

Nejkratší cesty grafem

Nejkratší cesty mezi dvojicemi uzlů. Dijkstra. Minimální kostra.
Borůvkův-Kruskalův algoritmus.

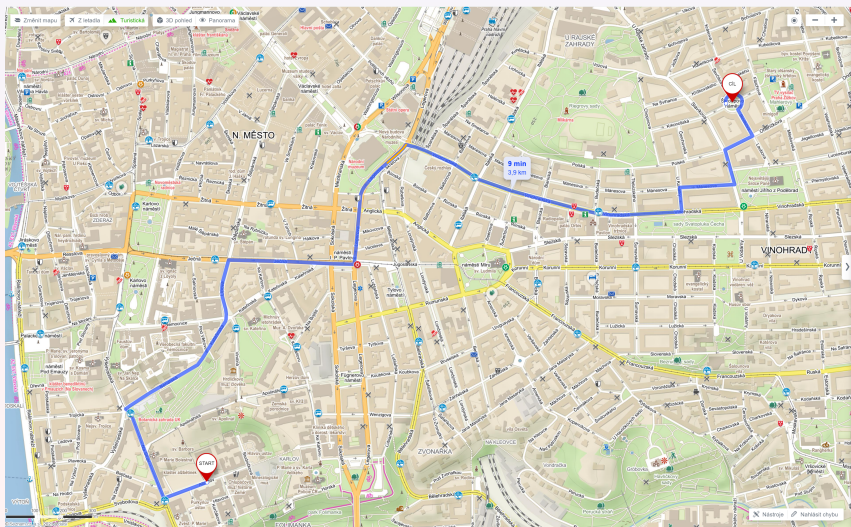
Tomáš Bayer | bayertom@natur.cuni.cz

Katedra aplikované geoinformatiky a kartografie, Přírodovědecká fakulta UK.

Obsah přednášky

- 1 Úvod
- 2 Nejkratší cesta mezi 2 uzly
 - Relaxace hrany
 - Dijkstra algoritmus
- 3 Minimální kostra grafu
 - Kosta grafu
 - Borůvkův/Kruskalův algoritmus
 - UNION-FIND
 - Weighted UNION-FIND

1. Ukázka nejkratší cesty mezi 2 uzly



2. Nejkratší cesta mezi 2 uzly

Nejčastěji řešená “dopravní” úloha.

Hledání nejkratší cesty z uzlu s do k v G .

Předpoklady pro G :

Orientovaný, neorientovaný, souvislý, konečný.

Popis G - spojový seznam (úspornější).

Většina metod vychází z BFS: prohledávání do šířky.

Provádí opakovanou relaxaci, netřeba značkovat.

Uzly uloženy v prioritní frontě (místo běžné fronty).

Ohodnocení hrany $h = (u, v)$:

Hrany grafů mají kladné ohodnocení $w(u, v) \in \mathbb{R}^+$.

Interpretace w : vzdálenost, čas jízdy, náklady, spotřeba, ...

Lze hledat nejkratší, nejlevnější či jinou cestu.

Přehled algoritmů:

- Dijkstra ($w \in \mathbb{R}^+$), běžně používán.
- Bellman+Ford ($w \in \mathbb{R}$), specializované případy.

Použití: navigační SW, logistika, doprava.

3. Délka cesty, vzdálenost uzlů

Cesta $C = \langle h_1, \dots, h_k \rangle$ v graf $G = \langle H, U, \rho \rangle$ tvořena k hranami h s délkou d_w

$$d_w(C) = \sum_{i=1}^k w(h_i).$$

Vzdálenost $d_w(u, v)$ uzlů u, v grafu G je nejmenší délka cesty z u do v

$$d_w(u, v) = \min_{\forall C} d_w(C)$$

Nejkratší cesta z vrcholu u do v neexistuje: $d_w(u, v) = \infty$.

Pro každé $u, v, x \in G$ platí axiomy:

- (1) $d_w(u, v) \geq 0$, ("nezápornost" vzdálenosti),
- (2) $d_w(u, v) = 0 \Leftrightarrow u = v$, (identita),
- (3) $d_w(u, v) = d_w(v, u)$, (symetrie, pro neorientované G),
- (4) $d_w(u, v) \leq d_w(u, x) + d_w(x, v)$, (trojúhelníková nerovnost)

Věta o nejkratší cestě:

Pokud $d_w(u, v)$ nejkratší cestou z u do v přes x , pak $d_w(u, x)$ nejkratší cestou z u do x a $d_w(x, v)$ nejkratší cestou z x do v .

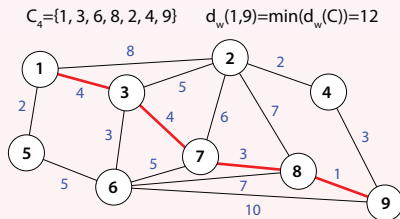
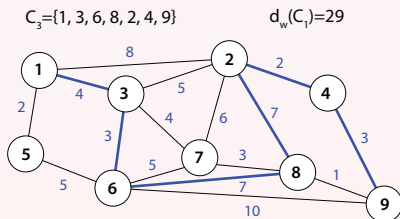
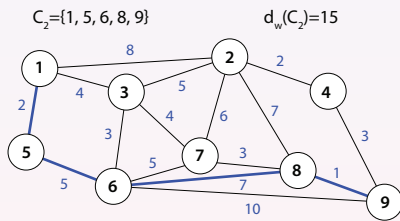
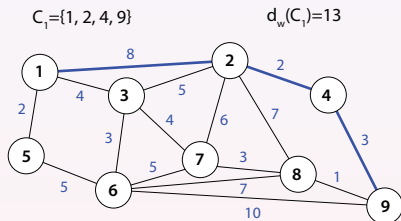
$$d_w(u, v) = d_w(u, x) + d_w(x, v) = d_w(u, x) + w(x, v).$$

Libovolná část nejkratší cesty je též nejkratší, důsledek TN.

4. W –délka vs. w – vzdálenost

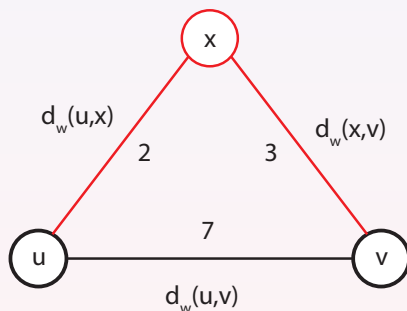
Mezi uzly 1, 9 mnoho cest C s různými délkami (modré).

Jedna z nich nejkratší (červeně), definuje vzdálenost, $d_w(1, 9) = 12$.

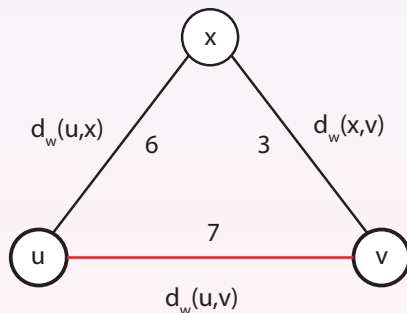


5. Ukázka trojúhelníkové nerovnosti

$$d_w(u,v) = d_w(u,x) + d_w(x,v) = 5$$



$$d_w(u,v) < d_w(u,x) + d_w(x,v) = 7$$



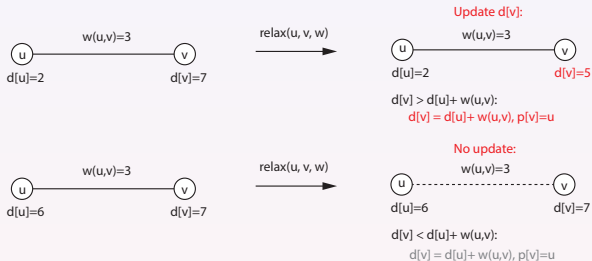
Vlevo, platí věta o nejkratší cestě.

Spojení obou případů

$$d_w(u, v) \leq d_w(u, x) + d_w(x, v).$$

6. Relaxace hrany (u, v)

Do operace vstupují uzly (u, v) a incidující hrana.



Hodnoty $d[u]$, $d[v]$ horními odhady $d_w(s, u)$, $d_w(s, v)$.

Pokud u neleží na nejkratší cestě s, v

$$d_w(s, v) \leq d_w(s, u) + d_w(u, v) \leq d_w(s, u) + w(u, v) \leq d[v] \leq d[u] + w(u, v),$$

Vlastní relaxace probíhá v *opačném směru*: opakovanou aktualizací odhadu $d[v]$ hledáme $d_w(s, v)$.

Odhad $d[v]$ se průběžně snižuje.

Aneb první nalezení v neznamená, že jsme k němu došli nejkratší cestou.

Pokud

$$d[v] > d[u] + w(u, v),$$

nejkratší cesta vede přes u a $p[v] = u$.

```
def relax(u, v, w):  
    if d[v] > d[u] + w(u, v):  
        d[v] = d[u] + w(u, v)  
        p[v] = u  
#Is path (s, u, v) shorter than (s, v)?  
#Update d[v]  
#Update predecessor
```

7. Dijkstra algoritmus

Autor Edsger W. Dijkstra (1956).

Nejznámější algoritmus pro hledání nejkratších cest mezi 2 uzly.

Ve skutečnost nejkratší cesta z s do všech ostatních uzlů G .

Zobecňuje strategii BFS + relaxace + prioritní fronta.

Uzly netřeba značkovat.

Předpoklady:

- nezáporné ohodnocení w ,
- G je souvislý.

Snaha o co nejmenší prodloužení cesty.

Realizuje se opakovanou relaxací hrany (u, v) .

Využívá *Greedy strategii*:

Heuristická optimalizace, zde úspěšná (obecně nemusí být).

Hledá globální minimum tak, že v každém kroku hledáme lokální.

Jednoduchá implementace.

8. Princip Dijkstra algoritmu

Hledána nejkratší cesta mezi uzly s a k .

Využívá postupného zpřesňování odhadu nejkratší délky od s do k .
Stávající nejkratší cestu se snažíme co nejméně prodloužit (+ 1 uzel).

Hodnota $d[u]$: aktuální odhad nejkratší vzdálenosti $d_w(s, u)$ k uzlu u .

Hodnota $d[v]$: aktuální odhad nejkratší vzdálenosti $d_w(s, v)$ k uzlu v .

Použití relaxace:

Pokud

$$d[v] > d[u] + w[u][v],$$

pak

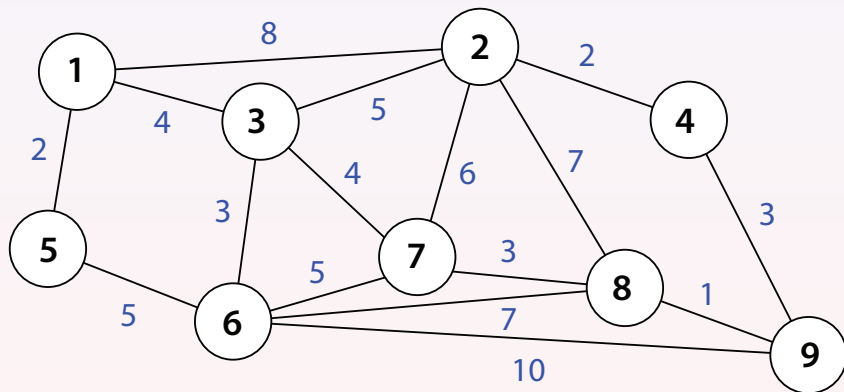
$$d[v] = d[u] + w[u][v],$$

je novým odhad $d[v]$ a

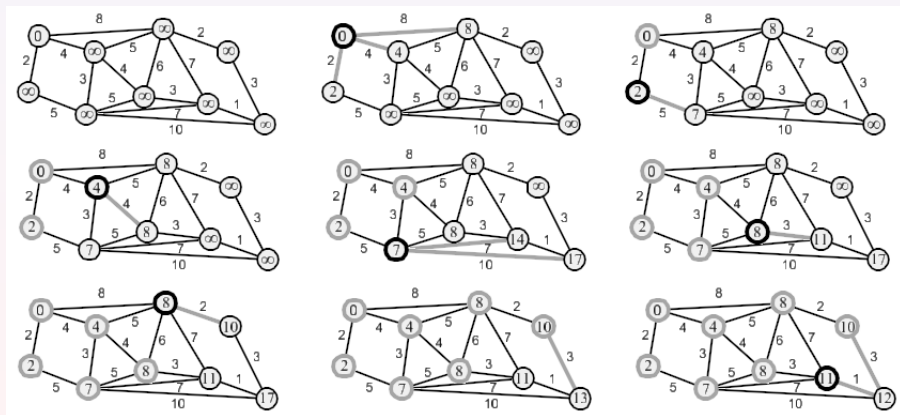
$$p[v] = u.$$

V každém kroku vybírán uzel u s nejmenší hodnotou $d[u]$.

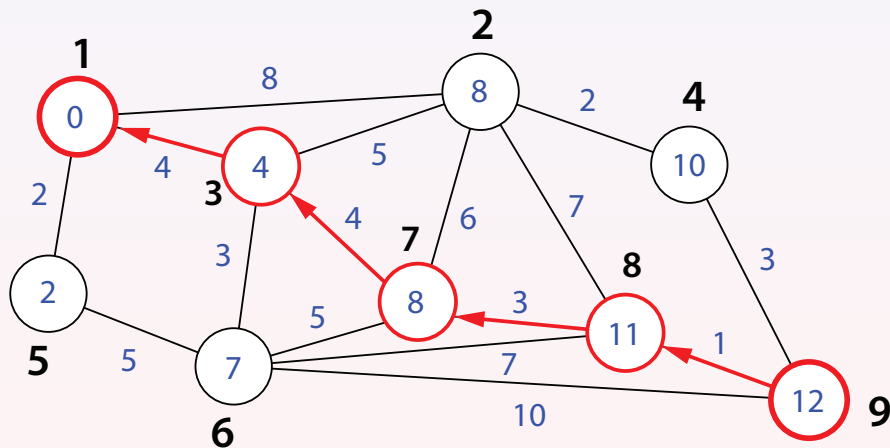
9. Ukázka vstupního grafu



10. Ukázka Dijkstra algoritmu



11. Výsledek Dijkstra algoritmu



12. Popis Dijkstra algoritmu

U každého uzlu uchováváno: $d[u]$, $p[u]$.

Uzly uloženy v prioritní frontě Q

$$Q = \{d[u], u\}.$$

Netřeba značkovat uzly.

Cesta rekonstruována ze seznamu předchůdců zpětně.

Startovní uzel s , jednotlivé fáze:

- 1 *Inicializační fáze:*
Inicializace vstupních hodnot: $d(u) = \infty$, $p(u) = -1$.
Nastavení $d[s] = 0$. Přidání $\langle d[s], s \rangle$ do Q .
- 2 *Iterativní fáze:*
Dokud Q není prázdná:
 - Z Q vybrán uzel u s nejmenší hodnotou $d(u)$.
 - Relaxaci z u na všechny sousedy v :
 - Pokud nové $d[v]$ menší než původní, aktualizujeme.
 - Aktualizujeme předchůdce: $p[v] = u$.
 - Přidáme $\langle d[v], v \rangle$ do Q .

Opakujeme (2), dokud Q není prázdná, tj. existuje nějaký otevřený uzel.

Snadná implementace, analogie BFS.

13. Datový model grafu s ohodnocením

Spojová reprezentace.

V Pythonu použít Dictionary

$$\langle K, V \rangle.$$

Klíč K : uzel.

Hodnota V : seznam incidujících vrcholů s ohodnocením hran.

```
G={V1 : {V1:W1, V2:W2, ..., Vk:Wk},
    V2 : {V1:W1, V2:W2, ..., Vk:Wk},
    ...
    Vk : {V1:W1, V2:W2, ..., Vk:Wk}}
```

Ukázka popisu G :

```
G3 = {
    1 : {2:8, 3:4, 5:2},
    2 : {1:8, 3:5, 4:2, 7:6, 8:7},
    3 : {1:4, 2:5, 6:3, 7:4},
    4 : {2:2, 9:3},
    5 : {1:2, 6:5},
    6 : {3:3, 5:5, 7:5, 8:7, 9:10},
    7 : {2:6, 3:4, 6:5, 8:3},
    8 : {2:7, 6:7, 7:3, 9:1},
    9 : {4:3, 6:10, 8:1}
}
```

14. Implementace Dijkstra algoritmu

Implementace s prioritní frontou.

```
def dijkstra(G, start, end):  
    d = [inf] * (len(G) + 1)           #Set infinite distance  
    p = [-1] * (len(G) + 1)           #No predecessors  
    Q = queue.PriorityQueue()          #Priority queue  
    Q.put((0, start))                  #Add start vertex  
    d[start] = 0                       #Start d[s] = 0  
    while not Q.empty():  
        du, u = Q.get()                 #Pop first element  
        for v, wuv in G[u].items():    #Relaxation, all (u,v)  
            if d[v] > d[u] + wuv:       #We found a better way  
                d[v] = d[u] + wuv      #Update distance  
                p[v] = u               #Update predecessor  
                Q.put((d[v], v))       #Add to Q
```

Ukázka:

```
dijkstra(G, 1, 9)  
path(p, 1, 9)  
>> 1 3 7 8 9
```

15. Úvod

Minimální kostra grafu: Minimum Spanning Tree.

Podgraf, faktor, strom, minimální váha hran.

Úloha může mít více řešení.

Často řešeno v dopravě, energetice, telekomunikacích.

Optimální zásobovací síť, udržení sjízdnosti silnic, železniční spojení.

Elektronika, návrh plošných spojů (L1 norma).

Z hlediska spolehlivosti *není optimální*.

Výpadek 1 hrany: rozpad G na 2 nesouvislé grafy.

Důsledek: přerušení dodávky energie, neprůjezdnost (nehoda), atd.

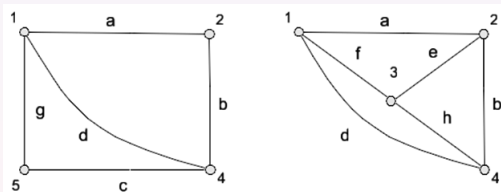
Kritické části infrastruktury nutno posílit: zdvojení.

Předpoklad G neorientovaný, souvislý, nezáporné hrany.

16. Podgraf a faktor

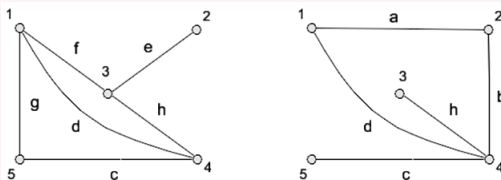
Podgraf $G' \subset G$:

$G' = (H', U', \rho')$ je podgraf $G = (H, U, \rho)$, platí-li $H' \subset H$ a $U' \subset U$ a pro každé $h \in H'$ je $\rho'(h) \subset \rho(h)$.



Faktor grafu:

Podgraf G' , jehož množina uzlů U' je totožná s množinou uzlů U grafu G .



17. Kostra grafu

Graf $G = (H, U, \rho)$, neorientovaný, $m = \|H\|$, $n = \|U\|$, souvislý.

Kostra grafu G :

$K = (H_k, U, \rho_k)$, podgraf G (faktor) který je stromem.

Obsahuje všechny uzly G , některé jeho hrany, netvoří cyklus.

Kostra tvořena n_k hranami

$$n_k = \|H_k\| = \|U\| - 1 = n - 1.$$

Pro G existuje “mnoho” koster: všechny podgrafy s n_k hranami neobsahující kružnici.

Exaktní stanovení počtu koster pro G obtížné.

Úplný graf G , počet koster N_k (Cayley, 1889)

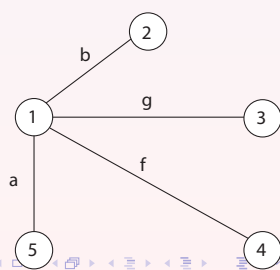
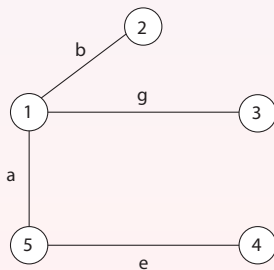
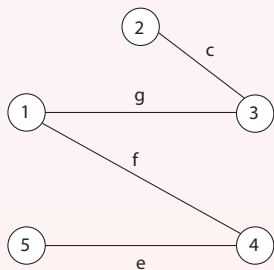
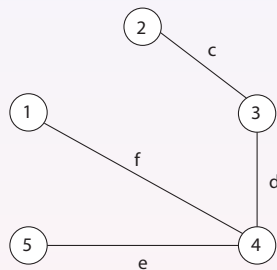
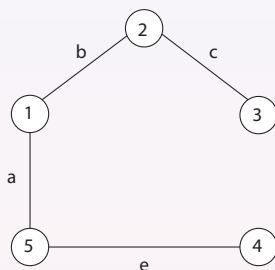
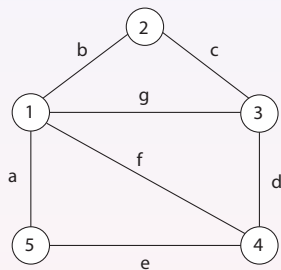
$$N_k = n^{n-2}.$$

Hledání všech koster neefektivní, exponenciální růst.

Lze provést pouze pro malé grafy.

V praxi nás zajímá kostra s minimálním ohodnocením, tzv. **minimální kostra**.

18. Graf a některé jeho kostry



19. Minimální kostra grafu

Bez ohodnocení grafu mají všechny K stejnou váhu.

Předpoklad: G souvislý, neorientovaný, bez záporných hran.

Hledáme $K = (H_k, U, \rho_k)$ s minimálním ohodnocením

$$C = \sum_{h \in H_k} w(h) = \min.$$

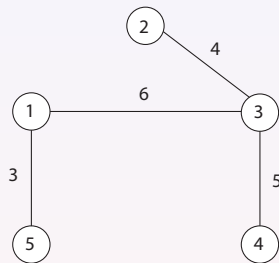
Úloha není jednoznačná, existence více minimálních koster.

Přehled algoritmů:

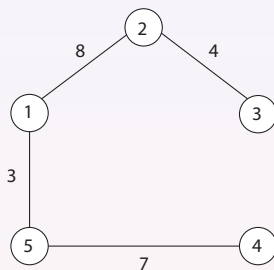
- *Borůvkův (modifikace Kruskal)*
Vhodný pro řídké grafy, $O(\|H\| \log \|U\|)$.
- *Jarníkův (modifikace Prim)*
Vhodný pro grafy s mnoha hranami, $O(\|U\| \log \|U\|)$.

Modifikované BFS vs. sjednocování podstromů v les.

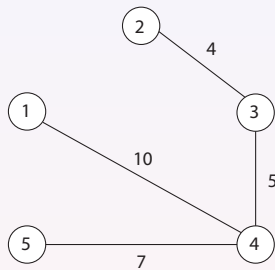
20. Ukázka minimální kostry



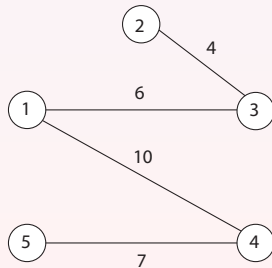
$C_{\min} = 18$



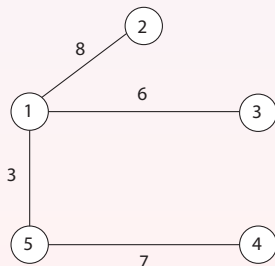
$C = 22$



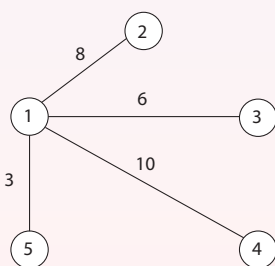
$C = 26$



$C = 27$



$C = 24$



$C = 27$

21. Borůvkův/Kruskalův algoritmus

Objeven prof. Otakarem Borůvkou (1899-1995) v roce 1926.

Připojení každé obce na jižní Moravě energetické sítě.

Minimalizace nákladů na připojení: délka vedení, počet stožárů.

V roce 1956 znovu objeven v USA J. P. Kruskalem.

Heuristika, v každém okamžiku hledáno lokální minimum.

Do kostry přidáváme hranu s minimální vahou spojující 2 podstromy.

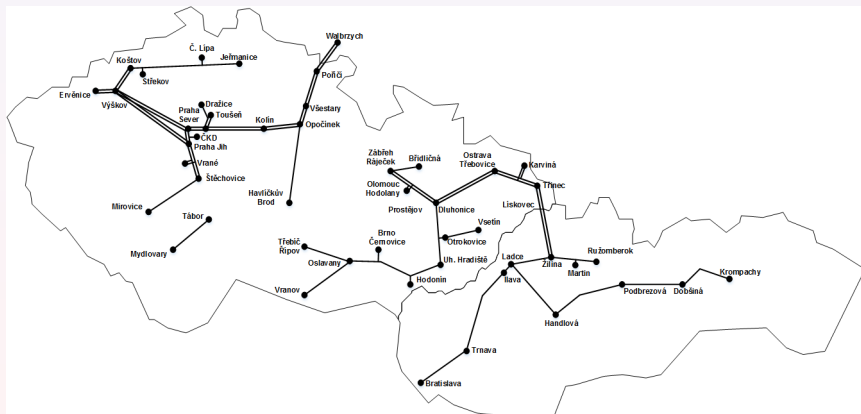
Pokračujeme, dokud les není stromem.

Opakované sjednocování: UNION, 2 podstromy.

Strom: souvislý, neobsahuje kružnici.

Les: nesouvislý, neobsahuje kružnici, tvořen podstromy.

22. Schéma elektrické sítě, ČSR



23. Princip Borůvkova/Kruskalova algoritmu

Předzpracování hran: seřídění hran dle w .

Implementace využívá prioritní frontu.

Opakované přidávání hrany h s aktuálně nejnižším w , aby nevznikla kružnice.

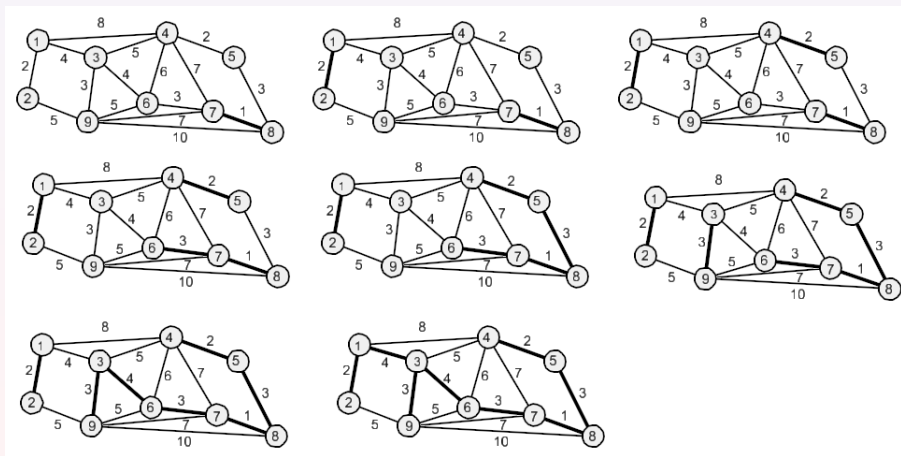
Mohou nastat 4 situace:

- 1 *Žádný uzel h neleží v jiném podstromu*
Hrana vytvoří nový podstrom.
- 2 *Jeden uzel h součástí některého z podstromů*
Hranu připojíme k podstromu.
- 3 *Oba uzly h v různých podstromech*
Oba podstromy spojíme do jednoho.
- 4 *Oba uzly h v jednom podstromu*
Přidání h vede ke kružnici, zahodíme.

Implementace množinových operací nad stromem:

- $\text{MAKE_SET}(x)$: vytvoření nové množiny s jedním prvkem x (identifikátor).
- $\text{FIND_SET}(x)$: vrací identifikátor množiny obsahující prvek x .
- $\text{UNION}(x, y)$: sjednocení podmnožin x, y do jedné podmnožiny.

24. Ukázka Borůvkova/Kruskalova algoritmu



25. Ukázka operace UNION(x, y)

Řádek 0: operace MAKE_SET.

#	h	$w(h)$	Disjunktní podmnožiny: $ID : \{items\}$								
0	-	-	1 : {1}	2 : {2}	3 : {3}	4 : {4}	5 : {5}	6 : {6}	7 : {7}	8 : {8}	9 : {9}
1 (7,8)	1		1 : {1}	2 : {2}	3 : {3}	4 : {4}	5 : {5}	6 : {6}	7 : {7,8}		9 : {9}
2 (1,2)	2		1 : {1,2}		3 : {3}	4 : {4}	5 : {5}	6 : {6}	7 : {7,8}		9 : {9}
3 (4,5)	2		1 : {1,2}		3 : {3}	4 : {4,5}		6 : {6}	7 : {7,8}		9 : {9}
4 (6,7)	3		1 : {1,2}		3 : {3}	4 : {4,5}		7 : {7,8,6}			9 : {9}
5 (3,9)	3		1 : {1,2}		3 : {3,9}	4 : {4,5}		7 : {7,8,6}			
6 (5,8)	3		1 : {1,2}		3 : {3,9}	7 : {7,8,6,4,5}					
7 (3,6)	4		1 : {1,2}		7 : {7,8,6,4,5,3,9}						
8 (1,3)	4		7 : {7,8,6,4,5,3,9,1,2}								

Minimální kostra G :

$$K = \{7, 8, 6, 4, 5, 3, 9, 1, 2\}, w(K) = 22.$$

26. UNION-FIND

Základem kombinovaná operace $\text{UNION-FIND} = \text{UNION} + \text{FIND_SET}$.

Umožňuje rychlé spojení podstromů.

Operace často používána v informatice (UNION).

Analogie s přebarvováním 2 různobarevných množin.

Po sjednocení mají obě množiny stejnou barvu.

Snaha o to, aby obě operace byly co nejrychlejší.

2 varianty UNION-FIND:

- *Array-based (pole)*
Rychlejší nalezení, pomalejší spojení.
 $\text{FIND_SET } O(1)$, $\text{UNION } O(\log_2(n))$.
- *Tree-based (strom)*
Pomalejší nalezení, rychlejší spojení.
 $\text{FIND_SET } O(\log_2(n))$, $\text{UNION } O(1)$.

V praxi preferována varianta 2.

Téměř všechny implementace Tree-based.

2 heuristiky vylepšující efektivitu:

- Weighted UNION: připojování menšího za větší (AB, TB).
- Path Compression: zkracování výšky stromu (TB).

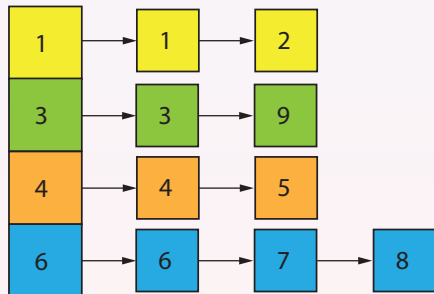
27. *UNION-FIND, AB

3 seznamy: podstromy, jejich délky a indexační struktura.

Pro podstrom uchovává seznam uzlů a identifikátor.

Identifikátorem zpravidla prvek s nejmenším ID.

UF: linked list, identifiers



UF: sizes

1	2
3	2
4	2
6	3

UF: 1D array, index

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

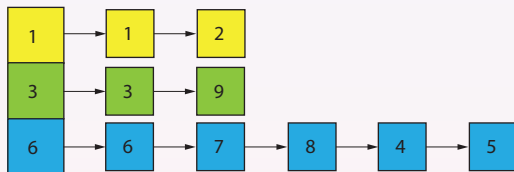
28. *Ukázka Weighted UNION(4,6), AB

Weighted Union:

Za delší pole připojíme kratší.

Změna identifikátorů pro kratší seznam, delší seznam neměněn.

UF: linked list, identifiers



UF: sizes

1	2
3	2
6	5

UF: 1D array, index

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

UNION(4, 6), $\|L_u\| = 2$, $\|L_v\| = 3$: Pak $L_v \leftarrow L_u$, úprava indexů delšího seznamu.

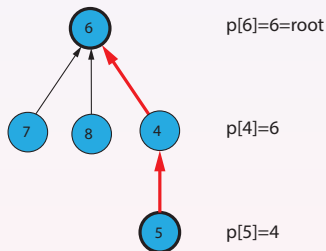
```
def union(u, v, index):  
    if len(u) < len(v):  
        for i in u:  
            index[i] = index[v]          #Process all elements in shorter array  
        size[v] = size[v] + size[u]      #Change all indices  
                                         #Change size
```

29. FIND, TB

Každá podmnožina **kořenovým stromem**:

Její prvky odkazují na rodiče (parent) $\Rightarrow$ kořen.

Aneb pro každý prvek stromu uložen kořen p , který je identifikátorem.



Operace FIND:

Kořeny uloženy v poli p , pro každý uzel u lze získat $p[u]$.

Postup vzhůru od uzlu u směrem ke kořenu dokud $p[u] \neq u$.

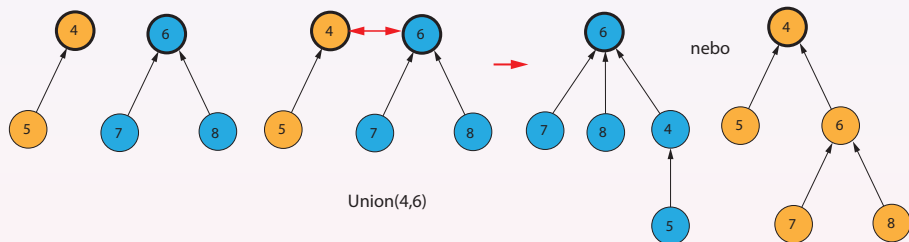
```
def find(u, p):  
    while(p[u] != u):  
        u = p[u]  
    return u
```

30. UNION, TB

Operace UNION:

Spojení kořenů R_U , R_V dvou stromů novou hranou.

Zatím neřešíme, zda připojujeme kratší za delší, či naopak.



Operace UNION FIND:

Nalezení kořenů R_U , R_V (FIND) + sjednocení(UNION).

```
def union(u, v, p):  
    root_u = find(u, p)    #Union + Find  
    root_v = find(v, p)    #Find first root  
    if root_u != root_v:   #Find second root  
        p[root_u] = root_v #u, v in different subtrees  
                           #Connect second tree to the first one
```

Připojování delšího stromu za kratší neefektivní.

31. *UNION-FIND, TB, zefektivnění

Hodnost stromu r (rank):

Odhad výšky stromu

$$r = \log_2 n_s,$$

kde n_s počet uzlu ve stromu.

Pro UNION-FIND netřeba znát přesně, zajímají nás rozdíly délek.

Weighted UNION:

Lepší připojovat kratší strom k delšímu.

Vyvažování stromu při sjednocení.

Kořen “menšího” stromu připojen na kořen “většího” dle r .

Pokud jsou stejné, vybereme libovolný z nich.

Path Compression:

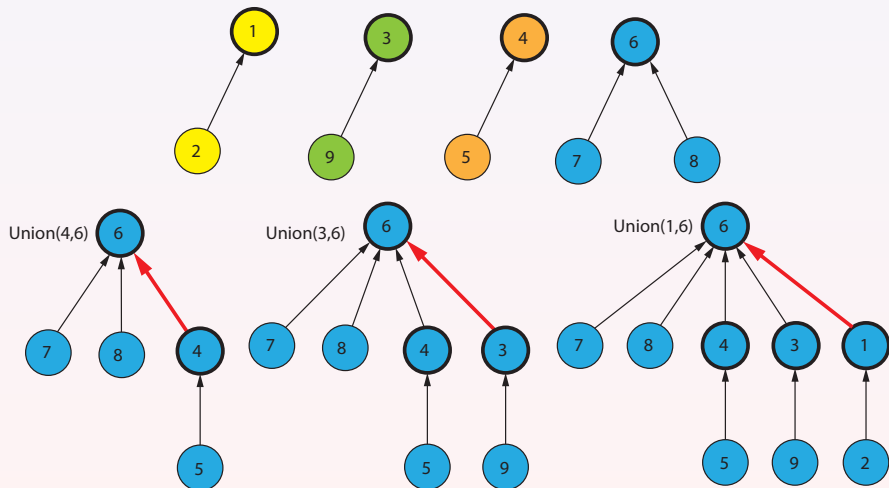
Zkracování cesty (stromu) $\Rightarrow$ snižování výšky.

Upravení odkazu na kořen všem uzlům ve stromě.

Heuristika, dvoupřechodová, rekurzivní/nerekurzivní implementace.

Významné zvýšení metody UNION-FIND: Weighted UNION-FIND.

32. *Weighted UNION-FIND, TB



33. *Operace Weighted UNION(u, v)

Weighted UNION:

Pro každý uzel u nutno uchovávat 2 informace:

- parent (kořen) $p[u]$, inicializace $p[u] = u$,
- hodnota (rank) podstromu $r[u]$, inicializace $r[u] = 0$.

Pro p, r použity seznamy.

Připojujeme-li menší podstrom k většímu, rank většího se nemění

$$r = \max(r[u], r[v]).$$

Rank updatován, pokud $r[u] = r[v]$,

$$r = r[u] + 1 = r[v] + 1.$$

Weighted UNION-FIND:

FIND + Weighted UNION.

1

FIND:

Nalezení kořenů R_u, R_v stromů obsahujících u, v .

2

Weighted UNION

Pokud u, v jiném podstromu, porovnáme výšky stromů:

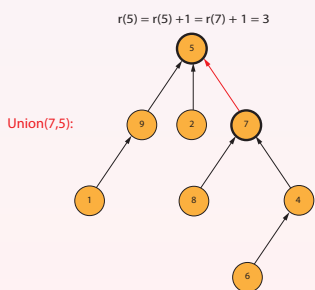
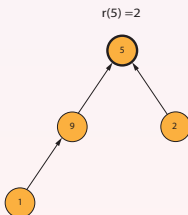
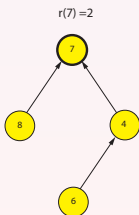
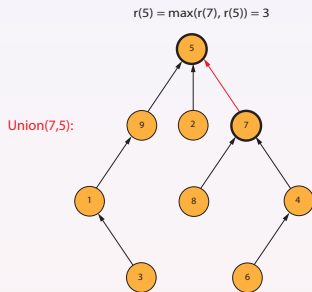
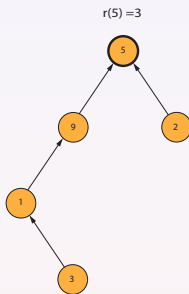
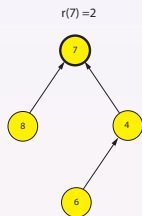
a) Jestliže $r[u] > r[v]$, připojíme v k u : $p[R_v] = R_u$.

b) Jestliže $r[v] > r[u]$, připojíme u k v : $p[R_u] = R_v$.

c) Při shodě $r[u] = r[v]$: $p[R_v] = R_u$ (např.).

Update výšky $r[u] = r[u] + 1$.

34. *Weighted UNION(u, v), update ranku



35. *Weighted UNION-FIND(u, v)

Nalezení kořenů R_u, R_v : FIND

Sjednocení: Weighted UNION.

```
def union(u, v, p, r):
    root_u = find(u, p)           #Find root for u + compress
    root_v = find(v, p)           #Find root for v + compress
    if root_u != root_v:          #u, v in different subtrees
        if r[root_u] > r[root_v]: #u subtree is longer
            p[root_v] = root_u    #Connect v to u
        elif r[root_v] > r[root_u]: #v subtree is longer
            p[root_u] = root_v    #Connect u to v
        else:                     #u, v have equal lengths
            p[root_u] = root_v    #Connect u to v
            r[root_v] = r[root_v] + 1 #Increment rank
```

36. *Path Compression

Postupné přepojení všech uzlů stromu ke kořenu.

Každému prvku na cestě ke kořenu upraven odkaz na kořen/prarodiče.

Provádí se při hledání kořene.

Snížení výšky stromu, urychlení operace FIND.

Dvě varianty:

- 1 *Dva průchody*
Nalezení kořenu stromu “klasicky”.
Oprava kořenů všech připojených uzlů.
- 2 *Jeden průchod*
Postupné přeskakování jedné generace.
Uzel přepojen na svého prarodiče.
Výška stromu klesne na $1/2$.

V praxi preferována varianta 2.

37. *Path Compression, implementace

Varianta s 1 průchodem:

Zkracování délky stromu přeskočením generace.

Postupné zkracování cesty (výšky stromu).

```
def find(u, p):
    while(p[u] != u):
        p[u] = p[p[u]]    #Move to grandparent
        u = p[u]          #Predecessor is grandparent
    return u
```

Varianta se 2 průchody:

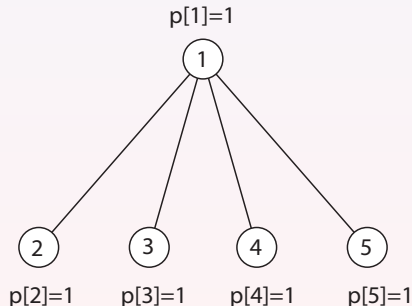
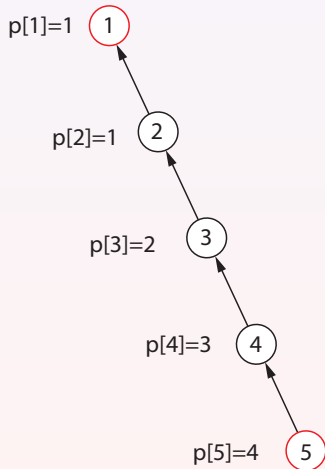
Přepojování všech uzlů na kořen.

Okamžité zkrácení stromu.

```
def find(u, p):
    while (p[u] != u):    #Find root
        u = p[u]
    root = u
    while u != root:
        up = p[u]         #Store predecessor
        p[u] = root       #Change predecessor to root
        u = up            #Go to parent
    return u
```

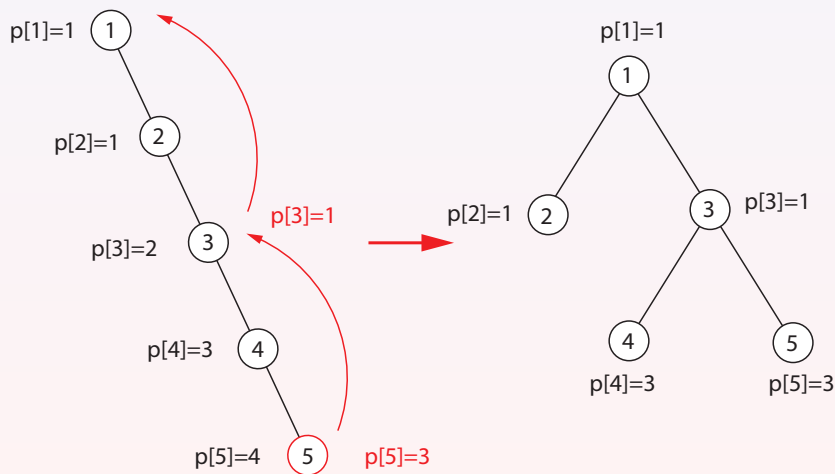
38. *Path Compression, I.

Přepojení všech uzlů cesty $< 2, 3, 4, 5 >$ ke kořeni 1.



39. *Path Compression, II.

Postupné zkracování cesty.



40. Datová reprezentace grafu

Kombinovaná reprezentace: seznam vrcholů V a hran E .

Reprezentace vrcholů:

Seznam vrcholů V v G .

$$V = [v_1, v_2, \dots, v_k]$$

Reprezentace hran:

Hrana: uspořádaná trojice - počáteční uzel u , koncový uzel v , ohodnocení w .

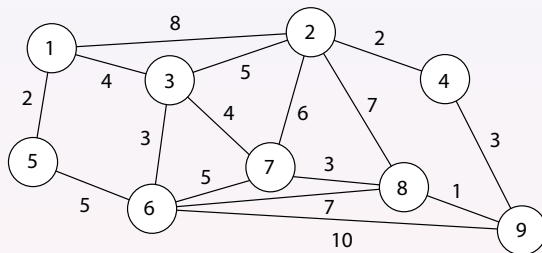
Hrany: spojová reprezentace, seznam hran E .

$$E = [\begin{array}{l} [u_1, v_1, w_1], \\ [u_2, v_2, w_2], \\ \dots \\ [u_k, v_k, w_k], \end{array}]$$

Hrany lze snadno seřadit dle w .

Dříve používané reprezentace umožňují snadnou konverzi na smíšenou.

41. Ukázka reprezentace grafu



Reprezentace uzlů:

$V = [1, 2, 3, 4, 5, 6, 7, 8, 9]$

Reprezentace hran:

$E = [[1, 2, 8], [1, 3, 4], [1, 5, 2], [2, 8, 7], [2, 1, 8], [2, 3, 5],$
 $[2, 4, 2], [2, 7, 6], [3, 1, 4], [3, 2, 5], [3, 6, 3], [3, 7, 4],$
 $[4, 9, 3], [4, 2, 2], [5, 1, 2], [5, 6, 5], [6, 8, 7], [6, 9, 10],$
 $[6, 3, 3], [6, 5, 5], [6, 7, 5], [7, 8, 3], [7, 2, 6], [7, 3, 4],$
 $[7, 6, 5], [8, 9, 1], [8, 2, 7], [8, 6, 7], [8, 7, 3], [9, 8, 1],$
 $[9, 4, 3], [9, 6, 10]]$

42. Ukázka operace MAKE_SET *

Vytvoření kořenových stromů tvořených 1 uzlem.

$p[1]=1$



$p[2]=2$



$p[3]=3$



$p[4]=4$



$p[5]=5$



$p[6]=6$



$p[7]=7$



$p[8]=8$



$p[9]=9$



43. Implementace Borůvkova/Kruskalova algoritmu

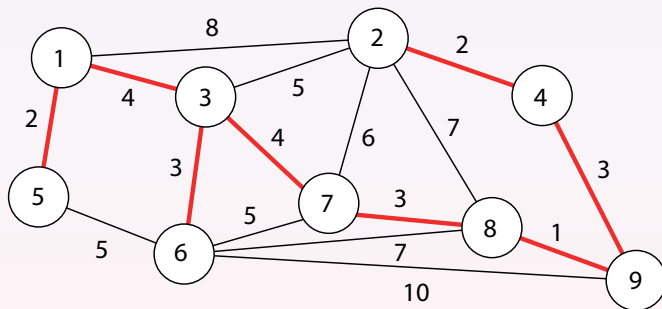
Operace MAKE_SET:

```
def make_set(u, p, r):
    p[u] = u
    r[u] = 0
```

Borůvkův/Kruskalův algoritmus:

```
def mstk(V, E):
    T=[]                                #Empty tree
    wt = 0                             #Sum of weights of T
    p = [inf] * (len(V) + 1)          #List of roots
    r = [inf] * (len(E) + 1)          #Rank of the node
    for v in V:                         #Make set
        make_set(v, p, r)              #Initilize p and r
    ES = sorted(E, key=lambda it:it[2]) #Sort edges by w
    for e in ES:                        #Process all edges
        u, v, w = e                   #Take an edge
        if (find(u, p) != find(v, p)): #roots u, v in different trees?
            union(u, v, p, r)          #Create union
            T.append([u, v, w])         #Add edge to tree
            wt = wt + w                 #Compute weight of T
    return wt, T
```

44. Výsledky, Borůvkův/Kruskalův algoritmus



```
wt, T = mst(V,E)
print(T)
>> [[8, 9, 1], [1, 5, 2], [2, 4, 2], [3, 6, 3], [4, 9, 3],
      [7, 8, 3], [1, 3, 4], [3, 7, 4]]
print(wt)
>> 22
```