Grafy

KIV/ADT – 9. - 10. přednáška

Miloslav Konopík, Libor Váša

26. dubna 2024

Obsah

ADT Graf

Graf

- podchycuje obecný vztah (relaci) mezi prvky
- Strom je speciální druh grafu

(Matematická interpretace pojmu graf, není to graf v excelovském smyslu)

Příklady

Prvek: Město.
Vztah: Města jsou spojená jedním úsekem silnice.

Příklady

Prvek: Město.
Vztah: Města jsou spojená jedním úsekem silnice.

Prvek: Slovo.
Vztah: Význam věty je dán vztahy mezi slovy.

Příklady

Prvek: Město.
Vztah: Města jsou spojená jedním úsekem silnice.

Prvek: Slovo.
Vztah: Význam věty je dán vztahy mezi slovy.

Prvek: Záznam osoby na Facebooku.
Vztah: Osoby jsou přátelé na Facebooku.

Orientovaný graf

Podchycuje situaci, kdy vztah není nutně symetrický.

Orientovaný graf

Podchycuje situaci, kdy vztah není nutně symetrický.

Příklady:
Prvek: Záznam osoby na Facebooku.
Vztah: Osoba požádala druhou osobu o přátelství
(některé mohou být symetrické).

Orientovaný graf

Podchycuje situaci, kdy vztah není nutně symetrický.

Příklady:
Prvek: Záznam osoby na Facebooku.
Vztah: Osoba požádala druhou osobu o přátelství
(některé mohou být symetrické).

Prvek: Popis činnosti.
Vztah: Druhou činnost nelze vykonat předtím, než bude vykonána první činnost
(žádné symetrické).

Formální definice

Neorientovaný graf G je dvojice (V, E):

- V: množina vrcholů (vertex, vertices).
- E: množina hran (edges).
- Hrana je dvouprvková množina $\{a, b\}, a \in V, b \in V$.

Formální definice

Neorientovaný graf G je dvojice (V, E):

- V: množina vrcholů (vertex, vertices).
- E: množina hran (edges).
- Hrana je dvouprvková množina $\{a, b\}, a \in V, b \in V$.

Orientovaný graf G je dvojice (V, E):

- V: množina vrcholů.
- E: množina hran.
- Hrana je uspořádaná dvojice prvků $(a, b), a \in V, b \in V$.

Značení

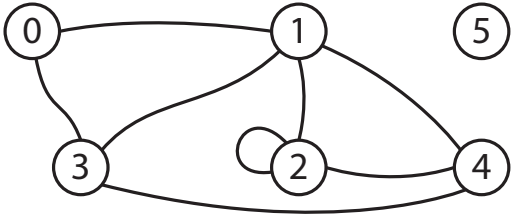
$|V|$ – počet vrcholů grafu
$|E|$ – počet hran grafu

$V(G)$ – množina vrcholů grafu G
$E(G)$ – množina hran grafu G

$y \in V$ je sousedem $x \in V$ právě když

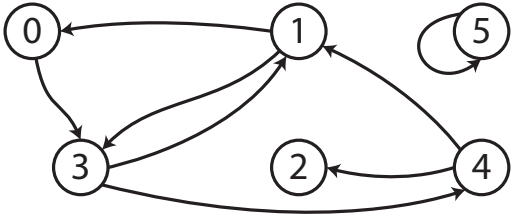
- existuje orientovaná hrana $E = (x, y)$
- existuje neorientovaná hrana $E, x \in E, y \in E$

Příklad neorientovaného grafu



$V = \{0, 1, 2, 3, 4, 5\}, |V| = 6$
$E = \{\{0, 1\}, \{1, 2\}, \{0, 3\}, \{1, 3\}, \{1, 4\}, \{4, 2\}, \{3, 4\}, \{2, 2\}\}, |E| = 8$

Příklad orientovaného grafu



$V = \{0, 1, 2, 3, 4, 5\}, |V| = 6$
$E = \{(1, 0), (0, 3), (1, 3), (3, 1), (3, 4), (4, 1), (4, 2), (5, 5)\}, |E| = 8$

ADT Graf

Operace

- vytvoření grafu s danou množinou vrcholů V (bez hran)
- přidání (ne)orientované hrany
- zjištění všech sousedů vrcholu $x \in V$
- zjištění, zda $y \in V$ je sousedem $x \in V$ (test sousednosti)

S vrcholy nebo hranami mohou být asociována data.

Kvíz I

Rozhraní ADT Graf – I

```
1  class iVertex:
2      def __init__(self, value):
3          self.value = value # hodnota vrcholu
4      @abstractmethod
5      def addNeighbour(self, neighbour: "iVertex") -> None:
6          # přidá souseda
7          pass
8      @abstractmethod
9      def getNeighbours(self) -> set["iVertex"]:
10         # vrátí seznam sousedů
11         pass
```

Rozhraní ADT Graf – II

```
1  class IGraph(ABC):
2      @abstractmethod
3      def addEdge(self, start: iVertex, end: iVertex) -> None:
4          # přidá hranu od startu do konce
5          pass

6      @abstractmethod
7      def getVertices(self) -> set[iVertex]:
8          # vrátí množinu vrcholů
9          pass
```

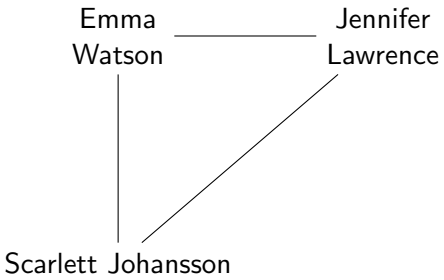
Použití ADT graf

```
1  # Vytvoříme vrcholy s názvy hereček
2  Emma = AdjListVertex("Emma Watson")
3  Jennifer = AdjListVertex("Jennifer Lawrence")
4  Scarlett = AdjListVertex("Scarlett Johansson")

5  graph = UndirectedGraph()

6  # přidáme hrany mezi vrcholy, které reprezentují
   ↪ přátelství
7  graph.addEdge(Emma, Jennifer)
8  graph.addEdge(Jennifer, Scarlett)
9  graph.addEdge(Scarlett, Emma)

10 print(Emma.getNeighbours())      # [Jennifer,
   ↪ Scarlett]
11 print(Jennifer.getNeighbours()) # [Emma, Scarlett]
12 print(Scarlett.getNeighbours()) # [Emma, Jennifer]
```



Implementace ADT Graf

Dvě možnosti:

- seznamy sousednosti
- matice sousednosti

- různé vlastnosti v závislosti na vlastnostech grafu
- implementace se liší pro orientované a neorientované grafy
- detailněji vysvětleno v KIV/IDT.

Jednoduchá implementace seznamem sousednosti – vrchol

```
1  class AdjListVertex(iVertex):
2      def __init__(self, value):
3          super().__init__(value)
4          self.neighbours = set()

5      def __repr__(self):
6          return f"Vertex({self.value}, # of neighbours: {len(self.neighbours)})"

7      def addNeighbour(self, neighbour: iVertex) -> None:
8          self.neighbours.add(neighbour)

9      def getNeighbours(self) -> set[iVertex]:
10         return self.neighbours

11     def isNeighbour(self, other: iVertex) -> bool:
12         return other in self.neighbours
```

Jednoduchá implementace seznamem sousednosti – graf

```
1  class UndirectedGraph(IGraph):
2      def __init__(self) -> None:
3          self.vertices: set[iVertex] = set() # množina vrcholů

4      def addEdge(self, start: iVertex, end: iVertex) -> None:
5          # přidá startovní a koncový vrchol do grafu, pokud tam ještě nejsou
6          if start not in self.vertices:
7              self.vertices.add(start)
8          if end not in self.vertices:
9              self.vertices.add(end)

10         # přidá koncový vrchol do seznamu sousednosti startovního vrcholu
11         start.addNeighbour(end)
12         end.addNeighbour(start)

13     def getVertices(self) -> set[iVertex]:
14         return self.vertices
```

Složitost operací

Sousedé jsou uloženy v množině.

Přidání hrany: $\Theta(1)$

Zjištění sousedů: $\Theta(1)$

Test sousednosti: $\Theta(1)$

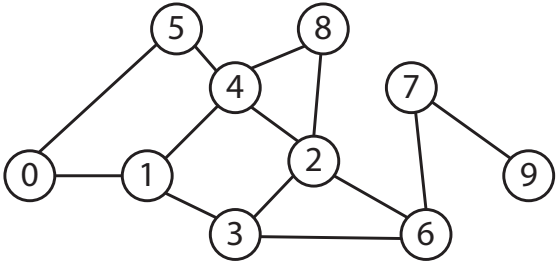
Procházení grafu

Procházení grafu

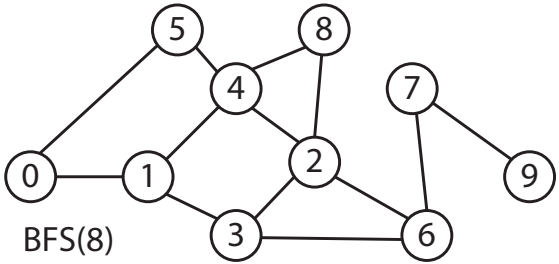
Procházení do šířky : prohledává graf po vrstvách, nejprve se prohledávají všechny vrcholy ve vzdálenosti 1, pak ve vzdálenosti 2, atd.

Procházení do hloubky : lze definovat rekurzivně, pokud data nejsou v prvním uzlu, rekurzivně zavolám hledání na graf bez tohoto uzlu. Prohledává graf tak, že zpracuje první sousedy všech uzlů a pak postupně další.

Příklad procházení grafu do šířky



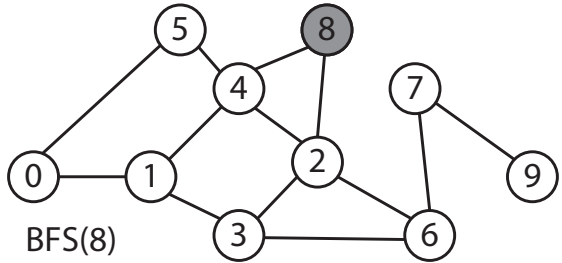
Příklad procházení grafu do šířky



BFS(8)

fronta:

Příklad procházení grafu do šířky

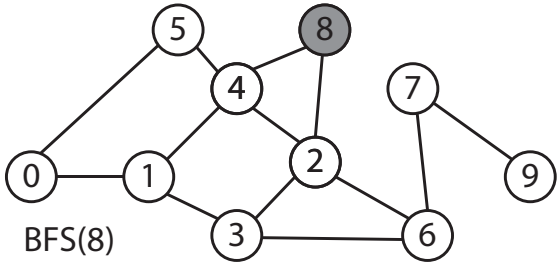


BFS(8)

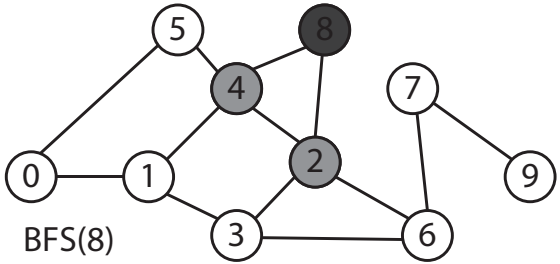
fronta:

8

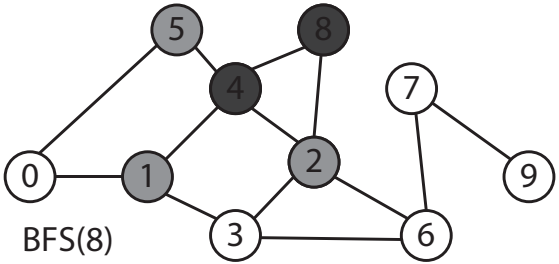
Příklad procházení grafu do šířky



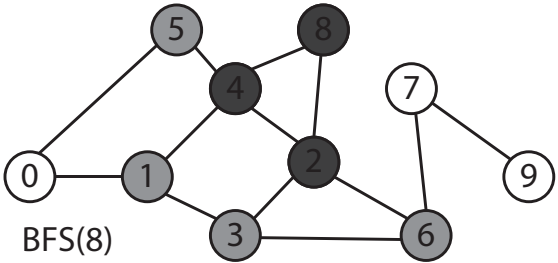
Příklad procházení grafu do šířky



Příklad procházení grafu do šířky



Příklad procházení grafu do šířky

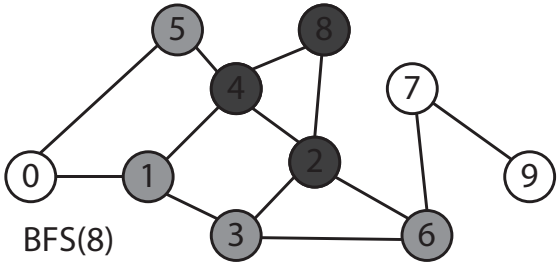


BFS(8)

fronta:

2	5	1	3	6			

Příklad procházení grafu do šířky

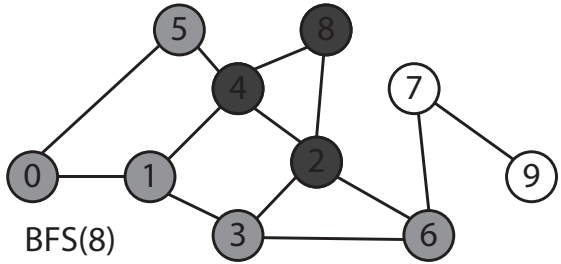


BFS(8)

fronta:

5	1	3	6	0		

Příklad procházení grafu do šířky

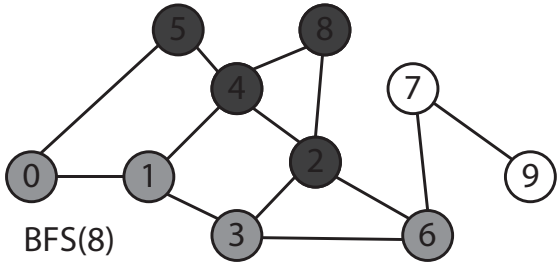


BFS(8)

fronta:

5	1	3	6	0		

Příklad procházení grafu do šířky

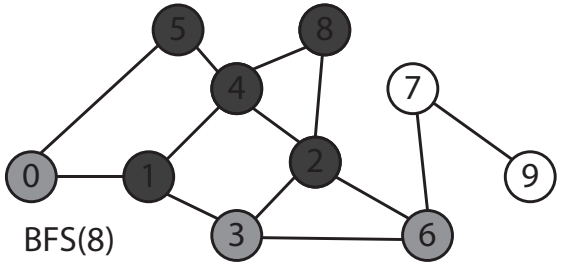


BFS(8)

fronta:

1	3	6	0		

Příklad procházení grafu do šířky

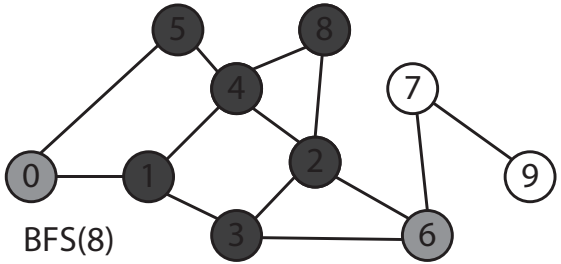


BFS(8)

fronta:

3	6	0		

Příklad procházení grafu do šířky

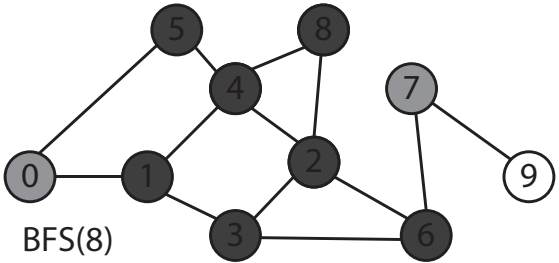


BFS(8)

fronta:

6	0	7	

Příklad procházení grafu do šířky

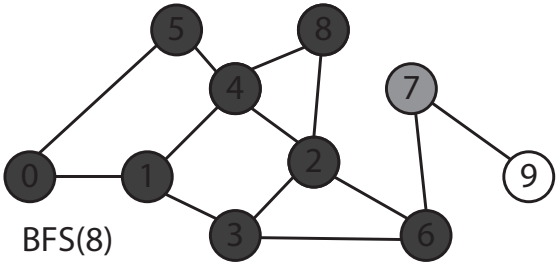


BFS(8)

fronta:

0	7	

Příklad procházení grafu do šířky

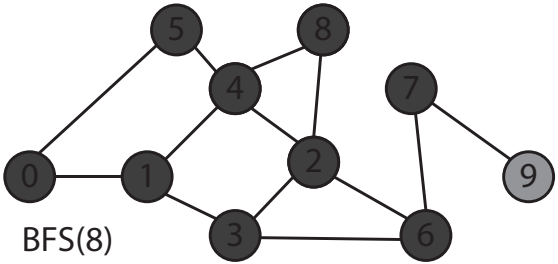


BFS(8)

fronta:

7	9

Příklad procházení grafu do šířky

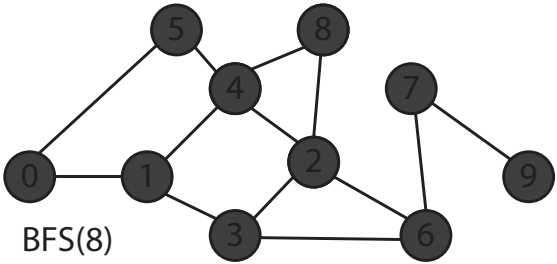


BFS(8)

fronta:

9

Příklad procházení grafu do šířky



BFS(8)

fronta:

Hledání cesty

Prohledávání grafu do šířky

```
1  def bfs(graph: IGraph, start: iVertex, end: iVertex) -> list[iVertex]:
2      # provede prohledávání do šířky od zadaného vrcholu a vrátí cestu k cílovému vrcholu,
       ↪ pokud existuje
3      queue = deque([(start, [start])])  # fronta pro navštívené vrcholy a jejich cesty
4      visited = set()  # množina pro označení navštívených vrcholů
5      while queue:  # dokud je fronta neprázdná
6          current, path = queue.popleft()  # vezmi vrchol a jeho cestu z fronty
7          if current == end:  # pokud je to cílový vrchol
8              return path  # vrať cestu
9          if current not in visited:  # pokud ještě nebyl navštíven
10             visited.add(current)    # označ ho jako navštívený
11             for neighbour in  current.getNeighbours():  # pro každého souseda
12                 queue.append((neighbour, path + [neighbour]))  # přidej ho do fronty
                   ↪ spolu s jeho cestou
13     return []  # nenalezena žádná cesta
```

Prohledávání grafu do hloubky

```
1  def dfs(graph: IGraph, start: Vertex, end: Vertex) -> List[Vertex]:
2      # provede prohledávání do hloubky od zadaného vrcholu a vrátí cestu k cílovému
       ↪ vrcholu, pokud existuje
3      stack = [(start, [start])]  # zásobník pro navštívené vrcholy a jejich cesty
4      visited = set()  # množina pro označení navštívených vrcholů
5      while stack:  # dokud je zásobník neprázdný
6          current, path = stack.pop()  # vezmi vrchol a jeho cestu ze zásobníku
7          if current == end:  # pokud je to cílový vrchol
8              return path  # vrať cestu
9          if current not in visited:  # pokud ještě nebyl navštíven
10             visited.add(current)    # označ ho jako navštívený
11             for neighbour in graph.neighbours(current):  # pro každého souseda
12                 stack.append((neighbour, path + [neighbour]))  # přidej ho na zásobník
                   ↪ spolu s jeho cestou
13     return []  # nenalezena žádná cesta
```

Kvíz II

Ohodnocené grafy

Ohodnocené grafy

Ohodnocený graf G je trojice (V, E, w):

- V: množina vrcholů.
- E: množina hran.
- $w : E \to R$: reálná funkce přiřazující každé hraně její ohodnocení (weight).
  - Orientovaný (hrana je dvouprvková množina).
  - Neorientovaný (hrana je uspořádaná dvojice).

Příklady:

Ohodnocené grafy

Ohodnocený graf G je trojice (V, E, w):

- V: množina vrcholů.
- E: množina hran.
- $w : E \to R$: reálná funkce přiřazující každé hraně její ohodnocení (weight).
  - Orientovaný (hrana je dvouprvková množina).
  - Neorientovaný (hrana je uspořádaná dvojice).

Příklady:

Neorientovaný ohodnocený graf:

$$G = (\{a, b, c, d\}, \{\{a, b\}, \{b, c\}, \{c, d\}, \{d, a\}\}, w)$$

kde $w(\{a, b\}) = 2$, $w(\{b, c\}) = 3$, $w(\{c, d\}) = 4$, $w(\{d, a\}) = 5$.

Ohodnocené grafy

Ohodnocený graf G je trojice (V, E, w):

- V: množina vrcholů.
- E: množina hran.
- $w : E \to R$: reálná funkce přiřazující každé hraně její ohodnocení (weight).
  - Orientovaný (hrana je dvouprvková množina).
  - Neorientovaný (hrana je uspořádaná dvojice).

Příklady:

Orientovaný ohodnocený graf:

$$G = (\{a, b, c, d\}, \{(a, b), (b, c), (c, d), (d, a)\}, w)$$

kde $w((a, b)) = 2$, $w((b, c)) = 3$, $w((c, d)) = 4$, $w((d, a)) = 5$.

Nejkratší cesta v ohodnoceném grafu

Velmi častý problém

- ohodnocení: vzdálenost, čas, ...

Nejkratší cesta v ohodnoceném grafu

Velmi častý problém

- ohodnocení: vzdálenost, čas, ...

Úkol: nalézt nejkratší vzdálenost ke všem vrcholům

Nejkratší cesta v ohodnoceném grafu

Velmi častý problém

- ohodnocení: vzdálenost, čas, ...

Úkol: nalézt nejkratší vzdálenost ke všem vrcholům

Ostatní úkoly se dají vyřešit analogicky

- nejkratší cesta ke konkrétnímu vrcholu
- všechny vrcholy až do vzdálenosti X

Nejkratší cesta v ohodnoceném grafu

Velmi častý problém

- ohodnocení: vzdálenost, čas, ...

Úkol: nalézt nejkratší vzdálenost ke všem vrcholům

Ostatní úkoly se dají vyřešit analogicky

- nejkratší cesta ke konkrétnímu vrcholu
- všechny vrcholy až do vzdálenosti X

Otázka

Dá se použít BFS?

Nejkratší cesta v ohodnoceném grafu

Velmi častý problém

- ohodnocení: vzdálenost, čas, ...

Úkol: nalézt nejkratší vzdálenost ke všem vrcholům

Ostatní úkoly se dají vyřešit analogicky

- nejkratší cesta ke konkrétnímu vrcholu
- všechny vrcholy až do vzdálenosti X

Otázka

Dá se použít BFS?

Odpověď

Ne zcela.

Rozhranní ohodnocených grafů

```python
class IWGraph(metaclass=ABCMeta):
    @abstractmethod
    def addEdge(self, start: Vertex, end: Vertex, weight: int) -> None:
        # přidá startovní a koncový vrchol do grafu se zadanou váhou
        pass

    @abstractmethod
    def neighbours(self, vertex: Vertex) -> dict[Vertex, int]:
        # vrátí sousedy daného vrcholu
        pass

    @abstractmethod
    def isNeighbour(self, v1: Vertex, v2: Vertex) -> bool:
        # vrátí True, pokud je v2 v seznamu sousednosti v1
        pass
```

BFS v ohodnoceném grafu

```
1  def bfs(graph: IWGraph, start: Vertex, end: Vertex) -> list[Vertex]:
2      # provede prohledávání do šířky od zadaného vrcholu a vrátí cestu k cílovému vrcholu
       ↪ s nejmenší váhou, pokud existuje
3      queue = deque([(start, [start], 0)]) # fronta pro navštívené vrcholy, jejich cesty a
       ↪ jejich váhy
4      visited = set() # množina pro označení navštívených vrcholů
5      while queue: # dokud je fronta neprázdná
6          current, path, weight = queue.popleft() # vezmi vrchol, jeho cestu a jeho váhu z
           ↪ fronty
7          if current == end: # pokud je to cílový vrchol
8              return path # vrať cestu
9          if current not in visited: # pokud ještě nebyl navštíven
10             visited.add(current) # označ ho jako navštívený
11             for neighbour, edge_weight in graph.neighbours(current).items(): # pro
               ↪ každého souseda a jeho váhu hrany
12                 queue.append((neighbour, path + [neighbour], weight + edge_weight)) #
                   ↪ přidej ho do fronty spolu s jeho cestou a celkovou váhou
13     return [] # nenalezena žádná cesta
```

BFS v ohodnoceném grafu – poznámky

Problém: nenalezne správné řešení.

Složitost:

BFS v ohodnoceném grafu – poznámky

Problém: nenalezne správné řešení.

Složitost: $\Theta(n)$

DFS v ohodnoceném grafu

```python
def dfs(graph: IWGraph, start: Vertex, end: Vertex, visited: frozenset[Vertex] = frozenset()) -> tuple[list[Vertex], int]:
    if start == end: # zastavovací podmínka: pokud dosáhneme cílového vrcholu
        return [start], 0 # vrať cestu a její váhu
    # rekurzivní případ
    visited = visited | {start} # označ ho jako navštívený
    best_path, best_weight = None,  float('inf')# nejlepší cesta None a váha nekonečno
    for neighbour, edge_weight in filter(lambda x: x[0] not in visited, graph.neighbours(start).items()): # pro nenavštívené sousedy a váhy
        result = shortest_path_rec(graph, neighbour, end, visited) # rekurzivní volání
            result_path, result_weight = result # rozbal výsledek na cestu a váhu
            result_path = [start] + result_path # přidej k cestě výchozí hranu
            result_weight += edge_weight # přičti k délce hranu do uzlu
            if result_weight < best_weight: # pokud je váha lepší než nejlepší váha
                best_path = result_path # aktualizuj nejlepší cestu
                best_weight = result_weight # aktualizuj nejlepší váhu
    return (best_path, best_weight) if best_path is not None else None # vrať nejlepší cestu a její váhu pokud existuje, nebo None jinak
```

DFS v ohodnoceném grafu

Řešení: nalezne správné řešení.

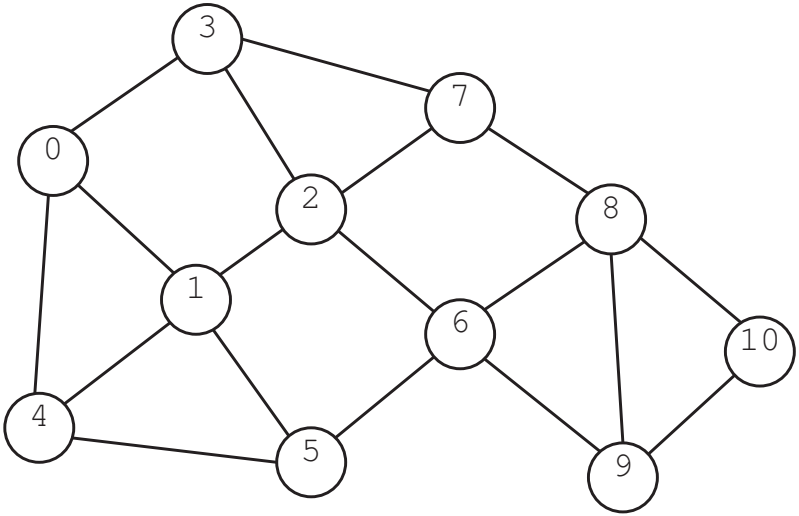
Složitost:

DFS v ohodnoceném grafu

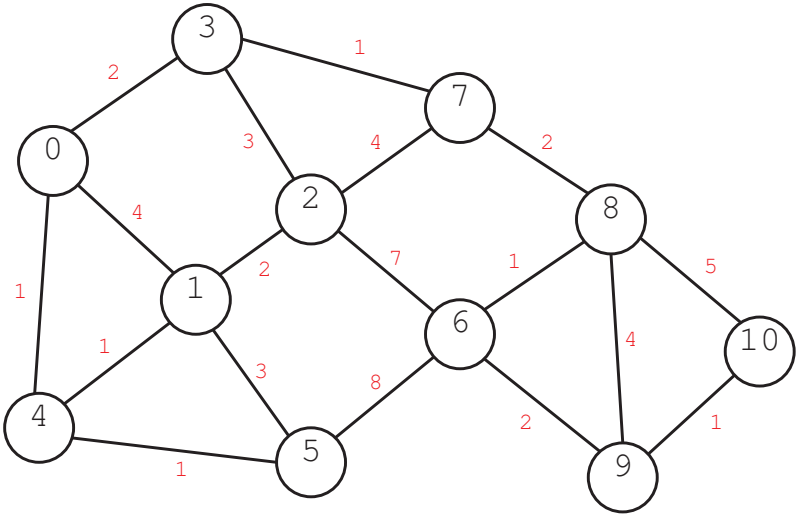
Řešení: nalezne správné řešení.

Složitost: $\Theta(n!)$

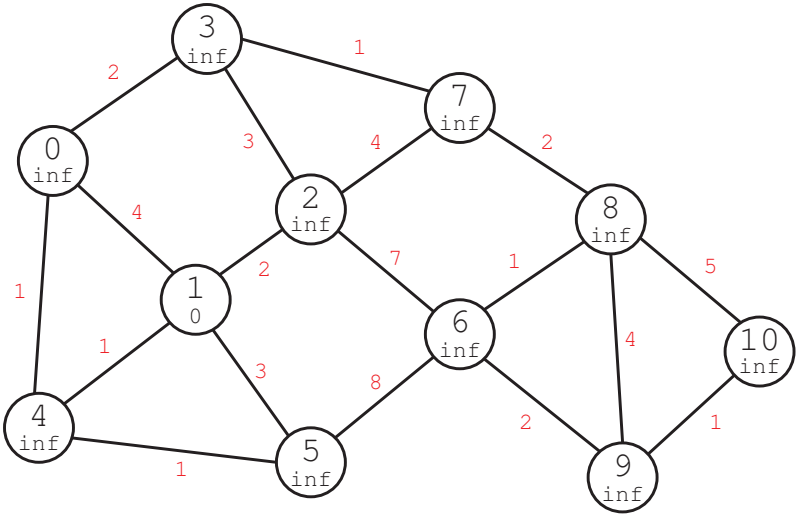
Dijkstrův algoritmus



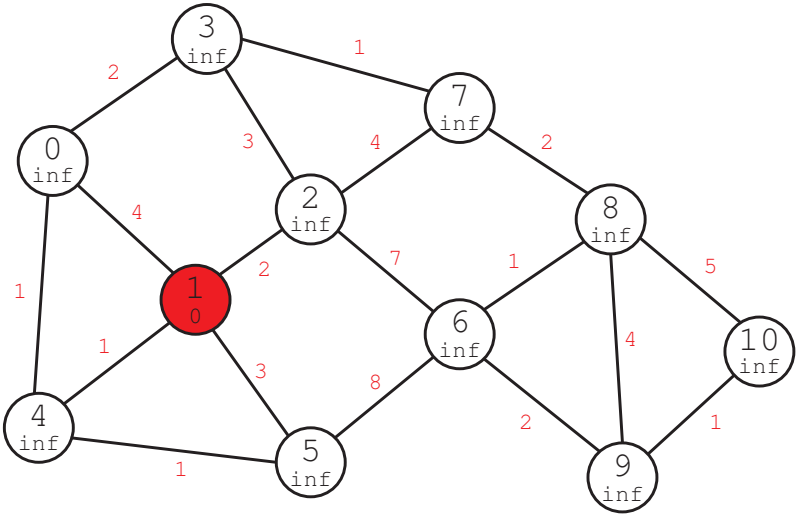
Dijkstrův algoritmus



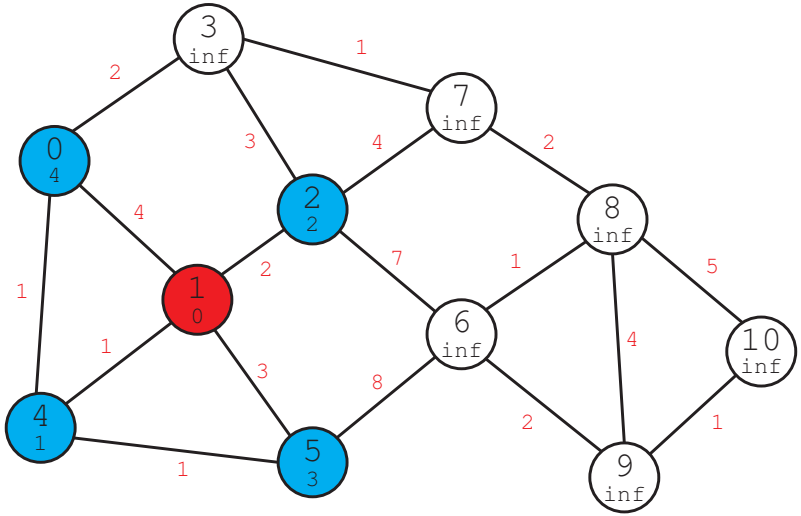
Dijkstrův algoritmus



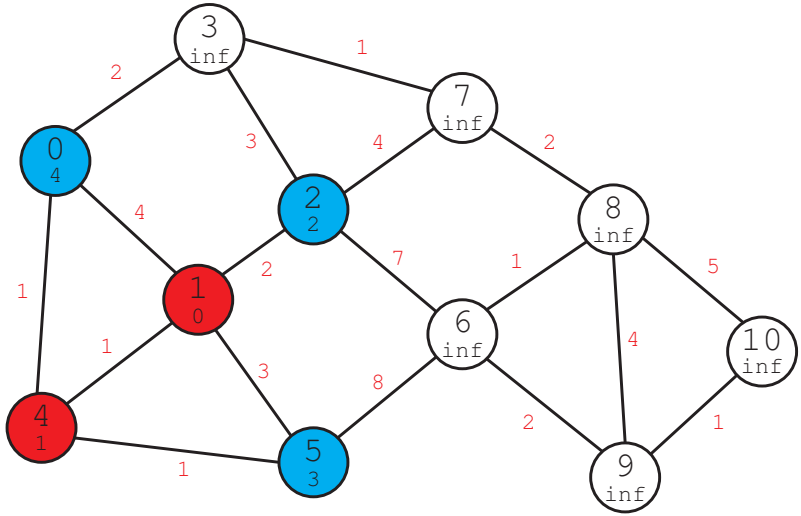
Dijkstrův algoritmus



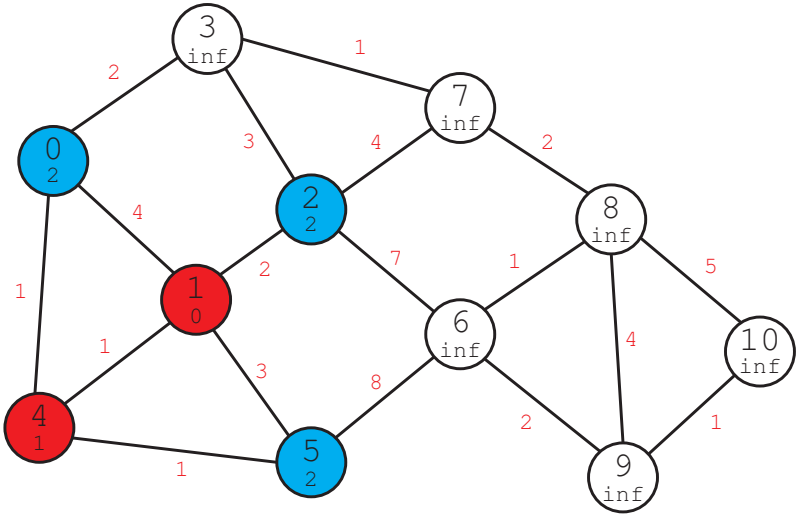
Dijkstrův algoritmus



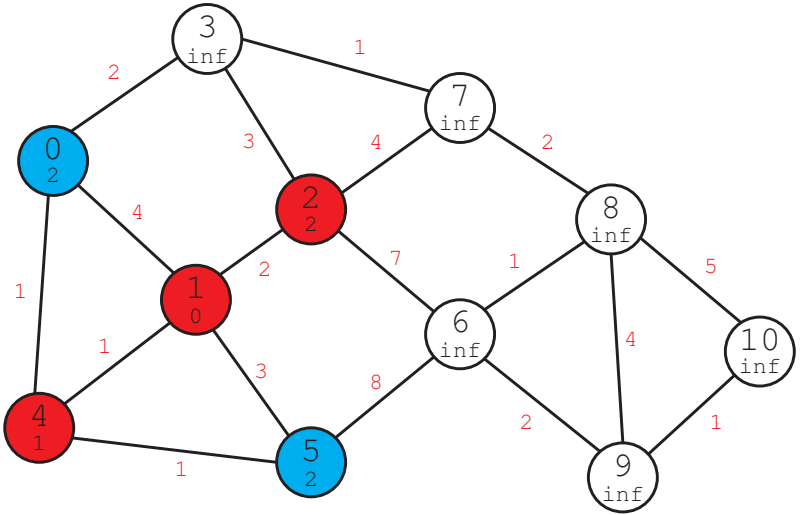
Dijkstrův algoritmus



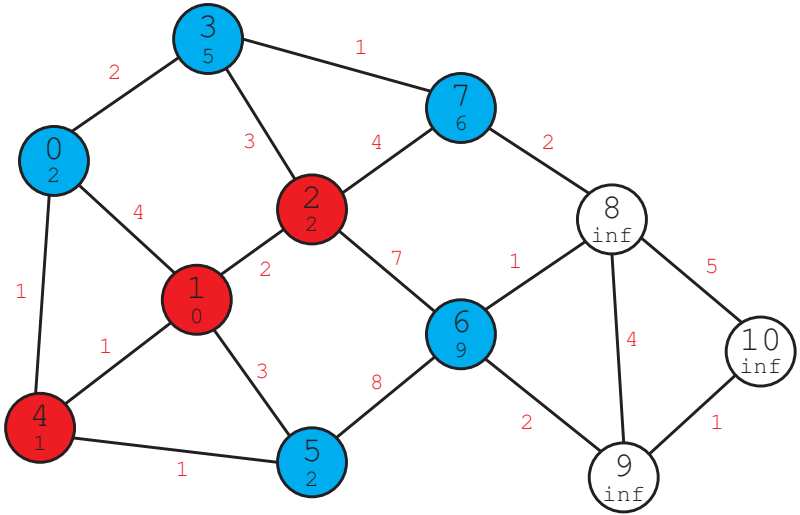
Dijkstrův algoritmus



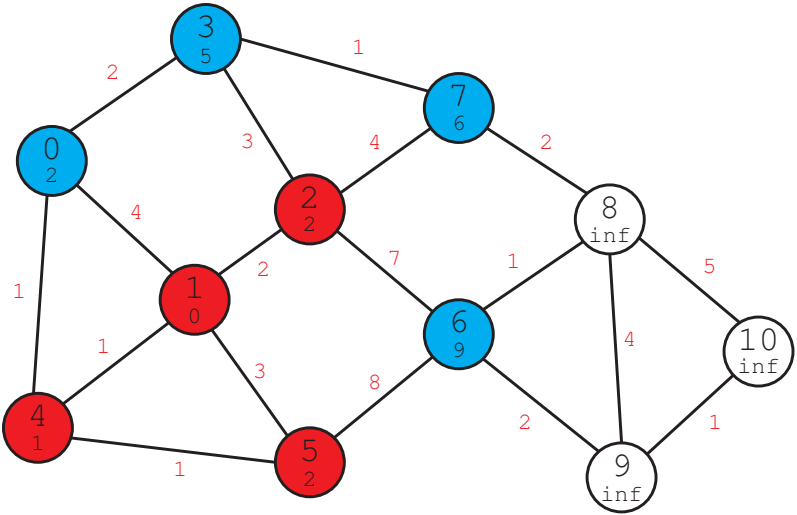
Dijkstrův algoritmus



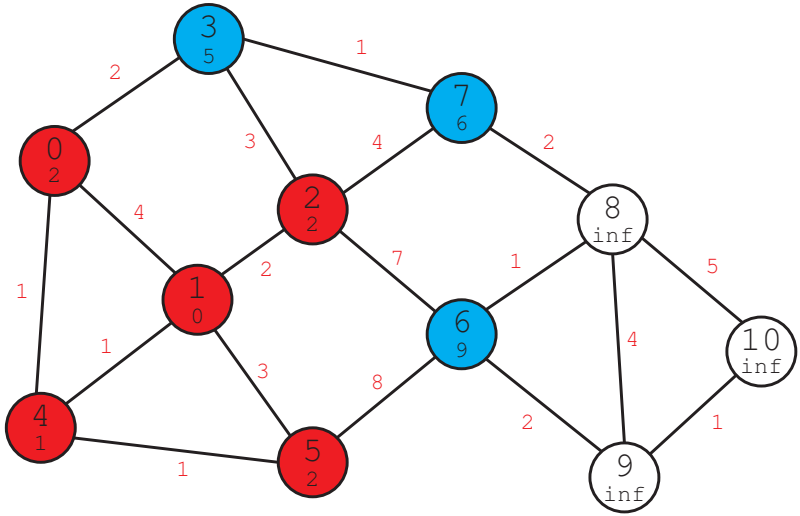
Dijkstrův algoritmus



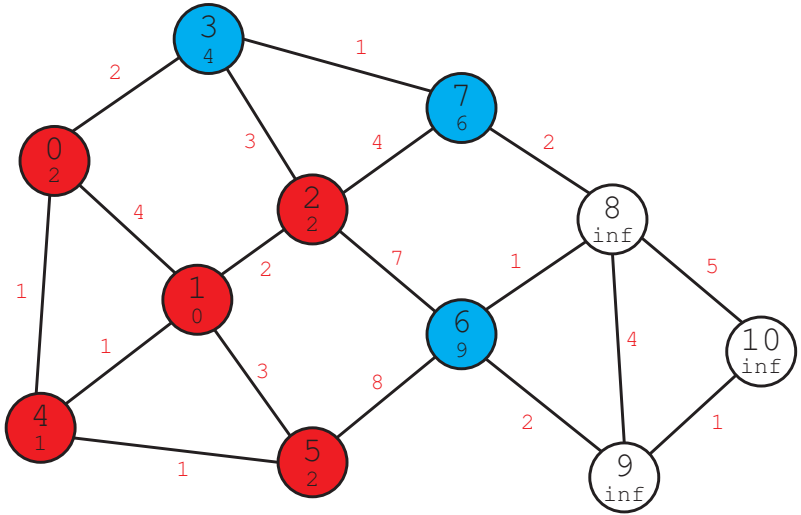
Dijkstrův algoritmus



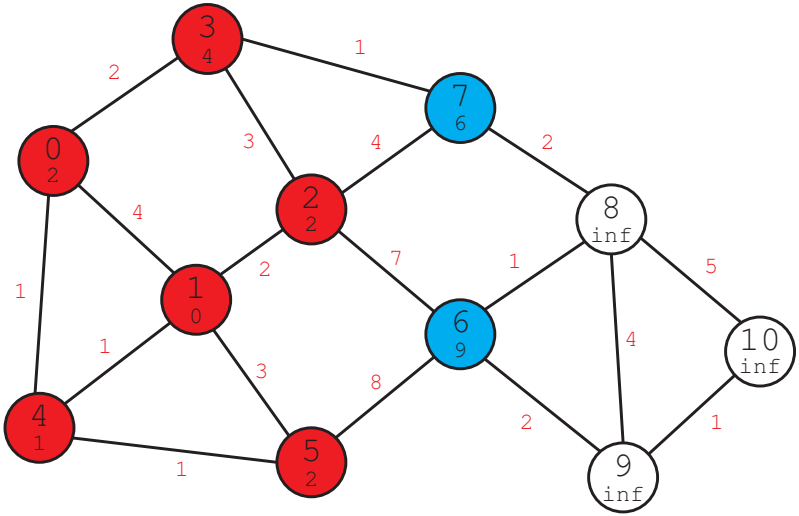
Dijkstrův algoritmus



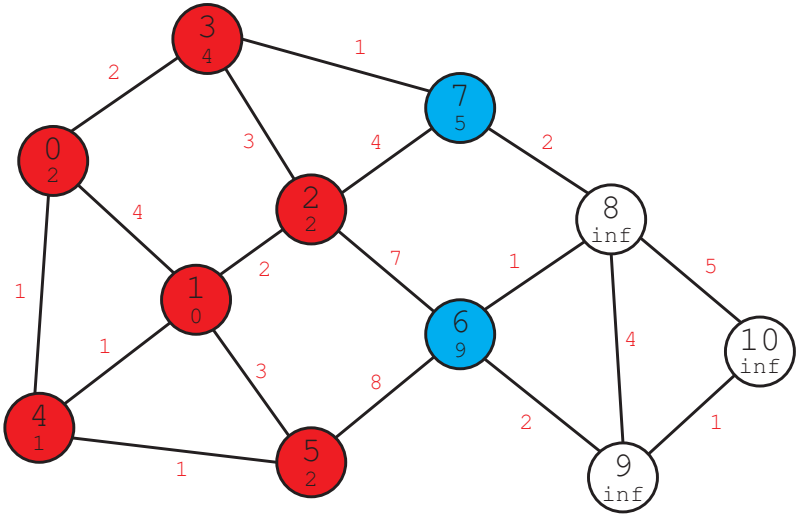
Dijkstrův algoritmus



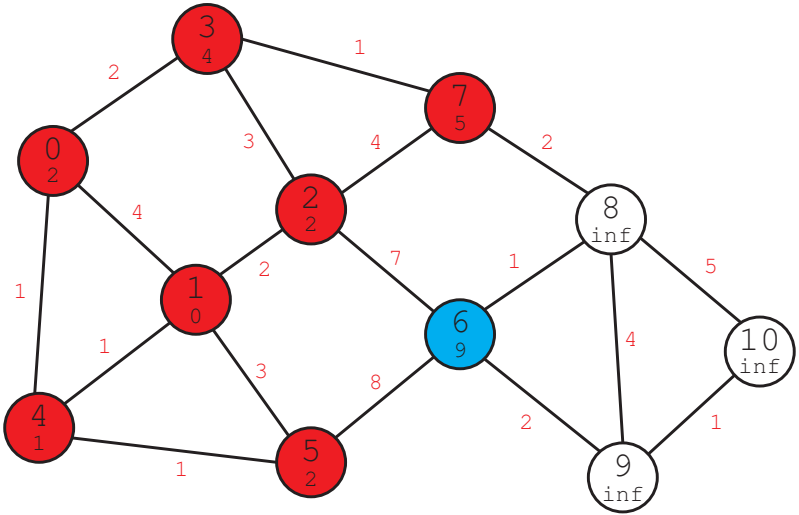
Dijkstrův algoritmus



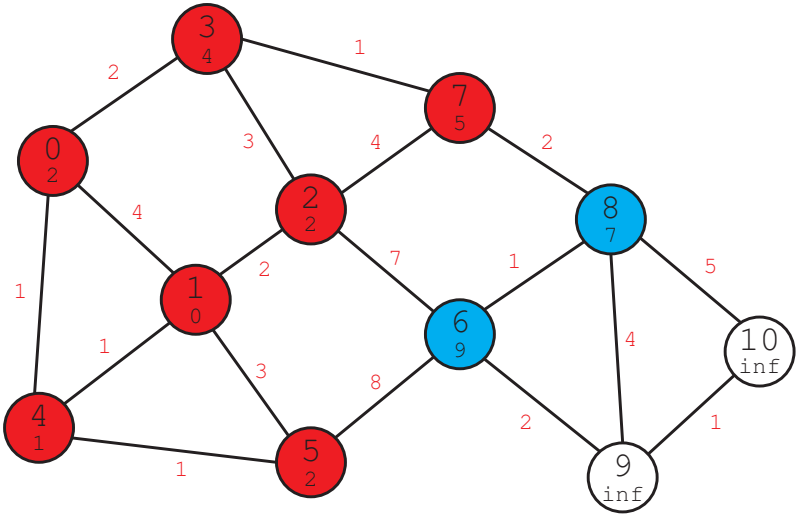
Dijkstrův algoritmus



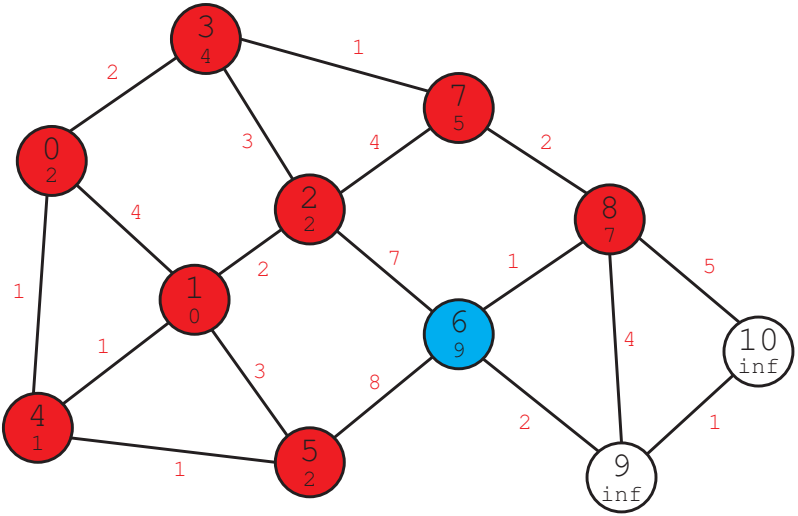
Dijkstrův algoritmus



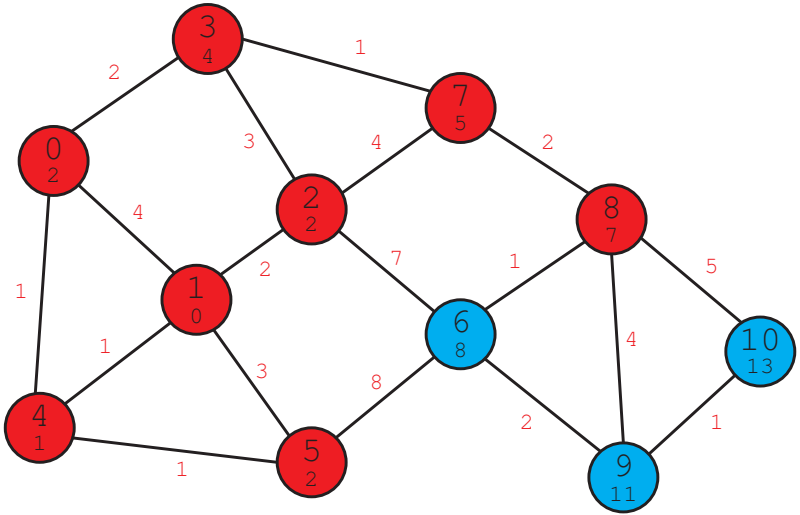
Dijkstrův algoritmus



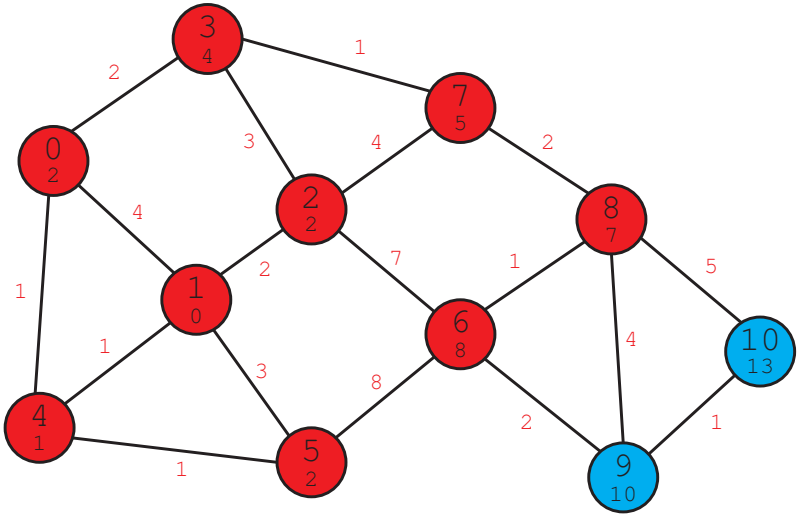
Dijkstrův algoritmus



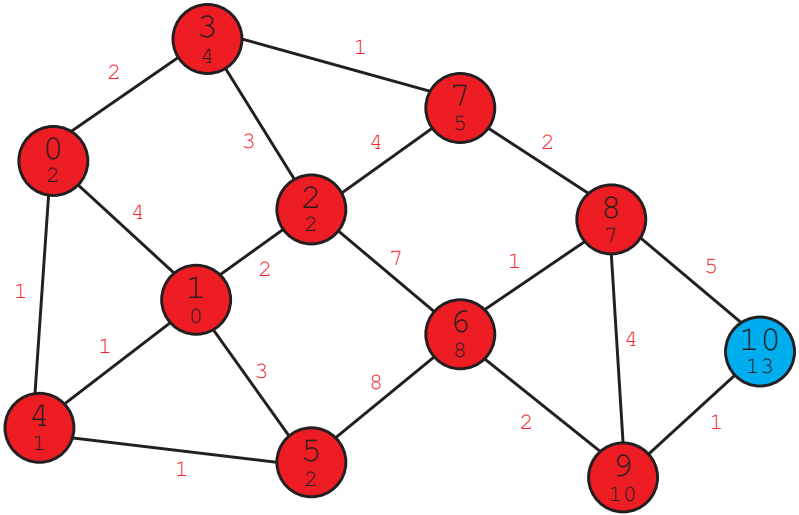
Dijkstrův algoritmus



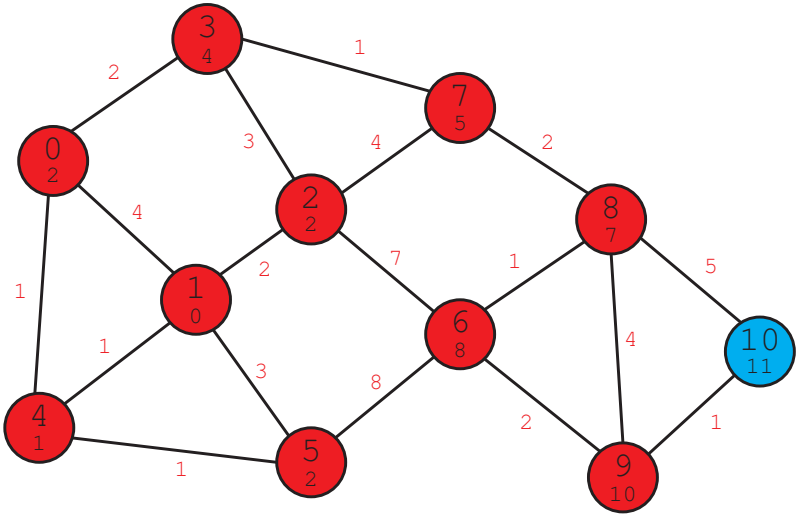
Dijkstrův algoritmus



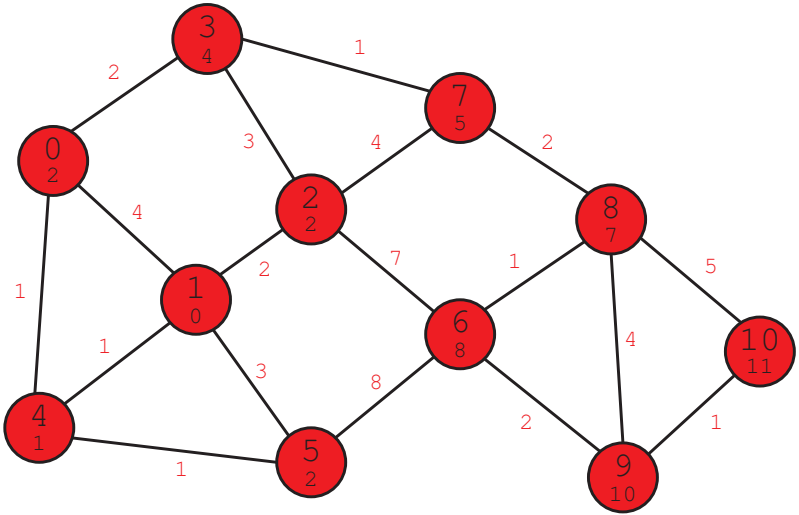
Dijkstrův algoritmus



Dijkstrův algoritmus



Dijkstrův algoritmus



Prioritní fronta

ADT Prioritní fronta

- Prioritní fronta uchovává prvky s přiřazenou prioritou.
- Podle priority jsou prvnky uspořádány.
- Operace s prioritní frontou zahrnují vkládání prvku, odebírání prvku s nejvyšší prioritou a zjištění nejvyšší priority.

ADT Prioritní fronta

- Prioritní fronta uchovává prvky s přiřazenou prioritou.
- Podle priority jsou prvnky uspořádány.
- Operace s prioritní frontou zahrnují vkládání prvku, odebírání prvku s nejvyšší prioritou a zjištění nejvyšší priority.

Základní operace ADT:

- Vložení prvku s prioritou – `put(item, priority)`.
- Vybrání prvku s nejvyšší prioritou – `get()`.
- Odebrání prvku s nejvyšší prioritou – `pop()`.
- Snížení priority prvnku – `decrease_key(item, new_priority)`.

ADT Prioritní fronta

- Prioritní fronta uchovává prvky s přiřazenou prioritou.
- Podle priority jsou prvnky uspořádány.
- Operace s prioritní frontou zahrnují vkládání prvku, odebírání prvku s nejvyšší prioritou a zjištění nejvyšší priority.

Základní operace ADT:

- Vložení prvku s prioritou – `put(item, priority)`.
- Vybrání prvku s nejvyšší prioritou – `get()`.
- Odebrání prvku s nejvyšší prioritou – `pop()`.
- Snížení priority prvnku – `decrease_key(item, new_priority)`.

Další operace:

- `size()` – vrátí počet prvků ve frontě
- `clear()` – vyprázdní frontu

Prioritní fronta – jednoduchá implementace neseřazným polem

```
1  class PriorityQueue:
2      def __init__(self):
3          self.items = [] # seznam pro ukládání položek

4      def is_empty(self): # zkontroluj, zda je  fronta prázdná
5          return len(self.items) == 0

6      def put(self, value, priority): # vloží novou položku s danou hodnotou a prioritou
7          self.items.append( (value, priority) ) # přidejte ji na konec seznamu

8      def get(self):  # vrátí a odeberte položku s nejnižší prioritou
9          if self.is_empty():
10             return None # vrátí None, pokud je fronta prázdná
11         else:
12             min_index = min(range(len(self.items)), key=lambda i: self.items[i][1]) #
               ↪  index minimální položky
13             min_item = self.items[min_index] # položka na tomto indexu
14             del self.items[min_index] # smaže položku na tomto indexu
15             return min_item[0] # vrátí její hodnotu
```

Prioritní fronta – implementace VS složitost

- Vložení prvku: `put()`:
  - Neseřazený seznam: čas $\Theta(1)$
  - Halda: čas $\Theta(\log n)$
  - Fibonacciho halda: amortizovaný čas $\Theta(1)$
- Odebrání minima `get()`:
  - Neseřazený seznam: čas $\Theta(n)$
  - Halda: čas $\Theta(\log n)$
  - Fibonacciho halda: amortizovaný čas $\Theta(logn)$
- Snížení hodnoty klíče `decrease_key()`:
  - Neseřazený seznam: $\Theta(n)$
  - Halda: čas $\Theta(\log n)$
  - Fibonacciho halda: amortizovaný čas $\Theta(1)$

Implementace:

Neseřazený seznam: viz výše.

Halda : viz KIV/IDT.

Fibonacciho halda : viz [1].

Dijkstrův algoritmus – implementace

```
1  def dijkstra(graph: IWGraph, start: Vertex):
2      D = {start:0} # inicializuj slovník vzdáleností, počáteční vrchol má vzdálenost nula
3      pq = PriorityQueue() # vytvoř prioritní frontu
4      pq.put((0, start)) # vlož do fronty dvojici (vzdálenost, vrchol)
5      visited = set() # množina pro uložení navštívených vrcholů

6      while not pq.empty(): # dokud není fronta prázdná
7          (dist, current_vertex) = pq.get() # vezmi z fronty dvojici (vzdálenost, vrchol)
8          visited.add(current_vertex) # přidej aktuální vrchol do množiny navštívených
9          for neighbor, distance in graph.neighbours(current_vertex).items():
10             if neighbor not in visited: # pokud soused ještě nebyl navštíven
11                 old_cost = D.get(neighbor, float('inf')) # stará vzdálenost souseda
12                 new_cost = D[current_vertex] + distance # nová vzdálenost souseda
13                 if new_cost < old_cost: # pokud je nová vzdálenost menší než stará
14                     pq.put((new_cost, neighbor)) # vlož do fronty dvojici (nová
                       ↪ vzdálenost, soused)
15                     D[neighbor] = new_cost # aktualizuj slovník pro vzdálenosti
16     return D # vrať slovník pro vzdálenosti
```

Dijsktra složitost

Složitost: $\Theta\left(|V|^2 \texttt{fce}(|V|)\right)$.

fce : záleží na složitosti `queue.get()` a `queue.put()`.

Prioritní fronta: optimalizuje operace:

- Vložení prvku (put).
- Odebrání minima (get).
- Snížení hodnoty klíče (decrease key).

Dijsktra složitost s prioritní frontou

Složitost: $\Theta\left(|V|^2 \texttt{fce}(|V|)\right)$

- Neseřazený seznam: $\Theta(|V|^3)$
- Prioritní fronta (halda): $\Theta(|V|^2 \log |V|)$
- Prioritní fronta (Fibonacciho halda): $\Theta(|V|^2 \log |V|)$

Nejkratší cesta

Nejkratší cesta

- Nejkratší cesta je cesta s nejmenším ohodnocením.
- Ohodnocení cesty je součet ohodnocení hran, kterými cesta prochází.
- Nejkratší cesta může být jedna nebo více.
- Nejkratší cesta může být orientovaná nebo neorientovaná.

Implementace:

- Dijkstrův algoritmus.
- Ukládáme se zpětné odkazy na předchozí vrcholy.

Dijstrův algoritmus pro hledání nejkratší cesty

```
1  def dijkstra_shortest(graph: IWGraph, start: Vertex, end: Vertex) -> list[Vertex]:
2      # provede prohledávání do šířky od zadaného vrcholu a vrátí cestu k cílovému vrcholu
       ↪ s nejmenší váhou, pokud existuje
3      queue = PriorityQueue()  # prioritní fronta pro navštívené vrcholy, jejich cesty a
       ↪ jejich váhy
4      visited = set()  # množina pro označení navštívených vrcholů
5      queue.put((0, (start, [start])))  # vlož startovní vrchol do fronty s nulovou váhou
6      while not queue.empty():  # dokud je fronta neprázdná
7          weight, (current, path) = queue.get()  # vezmi vrchol, jeho cestu a jeho váhu z
           ↪ fronty
8          if current == end:  # pokud je to cílový vrchol
9              return path  # vrať cestu
10         if current not in visited:  # pokud ještě nebyl navštíven
11             visited.add(current)    # označ ho jako navštívený
12             for neighbour, edge_weight in graph.neighbours(current):  # pro každého
               ↪ souseda a jeho váhu hrany
13                 queue.put((weight + edge_weight, (neighbour, path + [neighbour])))  #
                   ↪ vlož ho do fronty spolu s jeho cestou a celkovou váhou
14     return []  # nenalezena žádná cesta
```

Kvíz III

Kostra grafu

Kostra grafu

Definice:

- Kostra grafu je strom, který obsahuje všechny vrcholy původního grafu.
- Kostra grafu je souvislá.
- Kostra grafu neobsahuje cykly.

Kostra grafu

Definice:

- Kostra grafu je strom, který obsahuje všechny vrcholy původního grafu.
- Kostra grafu je souvislá.
- Kostra grafu neobsahuje cykly.

Otázka

Jak najít kostru grafu?

1. Prohledáváme graf.
2. Každý nově objevený uzel spojíme hranou s akutálním uzlem v novém grafu.
3. Pokud již uzel v novém grafu existuje, hranu nepřidáváme.

Kostra grafu – Algoritmus

```
1  def spanning_tree(graph: IGraph, start: iVertex) -> IGraph:
2      tree = UndirectedGraph()  # vytvoření nového grafu pro kostru
3      visited = set()  # množina pro sledování navštívených vrcholů
4      queue = deque([start])  # fronta pro provedení prohledávání do šířky
5      old2new = {vertex: AdjListVertex(vertex.value) for vertex in graph.getVertices()}  #
       ↪  slovník pro mapování původních vrcholů na nové

6      while queue:
7          current = queue.popleft()
8          if current not in visited:
9              visited.add(current) # označ uzel za navštívený
10             for neighbor in current.getNeighbours():
11                 if neighbor not in visited:
12                     queue.append(neighbor) # přidej sousedy do fronty pro další
                       ↪  prohledání
13                     if old2new[neighbor] not in tree.getVertices():
14                         tree.addEdge(old2new[current], old2new[neighbor])  # přidání
                           ↪  hrany do kostry
15     return tree
```

Minimální kostra ohodnoceného grafu

Definice:

- Minimální kostra grafu je kostra s nejmenším součtem ohodnocení hran.
- Minimální kostra grafu může být jedna nebo více.

Minimální kostra ohodnoceného grafu

Definice:

- Minimální kostra grafu je kostra s nejmenším součtem ohodnocení hran.
- Minimální kostra grafu může být jedna nebo více.

Použití:

- Propojení domů s nejmenšími náklady na budování cest.
- Propojení počítačových sítí s nejmenšími náklady na kabeláž.
- Plánování logistických a distribučních sítí pro optimalizaci dopravních tras.
- Zefektivnění rozvodných sítí elektřiny nebo vody pro minimalizaci ztrát.
- Plánování vysazení lesních a zemědělských oblastí pro maximalizaci užitku a minimalizaci nákladů.
- Analýza sociálních sítí pro identifikaci minimálních struktur propojení mezi klíčovými uzly.

Implementace Prim-Jarnikova algoritmu

Implementace Jarnik

1. Prohledáváme graf.
2. Vždy vyjmeme vrchol s nejmenším ohodnocením váhy, která do něj vede.
3. Pokud se z vyjmutého vrcholu dostaneme do nějakého souseda s menším ohodnocením, tak ohodnocení souseda nahradíme.

Prim-Jarnikův algoritmus pro hledání minimální kostry – I

```
1  def jarnik_algorithm(graph: IWGraph) -> dict[iVertex, iVertex]:
2      q: PriorityQueue[tuple[int, tuple[Optional[iVertex],iVertex]]] =
       ↪  PriorityQueue()  # vytvoř prioritní frontu
3      q.put((0, (None, list(graph.getVertices())[0])))  # vlož první vrchol
       ↪  do fronty s váhou 0
4      parent = {}  # Vytvoř slovník pro ukládání rodičovského vrcholu pro
       ↪  každý vrchol

5      visited: set[iVertex] = set()  # navštívené vrcholy
```

Prim-Jarnikův algoritmus pro hledání minimální kostry – II

```
9  while not q.empty(): # dokud je fronta neprázdná
10     weight, edge = q.get()  # Získej hranu s minimální váhou z prioritní fronty
11     from_vertex, vertex = edge  # Získej vrchol a jeho rodičovský vrchol

12     if vertex in visited:  # Pokud je vrchol již navštívený, pokračuj
13         continue

14     if from_vertex is not None:  # Pokud vrchol není první, přidej hranu do kostry
15         parent[vertex] = from_vertex

16     for neighbor, edge_weight in graph.neighbours(vertex).items():  # Pro každého souseda
17         if neighbor not in visited: # Pokud soused není navštívený
18             q.put((edge_weight, (vertex, neighbor))) # Přidej hranu do prioritní fronty

19     visited.add(vertex)  # Přidej vrchol do množiny navštívených

20 return parent
```

Kvíz IV

References I

[1] Michael L Fredman a Robert E Tarjan. “Fibonacci heaps and their uses in improved network optimization algorithms”. In: *Journal of the ACM (JACM)* 34.3 (1987), s. 596–615.