Sztuczna Inteligencja
3.1 Metody szukania na ślepo

Włodzisław Duch
Katedra Informatyki Stosowanej UMK
Google: Włodzisław Duch

Szukanie - metoda uniwersalna

Zakładamy, że problem jest zdefiniowany, tzn. jest:

1. Baza danych: fakty, stany, możliwości, opis sytuacji.
2. Możliwe operacje: zmieniają stan bazy danych.

Potrzebna jest strategia kontrolna poszukiwania rozwiązania.

Proces przeszukiwania wygodnie jest przedstawiać za pomocą drzew i grafów.

Droga na grafie od sytuacji startowej do rozwiązania

<=> znalezieniu sekwencji operacji prowadzących do rozwiązania

<=> rozumowaniu.

Ograniczenie: tylko problemy dyskretne, kombinatoryczne.
Ograniczone zastosowanie w przypadku percepcji sygnałów ciągłych,
ale czasami możliwa jest dyskretyzacja.

Procedury szukania

- Szukanie na ślepo - nie mamy żadnej informacji.
- Szukanie heurystycze - potrafimy ocenić postępy.

Na ślepo:

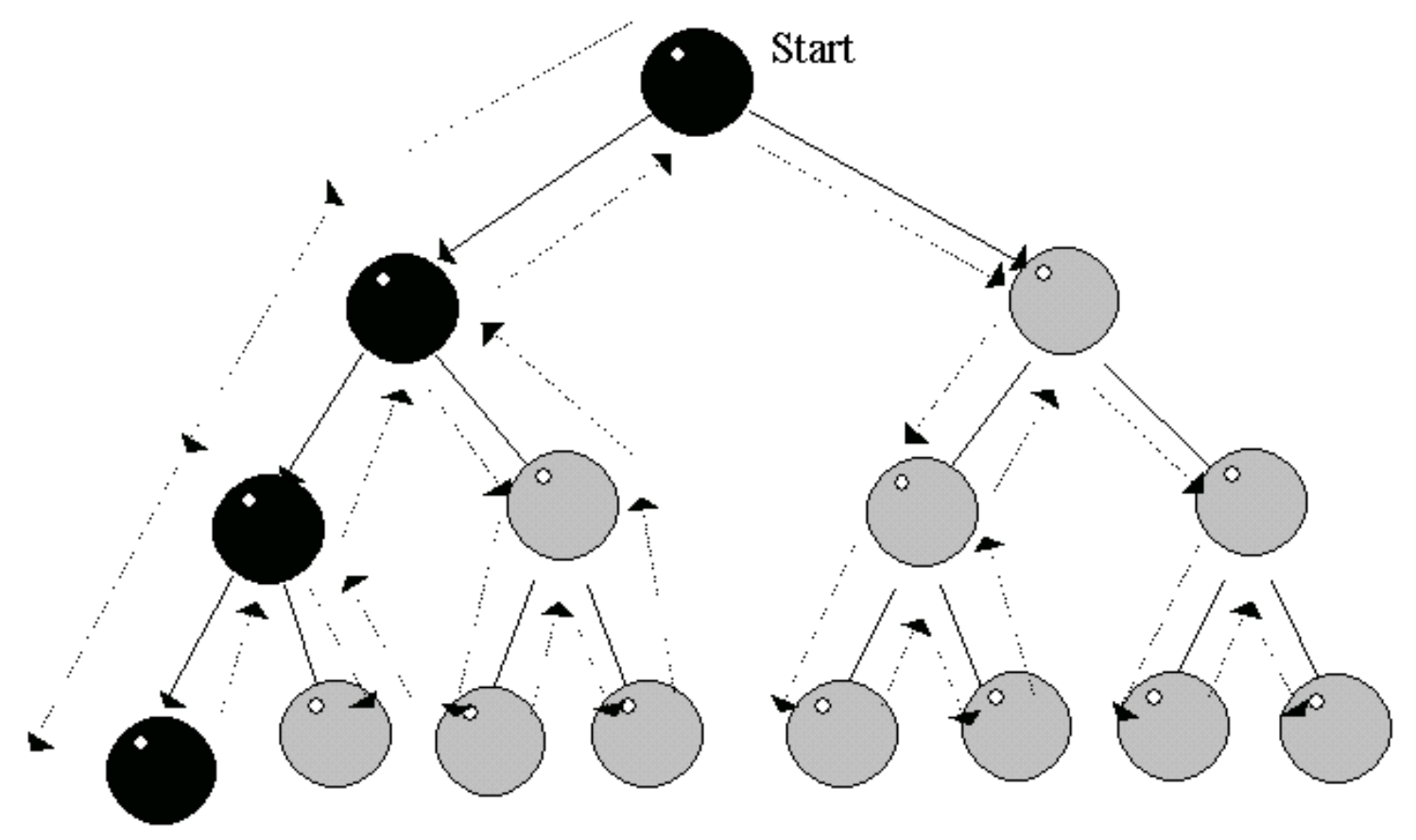
- Monte Carlo, czyli procedura Brytyjskiego Muzeum.

Dla większych problemów jeśli istnieje wiele rozwiązań może coś znaleźć ... i ślepej kurze ...

- Szukanie w głąb.
- Szukanie w szerz, sprawdzając wszystkie możliwości.

Ilustracje: Program PathFinding, PathDemo

W głąb, DFS



Szkic algorytmu Depth First Search (DFS)

0. uporządkuj operatory { $O_1\, O_2 \ldots O_{N-1}\, O_N$ }
   lista początkowa $L(1) = S_w$ (stan wyjściowy); $k=1$, $j(k)=1$
2. Generuj stan $L(k+1) = O_j\, L(k)$; $j(k) = j(k)+1$
3. if ($L(k+1) = S_f$) zakończ;
4. elseif ($L(k+1) = L(m)$, $m=1..k$) (check for loops) $k=k-1$;
5. if ($j(k) > N$) $k-1$; $j=0$

if ($k=0$) brak rozwiązania; else przejdź do 2

5. else $k=k+1$; przejdź do 2

Złożoność: b - średnia liczba możliwych rozgałęzień,
d - średnia głębokość rozwiązań

czas ~ b^d pamięć ~ bd, uwzględniając unikanie pętli pamięć ~ bd^2

Co-Pilot napisze nam taki algorytm w Python.

Wersja rekurencyjna w Pythonie

```
def dfs_recursive(graph, vertex, visited=None):

if visited is None:
    visited = set() # Zbiór przechowujący odwiedzone wierzchołki
    visited.add(vertex) # Oznaczamy bieżący wierzchołek jako odwiedzony
    print(vertex, end=' ') # Wypisujemy wierzchołek (można zmienić akcję)

# Rekurencyjnie odwiedzamy wszystkich nieodwiedzonych sąsiadów

for neighbor in graph[vertex]:
    if neighbor not in visited:
        dfs_recursive(graph, neighbor, visited)
```

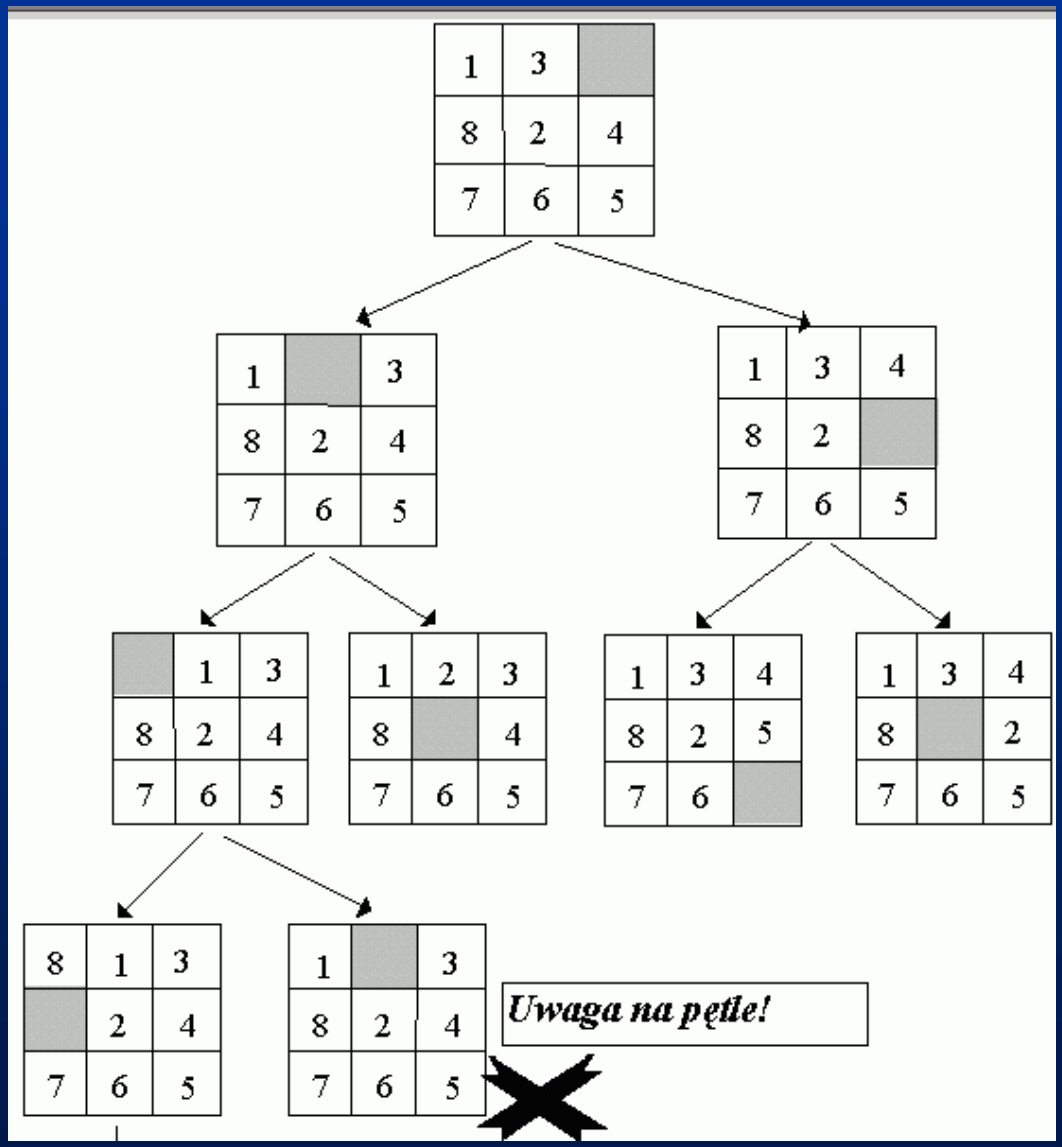
Grok 3 version, DFS algorytm w Python.

Wersja iteracyjna w Pythonie

```
def dfs_iterative(graph, start):
    visited = set()  # Zbiór odwiedzonych wierzchołków
    stack = [start]  # Stos do przechowywania wierzchołków do odwiedzenia
    while stack:
        vertex = stack.pop()  # Pobieramy wierzchołek ze stosu
        if vertex not in visited:
            visited.add(vertex)  # Oznaczamy jako odwiedzony
            print(vertex, end=' ')  # Wypisujemy wierzchołek
 # Dodajemy sąsiadów na stos w odwrotnej kolejności (LIFO)
            for neighbor in reversed(graph[vertex]):
                if neighbor not in visited:
                    stack.append(neighbor)
```

Grok 3 version, DFS algorytm w Python.

Przykład algorytmu DFS dla przesuwanki



Własności i modyfikacja DFS

Zalety: wystarczy niewielka pamięć $O(d)$

dla problemów, które mają wiele rozwiązań może być szybkie.

Wady: nieskończona głębokość, możliwe pętle w grafie,

nie gwarantuje prostoty znalezionego rozwiązania.

Modyfikacja metody DFS:

ograniczaj głębokość szukania do poziomu d.

Złożoność: czas b^d, pamięć bd.

Wada: jeśli d za małe nie znajdzie rozwiązania.

DFS z ograniczeniem głębokości

Zalety: wystarczy niewielka pamięć $O(d)$

dla problemów, które mają wiele rozwiązań może być szybkie

Wady: nieskończona głębokość, możliwe pętle w grafie,

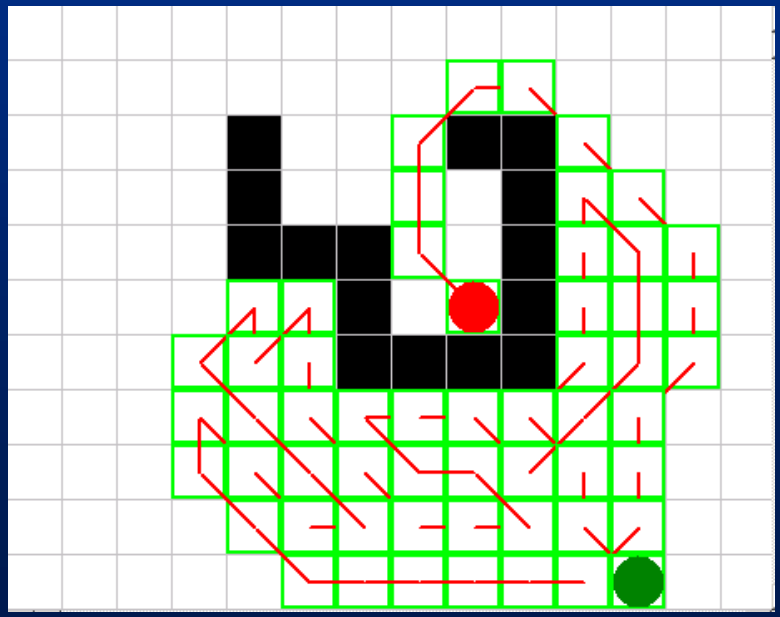
nie gwarantuje prostoty znalezionego rozwiązania.

Modyfikacja metody DF:

ograniczaj głębokość szukania do poziomu d.

Złożoność: czas b^d, pamięć bd.

Wada: jeśli d za małe nie znajdzie rozwiązania.



IDDF, iteracyjne pogłębianie.

- IDDF lub IDS, iteracyjnie pogłębiane szukanie w głąb.
- Próbuj szukać w głąb stopniowo zwiększając głębokość szukania - tani sposób na realizację szukania w głąb.

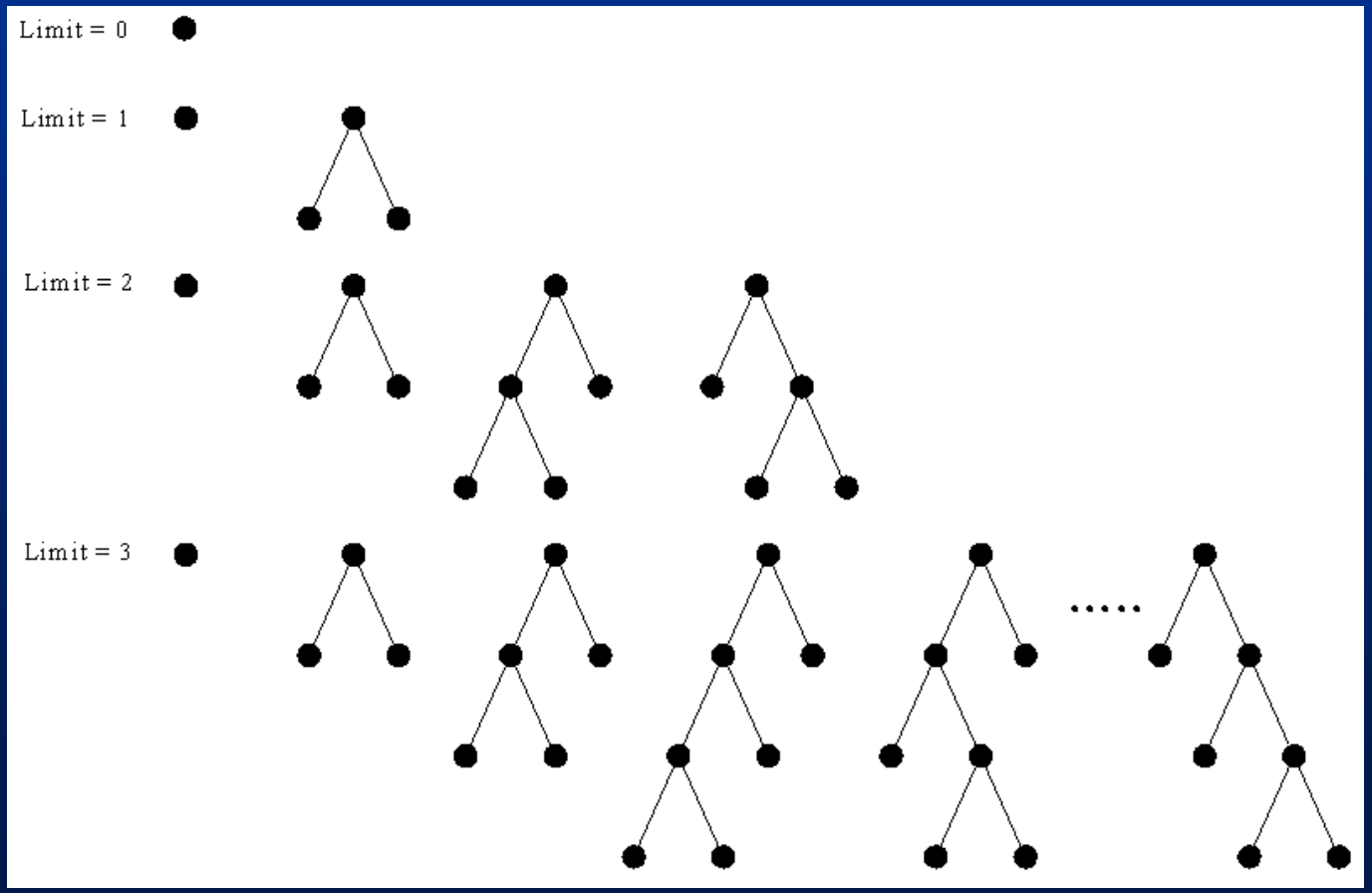
Zalety:

1. Zupełność – zawsze znajduje rozwiązanie.
2. Rozwiązanie jest optymalne (jeśli zwiększamy głębokość o 1)
3. Zużywa tylko bd. elementów pamięci (szukanie w głąb).

Wady: niewielkie zwiększenie kosztów na powtarzanie – ale ostatnie szukanie jest bardziej kosztowne niż wszystkie poprzednie, więc złożoność ~ $O(b^d)$.

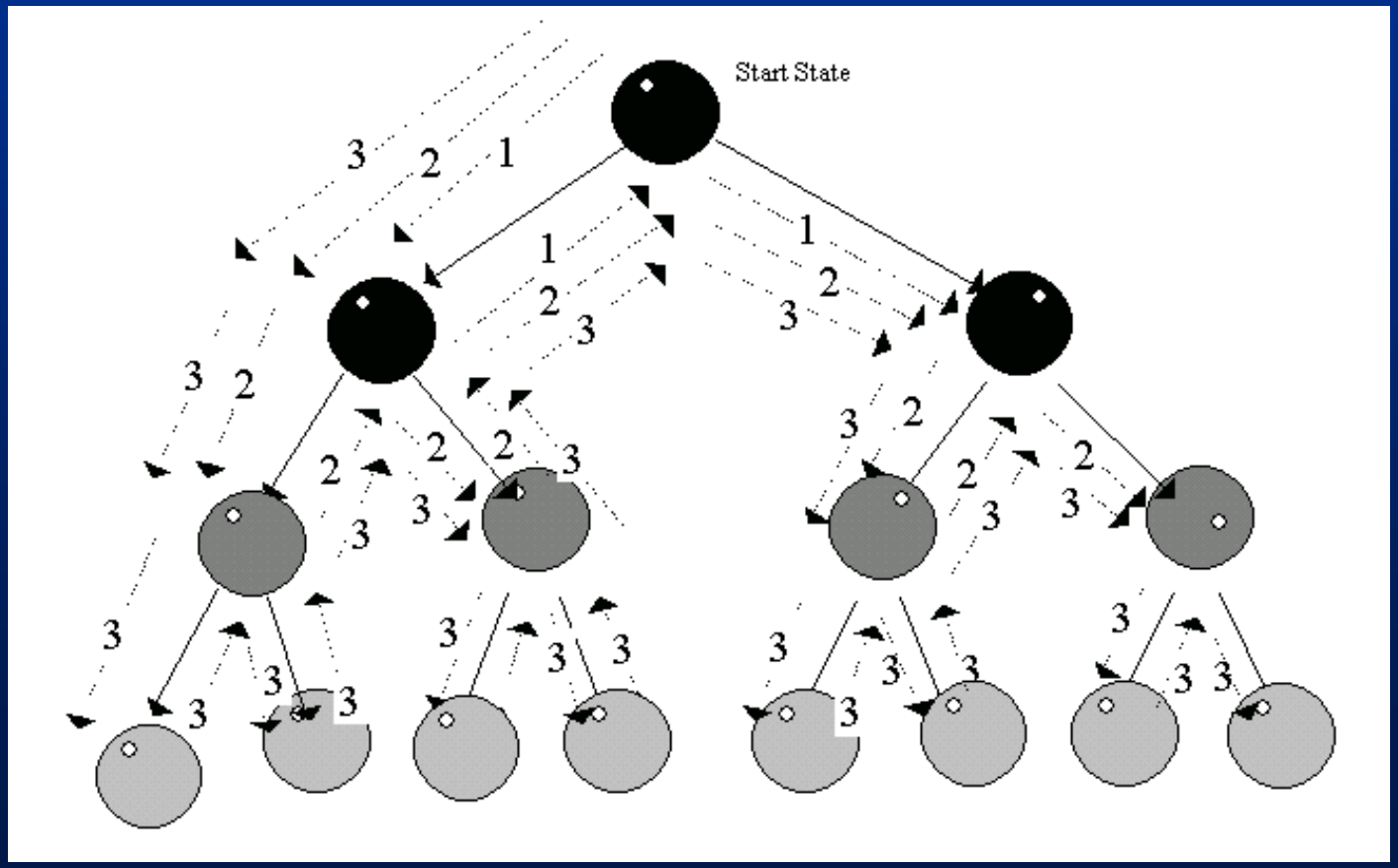
Ilustracja IDDF

- IDDF lub IDS, iteracyjnie pogłębiane szukanie w głąb.
- Próbuj szukać w głąb stopniowo zwiększając głębokość szukania - tani sposób na realizację szukania w głąb.



IDDF, iteracyjne pogłębianie.

- IDDF lub IDS, iteracyjnie pogłębiane szukanie w głąb.
- Próbuj szukać w głąb stopniowo zwiększając głębokość szukania; prosty sposób na realizację szukania w głąb.



Wszerz, BFS

1. lista początkowa $\mathcal{L}(1) = S_w$ (stan wyjściowy)
2. wygeneruj stany $S_{kj} = O_k\, \mathcal{L}(j)$ dla wszystkich **j**
3. if $S_{kj}=S_f$ zakończ;
4. else skopiuj S_{kj} -> $\mathcal{L}()$ i przejdź do 2

Zalety: znajduje najkrótszą listę operatorów jesli szuka do końca.

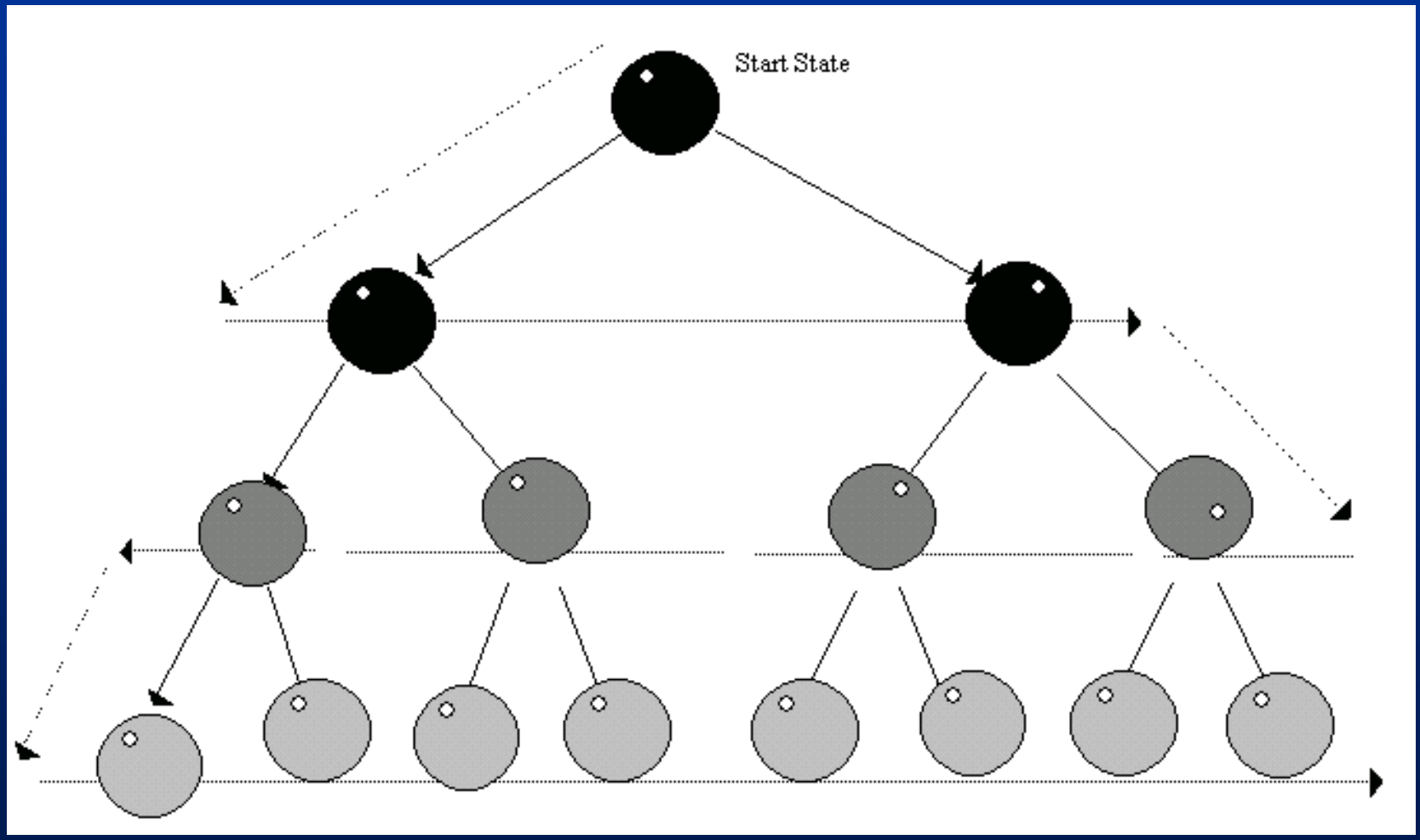
Wady: duża złożoność, przy liczbie operatorów b i kroków d

czas ~ O(b^d),
pamieć ~ O(b^d)

Nawet dla 8-ki trzeba pamiętać aż $(7/3)^{20} \approx 23$ mln stanów.

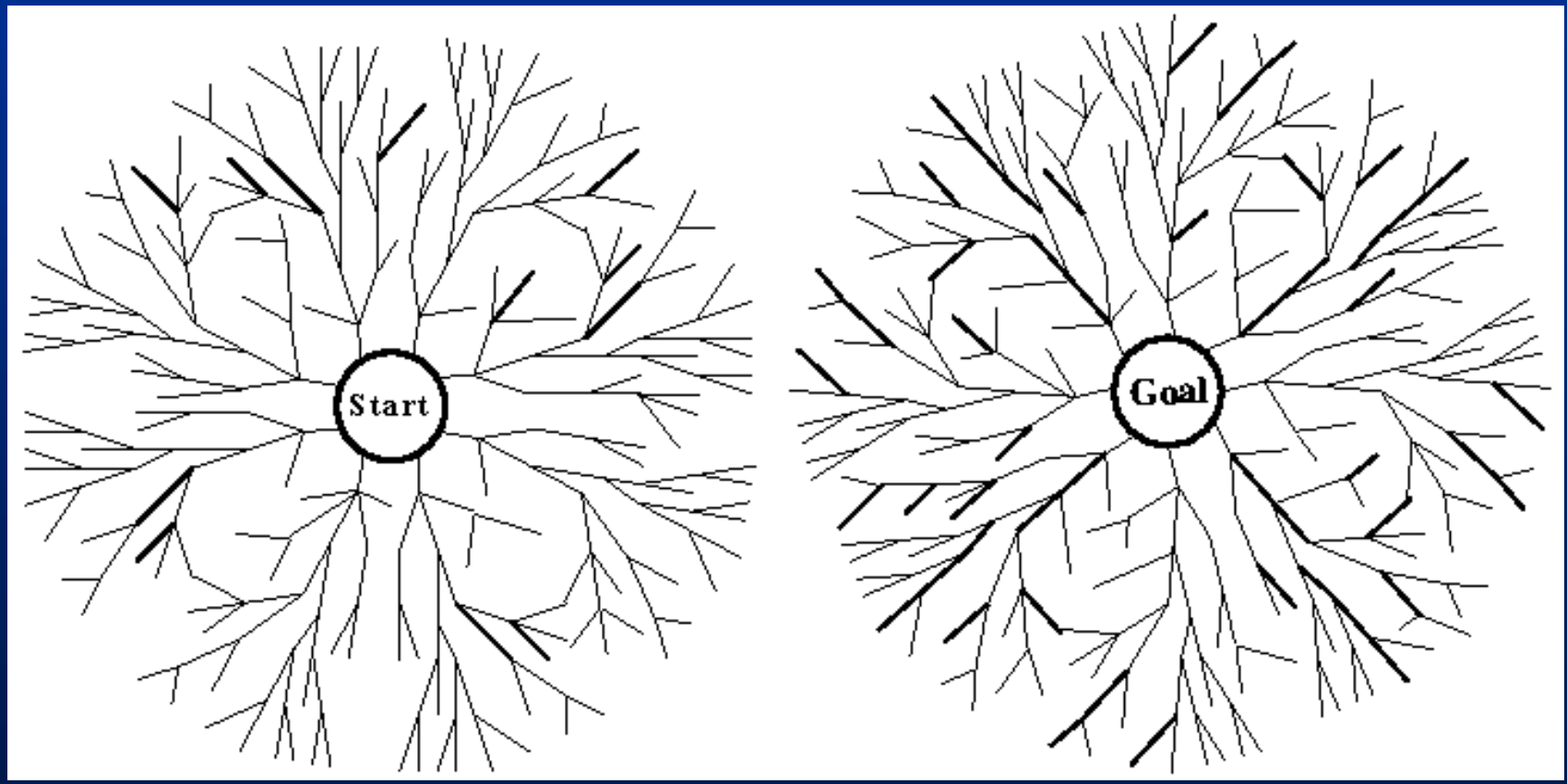
Co-pilot napisze nam ten algorytm w Python.

Drzewo BFS



BDS, szukanie dwukierunkowe.

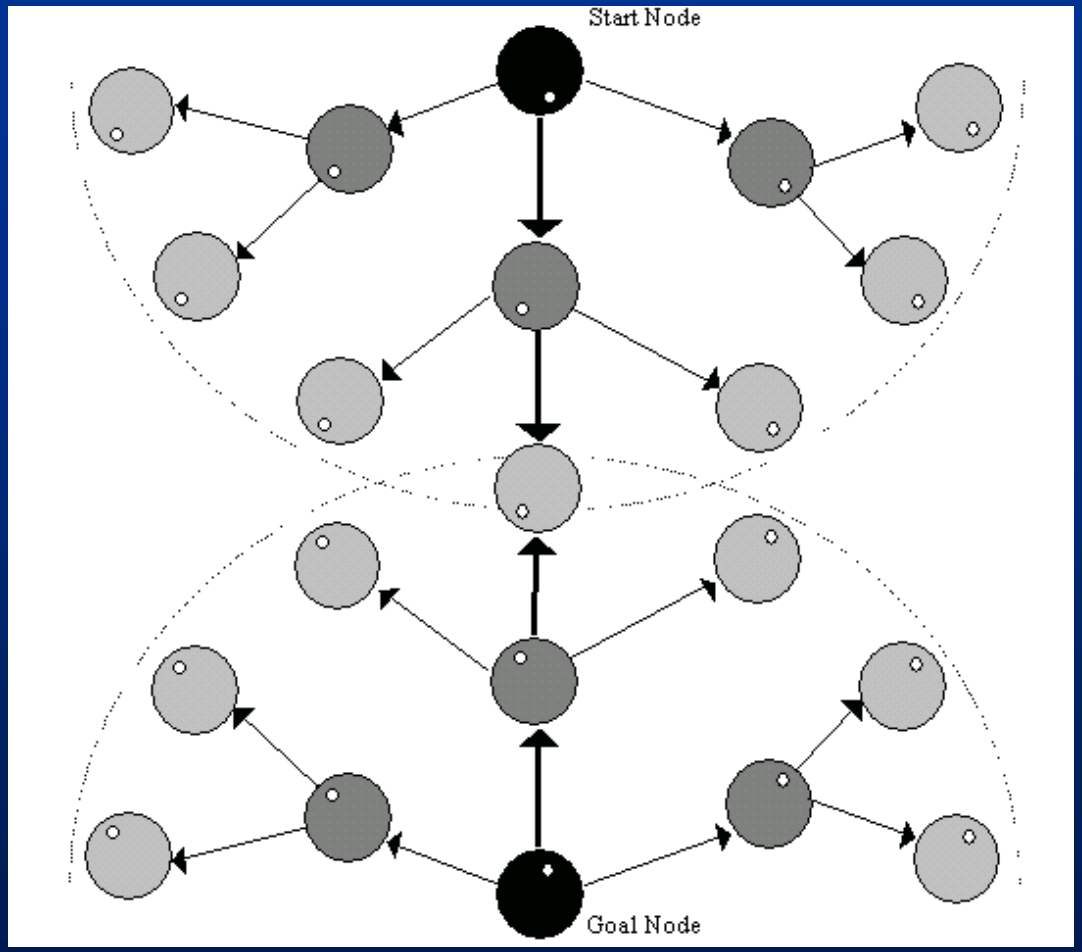
Szukaj wszerz startując od stanu wyjściowego i od stanu końcowego.
Złożoność: czas ~ $O(b^{d/2})$, pamięć ~ $O(b^{d/2})$



BDS, szukanie dwukierunkowe.

Szukaj wszerz startując
od stanu wyjściowego
i od stanu końcowego.

Złożoność:
czas ~ $O(b^{d/2})$,
pamięć ~ $O(b^{d/2})$



PathDemo

W programie używane są funkcje:

- Random Bounce. Po dojściu do przeszkody wykonaj jeden krok w przypadkowym kierunku i kontynuuj swój algorytm.
- Simple Trace. Obejdź przeszkodę aż będziesz mógł kontynuować w tym samym kierunku.
- Robust Trace. Oblicz kierunek pomiędzy blokującym klockiem a klockiem końcowym i obchodź przeszkody aż dojdziesz do kwadratu leżącego w tym kierunku.
- http://aispace.org/search/ przeszukiwanie grafów w Java
- PathFind alternatywa wersja na Github

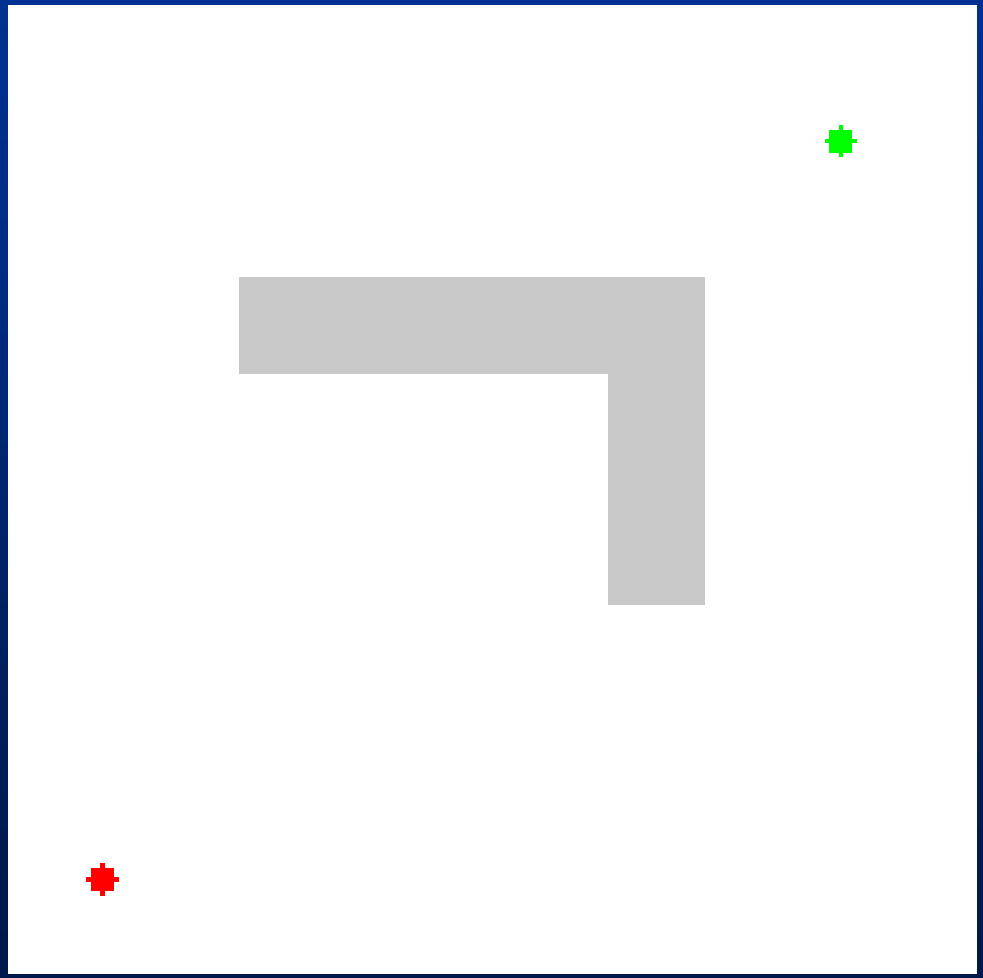
Algorytm Dijkstry

Pierwotnie algorytm najkrótszej drogi w grafie,
buduje drzewo najkrótszych dróg metodą zachłanną.

Znajduje drogi o minimalnej długości z wierzchołka *v* do wszystkich pozostałych, tworzy drzewo rozpinające zawierające najkrótsze drogi z wierzchołka *v*.

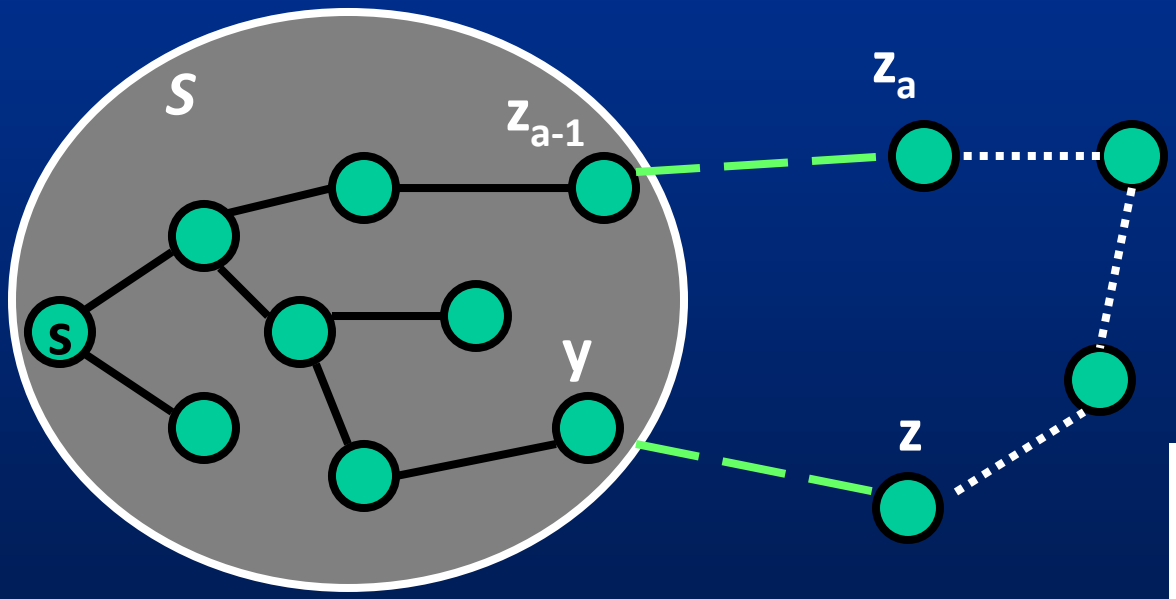
Dla *n* wierzchołków złożoność czasowa jest rzędu $O(n^2)$.

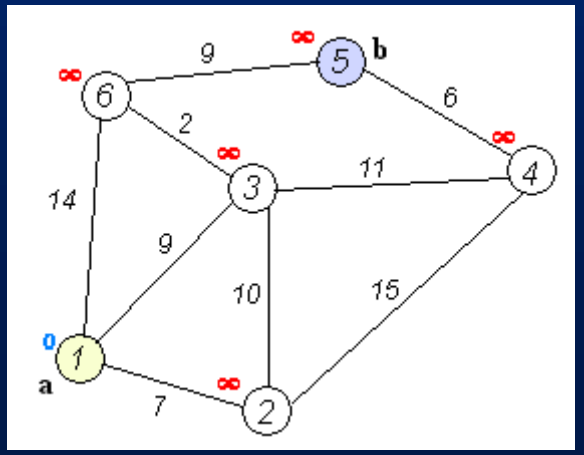
Przykład na geeksforgeeks



Algorytm Dijkstry

Algorytm najkrótszej drogi w grafie buduje drzewo najkrótszych dróg do kolejnych węzłów metodą zachłanną i pamięta najkrótsze.

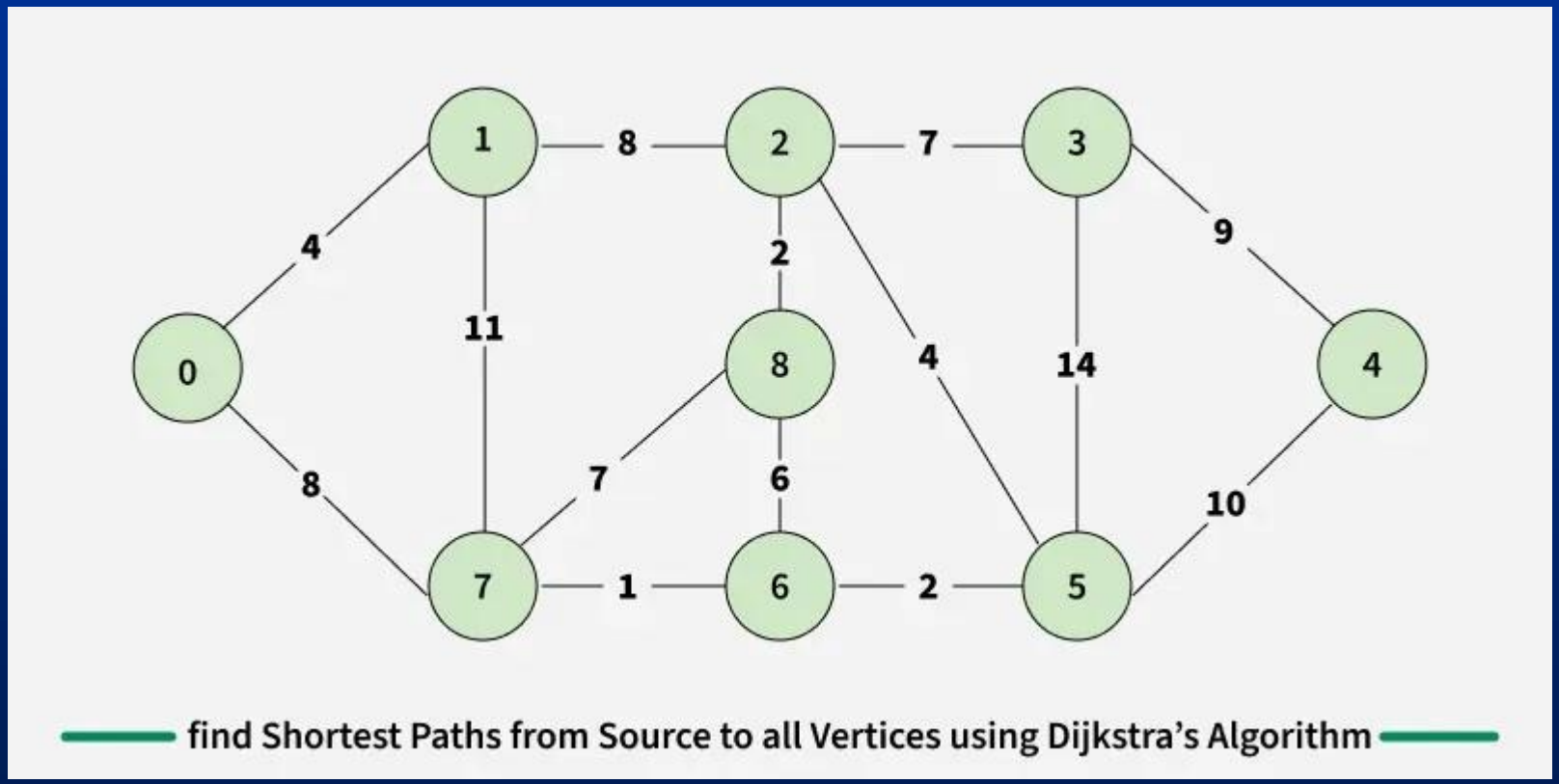




[Animacja algorytmu]

Ilustracja algorytmu Dijkstry

- Implementacja w kilku językach i animacja na stronie Geekforgeeks



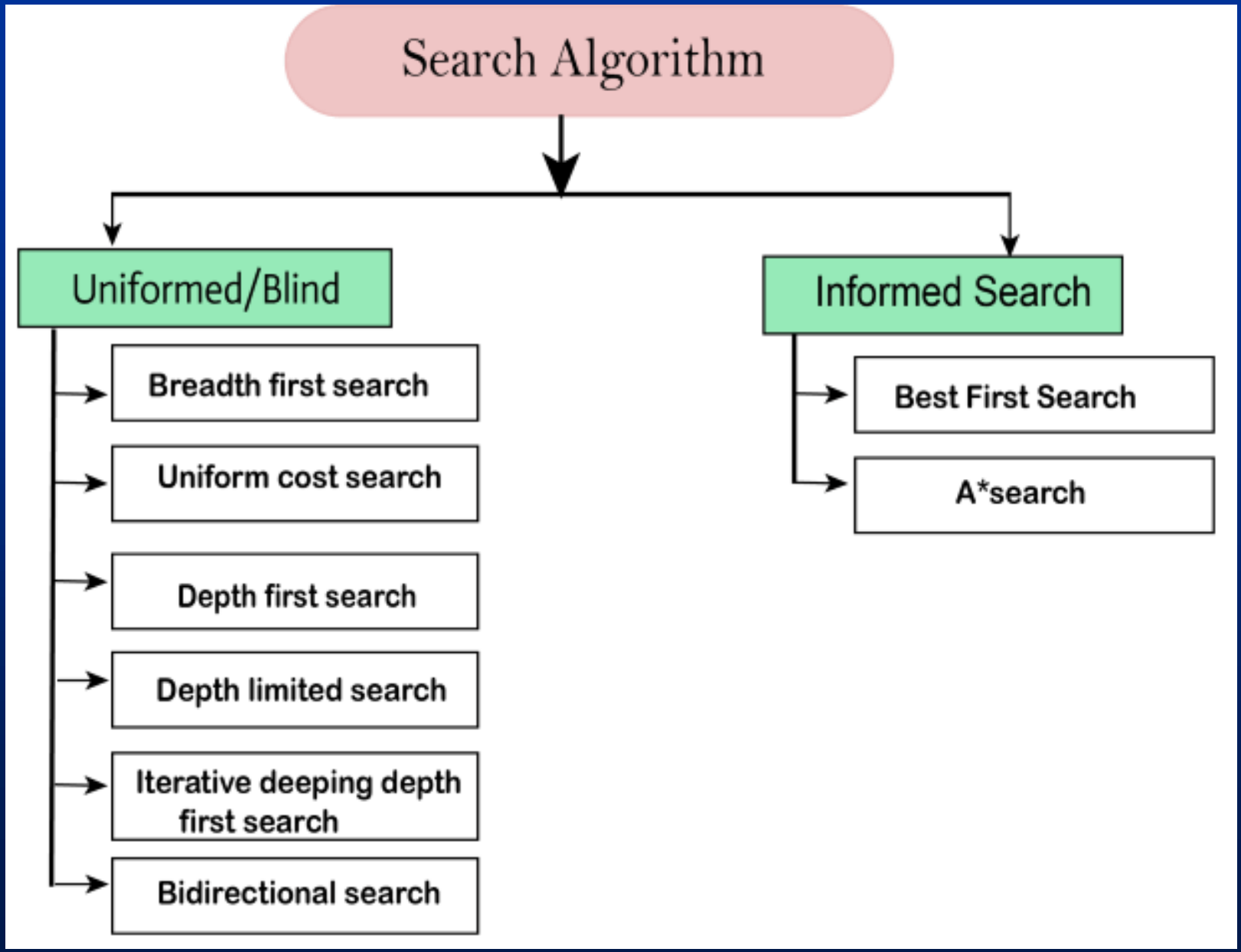
Algorytm Dijkstry, pseudokod

Oznaczenia: U=otwarte, Z=węzły, B=bieżący węzeł.
Drzewo S = {Start}, D(y)=D(y,S), koszt od S; d(y)=d(y,B) koszt od B

1. Budujemy drzewo S, bieżący węzeł D(B=S)=0, pozostałe $D(y \in U)=\infty$
2. Utwórz wszystkie możliwe węzły $y \in U$ połączone z bieżącym węzłem B.
3. Oblicz d(y)+d(B), wybierz D(y) = min(D(y),d(y)+d(B)), droga B<->y najkrótsza
4. Zamknij bieżący węzeł B, usuwając go z U.
5. Węzeł bieżący = następny otwarty węzeł o najniższym koszcie d(B).
6. Powtarzaj 2-5 aż dodanym węzłem będzie Cel lub $U=\varnothing$.

- Drzewo szukania rośnie przez dokładanie węzłów.
- Algorytm gwarantuje znalezienie najkrótszej drogi.

Złożoność dla grafu o *n* wierzchołkach jest rzędu $O(n^2)$
Mniejsza złożoność: implementacja z kolejką priorytetową z kopcem Fibonacciego ~O(n log n+ K) dla K krawędzi drzewa.


Search Algorithm
Uniformed/Blind
Informed Search
Breadth first search
Uniform cost search
Depth first search
Depth limited search
Iterative deeping depth first search
Bidirectional search
Best First Search
A*search

Literatura

- ChatGPT, Co-pilot i inne systemy potrafią napisać algorytmy i wyjaśnić dokładnie ich działania a także ocenić ich złożoność.

- Wiele repozytoriów zaiera opis i demonstracje algorytmów szukania, np..
- geeksforgeeks.org