DATOVÉ STRUKTURY – GRAFY

Kurz: **Datové struktury a algoritmy**

Lektor: Doc. Ing. Radim Burget, Ph.D.

Autor: Doc. Ing. Radim Burget, Ph.D.

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Vytvoření této videopřednášky bylo podpořeno projektem č. CZ.1.07/2.2.00/28.0098
Evropského sociálního fondu a státním rozpočtem České republiky.

Jak lidé interně reprezentují informaci?

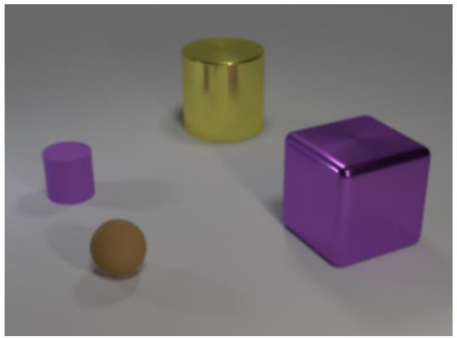
- Již víme, že pracovat s lineárními a stromovými strukturami je poměrně časově výhodné

- Stačí nám lineární struktury a stromy pro popis veškeré informace?

  X

- Je výpočetně časově srovnatelné pracovat s grafy

Skutečný svět



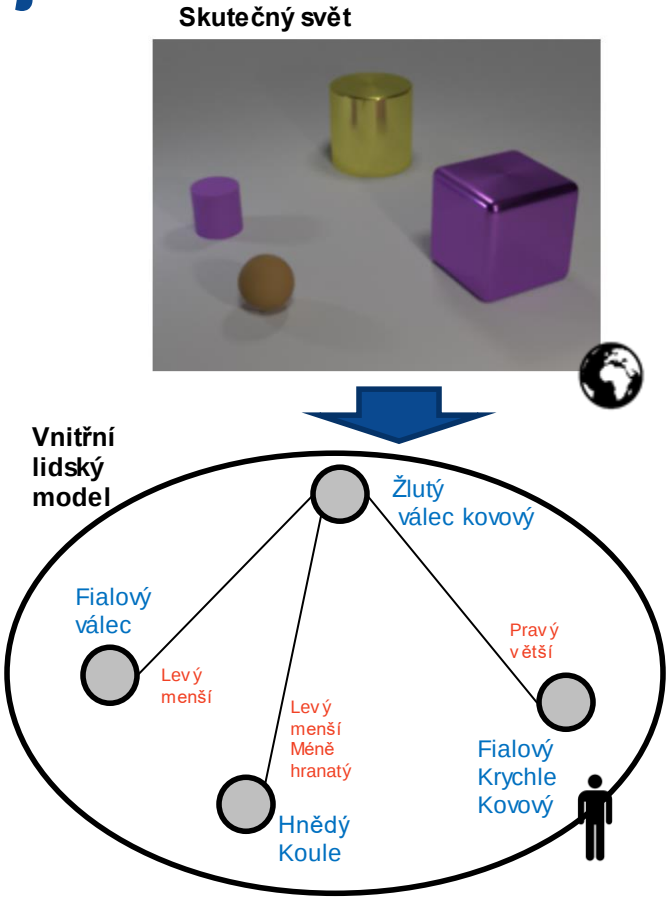
Motivace: Jak lidé interně reprezentují informaci?

- Z teorie psychologie:

  Vnitřní lidský model reprezentace informace

  - Rozklad na entity
    - + atributy
  - Relace mezi entitami
    - + atributy

Počítačová věda:
Multi-graph
+
atributy: $V \cup E \rightarrow A$

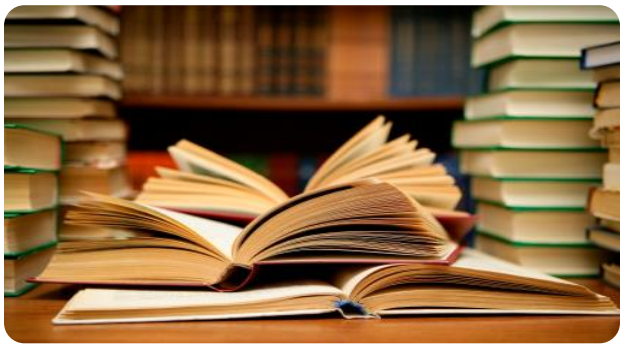


Cíl přednášky

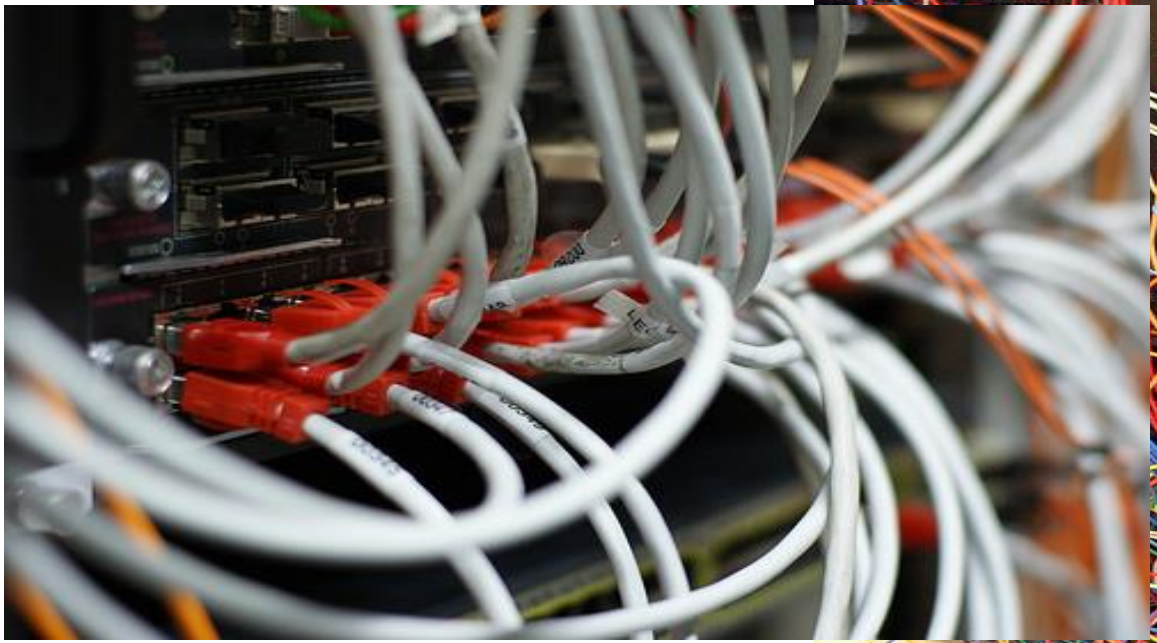
I. Teorie grafů – definice a členění typů

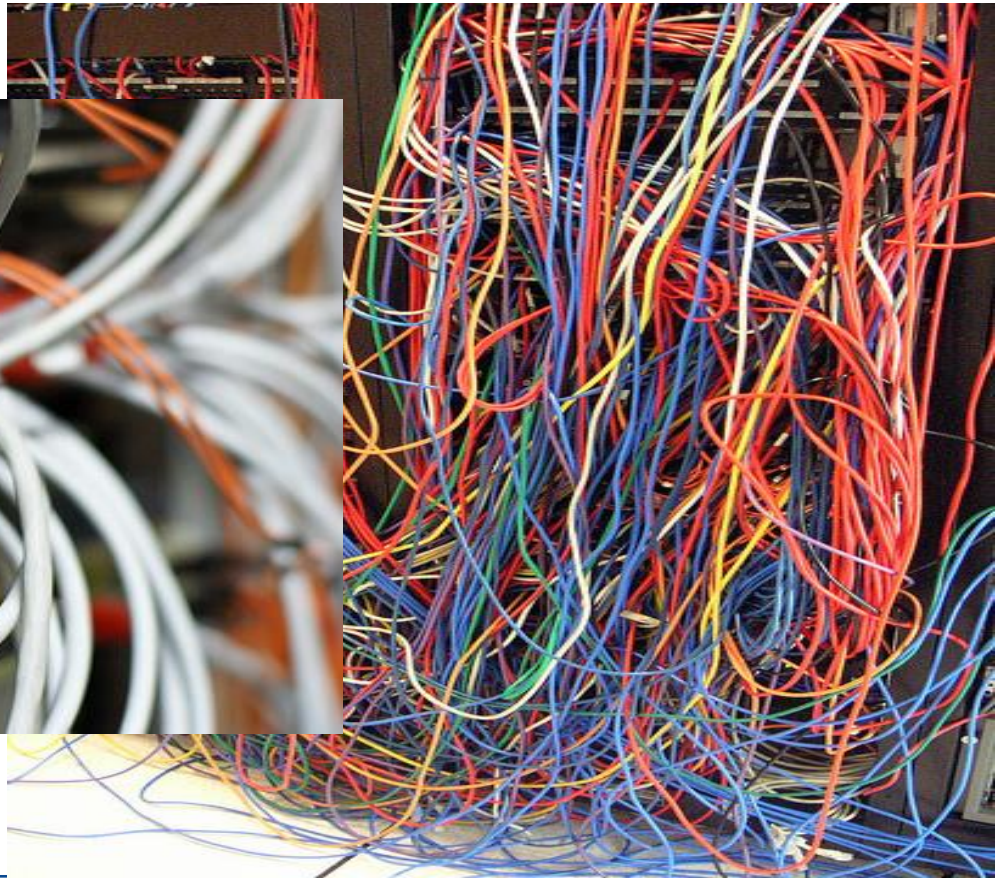
II. Převod grafu na stromovou strukturu = hledání kostry grafu (Spanning Tree)

- Centralizovaný algoritmus
- Distribuovaný algoritmus (sítě)



Teorie grafů



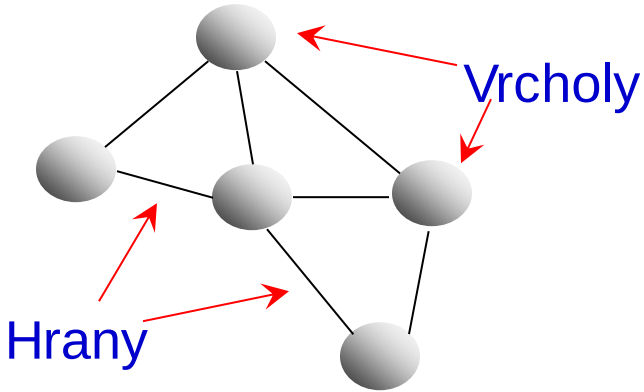


Některé aplikace teorie grafů

- Komunikační modely – veškeré **přepínače**, **směrovače** (OSPF, IS-IS, SpanningTree)
- Elektronické obvody
- Elektrické sítě
- Modely pro počítačové architektury
- Analýza konečných automatů
- Rozbor a optimalizace kódu v překladačích
- Optimalizační síťové modely pro funkci, analýzu a plánování a přidělení úkolů
- Umělá inteligence

Aplikace na Ad Hoc sítě

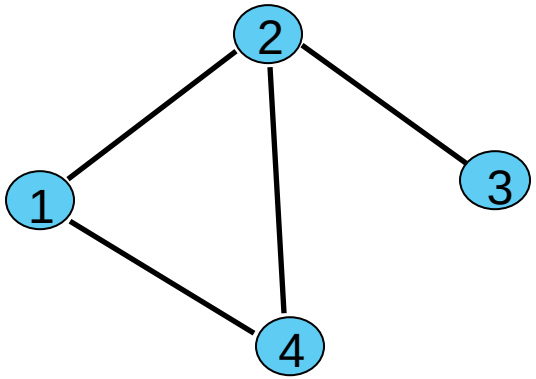
- Jakákoli síť může být reprezentována grafem
- Mobilní uzly jsou vrcholy grafu
- Komunikační linky jsou hrany



- Směrovací protokoly často používají algoritmus nejkratší cesty (OSPF, IS-IS)

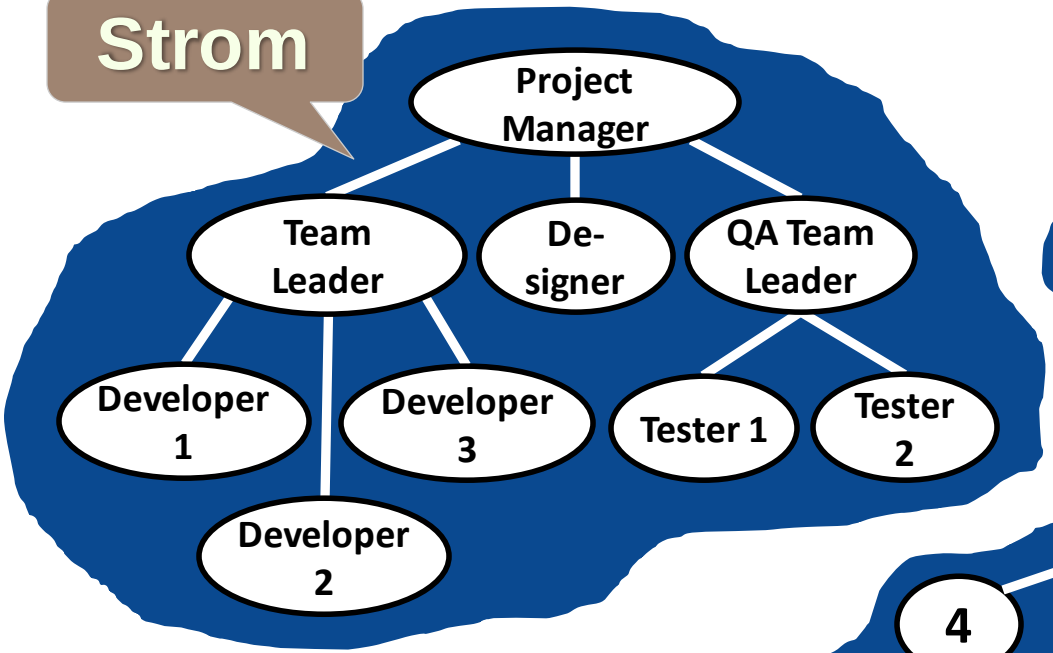
Graf: definice

- Graf je definován jako *G(V,E)* (uspořádaná dvojice množiny vrcholů V a množiny hran E)
  - Vrcholy (uzly), množina V
  - Hrany (spoje), množina E, kde $E \rightarrow V \times V$

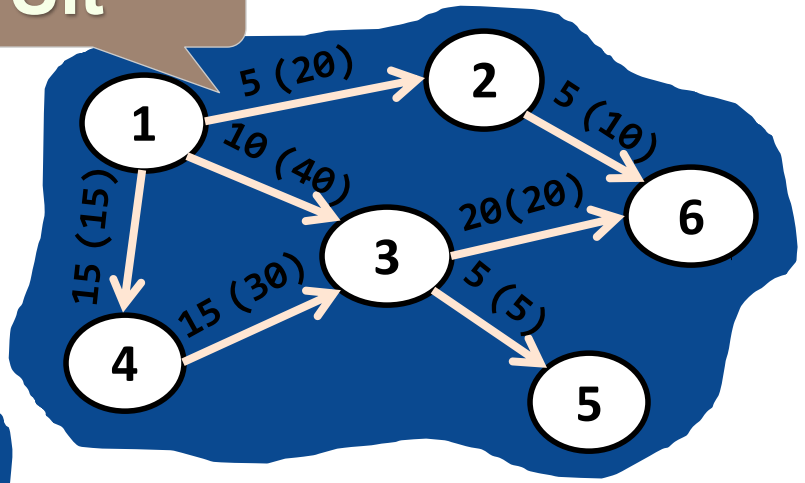


Grafy a jejich podoby

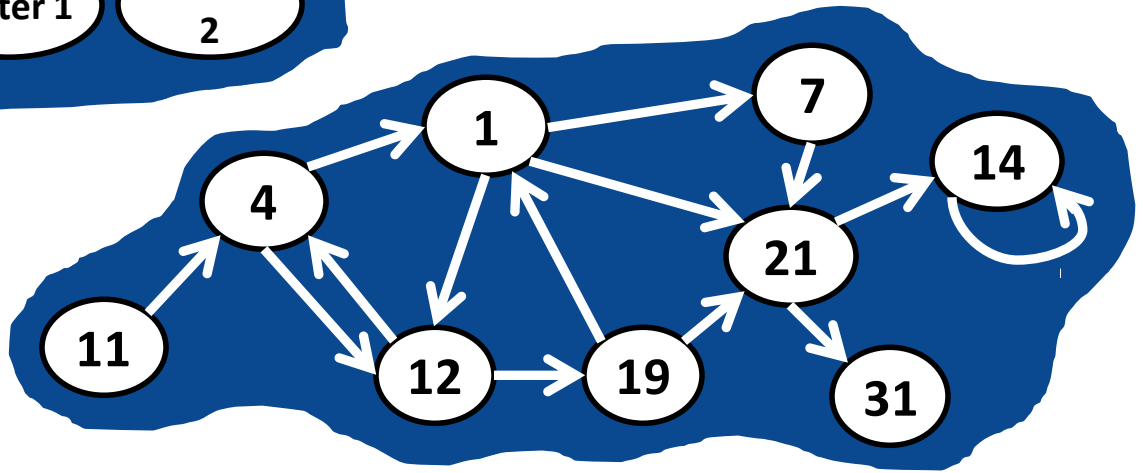
Strom



Síť



Graf



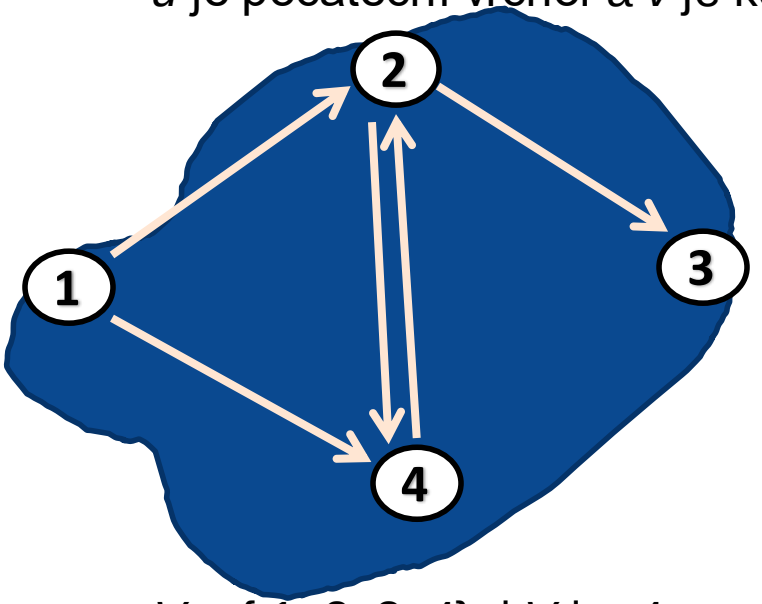
Grafy ↔ Sítě

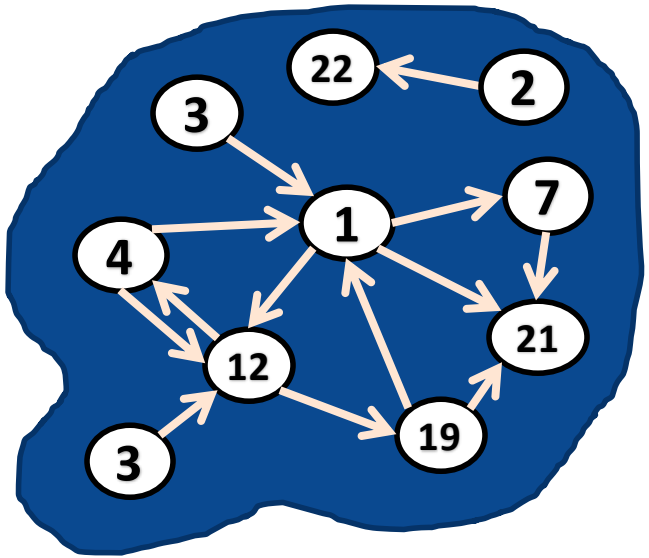
Grafy	Vrcholy	Hrany	Tok
Komunikace	Ústředny, telefony, satelity, směrovače, přepínače	Kabely, optická vlákna, mikrovlnné přenosy	Hlas, video, pakety
Obvody	Brány, registry, procesory	Dráty	El. proud
Mechanické	Spoje	Tyče, nosníky, pružiny	Teplo, energie
Hydraulika	Nádrže, přečerpávací stanice, jezera	Potrubí	Tekutiny, olej
Finančnictví	Burzy, měny	Transakce	Peníze
Doprava	Letiště, železnice, přestupní místa, křižovatky	dálnice, cesty, letecké koridory	Auta, letadla, cestující

Orientovaný graf

Hrana $e \in E$ orientovaného grafu je reprezentována neorientovanou uspořádanou dvojicí (u,v), kde $u, v \in V$.

u je počáteční vrchol a v je koncový vrchol.



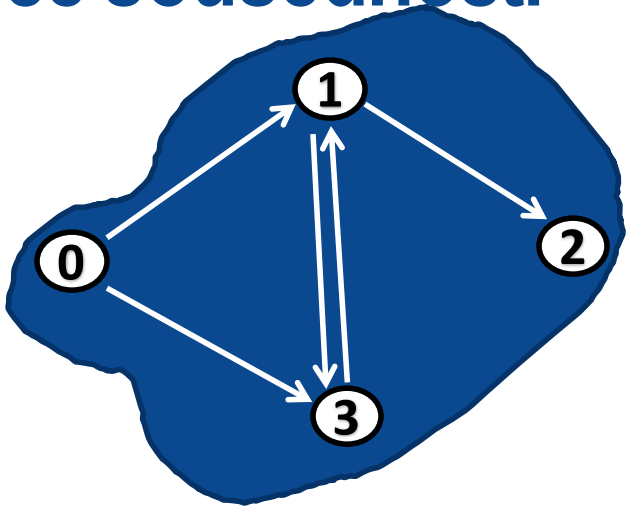


$V = \{ 1, 2, 3, 4\}$, $| V | = 4$

$E = \{(1,2), (1,4), (2,3), (2,4), (4,2)\}$, $| E | = 5$

Hustá reprezentace grafu: matice sousedností

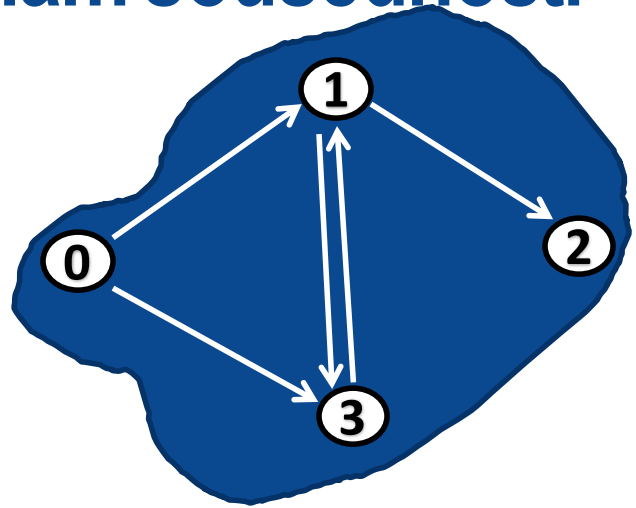
```
public class Graph
{
  private int[][] childNodes;
  public Graph(int[][] nodes)
  {
    this.childNodes = nodes;
  }
}
```




```
Graph g = new Graph(new int[][] {
  {0, 1, 0, 1}, // successors of vertice 1
  {0, 0, 1, 1}, // successors of vertice 2
  {0, 0, 0, 0}, // successors of vertice 3
  {0, 1, 0, 0}, // successors of vertice 4
});
```

Řídká reprezentace grafu: seznam sousedností

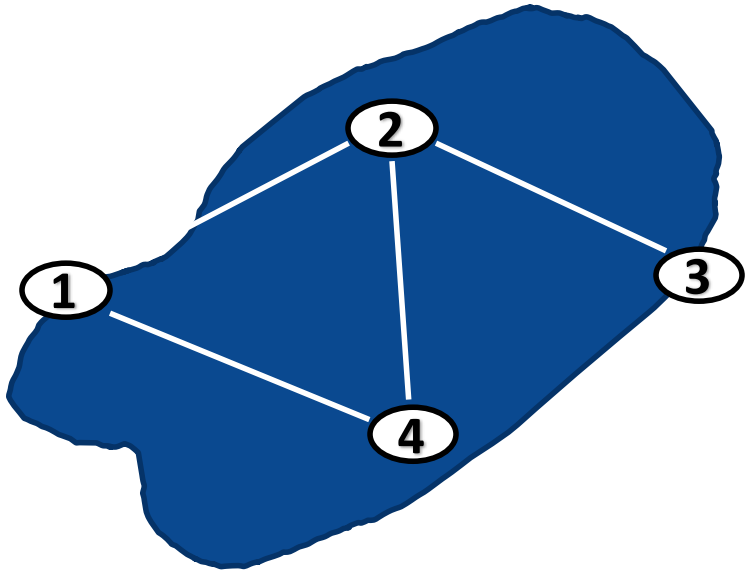
```
public class Graph
{
  private List<Vertex> vertices[];
}
```

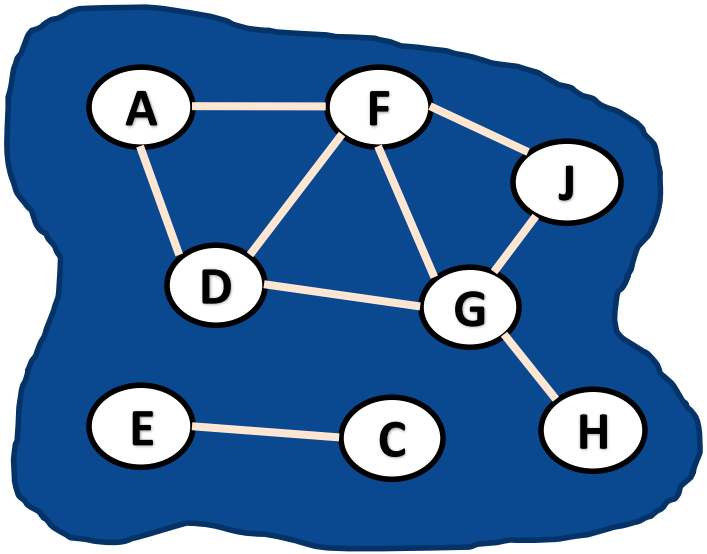



```
vertices[0] = new LinkedList(); vertices[0].add(1); vertices[0].add(3);
vertices[1] = new LinkedList(); vertices[1].add(2); vertices[1].add(3);
vertices[2] = new LinkedList();
vertices[3] = new LinkedList(); vertices[3].add(1);
```

Neorientovaný graf

Hrana $e \in E$ neorientovaného grafu je reprezentována **ne**uspořádanou dvojicí $(u,v)=(v,u)$, kde $u, v \in V$.





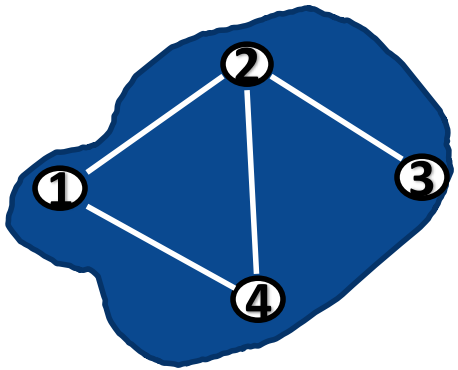
$V = \{ 1, 2, 3, 4\}, |V| = 4$

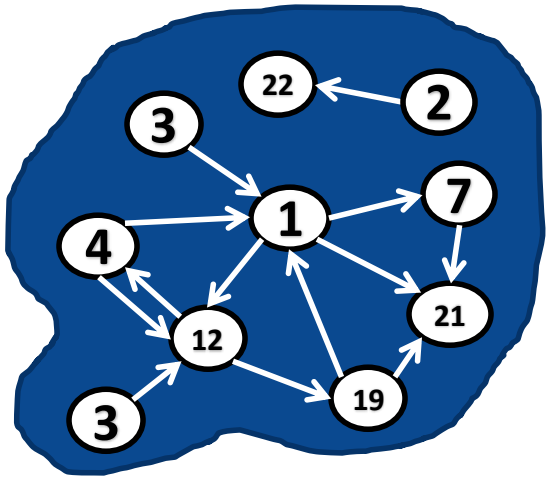
$E = \{(1,2), (2,3), (2,4), (4,1)\}, |E| = 4$

Stupeň vrcholu

Stupeň vrcholu je počet hran vycházející z vrcholu. V orientovaném grafu. V orientovaných grafech rozlišujeme *vstupní* a *výstupní stupeň*.

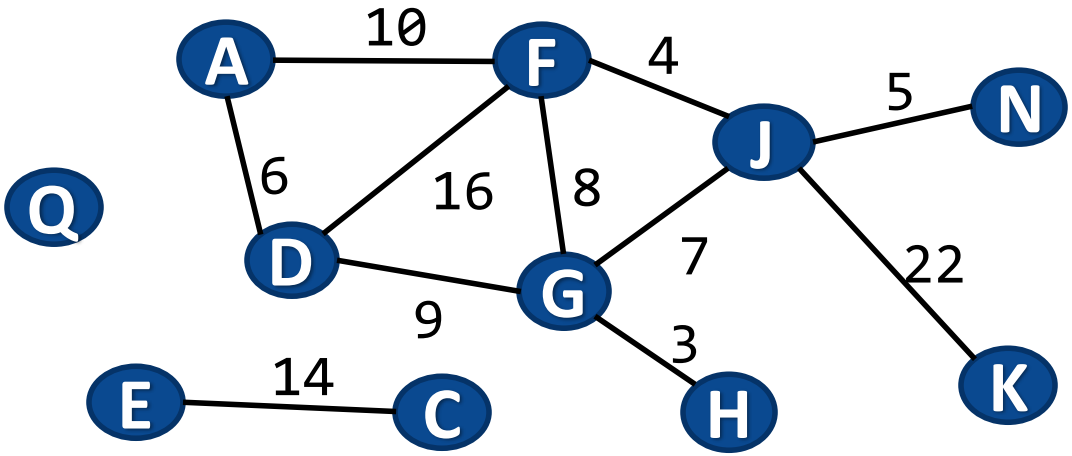
Stupeň grafu odpovídá nejvyšší hodnotě stupně vrcholu v grafu G.

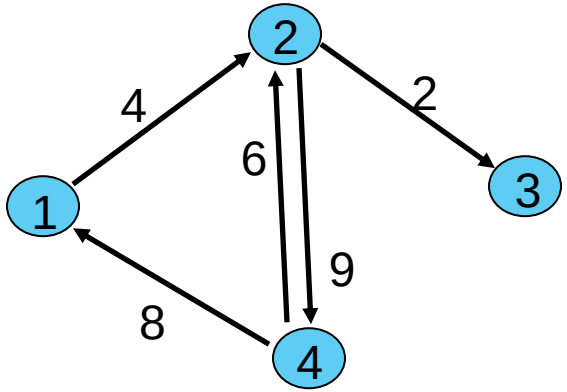




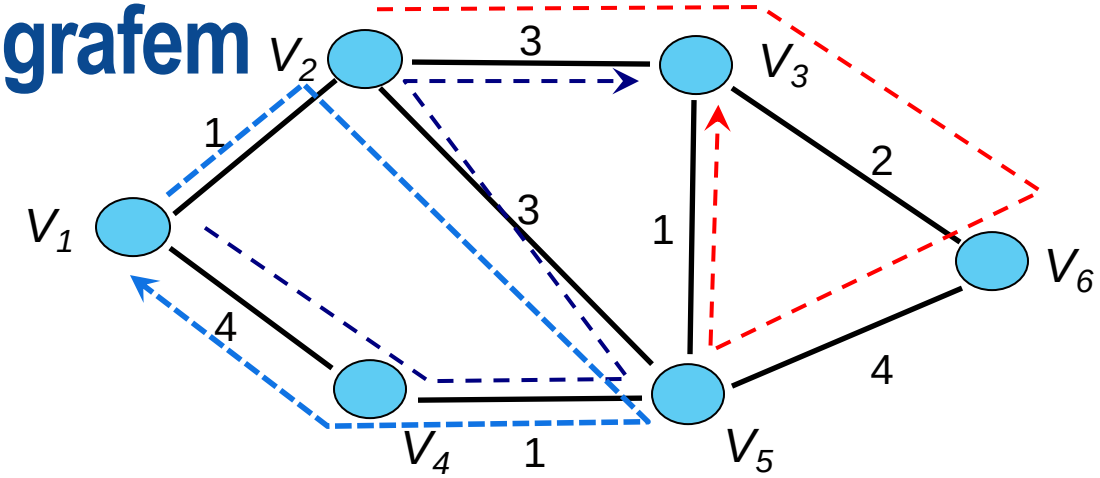
Vážený graf

- *Vážený graf* je graf, kde každé hraně *e* má přidělenou *váhu*, obvykle je dána *váhovou funkcí* $w: E \rightarrow \mathcal{R}$





Průchody a cesty grafem



Průchod je sekvence vrchlů ($v_1, v_2,..., v_L$) takových, že $\{(v_1, v_2), (v_2, v_3),..., (v_{L-1}, v_L)\} \subseteq E$, *např.* ($V_2, V_3, V_6, V_5, V_3$)

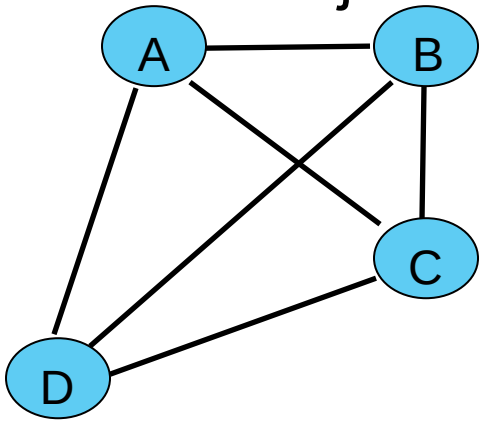
Jednoduchá cesta je průchod bez opakování vrcholů, např. (V_1, V_4, V_5, V_2, V_3)

Cyklus je průchod ($v_1, v_2,..., v_L$) kde $v_1=v_L$ (první = poslední) bez opakování uzlů a $L>=3$, například (V_1, V_2, V_5, V_4, V_1). Cyklus není cesta.

Graf je nazývaný *cyklický* jestliže obsahuje alespoň jeden cyklus; jinak se nazývá *acyklický*

Kompletní grafy

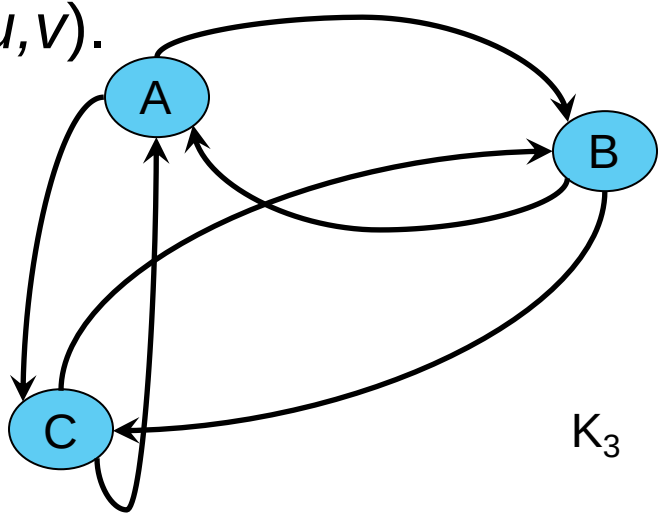
- Kompletní graf je orientovaný či neorientovaný graf kde pro každé dva uzly *u*, *v* grafu platí, že jsou přilehlé (v množině E existuje hrana mezi uzly *u*,*v*).



K_4

4 uzly a (4*3)/2 hran

|*V*| uzlů a |*V*|*(|*V*|-1)/2 hrany



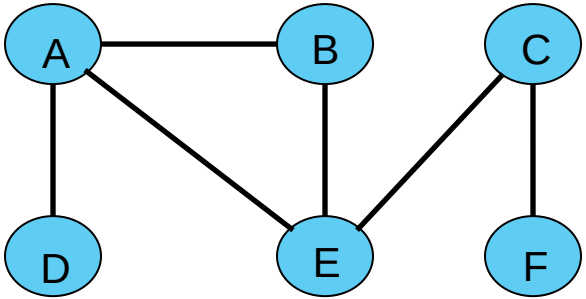
K_3

3 uzly a 3*2 hrany

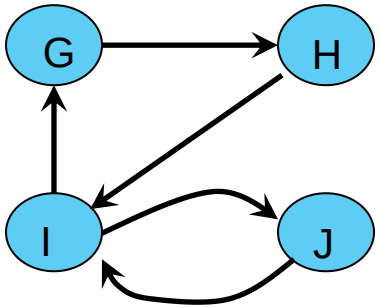
|*V*| uzly a |*V*|*(|*V*|-1) hran

Spojité grafy

- Neorientovaný graf je spojitý, jestliže pro libovolné dva vrcholy u,v existuje cesta z u do v.



- Orientovaný graf je *pevně spojitý* jestliže pro libovolné dva vrcholy u,v existuje orientovaná cesta



  - Graf je *řídký* jestliže $| E | \approx | V |$
  - Graf je *hustý* jestliže $| E | \approx | V |^2$

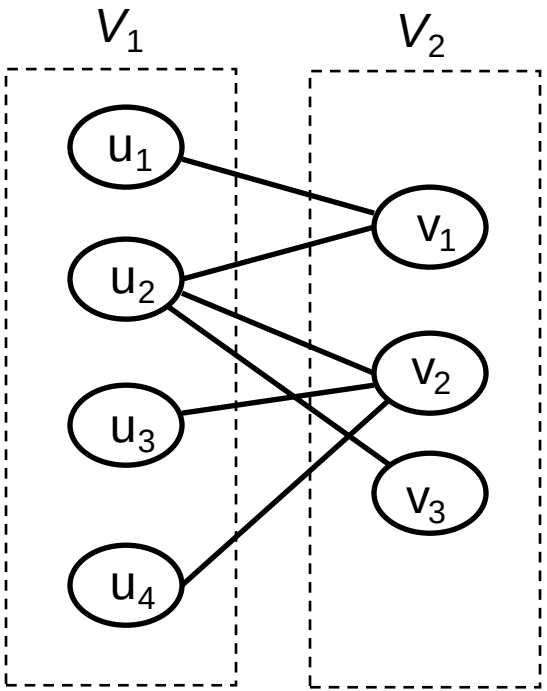
Bipartitní graf - definice

bipartitní graf je neorientovaný graf $G = (V,E)$ kde V mohou být rozděleny do dvou vzájemně disjunktních množin V_1 a V_2, kde $(u,v) \in E$ znamená současně:

$u \in V_1$ a $v \in V_2$

anebo

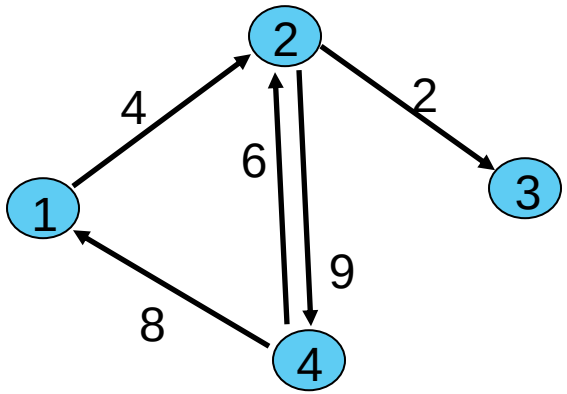
$v \in V_1$ a $u \in V_2$.



Příklad: reprezentujte v paměti počítače

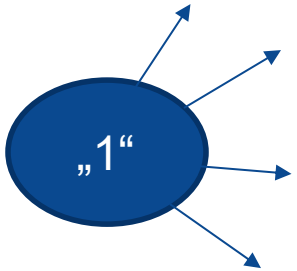
- Demo: Eclipse

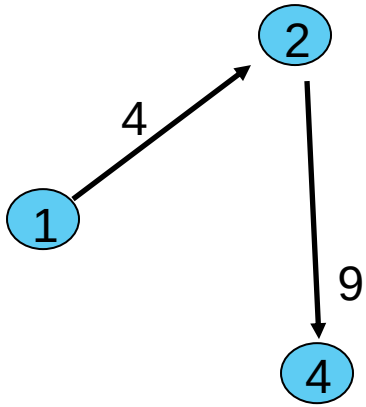
- Reprezentujte v paměti počítače



Objekt String „1“

Objekt Uzel:

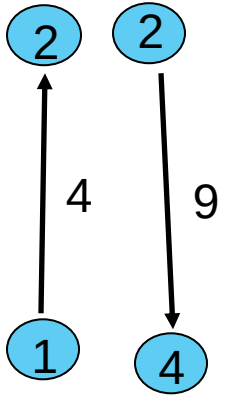




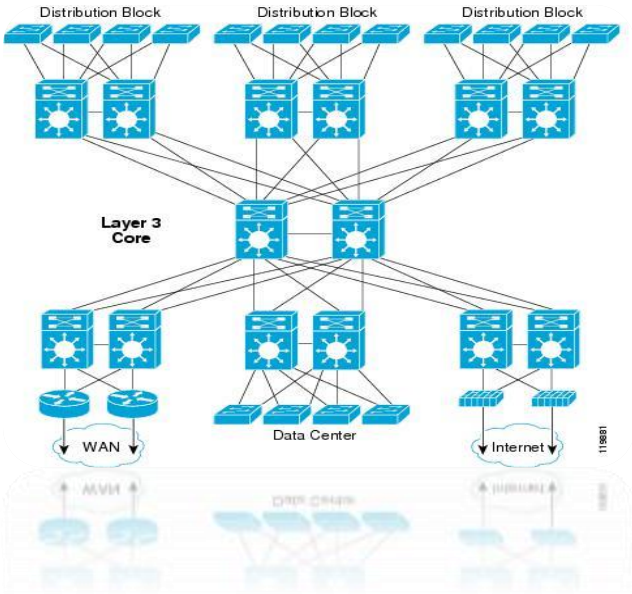

```
MyGraph g = new MyGraph();
g.pridej("1", "2", 4);
g.pridej("2", "4", 9);
```

Jak zajistit abychom neměli duplicitní uzly?

Jak mapovat řetězec „1“ na objekt Uzel?

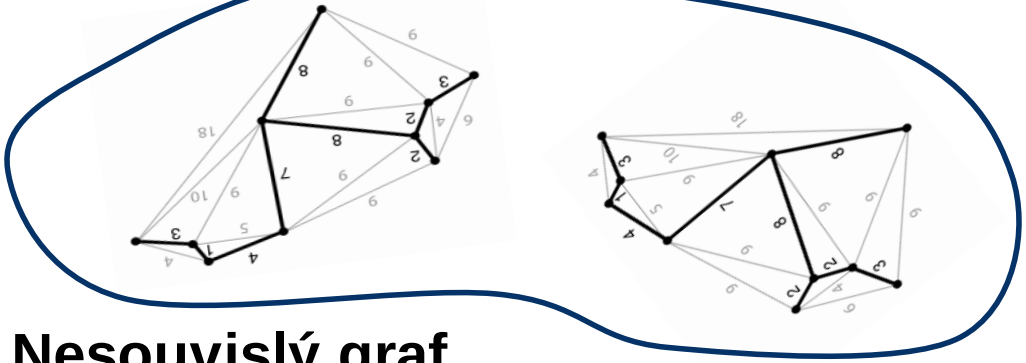


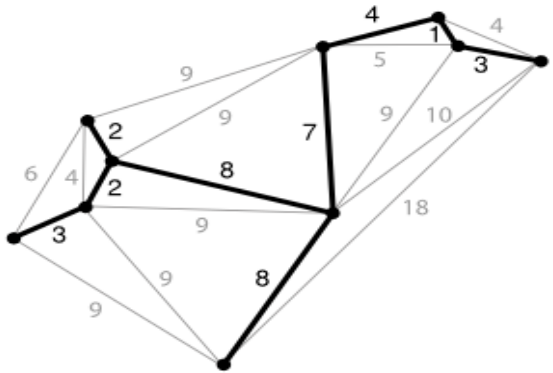
Nalezení kostry grafu (Spanning tree)



Kostra grafu (Spanning Tree)

- Kostra grafu = strom, spojující všechny vrcholy, (neobsahuje cykly, protože je to strom)
- **U souvislého grafu**
  - hledá podmnožinu hran, která tvoří strom (obsahující všechny uzly)
  - celková váha (součet délek) hran grafu je minimální.

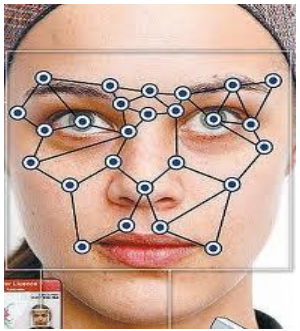




- **Nesouvislý graf**
  - V případě grafu o více komponentách, algoritmus hledá les minimálních koster, tedy minimální kostru každé komponenty

Uplatnění

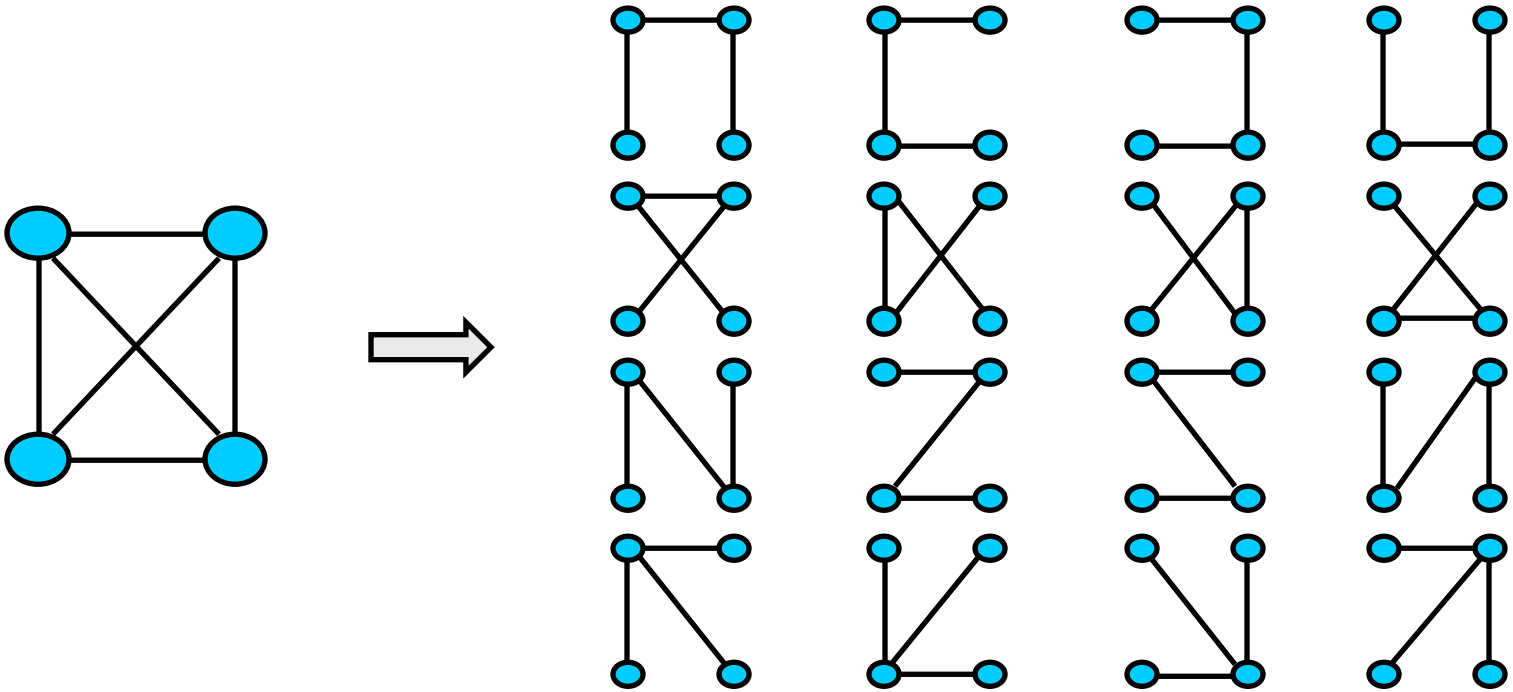
- **Odstranění cyklů z grafu** (např. přepínač)
- **Návrh sítí (nejen komunikačních)**
  - telefonické, elektrické, hydraulické, TV kabelové, počítačové, silniční, …
- **Analýza shluků**
  - Hledání shluků kvazarů a Seyfert galaxií
  - analyzování plísní prostorových modelů



- **Přibližná řešení tzv. NP-těžkých problémů**
  - Metrický TSP (Traveling Salesman Problem), Steiner stromy
- **Nepřímá uplatnění**
  - rakovinový výzkum
  - učení charakteristických rysů pro real-time verifikaci tváře
  - Modelování lokálnosti interakce částic turbulentního toku kapalin
  - snižuje množství ukládaných dat pro popis aminokyselin v proteinu

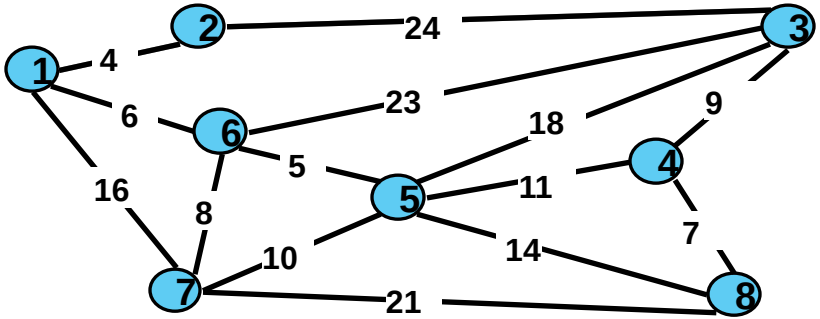
Kostra grafu = Spanning Tree

•Následující graf může mít několik možných koster grafu

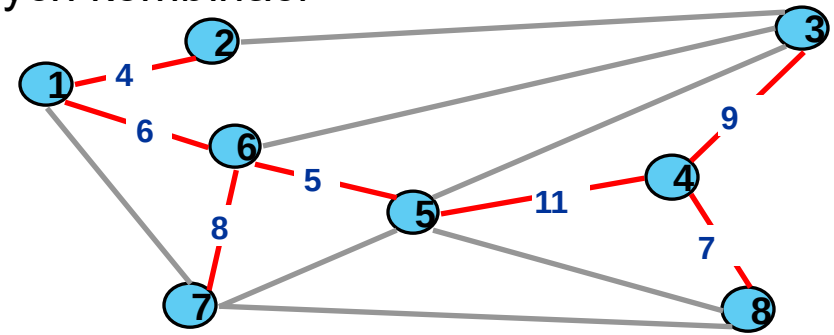


Minimální kostra grafu (Minimum Spanning Tree)

- Ohodnocené hrany grafu
- Suma hran je nejmenší ze všech možných kombinací



$G = (V, E)$



$T = (V, F)$ $w(T) = 50$

Cayley's Theorem (1889)

Pro kompletní graf K_n máme n^{n-2} koster grafu:

- $n = |V|, m = |E|$
- Nelze řešit hrubou silou - možností: n^{n-2}

Algoritmy pro nalezení kostry grafu (spanning tree)

- **Snaha nelézt minimální kostru grafu (Minimal Spanning Tree)**

- **Centralizované algoritmy**
  - Primův algoritmus (nebude probírán)
  - Borůvkův algoritmus (nebude probírán)
  - **Kruskalův algoritmus**

- **Distribuované algoritmy**
  - **Gallager-Humblet**

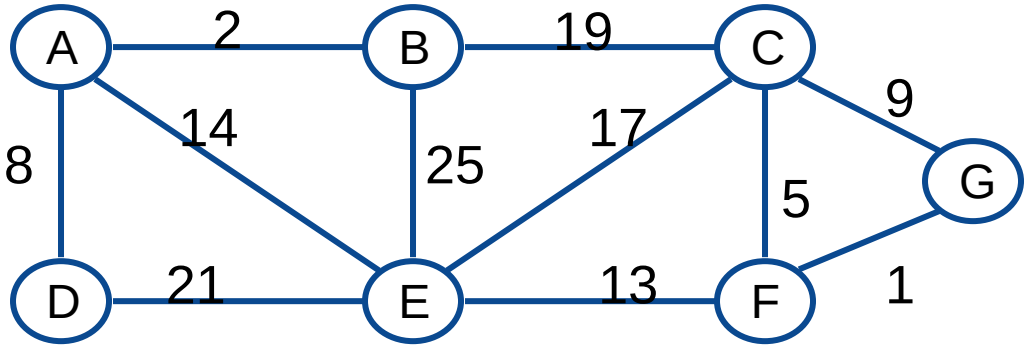
Kruskalův algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        VytvorMn(v);
    seřaď E vzestupně dle váhy hrany w
    for each (u,v) ∈ E (dle seřazení)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskalův algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        VytvorMn(v);
    seřaď E vzestupně dle váhy hrany w
    for each (u,v) ∈ E (dle seřazení)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

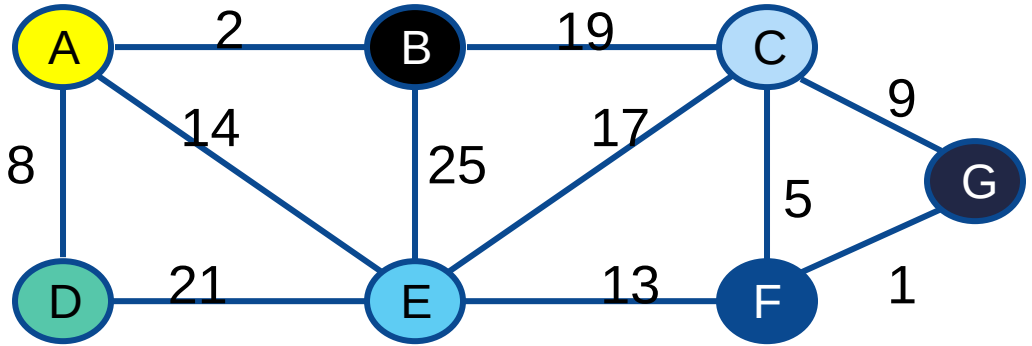
Běh algoritmu:



Kruskalův algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        VytvorMn(v);
    seřaď E vzestupně dle váhy hrany w
    for each (u,v) ∈ E (dle seřazení)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

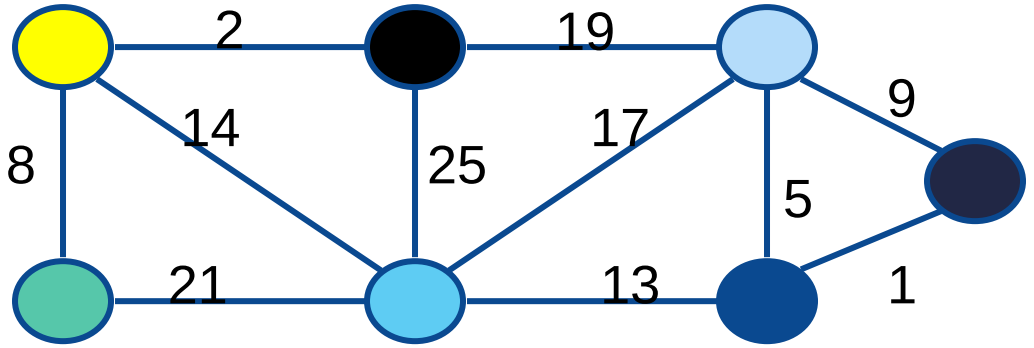
Běh algoritmu:



Kruskalův algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        VytvorMn(v);
    seřaď E vzestupně dle váhy hrany w
    for each (u,v) ∈ E (dle seřazení)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

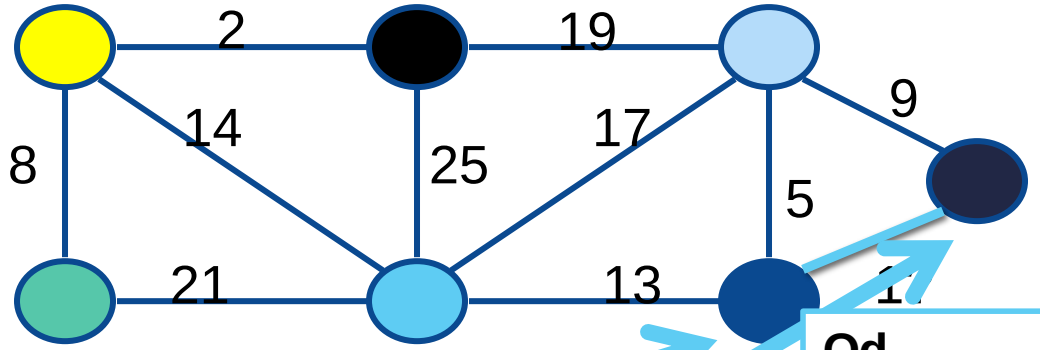
Běh algoritmu:



Kruskalův algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        VytvorMn(v);
    seřaď E vzestupně dle váhy hrany w
    for each (u,v) ∈ E (dle seřazení)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet
}
```

Běh algoritmu:



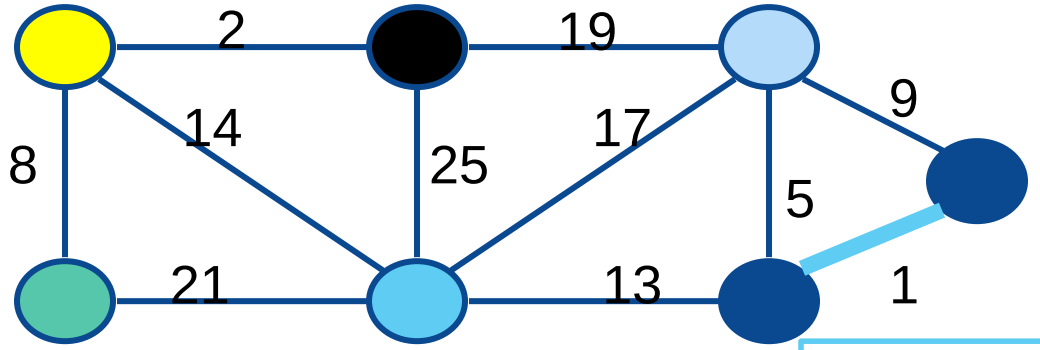
Od nejmenší hodnoty k největší

Jsou *u*,*v* z různých množin?

Kruskalův algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        VytvorMn(v);
    seřaď E vzestupně dle váhy hrany w
    for each (u,v) ∈ E (dle seřazení)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Běh algoritmu:

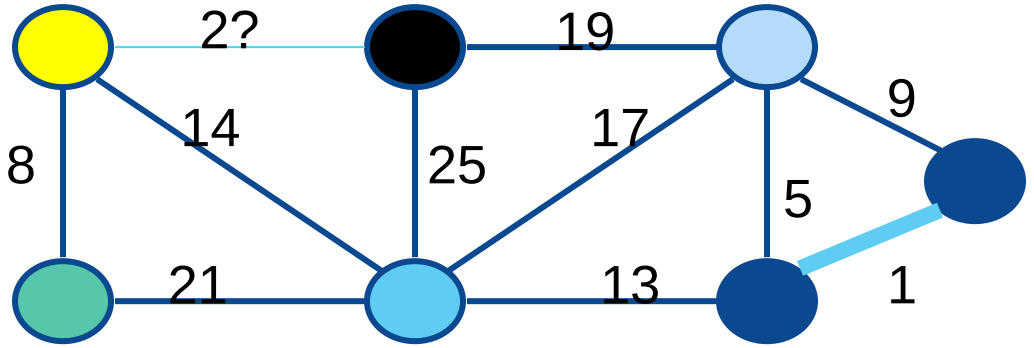


Nebyly ze stejné mn., Označ jako cestu

Kruskalův algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        VytvorMn(v);
    seřaď E vzestupně dle váhy hrany w
    for each (u,v) ∈ E (dle seřazení)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

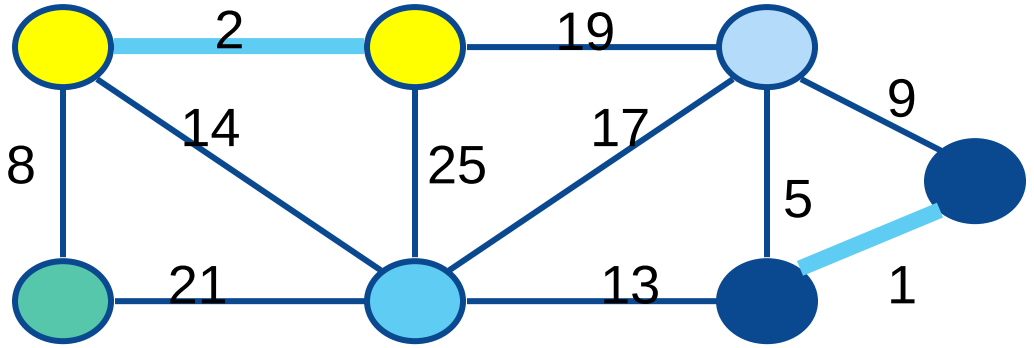
Běh algoritmu:



Kruskalův algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        VytvorMn(v);
    seřaď E vzestupně dle váhy hrany w
    for each (u,v) ∈ E (dle seřazení)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

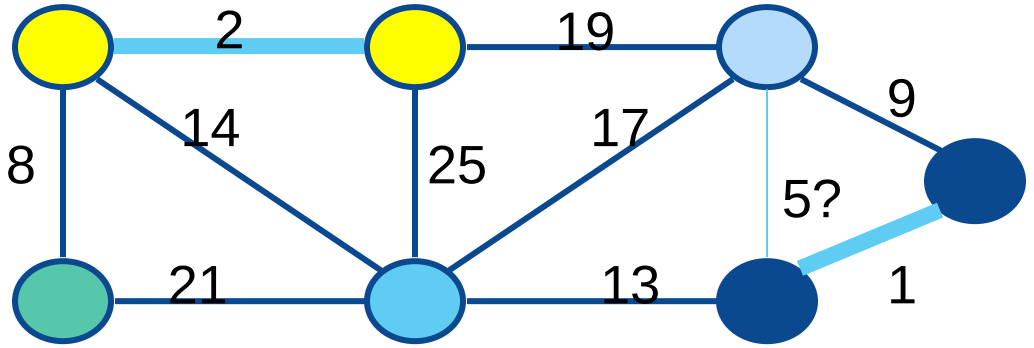
Běh algoritmu:



Kruskalův algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        VytvorMn(v);
    seřaď E vzestupně dle váhy hrany w
    for each (u,v) ∈ E (dle seřazení)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

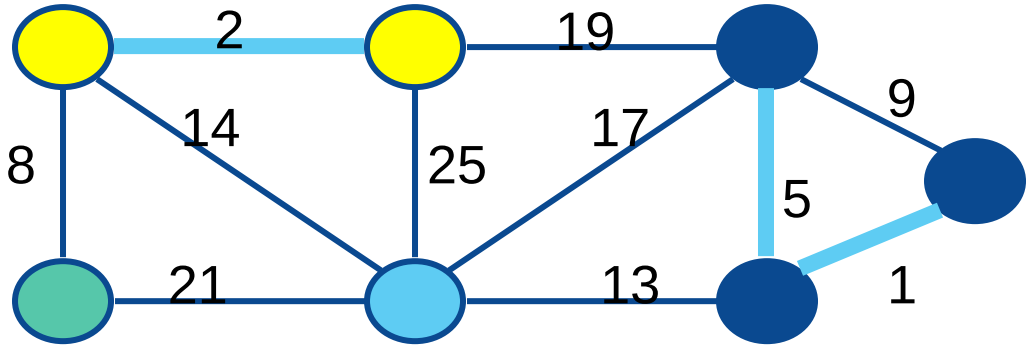
Běh algoritmu:



Kruskalův algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        VytvorMn(v);
    seřaď E vzestupně dle váhy hrany w
    for each (u,v) ∈ E (dle seřazení)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

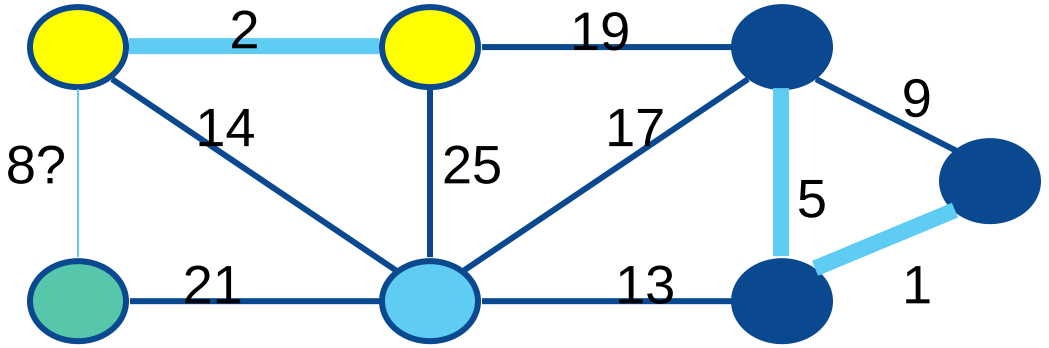
Běh algoritmu:



Kruskalův algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        VytvorMn(v);
    seřaď E vzestupně dle váhy hrany w
    for each (u,v) ∈ E (dle seřazení)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

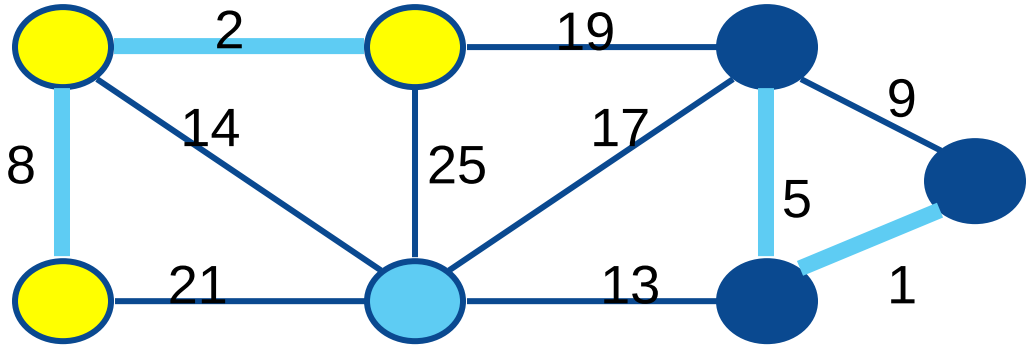
Běh algoritmu:



Kruskalův algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        VytvorMn(v);
    seřaď E vzestupně dle váhy hrany w
    for each (u,v) ∈ E (dle seřazení)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

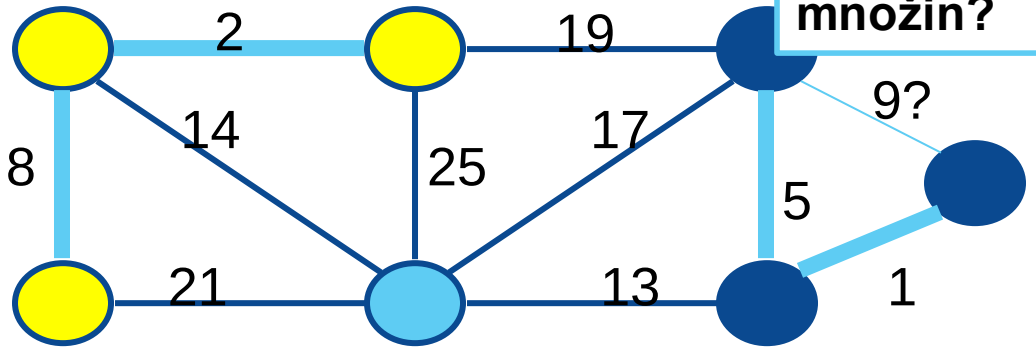
Běh algoritmu:



Kruskalův algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        VytvorMn(v);
    seřaď E vzestupně dle váhy hrany w
    for each (u,v) ∈ E (dle seřazení)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Běh algoritmu:

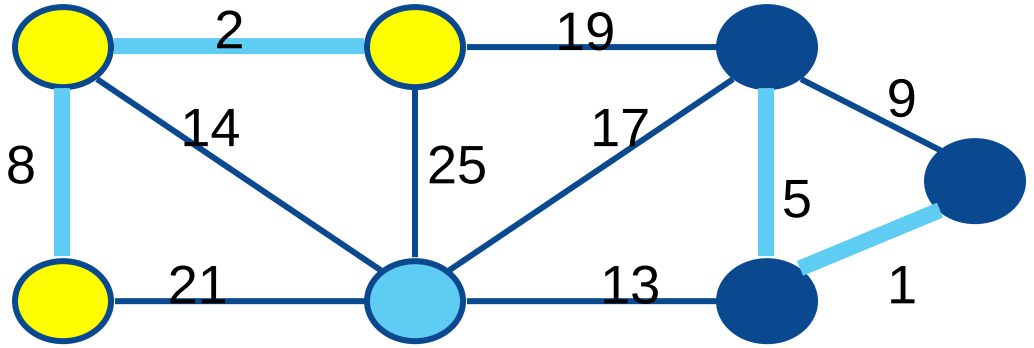


Jsou *u*,*v* z různých množin?

Kruskalův algoritmus

NE => není cesta

Běh algoritmu:

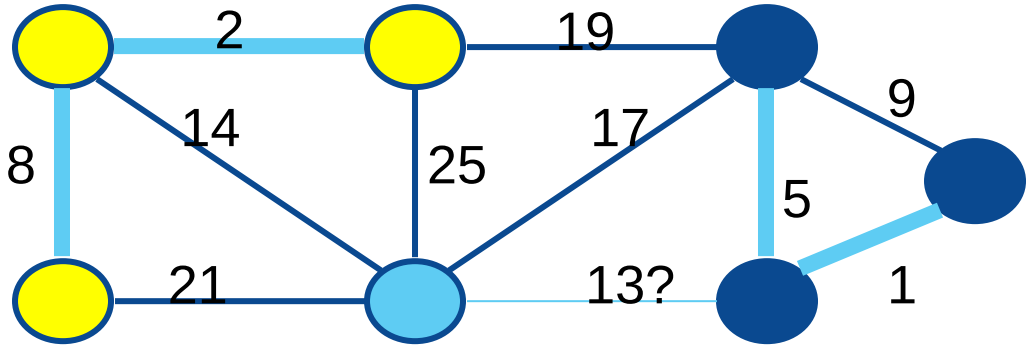



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        VytvorMn(v);
    seřaď E vzestupně dle váhy hrany w
    for each (u,v) ∈ E (dle seřazení)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskalův algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        VytvorMn(v);
    seřaď E vzestupně dle váhy hrany w
    for each (u,v) ∈ E (dle seřazení)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

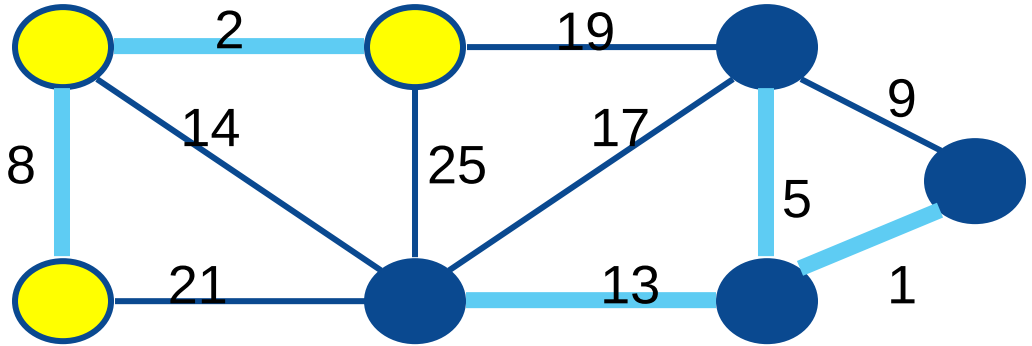
Běh algoritmu:



Kruskalův algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        VytvorMn(v);
    seřaď E vzestupně dle váhy hrany w
    for each (u,v) ∈ E (dle seřazení)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

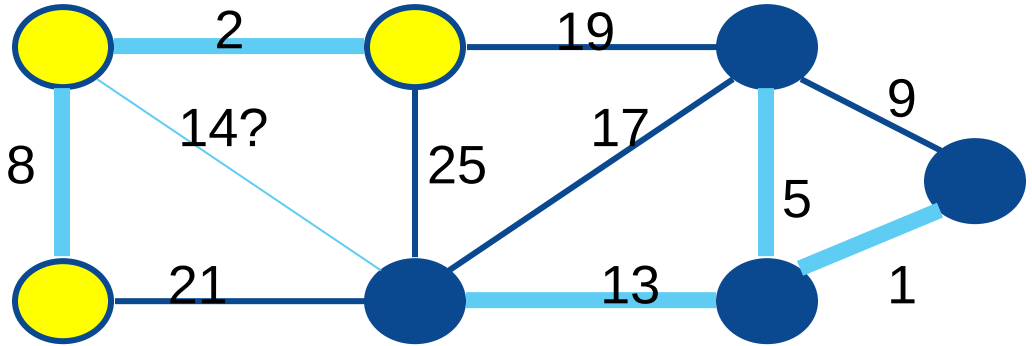
Běh algoritmu:



Kruskalův algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        VytvorMn(v);
    seřaď E vzestupně dle váhy hrany w
    for each (u,v) ∈ E (dle seřazení)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

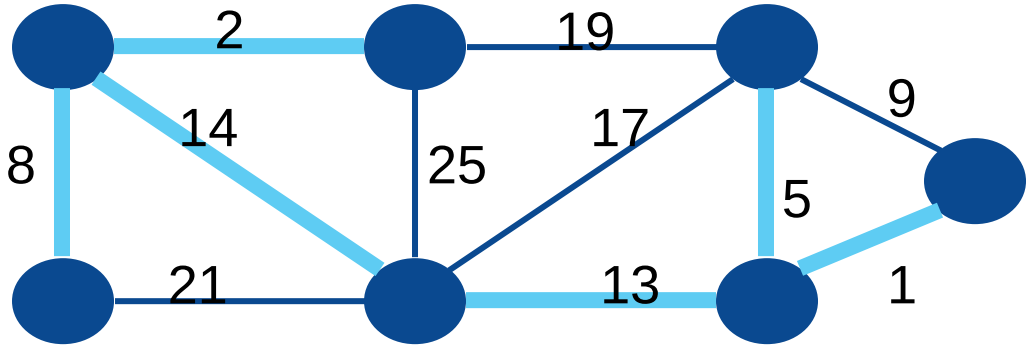
Běh algoritmu:



Kruskalův algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        VytvorMn(v);
    seřaď E vzestupně dle váhy hrany w
    for each (u,v) ∈ E (dle seřazení)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

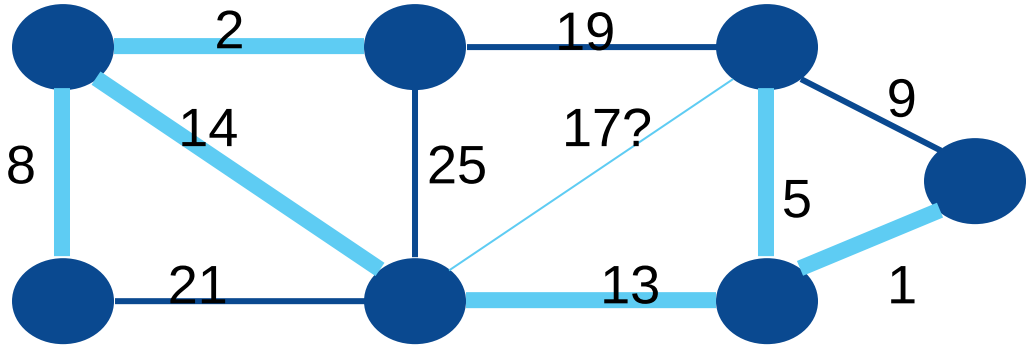
Běh algoritmu:



Kruskalův algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        VytvorMn(v);
    seřaď E vzestupně dle váhy hrany w
    for each (u,v) ∈ E (dle seřazení)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

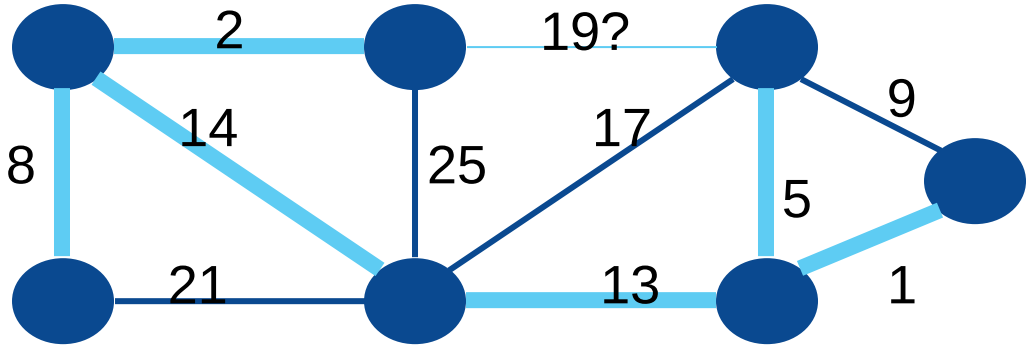
Běh algoritmu:



Kruskalův algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        VytvorMn(v);
    seřaď E vzestupně dle váhy hrany w
    for each (u,v) ∈ E (dle seřazení)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

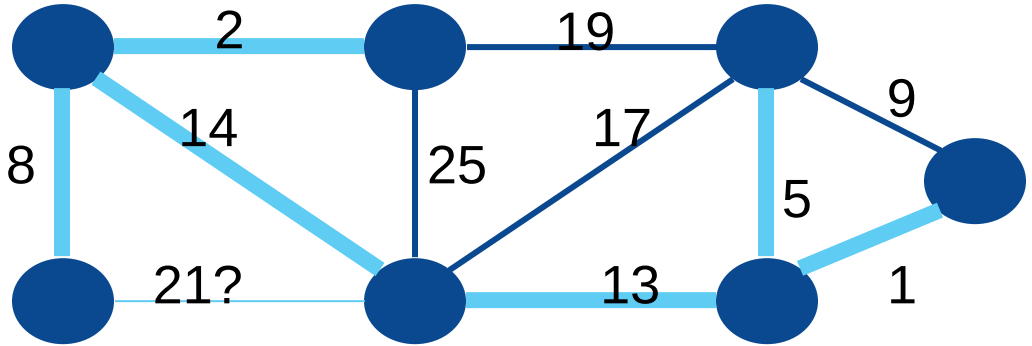
Běh algoritmu:



Kruskalův algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        VytvorMn(v);
    seřaď E vzestupně dle váhy hrany w
    for each (u,v) ∈ E (dle seřazení)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

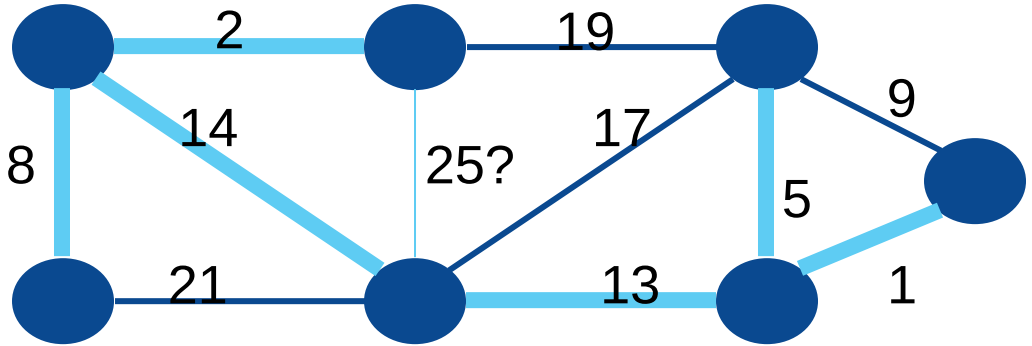
Běh algoritmu:



Kruskalův algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        VytvorMn(v);
    seřaď E vzestupně dle váhy hrany w
    for each (u,v) ∈ E (dle seřazení)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

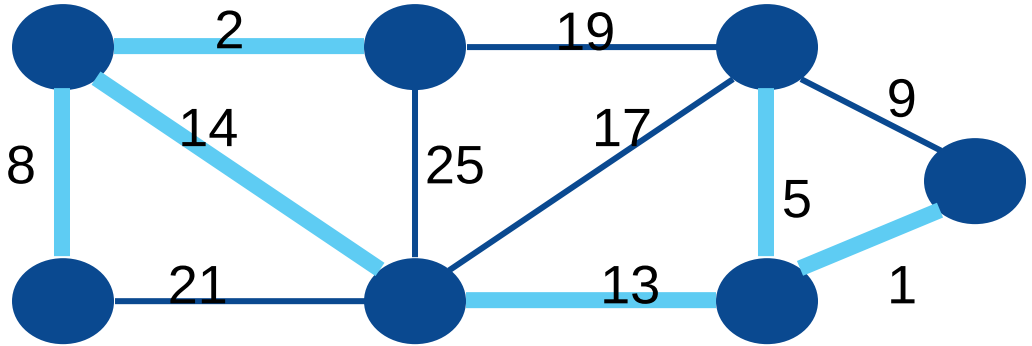
Běh algoritmu:



Kruskalův algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        VytvorMn(v);
    seřaď E vzestupně dle váhy hrany w
    for each (u,v) ∈ E (dle seřazení)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

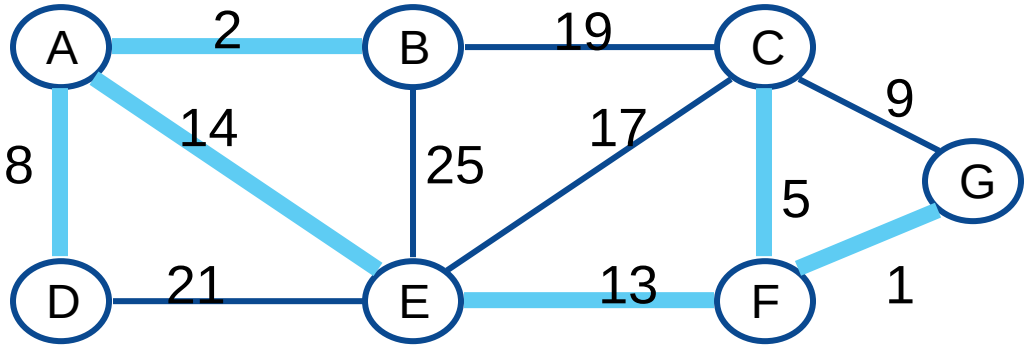
Běh algoritmu:



Kruskalův algoritmus

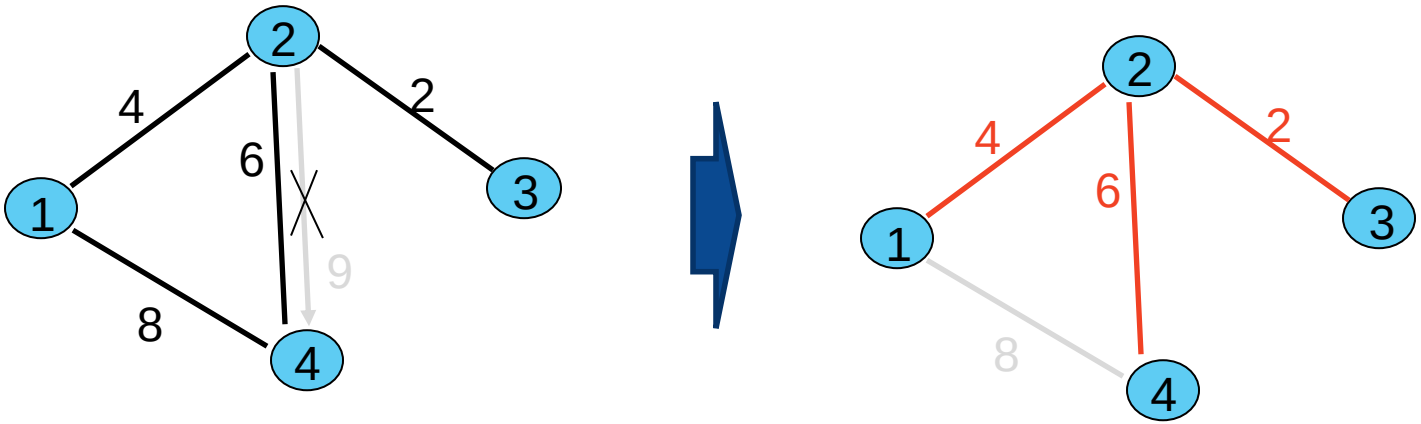
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        VytvorMn(v);
    seřaď E vzestupně dle váhy hrany w
    for each (u,v) ∈ E (dle seřazení)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Běh algoritmu:



Příklad

- Demo v Eclipse (neorientovaný graf)



Distribuované algoritmy

- Předchozí případ – znalost kompletní topologie a jeden procesor určí kostru grafu
- V telekomunikačních systémech – mnoho výpočetních jednotek (směrovače = vrcholy grafu) => **DISTRIBUOVANÝ ALGORITMUS**
- Jeden z návrhů: Gallager, Humblet, and Spira "**Distributed Algorithm for Minimum-Weight Spanning Trees**," ACM Transactions on Programming Languages and Systems, January 1983, pp. 66-67).
  - Začíná jedním fragmentem skládajícího se z jednoho vrcholu
  - Každý fragment vybere výstupní hranu s nejmenším ohodnocením
  - Spojí se se sousedícím fragmentem (vyjedná **zprávou**) pokud je také hranou s nejnižším ohodnocením
  - Generuje MST s časovou složitostí O(|V |x|V |)
  - Režie navíc O(|V |xlog|V |) + |E |) zpráv pro komunikaci

Distribuovaný MST

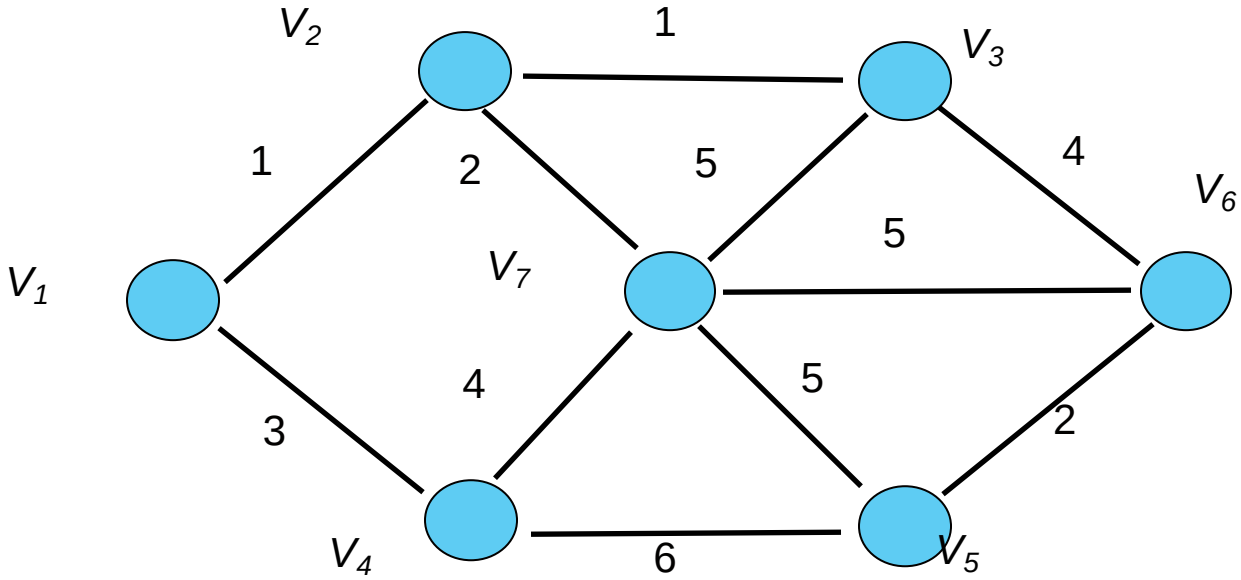
- Stavy:
  - Každý uzel je v jednom ze stavů:

Spící – počáteční stav

Hledající – po dobu hledání minimální výstupní hrany

Nalezeno – byla již nalezena výstupní linka

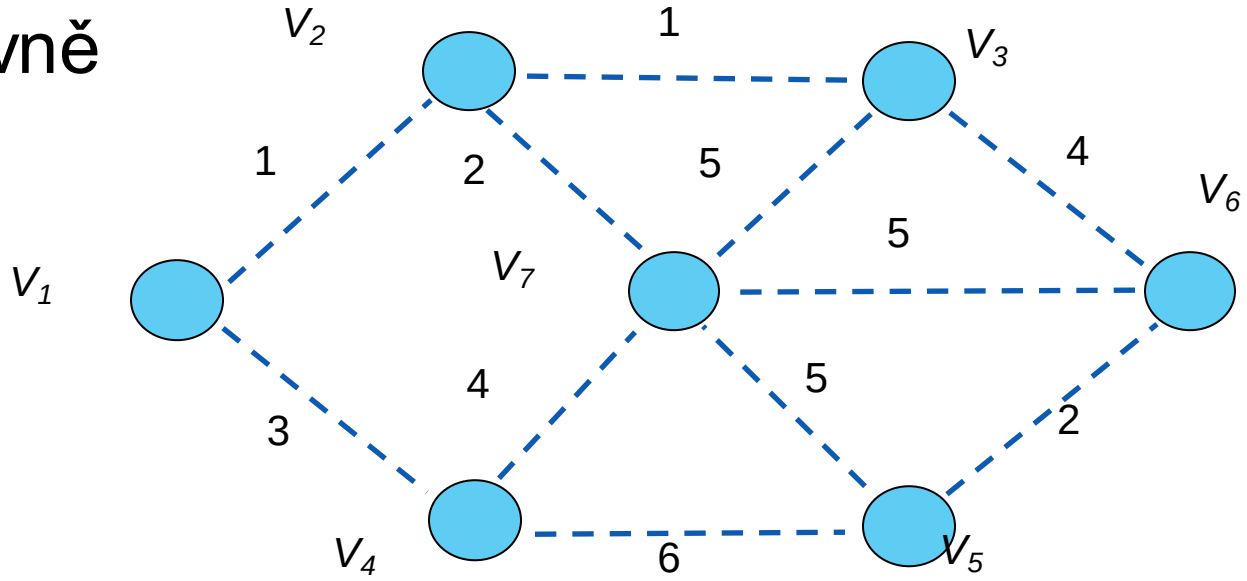
Distribuovaný algoritmus – Příklad



Distribuovaný algoritmus – Příklad

Fragmenty

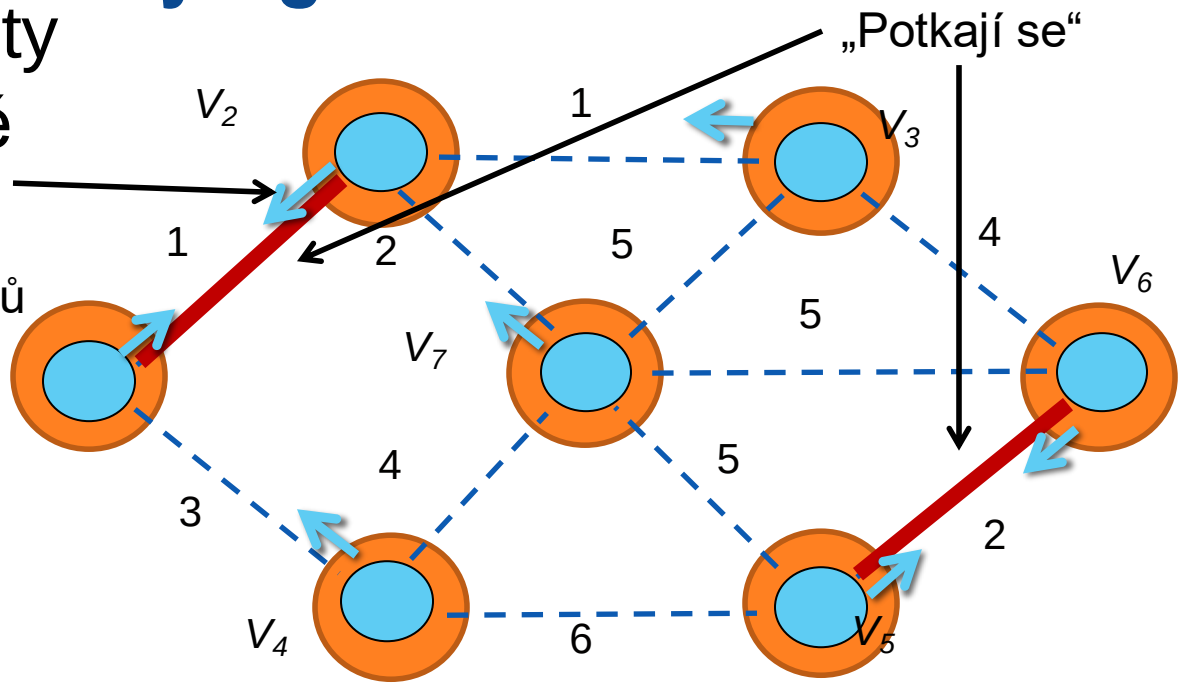
0. úrovně



Distribuovaný algoritmus – Příklad

Fragmenty 1. úrovně

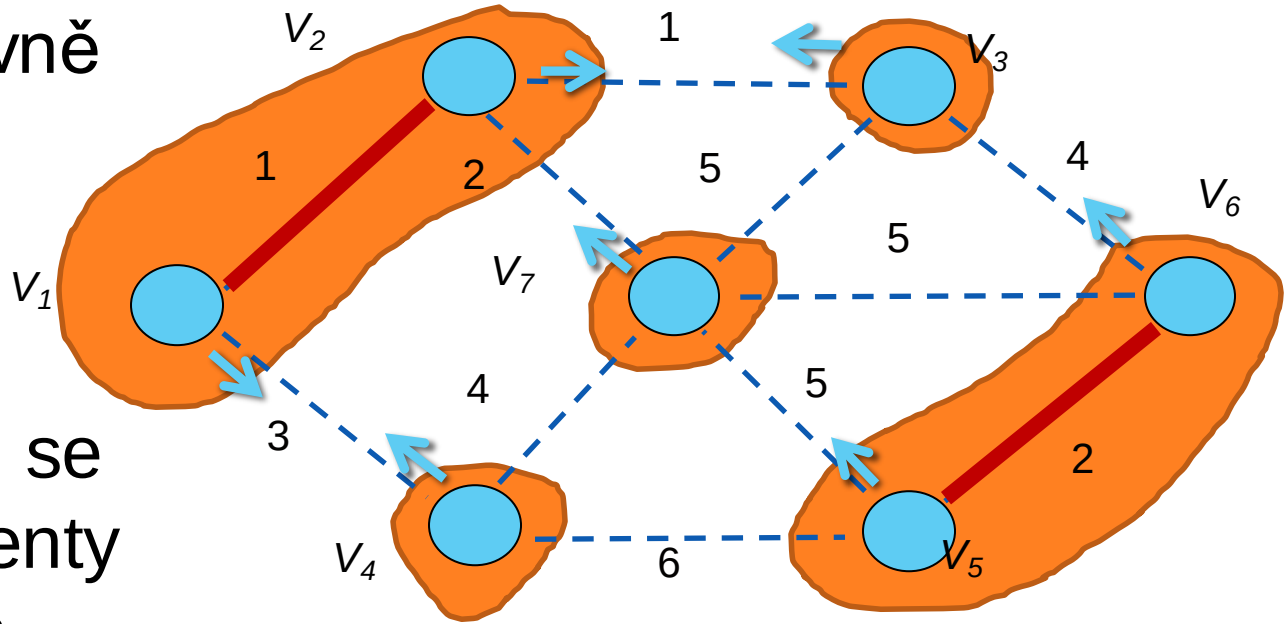
Vyber nejnižší (Bridge ID) z možných vrcholů



Zvol výstupní hranu fragmentu s nejnižším ohodnocením

Distribuovaný algoritmus – Příklad

Fragmenty
2. úrovně



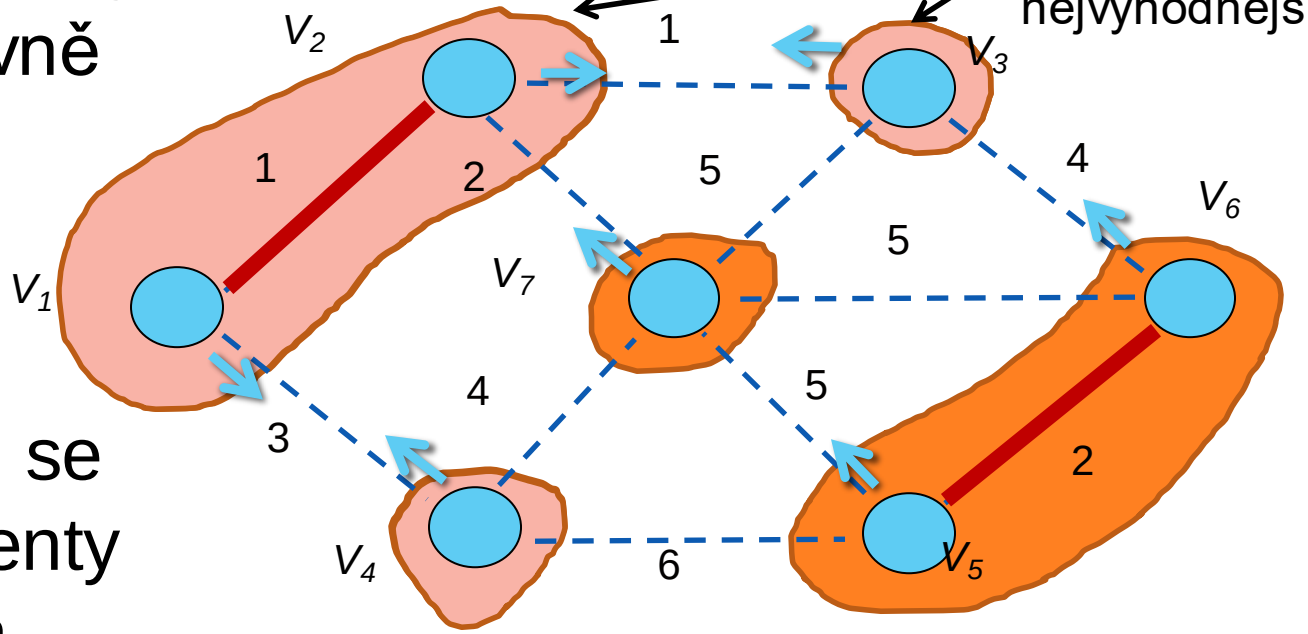
Ustaví se fragmenty {1,2} a {5,6}

Zvol výstupní hranu fragmentu s nejnižším ohodnocením

Distribuovaný algoritmus – Příklad

Fragmenty 2. úrovně

Tyto 2 fragmenty mají spol. cestu (jedinou) a ta je současně jejich nejvýhodnější

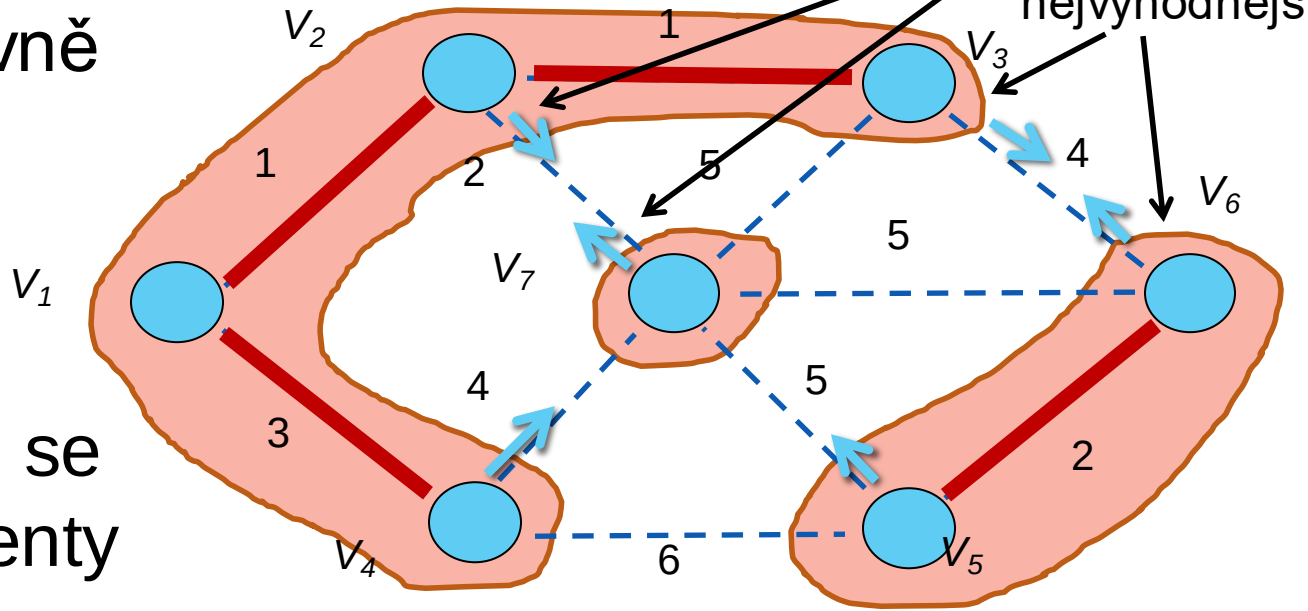


Ustaví se fragmenty {1,2} a {5,6}

Zvol výstupní hranu fragmentu s nejnižším ohodnocením

Distribuovaný algoritmus – Příklad

Fragmenty
3. úrovně



Tyto 2 fragmenty mají spol. cestu (jedinou) a ta je současně jejich nejvýhodnější

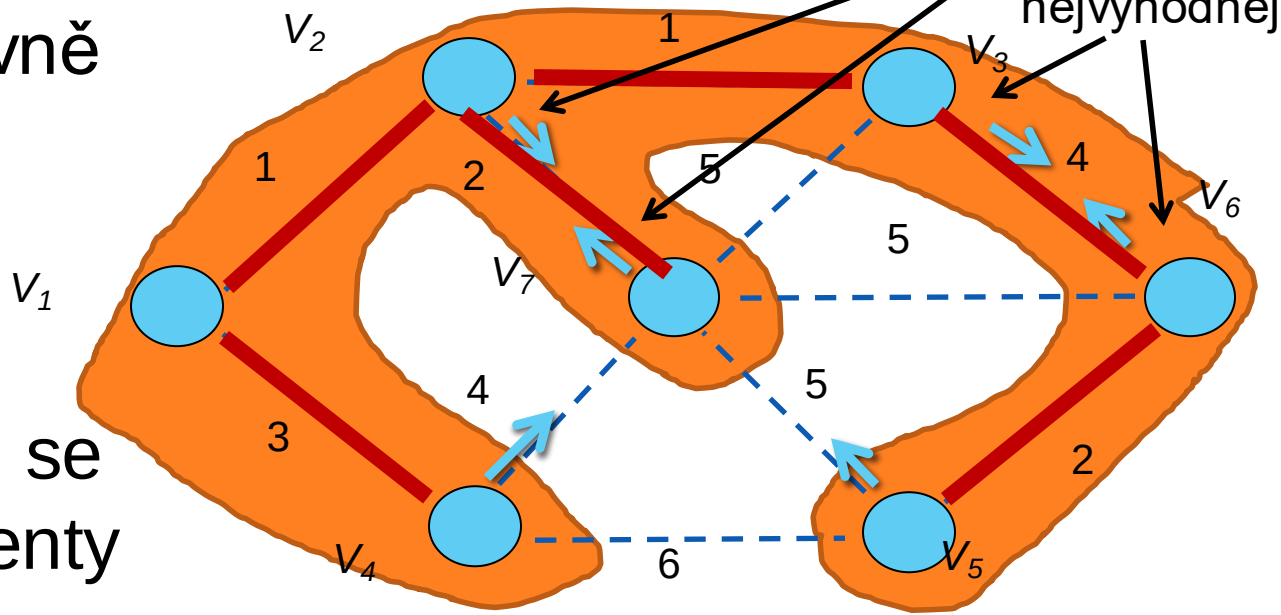
Ustaví se fragmenty {1,2} a {5,6}

Zvol výstupní hranu fragmentu s nejnižším ohodnocením

Distribuovaný algoritmus – Příklad

Fragmenty 4. úrovně

Ustaví se fragmenty {1,2} a {5,6}

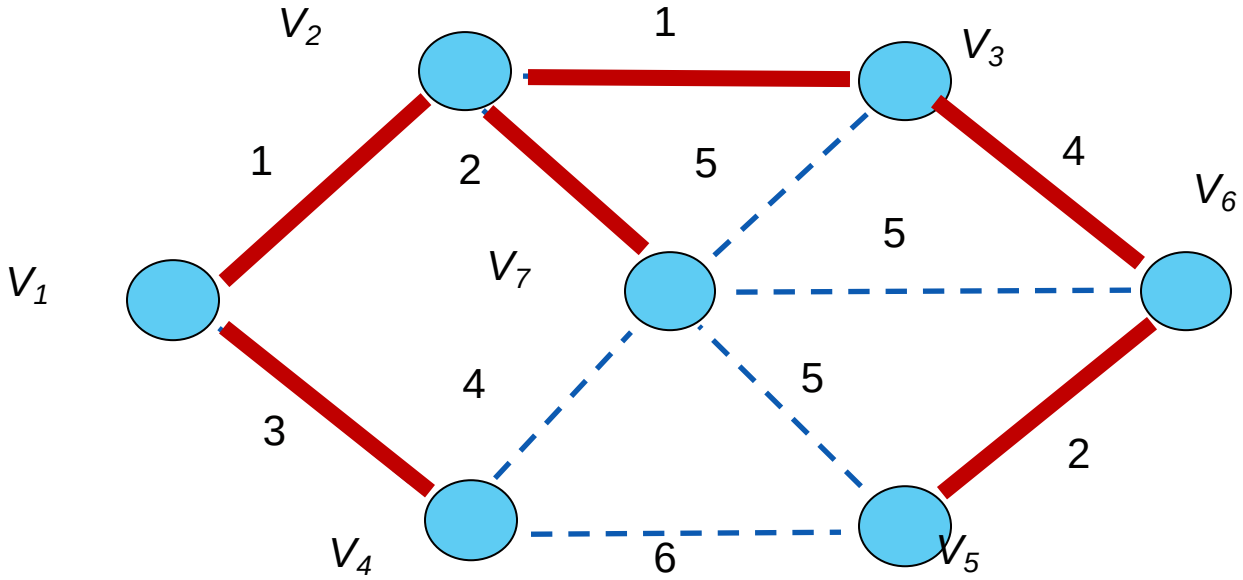


Tyto 2 fragmenty mají spol. cestu (jedinou) a ta je současně jejich nejvýhodnější

Zvol výstupní hranu fragmentu s nejnižším ohodnocením

Distribuovaný algoritmus – Příklad

Výsledek ve 3 krocích



Fragmenty {1,2,3,4,7} a {5,6} se spojí do nového fragmentu - {5,6} se připojí skrze nejnižší cestu.

Všichni ukončí činnost

Aplikace

- Unicast routing (jedna ku jedné) → SPST
- Multicast routing (jeden ku mnoha)
- Maximalizace pravděpodobnosti úspěchu v komunikaci „jeden ku mnoha“→ spanning tree s maximální vahou
- Load balancing → Stupněm omezený spanning tree

Děkuji za pozornost

Otázky – z pracovního pohovoru

- Mějme matici *mxn*. Každá buňka obsahuje číslo, která představuje množství zlata, 0 znamená, že je prázdná.
- Navrhněte algoritmus pro nalezení maxima zlata za podmínek:
  - Jakmile jste na místě, seberete veškeré zlato
  - Pohyb je možný jen nahoru, dolů, doleva či doprava
  - Políčko „0“ již nelze navštívit
  - Začít a skončit můžeme na jakémkoli políčku

0	6	0
5	8	7
0	9	0