Sztuczna inteligencja w grach

Krzysztof Grąbczewski
Katedra Informatyki Stosowanej
Uniwersytet Mikołaja Kopernika
Toruń
3 czerwca 2020

http://www.is.umk.pl/~kg/zajecia/SIwG/SIwG.pdf

Literatura

- slajdy,
- Brian Schwab – *AI game engine programming* (2004, Charles River Media; drugie wydanie 2009, Course Technology/Cengage Learning)
- Ian Millington, John Funge – *Artificial intelligence for games*, 2009, Elsevier,
- Steve Rabin (edytor) – *Game AI Pro* (2014, CRC Press) & *Game AI Pro 2* (2015, CRC Press) – http://www.gameaipro.com/,
- Stuart Rusell, Peter Norvig, *Artificial Intelligence. A Modern Approach.*, 2nd Edition 2003, 3rd Edition 2010, http://aima.cs.berkeley.edu/index.html,
- Patrick H. Winston, *Artificial Intelligence*, 3rd Edition, Addison Wesley 1992.

Wstępny program zajęć

- metody szukania,
  - szukanie dróg, również optymalnych (best-first, A*, algorytmy grafowe)
  - szukanie z adwersarzem (min-max, obcięcie alfa-beta, ...)
- proceduralne generowanie elementów światów gier,
  - losowe kształtowanie terenu, szum Perlina,
  - labirynty - tworzenie, szukanie dróg,
  - automaty komórkowe,
- systemy decyzyjne (drzewa zachowań, systemy regułowe),
- automaty stanów skończonych, również rozmyte, łańcuchy Markowa,
- algorytmy inteligencji zbiorowej (genetyczne, rojowe),
- systemy rozmyte,
- wszystko bardzo praktycznie.

Laboratorium

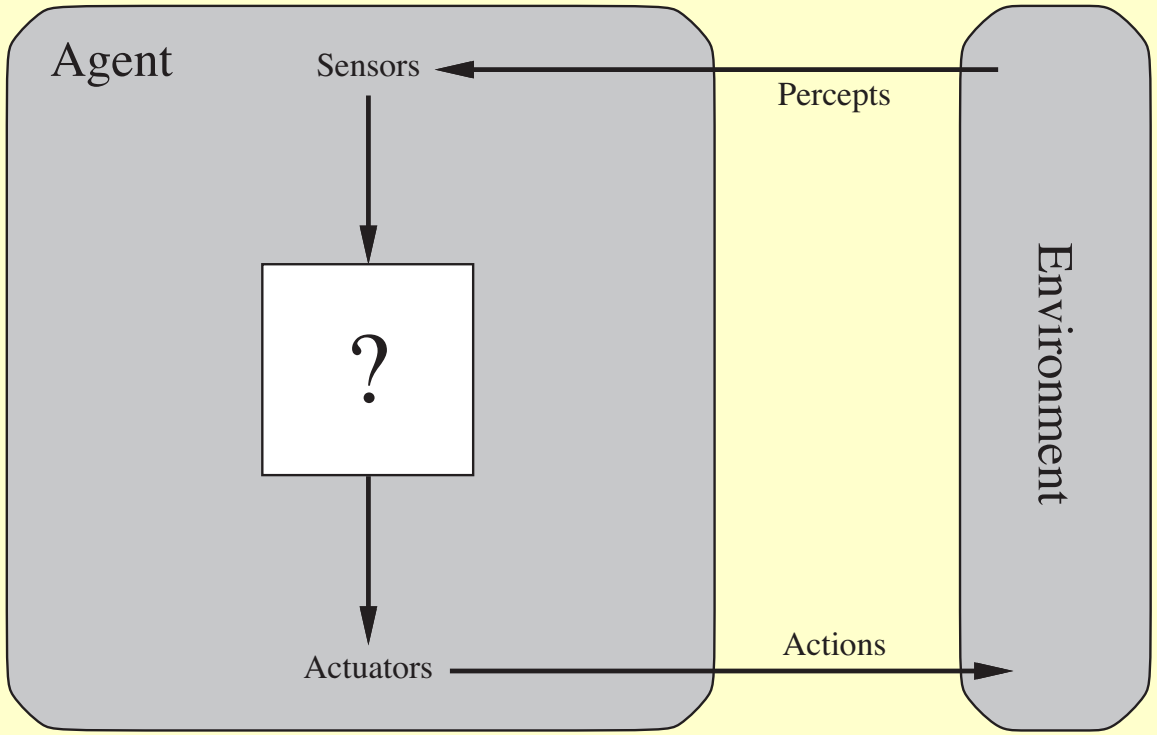
Cel: Biblioteka różnych ogólnych narzędzi przydatnych do programowania SI w grach oraz programy demonstracyjne.

- szukanie dróg na planszy 2D (różne kształty, różne sąsiedztwa pól, wagi połączeń?, przeszkody),
- szukanie z adwersarzem (gra w kółko i krzyżyk, warcaby, „oddawane” warcaby),
- generowanie labiryntów i szukanie dróg,
- generowanie ukształtowania terenu,
- wielu agentów na planszy (rywalizujących i współpracujących),

Wszystko z wizualizacją.

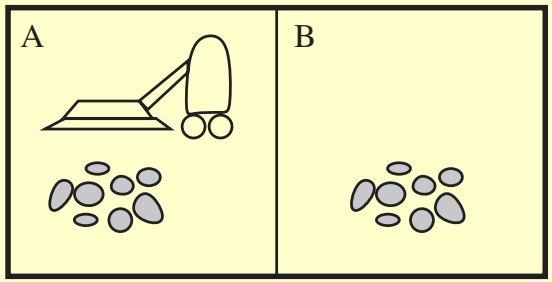
Agent, agenty

- **Agent** – ktoś kto (lub coś co) postrzega **środowisko** za pomocą **czujników** i wpływa na środowisko (działa) poprzez **urządzenia wykonawcze** (ang. *agent, environment, sensor, actuator*).
- **Spostrzeżenie** (ang. *percept*) – stan wejść agenta w danej chwili.
- Agent może analizować **ciąg spostrzeżeń** (ang. *percept sequence*) – całą historię stanów wejść.



Agenty

- Funkcja agenta – mapowanie ciągów spostrzeżeń na działania=akcje (wejść na wyjścia)
  - tabela działań,
  - program agenta.
- Prosty przykład środowiska dla agenta:



- Tabela działań:

Ciąg spostrzeżeń		Działanie
[A, czysto]	$\rightarrow$	w prawo
[A, brudno]	$\rightarrow$	odkurzaj
[A, czysto], [A, czysto]	$\rightarrow$	w prawo
[A, czysto], [A, brudno]	$\rightarrow$	odkurzaj
...		

Ocena działań agenta

- Agent **racjonalny**
- Miary jakości działań – zależne od celu, jakiemu agent ma służyć
  - **Racjonalność** = maksymalizacja miary jakości działań.
- Przykład odkurzacza – agent **„odkurzaj gdy brudno przejdź do drugiego pomieszczenia, gdy czysto”** – racjonalny, czy nie?
  - ilość zebranego kurzu,
  - ocena czystości pomieszczeń,
  - kary za zużywanie energii, ...
- **Warunki środowiska** istotnie wpływają na racjonalność działań
  - znajomość geografii – z pomieszczenia A nie ma sensu próbować iść w lewo,
  - ocena prawdopodobieństwa wystąpienia kurzu.
- Cechy zaawansowanych agentów:
  - gromadzenie informacji,
  - uczenie się,
  - autonomiczność.

Środowiska

- **Specyfikacja zadania** dla agenta:
  - Jakość działania (ang. *Performance*)
  - Środowisko (ang. *Environment*)
  - Urządzenia wykonawcze (ang. *Actuators*)
  - Czujniki (ang. *Sensors*)
- Przykład: **taksówkarz**
  - Miara jakości: bezpieczeństwo, czas dojazdu, przestrzeganie przepisów, wygoda podróży, maksymalizacja zysku
  - Środowisko: drogi, inne pojazdy, piesi, klienci
  - Urz. wykonawcze: kierownica, pedał gazu, pedał hamulca, przełączniki, klakson, wyświetlacz dla klientów
  - Czujniki: kamery, sonar, prędkościomierz, GPS, licznik, miernik przyspieszenia, czujniki silnika, klawiatura

Właściwości środowisk

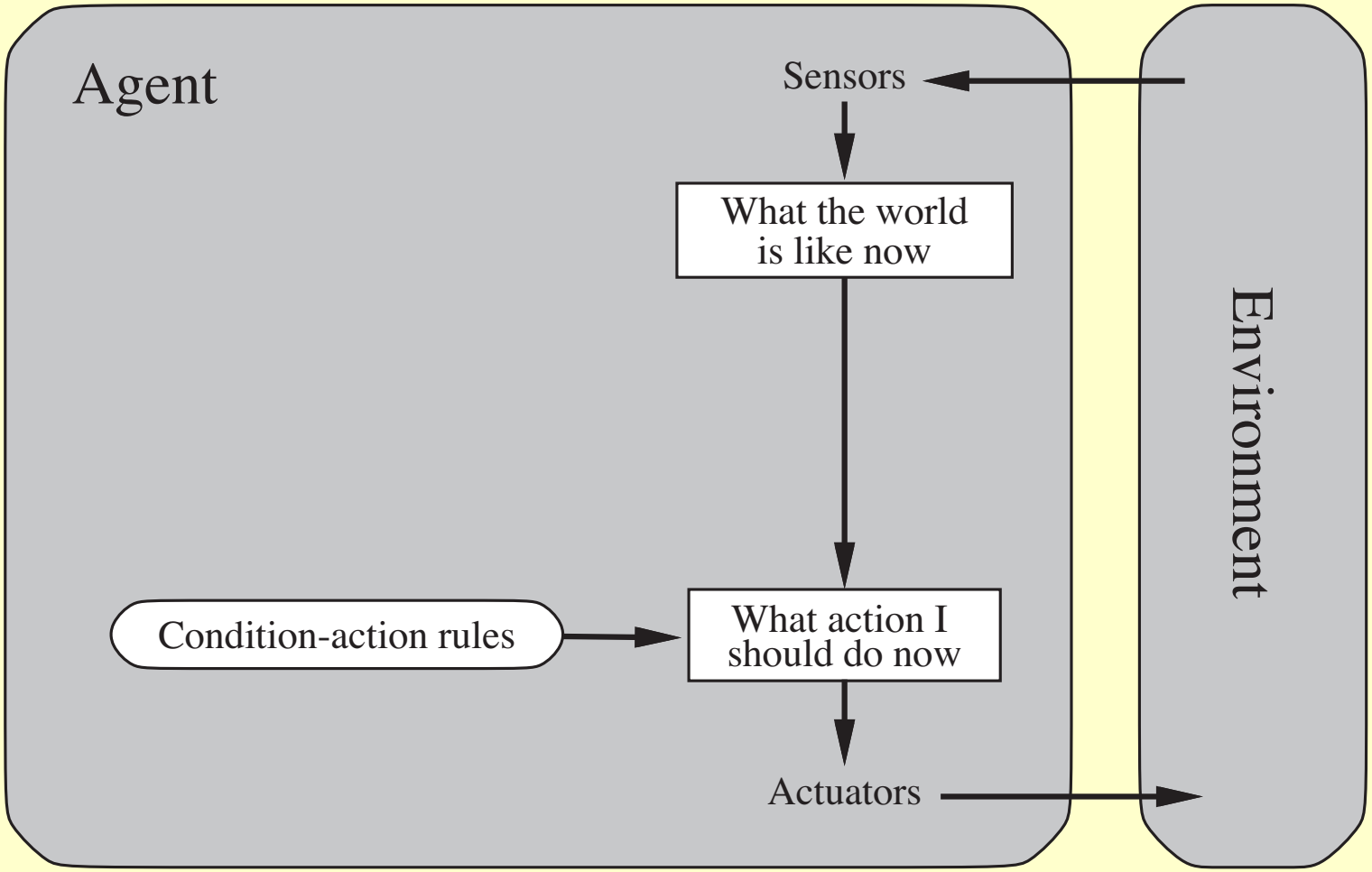
- w pełni i częściowo obserwowalne (ang. *fully/partially observable*)
- deterministyczne i stochastyczne (ang. *deterministic/stochastic*)
- epizodyczne i sekwencyjne (ang. *episodic/sequential*)
- statyczne i dynamiczne (ang. *static/dynamic*)
- dyskretne i ciągłe (ang. *discrete/continuous*)
- jedno- i wieloagentowe (ang. *single/multiagent*)
- współpracy i współzawodnictwa (ang. *cooperative/competitive*)

Struktury agentów

- Agent = architektura + program
- Architektura: czujniki i urządzenia wykonawcze
- Programy zapisywane różnymi językami
  - meta-kod,
  - schematy blokowe,
  - zestawy reguł warunek-działanie, ...
- Podstawowe typy agentów:
  - działania jako proste odruchy (ang. *simple reflex*),
  - działanie oparte na modelach (ang. *model-based*),
  - ukierunkowanie na cele (ang. *goal-based*),
  - użyteczność działań (ang. *utility-based*).

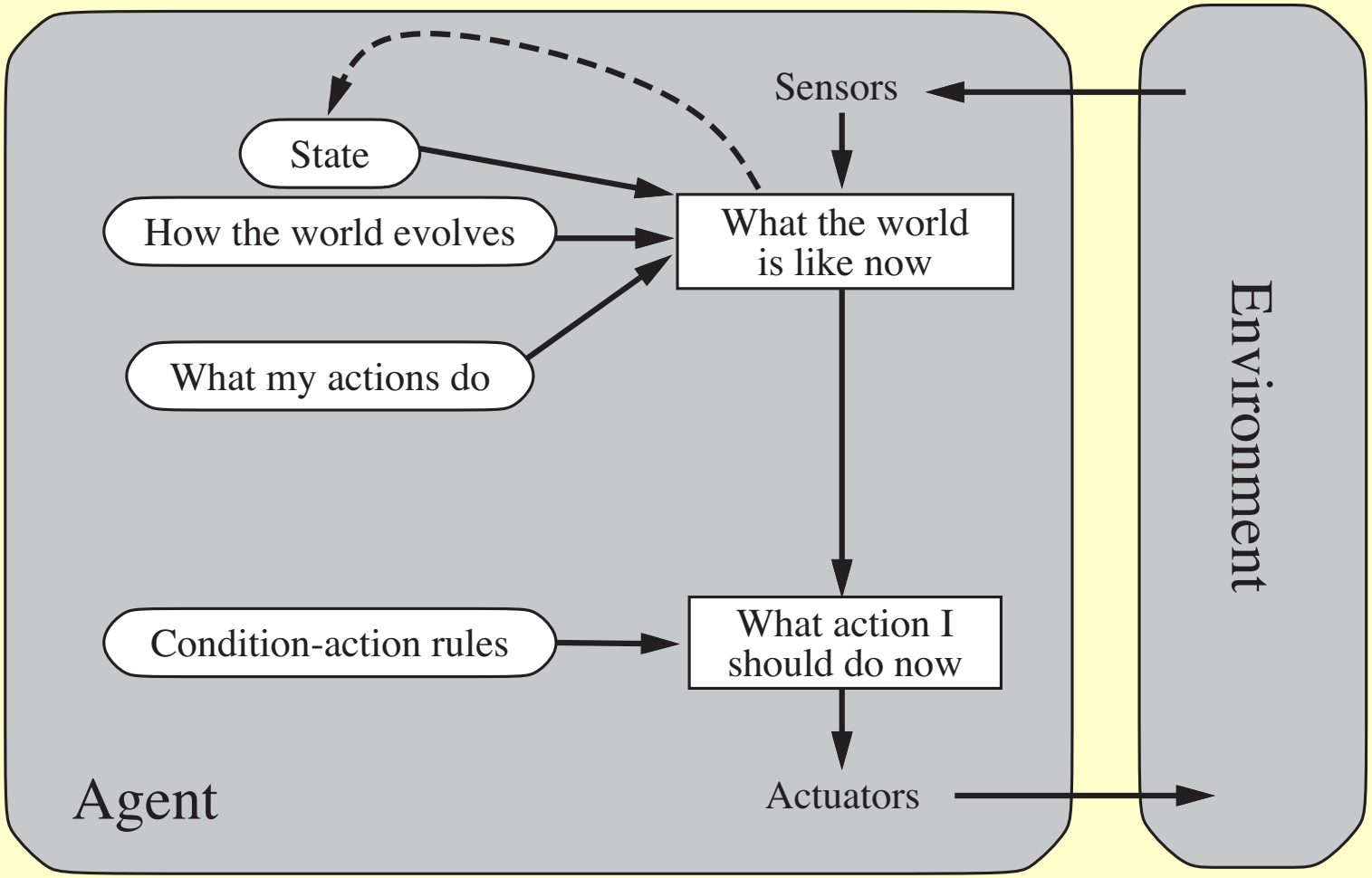
Agenty prostych odruchów

- Programy odpowiadają wprost tabelom działań bądź zestawom reguł warunek–działanie



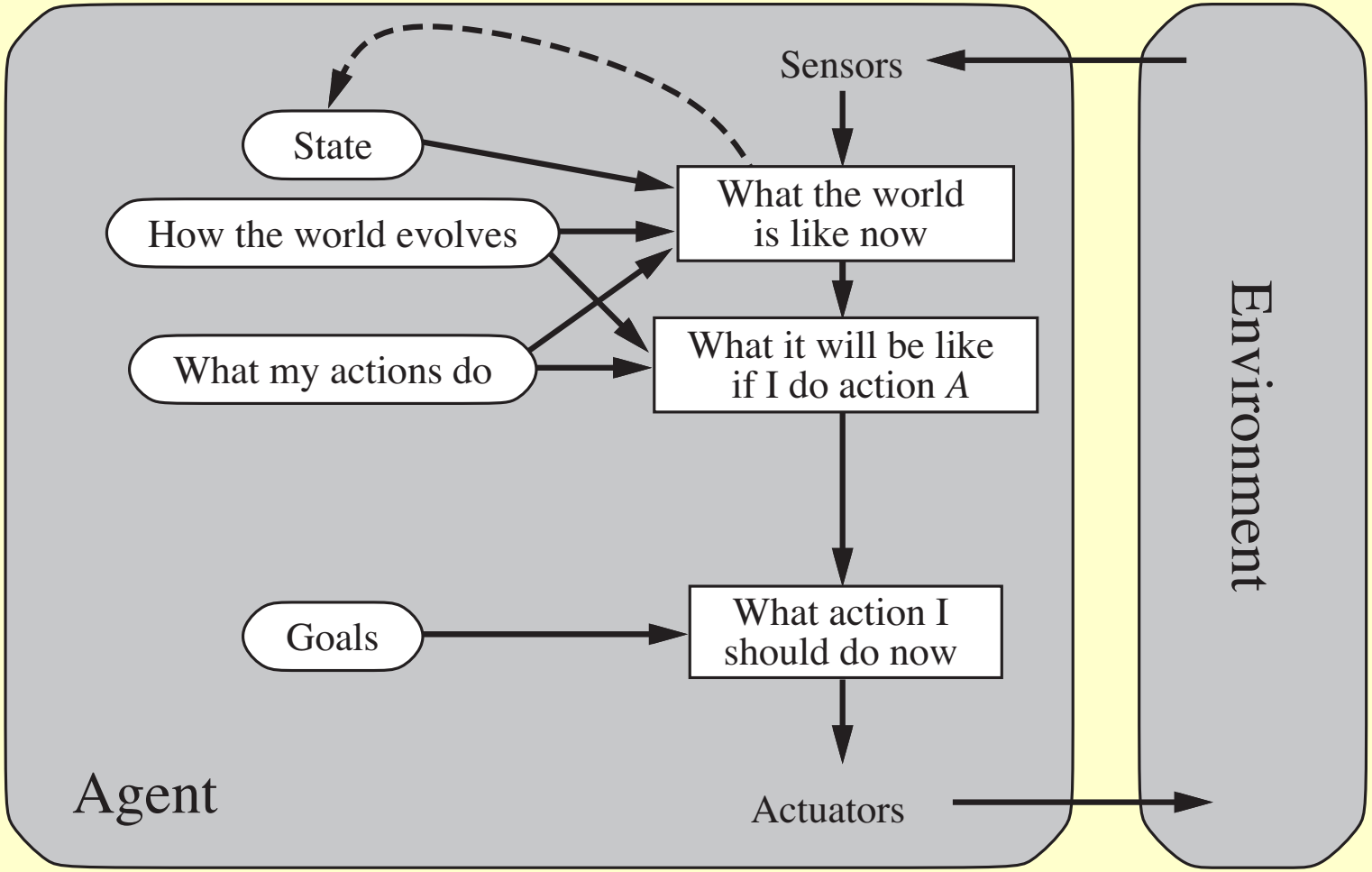
Odruchy oparte na modelach

- Wewnętrzna **pamięć** (**stan**) pozwala uwzględniać aspekty nieobserwowalne w bieżącym wejściu.
- **Model świata** uwzględniający wpływ działania agenta.



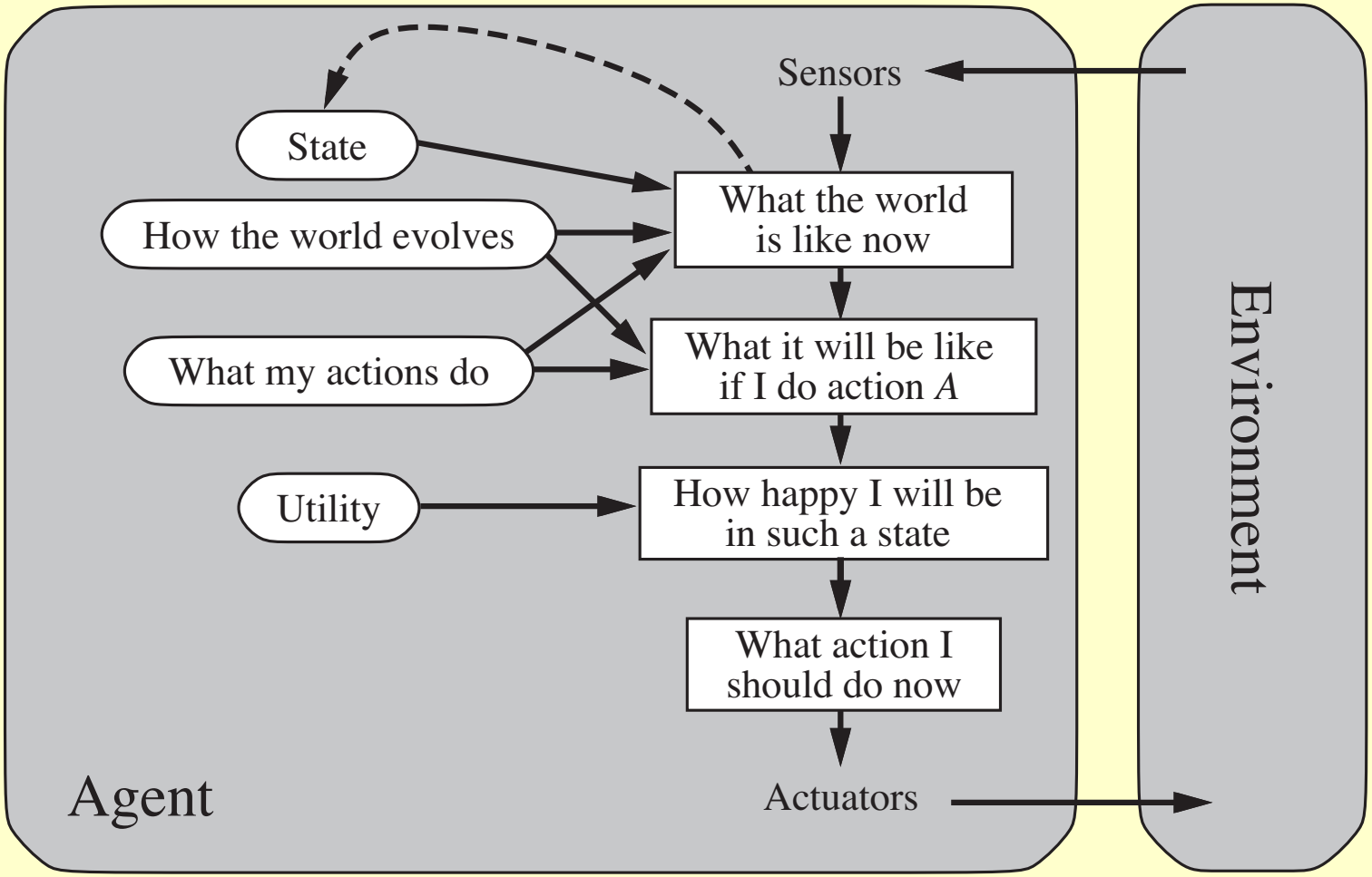
Agenty ukierunkowane na cele

- Ocena na ile działanie agenta zbliży go **do celu**.
- Bardziej złożone procedury maksymalizacji funkcji celu, **procesy szukania**, **planowanie**.



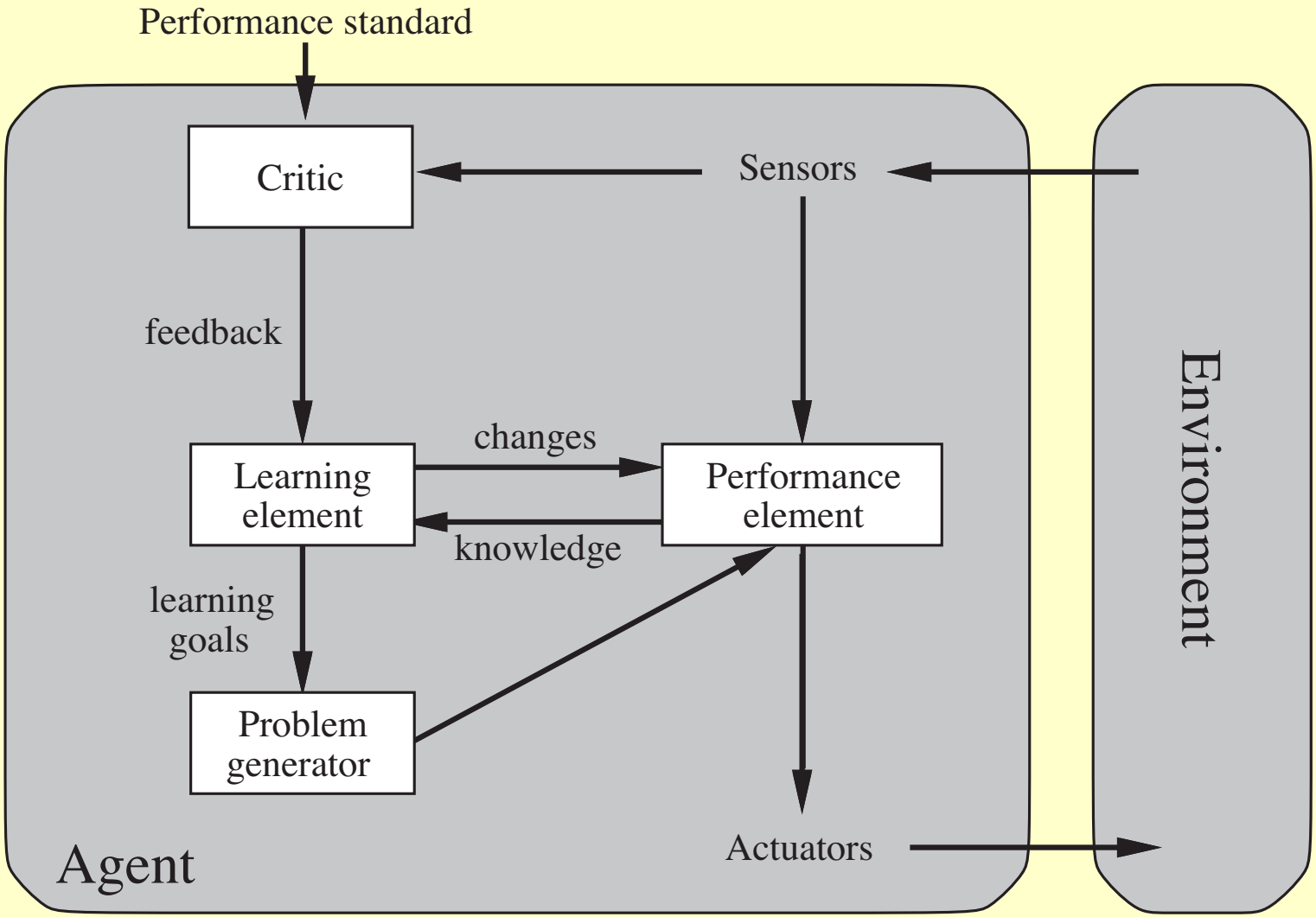
Agenty z funkcjami użyteczności

- Funkcje użyteczności oceniają jakość decyzji (poziom satysfakcji z danego działania)



Agenty uczące się

- Uczenie się może wzbogacić **każdy z typów** agentów.
- **Część wykonawcza** korzysta z wyników **procesów uczenia się**.
- Uczenie się **ze wzmocnieniem** (z nagrodą lub karą).



Rozwiązywanie problemów

Definicja problemu:

- **zbiór stanów** $\mathcal{S}$ (przestrzeń),
- stan początkowy $s_0 \in \mathcal{S}$,
- **cel** (podzbiór zbioru stanów $\mathcal{G} \in \mathcal{S}$, funkcja-test osiągnięcia celu $G : \mathcal{S} \to \{0, 1\}$),
- **zbiór** możliwych **działań** agenta $\mathcal{A}$ i ich skutków w danym kontekście, często funkcja następnika (ang. *successor*) $\text{succ} : \mathcal{S} \to 2^{\mathcal{A} \times \mathcal{S}}$,
- funkcja kosztu (koszt ścieżki, koszt kroku).

Rozwiązanie = **ścieżka** od stanu początkowego do celu.

Problem szukania

Proces szukania

- To proces wyznaczający sekwencję działań prowadzących do celu.
- Algorytm szukania otrzymuje na wejściu definicję problemu i zwraca ciąg działań (**rozwiązanie**).

Środowisko agenta stosującego problemy szukania:

- obserwowalne (przynajmniej znany stan początkowy),
- deteministyczne,
- statyczne (założenie dynamiczności istotnie zmienia problem),
- dyskretne,
- jednoagentowe.

Różne cele

- znalezienie **jakiejkolwiek** drogi,
- znalezienie **optymalnej** drogi,
- znalezienie **stanu** celu,
- działanie **w czasie rzeczywistym**, **z oponentem** itp.

Przykład – odkurzacz

Stany:

$\mathcal{S} = \{L, P\} \times \{\text{czysto}, \text{brudno}\}^2$ – pozycja odkurzacza i stan pomieszczeń ($2 \cdot 2^2 = 8$ stanów).

Stan początkowy:

Dowolny z możliwych.

Test celu:

Cel osiągnięty gdy w obu pomieszczeniach czysto.

Działania:

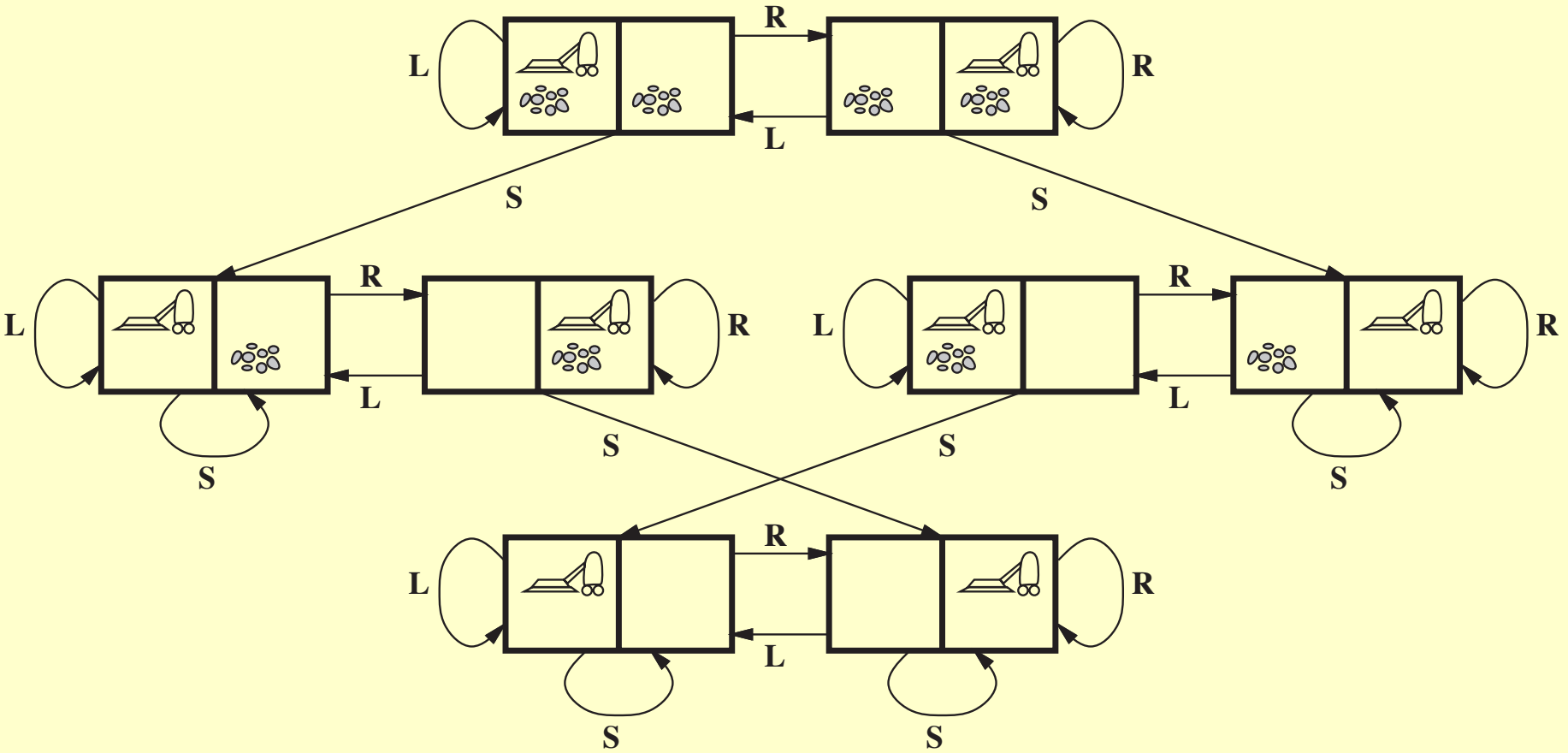
$\{\text{w lewo}, \text{w prawo}, \text{odkurzaj}\}$.

Koszt ścieżki:

Liczba kroków (fragmentów ścieżki).

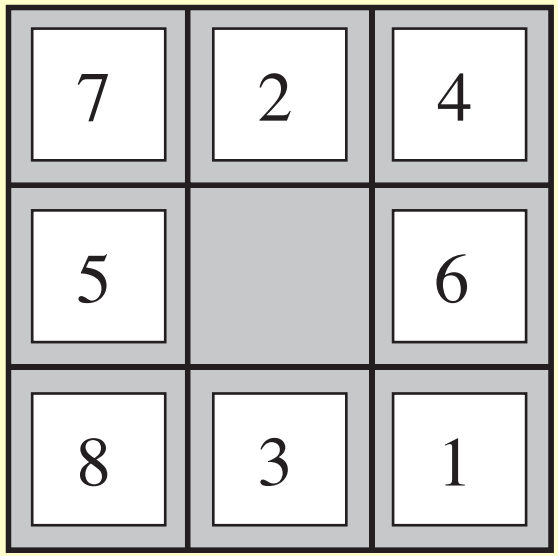
Ogólniej: dla n pomieszczeń mamy $n \cdot 2^n$ stanów.

Przykład – odkurzacz

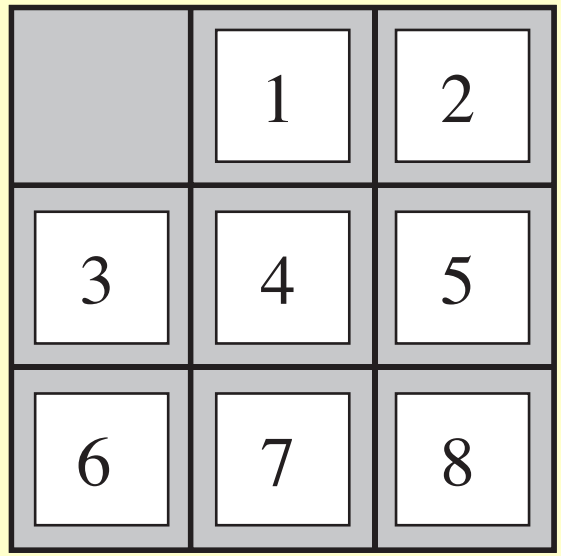


Przykład – ósemka

Popularna zabawka polegająca na ustawianiu w kolejności ośmiu ponumerowanych kostek poprzez przesuwanie ich w zamkniętej kwadratowej tabliczce 9 pól.



Start State



Goal State

Przykład – ósemka

Stany:

opis położeń wszystkich kostek w tabliczce (9! możliwości).

Stan początkowy:

Dowolny z możliwych. Każdy cel jest osiągalny z połowy stanów początkowych.

Test celu:

Kostki ułożone w kolejności, puste miejsce w lewym górnym rogu (oczywiście można inaczej).

Działania:

$\{$w lewo, w prawo, w górę, w dół$\}$.

Koszt ścieżki:

Liczba kroków (fragmentów ścieżki).

Osiągnięcie celu jest dość łatwe, ale **znalezienie optymalnej ścieżki NP-zupełne**.

Przykład – 8 hetmanów

Zadanie: ustawić 8 hetmanów na szachownicy tak by się żadna para nie szachowała.
Możliwe podejścia:

- inkrementacyjne/przyrostowe – dostawianie po jednym,
- całkowite – rozmieszczenie 8 od razu i ich przestawianie.

Stany:
Pozycje $k \in \{0, 1, \ldots, 8\}$ hetmanów.

Stan początkowy:
$k = 0$.

Test celu:
8 hetmanów poprawnie rozmieszczonych.

Działania:
Dostawienie kolejnego hetmana.

Koszt ścieżki:
nieistotny/stały.

Liczba sekwencji: $64 \cdot 63 \cdots 57 \approx 1.8 \times 10^{14}$.

Przykład – 8 hetmanów

Efektywniejsze podejście:

Stany:

Pozycje $k \in \{0, 1, \ldots, 8\}$ hetmanów w kolejnych kolumnach.

Działania:

Dostawienie kolejnego hetmana w kolejnej kolumnie.

Liczba sekwencji: $8^8 = 16 \times 10^6$

Jeszcze inne możliwości:

- sprawdzanie permutacji ($8! = 40320$ stanów),
- dostawianie hetmanów tylko na pozycjach nieszachowanych (2057 stanów, ale kosztowniejsze ich odnajdywanie).

Ogólnie, problem N hetmanów dla szachownicy $N \times N$, dla większych N, daje przestrzeń stanów niemożliwą do przeiterowania.

Przykład szukania optymalnych tras

Stany:
Aktualna lokalizacja i bieżący czas.

Stan początkowy i test celu:
definiowane indywidualnie dla problemu – miejsce startu, miejsce docelowe i pożądany czas dotarcia.

Działania:
Skorzystanie z dostępnego lotu.

Koszt ścieżki:
Różnie: koszt finansowy, czas lotu, czas oczekiwania, konieczność odpraw celnych i imigracyjnych, jakość miejsc, typ samolotu, pora dnia itp.

Inne przykłady ważnych problemów

- Problemy objazdowe – odwiedzić wszystkie miejsca i wrócić do źródła
  - komiwojażera,
  - podróży dookoła świata,
  - skoczka szachowego.
- Lis, gęś i ziarno lub podobny problem kanibali i misjonarzy.
- Problem układów VLSI – rozkład elementów i trasowanie ścieżek.
- Nawigacja robotem w ciągłym świecie.
- Automatyczny montaż – kolejność montażu wielu komponentów w przestrzeni 3D.
- Projektowanie białek.
- „Inteligentne” wyszukiwanie informacji w internecie.
- Gry planszowe, gry komputerowe, strategiczne.
- Dowodzenie twierdzeń matematycznych.

Procesy szukania

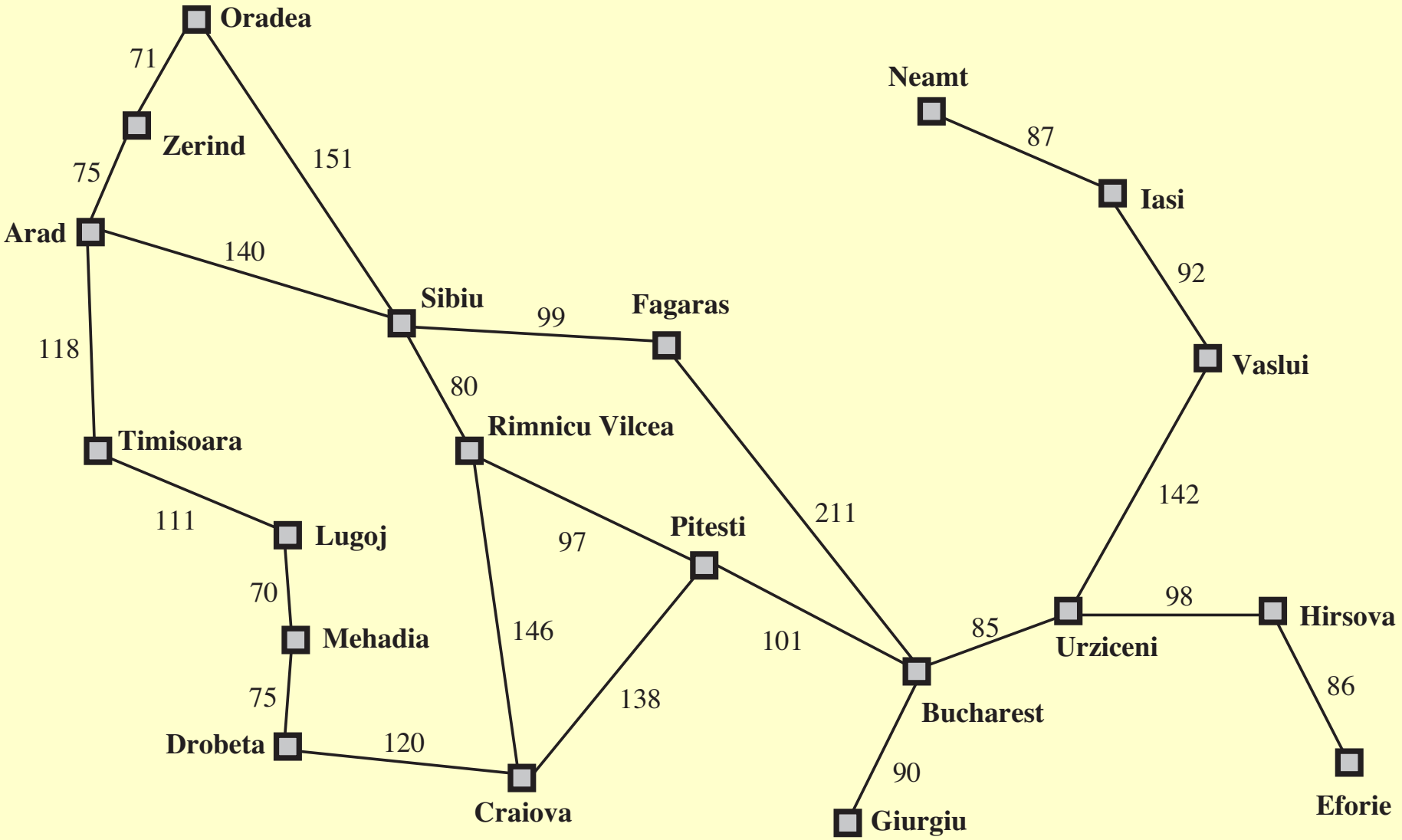
Struktura **drzewa**

- Stan początkowy to **korzeń**.
- W kolejnym kroku **rozwija się bieżący węzeł** według aktualnych możliwości działania.
- Różne **strategie szukania** – kolejność rozwijania wygenerowanych dotąd węzłów.
  - (ang. *fringe*) – zestaw węzłów-kandydatów do rozwijania.

Węzeł drzewa

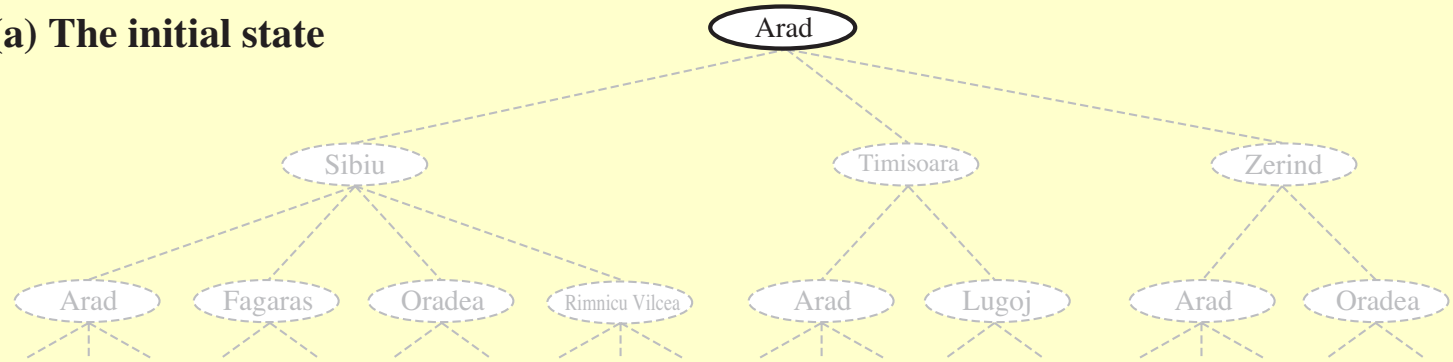
- bieżący **stan**,
- węzeł nadrzędny (**rodzic**),
- **działanie**, które doprowadziło do bieżącego stanu,
- **koszt** ścieżki,
- **głębokość**.

Przykład – fragment Rumunii

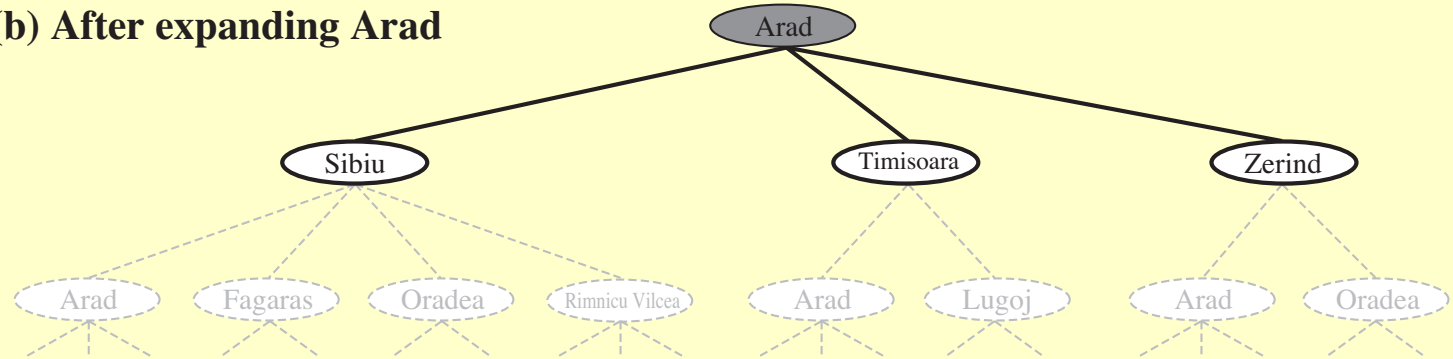


Przykład – fragment Rumunii

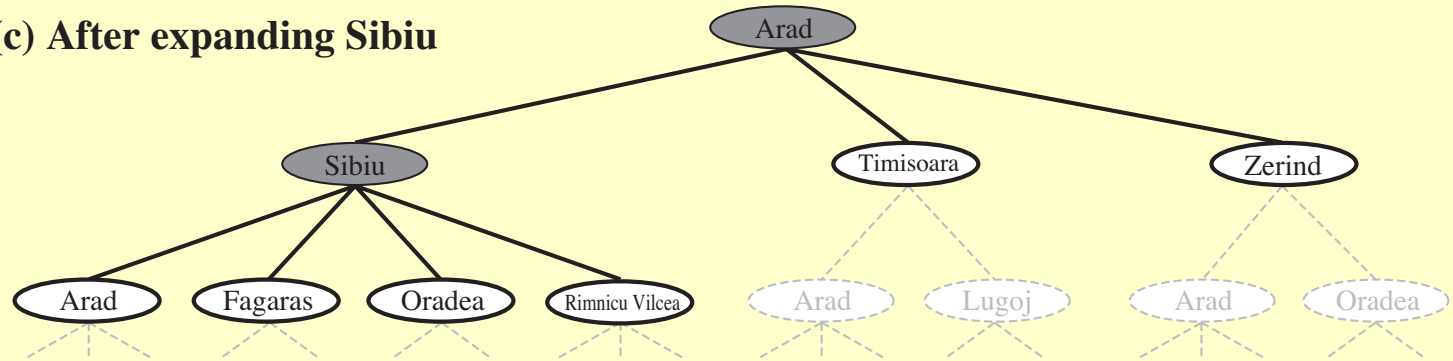
(a) The initial state



(b) After expanding Arad



(c) After expanding Sibiu



Szukanie w drzewie

Prototype: ***TreeSearch(problem,strategy)***
Input: Problem definition, search strategy.
Output: A solution or failure.

1. initialize the tree with the node of initial state
2. **while** exist candidates for expansion **do**
    1. use the strategy to select the node to examine/expand
    2. **if** a goal state reached **then return** the solution
    3. expand the node (grow the tree)
3. **return** failure

Szukanie z kolejką

Prototype: ***TreeSearchWithQueue(problem, fringe)***
Input: Problem definition, fringe interface.
Output: A solution or failure.

1. fringe ← one-element collection with the initial state
2. **while** fringe is not empty **do**
    1. node ← remove first node from the fringe
    2. **if** a goal state reached **then return** the solution
    3. expand the node – add new nodes to the fringe
3. **return** failure

Analiza procesów szukania

Najważniejsze aspekty:

- **Zupełność** – czy metoda gwarantuje dotarcie do celu, jeśli taki istnieje.
- **Optymalność** – czy znajdziemy optymalne rozwiązanie.
- **Złożoność czasowa**.
- **Złożoność pamięciowa**.

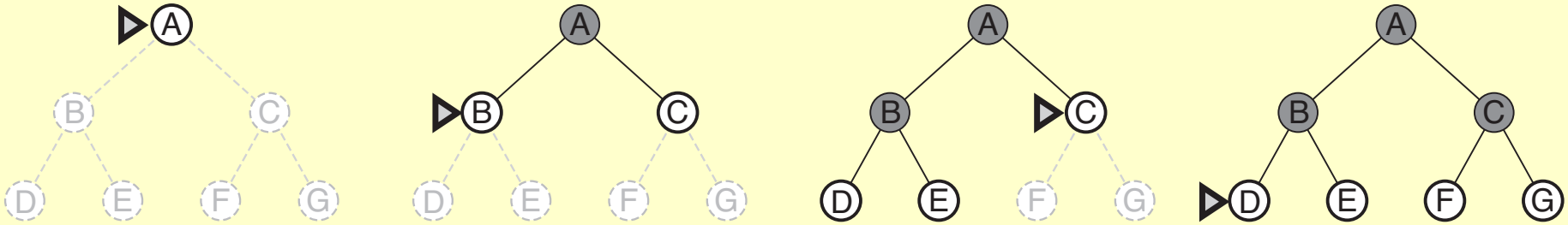
Podstawowe oznaczenia:

- b – czynnik rozgałęzień (ang. *branching factor*),
- d – minimalna głębokość drzewa, na której można znaleźć cel,
- m – maksymalna długość ścieżki.

Szukanie **na ślepo** (ang. *blind, uninformed*) i **heurystyczne** (ang. *heuristic, informed*).

Szukanie wszerz (ang. *breadth-first search*)

Szukanie w kolejności **od najpłytszych węzłów do coraz głębszych**



- Algorytm realizowany przez procedurę TreeSearchWithQueue i **zwykłą kolejkę** (FIFO).
- Liczba generowanych węzłów:

$$b + b^2 + \cdots + b^d + b^{d+1} - b \propto O(b^{d+1}) \tag{1}$$

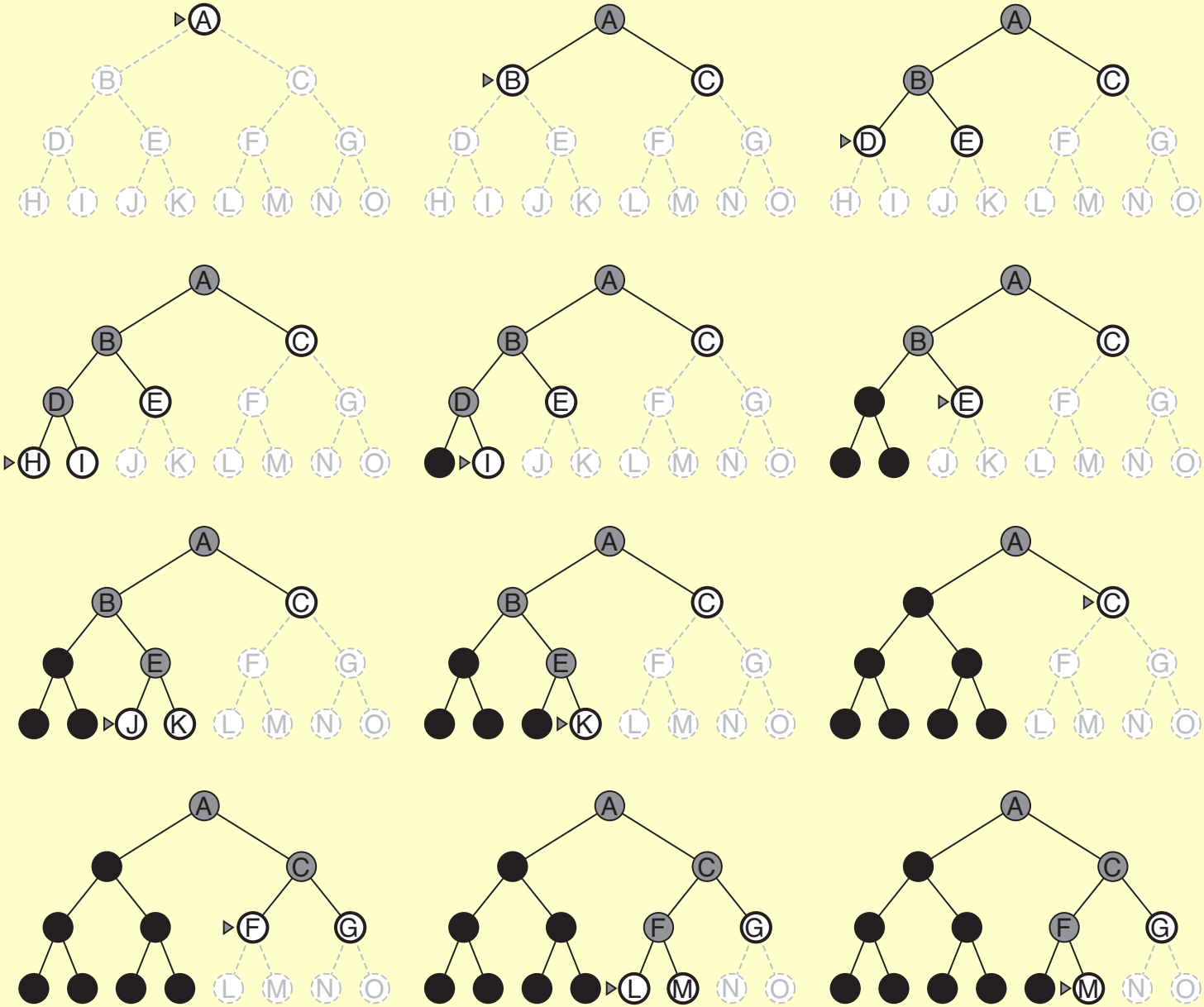
- Takie same: **złożoność czasowa i pamięciowa**, jednak pamięciowa doskwiera bardziej.
- Metoda **zupełna** i **optymalna**.

Szukanie wg kosztów (ang. *uniform-cost search*)

Tak jak wszerz, ale najpierw nie najkrótsze ścieżki, a te z **najniższym kosztem**.

- Zerowy koszt kroku może zapętlić metodę.
- Algorytm realizowany przez procedurę TreeSearchWithQueue i **kolejkę priorytetową**.
- **Zupełność** i **optymalność** zachowana.
- **Złożoność czasowa i pamięciowa**: $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, gdzie C^* to koszt optymalnego rozwiązania, a ϵ to dolne ograniczenie kosztu pojedynczego kroku.

Szukanie w głąb (ang. *depth-first search*)



Szukanie w głąb (ang. *depth-first search*)

Szukanie w kolejności **od najgłębszych węzłów**.

- Algorytm realizowany przez procedurę TreeSearchWithQueue i **stos** („kolejkę” LIFO).
- Możliwość utknięcia w nieskończonej gałęzi – metoda **niezupełna**.
- Metoda **nieoptymalna**.
- Złożoność pamięciowa $O(bm)$.
- **Szukanie z nawrotami** (ang. *backtracking search*) pozwala zredukować złożoność pamięciową do $O(m)$.
- Łatwiejsza i bardziej naturalna **implementacja rekurencyjna**.

Rekurencja

Funkcje rekurencyjne w bardzo prosty sposób wykonują złożone operacje na danych o rekurencyjnej naturze.

Definicje rekurencyjne:

- **Liczby naturalne** N
  - $0 \in N$,
  - $n \in N \Rightarrow succ(n) \in N$.
- **Silnia** liczby $n \in N$
  - $0! \in N$,
  - $n \in N \Rightarrow succ(n)! = succ(n) \cdot n!$.
- **Listy** elementów zbioru X jako złożenie głowy i ogona (pierwszy element i reszta)
  - $() \in L$,
  - $x \in X \wedge M \in L \Rightarrow x.M \in L$ ($[x|M] \in L$ w notacji prologowej).
- **Drzewa binarne** z etykietami ze zbioru X
  - $() \in D$,
  - $x \in X \wedge d_1, d_2 \in D \Rightarrow \begin{array}{c} x \\ / \ \backslash \\ d_1 \ \ d_2 \end{array} \in D$.

Operacje rekurencyjne

Długość listy

$$dl(L) = \begin{cases} 0 & \text{jeśli } L = (), \\ 1 + dl(M) & \text{jeśli } L = x.M. \end{cases} \tag{2}$$

Suma wartości z węzłów drzewa

$$suma(d) = \begin{cases} 0 & \text{jeśli } d = (), \\ x + suma(d_1) + suma(d_2) & \text{jeśli } d = \begin{array}{ccc} & x & \\ / & & \backslash \\ d_1 & & d_2 \end{array}. \end{cases} \tag{3}$$

Rekurencyjne szukanie w głąb

Prototype: ***RecursiveDFS(problem, node)***
Input: Problem definition, current node.
Output: A solution or failure.

1. **if** node reached the goal **then return** the solution
2. **for each** subnode sn **do**
    1. result ← RecursiveDFS(problem, sn)
    2. **if** result is a solution **then return** the solution
3. **return** failure

Prototype: ***RecursiveDFS(problem)***

1. **return** RecursiveDFS(problem, the root node)

Szukanie z ograniczeniem głębokości

Szukanie z ograniczeniem głębokości (ang. *depth-limited search*).

Jak w głąb, ale **nie rozwijamy** gałęzi po uzyskaniu zadanej **głębokości** l.

- Dodatkowe ograniczenie łatwe do wprowadzenia.
- Złożoność **czasowa** $O(b^l)$.
- Złożoność **pamięciowa** $O(bl)$.
- Brak **zupełności** i **optymalności**.

Iteracyjne pogłębianie

Iteracyjne pogłębianie szukania w głąb (ang. *iterative deepening depth-first search*).

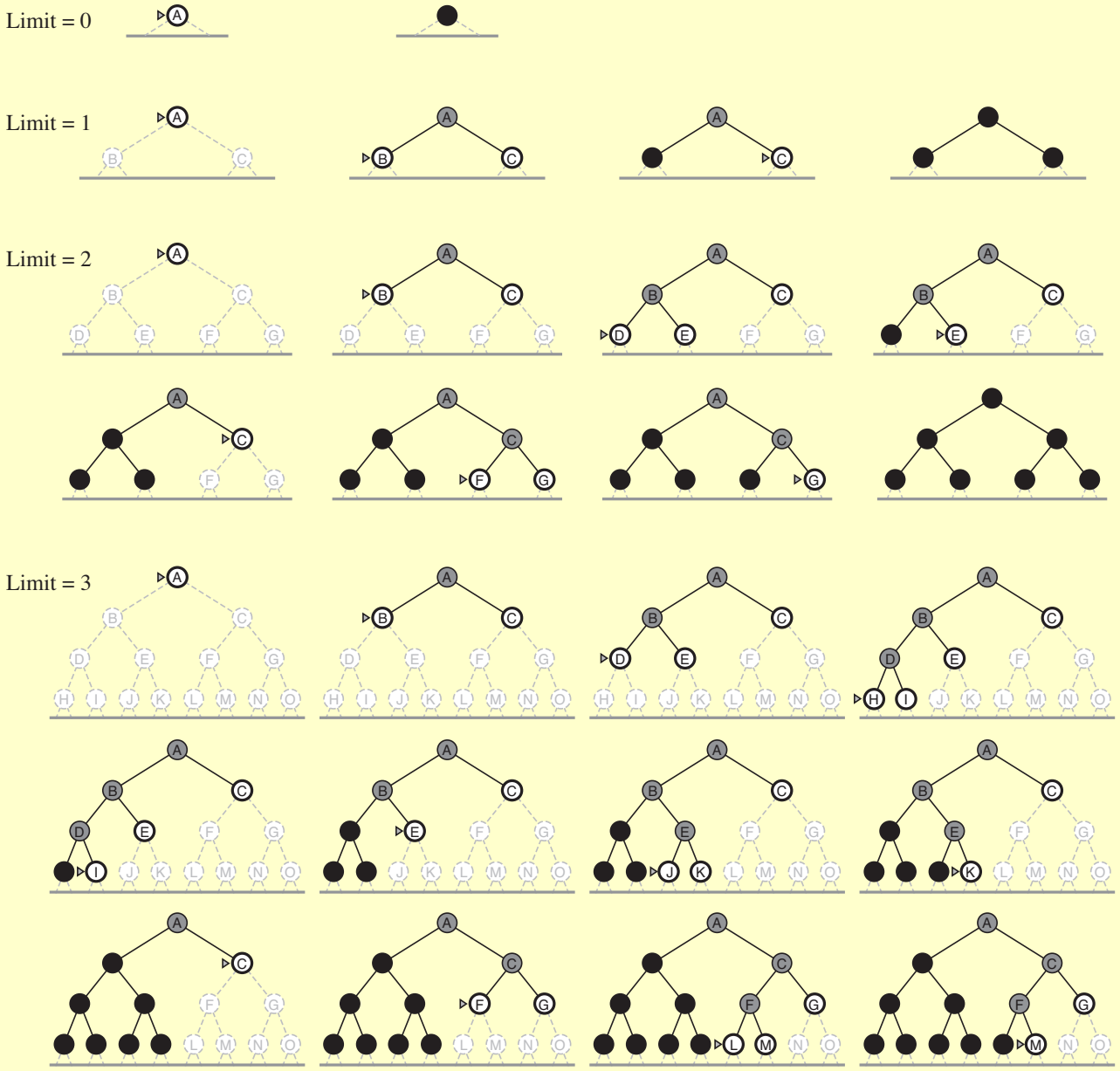
Prototype: ***IDDFS(problem)***
Input: Problem definition.
Output: A solution or failure.

1. **for** depth $\leftarrow 0$ **to** ∞ **do**
    1. result $\leftarrow$ DepthLimitedSearch(problem, depth)
    2. **if** result $\neq$ cutoff **then return** result

Właściwości:

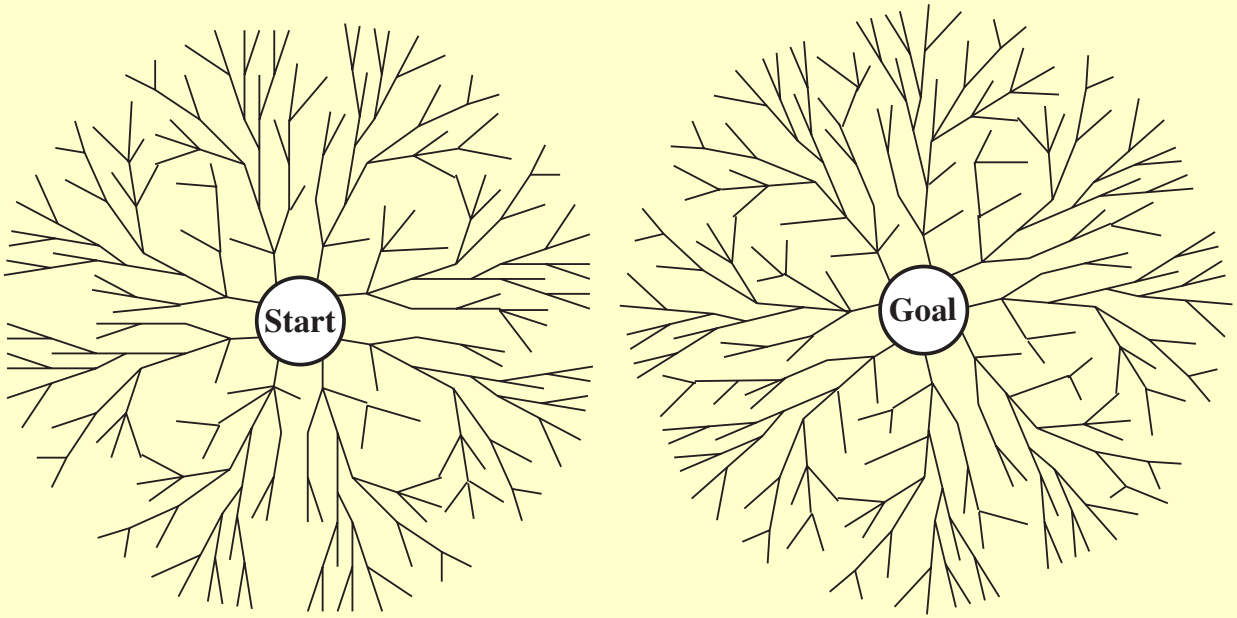
- Złożoność pamięciowa $O(bd)$.
- Złożoność czasowa $O(b^d)$ – powtórki nie kosztują tak dużo jakby się wydawało.
- Metoda zupełna i optymalna.

Iteracyjne pogłębianie



Szukanie dwukierunkowe

Dwa równoległe szukania – od startu do celu i odwrotnie:



Znacznie szybciej, bo $2 \cdot b^{d/2} << b^d$.

Problematyczne, gdy cel obejmuje wiele stanów (być może nawet nieskończenie wiele).

Różne problemy

Ponowne odwiedzanie tych samych stanów

- znacznie **spowalnia** szukanie
- może być wyeliminowane przez **oznaczanie sprawdzonych stanów** (flaga w obiekcie stanu lub tablica haszująca)

Brak pełnej informacji o bieżącym stanie

- **nieznany stan początkowy**,
- sytuacja niekoniecznie beznadziejna – przykład odkurzacza – sekwencja (w prawo, odkurzaj, w lewo, odkurzaj) zapewni **osiągnięcie celu niezależnie od stanu początkowego**,
- trudno oceniać **optymalność** rozwiązań,
- gotowość na **niespodzianki**,
- **gry z adwersarzem**.

Szukanie optymalnych dróg

Ogólny schemat – **generuj i sprawdzaj** (ang. *generate and test*)

Prototype: ***GenerateAndTest(P,G,SC)***
Input: Problem P definition, state generator G, stop condition SC.
Output: A solution or failure.

1. **while** SC not satisfied **do**
   1. S ← generate next state with G
   2. **if** S is a goal state **then return** the solution
2. **return** failure

Najbardziej prymitywne podejścia:

- Monte Carlo – losuj aż trafisz,
- brytyjskie muzeum – sprawdź wszystko i wybierz.

Szukanie optymalnych dróg

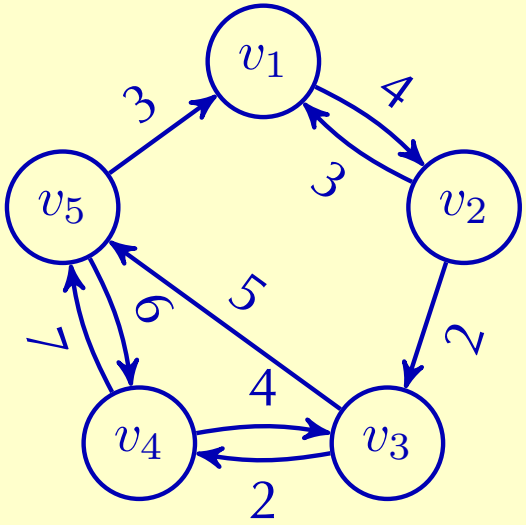
Rozwijaj i ograniczaj (ang. *branch and bound*) – ogólna technika polegająca na rozwijaniu kolejnych węzłów i ograniczaniu się do tych najbardziej obiecujących.

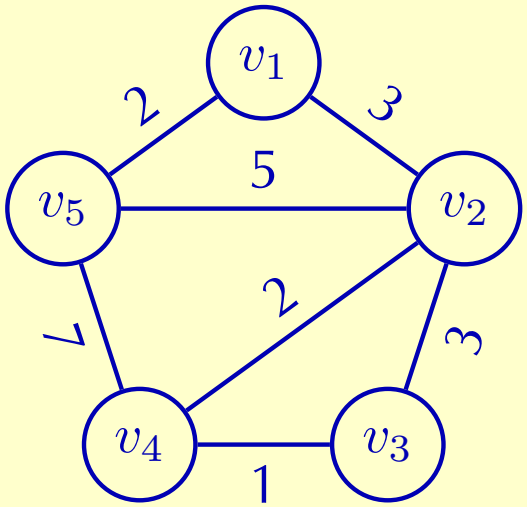
Najprostsze ujęcie:

- Badanie węzłów **w kolejności rosnących kosztów**.
- Rozwijane węzły do **kolejki priorytetowej**.
- **Początek: korzeń** do kolejki
- Potem pobieramy z kolejki ścieżkę **o najniższym koszcie** i ją rozwijamy. Następniki dodajemy **do kolejki**.
- Jeśli wiemy, że istnieje droga długości $<= 420$, to wszystkie dłuższe możemy odrzucać.

Podstawowe definicje:

- **Graf skierowany (digraf)** – para (V, E), gdzie V to dowolny zbiór (zbiór **wierzchołków**) a E to zbiór **krawędzi** (par wierzchołków, $E \subseteq V \times V$).
- **Graf nieskierowany** – jak skierowany, ale w krawędziach nieistotna jest kolejność wierzchołków.
- **Graf ważony** – graf, w którym dodatkowo każdej krawędzi przypisana jest wartość rzeczywista zwana **wagą**: (V, E, w), $w : E \to R$.





Grafy – reprezentacja tablicowa

Ważony graf skierowany $G = (V, E, w)$ najczęściej reprezentujemy **tablicą** W (**macierzą przyległości**, ang. adjacency matrix) taką, że:

$$W[i,j] = \begin{cases} 0 & \text{jeśli } i = j, \\ w(v_i, v_j) & \text{jeśli } (v_i, v_j) \in E \text{ (w G istnieje krawędź } (v_i, v_j)), \\ \infty & \text{jeśli } (v_i, v_j) \notin E. \end{cases}$$

Graf nieskierowany można traktować jako skierowany o symetrycznych połaczeniach.
Tablice dla grafów z poprzedniego slajdu:

	1	2	3	4	5
1	0	4	∞	∞	∞
2	3	0	2	∞	∞
3	∞	∞	0	2	5
4	∞	∞	4	0	7
5	3	∞	∞	6	0

	1	2	3	4	5
1	0	3	∞	∞	2
2	3	0	3	2	5
3	∞	3	0	1	∞
4	∞	2	1	0	7
5	2	5	∞	7	0

Grafy – definicji c.d.

- **Droga** – ciąg wierzchołków taki, że istnieje krawędź z każdego wierzchołka do jego następnika, **droga prosta** – droga, która nie przechodzi dwa razy przez ten sam wierzchołek.
- **Długość drogi** – suma wag krawędzi, z których składa się droga. Dla grafów nieważonych – liczba krawędzi.
- **Cykl** – droga, w której wierzchołek początkowy jest również końcowym, **cykl prosty** – cykl będący drogą prostą.
- **Graf cykliczny/acykliczny** – graf, w którym istnieje/nie istnieje cykl.
- Graf nieskierowany jest **spójny**, gdy między każdymi dwoma wierzchołkami istnieje droga.
- **Drzewo** – graf nieskierowany, acykliczny i spójny.
- **Drzewo rozpinające** graf (ang. spanning tree) – drzewo, które zawiera wszystkie wierzchołki grafu.
- **Minimalne drzewo rozpinające** – drzewo rozpinające o minimalnej sumie wag krawędzi.

Problemy grafowe

Najpopularniejsze problemy:

- Szukanie drogi o minimalnej długości.
- Wyznaczanie minimalnego drzewa rozpinającego:
  - jak najmniej asfaltu, by połączyć określone miasta,
  - jak najmniej kabli w sieci komputerowej, itp.

Podstawowe algorytmy:

- Szukanie dróg:
  - Dijkstry,
  - Floyda (alg. programowania dynamicznego).
- Wyznaczanie minimalnego drzewa rozpinającego:
  - Prima,
  - Kruskala.

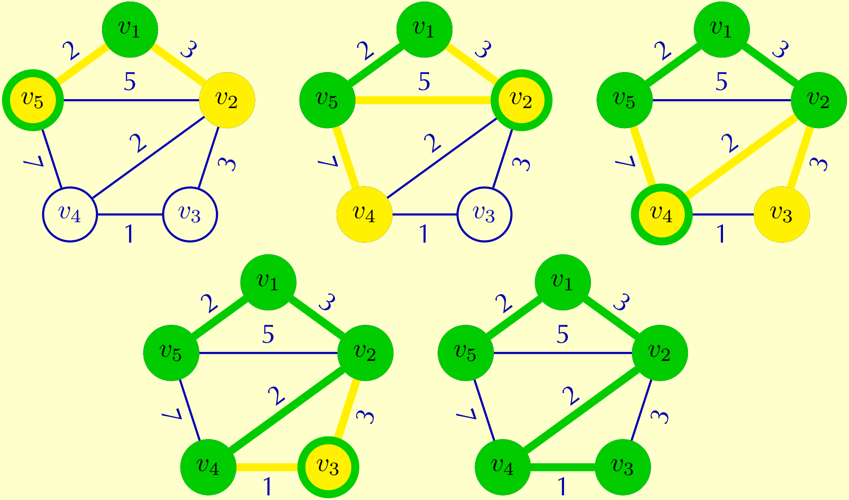
Algorytm Prima

Dane: Graf nieskierowany $G = (V, E, w)$.
Wynik: Minimalne drzewo rozpinające $D = (V, F, w|F)$ grafu G.

- $Y \leftarrow \{v_1\}$
- $F \leftarrow \emptyset$
- Aż do uzyskania drzewa rozpinającego ($Y = V$):
  - wyznaczamy wierzchołek y z $V - Y$ o minimalnej odległości do Y (jednocześnie zapamiętujemy krawędź f, która dała minimalną odległość),
  - $Y \leftarrow Y \cup \{y\}$ tzn. dodajemy wierzchołek y do zbioru Y,
  - $F \leftarrow F \cup \{f\}$ tzn. dodajemy krawędź f do zbioru F,
- Wynik: drzewo $D = (V, F, w|F)$.

Złożoność czasowa: $O(n^2)$ (n to liczba wierzchołków).

Algorytm Prima – przykład



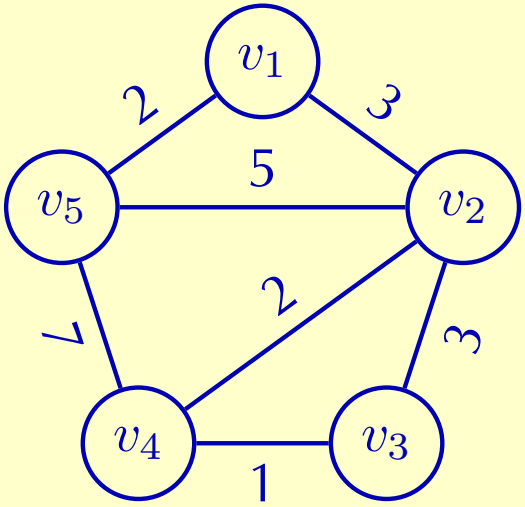
Algorytm Kruskala

Dane: Graf nieskierowany $G = (V, E, w)$.
Wynik: Minimalne drzewo rozpinające $D = (V, F, w|F)$ grafu G.

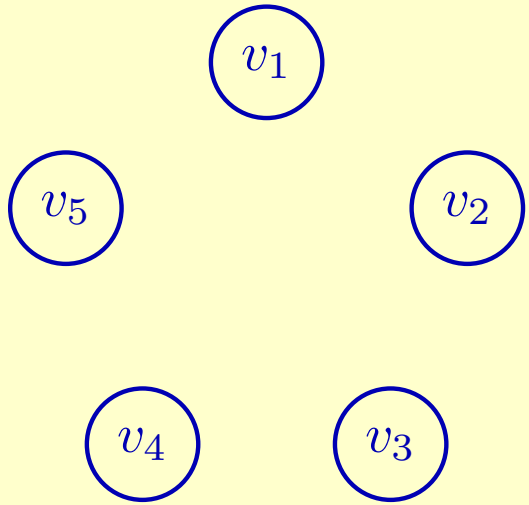
- $V_i \leftarrow \{v_i\}$ – rozłączne jednoelementowe zbiory (dla każdego wierzchołka),
- $F \leftarrow \emptyset$
- Sortujemy krawędzie ze zbioru E niemalejąco względem wag,
- Aż do uzyskania drzewa rozpinającego (wszystkie podzbiory V zostały scalone):
  - bierzemy kolejną krawędź $f \in E$ wg. ustalonego porządku,
  - jeśli krawędź f łączy wierzchołki z różnych podzbiorów to:
    - scalamy podzbiory zawierające wierzchołki,
    - $F \leftarrow F \cup \{f\}$ tzn. dodajemy krawędź f do zbioru F,
- Wynik: drzewo $D = (V, F, w|F)$.

Złożoność czasowa: (n – liczba wierzchołków, m – liczba krawędzi)
Inicjowanie zbiorów: $O(n)$, sortowanie: $O(m \log m)$, pętla w najgorszym przypadku $O(m \log m)$ (każda krawędź).
W najgorszym przypadku $m \approx n^2$, więc
$m \log m = n^2 \log n^2 = n^2 2 \log n$, czyli $O(n^2 \log n)$.

Algorytm Kruskala – przykład



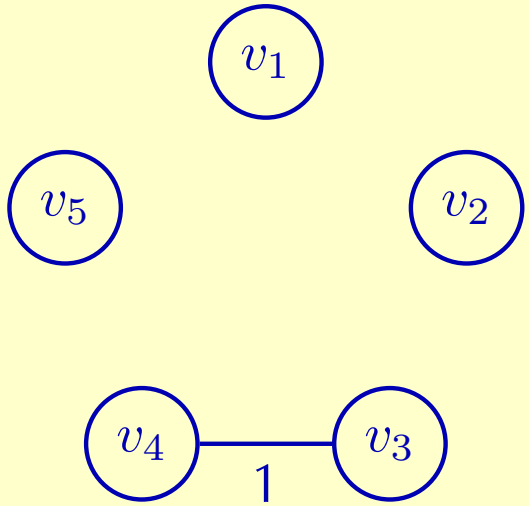
1	(v_3, v_4)	1
2	(v_1, v_5)	2
3	(v_2, v_4)	2
4	(v_2, v_3)	3
5	(v_1, v_2)	3
6	(v_2, v_5)	5
7	(v_4, v_5)	7



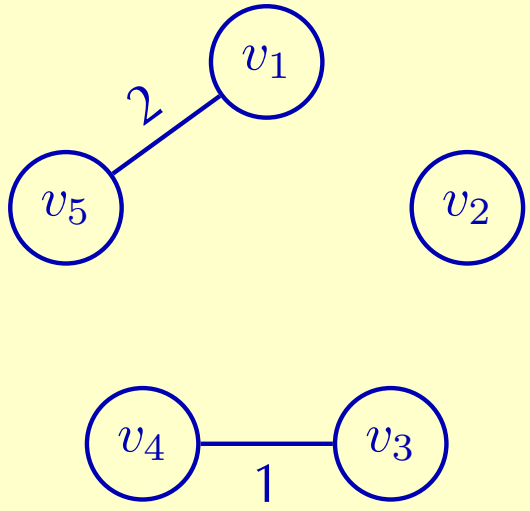
Algorytm Kruskala – przykład c.d.

1	(v_3, v_4)	1
2	(v_1, v_5)	2
3	(v_2, v_4)	2
4	(v_2, v_3)	3
5	(v_1, v_2)	3
6	(v_2, v_5)	5
7	(v_4, v_5)	7

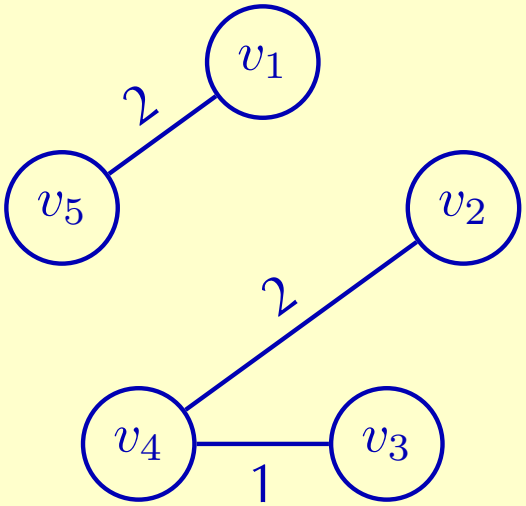
1. Dodajemy (v_3, v_4).



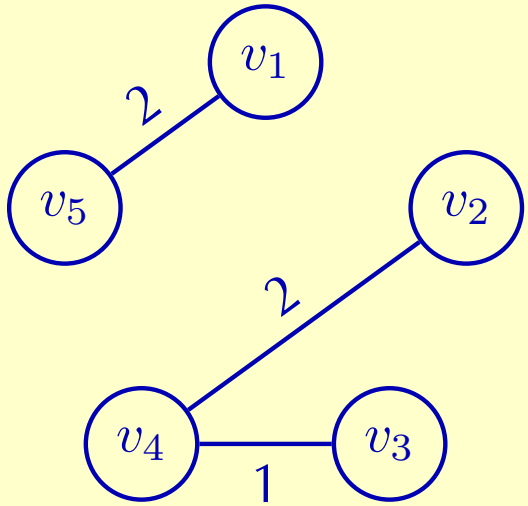
2. Dodajemy (v_1, v_5).



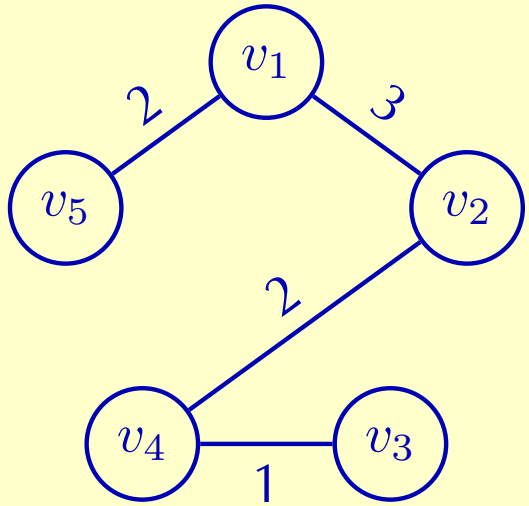
3. Dodajemy (v_2, v_4).



4. Nie dodajemy (v_2, v_3).



5. Dodajemy (v_1, v_2).



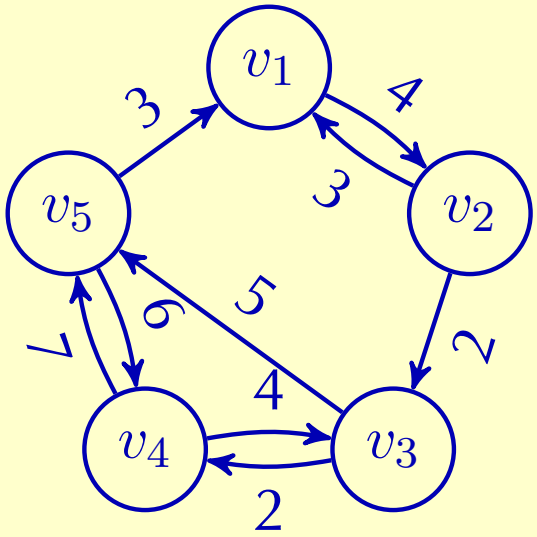
Algorytm Dijkstry

Dane: Graf $G = (V, E, w)$.
Wynik: Najkrótsze (dokładniej: minimalnej długości) drogi z wierzchołka $v \in V$ do wszystkich pozostałych, długości dróg; drzewo rozpinające zawierające najkrótsze drogi z wierzchołka v.

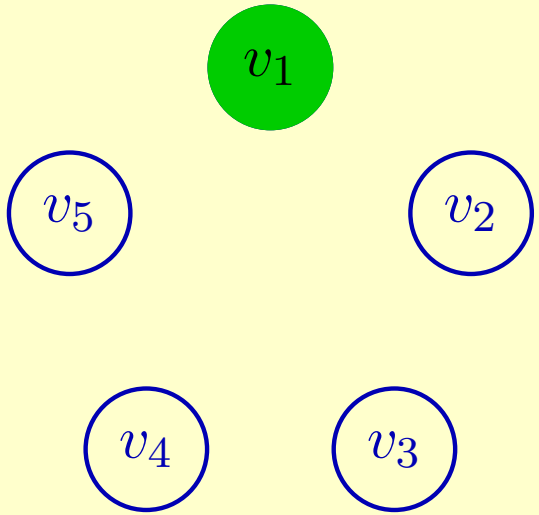
- $Y \leftarrow \{v\}$
- $F \leftarrow \emptyset$
- Tak długo jak $Y \neq V$:
  - wyznaczamy wierzchołek $y \in V - Y$, o minimalnej długości najkrótszej drogi z v poprzez wierzchołki z Y.
  - $Y \leftarrow Y \cup \{y\}$,
  - $F \leftarrow F \cup \{f\}$.

Złożoność czasowa: $O(n^2)$ (n to liczba wierzchołków).

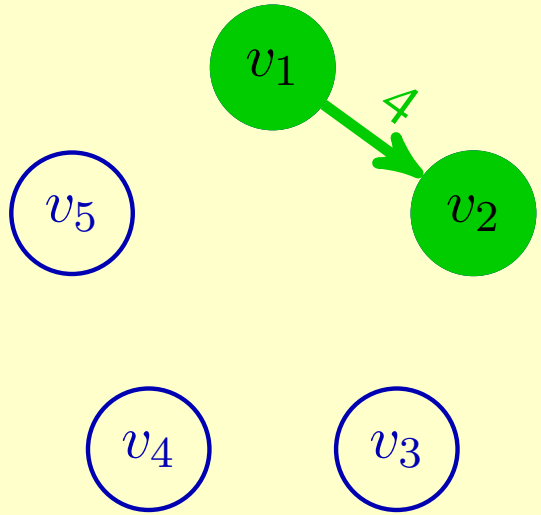
Algorytm Dijkstry – przykład



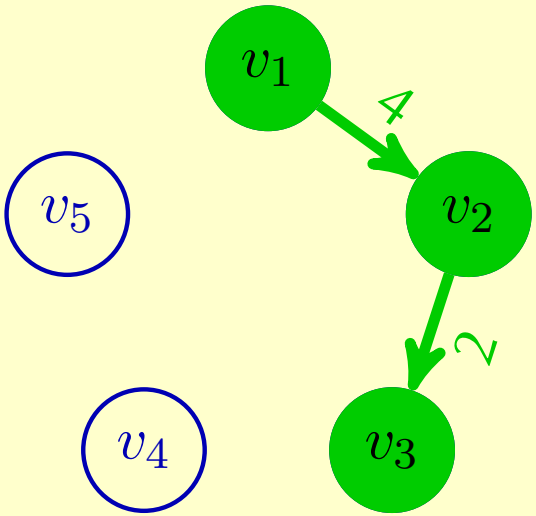
$Y = \{v_1\}$



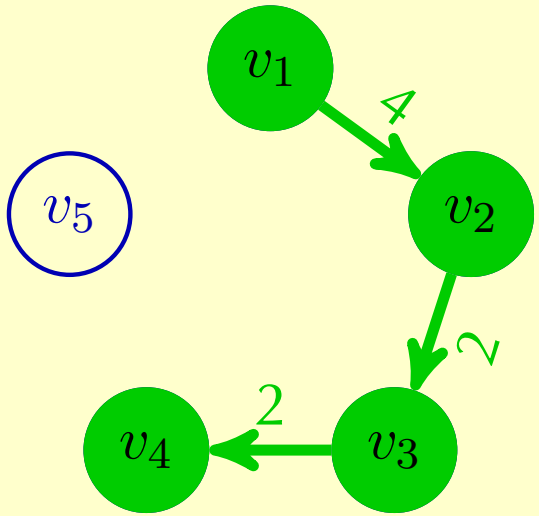
$v_2 \to Y,\ d(v_1, v_2) = 4$



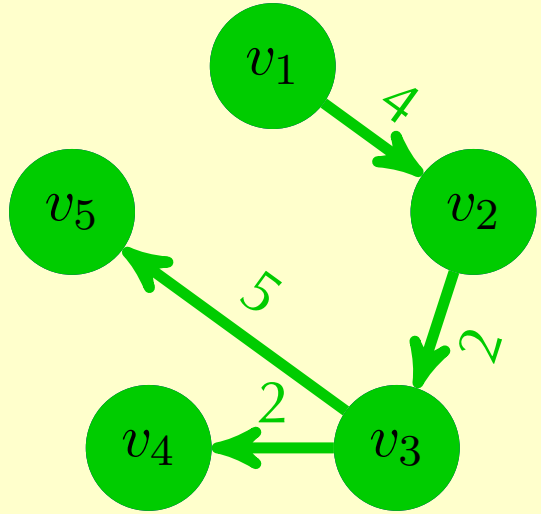
$v_3 \to Y,\ d(v_1, v_3) = 6$



$v_4 \to Y,\ d(v_1, v_4) = 8$



$v_5 \to Y,\ d(v_1, v_5) = 11$



Algorytm Floyda – programowanie dynamiczne

Idea: Najkrótsza droga z v_i do v_j wiodąca przez v_k składa się z najkrótszej drogi z v_i do v_k oraz najkrótszej drogi z v_k do v_j.
Dane: Graf $G = (V, E, w)$.
Wynik: Długości najkrótszych (dokładniej: minimalnej długości) dróg pomiędzy każdą parą wierzchołków.

```
1 Floyd(Array2D W, Array2D D)
2 {
3    D.CopyFrom(W);
4    for (int k = 1; k<=n; k++)
5       for (int i = 1; i<=n; i++)
6          for (int j = 1; j<=n; j++)
7             D[i,j] := minimum(D[i,j], D[i,k] + D[k,j]);
8    return D;
9 }
```

Złożoność czasowa: $O(n^3)$ (n to liczba wierzchołków).
Uwaga na indeksy!

Algorytm Floyda

Ważna własność: W k-tym kroku wartość $D[i,j]$ to długość najkrótszej drogi z v_i do v_j o wierzchołkach pośrednich ze zbioru $\{v_1, \ldots, v_k\}$.

Przykład:

D^0	1	2	3	4	5
1	0	4	∞	∞	∞
2	3	0	2	∞	∞
3	∞	∞	0	2	5
4	∞	∞	4	0	7
5	3	∞	∞	6	0

D^1	1	2	3	4	5
1	0	4	∞	∞	∞
2	3	0	2	∞	∞
3	∞	∞	0	2	5
4	∞	∞	4	0	7
5	3	7	∞	6	0

D^2	1	2	3	4	5
1	0	4	6	∞	∞
2	3	0	2	∞	∞
3	∞	∞	0	2	5
4	∞	∞	4	0	7
5	3	7	9	6	0

D^3	1	2	3	4	5
1	0	4	6	8	11
2	3	0	2	4	7
3	∞	∞	0	2	5
4	∞	∞	4	0	7
5	3	7	9	6	0

D^4	1	2	3	4	5
1	0	4	6	8	11
2	3	0	2	4	7
3	∞	∞	0	2	5
4	∞	∞	4	0	7
5	3	7	9	6	0

D^5	1	2	3	4	5
1	0	4	6	8	11
2	3	0	2	4	7
3	8	12	0	2	5
4	10	14	4	0	7
5	3	7	9	6	0

Szukanie heurystyczne

W przeciwieństwie do szukania na ślepo – **informacja zwrotna**, możliwość użycia funkcji oceny do oszacowania „jakości" bieżącego stanu.

Heurystyki – cenne wskazówki, oparte na doświadczeniu, które dają często, choć nie zawsze, pozytywne skutki. Zwykle to funkcje z przestrzeni stanów do zbioru liczb rzeczywistych

$$h : \mathcal{S} \to R, \tag{4}$$

takie, że

$$\underset{s \in G}{\forall} \; h(s) = 0. \tag{5}$$

Dla wygody, dla węzła n, $h(n) \overset{\text{ozn}}{=} h(s_n)$.

Proste **wykorzystanie heurystyk w szukaniu**:

- rozwijaj węzły drzewa w kolejności rosnącej heurystyki,
- kolejka priorytetowa.

Metoda „najpierw najlepszy”

- Wykorzystanie funkcji heurystycznych – nowa rodzina metod „**najpierw najlepszy**” (ang. *best-first search*).
- Podejście **zachłanne** (ang. *greedy best-first search*)

$$f(n) = h(s_n). \tag{6}$$

- Przykład dróg do Bukaresztu (str. 27) – dobra heurystyka: **jak daleko do celu w linii prostej**.

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Metoda A*

- Cel: **minimalizacja łącznego estymowanego kosztu** rozwiązania:

$$f(n) = g(n) + h(n). \tag{7}$$

- **Własność**: A* jest optymalna, gdy heurystyka h nigdy nie przeszacowuje rzeczywistego kosztu. Mówimy wtedy, że h jest **dopuszczalna** (ang. *admissible*).
  Dowód: Niech n będzie nieoptymalnym węzłem ze stanem końcowym oraz C^* kosztem optymalnego rozwiązania. Oczywiście $f(n) = g(n) > C^*$. Jeśli n_0 jest węzłem z optymalną ścieżką, to $f(n_0) = g(n_0) + h(n_0) \leq C^* < f(n)$, bo h (a więc również f) nie przeszacowuje kosztu. Stąd węzeł n nie zostanie nigdy rozwinięty, a rozwiązaniem będzie węzeł optymalny. □

Heurystyki

- Heurystyka jest **monotoniczna** (ang. *monotonic, consistent*) jeśli dla każdego węzła n oraz jego węzła potomnego n' zachodzi

$$h(n) \leq c(n, a, n') + h(n'), \tag{8}$$

  gdzie $c(n, a, n')$ jest kosztem działania a skutkującego zmianą stanu z s_n do $s_{n'}$.
- Jeśli h jest monotoniczna, to f jest **niemalejąca na danej ścieżce**.
- Monotoniczność heurystyki gwarantuje **optymalność** szukania **z odrzucaniem odwiedzonych** stanów.

Metoda A*

- Rozwija wszystkie węzły n, dla których $f(n) < C^*$.
- Nie rozwija węzłów n, dla których $f(n) > C^*$.
- Jest **zupełna**, jeśli zbiór węzłów n, dla których $f(n) < C^*$ jest skończony, w szczególności, jeśli koszt pojedynczego działania jest zawsze $\geq \epsilon > 0$ oraz b jest skończone.
- Jest **optymalnie efektywna** tzn. żaden inny algorytm optymalny nie rozwija mniej węzłów (chyba, że węzłów, dla których $f(n) = C^*$).
  Dowód: Algorytm, który nie rozwinie wszystkich węzłów o $f(n) < C^*$ nie jest optymalny. □

IDA*– Iterative Deepening A*

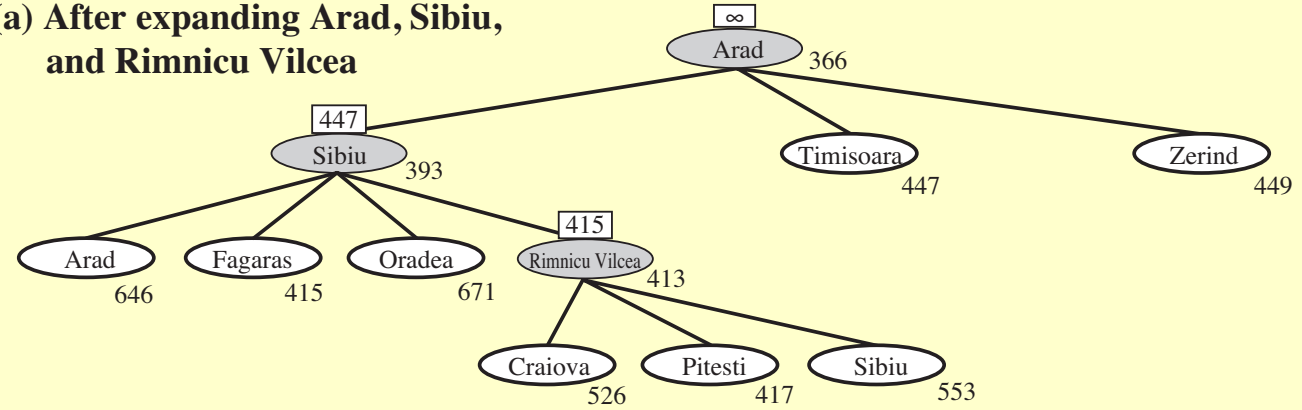
- Cel: ograniczenie zapotrzebowania na **pamięć**.
- IDA*- zwykły ID dla A*, ale **z limitem kosztów** $f = g + h$ **zamiast głębokości**.
- Próg f-kosztów – najmniejszy koszt węzła, przekraczający poprzedni próg odcięcia.

RBFS – recursive best-first search

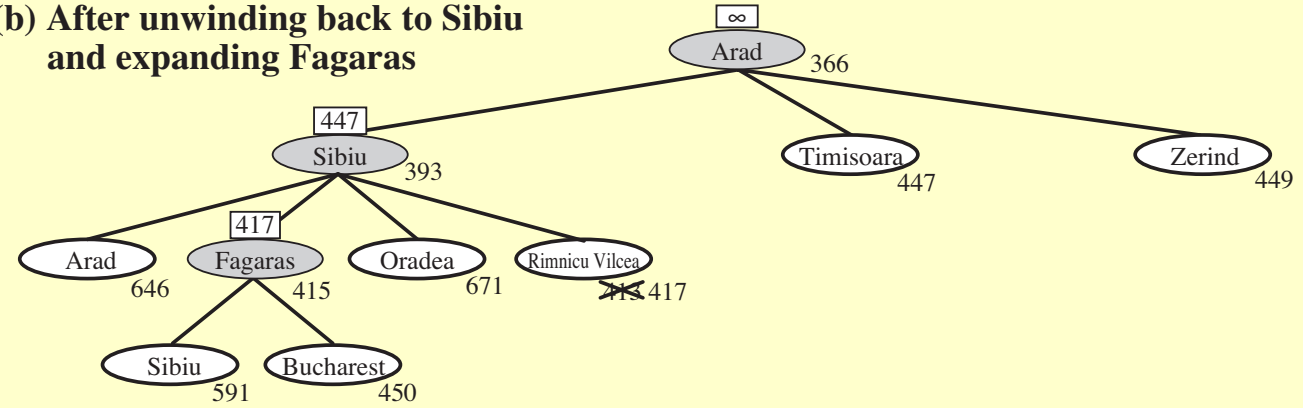
Prototype: ***RecursiveBestFS(P, node, fLimit)***
Input: Problem P, current node, f limit.
Output: A solution or failure and a new f-cost.

1. **if** node reached the goal **then return** the solution
2. successors $\leftarrow$ subnodes of the node
3. **if** successors is empty **then return** failure, ∞
4. **for each** s in successors **do** $f[s] \leftarrow \max(g(s) + h(s), f[node])$
5. **repeat**
   1. best $\leftarrow \arg\min_{s \in \text{successors}} f(s)$
   2. **if** f[best] $>$ fLimit **then return** failure, f[best]
   3. fAlt $\leftarrow$ second best $f(s)$ among successors
   4. result, f[best] $\leftarrow$ RecursiveBestFS(P, best, min(fLimit, fAlt))
   5. **if** result $\neq$ failure **then return** result

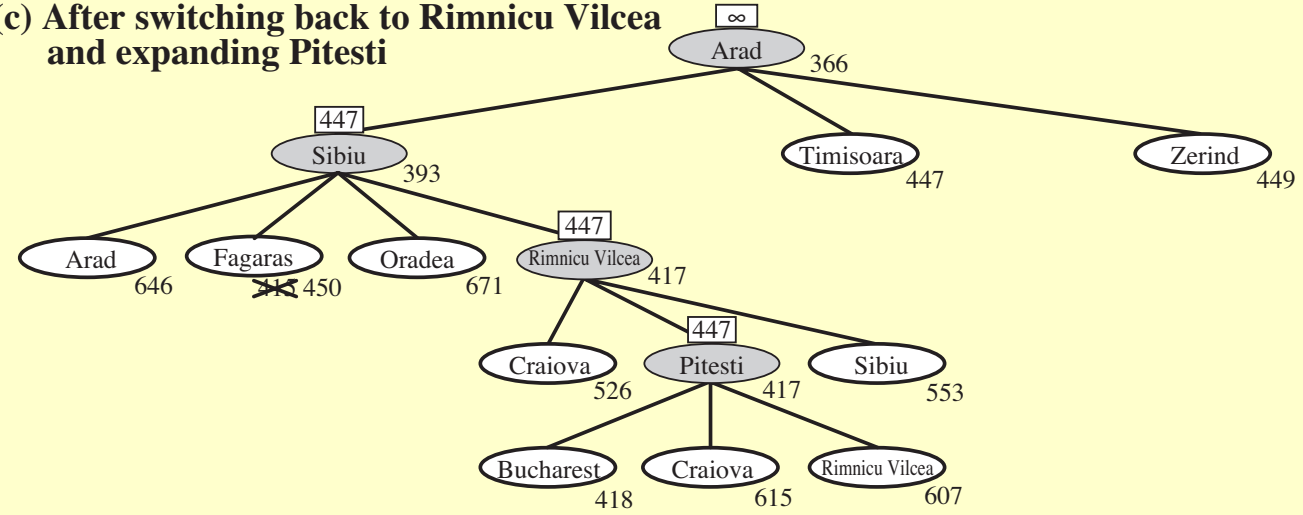
(a) After expanding Arad, Sibiu, and Rimnicu Vilcea



(b) After unwinding back to Sibiu and expanding Fagaras



(c) After switching back to Rimnicu Vilcea and expanding Pitesti



RBFS

- Niebezpieczeństwo częstych „**zmian zdania**” (zmian ścieżki szukania).
- **Złożoność czasowa** trudna do określenia.
- **Optymalny**, jeśli heurystyka h jest dopuszczalna.
- **Złożoność pamięciowa liniowa** względem głębokości najgłębszego rozwiązania optymalnego.
- Jak w IDA* **niebezpieczeństwo wielokrotnego badania** tych samych stanów.
- **Nadmierna oszczędność pamięci**.
- Algorytmy wykorzystujące dostępną pamięć: **MA*, SMA***.
  - SMA*– rozwija węzły aż do zapełnienia pamięci – wówczas **wyrzuca z pamięci najgorszy węzeł** (największe $f(n)$), zapamiętując jego jakość w węźle-rodzicu.
  - Na wypadek równych $f(n)$ – usuwany **najstarszy**, rozwijany **najmłodszy**.

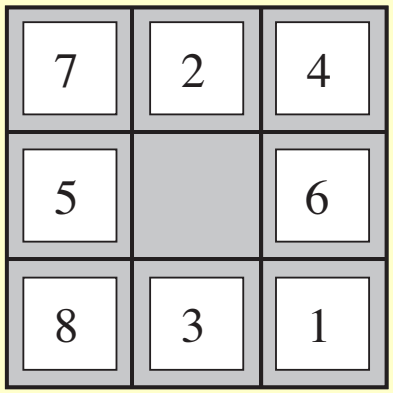
Szukanie wiązką (ang. *beam search*)

- Cel: **ograniczenie ilości pamięci**.
- Jak w szukaniu wszerz, ale po każdej iteracji ograniczanie zajętości pamięci do k **stanów**.
- k to **szerokość wiązki**.
- Porzucanie części stanów skutkuje brakiem **zupełności** i **optymalności**.
- Atrakcyjna złożoność **pamięciowa**.
- Złożoność **czasowa** trudna do określenia – w szczególności brak gwarancji zakończenia procesu sukcesem.
- Niebezpieczeństwo **degeneracji wiązki** – różne techniki dbania o **różnorodność wiązki**.

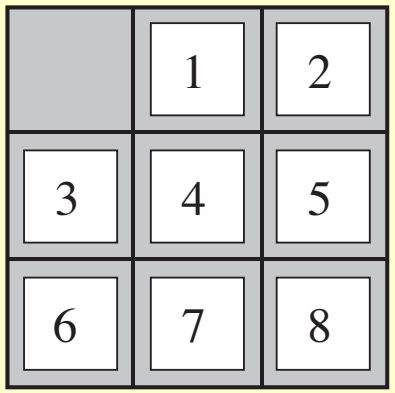
Heurystyki

Przykłady dla problemu 8-ki:

- h_1 – **liczba elementów** nie na swoim miejscu,
- h_2 – suma **odległości Manhattan** elementów od swoich pozycji docelowych.
- Dla przykładu obok: $h_1(n) = 8$, $h_2(n) = 18$.



Start State



Goal State

- h_1 i h_2 są **dopuszczalne**.
- Miara jakości heurystyki: **efektywny czynnik rozgałęzień** b^* – liczba rzeczywista spełniająca:

$$N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d, \tag{9}$$

gdzie N to liczba węzłów w procesie szukania, a d to głębokość rozwiązania.

- Im b^* jest bliższe 1, tym lepsza heurystyka.

Heurystyki

- h_2 **dominuje** nad h_1, tzn. dla dowolnego węzła n, $h_2(n) \geq h_1(n)$.
- Konsekwencja: każdy węzeł, rozwinięty przez h_2 będzie również rozwinięty przez h_1 (w A*, bo to metoda optymalna).
- Wniosek: im wyższe wartości heurystyki, tym lepiej!
- Zadania: porównać liczby węzłów rozwijanych heurystykami h_1 i h_2.

Poszukiwanie dopuszczalnych heurystyk

- **Problem uproszczony** (ang. *relaxed problem*) – problem ze zredukowaną liczbą ograniczeń.
- Ważna własność: **koszt optymalnego rozwiązania problemu uproszczonego jest dopuszczalną heurystyką problemu wyjściowego.**
  Dowód: Rozwiązanie problemu wyjściowego jest też rozwiązaniem problemu uproszczonego więc jest co najmniej tak kosztowne jak jego rozwiązanie optymalne. □

Poszukiwanie dopuszczalnych heurystyk

- Wyrażenie ograniczeń w języku formalnym i ich **upraszczanie**:

  Element można przesunąć:

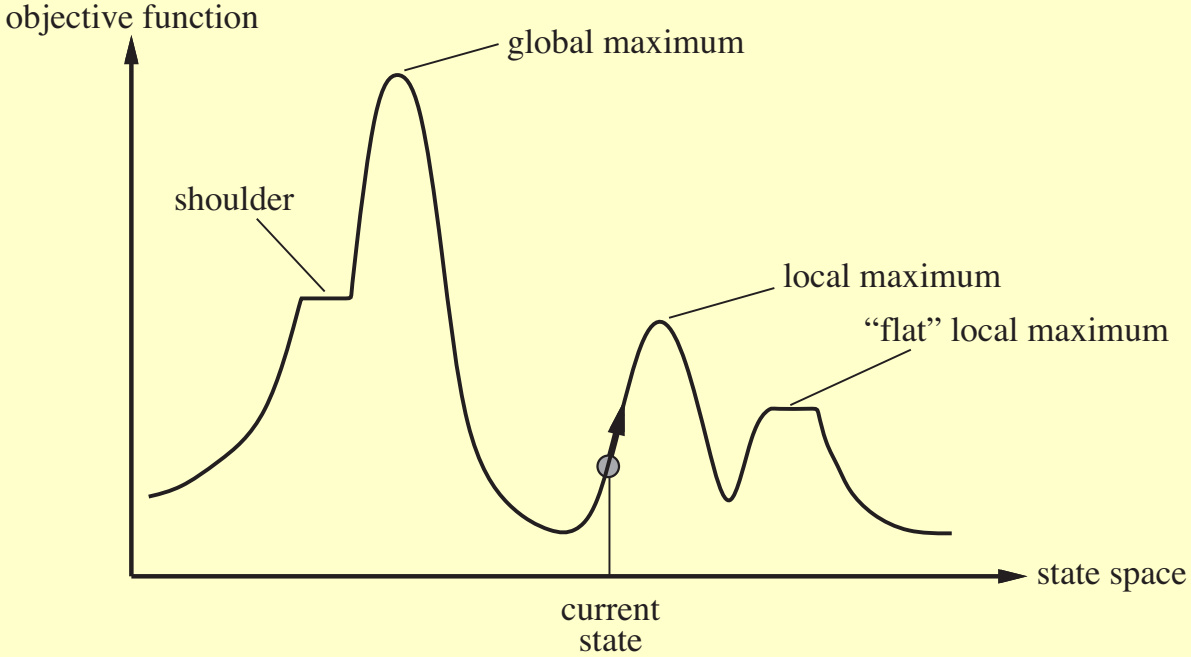
  - *poziomo lub pionowo o jedno pole,*
  - *pod warunkiem, że pole docelowe jest puste.*

- **Maksimum** heurystyk dopuszczalnych **jest heurystyką dopuszczalną**.
- Koszt rozwiązania podproblemu **jest heurystyką dopuszczalną**.
- **Bazy wzorców** – zapamiętanie rozwiązań wszystkich podproblemów i odnajdywanie kosztów dla konkretnych konfiguracji zgodnych z danym stanem.
  - dla 15-tki i maksimum kosztów dla 4 wybranych kostek – 1000 razy szybciej niż Manhattan.
- **Bazy rozłącznych wzorców** – liczymy tylko ruchy dla wybranych elementów i sumujemy dla różnych (rozłącznych) wzorców.
  - dla 15-tki i wzorców dla 4 kostek – 10000 razy szybciej niż Manhattan.

Lokalne szukanie

- Nie zawsze istotna jest **ścieżka dojścia** do rozwiązania (drzewa niepotrzebne).
- **Lokalne szukanie** operuje **tylko bieżącym stanem** zamiast ścieżki.
- Metody szukania mogą również rozwiązywać problemy optymalizacyjne (zadana jest funkcja do minimalizacji).
- Założenie „ciągłości” problemu w sensie, że podobne stany są podobnie oceniane.

- Optima globalne i lokalne.
- Ramię, płaskie maksimum, grzbiet, plateaux.



Metoda wspinaczki (ang. *hill climbing*)

- Zachłanne podążanie zawsze w kierunku rosnących wartości maksymalizowanej funkcji.
- Analizowane tylko bezpośrednie sąsiedztwo bieżącego stanu.

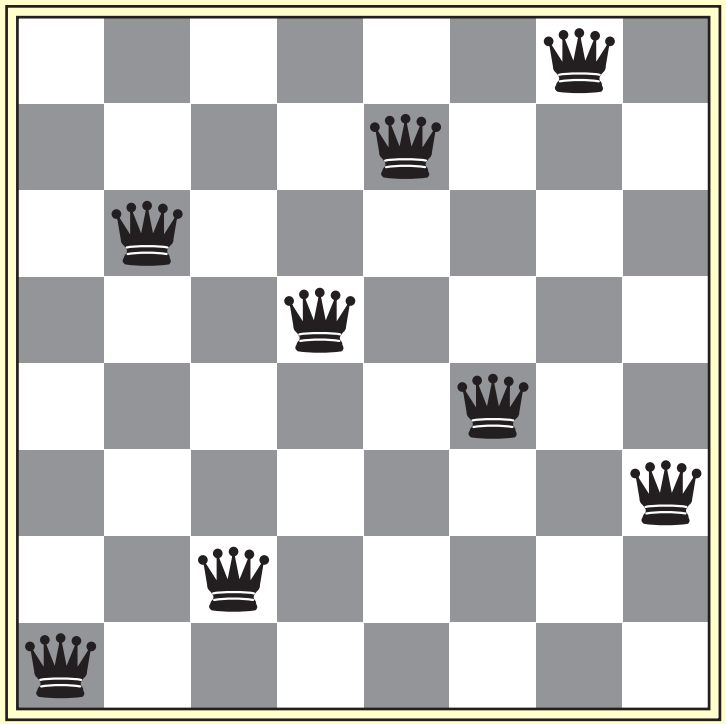
Prototype: ***HillClimbing(P)***
Input: Problem P, w tym funkcja maksymalizowana F.
Output: Stan lokalnego maksimum.

1. current $\leftarrow$ initial state
2. **repeat**
    1. neighbor $\leftarrow$ the highest-valued successor of current
    2. **if** F(neighbor) $\leq$ F(current) **then return** current
    3. current $\leftarrow$ neighbor

Przykład: problem 8 hetmanów

- Sformułowanie „pełnego stanu” (wszystkie hetmany na szachownicy, każdy ma jedną kolumnę).
- Funkcja następnika: **jeden hetman przestawiony** w inne miejsce.
- Heurystyka: **liczba szachujących się par** (do minimalizacji).

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♛	13	16	13	16
♛	14	17	15	♛	14	16	16
17	♛	16	18	15	♛	15	♛
18	14	♛	15	15	14	♛	16
14	14	13	17	12	14	12	18



Problemy wspinaczki

- Lokalne maksima/minima, grzbiety, plateaux...
- Przyzwolenie na **odejście w bok** z minimum lokalnego – przyzwolenie na 100 kolejnych ruchów w bok podnosi współczynnik sukcesu z 14% do 94%.
- **Stochastyczna metoda wspinaczki** – losuje kolejny stan spośród tych idących w górę.
- **Wspinaczka pierwszego wyboru** (ang. *first-choice hill climbing*) – losuje spośród następników aż wylosuje lepszy stan niż bieżący.
- Problem niezupełności – **losowe restarty**.

Symulowane wyżarzanie

Symulowanie fizycznych procesów wyżarzania (ang. *simulated annealing*).

Prototype: ***SimulatedAnnealing(P,H)***

Input: Problem P w tym funkcja maksymalizowana F, harmonogram schładzania H.

Output: Stan lokalnego minimum.

1. current $\leftarrow$ initial state
2. **for** $t = 1$ **to** ∞ **do**
    1. $T \leftarrow H(t)$
    2. **if** $T = 0$ **then return** current
    3. next $\leftarrow$ random successor of current
    4. $\Delta E \leftarrow$ F(next) - F(current)
    5. **if** $\Delta E > 0$ **then** current $\leftarrow$ next
       else current $\leftarrow$ next with probability $e^{\Delta E/T}$

Lokalne szukanie wiązką

- Lokalne szukanie wiązką (ang. *local beam search*) działa jak szukanie wiązką, tyle, że nie utrzymuje pełnych ścieżek, a **same stany bieżące**.
- Ograniczenie pamięci do k **stanów**.
- Start: k **losowo** wybranych stanów.
- Niebezpieczeństwo **degeneracji** wiązki – wszystkie stany z jednego rodzica.
- Podejście **stochastyczne** – wybór wiązki losowy, prawdopodobieństwa proporcjonalne do wartości maksymalizowanej funkcji.

Algorytmy genetyczne

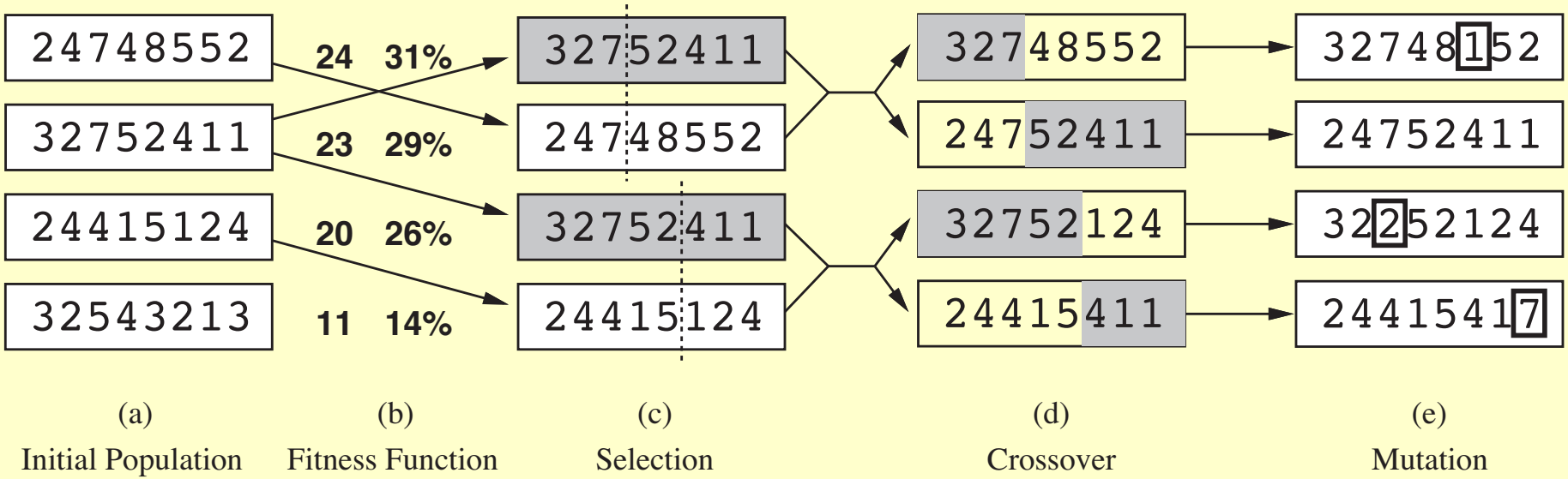
Inspiracje biologiczne

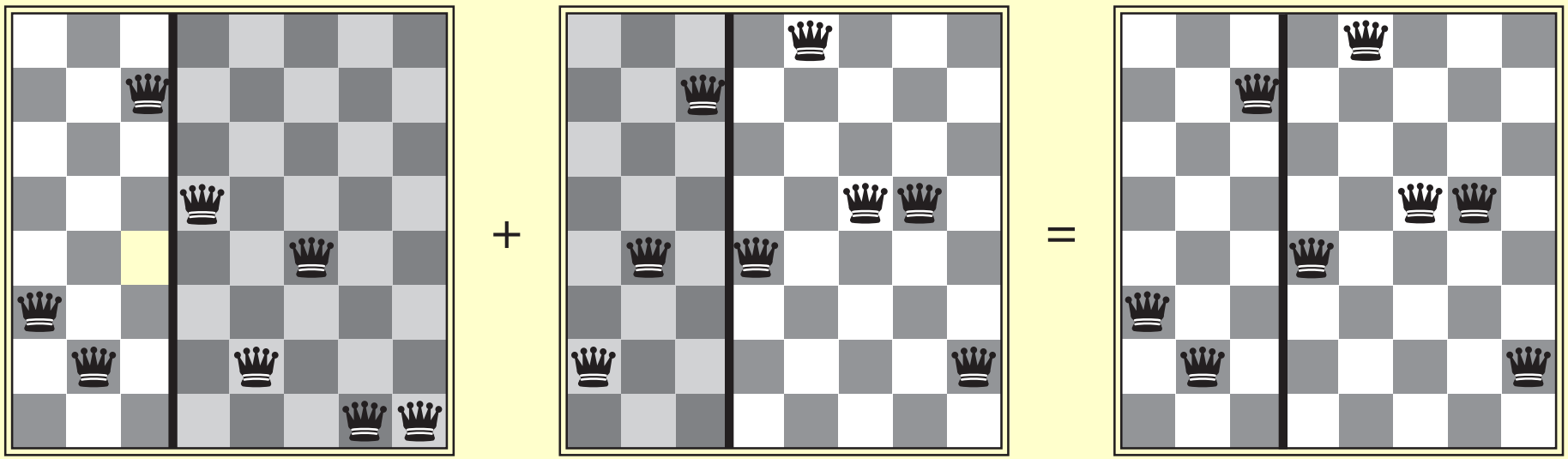
- kodowanie stanów (**osobników**) przez **chromosomy**,
- **naturalna selekcja** (funkcja **przystosowania**),
- **krzyżowanie**,
- **mutacje**.

Schemat algorytmu:

- Ustalamy **zasady kodowania** osobników.
- Wybieramy **losowo** pierwszą populację k osobników.
- Dla kolejnych populacji:
  - mierzymy **przystosowanie** (ang. *fitness*) osobników,
  - losujemy **pary osobników** do krzyżowania,
  - losujemy **punkt krzyżowania** (ang. *crossover point*),
  - krzyżujemy,
  - pozwalamy na **mutacje** z określonym p-stwem,
  - nowe osobniki to nowa populacja.

Algorytmy genetyczne – 8 hetmanów





Szukanie online

Szukanie na boku (**offline**) spisuje się dobrze w środowiskach **statycznych** i **deterministycznych**.

Środowiska **dynamiczne**, **semidynamiczne** (kary za opieszałość) i **stochastyczne** wymagają reakcji w trakcie szukania (**online**).

Szukanie online

- Wynik akcji poznajemy **po jej podjęciu**.
- **Przeskoki** do odległych punktów w przestrzeni **nie są możliwe** – metody typu „najpierw najlepszy” czy A* nie są odpowiednie.
- Właściwe są metody szukające w sposób „**zlokalizowany**” – szukanie w głąb, metoda wspinaczki itp.

Szukanie w głąb w wersji online

Potrzebujemy dodatkowo:

- **rejestracji skutków** naszych akcji (przy założeniu deterministyczności),
- umiejętności **rozróżniania odwiedzonych i nie odwiedzonych** miejsc,
- rejestracji **możliwości powrotów**.

Rekurencja może pomóc w realizacji tych zadań. Wersje nierekurencyjne są zwykle trudniejsze.

Szukanie online

Random walk

- wybieramy losowo dostępne akcje,
- preferencje dla akcji wcześniej nie próbowanych,
- dość powolny proces,
- niebezpieczeństwo pułapek odwodzących od celu.

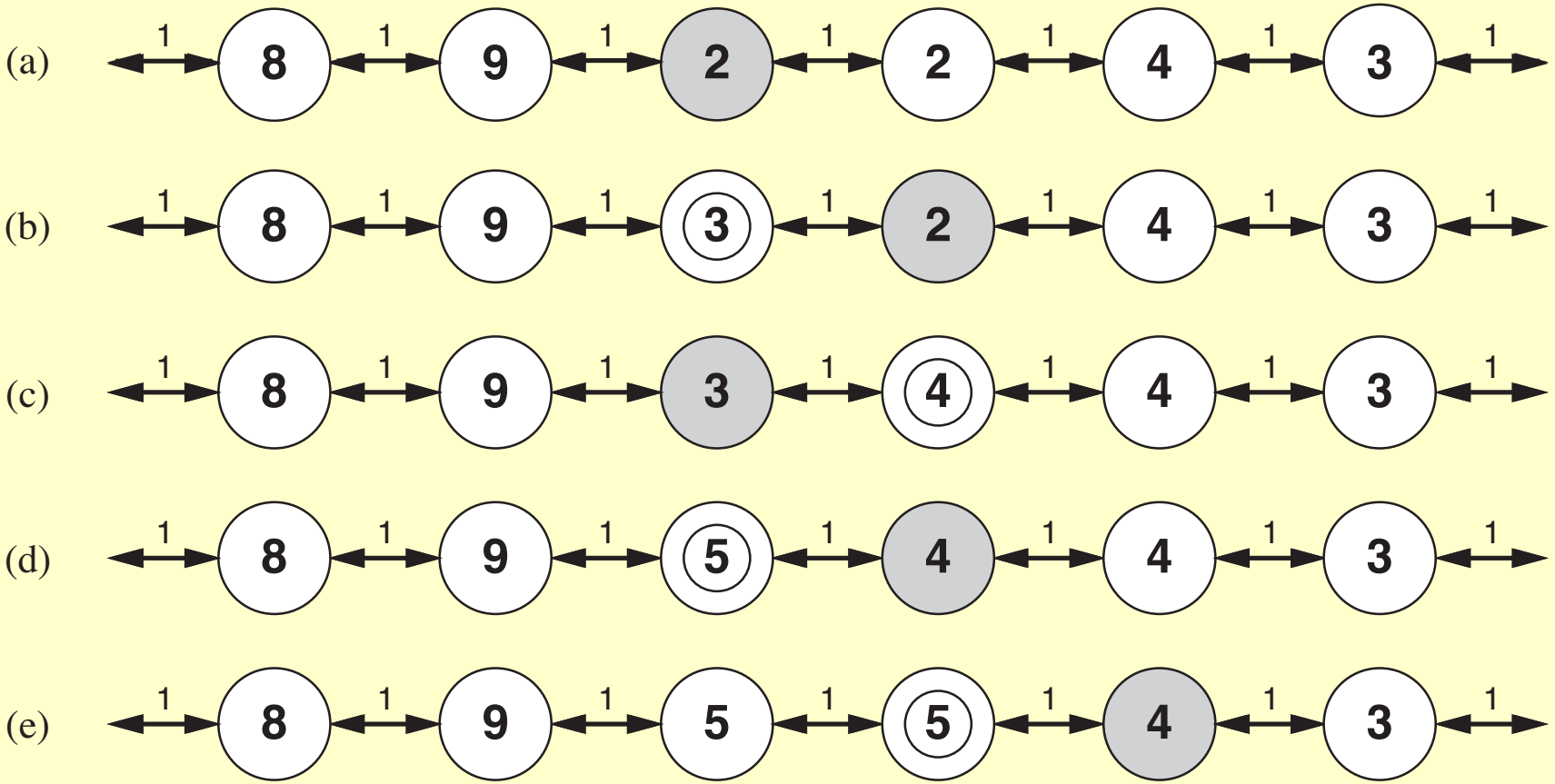
Metoda wspinaczki. Jak uciekać z minimów lokalnych?

- Przeskoki losowe nie są możliwe.
- Losowe kierunki w minimach lokalnych.
- Wspinaczka z pamięcią – LRTA*.

LRTA*– **learning real-time A***

- używamy **heurystyki** i **doświadczenia** – jak w A*, koszt jest sumą, tego co wiemy na pewno i oceny heurystyką,
- wybieramy akcję o najlepszym przewidywanym skutku,
- optymistycznie oceniamy nie próbowane akcje,
- aktualizujemy na bieżąco wiedzę o stanie, który opuszczamy.

LRTA* - przykład



Problemy z ograniczeniami

Problemy z **ograniczeniami**, **więzami** (ang. *Constraint satisfaction problems*)

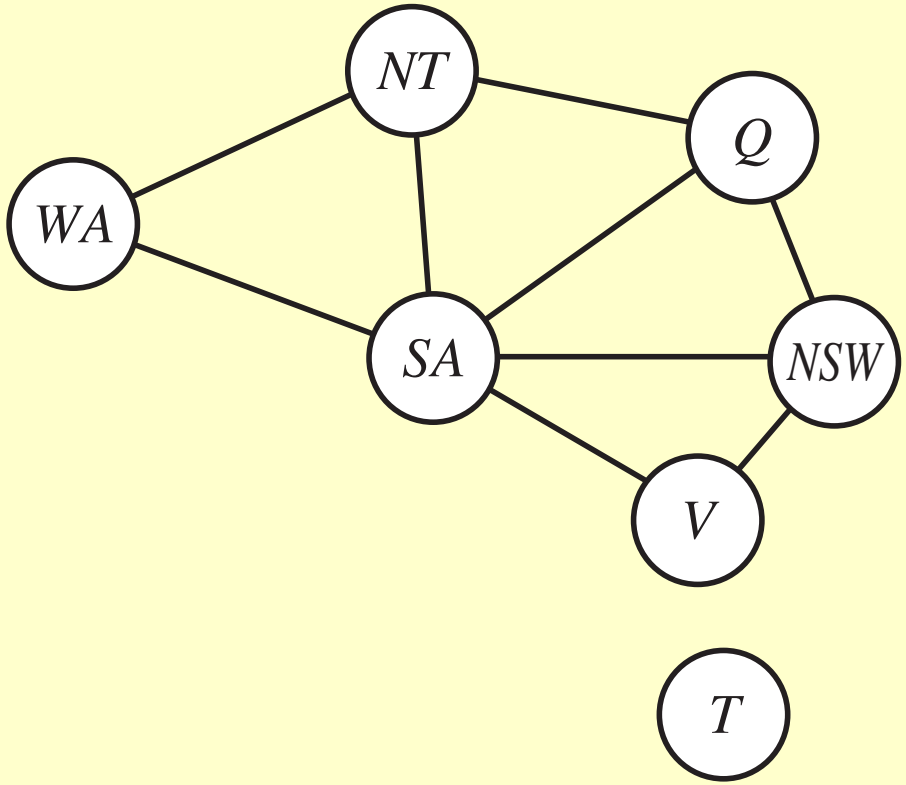
Wiedza o **wewnętrznej strukturze problemu** może pomóc w jego rozwiązywaniu.

Problem z ograniczeniami definiuje stany oraz cele przy użyciu **zbioru zmiennych** $X_1, X_2, \ldots, X_n$ oraz **zbioru ograniczeń** $C_1, C_2, \ldots, C_m$:

- stan – **przypisanie wartości podzbiorowi zmiennych**, np. $\{X_i = v_i, X_j = v_j, \ldots\}$,
- **stan zgodny** – stan nie naruszający żadnego ograniczenia,
- **stan pełny** – określający wartości wszystkich zmiennych,
- **cel** – stan pełny i zgodny,
- dodatkowo, możliwa **maksymalizacja** funkcji celu.

Problem kolorowania mapy

- Mapa i graf ograniczeń:



- Stany, ograniczenia, cel, koszt ścieżki...
- Naturalne podejście inkrementacyjne.

Problemy z ograniczeniami

- Dziedziny skończone i nieskończone, dyskretne i ciągłe.
- Języki dla ograniczeń.
- Ograniczenia liniowe, programowanie liniowe.
- Ograniczenia nieliniowe – znacznie trudniejsze.
- Liczba zmiennych objętych ograniczeniem: 1, 2, więcej (wszystkie różne, N spośród M, kryptarytmy, sudoku).
- Kryptarytmy (ten pierwszy z 1924 r.):

$$\begin{array}{r} \mathrm{S\ E\ N\ D} \\ +\ \mathrm{M\ O\ R\ E} \\ \hline \mathrm{M\ O\ N\ E\ Y} \end{array} \qquad \begin{array}{r} \mathrm{U\ S\ A} \\ +\ \mathrm{U\ S\ S\ R} \\ \hline \mathrm{P\ E\ A\ C\ E} \end{array} \qquad \begin{array}{r} \mathrm{S\ E\ R\ C\ E} \\ +\ \mathrm{S\ E\ R\ C\ E} \\ \hline \mathrm{M\ I\ Ł\ O\ Ś\ Ć} \end{array}$$

- Zadania typu: pięć osób pochodzących z pięciu różnych krajów interesuje się pięcioma różnymi dyscyplinami sportu i uwielbia pięć różnych potraw itd.
- Ograniczenia–preferencje, np. przy układaniu planów zajęć.

Szukanie z nawrotami (ang. *backtracking search*)

- Problemy z ograniczeniami są **przemienne** – kolejność wiązania zmiennych jest bez znaczenia.
- Podejście inkrementacyjne.
- Idea: w danym węźle drzewa (albo i na danej głębokości) wiążemy **jedną zmienną**.
- Szukanie z nawrotami to szukanie **w głąb** w takim drzewie, z kontrolą ograniczeń.
- Szukanie „**na ślepo**” – mało efektywne.
- Możliwość usprawnień **heurystycznych**:
  - kolejność zmiennych i ich wartości,
  - konsekwencje przypisań dla innych zmiennych,
  - unikanie takich samych niepowodzeń.

Kolejność zmiennych i wartości

- Przykład Australii – po przypisaniu WA=red oraz NT=green lepiej zająć się SA niż Q.
- Heurystyka **MRV** (ang. *minimum remaining values*) – wybieraj zmienną **z minimalną liczbą dozwolonych wartości**.
- **Stopień zmiennej** – liczba ograniczeń, w których zmienna występuje (z innymi, nie przypisanymi jeszcze zmiennymi) – najpierw zmienne o maksymalnym stopniu.
- Heurystyka **najmniej ograniczającej wartości** (ang. *least-constraining value*) – najpierw wartości, które eliminują najmniej możliwości innym zmiennym.

Problem	Nawroty	N.+MRV	FC	FC+MRV	MC
USA	(>1M)	(>1M)	2K	60	64
n-hetmanów	(>40M)	13,5M	(>40M)	817K	4K
Zebra	3859K	1K	35K	0,5K	2K

Analiza ograniczeń

- Testowanie możliwości eliminacji z wyprzedzeniem (ang. *forward checking*).

	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA=red*	(R)	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	(R)	B	(G)	R B	R G B	B	R G B
After *V=blue*	(R)	B	(G)	R	(B)		R G B

- Niektóre gałęzie można wcześnie odciąć
- Zależności wyższych rzędów
  - *WA=R, Q=G* pozostawia *NT=SA=B,*
  - parami różne.
- Niesprzeczność połączeń (ang. *arc consistency*) – trzeba uważać na złożoność!
- Kontrola niesprzeczności na początku procesu i w trakcie.

Lokalne szukanie z ograniczeniami

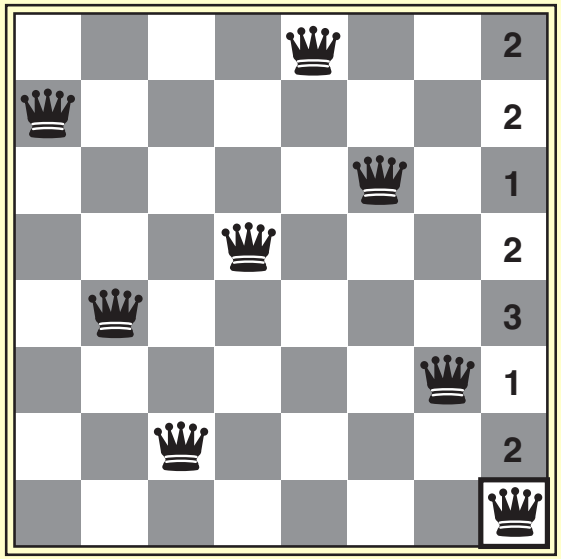
- **Pełna reprezentacja** i **zmiany** by spełniać ograniczenia
  - zmiana wartości **pojedynczych** zmiennych,
  - zamiana wartości **dwóch** zmiennych, itp.
- Naturalna heurystyka: wybór **wartości o jak najmniejszej liczbie konfliktów** (ang. *min-conflicts*).

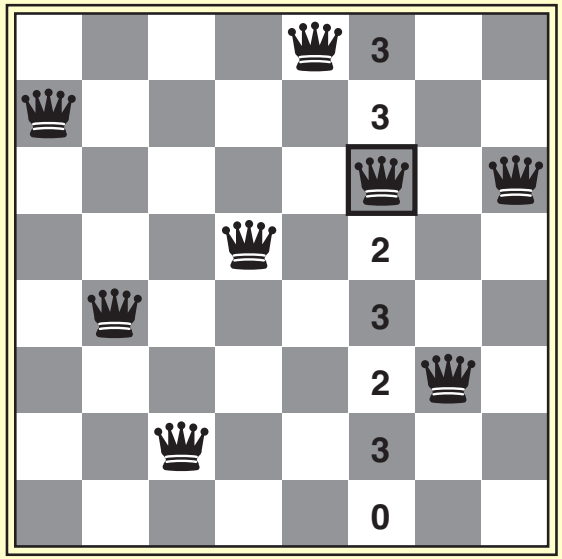
Prototype: ***MinConflicts(CSP, H, N)***
Input: Problem CSP, miara min-conflicts H, liczba N iteracji.
Output: Stan lokalnego minimum.

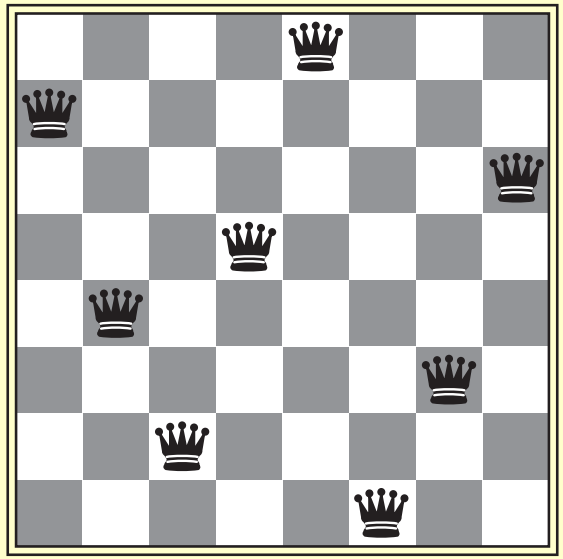
1. stan $\leftarrow$ (losowy pełny) stan początkowy
2. **for** $t = 1$ **to** N **do**
   1. **if** stan jest rozwiązaniem **then return** stan
   2. X $\leftarrow$ wylosowana zmienna z konfliktem
   3. Z $\leftarrow$ zmiana X minimalizująca miarę H
   4. stan $\leftarrow$ stan po zmianie Z
3. **return** porażka

Lokalne szukanie z ograniczeniami

- Dla hetmanów niemal niezależność od rozmiaru – nawet **milion hetmanów w 50 krokach**.
- Planowanie tygodnia obserwacji teleskopu Hubble'a – **redukcja czasu obliczeń z 3 tygodni do 10 minut**.







- Odpowiednie **w planowaniu**, gdy dochodzą drobne zmiany.

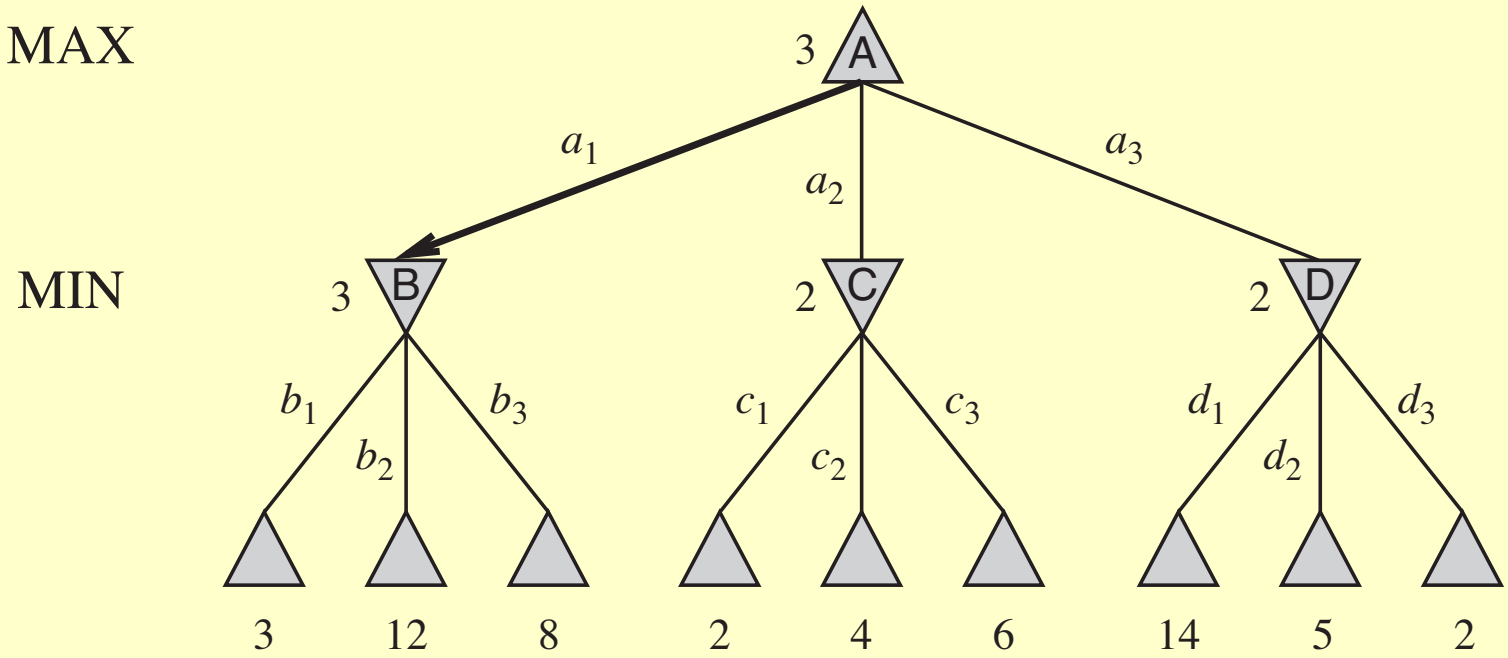
Gry - szukanie z oponentem

- Środowiska **wieloagentowe ze współzawodnictwem**.
- Szukanie **z przeciwnikiem**/oponentem/adwersarzem (ang. *adversarial search*).
- **Deterministyczne, w pełni obserwowalne środowisko**, w którym agenty/agenci działają naprzemiennie.
- Gry **z zerowym bilansem** - ktoś wygrywa, ktoś przegrywa.
- **Definicja** problemu gry z przeciwnikiem:
  - **stan początkowy**,
  - funkcja **następnika**,
  - **test zakończenia** gry,
  - **funkcja użyteczności**.
- Optymalna strategia – wartość minimax (MM):

$$MM(n) = \begin{cases} \text{użyteczność}(n) & \text{jeśli } n \text{ jest końcowy} \\ \max_{s \in \text{Następniki}(n)} MM(s) & \text{jeśli } n \text{ jest typu MAX} \\ \min_{s \in \text{Następniki}(n)} MM(s) & \text{jeśli } n \text{ jest typu MIN} \end{cases}$$

Minimax

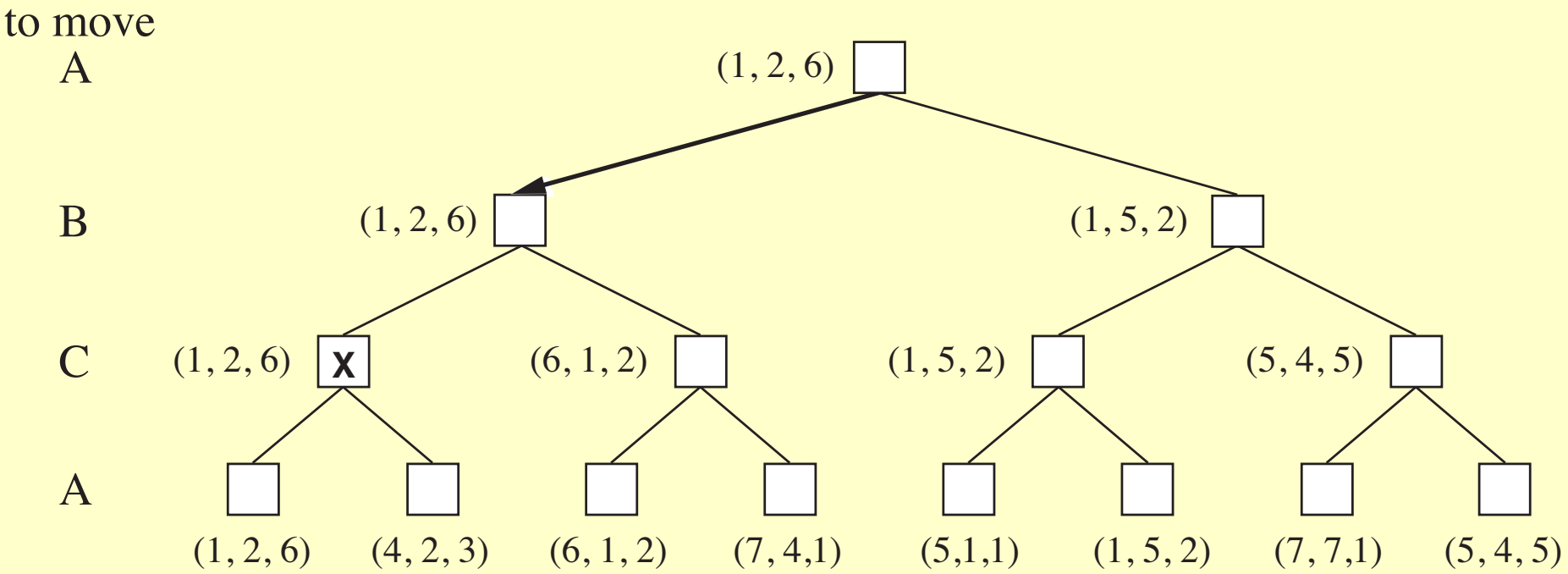
Drzewo gry:



- Złożoność $O(b^m)$, gdzie m to maksymalna głębokość rozwijanych węzłów.

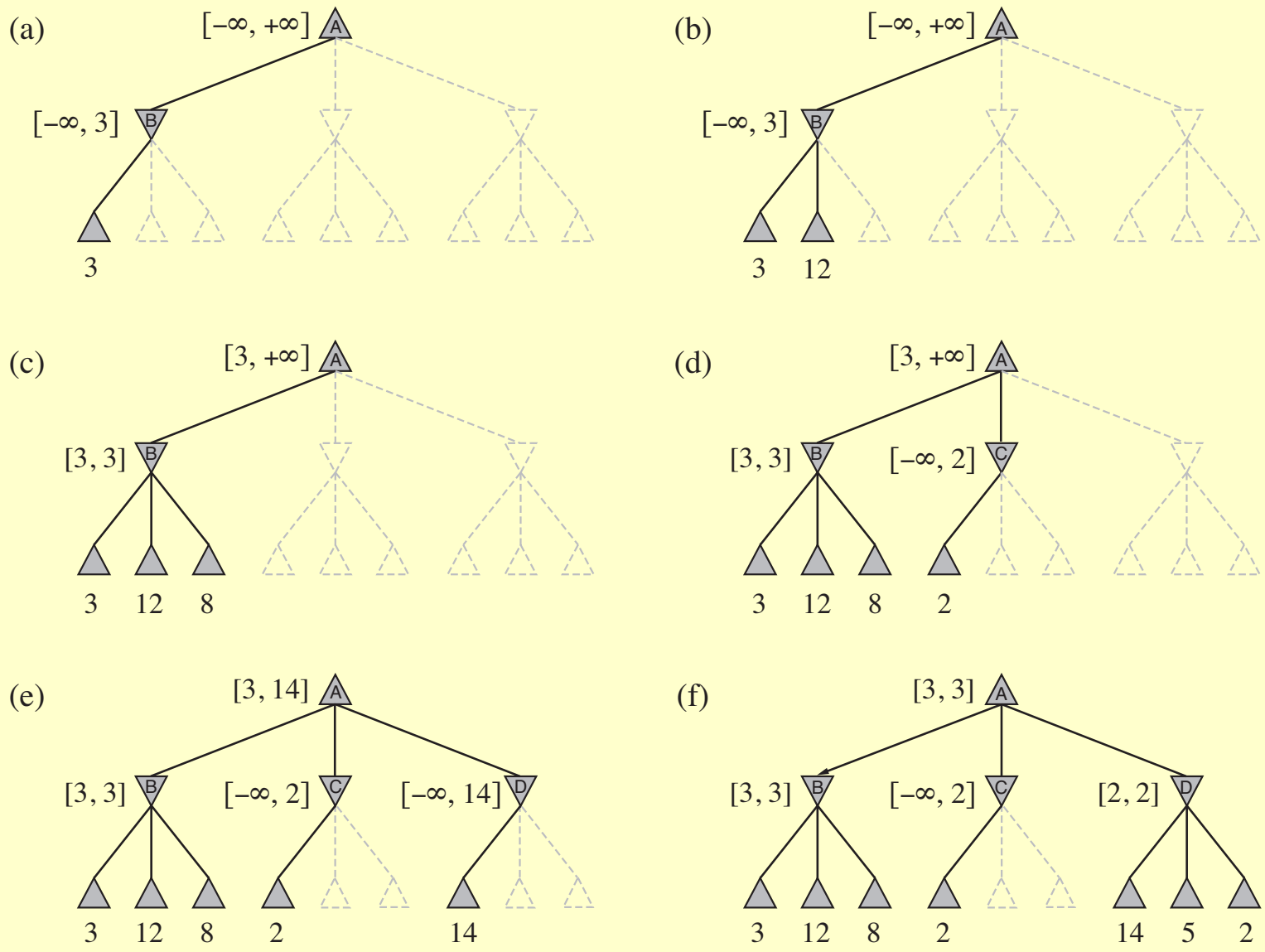
Minimax

Drzewo gry dla 3 graczy:



- Możliwość **współpracy** dwóch graczy **przeciwko** trzeciemu.
- Możliwość współpracy w grach dla dwóch graczy, gdy **niezerowy bilans**.

Obcinanie alfa-beta



Obcinanie alfa-beta

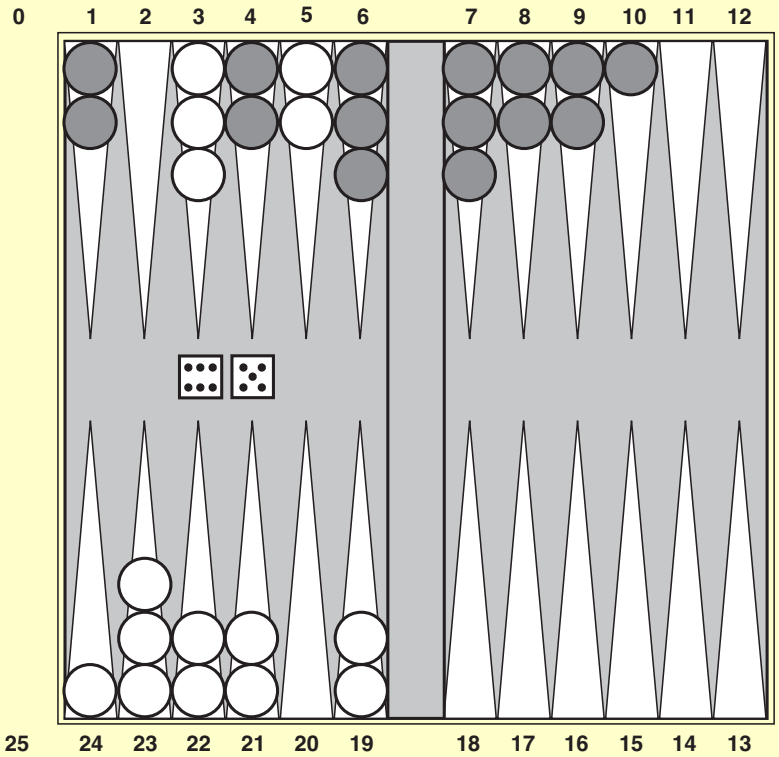
- **Parametry:**
  - α – maksymalna wartość zaobserwowana na ścieżce dla MAX,
  - β – minimalna wartość zaobserwowana na ścieżce dla MIN.
- Parametry są **aktualizowane** w trakcie szukania.
- **Kolejność analizy następników** wpływa na efektywność metody.
- Przy **optymalnej** kolejności rozwijanych jest $O(b^{\frac{m}{2}})$ węzłów – czynnik rozgałęzień $\sqrt{b}$ zamiast b.
- Przy **losowej** kolejności – ok. $O(b^{\frac{3m}{4}})$ węzłów.
- Dla szachów, **proste heurystyki porządkujące** (najpierw bicia, potem zagrożenia, ruchy naprzód, ruchy wstecz) osiągają wynik 2 razy optymalne $O(b^{\frac{m}{2}})$, dodatkowe techniki jeszcze bardziej zbliżają wynik do optymalnego.

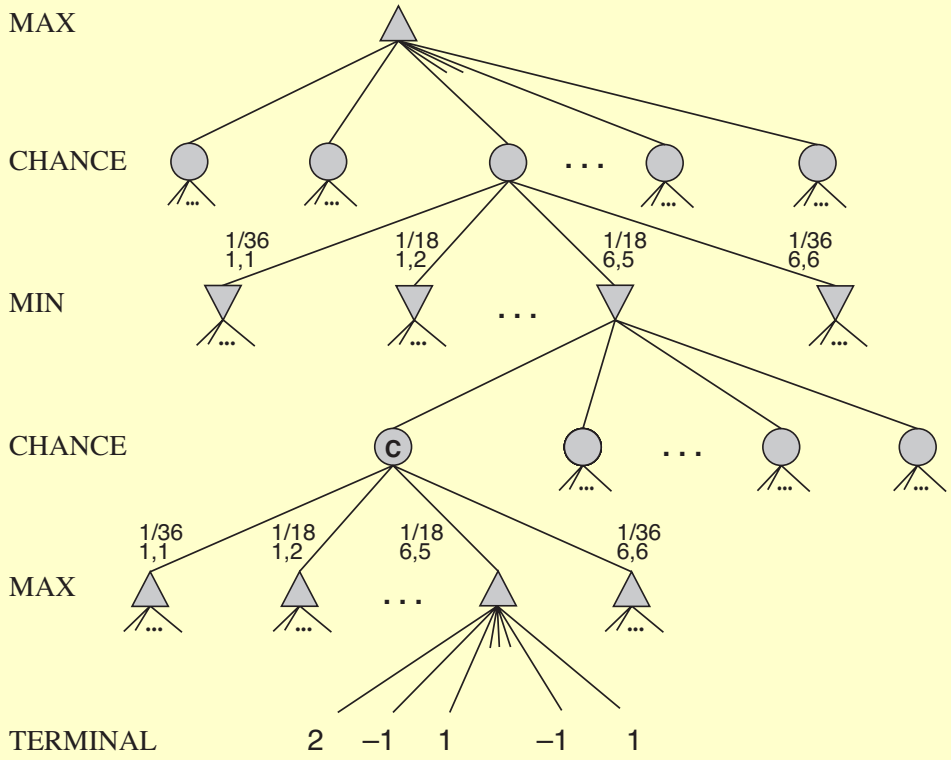
Przybliżone oceny stanu gry

- Nawet z obcinaniem $\alpha - \beta$, **koszty szukania są zwykle zbyt duże**, by szukać głęboko.
- **Obcinanie poddrzew** – test obcięcia zamiast testu stanu końcowego.
- **Miary oceny stanu gry** – generowanie cech i ich kombinacje liniowe i nieliniowe
  - przykład **szachów**: punkty dla figur (pion 1, koń i goniec po 3, wieża 5, hetman 9), – nie zawsze prawdziwe relacje, np. pion za chwilę zostanie zmieniony w hetmana,
  - **brydż**: punkty w kartach ściśle determinują reguły licytacji.
- Ocena nie powinna być kończona w pozycji **o dużej aktywności/zmienności**.
- **Efekt horyzontu** – gracz może odsunąć w czasie przykre konsekwencje (poza horyzont szukania), prowadząc do niewłaściwej oceny
  - przykład **szachów**: można bronić się przed stratą wielokrotnym szachowaniem.

Gry z elementami losowymi

- **Losowość** niezależna od graczy (np. rzut kostką).
- **Tryktrak** (ang. *backgammon*) – najpierw rzut dwiema kostkami, potem ruch zależny od wyniku losowania.
- Drzewa szukania wzbogacone o **poziomy losowe**.
- **Wartość oczekiwana** kryterium *minimax* (*expectiminimax*).





- Modyfikacje obcinania $\alpha - \beta$.

Tworzenie labiryntów (1)

Labirynt można postrzegać jako graf:

- węzły to pomieszczenia/komórki labiryntu,
- krawędzie to połączenia między komórkami (brak ścian)

Wówczas:

labirynt = drzewo rozpinające grafu

Wystarczy użyć znanych algorytmów:

- zrandomizowane szukanie w głąb
- zrandomizowany algorytm Kruskala
- zrandomizowany algorytm Prima
- algorytm Wilsona
  - dodajemy jedną losowo wybraną komórkę do labiryntu
  - dopóki istnieje komórka poza labiryntem
    - losujemy komórkę spoza labiryntu
    - generujemy losową ścieżkę aż do napotkania komórki labiryntu (usuwamy ewentualne pętle)
    - dodajemy ścieżkę do labiryntu

Tworzenie labiryntów (2)

- metoda rekurencyjnych podziałów
  - dzielimy prostokątny obszar cięciem poziomym i pionowym
  - losowe 3 z 4 powstałych (z krzyżowania się) odcinków przerywamy (robimy przejścia)
- wykorzystanie automatów komórkowych

Obciążenie (bias) metod

- w głąb – długie korytarze
- Prima i Kruskala – przeciwnie
- Wilsona – brak obciążenia

Proceduralne generowanie zawartości

- Proceduralne generowanie zawartości - *ang. procedural content generation* (PCG)
- Szum Perlina
  - https://en.wikipedia.org/wiki/Perlin_noise
  - https://mrl.nyu.edu/~perlin/noise/
  - https://pierpaololucarelli.com/2018/02/09/java-perlin-noise-visualisation/
  - https://www.ronja-tutorials.com/2018/09/15/perlin-noise.html
  - https://flafla2.github.io/2014/08/09/perlinnoise.html
  - https://medium.com/100-days-of-algorithms/day-88-perlin-noise-96d23158a44c
  - https://www.youtube.com/watch?v=MJ3bvCkHJtE

Automaty komórkowe

- https://natureofcode.com/book/chapter-7-cellular-automata/
- https://en.wikipedia.org/wiki/Conway's_Game_of_Life
- https://en.wikipedia.org/wiki/Life-like_cellular_automaton
- https://gamedevelopment.tutsplus.com/tutorials/generate-random-cave-levels-using-cellular-automata--game
- https://blog.jrheard.com/procedural-dungeon-generation-cellular-automata

Inteligencja zbiorowa – algorytmy rojowe (1)

Inspiracje biologiczne

- **mrówki** – oznaczanie dróg **feromonami**;
- **pszczoły** – zwiadowcy i ich **tańce** (ang. *waggle dance*);
- **bakterie** – **chemotaksja** dodatnia i ujemna (poruszanie się w kierunku rosnącego/malejącego gradientu stężeń bodźców chemicznych);
- **ptaki**, **ryby** – **synchronizacja** poruszania się stada/ławicy;
- **termity** – wyrafinowane **struktury** termitier (liczne komory, w tym komnata królewska, kanały wentylacyjne), **sortowanie** larw według wieku/rozmiaru.

Inteligencja zbiorowa – algorytmy rojowe (2)

Popularne algorytmy:

- Optymalizacja **rojem cząstek** (ang. *Particle Swarm Optimization*)
- Systemy **mrówkowe**
- Algorytmy **pszczele**
- Optymalizacja na wzór **kolonii bakterii** (ang. *Bacterial Foraging Optimization Algorithm*)
- Algorytmy stada – Reynolds' boids

Particle Swarm Optimization (1)

Idea PSO

- Pewna liczba cząstek **poszukuje** atrakcyjnych rozwiązań w przestrzeni.
- **Położenie cząstki** w przestrzeni to jedno z możliwych rozwiązań.
- Na początku **losowe** pozycje.
- **Pozycja** w kolejnej iteracji zależy od:
  - Wektora **prędkości** tej cząstki,
  - **Najlepszego rozwiązania** znalezionego przez **tę cząstkę**,
  - **Najlepszego rozwiązania globalnie**, tj. znalezionego przez którąkolwiek cząstkę.
- Z czasem cząstki **zbiegają** do optimów funkcji oceny rozwiązań.

Particle Swarm Optimization (2)

```
1  p^best ← null;
2  for i = 1..size
3      v_i ← RandomV()
4      p_i^best ← p_i ← RandomP()
5      if p^best = null or Cost(p_i^best) < Cost(p^best)
6          p^best ← p_i^best
7  while !ShouldStop()
8    for i = 1..size
9      v_i ← v_i + c_1 * r_1 * (p_i^best − p_i) + c_2 * r_2 * (p^best − p_i)
10     p_i ← p_i + v_i
11     if Cost(p_i) ≤ Cost(p_i^best)
12       p_i^best ← p_i
13           if Cost(p_i) ≤ Cost(p^best)
14               p^best ← p_i
15 return p^best
```

c_1 i c_2 to zadane współczynniki,

r_1 i r_2 to losowe wartości z rozkładu jednostajnego na $[0, 1]$.

Particle Swarm Optimization (3)

Uwagi

- Zwykle **mała liczba cząstek** (20–40).
- Wartości c_1 i c_2 zwykle w zakresie $[0, 4]$.
- Różne **kryteria stopu**:
  - liczba **iteracji**,
  - liczba ostatnich **iteracji bez poprawy**.
  - ...
- Ograniczenia **prędkości** (nominalne, bezwładność).
- Ograniczenia **przestrzeni**.

Systemy mrówkowe (1)

Idea systemów mrówkowych

- Na początku mrówki chodzą **po losowych ścieżkach**.
- Mrówka, która znajduje **pożywienie**, wraca do mrowiska **znacząc teren feromonami**.
- Feromony z czasem **wyparowują**.
- Mrówki **wyczuwają feromony** i **preferują** chodzenie oznaczonymi ścieżkami.

Uogólnienie na użytek rozwiązywania zadań

- **Minimalizacja kosztu rozwiązania** uwzględniającego informacje **heurystyczne** (strategia podążania do celu) i **historyczne** (preferowanie ścieżek, które w przeszłości okazywały się dobre).
- Rozwiązania są konstruowane z **komponentów** narastająco.
- Dane historyczne są **aktualizowane** w każdym kroku, **proporcjonalnie** do jakości rozwiązań, w których występują oceniane komponenty.

Systemy mrówkowe (2)

- Pseudokod algorytmu:

```
1  S_0 ← RozwiązanieHeurystyczne()
2  S^best ← S_0
3  feromony ← Feromony(S_0)
4  while !ShouldStop()
5      for i = 1..size
6          S_i ← RozwiązanieProbabilistyczne(feromony, α, β)
7          if Cost(S_i) ≤ Cost(S^best)
8              S^best ← S_i
9      feromony ← ZanikFeromonów(feromony,ρ)
10     for i = 1..size
11       feromony += Feromony(S_i)
12 return S^best
```

Systemy mrówkowe (3)

- α i β to współczynniki sterujące wpływem przesłanek heurystycznych i historycznych na prawdopodobieństwo wyboru danego składnika:

$$P_{i,j} \leftarrow \frac{\tau_{i,j}^{\alpha} \times \eta_{i,j}^{\beta}}{\sum_{k=1}^{c} \tau_{i,k}^{\alpha} \times \eta_{i,k}^{\beta}},$$

  gdzie $\tau_{i,j}$ to ilość feromonów krawędzi (i, j), a $\eta_{i,j}$ to ocena heurystyczna tej krawędzi (np. odwrotność odległości pomiędzy miastami w problemie komiwojażera).
- $\rho \in (0, 1)$ to współczynnik zaniku feromonów:

$$\tau_{i,j} \leftarrow (1 - \rho)\tau_{i,j}.$$

- Dla zróżnicowania zbioru rozwiązań można dodatkowo zmniejszać ilość feromonów dla ścieżek wybranych w kolejnych rozwiązaniach probabilistycznych.

Algorytmy pszczele (1)

Idea algorytmów pszczelich

- Pszczoły-zwiadowcy
  - szukają bogatych w nektar obszarów (nawet ponad 10km od ula),
  - gdy znajdą, wracają do ula i specyficznym tańcem (ang. *waggle dance*) zachęcają pszczoły-robotnice do podążenia za nimi w znalezione miejsce.
- Pszczoły-robotnice oceniają tańce zwiadowców i podążają za najatrakcyjniejszymi tancerzami.
- Tańce muszą być „proporcjonalne" do atrakcyjności odnalezionych zasobów.

Algorytmy pszczele (2)

Algorytmy pszczele dla minimalizacji funkcji

- Poszukiwanie atrakcyjnych obszarów (ang. *exploration*) – funkcja zwiadowców.
- Szczegółowe eksplorowanie znalezionych jako atrakcyjne obszarów (ang. *exploitation*) – funkcja robotnic.
- Pewna liczba pszczół/agentów cały czas poszukuje nowych obszarów, reszta je eksploruje.

Algorytmy pszczele (3)

Pseudokod algorytmu:

```
1  populacja ← PopulacjaPoczątkowa()
2  while !ShouldStop()
3      Oceń(populacja)
4      najlepszy ← Najlepszy(populacja)
5      następnaGeneracja ← ∅
6      foreach Obszar ∈ NajlepszeObszary(populacja, ileObszarów)
7          sąsiedztwo ← rozwiązania w sąsiedztwie Obszaru
8          następnaGeneracja ← następnaGeneracja ∪ {Najlepszy(sąsiedztwo)}
9      następnaGeneracja ← następnaGeneracja ∪ Losowe()
10     populacja ← następnaGeneracja
11 return najlepszy
```

Bacterial Foraging Optimization Algorithm (1)

Idee Bacterial Foraging Optimization Algorithm (BFOA)

- Naśladowanie poszukiwania żywności przez bakterie *escherichia coli* i *myxococcus xanthus*.
- **Chemotaksja** – poruszanie się w kierunku rosnącego/malejącego gradientu stężeń bodźców chemicznych.
- Bakterie wydzielają do środowiska **wabiące** lub **odpychające** substancje chemiczne.
- Witki umożliwiają poruszanie się **chaotyczne** lub **płynne, ukierunkowane**.
- Interakcje między komórkami pozwalają im gromadzić się przy pożywieniu lub odpychać się nawzajem.

Bacterial Foraging Optimization Algorithm (2)

Idee algorytmu

- Komórki poruszają się stochastycznie lub zbiorowo w kierunku optimów.
- Podstawowe zasady:
  - chemotaksja,
  - reprodukcja (z naturalną selekcją),
  - eliminacja i rozpraszanie się – jedne komórki giną, inne się pojawiają w losowych miejscach
- Miara interakcji komórki:

$$g(cell_k) = \Sigma_{i=1}^{S} - d_{attr} \times e^{-w_{attr} \times \Sigma_{m=1}^{P} (cell_{k,m} - cell_{i,m})^2}$$

$$+\Sigma_{i=1}^{S} h_{repel} \times e^{-w_{repel} \times \Sigma_{m=1}^{P} (cell_{k,m} - cell_{i,m})^2}$$

pomniejsza koszt komórki (rozwiązania).

Bacterial Foraging Optimization Algorithm (3)

Pseudokod algorytmu:

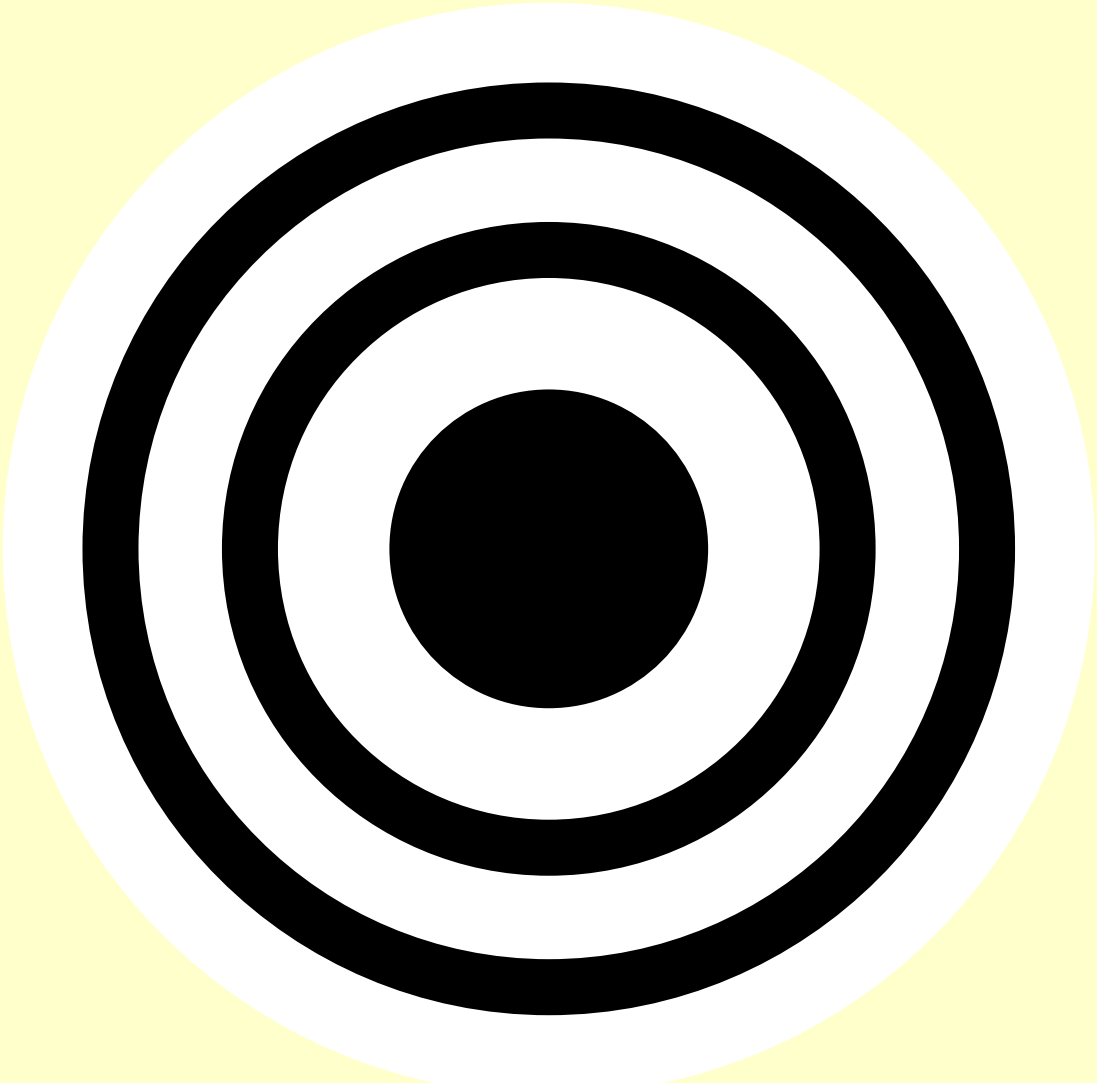
```
1  populacja ← PopulacjaPoczątkowa()
2  for i = 1..N_ed
3      for j = 1..N_re
4          for k = 1..N_ch
5              ChemotaksjaIPływanie(populacja)
6              foreach kom ∈ populacja
7                  if Cost(kom) ≤ Cost(najlepsza)
8                      najlepsza ← kom
9          wybrane ← ZdrowszaPołowa(populacja)
10         populacja ← 2×wybrane
11     foreach kom ∈ populacja
12         if Rand() ≤ p_ed
13             kom ← LosowaKomórka()
14 return najlepsza
```

Rozmycie a prawdopodobieństwo

- Oba systemy:
  - *opisują niepewność liczbami z zakresu* $[0, 1]$
  - *działają na zbiorach i formułach logicznych z zachowaniem zasad łączności, przemienności i rozdzielności*
- Rozmycie zaczyna się, gdy $A \cap A^c \neq \emptyset$
- Czy w lodówce jest jabłko?
  - *z prawdopodobieństwem 50%*
  - *jest pół jabłka*
- Niedokładny owal to
  - *prawdopodobnie elipsa czy rozmyta elipsa?*
- Łysy czy nie łysy?
- Biedny, bogaty, bardzo bogaty?

Geometria zbiorów rozmytych

- Zbiory rozmyte jako funkcje: $A \equiv m_A : X \to [0,1]$
- Zbiór rozmytych podzbiorów zbioru $X = \{x_1, \ldots, x_n\}$ to kostka $F(2^X) = I^n$

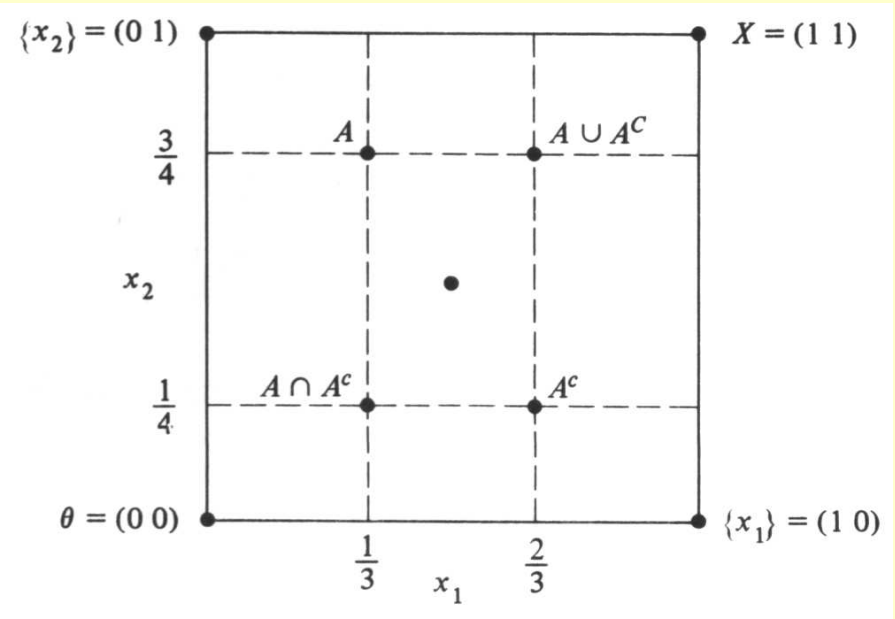


- Środek jest maksymalnie rozmyty: $M = M \cap M = M \cup M = M^c$
- Przykład: $X = \{x_1, x_2\}$ $A = (\frac{1}{3}, \frac{3}{4})$

Teoria zbiorów rozmytych

Operacje na zbiorach rozmytych:

$$
\begin{aligned}
m_{A\cap B} &\stackrel{\text{def}}{=} \min(m_A, m_B) \\
m_{A\cup B} &\stackrel{\text{def}}{=} \max(m_A, m_B) \\
m_{A^c} &\stackrel{\text{def}}{=} 1 - m_A \\
A \subseteq B &\stackrel{\text{def}}{\Leftrightarrow} \forall_{x\in X} m_A(x) \leq m_B(x)
\end{aligned}
$$



Często używana notacja zbiorów rozmytych ($\sum$, $\int$ i / to tylko symbole, nie oznaczają znanych matematycznych operacji!) :

$$
A = \begin{cases} \sum_X m_A(x_i)/x_i & \text{dla dyskretnego zbioru } X \\ \int_X m_A(x)/x & \text{dla ciągłego zbioru } X \end{cases}
$$

Przykłady operacji na zbiorach rozmytych

Dla zbiorów dyskretnych:

$$A = \{0.3/a, 0.5/b, 0.9/c\}$$

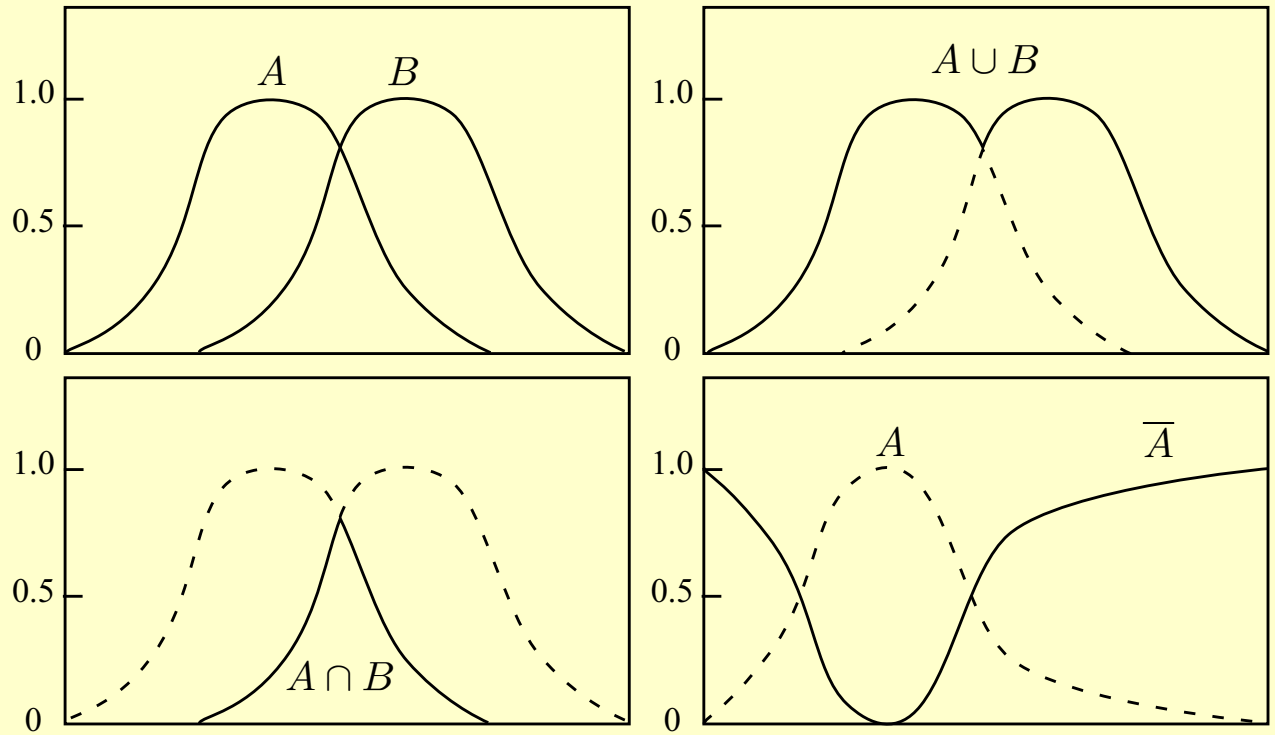
$$B = \{0/a, 0.5/b, 1/c\}$$

$$A \cap B = \{0/a, 0.5/b, 0.9/c\}$$

$$A \cup B = \{0.3/a, 0.5/b, 1/c\}$$

$$A^c = \{0.7/a, 0.5/b, 0.1/c\}$$

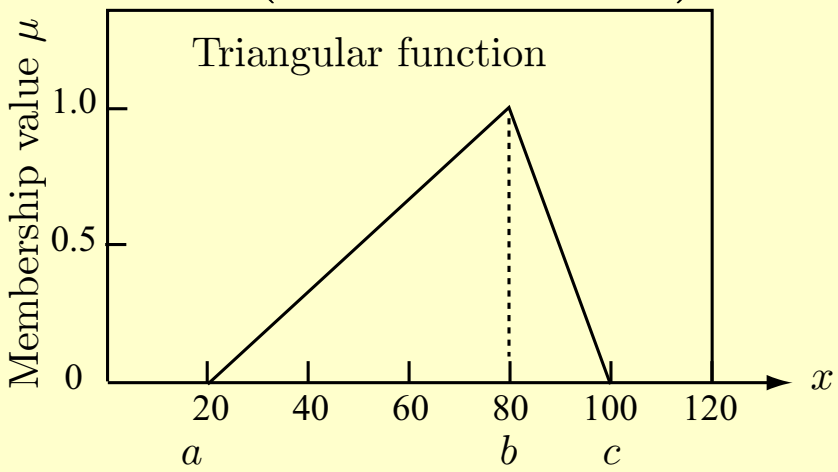
Dla zbiorów ciągłych:



Popularne funkcje przynależności

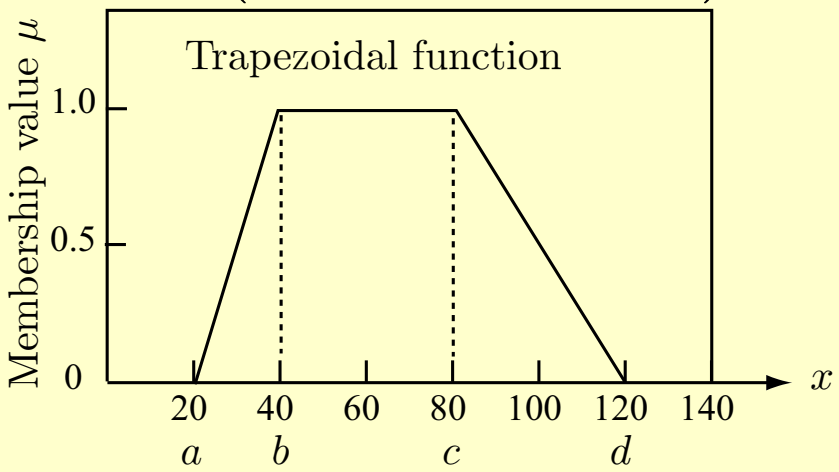
trójkątna

$$\max\left(\min\left(\frac{x-a}{b-a}, \frac{c-x}{c-b}\right), 0\right)$$



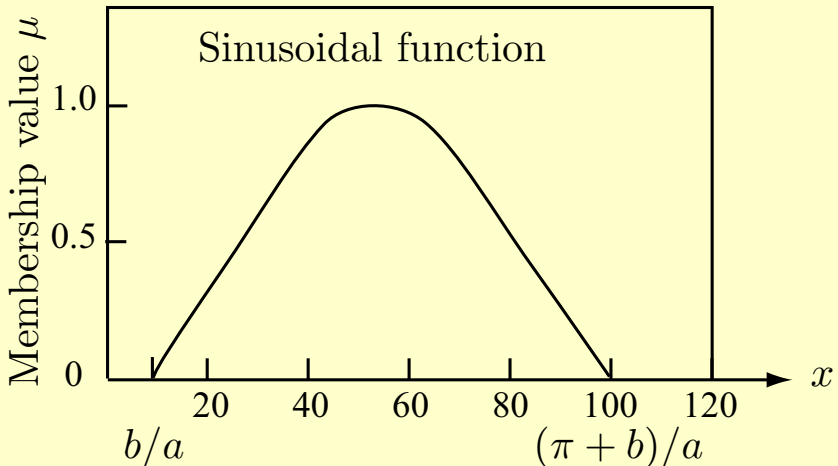
trapezowa

$$\max\left(\min\left(\frac{x-a}{b-a}, 1, \frac{d-x}{d-c}\right), 0\right)$$



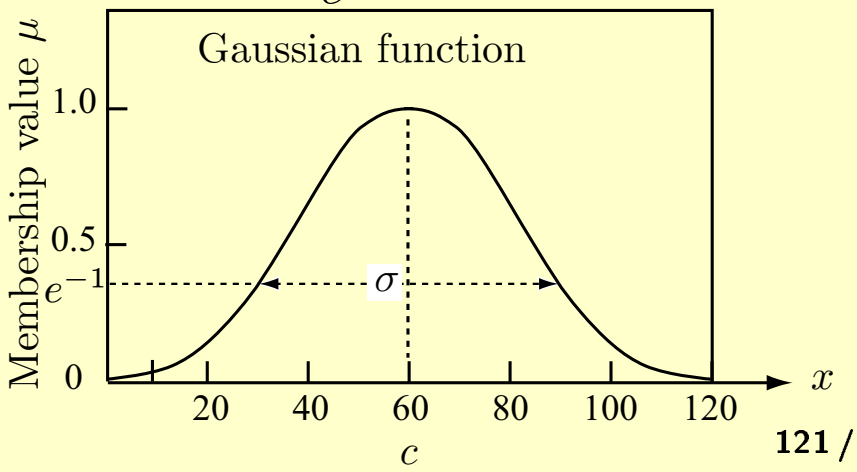
sinusoidalna

$$\begin{cases} \sin(ax-b), & \frac{b}{a} \le x \le \frac{\pi+b}{a} \\ 0, & \text{w przec. wyp.} \end{cases}$$



gausowska

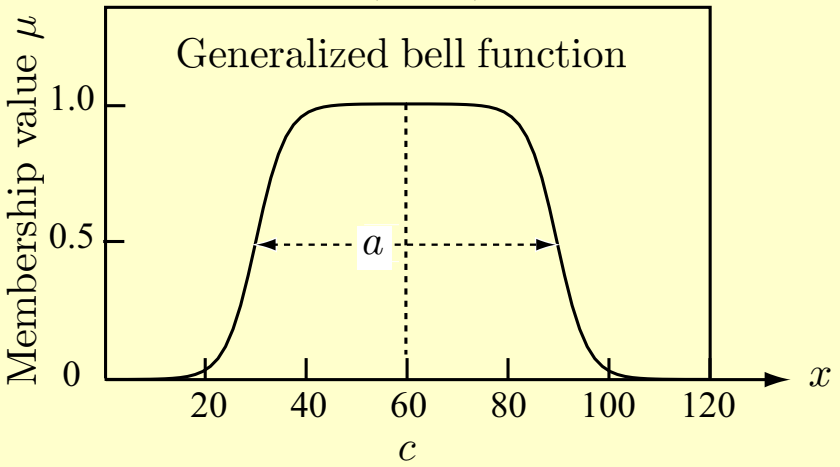
$$e^{-\left(\frac{x-c}{\sigma}\right)^2}$$



Popularne funkcje przynależności

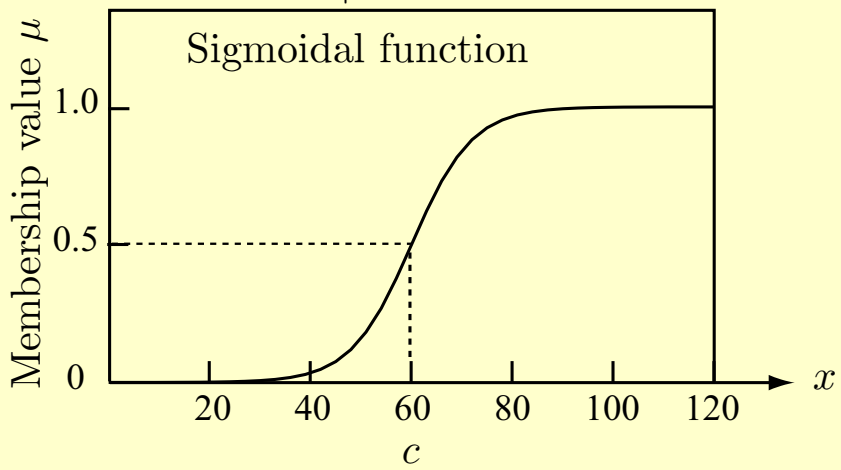
dzwonowa

$$\frac{1}{1+\left|\frac{x-c}{a}\right|^{2b}}$$



sigmoidalna

$$\frac{1}{1+e^{-a(x-c)}}$$



Systemy rozmyte

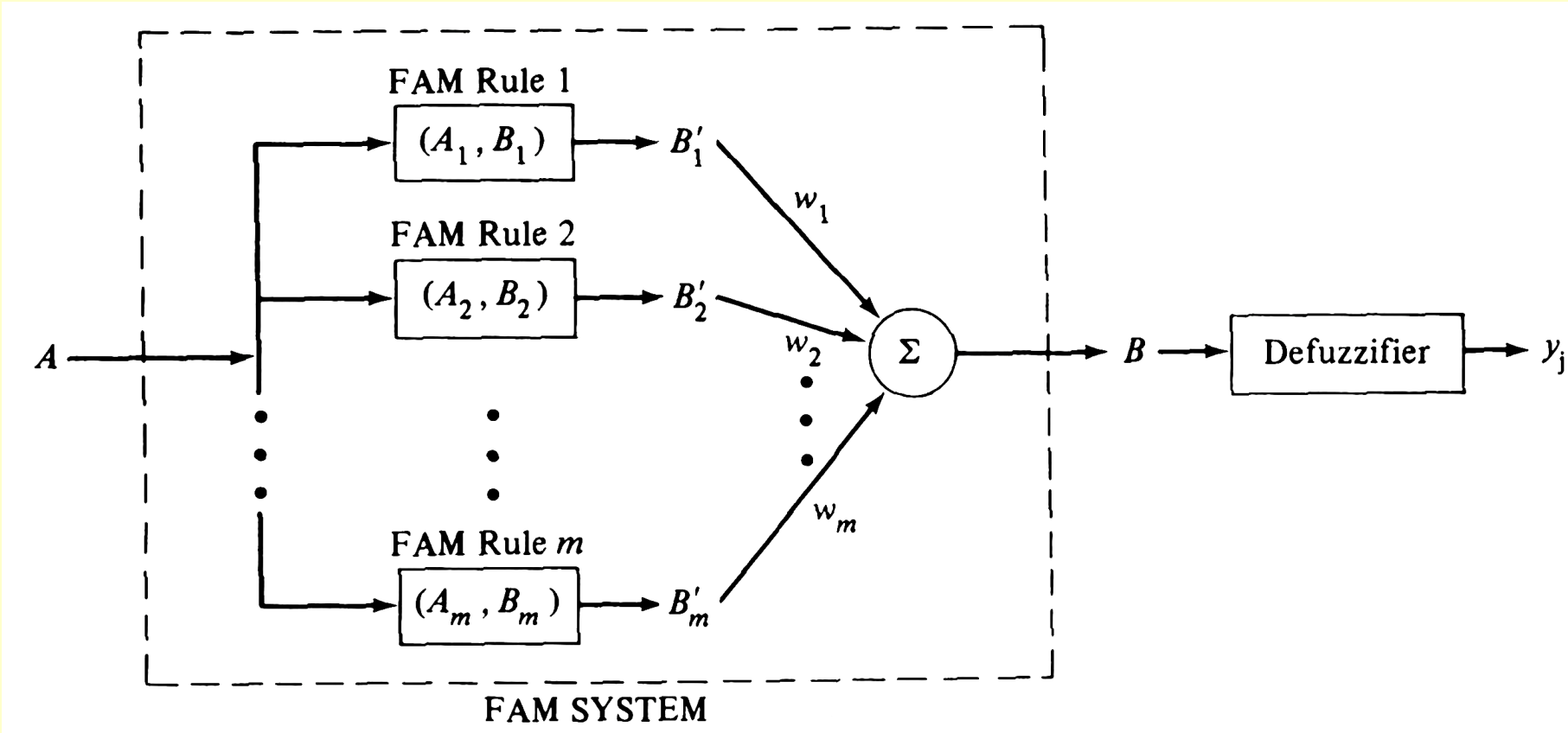
- Systemy rozmyte (rozmyte pamięci asocjacyjne, ang. fuzzy associative memories, FAM) to transformacje $S : I^n \to I^p$.
- Równoległe przetwarzanie m reguł rozmytych $(A_1, B_1), \ldots, (A_m, B_m)$.
- Reguły rozmyte (A, B) to pary zbiorów rozmytych, które można czytać jako:
  Jeżeli X jest A to Y jest B.
- Wejście aktywuje odpowiednio każdą regułę (A_i, B_i) i jest mapowane na wyjście B'_i (częściowo aktywowane B).
- Ostateczne wyjście jest ważoną sumą

$$B = w_1 B'_1 + \cdots + w_m B'_m,$$

gdzie w_i to wagi odpowiadające wiarygodności reguły, jej sile asocjacji itp.

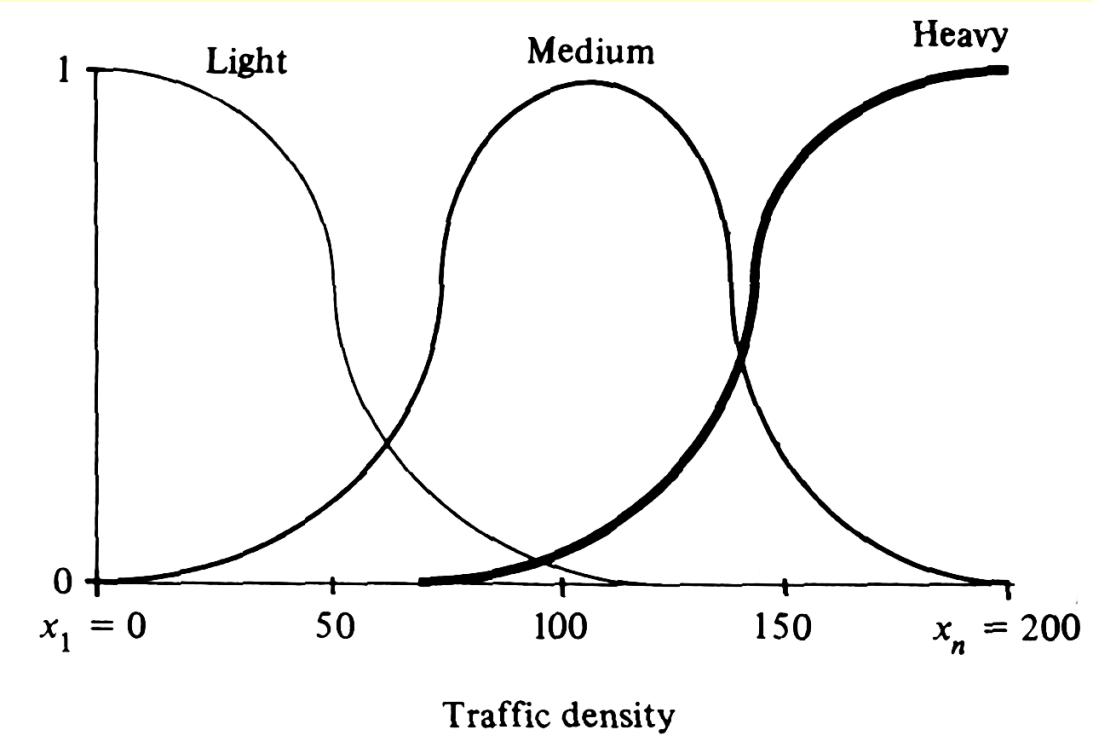
Systemy rozmyte

- Wyjście jest zwykle „wyostrzane” do pojedynczej wartości rzeczywistej y_j.
  - najprostsze wyostrzanie – wybór $y_{\max}$ t. że $m_B(y_{\max}) = \max_{1 \le j \le m} m_B(y_j)$
  - wyostrzanie jako środek masy: $y_j^c = \frac{\sum_{j=1}^{p} y_j m_B(y_j)}{\sum_{j=1}^{p} m_B(y_j)}$



Opracowywanie reguł rozmytych

- Kwantyzacja wejść i wyjść – przykład sterowania sygnalizacją świetlną:



- Proste reguły asocjacyjne: (Heavy, Longer)

Przykład odwróconego wahadła

- Zmienne rozmyte:
  - θ - odchylenie
  - $\Delta\theta$ - prędkość kątowa
  - v - prąd silnika
- Kwantyzacja wejść i wyjścia:
  - NL - negative large
  - NM - negative medium
  - NS - negative small
  - ZE - zero
  - PS - positive small
  - PM - positive medium
  - PL - positive large

$\Delta\theta$ \ θ	NL	NM	NS	ZE	PS	PM	PL
NL				PL			
NM				PM			
NS				PS	NS		
ZE	PL	PM	PS	ZE	NS	NM	NL
PS			PS	NS			
PM				NM			
PL				NL			

- Reguły to trójki zmiennych: (NM, ZE; PM), (ZE, ZE; ZE).

Przykład odwróconego wahadła