KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Projekt współfinansowany ze środków
Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Sztuczna Inteligencja

Krzysztof Grąbczewski
Katedra Informatyki Stosowanej
Uniwersytet Mikołaja Kopernika
Toruń
4 czerwca 2016

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

Program zajęć

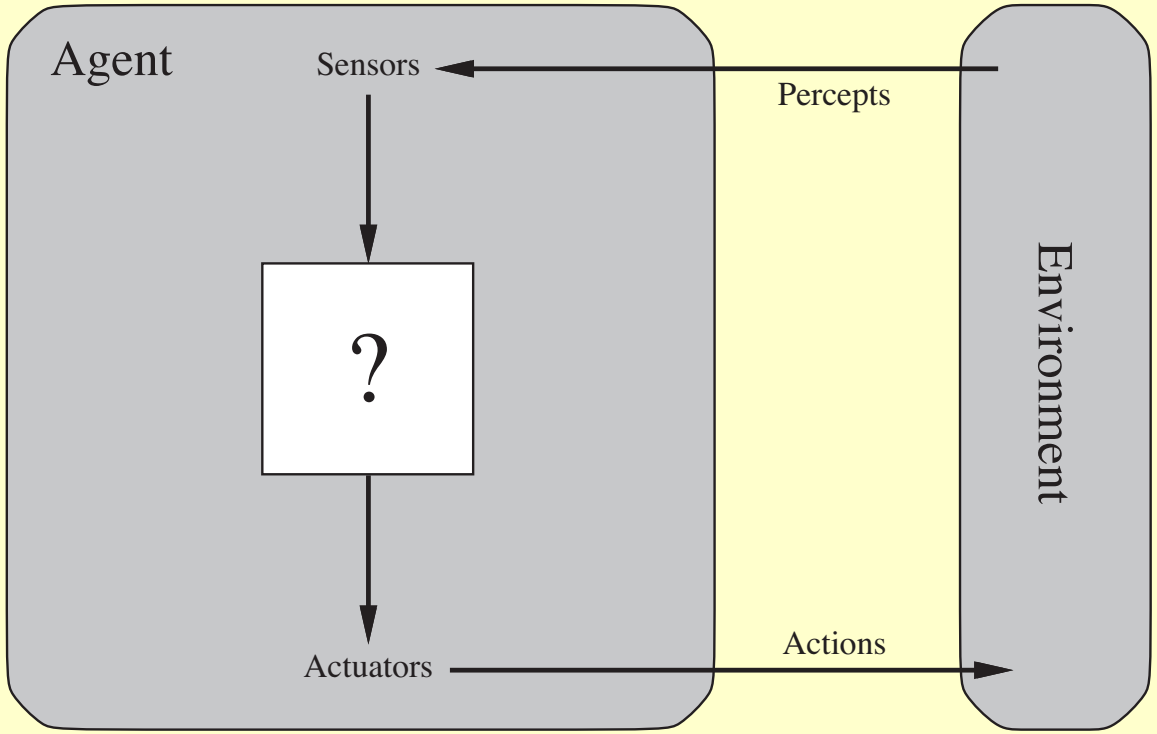
- Wprowadzenie, historia, filozofia AI
- Metody szukania (proste, heurystyczne, z oponentem, programowanie genetyczne)
- Logika w AI (zdaniowa, klasyczna, rozmyta, zbiorów przybliżonych)
- Języki AI - Prolog
- Języki AI - Lisp/Scheme
- Maszyny stanów skończonych, procesy Markowa
- Drzewa zachowań
- Sieci Bayesowskie
- Systemy agentowe
- Rozumienie języka naturalnego
- Uczenie maszynowe i odkrywanie wiedzy z danych

Literatura

- Notatki z wykładu.
- 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.
- Mariusz Flasiński, „Wstęp do sztucznej inteligencji”, Wydawnictwo Naukowe PWN, Warszawa 2011.

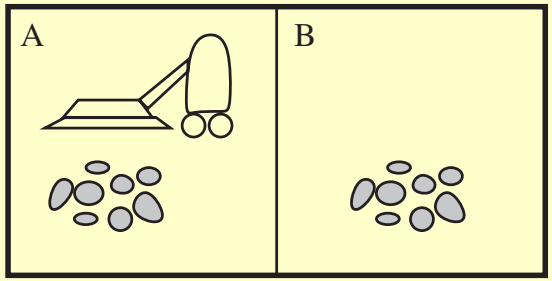
Agent, agenty

- **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

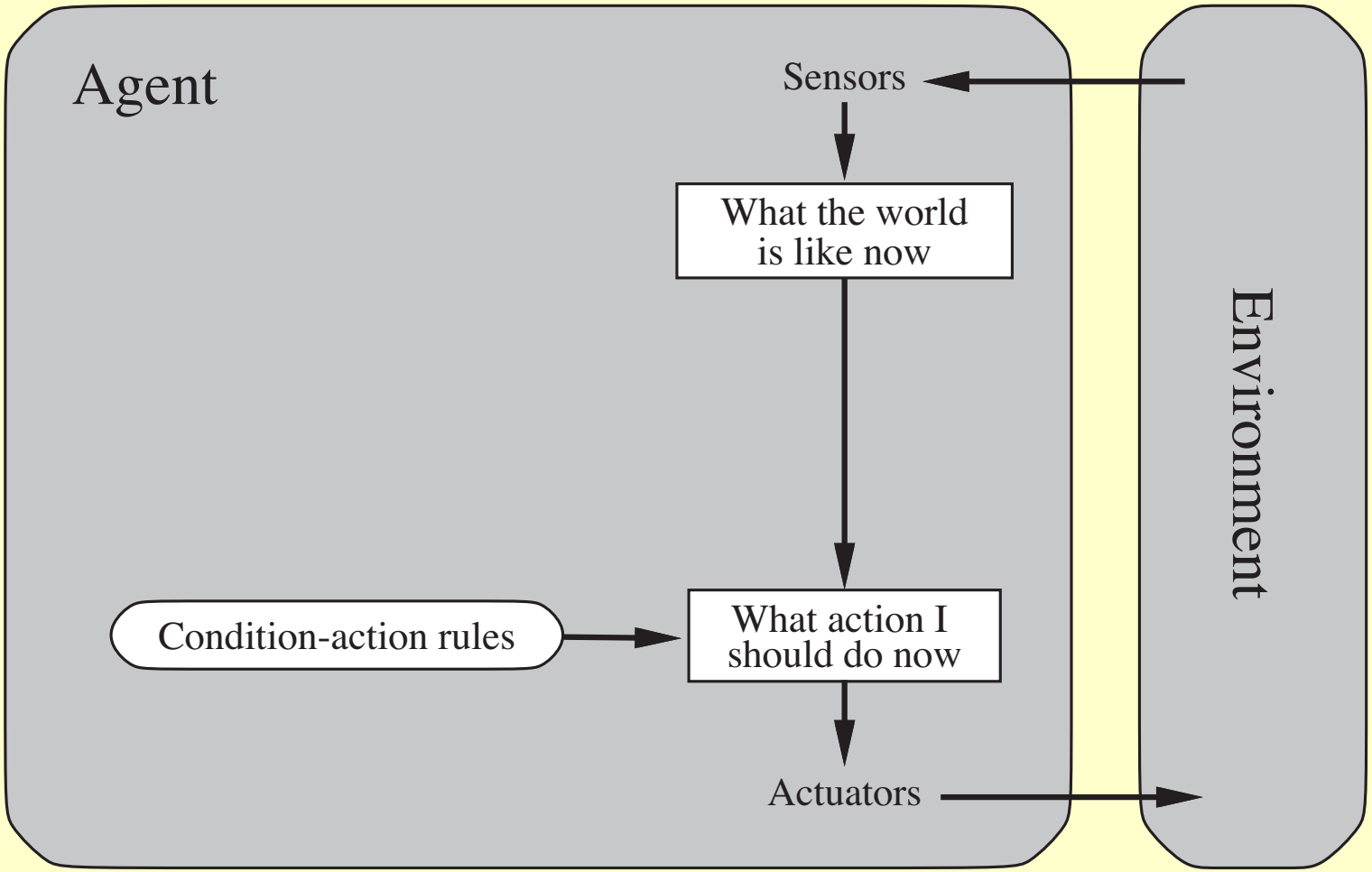
- w pełni i częściowo obserwowalne (ang. *fully/partially observable*)
- deterministyczne i stochastyczne (ang. *deterministic/stochastic*)
- epizodyczne i sekwencyjne (ang. *episodic/sequential*)
- statyczne i dynamiczne (ang. *static/dynamic*)
- dyskretne i ciągłe (ang. *discrete/continuous*)
- jedno-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

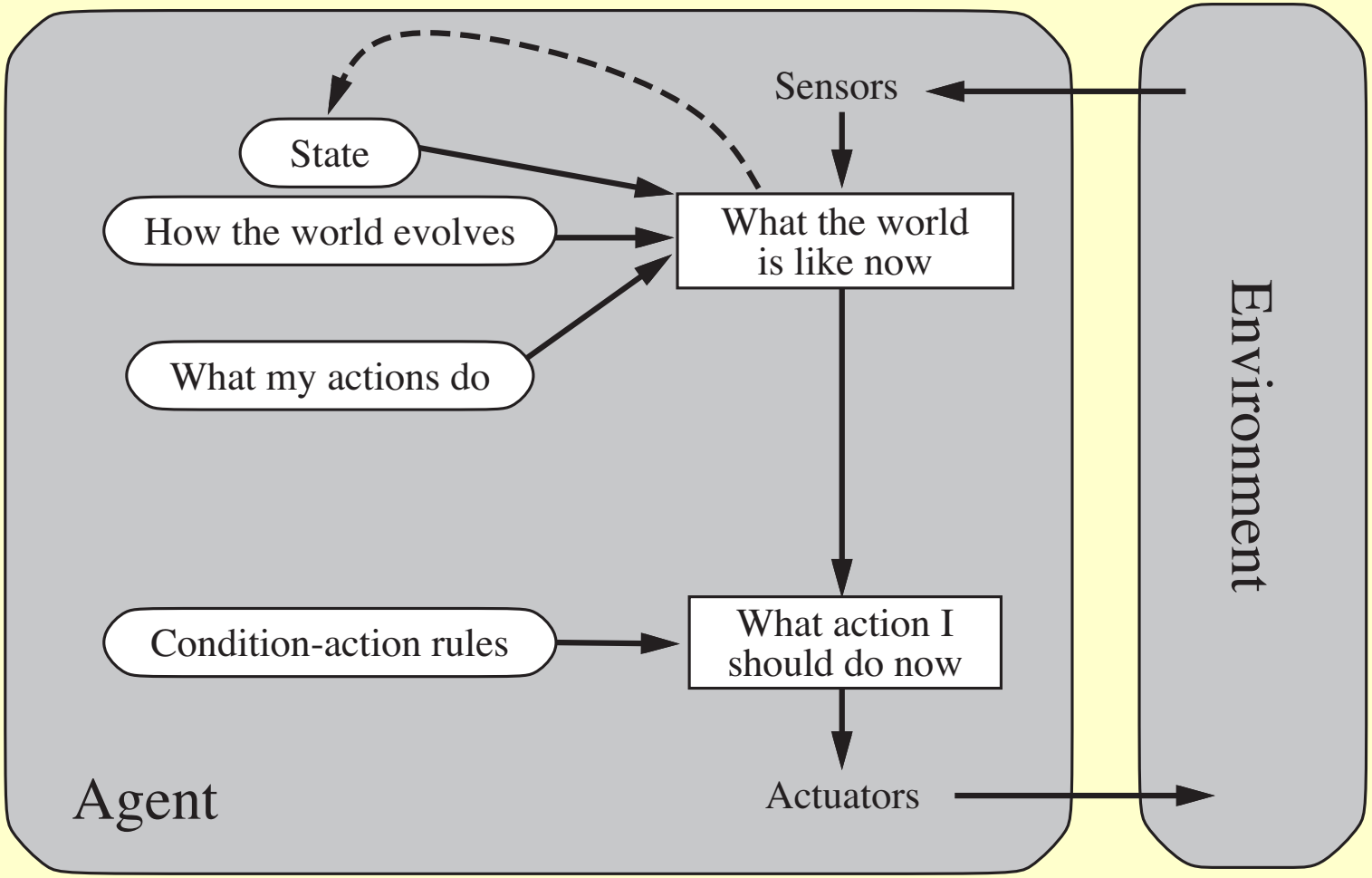
Agenty prostych odruchów

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



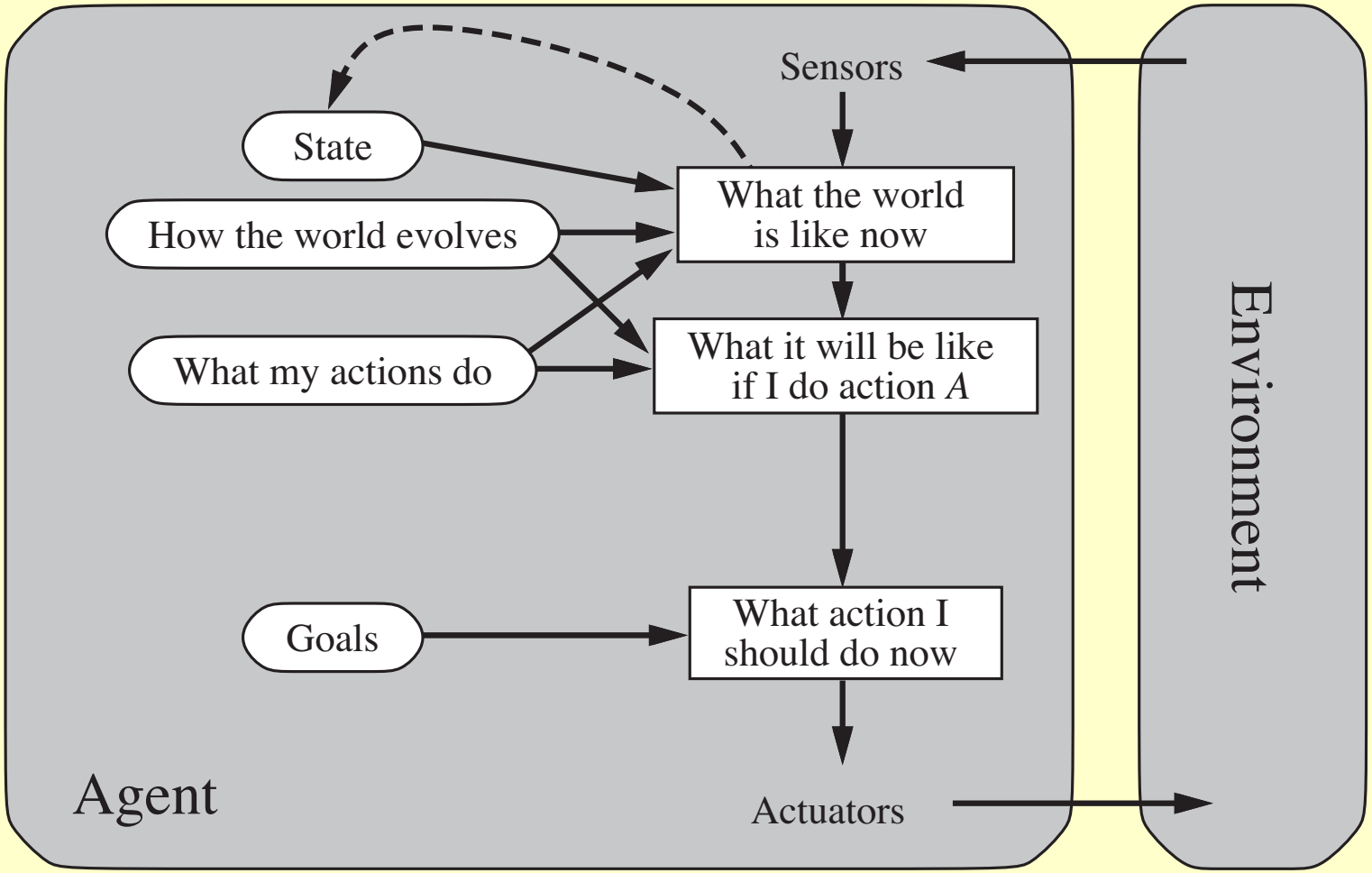
Odruchy oparte na modelach

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



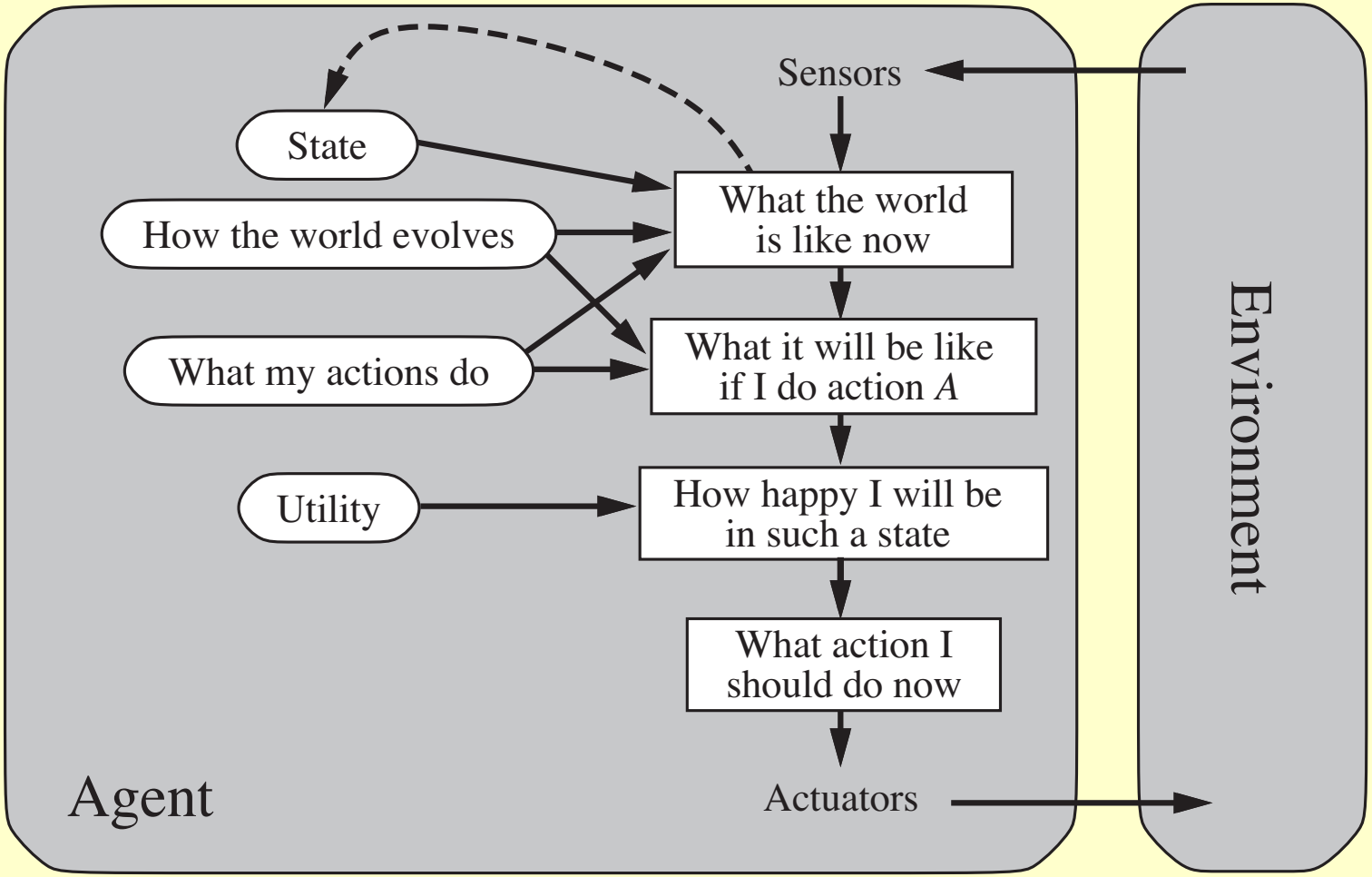
Agenty ukierunkowane na cele

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



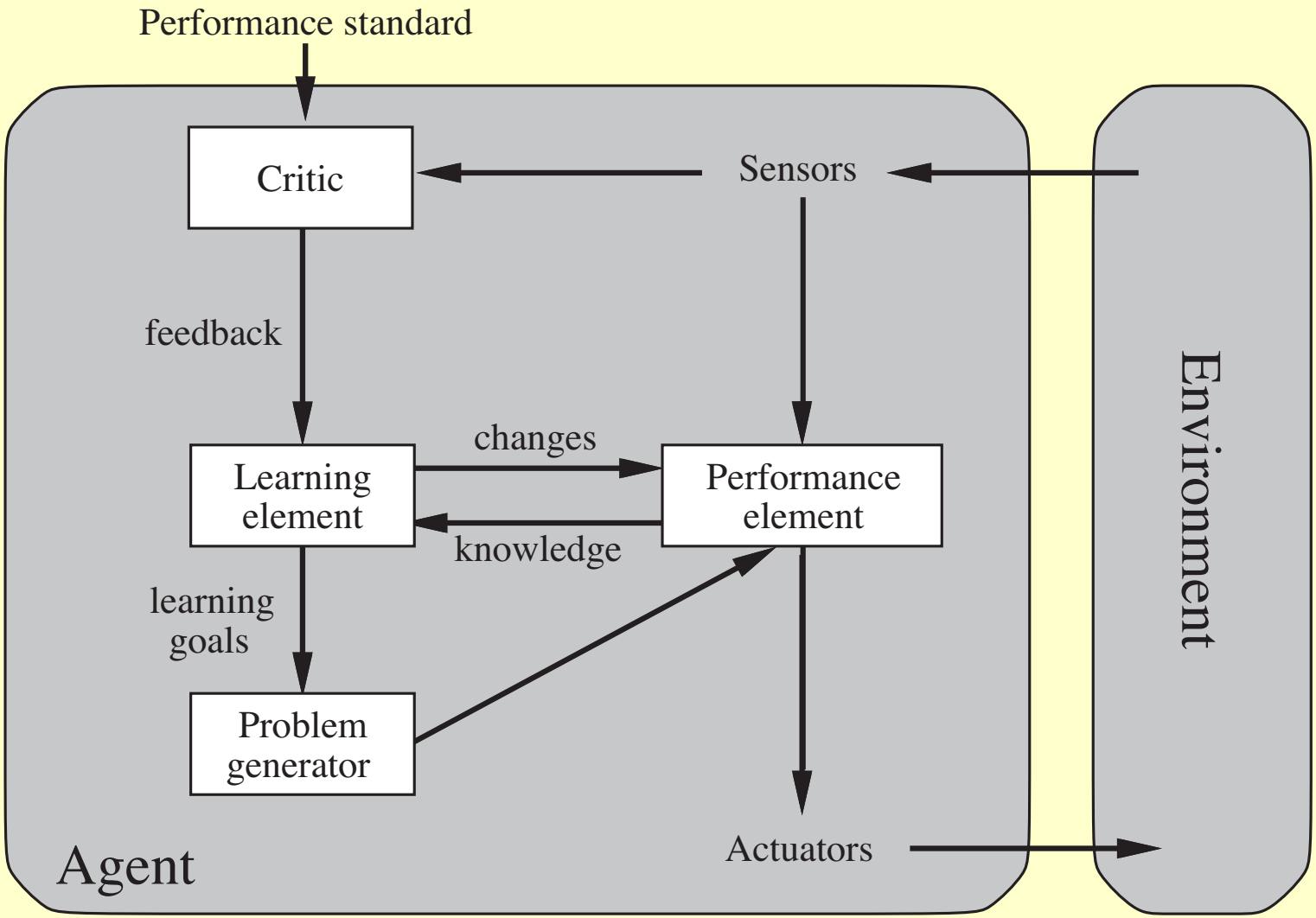
Agenty z funkcjami użyteczności

- 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:

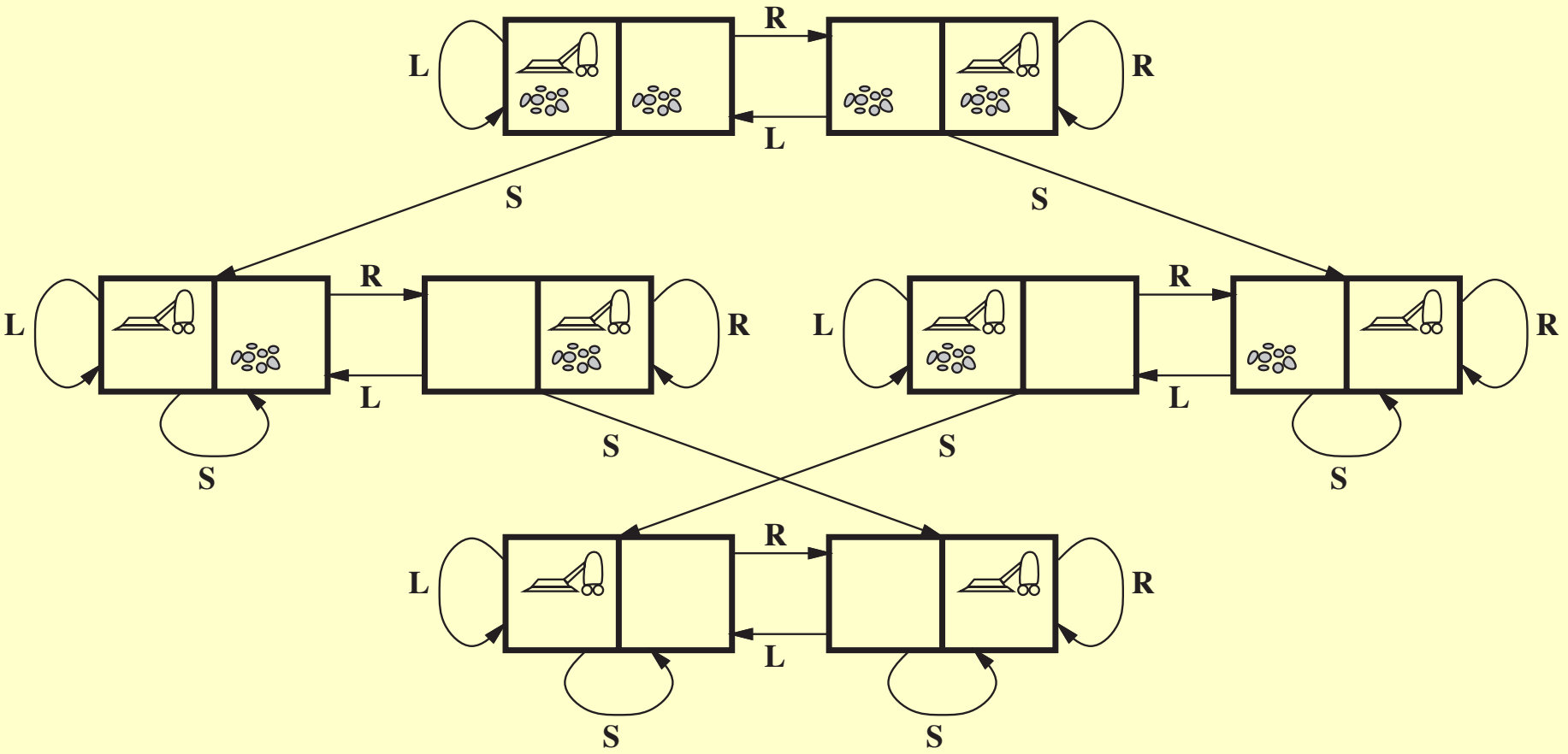
$\{\text{w lewo}, \text{w prawo}, \text{odkurzaj}\}$.

Koszt ścieżki:

Liczba kroków (fragmentów ścieżki).

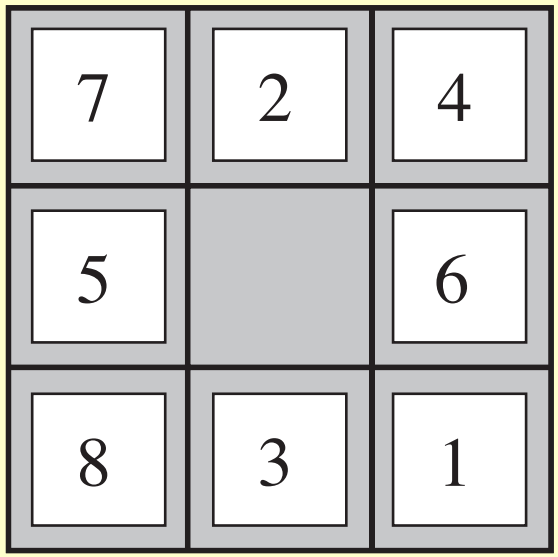
Ogólniej: dla n pomieszczeń mamy $n \cdot 2^n$ stanów.

Przykład – odkurzacz

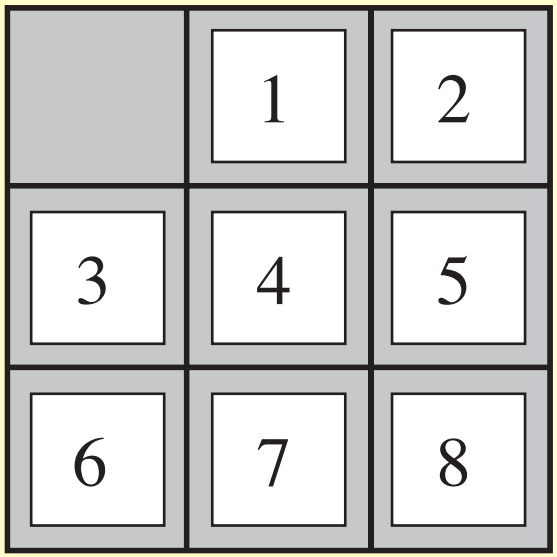


Przykład – ósemka

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

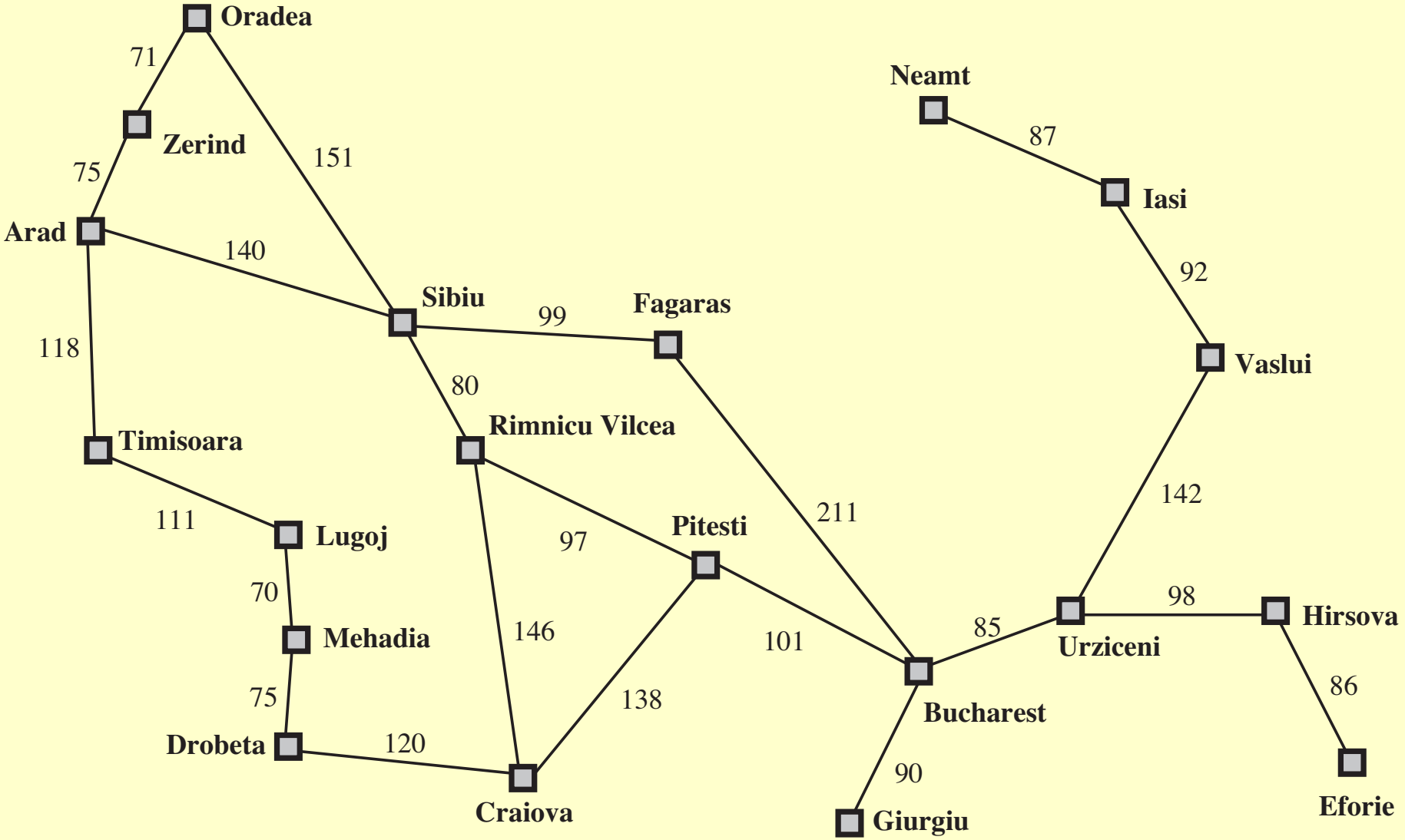
Struktura **drzewa**

- 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

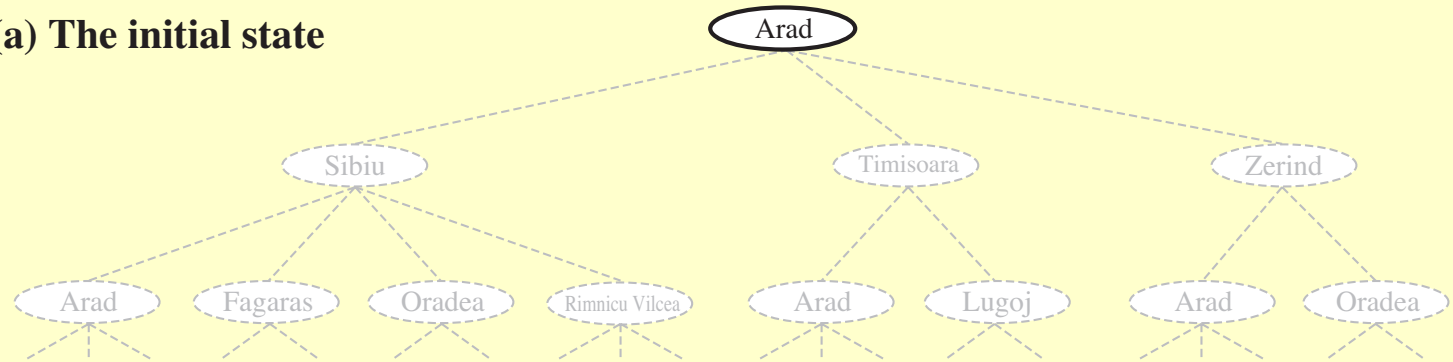
- bieżący **stan**,
- węzeł nadrzędny (**rodzic**),
- **działanie**, które doprowadziło do bieżącego stanu,
- **koszt** ścieżki,
- **głębokość**.

Przykład – fragment Rumunii

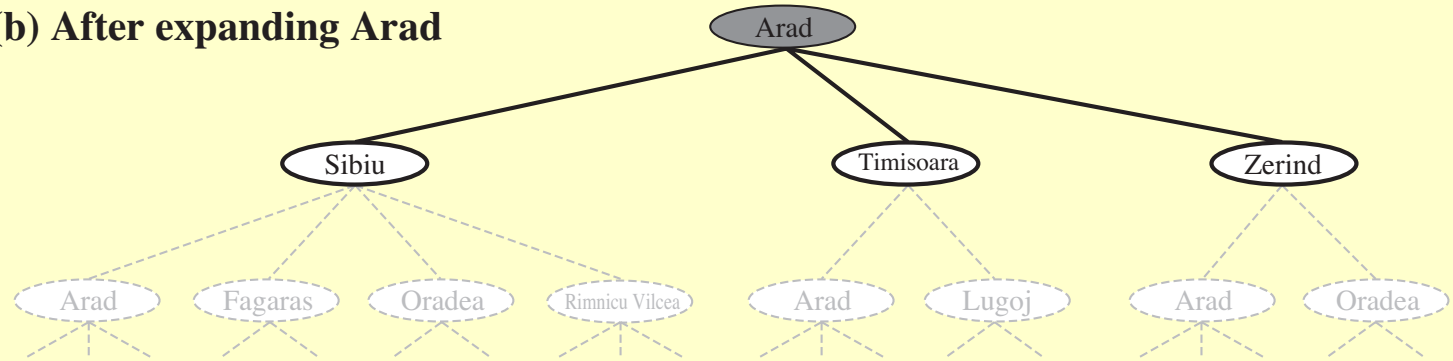


Przykład – fragment Rumunii

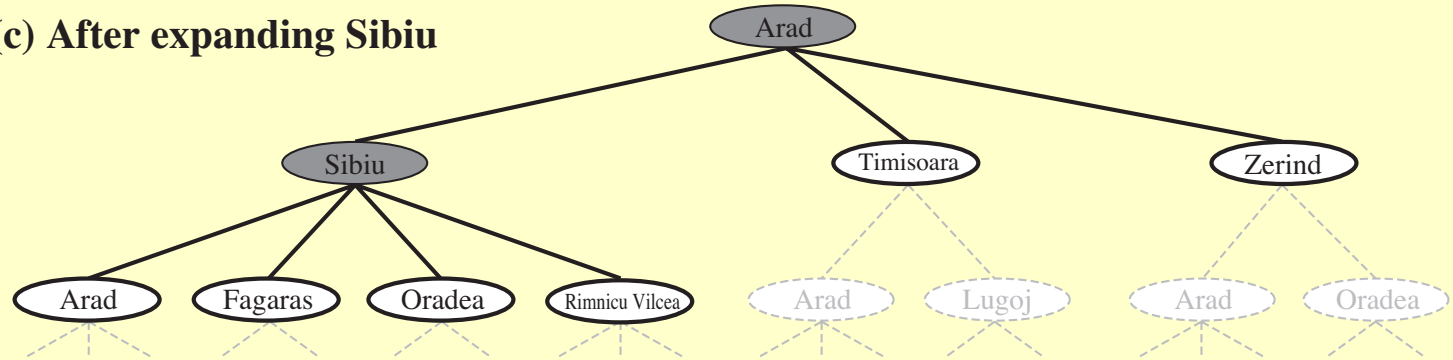
(a) The initial state



(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**.

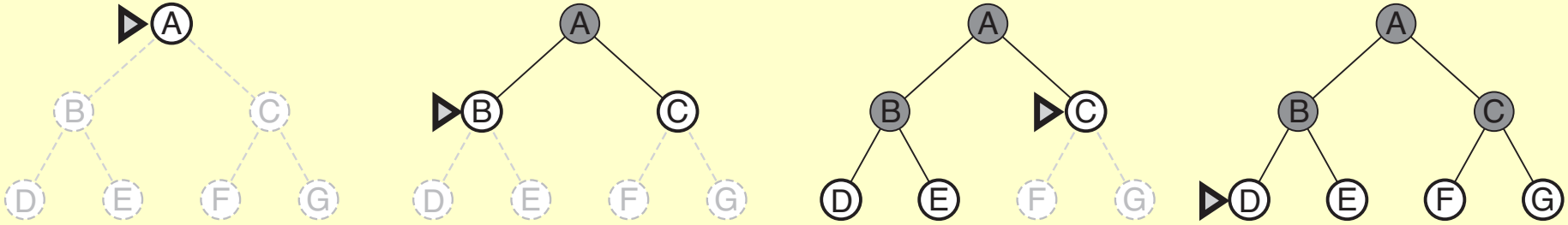
Podstawowe oznaczenia:

- 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:

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

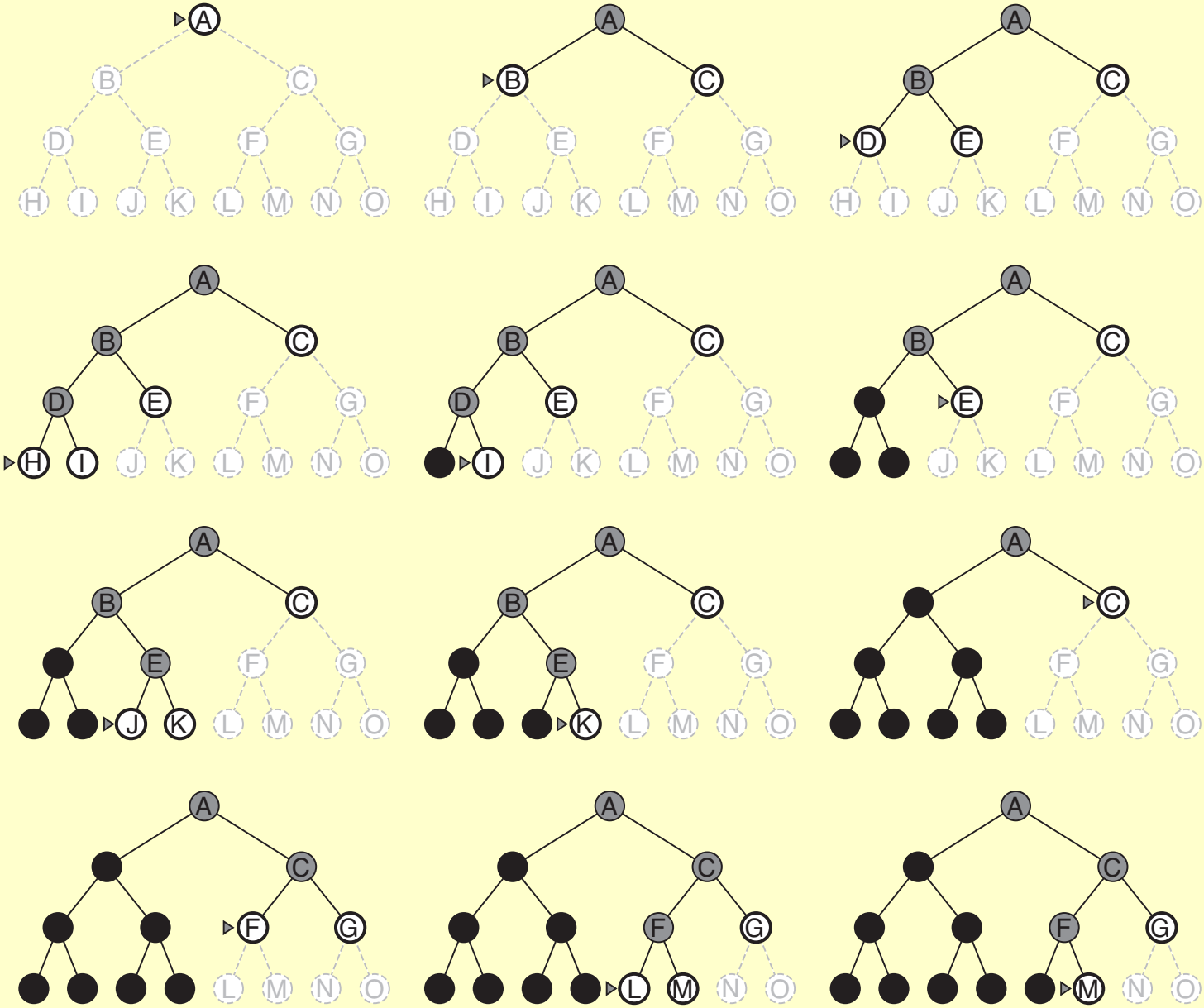
- Takie same: **złożoność czasowa i pamięciowa**, jednak pamięciowa doskwiera bardziej.
- Metoda **zupełna** i **optymalna**.

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*).

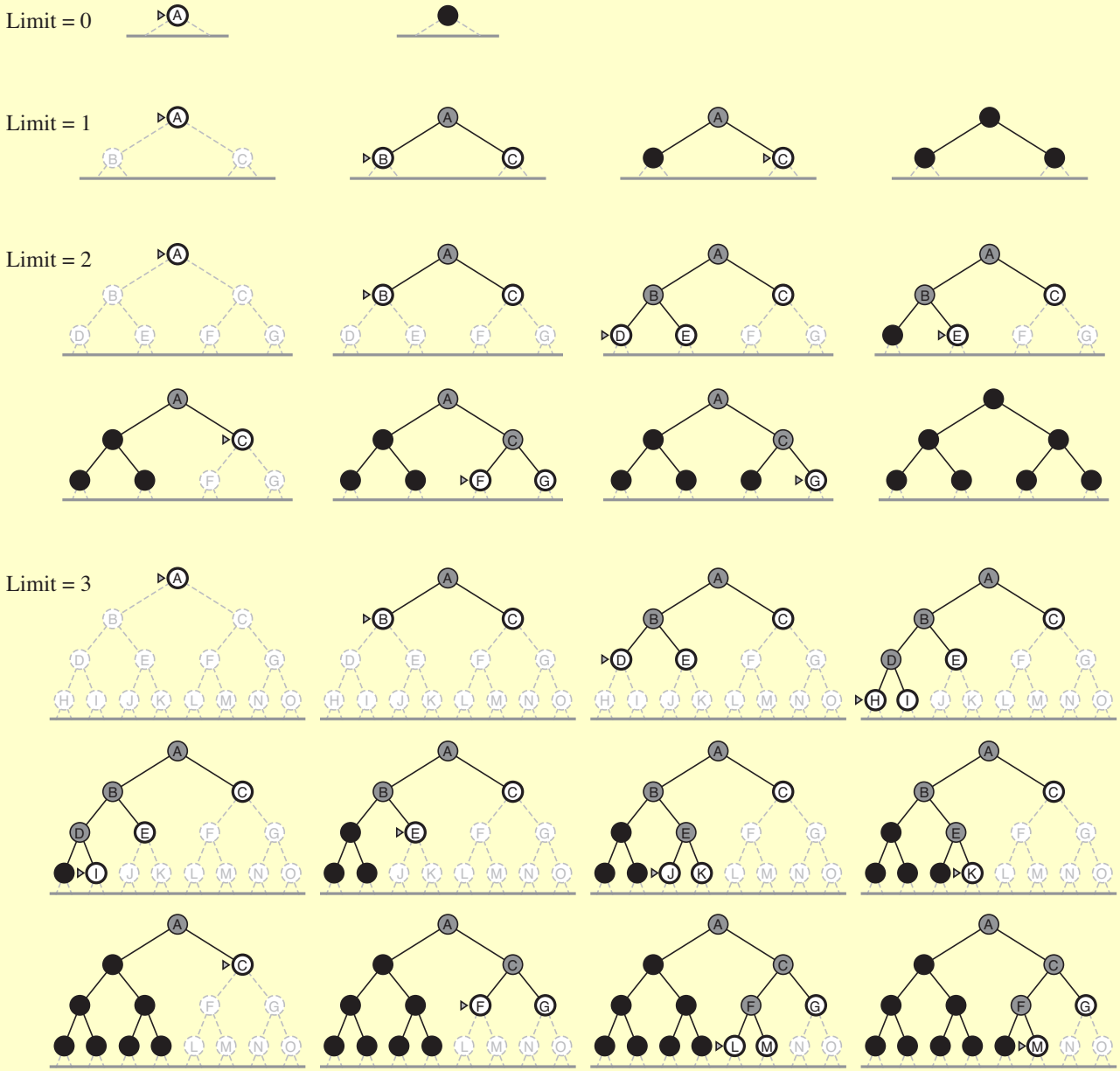
Prototype: ***IDDFS(problem)***
Input: Problem definition.
Output: A solution or failure.

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

Właściwości:

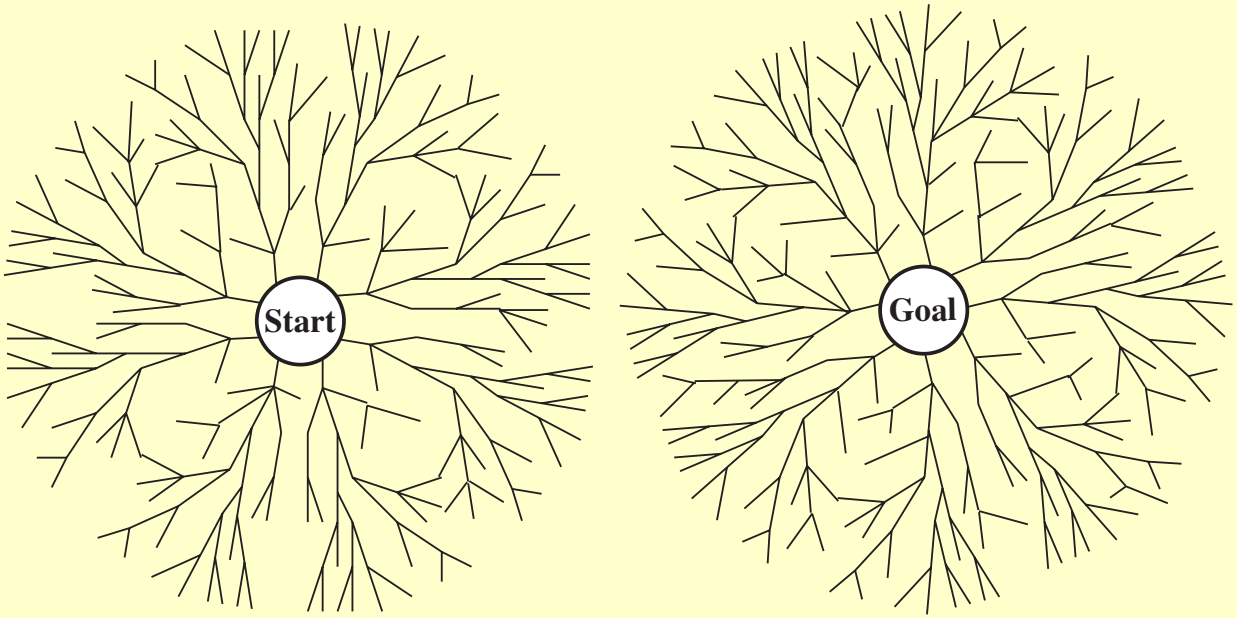
- Złożoność pamięciowa $O(bd)$.
- Złożoność czasowa $O(b^d)$ – powtórki nie kosztują tak dużo jakby się wydawało.
- Metoda zupełna i optymalna.

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 adwersażem**.

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.

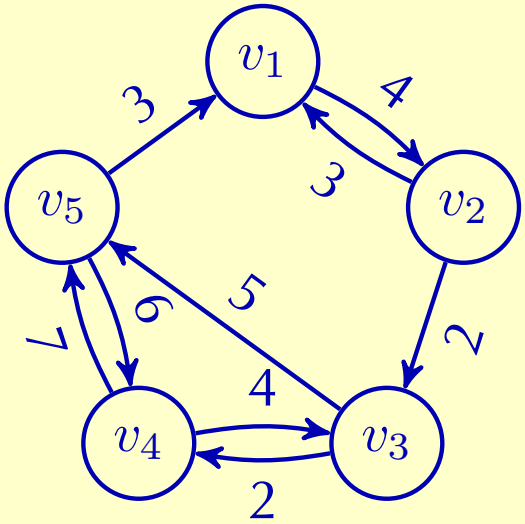
Najprostsze ujęcie:

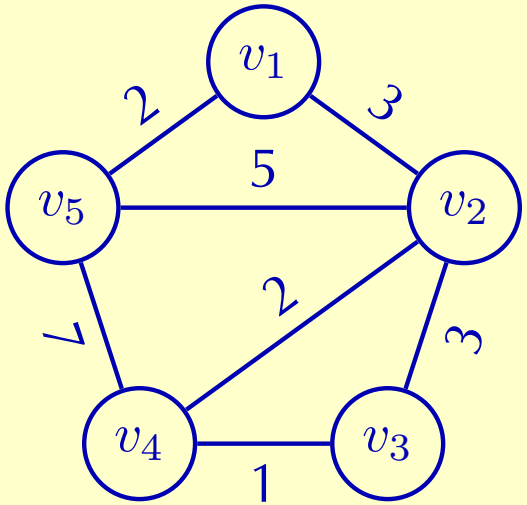
- Badanie węzłów **w kolejności rosnących kosztów**.
- Rozwijane węzły do **kolejki priorytetowej**.
- **Początek: korzeń** do kolejki
- Potem pobieramy z kolejki ścieżkę **o najniższym koszcie** i ją rozwijamy. Następniki dodajemy **do kolejki**.
- Jeśli wiemy, że istnieje droga długości $<= 420$, to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

Grafy

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.

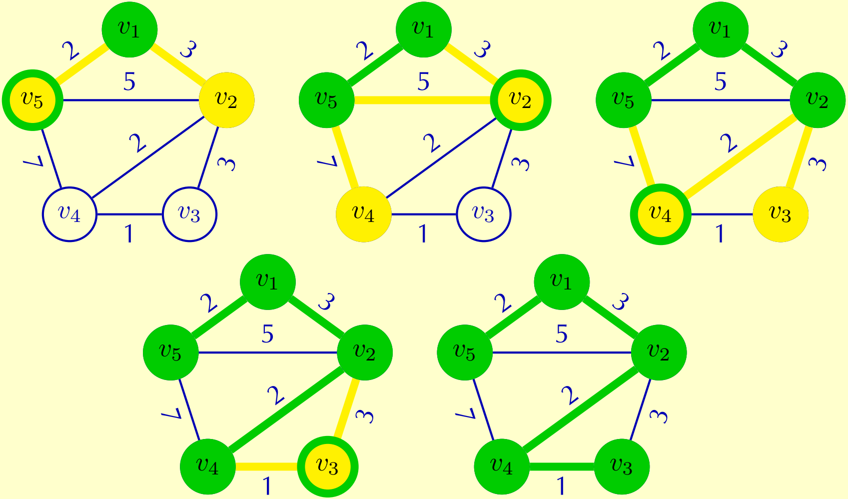
Algorytm Prima

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



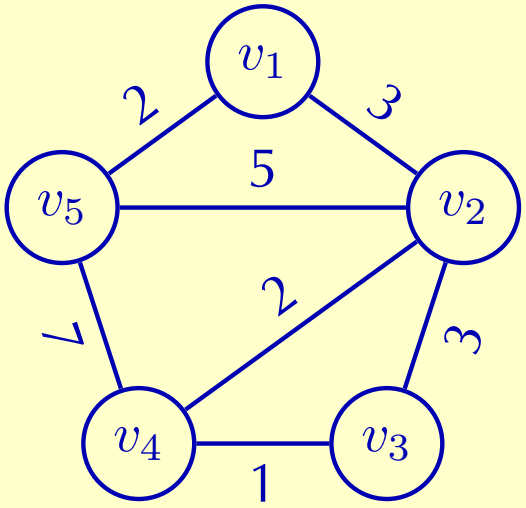
Algorytm Kruskala

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

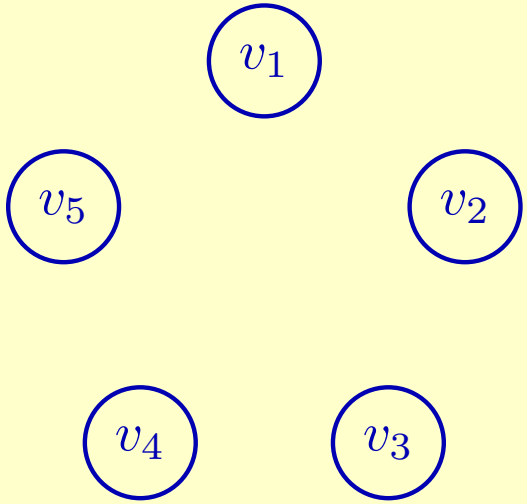
- $V_i \leftarrow \{v_i\}$ – rozłączne jednoelementowe zbiory (dla każdego wierzchołka),
- $F \leftarrow \emptyset$
- Sortujemy krawędzie ze zbioru E niemalejąco względem wag,
- Aż do uzyskania drzewa rozpinającego (wszystkie podzbiory V zostały scalone):
  - bierzemy kolejną krawędź $f \in E$ wg. ustalonego 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



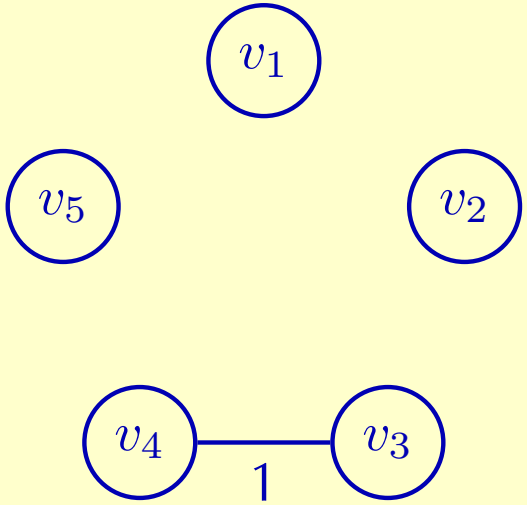
1	(v_3, v_4)	1
2	(v_1, v_5)	2
3	(v_2, v_4)	2
4	(v_2, v_3)	3
5	(v_1, v_2)	3
6	(v_2, v_5)	5
7	(v_4, v_5)	7



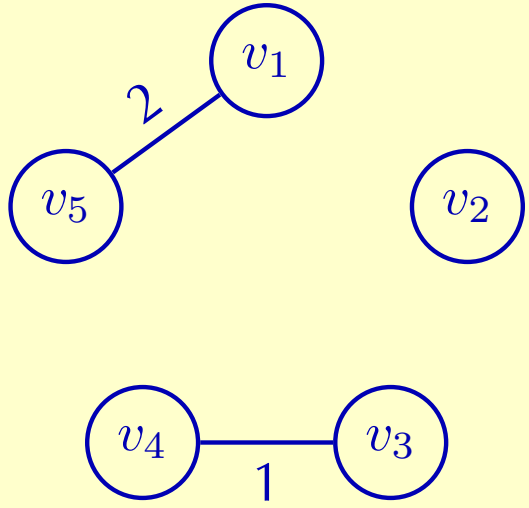
Algorytm Kruskala – przykład c.d.

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

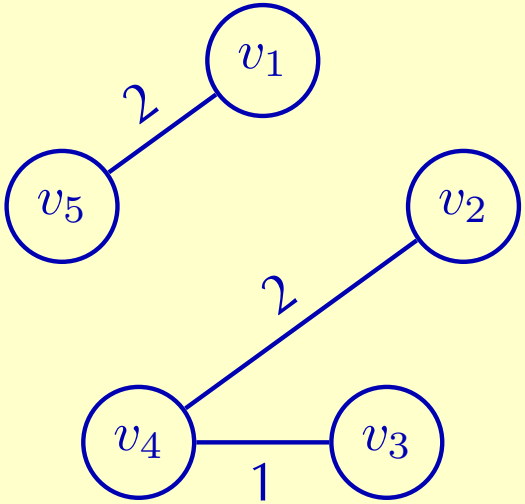
1. Dodajemy (v_3, v_4).



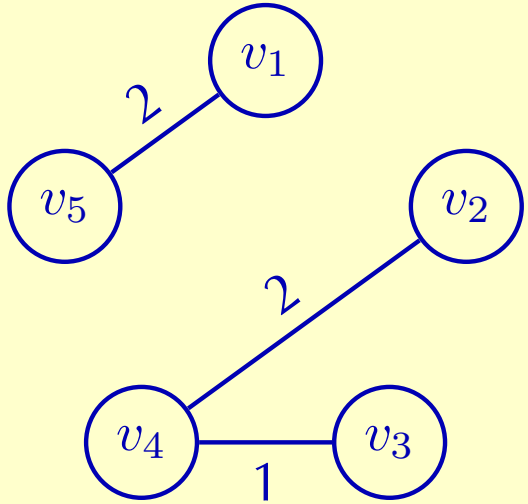
2. Dodajemy (v_1, v_5).



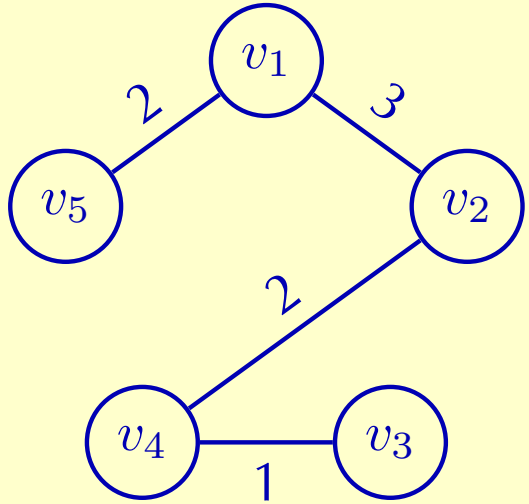
3. Dodajemy (v_2, v_4).



4. Nie dodajemy (v_2, v_3).



5. Dodajemy (v_1, v_2).



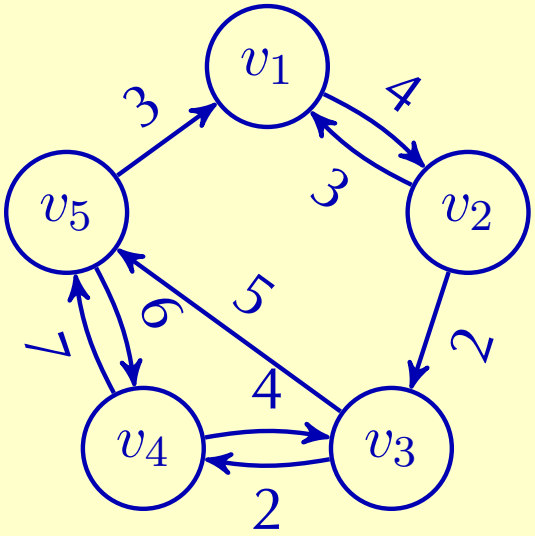
Algorytm Dijkstry

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.

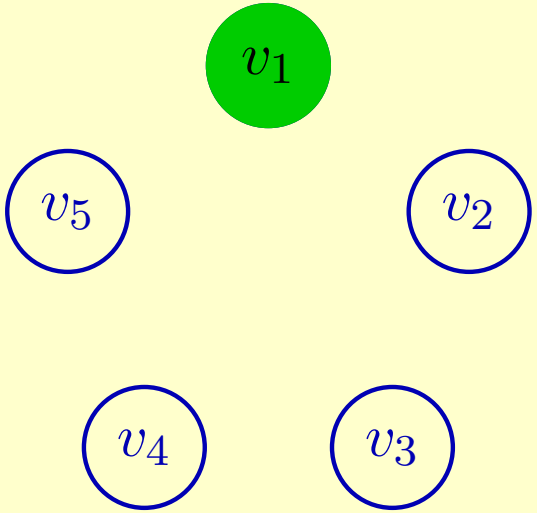
- $Y \leftarrow \{v\}$
- $F \leftarrow \emptyset$
- Tak długo jak $Y \neq V$:
  - wyznaczamy wierzchołek $y \in V - Y$, o minimalnej długości najkrótszej drogi z v poprzez wierzchołki z Y.
  - $Y \leftarrow Y \cup \{y\}$,
  - $F \leftarrow F \cup \{f\}$.

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

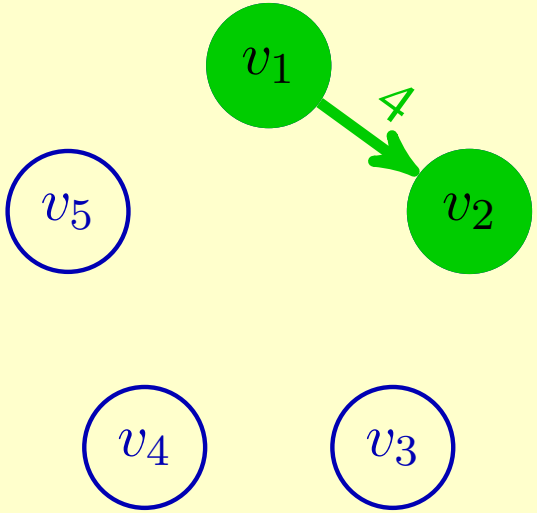
Algorytm Dijkstry – przykład



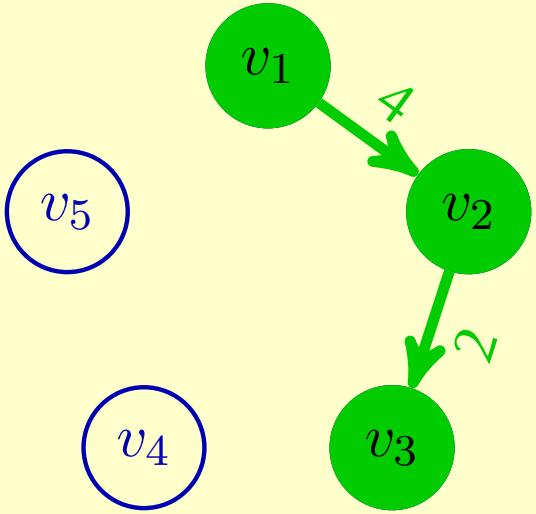
$Y = \{v_1\}$



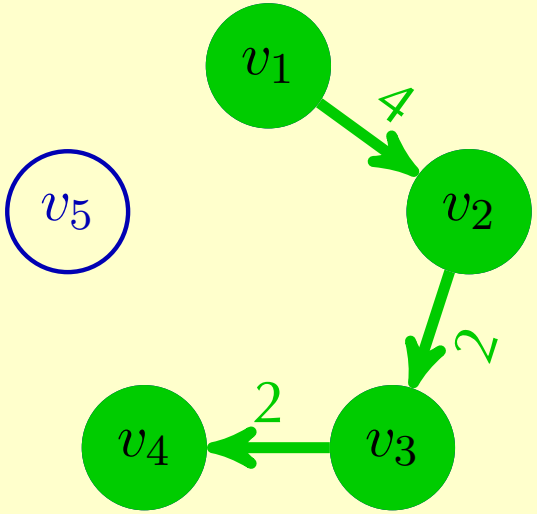
$v_2 \to Y,\ d(v_1, v_2) = 4$



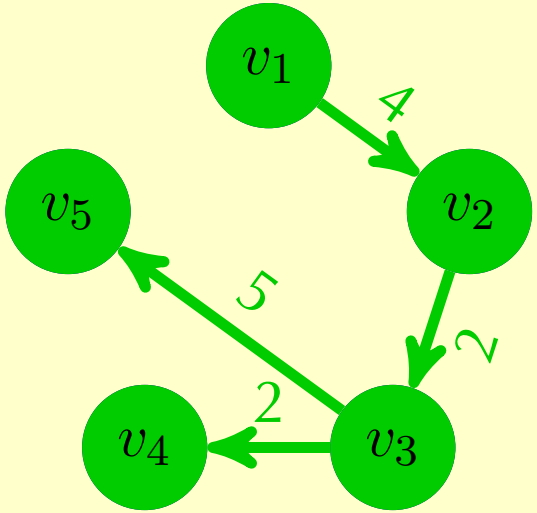
$v_3 \to Y,\ d(v_1, v_3) = 6$



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



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



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. 26) – 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*

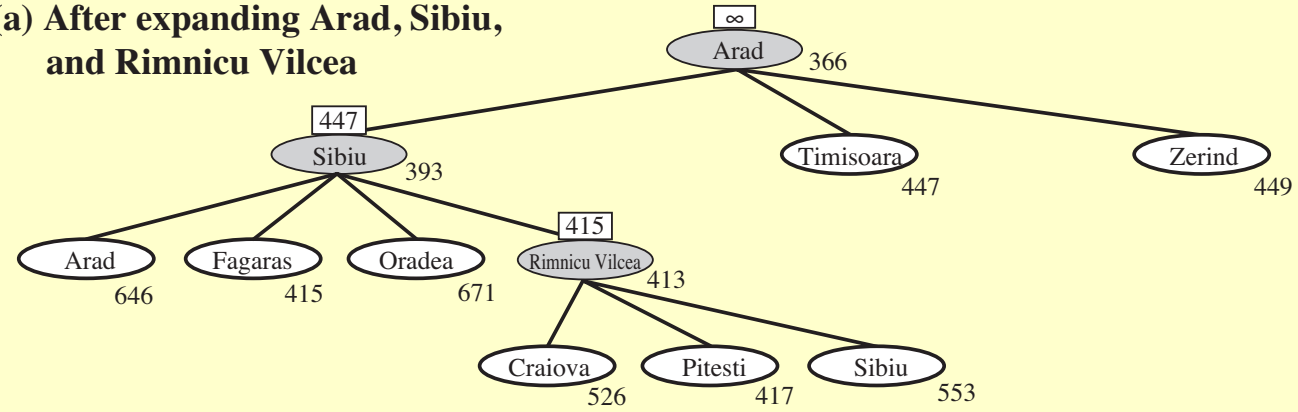
- Cel: ograniczenie zapotrzebowania na **pamięć**.
- IDA*- zwykły ID dla A*, ale **z limitem kosztów** $f = g + h$ **zamiast głębokości**.
- Próg f-kosztów – najmniejszy koszt węzła, przekraczający poprzedni próg odcięcia.

RBFS – recursive best-first search

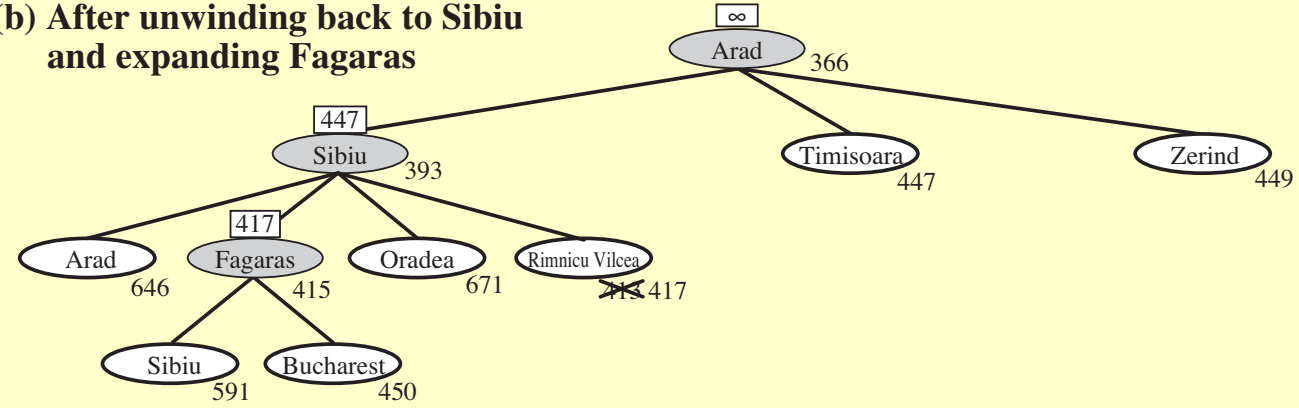
Prototype: ***RecursiveBestFS(P, node, fLimit)***
Input: Problem P, current node, f limit.
Output: A solution or failure and a new f-cost.

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

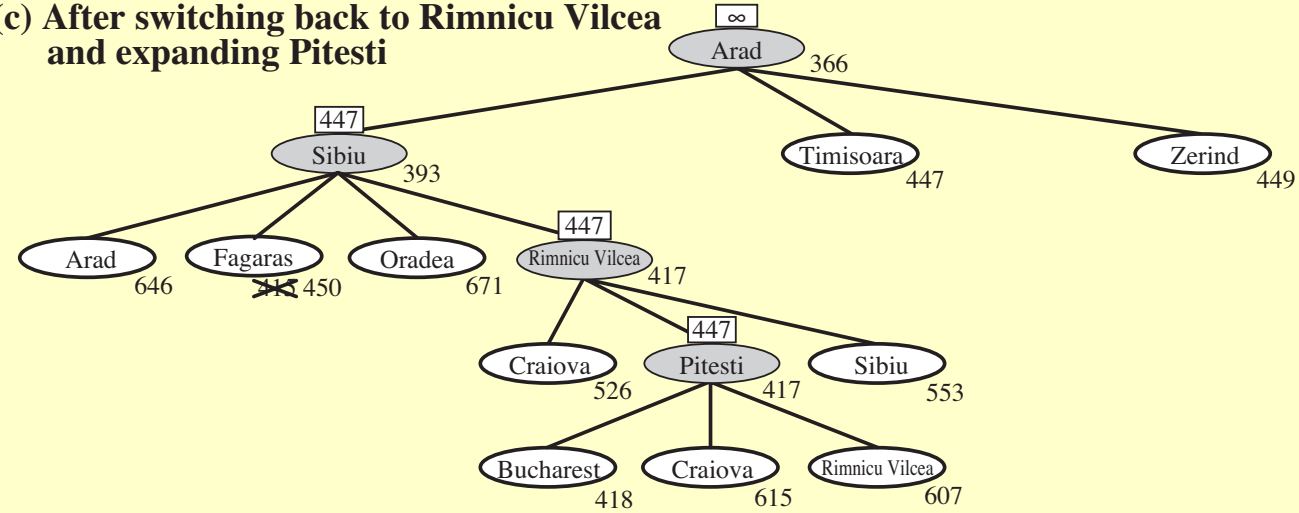
(a) After expanding Arad, Sibiu, and Rimnicu Vilcea



(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**.

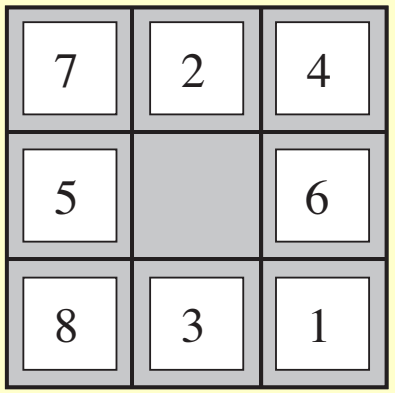
Szukanie wiązką (ang. *beam search*)

- 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**.

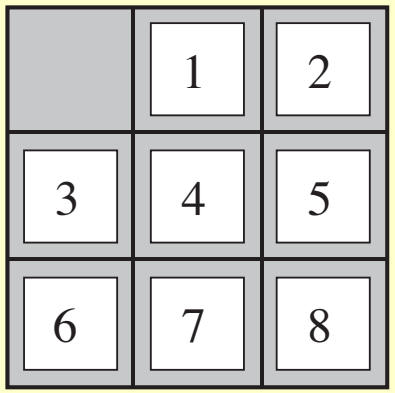
Heurystyki

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:

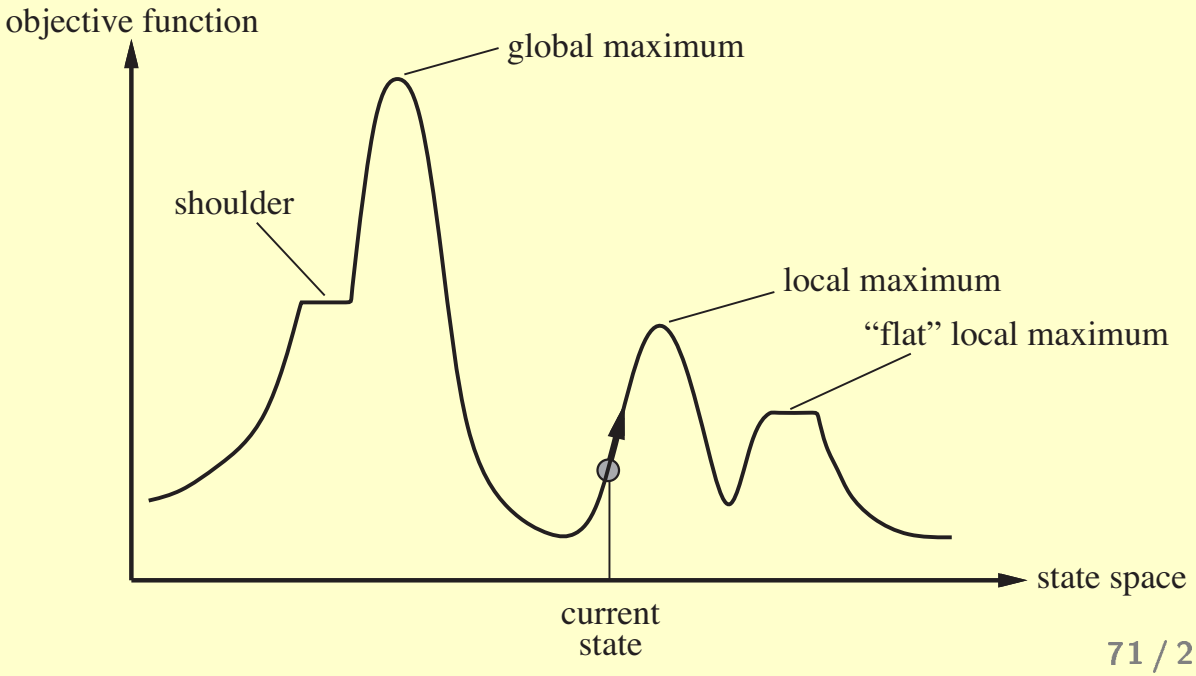
  - *poziomo lub pionowo o jedno pole,*
  - *pod warunkiem, że pole docelowe jest puste.*

- **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.

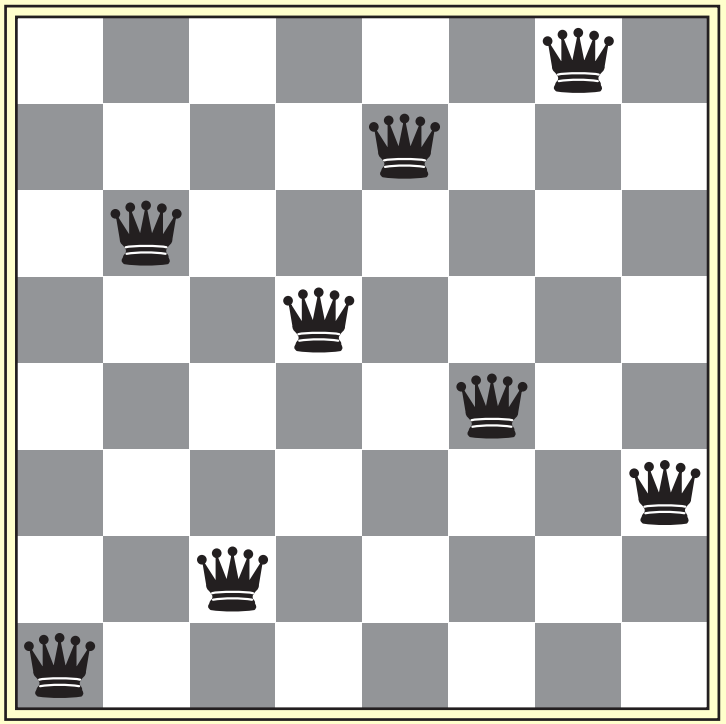
Prototype: ***HillClimbing(P)***
Input: Problem P, w tym funkcja maksymalizowana F.
Output: Stan lokalnego maksimum.

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

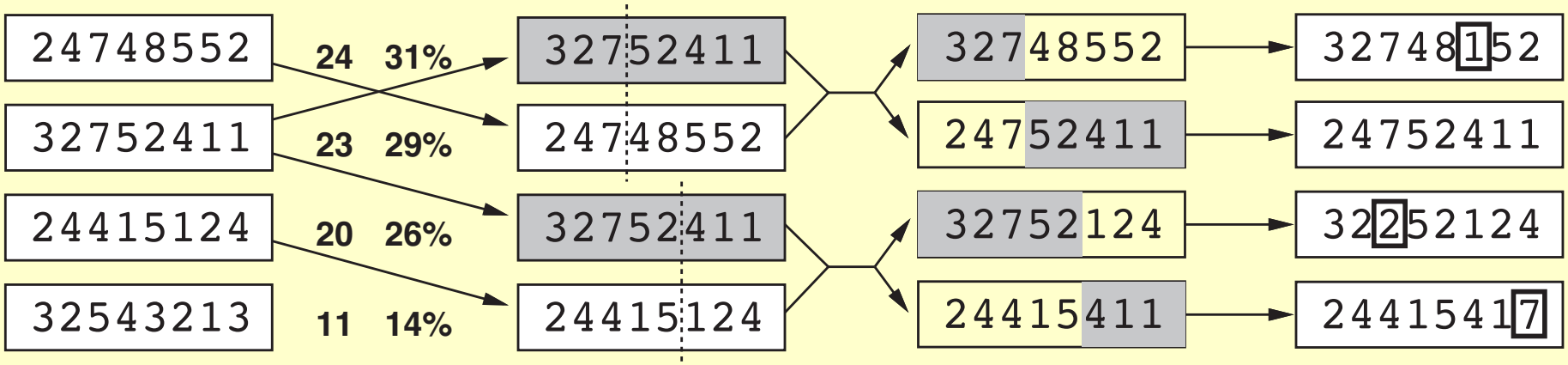
Inspiracje biologiczne

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

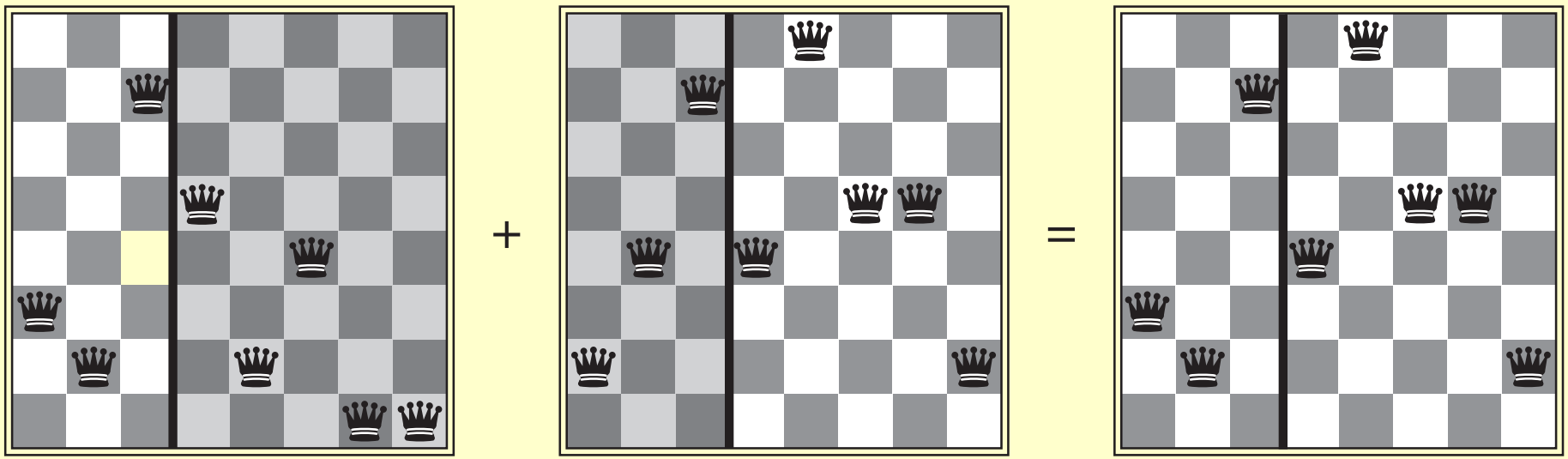
Schemat algorytmu:

- 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



(a) Initial Population | (b) Fitness Function | (c) Selection | (d) Crossover | (e) Mutation



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.

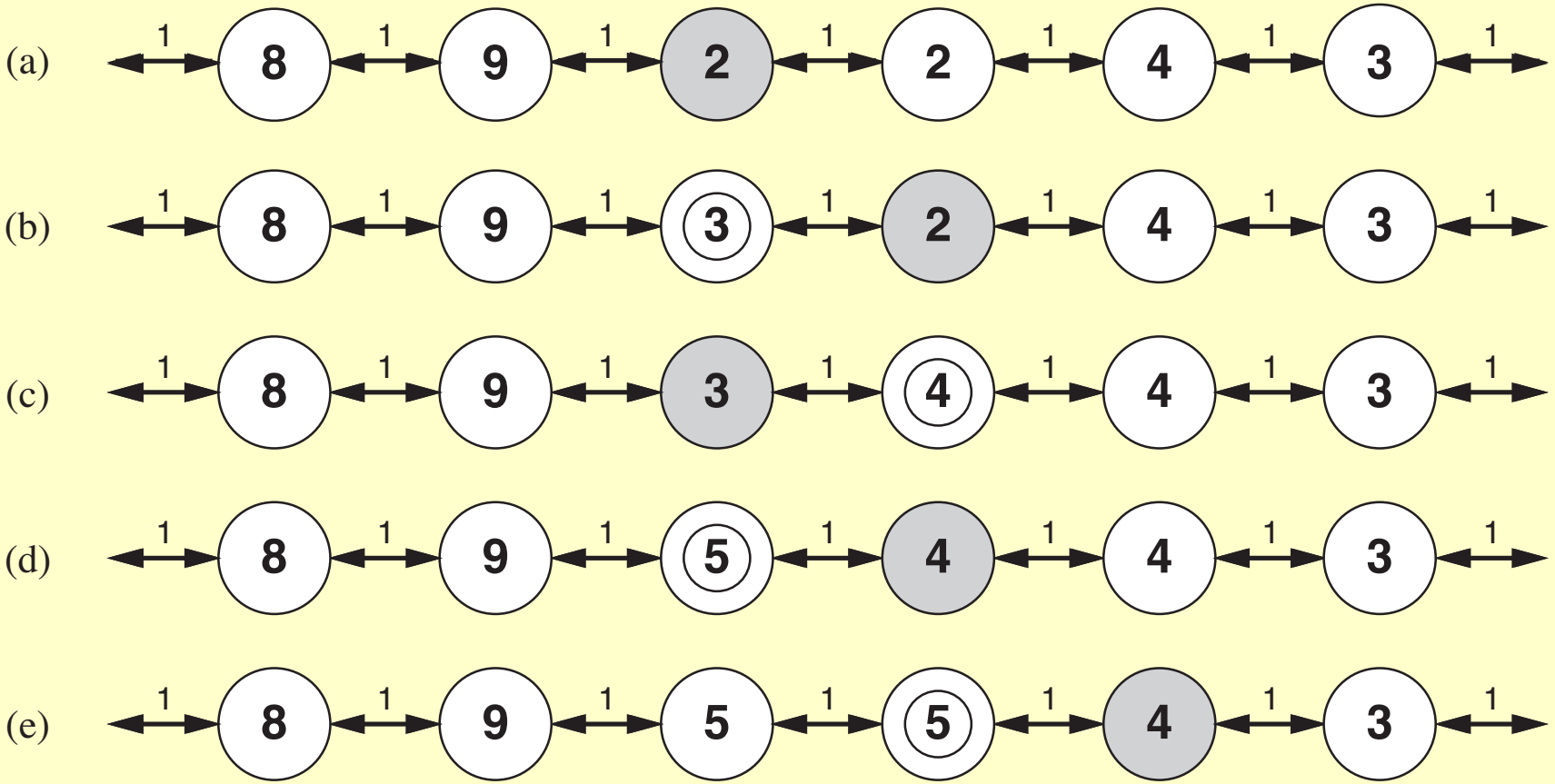
Metoda wspinaczki. Jak uciekać z minimów lokalnych?

- 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*)

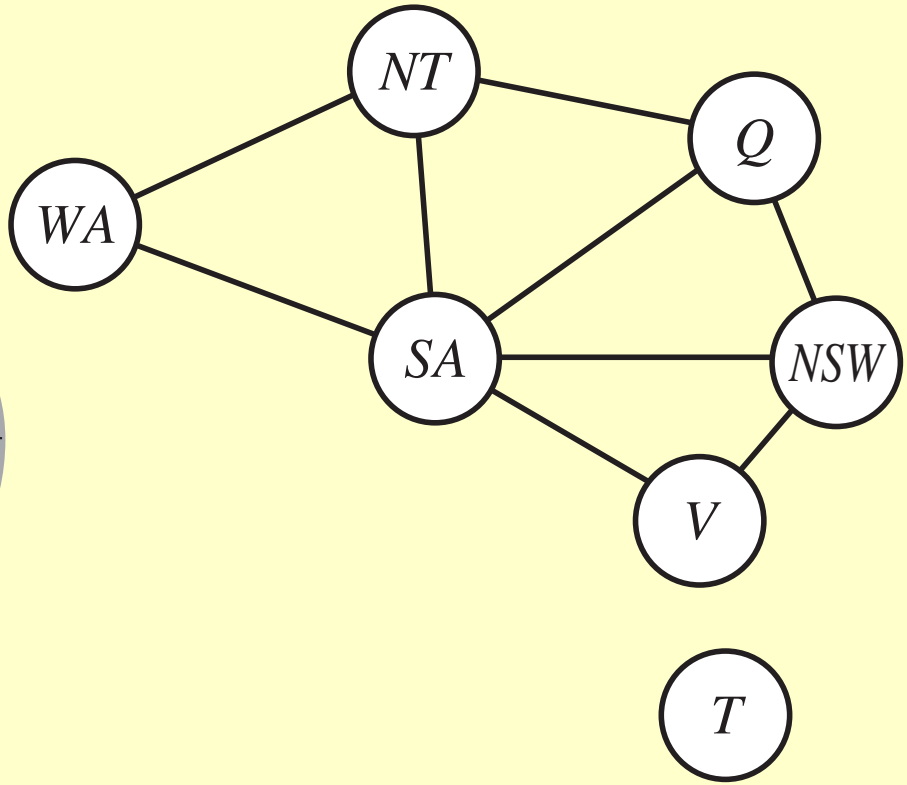
Wiedza o **wewnętrznej strukturze problemu** może pomóc w jego rozwiązywaniu.

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

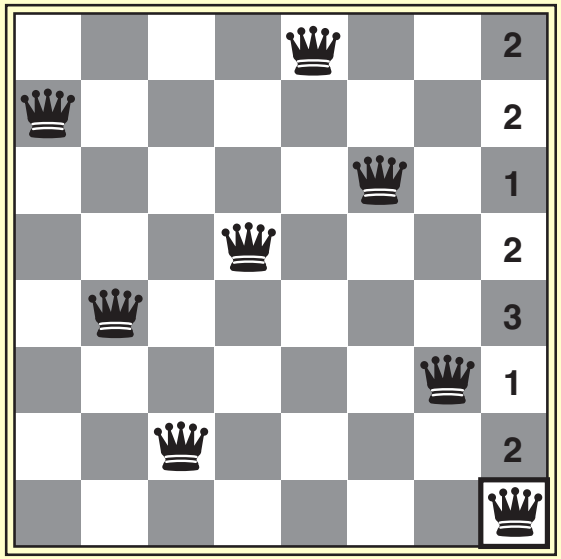
- **Pełna reprezentacja** i **zmiany** by spełniać ograniczenia
  - zmiana wartości **pojedynczych** zmiennych,
  - zamiana wartości **dwóch** zmiennych, itp.
- Naturalna heurystyka: wybór **wartości o jak najmniejszej liczbie konfliktów** (ang. *min-conflicts*).

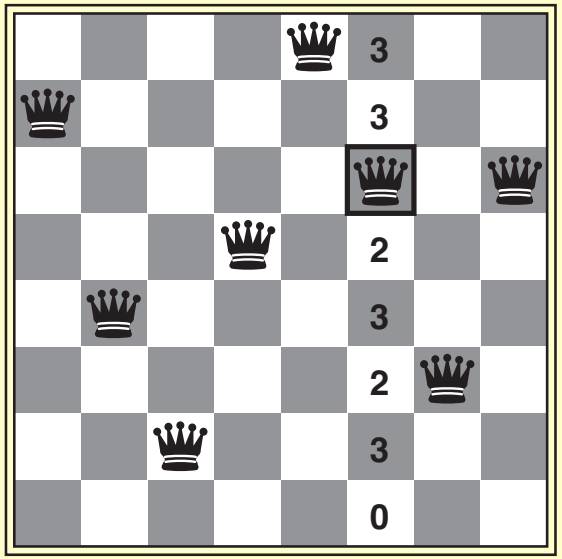
Prototype: ***MinConflicts(CSP, H, N)***
Input: Problem CSP, miara min-conflicts H, liczba N iteracji.
Output: Stan lokalnego minimum.

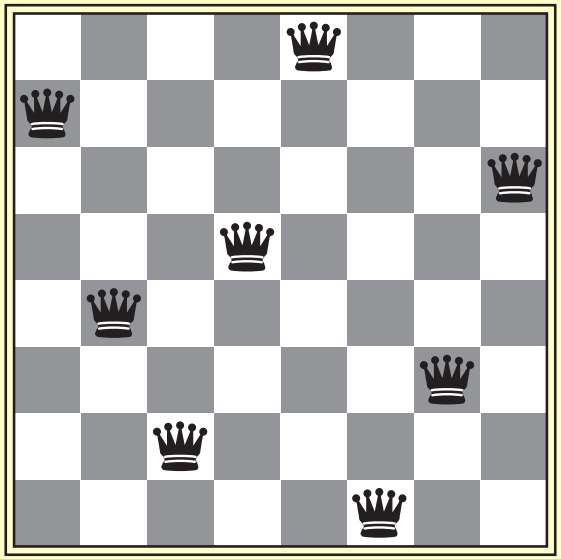
1. stan $\leftarrow$ (losowy pełny) stan początkowy
2. **for** $t = 1$ **to** N **do**
   1. **if** stan jest rozwiązaniem **then return** stan
   2. X $\leftarrow$ wylosowana zmienna z konfliktem
   3. Z $\leftarrow$ zmiana X minimalizująca miarę H
   4. stan $\leftarrow$ stan po zmianie Z
3. **return**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

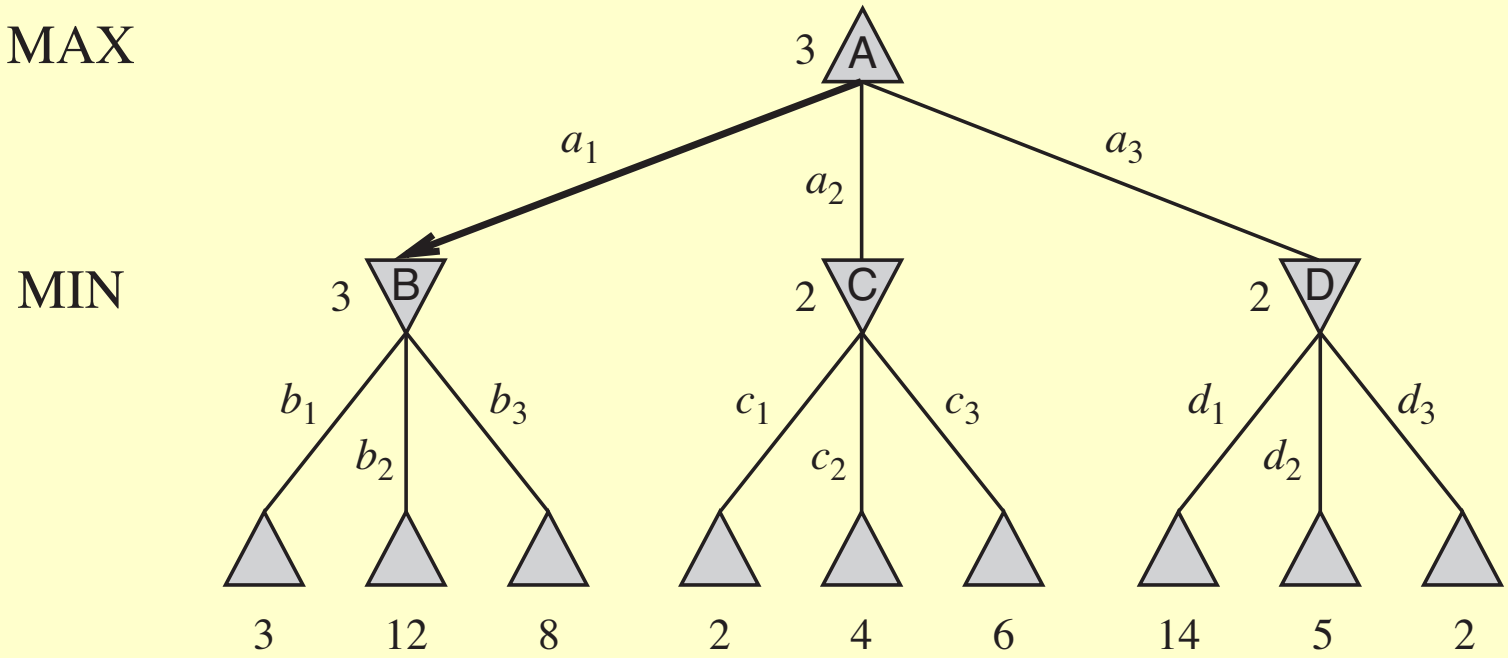
Gry - szukanie z oponentem

- 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):

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

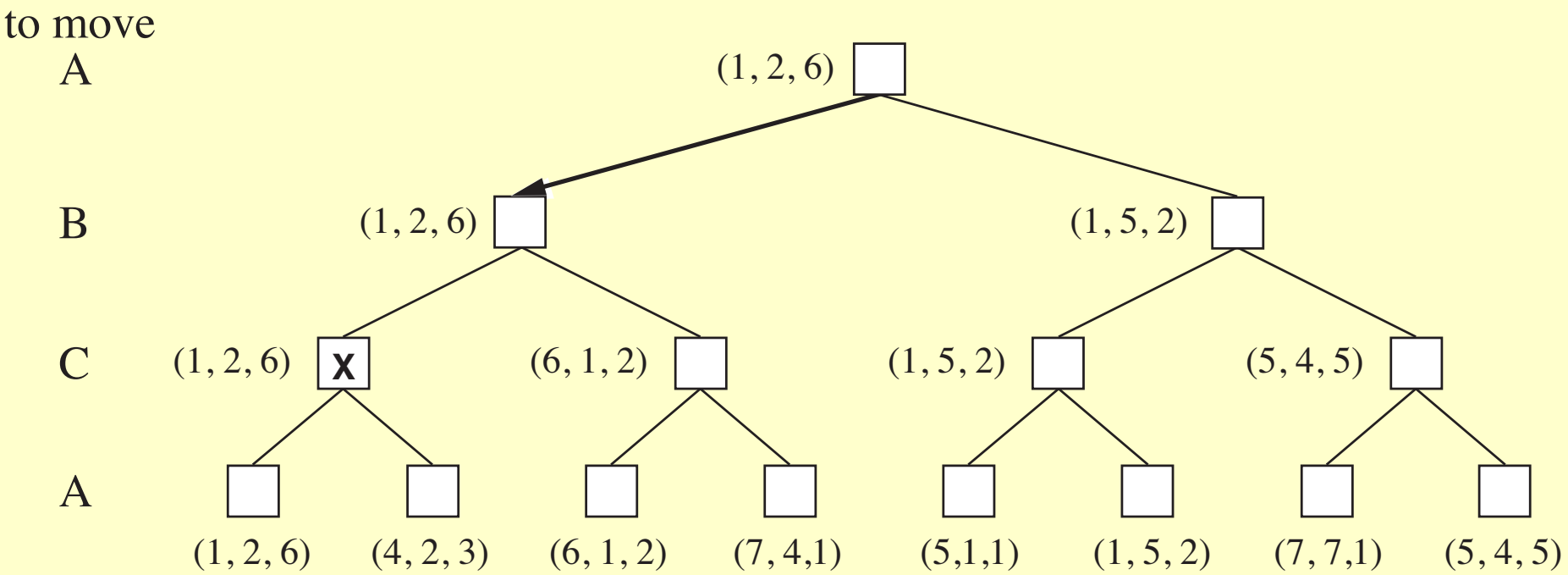
Minimax

Drzewo gry:



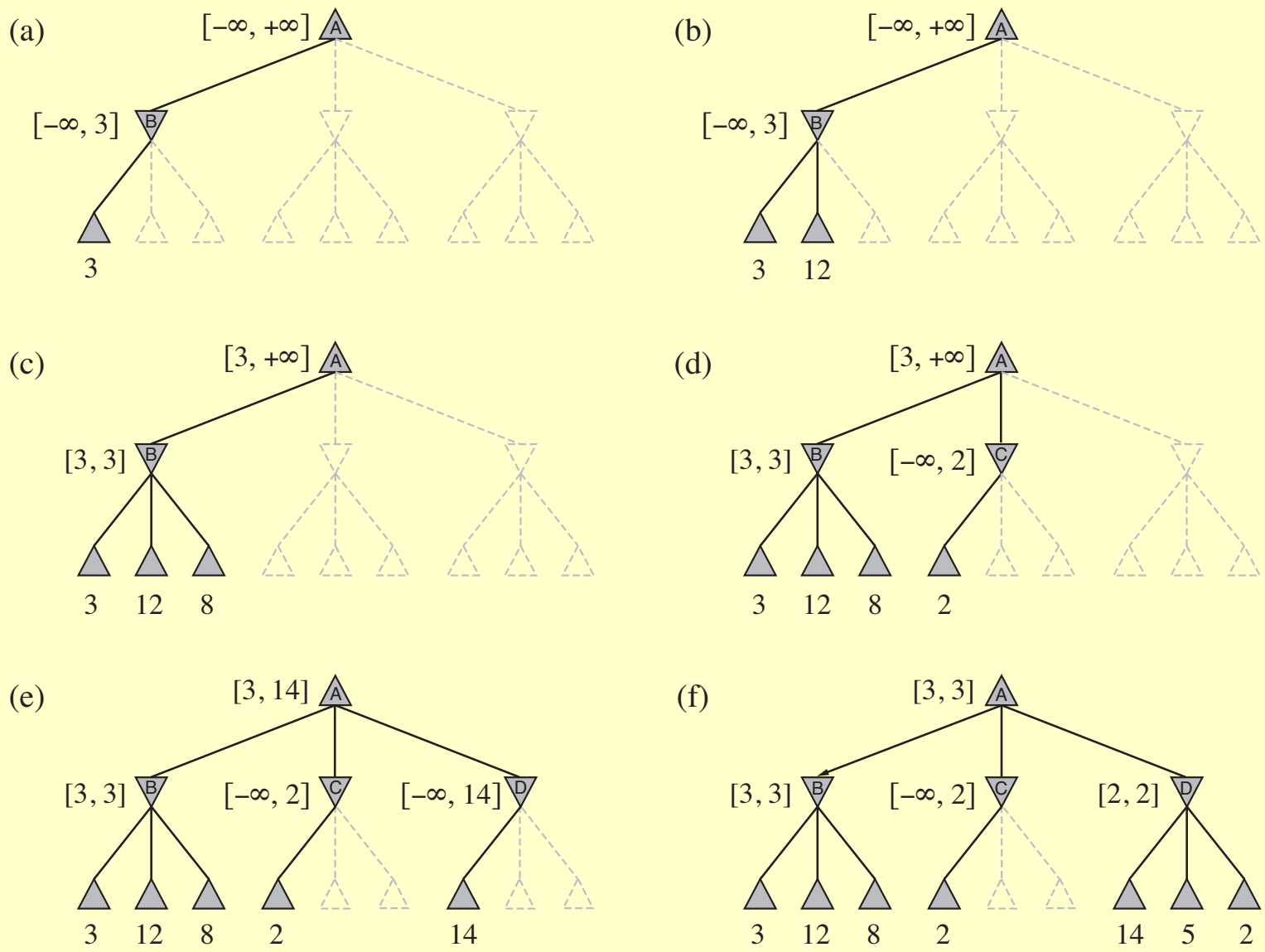
- Złożoność $O(b^m)$, gdzie m to maksymalna głębokość rozwijanych węzłów.

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

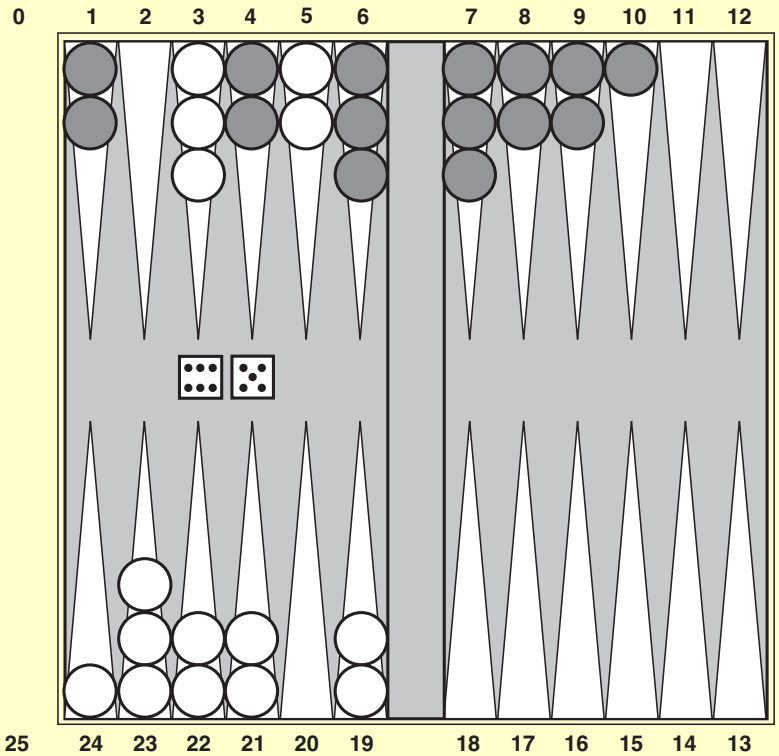
- **Parametry:**
  - α – maksymalna wartość zaobserwowana na ścieżce dla MAX,
  - β – minimalna wartość zaobserwowana na ścieżce dla MIN.
- Parametry są **aktualizowane** w trakcie szukania.
- **Kolejność analizy następników** wpływa na efektywność metody.
- Przy **optymalnej** kolejności rozwijanych 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.

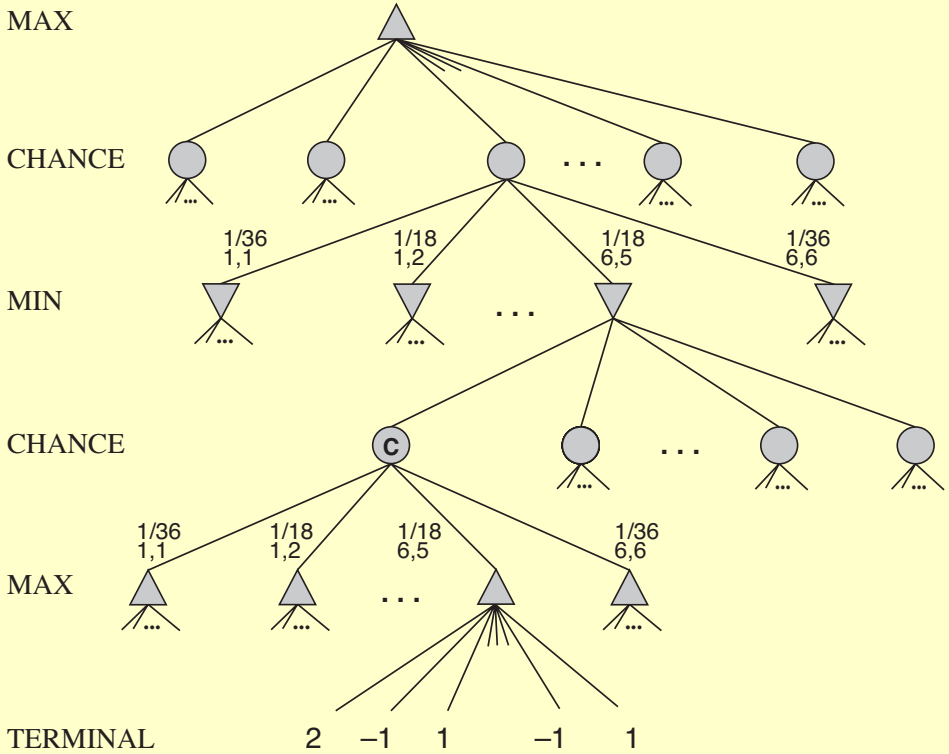
Przybliżone oceny stanu gry

- 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 $\alpha - \beta$.

Język Scheme – początek

Paradygmat języków funkcyjnych

- podejście **deklaratywne**, nie imperatywne!
- myślenie **matematyczne**, kategoriami **funkcji**,
- funkcja **odwzorowuje argumenty na wynik**,
- **interakcja** – użytkownik podaje wyrażenia, interpreter je wylicza i przedstawia wartości,
- „skażenia” podejściem imperatywnym,
- łączenie języków imperatywnych i deklaratywnych.

Racket

System Racket (wcześniej PLT Scheme) i środowisko **DrRacket** – http://racket-lang.org/

- **edytor** tekstów źródłowych,
- obsługa **interpretatora**,
- **debugger**,
- różne języki.

Podstawy języka *scheme* (1)

Scheme to dialekt LISPu

- LISt Processing,
- programy w schemie to wyrażenia do wyliczenia, w szczególności wywołania funkcji,
- pierwszy prosty program: "Hello world!",
- podstawowe **złożone struktury** danych to **listy**,
- lista zapisywana jako ciąg jej elementów oddzielonych znakami–odstępami i ujętych w nawiasy:
  (element1 element2 ... elementn)
- wywołania funkcji jako **listy**: funkcja i jej kolejne argumenty,
  (funkcja arg1 arg2 ... argk)
- przykład wywołania funkcji: (silnia 5).
- **zagnieżdżone** (często głęboko) wywołania,

Podstawy języka *scheme* (2)

- Proste wywołania – zawsze prefiksowo:

```
1 (+ 2 (* 3 4))
2 (define x 17)
3 (* x x)
4 (- (* b b) (* 4 a c))
5 (< 7 5)
```

- Pierwszy element to funkcja do wywołania:

```
1 (list 9 8 7 a b c)
2 (szukaj-w-glab pocz kon)
3 ((if dodawanie + -) 2 3)
```

Specyfika języka *Scheme* (1)

- identyfikatory – ciągi liter, cyfr i znaków dodatkowych, rozpoczynający się literą.
- znaki dodatkowe: `! % $ & * + - . / : < = > ? @ ^ _ ~`
- Konwencja:
  - `!` używany jest na końcu nazw funkcji z efektami ubocznymi, np. `set!`,
  - `?` używany jest na końcu nazw predykatów (funkcji o wartościach logicznych), np. `number?`,
  - `->` używany jest w nazwach funkcji realizujących konwersje, np. `string->number`,
- Literały:
  - funkcją: (quote a)
  - apostrofem: 'a
  - stałe numeryczne, łańcuchowe, znakowe i logiczne pozostają sobą: '78 $\Longrightarrow$ 78.

Specyfika języka *Scheme* (2)

- Zmienne, wartości
  - typy związane z wartościami, nie zmiennymi (*typowanie dynamiczne*),
  - dla danego obiektu, tylko jeden z predykatów może być spełniony:
    boolean? pair? symbol? **number**? **char**? **string**? **vector**? port? procedure?,
  - stałe logiczne #f #t,
  - nie zwalnia się obiektów, pamięć może być użyta powtórnie gdy jest pewność, że nikt się o obiekt nie upomni,
  - przykład definiowania zmiennych: (**define** s10 (silnia 10))
- Argumenty są przekazywane zawsze przez wartość. Wyrażenia definiujące argumenty są zawsze wyliczane przed wywołaniem funkcji.

Specyfika języka *Scheme* (3)

- Rekurencja ogonowa (ang. *tail recursion*) – redukcje wywołań z udziałem stosu.
- Programowanie z kontynuacjami.

```
1 ;;; klasyczna rekurencyjna silnia
2 (define (silnia n)
3   (if (< n 2) 1 (* n (silnia (- n 1)))))
4 ;;; silnia w stylu kontynuacyjnym (CPS−continuation passing style)
5 (define (silniaK n k)
6   (if (= n 1)
7       (k 1)
8       (silniaK (- n 1) (lambda (x) (k (* n x))))))
```

Elementarne funkcje języka *Scheme* (1)

- Definicje funkcji:

```
1 (lambda (x) (* x x))
2 (define kwadrat (lambda (x) (* x x)))
3 (define (kwadrat x) (* x x))
4
5 (lambda (x y) (+ x y))
6
7 (define bezDwoch (lambda (x y . z) z))
```

- zmiana wartości zmiennej

```
1 (set! x (sqrt delta))
```

Elementarne funkcje języka *Scheme* (2)

- Podstawowa konstrukcja warunkowa:

```
1 (if ⟨warunek⟩ ⟨wartość-dla-true⟩ ⟨wartość-dla-false⟩)
2 (if ⟨warunek⟩ ⟨wartość⟩)
```

Warunek jest fałszywy jeśli przyjmuje wartość #f, w przec. przyp. jest prawdziwy.
Uwaga! W innych dialektach Lispu, często lista pusta '() jest również uznawana za fałsz.

Przykłady:

```
1 (define (minimum x y) (if (< x y) x y))
2 (if (argumentWDziedzinie x) (funkcja x))
```

W postaci z jedną wartością, w przypadku niespełnionego warunku wynikiem jest #undefined.

Elementarne funkcje języka *Scheme* (3)

- Konstrukcja wielowarunkowa:

  (**cond** ⟨*klauzula*$_1$⟩ ⟨*klauzula*$_2$⟩ ...)

  gdzie każda ⟨*klauzula*$_i$⟩ ma jedną z postaci

  (⟨*test*⟩ ⟨*wyrażenie*$_1$⟩ ...)
  (⟨*test*⟩ => ⟨*wyrażenie*⟩ ...)

  lub w przypadku ostatniego wyrażenia

  (**else** ⟨*wyrażenie*$_1$⟩ ⟨*wyrażenie*$_2$⟩ ...)

  Przykład:

```
1 (cond ((< delta 0) 'zero)
2       ((eq? delta 0) 'jeden)
3       (else 'dwa)
4 )
```

Elementarne funkcje języka *Scheme* (4)

- Wyliczanie przypadków:

  (**case** ⟨*klucz*⟩ ⟨*klauzula*$_1$⟩ ⟨*klauzula*$_2$⟩ ...)

  gdzie ⟨*klucz*⟩ to dowolne wyrażenie, a każda ⟨*klauzula*$_i$⟩ ma postać

  ((⟨*wartość*$_1$ ...⟩) ⟨*wyrażenie*$_1$⟩ ⟨*wyrażenie*$_2$⟩ ...)

  lub w przypadku ostatniego wyrażenia

  (**else** ⟨*wyrażenie*$_1$⟩ ⟨*wyrażenie*$_2$⟩ ...)

  Przykład:

```
1 (define dniWMies (lambda (n)
2   (case n
3     ((2) 28)
4     ((4 6 9 11) 30)
5     (else 31))))
```

Elementarne funkcje języka *Scheme* (5)

- Funkcje logiczne

```
1 (and ⟨test1⟩ ⟨test2⟩ ...)
2 (or ⟨test1⟩ ⟨test2⟩ ...)
3 (not ⟨obiekt⟩)
```

  - Testy są sprawdzane od lewej do prawej.
  - Zwracana jest wartość pierwszego testu przesądzającego o wyniku.

Przykłady:

```
1 (and (< x y) (< y z))
2 (and (= x y) (= y z) (= z v))
3 (and 1 2 'c)                    ⟹ c
4 (or (< x 0) (> x n))
5 (not "napis")
6 (not #f)
```

Elementarne funkcje języka *Scheme* (6)

- Wiązanie zmiennych pomocniczych

  (**let** ⟨*wiązania*⟩ ⟨*wyrażenie-wynik*⟩)

  gdzie ⟨*wiązania*⟩ mają postać

  ((⟨*zmienna*$_1$⟩ ⟨*wyrażenie*$_1$⟩) ...)

  Wyrażenia-wartości zmiennych są wyznaczane w nieokreślonej kolejności.
  Przykład:

```
1 (let ((x 7) (y 8)) (sqrt (+ (* x x) (* y y))))
```

- Wiązania z ustaloną kolejnością – **let***

```
1 (let* ((x 7) (x2 (* x x)) (x3 (* x2 x))) (+ x x2 x3))
```

- Wiązania z rekurencją – letrec

Predykaty porównujące (1)

- Operator = działa tylko dla liczb.
- Predykaty porównujące **eq?**, **eqv?**, **equal?** – relacje równoważności.

```
1 (eq? ⟨obiekt1⟩ ⟨obiekt2⟩)
2 (eqv? ⟨obiekt1⟩ ⟨obiekt2⟩)
3 (equal? ⟨obiekt1⟩ ⟨obiekt2⟩)
```

- Stopniowanie rozróżniania:

$$\text{(eq? x y)} \Rightarrow \text{(eqv? x y)} \Rightarrow \text{(equal? x y)}$$

- Standard języka nie określa w pełni tych relacji.

Predykaty porównujące (2)

- Działanie **eqv?**
  - zgodne z intuicjami dla liczb, typu logicznego, symboli, znaków, pustych list,
  - dla par, wektorów i napisów, prawda gdy ta sama lokalizacja w pamięci,
  - dla funkcji również równość referencji,
  - nie wszystko określone w standardzie – różnice w implementacjach.
- **eq?** bardziej restrykcyjny niż **eqv?** – może być implementowany jako porównanie wskaźników.
- **equal?**
  - najmniej restrykcyjny,
  - porównuje rekurencyjnie pary, wektory i łańcuchy,
  - reguła kciuka – obiekty uznawane za równe, gdy „drukują się tak samo".

Przykłady definicji funkcji

```
1  (define (fkw a b c)
2      (let ((delta (- (* b b) (* 4 (* a c)))))
3        (cond ((< delta 0) '())
4              ((eq? delta 0) (list (/ (- b) (* 2 a))))
5              (else (list (/ (- (- b) (sqrt delta)) (* 2 a))
6        (/ (+ (- b) (sqrt delta)) (* 2 a))))
7  )))
8
9  (define potega (lambda (a n)
10     (if (< n 1) 1
11       (let* ((n2 (quotient n 2)) (p (potega a n2)))
12         (if (even? n)
13             (* p p)
14             (* a (* p p))
15 )))))
```

Świat liczb (1)

- Dość nieuporządkowany w implementacjach Lispu.
- Typy liczbowe:

  (integer? x) ⇒ **(rational?** x) ⇒ **(real?** x) ⇒
  (complex? x) ⇒ **(number?** x)

- Liczby dokładne (ang. *exact*) i niedokładne (ang. *inexact*)
- Operacje zwracające liczby dokładne:

  \+ − ∗ **quotient remainder modulo max min abs**
  numerator denominator gcd lcm floor ceiling
  truncate round rationalize expt

- Liczby niedokładne reprezentowane zwykle zmiennopozycyjnie (IEEE 32-bity lub 64 bity).

Świat liczb (2)

Podstawowe funkcje matematyczne:

```
 1 (+ z1 ...)
 2 (* z1 ...)
 3 (- z1 ...)
 4 (/ z1 ...)
 5 (= z1 z2 z3 ...)
 6 (< z1 z2 z3 ...)
 7 (> z1 z2 z3 ...)
 8 (<= z1 z2 z3 ...)
 9 (>= z1 z2 z3 ...)
10 (zero? z)
11 (positive? x)
12 (negative? x)
13 (odd? n)
14 (even? n)
15 (min x1 x2 ...)
16 (max x1 x2 ...)
17 (abs x)
18 (quotient n1 n2)
19 (remainder n1 n2)
20 (modulo n1 n2)
21 (gcd n1 ...)
22 (lcm n1 ...)
23 (floor x)
24 (ceiling x)
25 (truncate x)
26 (round x)
27 (sqrt z)
28 (expt z1 z2)
29 (exp z)
30 (log z)
31 (sin z)
32 (cos z)
33 (tan z)
34 (asin z)
35 (acos z)
36 (atan z)
37 (atan y x)
38 (exact->inexact z)
39 (inexact->exact z)
40 (number->string z)
41 (number->string z radix)
42 (string->number str)
43 (string->number str rdx)
```

„Zachwaszczenie" imperatywnością

Uwaga! Myślenie imperatywne w świecie funkcyjnym jest niewskazane i rzadko wygodne!

- Zmiana wartości zmiennych

```
1 (set! ⟨zmienna⟩ ⟨wyrażenie⟩)
```

- Sekwencje wyliczeń

```
1 (begin ⟨wyrażenie_1⟩ ⟨wyrażenie_2⟩ ...)
```

Przydatne do wyświetlania informacji w trakcie obliczeń

```
1 (begin (display "wywołanie silni dla ") (display n)
2        (silnia n))
```

- Obliczenia iteracyjne

```
1 (do ((⟨zmienna_1⟩ ⟨pocz_1⟩ ⟨krok_1⟩) ...)
2       (⟨test⟩ ⟨wyrażenie⟩ ...)
3       ⟨polecenie⟩ ...)
```

Pary i listy (1)

- **Para** – rekord z dwoma polami dostępnymi przez funkcje **car** i **cdr**
  - notacja z kropkami (w stylu drzew), '(3 . 4)
  - konstrukcja par funkcją **cons**,
  - ustawianie pól przez funkcje set−car! i set−cdr! – chwasty imperatywności?
- **Lista** – definicja rekurencyjna
  - **lista pusta** '() jest listą,
  - każda **para, której drugi element jest listą**, jest listą.
- **Głowa** i **ogon** listy – elementy pary.
- Przykłady list:

```
1 '(2 4 a b)
2 '(2 . (4 . (a . (b . ()))))
```

Pary i listy (2)

Podstawowe operacje na parach i listach:

1. (pair? ⟨*obiekt*⟩)
2. (**list**? ⟨*obiekt*⟩)
3. (**null**? ⟨*obiekt*⟩)
4. (**cons** ⟨$obiekt_1$⟩ ⟨$obiekt_2$⟩)
5. (**car** ⟨*para*⟩)
6. (**cdr** ⟨*para*⟩)
7. (set−car! ⟨*para*⟩ ⟨*obiekt*⟩)
8. (set−cdr! ⟨*para*⟩ ⟨*obiekt*⟩)
9. (caar ⟨*para*⟩)
10. (cadr ⟨*para*⟩)
11. ...
12. (cdddar ⟨*para*⟩)
13. (cddddr ⟨*para*⟩)
14. (**list** ⟨$obiekt_1$⟩ ...)
15. (**length** ⟨*lista*⟩)
16. (**append** ⟨*lista*⟩ ...)
17. (**reverse** ⟨*lista*⟩)
18. (list−tail ⟨*lista*⟩ k)
19. (list−ref ⟨*lista*⟩ k)
20. (memq ⟨*obiekt*⟩ ⟨*lista*⟩)
21. (memv ⟨*obiekt*⟩ ⟨*lista*⟩)
22. (**member** ⟨*obiekt*⟩ ⟨*lista*⟩)

Zadania dla początkujących

1. Silnia,
2. Rozwiązywanie równań kwadratowych,
3. Operacje na listach:
   - długość listy,
   - zawieranie elementu,
   - k-ty element,
   - mapowanie, filtrowanie,
   - usuń k pierwszych, k ostatnich,
   - weź k pierwszych, k ostatnich,
   - połączenie dwóch list,
   - odwracanie list.
4. Metody szukania:
   - w głąb, wszerz,
   - best first, A*,
   - lokalne szukanie wiązką,
   - algorytmy genetyczne.

Agenty oparte na wiedzy (1)

- **Baza wiedzy**
  - to **zbiór zdań** w odpowiednim języku logiki
  - może pochodzić **od ekspertów** z dziedziny,
  - może być **rozszerzana** w trakcie pracy agenta,
  - może być budowana przy użyciu metod **uczenia maszynowego**.
- **Metody wnioskowania** (ang. *inference*)
  - odpowiednio sformułowane **pytania**,
  - **odpowiedzi** na podstawie bazy wiedzy,
  - mechanizmy **wnioskowania**.
  - podejścia **proceduralne**, **imperatywne** i **mieszane**.
- **Agenty oparte na wiedzy** (ang. *knowledge-based agents*).

Agenty oparte na wiedzy (2)

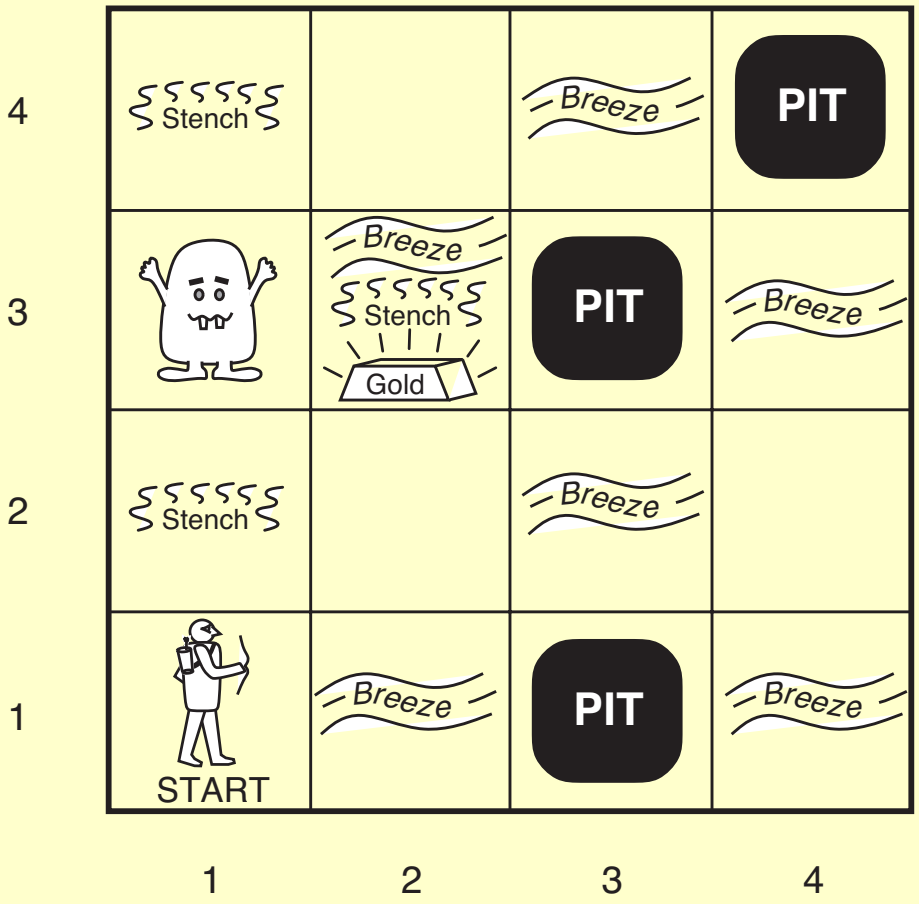
Prototype: ***AgentOpartyNaWiedzy(BW)***
Input: Baza wiedzy BW.

1. **while** nie koniec **do**
    1. S ← Spostrzeżenie()
    2. DecyzjaOpartaNaWiedzy(BW,S)

Prototype: ***DecyzjaOpartaNaWiedzy(BW,S)***
Input: Baza wiedzy BW, spostrzeżenie S.
Output: Działanie.

1. BW.DodajSpostrzeżenie(S)
2. działanie ← BW.SugerowanaAkcja()
3. BW.DodajInformacjęOSkutkach(działanie)
4. **return** działanie

Świat wumpusa (1)



Świat wumpusa (2)

- **Środowisko**:
  - jaskinia o 16 (4×4) pomieszczeniach,
  - agent startuje w [1,1] zwrócony w prawo,
  - położenia wumpusa i złota losowe, rozkład jednostajny poza [1,1],
  - przepaści z p-stwem 0,2 wszędzie poza startem
- **Sygnały z czujników**:
  - smród wumpusa w pomieszczeniach sąsiadujących,
  - powiew w sąsiedztwie przepaści,
  - lśnienie złota tylko w jego pomieszczeniu,
  - odbicie od ściany na brzegach jaskini,
  - wrzask wumpusa trafionego strzałą (słychać w całej jaskini).
- **Działania**: obrót w lewo o 90°, w prawo o 90°, ruch naprzód, zabranie złota, strzał (przez wszystkie pomieszczenia na wprost, jedna strzała).
- **Miara jakości**: suma nagród i kar: +1000 za złoto, -1000 śmierć od wumpusa lub w przepaści, -1 za każde działanie i -10 za użycie strzały.

Systemy logiczne

- **Syntaktyka** – wnioskowanie jako przetwarzanie wyrażeń
  - **wyrażenia** konstruowane wg ścisłych zasad,
  - **algorytmy wnioskujące** i relacja **dowodliwości**, wnioskowania (ang. *inference*)

$$BW \vdash_A \alpha$$

- **Semantyka** – znaczenie przypisywane zdaniom
  - **modele** (pot. „możliwe światy”),
  - **prawdziwość** zdań,
  - relacja **logicznej konsekwencji**

$$\alpha \models \beta$$

  – *w każdym modelu, w którym α jest prawdziwe, również β jest prawdziwe,*

- System formalny jest
  - **zgodny** (ang. *sound*), jeśli każde zdanie dowodliwe (przy jego użyciu) z określonych przesłanek jest ich logiczną konsekwencją,
  - **zupełny** (ang. *complete*), jeśli potrafi dowieść każdego zdania wynikającego logicznie z przesłanek.

Syntaktyka

- Zdania **atomowe** i **symbole** zdaniowe.
- Zdania **złożone** konstruowane z użyciem **spójników** logicznych ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$).
- Notacja Backusa-Naura:

$$
\begin{aligned}
\mathit{Zdanie} &\rightarrow \mathit{ZdanieAtomowe} \mid \mathit{ZdanieZłożone} \\
\mathit{ZdanieAtomowe} &\rightarrow \boldsymbol{Prawda} \mid \boldsymbol{Fałsz} \mid \mathit{Symbol} \\
\mathit{Symbol} &\rightarrow \boldsymbol{P} \mid \boldsymbol{Q} \mid \boldsymbol{R} \mid \ldots \\
\mathit{ZdanieZłożone} &\rightarrow \neg \mathit{Zdanie} \\
&\quad \mid (\mathit{Zdanie} \wedge \mathit{Zdanie}) \\
&\quad \mid (\mathit{Zdanie} \vee \mathit{Zdanie}) \\
&\quad \mid (\mathit{Zdanie} \Rightarrow \mathit{Zdanie}) \\
&\quad \mid (\mathit{Zdanie} \Leftrightarrow \mathit{Zdanie})
\end{aligned}
$$

- Nawiasy można eliminować stosując **priorytety** i **łączność**.

Semantyka

- Różne **modele** powstają przez różne **wartościowania** – **przypisania** prawdy bądź fałszu symbolom zdaniowym.
- Prawda bądź fałsz pozostałych zdań:
  - ***Prawda*** jest prawdziwe w każdym modelu,
  - ***Fałsz*** jest nieprawdziwe w każdym modelu,
  - Prawdziwość zdań zbudowanych z użyciem spójników jest określona wg tabeli:

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
0	0	1	0	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	0
1	1	0	1	1	1	1

gdzie 1 oznacza prawdę, a 0 fałsz.

- Bazę wiedzy można zapisać równoważnie jako jedno zdanie (koniunkcję elementów bazy).

Systemy formalne

- **Spełnialność**
  Model M spełnia zdanie α wtw gdy α jest prawdziwe w M.
  - Problem sprawdzania spełnialności w logice zdaniowej jest **NP-zupełny** – pierwszy dowód NP-zupełności.
- **Logiczna równoważność**
  Zdania α i β są logicznie równoważne ($\alpha \equiv \beta$) wtw gdy są prawdziwe w tych samych modelach.
- **Tautologie** (zdania prawdziwe) – zdania prawdziwe w każdym modelu.
- **Twierdzenia** – zdania/formuły posiadające dowód.
- Twierdzenie o **dedukcji**:
  $\alpha \models \beta$ wtw gdy $\alpha \Rightarrow \beta$ jest tautologią.
  - Sprawdzanie logicznej konsekwencji przez sprawdzanie prawdziwości zdań.
- $\alpha \models \beta$ wtw gdy zdanie $\alpha \wedge \neg\beta$ jest niespełnialne (nie jest spełnione w żadnym modelu) – **dowodzenie przez sprzeczność**.

Logika zdaniowa – przykłady ze świata wumpusa

Symbole zdaniowe:

- P_{11}, P_{12}, …, P_{44} – prawdziwe, gdy przepaść w $[1,1]$, $[1,2]$, itd.
- B_{11}, B_{12}, …, B_{44} – prawdziwe, gdy wietrzyk w $[1,1]$, $[1,2]$, itd.
- W_{11}, W_{12}, …, W_{44} – prawdziwe, gdy wumpus w $[1,1]$, $[1,2]$, itd.
- G_{11}, G_{12}, …, G_{44} – prawdziwe, gdy złoto w $[1,1]$, $[1,2]$, itd.

Zdania opisujące reguły świata:

- R_{P11}: $\neg P_{11}$
- R_{B11}: $B_{11} \Leftrightarrow (P_{12} \vee P_{21})$
- R_{B21}: $B_{21} \Leftrightarrow (P_{11} \vee P_{22} \vee P_{31})$
- …

Zdania opisujące pierwsze spostrzeżenia:

- R_{S1}: $\neg B_{11}$
- R_{S2}: B_{21}

Logika zdaniowa – wnioskowanie (1)

Pierwszy algorytm sprawdzania czy $BW \models \alpha$ – **sprawdzenie wszystkich możliwych modeli**.

Przykład świata wumpusa:

- Baza wiedzy i sprawdzane zdanie:
  - BW : $R_{P11} \wedge R_{B11} \wedge R_{B21} \wedge R_{S1} \wedge R_{S2}$
  - α : P_{22}
- Istotne symbole: $B_{11}, B_{21}, P_{11}, P_{12}, P_{21}, P_{22}, P_{31}$.
- 7 symboli to $2^7 = 128$ możliwych modeli
  - tylko trzy z nich są modelami dla BW,
  - we wszystkich trzech P_{21} jest fałszywe,
  - niejednoznacznie dla P_{22}.

Realizacja algorytmu w tabeli prawdziwości.

Logika zdaniowa – wnioskowanie (2)

Przykłady reguł wnioskowania (wyprowadzania):

- **Modus ponens**:

  MP: $$\frac{\alpha \Rightarrow \beta \qquad \alpha}{\beta}$$

- **Eliminacja koniunkcji**:

  AndE1: $$\frac{\alpha \wedge \beta}{\alpha}$$ AndE2: $$\frac{\alpha \wedge \beta}{\beta}$$

- Z prawa **kontrapozycji**:

  K1: $$\frac{\alpha \Rightarrow \beta}{\neg\beta \Rightarrow \neg\alpha}$$ K2: $$\frac{\neg\beta \Rightarrow \neg\alpha}{\alpha \Rightarrow \beta}$$

- Z praw **de Morgana**:

  dM1: $$\frac{\neg(\alpha \vee \beta)}{\neg\alpha \wedge \neg\beta}$$ dM2: $$\frac{\neg(\alpha \wedge \beta)}{\neg\alpha \vee \neg\beta}$$

- Z własności **równoważności**:

  IffE: $$\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}$$ IffI: $$\frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}$$

Logika zdaniowa – wnioskowanie (3)

- Przykład dowodu:

$$\dfrac{\dfrac{\dfrac{\dfrac{B_{11} \Leftrightarrow P_{12} \vee P_{21}}{(B_{11} \Rightarrow P_{12} \vee P_{21}) \wedge (P_{12} \vee P_{21} \Rightarrow B_{11})}\,\text{IffE}}{P_{12} \vee P_{21} \Rightarrow B_{11}}\,\text{AndE2}}{\neg B_{11} \Rightarrow \neg(P_{12} \vee P_{21})}\,\text{K1} \qquad \neg B_{11}}{\dfrac{\dfrac{\neg(P_{12} \vee P_{21})}{\neg P_{12} \wedge \neg P_{21}}\,\text{dM1}}{\neg P_{12}}\,\text{AndE1}}\,\text{MP}$$

Dowód dla $\neg P_{12}$ uzyskamy przez zastosowanie na końcu reguły AndE2 zamiast AndE1.

Logika zdaniowa – systemy formalne

- W systemach formalnych, **klasyczny język zdaniowy definiuje się** jako relację konsekwencji syntaktycznej (zdefiniowanej jako dowodliwość) z **trzema schematami aksjomatów**:

$$
\begin{array}{ll}
A1: & \alpha \Rightarrow (\beta \Rightarrow \alpha) \\
A2: & (\alpha \Rightarrow (\beta \Rightarrow \gamma)) \Rightarrow ((\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \gamma)) \\
A3: & (\neg\alpha \Rightarrow \neg\beta) \Rightarrow (\beta \Rightarrow \alpha)
\end{array}
$$

  oraz regułą **modus ponens**.

- **Pozostałe symbole logiczne** są definiowane na bazie dwóch podstawowych $\Rightarrow$ i $\neg$ następująco:

$$
\begin{array}{rcl}
\alpha \vee \beta & \equiv & \neg(\alpha \Rightarrow \beta) \\
\alpha \wedge \beta & \equiv & \neg(\alpha \Rightarrow \neg\beta) \\
\bot \textit{ (fałsz)} & \equiv & \neg(\alpha \Rightarrow \alpha)
\end{array}
$$

Formalne systemy wnioskowania

- Różne reguły i środki **budowania dowodów**. Trzy podstawowe **reprezentacje**:
  - reprezentacja **Hilberta**,
  - **naturalna dedukcja**,
  - reprezentacja **sekwentowa**.
- Wszystkie trzy są **wzajemnie równoważne** – jeśli istnieje dowód w jednym systemie to i w pozostałych dwóch.
- Każdy z systemów jest **zgodny** i **zupełny**.
- Problem znajdowania dowodów zadanych formuł jest **NP-zupełny**.

Reprezentacja Hilberta (1)

Reprezentacja Hilberta

- **Dowód formuły** α z przesłanek Γ to **skończony ciąg formuł** zakończony formułą α taki, że każda z pozostałych formuł spełnia jeden z warunków:
  - jest jednym z aksjomatów,
  - jest elementem zbioru Γ,
  - można ją otrzymać z poprzedzających ją formuł stosując jedną z reguł dowodzenia.
- **Formuły** to uogólnienie zdań na użytek innych (bardziej złożonych) systemów logiki.

Reprezentacja Hilberta (2)

Dowód ze str. 131 jest w reprezentacji **Hilberta** przekonwertowanej do postaci **drzewa**. **Jako lista** wyglądałby tak:

$1:$	$B_{11} \Leftrightarrow P_{12} \vee P_{21}$	*założenie*
$2:$	$(B_{11} \Rightarrow P_{12} \vee P_{21}) \wedge (P_{12} \vee P_{21} \Rightarrow B_{11})$	*IffE(1)*
$3:$	$P_{12} \vee P_{21} \Rightarrow B_{11}$	*AndE2(2)*
$4:$	$\neg B_{11} \Rightarrow \neg(P_{12} \vee P_{21})$	*K1(3)*
$5:$	$\neg B_{11}$	*spostrzeżenie*
$6:$	$\neg(P_{12} \vee P_{21})$	*MP(4,5)*
$7:$	$\neg P_{12} \wedge \neg P_{21}$	*dM1(6)*
$8:$	$\neg P_{12}$	*AndE1(7)*

Naturalna dedukcja (1)

Dowodem formuły α z przesłanek Γ (w reprezentacji naturalnej dedukcji) nazywamy **skończony ciąg formuł** taki, że

1. Ostatnią formułą ciągu jest formuła α.
2. Każda z pozostałych formuł ciągu spełnia jeden z warunków:
   - jest elementem zbioru Γ,
   - powstaje z formuł poprzedzających ją przez zastosowanie jednej z reguł wnioskowania bądź przez przepisanie,
   - jest dowolną inną formułą i wtedy mówimy, że jest **hipotezą**.
3. Wszystkie **hipotezy są wyeliminowane**.

Eliminacji hipotezy dokonuje się przez zastosowanie reguły eliminującej założenia.

Eliminować można **tylko ostatnią hipotezę** w ciągu (dowodzie).

Naturalna dedukcja (2)

Dowód formuły $\neg\beta \Rightarrow (\beta \Rightarrow \alpha)$:

$1:$	$\neg\beta$	hipoteza
$2:$	β	hipoteza
$3:$	$\bot$	$\neg e$ 1,2
$4:$	α	$\bot e$ 3
$5:$	$\beta \Rightarrow \alpha$	$\Rightarrow i$ 2–4
$6:$	$\neg\beta \Rightarrow (\beta \Rightarrow \alpha)$	$\Rightarrow i$ 1–5

Naturalna dedukcja (3)

Reguły dla logiki klasycznej pierwszego rzędu (c.d.n.):

$$\frac{A \quad B}{A \wedge B}\ \wedge i \qquad \frac{A \wedge B}{A}\ \wedge e_1 \qquad \frac{A \wedge B}{B}\ \wedge e_2$$

$$\frac{A}{A \vee B}\ \vee i_1 \qquad \frac{B}{A \vee B}\ \vee i_2 \qquad \frac{A \vee B \quad \boxed{\begin{array}{c} A \\ \vdots \\ C \end{array}} \quad \boxed{\begin{array}{c} B \\ \vdots \\ C \end{array}}}{C}\ \vee e$$

$$\frac{\boxed{\begin{array}{c} A \\ \vdots \\ B \end{array}}}{A \Rightarrow B}\ \Rightarrow i \qquad \frac{A \quad A \Rightarrow B}{B}\ \Rightarrow e$$

$$\frac{\boxed{\begin{array}{c} A \\ \vdots \\ \bot \end{array}}}{\neg A}\ \neg i \qquad \frac{A \quad \neg A}{\bot}\ \neg e$$

Naturalna dedukcja (4)

Reguły dla logiki klasycznej pierwszego rzędu (c.d.):

$$\frac{\bot}{A}\ \bot e$$

$$\frac{\begin{array}{|l}\hline y \\ \\ \vdots \\ \\ A(y) \\ \hline\end{array}}{\forall x A(x)}\ \forall i$$

$$\frac{\forall x A(x)}{A(t)}\ \forall e$$

$$\frac{A(t)}{\exists x A(x)}\ \exists i$$

$$\frac{\exists x A(x) \qquad \begin{array}{|ll}\hline y & A(y) \\ & \vdots \\ & B \\ \hline\end{array}}{B}\ \exists e$$

$$\frac{\neg\neg A}{A}\ \neg\neg$$

Reprezentacja sekwentowa (1)

- **Sekwentem** nazywamy wyrażenie postaci $\Gamma \vdash \Delta$, gdzie Γ, Δ są zbiorami formuł.
- Γ – **poprzednik**, Δ – **następnik sekwentu** $\Gamma \vdash \Delta$.
- **Dowód** w reprezentacji sekwentowej to **drzewo sekwentów** spełniające następujące warunki :
  - **liście** drzewa to **aksjomaty**,
  - każdy sekwent drzewa jest górnym sekwentem **reguły wyprowadzania**, której dolny sekwent występuje w drzewie bezpośrednio pod tym sekwentem.

Reprezentacja sekwentowa (2)

Reguły dla logiki klasycznej pierwszego rzędu (c.d.n.):

$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B} \wedge i$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge e_1 \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge e_2$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee i_1 \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee i_2$$

$$\frac{\Gamma \vdash A \vee B \quad \Delta, A \vdash C \quad \Theta, B \vdash C}{\Gamma, \Delta, \Theta \vdash C} \vee e$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \Rightarrow i$$

$$\frac{\Gamma \vdash A \quad \Delta \vdash A \Rightarrow B}{\Gamma, \Delta \vdash B} \Rightarrow e$$

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A} \neg i$$

$$\frac{\Gamma \vdash A \quad \Delta \vdash A \Rightarrow B}{\Gamma, \Delta \vdash \bot} \neg e$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A} \bot e$$

Reprezentacja sekwentowa (3)

Reguły dla logiki klasycznej pierwszego rzędu (c.d.):

$$\frac{\Gamma \vdash A(y)}{\Gamma \vdash \forall x A(x)}\ \forall i \qquad \frac{\Gamma \vdash \forall x A(x)}{\Gamma \vdash A(t)}\ \forall e$$

$$\frac{\Gamma \vdash A(t)}{\Gamma \vdash \exists x A(x)}\ \exists i \qquad \frac{\Gamma \vdash \exists x A(x) \quad \Delta, A(y) \vdash B}{\Gamma, \Delta \vdash B}\ \exists e$$

$$\frac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}\ \neg\neg$$

Uwaga! *W regułach* $\forall i$ *oraz* $\exists e$ *zmienna* y *nie występuje w sposób wolny w* Γ, $A(x)$ *oraz* B.

System Gentzena (1)

- **Relacja podformuły**:
  - $\bot$ jest podformułą każdej formuły,
  - A jest podformułą formuł A i $\neg A$,
  - A i B są podformułami formuł $A \vee B$, $A \wedge B$, $A \Rightarrow B$, $A \Leftrightarrow B$,
  - dla dowolnego termu t, formuła $A(t)$ jest podformuła formuł $\forall_x A(x)$ i $\exists_x A(x)$.
- Reguła wnioskowania $\dfrac{\Gamma \vdash A}{\Gamma' \vdash A'}$ spełnia **własność podformuły** wtedy i tylko wtedy, gdy każda formuła ze zbioru $\Gamma \cup \{A\}$ jest podformułą pewnej formuły ze zbioru $\Gamma' \cup \{A'\}$.
- Reguły wprowadzania (i) rachunku sekwentowego spełniają własność podformuły.
- Reguły eliminacji (e) rachunku sekwentowego **nie spełniają** własności podformuły.
- Sekwentowy **rachunek Gentzena** zawiera reguły spełniające własność podformuły.

System Gentzena (2)

Reguły systemu Gentzena (c.d.n.) i reguła obcinania:

Reguły strukturalne

$$\frac{}{\Gamma, A \vdash A, \Delta}(inkluzja) \qquad \frac{\Gamma \vdash \Theta, C \quad C, \Delta \vdash \Lambda}{\Gamma, \Delta \vdash \Theta, \Lambda}\mathsf{cut}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\mathsf{mono}_l \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}\mathsf{mono}_r$$

Reguły logiczne

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \wedge l_1 \quad \frac{\Gamma, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \wedge l_2 \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta} \wedge r$$

$$\frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta} \vee l \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \vee B, \Delta} \vee r_1 \quad \frac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \vee B, \Delta} \vee r_2$$

System Gentzena (3)

Reguły systemu Gentzena (c.d.):

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \Rightarrow B \vdash \Delta} \Rightarrow l \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta} \Rightarrow r$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta} \neg l \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta} \neg r$$

$$\frac{\Gamma, A(t) \vdash \Delta}{\Gamma, \forall x A(x) \vdash \Delta} \forall l \qquad \frac{\Gamma \vdash A(y), \Delta}{\Gamma \vdash \forall x A(x), \Delta} \forall r$$

$$\frac{\Gamma, A(y) \vdash \Delta}{\Gamma, \exists x A(x) \vdash \Delta} \exists l \qquad \frac{\Gamma \vdash A(t), \Delta}{\Gamma \vdash \exists x A(x), \Delta} \exists r$$

Uwaga! *W regułach* $\forall r$ *oraz* $\exists l$ *zmienna* y *nie występuje w sposób wolny w* Γ, Δ *oraz* $A(x)$.

System Gentzena (4)

- Dowód równoważny Hilbertowskiemu ze strony 131:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg B_{11}, P_{12} \vdash P_{12}}\,\textit{inkluzja}}{\neg B_{11}, P_{12} \vdash P_{12} \vee P_{21}}\,\vee r_1}{\neg B_{11} \vdash P_{12} \vee P_{21}, \neg P_{12}}\,\neg r \qquad \dfrac{\dfrac{}{B_{11} \vdash B_{11}, \neg P_{12}}\,\textit{inkluzja}}{\neg B_{11}, B_{11} \vdash \neg P_{12}}\,\neg l}{(P_{12} \vee P_{21} \Rightarrow B_{11}), \neg B_{11} \vdash \neg P_{12}}\,\Rightarrow l}{(B_{11} \Rightarrow P_{12} \vee P_{21}) \wedge (P_{12} \vee P_{21} \Rightarrow B_{11}), \neg B_{11} \vdash \neg P_{12}}\,\wedge l_2}{B_{11} \Leftrightarrow P_{12} \vee P_{21}, \neg B_{11} \vdash \neg P_{12}}\,\text{def.}\,\Leftrightarrow$$

Reguła rezolucji (1)

Przykład wumpusa:

- Był powiew w [1,2]. Stąd dowód $P_{11} \vee P_{22} \vee P_{31}$.
- Nie było powiewu w [2,1]. Stąd dowód $\neg P_{11} \wedge \neg P_{22} \wedge \neg P_{13}$ oraz dowody $\neg P_{11}$ i $\neg P_{22}$.
- Z $P_{11} \vee P_{22} \vee P_{31}$ oraz $\neg P_{11}$ wynika $P_{22} \vee P_{31}$.
- Z $P_{22} \vee P_{31}$ oraz $\neg P_{22}$ wynika P_{31}.

Reguła rezolucji:

$$\frac{l_1 \vee \cdots \vee l_k \qquad m_1 \vee \cdots \vee m_n}{l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \vee \cdots \vee l_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n}$$

gdzie l_i oraz m_j są **literałami komplementarnymi** (atom i jego zaprzeczenie).

- *Uwaga techniczna:* powtarzające się literały są eliminowane z **rezolwenty** (zdania pod kreską).

Reguła rezolucji (2)

Własność 1: Reguła rezolucji jest zgodna.
Dowód: Albo l_i nie jest prawdziwe, albo m_j. Jeśli l_i, to prawdziwe jest $l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \vee \cdots \vee l_k$, a więc i pełna konkluzja reguły. W przeciwnym przypadku zdanie $m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n$ jest prawdziwe, a więc również pełna konkluzja reguły. □

Własność 2: Każdy **zupełny algorytm szukania**, stosujący jedynie regułę rezolucji **znajdzie dowód każdego zdania wynikającego** z każdej bazy wiedzy w logice zdaniowej.

Uwaga: Własność 2

- nie oznacza, że możemy wyliczyć wszystkie zdania wynikające z danej bazy wiedzy,
- mówi o **zupełności odrzucania** (ang. *refutation completness*) tzn, że jesteśmy w stanie weryfikować prawdziwość (bądź nie) każdego zdania w kontekście każdej bazy wiedzy.

Reguła rezolucji (3)

Koniunkcyjna postać normalna (KPN)

- **Każde zdanie** logiki zdaniowej jest logicznie równoważne zdaniu będącemu **koniunkcją alternatyw literałów**. Mówimy, że zdania w takiej postaci są w **koniunkcyjnej postaci normalnej**.
- Idea algorytmu transformującego zdania do KPN:
  - eliminujemy spójniki $\Leftrightarrow$ i $\Rightarrow$ na podstawie ich definicji,
  - używamy zasad rozdzielności, podwójnej negacji, praw de Morgana.

Algorytm rezolucji (1)

- Aby wykazać, że $BW \vdash \alpha$ pokazuje się, że $BW \wedge \neg\alpha$ jest **niespełnialne** (dowód przez zaprzeczenie).
- **Idea algorytmu**:
  - przekształcamy $BW \wedge \neg\alpha$ **do KPN**,
  - **stosujemy regułę rezolucji** do wszystkich możliwych par zdań (części KPN),
  - powtarzamy poprzedni krok aż uzyskamy **zdanie puste** (fałsz, sprzeczność) lub **nie** jesteśmy w stanie wygenerować **nowych zdań** (brak sprzeczności, brak wynikania).
- Algorytm **zawsze się kończy**, bo liczba możliwych zdań zbudowanych ze zdań występujących w $BW \wedge \alpha$ jest skończona.
- **Domknięcie rezolucyjne** $DR(S)$ zbioru zdań S, to zbiór wszystkich zdań , które można uzyskać z S poprzez stosowanie reguły rezolucji dowolną liczbę razy.

Algorytm rezolucji (2)

Twierdzenie: *Jeśli zbiór zdań jest niespełnialny, to jego domknięcie rezolucyjne zawiera zdanie puste.*

Dowód: Niech $S = \{P_1, \ldots, P_k\}$ będzie zbiorem zdań takim, że $DR(S)$ nie zawiera zdania pustego. Pokażemy, że S jest spełnialny przez konstrukcję modelu:

- Dla $i = 1, \ldots, k$
  - Jeśli w $DR(S)$ istnieje zdanie zawierające literał $\neg P_i$, w którym pozostałe literały są fałszywe przy dotychczasowym przypisaniu dla $P_1, \ldots, P_{i-1}$, to przypisujemy fałsz do P_i.
  - W przec. razie przypisujemy prawdę do P_i.

Fakt, że takie przypisanie jest modelem możemy dowieść (również) przez sprzeczność.

Wnioskowanie w przód i w tył (1)

Klauzula Hornowska (Horna) – alternatywa literałów, z których co najwyżej jeden jest pozytywny.

- $\neg P_1 \vee \cdots \vee \neg P_k \vee P_{k+1} \equiv P_1 \wedge \cdots \wedge P_k \Rightarrow P_{k+1}$
- Reguły tej postaci pozwalają na efektywne wnioskowanie w przód i w tył.

Wnioskowanie w przód

- iteracyjne poszerzanie listy udowodnionych zdań atomowych,
- działa **w czasie liniowym** względem rozmiaru bazy wiedzy.

Wnioskowanie w tył

- startuje od weryfikowanego zdania i zastępuje kolejne zdania do udowodnienia przez przesłanki odpowiednich reguł,
- działa **w czasie liniowym**, a zwykle znacznie szybciej.

Wnioskowanie w przód – zdania Horna

Prototype: ***SprawdźWPrzód***(BW, q)

Input: Baza wiedzy BW, zapytanie q.

1. **for each** $\alpha \in BW$ **do** $ile[\alpha] \leftarrow$ liczba przesłanek w α
2. kolejka $\leftarrow$ symbole prawdziwe z BW
   1. **while** kolejka niepusta **do**
   2. $p \leftarrow$ pobrane z kolejki
   3. **if** $p = q$ **then return** prawda
   4. **if** $p \notin$ udowodnione **then**
      1. udowodnione += p
      2. **for each** $\alpha \in BW$ t.że p występuje w α **do**
         - $ile[\alpha] - -$
         - **if** $ile[\alpha] = 0$ **then** kolejka += konkluzja z α

Wnioskowanie w tył – zdania Horna

Prototype: ***SprawdźWTył***(BW, q)
Input: Baza wiedzy BW, zapytanie q.

1. **for each** $\alpha \in BW$ t.że q jest konkluzją α **do**
   - **for each** $p \in$ przesłanki α **do**
     - **if** SprawdźWTył(BW,p) = fałsz **then return** fałsz
   - **return** prawda
2. **return** fałsz

- Ten algorytm nie chroni przed nieskończoną rekurencją, gdy w BW znajdą się formuły typu $P \Rightarrow Q$ i $Q \Rightarrow P$.

Algorytm Davisa-Putnama (1)

- Zupełny algorytm szukania z nawrotami (jak w problemach z ograniczeniami).
- Próba znalezienia modelu (wartościowania symboli) z technikami usprawniającymi:
  - Wczesne rozstrzyganie (gdy można ocenić prawdziwość zdania bez wyznaczania pełnego wartościowania).
  - Heurystyka **czystych symboli**. Symbol jest **czysty** w danym zbiorze zdań, gdy w zdaniach z tego zbioru występuje zawsze jako pozytywny bądź zawsze jako negatywny. Wówczas wystarczy sprawdzić tylko przypisanie zapewniające prawdziwość występujących literałów.
  - **Zdania jednostkowe** – zdania w których wszystkie literały oprócz jednego zostały ocenione jako fałszywe. Determinują przypisanie symbolu występującego w tym jednym literale.

Algorytm Davisa-Putnama (2)

Prototype: ***Davis-Putnam(*z*)***
Input: Zdanie z logiki zdaniowej.

1. $klauzule \leftarrow$ zbiór klauzul z KPN zdania z
2. $symbole \leftarrow$ kolekcja symboli występujących w zdaniu z
3. **return** DP($klauzule$, $symbole$, $\emptyset$)

Prototype: ***DP(*$klauzule$*,* $symbole$*,* $model$*)***

1. **if** wszystkie $klauzule$ są prawdziwe w $modelu$ **then return** prawda
2. **if** istnieje $\alpha \in klauzule$ fałszywa w $modelu$ **then return** fałsz
3. $C \leftarrow$ WartościDlaCzystych($symbole$, $klauzule$, $model$)
4. **if** $C \neq \emptyset$ **then return** DP($klauzule$, $symbole - C$, Rozszerz($model$, C))
5. $J \leftarrow$ WartościDlaJedn($symbole$, $klauzule$, $model$)
6. **if** $J \neq \emptyset$ **then return** DP($klauzule$, $symbole - J$, Rozszerz($model$, J))
7. $P, R \leftarrow$ pierwszy i reszta spośród $symboli$
8. **return** DP($klauzule$, $symbole + P$, Rozszerz($model$,$P = prawda$) $\vee$ DP($klauzule$, $symbole + P$, Rozszerz($model$,$P = fasz$)

Algorytm WalkSAT

- Implikacja (przesłanki $\Rightarrow$ konkluzja) $\Leftrightarrow$ KPN $\Leftrightarrow$ zbiór klauzul-alternatyw KPN.
- Podobieństwo do problemów **z ograniczeniami** (zmienne vs zdania).
- **Metody lokalnego szukania** są często skuteczne w takich zadaniach.
- **Schemat algorytmu**:
  - wybieramy losowy model (losowe wartościowanie symboli zdaniowych),
  - określoną liczbę razy
    - jeśli model spełnia sprawdzane klauzule, to zwracamy go jako wynik,
    - dokonujemy zmiany przypisania (modelu) dla jednej ze zmiennych.
- **Zmiany przypisania** można robić na różne sposoby
  - całkowicie losowo (ang. *random walk*),
  - wg kryterium *min-conflicts*,
  - wiele innych możliwości.

Agent w świecie wumpusa

- Opis świata 4×4 w logice zdaniowej.
  - 16 reguł – powiew w pomieszczeniach, np. $B_{22} \Leftrightarrow P_{12} \vee P_{21} \vee P_{23} \vee P_{32}$
  - 16 reguł – fetor wumpusa (analogicznie do powiewu),
  - 1 reguła – istnienie wumpusa ($W_{12} \vee W_{13} \vee \ldots \vee W_{44}$),
  - 105 reguł – niemożliwość spotkania wumpusa w dwóch pomieszczeniach,
  - 2 reguły startowe: $\neg P_{11}$, $\neg W_{11}$.
  - Razem: **140 reguł** z wykorzystaniem **64 symboli**. Wyliczenie wszystkich 2^{64} przypisań niemożliwe.
- **Pytania** o bezpieczeństwo pomieszczeń, np. $\neg P_{12} \wedge \neg W_{12}$.
- **Decyzje** agenta
  - **logiczne wnioskowanie** ocenia kolejne nieznane pomieszczenia,
  - baza wiedzy **aktualizowana** na bieżąco wg spostrzeżeń,
  - **metody szukania** wyznaczają optymalne, bezpieczne ścieżki do nowych pomieszczeń.

Ograniczenia logiki zdaniowej

Cechy systemów logiki zdaniowej

- **Deklaratywność**.
- **Kompozycyjność** – znaczenie zdania złożonego jest funkcją znaczeń jego składowych.
- Zdania są zbyt ogólnym narzędziem – **wymagają mnóstwa symboli** w rozwiązywaniu rzeczywistych problemów.
- Brak narzędzi do reprezentowania **niepewności**.
- **Eliminowanie dwuznaczności**.

Pożądane cechy języka naturalnego:

- rzeczowniki i wyrażenia rzeczownikowe do **reprezentacji obiektów**,
- **własności**, **relacje** i **funkcje**, np. „niesprawny komputer”, „mój samochód”, „pies ogrodnika”.

Bardziej złożone systemy logiczne

- Logika **klasyczna I rzędu**
  - termy – „świat” obiektów
- Logika **temporalna**
  - uwzględnia zmienność w czasie
- Logika **wyższego rzędu**
  - pozwala wyartykułować bardziej złożone relacje i meta-relacje, np. kwantyfikowanie po podzbiorach, założenie przechodniości relacji itp.
- Systemy **probabilistyczne**
  - prawdopodobieństwa, sieci przekonań, itp.
- Logika **rozmyta**
  - stopniowanie przynależności do zbiorów
- Logika **zbiorów przybliżonych**
  - określanie zbiorów przez ograniczenia z góry i z dołu

Język predykatowy (1)

Język predykatowy to dowolna szóstka $L = (P, T, V, O, Q, arg)$, t.że:

1. P, T, O, Q to dowolne parami rozłączne zbiory
2. $V \subseteq T$
3. $arg : P \cup O \to N$

Kolejne składowe języka predykatowego noszą następujące nazwy:

- P to zbiór **symboli predykatowych**
- T to zbiór **termów**
- V to zbiór **zmiennych**
- O to zbiór **operatorów**
- Q to zbiór **kwantyfikatorów**
- arg to argumentowość (arność) operatorów i predykatów

Oznaczenie: $var(t)$ – zbiór zmiennych termu t.

Język predykatowy (2)

Termy (elementy zbioru T) są konstruowane ze zmiennych (ze zbioru V) oraz **symboli funkcyjnych** z określonego zbioru F o **określonej argumentowości**:

- $V \subseteq T$,
- jeśli $f \in F$, $arg(f) = n$ oraz $t_1, \ldots, t_n \in T$, to $f(t_1, \ldots, t_n) \in T$.

Formuły atomowe języka $L = (P, T, V, O, Q, arg)$ powstają przy użyciu symboli predykatowych i termów zgodnie z następującą zasadą:

$$\textit{jeśli } p \in P, arg(p) = n, t_1, \ldots, t_n \in T,$$
$$\textit{to } p(t_1, \ldots, t_n) \textit{ jest } \textbf{\textit{formułą atomową}}.$$

Jeśli $arg(p) = 0$ to p jest **stałą formułą atomową**.

Język predykatowy (3)

Zbiorem formuł języka L nazywamy najmniejszy zbiór spełniający następujące własności:

1. Jeśli A jest formułą atomową to A jest formułą. Mówimy, że A nie ma głównego symbolu logicznego.
2. Jeśli $A_1, \ldots, A_n$ są formułami i o jest operatorem arności n to napis $o(A_1, \ldots, A_n)$ jest formułą. Mówimy, że jego głównym symbolem logicznym jest symbol o.
3. Jeśli A jest formułą, $q \in Q$ oraz $x \in V$ to napis $q_x A$ jest formułą. Głównym symbolem logicznym formuły $q_x A$ jest symbol q.

Zmienne wolne i związane

Zbiór zmiennych wolnych formuły A (oznaczenie $free(A)$) określają następujące reguły:
Niech $p \in P$, $arg(p) = n$, $o \in O$, $arg(o) = m$, $t_1, \ldots, t_n \in T$, $q \in Q$ oraz niech $A_1, \ldots, A_m \in L$

1. $free(p(t_1, \ldots, t_n)) = var(t_1) \cup \ldots \cup var(t_n)$
2. $free(o(A_1, \ldots, A_m)) = free(A_1) \cup \ldots \ free(A_m)$
3. $free(q_x A) = free(A) - \{x\}$. W tym przypadku każde wystąpienie zmiennej x w formule A nazywa się związanym.

Logika klasyczna I-go rzędu (1)

Logika klasyczna I-go rzędu to logika predykatowa
$L = (P, T, V, O, Q, arg)$, gdzie:

- $O = \{\vee, \wedge, \rightarrow, \neg, \bot\}$,
- $V = \{x, y, z, x_1, x_2, \ldots\}$,
- $Q = \{\forall, \exists\}$,
- $arg(\vee) = arg(\wedge) = arg(\rightarrow) = 2$, $arg(\neg) = 1$, $arg(\bot) = 0$.

Uwaga: L to nie jeden konkretny język, ale **cała rodzina**. Różne definicje V i P określają różne konkretne języki.

Logika klasyczna I-go rzędu (2)

Uwagi: Przyjmijmy, że t_1, t_2, t_3 są termami, oraz $\{p, r\} \subseteq P$, gdzie $arg(p) = 2, arg(r) = 0$. Wówczas:

- Formułami atomowymi są napisy: $p(t_1, t_3)$, $p(t_2, t_2)$, r ...
- Do zbioru formuł należą na przykład napisy: $\neg p(t_2, t_3)$, $p(t_1, t_2) \to r$, $\forall_x p(t_1, t_1)$, ...
- W formułach $\neg p(t_2, t_3)$, $p(t_1, t_2) \to r$ wszystkie występujące zmienne są wolne.
- W $\forall_x p(t_1, t_1)$ zmienna x występuje w termach t_1 i t_2 jako związana, natomiast zbiór zmiennych wolnych tej formuły $free(\forall_x p(t_1, t_1)) = var(t_1) \cup var(t_2) - \{x\}$.

Semantyka: systemy waluacji, przypisania i interpretacje

Systemem waluacji dla języka $L = (P, T, V, O, Q, arg)$ nazywamy czwórkę $\underline{M} = (M, D, W_O, W_Q)$ określoną następująco:

1. M jest zbiorem co najmniej dwuelementowym
2. D jest niepustym, właściwym podzbiorem zbioru M
3. $W_O = \{g_o : M^{arg(o)} \to M; o \in O\}$
4. $W_Q = \{h_q : 2^M \to M; q \in Q\}$

Semantyka:

- M to zbiór **wartości logicznych**, np. $\{\text{fałsz}, \text{prawda}\}$.
- D to zbiór wartości oznaczających **spełnianie formuł** (p. dalej).
- W_O to **interpretacja operatorów**.
- W_Q to **interpretacja kwantyfikatorów**.

Logika klasyczna I-go rzędu (4)

Przypisaniem dla języka predykatowego $L = (P, T, V, O, Q)$ w systemie waluacji $\underline{M} = (M, D, W_O, W_Q)$ nazywamy parę (a, I) taką, że I jest niepustym zbiorem (zwanym **dziedziną indywiduową**), natomiast

$$a : T \cup P \to I \cup \bigcup_{n \in N} (I^n \to M)$$

jest funkcją spełniającą następujące warunki:

1. Jeśli $t \in T$ to $a(t) \in I$
2. Jeśli $p \in P$, $arg(p) = n$ to $a(p) : I^n \to M$

Uwaga: Przypisanie (a, I) zwykle utożsamia się z funkcją a.

Logika klasyczna I-go rzędu (5)

Oznaczenie: Niech a i a' będą przypisaniami o tej samej dziedzinie indywiduowej, oraz niech $x \in V$. Piszemy

$$a \sim_x a'$$

o ile spełnione są następujące warunki:

1. Jeśli $t \in T$ oraz $x \notin var(t)$ to $a(t) = a'(t)$
2. Jeśli $y \in V$ oraz $y \neq x$ to $a(y) = a'(y)$
3. Jeśli $p \in P$ to $a(p) = a'(p)$

Logika klasyczna I-go rzędu (6)

Niech $\underline{M} = (M, D, W_O, W_Q)$ będzie systemem waluacji dla języka predykatowego L, oraz niech a będzie przypisaniem w $\underline{M}$. **Waluacją** nazywamy funkcję

$$v_a : L \to M$$

określoną następująco:

1. Jeśli $p \in P$, $arg(p) = n$ oraz $t_1, \ldots, t_n \in T$ to

$$v_a(p(t_1, \ldots, t_n)) = a(p)(a(t_1), \ldots, a(t_n))$$

2. Jeśli $o \in O, arg(o) = m$ oraz $A_1, \ldots, A_m \in L$ to

$$v_a(o(A_1, \ldots, A_n)) = g_o(v_a(A_1), \ldots, v_a(A_n))$$

3. Jeśli $q \in Q, x \in V, A \in L$ to

$$v_a(q_x A) = h_q(\{v_{a'}(A); a' \sim_x a\})$$

Uwaga: Wartość logiczna formuły $q_x A$ dla przypisania a nie zależy od wartości tego przypisania dla zmiennej x.

Spełnianie i prawdziwość formuł

Formuła A jest **spełniona w systemie waluacji** $\underline{M} = (M, D, W_O, W_Q)$ **przy interpretacji** v_a, o ile $v_a(A) \in D$.

Formuła A jest **prawdziwa w systemie waluacji** $\underline{M} = (M, D, W_O, W_Q)$, o ile jest spełniona przy każdej interpretacji v_a.

Waluacja w logice klasycznej I-go rzędu

System waluacji dla języka $L_{K1R} = (P, T, V, O, Q, arg)$ to czwórka $\underline{M} = (M, D, W_O, W_Q)$, taka, że

- $M = \{t, f\}$,
- $D = \{t\}$,
- $W_O = \{g_\vee, g_\wedge, g_\neg, g_\to, g_\bot\}$ – określone jak dla algebry Boole'a, dodatkowo $g_\bot =$ fałsz.
- $W_Q = \{h_\forall, h_\exists\}$ – określone następująco:

$$h_\forall(X) = \begin{cases} f & \text{o ile } f \in X \\ t & \text{w przeciwnym wypadku} \end{cases}$$

$$h_\exists(X) = \begin{cases} t & \text{o ile } t \in X \\ f & \text{w przeciwnym wypadku} \end{cases}$$

Logika klasyczna I-go rzędu (9)

Przykład interpretacji formuł z kwantyfikatorami dla konkretnego języka.

Niech $T = V = \{x, y\}$, $P = \{p, r\}$, $arg(p) = 1$, oraz $arg(r) = 0$.

W takim języku mamy tylko **trzy formuły atomowe**: $p(x)$, $p(y)$, r.

Niech a będzie **przypisaniem** takim, że: $a(x) = t$, $a(y) = f$, $a(r) = t$, oraz $a(p) = id_{M_2}$.

Wówczas:

$$v_a(\forall_x p(x)) = g_\forall(\{v_{a'}(p(x)); a' \sim_x a\}) = g_\forall(\{t, f\}) = \text{fałsz}$$

$$v_a(\exists_x p(x)) = g_\exists(\{t, f\}) = \text{prawda}$$

$$v_a(\exists_y r) = g_\exists(\{v_{a'}(r); a' \sim_y a\}) = g_\exists(\{t\}) = \text{prawda}$$

Przykład liczb naturalnych

- $V = \{x, y, \ldots\}$
- $F = \{0, s\}$, $arg(0) = 0$, $arg(s) = 1$
- $T = \{x, y, 0, s(0), s(x), s(y), s(s(0)), \ldots\}$
- $P = \{parz, nieparz\}$, $arg(parz) = 1$, $arg(nieparz) = 1$

Przykłady formuł:

$$parz(x) \Rightarrow nieparz(s(x))$$

$$\forall_x parz(x) \vee nieparz(x)$$

$$\exists_x parz(x) \wedge nieparz(x)$$

Uwaga: Aksjomatyka Peana pozostaje poza możliwościami logiki klasycznej I-go rzędu (aksjomat indukcji).

Świat wumpusa w języku l. k. I-go rzędu (1)

- Termy i kwantyfikatory pozwalają na **bardziej zwięzłą** reprezentację wiedzy.
- **Problem czasu** - zmienności stanu wiedzy
  - predykaty rozszerzone o **parametr czasu** (liczby naturalne),
  - **osobna baza wiedzy** na użytek każdej decyzji (spostrzeżenia modyfikują BW).
- Oczywiście jest wiele możliwości organizacji systemu dla tego zadania.

Świat wumpusa w języku l. k. I-go rzędu (2)

Przymiarki do **reprezentacji wiedzy**:

- **Obiekty** (do reprezentowania termami):
  - wumpus, agent, przepaść,
  - sygnały: smród, powiew, lśnienie, odbicie od ściany, wrzask wumpusa,
  - działania: obroty, ruch naprzód, strzał z łuku, zabranie złota,
  - „dom” wumpusa – lepiej reprezentować termem niż predykatem i faktami.
- **Predykaty**:
  - lokalizacja agenta, spostrzeżenie, najlepsze działanie,
  - zjawiska w bieżącym pomieszczeniu: powiew, lśnienie,
  - cechy pomieszczeń: wietrzne, z przepaścią,
  - sąsiedztwo pomieszczeń.

Świat wumpusa w języku l. k. I-go rzędu (3)

Przykłady reguł:

$$\forall_x \forall_y Obok(x,y) \Leftrightarrow$$
$$x = [a,b] \wedge y = [c,d] \wedge (a = c \wedge |b-d| = 1 \ \vee \ b = d \wedge |a-c| = 1).$$

Diagnostyczne:

$$\forall_x Powiew(x) \Leftrightarrow \exists_y Obok(x,y) \wedge Przepaść(y).$$

Przyczynowe:

$$\forall_x Przepaść(x) \Rightarrow [\forall_y Obok(x,y) \Rightarrow Powiew(y)]\,.$$

Wnioskowanie w logice I-go rzędu (1)

Reguły podstawiania dla kwantyfikatorów:

- **ogólnego**:

$$\frac{\forall_x \alpha}{Podst(\{x/t\}, \alpha)}$$

  $Podst(\{x/t\}, \alpha)$ oznacza podstawienie za x **termu zamkniętego** (nie zawierającego zmiennych) t.

- **szczegółowego**:

$$\frac{\exists_x \alpha}{Podst(\{x/k\}, \alpha)}$$

  k jest stałą (symbolem funkcyjnym arności 0) nie występującą w bazie wiedzy (stała Skolema).

Wnioskowanie w logice I-go rzędu (2)

Redukcja do języka zdaniowego:

- formuła z kwantyfikatorem **ogólnym** zastępowana jest przez **kolekcję reguł** otrzymaną przez wszystkie możliwe podstawienia termów zamkniętych,
- formuła z kwantyfikatorem **szczegółowym** – **formułą z nową stałą**.

Problemy:

- **Liczba termów zamkniętych** może być duża.
- W przypadku istnienia symboli funkcyjnych niezerowej arności mamy wręcz **nieskończenie wiele** termów zamkniętych – ratunkiem **twierdzenie Herbranda** (skończony podzbiór wystarczy).

Wnioskowanie w logice I-go rzędu (3)

Uogólniona reguła Modus Ponens:

$$\frac{p'_1, \ldots, p'_n, (p_1 \wedge \cdots \wedge p_n) \Rightarrow q}{Podst(\theta, q)}$$

gdzie p_i, p'_i i q są formułami atomowymi, a θ takim podstawieniem, że dla każdego $i = 1, \ldots, n$, $Podst(\theta, p_i) = Podst(\theta, p'_i)$.

Przykład:

$$\frac{\textit{Podał}(\textit{Nowak}, \textit{Kowal}),\ \textit{Strzelił}(\textit{Kowal}),\ \textit{Podał}(x, y) \wedge \textit{Strzelił}(y) \Rightarrow \textit{Asysta(x)}}{Asysta(\textit{Nowak})}$$

Podstawienie $\theta = \{x/\textit{Nowak}, y/\textit{Kowal}\}$.

Wnioskowanie w logice I-go rzędu (4)

Proces unifikacji wyznacza (o ile się da, tzn. o ile istnieje) podstawienie unifikujące dwie formuły (czyniące z nich syntaktycznie tożsame):

$$\text{Unif}(p,q) = \theta \Rightarrow \text{Podst}(\theta,p) = \text{Podst}(\theta,q).$$

Przykłady:

$$\text{Unif}(\text{Lubi}(x,\text{lody}), \text{Lubi}(\text{Jan},y)) = \{x/\text{Jan}, y/\text{lody}\}$$

$$\text{Unif}(\text{Lubi}(x,\text{Córka}(\text{Szef}(\text{Piotr}))), \text{Lubi}(\text{Piotr},\text{Córka}(y))) = \{x/\text{Piotr}, y/\text{Szef}(\text{Piotr})\}$$

$$\text{Unif}(2+2,x) = \{x/2+2\}$$

Wnioskowanie w logice I-go rzędu (5)

Prototype: ***Unif(x, y, θ)***
Input: x,y – zmienne, stałe, listy bądź operatory z arg., θ – podstawienie

1. **if** x=y **then return** θ
2. **if** x jest zmienną **then return** UnifZ(x, y, θ)
3. **if** y jest zmienną **then return** UnifZ(y, x, θ)
4. **if** x i y są oper. z arg. **then**
   return Unif(op(x), op(y), θ) $\wedge$ Unif(arg(x), arg(y), θ)
5. **if** x i y są niepustymi listami **then**
   return Unif(głowa(x), głowa(y), θ) $\wedge$ Unif(ogon(x), ogon(y), θ)
6. **return** porażka

Prototype: ***UnifZ(z, x, θ)***
Input: z – zmienna, x – wyrażenie, θ – podstawienie

1. **if** $\{z/v\} \in \theta$ **then return** Unif(v, x, θ)
2. **if** $\{x/v\} \in \theta$ **then return** UnifZ(z, v, θ)
3. **if** z występuje w x **then return** porażka
4. **return** dodaj $\{z/x\}$ do θ

Wnioskowanie w logice I-go rzędu (6)

Prototype: ***WPrzód(BW,*** α***)***

1. **do**
   1. nowe $\leftarrow \emptyset$
   2. **for each** zdanie r w BW **do**
      - $(p_1 \wedge \cdots \wedge p_n \Rightarrow q) \leftarrow$ Standaryzacja(r)
      - **for each** θ t.że Podst(θ,$p_1 \wedge \cdots \wedge p_n$) = Podst($\theta$,$p'_1 \wedge \cdots \wedge p'_n$) dla pewnych $p'_1, \ldots, p'_n \in$ BW **do**
        1. $q' \leftarrow$Podst(θ,q)
        2. **if** nowe lub BW zawierają formułę równoważną z q' **then** kontynuuj
        3. nowe += q'
        4. $\phi \leftarrow$ Unif(q', α)
        5. **if** $\phi \neq$ porażka **then return** ϕ
   3. dodaj nowe do BW
2. **while** nowe $\neq \emptyset$
3. **return** fałsz

Wnioskowanie w logice I-go rzędu (7)

Przykład:

Prawo mówi, że Amerykanin popełnia przestępstwo sprzedając broń wrogim narodom. Państwo Nono, nieprzyjaciel Ameryki, posiada pewne pociski, wszystkie sprzedane mu przez Amerykanina, pułkownika Westa.

Formalnie:

1. Amerykanin(x) $\wedge$ Broń(y) $\wedge$ Sprzedaje(x,y,z) $\wedge$ Wrogi(z) $\Rightarrow$ Przestępca(x)
2. $\exists_x$ Posiada(Nono,x) $\wedge$ Pocisk(x)
   - Posiada(Nono, P0)
   - Pocisk(P0)
3. Pocisk(x) $\wedge$ Posiada(Nono,x) $\Rightarrow$ Sprzedaje(West, x, Nono)
4. Pocisk(x) $\Rightarrow$ Broń(x)
5. Nieprzyjaciel(x, Ameryka) $\Rightarrow$ Wrogi(x)
6. Amerykanin(West)
7. Nieprzyjaciel(Nono, Ameryka)

Wnioskowanie w logice I-go rzędu (8)

Wnioskowanie w przód

- Dopasowywanie wzorców (ang. *pattern matching*) może być **kosztowne**.
- **Powtórne sprawdzanie** dopasowań jest rozrzutnością.
- Generowane są całkiem **niepotrzebne konkluzje**.

Metody na **przyspieszenie**:

- **porządkowanie** czynników koniunkcji, heurystyka **najbardziej ograniczonej zmiennej** znana z szukania z ograniczeniami,
- **przyrostowe wnioskowanie w przód** – tylko wnioskowania z udziałem formuł wygenerowanych w poprzednim kroku,
- **ograniczanie** pasujących obiektów do „**magicznych zbiorów**” – hybryda wnioskowania w przód i wstecz.

Wnioskowanie w logice I-go rzędu (9)

Wnioskowanie wstecz

Prototype: ***Wstecz(BW, cele,*** θ***)***

Input: BW – baza wiedzy, cele – do udowodnienia, θ – bieżące podstawienie

1. **if** cele = $\emptyset$ **then return** $\{\theta\}$
2. $q' \leftarrow$ Podst(θ,głowa(cele))
3. **for each** r $\in$ BW **do**
   1. $(p_1 \wedge \cdots \wedge p_n \Rightarrow q) \leftarrow$ Standaryzacja(r)
   2. $\theta' \leftarrow$ Unif(q,q′)
   3. **if** $\theta' \neq$ porażka **then**
      1. noweCele $\leftarrow [p_1, \ldots, p_n | ogon(cele)]$
      2. odpowiedzi $\leftarrow$ Wstecz(BW, noweCele, Złożenie(θ',θ)) $\cup$ odpowiedzi
4. **return** odpowiedzi

Rezolucja w logice klasycznej pierwszego rzędu

Koniunkcyjna postać normalna – koniunkcja klauzul (dysjunkcji literałów) z wszystkimi zmiennymi objętymi kwantyfikatorami ogólnymi (kwantyfikatory ukrywa się).

Przykład: Reguła

$$\forall_{x,y,z}\ \textit{Amerykanin(x)} \wedge \textit{Broń(y)} \wedge \textit{Sprzedaje(x,y,z)} \wedge \textit{Wrogi(z)} \Rightarrow \textit{Przestępca(x)}$$

wygląda w KPN następująco:

$$\neg\textit{Amerykanin(x)} \vee \neg\textit{Broń(y)} \vee \neg\textit{Sprzedaje(x,y,z)} \vee \neg\textit{Wrogi(z)} \vee \textit{Przestępca(x)}$$

Wnioskowanie w logice I-go rzędu (11)

Procedura **konwersji do KPN**:

- Eliminacja implikacji
- Przeniesienie negacji w głąb
- Standaryzacja zmiennych
- **Skolemizacja** – usuwanie kwantyfikatorów szczegółowych przez wprowadzanie symboli funkcyjnych (Skolema; zero lub więcej argumentowych – arność równa liczbie zmiennych pod kwantyfikatorami ogólnymi)
- Porzucenie kwantyfikatorów ogólnych
- Przenoszenie operatorów $\vee$ i $\wedge$

Wnioskowanie w logice I-go rzędu (12)

Reguła rezolucji

$$\frac{l_1 \vee \cdots \vee l_k \qquad m_1 \vee \cdots \vee m_n}{\text{Podst}(\theta, l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \vee \cdots \vee l_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)}$$

gdzie $\theta = \mathit{Unif}(l_i, m_j)$.

- **Pełna rezolucja** szuka podstawień w podzbiorach literałów a nie tylko w dwóch pojedyńczych literałach.
- **Faktoring** – usuwanie unifikowalnych literałów.

Schemat dowodu zupełności odrzucania:

1. **Tw. Herbranda**: Jeśli zbiór S jest niespełnialny, to istnieje niespełnialny zbiór formuł zamkniętych $S_0 \subseteq S$.
2. Rezolucja **zdaniowa jest zupełna** dla zdań zamkniętych.
3. **Lemat podnoszenia** – dla każdego dowodu zdaniowego istnieje odpowiadający mu dowód w l.k. I-go rzędu.

Twierdzenie Gödla o niezupełności

Dla każdego niesprzecznego systemu formalnego z regułą indukcji ***istnieje prawdziwa formuła, która nie posiada dowodu****.*

Innymi słowy: *Istnieją prawdziwe zdania o arytmetyce, które nie mogą być udowodnione.*

Idee dowodu:

- Numerowanie formuł/programów α – liczby Gödla $\#\alpha$
- Dla liczby naturalnej j i zbioru prawdziwych formuł A, formuła $\alpha(j, A)$: $\forall_i$ *i nie jest numerem Gödla dowodu formuły o numerze Gödla j używającego tylko przesłanek z A.*
- $\sigma = \alpha(\#\sigma, A)$
- Załóżmy, że σ jest dowodliwe z A. σ jest fałszywa, bo twierdzi, że nie ma dowodu. Skoro fałszywa formuła σ jest dowodliwa z A, to A nie może zawierać tylko prawdziwych formuł – sprzeczność. Zatem σ nie jest dowodliwe z A, a ono właśnie to mówi, więc jest prawdziwe.

Programowanie w logice – język prolog (1)

Prolog – podstawowe idee:

- Baza wiedzy składa się z klauzul hornowskich
  reguła: $p_1 \wedge \cdots \wedge p_n \Rightarrow q$
  fakt: q
- Pytania:
  $p_1 \wedge \cdots \wedge p_n$
- Wnioskowanie wstecz – szukanie w głąb – niezupełność!
- Unifikacja
- Podstawienia wyznaczane w trakcie dowodzenia

Syntaktyka prologu:

- Przykład reguły:
  `ojciec(X,Y) :- rodzic(X,Y), mezczyzna(X).`
- reguły w notacji Kowalskiego – operator „`:-`” (czyt. „o ile”)
- kropka kończy deklarację
- semantyka operatorów:

`p :- q`	$q \Rightarrow p$
`p, q`	$p \wedge q$
`p; q`	$p \vee q$
`\+ p`	$\neg p$ (niedowodliwość, a nie nieprawdziwość)

Programowanie w logice – język prolog (3)

Podstawowe zasady:

- dany predykat definiowany (nieprzerwanym) ciągiem deklaracji w jednym pliku,
- ładowanie do bazy wiedzy przez
  `?- consult(nazwaPliku).`
  lub
  `?- [nazwaPliku].`
- symbole funkcyjne i predykatowe piszemy małą literą,
- zmienne rozpoczynamy wielka literą,
- podkreślenie rozpoczyna nazwy zmiennych ignorowanych w rozwiązaniach,
- komentarze do końca linii – znak %.

Programowanie w logice – język prolog (4)

Różne ważne szczegóły prologu:

- arytmetyka: `X is 2+3`
- operator = to unifikacja, a nie przypisanie (`X = 2+3`) się powiedzie i podstawi wyrażenie (term) `2+3` za `X`
- operator \= zastępuje `\+ term1 = term2`
- operatory == i \== sprawdzają równoważność termów (różne zmienne są równoważne, gdy są zunifikowane)
- uwaga na kolejności deklaracji – przykład predykatu `silnia`

```
silnia(0,1).
silnia(N,S) :- N1 is N-1, silnia(N1,S1), S is S1*N.
```

Deklaratywność a imperatywność – elementarna uwaga!
Definiujemy kiedy **predykaty są prawdziwe bądź nie**, a nie „co mają robić”.

Listy w prologu

- Lista to para głowa-ogon, jak w języku scheme.
- Notacja:
  - lista pusta: `[]`
  - lista elementów: `[1,2,3,4,5]`
  - lista niepusta: `[H|T]` (**Uwaga!** kreska pionowa, a nie przecinek)
- Funkcje obsługi list są zwykle rekurencyjne (jak w schemie).
- Przykład: długością listy jest N:

```
dlugosc([],0).
dlugosc([_|T], N) :- dlugosc(T,N2), N is N2+1.
```

- Podstawowe zadania: zdefiniować predykaty `sumaElementow/2`, `polaczenieList/3`, `odwrocenieListy/2`, `mapowanie/3`, `filtrListy/3`.

Programowanie w logice – język prolog (6)

Śledzenie programów

- predykaty `trace`, `notrace`, `debug`, `nodebug`, `spy`, `nospy`

Odcięcie – bardzo ważne narzędzie sterujące szukaniem dowodów

- symbol `!` – odcięcie możliwości powrotów procesu szukania,
- przykład silni, równoważności.

Predykaty dynamiczne – zmiana bazy danych w runtime

- `dynamic/1`
- `asserta` i `assertz`
- `retract` i `retractAll`

Zagadka logiczna Mr S and Mr P problem.

There are two numbers M and N such that $1 < M, N < 100$. Mr S is told their sum S and Mr P is told their product P. The following dialogue takes place:

Mr P: *I don't know the numbers.*

Mr S: *I knew you didn't know them. I don't know them either.*

Mr P: *Now I know the numbers!*

Mr S: *Now I know them too!*

What are the numbers?

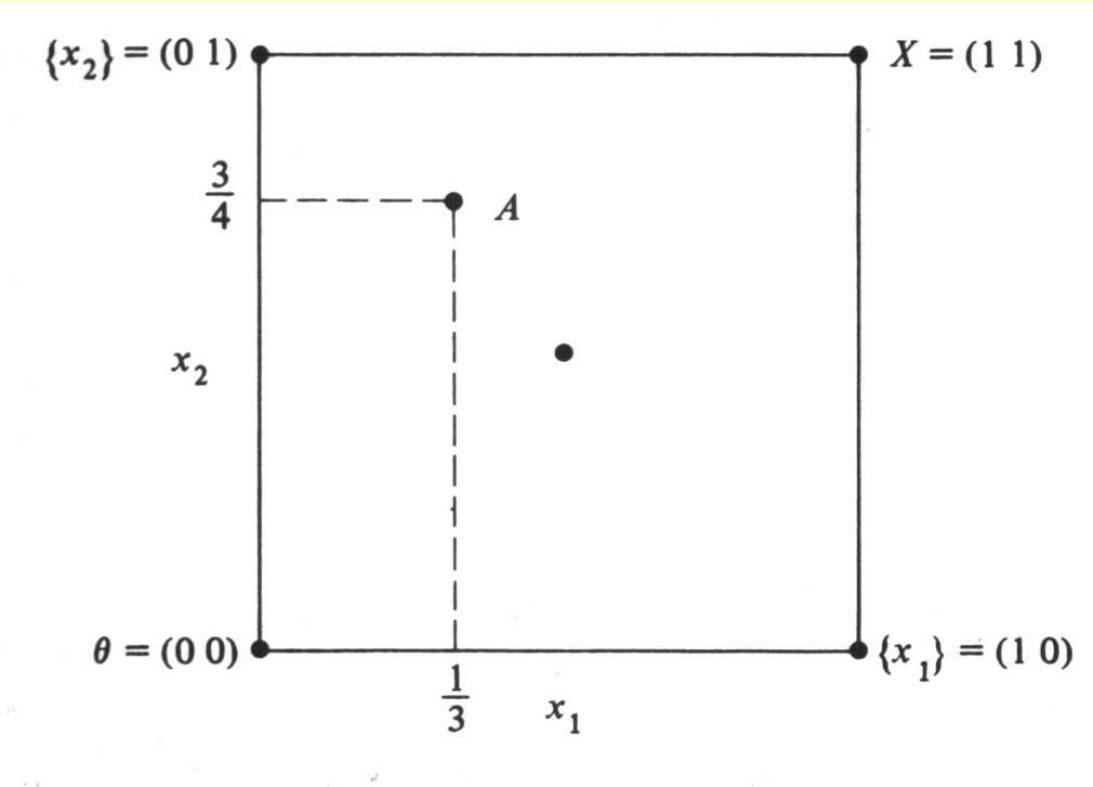
Rozwiązanie w następnym odcinku.

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

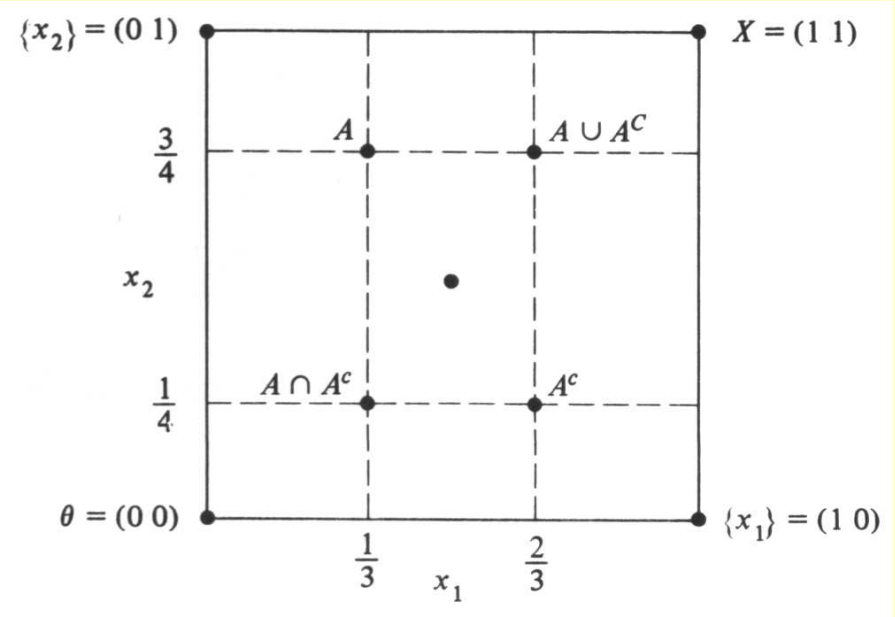
- Zbiory rozmyte jako funkcje: $A \equiv m_A : X \to [0,1]$
- Zbiór rozmytych podzbiorów zbioru $X = \{x_1, \ldots, x_n\}$ to kostka $F(2^X) = I^n$



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

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:

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

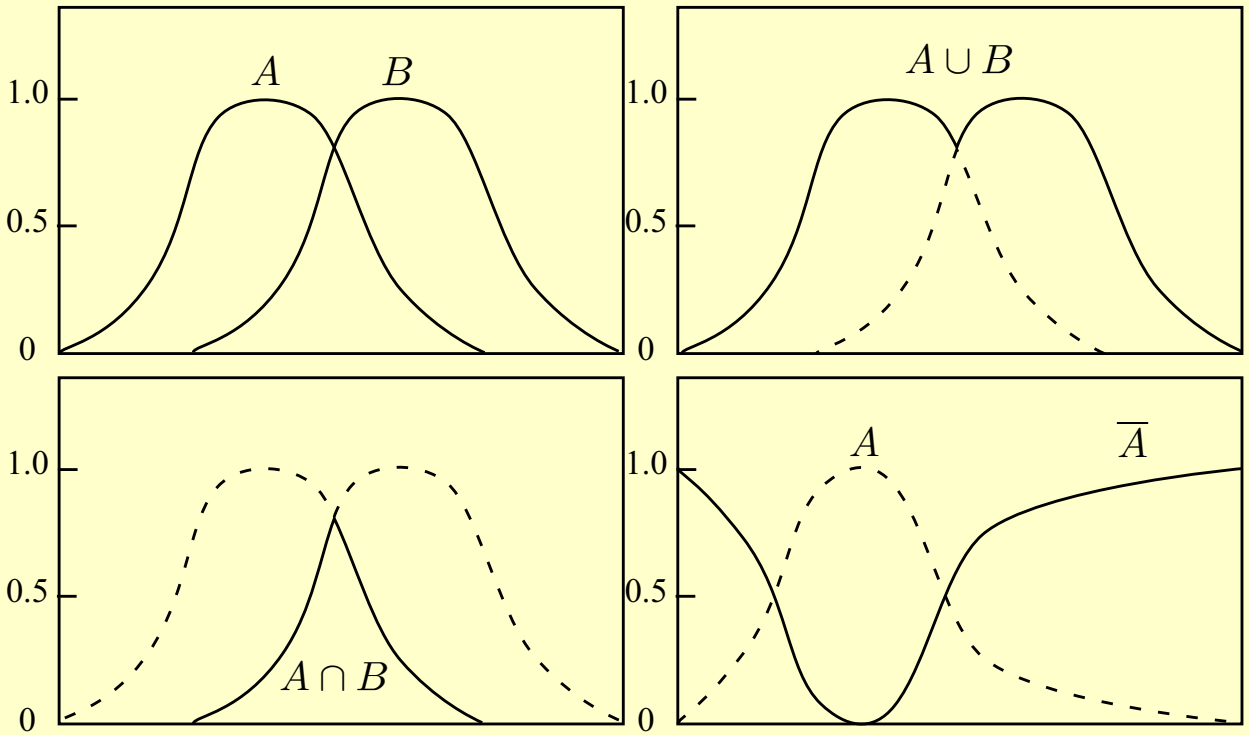
$$B = \{0/a, 0.5/b, 1/c\}$$

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

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

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

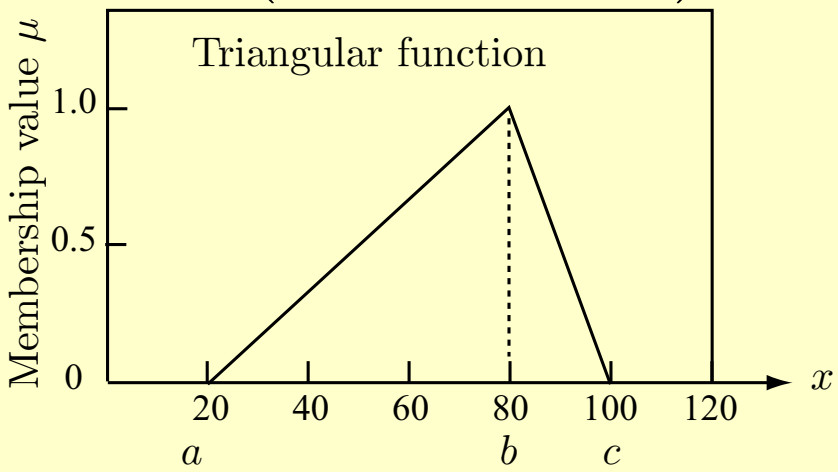
Dla zbiorów ciągłych:



Popularne funkcje przynależności

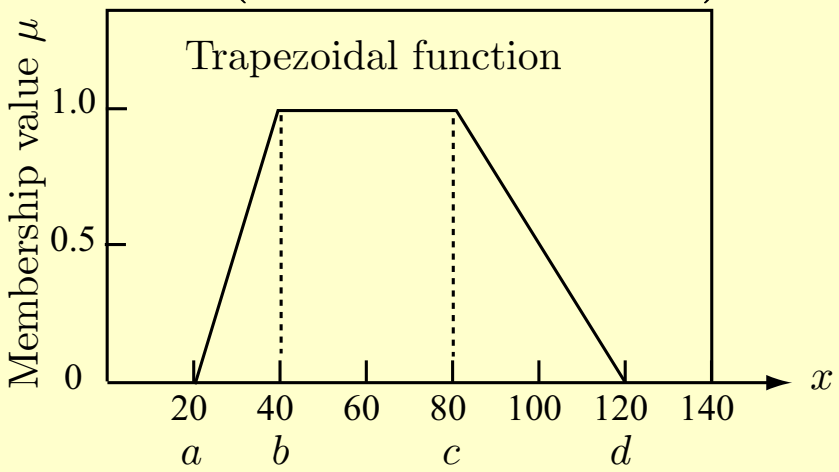
trójkątna

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



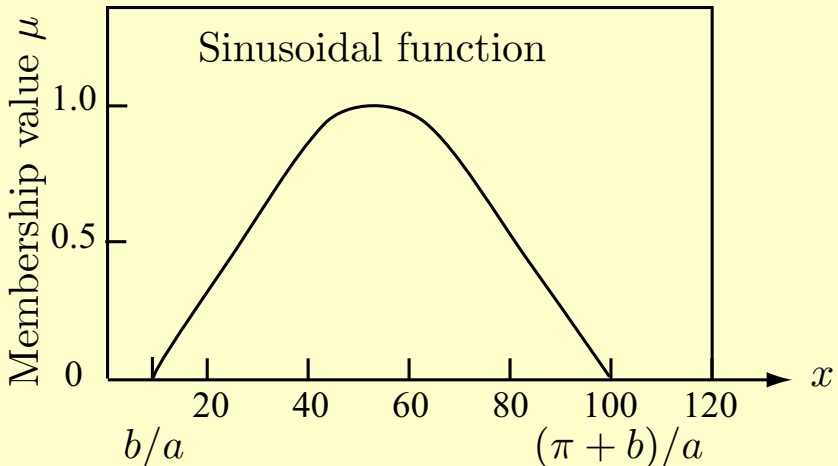
trapezowa

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



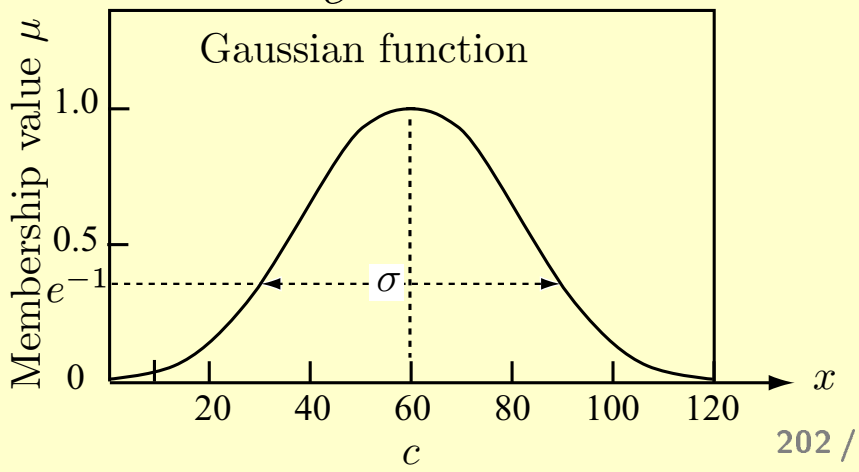
sinusoidalna

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



gausowska

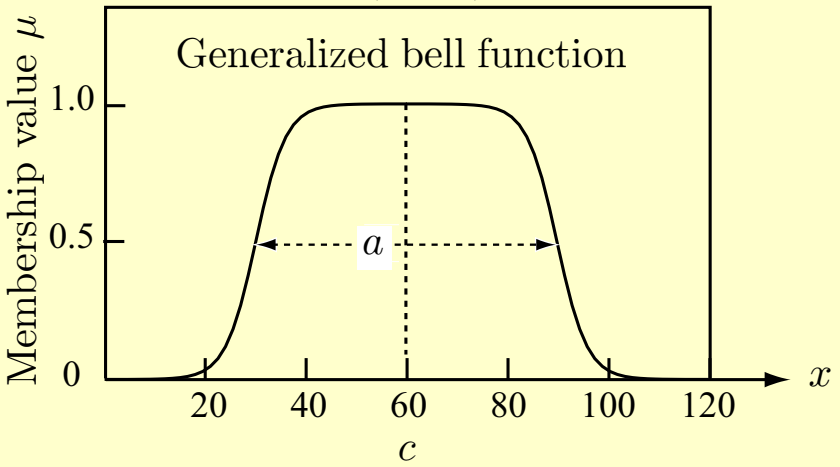
$$e^{-\left(\frac{x-c}{\sigma}\right)^2}$$



Popularne funkcje przynależności

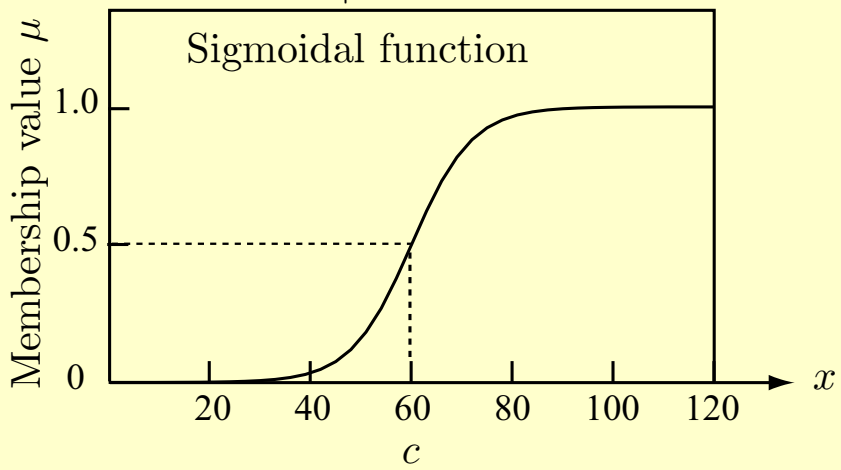
dzwonowa

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



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:

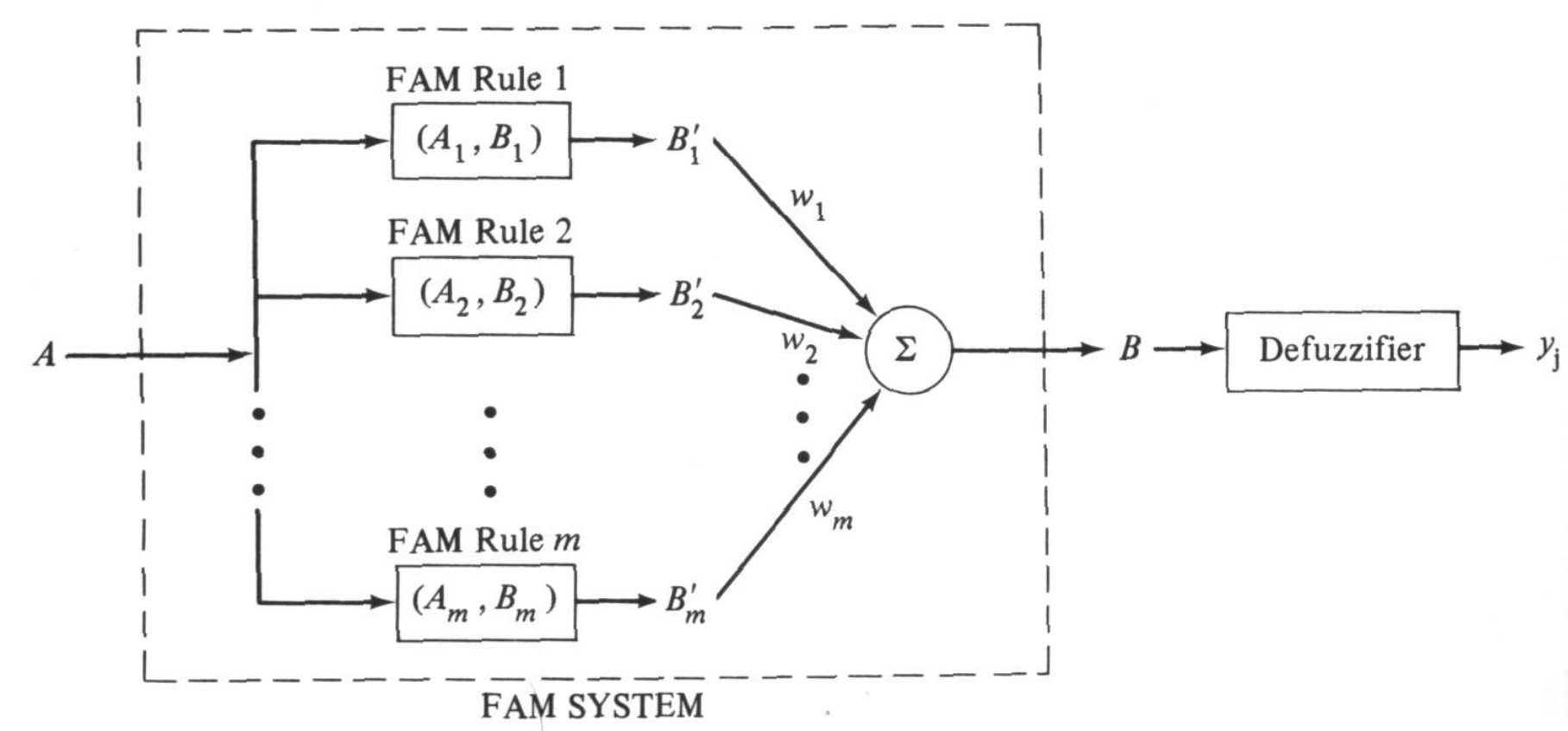
  Jeżeli X jest A to Y jest B.

- Wejście aktywuje odpowiednio każdą regułę (A_i, B_i) i jest mapowane na wyjście B'_i (częściowo aktywowane B).
- Ostateczne wyjście jest ważoną sumą

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

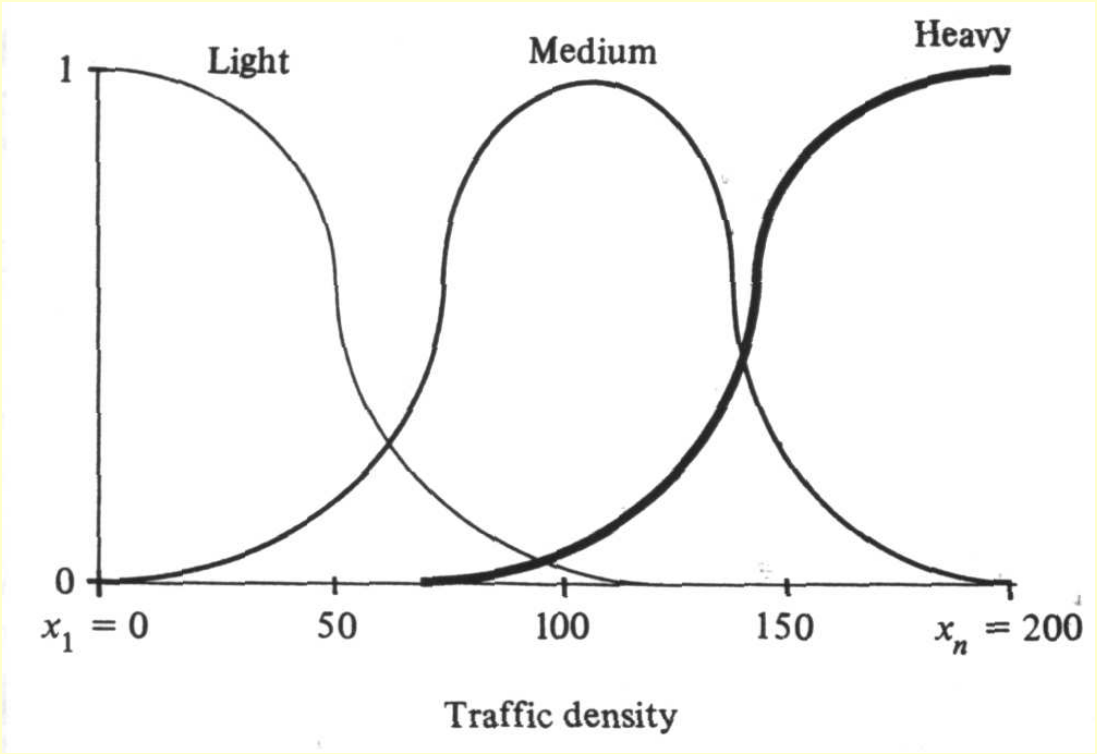
gdzie w_i to wagi odpowiadające wiarygodności reguły, jej sile asocjacji itp.

- 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