Minimální kostra.

Borůvkův/Kruskalův algoritmus. UNION-FIND. Jarníkův/Primův algoritmus.

Tomáš Bayer | bayertom@natur.cuni.cz

Katedra aplikované geoinformatiky a kartografie, Přírodovědecká fakulta UK.

Obsah přednášky

1. Úvod
2. Kostra grafu
3. Borůvkův/Kruskalův algoritmus
   - UNION-FIND
   - Weighted UNION-FIND
4. Jarníkův/Primův algoritmus

1. 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).

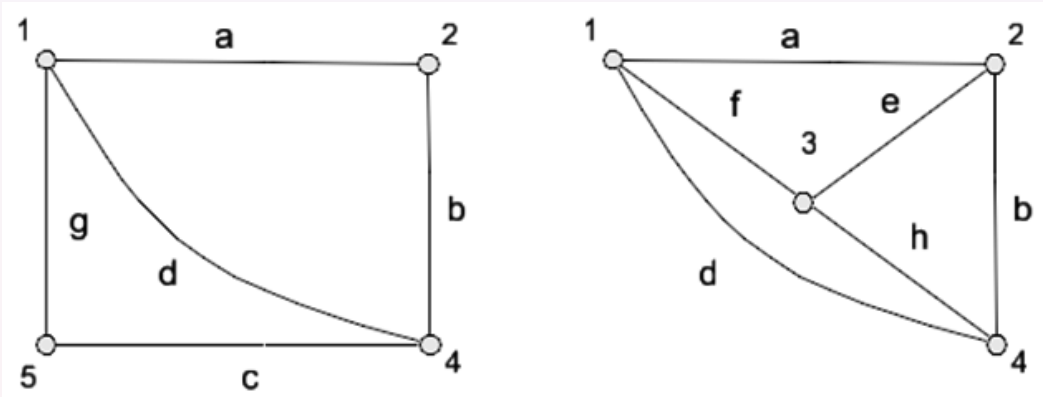
Z hlediska spolehlivosti *není optimální*.
Výpadek 1 hrany: rozpad G na 2 nesouvislé grafy.
Důsledek: přerušení dodávky energie, neprůjezdnost (nehoda), atd.
Kritické části infrastruktury nutno posílit: zdvojení.

Předpoklad G neorientovaný, souvislý, nezáporné hrany.

2. Podgraf a faktor

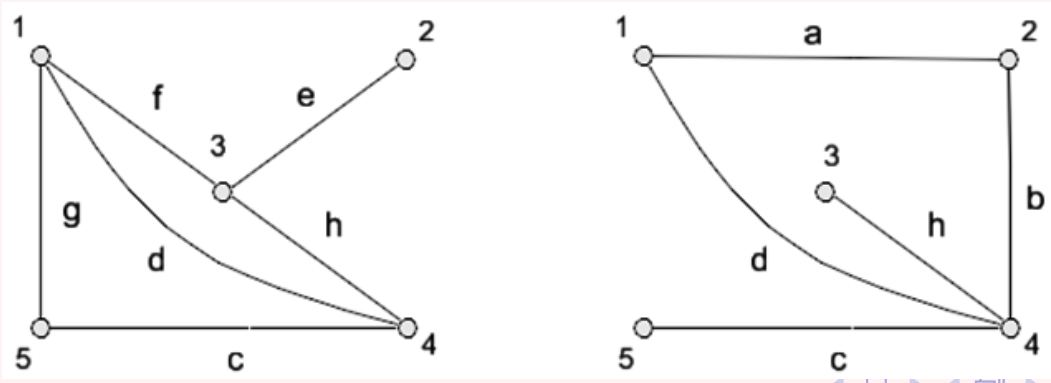
Podgraf $G' \subset G$:

$G' = (H', U', \rho')$ 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.



3. 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.

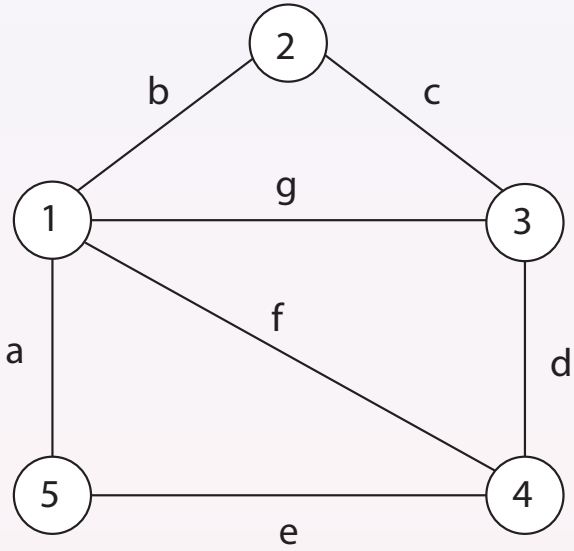
Úplný graf G, počet koster N_k (Cayley, 1889)

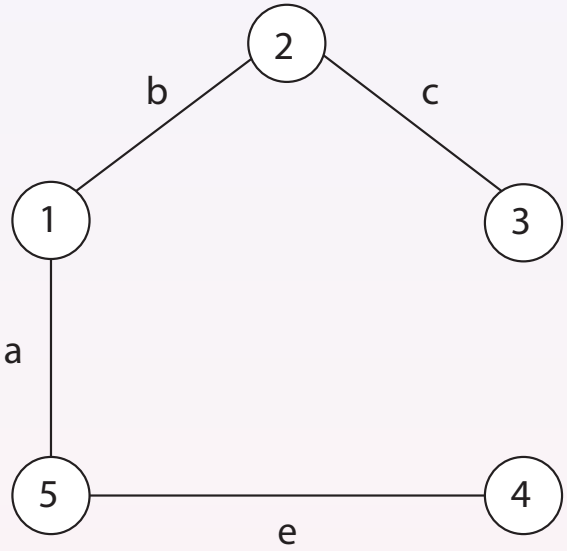
$$N_k = n^{n-2}.$$

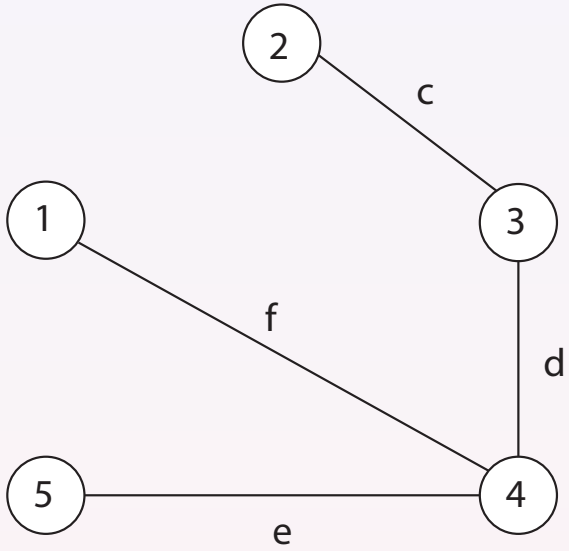
Hledání všech koster neefektivní, exponenciální růst.
Lze provést pouze pro malé grafy.

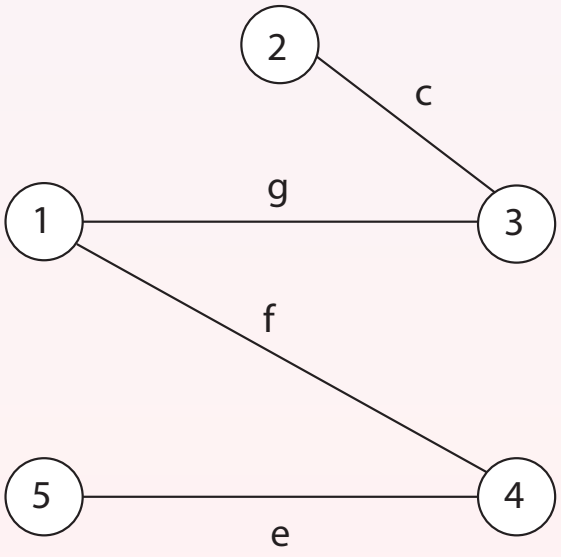
V praxi nás zajímá kostra s minimálním ohodnocením, tzv. **minimální kostra**.

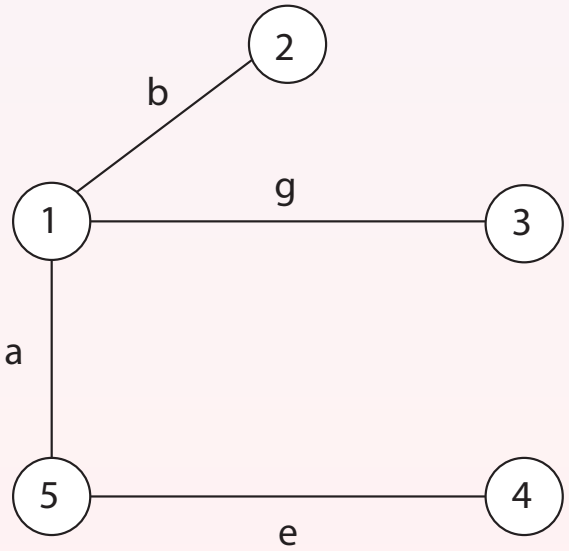
4. Graf a některé jeho kostry

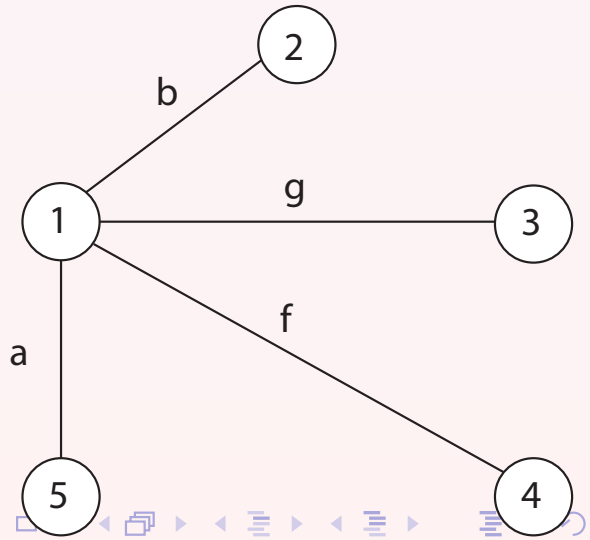












5. Minimální kostra grafu

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

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

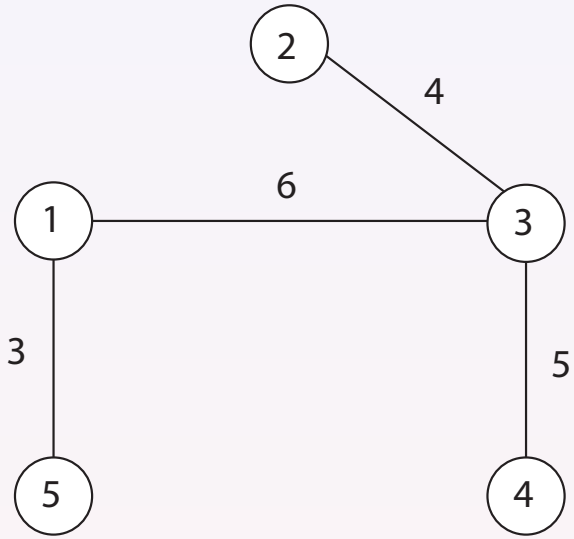
Úloha není jednoznačná, existence více minimálních koster.

Přehled algoritmů:

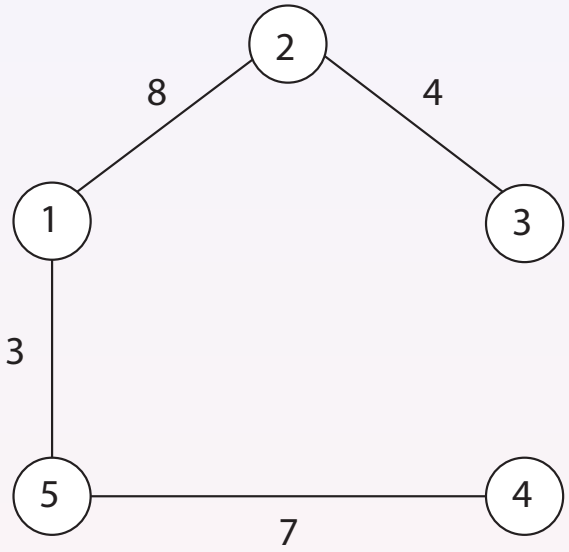
- *Borůvkův (modifikace Kruskal)*
  Vhodný pro řídké grafy, $O(\|H\| \log \|U\|)$.
- *Jarníkův (modifikace Prim)*
  Vhodný pro grafy s mnoha hranami, $O(\|U\| \log \|U\|)$.

Modifikované BFS vs. sjednocování podstromů v les.

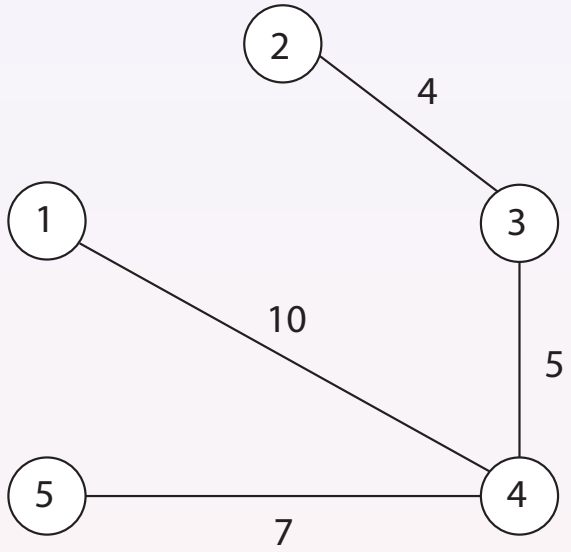
6. Ukázka minimální kostry



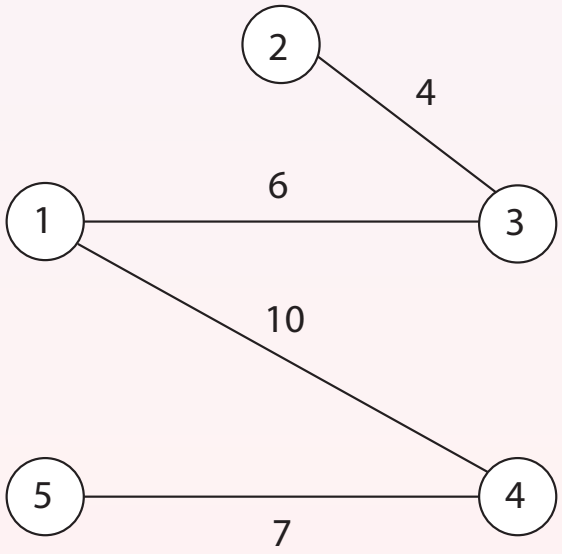
$C_{min} = 18$



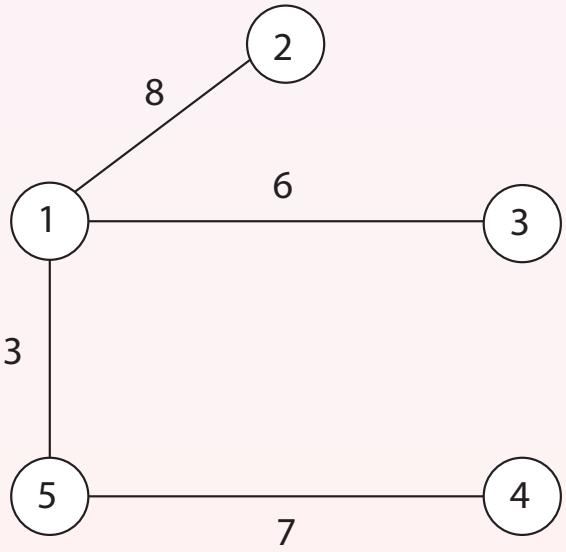
C = 22



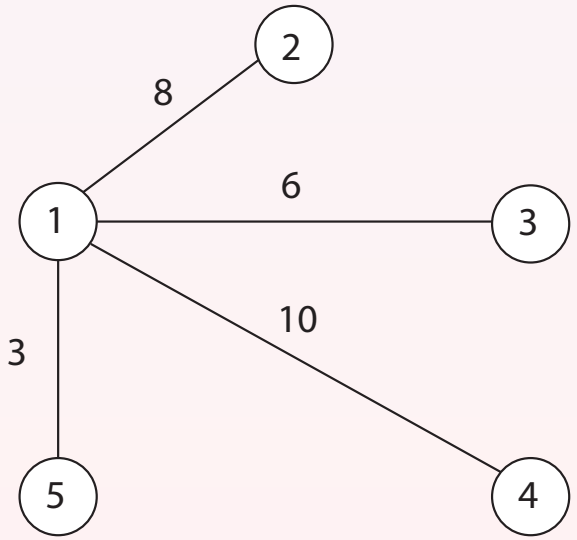
C = 26



C = 27



C = 24



C = 27

7. 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.

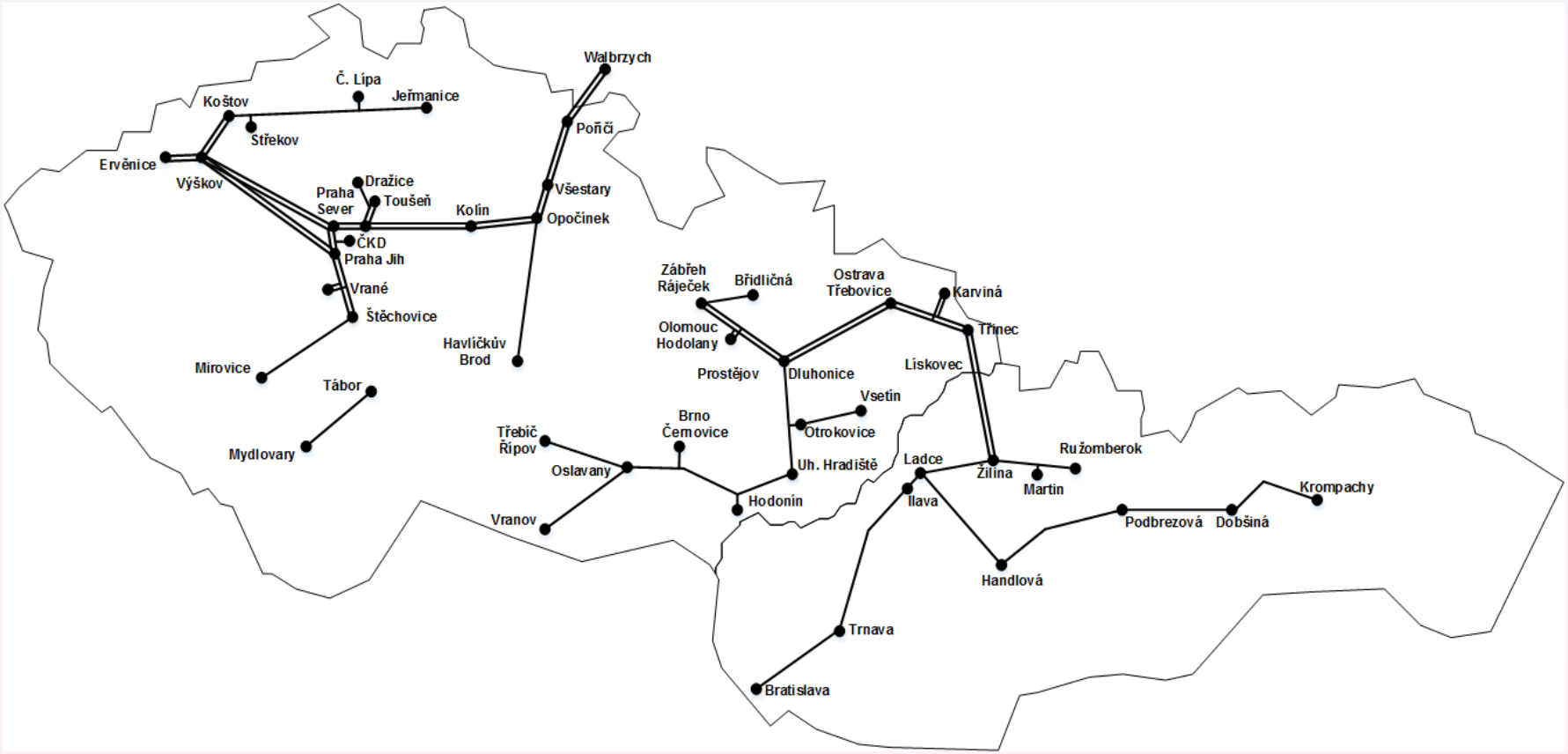
Pokračujeme, dokud les není stromem.

Opakované sjednocování: UNION, 2 podstromy.

Strom: souvislý, neobsahuje kružnici.

Les: nesouvislý, neobsahuje kružnici, tvořen podstromy.

8. Schéma elektrické sítě, ČSR



9. Princip Borůvkova/Kruskalova algoritmu

Předzpracování hran: setřídění hran dle w.

Implementace využívá prioritní frontu.

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

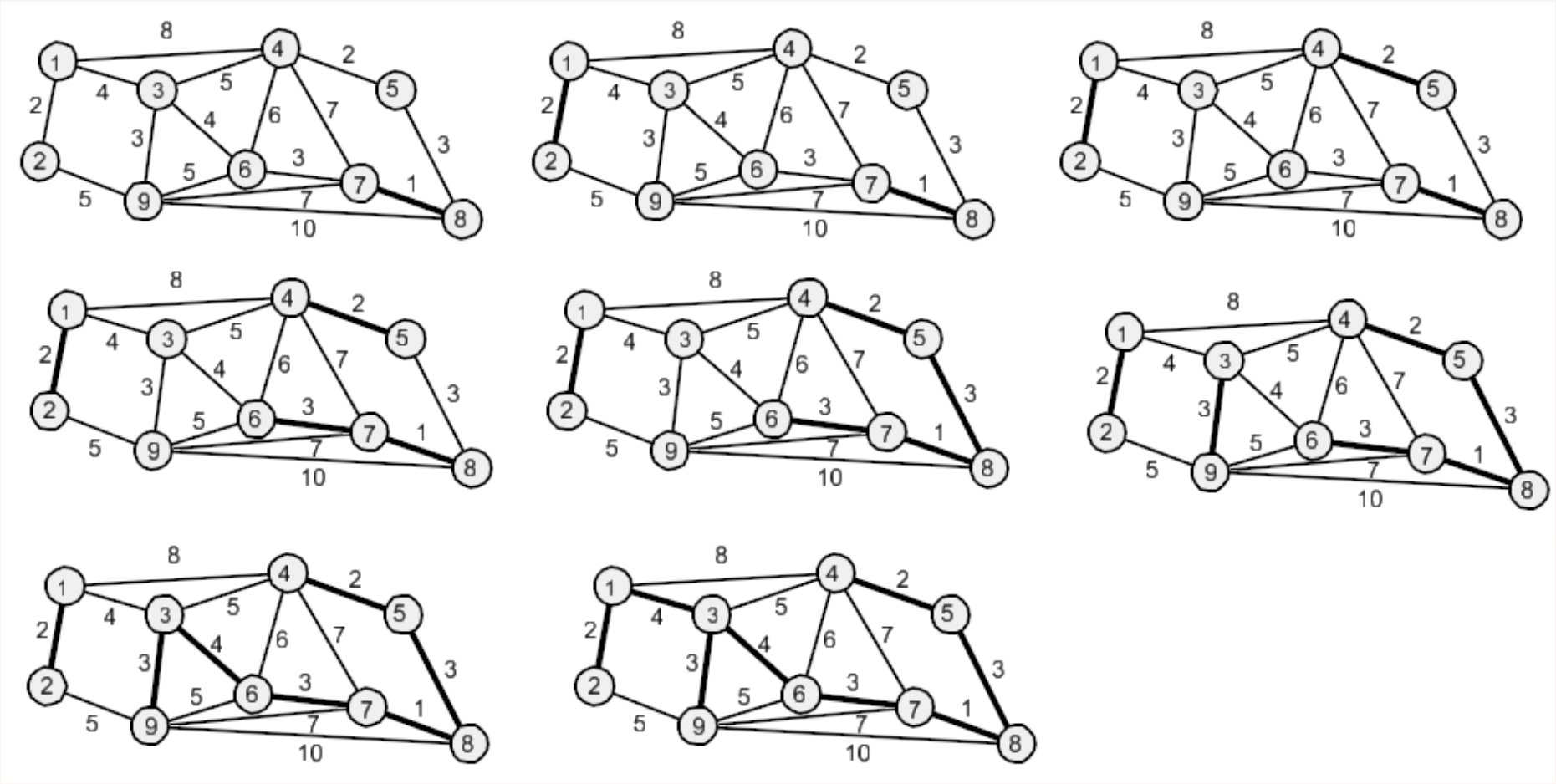
Mohou nastat 4 situace:

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:

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

10. Ukázka Borůvkova/Kruskalova algoritmu



11. 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 : {6,7,8}			9 : {9}
5	(3,9)	3	1 : {1,2}		3 : {3,9}	4 : {4,5}		7 : {6,7,8}			
6	(5,8)	3	1 : {1,2}		3 : {3,9}	7 : {6,7,8,4,5}					
7	(3,6)	4	1 : {1,2}		7 : {6,7,8,4,5,3,9}						
8	(1,3)	4	7 : {6,7,8,4,5,3,9,1,2}								

Minimální kostra G:

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

12. UNION-FIND

Základem kombinovaná operace UNION-FIND = UNION + 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.
  FIND_SET $O(1)$, UNION $O(\log_2(n))$.
- *Tree-based (strom)*
  Pomalejší nalezení, rychlejší spojení.
  FIND_SET $O(\log_2(n))$, 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).

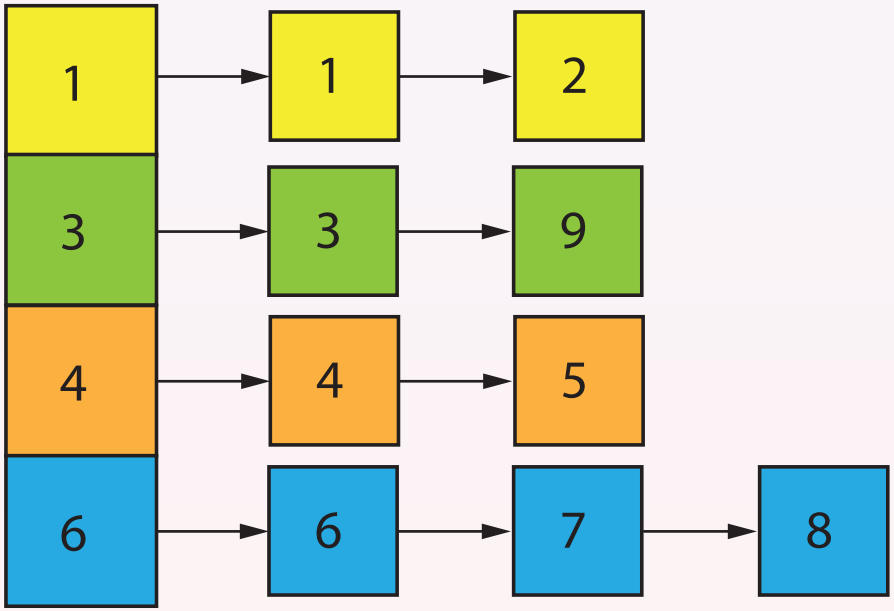
13. UNION-FIND, AB

3 seznamy: podstromy, jejich délky a indexační struktura.

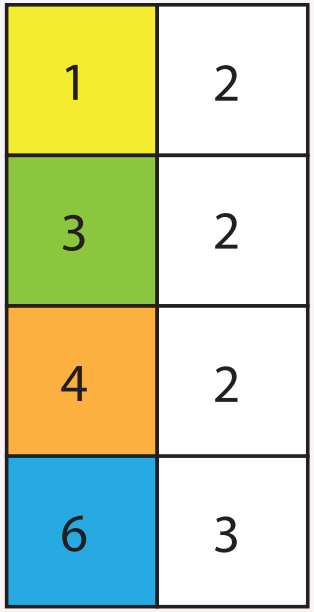
Pro podstrom uchovává seznam uzlů a identifikátor.

Identifikátorem zpravidla prvek s nejmenším ID.

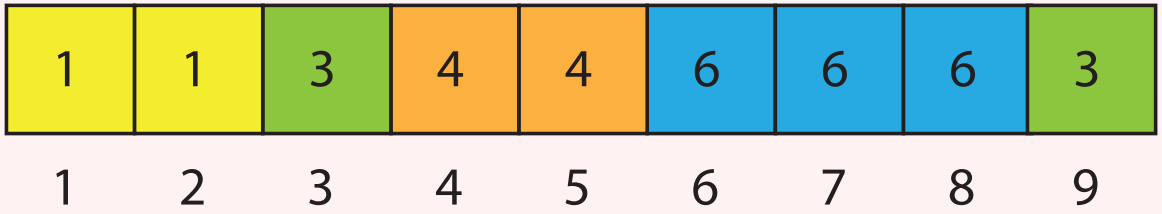
UF: linked list, identificators



UF: sizes



UF: 1D array, index

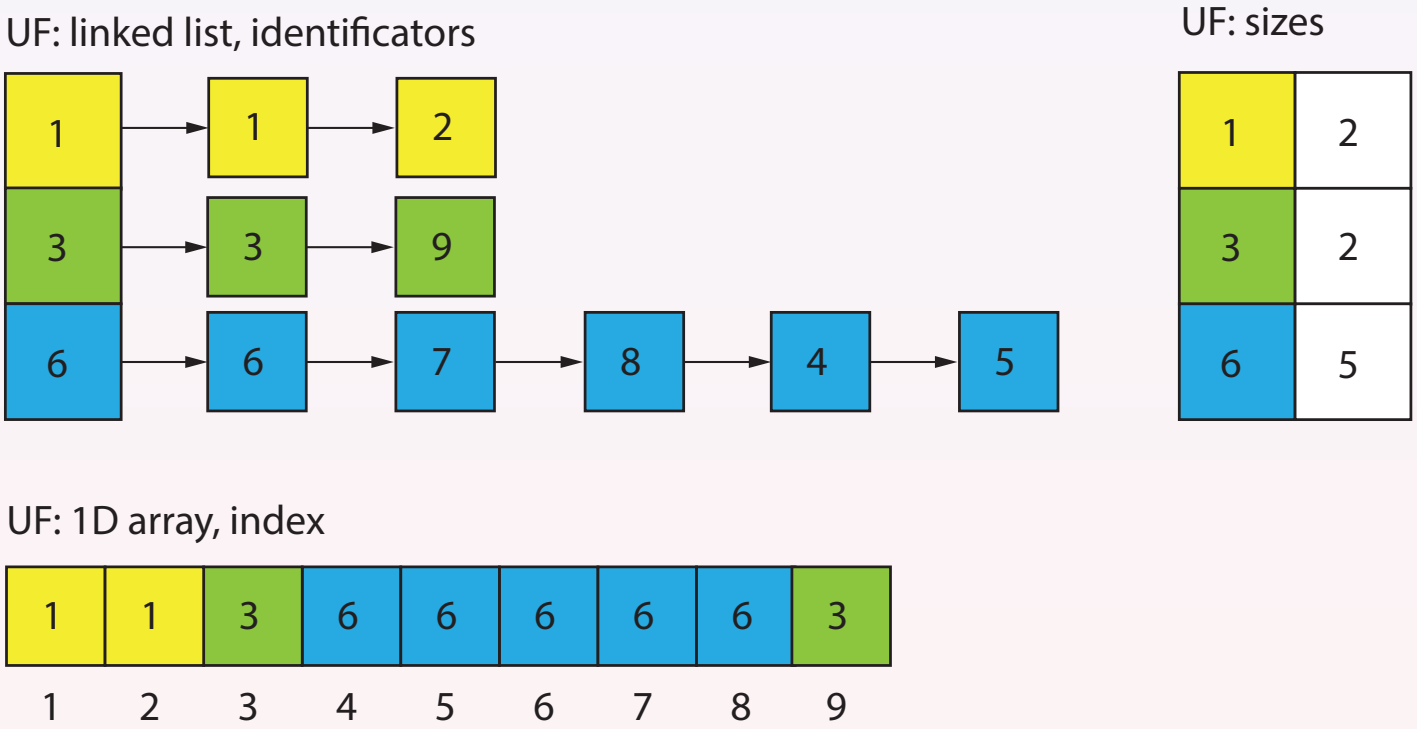


14. Ukázka Weighted UNION(4,6), AB

Weighted Union:

Za delší pole připojíme kratší.

Změna identifikátorů pro kratší seznam, delší seznam neměněn.



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

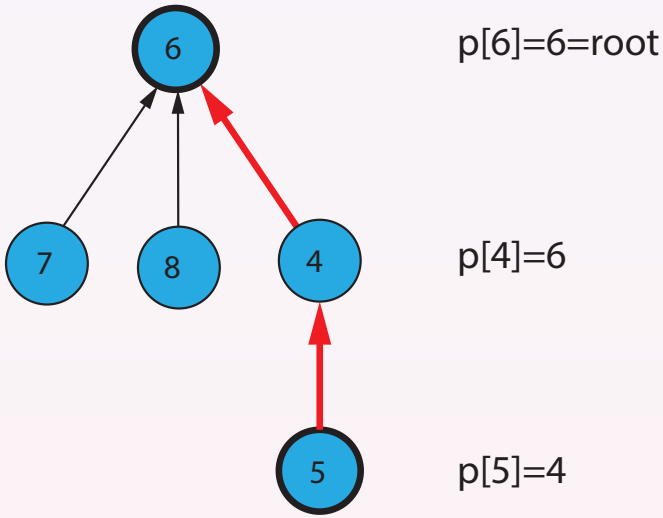
```
def union(u, v, index):
    if len(u)<len(v):
        for i in u:                    #Process all  elements in shorter array
            index[i] = index[v]        #Change all indices
        size[v] = size[v] + size[u]    #Change size
```

15. 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 $p[u] != u$.

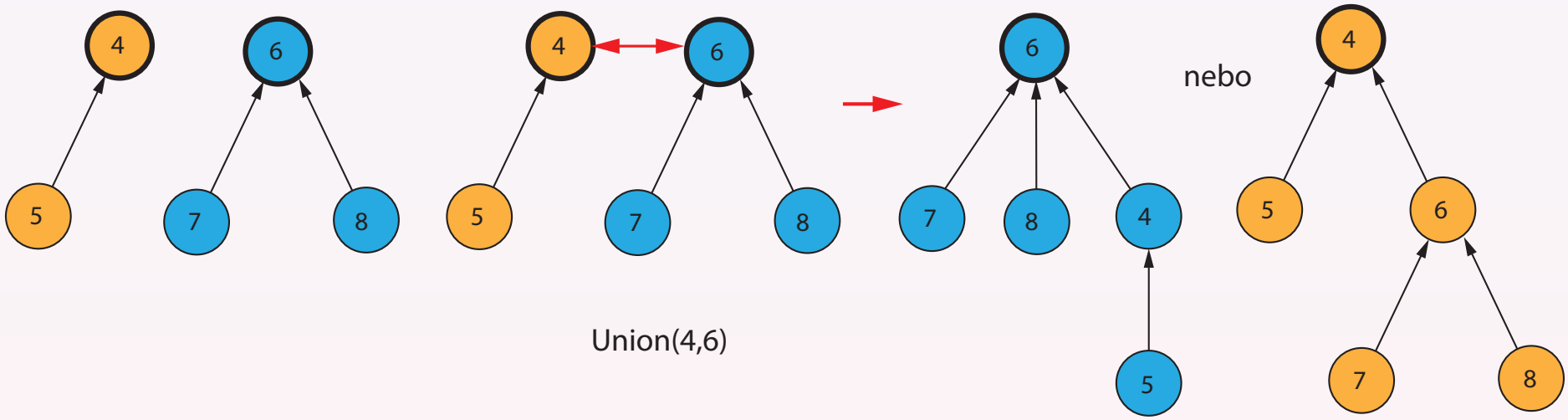
```
def find(u, p):
    while(p[u] != u):
        u = p[u]
    return u
```

16. 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):                 #Union + Find
    root_u = find(u, p)             #Find first root
    root_v = find(v, p)             #Find second root
    p[root_u] = root_v              #Connect second tree to the first one
```

Připojování delšího stromu za kratší neefektivní.

17. 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:

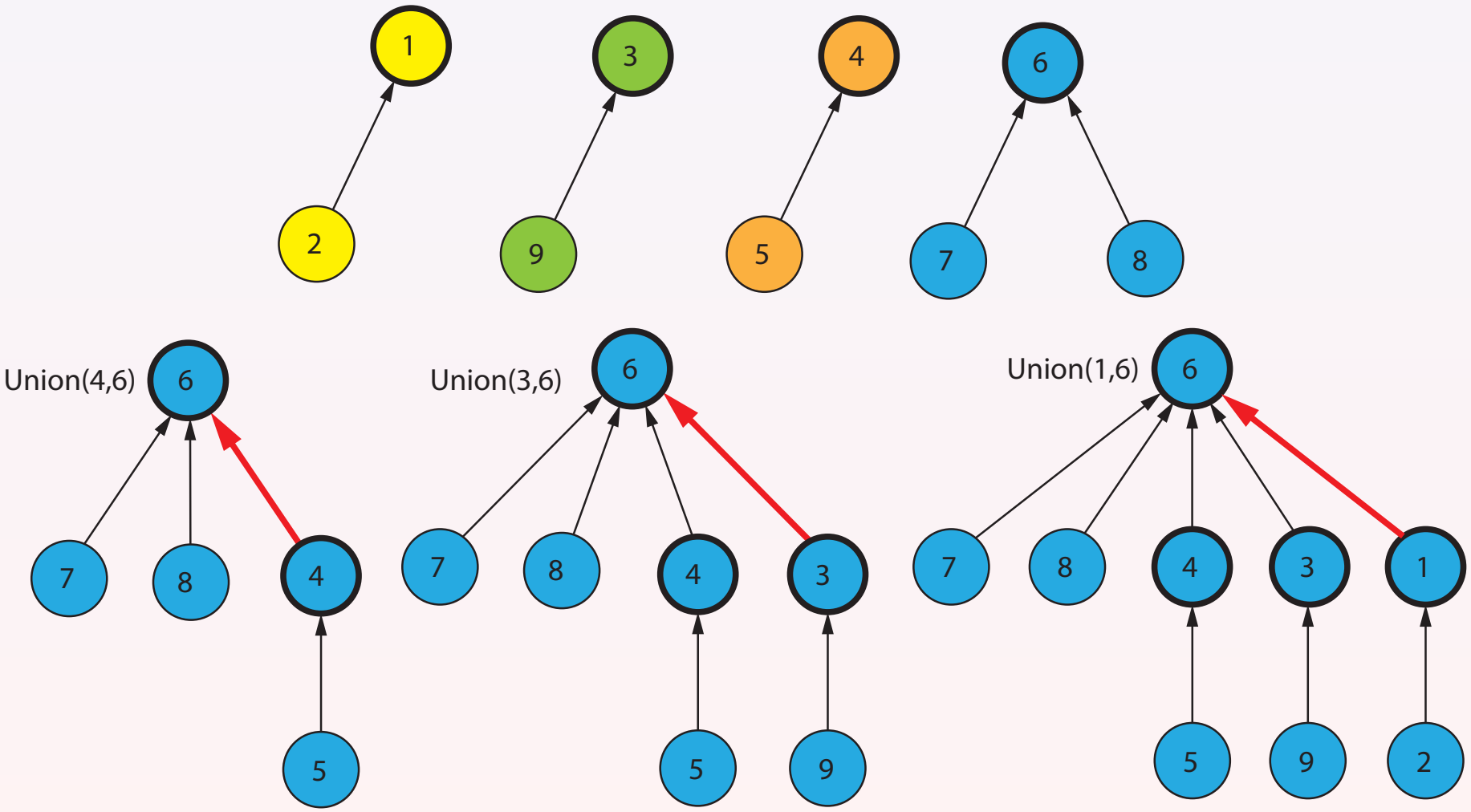
Zkracování cesty (stromu) $\Rightarrow$ snižování výšky.

Upravení odkazu na kořen všem uzlům ve stromě.

Heuristika, dvouprůchodová, rekurzivní/nerekurzivní implementace.

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

18. Weighted UNION-FIND, TB *



19. Operace Weighted UNION(u, v) *

Weighted UNION:

Pro každý uzel u nutno uchovávat 2 informace:

- parent (kořen) $p[u]$, inicializace $p[u] = u$,
- hodnost (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 $r[u] = r[v]$,

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

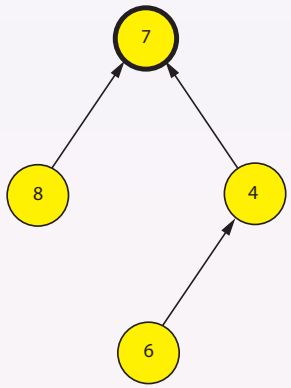
Weighted UNION-FIND:

FIND + Weighted UNION.

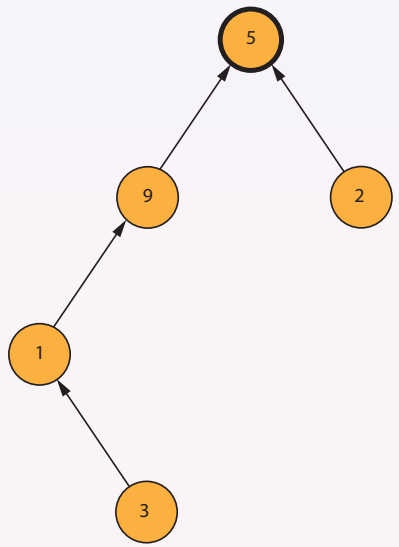
1. FIND:
   Nalezení kořenů R_u, R_v stromů obsahujících u, v.
2. Weighted UNION
   Pokud u, v jiném podstromu, porovnáme výšky stromů:
   a) Jestliže $r[u] > r[v]$, připojíme v k u: $p[R_v] = R_u$.
   b) Jestliže $r[v] > r[u]$, připojíme u k v: $p[R_u] = R_v$.
   c) Při shodě $r[u] = r[v]$: $p[R_v] = R_u$ (např.).
   Update výšky $r[u] = r[u] + 1$.

20. Weighted UNION(u, v), update rank$_u$ *

r(7) =2

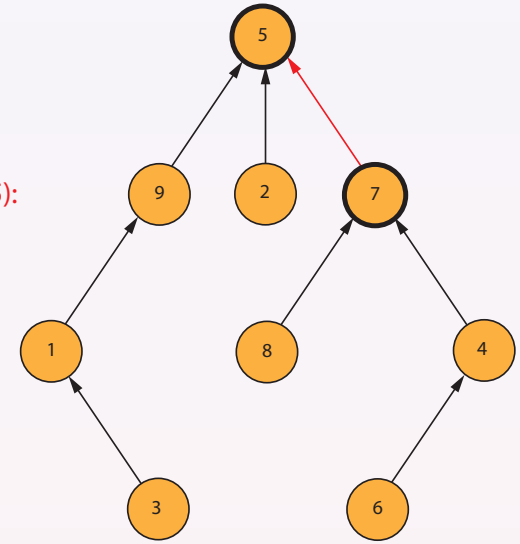


r(5) =3

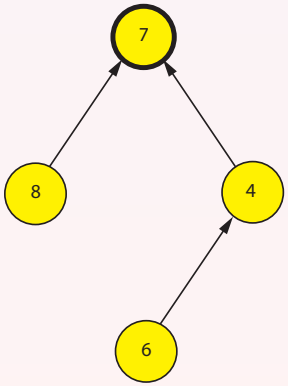


Union(7,5):

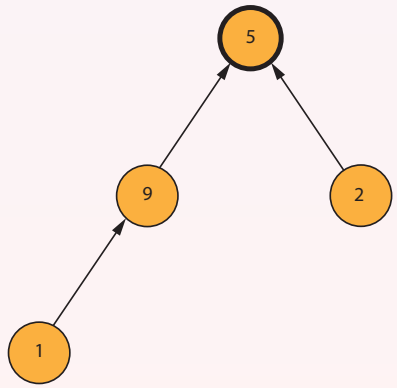
r(5) = max(r(7), r(5)) = 3



r(7) =2

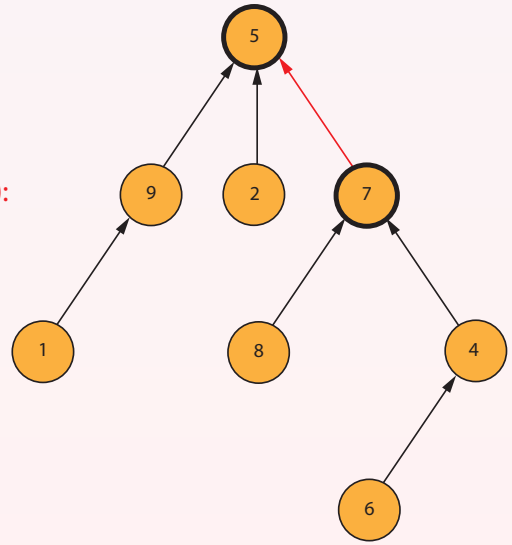


r(5) =2



Union(7,5):

r(5) = r(5) +1 = r(7) + 1 = 3



21. 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
        elif u!= v:                    #u, v have equal lengths
            p[root_u] = root_v         #Connect u to v
            r[root_v] = r[root_v]+1    #Increment rank
```

22. 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.

23. 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:

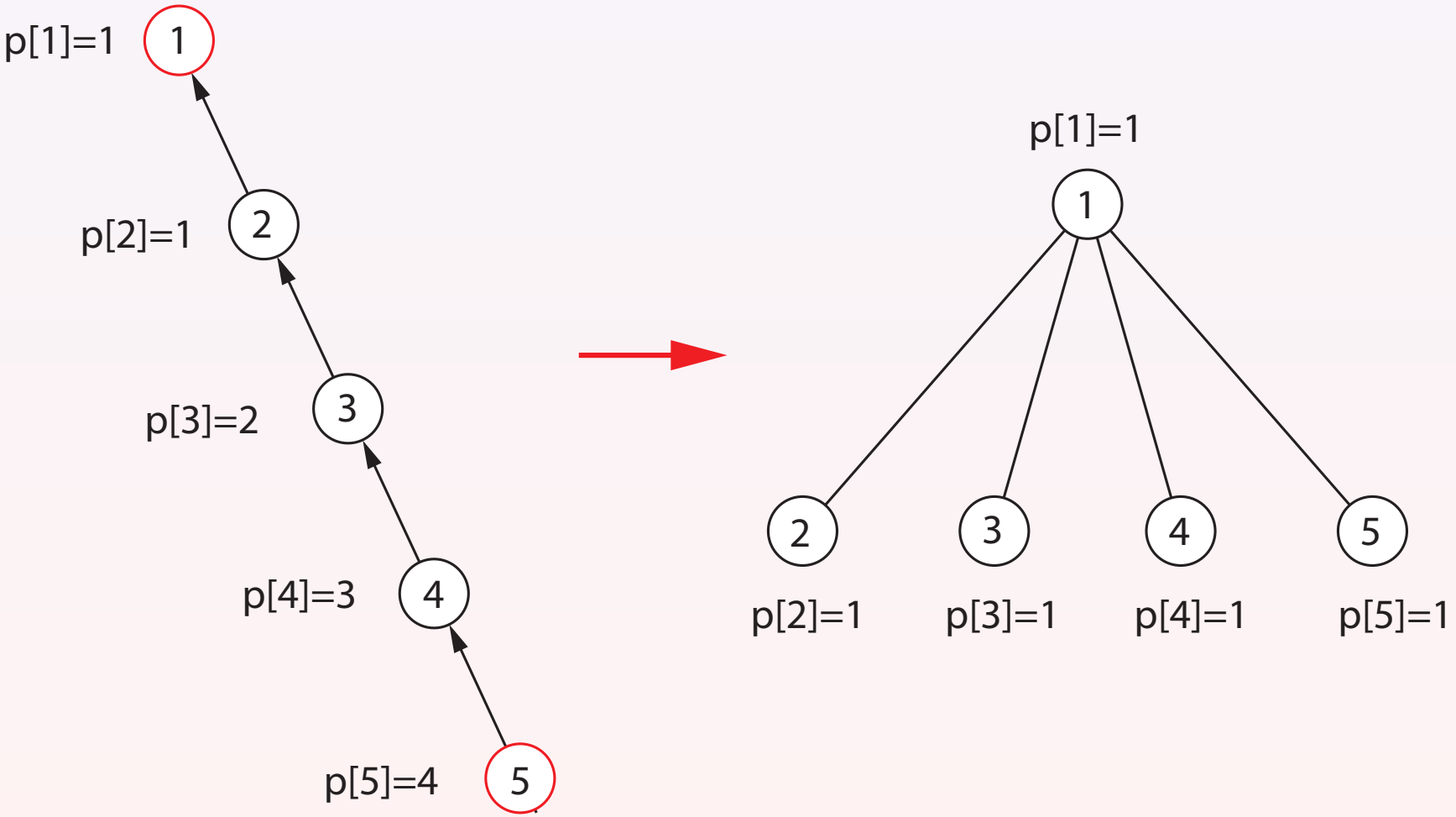
Přepojování všech uzlů na kořen.

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
```

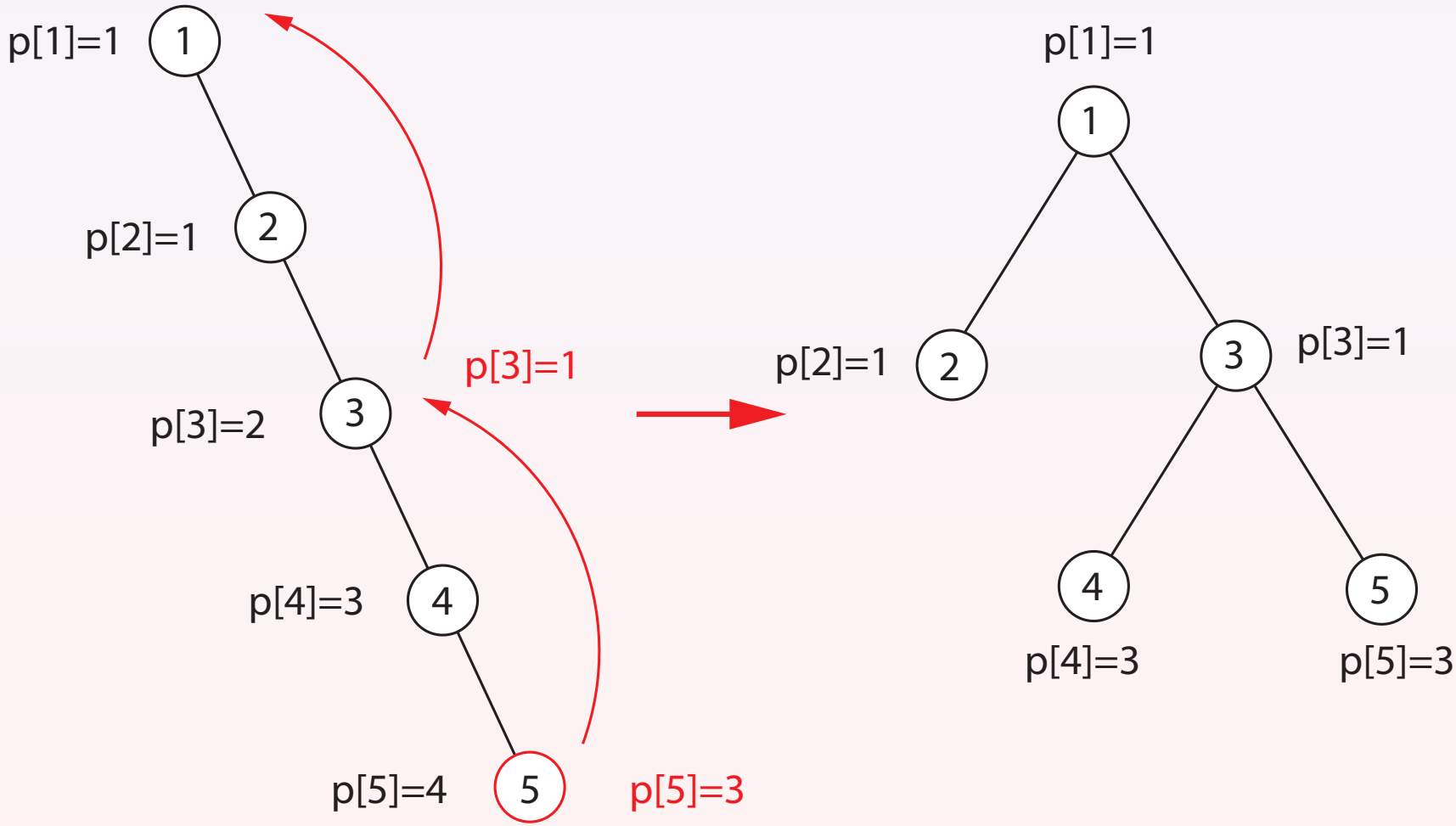
24. Path Compression, I. *

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



25. Path Compression, II. *

Postupné zkracování cesty.



26. Datová reprezentace grafu

Kombinovaná reprezentace: seznam vrcholů V a hran E.

Reprezentace vrcholů:

Seznam vrcholů V v G.

```
V = [v1, v2, ...,vk]
```

Reprezentace hran:

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

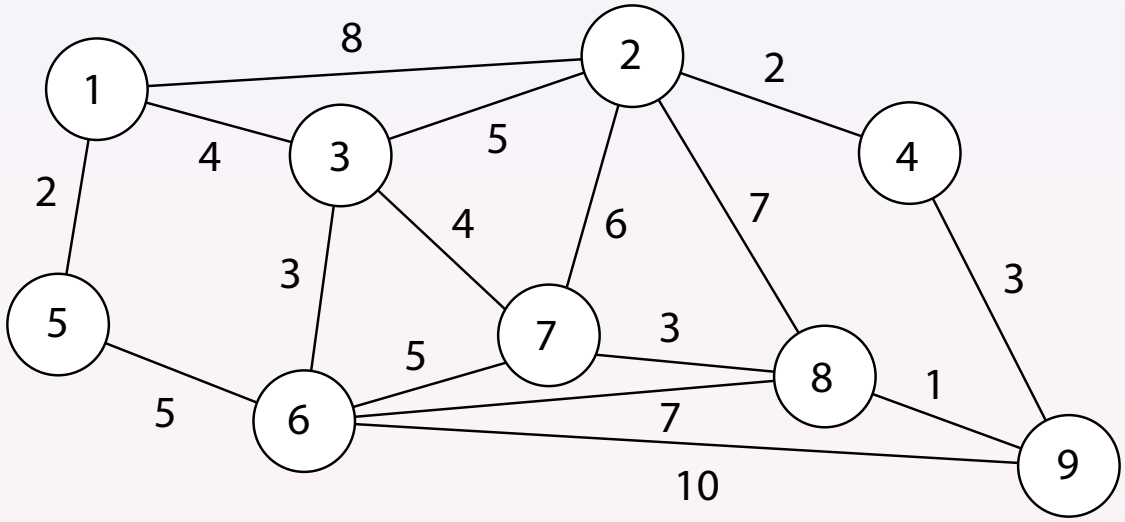
Hrany: spojová reprezentace, seznam hran E.

```
E = [
      [u1, v1, w1],
      [u2, v2, w2],
       ....
      [uk, vk, wk],
    ]
```

Hrany lze snadno setřídit dle w.

Dříve používané reprezentace umožňují snadnou konverzi na smíšenou.

27. Ukázka reprezentace grafu



Reprezentace uzlů:

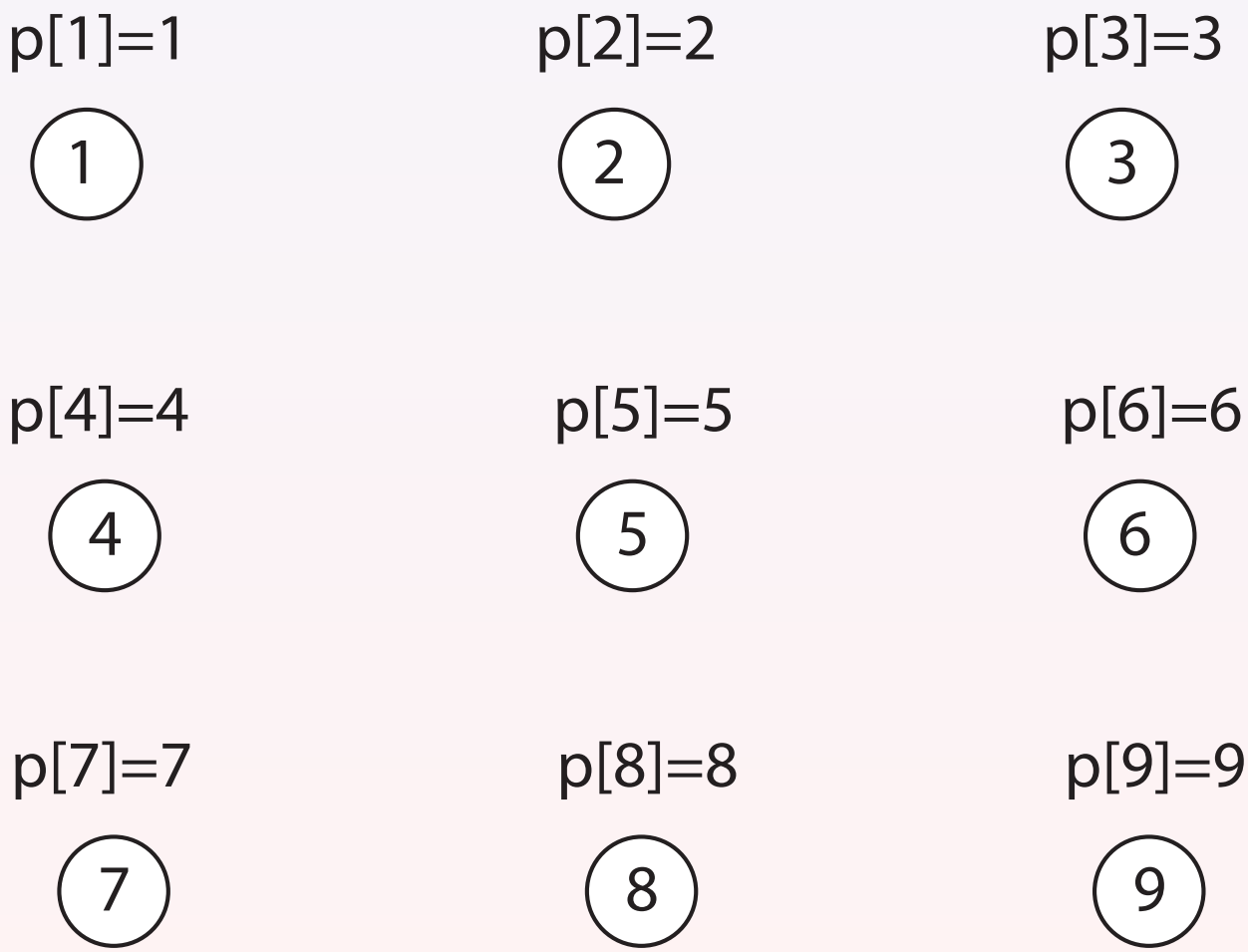
```
V = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

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]]
```

28. Ukázka operace MAKE_SET *

Vytvoření kořenových stromů tvořených 1 uzlem.



29. Implementace Borůvkova/Kruskalova algoritmu

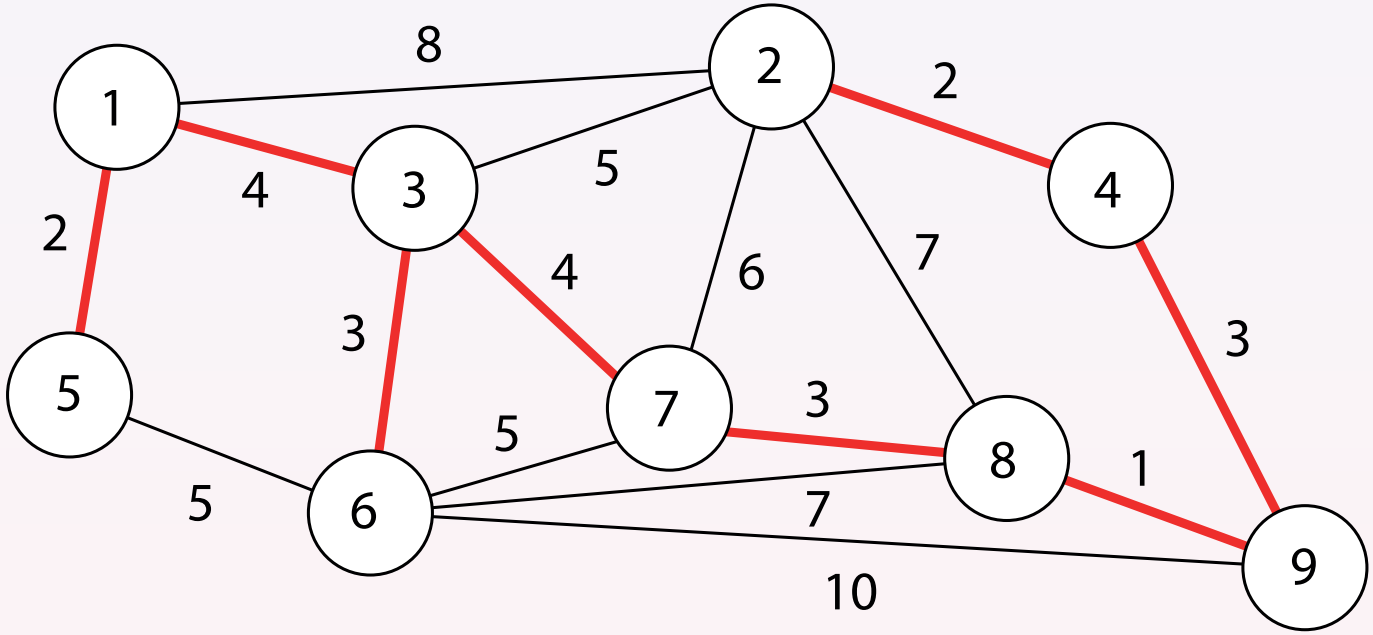
Operace MAKE_SET:

```
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
```

30. 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
```

31. Jarníkův/Primův algoritmus

Objeven prof. Vojtěchem Jarníkem (1897-1970) v roce 1930.
Prof. Jarník působil v letech 1921-23 na PřF UK.
Znovu objeven: 1957, R. C. Prime, 1959 E. W. Dijkstra.

Borůvkův algoritmus pracuje současně s více podstromy, které sjednocuje.
Jarníkův algoritmus pracuje pouze s jedním podstromem, který rozšiřuje.

Využívá greedy strategii.
Velmi podobný Dijkstra, liší se *způsobem relaxace*.
K aktuálnímu uzlu u hledáme v minimalizující $d[v] = w(u, v)$.
Vrcholy možno značkovat.

Implementačně jednodušší než Borůvkův/Kruskalův.
Klíčový prvkem rychlost hledání v prioritní frontě Q.
Při implementaci Q haldou $O(n)$, pomalé, nutno značkovat $O(1)$.

32. Popis Jarníkova/Primova algoritmu

Uzly, které nejsou v podstromu, uloženy v prioritní frontě Q setříděné dle $d[v]$.
Pokud nelze v Q hledat $O(n)$, nutno značkovat $O(1)$.
Při inicializaci všem $u \in U$ nastaveny $d[u] = \infty$.

Lze začít od libovolného uzlu.
Ten označen jako root r, $d[r] = 0$, zpravidla $r = 0$.

Dokud Q není prázdná vyjmeme uzel $(d[u], u)$.
Pro každé dosud nenavštívené v incidující s u provádíme relaxaci.

Upravená relaxace:
Relaxační pravidlo má tvar

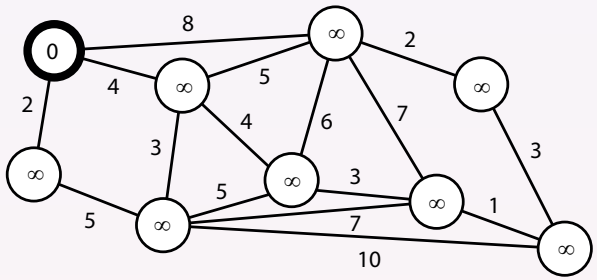
$$d[v] > w[u][v] \rightarrow d[v] = w(u, v), \qquad p[v] = u.$$

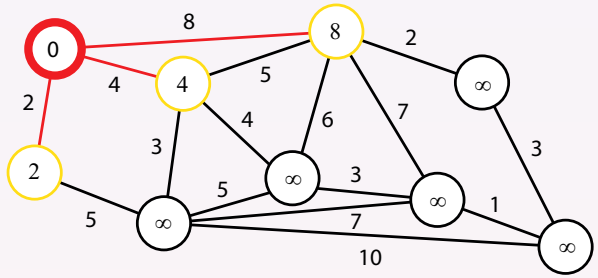
Hodnota $d[v]$ neobsahuje celkovou délku cesty $s \rightarrow u$ (Dijkstra).
Reprezentuje vzdálenost v od kostry: $d[v] = w(u, v)$.

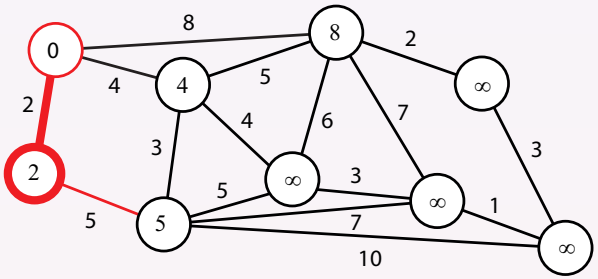
Pro uzel v uložíme předchůdce $p[v]$.
Následně přidáme $(d[v], v)$ do Q.

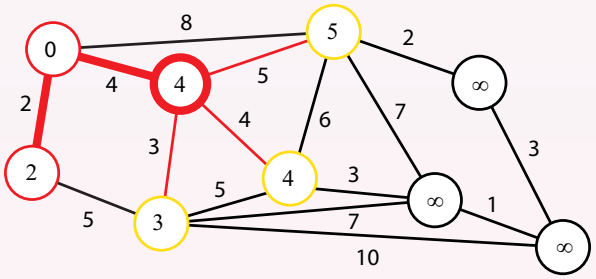
Rekonstrukce kostry: nalezneme $p[u]$ pro všechna $u \in U$.

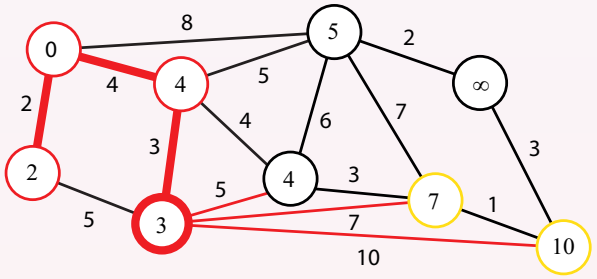
33. Ukázka Jarníkova/Primova algoritmu

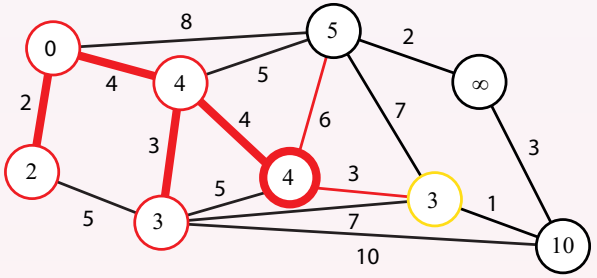


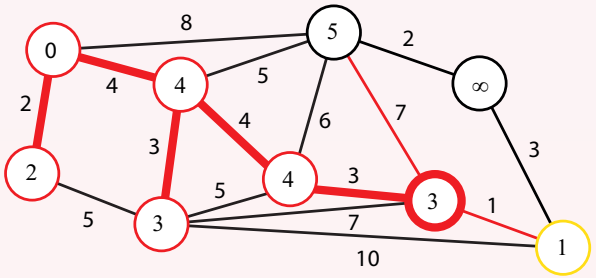


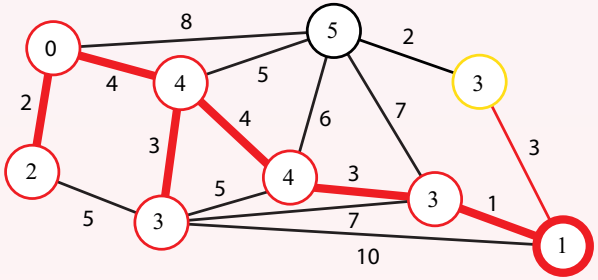


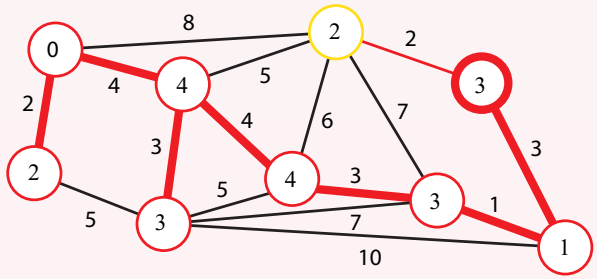












34. Implementace Jarníkova/Primova algoritmu

Reprezentace *G* spojovým seznamem.

```
def mstj(G, r):
    s = ['N'] * (len(G) + 1)           #No vertex is in mst
    d = [float('Inf')] * (len(G) + 1) #Set infinite distances
    p = [-1] * (len(G) + 1)            #No predecessors
    Q = queue.PriorityQueue()          #Priority queue
    d[r] = 0                           #Root d[r] = 0
    Q.put((0, r))                      #Add start vertex
    while not Q.empty():
        du, u = Q.get()                #Pop first element
        s[u] = 'C'                     #Node belongs to mst, close
        for v, wuv in G[u].items():
            if s[v] == 'N':            #Relaxation, v not in mst
                if d[v] > wuv:         #We found a better node
                    d[v] = wuv         #Update distance from mst
                    p[v] = u           #Update predecessor
                    Q.put((d[v], v))   #Add to Q
```

Ukázka:

```
mstj(G,1)
>> [-1, 4, 1, 9, 1, 3, 3, 7, 8]
```

35. Výsledky, Jarníkův/Primův algoritmus