Algoritmy a programování

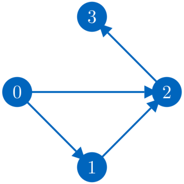
Grafy

Vojtěch Vonásek

Department of Cybernetics
Faculty of Electrical Engineering
Czech Technical University in Prague

Graf $G = (V, E)$

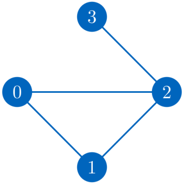
- V jsou uzly (vrcholy/vertices/nodes)
- E je seznam hran $E = \{(i, j) | i, j \in V\}$ (existující hrany)
- Orientovaný graf:
  - hrana (i, j): z i můžeme přejít do j (ale ne naopak)
- Neorientovaný graf:
  - hrana (i, j) umožňuje přechod oběma směry



Orientovaný

Reprezentace grafu

- Matice sousednosti
- Seznam hran
- Seznam sousedních vrcholů



Neorientovaný

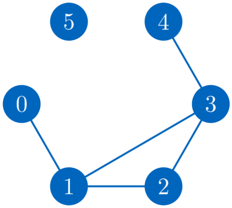
Graf: matice sousednosti

- Čtvercová matice $n \times n$ M, $n = |V|$
- $M_{i,j} = 1$ pokud je hrana $(i, j) \in E$, jinak 0

Neorientovaný graf

```
m=[[0, 1, 0, 0, 0, 0], [1, 0, 1, 1, 0, 0], [0, 1, 0, 1, 0, 0], [0,
    1, 1, 0, 1, 0], [0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 0, 0]]
```

```
0 1 0 0 0 0
1 0 1 1 0 0
0 1 0 1 0 0
0 1 1 0 1 0
0 0 0 1 0 0
0 0 0 0 0 0
```



- Co hrana, to jeden prvek v matici
- Matice je symetrická

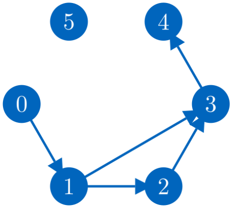
Graf: matice sousednosti

- Čtvercová matice $n \times n$ M, $n = |V|$
- $M_{i,j} = 1$ pokud je hrana $(i, j) \in E$, jinak 0

Orientovaný graf

```
m=[[0, 1, 0, 0, 0, 0], [0, 0, 1, 1, 0, 0], [0, 0, 0, 1, 0, 0], [0,
    0, 0, 0, 1, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]]
```

```
0 1 0 0 0 0
0 0 1 1 0 0
0 0 0 1 0 0
0 0 0 0 1 0
0 0 0 0 0 0
0 0 0 0 0 0
```



- Matice není symetrická

Graf: matice sousednosti

- Čtvercová matice $n \times n$ M, $n = |V|$
- $M_{i,j} = 1$ pokud je hrana $(i, j) \in E$, jinak 0

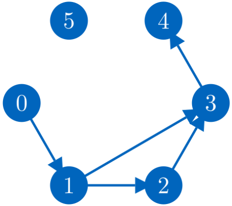
Vlastnosti

- Jednoduché přidání/odebrání hran (změna hodnot M_{ij})
- Přidání/odebrání nového vrcholu do grafu — vyžaduje přidat/smazat řádek a sloupec
- Paměťově náročná ($|V|^2$ buněk)
- Nevhodná pro řídké grafy ($|E| \ll |V|^2$)
- Odchozí hrany z uzlu k: všechny (k, j) kde $M_{k,j} = 1$
- Příchozí hrany do uzlu k: všechny (i, k) kde $M_{i,k} = 1$

Graf: seznam hran

- Pole hran, hrana je například `(i,j)` nebo `[i,j]`
- Jednoduché přidání nových hran $\mathcal{O}(1)$
- Smazaní hrany (i, j) vyžaduje její vyhledání, složitost $\mathcal{O}(n)$
- Nevhodné pro zjištění všech odchozích/příchozích hran uzlu $\mathcal{O}(n)$
- Vhodné pro řídké grafy

```
edges=[(0, 1), (1, 2), (2, 3),
    (1, 3), (3, 4)]
```



Graf: seznam sousedů

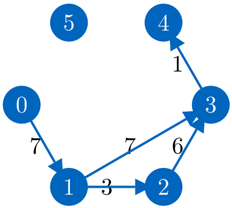
- Graf reprezentujeme polem (nebo dictionary)
- Index je jméno uzlu
- Každá položka je seznam odchozích hran (případně jejich vah)
- V případě nečíselných jmen uzlů je vhodnější použít dictionary

- Pouze sousední vrcholy

```
neighbors={0: [1], 1: [2, 3],
     2: [3], 3: [4]}
```

- Sousední vrcholy + váhy

```
neighbors={0: [[1, 7]], 1:
    [[2, 3], [3, 7]], 2: [[3,
     6]], 3: [[4, 1]]}
```



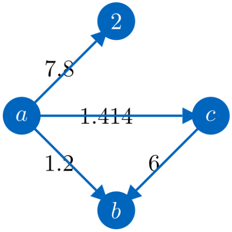
Graf: seznam sousedů

- Výhoda použití dictionary: uzly lze pojmenovávat libovolně

```python
neighbors={'a': [['b', 1.2], ['c', 1.414], [2, 7.8]], 'c': [['b', 6]]}

node = "a"
print("All outgoing from", node)
for edge in neighbors[node]:
    nextNodeName, weight = edge
    print("name:", nextNodeName, "weight:", weight)
```

```
All outgoing from a
name: b weight: 1.2
name: c weight: 1.414
name: 2 weight: 7.8
```



Graf: seznam sousedů

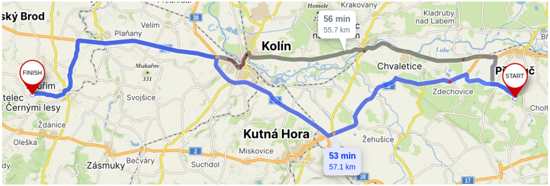
- Seznam sousedů uložený v dictionary
- Libovolná jména uzlů (string, int)
- Přístup do dictionary v $\mathcal{O}(1)$ čase
- Jednoduché získání všech odchozích hran z uzlu
- Jednoduché přidání uzlu nebo hrany
- Složitější odebrání uzlu a hrany

```
neighbors={'a': [['b', 1.2], ['c', 1.414], [2, 7.8]], 'c': [['b', 6]]}

node = "a"
print("All outgoing from", node)
for edge in neighbors[node]:
    nextNodeName, weight = edge
    print("name:", nextNodeName, "weight:", weight)
```

Prohledávání grafů

- Mnoho úloh vyžaduje prohledání grafů, například:
- Najít (jakoukoliv) cestu ze startu do cíle
- Najít optimální cestu ze startu do cíle
- Najít všechny cesty ze startu do cíle
- Existuje (nepřímé) spojení mezi dvěma vrcholy?
- Hledání komponent souvislosti
- atd.



Prohledávání do šířky

- Prohledávání grafu ze startu do cíle
- Využívá fronty
- Stav prohledání reprezentujeme třídou GNode (obsahuje jméno uzlu a jeho rodiče v nalezené cestě)

```python
class GNode:
    def __init__(self, name, parent = None):
        self.name = name
        self.parent = parent
```

Prohledávání do šířky

- Start vložíme do fronty
- Dokud není fronta prázdná:
    - Vezmene prvek z fronty — `actual`
    - Pokud je `actual` cílový uzel, konec
    - Expanze: projdeme všechny sousedy actual a pokud nejsou known, vložíme je do fronty a označíme jako known

Prohledávání do šířky

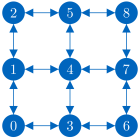
```python
from gnode import *

def bfs(graph, start, goal):
    #graph as list of neighbors
    #start,goal are names of vertices
    queue = [ GNode(start) ]
    known = {}
    known[ start ] = True
    while len(queue) > 0:
        node = queue.pop(0)
        if node.name == goal:
            path = traverse(node)
            return path[::-1]
        if not node.name in graph:
            continue
        for neighbor in graph[node.name]:
            if not neighbor in known:
                known[neighbor] = True
                queue.append(GNode(neighbor, node) )
    return []
```

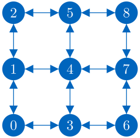
Prohledávání do šířky: rekonstrukce cesty

```
def traverse(node):
    result = []
    while node != None:
        result.append(node.name)
        node = node.parent
    return result
```

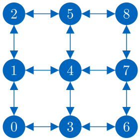
Prohledávání do šířky

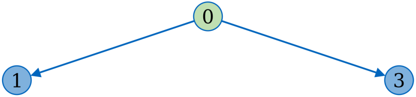


Prohledávání do šířky

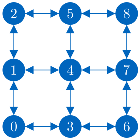


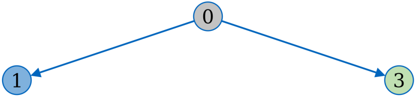
Prohledávání do šířky



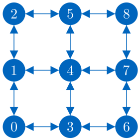


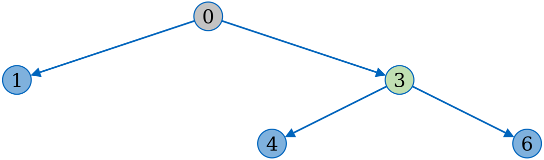
Prohledávání do šířky



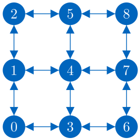


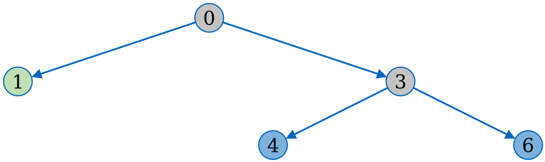
Prohledávání do šířky



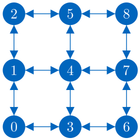


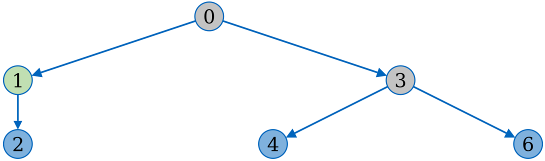
Prohledávání do šířky



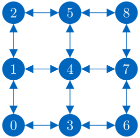


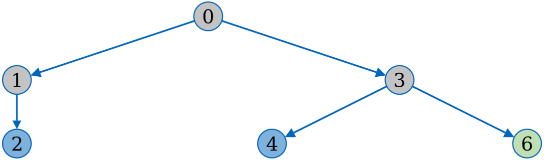
Prohledávání do šířky



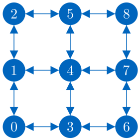


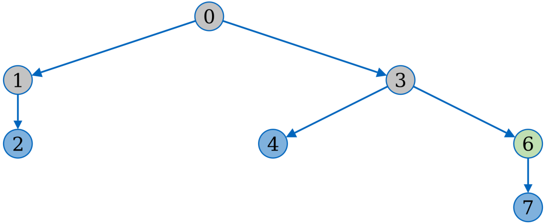
Prohledávání do šířky



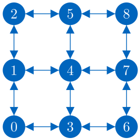


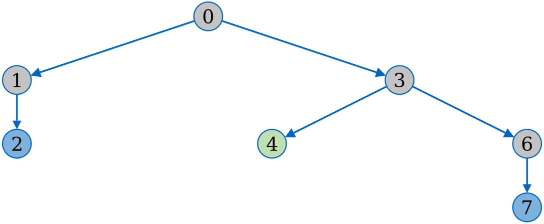
Prohledávání do šířky



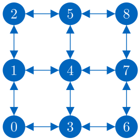


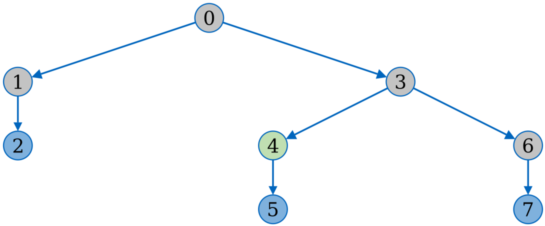
Prohledávání do šířky



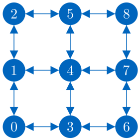


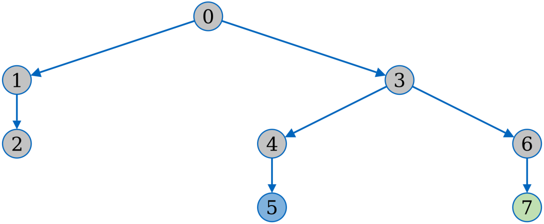
Prohledávání do šířky



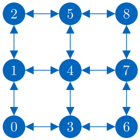


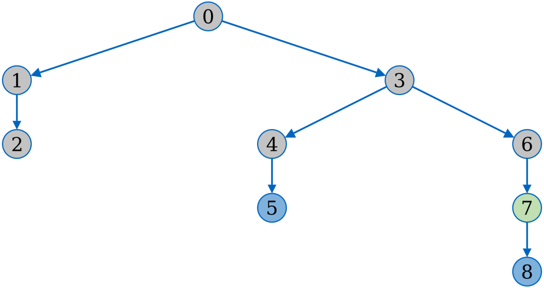
Prohledávání do šířky



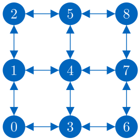


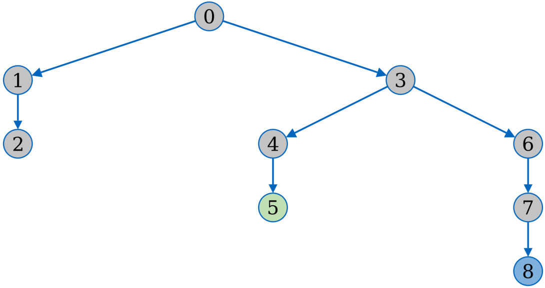
Prohledávání do šířky



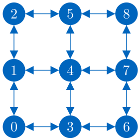


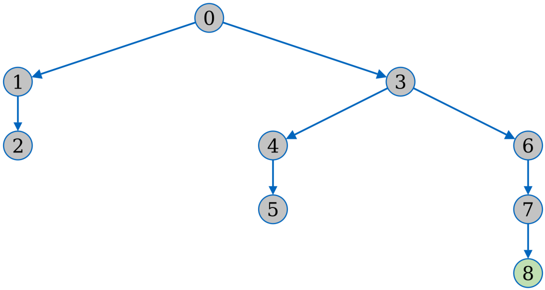
Prohledávání do šířky

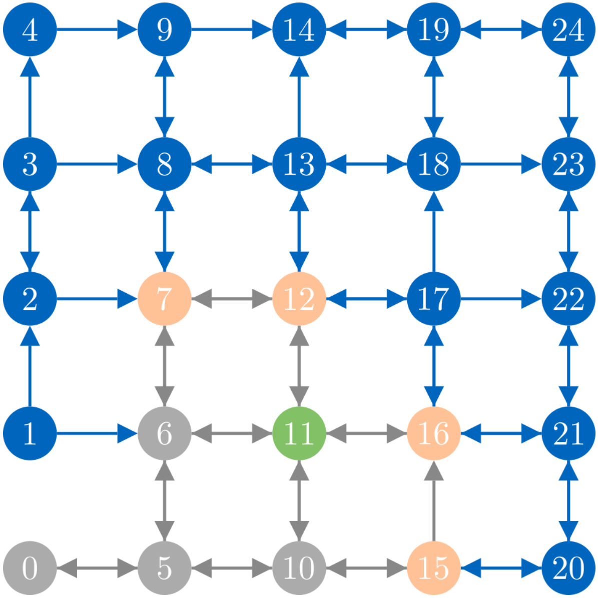




Prohledávání do šířky






4
9
14
19
24
3
8
13
18
23
2
7
12
17
22
1
6
11
16
21
0
5
10
15
20

Prohledávání do šířky

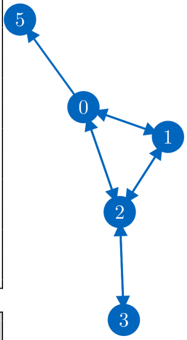
```python
from bfs import bfs

G = {}
G[0] = [1,2,5]
G[1] = [0,2]
G[2] = [0,1,3]
G[3] = [2]

path = bfs(G, 0, 3)
print(path)

path = bfs(G, 5,0)
print(path)
```

```
[0, 2, 3]
[]
```



Prohledávání do šířky

Vlastnosti

- BFS je tzv. complete algoritmus
  - Pokud řešení existuje, tak ho v konečném čase buď najde, nebo reportuje, že neexistuje
  - Předpoklad: graf je konečný
- Časová složitost $\mathcal{O}(|V| + |E|)$, kde $\mathcal{O}(|E|)$ je mezi $\mathcal{O}(1)$ až $\mathcal{O}(|V|^2)$
- Paměťová složitost $\mathcal{O}(|V|)$
- Pokud hledání ukončíme při prvním nalezení cílového stavu, pak řešení obsahuje nejmenší počet hran
- Takové řešení nemusí být nejkratší ve smyslu jiného kritéria (např. délka cesty jako součet vah hran, ...)

Prohledávání do hloubky

- Prohledávání grafu ze startu do cíle
- Využívá zásobníku

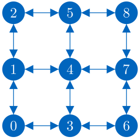
- Start vložíme do zásobníku, označíme ho jako known
- Dokud je něco v zásobníku:
  - Vezmene prvek ze zásobníku — `actual`
  - Pokud `actual` je cílový uzel, konec
  - Expanze: projdeme všechny sousedy `actual` a pokud nejsou known, vložíme je do zásobníku a označíme jako known

Prohledávání do hloubky

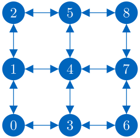
```
from gnode import *

def dfs(graph, start, goal):
    #graph as list of neighbors
    #start,goal are names of vertices
    stack = [ GNode(start) ]
    known = {}
    known[ start ] = True
    while len(queue) > 0:
        node = stack.pop()
        if node.name == goal:
            path = traverse(node)
            return path[::-1]
        if not node.name in graph:
            continue
        for neighbor in graph[node.name]:
            if not neighbor in known:
                known[neighbor] = True
                stack.append(GNode(neighbor, node) )
    return []
```

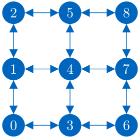
Prohledávání do hloubky

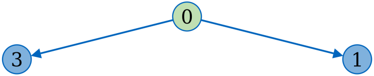



0

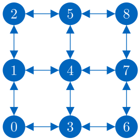

2
5
8
1
4
7
0
3
6

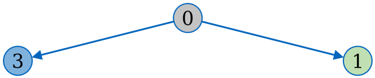
Prohledávání do hloubky



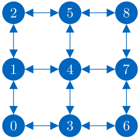


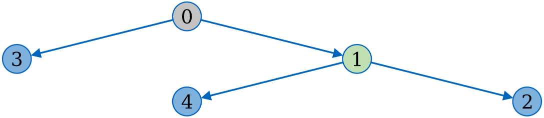
Prohledávání do hloubky



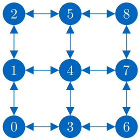


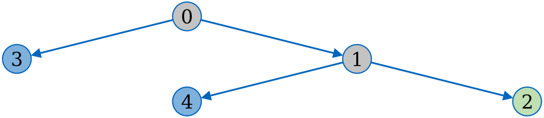
Prohledávání do hloubky



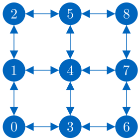


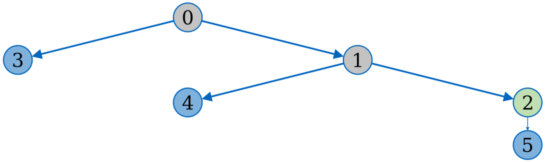
Prohledávání do hloubky



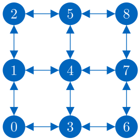


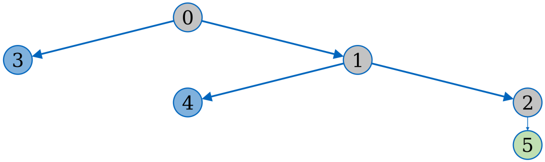
Prohledávání do hloubky



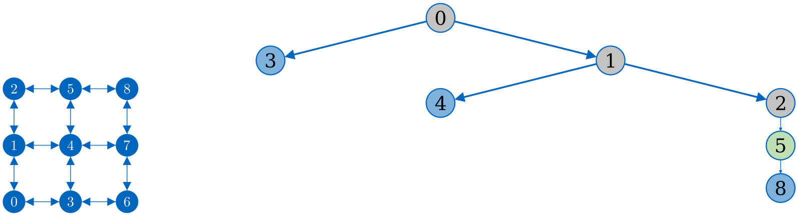


Prohledávání do hloubky

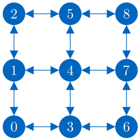


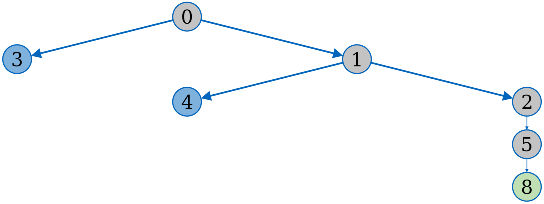


Prohledávání do hloubky

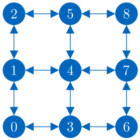


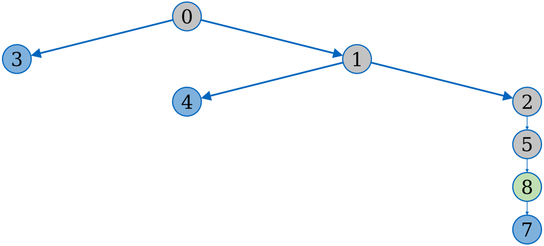
Prohledávání do hloubky



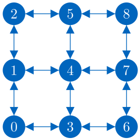


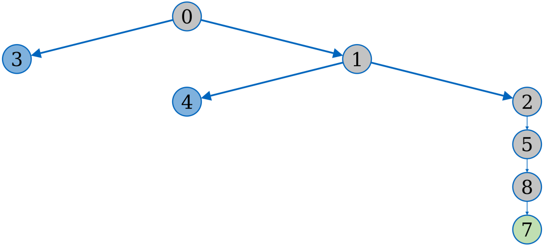
Prohledávání do hloubky

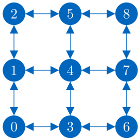


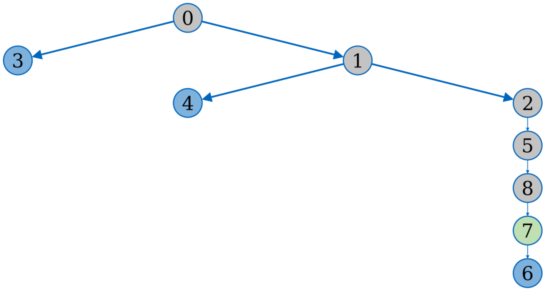


Prohledávání do hloubky

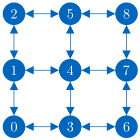


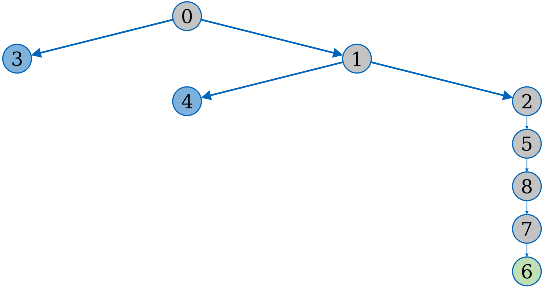



2
5
8
1
4
7
0
3
6

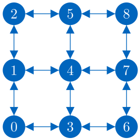

0
3
1
4
2
5
8
7
6

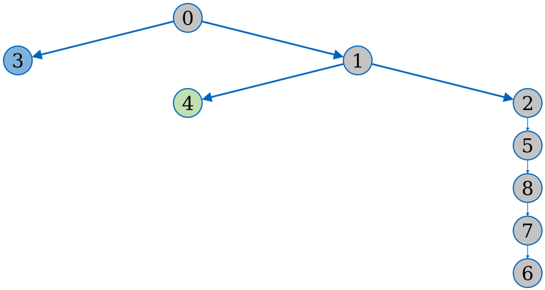
Prohledávání do hloubky



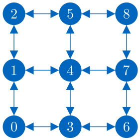


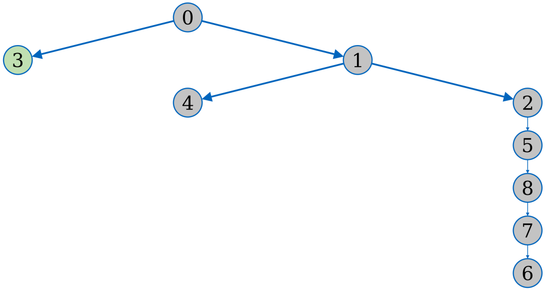
Prohledávání do hloubky





Prohledávání do hloubky





Vlastnosti

- Časová složitost $\mathcal{O}(|V| + |E|)$ (pro konečný graf)
- Paměťová složitost $\mathcal{O}(|V|)$ (data na zásobníku)
- První nalezené řešení negarantuje optimalitu (počet hran, délka cesty, nebo jiné kritérium)
- Pokud si pamatujeme navštívené stavy (known), pak je DFS kompletní
- V některých případech (např. prohledání velkých grafů, implicitně zadaných grafů a stavového prostoru) se pamět known nepoužívá, DFS se pak může zacyklit

Prohledávání do hloubky