Strategie przeszukiwania ślepe i heurystyczne

dr inż. Joanna Kołodziejczyk

jkolodziejczyk@wi.ps.pl

Zakład Sztucznej Inteligencji ISZiMM

Przeszukiwanie — charakterystyka zadań problem-solving

- Nie jest znana a priori sekwencja kroków (stanów) prowadząca do rozwiązania.
- Kroki owe muszą być wyznaczone przez analizę bardzo licznych alternatyw.
- Szukanie to proces polegający na testowaniu różnych kierunków poszukiwań i odrzucaniu błędnych.
- Zaletą jest łatwość formułowania problemu: zbioru stanów, zbioru operatorów, stan początkowy i cel.

Plan wykładu

Strategie ślepe (*ang. blind search*) inaczej bez wiedzy (*ang. uninformed search*).

- Przeszukiwanie wszerz — BFS (Breadth-first search).
- Przeszukiwanie o stałym koszcie — UCS (Uniform-cost search).
- Przeszukiwanie w głąb — DFS (Depth-first search).
- Przeszukiwanie w głąb z ograniczeniem (Depth-limited search).
- Iteracyjne zagłębianie (Iterative deepening search).

Strategie heurystyczne (szukanie z wiedzą (*ang. informed search*))

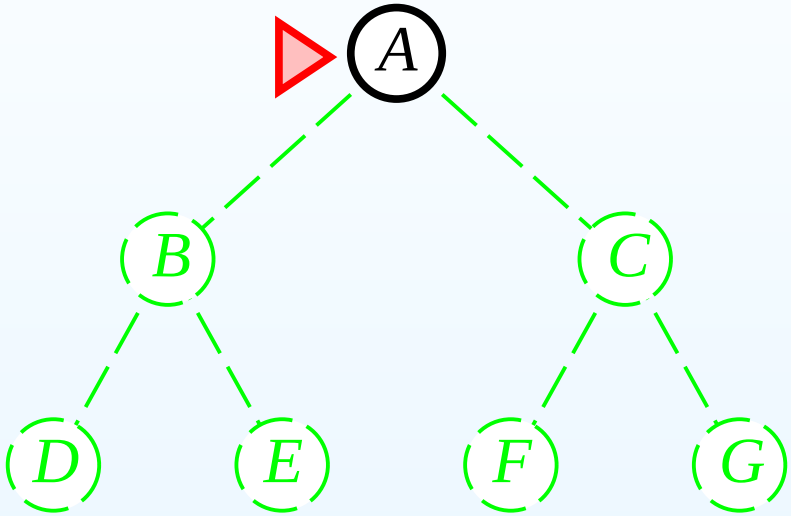
- Najpierw najlepszy - Best-first search
- A^* search
- Heurystyki

Przeszukiwanie wszerz — BFS

Breadth First Search (BFS)

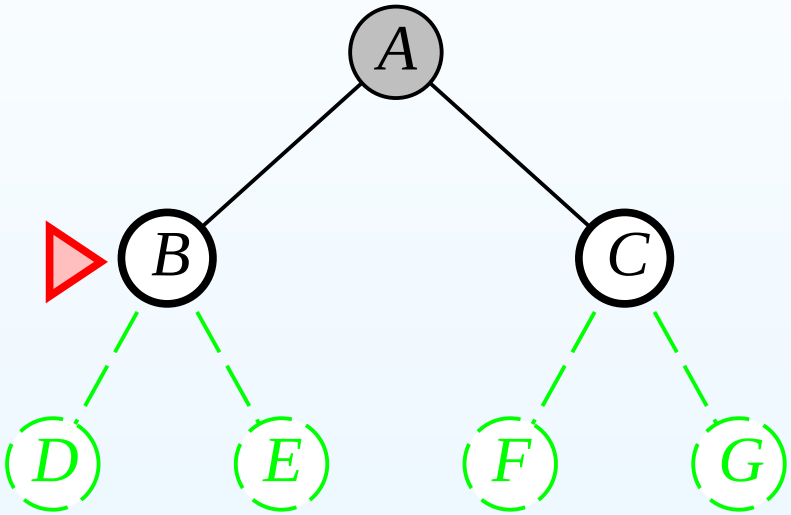
- Pierwszy rozwijany jest węzeł w korzeniu drzewa — jest to stan początkowy wyznaczony ze specyfikacji rozwiązywanego zadania.
- Następnie rozwija wszystkich potomków wyznaczonych w kroku poprzednim, a potem ich potomków itd.
- Uogólniając: BFS rozwija wszystkie węzły drzewa na poziomie d.
- BFS może być zaimplementowany jako kolejka FIFO tj. nowy węzeł potomny z rozwinięcia węzła bieżącego jest dopisywany na końcu kolejki węzłów czekających do rozwinięcia.

BFS — działanie



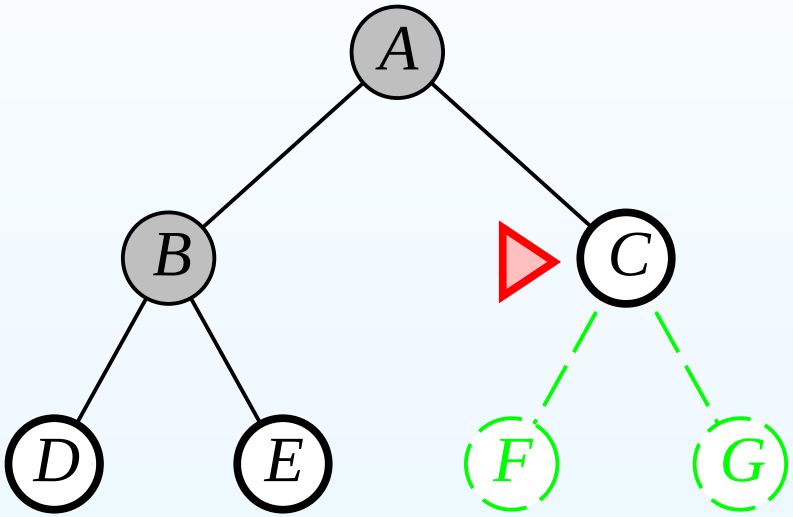
$$LISTA = \{A\}$$

BFS — działanie



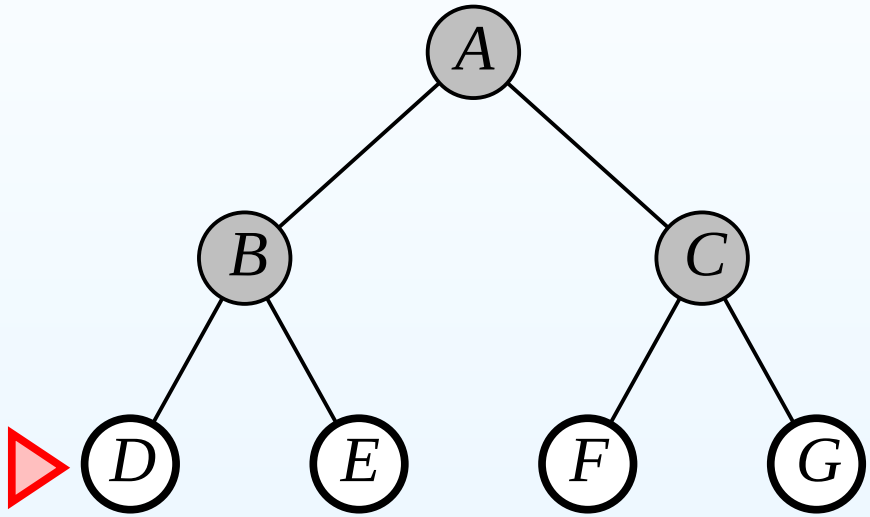
$$LISTA = \{B, C\}$$

BFS — działanie



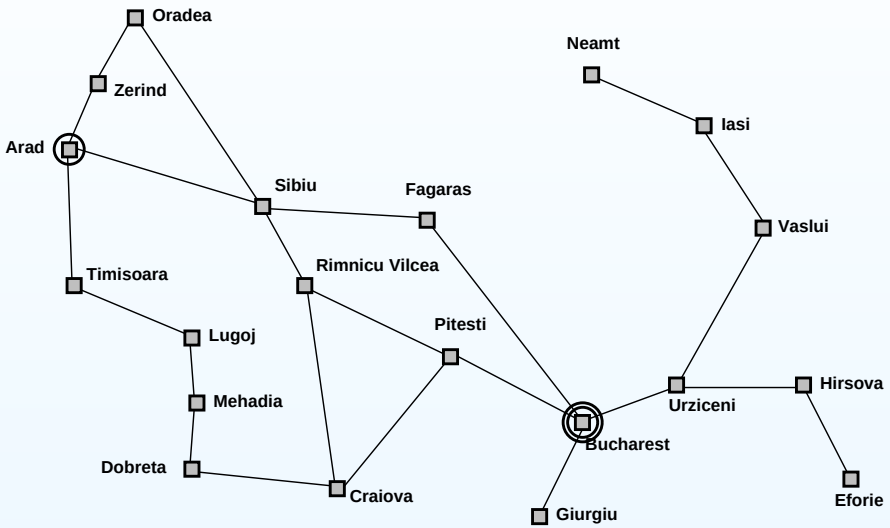
$$LISTA = \{C, D, E\}$$

BFS — działanie



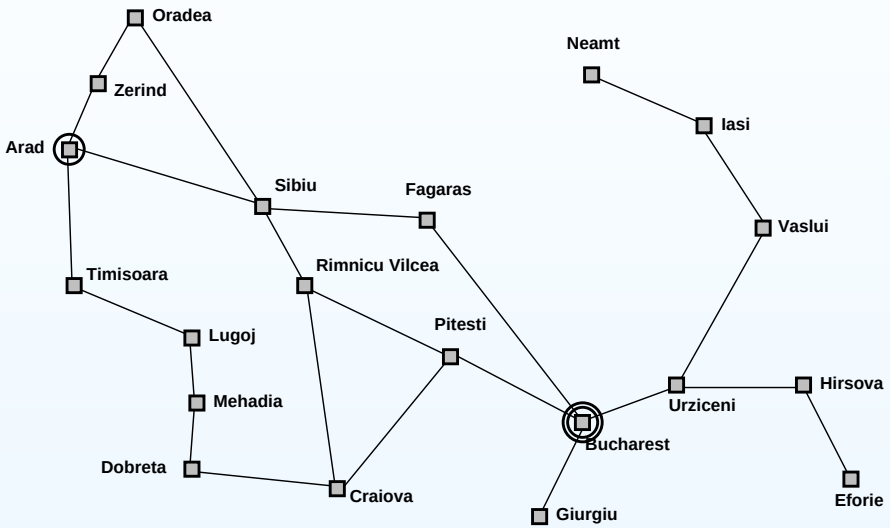
$$LISTA = \{D, E, F, G\}$$

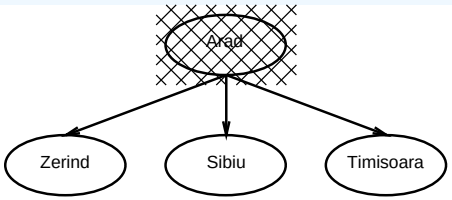
Przeszukiwanie wszerz — przykład



$$LISTA = \{Arad\}$$

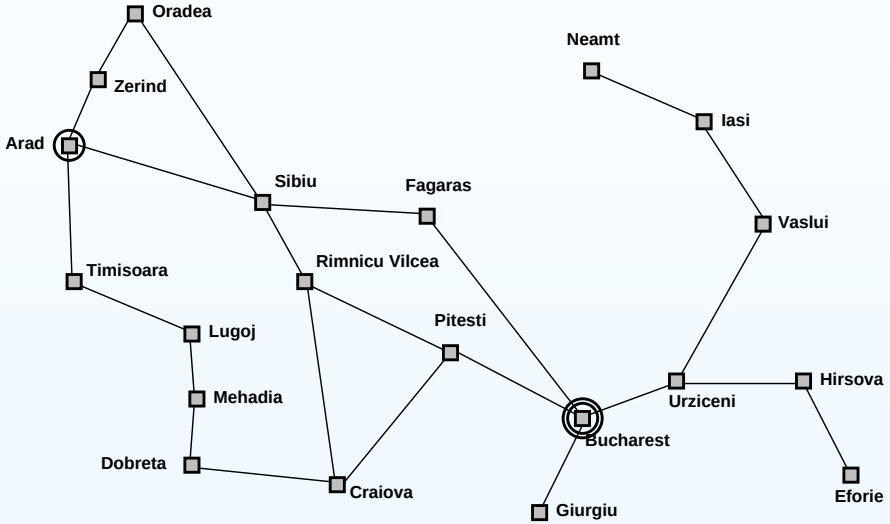
Przeszukiwanie wszerz — przykład

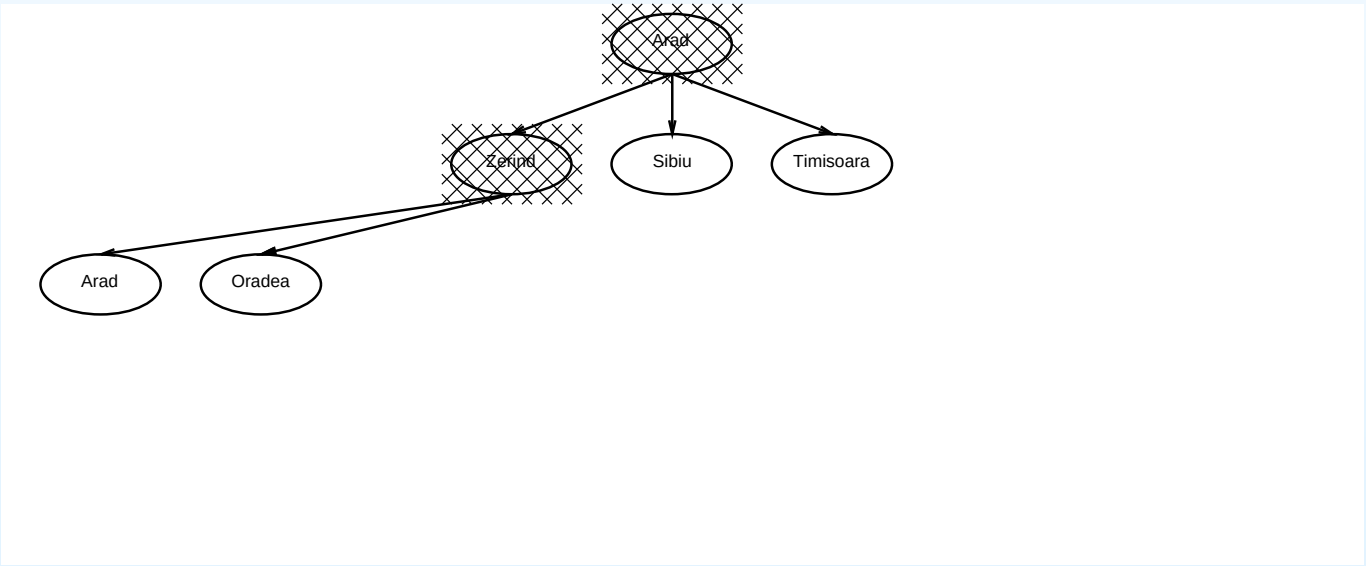




$$LISTA = \{Zerind, Sibiu, Tisimora\}$$

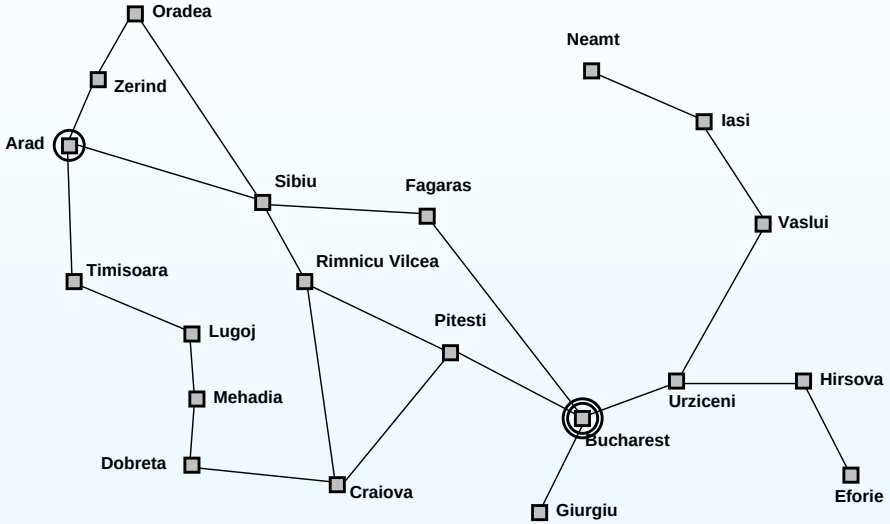
Przeszukiwanie wszerz — przykład

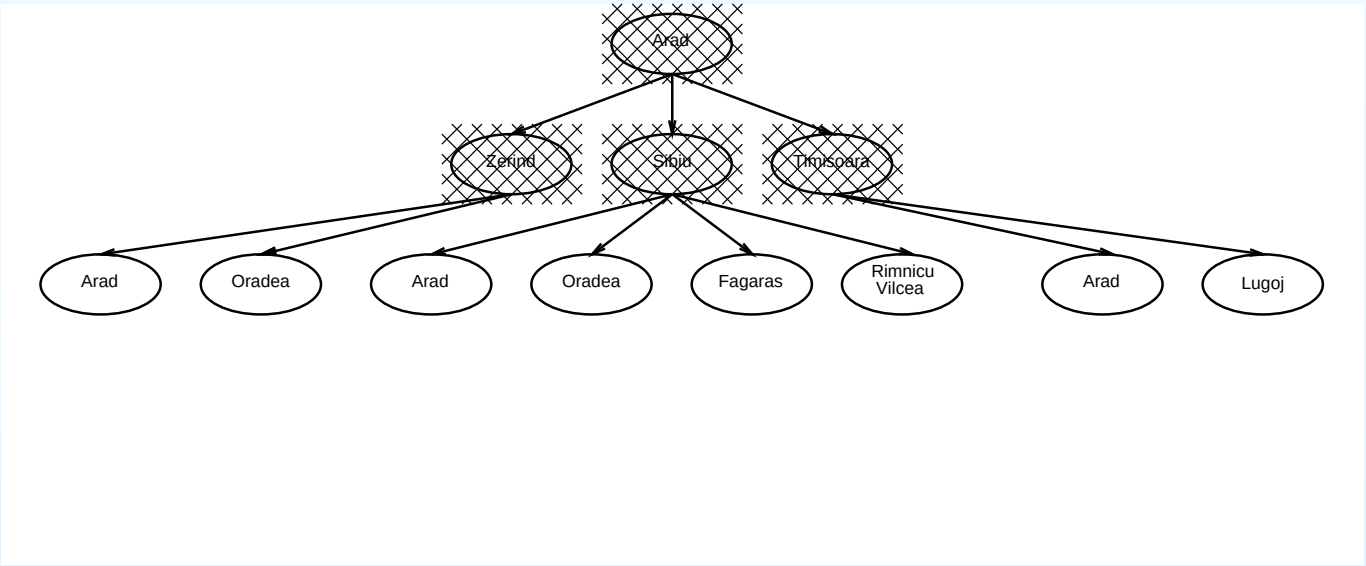




$$LISTA = \{Sibiu, Tisimora, Arad, Oradea\}$$

Przeszukiwanie wszerz — przykład





$$LISTA = \{Arad, Oradea, Arad, Oradea, Fagaras, RimnicuVilcea, Arad, Lugoj\}$$

Złożoność algorytmu — notacja $O()$ — powtórzenie

```
1  function Summation(sequence) return a number
2     sum = 0
3     for i =1 to LENGTH(sequence)
4        sum = sum + sequence[i]
5     end
6     return sum
```

Jeżeli n długość ciągu, to liczba wykonanych linii kodu (wszystkich operacji) dla rozpatrywanego przykładu będzie $T(n) = 2n + 2$. Liczbę kroków przedstawia się jako funkcję od n -liczby wejść.

Ze względu na trudność określenia dokładnej wartości $T(n)$ stosuje się przybliżenie. Np. dla rozpatrywanego przykładu złożoności algorytmiczna jest $O(n)$, co oznacza, że czas obliczeń wzrasta liniowo z n.

Uogólniając: $T(n)$ jest $O(f(n))$, jeżeli $T(n) < k \cdot f(n)$ dla pewnego k, dla wszystkich $n > n_0$

Notacja $O()$ pozwala na analizę asymptotyczną. Np. Dla $n \to \infty$ algorytm $O(n)$ jest lepszy od algorytmu o złożoności $O(n^2)$.

Klasy złożoności

- Analiza algorytmów i notacja $O()$ pozwala na określenie efektywności danego algorytmu.
- Analiza złożoności zajmuje się raczej problemem niż algorytmem.
- Zgrubnie dzieli się problemy na rozwiązywalne w czasie wielomianowym i takie, które nie są w nim rozwiązywalne niezależnie od użytego algorytmu.
- Niektóre klasy złożoności problemów:
  - P (polynomial problems) - tzw. problemy łatwe, których złożoność to $O(\log n)$ lub $O(n)$ $O(n^c)$. Np. Czy $O(n^{1000})$ jest "łatwy"?
  - NP (nondeterministic polynomial problems) - jeżeli istnieje algorytm zgadujący rozwiązanie i potrafiący je zweryfikować jako poprawne w czasie wielomianowym.
  - NP-complete „most extreme NP problems”- duże zainteresowanie tą klasą ze względu na mnogość problemów do niej należących. Jak np. problem spełnialności w logice zdań.

Ocena algorytmu BFS

1. Zupełność (completeness): czy zawsze znajdzie rozwiązanie jeżeli ono istnieje? Tak, jeżeli b jest skończone
2. Złożoność czasowa (time complexity):
3. Obszar zajmowanej pamięci (space complexity):
4. Optymalność (optimality):

Ocena algorytmu BFS

1. Zupełność (completeness): Tak, jeżeli b jest skończone
2. Złożoność czasowa (time complexity): liczba węzłów generowana/rozwijana.

   Zakłada się, że węzeł w korzeniu może mieć b potomków. Każdy jego potomek też może mieć b potomków. Rozwiązanie jest na poziomie drzewa d. W najgorszym przypadku trzeba rozwinąć wszystkie węzły z poziomu d oprócz ostatniego węzła.
   $1 + b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = O(b^{d+1})$
3. Obszar zajmowanej pamięci (space complexity):
4. Optymalność (optimality):

Ocena algorytmu BFS

1. Zupełność (completeness): Tak, jeżeli b jest skończone
2. Złożoność czasowa (time complexity):
   $1 + b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = O(b^{d+1})$
3. Obszar zajmowanej pamięci (space complexity):
   maksymalna liczba węzłów w pamięci. $O(b^{d+1})$
   Przechowuje każdy węzeł w pamięci, bo albo węzły są liśćmi i czekają na rozwinięcie, lub są przodkami, których trzeba pamiętać.
4. Optymalność (optimality):

Ocena algorytmu BFS

1. Zupełność (completeness): Tak, jeżeli b jest skończone
2. Złożoność czasowa (time complexity):
   $1 + b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = O(b^{d+1})$
3. Obszar zajmowanej pamięci (space complexity): $O(b^{d+1})$
   Przechowuje każdy węzeł w pamięci, bo albo węzły są liśćmi i czekają na rozwinięcie, lub są przodkami, których trzeba pamiętać.
4. Optymalność (optimality): czy zawsze znajdzie najmniej kosztowne rozwiązanie? Tak, jeżeli w każdym kroku koszt ścieżki jest stały. Nie musi być optymalny (najbliższe rozwiązanie nie musi być optimum).

Wymagania algorytmu wszerz

Wnioski

1. Wymagania pamięciowe algorytmu są dużo większe niż czasowe.
2. Problemy o złożoności wykładniczej nie mogą być rozwiązane metodą BFS. Żadna z metod ślepych nie jest efektywna dla dużego n.

Depth	Nodes	Time	Memory
2	1100	0.11 seconds	1 MB
4	111100	11 seconds	106 MB
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabyte
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3523 years	1 exabyte

Wymagania czasowe i pamięciowe dla BFS. Założono, że $b = 10$, 10000 węzłów/sek, 1000 bytów/węzeł.

Szukanie przy stałych kosztach

BFS nie może znaleźć optimum, stąd dla zadań o zmiennym koszcie ścieżki stosuje się algorytm Uniform-cost Search.

Uniform-cost Search - Szukanie przy stałych kosztach: rozwija węzły z najmniejszym kosztem ścieżki. Brzeg drzewa stanowi kolejka uporządkowana wg kosztu ścieżki.

U-CS jest tym samym co BFS, gdy koszt wszystkich kroków jest taki sam.

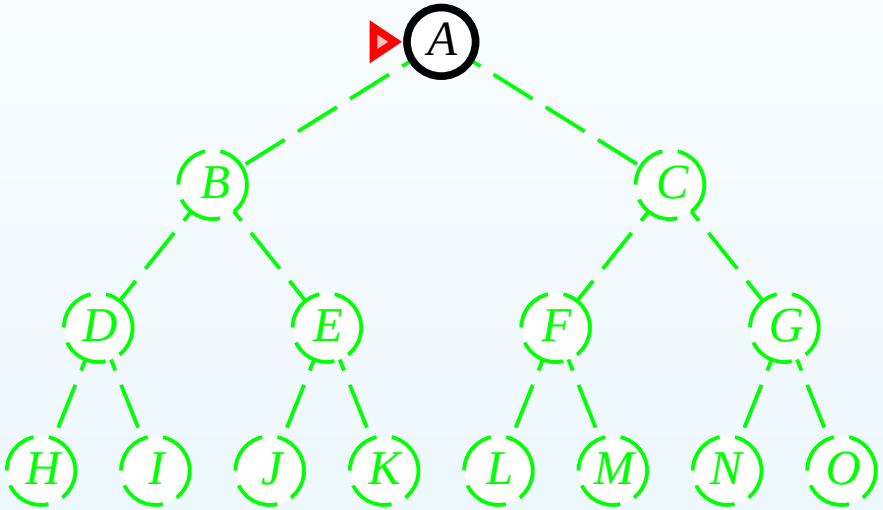
1. Zupełność (completeness): Tak, jeżeli koszt $> \epsilon$ mała dodatnia liczba. Dla ścieżek o zerowym koszcie wpadnie w pętle.
2. Złożoność czasowa (time complexity): Zakłada się, że $C*$ — koszt optymalnego rozwiązania. Każda akcja kosztuje co najmniej ϵ. Najgorszy przypadek: $O(b^{1+\frac{C*}{\epsilon}})$, co może być dużo większe niż b^d.
3. Obszar zajmowanej pamięci (space complexity): Taka sama jak złożoność czasowa.
4. Optymalność (optimality): Węzły rozwijane w kolejności rosnących kosztów ścieżek. Jest optymalny tylko wtedy, gdy zapewniony jest warunek zupełności.

Szukanie w głąb — DFS

Depth-first search: zawsze rozwija najgłębsze jeszcze nierozwinięte węzły.

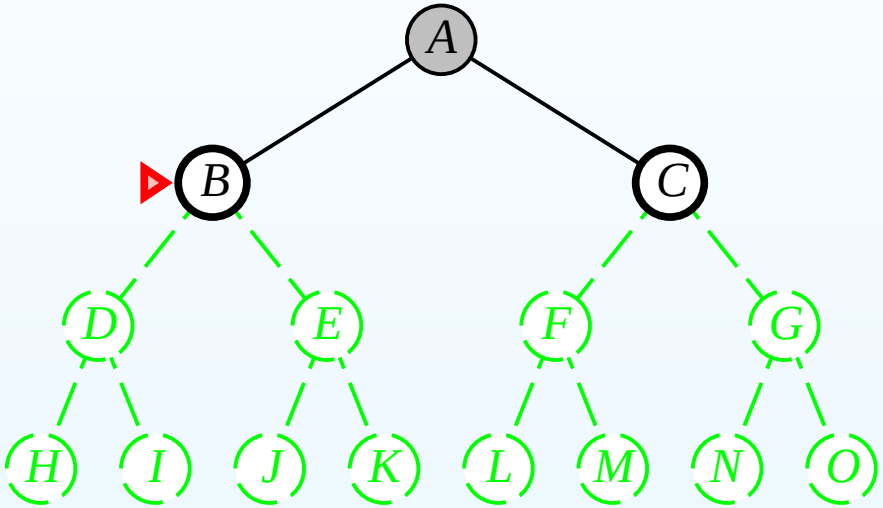
- Zaczyna od korzenia w drzewie.
- Rozwija jedną gałąź aż do osiągnięcia maksymalnej głębokości w drzewie.
- Jeżeli liść nie jest rozwiązaniem, to wraca się do najbliższego, płytszego, jeszcze nie rozwiniętego węzła i rozwija się go.
- Implementacją tej strategii może być kolejka LIFO (stos).
- Procedura implementacyjna może być wykonana jako funkcja rekurencyjna.

DFS — działanie



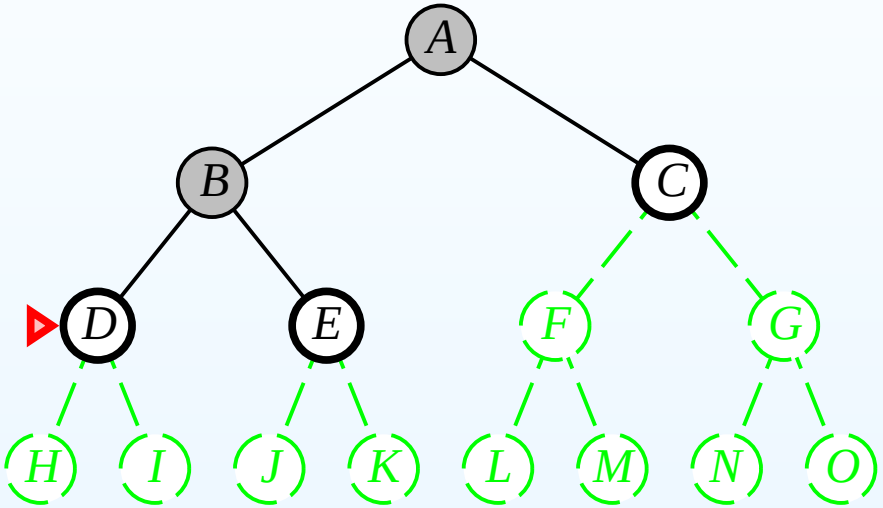
$$LISTA = \{A\}$$

DFS — działanie



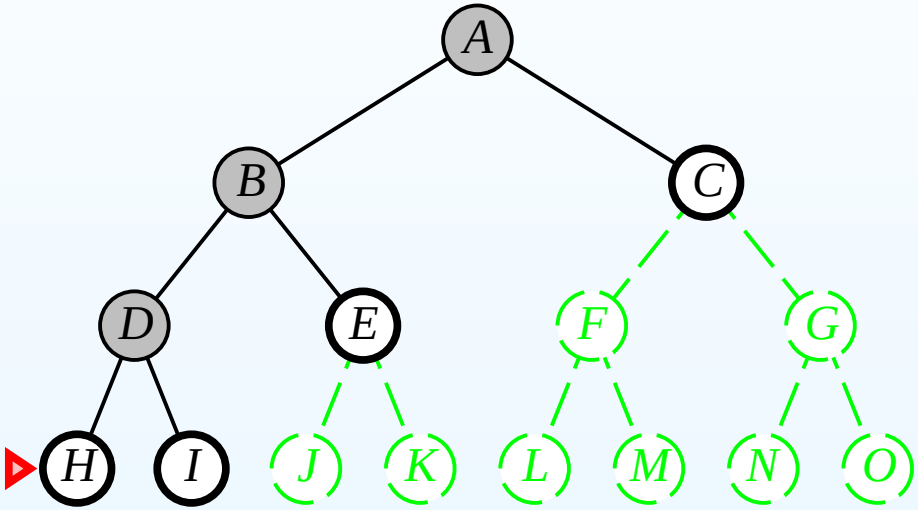
$$LISTA = \{B, C\}$$

DFS — działanie



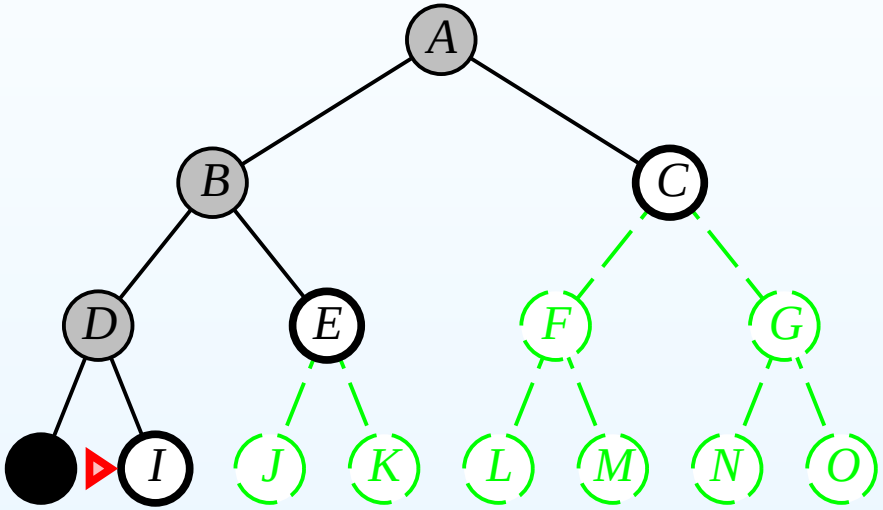
$$LISTA = \{D, E, C\}$$

DFS — działanie



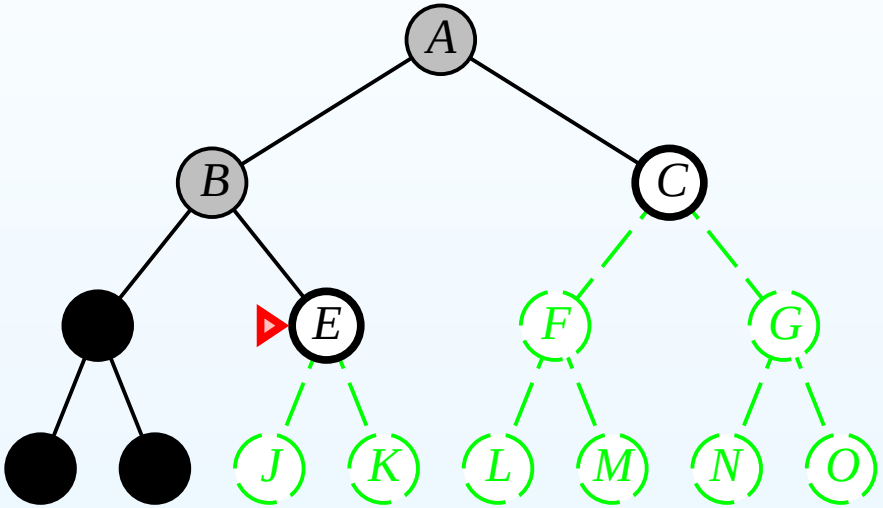
$$LISTA = \{H, I, E, C\}$$

DFS — działanie



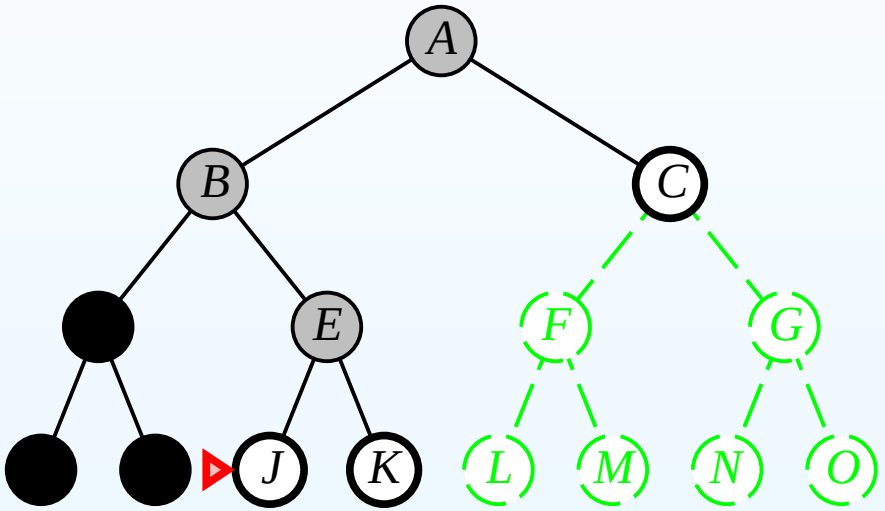
$$LISTA = \{I, E, C\}$$

DFS — działanie



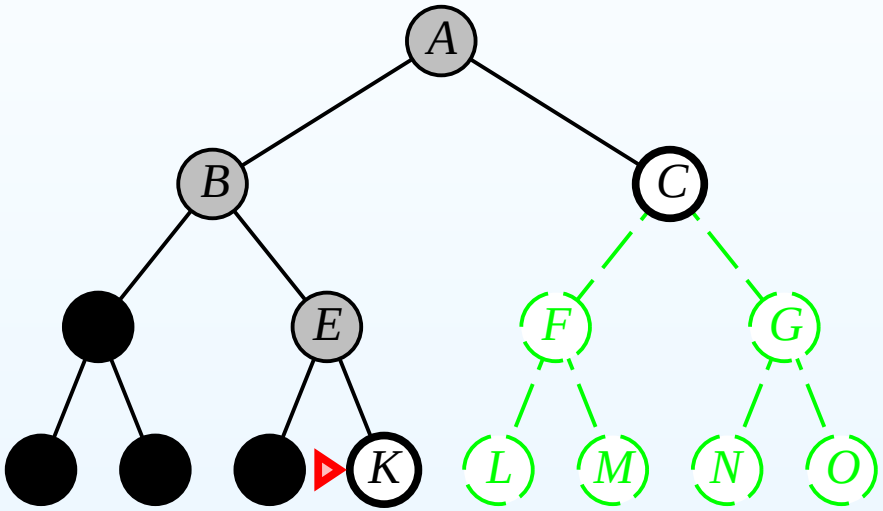
$$LISTA = \{E, C\}$$

DFS — działanie



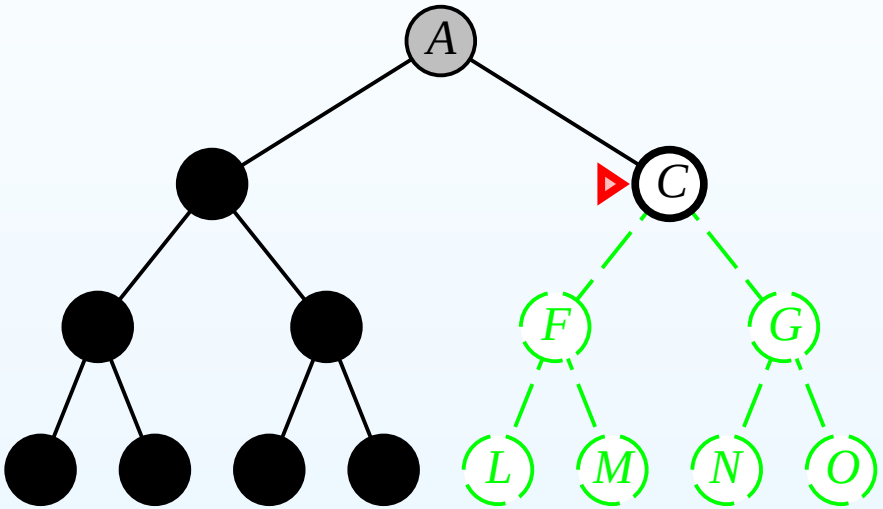
$$LISTA = \{J, K, C\}$$

DFS — działanie



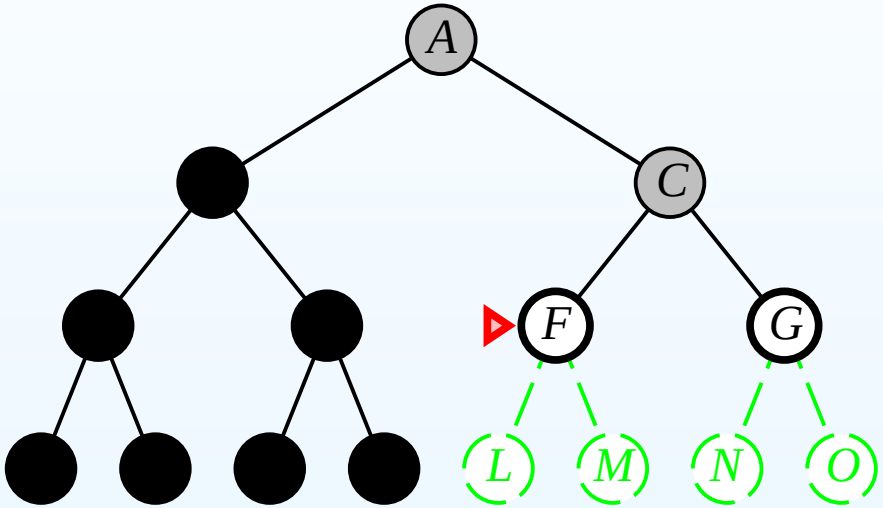
$$LISTA = \{K, C\}$$

DFS — działanie



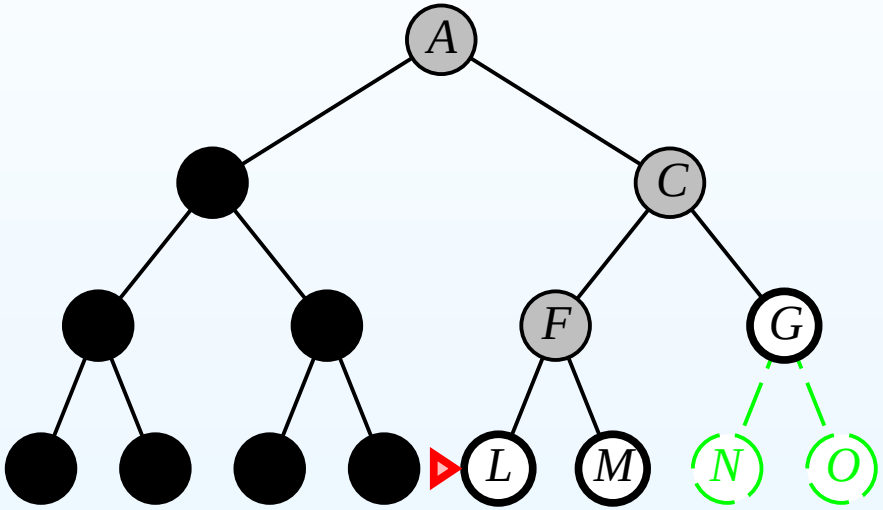
$$LISTA = \{C\}$$

DFS — działanie



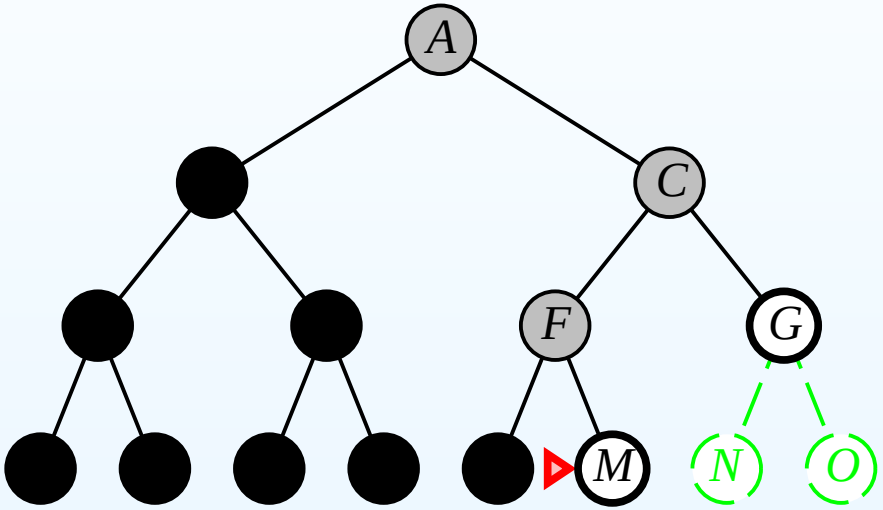
$$LISTA = \{F, G\}$$

DFS — działanie



$$LISTA = \{L, M, G\}$$

DFS — działanie



$$LISTA = \{M, G\}$$

Ocena algorytmu DFS

1. Zupełność (completeness): czy zawsze znajdzie rozwiązanie jeżeli ono istnieje?
   Nie, chyba że przeszukiwana przestrzeń jest skończona (nie pojawiają się pętle).
2. Złożoność czasowa (time complexity):
3. Obszar zajmowanej pamięci (space complexity):
4. Optymalność (optimality):

Ocena algorytmu DFS

1. Zupełność (completeness):
   Nie, chyba że przeszukiwana przestrzeń jest skończona (nie pojawiają się pętle).
2. Złożoność czasowa (time complexity): liczba węzłów generowana/rozwijana.

   m — maksymalna głębokość drzewa. Złożoność jest olbrzymia, gdy m jest dużo większe od d. Natomiast, gdy jest wiele rozwiązań może być szybszy niż BFS.
   Zatem złożoność jest $O(b^m)$
3. Obszar zajmowanej pamięci (space complexity):
4. Optymalność (optimality):

Ocena algorytmu DFS

1. Zupełność (completeness):
   Nie, chyba że przeszukiwana przestrzeń jest skończona (nie pojawiają się pętle).
2. Złożoność czasowa (time complexity):
   Zatem złożoność jest $O(b^m)$
3. Obszar zajmowanej pamięci (space complexity):
   maksymalna liczba węzłów w pamięci.
   Przechowuje pojedynczą ścieżkę w pamięci od korzenia do liścia i braci węzłów na każdym poziomie. Rozwinięte węzły mogą zostać usunięte z pamięci, bo ścieżka do nich nie zawiera rozwiązania.
   Niewielkie wymagania pamięci - złożoność liniowa: $O(bm)$
4. Optymalność (optimality):

Ocena algorytmu DFS

1. Zupełność (completeness):
   Nie, chyba że przeszukiwana przestrzeń jest skończona (nie pojawiają się pętle).
2. Złożoność czasowa (time complexity):
   Zatem złożoność jest $O(b^m)$
3. Obszar zajmowanej pamięci (space complexity):
   Niewielkie wymagania pamięci - złożoność liniowa: $O(bm)$
4. Optymalność (optimality): czy zawsze znajdzie najmniej kosztowne rozwiązanie?
   Nie Z tych samych powodów co zupełność.

Przeszukiwanie w głąb z ograniczoną głębokością

Depth-limited search - jest to wariant DFS z oraniczoną głębokością przeszukiwania

- Zmienna l określa limit przeszukiwania — nowa graniczna głębokość w drzewie.
- Każdy węzeł na poziomie l traktowany jest tak, jakby nie miał potomków.
- Takie podejście rozwiązuje problem nieskończonych ścieżek.
- Problemy:
  - Jeżeli $l < d$ - rozwiązanie nie jest zupełne.
  - Jeżeli $l > d$ - rozwiązanie nie jest optymalne.
- l powinno być wyznaczane na podstawie wiedzy o rozwiązywanym problemie.
- Złożoność czasowa: $O(b^l)$.
- Obszar zajmowanej pamięci: $O(bl)$
- Może być implementowane jako rekurencyjne DFS z parametrem l.

Rekurencyjne Depth-limited search

```
1  function DEPTH-LIMITED-SEARCH(problem,limit)
2              return a solution or failure/cutoff
3     return RECURSIVE-DLS(MAKE-NODE(
4           INITIAL-STATE[problem]),problem,limit)
5
6  function RECURSIVE-DLS(node, problem, limit)
7              return a solution or failure/cutoff
8     cutoff_occurred = false
9
10    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
11    else if DEPTH[node] == limit then return cutoff
12        else for each successor in EXPAND(node, problem) do
13             result = RECURSIVE-DLS(successor, problem, limit)
14             if result == cutoff then cutoff_occurred = true
15             else if result != failure then return result
16       if cutoff_occurred then return cutoff
17       else return failure
```

Iteracyjne zagłębianie

Iterative deepening depth-first search jest modyfikacją DFS z ograniczoną głębokością. Łączy w sobie zalety DFS i BFS.

- Algorytm poszukuje najlepszej wartości ograniczenia poziomu przeszukiwań l.
- Jest to realizowane przez stopniowe zwiększanie l od 0, potem 1 itd. Dopóki nie znajdzie rozwiązania.
- Rozwiązanie zostanie znalezione na poziomie d (najbliższe rozwiązanie).
- Jak BFS jest zupełny i optymalny, gdy koszt ścieżki jest niemalejącą funkcją głębokości drzewa.
- Sprawdza się w zadaniach, w których nie znany jest poziom rozwiązania.

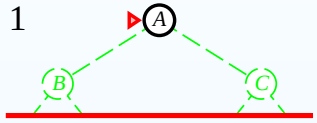
```
1  function ITERATIVE_DEEPENING_SEARCH(problem)
2                    return a solution or failure
3     for depth = 0 to inf do
4        result = DEPTH-LIMITED_SEARCH(problem, depth)
5        if result != cutoff then return result
```

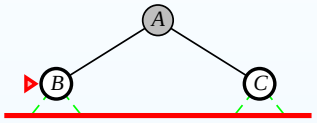
Działanie IDS

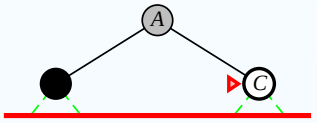
Limit = 0

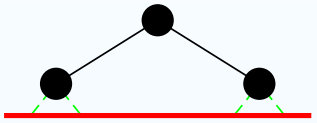
Działanie IDS

Limit = 1



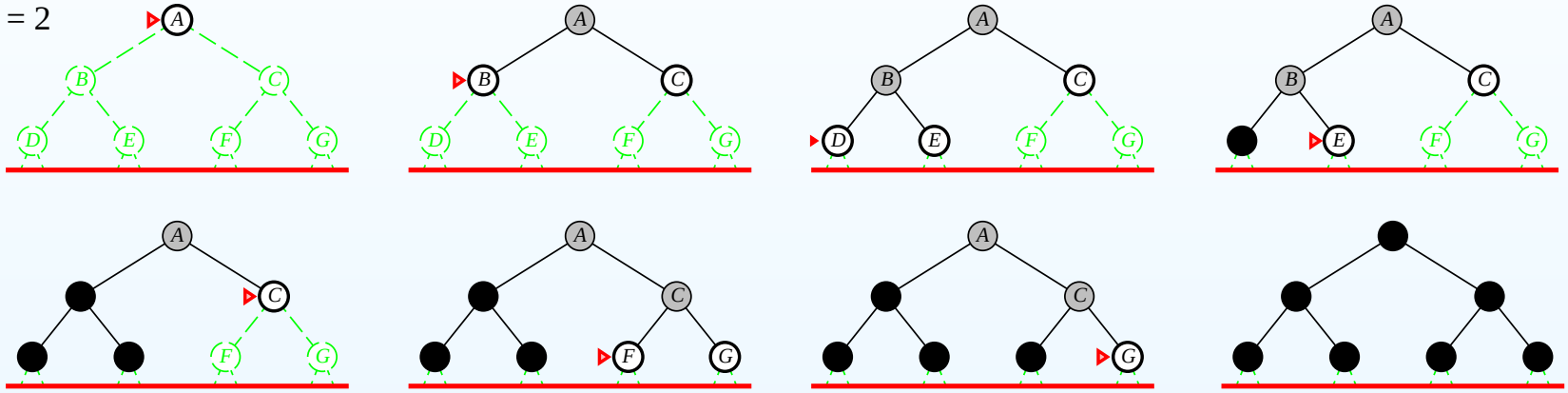




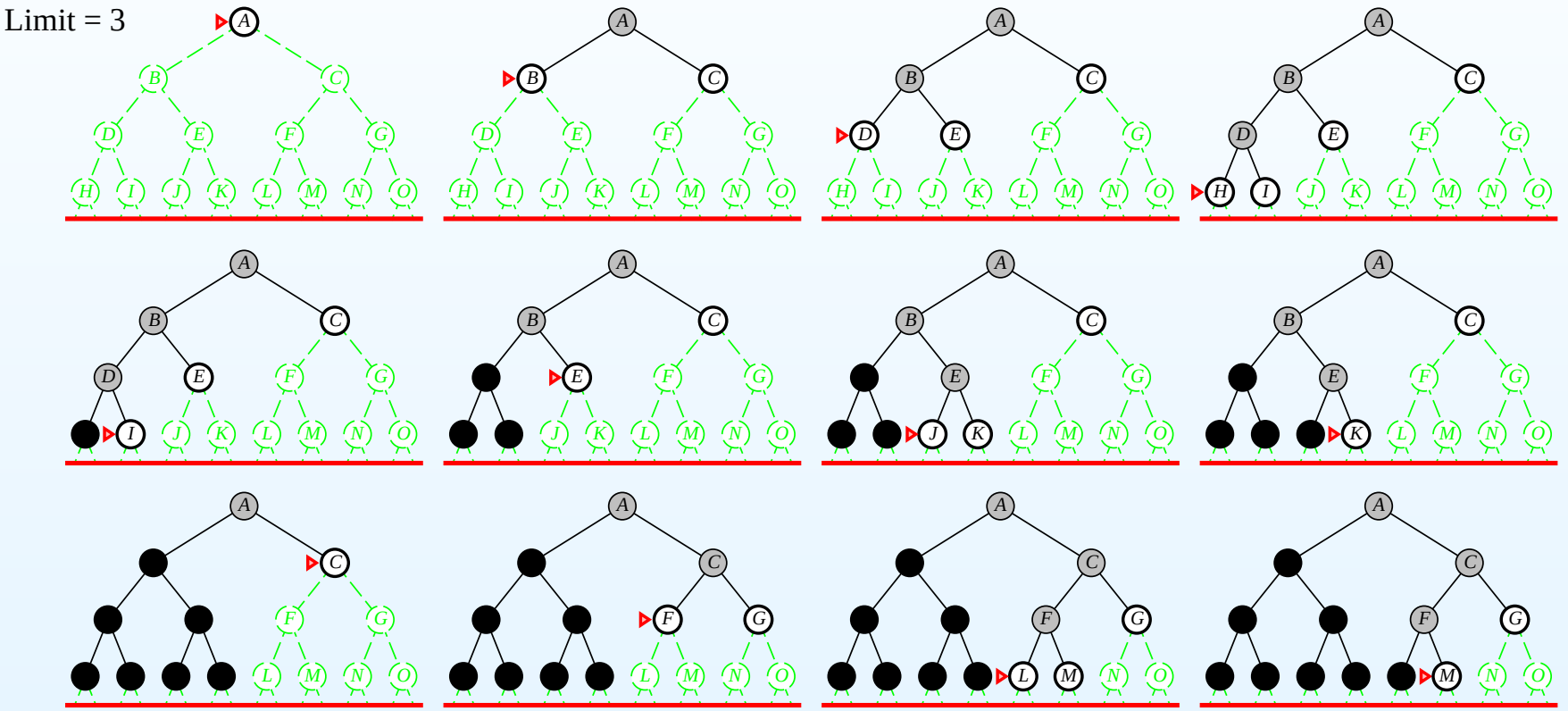


Działanie IDS

Limit = 2



Działanie IDS



Ocena algorytmu IDS

1. Zupełność (completeness): czy zawsze znajdzie rozwiązanie jeżeli ono istnieje?
   Tak, gdyż nie ma nieskończonych ścieżek.
2. Złożoność czasowa (time complexity):
3. Obszar zajmowanej pamięci (space complexity):
4. Optymalność (optimality):

Ocena algorytmu IDS

1. Zupełność (completeness):
   Tak, gdyż nie ma nieskończonych ścieżek.
2. Złożoność czasowa (time complexity): liczba węzłów generowana/rozwijana.
   Algorytm wydaje się być kosztowny ze względu na wielokrotne powtarzanie szukania na niektórych poziomach. Węzły na poziomie d będą przeszukiwane 1 raz, na poziomie $d-1$ dwa razy itd. aż do korzenia, który będzie rozwinięty $d+1$ razy. Stąd całkowita liczba węzłów to:
   $N(IDS) = (d+1)1 + db + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + b^d$
   dla BFS to: $N(BFS) = b + b^2 + \ldots + b^d + (b^{d+1} - b)$
   Porównanie: $b = 10$ i $d = 5$:
   $N(IDS) = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456$, gdy
   $N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$
   Złożoność jest $O(b^d)$
3. Obszar zajmowanej pamięci (space complexity):
4. Optymalność (optimality):

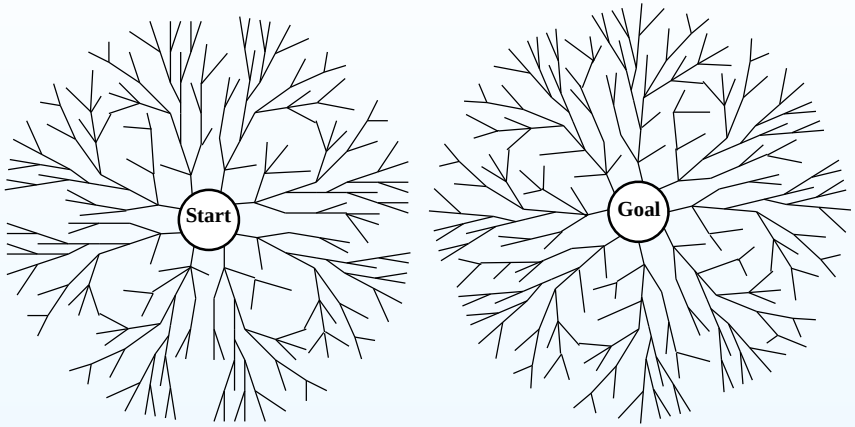
Ocena algorytmu IDS

1. Zupełność (completeness):
   Tak, gdyż nie ma nieskończonych ścieżek.
2. Złożoność czasowa (time complexity):
   Złożoność jest $O(b^d)$
3. Obszar zajmowanej pamięci (space complexity):
   maksymalna liczba węzłów w pamięci.
   Przechowuje w pamięci pojedynczą ścieżkę od korzenia do liścia i wszystkich braci.
   Złożoność algorytmu DFS: $O(bd)$
4. Optymalność (optimality):

Ocena algorytmu IDS

1. Zupełność (completeness):
   Tak, gdyż nie ma nieskończonych ścieżek.
2. Złożoność czasowa (time complexity):
   Złożoność jest $O(b^d)$
3. Obszar zajmowanej pamięci (space complexity):
   Złożoność algorytmu DFS: $O(bd)$
4. Optymalność (optimality): czy zawsze znajdzie najmniej kosztowne rozwiązanie?
   Tak, jeżeli koszt ścieżki jest stały lub jest niemalejącą funkcją głębokości w drzewie.

Przeszukiwanie dwukierunkowe



- Symultanicznie zaczyna się przeszukiwanie do stanu inicjalizującego i docelowego.
- Jeżeli założymy, że rozwiązanie jest na poziomie d, to zostanie ono znalezione w czasie $O(2b^{d/2}) = O(b^{d/2})$, czyli krótszym niż przy przeszukiwaniu w jednym kierunku.
- W każdym kroku trzeba sprawdzać, czy rozwijany węzeł nie znajduję się w liście węzłów do rozwinięcia w drugiej części.
- Zajętość pamięci $O(b^{d/2})$.
- Jest zupełny i optymalny, gdy z dwóch stron szukamy metodą wszerz.
- Tylko dla odwracalnych przestrzeni stanów.

Podsumowanie algorytmów przeszukiwania bez wiedzy (blind search)

Kryterium	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bi-directional
Zupełny?	tak^{a}	tak^{a,b}	nie	nie	tak^{a}	tak^{a,d}
Czas	b^{d+1}	$b^{1+C^*/\epsilon}$	b^{m}	b^{l}	b^{d}	$b^{d/2}$
Pamięć	b^{d+1}	$b^{1+C^*/\epsilon}$	bm	bl	bd	$b^{d/2}$
Optymalny?	tak^{c}	tak	nie	nie	tak^{c}	tak^{c,d}

b — branch factor, d — głębokość, na której jest najbliższe rozwiązanie, m — maksymalna głębokość drzewa, l — limit przeszukiwania

^{a} — zupełny, jeżeli b jest skończone;

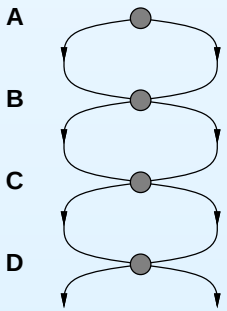
^{b} — zupełne jeżeli koszt ścieżki $\geq \epsilon$, dla $\epsilon > 0$;

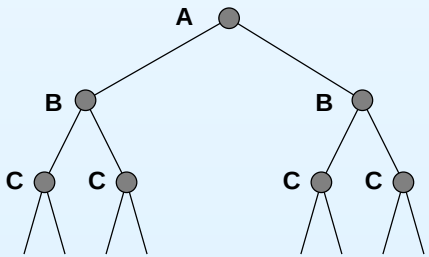
^{c} — optymalny jeżeli koszt kroku stały;

^{d} — jeżeli oba kierunki szukane BFS;

Powtarzające się stany

- W dotychczas omawianych algorytmach zignorowano problem odwiedzania tych samych stanów.
- W niektórych problemach, jak np. wyznacznie drogi z punktu do punktu (routing problem) powtarzanie jest naturalne. Zazwyczaj dotyczy to zadań z przestrzenią odwracalną (odpowiednie dla dwukierunkowego szukania). Dla takich zadań drzewo przeszukiwań jest nieskończone.
- Brak ropoznawania powtarzających się stanów może prowadzić do wzrostu złożoności problemu z liniowego do wykładniczego!
- Jedynym sposobem na uniknięcie tego problemu jest trzymanie rozwiniętych już węzłów w pamięci. Dodanie takiej cechy do algorytmu prowadzi do przeszukiwania przestrzeni stanów na grafie.
- Algorytm, który nie pamięta swojej historii jest skazany na powtórzenia.





Ogólny algorytm przeszukiwania grafu

Zbiór CLOSED powinien być zaimplementowany jako tablica hashująca, co zapewni efektywne sprawdzanie powtarzających się węzłów.

Algorytmy Uniform-cost search i BFS są również optymalne do przeszukiwania grafu.

DFS i wszystkie jego odmiany nie mają już liniowych wymagań pamięciowych, bo należy pamiętać wszystkie węzły.

```
1  function GRAPH-SEARCH(problem,fringe) return a solution or failure
2     closed = an empty set
3     fringe = INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
4
5     loop do
6        if EMPTY(fringe) then return failure
7        node = REMOVE-FIRST(fringe)
8        if GOAL-TEST[problem] applied to STATE[node] succeeds
9                    then return SOLUTION(node)
10       if STATE[node] is not in closed then
11          add STATE[node] to closed
12          fringe = INSERT-ALL(EXPAND(node, problem),fringe)
```

Strategie z informacją — informed search

Strategie heurystyczne — wykorzystują dodatkową informację, specyficzną wiedzę o rozwiązywanym problemie, co pozwoli na zwiększenie wydajności algorytmów przeszukiwania w stosunku do metod ślepych.

Eksplozja kombinatoryczna to gwałtowny wzrost obliczeń przy niewielkim wzroście rozmiaru danych wejściowych (cecha zadań z klasy NP-trudnych).

Przykład wzrostu obliczeń na problemie TSP (*Traveling Salesman Person*) (komiwojażera). Założenia:

- dana jest mapa
- dane są dystanse pomiędzy parami miast
- każde miasto odwiedzane jest tylko raz
- cel: znaleźć najkrótsza drogę drogę, jeżeli podróż zaczyna się i kończy w jednym z miast

Dla N miast istnieje $1 \cdot 2 \cdot 3 \cdot \ldots \cdot (N-1)$ możliwych kombinacji. Czas jest proporcjonalny do N, a całkowity do $N!$, zatem dla 10 miast jest $10! = 3,628,800$ stanów do sprawdzenia przez metody zachłanne badające całą przestrzeń, a dla 11 miast aż $39,916,800$.

Ogólna strategia heurystyczna — best-first search

Best-first search (najpierw najlepszy) wykorzystuje funkcję oceny ($f(n)$).

- Dla węzła n funkcja $f(n)$ oszacowuje jego „użyteczność”.
- Funkcja oceny mierzy odległość do rozwiązania.
- Rozwija się najbardziej użyteczne (takie, które wydają się być najlepsze w bieżącej chwili) jeszcze nie rozwinięte węzły.
- Implementacja strategii to lista węzłów posortowanych według malejącej wartości $f(n)$ (przy założeniu, że $f(goal) = 0$, a $f(n) > 0$).

Warianty best-first search:

- Szukanie zachłanne (greedy best-first search)
- A^*

Funkcja heurystyczna

Funkcja odwzorowująca stany we współczynnik ich użyteczności.

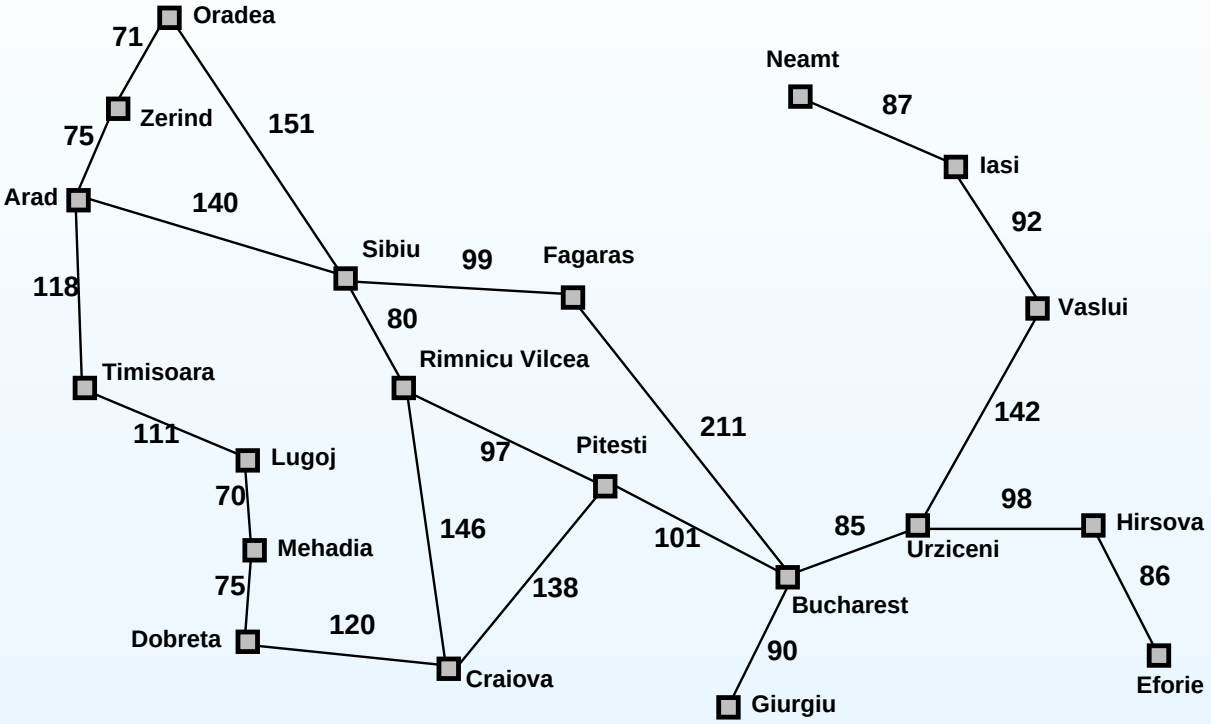
- Najczęściej jej przeciwdziedzina to $\mathcal{R}^+$.
- Zazwyczaj określa „odległość” od rozwiązania.
- Przyjmuje się, że $h(n) = 0$, gdy $n = cel$ (rozwiązanie).

$$h : \Omega \rightarrow \mathcal{R}$$

gdzie Ω przestrzeń stanów, a $\mathcal{R}$ zbiór liczb rzeczywistych.

$h(n)$ — jest oszacowaniem! kosztu przejścia od węzła n do rozwiązania.

Podróż po Rumunii z funkcją heurystyczną



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

h_{SLD} — straight-line distance heuristic.

- $h_{SLD}(Arad) = 366$
- h_{SLD} nie może być policzona z danych wejściowych problemu

Przeszukiwanie zachłanne

Greedy best-first search — próbuje rozwijać węzły, które są najbliższe rozwiązania, zakładając, że prawdopodobnie prowadzi to do najszybszego rozwiązania.

Oszacowuje koszt węzła stosując tylko funkcję heurystyczną

$$f(n) = h(n)$$

Minimalizowanie $h(n)$ jest podatne na niewłaściwy punkt startowy.

Greedy best-first search przypomina DFS, bo wybiera raczej jedną ścieżkę podążając do celu.

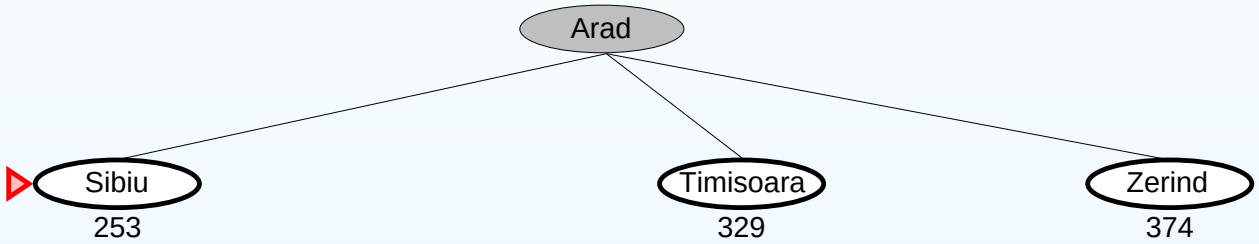
Przeszukiwanie zachłanne — przykład

Zakłada się, że za pomocą greedy search szukamy najkrótszej drogi z Aradu do Bukaresztu.

Stan początkowy: Arad

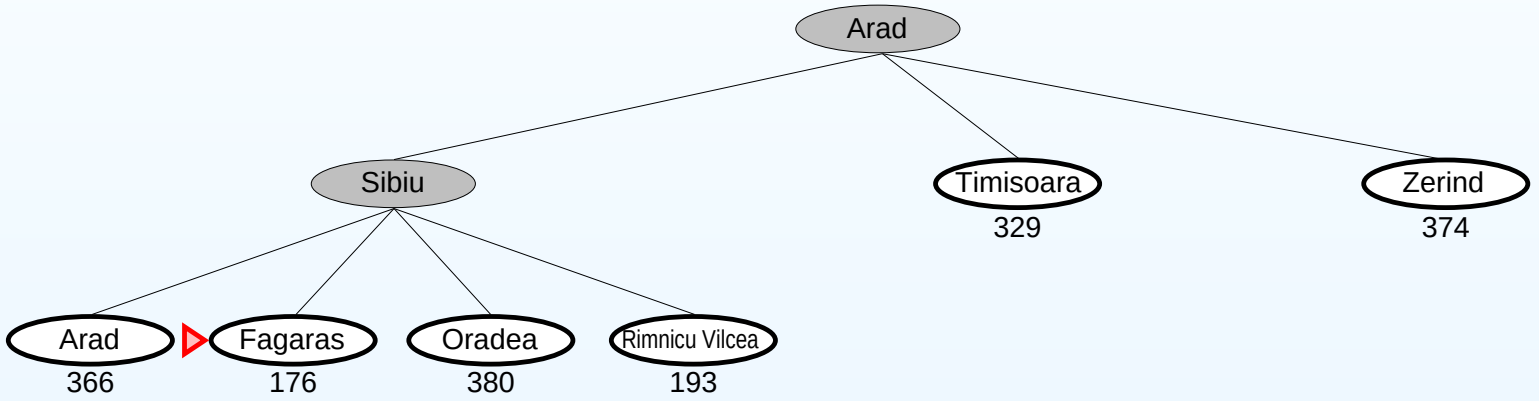
$$Lista = \{Arad.h(Arad) = 366\}$$

Przeszukiwanie zachłanne — przykład



Pierwszy krok rozwija węzeł Arad i generuje potomków: Sibiu, Timisoara and Zerind
$Lista = \{Sibiu.h(Sibiu) = 253, Timisoara.h(Timisoara) = 329, Zerind.h(Zerind) = 374\}$ Greedy best-first wybierze Sibiu.

Przeszukiwanie zachłanne — przykład

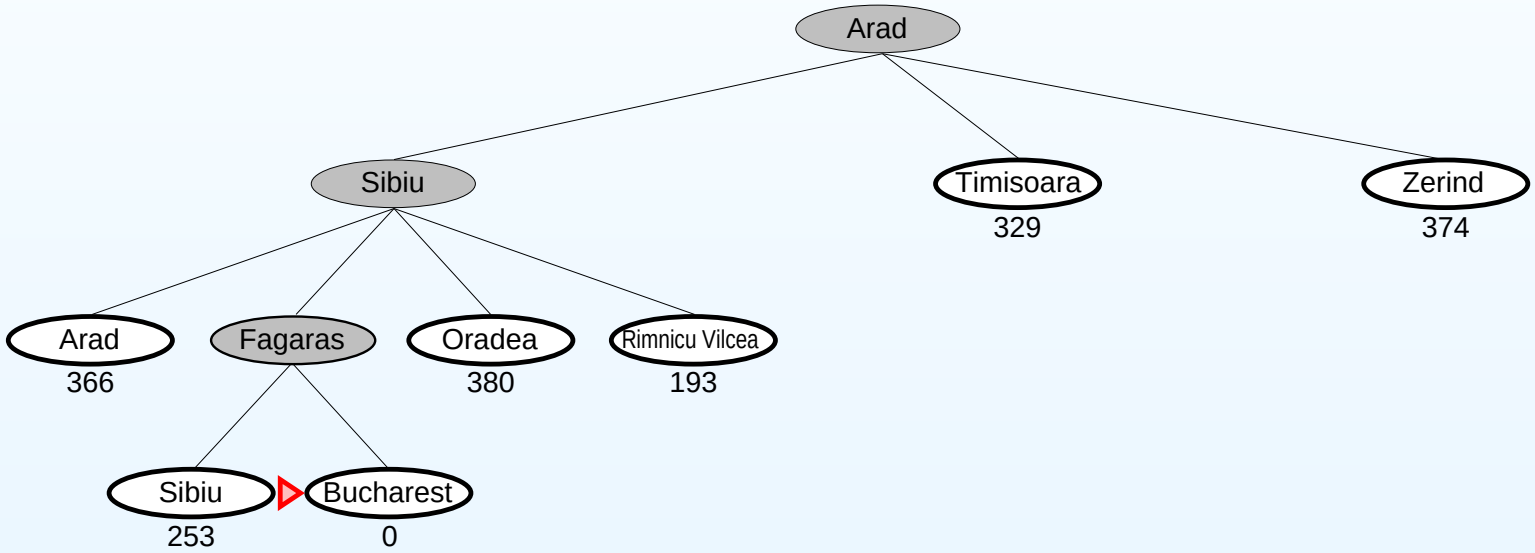


Jeżeli rozwiniemy Sibiu to otrzymamy listę:

$$Lista = \{Fagaras.h(Fagaras) = 176, RimnicuVilcea.h(RimnicuVilcea) = 193, Timisoara.h(Timisoara) = 329, Arad.h(Arad) = 366, Zerind.h(Zerind) = 374, Oradea.h(Oradea) = 380\}$$

Greedy best-first wybierze Fagaras.

Przeszukiwanie zachłanne — przykład



Jeżeli Fagaras jest rozwinięte to otrzymamy :Sibiu and Bukareszt.

Cel został osiągnięty $h(Bukareszt) = 0$, acz nie jest optymalny: (zobacz Arad, Sibiu, Rimnicu Vilcea, Pitesti)

Ocena algorytmu greedy best-first search

1. Zupełność (completeness): czy zawsze znajdzie rozwiązanie jeżeli ono istnieje?
   Nie, może utknąć w pętlach.
   Iasi → Neamt → Iasi → Neamt →
   Zupełny tylko w warunkach kontroli powtórzeń (graf) i skończonej przestrzeni stanów.
2. Złożoność czasowa (time complexity):
3. Obszar zajmowanej pamięci (space complexity):
4. Optymalność (optimality):

Ocena algorytmu greedy best-first search

1. Zupełność (completeness):
   Nie, może utknąć w pętlach.
   Zupełny tylko w warunkach kontroli powtórzeń (graf) i skończonej przestrzeni stanów.
2. Złożoność czasowa (time complexity): liczba węzłów generowana/rozwijana.
   $O(b^m)$, choć dobra heurystyka może dać znaczne zmniejszenie złożoności czasowej.
3. Obszar zajmowanej pamięci (space complexity):
4. Optymalność (optimality):

Ocena algorytmu greedy best-first search

1. Zupełność (completeness):
   Nie, może utknąć w pętlach.
   Zupełny tylko w warunkach kontroli powtórzeń (graf) i skończonej przestrzeni stanów.
2. Złożoność czasowa (time complexity):
   $O(b^m)$, choć dobra heurystyka może dać znaczne zmniejszenie złożoności czasowej.
3. Obszar zajmowanej pamięci (space complexity):
   maksymalna liczba węzłów w pamięci.
   Przechowuje wszystkie węzły $O(b^m)$.
4. Optymalność (optimality):

Ocena algorytmu greedy best-first search

1. Zupełność (completeness):
   Nie, może utknąć w pętlach.
   Zupełny tylko w warunkach kontroli powtórzeń (graf) i skończonej przestrzeni stanów.
2. Złożoność czasowa (time complexity):
   $O(b^m)$, choć dobra heurystyka może dać znaczne zmniejszenie złożoności czasowej.
3. Obszar zajmowanej pamięci (space complexity): $O(b^m)$.
4. Optymalność (optimality): czy zawsze znajdzie najmniej kosztowne rozwiązanie?
   Nie

Algorytm A^*

Funkcja oceny ma postać:

$$f(n) = g(n) + h(n)$$

gdzie:

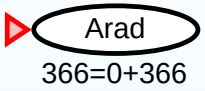
- $g(n)$ = koszt osiągnięcia bieżącego węzła n z węzła początkowego
- $h(n)$ = oszacowany koszt przejścia z węzła n do rozwiązania
- $f(n)$ = oszacowany pełny koszt ścieżki przez węzeł n do rozwiązania

Algorytm A^* ma szansę być zupełny i optymalny jeżeli $h(n)$ spełni pewne warunki.
Twierdzenie Algorytm A^* jest optymalny jeżeli funkcja heurystyczna jest dopuszczalna (admissible), tj. taką że:

- Nigdy nie przeceni kosztu dotarcia do rozwiązania: $h(n) \leq h^*(n)$, gdzie $h^*(n)$ jest rzeczywistym kosztem osiągnięcia rozwiązania z n
- Jest z natury optymistyczna bo zakłada że koszt rozwiązania jest mniejszy od rzeczywistego.
- $h(n) \geq 0$, stąd $h(G) = 0$ dla każdego rozwiązania G.

Przykład: $h_{\mathrm{SLD}}(n)$ - nigdy nie zawyży rzeczywistego dystansu.

Algorytm A^* — przykład

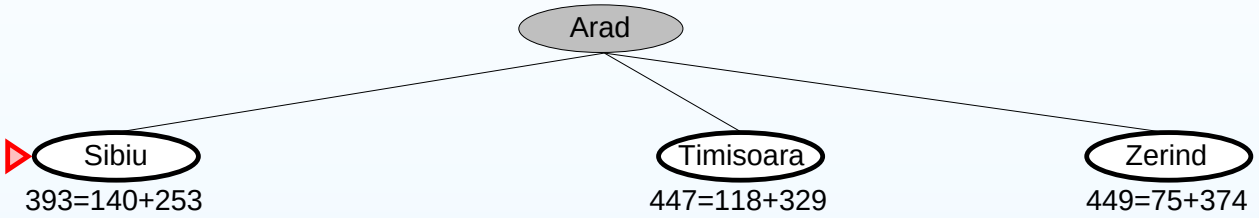


Szukamy najkrótszej drogi z Aradu do Bukaresztu.

Stan początkowy: Arad

$$f(Arad) = g(Arad) + h(Arad) = 0 + 366 = 360$$

Algorytm A^* — przykład

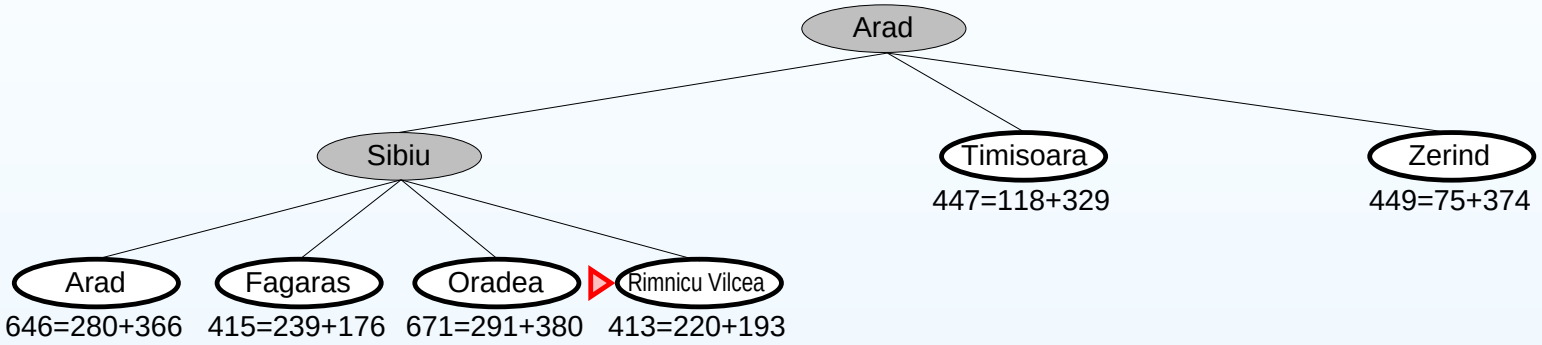


Pierwszy krok rozwija węzeł Arad i generuje potomków:

$$f(Sibiu) = g(Arad, Sibiu) + h(Sibiu) = 140 + 253 = 393$$
$$f(Timisoara) = g(Arad, Timisoara) + h(Timisoara) = 118 + 329 = 447$$
$$f(Zerind) = g(Arad, Zerind) + h(Zerind) = 75 + 374 = 449$$

A^* wybierze Sibiu.

Algorytm A^* — przykład



Jeżeli rozwiniemy Sibiu to otrzymamy:

$$f(Arad) = g(Sibiu, Arad) + h(Arad) = 280 + 366 = 646$$

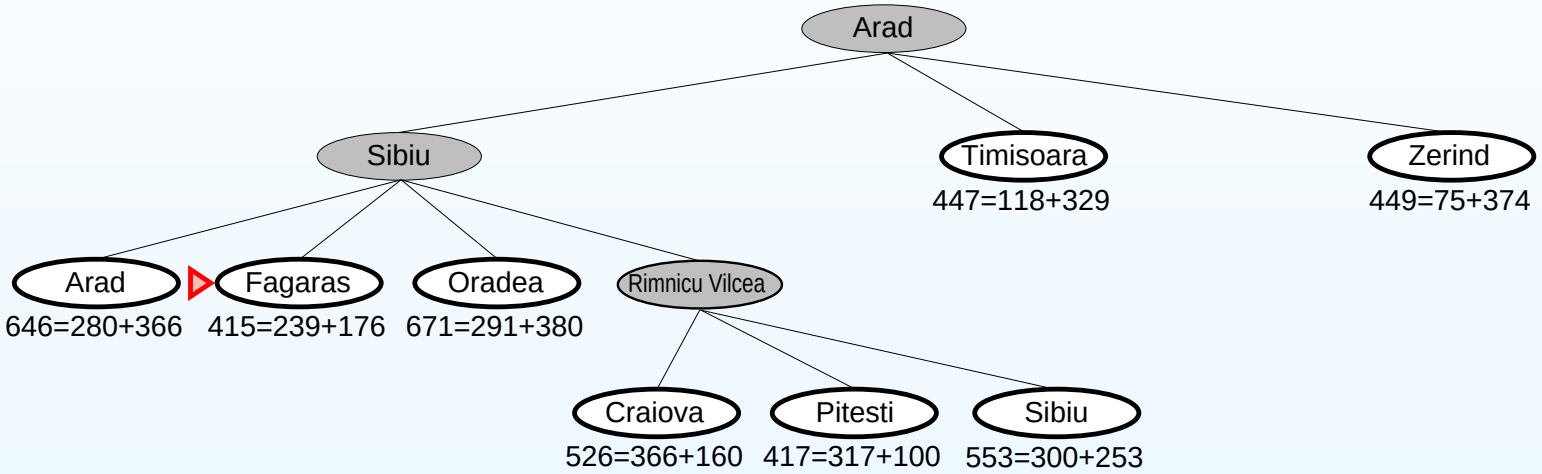
$$f(Fagaras) = g(Sibiu, Fagaras) + h(Fagaras) = 239 + 179 = 415$$

$$f(Oradea) = g(Sibiu, Oradea) + h(Oradea) = 291 + 380 = 671$$

$$f(RimnicuVilcea) = g(Sibiu, RimnicuVilcea) + h(RimnicuVilcea) = 220 + 192 = 413$$

A^* wybierze Rimnicu Vilcea.

Algorytm A^* — przykład

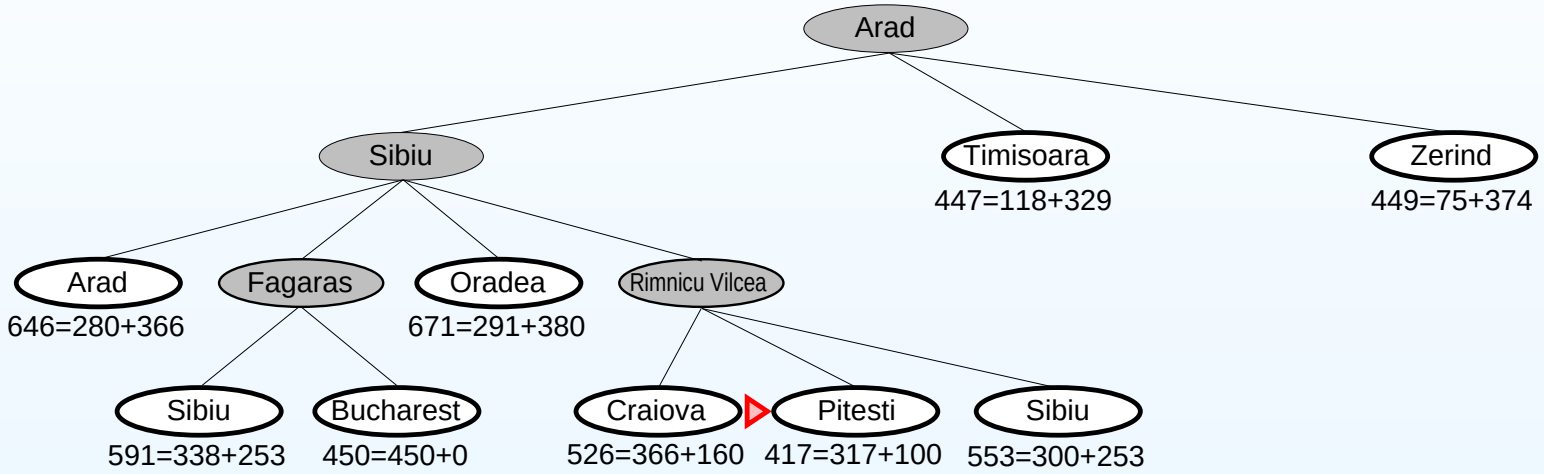


Jeżeli rozwiniemy Rimnicu Vilcea to otrzymamy:

$$f(Craiova) = g(RimnicuVilcea, Craiova) + h(Craiova) = 360 + 160.326$$
$$f(Pitesti) = g(RimnicuVilcea, Pitesti) + h(Pitesti) = 317 + 100 = 417$$
$$f(Sibiu) = g(RimnicuVilcea, Sibiu) + h(Sibiu) = 300 + 253 = 553$$

A^* wybierze Fagaras - węzeł pamiętany z poprzednich rozwinięć.

Algorytm A^* — przykład

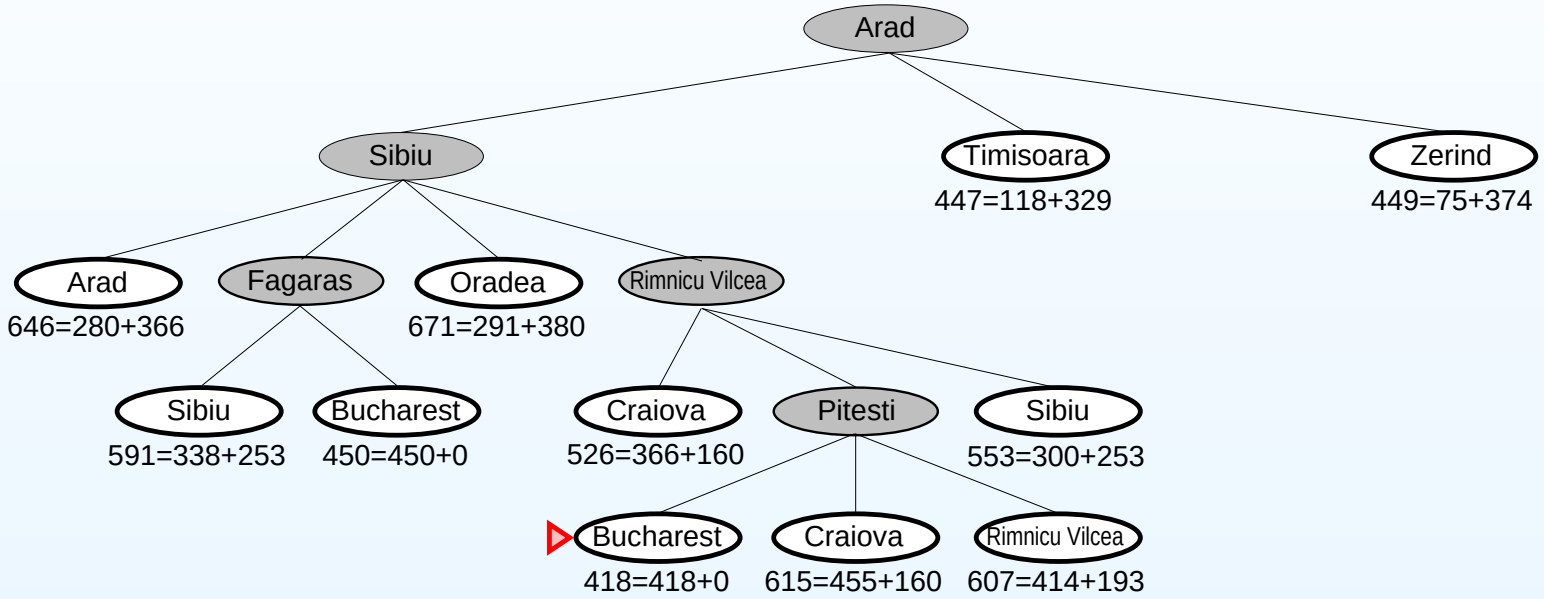


Jeżeli rozwiniemy Fagaras to najmniejszą wartość f ma Pitesi (417km):

$$f(Sibiu) = g(Fagaras, Sibiu) + h(Sibiu) = 338 + 253 = 591$$

$$f(Bucharest) = g(Fagaras, Bucharest) + h(Bucharest) = 450 + 0 = 450$$

Algorytm A^* — przykład

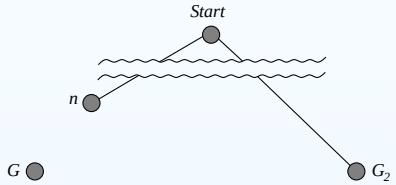


Wybieramy do rozwinięcia Pitesi, bo f ma najmniejszą wartość i dotrzemy najkrótszą drogą do Bukaresztu (418km):

$$f(Bucharest) = g(Pitesti, Bucharest) + h(Bucharest) = 418 + 0 = 418$$

Dowód na optymalność A^*

Zakładamy, że pewien suboptymalny cel G_2 został wygenerowany i znajduje się w kolejce do rozwinięcia. Niech n będzie nierozwiniętym węzłem na optymalnej drodze czyli do rozwiązania optymalnego G.



Jeżeli G_2 suboptymalne i dlatego, że $h(G_2) = 0$ to zachodziłoby

$$f(G_2) = h(G_2) + g(G_2) > f(G)$$

Rozważmy węzeł n leżący na ścieżce do rozwiązania. Z założenia o dopuszczalności heurystyki, $h(n)$ nie przeszacuje ścieżki i prawdziwa będzie zależność:

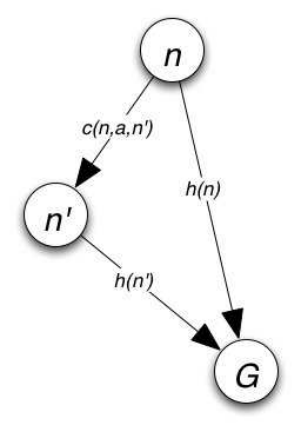
$f(G) \geq f(n) = h(n) + g(n)$

n nie zostało rozwinięte, co oznacza, że $f(n) \geq f(G_2)$. Zatem:

$$f(G) > f(G_2)$$

Zachodzi zatem sprzeczność i A^* nigdy nie wybierze G_2 do rozwinięcia.

Monotoniczność heurystyki



Dla grafów należy dodać jeszcze jeden warunek do heurystyki, by zapewnić jej optymalność. Cechę tę nazywa się monotonicznością lub spójnością. Dopuszczalna heurystyka h jest spójna (lub spełnia wymóg monotoniczności), jeżeli dla każdego węzła n i każdego jego następcy n' powstałego w wyniku akcji a zachodzi:

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

Jeżeli heurystyka h jest spójna, wówczas funkcja f wzdłuż dowolnej ścieżki jest niemalejąca:

$f(n) = g(n) + h(n)$

$f(n') = g(n) + c(n, n') + h(n')$

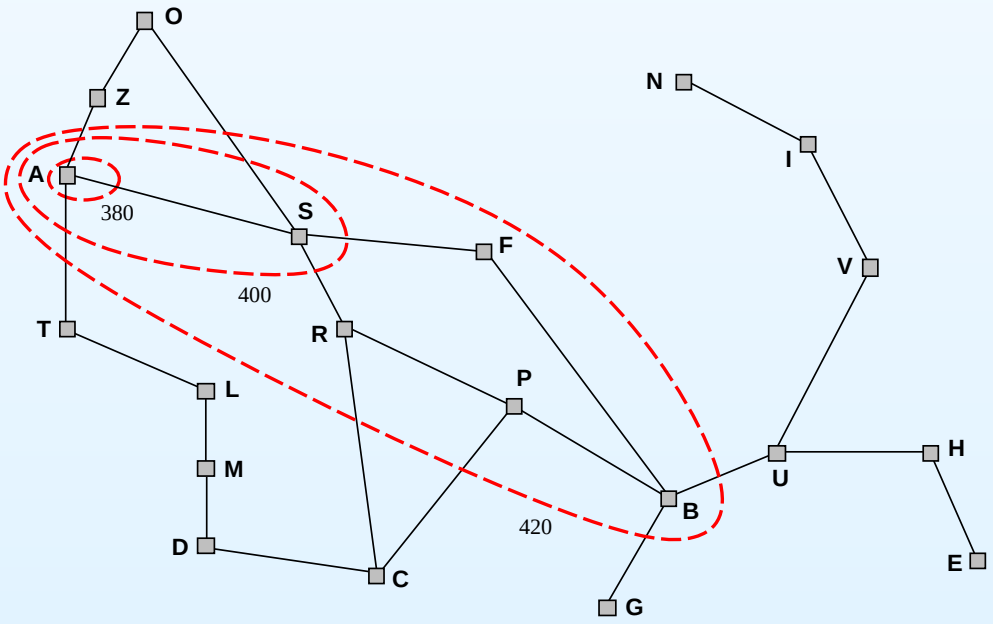
$h(n) \leq c(n, n') + h(n')$

$f(n) \leq f(n')$

Monotoniczność A^*

Z monotoniczności wynika, iż

- A^* rozwija węzły w kolejności rosnącej wartości f.
- Stopniowo wyznacza się f-kontur na węzłach. Kontur i zawiera wszystkie węzły z $f = f_i$, gdzie $f_i < f_{i+1}$.
- Jeżeli $h(n) = 0$ kontur ma kształt kół.
- A^* nigdy nie rozwinie węzłów z $f(n) > f(G)$, czyli ma własności odcinania nieobiecujących stanów.



Ocena algorytmu A^*

1. Zupełność (completeness): czy zawsze znajdzie rozwiązanie jeżeli ono istnieje?
   Tak, chyba że jest nieskończenie wiele węzłów z $f \leq f(G)$.
2. Złożoność czasowa (time complexity):
3. Obszar zajmowanej pamięci (space complexity):
4. Optymalność (optimality):

Ocena algorytmu A^*

1. Zupełność (completeness):
   Tak, chyba że jest nieskończenie wiele węzłów z $f \leq f(G)$.
2. Złożoność czasowa (time complexity): liczba węzłów generowana/rozwijana.
   Niestety wykładniczy
3. Obszar zajmowanej pamięci (space complexity):
4. Optymalność (optimality):

Ocena algorytmu A^*

1. Zupełność (completeness):
   Tak, chyba że jest nieskończenie wiele węzłów z $f \leq f(G)$.
2. Złożoność czasowa (time complexity):
   Niestety wykładniczy
3. Obszar zajmowanej pamięci (space complexity):
   maksymalna liczba węzłów w pamięci. Przechowuje wszystkie węzły.
4. Optymalność (optimality):

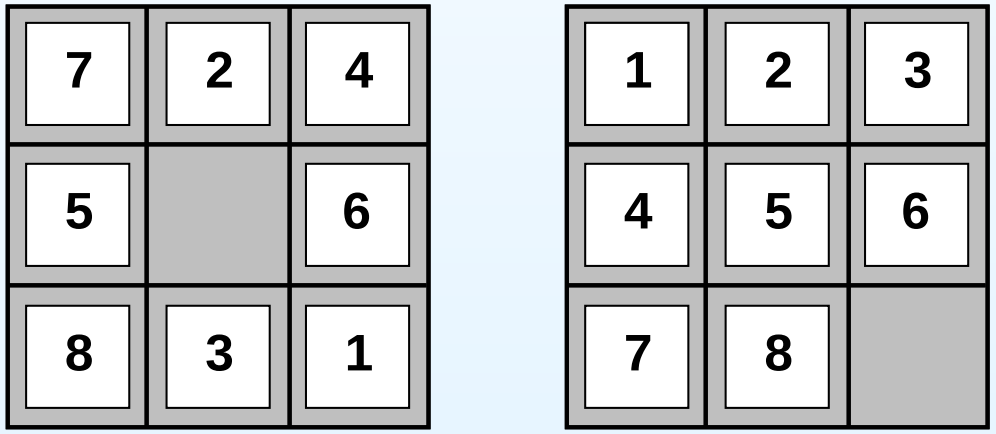
Ocena algorytmu A^*

1. Zupełność (completeness):
   Tak, chyba że jest nieskończenie wiele węzłów z $f \leq f(G)$.
2. Złożoność czasowa (time complexity):
   Niestety wykładniczy
3. Obszar zajmowanej pamięci (space complexity):
   Przechowuje wszystkie węzły.
4. Optymalność (optimality): czy zawsze znajdzie najmniej kosztowne rozwiązanie?
   Tak jeżeli $h(n)$ dopuszczalne,
   A^* rozwija wszystkie węzły o $f(n) < C^*$
   A^* rozwija niektóre węzły o $f(n) = C^*$
   A^* nie rozwija węzłów o $f(n) > C^*$

Jak dobrać heurystykę dla układanki 8-elementowej?

Liczba węzłów na drzewie do przeszukania to 3^{22} a na grafie 181440.

Proponowane heurystyki:
$h_1(n)$ = liczba niewłaściwie ułożonych elementów
$h_2(n)$ = całkowita odległość Manhattan (tj. liczba pól do lokalizacji końcowej)

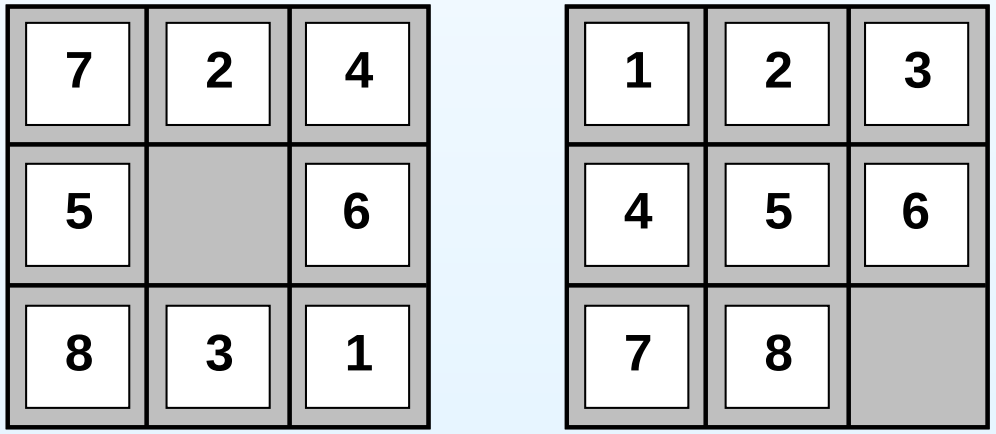


$h_1(S) =$
$h_2(S) =$

Jak dobrać heurystykę dla układanki 8-elementowej?

Liczba węzłów na drzewie do przeszukania to 3^{22} a na grafie 181440.

Proponowane heurystyki:
$h_1(n)$ = liczba niewłaściwie ułożonych elementów
$h_2(n)$ = całkowita odległość Manhattan (tj. liczba pól do lokalizacji końcowej)



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

Porównanie

Jeżeli $h_2(n) \geq h_1(n)$ dla każdego n (przy czym obie dopuszczalne), to h_2 zdominuje h_1 i gwarantuje przeszukanie mniejszej liczby węzłów.

Przykładowe koszty poszukiwania do zadanego poziomu dla układanki 8-elementowej:

$d = 8$	IDS = 6384 węzłów
	A*(h_1) = 39 węzłów
	A*(h_2) = 25 węzłów
$d = 14$	IDS — nieopłacalne
	A*(h_1) = 539 węzłów
	A*(h_2) = 113 węzłów

Dla k heurystyk dopuszczalnych $h_a, h_b, \ldots h_k$, określonych dla zadania wybiera się:

$$h(n) = \max(h_a(n), h_b(n), \ldots, h_k(n))$$

Dla układanki h_2 jest zatem lepsza.

Jak znaleźć funkcję heurystyczną?

Relaxed problem polega na zmniejszeniu liczby reguł ograniczających rozwiązanie i wówczas poszukiwanie heurystyki. Np. dla układanki.

- Można przesuwać klocki gdziekolwiek, zamiast na puste sąsiednie pole, to $h_1(n)$ dałoby najkrótsze rozwiązanie.
- Jeżeli klocki można przesuwać na sąsiednie pola, nawet gdy są zajęte, to $h_2(n)$ dałoby najkrótsze rozwiązanie.

Takie uproszczenia dają ocenę nie wyższą niż koszt dokładny problemu w rzeczywistym ujęciu.

Opracowano program (ABSolver), który znalazł heurystyki dla kostki Rubika.

Metody haurystyczne przeszukiwania z ograniczeniem pamięci

- Iteracyjny A^* (IDA^*) - iterative-deepening A^*: podobnie jak IDS tylko $cutoff = f$ (w IDS - głębokość drzewa)
- Rekurencyjny najpierw najlepszy - Recursive best-first search(RBFS)
- Memory-bounded A^* ($(S)MA^*$): Odrzuca najgorsze liście gdy brakuje pamięci.