Úvod do grafových algoritmů.

Orientovaný, neorientovaný graf. BFS. DFS. Dijkstra algoritmus.

Tomáš Bayer | bayertom@natur.cuni.cz

Katedra aplikované geoinformatiky a kartografie, Přírodovědecká fakulta UK.

Obsah přednášky

1. Graf
   - Spojová reprezentace grafů
2. Prohledávání grafu
   - Prohledávání grafu do šířky
   - Prohledávání grafu do hloubky
3. Nejkratší cesta mezi 2 uzly

1. Graf (1/2)

Graf: datová struktura popisující vztahy mezi objekty.

Datová struktura tvořená uzly U a hranami H.

Zpravidla konečný počet $\Rightarrow$ konečný graf.

Topologicko/geometrický model skutečnosti.

Informace o poloze méně důležitá než vzájemný vztah.

Lze znázornit nekonečně mnoha způsoby.

Široké využití v mnoha oblastech:

Úlohy o dopravním spojení, logistické problémy, optimální trasa, plánování, navigace, propustnost sítě, přenos energie, komprese dat.

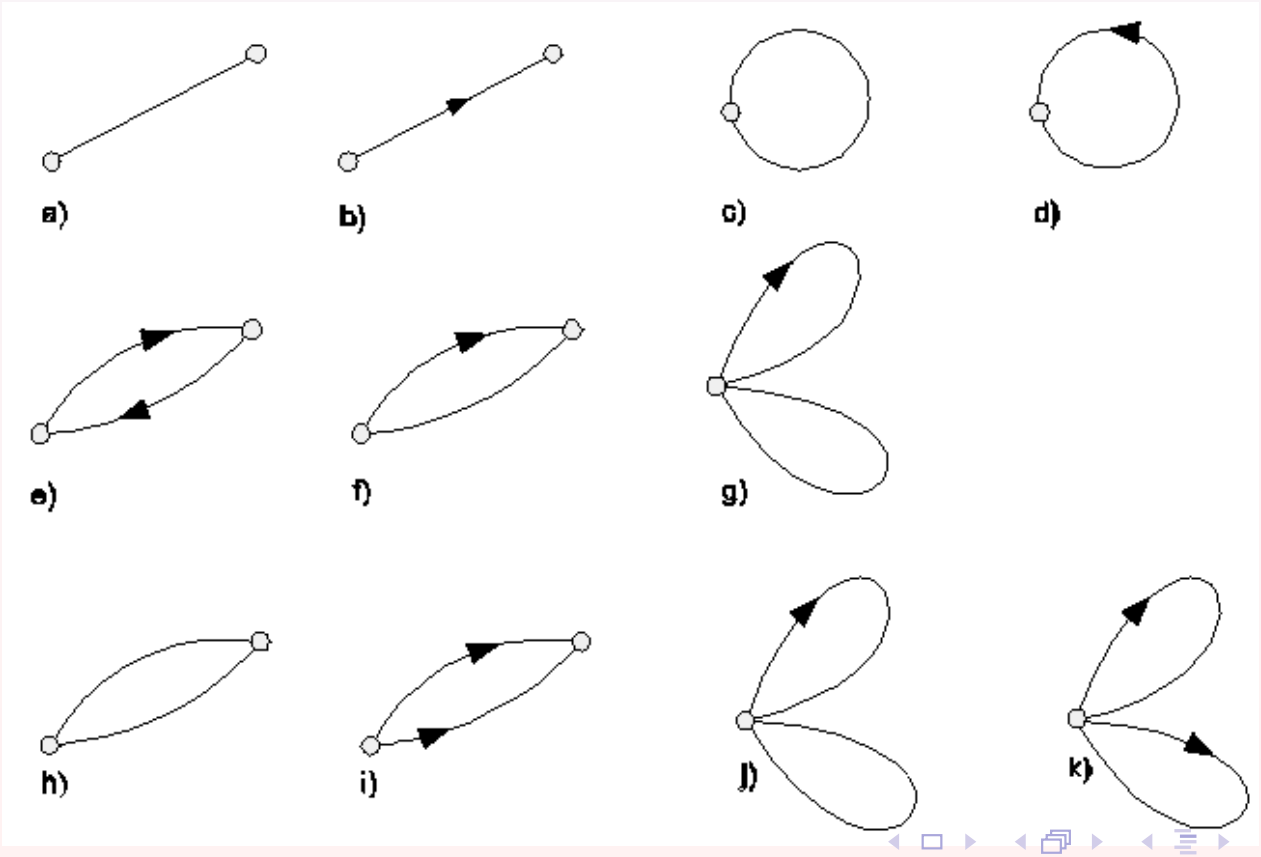
Dělení grafů:

- neorientované,
- orientované,
- částečně orientované.

2. Graf (2/2)

Datová struktura tvořená uzly U a hranami H.
Zpravidla konečný počet $\Rightarrow$ konečný graf.
Lze znázornit nekonečně mnoha způsoby.

Přímá neorientovaná hrana (a), orientovaná hrana (b), neorientovaná smyčka (c), orientovaná smyčka (d), rovnoběžné hrany (e, f), rovnoběžná hrana se smyčkou (g), násobné hrany (h, i), násobné hrany se smyčkou (j, k).



3. Neorientovaný graf

Uspořádaná trojice disjunktních množin

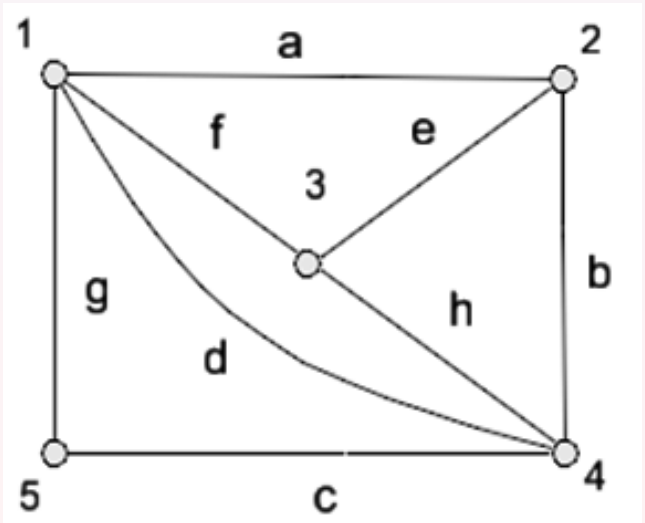
$$G = \langle H, U, \rho \rangle ,$$

kde H představují hrany, U uzly, ρ incidenci grafu G.

Incidence

$$\rho : H \to U \otimes U,$$

přiřazuje každé hraně z množiny H neprázdnou množinu dvojic uzlů z množiny U.



Množina uzlů: $U = \{1, 2, 3, 4, 5\}$,

Množina hran: $H = \{a, b, c, d, e, f, g, h\}$,

Incidence: $\rho(a) = \{1, 2\}$, $\rho(b) = \{2, 4\}$, $\rho(c) = \{4, 5\}$, $\rho(d) = \{1, 4\}$, $\rho(e) = \{2, 3\}$, $\rho(f) = \{1, 3\}$, $\rho(g) = \{1, 5\}$, $\rho(h) = \{3, 4\}$.

4. Sled, tah, cesta

Sled: uspořádaná posloupnost uzlů a hran,

$$\langle 1, a, 2, e, 3, f, 1, d, 4, b, 2, e, 3, h, 4, c, 5 \rangle .$$

Tah: sled, ve kterém se vyskytuje každá hrana nejvýše jednou,

$$\langle 1, a, 2, e, 3, f, 1, g, 5, c, 4, b, 2 \rangle .$$

Uzavřený tah: začíná a končí ve stejném uzlu.

Cesta: tah, ve kterém se každý uzel vyskytuje nejvýše jednou,

$$\langle 3, f, 1, a, 2, b, 4, c, 5 \rangle .$$

Kružnice: uzavřená cesta, začíná a končí ve stejném uzlu,

$$\langle 3, f, 1, g, 5, c, 4, b, 2, e, 3 \rangle .$$

Stupeň uzlu $d(u)$:

Počet hran, které incidují s uzlem u, důležitá sudost, lichost.

$$\sum_{u \in G} d(u) = 2 \, \|H\|$$

Příklady: $d(1) = 2$, $d(5) = 1$

5. Orientovaný graf

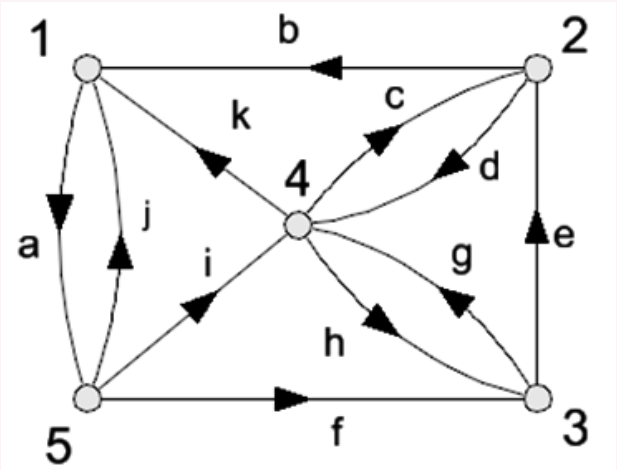
Uspořádaná trojice disjunktních množin $G = \langle H, U, \sigma \rangle$, kde

$$\sigma : H \to U \times U.$$

Pro $h \in H$

$$\sigma(h) = \{u, v\},$$

kde u je počáteční a v koncový uzel.



Stupeň uzlu: suma vstupujících $d^+(u)$ a vystupujících hran $d^-(u)$

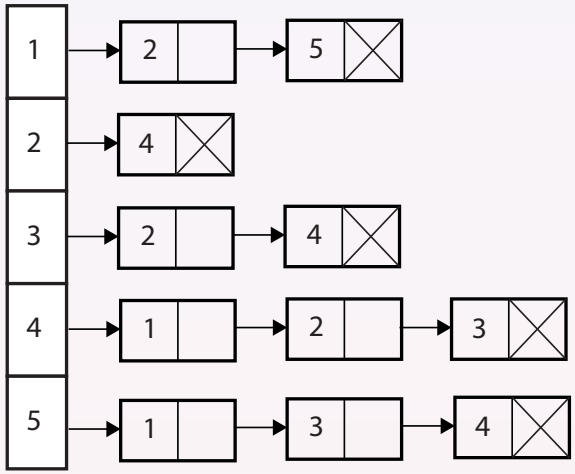
$$d(u) = d^+(u) + d^-(u).$$

6. Reprezentace grafů

Maticová reprezentace:

Výhodná pro výpočty, u řídkých grafů mnoho nulových prvků.

Maticová reprezentace neefektivní, velké paměťové nároky.



Reprezentace spojovým seznamem :

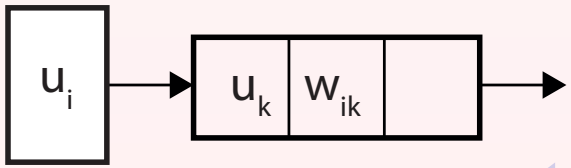
Ukládáme jen uzly, mezi kterými jsou vztahy, menší paměťové nároky.

Méně přehledné pro rozsáhlé grafy.

V praxi použit 2D seznam: 1D seznam, každá položka opět 1D seznam.

Ohodnocené grafy:

Přidáno ohodnocení w_{ik} hrany u_{ik}.



7. Spojová reprezentace v Pythonu

Spojová reprezentace grafu prostřednictvím Dictionary

```
key: value
```

Klíčem uzel, hodnotou vnořený seznam incidujících uzlů.

```
V1 : [V1, V2, ..., Vk]
```

Obecný tvar reprezentace:

```
G={
   V1 : [V1, V2, ..., Vk],
   V2 : [V1, V2, ..., Vk],
   ...
   Vk : [V1, V2, ..., VK]
  }
```

Grafy s ohodnocením:

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

8. Prohledávání grafu

Systematické procházení hran a uzlů grafu dle zadané strategie.

Nejčastěji řešené úlohy:

1. Dostupnost uzlu v z u.
2. Množina všech uzlů dostupných z u.
3. Nalezení cesty z u do v (libovolná, optimální).
4. Nalezení cesty z u do všech dosažitelných vrcholů.

Aplikace v oblasti geoinformatiky, logistiky, dopravy, navigace.

Techniky prohledávání grafu:

- Prohledávání do šířky.
- Prohledávání do hloubky.
- Kombinace + heuristiky.

9. Prohledávání grafu do šířky

BFS (Breadth First Search).

Častá strategie pří procházení grafu.

Vychází z něj: hledání minimální kostry, nejkratší cesty.

Lze použít pro neorientované/orientované grafy.

Idea:

- Postupné rozhlížení z uzlu u do všech jeho doposud neprohledaných sousedů v.
- Získáme uzly v jsou z tohoto uzlu dostupné.
- Následně skok na prvního souseda u, opakování pro dosud neprohledané uzly u.

Výsledek:

Strom hledání do šířky: BF strom.

- Seznam všech uzlů dosažitelných z výchozího uzlu.
- Cesta do těchto uzlů tvořená minimem hran.

BF strom uložen jako P-strom, strom předchůdců.

Velmi efektivní reprezentace.

Složitost $O(\|U\| + \|H\|)$.

10. Rozdělení uzlů

rovádíme značkování uzlů do kategorií.

Aby bylo jasné, které lze ještě použít, a které již nikoliv.

3 kategorie uzlů:

- *Nové uzly (New, White)*
  Dosud neobjevené uzly.
  Uzel, na který narazíme poprvé.
- *Otevřené uzly (Open, Gray)*
  Již objevený uzel.
  Prozkoumáni někteří sousedé.
- *Uzavřené uzly (Closed, Black)*
  Již objevený uzel.
  Prozkoumáni všichni sousedé.

Grafy neobsahující cykly: není třeba značkovat uzly.

11. Princip BFS

Nezohledňuje ohodnocení hran grafu: všechny jednotkové.

Na začátku všechny uzly “N”, následně “O”, potom “C”.

Uzly a atributem “O” uloženy v Q.

Princip BFS:

- Z fronty Q vezmi aktuální uzel u .
- Pro každé v sousedící s u opakuj následující kroky:
  Pokud má v atribut “N”, změň ho na “O”.
  Vytvoř novou hranu $\{u, v\}$ a přidej ji do BF stromu.
- Poté změň stav u na “C”.

Místo přidávání $\{u, v\}$ do BF stromu lze využít *předchůdce uzlu v.*

Předchůdce uzlu.

Uzel u je předchůdcem p uzlu v

$$p(v) = u.$$

Na základě znalosti $p(v)$ lze cestu zpětně zrekonstruovat (netřeba přidávat $\{u, v\}$ do BF).

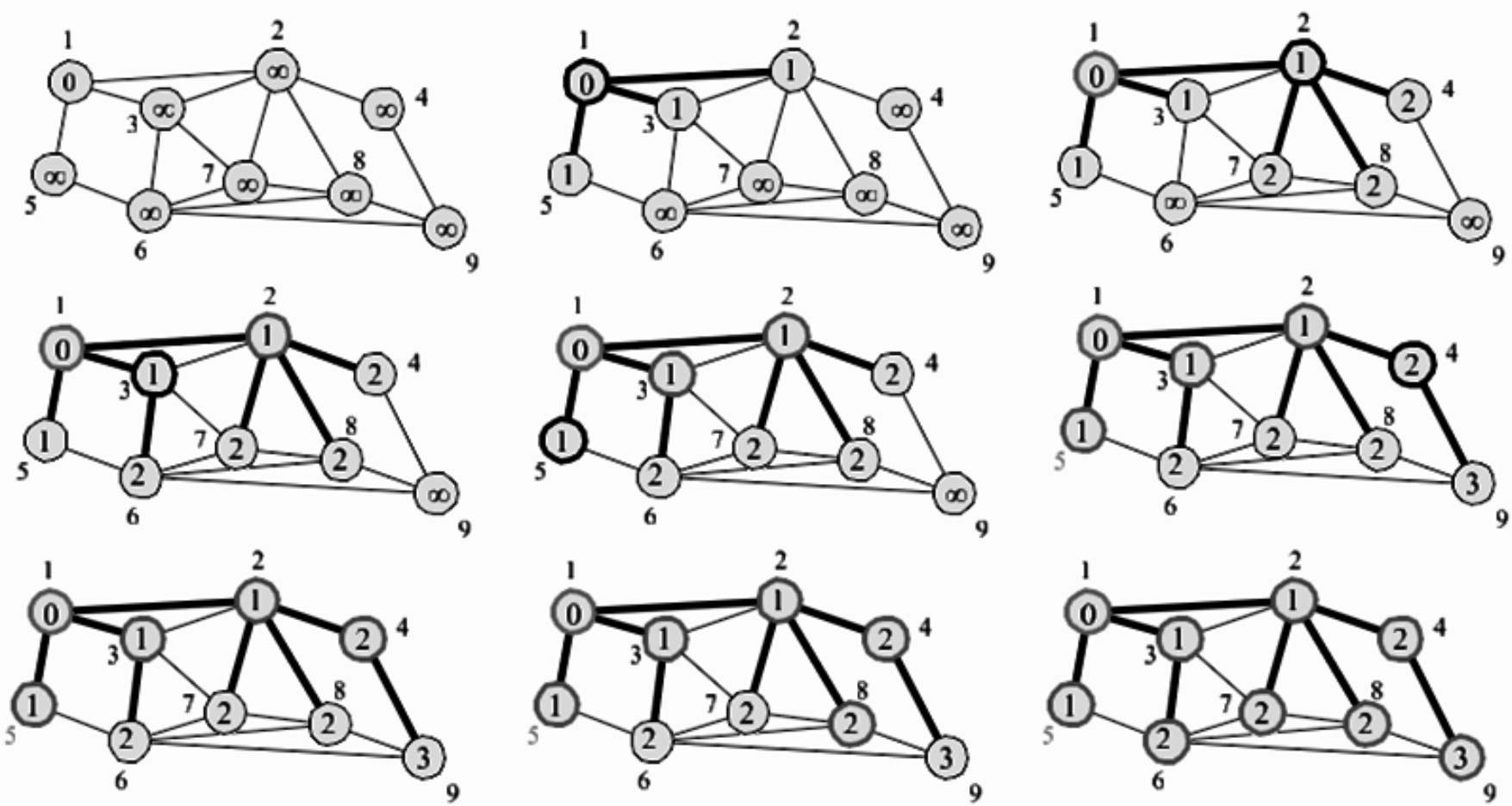
Rekonstrukce cesty od uzlu v ke kořenu u

$$\langle v, p(v), p(p(v)), ..., u \rangle .$$

Pomocné proměnné.

BFS využívá 3 pomocné proměnné:

12. Ukázka BFS



13. Implementace BFS

```
def BFS(G, u):
    s = ['N'] * (len(G)+1)       #All vertices are new
    p = [None] * (len(G)+1)      #Without predecessors
    Q = []                       #Empty queue
    Q.append(u)                  #Add start vertex to Q
    s[u] = 'O'                   #Set as open
    while Q:
        u = Q.pop(0)             #Pop first node
        for v in G[u]:           #Browse incident nodes
            if s[v] == 'N':      #We found a new node
                s[v] = 'O'       #Change status to Open
                p[v] = u         #Set predecessors
                Q.append(v)      #Add v to Q
        s[u] = 'C'               #Change status
```

Využívá frontu.

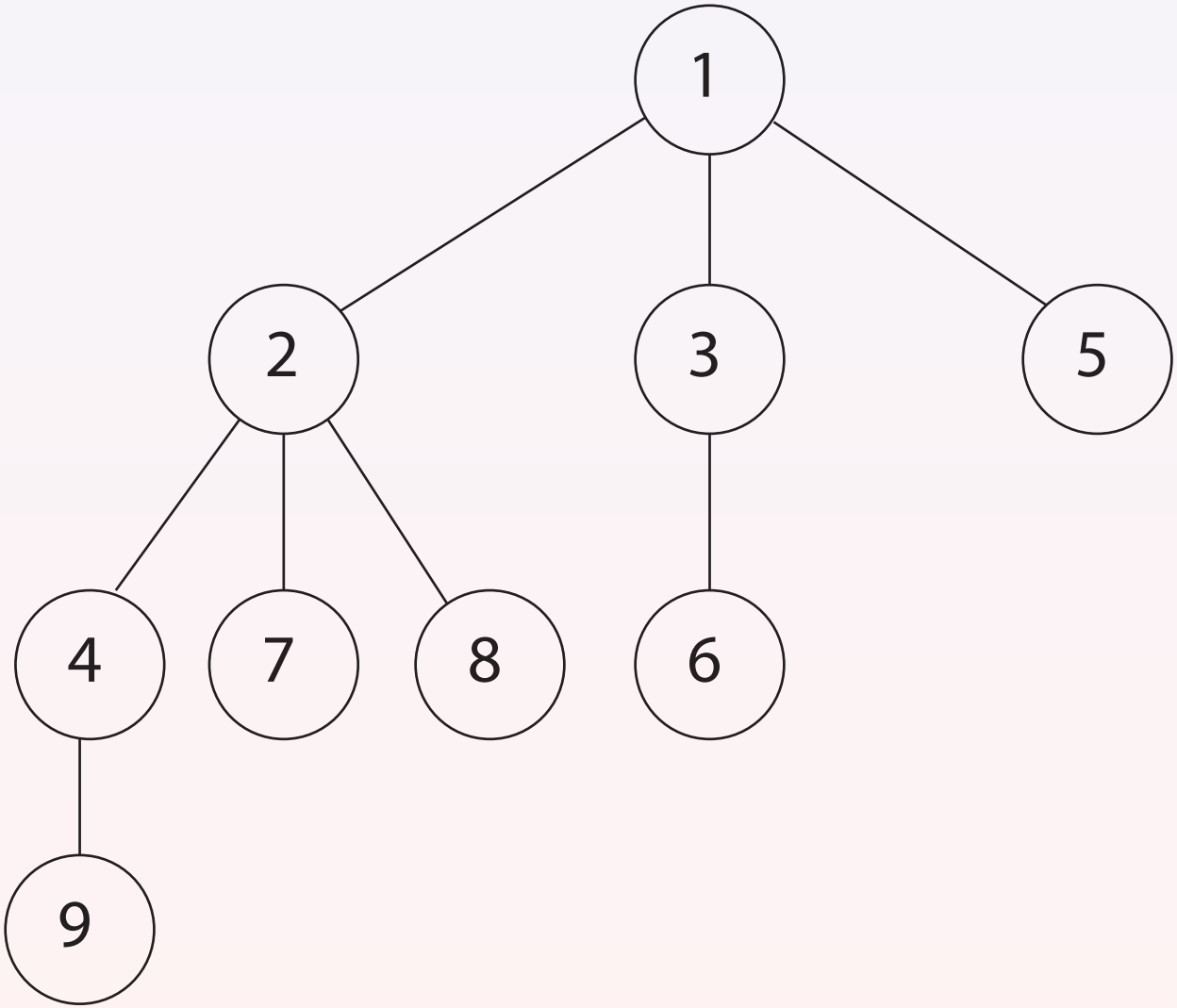
14. Ukázka činnosti algoritmu, $u = 1$

Aktualizace předchůdců a stav Q.

#	u_1	u_2	u_3	u_4	u_5	u_6	u_7	u_8	u_9	Q	$Q \leftarrow$
1	-1	-1	-1	-1	-1	-1	-1	-1	-1	1,	2,3,5
2	-1	1	-1	-1	-1	-1	-1	-1	-1	2, 3, 5	4,7,8
3	-1	1	1	-1	-1	-1	-1	-1	-1	3, 5, 4, 7, 8	6
4	-1	1	1	-1	1	-1	-1	-1	-1	5, 4, 7, 8, 6	
5	-1	1	1	2	1	-1	-1	-1	-1	4, 7, 8, 6	9
6	-1	1	1	2	1	-1	2	-1	-1	7, 8, 6, 9	
7	-1	1	1	2	1	-1	2	2	-1	8, 6, 9	
8	-1	1	1	2	1	3	2	2	-1	6, 9	
9	-1	1	1	2	1	3	2	2	4	9	

Obsah Q: procházení BF stromu po úrovních.

15. Výsledný BF strom



16. Graf a jeho popis

Popis grafu:

```
G = {
    1 : [2, 3, 5],
    2 : [1, 3, 4, 7, 8],
    3 : [1, 2, 6, 7],
    4 : [2, 9],
    5 : [1, 6],
    6 : [3, 5, 7, 8, 9],
    7 : [2, 3, 6, 8],
    8 : [2, 6, 7, 9],
    9 : [4, 6, 8]
}
```

Výsledky:

```
print(p)
print(d)
print(s)
>>[None, 1, 1, 2, 1, 3, 2, 2, 4]
>>[0, 1, 1, 2, 1, 2, 2, 2, 3]
>>['C', 'C', 'C', 'C', 'C', 'C', 'C', 'C', 'C']
```

17. Zpětná rekonstrukce cesty

Cesta mezi uzly $\langle u, v \rangle$.

Postup od koncového uzlu v blížíme k počátečnímu uzlu u.

Využití předchůdce $p[v]$.

Opakujeme, dokud $v \neq u$.

```
def createPath(u, v, p):
    path = []                       //Create empty path
    while v != u and v != -1:       //Until we reach a start node
        path.append(v)              //Add current node
        v = p[v]                    //Go to the predeccessor
    path.append(v)                  //Add last vertex
    return path
```

Volání:

```
createPath(1, 9, p)
>> 1 2 4 9
```

Existuje i rekurzivní implementace.

Založena na opakovaném zkracování cesty.

18. Prohledávání grafu do hloubky

DFS (Depth First Search)

Prohledávání grafu s cílem dostat se do grafu co nejhlouběji.

Postupné procházení všech možných cest grafem.

Analogie bludiště: vracíme se zpět, když cesta nepokračuje.

Strategie známa jako *Backtracking*.

Jedna ze základních metod procházení stavového prostoru.

Idea:

- Z uzlu u jdeme do prvního dosud neprohledaného souseda v.
- Pokud takový uzel v neexistuje, návrat do uzlu, ze kterého jsme vstoupili do u, tj. do $p[u]$.
- Opakováno, dokud neobjeveny všechny dosažitelné uzly z výchozího uzlu.

Výsledek:

Strom hledání do šířky: DF strom.

- Seznam všech uzlů dosažitelných z výchozího.
- Cesta do těchto uzlů není nejkratší.

DF strom uložen jako P-strom, strom předchůdců.

Složitost $O(\|U\| + \|H\|)$.

19. Princip DFS

Uchovávané informace o uzlu: stav uzlu $s[u]$, předchůdce $p[u]$ uzlu u.

Pro nový uzel u projdi všechny jeho sousedy v.

- nastavení předchůdce $p[v] = u$,
- rekurzivně projdi všechny sousedy uzlu v.

Implementace: Iterativní vs. rekurze.

Iterativní implementace:

Zásobník, vrcholy procházeny v opačném směru.

Upravené otevírání uzlu: při 2 průchodu.

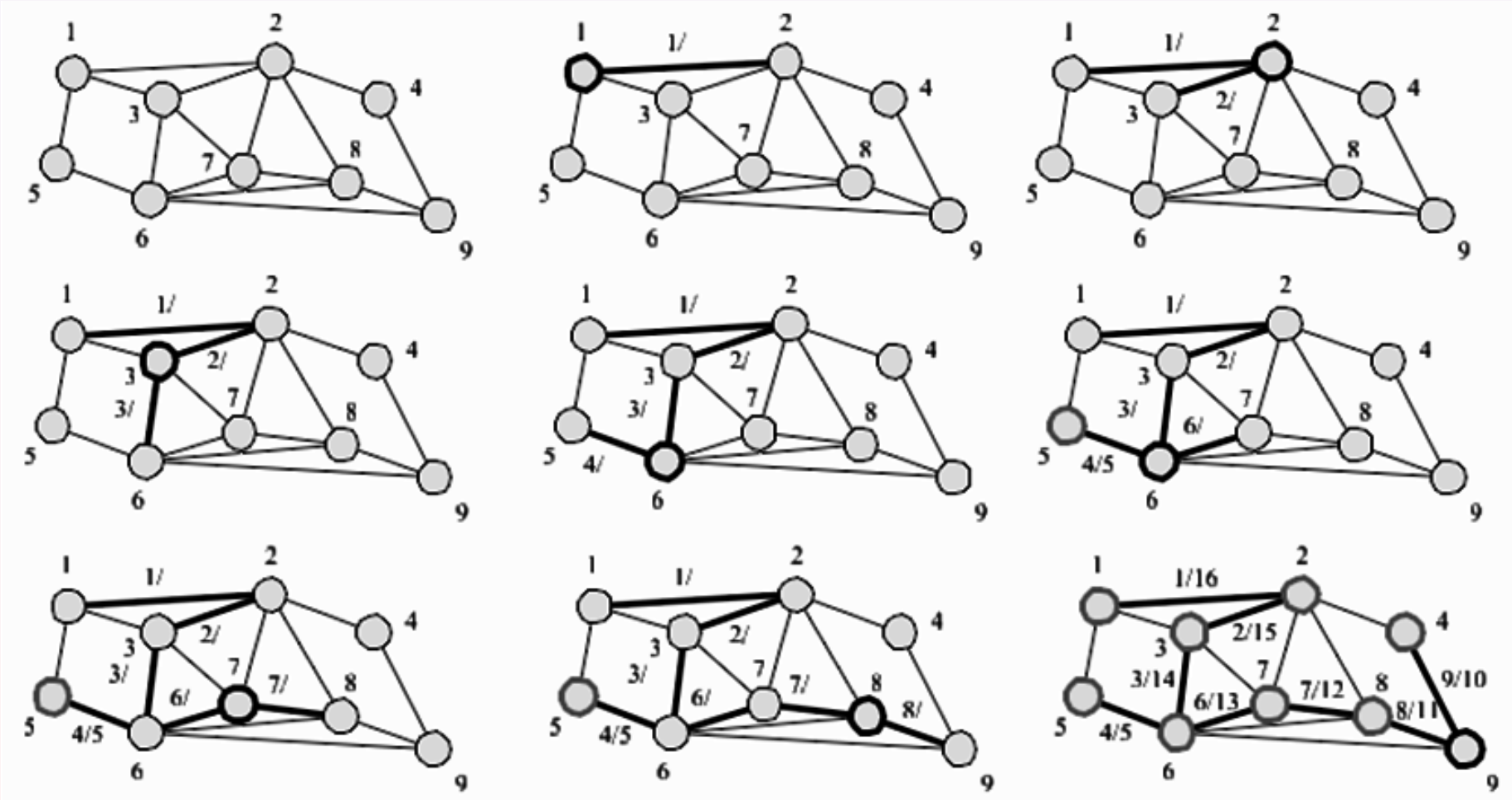
Rekurzivní implementace:

Procedura DFSR spuštěna 1x nad každým novým uzlem.

Pokud G souvislý, stačí 1x nad libovolným uzlem.

Výsledkem DF les: tvořen DF stromy (>, pokud G nesouvislé).

20. Ukázka DFS



21. Iterativní implementace DFS

```
def DFS(G, u):
    s = ['N'] * (len(G)+1)          #All vertices are new
    p = [None] * (len(G)+1)         #Without predecessors
    S = []                          #Empty stack
    S.append(u)                     #Add start vertex to S
    while S:
        u = S.pop()                 #Pop node
        s[u] = 'O'                  #Set node as open
        for v in reversed(G[u]):    #Browse incident nodes
            if s[v] == 'N':         #We found a new node
                p[v] = u            #Set predecessor
                S.append(v)         #Add v to Q
        s[u] = 'C'                  #Change status
```

Využívá zásobník.

22. Rekurzivní implementace DFS

Souvislý graf, 1x z libovolného uzlu:

```
def DFS(G, u):
    s = ['N'] * (len(G)+1)          #All vertices are new
    p = [None] * (len(G)+1)         #Without predecessors
    DFSR(G, s, p, u)                #Run DFS for connected graph
```

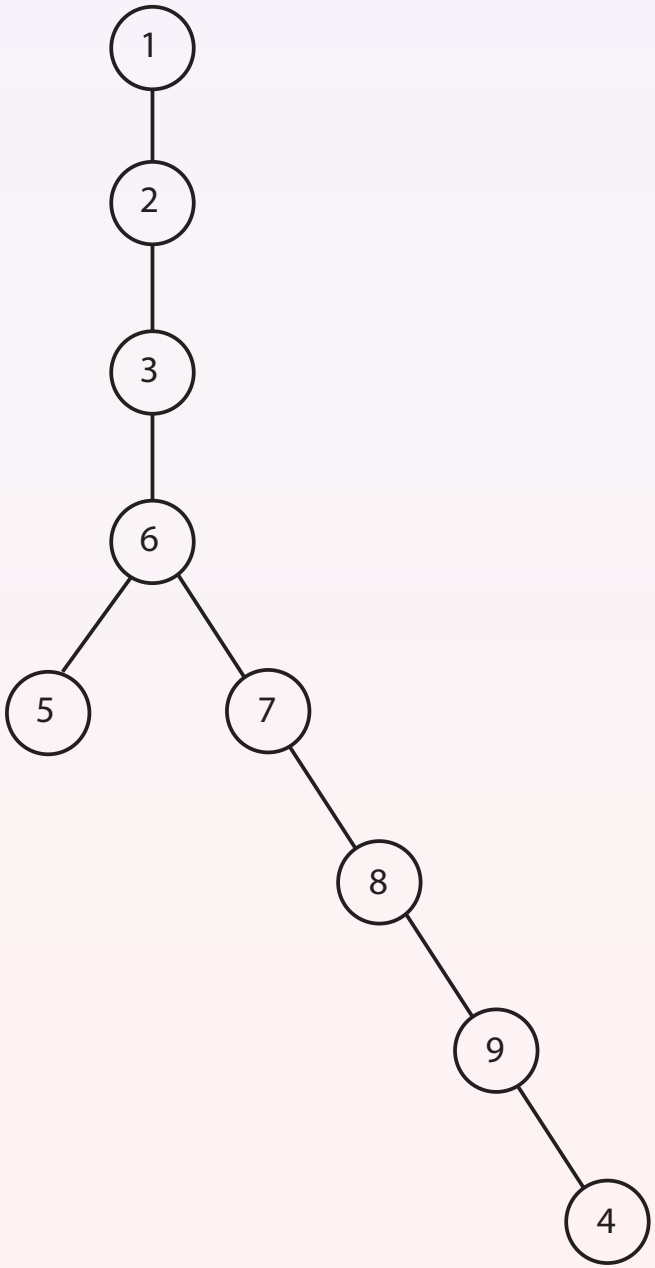
Nesouvislý graf:

```
def DFS(G, u):
    s = ['N'] * (len(G)+1)          #All vertices are new
    p = [None] * (len(G)+1)         #Without predecessors
    for u, value in G.items():      #Process all new nodes
        if s[u] == 'N':
           DFSR(G, s, p, u)         #Run DFS for disconnected graph
```

Rekurzivní procedura:

```
def DFSR(G, s, p, u):
    s[u] = 'O'                      #Set node as open
    for v in G[u]:                  #Browse all edges from u
        if s[v] == 'N':             #For each new node
            p[v] = u                #Set predeccessor
            DFSR (G, s, p, v)       #Browse its neighbors
    s[u] = 'C'                      #Node u is closed
```

23. Výsledný DF strom



24. Použití BFS/DFS

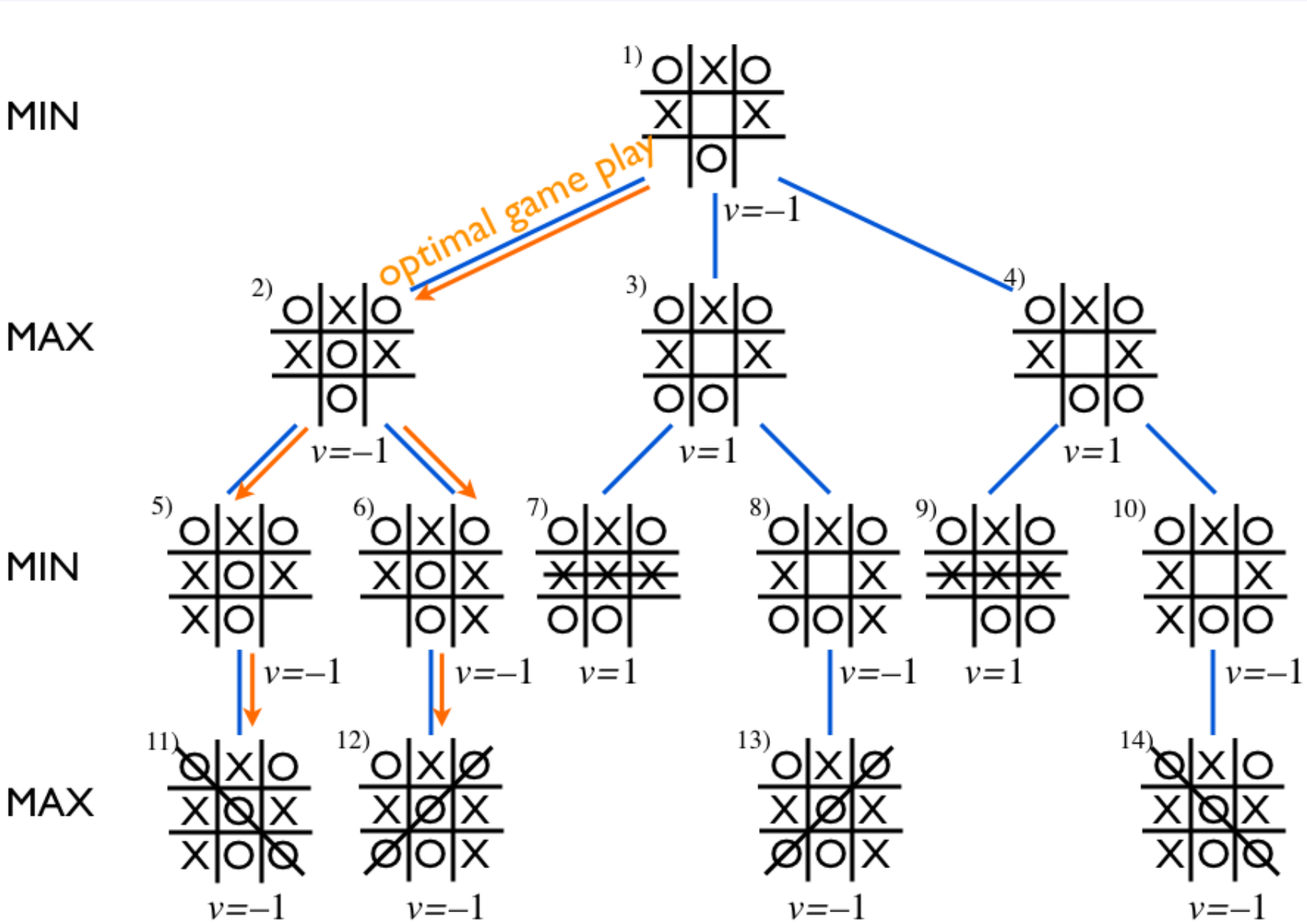
Jedny z nejčastěji používaných algoritmů.

Častá aplikace i pro problémy s grafy zdánlivě nesouvisejícími.

Nejčastější použití:

- *Optimalizační strategie:*
  Heuristika, TSP (přibližné řešení), turnajová schémata, procházení složek/podsložek.
- *Stolní hry:*
  Šachy, dáma, GO, piškvorky + doplněno efektivnějšími strategiemi.
- *Hlavolamy:*
  Magické čtverce, SUDOKU, křížovky, Rubikova kostka, bludiště.
- *Dopravní a logistické problémy*
  Existence cesty, optimální cesty, vzdálenosti centra/pobočky, údržba silnic, elektrifikace.

25. Piškvorky, stavový strom



26. 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 hran $w > 0$ (obecně $w \in \mathbb{R}$).

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

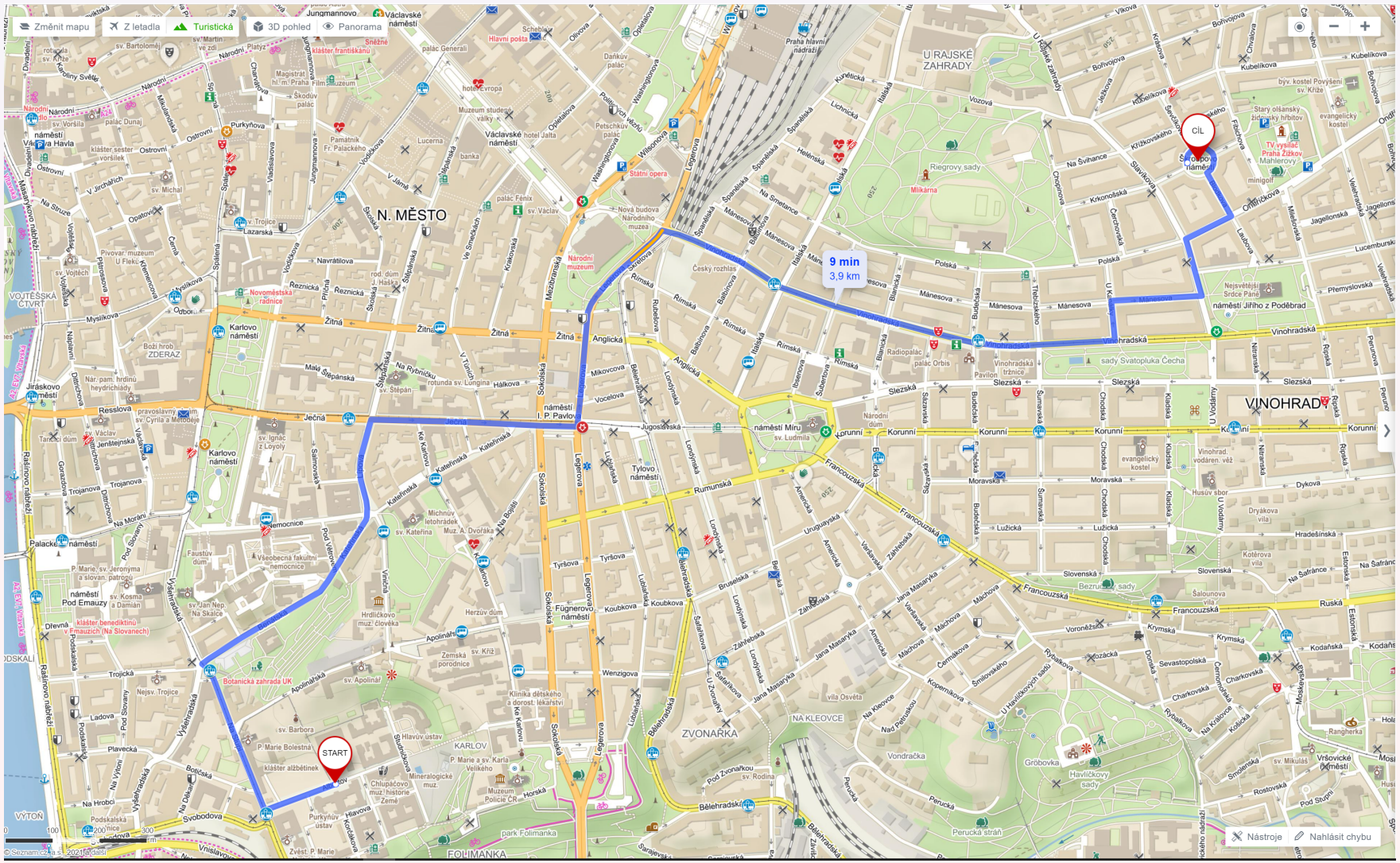
Lze hledat nejkratší, nejlevnější či jinou cestu.

Přehled algoritmů:

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

Použití: navigační SW, logistika, doprava.

27. Ukázka nejkratší cesty mezi 2 uzly



28. Datový model grafu s ohodnocením

Každá hrana $h = (u, v)$ má ohodnocení $w(u, v) \in \mathbb{R}^+$ (nezáporné ohodocení).

Spojová reprezentace, v Pythonu použit Dictionary

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

Klíč K : uzel.

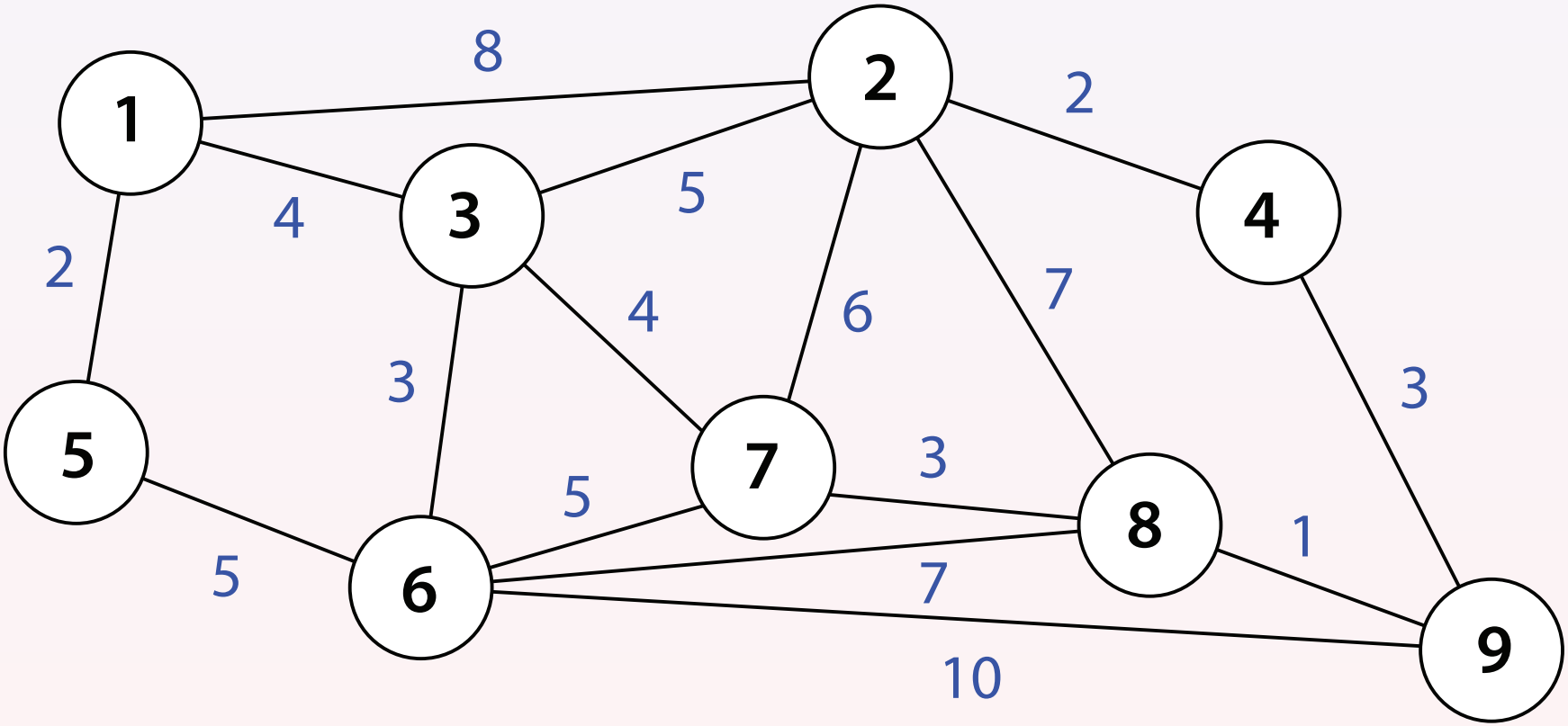
Hodnota V: seznam incidujících vrcholů s ohodnocením hran.

```
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}
```

29. Ukázka grafu s ohodnocením



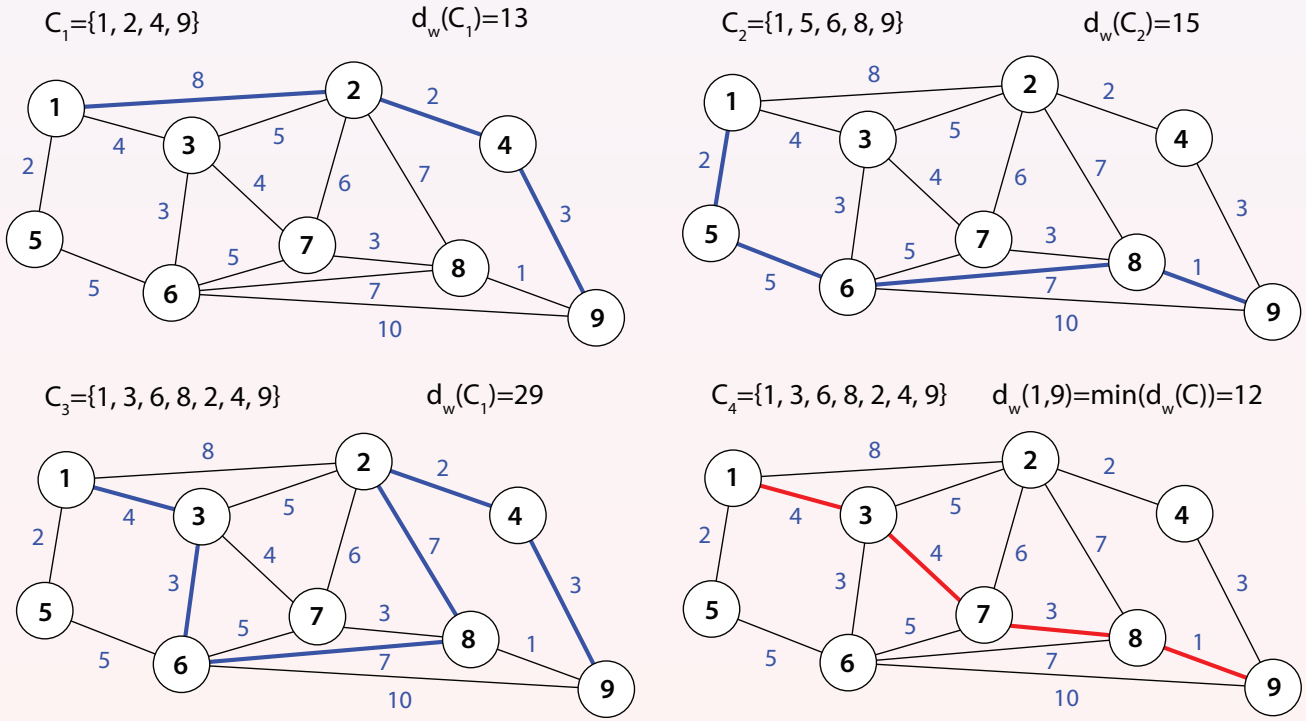
30. *W*-délka a *w*-vzdálenost

W-délka d_w cesty $C = \langle h_1, \ldots, h_k \rangle$ s k hranami h_i

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

W-vzdálenost $d_w(u, v)$ uzlů u, v, nejmenší w-délka cesty z u do v

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



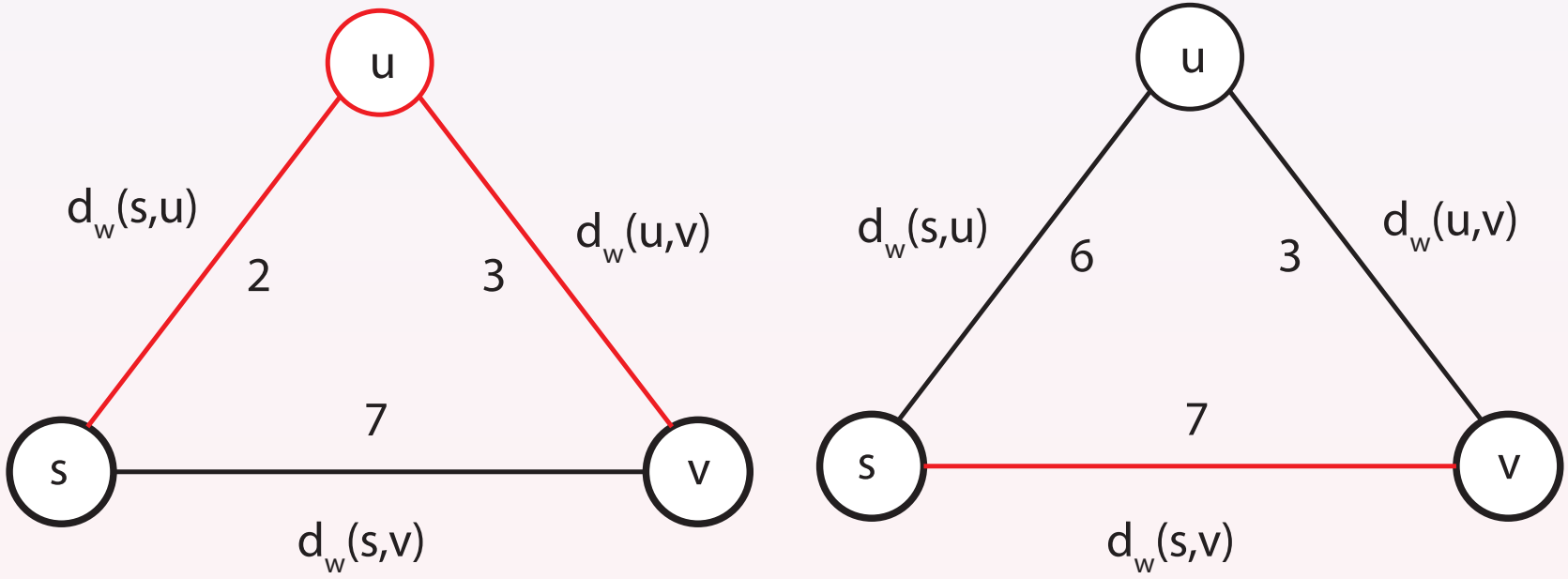
Mezi uzly 1, 9 mnoho cest C s různými $w-$délkami (modré).

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

31. Základní věty

Trojúhelníková nerovnost

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



Věta o nejkratší cestě:

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

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

32. Relaxace hrany

Pracuje s hodnotami $d[u]$, $d[v]$: horní odhady $d_w(s,u)$, $d_w(s,v)$.

Trojúhelníková nerovnost platí pro w-vzdálenosti, ale i pro horní odhady

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

Relaxace probíhá v *opačném směru*.

Princip relaxace:

Opakovanou aktualizací odhadu $d[v]$ hledáme $d_w(s,v)$.

Odhad $d[v]$ se průběžně snižuje.

Pokud

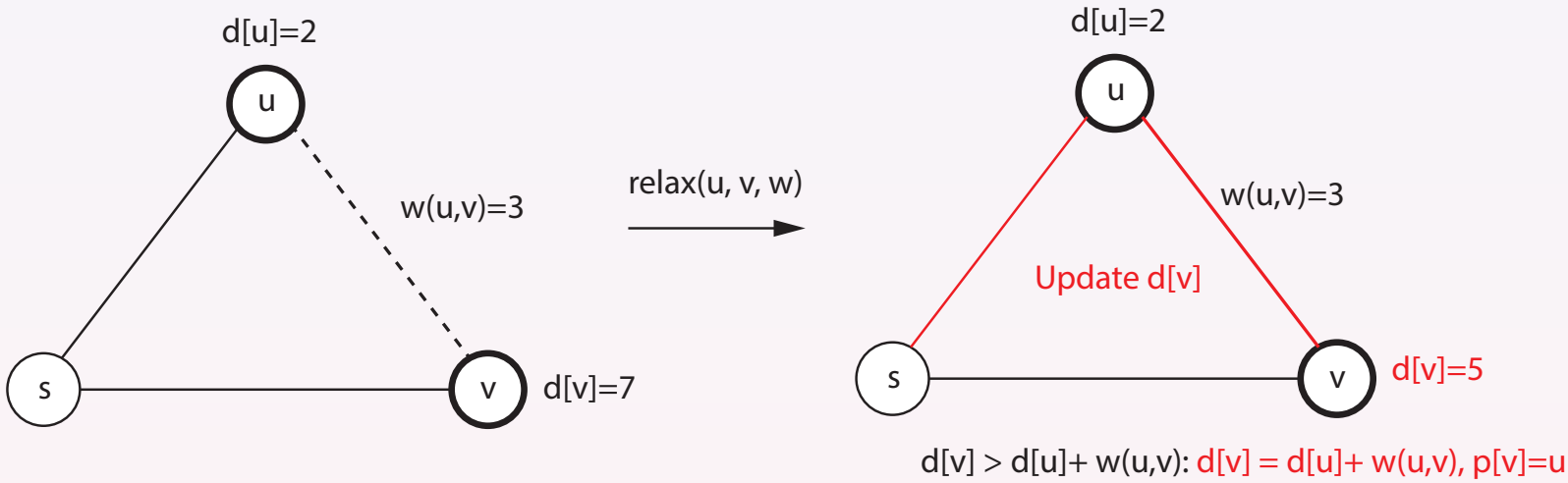
$$d[v] > d[u] + w(u,w),$$

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

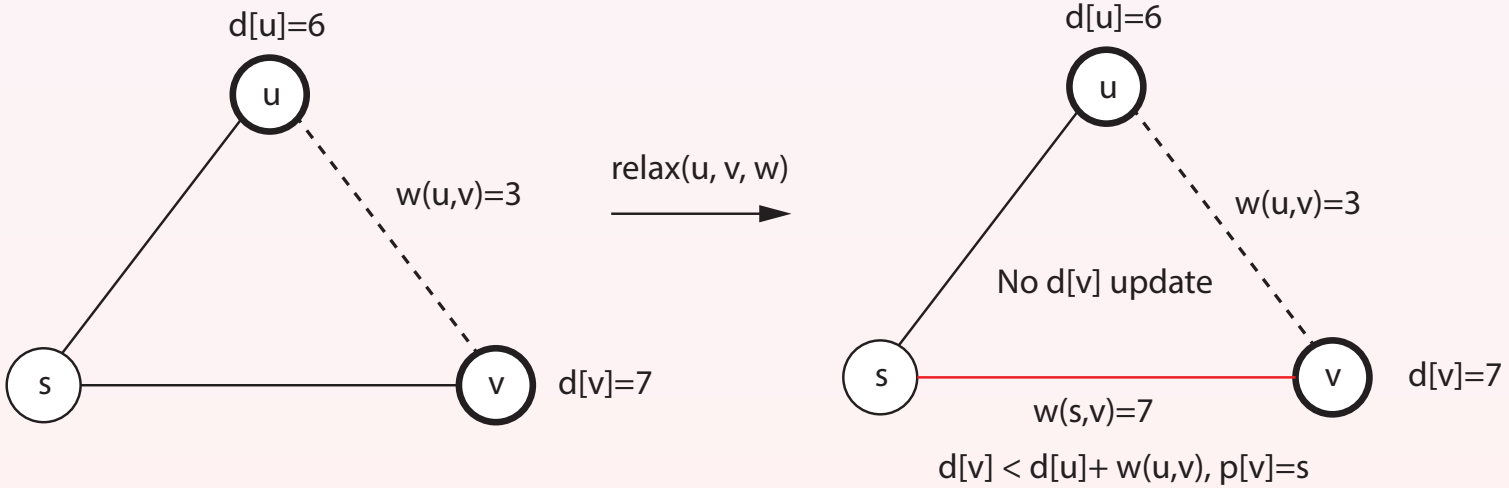
```
def relax(u, v, w):
   if d[v] > d[u]+w(u,v):        #Is path (s, u, v) shorter than (s, v)?
      d[v] = d[u]+w(u,v)         #Update d[v]
      p[v] = u                   #Update predecessor
```

33. Ukázka relaxace hrany (u, v)

Výhodnější cesta $s \to u \to v$ nebo $s \to v$?



d[v] > d[u]+ w(u,v): d[v] = d[u]+ w(u,v), p[v]=u



d[v] < d[u]+ w(u,v), p[v]=s

34. 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.

35. 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

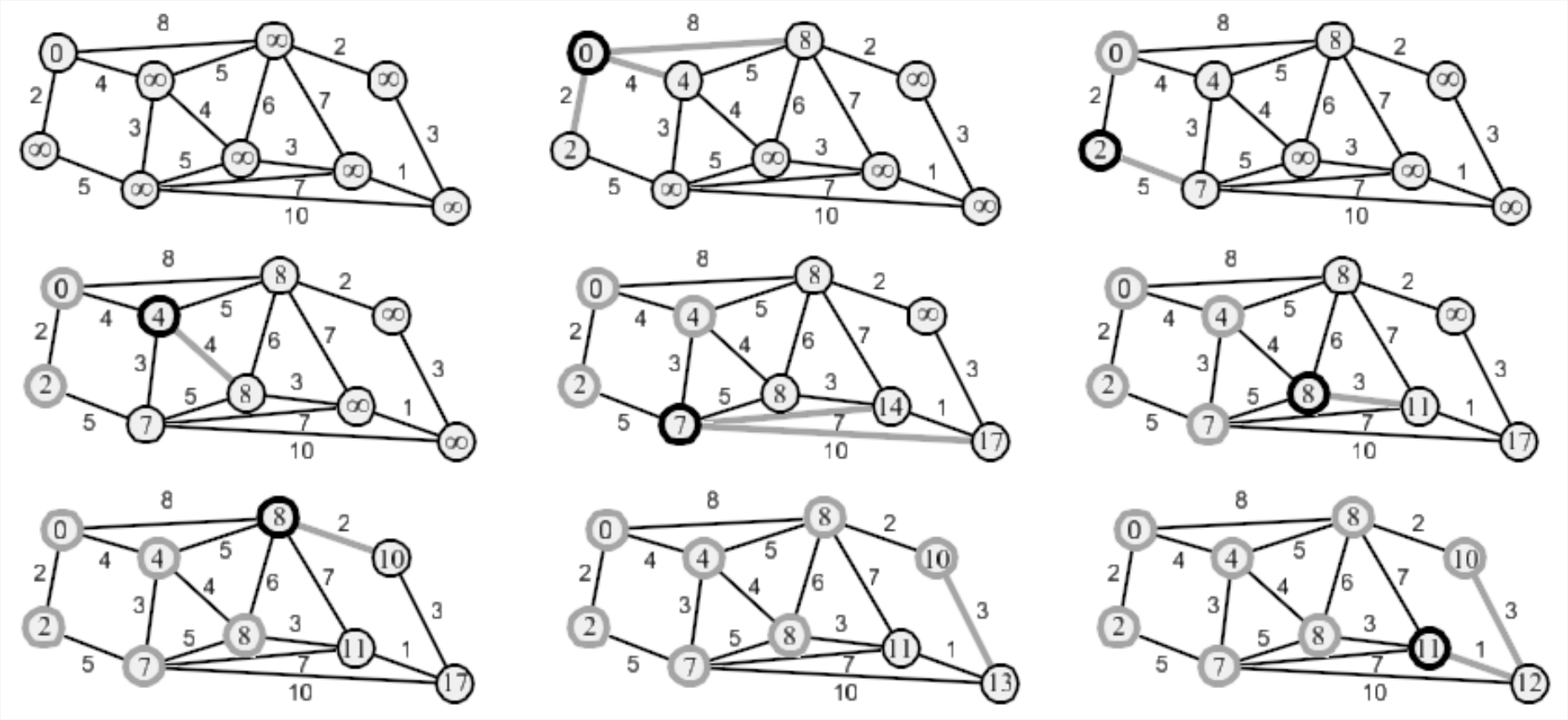
$$d[v] = d[u] + w[u][v],$$

je novým odhad $d[v]$ a

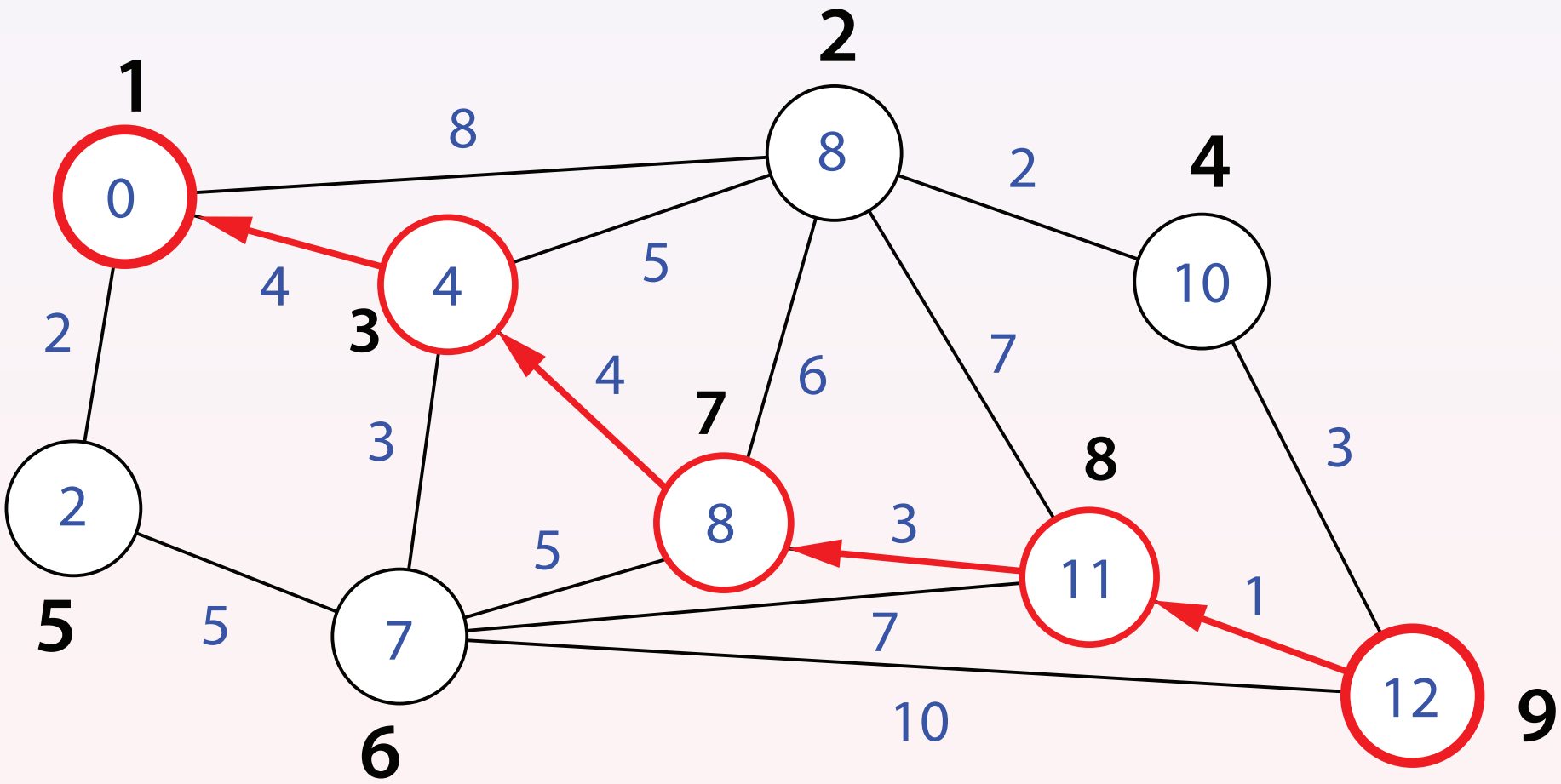
$$p[v] = u.$$

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

36. Ukázka Dijkstra algoritmu



37. Výsledek Dikjstra algoritmu



38. 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 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.

39. 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
```