Grafové algoritmy II.

Nejkratší cesty grafem. Dijkstra. Bellman-Ford. Floyd-Warshall.

Tomáš Bayer | bayertom@natur.cuni.cz

Katedra aplikované geoinformatiky a kartografie, Přírodovědecká fakulta UK.

Obsah přednášky

1. Nejkratší cesta grafem

Často řešená úloha v geoinformatice, logistice, dopravě.

Optimalizace průchodu grafem tak, aby délka cesty byla minimální.

Výsledná cesta je orientovaná.

Varianty problému:

1. nejkratší cesta ze zadaného uzlu do cílového uzlu,
2. nejkratší cesta ze zadaného uzlu do každého uzlu,
3. nejkratší cesty ze všech uzlů do zadaného uzlu,
4. nejkratší cesty mezi všemi dvojicemi uzlů.

Variantu ad 1) lze poměrně jednoduše převést na ad 2) nebo ad 4).

Variantu ad 3) lze změnou orientace hran převést na ad 2).

Lze aplikovat na orientované/neorientované grafy.

Předpoklád: hrany grafů mají kladné ohodnocení, $w > 0$.

2. Nejkratší cesta z 1 uzlu

Předpoklady pro G: orientovaný/neorientovaný, souvislý, konečný.

Popis G: spojový seznam.

Většina metod vychází z BFS: prohledávání do šířky.

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

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

Přehled algoritmů:

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

Často řešený problém v geoinformatice.

Použití: navigační SW, logistika, doprava.

3. *W*-délka a *w*-vzdálenost

Cesta $P = \langle h_1, ..., h_k \rangle$ v graf $G = \langle H, U, \rho \rangle$ tvořena k hranami h s w-délkou d_w

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

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

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

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

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

$$\begin{aligned}
&(1)\ d_w(u, v) &&\geq && 0, \text{(“nezápornost” vzdálenosti)},\\
&(2)\ d_w(u, v) &&= && 0 \Leftrightarrow u = v, \text{(identita)},\\
&(3)\ d_w(u, v) &&= && d_w(v, u), \text{(symetrie, pro neorientované G)},\\
&(4)\ d_w(u, v) &&\leq && d_w(u, x) + d_w(x, v), \text{(trojúhelníková nerovnost)}.
\end{aligned}$$

Věta o nejkratší cestě:

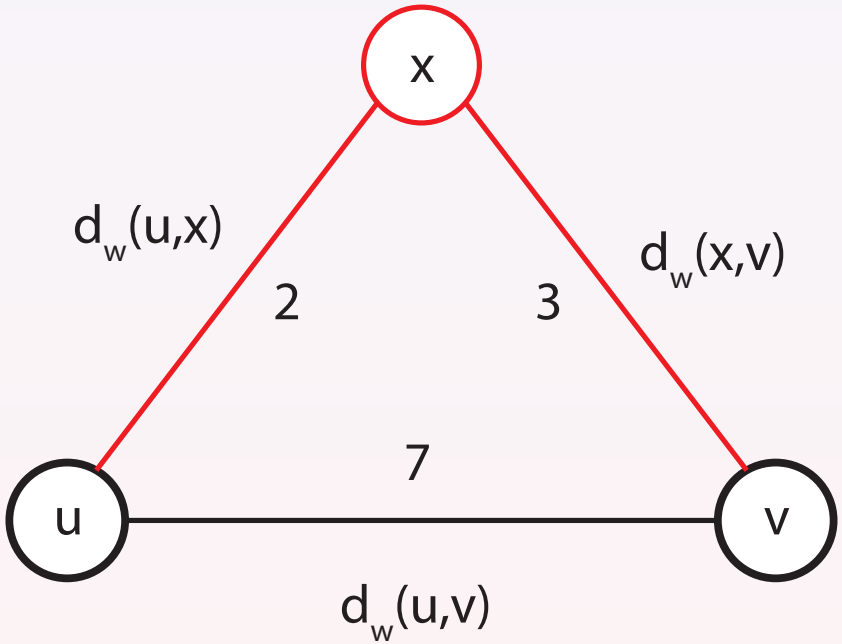
Pokud $d_w(u, v)$ nejkratší cestou z u do v přes x, pak $d_w(u, x)$ nejkratší cestou z u do x a $d_w(x, v)$ nejkratší cestou z x do v.

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

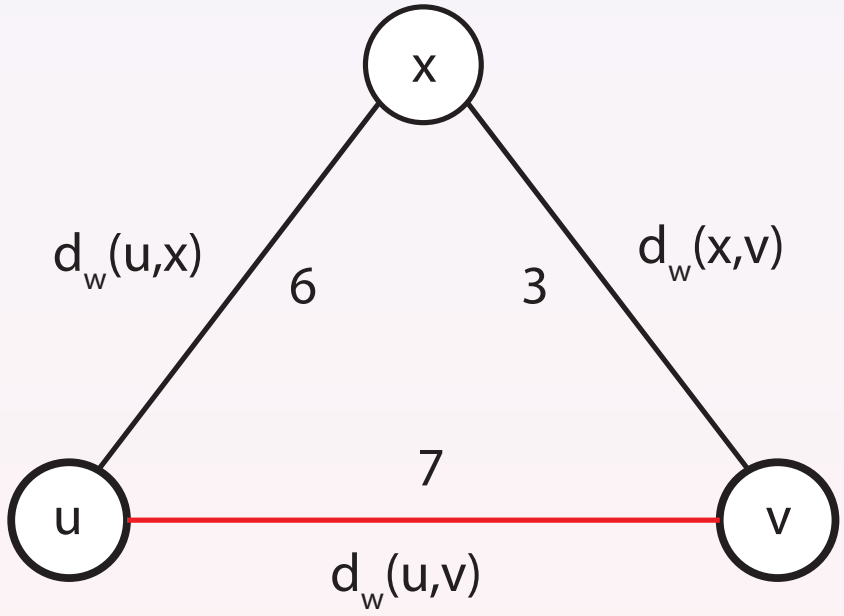
Libovolná část nejkratší cesty je též nejkratší, důsledek TN.

4.Ukázka trojúhelníkové nerovnosti

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



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



Vlevo, platí věta o nejkratší cestě.

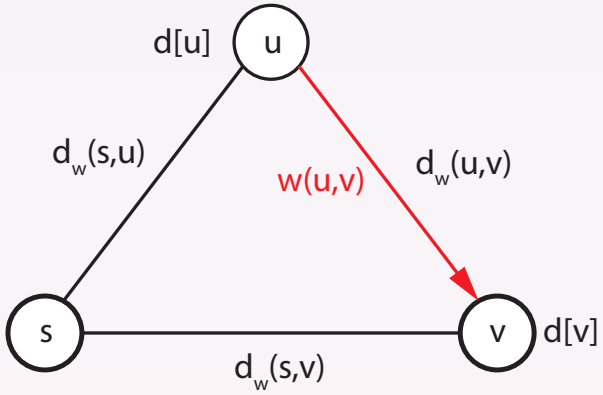
Spojení obou případů

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

5. Relaxace hrany

Nalezení nejkratší spojnice z uzlu s do v + aktualizace předchůdce v.

2 varianty: $s \to v$ nebo $s \to u \to v$.



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

Pro uzly u, v, s platí automaticky trojúhelníková nerovnost, avšak ověřujeme

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

Pokud

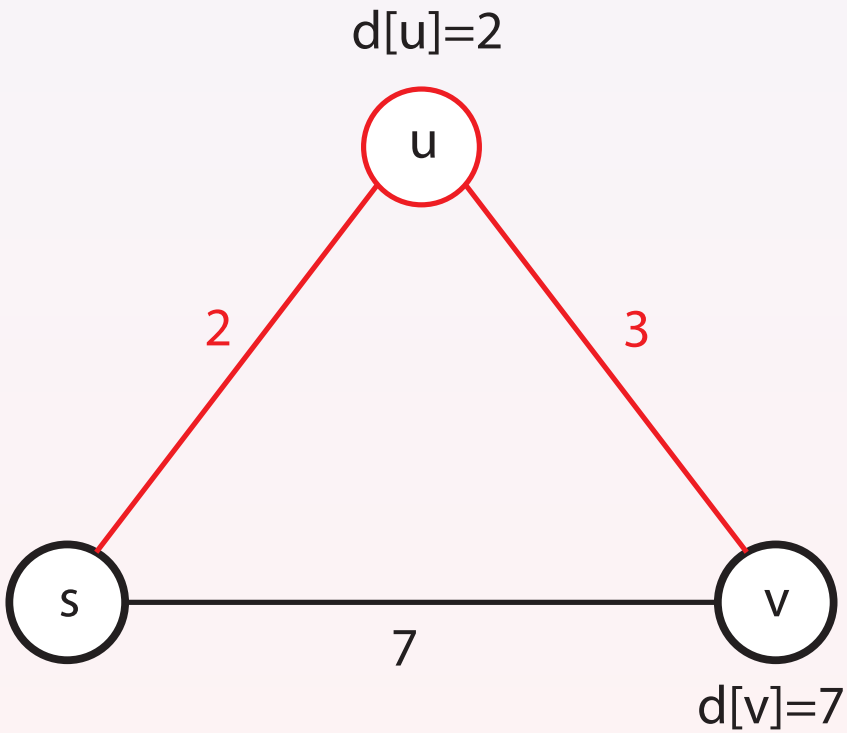
$$d[v] > d[u] + w(u,w),$$

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

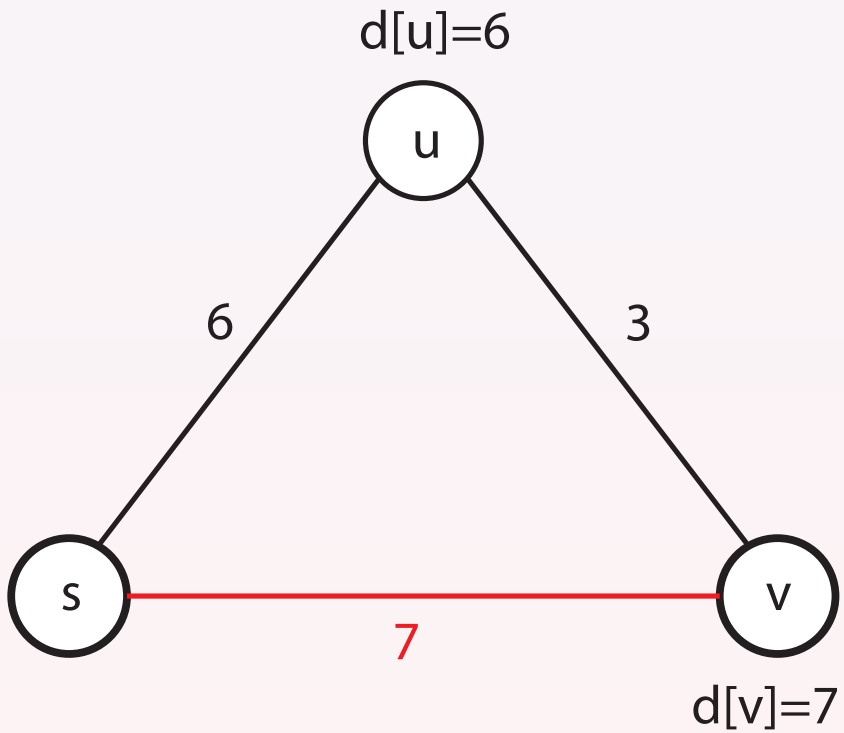
```
def relax(s,u,v):
    if d[v] > d[u]+w(u,v):
        d[v] = d[u]+w(u,v)
        p[v] = u
```

6. Ukázka relaxace hrany

Výhodnější cesta $s \to u \to v$ (vlevo) nebo $s \to v$ (vpravo).



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



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

7. Dijkstrův algoritmus

Nejznámější algoritmus pro hledání nejkratších cest mezi u, v.

Zobecňuje strategii BFS.

Předpoklady:

- nezáporné ohodnocení w,
- G je souvislý.

Provádí opakovanou relaxaci hrany, co nejmenší prodloužení cesty.

Využívá *Greedy strategii*:

Heuristická optimalizace, zde úspěšná (obecně nemusí být)

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

Jednoduchá implementace.

Složitost $O(\|U\|^2)$.

8. Princip Dijkstrova 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.

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

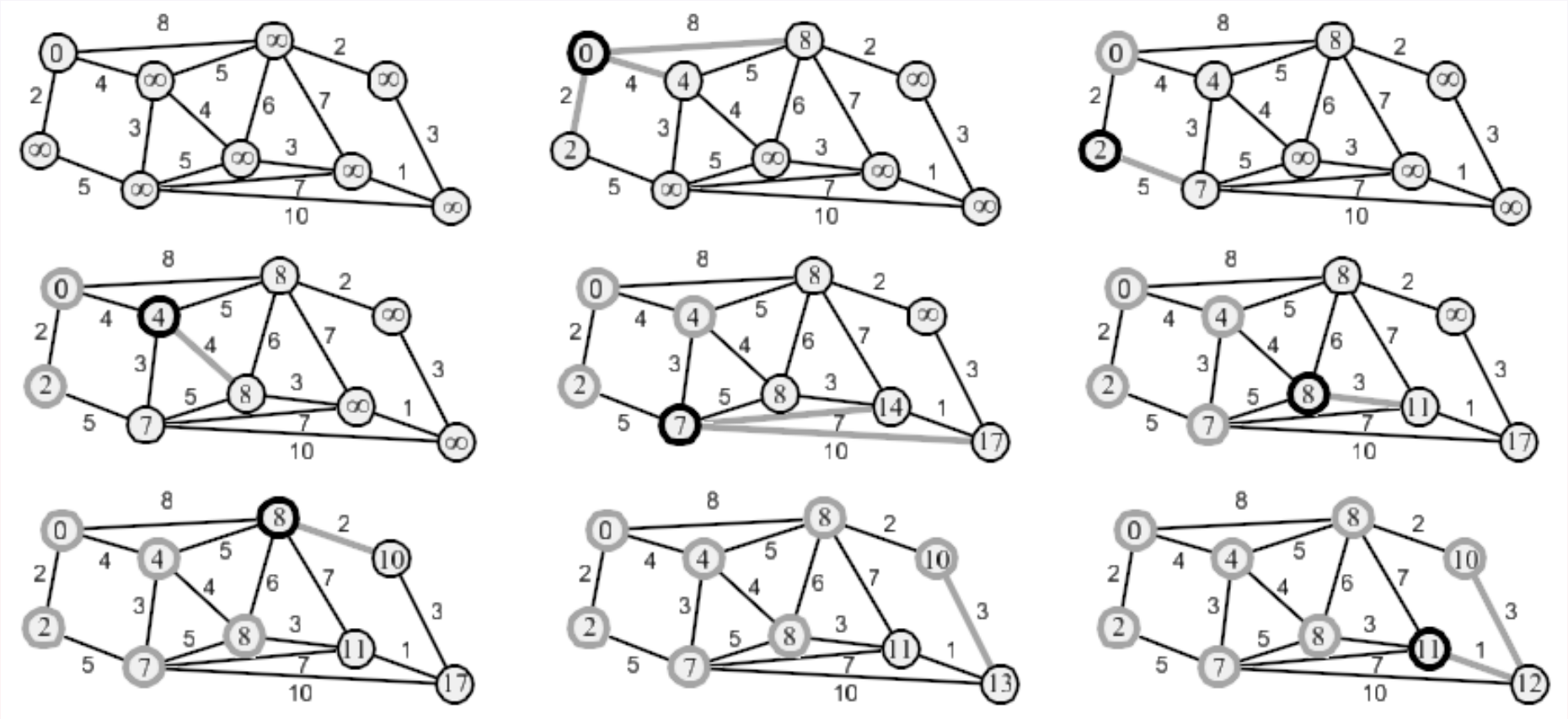
$$d[v] = d[u] + w[u][v],$$

je novým odhad $d[v]$ a

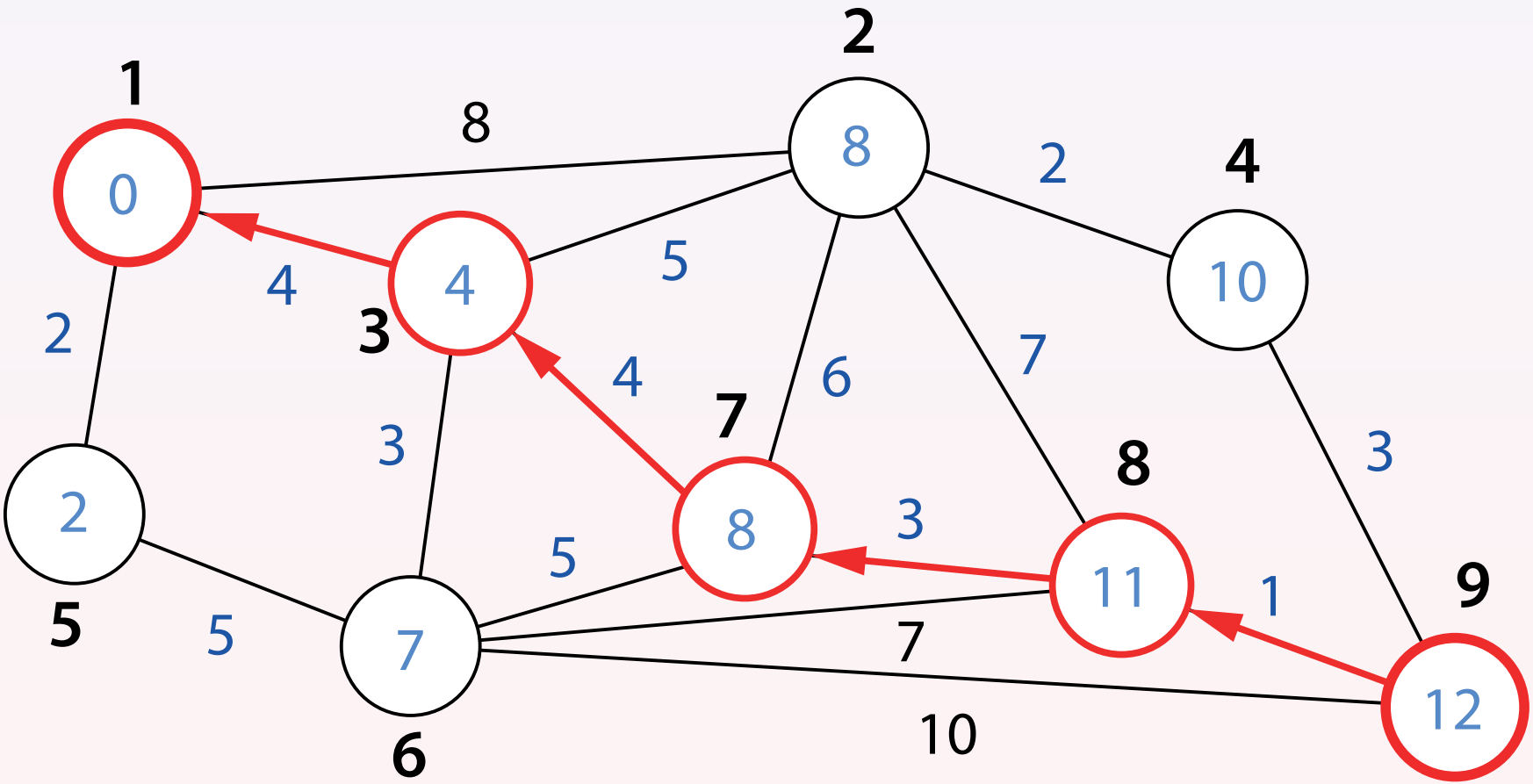
$$p[v] = u.$$

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

9. Ukázka Dikjstrova algoritmu (1/2)



10. Ukázka Dikjstrova algoritmu (2/2)



11. Popis Dijkstrova algoritmu

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

Cesta rekonstruována ze seznamu předchůdců zpětně.

Uloženy v prioritní frontě Q

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

Startovní uzel s.

1. *Inicializační fáze:*
   Inicializace vstupních hodnot: $d(u) = \infty$, $p(u) = -1$,.
   Nastavení $d[start] = 0$. Přidání $< d[start], start >$ 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, nahradíme ho aktualizovanou hodnotou.
     - Aktualizujeme předchůdce: $p[v] = u$.
     - Přidáme $< d[v], v >$ do Q.

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

Snadná implementace.

12. Datový model grafu s ohodnocením

Spojová reprezentace.

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}
}
```

13. Implementace Dijkstrova 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, dv in G[u].items():    #Relaxation, all (u,v)
            if d[v] > d[u] + dv:      #We found a better way
                d[v] = d[u] + dv      #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
```

14. Bellman-Fordův algoritmus

Vhodný pro grafy se záporným ohodnocením hran.

Dijkstra zde nefunguje, u může být otvírán opakovaně, $d[u] \to -\infty$.

B-F relaxuje všechny hrany opakovaně v abecedním pořadí.

D realaxuje pouze hrany vycházející z uzlu u.

Výhodou jednoduchá implementace.

Uzly není nutno značkovat.

Vzniká snadnou modifikací D: místo prioritní fronty použita fronta.

V takovém případě otevíráme nejstarší uzel, ne nejbližší.

B-F se zastaví, pokud neexistuje h,kde $w(h) < 0$.

Generuje nejkratší cestu tvořenou nejméně hranami.

Složitost $O(\|U\| \cdot \|H\|)$.

15. Princip B-F algoritmu

Inicializace $d[u] = \infty$ a $p[v] = -1$ pro všechny uzly.

Z každého uzlu relaxovány všechny hrany v abecedním pořadí.

Výběr uzlu na rozdíl od D. není vázán na $d[u]$.

Relaxací aktualizována hodnoty $d[v]$.

Algoritmus 2 fázový:

1. *Relaxace všech hran G.*
   Provedeno $\|U - 1\|$ x.
2. *Hledání hrany snižující $d[V]$*
   Opakovaná relaxace každé hrany.
   Pokud alespoň 1x sníženo $d[v]$, existuje $w(h) < 0$.
   Jinak $w(h) < 0$ neexistuje.

Klasická implementace:

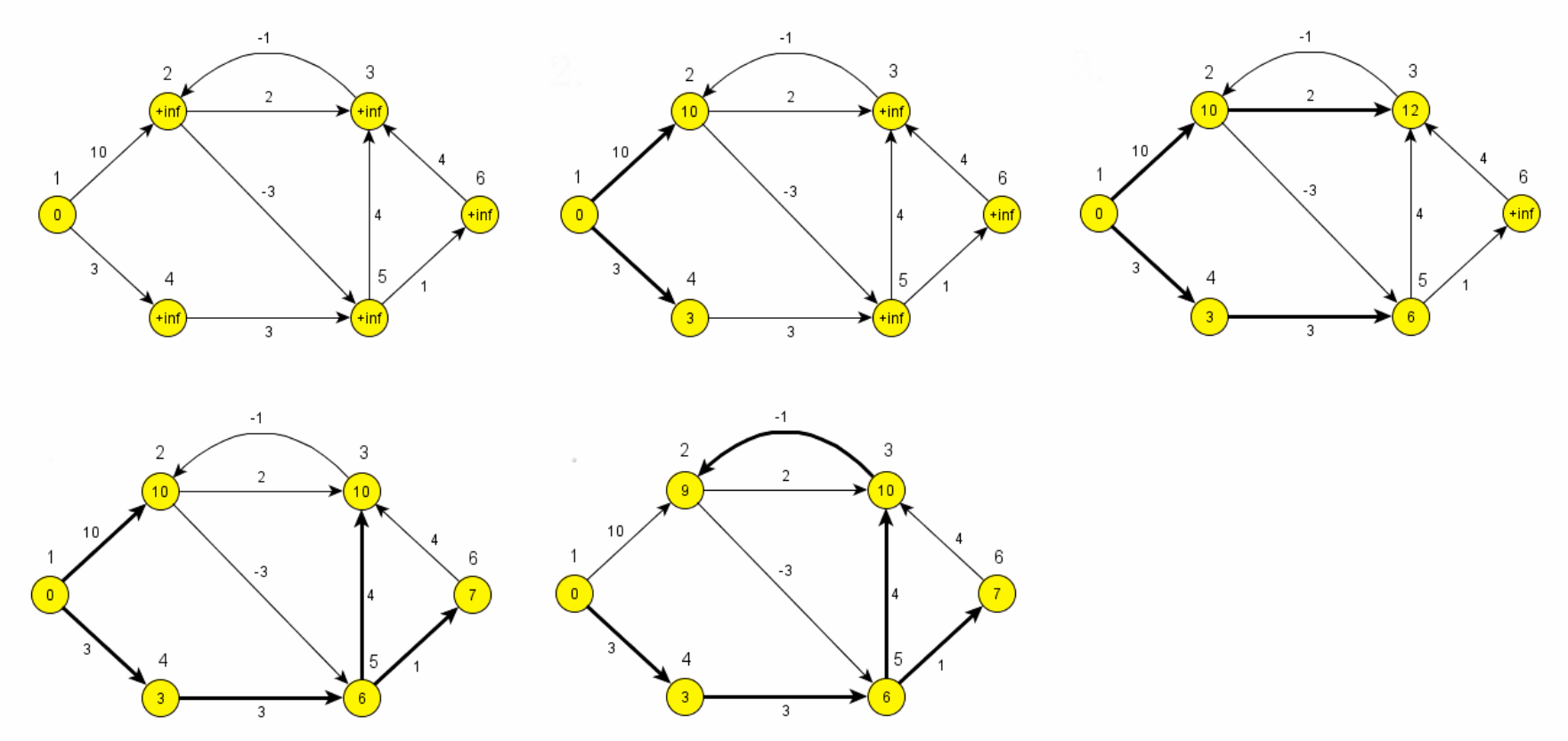
Dvoufázová s vnořeným cyklem.

Implementace frontou:

Efektivnější, do Q přidávány u, u kterých relaxace snížila $d[v]$.

Počet opakování není fixní, vázán na Q.

16. Ukázka B-F algoritmu



17. Klasická implementace B-F

Fáze 2 nepovinná, detekuje negativní hrany.

```
def bellman(G, start, end):
    d = [inf] * (len(G) + 1)          #Set infinite distance
    p = [-1] * (len(G) + 1)           #No predecessors
    d[start] = 0                      #Start d[s] = 0
    for i in range(len(G)-1):         #First phase
        for u in G:
            for v, dv in G[u].items():
                if d[v] > d[u] + dv:  #We found a better way
                    d[v] = d[u] + dv  #Update distance
                    p[v] = u          #Update predecessor
    for u in G:                       #Second phase
        for v, dv in G[u].items():
            if d[v] > d[u] + dv:      #Found edge decreasing d[v]
                return True           #Negative edge found
    return False                      #No negative edge
```

18. Implementace B-F frontou

```
def bellman(G, start, end):
    d = [inf] * (len(G) + 1)          #Set infinite distance
    p = [-1] * (len(G) + 1)           #No predecessors
    Q = []                            #Queue
    Q.append(start)                   #Add start vertex
    d[start] = 0                      #Start d[s] = 0
    while Q:
        u = Q.pop(0)                  #Pop first element
        for v, dv in G[u].items():    #Relaxation, all (u,v)
            if d[v] > d[u] + dv:      #We found a better way
                d[v] = d[u] + dv      #Update distance
                p[v] = u              #Update predecessor
                Q.append(v)           #Add to Q
```

Ukázka:

```
bellman(G, 1, 2)
path(p, 1, 2)
>> 1 4 5 6 3 2
```

19. Nejkratší cesty mezi všemi páry uzlů

Hledání nejkratších cest mezi všemi dvojicemi uzlů v G.

Pro orientované i neorientované grafy.

Předpoklad: graf neobsahuje záporné w.

Metody:

- Opakování Dijkstry nebo B-F pro všechny kombinace
  Neefektivní, používáno jen pro řídké grafy.
- Specializované algoritmy
  Floyd-Warshal, prodlužování w-délky.

Přechod od spojové reprezentaci k maticové: D, W, P.

Kombinace matice sousednosti a ohodnocení.

Matice w-délek a w-vzdáleností, předchůdců.

20. Matice w-délek W

Graf $G = \langle H, U, \rho \rangle$, ohodnocení $w : H \mapsto \mathbb{R}$.

Čtvercová matice $W = [w_{i,j}]$ řádu n je maticí w-délek

$$w_{ij} = \begin{cases} 0, & \text{pokud } i = j, \\ w(u_i, u_j), & \text{pokud } i \neq j,\ (u_i, u_j) \in H, \\ \infty, & \text{pokud } i \neq j,\ (u_i, u_j) \notin H. \end{cases}$$

Pro G lze snadno sestavit.

Pro neorientovaný graf symetrická, kombinuje incidenci a ohodnocení.

Výchozí vstupní matice pro výpočty.

Reprezentace v Pythonu: 2D seznam

```
W=[
    [w11, w12, ..., w1n],
    [w21, w22, ..., w2n],
    ...
    [wn1, wn2, ..., wnn],
  ]
```

21. Matice *w*-vzdáleností *D*

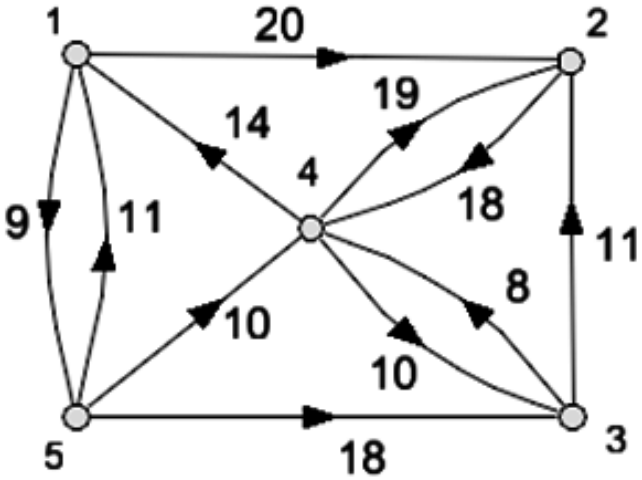
Čtvercová matice $D = [d_{i,j}]$ řádu n je maticí w vzdáleností

$$d_{ij} = d_w(u_i, u_j).$$

Nejkratší vzdálenosti mezi páry uzlů.

Pro neorientovaný G symetrická.

Získána následným výpočtem, výsledek grafových algoritmů.



$$W = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 20 & \infty & \infty & 9 \\ 2 & \infty & 0 & \infty & 19 & \infty \\ 3 & \infty & 11 & 0 & 10 & \infty \\ 4 & 14 & 18 & 8 & 0 & \infty \\ 5 & 11 & \infty & 18 & 10 & 0 \end{array}$$

$$D = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 20 & 27 & 19 & 9 \\ 2 & 33 & 0 & 27 & 19 & 42 \\ 3 & 24 & 11 & 0 & 10 & 33 \\ 4 & 14 & 18 & 8 & 0 & 23 \\ 5 & 11 & 28 & 18 & 10 & 0 \end{array}$$

Ilustruje dosažitelnost uzlů z jiných uzlů (konečné vzdálenosti).

22. Matice předchůdců P

Čtvercová matice $P = [p_{i,j}]$ řádu n je maticí předchůdců

$$p_{ij} = \begin{cases} 0, & \text{pokud } i = j \text{ nebo } \nexists \text{ min. cesta } u_i \to u_j, \\ u_k, & u_k \text{ předchůdce } u_j, \exists \text{ min. cesta } u_i \to u_j. \end{cases}$$

Analogie s předchůdcem uzlu ve spojové reprezentaci.

Prvek p_{ij} obsahuje stejnou informaci jako 1D pole $p[u]$.

Umožňuje rekonstrukci *nejkratší cesty*.

Provádí se opět od koncového bodu k počátečnímu: rekurze, iterace.

Získána výpočtem.

$$P = \begin{bmatrix} 0 & 1 & 5 & 5 & 1 \\ 4 & 0 & 4 & 2 & 1 \\ 4 & 3 & 0 & 3 & 1 \\ 4 & 4 & 4 & 0 & 1 \\ 5 & 4 & 5 & 5 & 0 \end{bmatrix}.$$

23. Rekonstrukce nejkratší cesty, nerekurzivní

Od posledního uzlu *j* postupujeme k prvnímu uzlu *i*.

Nutno prohodit počáteční a koncový uzel.

```
def path(P, i, j):                      #Path from i to j
    print(j)                            #print end vertex j
    #print(j + 1)                       #If index starts at i=0
    while i != j:                       #Until we start=stop
        j = P[i][j]                     #Predecessor P[i][j]
        #j = P[i][j]-1                  #If index starts at i=0
        if j == 0:                      #No predecessor, stop
        #if j == -1:                    #If index starts at i=0
            print("No Path.")
            return
        print(j)                        #Print point
        #print(j+1)                     #If index starts at i=0
```

24. Rekonstrukce nejkratší cesty, rekurze

Rekurzivní implementace využívající P.

Analogie postupu s polem předchůdců.

```
def pathm(P, i, j):                          #Path from i to j
    if i==j:                                 #Start = end, stop
        print(i)
        #print(i+1)                          #If index starts at i=0
    else:
        if P[i][j] == 0:                     #No predecessor
        #if P[i][j] -1 == 0:                 #If index starts at i=0
            print("No Path.")
        else:
            pathm(P, i, P[i][j])             #Path to P[i][j]
            #pathm(P, i, P[i][j]-1)          #If index starts at i=0
            print(j)
```

25. Metody výpočtu D

2 základní metody výpočtu:

1. Opakovaným zpřesňováním w-délky.
2. Floyd-Warshalův algoritmus.

26. Výpočet zpřesňováním *w* délky

Graf $G = \langle H, U, \rho \rangle$, $\|U\| = n$.

Nejkratší cesta uzly v G tvořena nejvýše $n - 1$ hranami.

Část nejkratší cesty je také nejkratší.

Mezi u_i, $u_j \in U$ hledáme nejkratší cestu $d_w(u_i, u_j)$ s nejvýše m hranami.

Nechť $u_k = p(u_j)$

$$u_k = p(u_j),$$

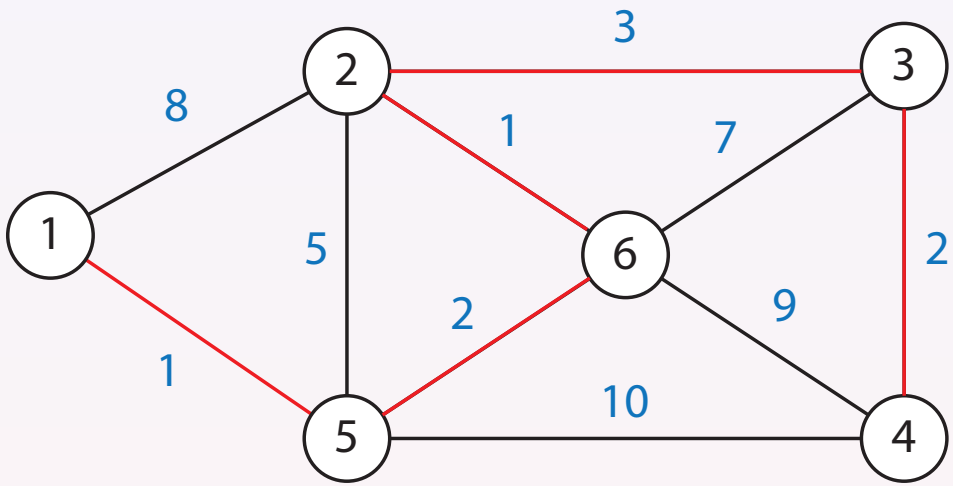
pak $u_i \to u_k$ také nejkratší, tvoří ji nejvýše $m - 1$ hran

$$d_w(u_i, u_j) = d_w(u_i, u_k) + w_j.$$

Resymbolizace: $d_{i,j}^{(m)} \equiv d_w(u_i, u_j)$, $d_{ik}^{(m-1)} \equiv d_w(u_i, u_k)$, pak

$$d_{ij}^{(m)} = d_{ik}^{(m-1)} + w_{kj}.$$

27. Rozklad nejkratší cesty



$$
\begin{aligned}
d_{14}^{(5)} &= d_{13}^{(4)} + w_{34}, \\
&= d_{12}^{(3)} + w_{23} + w_{34}, \\
&= d_{16}^{(2)} + w_{62} + w_{23} + w_{34}, \\
&= d_{15}^{(1)} + w_{56} + w_{62} + w_{23} + w_{34}, \\
&= d_{11}^{(0)} + w_{15} + w_{56} + w_{62} + w_{23} + w_{34}, \\
&= w_{15} + w_{56} + w_{62} + w_{23} + w_{34}.
\end{aligned}
$$

Každý úsek nejkratší spojnicí mezi 2 uzly.

28. Princip metody prodlužování w délky

Inverzní postup vzhledem ke zkracování délky, obtížnější $\Rightarrow$ optimalizace.

Pro cestu délky m prohledány všechny existující cesty o délce $m-1$.

Předpoklad: G neobsahuje záporné hrany.

Opakované prodlužování nejkratší cesty $u_i \to u_j$ s $m-1$ hranami o 1 hranu.

Hledáme nejkratší cestu přes všechny předchůdce u_k uzlu u_j.

Známe $d_{ik}^{(m-1)}$, hledáme $d_{ij}^{(m)}$, optimalizační problém

$$d_{ij}^{(m)} = \min_{1 \le k \le n} \left(d_{ik}^{(m-1)} + w_{kj} \right),$$

kde

$$d_{ij}^{(0)} = \begin{cases} 0, & i = j, \\ \infty, & i \neq j, \end{cases}$$

a $d_{ij}^{(1)} = w_{ij}$.

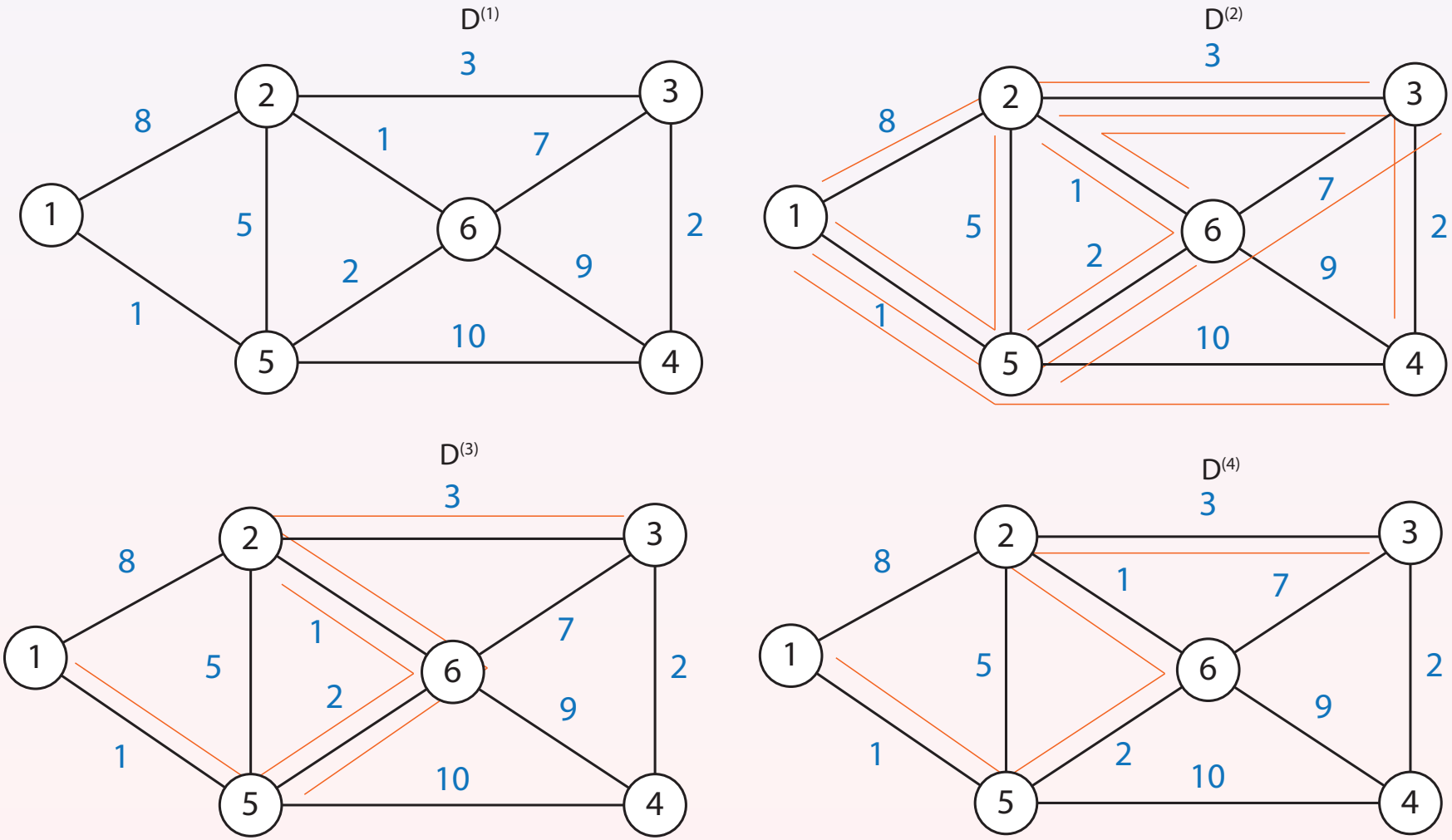
Provádíme pro $m \geq 2$, pro $m = 1$ $d_{ij}^{(1)} = w_{ij}$.

Neefektivní metoda, pouze pro malé grafy: $O(\|U\|^4)$.

V praxi používán Floyd-Warshal.

29. Ukázka funkcionality algoritmu

Zvýrazněny $d_{ij}^{(m)}$, kde došlo ke zlepšení.



30. Algoritmus prodlužování *w* délky

Klasické násobení matic $C = AB$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

Pro C, A, B prodlužování trasy

$$c_{ij} = \min_{1 \le k \le n} \left(a_{ik} + b_{kj} \right),$$

představuje modifikované násobení $C = A \otimes B$.

Opakováno $n - 1$ x.

Problém lze přepsat do tvaru

$$\begin{aligned} D^{(1)} &= W, \\ D^{(2)} &= D^{(1)} \otimes W = W \otimes W = W^2, \\ D^{(3)} &= D^{(2)} \otimes W = W \otimes W \otimes W = W^3, \\ \dots &\quad \dots \quad \dots \\ D^{(n-1)} &= D^{(n-2)} \otimes W = W^{n-1}. \end{aligned}$$

Na matici W aplikujeme $n - 1$ modifikované násobení.

Postupné hledání nejkratší cesty tvořené 2, 3, 4, ..., uzly

31. Implementace metody prodlužování *w* délky

Prodloužení *w* délky o 1, upravené násobení matic

```
def extend(D, W):
    n = len(W)
    Dn = [[inf] * n for i in range(n)]          #Empty matrix of Inf
    for i in range(0, n):                       #For each start node u
        for j in range(0, n):                   #For each end node uj
            x = inf                             #Assign minimum
            for k in range(0, n):               #Find shortest path
                x = min(x, D[i][k] + W[k][j])   #by extending old one
            Dn[i][j] = x                        #Store path
    return Dn
```

Prodloužení *w* délek všech nejkratších cest:

```
def mmshortest(W):
    D = W                       #Initialize D(1)=W
    n = len(W)
    for m in range(2, n):       #Extend path
        D = extend(D, W)
        print(D)
    return D                    #Return D(n)
```

32. Výsledky

Matice $D^{(1)} = W$:

```
W = [
      [0,   8,   inf, inf, 1,   inf],
      [8,   0,   3,   inf, 5,   1],
      [inf, 3,   0,   2,   inf, 7],
      [inf, inf, 2,   0,   10,  9],
      [1,   5,   inf, 10,  0,   2],
      [inf, 1,   7,   9,   2,   0]
    ]
```

Matice $D^{(2)} - D^{(5)}$:

$$D^{(2)} = \begin{bmatrix} 0 & 6 & 11 & 11 & 1 & 3 \\ 6 & 0 & 3 & 5 & 3 & 11 \\ 11 & 3 & 0 & 2 & 8 & 4 \\ 11 & 5 & 2 & 0 & 10 & 9 \\ 1 & 3 & 8 & 10 & 0 & 2 \\ 3 & 1 & 4 & 9 & 2 & 0 \end{bmatrix}, D^{(3)} = \begin{bmatrix} 0 & 4 & 9 & 11 & 1 & 3 \\ 4 & 0 & 3 & 5 & 3 & 1 \\ 9 & 3 & 0 & 2 & 6 & 4 \\ 11 & 5 & 2 & 0 & 10 & 6 \\ 1 & 3 & 6 & 10 & 0 & 2 \\ 3 & 1 & 4 & 6 & 2 & 0 \end{bmatrix},$$

$$D^{(4)} = \begin{bmatrix} 0 & 4 & 7 & 11 & 1 & 3 \\ 4 & 0 & 3 & 5 & 3 & 1 \\ 7 & 3 & 0 & 2 & 6 & 4 \\ 11 & 5 & 2 & 0 & 8 & 6 \\ 1 & 3 & 6 & 18 & 0 & 2 \\ 3 & 1 & 4 & 6 & 2 & 0 \end{bmatrix}, D^{(5)} = \begin{bmatrix} 0 & 4 & 7 & 9 & 1 & 3 \\ 4 & 0 & 3 & 5 & 3 & 1 \\ 7 & 3 & 0 & 2 & 6 & 4 \\ 9 & 5 & 2 & 0 & 8 & 6 \\ 1 & 3 & 6 & 18 & 0 & 2 \\ 3 & 1 & 4 & 6 & 2 & 0 \end{bmatrix}.$$

33. Floyd-Warshallův algoritmus

Nejčastější metoda hledání nejkratších cest mezi páry uzlů.

Předpoklad: G nemá hrany $w < 0$.

Postupné zpřesňování rozšiřováním množiny vnitřních uzlů.

Vnitřní uzel cesty $u_1 \to u_k$

$$\langle u_1, u_2, u_3, \ldots, u_{k-1}, u_k \rangle ,$$

je její libovolný nekrajní uzel

$$\{u_2, u_3, \ldots, u_{k-1}\} .$$

V každém kroku se jeden uzel stává vnitřním a hledáme cesty přes něj.

Snadná implementace, trojice vnořených cyklů.

Složitost $O(\|U\|^3)$.

Řídké grafy: n-násobné použití Dijkstry + B-F.

34. Matice D w- délek

Matice D w-délek

$$D^{(k)} = [d_{ij}^{(k)}],$$

kde $d_{ij}^{(k)}$ je w-délka mezi uzly u_i, u_j tvořená k vnitřními uzly

$$\{u_1, u_2, u_3, \ldots, u_k\}.$$

Platí

$$d_{i,j}^{(k)} = \begin{cases} w_{i,j}, & k = 0, \\ \min(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}), & k \geq 1. \end{cases}$$

Nejkratší cesta $u_i \to u_j$ s k vnitřními uzly:

- nejde přes u_k, pak $d_{i,j}^{(k)} = d_{i,j}^{(k-1)}$,
- jde přes u_k, lze rozdělit na $u_i \to u_k$ a $u_k \to u_j$.

Ověření, zda cesta $u_i \to u_j$ není horší než cesta z uzlu $u_i \to u_j$ přes u_k.

Pokud cesta přes u_k výhodnější, aktualizován předchůdce $p_{ij} = p_{kj}$, obdoba relaxace.

Pomůcka: $D^{(k)}$, k vrchol přes který hledáme kratší cestu $u_i \to u_j$.

35. Matice P

Matice $P = [p_{ij}]$ předchůdců.

p_{ij} předchůdcem u_j hrany $\{u_i, u_j\}$, $p(u_j) = u_i$.

Inicializace P:

$$p_{i,j}^{(0)} = \begin{cases} 0, & i = j \vee w_{i,j} = \infty, \\ i, & \text{jinak.} \end{cases}$$

Žádný vnitřní uzel, předchůdcem uzlu u_j hrany $\{u_i, u_j\}$ uzel u_i.

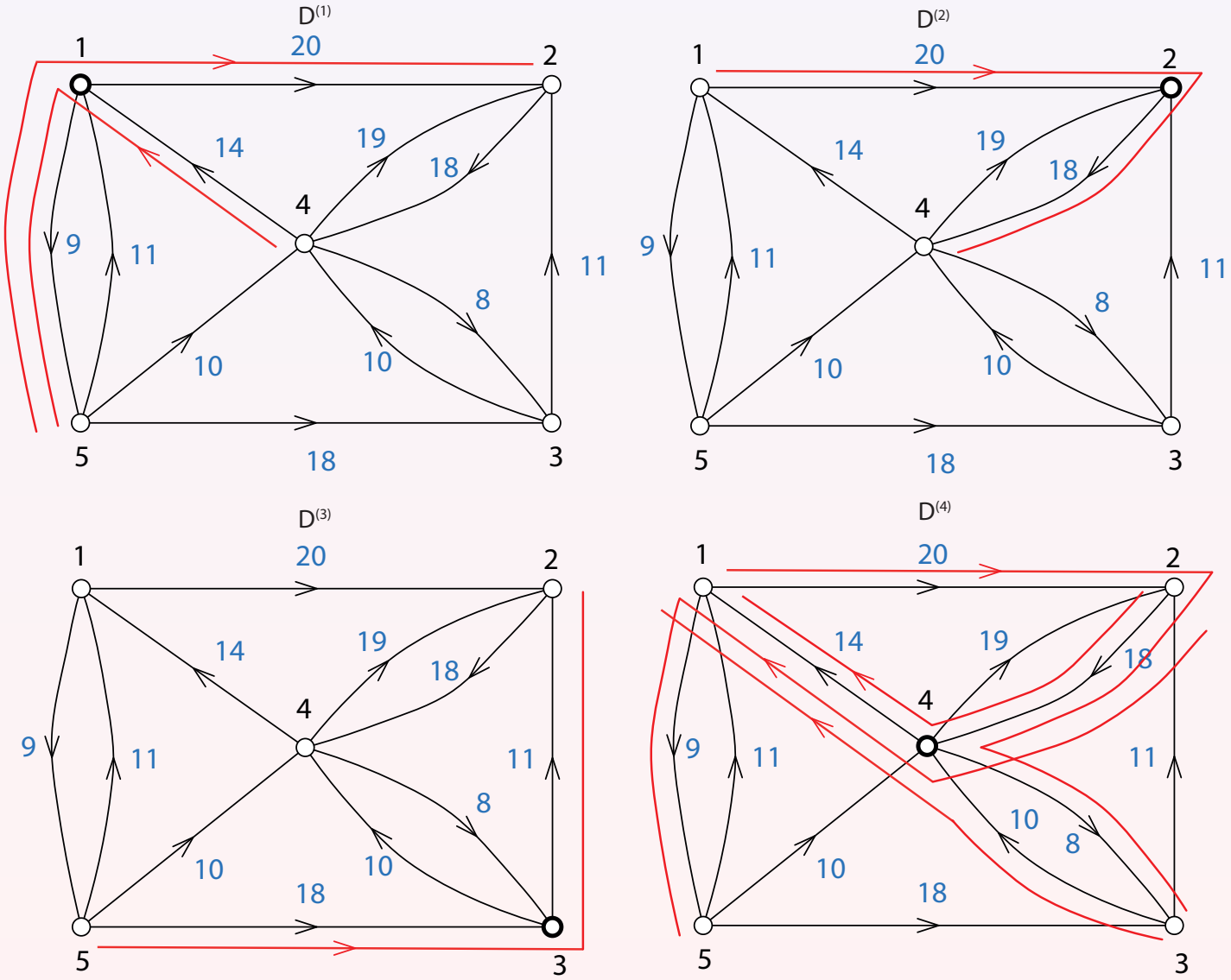
Aktualizace P:

$$p_{i,j}^{(k)} = \begin{cases} p_{i,j}^{(k-1)}, & d_{i,j}^{(k)} \leq d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}, \\ p_{k,j}^{(k-1)}, & d_{i,j}^{(k)} > d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}. \end{cases}$$

Pokud je výhodnější cesta přes u_k, aktualizujeme předchůdce u_i uzlu u_j na u_k.

Rekonstrukce cesty $u_i \to u_j$ zpětně od koncového uzlu u_j.

36. Ukázka funkcionality Floyd-Warshall



37. Implementace Floyd-Warshall

Snadná implementace.

Tři vnořené cykly + neúplná podmínka.

```
def fw(P, W):
    D = W                                          #Initialize D(1)=W
    n = len(W)
    for k in range(0, n):                          #For growing inner nodes
        for i in range(0, n):                      #For each start node ui
            for j in range(0, n):                  #For each end node uj
                if D[i][j] > D[i][k] + D[k][j]: #Shorter path over uk
                    D[i][j] = D[i][k] + D[k][j] #Assign shorter path
                    P[i][j] = P[k][j]              #Update predecessor
    return P, D
```

Doba běhu $O(\|U\|^3)$.

Neefektivní pro velké řídké grafy.

38. Výsledky, Floyd-Warshall, 1/2

Matice W, P :

```
W = [                                    P = [
     [0,   20,   inf, inf,  9],               [0, 1, 0, 0, 1],
     [inf, 0,    inf, 18,  inf],              [0, 0, 0, 2, 0],
     [inf, 11,   0,   10,  inf],              [0, 3, 0, 3, 0],
     [14,  19,   8,   0,   inf],              [4, 4, 4, 0, 0],
     [11,  inf, 18,   10,  0],                [5, 0, 5, 5, 0],
     ]      ]
```

Volání funkce:

```
P,D = fw(P, W)
```

Matice $D^{(2)} - D^{(5)}$:

$$D^{(1)} = \begin{bmatrix} 0 & 20 & \infty & \infty & 9 \\ \infty & 0 & \infty & 18 & \infty \\ \infty & 11 & 0 & 10 & \infty \\ 14 & 19 & 8 & 0 & 23 \\ 11 & 31 & 18 & 10 & 0 \end{bmatrix}, D^{(2)} = \begin{bmatrix} 0 & 20 & \infty & 38 & 9 \\ \infty & 0 & \infty & 18 & \infty \\ \infty & 11 & 0 & 10 & \infty \\ 14 & 19 & 8 & 0 & 23 \\ 11 & 31 & 18 & 10 & 0 \end{bmatrix}, D^{(3)} = \begin{bmatrix} 0 & 20 & \infty & 38 & 9 \\ \infty & 0 & \infty & 18 & \infty \\ \infty & 11 & 0 & 10 & \infty \\ 14 & 19 & 8 & 0 & 23 \\ 11 & 29 & 18 & 10 & 0 \end{bmatrix}$$

$$D^{(4)} = \begin{bmatrix} 0 & 20 & 46 & 38 & 9 \\ 32 & 0 & 26 & 18 & 41 \\ 24 & 11 & 0 & 10 & 33 \\ 14 & 19 & 8 & 0 & 23 \\ 11 & 29 & 18 & 10 & 0 \end{bmatrix}, D^{(5)} = \begin{bmatrix} 0 & 20 & 27 & 19 & 9 \\ 33 & 0 & 26 & 18 & 41 \\ 24 & 11 & 0 & 10 & 33 \\ 14 & 19 & 8 & 0 & 23 \\ 11 & 29 & 18 & 10 & 0 \end{bmatrix}.$$

39. Výsledky, Floyd-Warshall, 2/2

Postup:

$D^{(1)}$: hledáme všechny $u_i \to u_j$ přes u_1

$1 \to (1) \to 2, 1 \to (1) \to 3, 1 \to (1) \to 4, 1 \to (1) \to 5$: $2 \to (1) \to 1, 2 \to (1) \to 3, 2 \to (1) \to 4, 2 \to (1) \to 5$.

$3 \to (1) \to 1, 3 \to (1) \to 2, 3 \to (1) \to 4, 3 \to (1) \to 5$: $4 \to (1) \to 1, 4 \to (1) \to 2, 4 \to (1) \to 3, 4 \to (1) \to 5$.

$5 \to (1) \to 1, 5 \to (1) \to 2, 5 \to (1) \to 3, 5 \to (1) \to 4$: $4 \to (1) \to 1, 4 \to (1) \to 2, 4 \to (1) \to 3, 4 \to (1) \to 5$.

$D^{(2)}$: hledáme všechny $u_i \to u_j$ přes u_2

$1 \to (2) \to 2, 1 \to (2) \to 3, 1 \to (2) \to 4, 1 \to (2) \to 5$: $2 \to (2) \to 1, 2 \to (2) \to 3, 2 \to (2) \to 4, 2 \to (2) \to 5$.

$3 \to (2) \to 1, 3 \to (2) \to 2, 3 \to (2) \to 4, 3 \to (2) \to 5$: $4 \to (2) \to 1, 4 \to (2) \to 2, 4 \to (2) \to 3, 4 \to (2) \to 5$,

$5 \to (2) \to 1, 5 \to (2) \to 2, 5 \to (2) \to 3, 5 \to (2) \to 4$: $4 \to (2) \to 1, 4 \to (2) \to 2, 4 \to (2) \to 3, 4 \to (2) \to 5$.

Matice $P^{(0)} - P^{(5)}$:

$$P^{(0)} = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 3 & 0 & 3 & 0 \\ 4 & 4 & 4 & 0 & 0 \\ 5 & 0 & 5 & 5 & 0 \end{bmatrix}, \quad P^{(1)} = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 3 & 0 & 3 & 0 \\ 4 & 4 & 4 & 0 & 1 \\ 5 & 1 & 5 & 5 & 0 \end{bmatrix}, \quad P^{(2)} = \begin{bmatrix} 0 & 1 & 0 & 2 & 1 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 3 & 0 & 3 & 0 \\ 4 & 4 & 4 & 0 & 1 \\ 5 & 1 & 5 & 5 & 0 \end{bmatrix},$$

$$P^{(3)} = \begin{bmatrix} 0 & 1 & 0 & 2 & 1 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 3 & 0 & 3 & 0 \\ 4 & 4 & 4 & 0 & 1 \\ 5 & 3 & 5 & 5 & 0 \end{bmatrix}, \quad P^{(4)} = \begin{bmatrix} 0 & 1 & 4 & 2 & 1 \\ 4 & 0 & 4 & 2 & 1 \\ 4 & 3 & 0 & 3 & 1 \\ 4 & 4 & 4 & 0 & 1 \\ 5 & 4 & 5 & 5 & 0 \end{bmatrix}, \quad P^{(5)} = \begin{bmatrix} 0 & 1 & 5 & 5 & 1 \\ 4 & 0 & 4 & 2 & 1 \\ 4 & 3 & 0 & 3 & 1 \\ 4 & 4 & 4 & 0 & 1 \\ 5 & 4 & 5 & 5 & 0 \end{bmatrix}.$$