Sztuczna Inteligencja i Systemy Doradcze

Przeszukiwanie przestrzeni stanów — algorytmy heurystyczne

Strategie heurystyczne

Strategie *heurystyczne* korzystają z dodatkowej, heurystycznej funkcji oceny stanu (np. szacującej koszt rozwiązania od bieżącego stanu do celu):

- ♢ Przeszukiwanie pierwszy najlepszy
  - ♢ Przeszukiwanie zachłanne
  - ♢ Przeszukiwanie A*
  - ♢ Rekurencyjne przeszukiwanie pierwszy najlepszy
- ♢ Iteracyjne poprawianie
  - ♢ Przeszukiwanie lokalne zachłanne (hill-climbing)
  - ♢ Przeszukiwanie z tabu
  - ♢ Symulowane wyżarzanie
  - ♢ Przeszukiwanie o zmiennej głębokości
  - ♢ Algorytm genetyczny

Przeszukiwanie pierwszy najlepszy

Używa *funkcji użyteczności*, która dla każdego stanu ocenia jego "użyteczność"

⇒ Wykonuje ekspansję najbardziej użytecznego stanu spośród wcześniej nieodwiedzonych stanów

Implementacja: *fringe* jest kolejką porządkującą węzły rosnąco według wartości funkcji użyteczności (im mniejsza wartość funkcji, tym węzeł jest bardziej "użyteczny").

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
       if fringe is empty then return failure
       node ← REMOVE-FRONT(fringe)
       if GOAL-TEST[problem] applied to STATE(node) succeeds return node
       fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

Przypadki szczególne: przeszukiwanie zachłanne, przeszukiwanie A*

Przeszukiwanie zachlanne

Funkcja użyteczności = heurystyczna funkcja oceny stanu $h(n)$

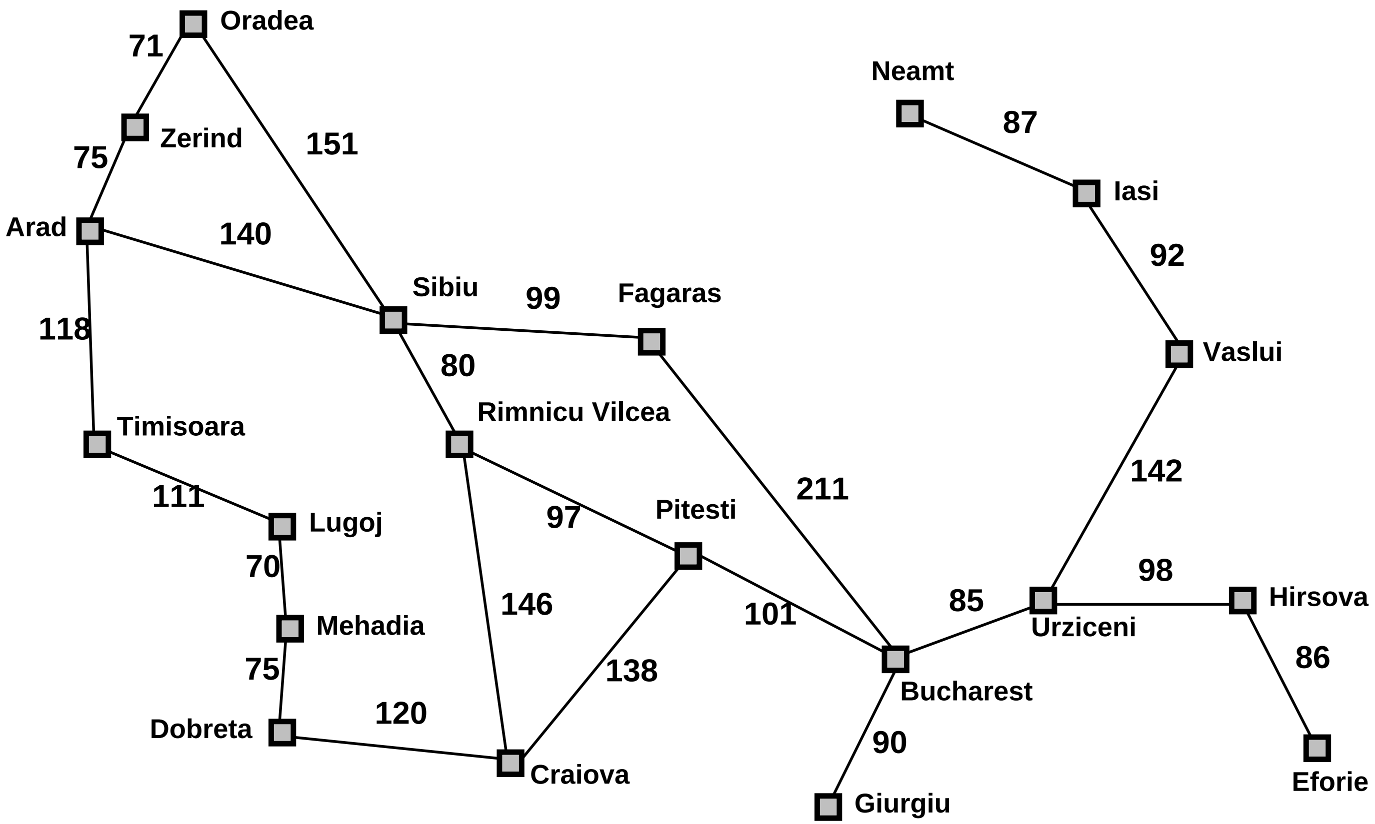
$\Rightarrow$ Szacuje koszt rozwiązania z bieżącego stanu n do najbliższego stanu docelowego

Przeszukiwanie zachłanne wykonuje ekspansję tego węzła, który wydaje się być najbliżej celu

function TREE-SEARCH(*problem*, *fringe*) **returns** a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)
    loop do
        if *fringe* is empty **then return** failure
        node ← REMOVE-FRONT(*fringe*)
        if GOAL-TEST[*problem*] applied to STATE(*node*) succeeds **return** *node*
        fringe ← INSERTALL(EXPAND(*node*, *problem*), *fringe*)

Przeszukiwanie zachlanne: przyklad

$h_{\mathrm{SLD}}(n)$ = odległość w linii prostej z miasta n do Bukaresztu



Straight−line distance
to Bucharest

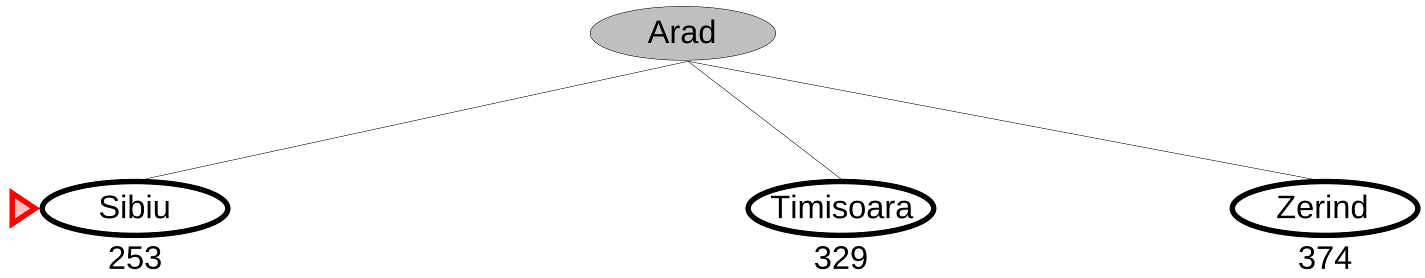
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Przeszukiwanie zachlanne: przyklad

$h_{\mathrm{SLD}}(n)$ = odległość w linii prostej z miasta n do Bukaresztu

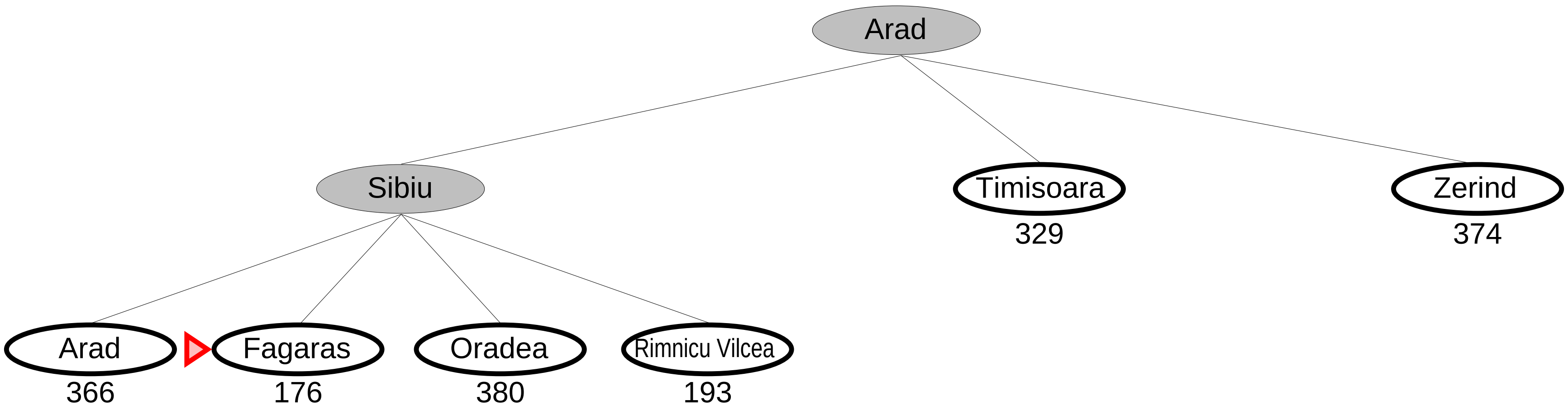
Przeszukiwanie zachlanne: przyklad

$h_{\mathrm{SLD}}(n)$ = odległość w linii prostej z miasta n do Bukaresztu



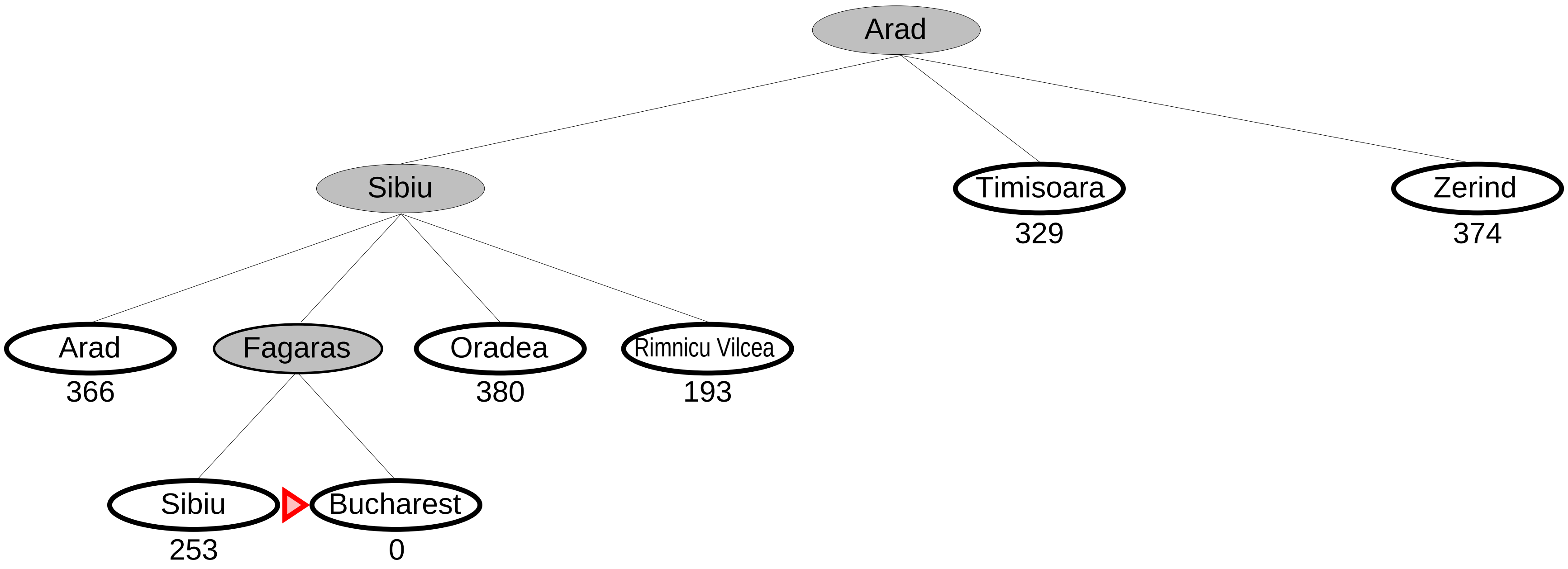
Przeszukiwanie zachlanne: przyklad

$h_{\mathrm{SLD}}(n)$ = odległość w linii prostej z miasta n do Bukaresztu



Przeszukiwanie zachlanne: przyklad

$h_{\mathrm{SLD}}(n)$ = odległość w linii prostej z miasta n do Bukaresztu



Przeszukiwanie zachlanne: wlasnosci

Zupełność??

Przeszukiwanie zachlanne: wlasnosci

Zupełność??

Brak – może się zapętlać, np. jeśli celem jest Oradea,
Iasi → Neamt → Iasi → Neamt →

Zupełne w przestrzeniach skończonych z wykrywaniem i eliminacją powtarzających się stanów

Złożoność czasowa??

Przeszukiwanie zachlanne: wlasnosci

Zupełność??

Brak – może się zapętlać, np. jeśli celem jest Oradea,
Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$

Zupełne w przestrzeniach skończonych z wykrywaniem i eliminacją powtarzających się stanów

Złożoność czasowa??

$O(b^m)$, ale dobra heurystyka może dramatycznie przyspieszać

Złożoność pamięciowa??

Przeszukiwanie zachlanne: wlasnosci

Zupełność??

Brak – może się zapętlać, np. jeśli celem jest Oradea,
Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$

Zupełne w przestrzeniach skończonych z wykrywaniem i eliminacją powtarzających się stanów

Złożoność czasowa??

$O(b^m)$, ale dobra heurystyka może dramatycznie przyspieszać

Złożoność pamięciowa??

$O(b^m)$ — przechowuje wszystkie węzły w pamięci

Optymalność??

Przeszukiwanie zachlanne: wlasnosci

Zupełność??

Brak – może się zapętlać, np. jeśli celem jest Oradea,
Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$

Zupełne w przestrzeniach skończonych z wykrywaniem i eliminacją powtarzających się stanów

Złożoność czasowa??

$O(b^m)$, ale dobra heurystyka może dramatycznie przyspieszać

Złożoność pamięciowa??

$O(b^m)$ — przechowuje wszystkie węzły w pamięci

Optymalność??

Brak

Przeszukiwanie A*

Pomysł: unika ekspansji stanów, dla których dotychczasowa ścieżka jest już kosztowna

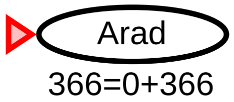
Funkcja użyteczności: $f(n) = g(n) + h(n)$

$g(n)$ = dotychczasowy koszt dotarcia do stanu n
$h(n)$ = oszacowanie kosztu od stanu bieżącego n do stanu docelowego
$f(n)$ = oszacowanie pełnego kosztu ścieżki od stanu początkowego do celu prowadzącej przez stan n

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem] applied to STATE(node) succeeds return node
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

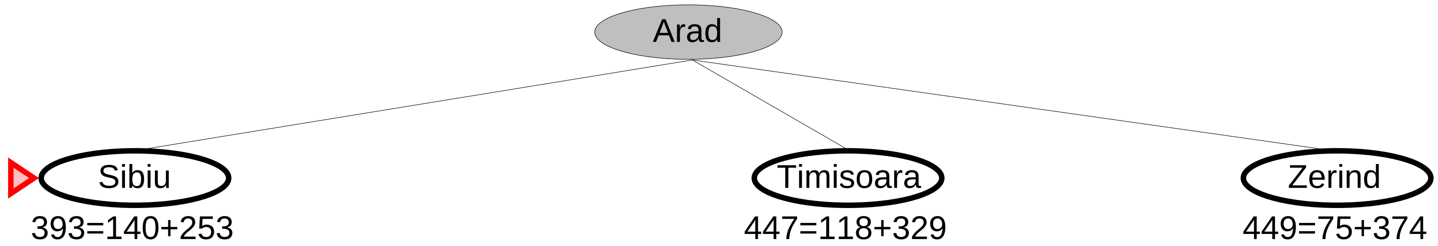
Przeszukiwanie A*: przyklad

Funkcja użyteczności: $f(n) = g(n) + h(n)$
$g(n)$ = dotychczasowy koszt dotarcia do stanu n
$h(n)$ = oszacowanie kosztu od stanu bieżącego n do stanu docelowego



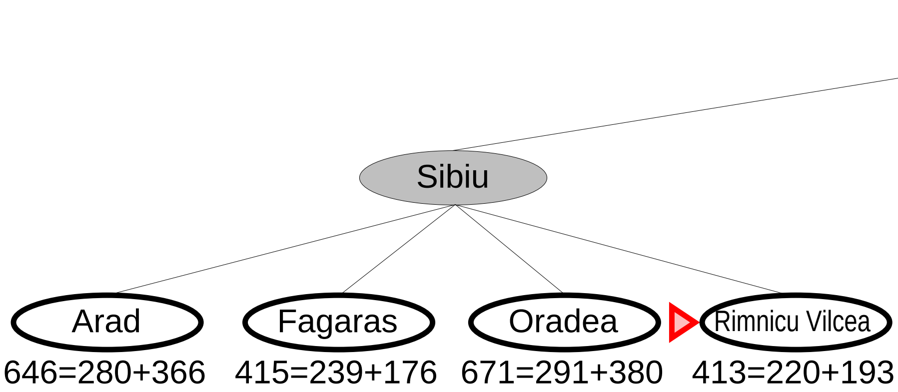
Przeszukiwanie A*: przyklad

Funkcja użyteczności: $f(n) = g(n) + h(n)$
$g(n)$ = dotychczasowy koszt dotarcia do stanu n
$h(n)$ = oszacowanie kosztu od stanu bieżącego n do stanu docelowego



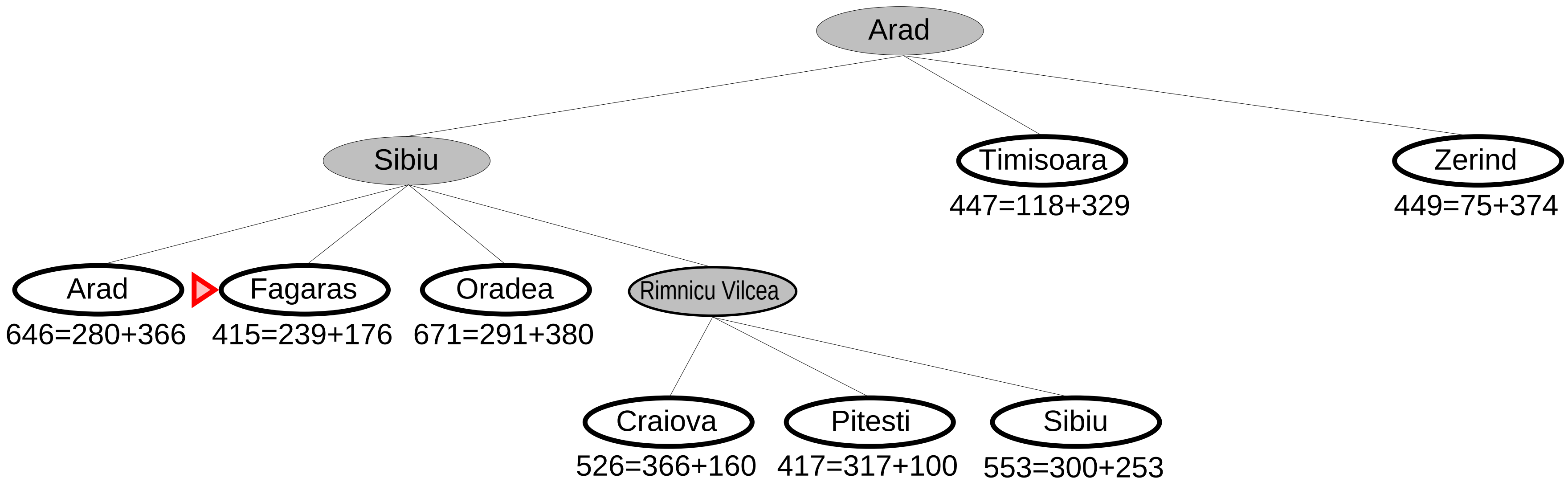
Przeszukiwanie A*: przyklad

Funkcja użyteczności: $f(n) = g(n) + h(n)$
$g(n)$ = dotychczasowy koszt dotarcia do stanu n
$h(n)$ = oszacowanie kosztu od stanu bieżącego n do stanu docelowego



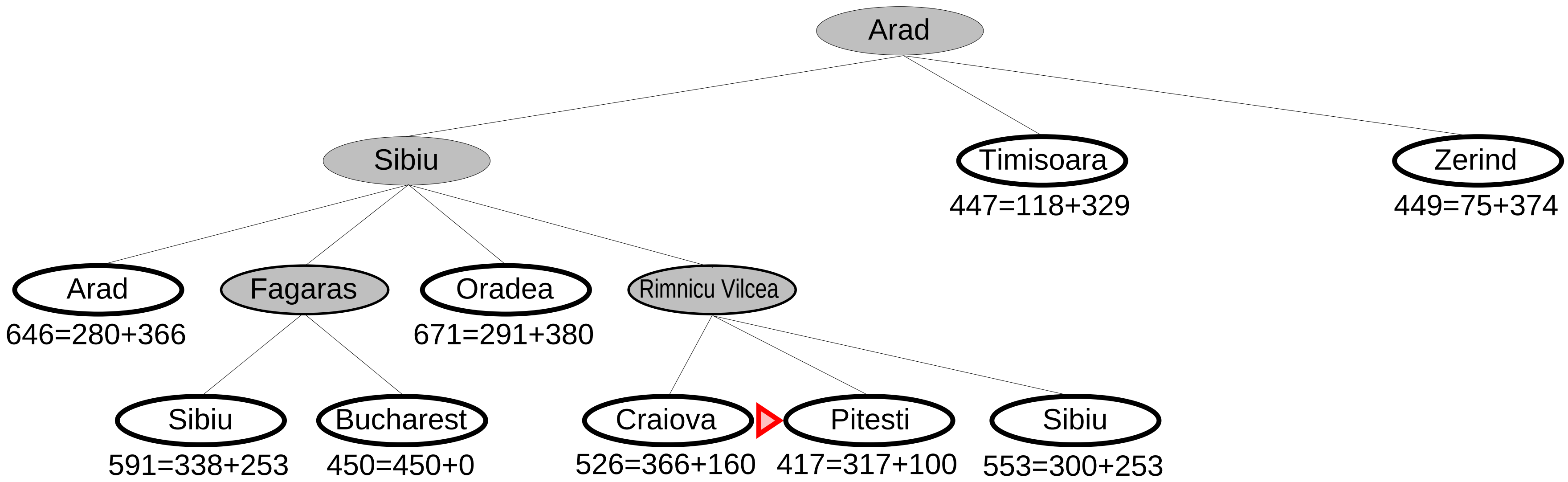
Przeszukiwanie A*: przyklad

Funkcja użyteczności: $f(n) = g(n) + h(n)$
$g(n)$ = dotychczasowy koszt dotarcia do stanu n
$h(n)$ = oszacowanie kosztu od stanu bieżącego n do stanu docelowego



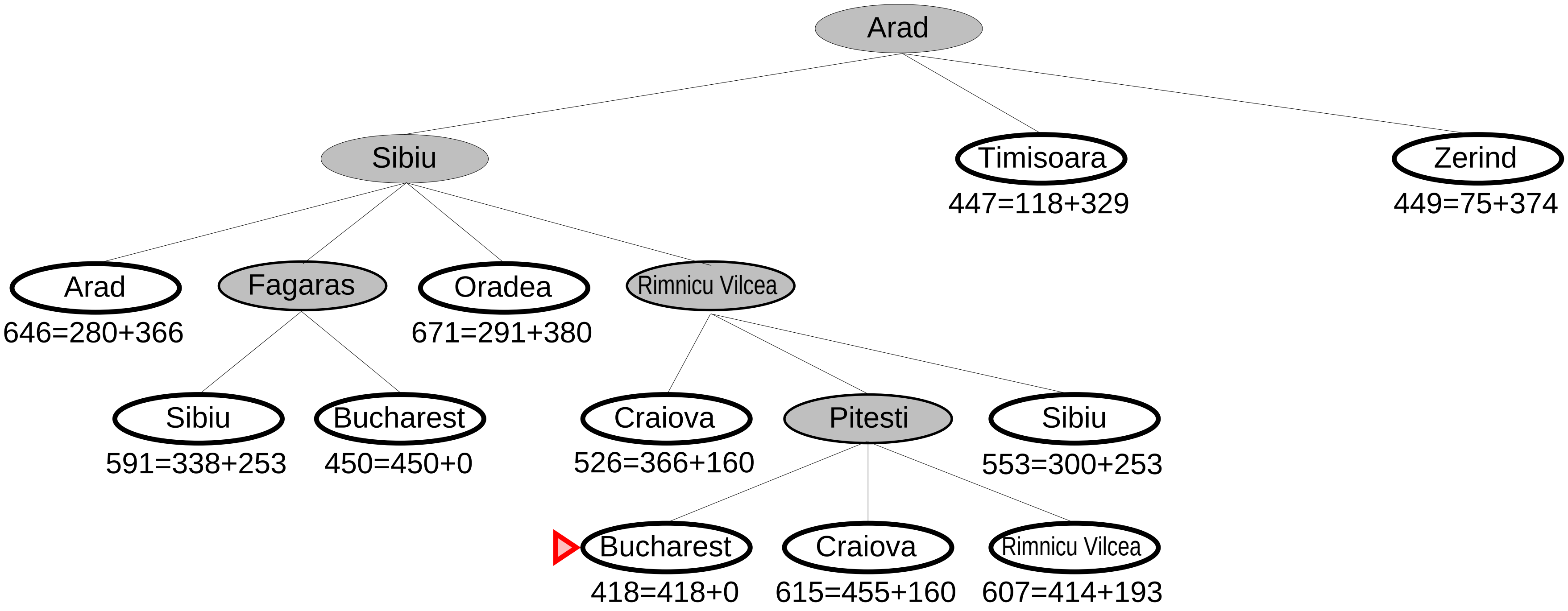
Przeszukiwanie A*: przyklad

Funkcja użyteczności: $f(n) = g(n) + h(n)$
$g(n)$ = dotychczasowy koszt dotarcia do stanu n
$h(n)$ = oszacowanie kosztu od stanu bieżącego n do stanu docelowego



Przeszukiwanie A*: przyklad

Funkcja użyteczności: $f(n) = g(n) + h(n)$
$g(n)$ = dotychczasowy koszt dotarcia do stanu n
$h(n)$ = oszacowanie kosztu od stanu bieżącego n do stanu docelowego



Przeszukiwanie A*: wlasnosci

Zupełność??

Przeszukiwanie A*: wlasnosci

Zupełność?? Tak, jeśli nie ma nieskończonie wiele stanów z $f \leq f(G)$

Złożoność czasowa??

Przeszukiwanie A*: wlasnosci

Zupełność?? Tak, jeśli nie ma nieskończonie wiele stanów z $f \leq f(G)$

Złożoność czasowa?? Wykładniczy względem [bład względny $h \times$ dług. rozw.]

Złożoność pamięciowa??

Przeszukiwanie A*: wlasnosci

Zupełność?? Tak, jeśli nie ma nieskończonie wiele stanów z $f \leq f(G)$

Złożoność czasowa?? Wykładniczy względem [bład względny $h \times$ dług. rozw.]

Złożoność pamięciowa?? Przechowuje wszystkie węzły w pamięci

Optymalność??

Przeszukiwanie A*: wlasnosci

Zupełność?? Tak, jeśli nie ma nieskończonie wiele stanów z $f \leq f(G)$

Złożoność czasowa?? Wykładniczy względem [błąd względny $h \times$ dług. rozw.]

Złożoność pamięciowa?? Przechowuje wszystkie węzły w pamięci

Optymalność?? Tak, jeśli heurystyka h jest dopuszczalna

Przeszukiwanie A*: wlasnosci

Zupełność?? Tak, jeśli nie ma nieskończonie wiele stanów z $f \leq f(G)$

Złożoność czasowa?? Wykładniczy względem [błąd względny $h \times$ dług. rozw.]

Złożoność pamięciowa?? Przechowuje wszystkie węzły w pamięci

Optymalność?? Tak, jeśli heurystyka h jest dopuszczalna

A* eksploruje wszystkie węzły z $f(n) < C^*$
A* eksploruje niektóre węzły z $f(n) = C^*$
A* nie eksploruje żadnych węzłów z $f(n) > C^*$

Heurystyka dopuszczalna

Ogólnie o każdej funkcji heurystycznej $h(n)$ zakłada się, że $h(n) \geq 0$

Funkcja heurystyczna $h(n)$ jest *dopuszczalna*, jeśli w każdym stanie n spełnia następujący warunek:

$$h(n) \leq h^*(n)$$

gdzie $h^*(n)$ jest *rzeczywistym* kosztem ścieżki od stanu n do celu.

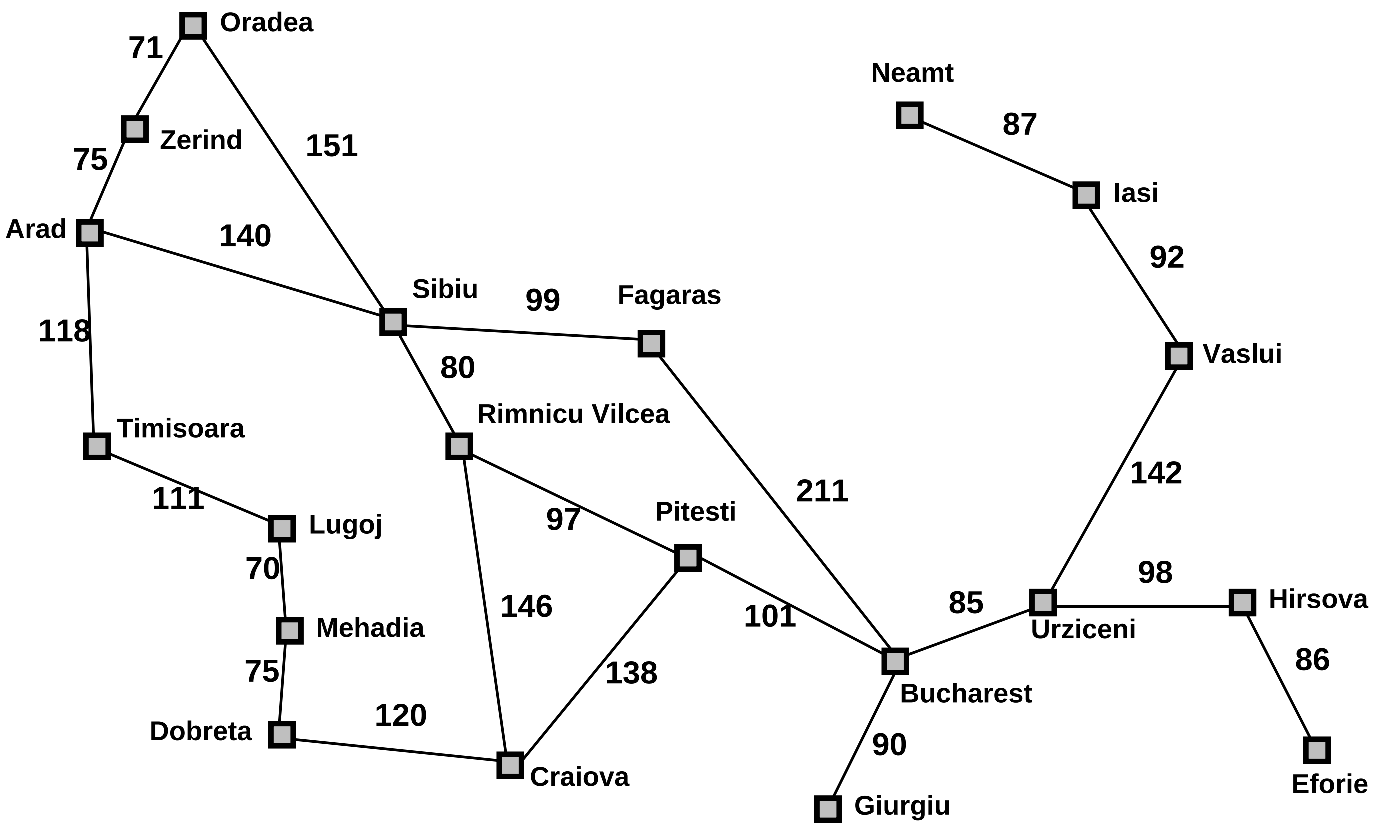
Problem uproszczony

Wersja oryginalnego problemu, dla której koszt rozwiązania jest zawsze nie większy niż koszt rozwiązania problemu oryginalnego

Heurystyka dopuszczalna może być *dokładnym* kosztem rozwiązania uproszczonej wersji problemu

Heurystyka dopuszczalna: najkrotsza droga

Funkcja odległości w linii prostej $h_{\mathrm{SLD}}(n)$ jest dopuszczalna — nigdy nie przekracza rzeczywistej odległości drogowej

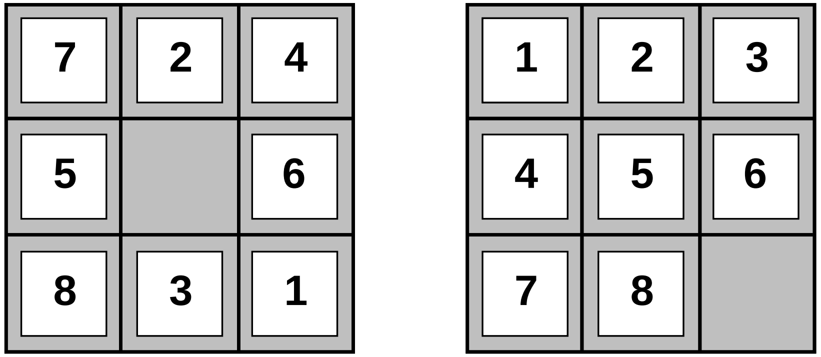


Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Heurystyki dopuszczalne: 8-elementowe puzzle

Uproszczenie 1: klocek może być przesunięty na *dowolne* pole:
$h_1(n)$ = liczba klocków nie będących w docelowym położeniu

Uproszczenie 2: klocek może być przesunięty na *dowolne sąsiednie* pole:
$h_2(n)$ = suma odległości miejskiej (ilości ruchów)
od docelowych miejsc dla poszczególnych klocków



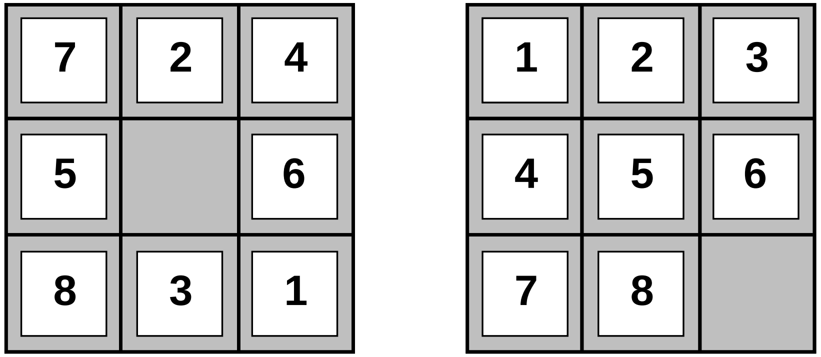
Start State　　Goal State

$h_1(S) =??$
$h_2(S) =??$

Heurystyki dopuszczalne: 8-elementowe puzzle

Uproszczenie 1: klocek może być przesunięty na *dowolne* pole:
$h_1(n)$ = liczba klocków nie będących w docelowym położeniu

Uproszczenie 2: klocek może być przesunięty na *dowolne sąsiednie* pole:
$h_2(n)$ = suma odległości miejskiej (ilości ruchów)
od docelowych miejsc dla poszczególnych klocków

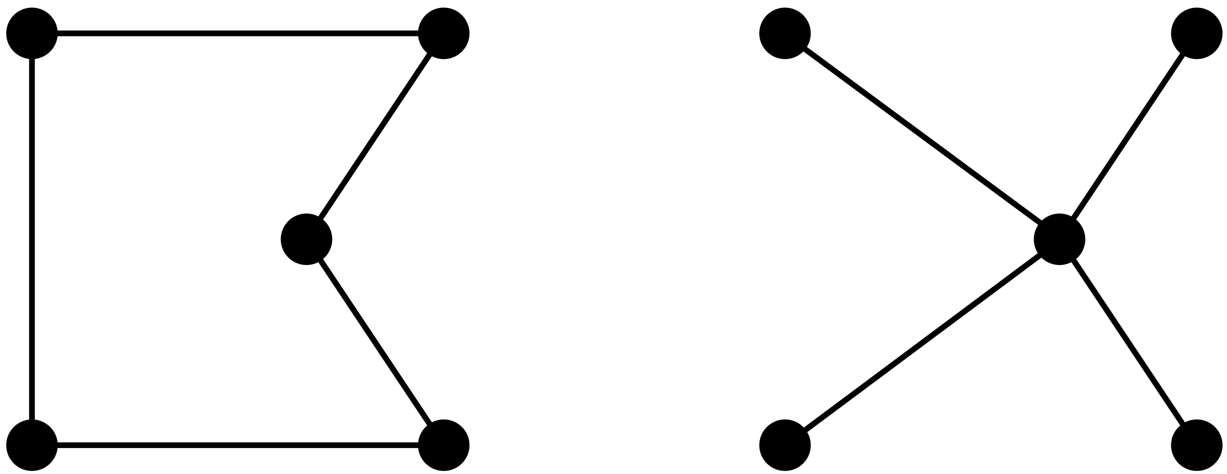


Start State **Goal State**

$h_1(S)$ =?? 6
$h_2(S)$ =?? 4+0+3+3+1+0+2+1 = 14

Heurystyka dopuszczalna: problem komiwojazera

Problem: znajdź najkrótszy *cykl Hamiltona* w grafie, tzn.
najkrótszy cykl przechodzący przez każde miasto dokładnie raz



Fakt: Znalezienie najkrótszego cyklu Hamiltona jest NP-trudne

Uproszczenie: znajdź *minimalne drzewo rozpinające*
Minimalne drzewo rozpinające można znaleźć w czasie $O(n^2)$
i jest uproszczeniem problemu minimalnego cyklu Hamiltona,
bo wszystkie cykle Hamiltona zawierają w sobie drzewo rozpinające

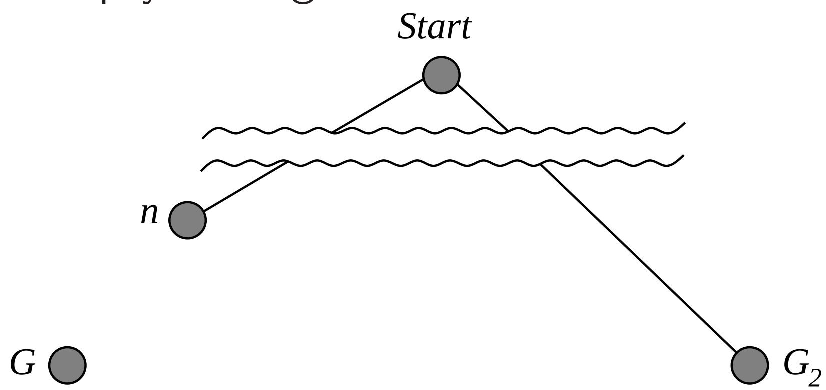
Heurystyka dopuszczalna: optymalnosc A*

Twierdzenie:

Przeszukiwanie A* bez eliminacji powtarzających się stanów przy użyciu heurystyki dopuszczalnej znajduje zawsze rozwiązanie optymalne

Heurystyka dopuszczalna: optymalnosc A*

Dowód: Załóżmy, że stan docelowy G_2 o nieoptymalnym koszcie rozwiązania został wstawiony do kolejki stanów. Niech n będzie dowolnym stanem na najkrótszej ścieżce do optymalnego celu G.



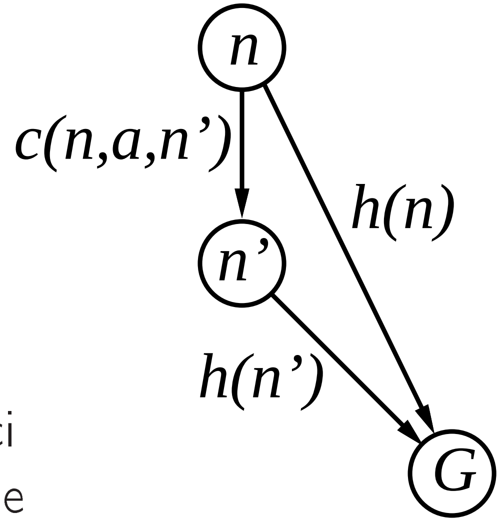
$$
\begin{aligned}
f(G_2) &= g(G_2) && \text{poniewaz } h(G_2) = 0 \\
&> g(G) && \text{poniewaz } G_2 \text{ jest nieoptymalny} \\
&\geq f(n) && \text{poniewaz } g(G) = g(n) + h^*(n) \geq g(n) + h(n) = f(n)
\end{aligned}
$$

$f(G_2) > f(n)$ dla wszystkich n z optymalnej ścieżki, A* wyjmie je przed G_2

Heurystyka spojna

Heurystyka jest *spójna* jeśli dla każdego stanu n i każdej akcji a z tego stanu:

$$h(n) \leq c(n, a, n') + h(n')$$



gdzie $c(n, a, n')$ jest kosztem wykonania akcji a.

Fakt 1: Każda heurysytka spójna jest dopuszczalna (dowód jako ćwiczenie)

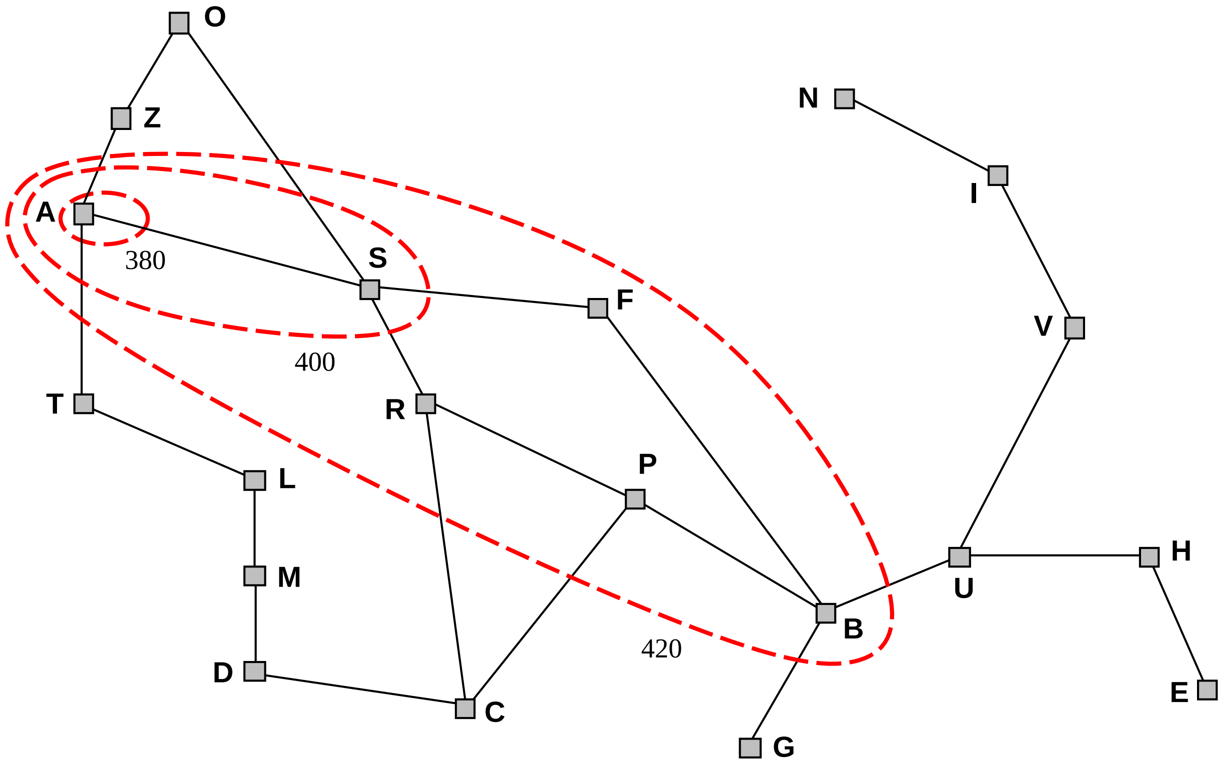
Fakt 2: Jeśli heurystyka h jest spójna to ciąg wartości funkcji $f(n)$ wzdłuż dowolnej ścieżki w drzewie przeszukiwań stanów jest niemalejący

$$\begin{aligned} \text{Dowód}: f(n') &= g(n') + h(n') \\ &= g(n) + c(n, a, n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$

Heurystyka spojna

Lemat: A* eksploruje stany w kolejności rosnących wartości funkcji f

Stopniowo dodaje "f-kontury" stanów (anal. przesz. wszerz dodaje warstwy)
Kontur i ma wszystkie stany $f = f_i$, gdzie $f_i < f_{i+1}$



Heurystyka spojna: optymalnosc A*

Twierdzenie:

Przeszukiwanie A* z eliminacją powtarzających się stanów przy użyciu heurystyki spójnej znajduje zawsze rozwiązanie optymalne

Heurystyka spojna: optymalnosc A*

Twierdzenie:

Przeszukiwanie A* z eliminacją powtarzających się stanów przy użyciu heurystyki spójnej znajduje zawsze rozwiązanie optymalne

Dowód:

$f(G) = g(G)$ dla każdego stanu docelowego $G \Longrightarrow f(G_{opt}) \leq f(G)$

Z lematu stan docelowy z optymalnym kosztem ścieżki G_{opt} będzie wyjęty z kolejki jako pierwszy spośród wszystkich stanów docelowych G

Heurystyka dominujaca

h_1, h_2 - heurystyki dopuszczalne

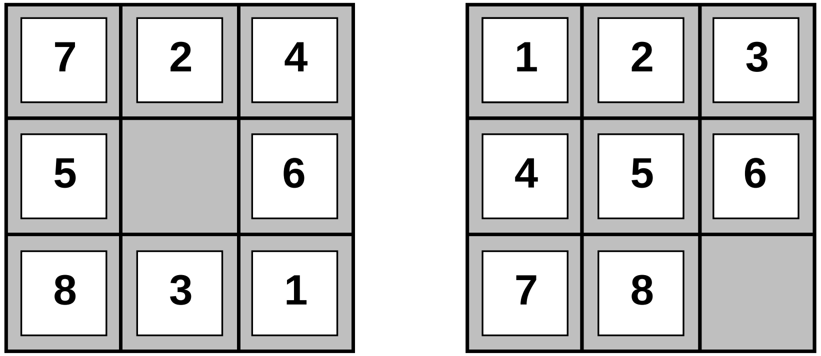
Heurystyka h_2 *dominuje* heurystykę h_1 jeśli
$h_2(n) \geq h_1(n)$ dla wszystkich stanów n

Twierdzenie: Wszystkie węzły odwiedzone przez algorytm A* z heurystyką h_2 będą odwiedzone również przez algorytm A* z heurystyką h_1

Wniosek: jeśli h_2 dominuje h_1, to opłaca się użyć h_2

Heurystyka dominujaca: przyklad

$h_1(n)$ = liczba klocków nie będących w docelowym położeniu
$h_2(n)$ = suma odległości miejskiej (ilości ruchów)
od docelowych miejsc dla poszczególnych klocków



Start State Goal State

h_2 dominuje h_1, bo dla każdego stanu n zachodzi $h_2(n) \geq h_1(n)$, np.

$h_1(n) = 6$
$h_2(n) = 4+0+3+3+1+0+2+1 = 14$

Efektywnosc A*

Średnia liczba generowanych węzłów dla zadań o długości optymalnego rozwiązania równej 14 oraz 24 w problemie 8-elementowej układanki:

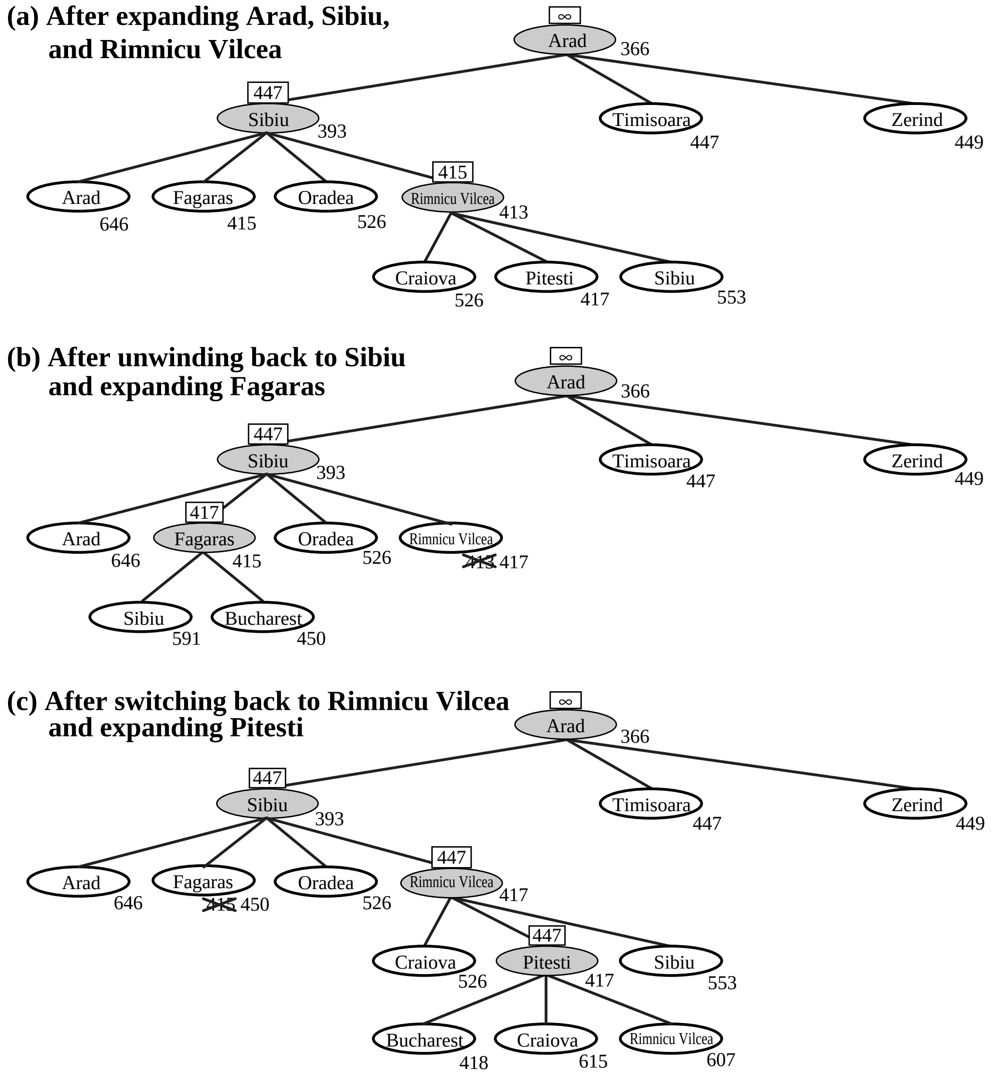
$d = 14$ Iter. pogłęb. = 3,473,941 węzłów
A*(h_1) = 539 węzłów
A*(h_2) = 113 węzłów
$d = 24$ Iter. pogłęb. ≈ 54,000,000,000 węzłów
A*(h_1) = 39,135 węzłów
A*(h_2) = 1,641 węzłów

Rekurencyjne przeszukiwanie pierwszy najlepszy

function RECURSIVE-BEST-FIRST-SEARCH(*problem*) **returns** soln/fail
  return RBFS(*problem*, MAKE-NODE(INITIAL-STATE[*problem*]), ∞)

function RBFS(*problem*, *node*, *f_ limit*) **returns** soln/fail and a new f-cost limit
  if GOAL-TEST[*problem*](*state*) **then return** *node*
  successors $\leftarrow$ EXPAND(*node*, *problem*)
  if *successors* is empty **then return** *failure*, ∞
  for each s **in** *successors* **do**
    $f[s] \leftarrow \max(g(s)+h(s), f[node])$
  repeat
    best $\leftarrow$ the lowest f-value in *successors*
    if $f[best] > f\_limit$ **then return** *failure*, $f[best]$
    alternative $\leftarrow$ the second-lowest f-value among *successors*
    result, $f[best] \leftarrow$ RBFS(*problem*, *best*, min(*f_ limit*, *alternative*))
    if $result \neq failure$ **then return** *result*

Rekurencyjne przeszukiwanie pierwszy najlepszy



Rekurencyjne przeszukiwanie pierwszy najlepszy

Zupełność?? Tak, jeśli stanów z $f \leq f(G)$ jest skończona ilość ($\Leftrightarrow A^*$ zup.)

Złożoność czasowa?? Trudne...

Złożoność pamięciowa?? $O(bd)$!

Optymalność?? Tak, jeśli heurystyka h jest dopuszczalna ($\Leftrightarrow A^*$ optymalne)

Iteracyjne poprawianie

Przy wielu problemach optymalizacyjnych *ścieżka* jest nieistotna:
stan docelowy sam w sobie jest rozwiązaniem

Przestrzeń stanów = zbiór konfiguracji "pełnych";
problem wymaga znalezienia konfiguracji *optymalnej*
lub spełniającej pewne warunki, np. alokacja zasobów w czasie

Dla tego typu problemów można użyć algorytmu *iteracyjnego poprawiania*:
przechowuje tylko stan "bieżący" i próbuje go poprawić

Fakt: Algorytmy iteracyjnego poprawiania wykonywane są w stałej pamięci

Przeszukiwanie lokalne

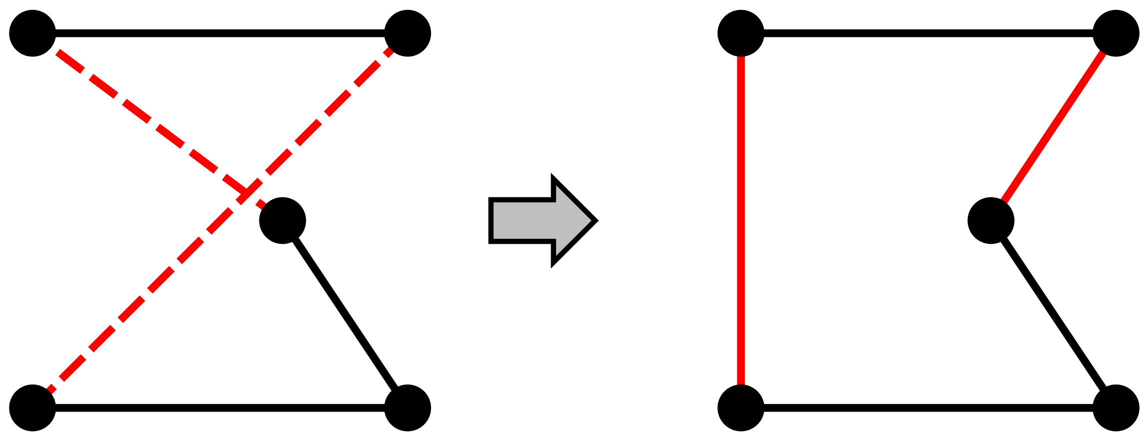
Przeszukiwanie lokalne zastępuje stan bieżący jednym z jego bezpośrednich sąsiadów

```
function LOCAL-SEARCH(problem) returns a state
 inputs: problem, a problem
         VALUE, a function that evaluates a state
 local variables: current, a node

 current ← MAKE-NODE(INITIAL-STATE[problem])
 best ← current
 repeat
     current ← any successor of current
     if VALUE[current] > VALUE[best] then best ← current
 until best is optimal, or VALUE[best] is high enough, or enough time has elapsed
 return best
```

Przeszukiwanie lokalne: problem komiwojazera

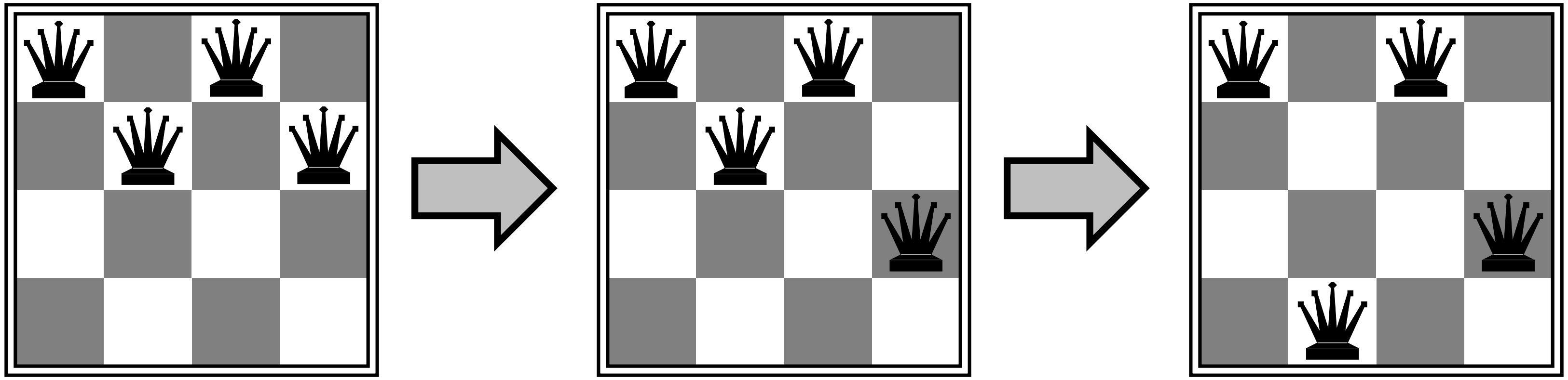
Przeszukiwanie lokalne zaczyna od dowolnego cyklu Hamiltona i wykonuje zamianę krawędzi parami



Przeszukiwanie lokalne: problem n-hetmanow

Problem: Znaleźć rozstawienie n hetmanów na szachownicy $n \times n$
tak, żeby żadne dwa nie biły się nawzajem, tzn.
nie znajdowały się w tym samym rzędzie, kolumnie lub przekątnej

Przeszukiwanie lokalne przesuwa hetmanów tak, żeby redukować liczbę konfliktów



Hill-climbing

Inaczej *przeszukiwanie lokalne zachłanne* lub *wspinanie wzdłuż gradientu*

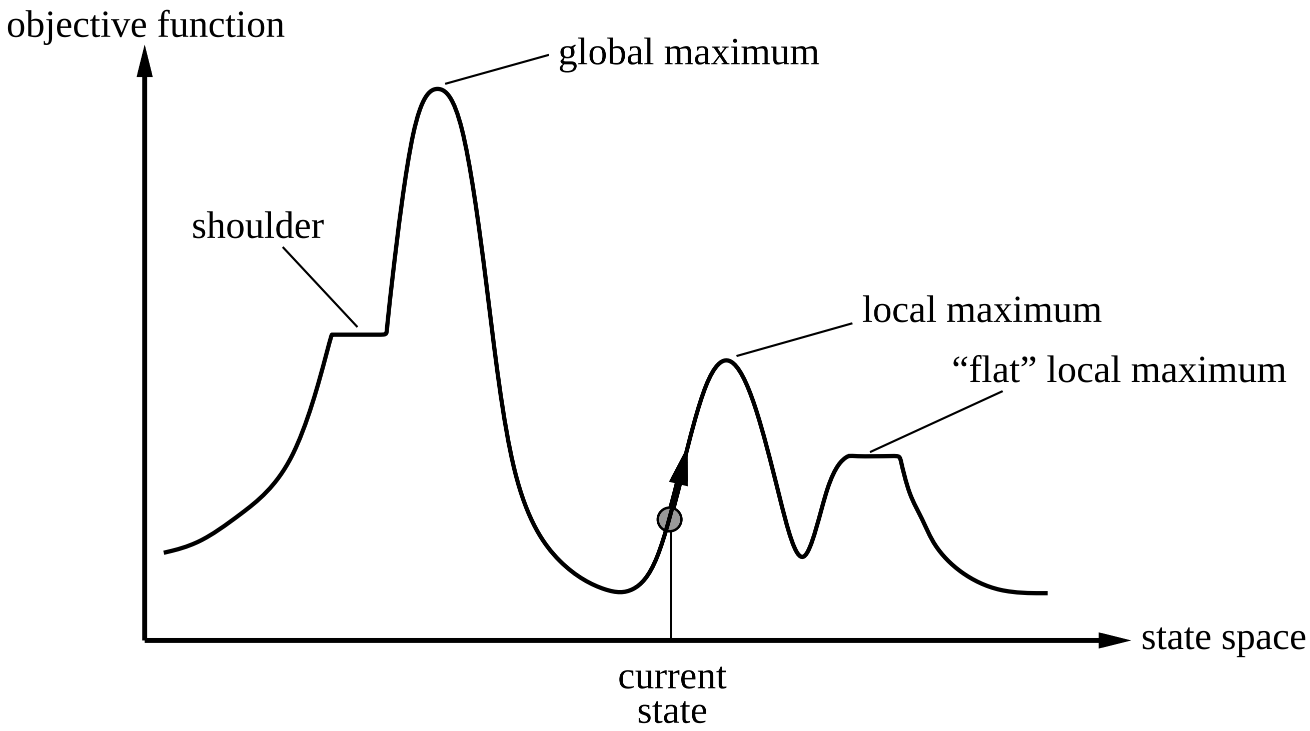
Wybiera zawsze sąsiada z największą wartością funkcji oceny, tzn. wyznaczanego przez *gradient funkcji*

```
function HILL-CLIMBING( problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node

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

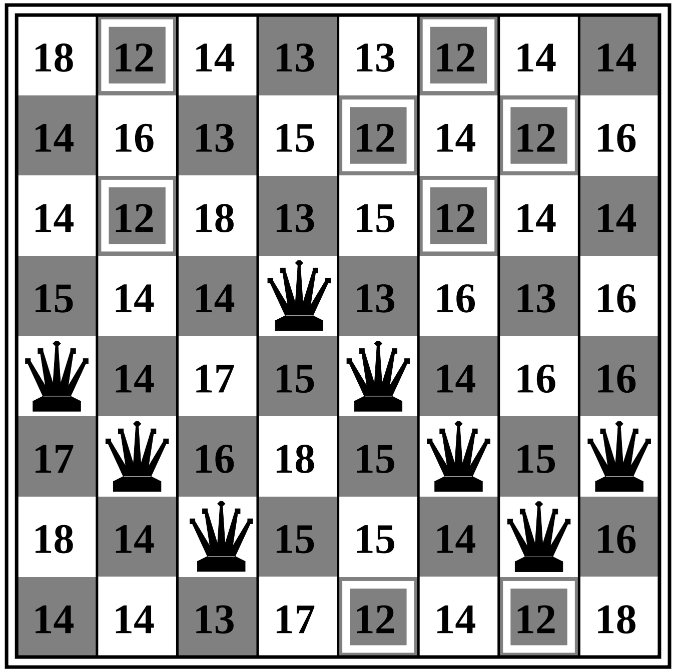
Hill-climbing: lokalne maksima

Algorytm może "utknąć" w lokalnym maksimum funkcji oceny stanów

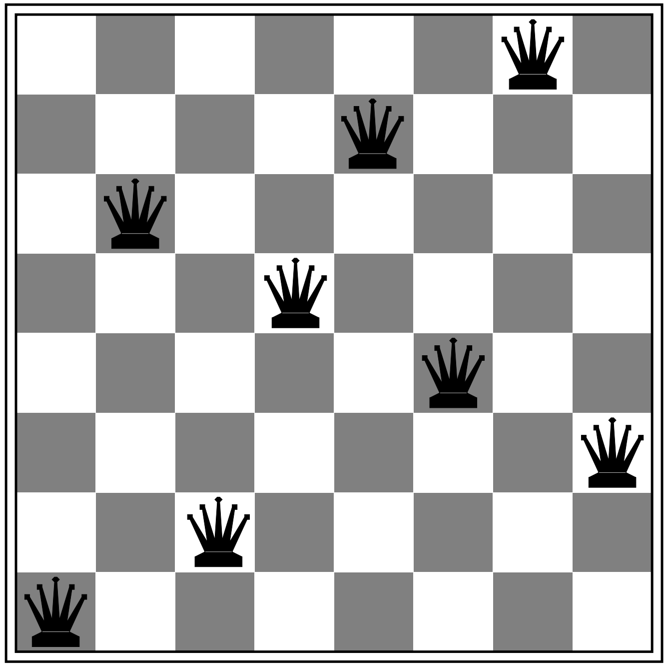


Hill-climbing: problem 8-hetmanow

Stan z liczbą konfliktów = 17,
przedstawia liczbą konfliktów
dla wszystkich sąsiadów osiągalnych
przez przesunięcie hetmana w kolumnie



Lokalne minimum:
stan ma 1 konflikt
i każde przesunięcie w kolumnie
zwiększa liczbę konfliktów



Unikanie lokalnego maksimum

- Start wielokrotny
- Dopuszczenie "złych" posunięć:
  - Przeszukiwanie z tabu
  - Symulowane wyżarzanie
  - Przeszukiwanie o zmiennej głębokości

Start wielokrotny

function MULTISTART-HILL-CLIMBING(*problem*) **returns** a local maximum
  inputs: *problem*, a problem
  local variables: *initial*, an initial node in an iteration
    localmax, the result of a single hill-climbing
    best, a best node

  repeat a number of iterations
    initial ← a random node
    localmax ← HILL-CLIMBING(*problem*, *initial*)
    if VALUE[localmax] > VALUE[best] **then** *best* ← *localmax*
  return *best*

Zaleta: Zwiększa szansę na znalezienie lokalnego maksimum bliskiego optymalnemu rozwiązaniu

Cena: Wielokrotnie większy koszt czasowy

Przeszukiwanie z tabu

Pomysł: wybiera optymalny ruch przestrzegając zakazu powrotu do ostatnio odwiedzonych stanów, może natomiast wykonać ruch do "gorszego" stanu

function TABU-SEARCH(*problem*, *k*) **returns** a solution state
  local variables: *current*, *next*, nodes
    best, a node with the best value
    tabu, a set of forbidden states

  $current \leftarrow$ MAKE-NODE(INITIAL-STATE[*problem*])
  $best \leftarrow current$
  $tabu \leftarrow \{current\}$
  repeat
    $next \leftarrow$ a highest-valued successor of $current \notin tabu$
    if VALUE[*next*] $>$ VALUE[*best*] **then** $best \leftarrow next$
    replace the k-th oldest state $\in tabu$ with *next*
    $current \leftarrow next$
  until the same state and tabu are reached twice or time has elapsed
  return *best*

Symulowane wyzarzanie

Pomysł: dopuszcza "złe" posunięcia,
ale stopniowo maleje ich częstość wraz z upływem czasu

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to "temperature"
  local variables: *current*, a node
    next, a node
    T, a "temperature" controlling prob. of downward steps

  current ← MAKE-NODE(INITIAL-STATE[*problem*])
  for $t \leftarrow 1$ **to** ∞ **do**
    $T \leftarrow$ *schedule*[t]
    if $T = 0$ **then return** *current*
    next ← a randomly selected successor of *current*
    $\Delta E \leftarrow$ VALUE[*next*] – VALUE[*current*]
    if $\Delta E > 0$ **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{\Delta E/T}$

Symulowane wyzarzanie: funkcja temperatury

Wybór temperatury początkowej:

Na początku powinna umożliwiać akceptowanie wszystkich posunięć:

$$e^{\Delta E/T_0} \approx 1$$

Wybór funkcji redukcji temperatury:

Redukcja co d kroków ($d \approx$ stopień rozgałęzienia przestrzeni)

— czynnikiem geometrycznym $T := T * r$, $r \in [0.8; 0.99]$

— $T_k = \frac{T_0}{\log_2(k+2)}$, wartość po k-tej redukcji

Symulowane wyzarzanie: wlasnosci

Dla stałej wartości "temperatury" T, prawdopodobieństwo osiągnięcia stanów zbiega do rozkładu Boltzmana

$$p(x) = \alpha e^{\frac{E(x)}{kT}}$$

jeśli T maleje odpowiednio wolno $\Longrightarrow$ najlepszy stan będzie zawsze osiągnięty

Opracowanie: Metropolis i inni, 1953, do modelowania procesów fizycznych

Szeroko stosowane m. in. w projektowaniu układów o dużym stopniu scalenia, w planowaniu rozkładu lotów pasażerskich

Przeszukiwanie o zmiennej glebokosci

Pomysł: szukanie lepszego stanu poprzez wykonanie na raz
kilku kroków zamiast jednego

Algorytm Kernigana-Lina:

Założenie:
Każdy stan jest opisany przez listę ustalonych dla problemu lokalnych specyfikacji $(p_1, \ldots, p_n)$, lokalny krok polega na zmianie jednej specyfikacji p_i lub kilku z nich (założenie zazwyczaj prawdziwe).

Idea algorytmu:
wykonanie kilku kolejnych zmian lokalnych specyfikacji i wybranie najlepszego stanu spośród wygenerowanych

Ograniczenie:
W trakcie jednego "dużego" kroku i-ta specyfikacja może być zmieniona conajwyżej raz

Algorytm Kernigana-Lina

function VARIABLE-DEPTH-SEARCH(*problem*) **returns** a solution state
  local variables: *current*, local specifications of the current state
    new, states generated in the current step
    used, specifications used in the current step

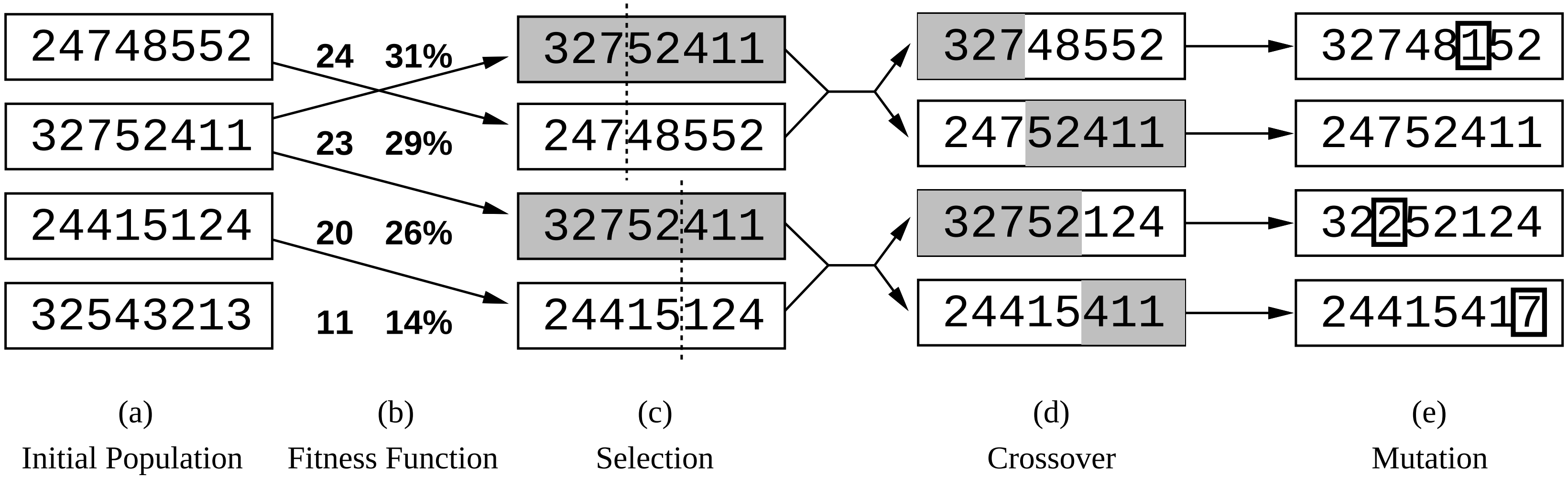
  $current \leftarrow$ MAKE-NODE(INITIAL-STATE[*problem*])
  loop do
    $(p_1, \ldots, p_i, \ldots, p_n) \leftarrow current$
    $new \leftarrow \{\}$
    $used \leftarrow \{\}$
    loop for j **from** 1 **to** n
      $(p_1, .., p'_i, .., p_n) \leftarrow$ highest-valued successor of $(p_1, .., p_i, .., p_n)$ with $i \notin used$
        add $(p_1, .., p'_i, .., p_n)$ **to** *new*
        add i **to** *used*
        $(p_1, .., p_i, .., p_n) \leftarrow (p_1, .., p'_i, .., p_n)$
    $next \leftarrow$ the highest-valued state in *new*
    if VALUE[*next*] > VALUE[*current*] **then** $current \leftarrow next$
    else return *current*

Algorytm genetyczny

function GENETIC-ALGORITHM(*problem*, FITNESS-FN) **returns** an individual
  inputs: *population*, a set of individuals
    FITNESS-FN, a function that measures the fitness of an individual
  new_population ← empty set
  loop for *i* **from** 1 **to** SIZE(*population*)
    $x \leftarrow$ RANDOM-SELECTION(*population*, FITNESS-FN)
    $y \leftarrow$ RANDOM-SELECTION(*population*, FITNESS-FN)
    child ← REPRODUCE(x, y)
    if (small random probability) **then** *child* ← MUTATE(*child*)
    add *child* to *new_population*
  population ← *new_population*
  until some individual is fit enough, or enough time has elapsed
  return the best individual in *population*, according to FITNESS-FN

Funkcja REPRODUCE zwraca nowy stan będący losowym *skrzyżowaniem (kombinacją)* dwóch stanów-rodziców.
Funkcja MUTATE zmienia losowo *pojedynczą informację* w stanie.

Algorytm genetyczny: kombinowanie i mutacja



function REPRODUCE(x, y) **returns** an individual
  inputs: x, y, parent individuals
  $n \leftarrow$ LENGTH(x)
  $c \leftarrow$ random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c+1, n))

Algorytm genetyczny: problem 8 hetmanow

Skrzyżowanie dwóch stanów w problemie 8 hetmanów

niezacienione kolumny są tracone przy operacji krzyżowania

zacienione kolumny pozostają