Sztuczna Inteligencja i Systemy Doradcze

Przeszukiwanie przestrzeni stanów — algorytmy ślepe

Strategie slepe

Strategie *ślepe* korzystają z informacji dostępnej jedynie w definicji problemu:

- ♢ Przeszukiwanie wszerz
- ♢ Strategia jednolitego kosztu
- ♢ Przeszukiwanie wgłąb
- ♢ Przeszukiwanie ograniczone wgłąb
- ♢ Przeszukiwanie iteracyjnie pogłębiane
- ♢ Przeszukiwanie dwukierunkowe

Przeszukiwanie wszerz

Wykonuje ekspansję najpłytszego węzła spośród tych, ktore nie były jeszcze rozszerzone

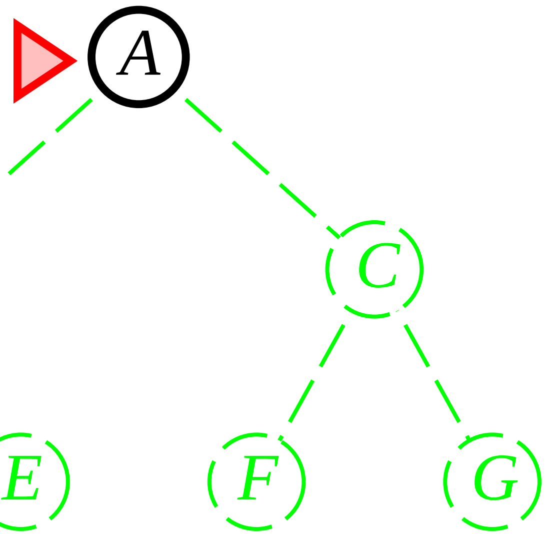
Implementacja: *fringe* jest kolejką FIFO, tzn. nowe następniki dodawane są na koniec kolejki

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 wszerz

Wykonuje ekspansję najpłytszego węzła spośród tych, ktore nie były jeszcze rozszerzone

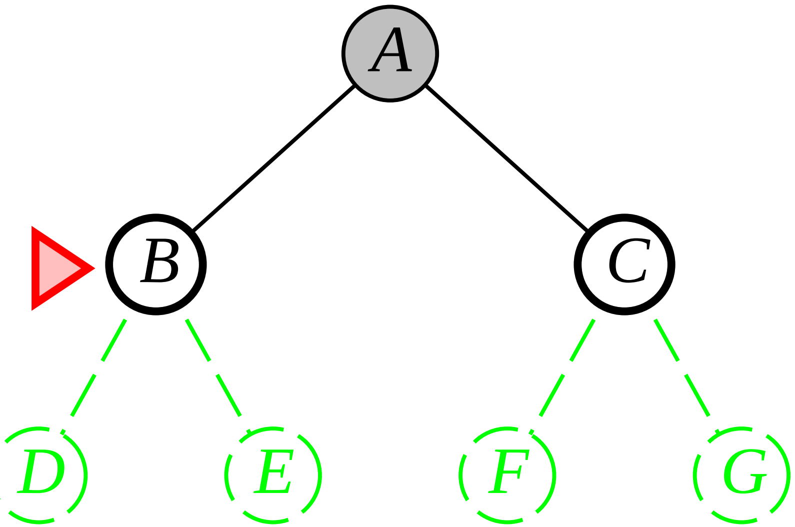
Implementacja: *fringe* jest kolejką FIFO, tzn. nowe następniki dodawane są na koniec kolejki



Przeszukiwanie wszerz

Wykonuje ekspansję najpłytszego węzła spośród tych, ktore nie były jeszcze rozszerzone

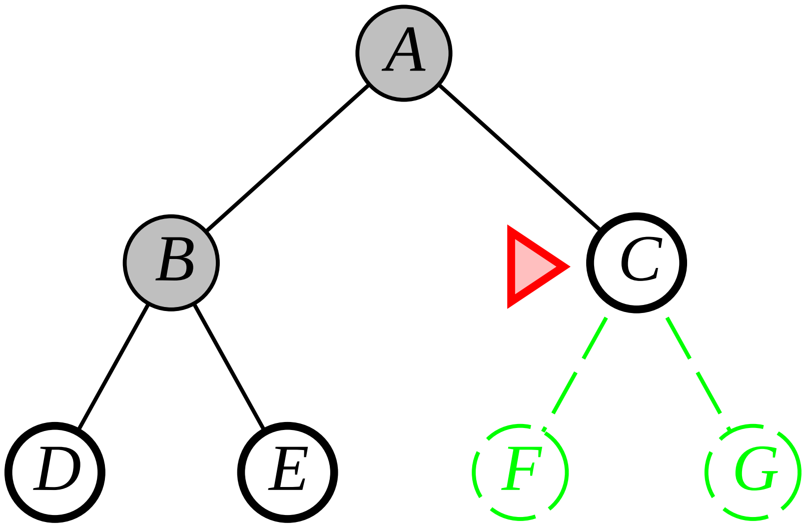
Implementacja: *fringe* jest kolejką FIFO, tzn. nowe następniki dodawane są na koniec kolejki



Przeszukiwanie wszerz

Wykonuje ekspansję najpłytszego węzła spośród tych, ktore nie były jeszcze rozszerzone

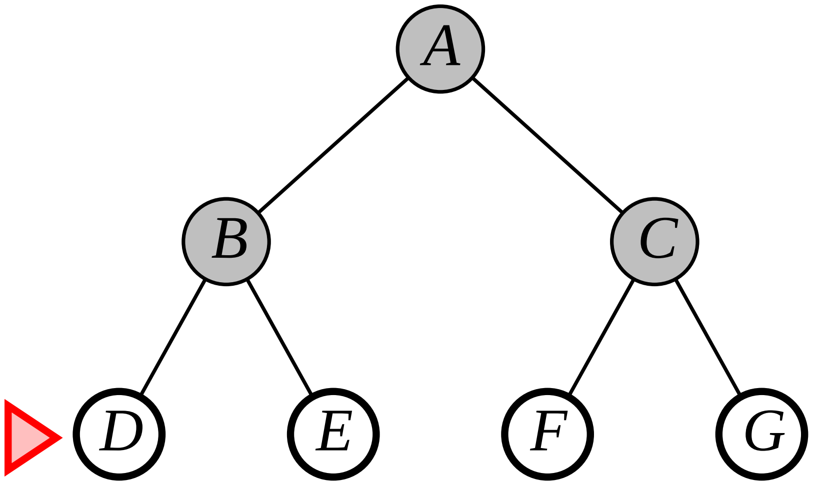
Implementacja: *fringe* jest kolejką FIFO, tzn. nowe następniki dodawane są na koniec kolejki



Przeszukiwanie wszerz

Wykonuje ekspansję najpłytszego węzła spośród tych, ktore nie były jeszcze rozszerzone

Implementacja: *fringe* jest kolejką FIFO, tzn. nowe następniki dodawane są na koniec kolejki



Przeszukiwania wszerz: wlasnosci

Zupełność??

Przeszukiwania wszerz: wlasnosci

Zupełność?? Tak (jeśli b jest skończone)

Złożoność czasowa??

Przeszukiwania wszerz: wlasnosci

Zupełność?? Tak (jeśli b jest skończone)

Złożoność czasowa?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$
tzn. wykładnicza względem d

Złożoność pamięciowa??

Przeszukiwania wszerz: wlasnosci

Zupełność?? Tak (jeśli b jest skończone)

Złożoność czasowa?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$
tzn. wykładnicza względem d

Złożoność pamięciowa?? $O(b^{d+1})$ (przechowuje każdy węzeł w pamięci)

Optymalność??

Przeszukiwania wszerz: wlasnosci

Zupełność?? Tak (jeśli b jest skończone)

Złożoność czasowa?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$
tzn. wykładnicza względem d

Złożoność pamięciowa?? $O(b^{d+1})$ (przechowuje każdy węzeł w pamięci)

Optymalność?? Tak (jeśli koszt każdego kroku $= 1$);
w ogólności nieoptymalny

Przeszukiwania wszerz: wlasnosci

Zupełność?? Tak (jeśli b jest skończone)

Złożoność czasowa?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$
tzn. wykładnicza względem d

Złożoność pamięciowa?? $O(b^{d+1})$ (przechowuje każdy węzeł w pamięci)

Optymalność?? Tak (jeśli koszt każdego kroku $= 1$);
w ogólności nieoptymalny

Złożoność pamięciowa jest dużym problemem; można łatwo generować węzły z szybkością 10MB/sek czyli 24godz = 860GB.

Strategia jednolitego kosztu

Wykonuje ekspansję węzła o najmniejszym koszcie spośród tych, ktore nie były jeszcze rozszerzone

Implementacja: *fringe* = kolejka priorytetowa porządkująca węzły według kosztu ścieżki od korzenia

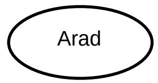
```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem] applied to STATE(node) succeeds return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

Odpowiada przeszukiwaniu wszerz jeśli koszt wszystkich pojedynczych akcji jest ten sam

Strategia jednolitego kosztu

Wykonuje ekspansję węzła o najmniejszym koszcie spośród tych, ktore nie były jeszcze rozszerzone

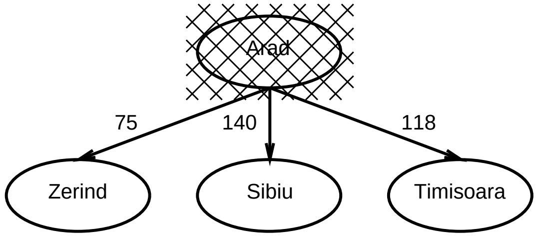
Implementacja: $fringe$ = kolejka priorytetowa porządkująca węzły według kosztu ścieżki od korzenia



Strategia jednolitego kosztu

Wykonuje ekspansję węzła o najmniejszym koszcie spośród tych, ktore nie były jeszcze rozszerzone

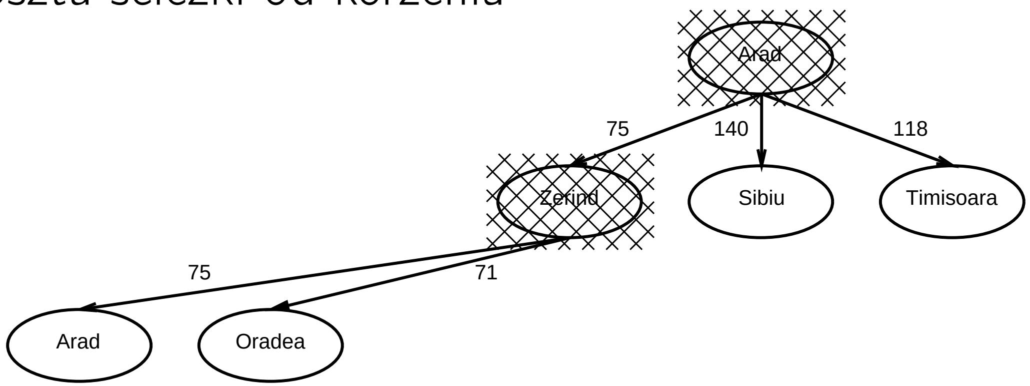
Implementacja: $fringe$ = kolejka priorytetowa porządkująca węzły według kosztu ścieżki od korzenia



Strategia jednolitego kosztu

Wykonuje ekspansję węzła o najmniejszym koszcie spośród tych, ktore nie były jeszcze rozszerzone

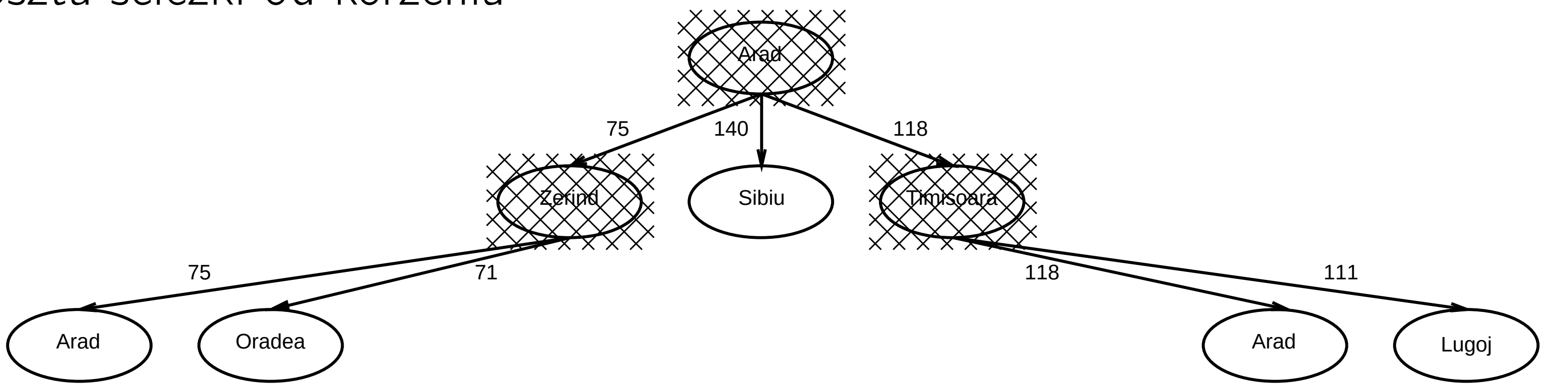
Implementacja: $fringe$ = kolejka priorytetowa porządkująca węzły według kosztu ścieżki od korzenia



Strategia jednolitego kosztu

Wykonuje ekspansję węzła o najmniejszym koszcie spośród tych, ktore nie były jeszcze rozszerzone

Implementacja: $fringe$ = kolejka priorytetowa porządkująca węzły według kosztu ścieżki od korzenia



Strategia jednolitego kosztu: wlasnosci

Zupełność??

Tak, jeśli koszt wszystkich akcji $\geq \epsilon$, dla pewnego $\epsilon > 0$

Złożoność czasowa??

Liczba węzłów, dla których $g \leq$ koszt optymalnego rozwiązania
$O(b^{\lceil C^*/\epsilon \rceil})$, gdzie C^* jest kosztem optymalnego rozwiązania

Złożoność pamięciowa??

Liczba węzłów, dla których $g \leq$ koszt optymalnego rozwiązania
$O(b^{\lceil C^*/\epsilon \rceil})$

Optymalność??

Tak — węzły są uporządkowane rosnąco względem $g(n)$

Przeszukiwanie wglab

Wykonuje ekspansję najgłebszego węzła spośród tych, ktore nie były jeszcze rozszerzone

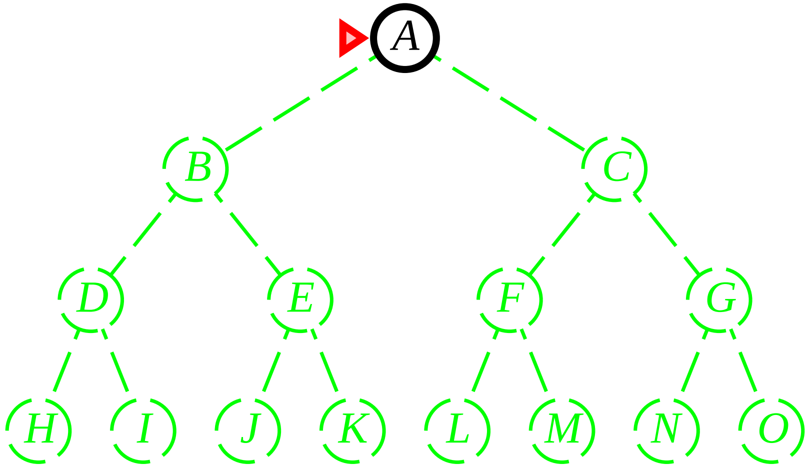
Implementacja: *fringe* jest kolejką LIFO, tzn. nowe następniki dodawane są na początek kolejki

```
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 wglab

Wykonuje ekspansję najgłebszego węzła spośród tych, ktore nie były jeszcze rozszerzone

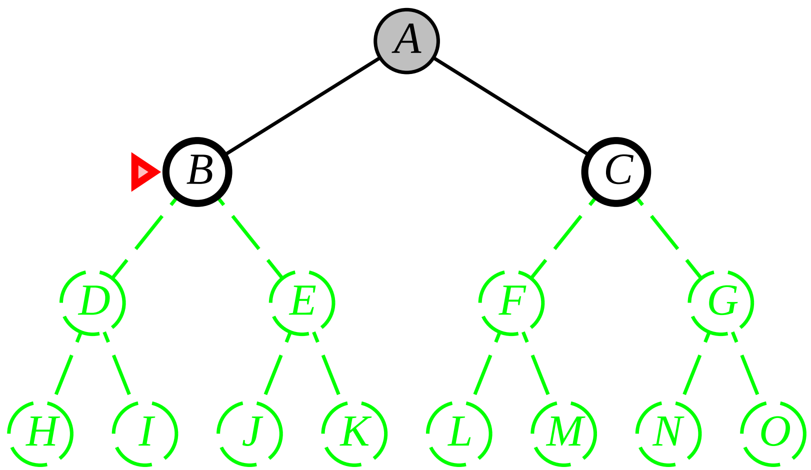
Implementacja: *fringe* jest kolejką LIFO, tzn. nowe następniki dodawane są na początek kolejki



Przeszukiwanie wglab

Wykonuje ekspansję najgłebszego węzła spośród tych, ktore nie były jeszcze rozszerzone

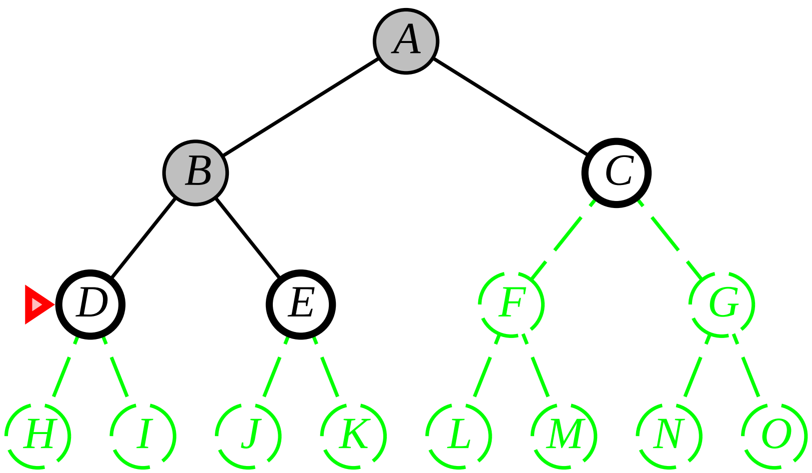
Implementacja: *fringe* jest kolejką LIFO, tzn. nowe następniki dodawane są na początek kolejki



Przeszukiwanie wglab

Wykonuje ekspansję najgłebszego węzła spośród tych, ktore nie były jeszcze rozszerzone

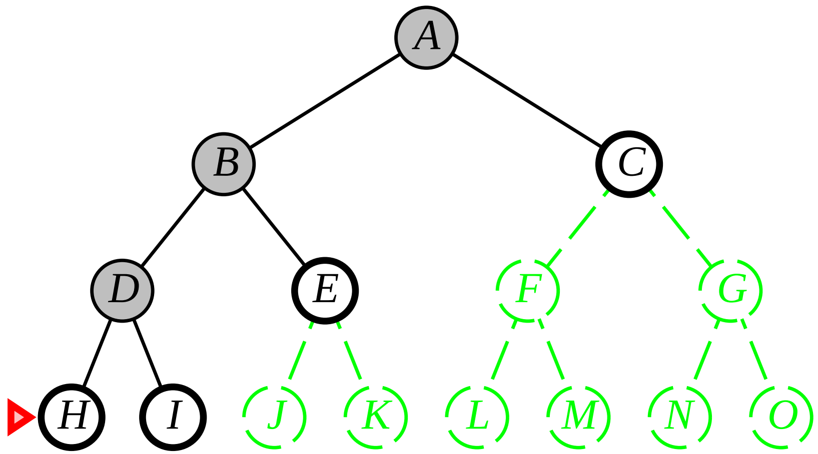
Implementacja: *fringe* jest kolejką LIFO, tzn. nowe następniki dodawane są na początek kolejki



Przeszukiwanie wglab

Wykonuje ekspansję najgłebszego węzła spośród tych, ktore nie były jeszcze rozszerzone

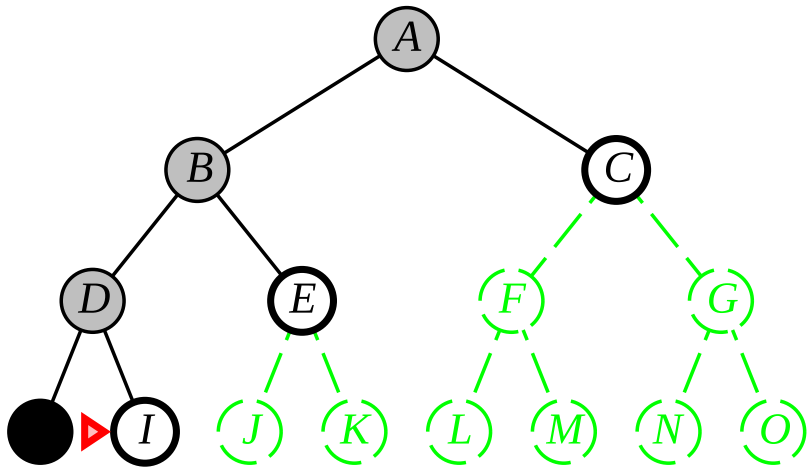
Implementacja: *fringe* jest kolejką LIFO, tzn. nowe następniki dodawane są na początek kolejki



Przeszukiwanie wglab

Wykonuje ekspansję najgłebszego węzła spośród tych, ktore nie były jeszcze rozszerzone

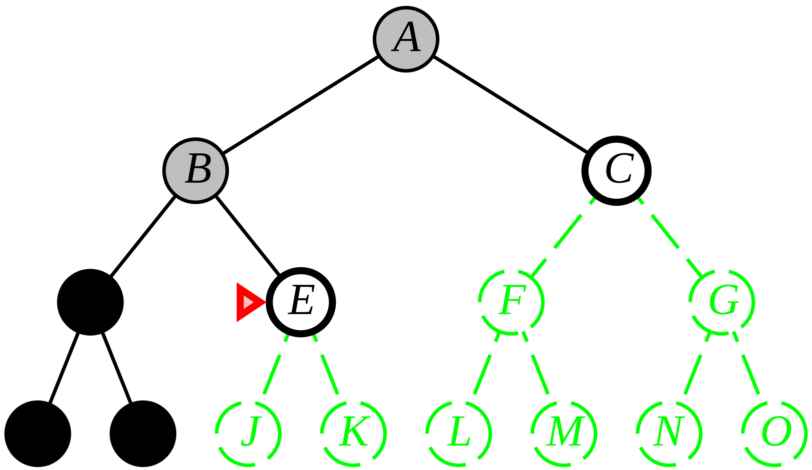
Implementacja: *fringe* jest kolejką LIFO, tzn. nowe następniki dodawane są na początek kolejki



Przeszukiwanie wglab

Wykonuje ekspansję najgłebszego węzła spośród tych, ktore nie były jeszcze rozszerzone

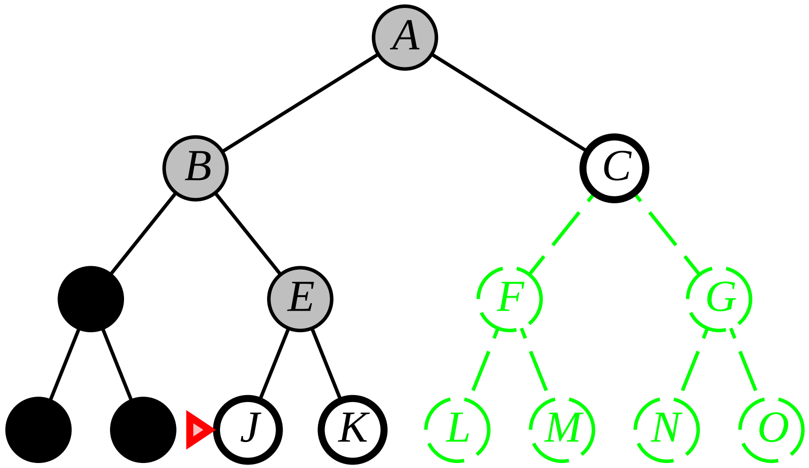
Implementacja: *fringe* jest kolejką LIFO, tzn. nowe następniki dodawane są na początek kolejki



Przeszukiwanie wglab

Wykonuje ekspansję najgłebszego węzła spośród tych, ktore nie były jeszcze rozszerzone

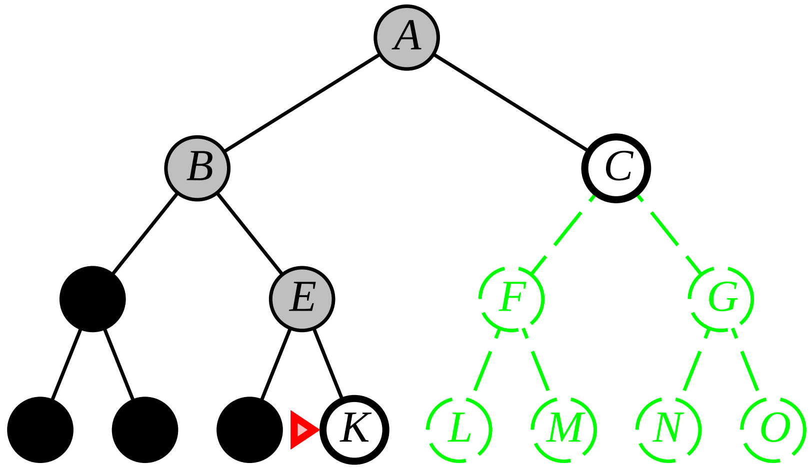
Implementacja: *fringe* jest kolejką LIFO, tzn. nowe następniki dodawane są na początek kolejki



Przeszukiwanie wglab

Wykonuje ekspansję najgłebszego węzła spośród tych, ktore nie były jeszcze rozszerzone

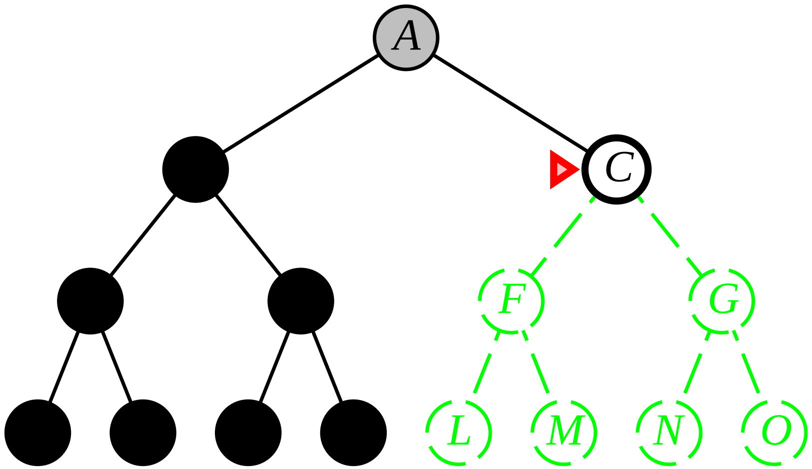
Implementacja: *fringe* jest kolejką LIFO, tzn. nowe następniki dodawane są na początek kolejki



Przeszukiwanie wglab

Wykonuje ekspansję najgłebszego węzła spośród tych, ktore nie były jeszcze rozszerzone

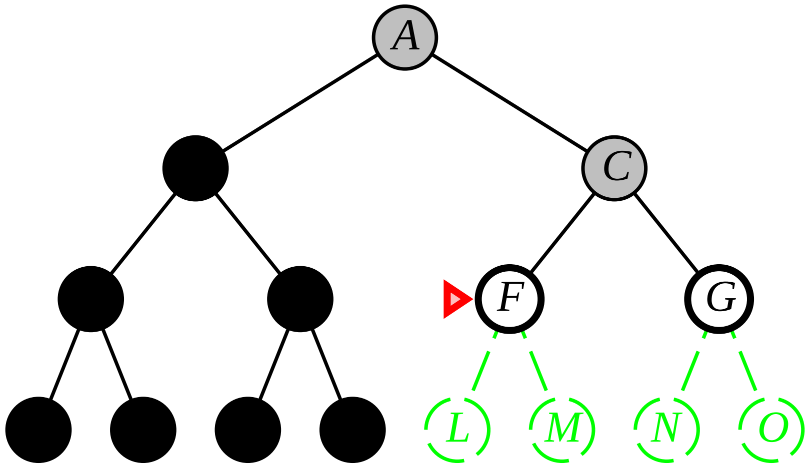
Implementacja: *fringe* jest kolejką LIFO, tzn. nowe następniki dodawane są na początek kolejki



Przeszukiwanie wglab

Wykonuje ekspansję najgłebszego węzła spośród tych, ktore nie były jeszcze rozszerzone

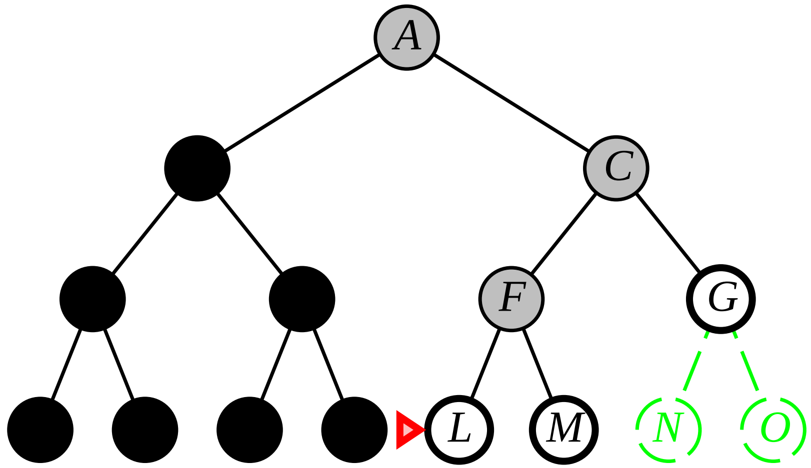
Implementacja: *fringe* jest kolejką LIFO, tzn. nowe następniki dodawane są na początek kolejki



Przeszukiwanie wglab

Wykonuje ekspansję najgłebszego węzła spośród tych, ktore nie były jeszcze rozszerzone

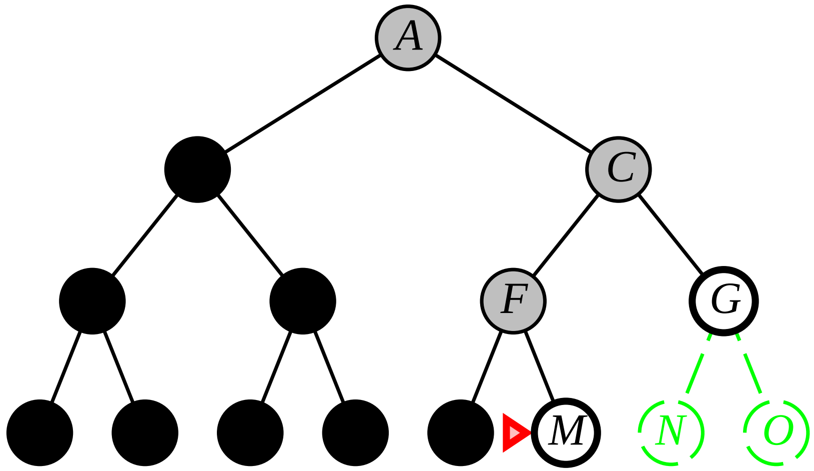
Implementacja: *fringe* jest kolejką LIFO, tzn. nowe następniki dodawane są na początek kolejki



Przeszukiwanie wglab

Wykonuje ekspansję najgłebszego węzła spośród tych, ktore nie były jeszcze rozszerzone

Implementacja: *fringe* jest kolejką LIFO, tzn. nowe następniki dodawane są na początek kolejki



Przeszukiwanie wglab: wlasnosci

Zupełność??

Przeszukiwanie wglab: wlasnosci

Zupełność??

Brak, zawodzi w przestrzeniach o nieskończonej głębokości
oraz w przestrzeniach z pętlami

Po dodaniu eliminacji stanów powtarzających się wzdłuż ścieżki
$\Rightarrow$ zupełność w przestrzeniach skończonych

Złożoność czasowa??

Przeszukiwanie wglab: wlasnosci

Zupełność??

Brak, zawodzi w przestrzeniach o nieskończonej głębokości
oraz w przestrzeniach z pętlami
Po dodaniu eliminacji stanów powtarzających się wzdłuż ścieżki
$\Rightarrow$ zupełność w przestrzeniach skończonych

Złożoność czasowa??

$O(b^m)$: okropne jeśli m jest dużo większe niż d
jeśli rozwiązania są gęste, może być szybsze niż przeszukiwanie wszerz

Złożoność pamięciowa??

Przeszukiwanie wglab: wlasnosci

Zupełność??

Brak, zawodzi w przestrzeniach o nieskończonej głębokości
oraz w przestrzeniach z pętlami
Po dodaniu eliminacji stanów powtarzających się wzdłuż ścieżki
$\Rightarrow$ zupełność w przestrzeniach skończonych

Złożoność czasowa??

$O(b^m)$: okropne jeśli m jest dużo większe niż d
jeśli rozwiązania są gęste, może być szybsze niż przeszukiwanie wszerz

Złożoność pamięciowa??

$O(bm)$, tzn. pamięć liniowa!

Optymalność??

Przeszukiwanie wglab: wlasnosci

Zupełność??

Brak, zawodzi w przestrzeniach o nieskończonej głębokości
oraz w przestrzeniach z pętlami
Po dodaniu eliminacji stanów powtarzających się wzdłuż ścieżki
$\Rightarrow$ zupełność w przestrzeniach skończonych

Złożoność czasowa??

$O(b^m)$: okropne jeśli m jest dużo większe niż d
jeśli rozwiązania są gęste, może być szybsze niż przeszukiwanie wszerz

Złożoność pamięciowa??

$O(bm)$, tzn. pamięć liniowa!

Optymalność?? Brak

Przeszukiwanie ograniczone wglab

Przeszukiwanie wgłąb z ograniczeniem na głębokość l, tzn. węzły na głębokości l nie mają następników

Implementacja rekurencyjna:

function DEPTH-LIMITED-SEARCH(*problem*, *limit*) **returns** soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[*problem*]), *problem*, *limit*)

function RECURSIVE-DLS(*node*, *problem*, *limit*) **returns** soln/fail/cutoff
  cutoff-occurred? ← false
  if GOAL-TEST[*problem*](STATE[*node*]) **then return** *node*
  else if DEPTH[*node*] = *limit* **then return** *cutoff*
  else for each *successor* **in** EXPAND(*node*, *problem*) **do**
    result ← RECURSIVE-DLS(*successor*, *problem*, *limit*)
    if *result* = *cutoff* **then** *cutoff-occurred?* ← true
    else if *result* ≠ *failure* **then return** *result*
  if *cutoff-occurred?* **then return** *cutoff* **else return** *failure*

Przeszukiwanie iteracyjnie poglebiane

Powtarza przeszukiwanie ograniczone wgłąb z rosnącym ograniczeniem na głębokość przeszukiwania

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
    end
```

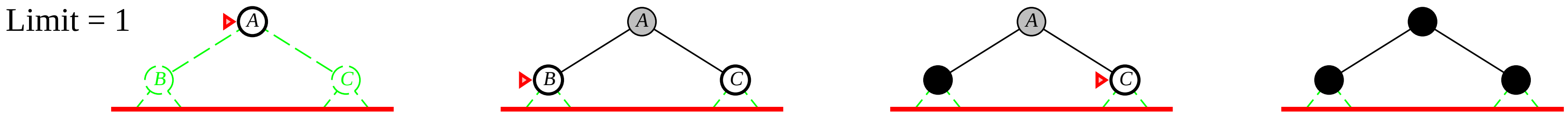
Przeszukiwanie iteracyjnie poglebiane $l = 0$

Powtarza przeszukiwanie ograniczone wgłąb z rosnącym ograniczeniem na głębokość przeszukiwania

Limit = 0

Przeszukiwanie iteracyjnie poglebiane $l = 1$

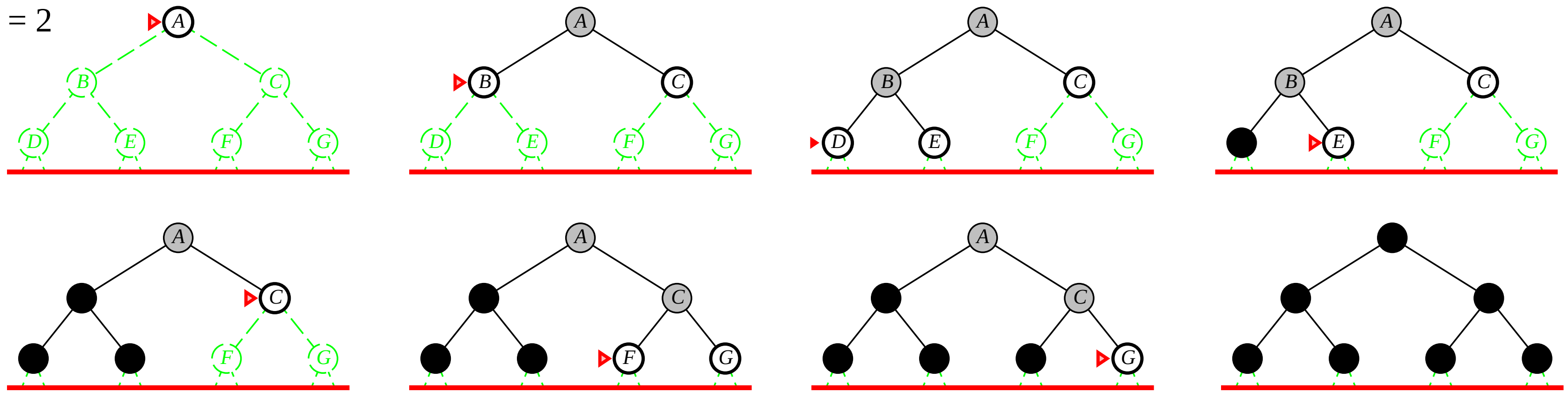
Powtarza przeszukiwanie ograniczone wgłąb z rosnącym ograniczeniem na głębokość przeszukiwania



Przeszukiwanie iteracyjnie poglebiane $l = 2$

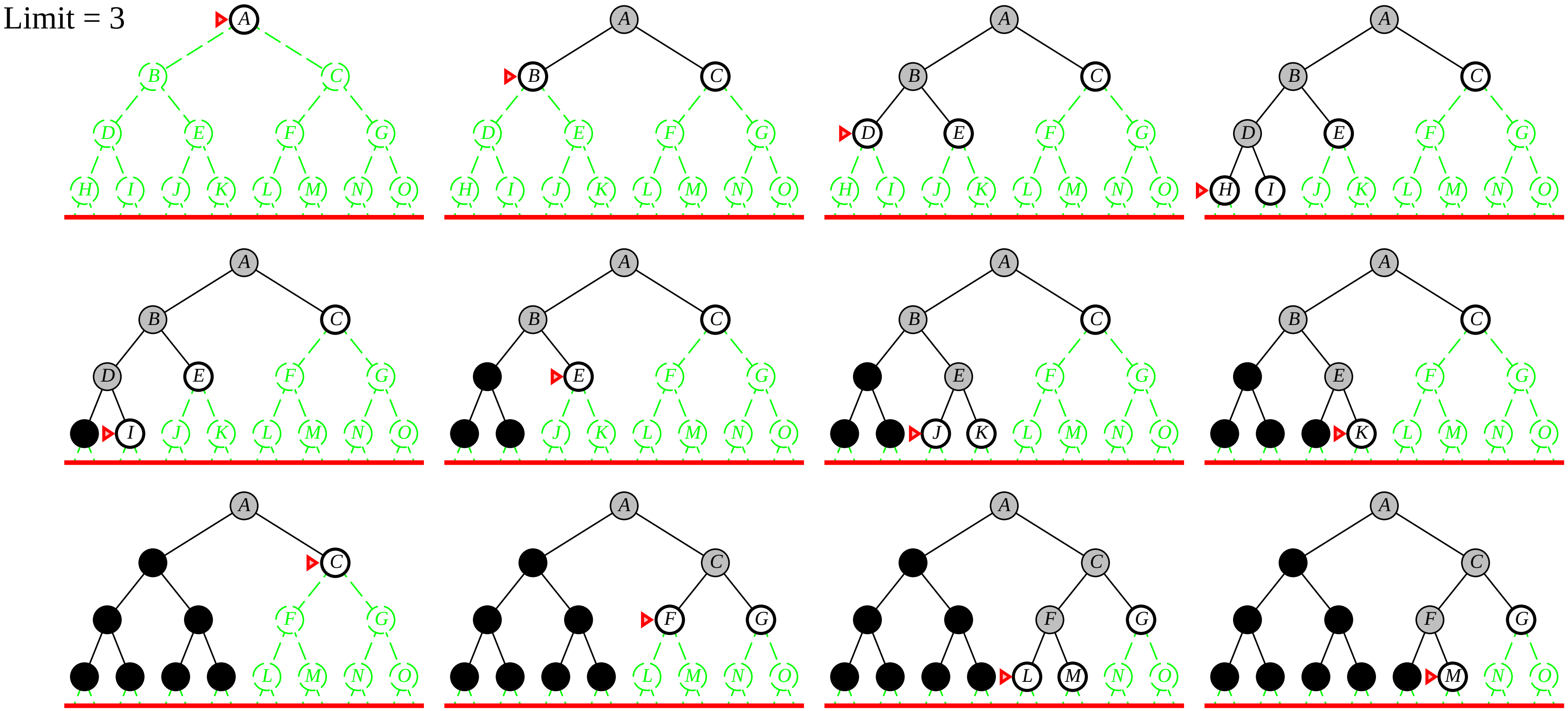
Powtarza przeszukiwanie ograniczone wgłąb z rosnącym ograniczeniem na głębokość przeszukiwania

Limit = 2



Przeszukiwanie iteracyjnie poglebiane $l = 3$

Powtarza przeszukiwanie ograniczone wgłąb z rosnącym ograniczeniem na głębokość przeszukiwania



Przeszukiwanie iteracyjnie poglebiane: wlasnosci

Zupełność??

Przeszukiwanie iteracyjnie poglebiane: wlasnosci

Zupełność?? Tak

Złożoność czasowa??

Przeszukiwanie iteracyjnie poglebiane: wlasnosci

Zupełność?? Tak

Złożoność czasowa?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Złożoność pamięciowa??

Przeszukiwanie iteracyjnie poglebiane: wlasnosci

Zupełność?? Tak

Złożoność czasowa?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Złożoność pamięciowa?? $O(bd)$

Optymalność??

Przeszukiwanie iteracyjnie poglebiane: wlasnosci

Zupełność?? Tak

Złożoność czasowa?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Złożoność pamięciowa?? $O(bd)$

Optymalność?? Tak, jeśli koszt wszystkich akcji jest taki sam
Można zmodyfikować tak, żeby przeszukiwać drzewo jednolitego kosztu

Przeszukiwanie iteracyjnie poglebiane: wlasnosci

Zupełność?? Tak

Złożoność czasowa?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Złożoność pamięciowa?? $O(bd)$

Optymalność?? Tak, jeśli koszt wszystkich akcji jest taki sam
Można zmodyfikować tak, żeby przeszukiwać drzewo jednolitego kosztu

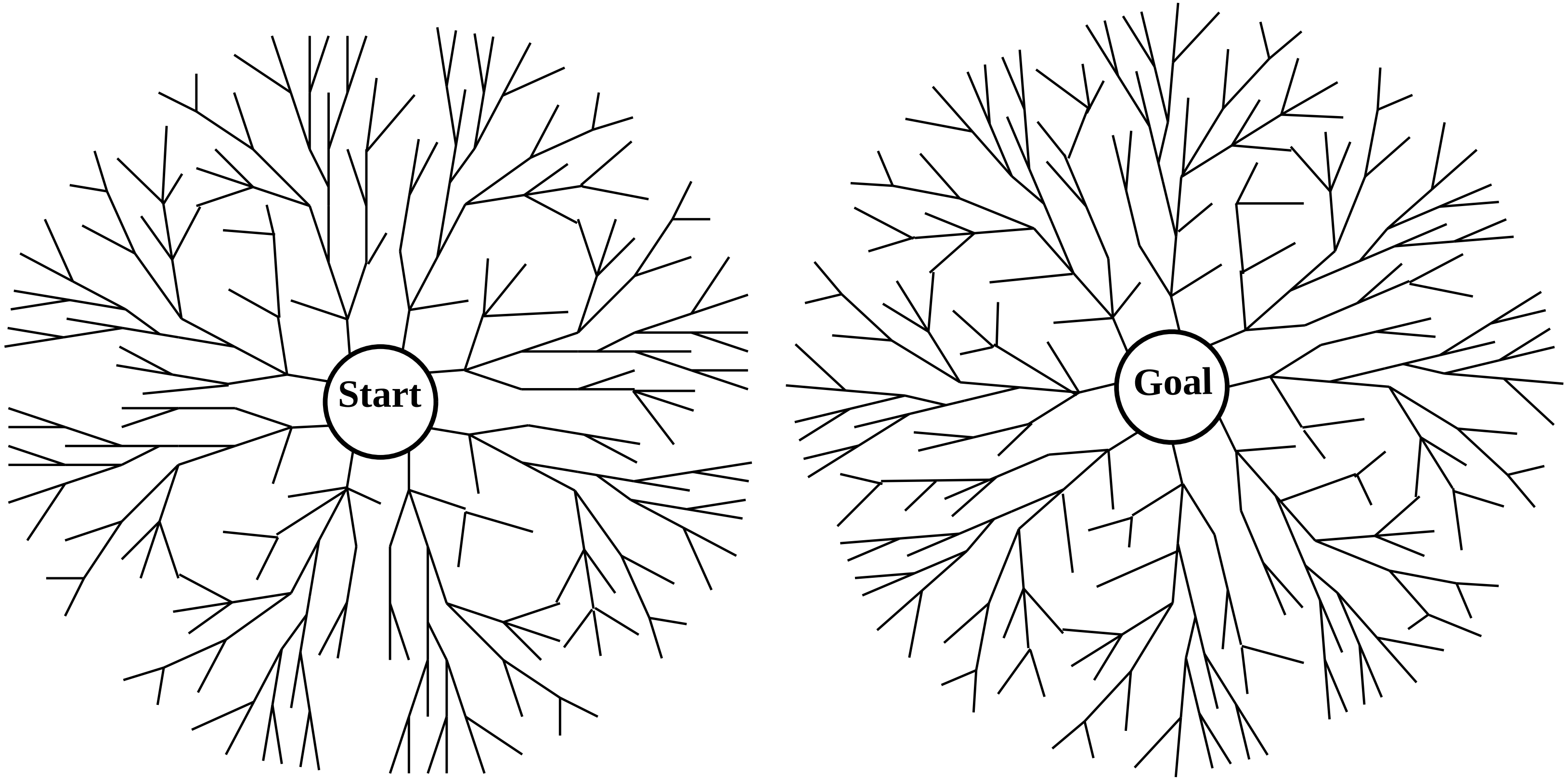
Numeryczne porównanie czasu wykonania dla współcz. rozgałęzienia $b = 10$ przy założeniu, że rozwiązanie w drzewie przeszukiwań ma głębokość $d = 5$ i znajduje się w skrajnie prawym węźle drzewa przeszukiwań:

$$\begin{aligned} N(\mathsf{IDS}) &= 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \\ N(\mathsf{BFS}) &= 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100 \end{aligned}$$

Przeszukiwanie dwukierunkowe

Wykonuje równolegle dwa przeszukiwania:

1) przeszukiwanie wprzód od stanu początkowego
2) przeszukiwanie w tył od stanu końcowego



Przeszukiwanie dwukierunkowe: wlasnosci

Zupełność?? Tak, jeśli oba przeszukiwania wykonywane są wszerz

Złożoność czasowa?? $O(b^{d/2})$

To główna motywacja

Złożoność pamięciowa?? $O(b^{d/2})$

Cena płacona za oszczędność czasu

Optymalność??

Tak, jeśli oba przeszukiwania wykonywane są wszerz
(w grafie z takim samym kosztem wszystkich akcji)
lub jeśli oba przeszukiwania używają strategii jednolitego kosztu
(w grafie z różnym kosztem akcji)

Podsumowanie algorytmow

b — maksymalne rozgałęzienie d — głębokość optymalnego rozwiązania
m — maksymalna głębokość drzewa przeszukiwań (może być ∞)

Kryterium	Wszerz	Jednolity Koszt	Wgłąb	Ograniczone Wgłąb	Iter. Pogłęb.	Dwukie-runkowe
Zupełne?	Tak^a	Tak^{a,b}	Nie	Tak, dla $l \geq d$	Tak^a	Tak^{a,d}
Czas	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d	$b^{d/2}$
Pamięć	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd	$b^{d/2}$
Optymalne?	Tak^c	Tak	Nie	Nie	Tak^c	Tak^d

a) zupełne, jeśli b jest skończone
b) zupełne, jeśli koszt akcji $\geq \epsilon$, dla pewnego $\epsilon > 0$
c) optymalne, jeśli koszt wszystkich akcji jest taki sam
d) zupełne i optymalne, jeśli oba przeszukiwania wszerz lub wg jednol. kosztu

Podsumowanie algorytmow

b — maksymalne rozgałęzienie $\quad$ d — głębokość optymalnego rozwiązania
m — maksymalna głębokość drzewa przeszukiwań (może być ∞)

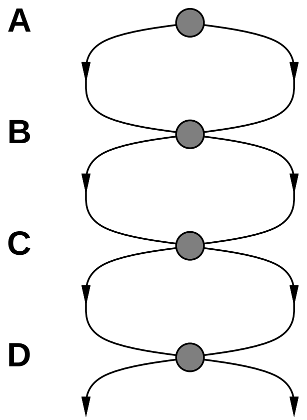
Kryterium	Wszerz	Jednolity Koszt	Wgłąb	Ograniczone Wgłąb	Iter. Pogłęb.	Dwukie-runkowe
Zupełne?	Tak^a	Tak^{a,b}	Nie	Tak, dla $l \geq d$	Tak^a	Tak^{a,d}
Czas	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d	$b^{d/2}$
Pamięć	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd	$b^{d/2}$
Optymalne?	Tak^c	Tak	Nie	Nie	Tak^c	Tak^d

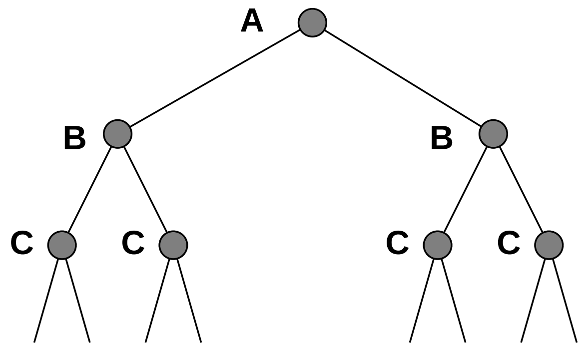
a) zupełne, jeśli b jest skończone
b) zupełne, jeśli koszt akcji $\geq \epsilon$, dla pewnego $\epsilon > 0$
c) optymalne, jeśli koszt wszystkich akcji jest taki sam
d) zupełne i optymalne, jeśli oba przeszukiwania wszerz lub wg jednol. kosztu

Podsumowanie: Iteracyjne pogłębianie używa tylko liniowej pamięci i czasu porównywalnego z innymi algorytmami

Wykrywanie stanow odwiedzonych

Niepełna eliminacja powtarzających się stanów może zamienić problem liniowy w wykładniczy!





Funkcja sprawdzania, czy stan był już odwiedzony, może działać szybko, jeśli zbiór stanów odwiedzonych jest pamiętany i zaimplementowany przy pomocy efektywnej struktury danych, np. kolejki priorytetowej lub tablicy haszującej.

Przeszukiwanie grafu

Jesli algorytm przeszukiwania przestrzeni stanów wykrywa i eliminuje z przeszukiwania stany wcześniej odwiedzone, to taki algorytm jest dobry również do przeszukiwania grafu

Zmienna *closed* pamięta wszystkie wcześniej odwiedzone stany

function GRAPH-SEARCH(*problem, fringe*) **returns** a solution, or failure

    closed ← an empty set
    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*](STATE[*node*]) **then return** *node*
        if STATE[*node*] is not in *closed* **then**
            add STATE[*node*] to *closed*
            fringe ← INSERTALL(EXPAND(*node*, *problem*), *fringe*)
    end

Przeszukiwanie grafu: wlasnosci

Zupełność?? Tak, jeśli graf skończony

Złożoność czasowa?? $\leq$ liczba wierzchołków grafu $\times$ koszt wyszukania stanu

Złożoność pamięciowa?? $\leq$ liczba wierzchołków grafu

Optymalność?? $\Leftrightarrow$ optymalne przy przeszukiwaniu drzewa