Grafové algoritmy

Programovací techniky

Grafy – Úvod - Terminologie

- Graf je datová struktura, skládá se z množiny vrcholů “V” a množiny hran mezi vrcholy “E”
- Počet vrcholů a hran musí být konečný a nesmí být nulový u vrcholů ani u hran
- Grafy
  - Orientované – hrana (u,v) označena šipkou u->v
  - Neorientované – pokud ex. (u,v), existuje také (v,u)
- Souvislost – graf je souvislý, jestliže pro všechny v(i) z V existuje cesta do libovolného v(j) z V, nesouvislý graf je rozdělen na komponenty
- Strom – graf, ve kterém pro každé 2 vrcholy existuje právě jedna cesta

Grafy – Úvod - Terminologie

FIGURE 12.4
Example of a Weighted Graph

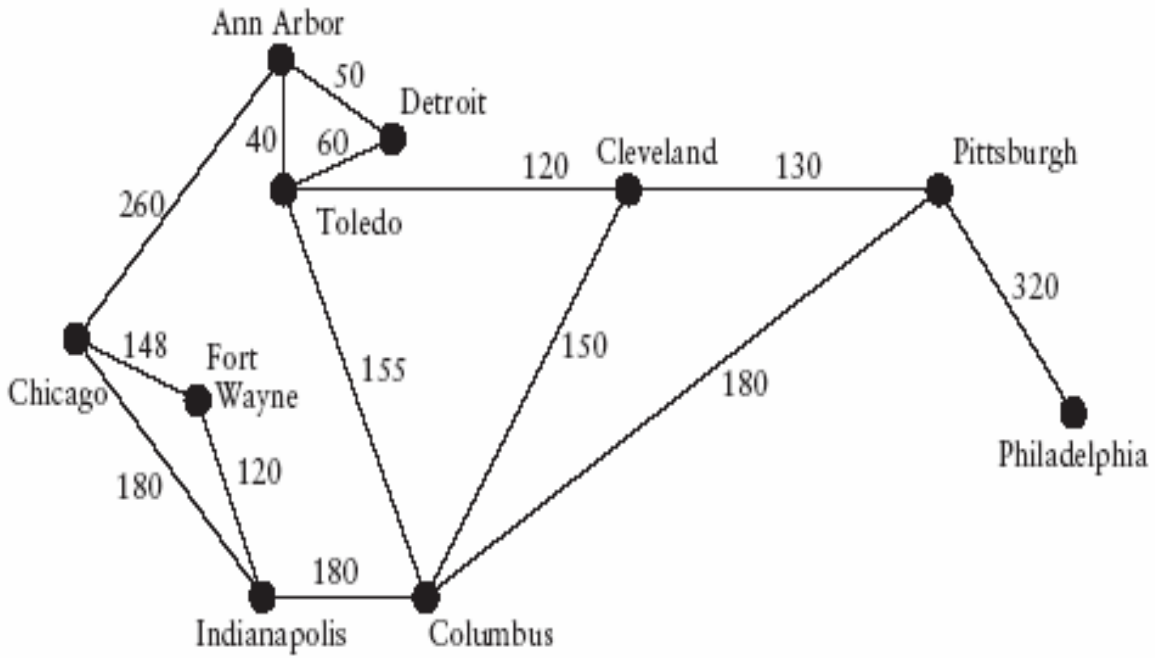
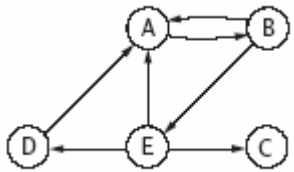


FIGURE 12.3
Example of a Directed Graph

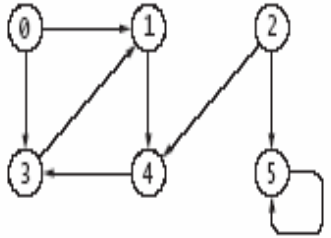


Grafy – Úvod – Reprezentace grafu

Matice sousednosti

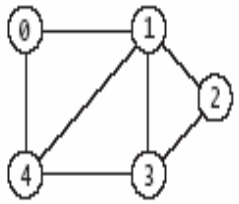
- Dvou dimenzionální pole, hodnoty A(i,j) jsou ohodnocením hrany mezi vrcholy i,j
- Pro neohodnocený graf jsou prvky matice typu Boolean
- Neorientované grafy mají A symetrickou podle diagonály

FIGURE 12.12
A Directed Graph and the Corresponding Adjacency Matrix



Row \ Column	[0]	[1]	[2]	[3]	[4]	[5]
[0]		1.0		1.0		
[1]					1.0	
[2]					1.0	1.0
[3]		1.0				
[4]				1.0		
[5]						1.0

FIGURE 12.13
Undirected Graph and Adjacency Matrix Representation

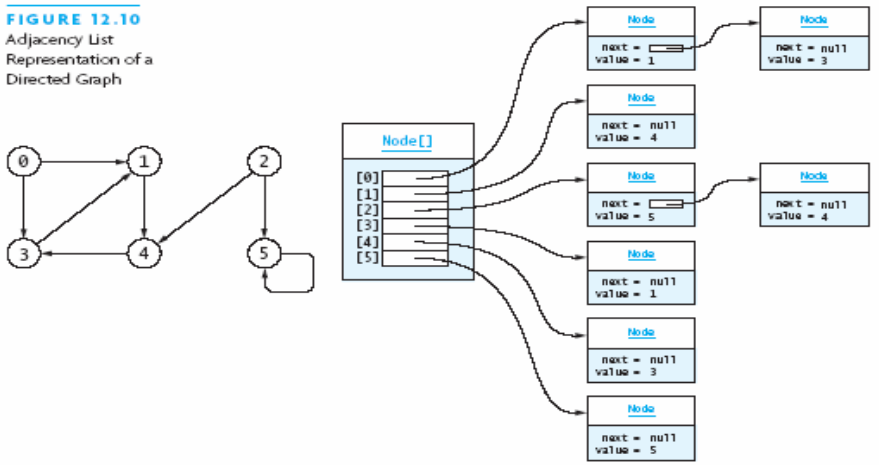


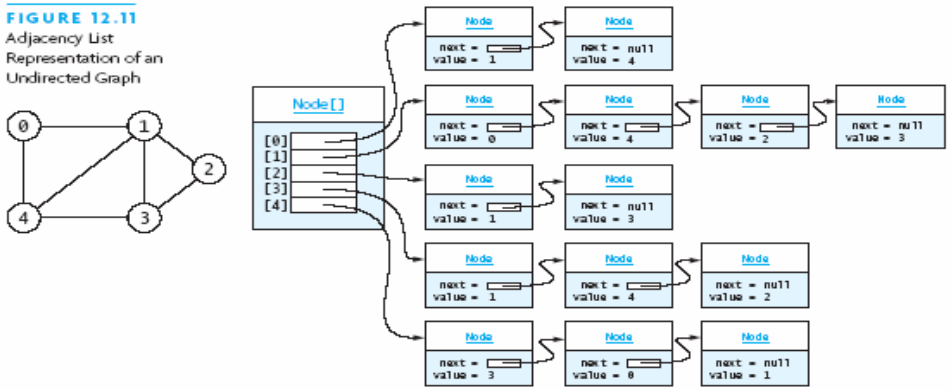
Row \ Column	[0]	[1]	[2]	[3]	[4]
[0]		1.0			1.0
[1]	1.0		1.0	1.0	1.0
[2]		1.0		1.0	
[3]		1.0	1.0		1.0
[4]	1.0	1.0		1.0	

Grafy – Úvod – Reprezentace grafu

Seznamem sousednosti

- Využívá pole seznamů
- Jeden seznam (sousedů) pro každý vrchol
- Vhodné, když chceme měnitelný počet hran



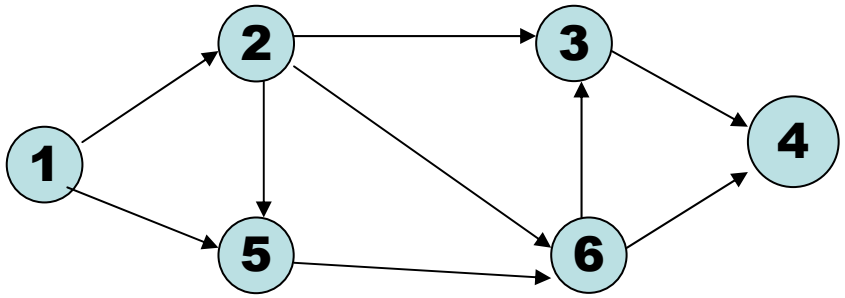


Grafy – Úvod – Reprezentace grafu

Seznamem sousednosti (reprezentace polem)

V:	i
1	1
2	3
3	6
4	7
5	7
6	8
7	10

E:	i
1	2
2	5
3	3
4	5
5	6
6	4
7	6
8	3
9	4



- **Snadno modifikovatelné pro ohodnocený graf**
- **Úsporná struktura**
- **Vhodné pro práci s neměnným počet vrcholů a hran**

Grafy – Úvod – Reprezentace grafu

- Plexová struktura
  - Vypadá jako reprezentace seznamem sousednosti, akorát vrcholy nejsou uloženy ve statickém poli, ale v dyn. struktuře (spojový seznam)
    - => Můžeme měnit počet hran i vrcholů

Hledání nejkratší cesty - v neohodnoceném grafu

Prohledáváním do šířky

```
void unweighted(Vertex s)
{
Queue q;
Vertex v,w;

q = new Queue();
q.enqueue(s); s.dist=0;

while (!q.isEmpty())
  {
    v = q.dequeue();
    v.known = true; // Not really needed anymore

    for each w adjacent to v
    if (w.dist == INFINITY)
    {
       w.dist = v.dist + 1;
       w.path = v;
       q.enqueue(w);
    }
  }
}
```

Slovníček:

Adjacent to – sousedící s

Enqueue – zařadit do fronty

Dequeue – vyndat z fronty

Složitost: O(m+n)

n – počet vrcholů, m – počet hran

Hledání nejkratší cesty - v neohodnoceném grafu

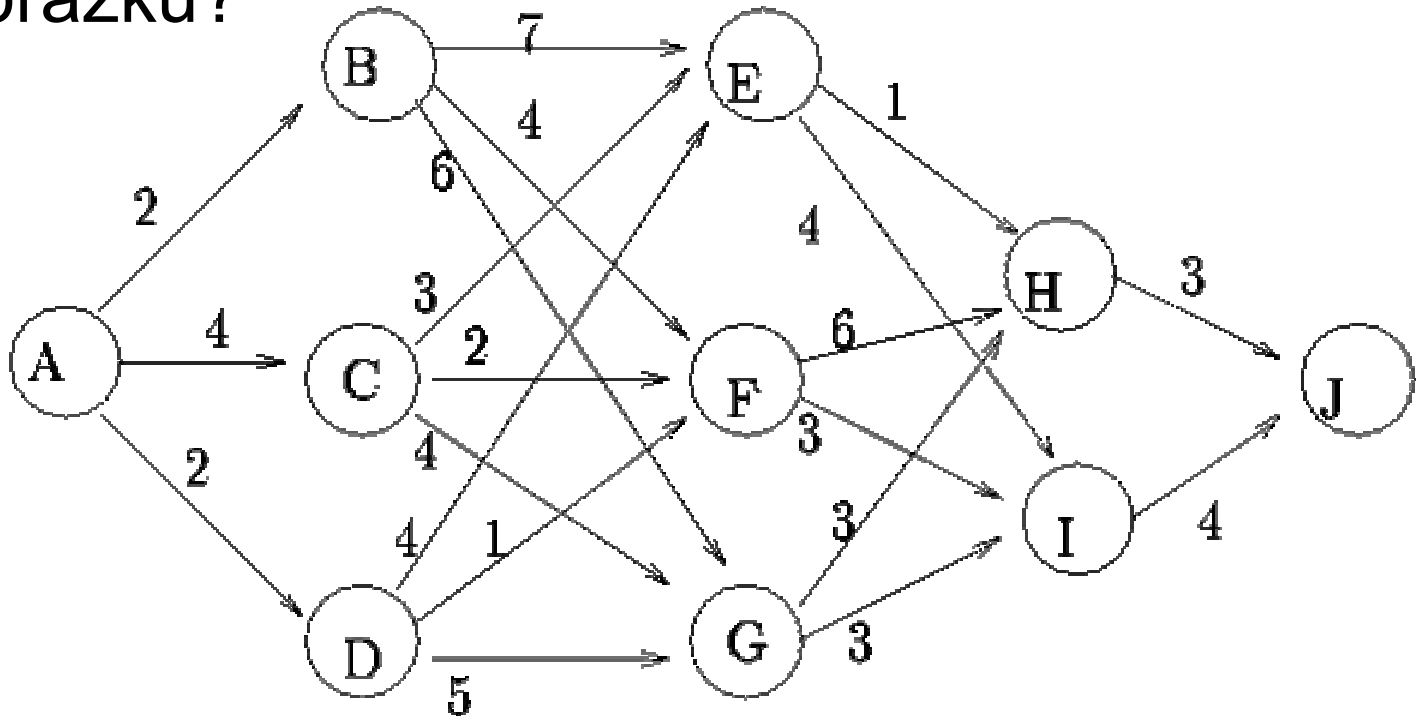
- Jak je to s hledáním do hloubky?
  - Jak se implementuje?
  - Lze využít při hledání cyklů a odhalování souvislosti grafu?
  - Dá se použít pro hledání nejkratší cesty?

Poznámka - Hledání do hloubky (backtracking) - upgrade

– Ořezávání
  - **pokud vidím, že v nějakém uzlu (stavu) prohledáváním jeho následníků URČITĚ nenajdu řešení, neprohledávám dále, ale vrátím se o úroveň výš**
    – Příklad – v grafu projít všechny uzly a a neurazit přitom délku větší než D
– Heuristika
  - **Upřednostním nějakého následníka mezi jinými, podle nějakého kritéria, které mi SNAD urychlí hledání řešení**
    – Příklad(negrafový) – proskákat šachovnici koněm – půjdeme na ta pole, z kterých budeme mít nejméně možností dalšího skoku
- A* algoritmus – zahrnuje prohledávání do šířky i do hloubky s heuristikou i ořezáváním, viz dále.

Hledání nekratší cesty v acyklickém orientovaném grafu

- DAG (directed acyclic graph) shortest path algorithms
- Jaká je nejkratší cesta z A do J v tomto obrázku?



Hledání nekratší cesty v acyklickém orientovaném grafu

- Graf je kvůli své struktuře rozložitelný na etapy

Označíme:

1. **etapa: A**
2. **etapa: B, C, D**
3. **etapa: E, F, G**
4. **etapa: H, I**
5. **etapa: J**

Nechť S udává uzel v etapě j a $f_j(S)$ je nejkratší cesta mezi uzly S a J, platí

$$f_j(S) = \min_{\text{nodes } z \text{ in stage } j+1} \{c_{sz} + f_{j+1}(Z)\}$$

Kde c_{sz} udává spočtený "oblak hran" SZ**. Tímto dostáváme rekurentní vztah, který řeší náš problém.**

DAG – řešení úlohy

- **Začneme s** $f_3(J)$
- **Etapa 4**
  - Zde se nic nerozhoduje, jen přejdeme do J. Tím, že jdeme do J tedy dostáváme $f_4(H)=3$ a $f_4(I)=4$.
- **Etapa 3 (viz tabulka S_3)**
  - Jak spočítat $f_3(F)$. Z F můžeme jít do H nebo do I.
    - Hrana do H má hodnotu 6, následující je $f_4(H)=3$. Celkem tedy 9.
    - Hrana do I má hodnotu 3, následující je $f_4(I)=4$. Celkem tedy 7.
  - Jakmile se tedy dostaneme do F, je nelepší jít přes I s vydáním = 7.
  - Stejně pro E, G.
- **Pokračujeme takto až do etapy 1**

S_3	$cs_3z_3 + f_4(Z_3)$		$f_3(S_3)$	Decision
	H	I		Go to
E	4	8	4	H
F	9	7	7	I
G	6	7	6	H

S_2	$cs_2z_2 + f_3(Z_2)$			$f_2(S_2)$	Decision
	E	F	G		Go to
B	11	11	12	11	E or F
C	7	9	10	7	E
D	8	8	11	8	E or F

S_1	$cs_1z_1 + f_2(Z_1)$			$f_2(S_1)$	Decision
	B	C	D		Go to
A	13	11	11	11	C or D

DAG shortest path

- Složitost hledání – O(n+m)
- Nejdelší cesta v grafu

$$EC_1 = 0$$

$$EC_w = max_{(v,w)\in E}(EC_v + c_{v,w})$$

•V diagramu činností – ukazuje nejkratší čas ukončení projektu

Floyd-Warshallův algoritmus

- Algoritmus hledající minimální cestu mezi všemi páry vrcholů (all-pair shortest path algorithm)
- Vhodný pro husté grafy – v tom případě rychlejší než Dijkstra opakovaný pro všechny vrcholy
- Pracuje s maticí sousednosti
- Složitost $O(n^3)$
- Můžeme stanovit max. počet vrcholů, přes které se jde
- Je technikou dynamického programování – viz dále

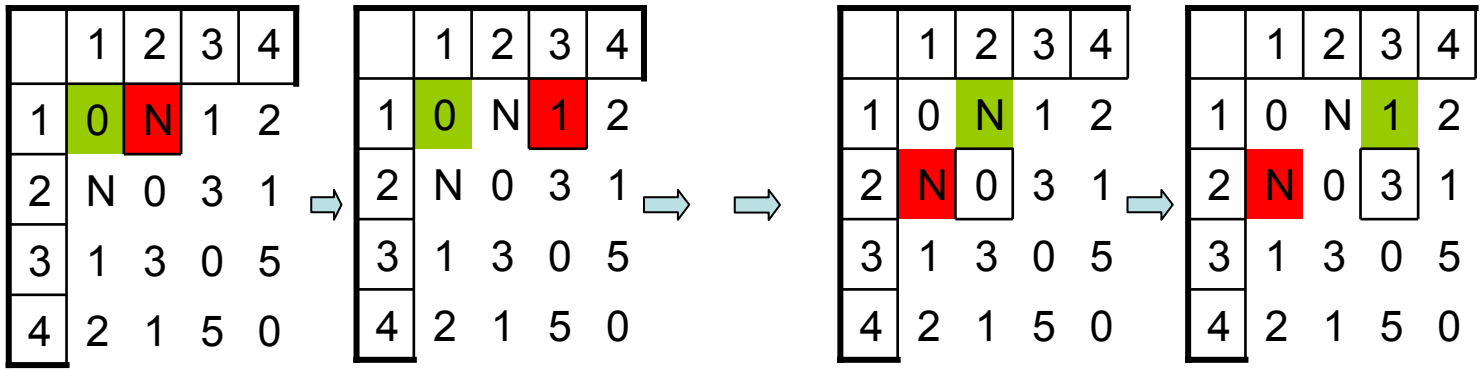
Floyd-Warshallův algoritmus

```
public static void allPairs(
int [][]a, int[][]d, int [][]path)
{

int n = a.length;
//initialize d and path
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++) {
    d[i][j] = a[i][j];
    path[i][j] = NOT_A_VERTEX;
  }

for (int k = 0; k < n; k++)
  for (int i = 0; i < n; i ++)
    for (int j = 0; j < n; j++)
      if (d[i][k]+d[k][j] < d[i][j]) {
        //update shortest path
        d[i][j] = d[i][k]+d[k][j];
        path[i][j] = k;
      }
}
```

- Matice sousednosti je uložena v **a**
- **d** je matice aktuálně spočtených nejkratších vzdáleností
- **path** je matice nejkratších mezicest, je nainicializována na -1
- V každém kroku algoritmu zjišťuji, jestli existuje mezi vrcholy i,j kratší cesta přes vrchol k, pokud ano, nastavím vzdálenost v **d[i][j]** na novou velikost a do **path[i][j]** zaznamenám vrchol k

Floyd-Warshallův algoritmus

- **Příklad –** je dána matice sousednosti grafu. FW algoritmem nalezněte nejkratší cestu mezi všemi vrcholy



(k,i,j)=(0,0,1)

0+"N" < "N" ?

•NENÍ

(k,i,j)=(0,0,2)

0+1 < 1 ?

•NENÍ

(k,i,j)=(0,1,1)

"N"+"N" < 0 ?

•NENÍ

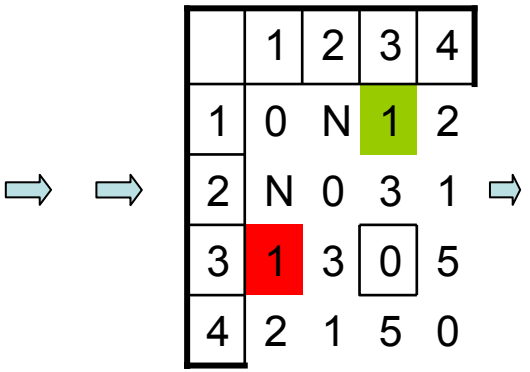
(k,i,j)=(0,1,2)

"N"+1 < 3 ?

•NENÍ

Floyd-Warshallův algoritmus

- Příklad - pokračování

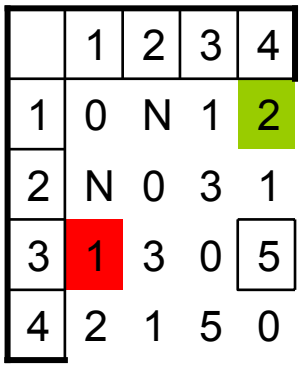


	1	2	3	4
1	0	N	1	2
2	N	0	3	1
3	1	3	0	5
4	2	1	5	0

(k,i,j)=(0,2,2)

1+1 < 0 ?

•NENÍ



	1	2	3	4
1	0	N	1	2
2	N	0	3	1
3	1	3	0	5
4	2	1	5	0

(k,i,j)= (0,2,3)

1+2 < 5 ?

•JE

Nová d

	1	2	3	4
1	0	N	1	2
2	N	0	3	1
3	1	3	0	3
4	2	1	5	0

Nová path

	1	2	3	4
1	-1	-1	-1	-1
2	-1	-1	-1	-1
3	-1	-1	-1	0
4	-1	-1	-1	-1

Kratší cesta z 3 do 4 vede přes 0

...

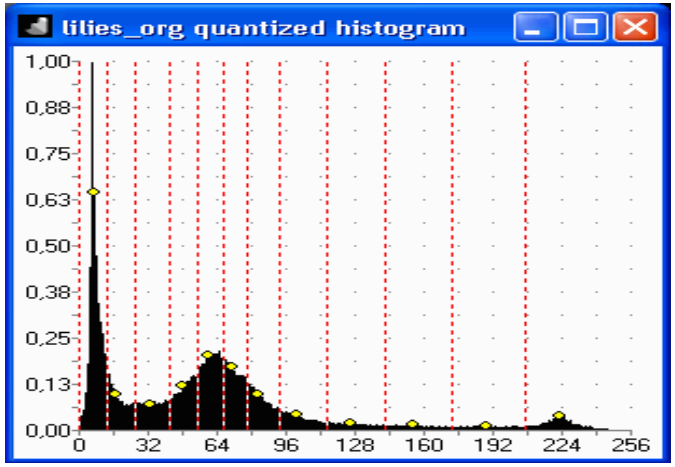
Floyd-Warshallův algoritmus

- Rekonstrukce nejkratší cesty mezi i,j
  - V **path[i][j]** je index vrcholu k, přes který se má jít. Pokud k není -1 podívám se do **path** na nejkratší cestu mezi i,k a k,j
  - Toto opakuji rekurzivně dokud nenajdu **path[i][j] == -1**.

Dynamické programování

- Algoritmus na hledání nejkratší cesty v acyklickém grafu a Floyd-Warshall jsou zástupci techniky **dynamického programování (DP)**
- **DP**
  - **Se snaží rozdělit původní problém na podproblémy a pro ně nalézt nejlepší řešení. Řešení většího problému pomocí nejlepších malých řešení se provádí na základě rekurzivního vztahu. Až sem by se jednalo čistě o techniku Rozděl a panuj (třeba quicksort je RP). Nové a speciální je to, že se nejdříve zbavíme kandidátů, kteří nemají šanci na úspěch. Navíc se ukládá (do tabulky) informace o kandidátech, kteří mají šanci, že budou vést posloupnost vedoucí k nejlepšímu řešení.**
  - **Problémy řešící DP**
    - **Negrafové - dají se na graf převést, nebo ne (uzávorkování matic, Fibonacciho čísla, ...)**

DP – zajímavý, nepříliš školní příklad

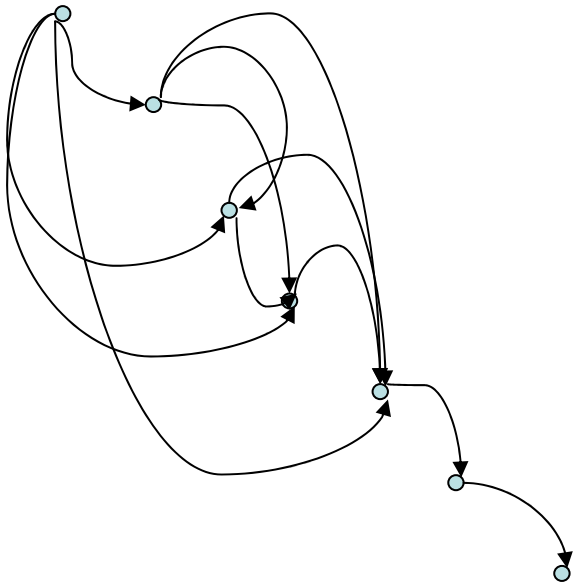


- **Úkolem je rozdělit histogram rel. četností šedi obrázku na předem daný počet oblastí (barev) M**
- **V každém intervalu mezi 0-255 se podle daných vzorců spočte chyba a naším úkolem je vybrat takové rozložení intervalů, aby součet chyb k příslušným intervalům byl co nejmenší -> dostáváme dyn. formulaci problému**

$$D_m(0,n] = \min_{m-1 \le r_{m-1} < n} \left\{ D_{m-1}(0, r_{m-1}] + e^2(r_{m-1}, n] \right\}$$

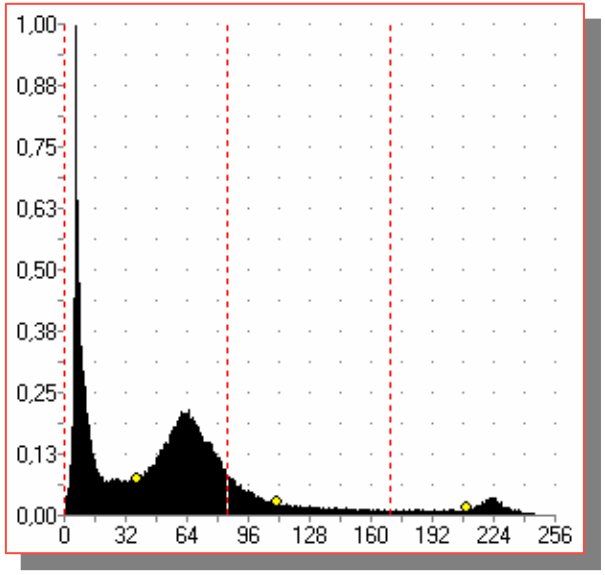
DP – zajímavý, nepříliš školní příklad

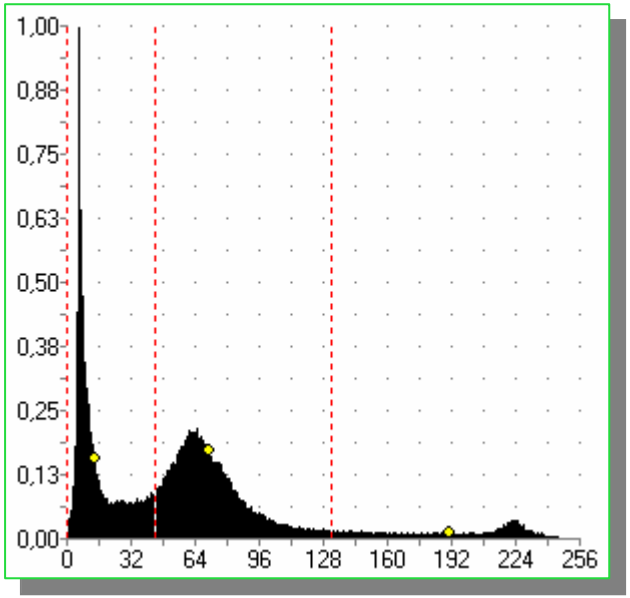
- **Převedení problém na hledání nekratší cesty v grafu délky M, ohodnocení hran je velikost chybové funkce**



- **Obrázek ukazuje interpretaci na graf**
  - **7 barev (místo 256), nejkratší cesta délky 3**
- **Nechá se použít například Floyd-Warshallův algoritmus, protože je snadno modifikovatelný pro nalezení nejratší cesty předem dané délky**
  - **Stačí horní mez pro k v algoritmu omezit na M, místo na n (počet vrcholů)**

DP – zajímavý, nepříliš školní příklad





Uniform

Optimal

Dijkstrův algoritmus

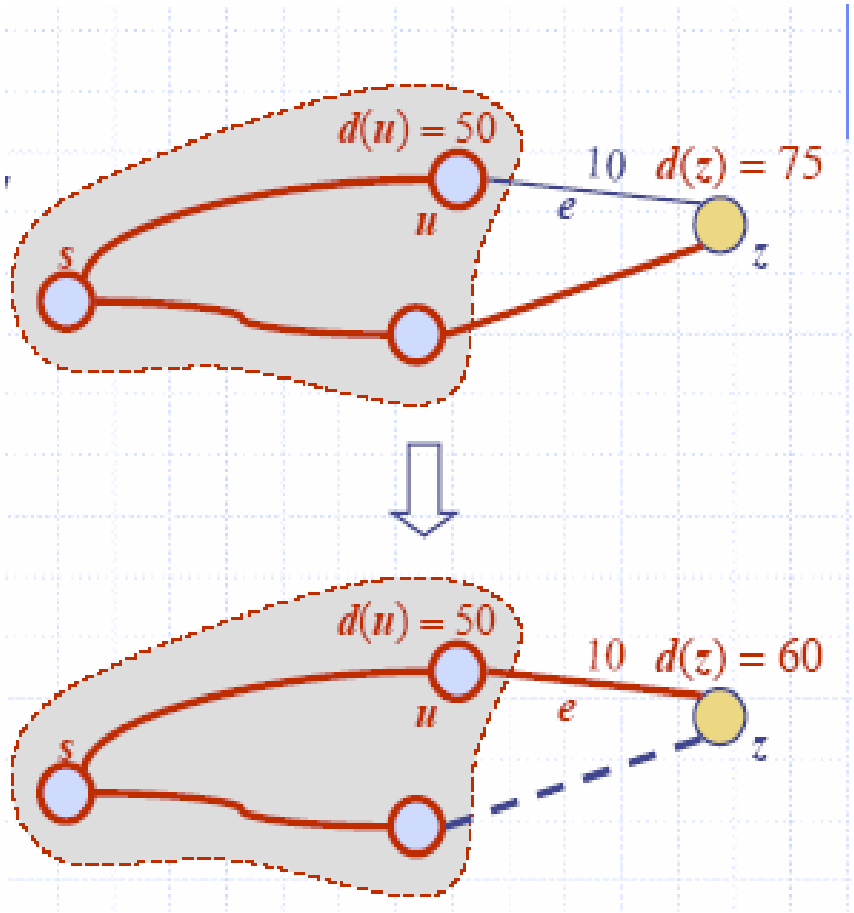
- **Hledání nejkratší cesty v ohodnoceném grafu**
- **Počítá vzdálenosti VŠECH vrcholů od počátečního**
- **Předpokládáme:**
  - **Graf je souvislý**
  - **Neorientované hrany**
  - **Má nezáporně ohodnocené hrany**

- **Z navštívených vrcholů vytváříme mrak. Zpočátku obsahuje počáteční vrchol S, nakonci všechny vrcholy**
- **V každém kroku – přidáme do mraku vrchol v vně mrak s nejmenší vzdáleností d(v) (prioritní fronta)**
- **Opravíme vzdálenosti vrcholů sousedících s v (vede často k relaxaci hran)**

Dijkstrův algoritmus

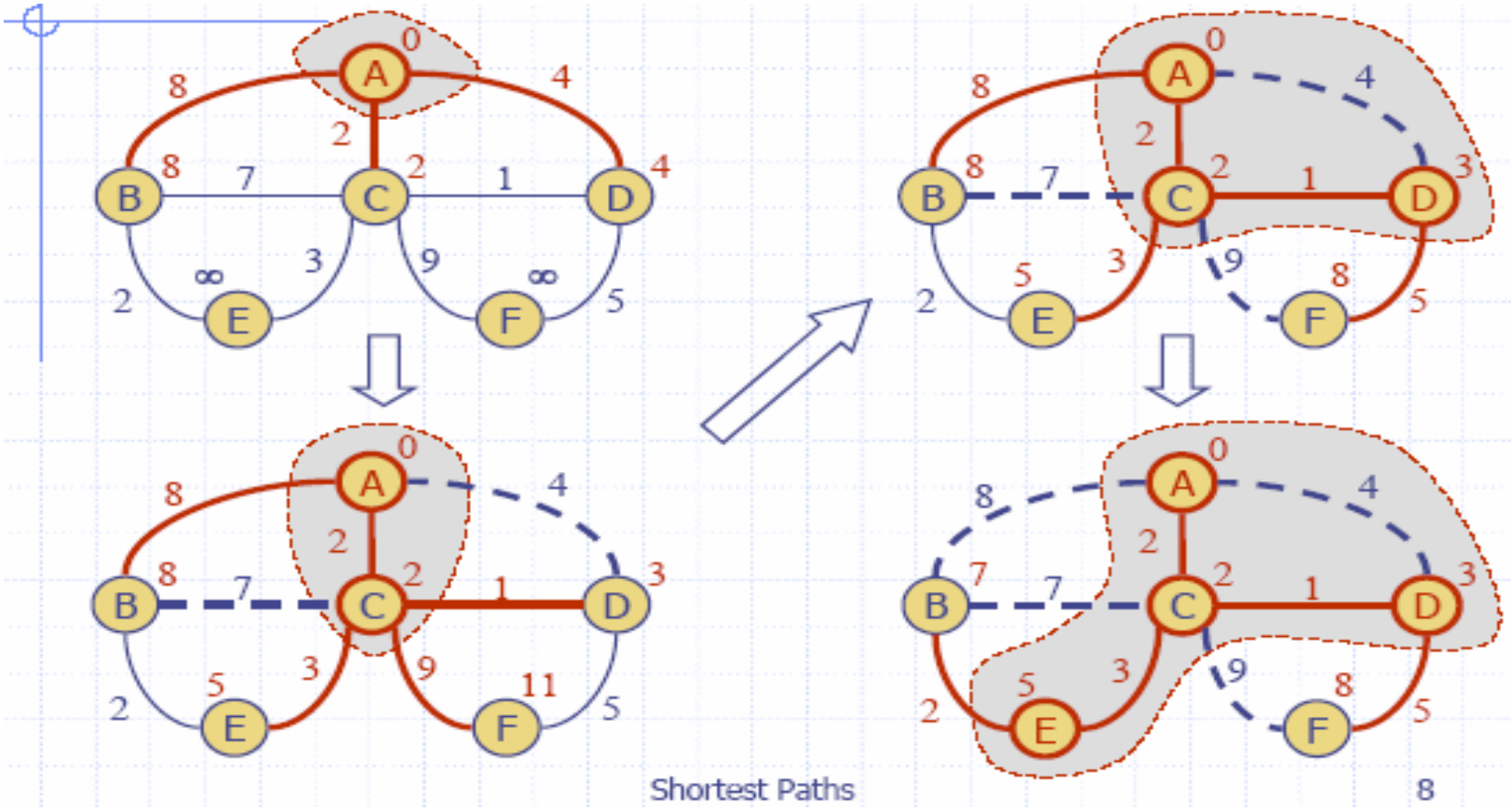
- Relaxace hran
  - Relaxace hrany e upravuje vzdálenost d(z) takto:

$d(z) \leftarrow \min\{d(z), d(u)+\text{weight}(e)\}$



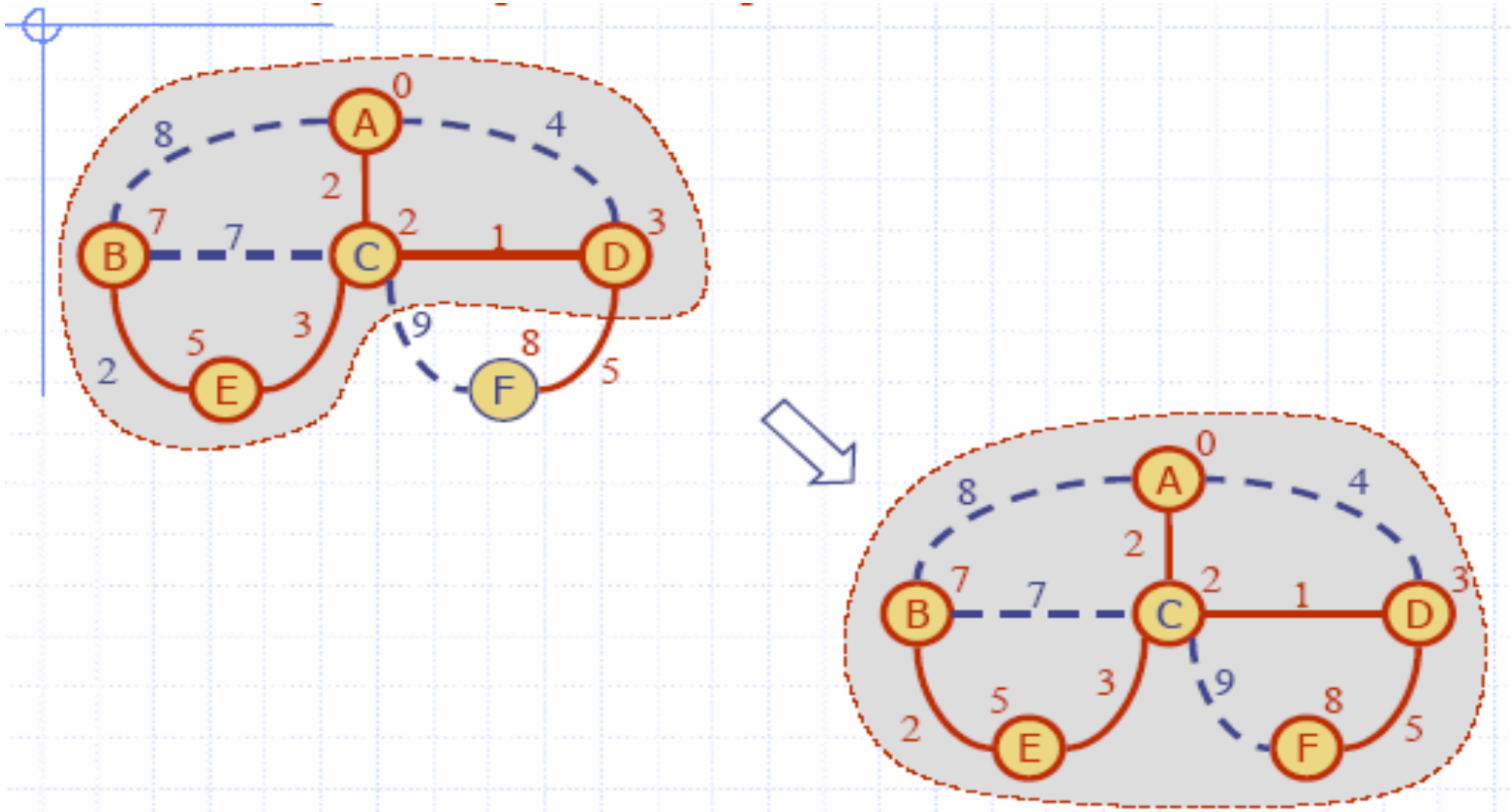
Dijkstrův algoritmus

- Příklad(1)



Dijkstrův algoritmus

- Příklad(2)



Dijkstrův algoritmus

Pseudokód

- V prioritní frontě jsou uložené nenavštívené vrcholy, které nebyly propagovány společně se vzdálenostmi, s kterými jsme se k nim dostali
- V každé iteraci z prioritní fronty vyhodíme vrchol s nejmenší vzdáleností pak provádíme relaxaci
- Vrchol z musí být v Q!

```
Algorithm DijkstraDistances(G, s)
    Q ← new heap-based priority queue
    for all v ∈ G.vertices()
        if v = s
            setDistance(v, 0)
        else
            setDistance(v, ∞)
        l ← Q.insert(getDistance(v), v)
        setLocator(v,l)
    while ¬Q.isEmpty()
        u ← Q.removeMin()
        for all e ∈ G.incidentEdges(u)
            { relax edge e }
            z ← G.opposite(u,e)
            r ← getDistance(u) + weight(e)
            if r < getDistance(z)
                setDistance(z,r)
                Q.replaceKey(getLocator(z),r)
```

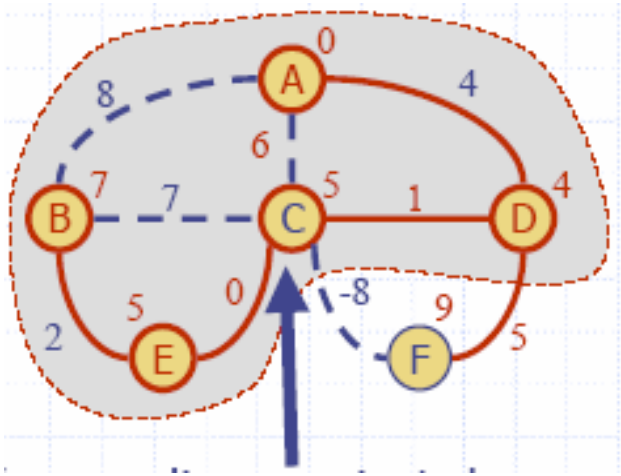
Dijkstrův algoritmus

- Rozšíření
  - jak to provést, abychom mohli vypsat všechny nejkratší cesty?
  - Pro každý vrchol si budeme pamatovat hranu, po které jsme se k němu dostali

```
Algorithm DijkstraShortestPathsTree(G, s)
    …
    for all v ∈ G.vertices()
        …
        setParent(v, ∅)
        …
    for all e ∈ G.incidentEdges(u)
        { relax edge e }
        z ← G.opposite(u,e)
        r ← getDistance(u) + weight(e)
        if r < getDistance(z)
            setDistance(z,r)
            setParent(z,e)
            Q.replaceKey(getLocator(z),r)
```

Dijkstrův algoritmus

- **Složitost**
  - Vkládání a hledání v prioritní frontě: log(n)
  - Projdeme n vrcholů a m hran
  - O((n+m) log(n)) průměrně při dobré implementaci
- **Proč to nefunguje i pro záporně ohodnocené hrany?**
  - Protože algoritmus je Greedy (Hladový)
  - Kdyby se přidal do mraku (hotové vrcholy), mohlo by to narušit vzdálenosti k vrcholům, které už v mraku jsou



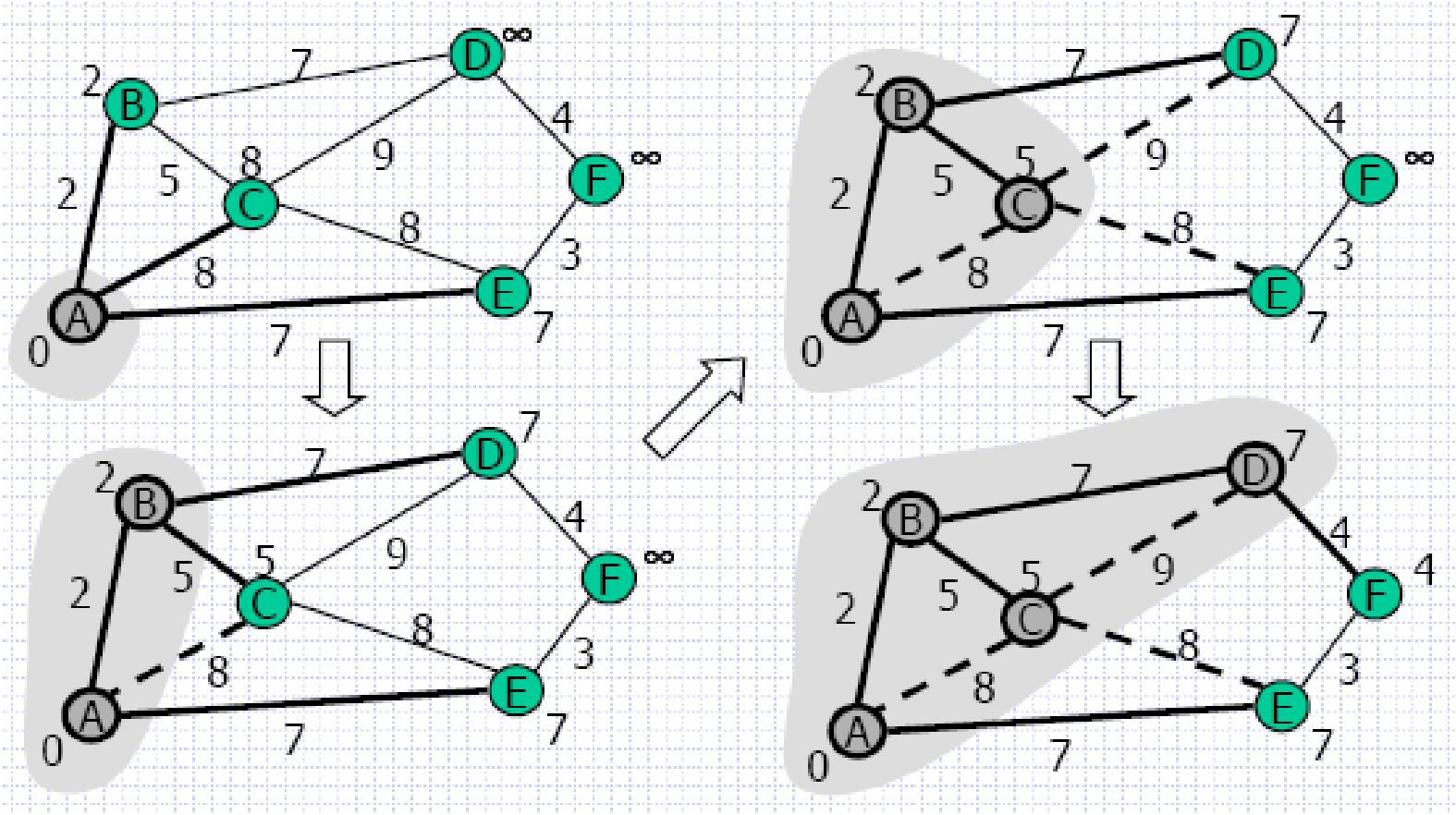
Vzdálenost C od A je 1, ale v mraku už máme hotovou vzdálenost 5

Primův-Jarníkův algoritmus

- Algorimus na hledání minimální kostry grafu (Minimum Spannig Tree)
- Kostra grafu
  - minimální souvislý podgraf obsahující všechny vrcholy
  - Nemá žádný cyklus, je tudíž stromem
- Postup stejný jako u Dijkstrovo algoritmu
- Další algoritmus na hledání min. kostry grafu – Kruskalovo hladový (greedy) algoritmus (podobně jako Primův)

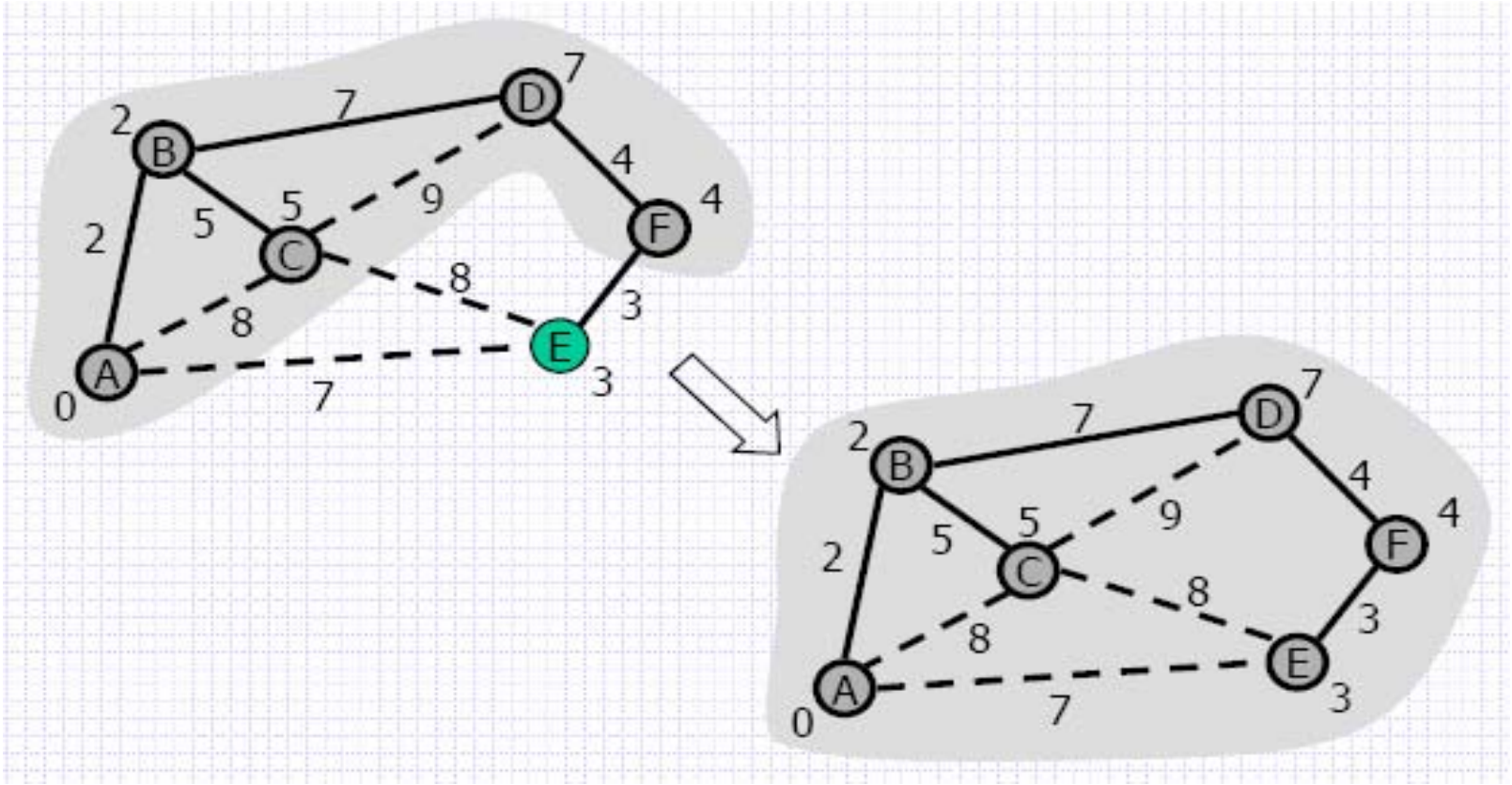
Primův-Jarníkův algoritmus

- Příklad(1)



Primův-Jarníkův algoritmus

- Příklad(2)



Prim-Jarník vs. Dijkstra

```
Algorithm DijkstraShortestPaths(G, s)
    Q ← new heap-based priority queue

    for all v ∈ G.vertices()
        if v = s
            setDistance(v, 0)
        else
            setDistance(v, ∞)
        setParent(v, ∅)
        l ← Q.insert(getDistance(v), v)
        setLocator(v,l)
    while ¬Q.isEmpty()
        u ← Q.removeMin()
        for all e ∈ G.incidentEdges(u)
            z ← G.opposite(u,e)
            r ← getDistance(u) + weight(e)
            if r < getDistance(z)
                setDistance(z,r)
                setParent(z,e)
                Q.replaceKey(getLocator(z),r)
```

```
Algorithm PrimJarnikMST(G)
    Q ← new heap-based priority queue
    s ← a vertex of G
    for all v ∈ G.vertices()
        if v = s
            setDistance(v, 0)
        else
            setDistance(v, ∞)
        setParent(v, ∅)
        l ← Q.insert(getDistance(v), v)
        setLocator(v,l)
    while ¬Q.isEmpty()
        u ← Q.removeMin()
        for all e ∈ G.incidentEdges(u)
            z ← G.opposite(u,e)
            r ← weight(e)
            if r < getDistance(z)
                setDistance(z,r)
                setParent(z,e)
                Q.replaceKey(getLocator(z),r)
```

Hladové (greedy) algoritmy obecně

- Dijkstra, Kruskal, Prim
- Globálního optimálního řešení se dosáhne provedením lokální nejvhodnější operace
  - => v každém kroku výpočtu dochází k seřazení kandidátů podle jejich kvality (prioritní fronta) a použitím nejlepšího
  - => greedy algoritmy nemusí vždy dosahovat optimálního řešení jako Dijkstra, avšak jejich použitím můžeme dojít nějakému, uspokojivému řešení
- Co je to “optimální”?
- Příklad - negrafová úloha – problém batohu (0/1 Knapsack)

A* algoritmus

- Hledání nejkratší cesty stavovým prostorem
- K čemu je dobrý:
  - Situace: Mám neorientovaný graf, hledám cestu A->B.
    - Pokud bych bod B může být od A kdekoliv, tzn. nemám žádnou doplňující informaci o poloze B, budu zvětšovat okruh navštívených vrcholů kolem A podle Dijkstrova algoritmu.
    - Mám aditivní informaci o B, např. B leží na sever od A. V tomto případě je vhodnější použít A* algoritmus

Vlastnosti A*

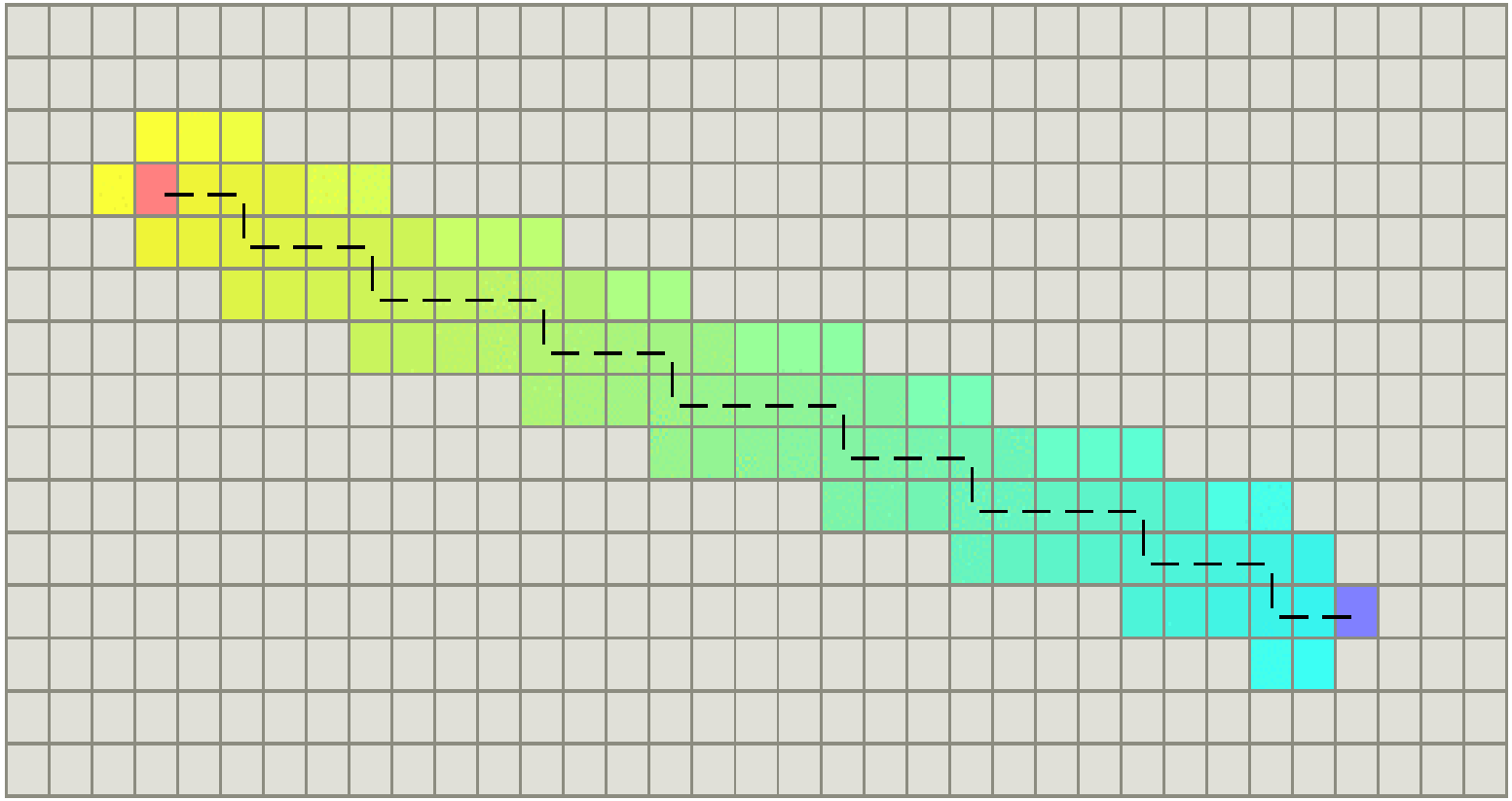
- Každý zpracovávaný uzel je ohodnocen funkcí f = g + h, g je suma všech ohodnocení, kterými jsme doposud prošli, h je heuristická funkce - odhad cesty k finálnímu uzlu (stavu).
- Garantuje nalézt nejkratší cestu z A->B, pokud h nikdy nepřecení ohodnocení zbytku cesty k finál. uzlu.
- Čím více máme relevantní informace o hledaném uzlu, tím rychleji jej dosáhneme.
- OPEN
  - všechny uzly, které byly otevřeny a neprošlo se jimi
  - prioritní fronta
- CLOSED - všechny uzly, kterými se prošlo

A* algoritmus

```
inicializuj seznam OPEN
inicializuj seznam CLOSED
vloz startovní uzel do OPEN (f poloz rovne 0)

while seznam OPEN neni prazdny
    najdi uzel s nejmensi f v OPEN, nazveme ho "q"
    vyndej q z OPEN
    vygeneruj N nasledovniku q a nastav jejich rodice na q
    for kazdyho nasledovnika
        pokud nasledovnik je cil, zastav hledani
        nasledovnik.g = q.g + vzdalenost mezi nasledovnikem a q
        nasledovnik.h = domela vzdalenost od naslednika k cili
        nasledovnik.f = nasledovnik.g + nasledovnik.h

        if uzel na stejne pozici jako je nasledovnik, je v OPEN
            s mensi f nez nasledovnik, preskoc ho
        if uzel na stejne pozici jako je nasledovnik, je v CLOSED
            s mensi f nez nasledovnik, preskoc ho
        jinak vloz uzel do OPEN
    end
    vloz q do CLOSED
end
```

A* algoritmus – příklad negrafový

- h(n) = D * (abs(n.x-goal.x) + abs(n.y-goal.y))
  - Funkce pro odhad délky zbytku cesty



NP-úplné grafové problémy

- TSP (Travelling Salesman Problem)
  - Acyklický, neorientovaný, souvislý graf., naším úkolem je počínaje vrcholem S navštívit všechny vrcholy v grafu právě jednou tak, aby cesta byla co nejkratší, a vrátit se do S
- Hamiltonovská kružnice
  - To samé co TSP, akorát cesta nemusí být nejkratší
- Pokrytí vrcholů grafu max. velikosti X.
  - Výběr vrcholů grafu tak, že se budou dotýkat všech hran z nich vedoucí mají dohromady ohodnocení co nejblíže danému X
- …

NP-úplnost

- P – problémy řešitelné polynom. algoritmy
- NP – nedeterministickými polynomiálními algoritmy
- NP – úplné – NP, které jsou na sebe deterministicky v polynom. čase převeditelné
- Co to znamená slovo NP?
- Co jak prakticky poznáme, že je problém NP?
- P = NP?