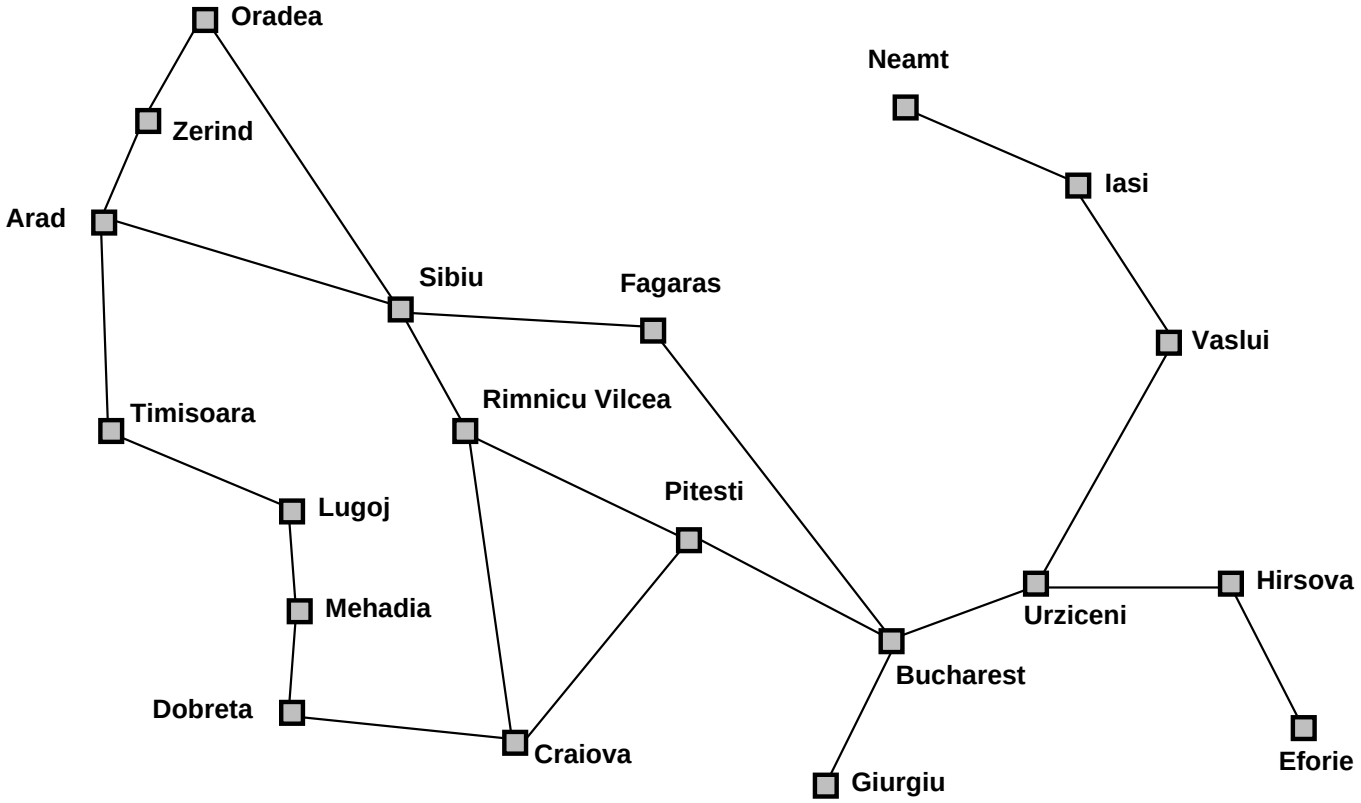
Akcje, cel, koszt: jak wyżej.

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.

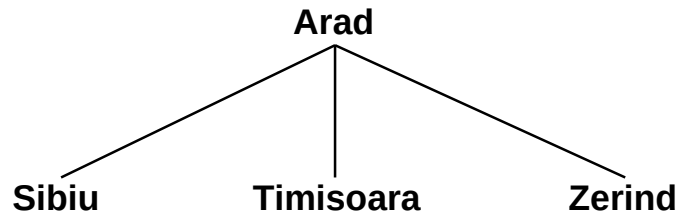
Stan początkowy: Arad.

Koszt: Liczba miast przez które jedziemy.

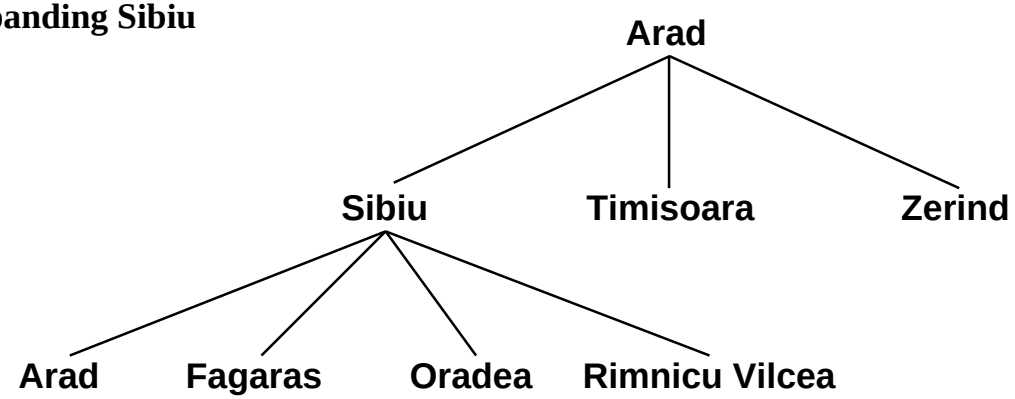
Przeszukiwanie przestrzeni stanów sprowadza się do budowy *drzewa przeszukiwań*. Korzeniem drzewa jest stan początkowy.

(a) The initial state

(b) After expanding Arad



(c) After expanding Sibiu



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.

Strategie przeszukiwania

Kryteria oceny strategii:

- **Pełność** – czy zawsze znajduje rozwiązanie, o ile istnieje.
- **Złożoność czasowa**.
- **Złożoność pamięciowa**.
- **Optymalność** – czy znalezione rozwiązanie jest rozwiązaniem optymalnym (w sensie funkcji kosztu).

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 $d + 1$.

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

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

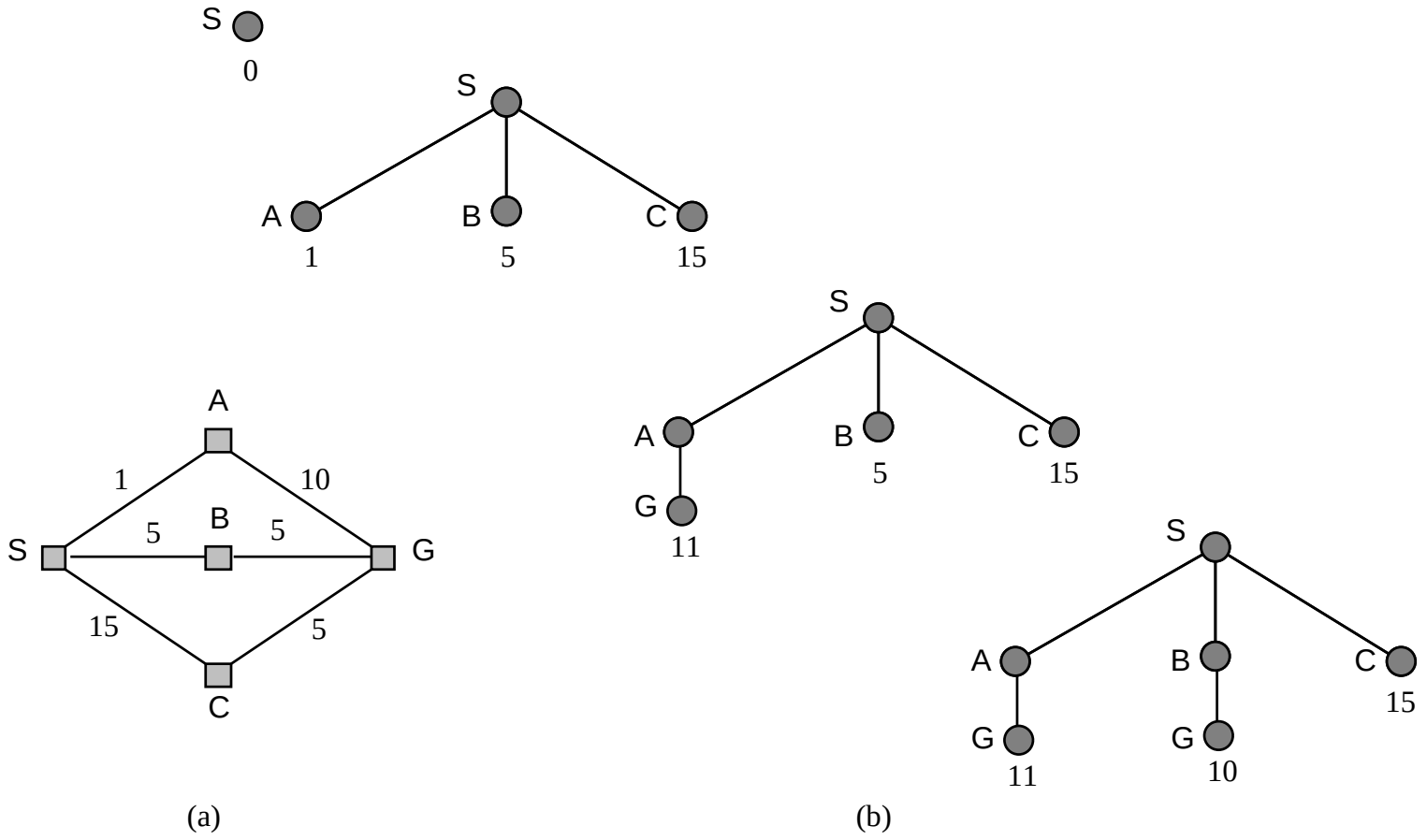
Dla $b = 10$

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.

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* $\leftarrow$ 0 **to** ∞ **do**
    if DEPTH-LIMITED-SEARCH(*problem*, *depth*) succeeds **then return** its result
  end
  return failure

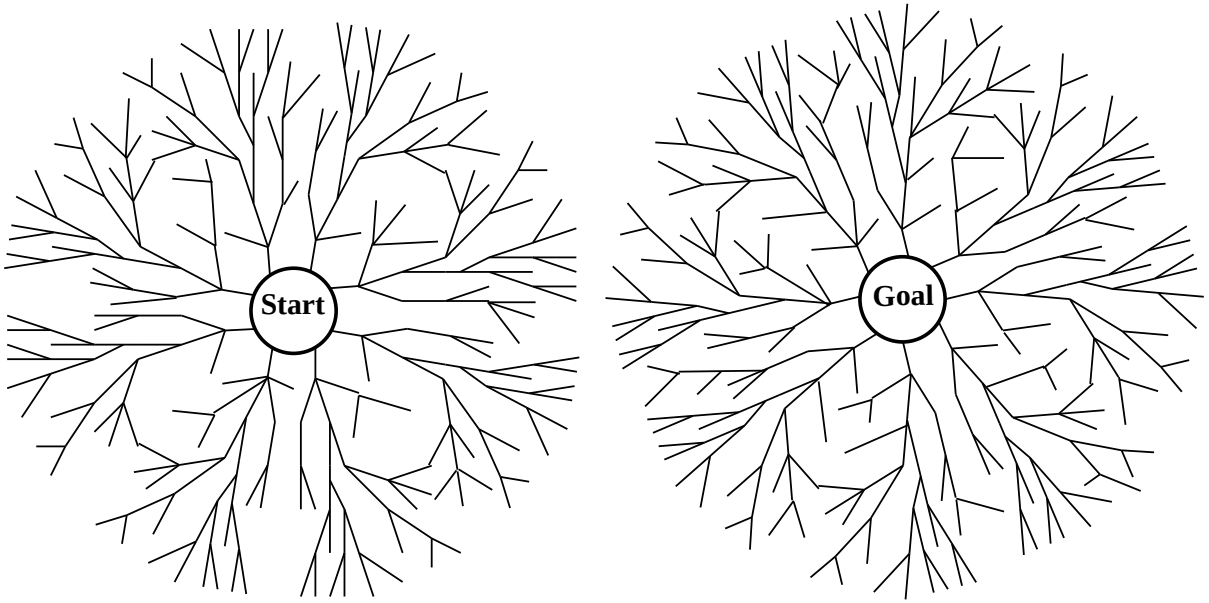
b – współczynnik rozgałęzienia.
d – długość najkrótszego rozwiązania.

- 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*)

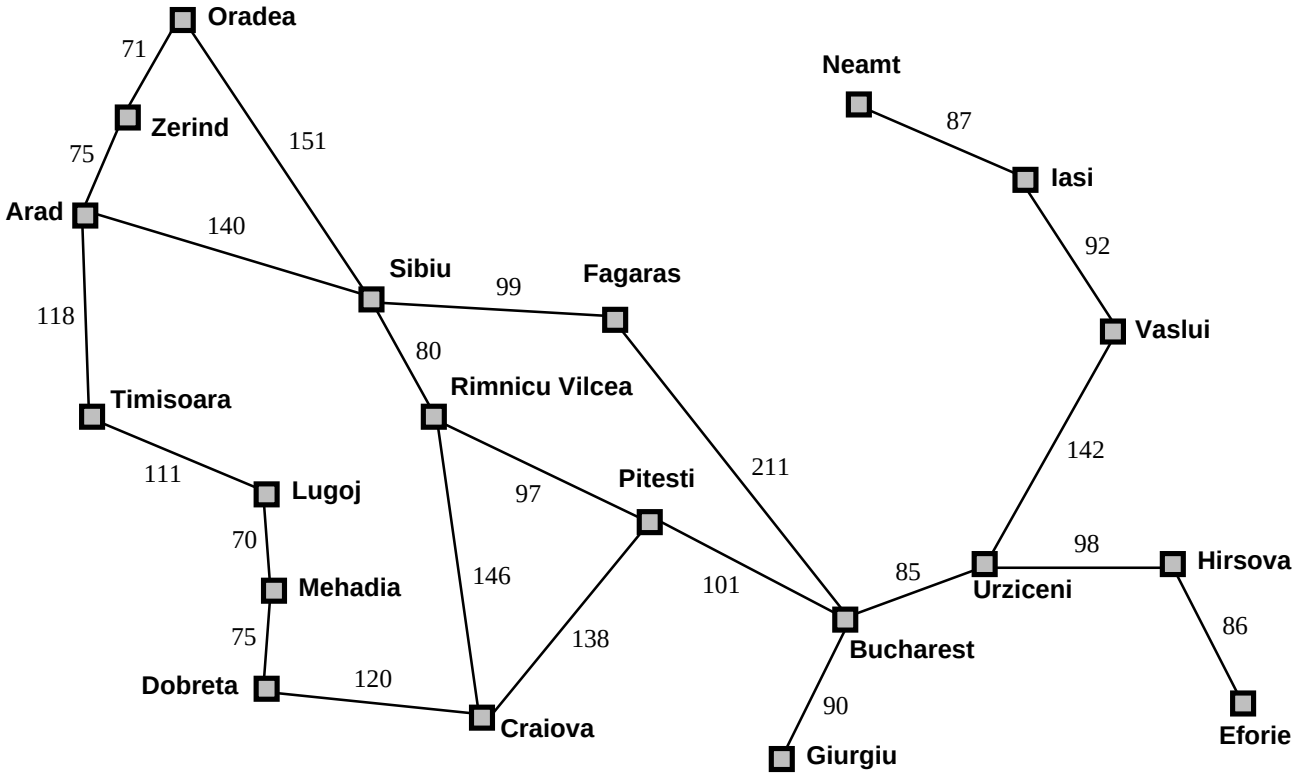
Problem: Jak określić funkcję EVAL-FN?

Przeszukiwanie zachłanne (greedy search)

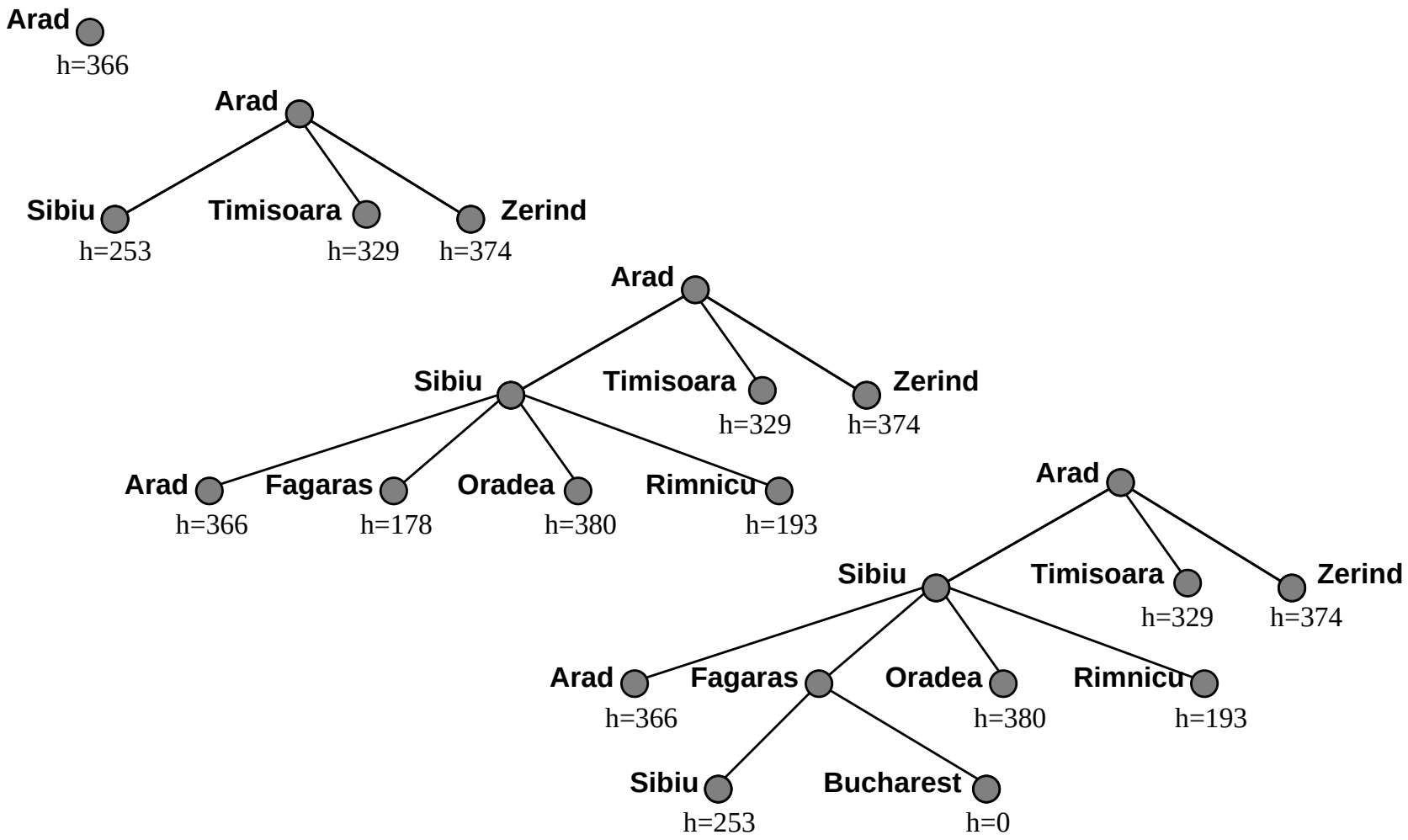
Stany porządkujemy wprowadzając *funkcję heurystyczną* h:

$h(n)$ = 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


Arad
h=366
Arad
Sibiu
Timisoara
Zerind
h=253
h=329
h=374
Arad
Sibiu
Timisoara
Zerind
h=329
h=374
Arad
Fagaras
Oradea
Rimnicu
h=366
h=178
h=380
h=193
Arad
Sibiu
Timisoara
Zerind
h=329
h=374
Arad
Fagaras
Oradea
Rimnicu
h=366
h=380
h=193
Sibiu
Bucharest
h=253
h=0

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)

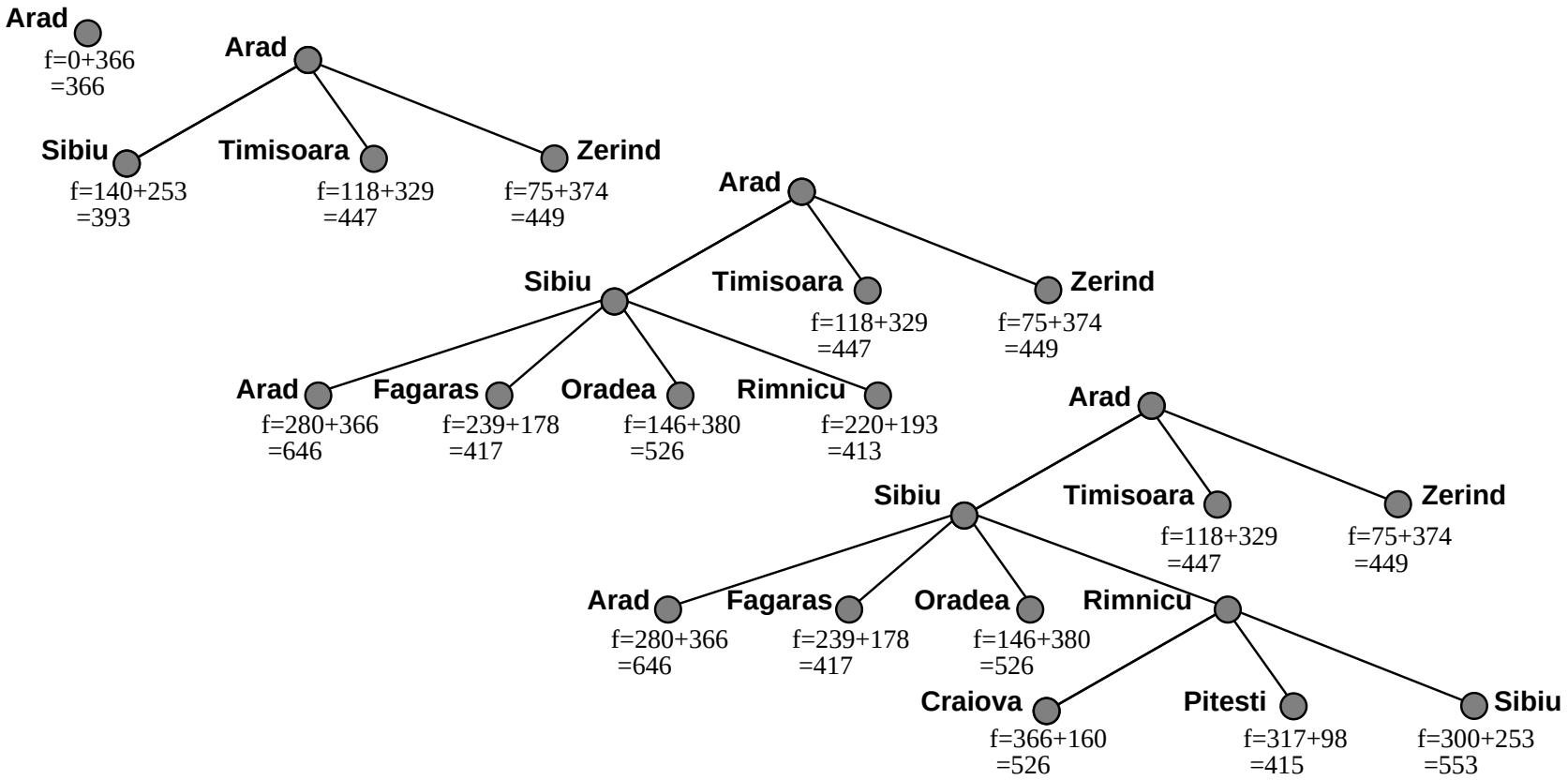
Łączy przeszukiwanie zachłanne i strategię jednolitego kosztu. Z każdym skonstruowanym dotąd stanem n wiążemy wartość $g(n)$, definiującą rzeczywisty koszt ścieżki od stanu początkowego do stanu n. Jak poprzednio wprowadzamy funkcję heurystyczną $h(n)$. Jako funkcji porządkującej stany używamy funkcji $f(n) = g(n)+h(n)$. 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.


Arad
f=0+366
=366
Arad
Sibiu
f=140+253
=393
Timisoara
f=118+329
=447
Zerind
f=75+374
=449
Arad
Sibiu
Timisoara
f=118+329
=447
Zerind
f=75+374
=449
Arad
f=280+366
=646
Fagaras
f=239+178
=417
Oradea
f=146+380
=526
Rimnicu
f=220+193
=413
Arad
Sibiu
Timisoara
f=118+329
=447
Zerind
f=75+374
=449
Arad
f=280+366
=646
Fagaras
f=239+178
=417
Oradea
f=146+380
=526
Rimnicu
Craiova
f=366+160
=526
Pitesti
f=317+98
=415
Sibiu
f=300+253
=553

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.

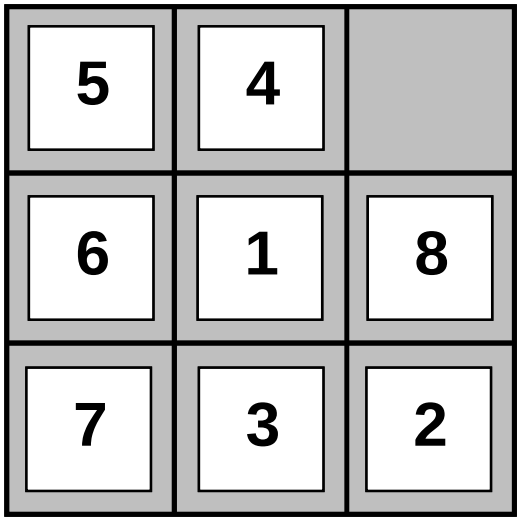
- Jeśli funkcja f nie jest monotoniczna, można zastąpić ją funkcją

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

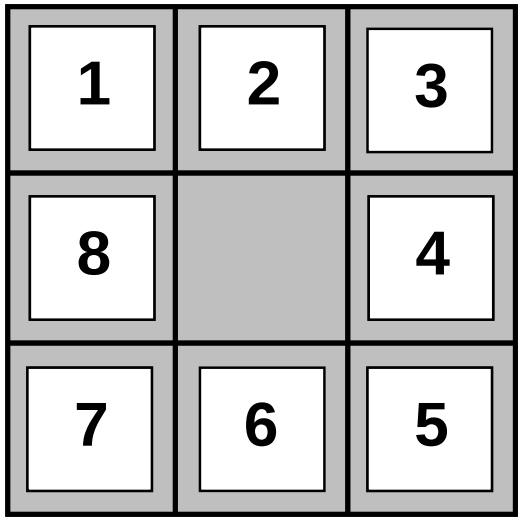
- Przeszukiwanie A^* jest znacznie efektywniejsze od przeszukiwania wg strategii jednolitego kosztu. Niestety jego koszt jest również wykładniczy.

Funkcje heurystyczne

Problem:



Start State



Goal State

- 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.

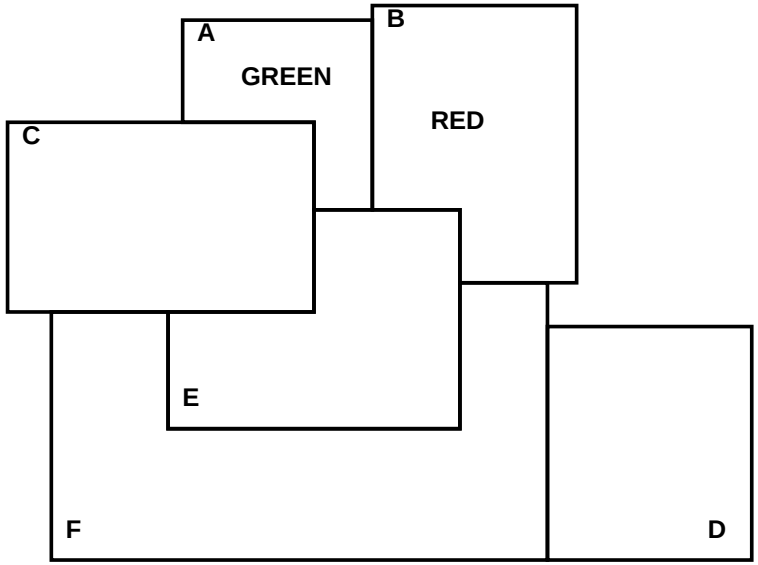
Jak znaleźć dobrą funkcję heurystyczną?

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)

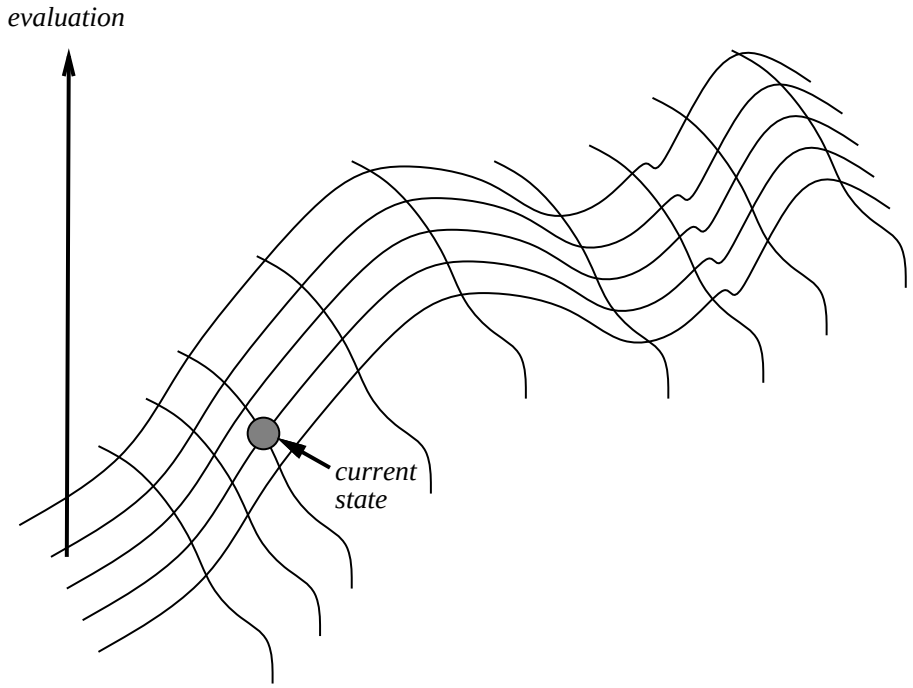
- Dotyczą problemów, dla których nieistotna jest droga uzyskania rozwiązania (np. problem 8 hetmanów).

- Idea: Startujemy od pewnego stanu (na ogół wybranego losowo), ktory modyfikujemy, aby 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)



Algorytm jest pętlą, w której kolejne ruchy polegają na wyborze stanu następnika według wzrastającej wartości funkcji VALUE. Jeśli jest więcej niż jeden najlepszy następnik do wyboru, wybór jest losowy.

```
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. W takim przypadku algorytm się zatrzymuje nie znajdując rozwiązania.

- Plateaux. Funkcja wartościująca jest płaska. Przeszukiwanie przebiega losowo.

- Grzbiety. Grzbiet może być stromo pochylony, więc osiągnięcie go będzie łatwe, ale potem postęp może być bardzo powolny, a nawet może oscylować z jednej strony na drugą.

W każdym z tych przypadków trzeba wystartować algorytm od nowa. Jeśli dopuszcza się dostatecznie dużo iteracji, algorytm znajduje optymalne rozwiązanie. Sukces zależy tu od kształtu "powierzchni" przestrzeni stanów.

Zastosowanie: GSAT, learning, trudne problemy praktyczne.

Symulowane wyżarzanie (Simulated annealing)

Idea: Zamiast rozpoczynać przeszukiwanie od nowego losowego punktu startowego w przypadku napotkania maksimum lokalnego pozwalamy, aby algorytm wykonał kilka kroków wstecz i ominął maksimum lokalne.

Zamiast wybierać najlepszy ruch (jak w hill-climbing), algorytm wybiera ruch losowy. Jeśli poprawia on sytuację, jest zawsze wykonywany. W przeciwnym przypadku wykonuje się ruch z prawdopodobieństwem mniejszym od 1. Prawdopodobieństwo zmniejsza się wykładniczo wraz z "pogarszaniem" się ruchu.

Pomysł algorytmu - analogia z procesem wyżarzania - stopniowego ochładzania płynu aż do jego zamarznięcia.

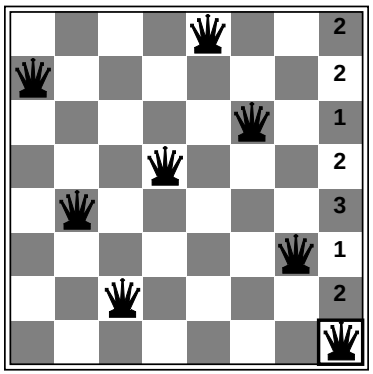
```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  static: current, a node
          next, a node
          T, a "temperature" controlling the probability of downward steps

  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)
```

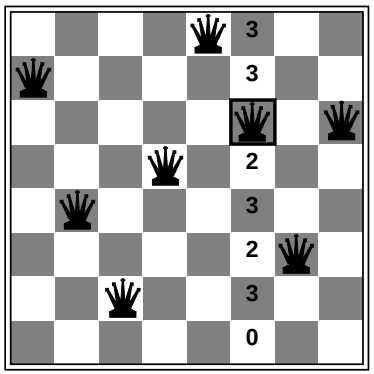
- Funkcja VALUE odpowiada całkowitej energii atomów materiału. T odpowiada temperaturze. Lista $schedule$ określa stopień obniżania się temperatury. Zmiany stanów odpowiadają losowym fluktuacjom materiału na skutek obniżania się temperatury.
- Algorytm okazał się bardzo skuteczny dla problemów optymalizacji.

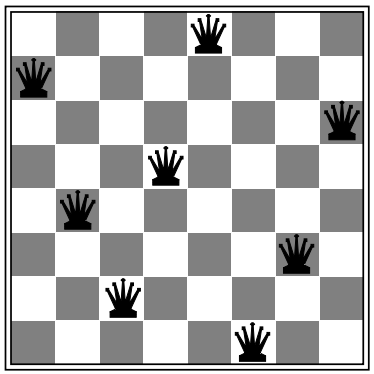
Heurystyka minimalizacji konfliktów dla problemów z warunkami ograniczającymi

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!

Poniżej: dwukrokowe rozwiązanie problemu 8 hetmanów:







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

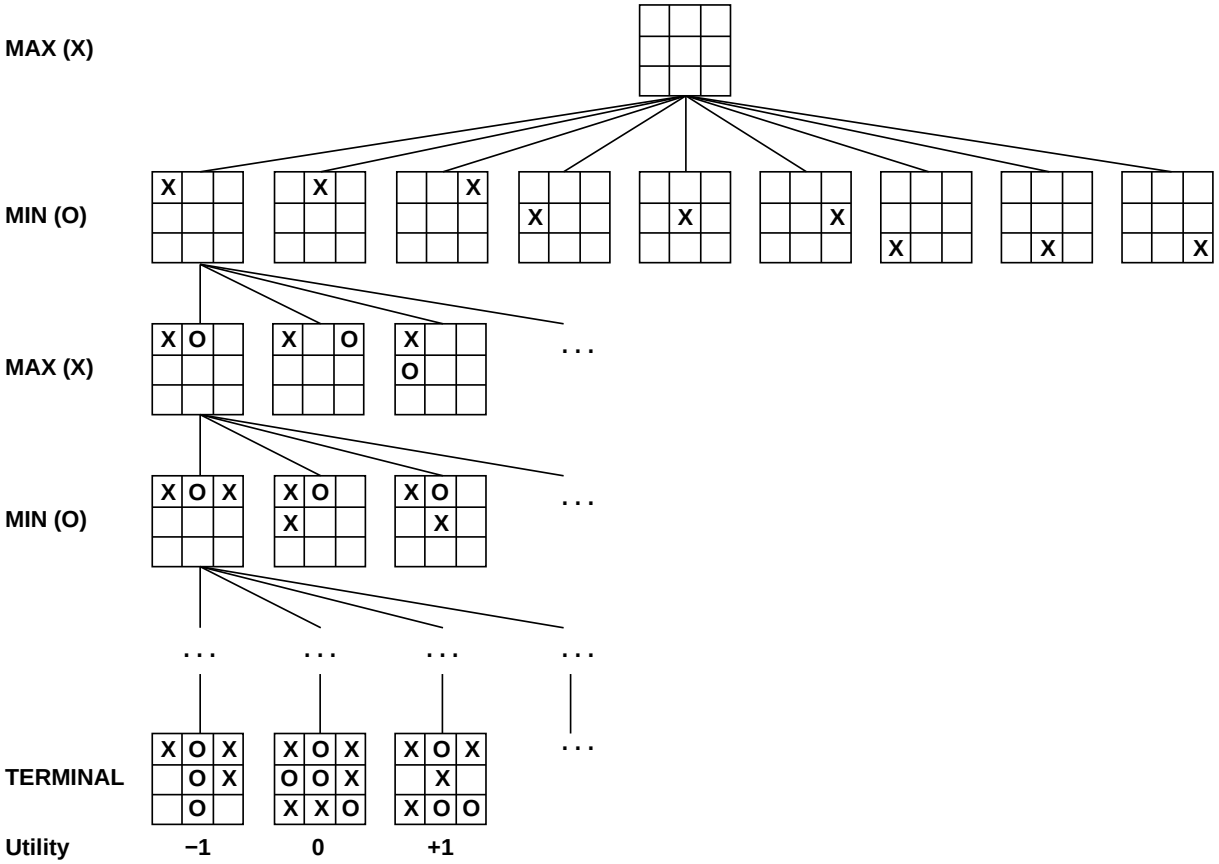
Grają MIN i MAX; MAX rozpoczyna.

Gra jako problem przeszukiwania:

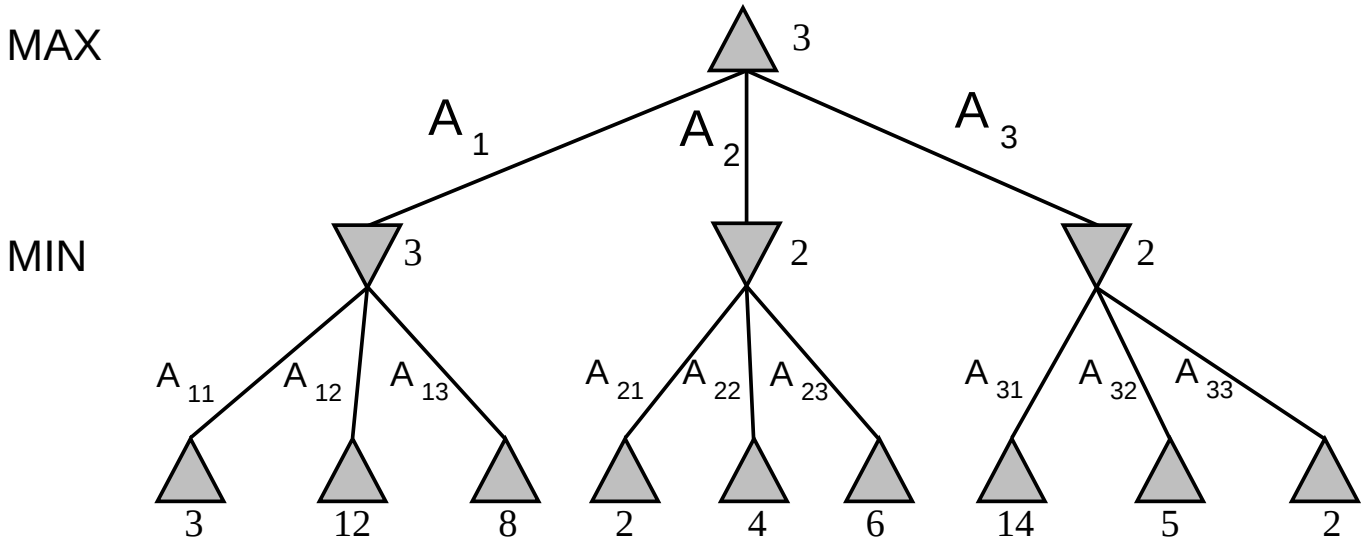
- **Stan początkowy:** zawiera pozycję planszy i informację, kto rozpoczyna grę.
- Zbiór **akcji:** zbiór poprawnych ruchów.
- **Test końcowy:** procedura stwierdzająca, czy nastąpił koniec gry.
- **Funkcja wypłaty (payoff function):** numeryczna wartość partii dla obu graczy.

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:

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

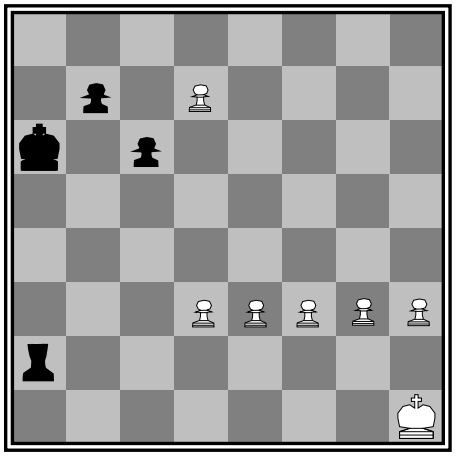
gdzie w_i jest numeryczną wartością przypisaną figurom (pion = 1, goniec i skoczek = 3, wieża = 5, hetman = 9), natomiast f_i oznacza liczbę figur danego rodzaju w danej pozycji.

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 - specjalne techniki do przypadku bicia.

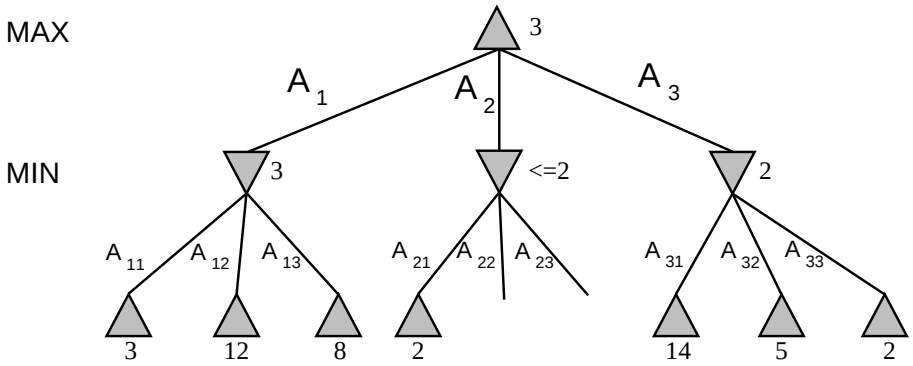
- Problem horyzontu: brak rozwiązania.

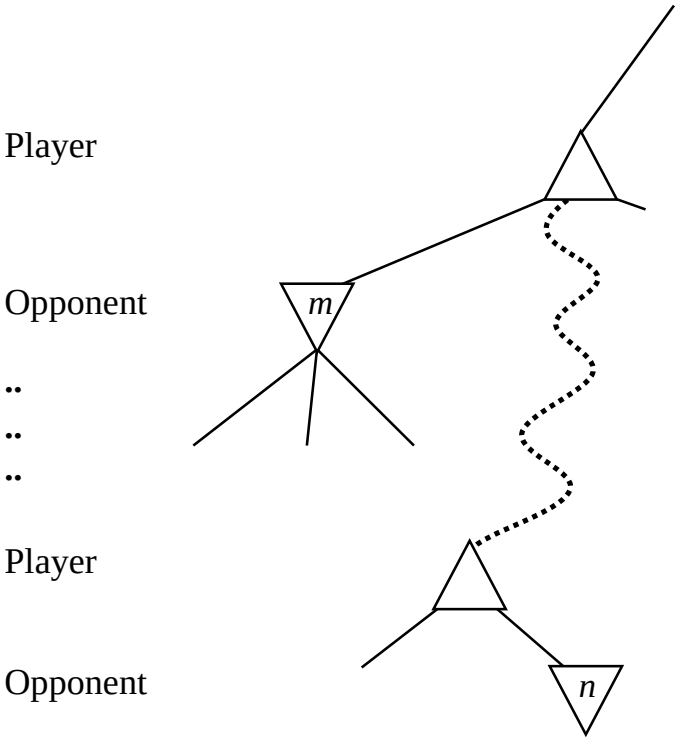


Black to move

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
      α ← MAX(α, MIN-VALUE(s, game, α, β))
      if α ≥ β 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
      β ← MIN(β, MAX-VALUE(s, game, α, β))
      if β ≤ α 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.

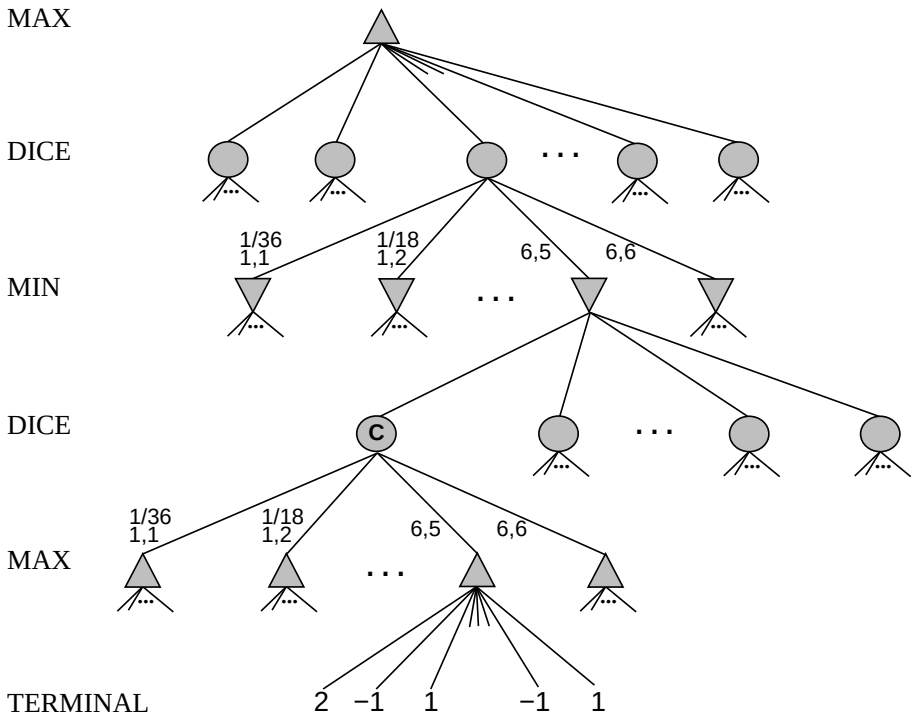
Mini-max: $O(b^d)$.

Alfa-beta:

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

Gry z elementem losowości

Drzewo gry dla gry backgammon:

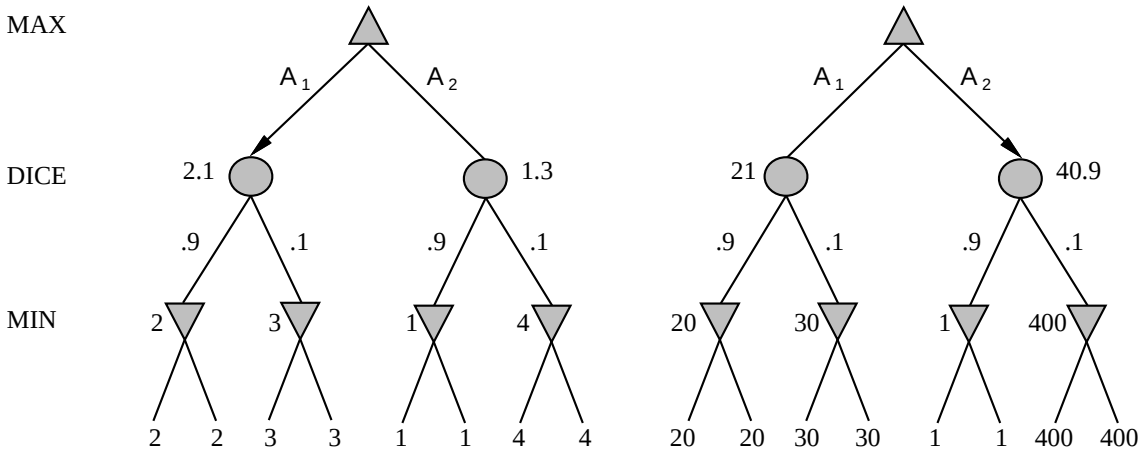


Nie możemy obliczyć wartości węzła C. Możemy tylko obliczyć jego *wartość oczekiwaną*.

Niech d_i będzie możliwym wynikiem rzutu kośćmi, $P(d_i)$ prawdopodobieństwem tego rzutu. Niech $S(\mathsf{C}, d_i)$ oznacza zbiór wszystkich poprawnych pozycji, do których można dojść z C wyrzucając d_i. Wówczas wartość oczekiwana węzła C wynosi

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

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:

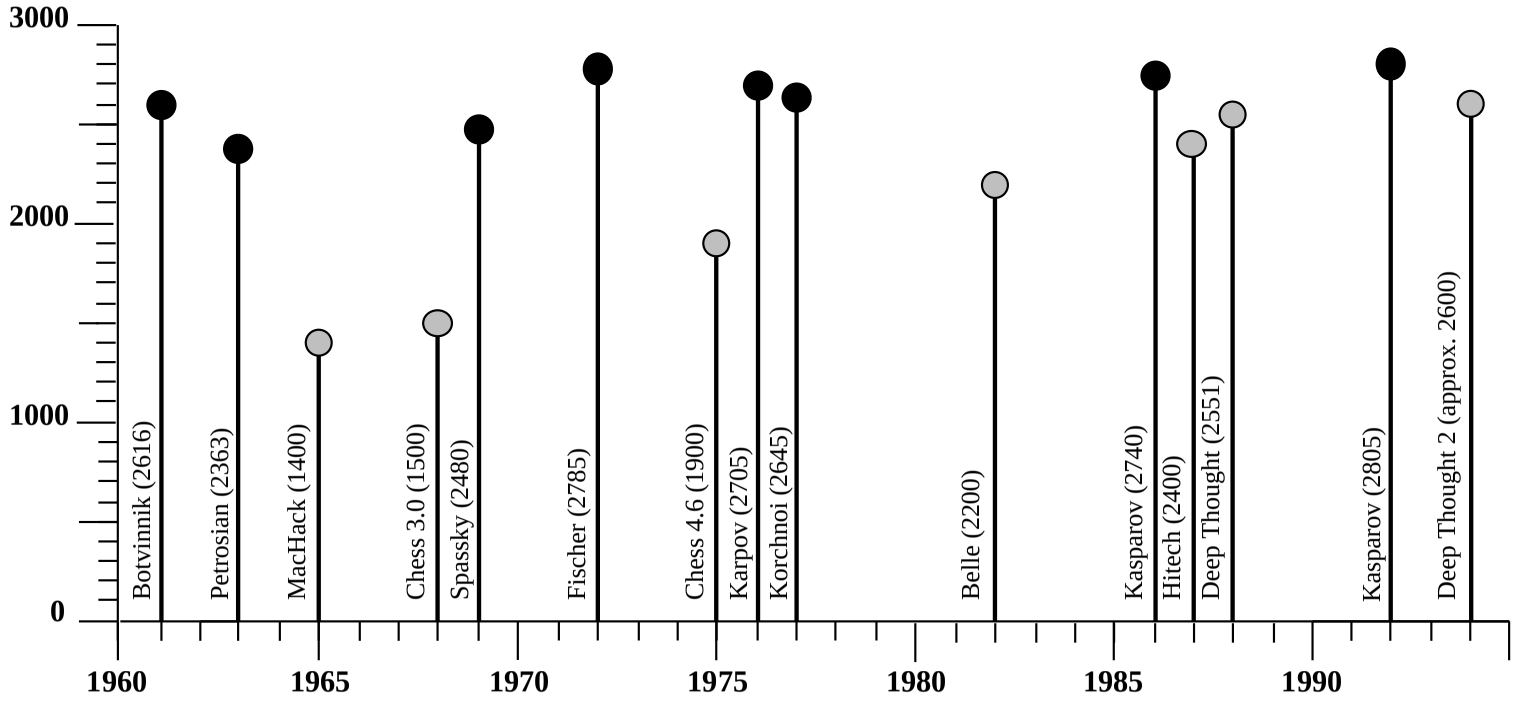


Uwaga2: Złożoność procedury expectminimax: $O(b^d * n^d)$, gdzie n jest liczbą możliwych wyników rzutu kostką (kostkami).

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 pierwszej trójce na świecie.

5. Go

Nie ma dobrych programów (b $\approx$ 360).