PPA2, Přednáška 10

Libor Váša

Katedra informatiky a výpočetní techniky, Západočeská univerzita v Plzni

3. 5. 2019

Zkouškové termíny

Byly vypsány

1. 28.5. 2019 9:00
2. 4.6. 2019 9:00
3. 11.6. 2019 9:00
4. 27.6. 2019 9:00
5. 27.8. 2019 9:00
6. další nebudou

Zkouškové termíny

Byly vypsány

1. 28.5. 2019 9:00
2. 4.6. 2019 9:00
3. 11.6. 2019 9:00
4. 27.6. 2019 9:00
5. 27.8. 2019 9:00
6. další nebudou

Kdo má 24.5. nárok na zápočet, tomu bude na STAG zapsán

Zkouškové termíny

Byly vypsány

1. 28.5. 2019 9:00
2. 4.6. 2019 9:00
3. 11.6. 2019 9:00
4. 27.6. 2019 9:00
5. 27.8. 2019 9:00
6. další nebudou

Kdo má 24.5. nárok na zápočet, tomu bude na STAG zapsán
Komu vznikne nárok později, ten musí požádat cvičícího o zapsání na STAG

Zkouškové termíny

Byly vypsány

1. 28.5. 2019 9:00
2. 4.6. 2019 9:00
3. 11.6. 2019 9:00
4. 27.6. 2019 9:00
5. 27.8. 2019 9:00
6. další nebudou

Kdo má 24.5. nárok na zápočet, tomu bude na STAG zapsán
Komu vznikne nárok později, ten musí požádat cvičícího o zapsání na STAG
Do indexu se zápočty budou zapisovat až spolu s výsledkem zkoušky

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: Trojúhelník
Vztah: Trojúhelníky sousedí v trojúhelníkové síti

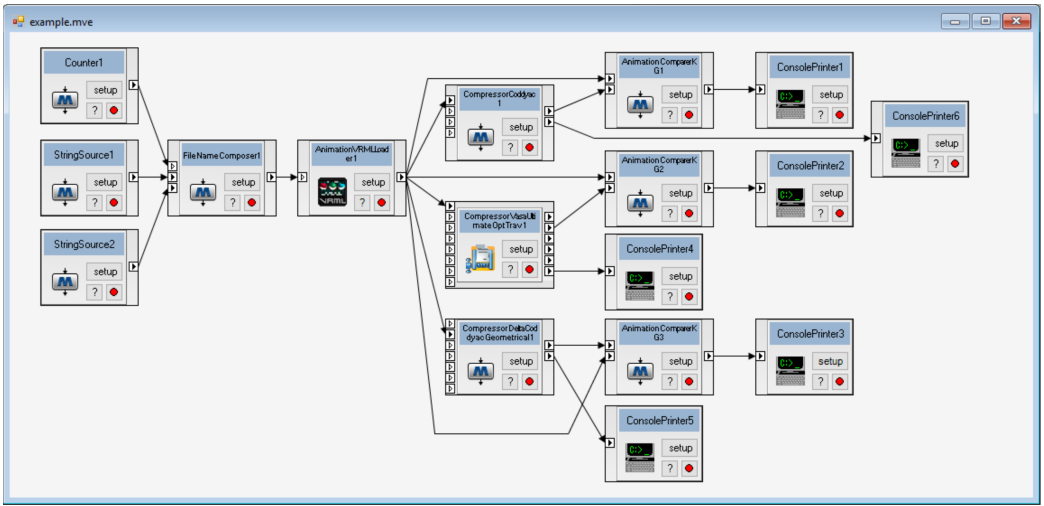
Příklady

Prvek: Město
Vztah: Města jsou spojená jedním úsekem silnice

Prvek: Trojúhelník
Vztah: Trojúhelníky sousedí v trojúhelníkové síti

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

Modular Visualization Environment 2



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

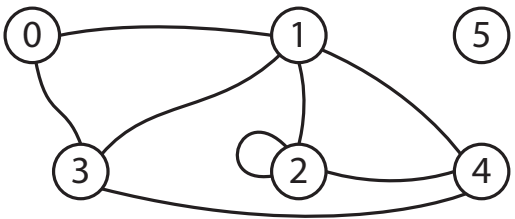
$y \in V$ je sousedem $x \in V$ právě když

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

Příklad neorientovaného grafu

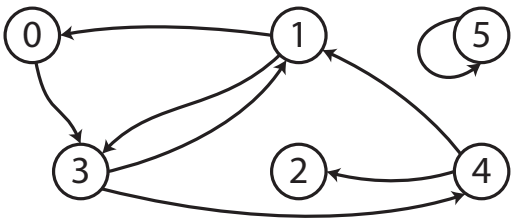
Příklad orientovaného grafu

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)

Reprezentace vrcholů

- s vrcholy mohou být asociována data
- ADT Graf ale neumožňuje vrcholy přidávat a odebírat

Reprezentace vrcholů

- s vrcholy mohou být asociována data
- ADT Graf ale neumožňuje vrcholy přidávat a odebírat

Důsledek

- data přiřazená vrcholům je možné držet mimo ADT v poli
- jedinou reprezentací vrcholu je jeho index

Rozhraní ADT Graf

```
interface IGraph{
  void initialize(int vertexCount);
  void addEdge(int start, int end);
  ArrayList<Integer> neighbours(int vertex);
  boolean isNeighbour(int v1, int v2);
}
```

Použití ADT graf

```
Person[] people = new Person[3];
people[0] = new Person("Jennifer␣Aniston");
people[1] = new Person("Brad␣Pitt");
people[2] = new Person("Angelina␣Jolie");
```

Použití ADT graf

```
Person[] people = new Person[3];
people[0] = new Person("Jennifer␣Aniston");
people[1] = new Person("Brad␣Pitt");
people[2] = new Person("Angelina␣Jolie");

IGraph relations = new Graph();
relations.initialize(3);
relations.addEdge(0,1);
relations.addEdge(1,2);
```

Použití ADT graf

```
Person[] people = new Person[3];
people[0] = new Person("Jennifer␣Aniston");
people[1] = new Person("Brad␣Pitt");
people[2] = new Person("Angelina␣Jolie");

IGraph relations = new Graph();
relations.initialize(3);
relations.addEdge(0,1);
relations.addEdge(1,2);

ArrayList<Integer> bradsRelations = relations.neighbours(1);
for (int i = 0;i<bradsRelations.size();i++)
  System.out.println(people[bradsRelations.get(i)].name);
```

Implementace ADT Graf

Dvě možnosti:

- seznamy sousednosti
- matice sousednosti

- různé vlastnosti v závislosti na vlastnostech grafu
- implementace se liší pro orientované a neorientované grafy

Implementace grafu seznamem sousednosti

- sousedé každého vrcholu jsou uloženi v ADT Seznam

Implementace grafu seznamem sousednosti

- sousedé každého vrcholu jsou uloženi v ADT Seznam
  - mohli bychom použít existující implementaci (třeba `LinkedList<Integer>`), ale
    - většinu funkcionality nevyužijeme
    - program by byl neefektivní (implicitní konverze `int`/`Integer`)
    - chceme vidět jak věci fungují uvnitř

Implementace grafu seznamem sousednosti

- sousedé každého vrcholu jsou uloženi v ADT Seznam
  - mohli bychom použít existující implementaci (třeba `LinkedList<Integer>`), ale
    - většinu funkcionality nevyužijeme
    - program by byl neefektivní (implicitní konverze `int`/`Integer`)
    - chceme vidět jak věci fungují uvnitř
  - použijeme vlastní, zjednodušenou implementaci
  - typ dat: `int`

Implementace grafu seznamem sousednosti

- sousedé každého vrcholu jsou uloženi v ADT Seznam
  - mohli bychom použít existující implementaci (třeba `LinkedList<Integer>`), ale
    - většinu funkcionality nevyužijeme
    - program by byl neefektivní (implicitní konverze `int`/`Integer`)
    - chceme vidět jak věci fungují uvnitř
  - použijeme vlastní, zjednodušenou implementaci
  - typ dat: `int`
- reference na první prvek každého seznamu jsou uloženy v poli velikosti $|V|$

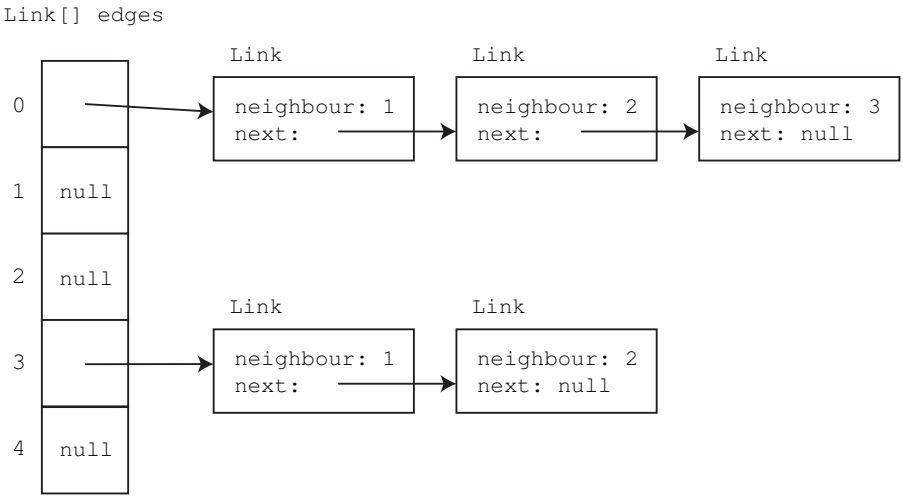
Spojovací prvek

```
class Link{
  int neighbour;
  Link next;

  public Link(int neighbour, Link next) {
    this.neighbour = neighbour;
    this.next = next;
  }
}
```

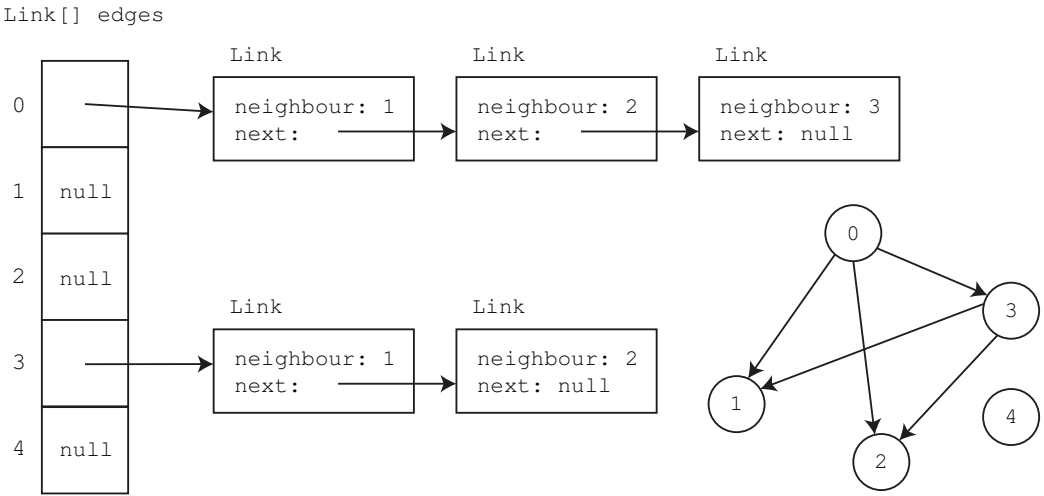
Implementace

```
class Graph implements IGraph{
  Link[] edges;

  public void initialize(int vertexCount){
    this.edges = new Link[vertexCount];
  }

  ... // metody
}
```

Seznam sousednosti



Seznam sousednosti



Přidání hrany

Orientovaný graf

```
void addEdge(int start, int end){
  edges[start] = new Link(end, edges[start]);
}
```

Přidání hrany

Orientovaný graf

```
void addEdge(int start, int end){
  edges[start] = new Link(end, edges[start]);
}
```

Neorientovaný graf

```
void addEdge(int i, int j){
  edges[i] = new Link(j, edges[i]);
  edges[j] = new Link(i, edges[j]);
}
```

Přidání hrany

Orientovaný graf

```
void addEdge(int start, int end){
  edges[start] = new Link(end, edges[start]);
}
```

Neorientovaný graf

```
void addEdge(int i, int j){
  edges[i] = new Link(j, edges[i]);
  edges[j] = new Link(i, edges[j]);
}
```

Vkládáme na začátek seznamu

- na pořadí nezáleží
- vložení na začátek je rychlé

Sousedi vrcholu

```
ArrayList<Integer> neighbours(int v){
  ArrayList<Integer> result = new ArrayList<Integer>();
  Link n = edges[v];
  while(n!=null) {
    result.add(n.neighbour);
    n = n.next;
  }
  return result;
}
```

Složitost:

- průchod všemi sousedy závisí na jejich počtu
  - husté grafy: počet sousedů může být $\Omega(|V|)$
  - řídké grafy: (průměrný) počet sousedů $\mathcal{O}(1)$
    - např. pro planární trojúhelníkovou síť je průměrný počet sousedů 6 (konstanta!)

Test sousednosti

```
boolean isNeighbour(int i, int j){
  Link n = edges[i];
  while(n!=null) {
    if (n.neighbour == j)
      return true;
    n = n.next;
  }
  return false;
}
```

Složitost závisí na počtu sousedů!
(prochází se seznam)
Složitost: v nejhorším případě (hustý graf): $\Omega(|V|)$

Úprava na ohodnocený graf

Ohodnocený graf

- každé hraně je přiřazeno navíc číslo představující nějakou dodatečnou vlastnost
  - délka cesty
  - propustnost potrubí
  - ...

Seznamy sousedů pak musí obsahovat instance složitější třídy, která zachycuje i ohodnocení

```
class Link {
  int neighbour;
  double edgeValue;
  Link next;
  ...
}
```

Implementace maticí sousednosti

Orientovaný graf
Matice $|V| \times |V|$ obsahuje na pozici $[i, j]$

- hodnotu 1 pokud z i-tého vrcholu vede hrana do j-tého
- hodnotu 0 v ostatních případech

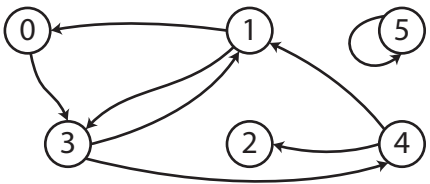
Neorientovaný graf
Matice $|V| \times |V|$ obsahuje na pozici $[i, j]$ a $[j, i]$

- hodnotu 1 pokud z i-tého vrcholu vede hrana do j-tého
- hodnotu 0 v ostatních případech

Příklad orientovaného grafu

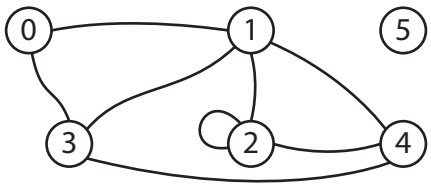
kam

odkud	0	1	2	3	4	5
0	0	0	0	1	0	0
1	1	0	0	1	0	0
2	0	0	0	0	0	0
3	0	1	0	0	1	0
4	0	1	1	0	0	0
5	0	0	0	0	0	1

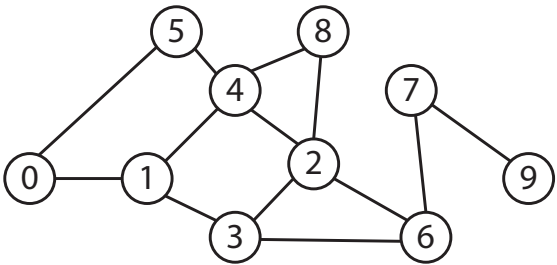


Příklad neorientovaného grafu

	0	1	2	3	4	5
0	0	1	0	1	0	0
1	1	0	1	1	1	0
2	0	1	1	0	1	0
3	1	1	0	0	1	0
4	0	1	1	1	0	0
5	0	0	0	0	0	0



Zapište první dvě řádky matice sousednosti (oddělte středníkem)



Reprezentace matice

- `int[][]` (dvourozměrné pole integerů)
  - alokováno množství paměti zbytečně
- `byte[][]` (dvourozměrné pole bytů)
- `boolean[][]` (dvourozměrné pole bitů)
  - překladač ale stejně obvykle pro každý `boolean` alokuje celý `byte`

Implementace

```
class Graph{
  byte[][] matrix;

  public void initialize(int vertexCount){
    this.matrix = new byte[vertexCount][vertexCount];
  }

  ... // metody
}
```

Přidání hrany

Orientovaný graf

```
void addEdge(int i, int j){
  matrix[i][j] = 1;
}
```

Neorientovaný graf

```
void addEdge(int i, int j){
  matrix[i][j] = 1;
  matrix[j][i] = 1;
}
```

Sousedi vrcholu

```
ArrayList<Integer> neighbours(int v){
  ArrayList<Integer> result = new ArrayList<Integer>();
  for (int i = 0;i<matrix[0].length;i++)
    if (matrix[v][i] == 1) result.add(i);
  return result;
}
```

Složitost $\Omega(|V|)$ nezávisle na hustotě grafu

Test sousednosti

```
boolean isNeighbour(int i, int j){
  return matrix[i][j]==1;
}
```

Složitost $\mathcal{O}(1)$ nezávisle na hustotě grafu

Úprava na ohodnocený graf

- matice reprezentována jako `double[][]`
- v matici na pozici sousedících vrcholů uloženo ohodnocení
- na ostatních pozicích hodnota mimo rozsah možných ohodnocení
  - -1, pokud ohodnocení musí být nezáporné
  - `NaN`, není-li `NaN` přípustným ohodnocením
  - záleží na aplikaci

Porovnání implementací

Paměťová složitost

- seznam: $\mathcal{O}(|V| + |E|)$
- matice: $\Omega(|V|^2)$

Porovnání implementací

Paměťová složitost

- seznam: $\mathcal{O}(|V| + |E|)$
- matice: $\Omega(|V|^2)$

Vložení hrany

- seznam i matice: $\mathcal{O}(1)$

Porovnání implementací

Paměťová složitost

- seznam: $\mathcal{O}(|V| + |E|)$
- matice: $\Omega(|V|^2)$

Vložení hrany

- seznam i matice: $\mathcal{O}(1)$

Sousedi vrcholu

- seznam: $\Omega(n)$, n je počet sousedů vrcholu
- matice: $\Omega(|V|)$

Porovnání implementací

Paměťová složitost
- seznam: $\mathcal{O}(|V| + |E|)$
- matice: $\Omega(|V|^2)$

Vložení hrany
- seznam i matice: $\mathcal{O}(1)$

Sousedi vrcholu
- seznam: $\Omega(n)$, n je počet sousedů vrcholu
- matice: $\Omega(|V|)$

Test sousednosti
- seznam: $\Omega(n)$, n je počet sousedů vrcholu
- matice: $\mathcal{O}(1)$

Hustota grafu

Řídký graf

- $|E| << |V^2|$
- obvykle vhodnější reprezentace seznamem sousedů
- např. pro planární trojúhelníkové sítě se dá dokázat, že průměrný počet sousedů vrcholu je blízký 6
  - test sousednosti pak také probíhá v průměru v $\mathcal{O}(1)$

Hustota grafu

Řídký graf

- $|E| << |V^2|$
- obvykle vhodnější reprezentace seznamem sousedů
- např. pro planární trojúhelníkové sítě se dá dokázat, že průměrný počet sousedů vrcholu je blízký 6
  - test sousednosti pak také probíhá v průměru v $\mathcal{O}(1)$

Hustý graf

- $|E| \simeq |V^2|$, popř. $|E| = k|V^2|$ pro nějakou konstantu k
- obvykle vhodnější reprezentace maticí sousednosti (rychlejší test sousednosti)

Procházení grafu

Motivace

Typické úlohy:

Existuje v grafu cesta z vrcholu A do vrcholu B?

Graf: bludiště
Vrchol: křižovatka v bludišti
Hrana: cesta mezi křižovatkami
Úkol: Zjistit, zda existuje cesta z jednoho místa na jiné

Motivace

Typické úlohy:

Existuje v grafu cesta z vrcholu A do vrcholu B?

Graf: bludiště
Vrchol: křižovatka v bludišti
Hrana: cesta mezi křižovatkami
Úkol: Zjistit, zda existuje cesta z jednoho místa na jiné

Jak dlouhá (kolik hran) je nejkratší cesta z vrcholu A do vrcholu B?

Graf: Vlaková spojení
Vrchol: Nádraží
Hrana: Mezi nádražími jede přímý spoj
Úkol: Vyhledat spojení s nejmenším počtem přestupů

Typické úlohy

Které vrcholy se v grafu vyskytují ve vzdálenosti menší než k (počet hran)?

Graf: síť kontaktů LinkedIn
Vrchol: záznam osoby
Hrana: konexe
Úkol: prohledat konexe do úrovně k, zda obsahují hledanou osobu

Typické úlohy

Které vrcholy se v grafu vyskytují ve vzdálenosti menší než k (počet hran)?

Graf: síť kontaktů LinkedIn
Vrchol: záznam osoby
Hrana: konexe
Úkol: prohledat konexe do úrovně k, zda obsahují hledanou osobu

Existuje v orientovaném grafu cyklus?

Graf: vztahy buněk v tabulkovém kalkulátoru
Vrchol: buňka
Hrana A→B: hodnota buňky A závisí na hodnotě buňky B
Úkol: zjistit, zda je možné tabulku vyhodnotit (nesmí obsahovat cyklus!)

Typické úlohy

Přiřadit vrcholům orientovaného grafu indexy tak, že hrany vedou vždy od menšího indexu k většímu

Graf: závislosti činností
Vrchol: činnost
Hrana A→B: činnost B může být vykonána, teprve když je činnost A hotová
Úkol: zjistit, v jakém pořadí je možné činnosti vykonat

Prohledávání do šířky

Breadth-First Search (BFS)

Postup zpracovává vrcholy grafu od vrcholu s v pořadí od blízkých ke vzdáleným
Postup zpracování vyžaduje označování vrcholů

- označení uložíme do pole délky $|V|$

Prohledávání do šířky

Breadth-First Search (BFS)

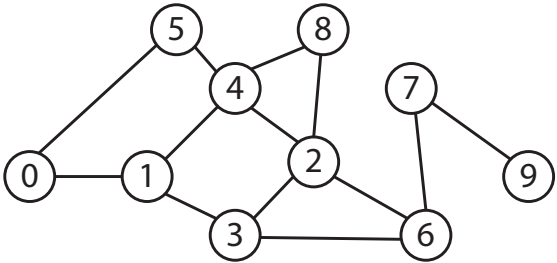
Postup zpracovává vrcholy grafu od vrcholu s v pořadí od blízkých ke vzdáleným
Postup zpracování vyžaduje označování vrcholů

- označení uložíme do pole délky $|V|$

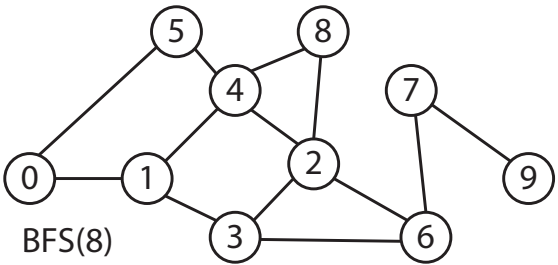
Možná označení vrcholu:

- nenavštívený (”bílý”), kód 0
- čekající na zpracování (”šedý”), kód 1
- hotový (”černý”), kód 2

Příklad



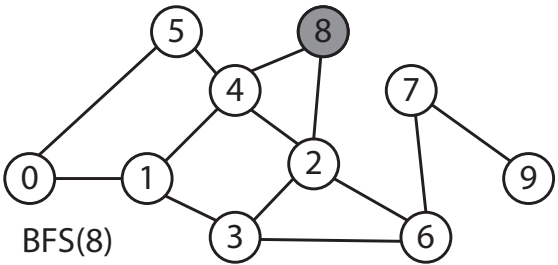
Příklad



BFS(8)

fronta:

Příklad

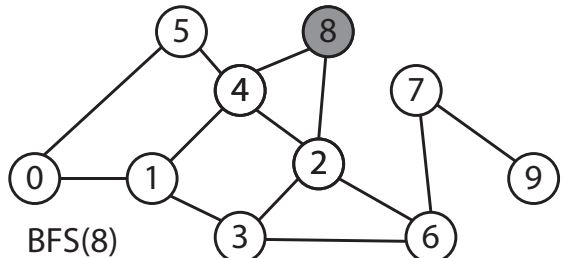


BFS(8)

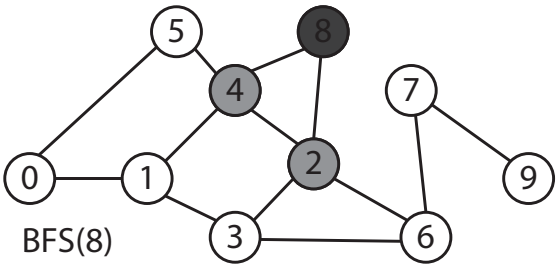
fronta:

8

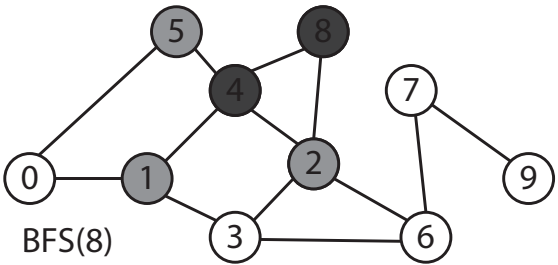
Příklad



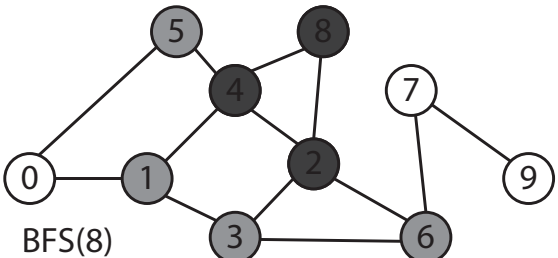
Příklad



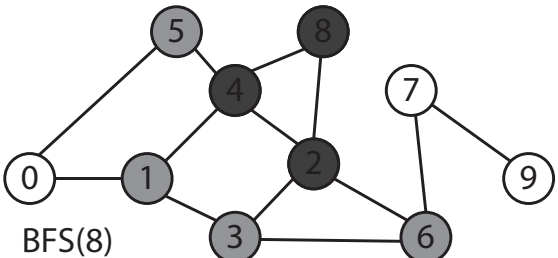
Příklad



Příklad



Příklad

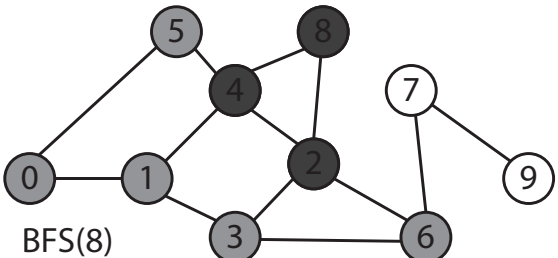


BFS(8)

fronta:

5	1	3	6	0		

Příklad

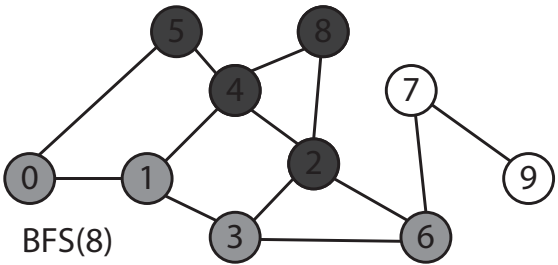


BFS(8)

fronta:

5	1	3	6	0		

Příklad

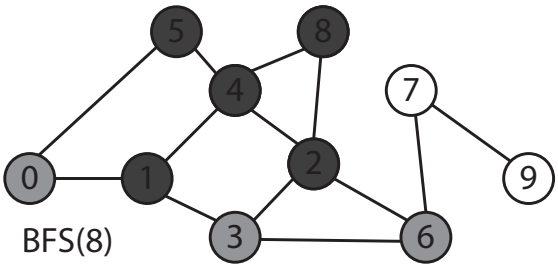


BFS(8)

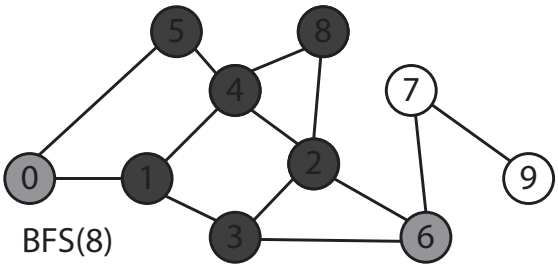
fronta:

1	3	6	0		

Příklad

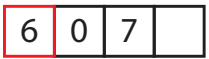


Příklad

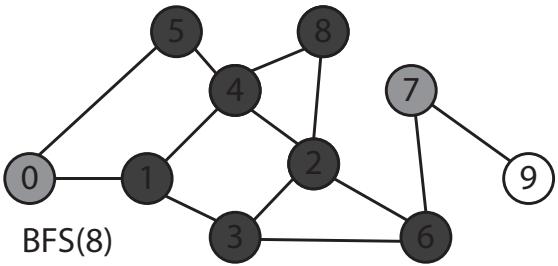


BFS(8)

fronta:



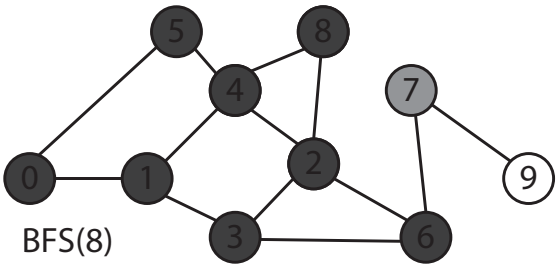
Příklad



BFS(8)

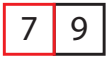
fronta:

Příklad

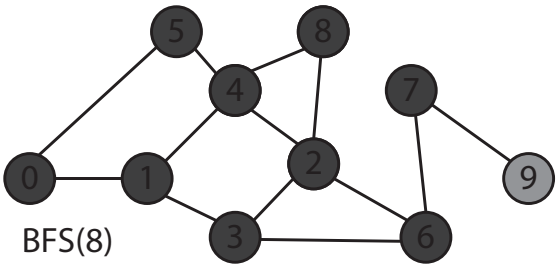


BFS(8)

fronta:



Příklad

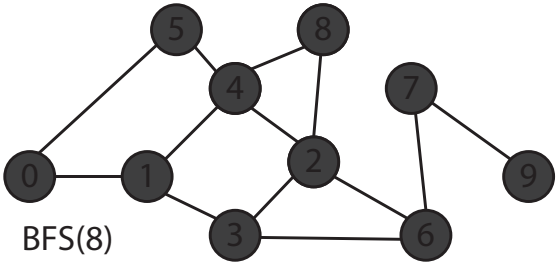


BFS(8)

fronta:

9

Příklad



BFS(8)

fronta:

Implementace

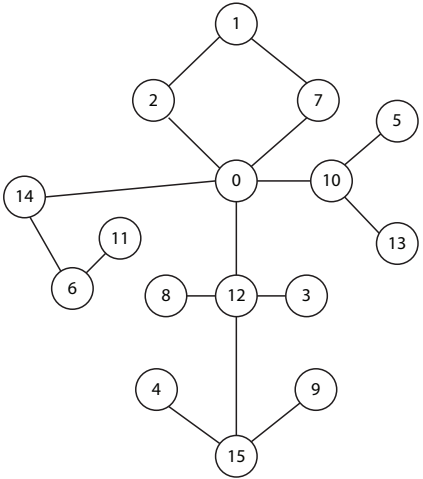
Díky použití Fronty se zpracují všchny vrcholy ve vzdálenosti k před vrcholy ve vzdálenosti $> k$

```
void BFS(int s) {
  int[] mark = new int[edges.length]; // pocet vrcholu!
  mark[s] = 1;

  Queue q = new Queue();
  q.add(s);
  ...
```

Implementace

```
  ...
  while(!q.isEmpty()) {
    int v = q.pop();
    ArrayList<Integer> nbs = neighbours(v);
    for(int i = 0;i<nbs.size();i++) {
      int n = nbs.get(i)
      if (mark[n] == 0) {
        mark[n] = 1;
        q.add(n);
      }
    }
    mark[v] = 2;
  }
}
```

Příklad



Důležitá pozorování

Pozorování

BFS v této podobě nic nedělá

Důležitá pozorování

Pozorování

BFS v této podobě nic nedělá

- je nutné doplnit „užitečný“ kód
- kam a jaký záleží na řešeném problému

Důležitá pozorování

Pozorování

BFS v této podobě nic nedělá

- je nutné doplnit „užitečný“ kód
- kam a jaký záleží na řešeném problému

Pozorování

BFS zpracuje jen jednu komponentu grafu

Zpracování všech vrcholů

- metoda BFS zpracuje jen vrcholy dosažitelné z počátečního vrcholu
- chceme-li zpracovat všechny vrcholy, musíme BFS restartovat v nezpracovaných vrcholech
- nezpracovaný vrchol se pozná podle toho, že mu zůstane bílé obarvení

```
void BFS_All() {
  int[] mark = new int[edges.length];
  for (int s = 0;s<edges.length;s++) { // smycka pres vrcholy!
    if (mark[s]!=0)
      continue; // s byl jiz zpracovan
    mark[s] = 1; // s je bily, pouzije se jako startovni vrchol

    Queue q = new Queue();
    q.add(s);
    ...
```

Složitost algoritmu BFS

- vkládání do fronty (každý vrchol je vložen do fronty právě jednou): $\Omega(|V|)$
- procházení sousedů (seznam sousedů pro každý vrchol, každá hrana je zpracována jednou):
  - implementace grafu seznamem: $\mathcal{O}(|E|)$
  - implementace grafu maticí: $\Omega(|V|^2)$

Složitost algoritmu BFS

- vkládání do fronty (každý vrchol je vložen do fronty právě jednou): $\Omega(|V|)$
- procházení sousedů (seznam sousedů pro každý vrchol, každá hrana je zpracována jednou):
  - implementace grafu seznamem: $\mathcal{O}(|E|)$
  - implementace grafu maticí: $\Omega(|V|^2)$

Celkem:

- úplný, popř. hustý graf nebo implementace sousednosti maticí: $\Omega(|V|^2)$
- implementaci sousednosti seznamem: $\mathcal{O}(|V| + |E|))$
  - graf může mít počet hran až $|V|^2$

Aplikace

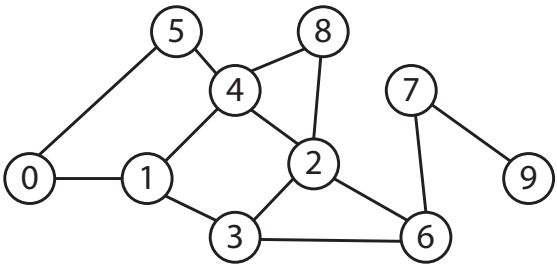
Určení všech vzdáleností od vrcholu s

```
int[] BFSDistance(int s) {
  int[] result = new int[edges.length];
  for (int i = 0;i<edges.length;i++)
    result[i] = -1;
  result[s] = 0;

  int[] mark = new int[vertices.length];
  mark[s] = 1;

  Queue q = new Queue();
  q.add(s);
  ...
```

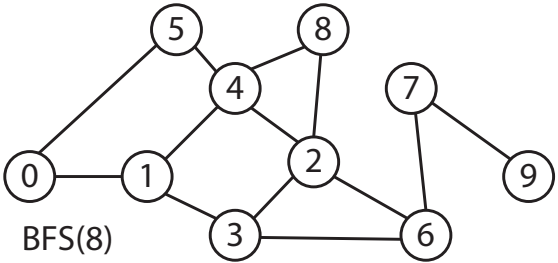
Implementace

```
  ...
  while(!q.isEmpty()) {
    int v = q.pop();
    ArrayList<Integer> nbs = neighbours(v);
    for(int i = 0;i<nbs.size();i++) {
      int n = nbs.get(i)
      if (mark[n] == 0) {
        mark[n] = 1;
        q.add(n);
        result[n] = result[v]+1;
      }
    }
    mark[v] = 2;
  }
  return result;
}
```

Příklad



Příklad



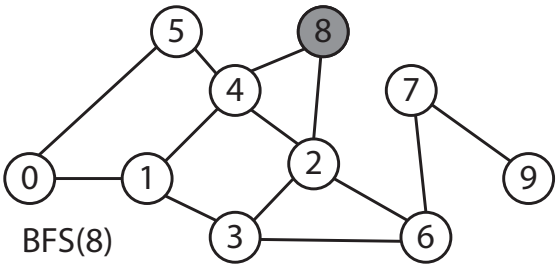
BFS(8)

fronta:

výsledek:

0	1	2	3	4	5	6	7	8	9

Příklad



BFS(8)

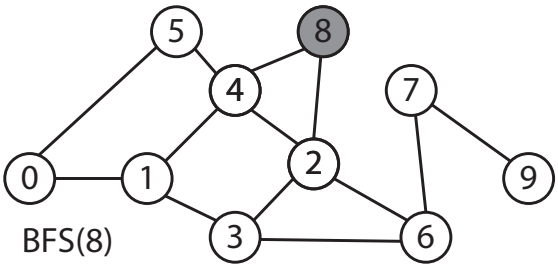
fronta:

8

výsledek:

0	1	2	3	4	5	6	7	8	9
								0	

Příklad



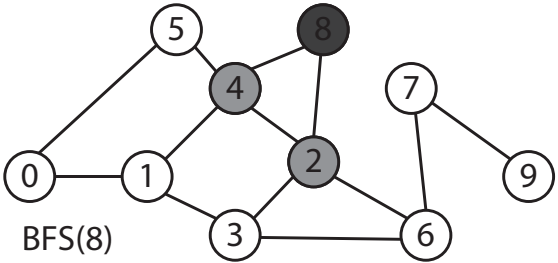
fronta:

8	4	2							

výsledek:

0	1	2	3	4	5	6	7	8	9
		1		1				0	

Příklad



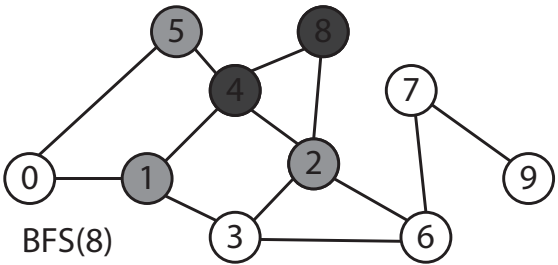
fronta:

4	2	5	1					

výsledek:

0	1	2	3	4	5	6	7	8	9
	2	1		1	2			0	

Příklad



BFS(8)

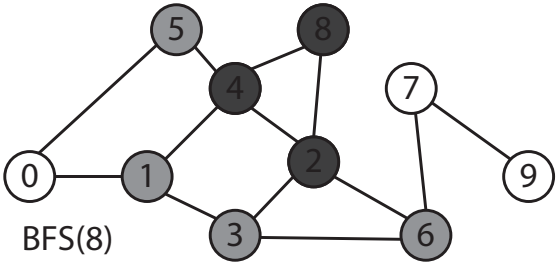
fronta:

2	5	1	3	6			

výsledek:

0	1	2	3	4	5	6	7	8	9
	2	1	2	1	2	2		0	

Příklad



BFS(8)

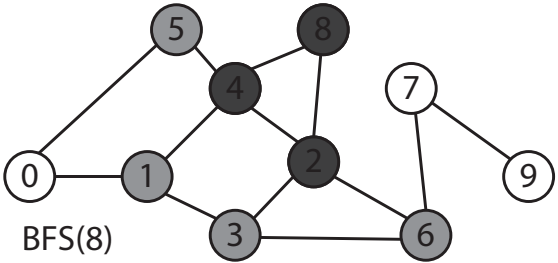
fronta:

2	5	1	3	6			

výsledek:

0	1	2	3	4	5	6	7	8	9
	2	1	2	1	2	2		0	

Příklad



BFS(8)

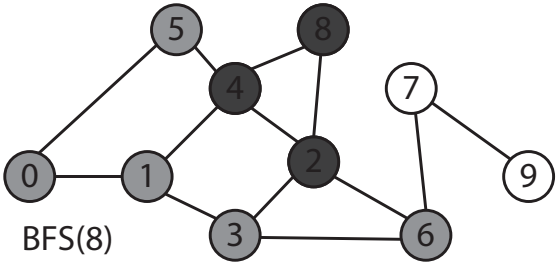
fronta:

5	1	3	6	0		

výsledek:

0	1	2	3	4	5	6	7	8	9
3	2	1	2	1	2	2		0	

Příklad



BFS(8)

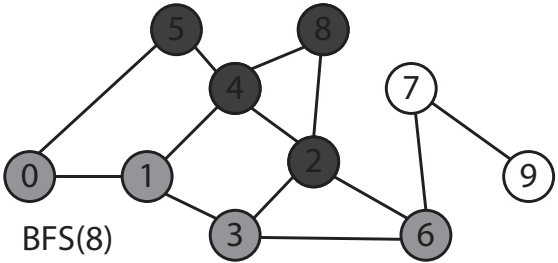
fronta:

5	1	3	6	0		

výsledek:

0	1	2	3	4	5	6	7	8	9
3	2	1	2	1	2	2		0	

Příklad



BFS(8)

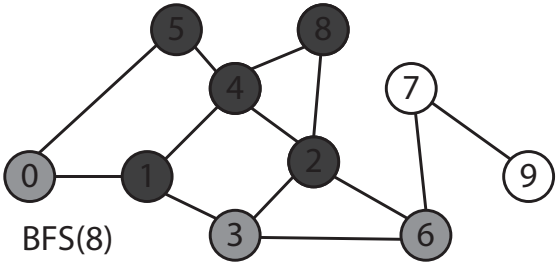
fronta:

1	3	6	0		

výsledek:

0	1	2	3	4	5	6	7	8	9
3	2	1	2	1	2	2		0	

Příklad



BFS(8)

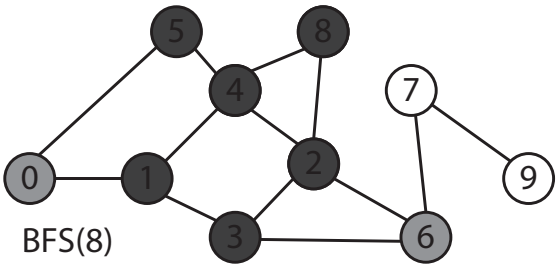
fronta:

3	6	0		

výsledek:

0	1	2	3	4	5	6	7	8	9
3	2	1	2	1	2	2		0	

Příklad



BFS(8)

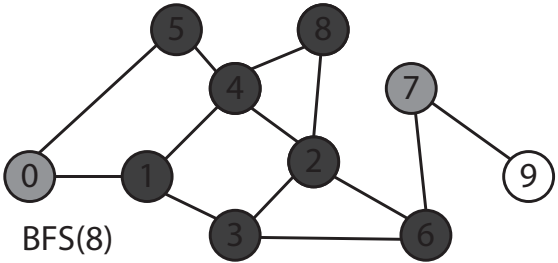
fronta:

6	0	7	

výsledek:

0	1	2	3	4	5	6	7	8	9
3	2	1	2	1	2	2	3	0	

Příklad



BFS(8)

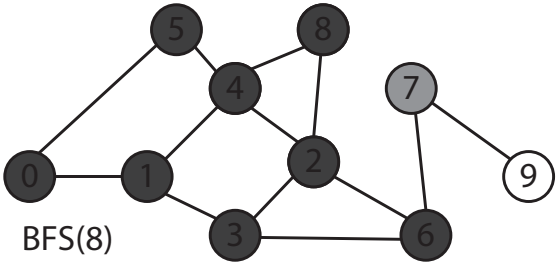
fronta:

0	7	

výsledek:

0	1	2	3	4	5	6	7	8	9
3	2	1	2	1	2	2	3	0	

Příklad



BFS(8)

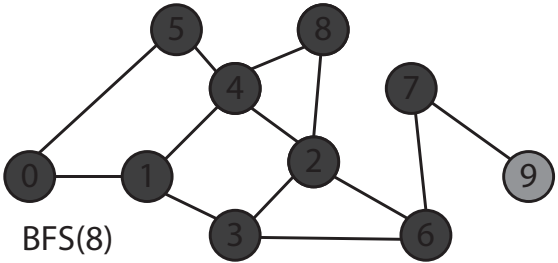
fronta:

7	9

výsledek:

0	1	2	3	4	5	6	7	8	9
3	2	1	2	1	2	2	3	0	4

Příklad



BFS(8)

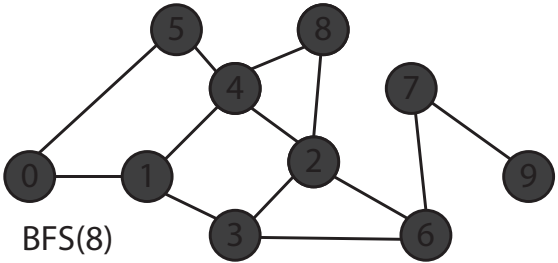
fronta:

9

výsledek:

0	1	2	3	4	5	6	7	8	9
3	2	1	2	1	2	2	3	0	4

Příklad



BFS(8)

fronta:

výsledek:

0	1	2	3	4	5	6	7	8	9
3	2	1	2	1	2	2	3	0	4

Použití algoritmu

Zjištění zda existuje cesta z s do t:

```
boolean isPath(int s, int t){
  int[] d = BFSDistance(s);
  return (d[t]>=0);
}
```

Použití algoritmu

Zjištění zda existuje cesta z *s* do *t*:

```
boolean isPath(int s, int t){
  int[] d = BFSDistance(s);
  return (d[t]>=0);
}
```

Určení délky nejkratší cesty z *s* do *t*:

```
int distance(s, t){
  int[] d = BFSDistance(s);
  return d[t];
}
```

Použití algoritmu

Zjištění zda existuje cesta z *s* do *t*:

```
boolean isPath(int s, int t){
  int[] d = BFSDistance(s);
  return (d[t]>=0);
}
```

Určení délky nejkratší cesty z *s* do *t*:

```
int distance(s, t){
  int[] d = BFSDistance(s);
  return d[t];
}
```

šlo by významně zrychlit předáním *t* do BFS a ukončením smyčky ve chvíli, kdy je *t* přiřazena vzdálenost

BFSDistanceTo

Určení vzdálenosti od vrcholu s

```
int BFSDistanceTo(int s, int t) {
  int[] result = new int[edges.length];
  for (int i = 0;i<edges.length;i++)
    result[i] = -1;
  result[s] = 0;

  int[] mark = new int[vertices.length];
  mark[s] = 1;

  Queue q = new Queue();
  q.add(s);
  ...
```

BFSDistanceTo

```
  ...
  while(!q.isEmpty()) {
    int v = q.pop();
    ArrayList<Integer> nbs = neighbours(v);
    for(int i = 0;i<nbs.size();i++) {
      int n = nbs.get(i)
      if (mark[n] == 0) {
        mark[n] = 1;
        q.add(n);
        result[n] = result[v]+1;
        if (n == t)
          return result[t];
      }
    }
    mark[v] = 2;
  }
  return -1;
}
```

Zjištění počtu komponent

- přidáme užitečný kód do `BFS_All()`

Zjištění počtu komponent

- přidáme užitečný kód do `BFS_All()`

```
int componentCount() {
  int result = 0;
  int[] mark = new int[edges.length];
  for (int s = 0;s<edges.length;s++) {
    if (mark[s]!=0)
      continue; // s byl jiz zpracovan
    result++; // nova komponenta
    mark[s] = 1; // s je bily, pouzije se jako startovni vrchol

    Queue q = new Queue();
    q.add(s);
    ...
  }
  return result;
}
```

Příště

Pokračování grafů

- další aplikace prohledávání do šířky (strom dostupnosti)
- prohledávání do hloubky
- topologické řazení