

Algoritmy a datové struktury I

Spojovaný seznam, fronta a zásobník

Datová struktura - datový typ, který je definován rozsahem hodnot, kterých může nabývat

Statické datové struktury - datové struktury s pevnou velikostí

Dynamické datové struktury - datové struktury jejichž velikost není před během programu známá, musí být alokována v průběhu programu

Operace nad dynamickými strukturami - operace, kterými lze modifikovat nebo využívat obsah datové struktury

Zřetěžený seznam

Vložení uzlu do zřetěženého seznamu

83 of 176 248,83%

Procedura $\text{INSERT}(L, key)$

vstup: seznam L typu $List$, vkládaný klíč key

výstup: ukazatel na nově přidaný prvek

```
1 new ← NEW (key) // vytvoří nový prvek seznamu
2 new.next ← nil // následující prvek není
3 new.prev ← L.last // předchozí prvek je bývalý poslední prvek
4 if L.first = nil then
5     L.first ← new // případ prázdného seznamu
6 else
7     L.last.next ← new
8 fi
9 L.last ← new // nový poslední prvek
10 return new
```

Odstranění uzlu ze zřetěženého seznamu

Procedura DELETE($L, node$)

vstup: seznam L typu $List$, ukazatel $node$ na prvek seznamu, který chceme odstranit

```
1 if  $node = nil$  then
2     return Chyba, byl zadán prázdný ukazatel
3 fi
4 if  $node.prev = nil$  then
5      $L.first \leftarrow node.next$  // nemá předchůdce
6 else
7      $node.prev.next \leftarrow node.next$ 
8 fi
9 if  $node.next = nil$  then
10     $L.last \leftarrow node.prev$  // nemá následníka
11 else
12     $node.next.prev \leftarrow node.prev$ 
13 fi
14 RELEASE( $node$ ) // až po úpravě ukazatelů uvolňujeme prvek z paměti
```

Zásobník

Push funkce

```
push(stack, key) {
    Element e = new Element();
    e->key = key;
    if(stack->top != NULL)
        e->below = stack->top
    else
        e->below = NULL;
    stack->top = e;
}
```

Pop funkce

```
pop(stack) {
    if(stack->top == NULL)
        return NULL;
    Element e = stack->top->key;
    int key = e->key;
    stack->top = stack->top->below;
    delete(e);
    return key;
}
```

Fronta

Enqueue funkce

```
enqueue(queue, key) {  
    Element e = new Element();  
    e->key = key;  
    if(queue->last == NULL)  
        queue->first = e;  
    else  
        queue->last->left = e;  
    queue->last = e;  
}
```

Dequeue funkce

```
dequeue(queue) {  
    if(queue->first == NULL)  
        return NULL  
    Element e = queue->first;  
    int key = e->key;  
    if(queue->first == queue->last) {  
        queue->first = NULL;  
        queue->last = NULL;  
    } else {  
        queue->first = queue->first->left;  
    }  
    delete(e);  
    return key;  
}
```

Algoritmy a korektnost

Algoritmus - přesný popis toho, jak máme postupovat, abychom po provedení tohoto postupu na vstupních hodnotách dostali kýžený výsledek.

Vstupní podmínka - ze všech možných vstupů do daného algoritmu vymezuje, ty pro které je algoritmus definován

Výstupní podmínka - pro každý vstup daného algoritmu splňující vstupní podmínku určuje, jak má vypadat výsledek odpovídající danému vstupu

Částečně (parciálně) korektní algoritmus - algoritmus, který pro každý vstup, který splňuje *vstupní podmínku* a algoritmus na něm skončí, výstup splňuje *výstupní podmínku*

Úplný (konvergentní) algoritmus - algoritmus, který pro každý vstup splňující vstupní podmínku výpočet skončí

Totálně korektní algoritmus - algoritmus, který je *parciálně korektní* a *konvergentní*

Invarianta cyklu - každá takové tvrzení o algoritmu, které platí před a po vykonání každé iterace cyklu

Matematická indukce - běžný způsob dokazování korektnosti rekurzivního algoritmu.

Nejdříve tvrzení musíme dokázat pro funkci bez zanoření, dále předpokládáme platnost pro obecné zanoření dohloubky m a indukčním krokem musíme potvrdit, že pokud platí pro m , pak platí i pro $m + 1$.

Hladové algoritmy - algoritmy, které vždy volí v danou chvíli nejvýhodnější volbu, aniž by se snažily "myslet do budoucnosti"

Grafy I

Ohodnocená hrana má přiřazenou hodnotu. Pokud je délka kladné celé číslo, pak lze graf s ohodnocenými hranami převést na graf s hranami neohodnocenými (tedy ohodnocenými jedničkou), hranu délky n nahradíme n po sobě jdoucími hranami délky 1. V případě, že jednu dvojici vrcholů spojuje více hran, hovoříme o paralelních hranách (a multigrafu).

Výstupní stupeň vrcholu je počet hran, které z vrcholu vychází.

Transponovaný graf je graf obsahující hrany opačně orientované než graf původní. Má smysl jej definovat pouze pro orientovaný graf. Pro reprezentaci pomocí matice sousednosti je transponování grafu totožné s transponováním matice, která graf popisuje.

Průchody grafu

1. **BFS** (breadth-first search), tedy prohledávání do šířky. Používá datovou strukturu **fronta** (ve které uchováváme vrcholy čekající na zpracování) a hodí se pro hledání **nejkratší cesty** nebo testování, zdali je graf **bipartitní**.
2. **DFS** (depth-first search), tedy prohledávání do hloubky. Používá datovou strukturu **zásobník** (ve kterém ukládáme cestu). Používá se k hledání **cyklů v grafu**, nalezení **topologického uspořádání grafů** nebo rozdělení grafu na **silně souvislé komponenty**. K těmto aplikacím využívá časové známky, které popisují, kdy jsme vrchol objevili (u.d) a kdy jsme jej opustili (u.f).

Průzkum do šířky

Složitost je $O(V + E)$

Atributy vrcholu

v.color

- vrchol má *černou* barvu, když je dosažitelný z iniciálního vrcholu a byl již prozkoumán
- vrchol má *šedivou* barvu, když je dosažitelný z iniciálního vrcholu a byl již objeven, ale nebyl ještě prozkoumán
- vrchol má *bílou* barvu, když není dosažitelný z iniciálního vrcholu nebo nebyl ještě prozkoumán

v.π

- vrchol, ze kterého byl v objeven

v.d

- délka (počet hran) cesty z s do v , na které byl v objeven (= délka nejkratší cesty z s do

v)

Pseudokód

The image is a screenshot of a presentation slide. At the top, there is a navigation bar with the text "Průzkum grafů a grafová souvislost" and "Průzkum do šířky". The slide title is "BFS - kompaktní verze". Below the title, the pseudocode for BFS is presented in a light blue box. The code is as follows:

```
BFS( $G, s$ )  
1 foreach  $u \in V \setminus \{s\}$   
2   do  $u.d \leftarrow \infty$  od  
3  $s.d \leftarrow 0$   
4  $Q \leftarrow \emptyset$   
5 Enqueue( $Q, s$ )  
6 while  $Q \neq \emptyset$  do  
7    $u \leftarrow \text{Dequeue}(Q)$   
8   foreach  $v \in \text{Adj}[u]$  do  
9     if  $v.d = \infty$   
10      then  $v.d \leftarrow u.d + 1$   
11        Enqueue( $Q, v$ ) fi  
12   od  
13 od
```

Nejkratší cesta v BFS

Délka nejkratší cesty z s do v , značíme $\delta(s, v)$, je definována jako minimální počet hran na cestě z s do v . Když neexistuje žádná cesta z s do v , tak $\delta(s, v) = \infty$.

Nejkratší cestou z s do v je každá cesta z s do v která má $\delta(s, v)$ hran.

Věta 1

Nechť $G = (V, E)$ je graf a $s \in V$ jeho vrchol, na které aplikujeme algoritmus BFS. Pak po ukončení výpočtu pro každý vrchol $v \in V$ platí **$v.d = \delta(s, v)$**

BFS strom

- je graf předchůdců definován přes \in atributy
- je kostrou grafu
- pro každý vrchol $v \in \pi$ obsahuje BFS strom jedinou cestu z s do v , která je současně **nejkratší cestou z s do v**

BFS strom s ohodnocenými hranami

Primův algoritmus

- namísto fronty použijeme **prioritní frontu**
- do fronty vkládáme dvojici (vrchol, délka hrany po které byl objeven)
- prioritou je délka hrany
- z fronty vybíráme vždy **nejkratší hranu**
- **BFS strom je nejlevnější kostrou grafu**

Dijkstrův algoritmus

- vrcholu ve frontě aktualizujeme hodnotu $v.d$ pokaždé, když je po nějaké hraně objeven
- prioritou je hodnota $v.d$
- z fronty vybíráme vrchol s nejnižší hodnotou **$v.d$**
- **BFS strom je strom nejkratších cest z s do ostatních vrcholů grafu**

Bipartitní grafy

Bipartitní graf je takový graf, jehož množina vrcholů se dá rozdělit na dvě disjunktní množiny tak, že žádné dva vrcholy patřící do stejné množiny nejsou spojeny hranou. Bipartitní graf neobsahuje cyklus **liché délky**.

Testování bipartitnosti s využitím BFS

- zvolíme libovolný vrchol grafu jako iniciální vrchol s
- BFS průzkum z vrcholu s definuje vrstvy L_0, L_1, L_2, \dots
- do vrstvy L_i patří vrcholy, jejichž vzdálenost od s je i (tj. $v.d = i$)

Žádné dva vrcholy patřící do stejné vrstvy nejsou spojeny hranou

- obarvení vrcholů je určeno vrstvami: vrcholy jejichž vzdálenost od s je sudá mají modrou, vrcholy jejichž vzdálenost je lichá od s mají červenou
- korektnost obarvení plyne z předpokladu o neexistenci hrany mezi vrcholy ze stejné vrstvy

Existují dva vrcholy spojeny hranou a patřící do stejné vrstvy

- nechť u, v jsou vrcholy takové, že $u, v \in L_i$ a $\{u, v\} \in E$
- nechť y je nejmenší společný předchůdce vrcholů u a v v BFS stromu
- cesta z y do u , hrana $\{u, v\}$ a cesta z v do y tvoří cyklus, jehož délka je lichá, protože cesta z y do u a cesta z v do y mají stejnou délku
- graf není bipartitní

Průzkum do hloubky

Složitost je $O(V + E)$

Atributy vrcholu

- **v.d** - první návštěva vrcholu
- **v.f** - ukončení průzkumu vrcholu

Pseudokód

Průzkum grafů a grafová souvislost Průzkum do hloubky

Průzkum do hloubky - iterativní implementace

DFS_Iterative_Visit(G, u)

```
1  $S \leftarrow \emptyset$ 
2  $S.push(u)$ 
3  $time \leftarrow time + 1$ ;  $u.d \leftarrow time$ 
4  $u.color \leftarrow gray$ 
5 while  $S \neq \emptyset$  do
6    $u \leftarrow S.pop()$ 
7   if existuje hrana  $(u, v)$  taková, že  $v.color = white$ 
8     then  $S.push(u)$ 
9          $S.push(v)$ 
10         $v.color \leftarrow gray$ 
11         $v.\pi \leftarrow u$ 
12         $time \leftarrow time + 1$ ;  $v.d \leftarrow time$ 
13   else  $u.color \leftarrow black$ 
14    $time \leftarrow time + 1$ ;  $u.f \leftarrow time$  fi od
```

0 2. května 2016 31 / 80

Časové značky

- DFS přiřadí vrcholům grafu časové značky
- obsahují informace o struktuře grafu a DFS stromu

Platí

- pro všechny vrcholy u platí $u.v < u.f$
- s každým vrcholem u je asociován interval $[u.d, u.f]$
- má-li vrchol a časové známky v intervalu časových známek vrcholu b , pak existuje cesta z b do a , **opačné tvrzení neplatí**

Uspořádání vrcholů

- **preorder** - uspořádání podle značky $.d$ v rostoucím pořadí
- **postorder** - uspořádání podle značky $.f$ v rostoucím pořadí
- **reverse postorder** - uspořádání podle značky $.f$ v klesajícím pořadí

Pro každé dva vrcholy u, v platí jedna z podmínek

1. intervaly $[u.d, u.f]$ a $[v.d, v.f]$ jsou **disjunktní**
 - u není následníkem v v DFS stromu
 - v není následníkem u v DFS stromu
2. interval $[u.d, u.f]$ je celý obsažen v intervalu $[v.d, v.f]$
 - u je následníkem v v DFS stromu
3. interval $[v.d, v.f]$ je celý obsažen v intervalu $[u.d, u.f]$
 - v je následníkem u v DFS stromu

Vrchol v je dosažitelný z vrcholu u v DFS stromu grafu G právě když **$u.d < v.d < v.f < u.f$**

Vlastnost bílé cesty - v DFS stromu grafu G je vrchol v následníkem vrcholu u právě když v čase $u.d$ existuje cesta z u do v obsahující jen bílé vrcholy

Klasifikace hran

stromová hrana

- hrana (u, v) obsažená v lese
- při průzkumu je vrchol v bílý
- $u.d < v.d < v.f < u.f$

zpětná hrana

- hrana (u, v) která spojuje vrchol u s jeho předchůdcem v v DFS stromu
- při průzkumu hrany je vrchol v šedivý
- $v.d < u.d < u.f < v.f$

dopředná hrana

- hrana (u, v) , která nepatří do DFS stromu a která spojuje vrchol u s jeho následníkem v DFS stromu
- vrchol který s její pomocí objevily už byl objeven dříve
- při průzkumu hrany je vrchol v černý
- $u.d < v.d < v.f < u.f$

příčná hrana

- všechny ostatní hrany
- při průzkumu hrany je vrchol v černý
- $v.d < v.f < u.d < u.f$

Všechny hrany v neorientovaném grafu jsou buď stromové nebo zpětné.

Topologické uspořádání

Topologické uspořádání vrcholů v orientovaném acyklickém grafu je takové očíslování vrcholů čísly 1 až n (n je počet vrcholů grafu), že každá hrana vede z vrcholu s nižším číslem do vrcholu v vyšším číslem.

Lemma

Orientovaný graf G má topologické uspořádání právě když je **acyklický**.

Lemma

Orientovaný graf G je acyklický právě když DFS průzkum grafu neoznačí žádnou hranu jako zpětnou.

Topologické uspořádání naivní algoritmus

- najdeme vrchol v do kterého nevstupuje žádná hrana
- v vložíme do seznamu, který určuje uspořádání
- odstraníme v a všechny jeho hrany z grafu
- a pokračujeme prvním bodem, tj. najdeme vrchol do kterého nevstupuje žádná hrana...
- složitost $O(VE)^*$

Topologické uspořádání algoritmus

- aplikuj **DFS** na graf G
- když průzkum označí nějakou hranu jako **zpětnou**, tak graf G nemá topologické uspořádání
- v opačném případě vypiš vrcholy v uspořádání **reverse postorder**, tj. podle značky $.f$ v klesajícím pořadí
- neobjevy všechny cykly

Souvislost v grafu

Souvislost v orientovaném grafu

- vrchol v je **dosažitelný** z vrcholu u , načítáme $u \rightsquigarrow v$, právě když v G existuje orientovaná cesta z u do v
- $Reach(u)$ je množina **všech** vrcholů dosažitelných z u
- vrcholy u a v jsou **silně dosažitelné**, právě když je u dosažitelné z v a současně je v dosažitelné z u
- **silně souvislá komponenta** je maximální množina vrcholů $C \subseteq V$ taková, že pro každé $u, v \in C$ platí $u \rightsquigarrow v$ a současně $v \rightsquigarrow u$

Souvislost v neorientovaném grafu

- pojmy *dosažitelnost* a *silná dosažitelnost* jsou **totožné**
- pro hledání silně souvislé komponenty v grafu můžeme použít **BFS nebo DFS**
- jednotlivé BFS (DFS) stromy **korespondují s komponentami souvislosti**
- složitost je **$O(V+E)$**

Silně souvislé komponenty obsahující daný vrchol u

- najdi množinu $Reach(u)$ všech vrcholů **dosažitelných z u** aplikací $DFS_Visit(G, u)$
- najdi množinu $1/Reach(u)$ všech vrcholů, **ze kterých je dosažitelný u**
- pro výpočet $1/Reach(u)$ využijeme transponovaný graf $rev(G)$, na který aplikujeme $DFS_Visit(rev(G), u)$

- $rev(G)$ je graf G s obrácenou orientací hran
- silně souvislá komponenta obsahující u je průnikem $Reach(u)$ a $1/u(Reach)$
- časová složitost je $O(V+E)$

Silně souvislé komponenty v orientovaném grafu

komponentový graf (graf silně souvislých komponent) je orientovaný graf, který vznikne kontrakcí každé silně souvislé komponenty grafu do jednoho vrcholu a kontrakcí paralelních hran do jedné hrany, značíme $scc(G)$

- $scc(g)$ je orientovaný acyklický graf
- **listová komponenta** = list grafu $scc(G)$
- **kořenová komponenta** = kořen grafu $scc(G)$
- platí $rev(scc(G)) = scc(rev(G))$

Věta

Vrchol s nejvyšší časovou značkou $.f$ leží v kořenové komponentě.

Věta

Nechť C a D jsou silně souvislé komponenty takové, že z C vede hrana do D . Potom největší hodnota $.f$ v komponentě C je větší než největší hodnota $.f$ v komponentě D .

Algoritmus pro detekci silně souvislých komponent

- použijeme **DFS** na G s uložením časových známek
- vypočteme **transponovaný graf T** , který vznikne otočením směru šipek
- vrcholy grafu uspořádej v obráceném pořadí, tj. podle značky $.f$ v klesajícím pořadí
- proved' DFS průzkum z vrcholů v daném pořadí
- vrcholy každého **DFS stromu** vzniklé při aplikaci DFS na graf T tvoří samostatné silně souvislé komponenty

Cykly

Nalezení cyklu v orientovaném grafu

- použijeme **DFS**
- odpovídá nalezení **zpětné hrany**
- použijeme časové známky k nalezení zpětné hrany

Druhý způsob

- použijeme **DFS** s poznamenáváním aktuální procházené větve
- použijeme značku, která říká, že vrchol je procházen
- když vrchol začneme procházet nastavíme značku na *true*, po prozkoumání nastavíme značku na *false*
- graf obsahuje cyklus, když při prozkoumávání narazíme na vrchol, který má značku nastavenou na *true*

Nalezení nejkratšího cyklu

- dle předchozího najdu zpětné hrany
- pro každou zpětnou hranu (u, v) najdu nejkratší zbytek cyklu, tj. nejkratší cestu z v do u

- pro nalezení nejkratší cesty použijí BFS z v

Souvislý neorientovaný graf

Z každého souvislého neorientovaného grafu lze odebrat jeden vrchol, tak že nedojde k rozpojení grafu na samostatné části.

Důkaz

- k řešení použijeme DFS průchod
- tento průchod nám vytvoří DFS strom
- libovolný list DFS stromu lze odstranit
- obecně platí že libovolný list lze odstranit z libovolné kostry grafu, tak aby byl graf stále souvislý

U orientovaných grafů toto neplatí! Protipříklad: kružnice na 3 vrcholech

Grafy II

Cesta v grafu $G = (V, E)$ je posloupnost vrcholů $p = v_0, v_1, \dots, v_k$ taková, že $(v_{i-1}, v_i) \in E$ pro $i = 1, \dots, k$.

Jednoduchá cesta je cesta, která neobsahuje dva stejné vrcholy.

Relaxace cesty je procedura volaná na dvojici vrcholů, která v případě existence kratší cesty, než kterou zatím známe, aktualizuje vzdálenost vrcholu na novou kratší hodnotu.

Strom nejkratších cest grafu G definujeme jako strom, kde od fixního kořene v k libovolnému vrcholu u je cesta ve stromě nejkratší cestou v grafu G .

Bellman-Fordův algoritmus

- hledá nejkratší cestu ze zadaného vrcholu do všech ostatních vrcholů
- pracuje s hranami **záporné délky**
- **detekuje záporný cyklus**
- využívá frontu
- celková složitost je $O(V \cdot E)$

Pseudokód

Bellman-Ford(G, w, s)

```
1 INIT_SSSP( $G, s$ )
2 for  $i = 1$  to  $|V| - 1$  do
3   foreach  $(u, v) \in E$  do
4     if  $v.d > u.d + w(u, v)$  then RELAX( $u, v, w$ ) fi
5   od
6 od
7 foreach  $(u, v) \in E$  do
8   if  $v.d > u.d + w(u, v)$  then return FALSE fi
9 od
10 return TRUE
```

Nejkratší cesty v orientovaném acyklickém grafu

- nalezneme topologické uspořádání grafu
- relaxujeme hrany v grafu v pořadí, které respektuje topologické uspořádání

```
Dag( $G, w, s$ ) {
  najdi topologické uspořádání vrcholů grafu  $G$ 
  Init_SSSP( $G, s$ );
  foreach vrchol  $*u*$  v topologickém uspořádání do
    foreach  $(u, v) \in E$  do
      if( $v.d > u.d + w(u, v)$ ) then Realx( $u, v, w$ )
    od
  od
}
```

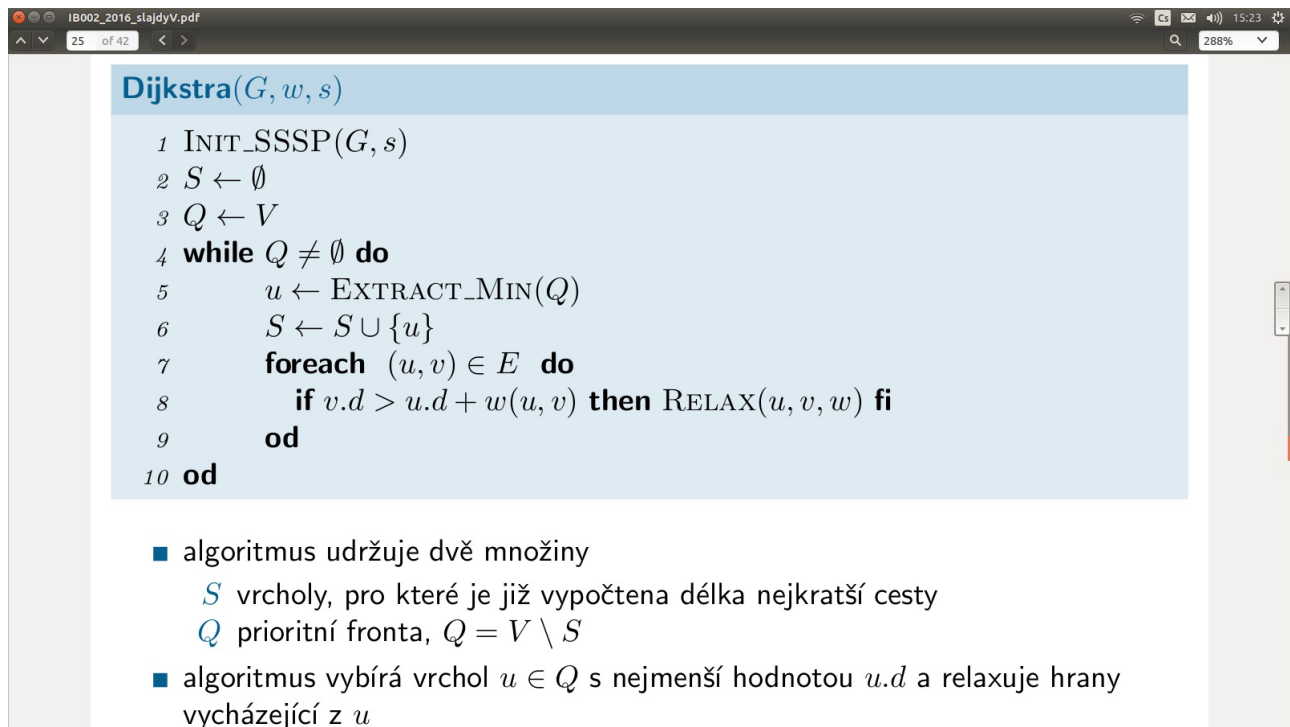
- časová složitost $\Theta(V + E)$
- topologické uspořádání garantuje, že hrany *každé* cesty jsou relaxované v pořadí, v jakém se vyskytují na cestě

Dijkstrův algoritmus

- hledá nejkratší cestu ze zadaného vrcholu do všech ostatních vrcholů
- **neumí projít** graf s hranami záporné délky
- **lepší** časová složitost než Bellman-Fordův algoritmus
- využívá **prioritní frontu**, kde priorita vrcholu je značena $v.d$
- na dijkstrův algoritmus můžeme nahlížet i jako na efektivní implementaci prohledávání grafu do šířky, na rozdíl od BFS neukládáme vrcholy, které mají být prozkoumané, do fronty, ale do prioritní fronty
- složitost závisí od způsobu implementace prioritní fronty Q : pole $\Theta(V^2)$, binární halda $\Theta(V \log V + E \log V)$, Fibonacciho halda $\Theta(V \log V + E)$

Pokud v algoritmu chceme změnit minimum na **maximum**, tj. algoritmus bude hledat nejdelší cestu mezi dvěma body, tak to nelze.

Pseudokód



Dijkstra(G, w, s)

```
1 INIT_SSSP( $G, s$ )
2  $S \leftarrow \emptyset$ 
3  $Q \leftarrow V$ 
4 while  $Q \neq \emptyset$  do
5      $u \leftarrow \text{EXTRACT\_MIN}(Q)$ 
6      $S \leftarrow S \cup \{u\}$ 
7     foreach  $(u, v) \in E$  do
8         if  $v.d > u.d + w(u, v)$  then RELAX( $u, v, w$ ) fi
9     od
10 od
```

- algoritmus udržuje dvě množiny
 - S vrcholy, pro které je již vypočtena délka nejkratší cesty
 - Q prioritní fronta, $Q = V \setminus S$
- algoritmus vybírá vrchol $u \in Q$ s nejmenší hodnotou $u.d$ a relaxuje hrany vycházející z u

Optimalizace Dijkstrova algoritmu pro hledání nejkratší cesty mezi dvěma vrcholy s a t

- **Optimalizace 1 - výpočet ukončíme když vrchol t odebereme z prioritní fronty**
- **Optimalizace 2 - dvousměrné hledání**
- současně spouštíme dopředný výpočet Dijkstrova algoritmu z vrcholu s a zpětný výpočet z vrcholu t , vždy jednou iterací každého výpočtu
- dopředný výpočet používá frontu Q_f a přiřazuje vrcholům hodnoty $.d_f$ a $.p_f$, zpětný frontu Q_b a přiřazuje hodnoty $.d_b$ a $.p_b$
- výpočet ukončíme když nějaký vrchol w je odstraněn z obou front Q_f a Q_b
- po ukončení najdeme vrchol x s minimální hodnotou $x.d_f + x.d_b$
- pomocí atributů počítaných v dopředném výpočtu najdeme nejkratší cestu z s do x a pomocí atributů z zpětného výpočtu najdeme nejkratší cestu z x do t
- **Optimalizace 3 - heuristika**
- jestliže jsou dva vrcholy stejně daleko od s , chceme preferovat ten, který je blíže t
- pro odhad preferencí používáme ohodnocení vrcholů - potenciál, což je nějaká funkce
- dijkstrův algoritmus pak při výběru vrcholů z prioritní fronty vybírá vrchol s nejnižší hodnotou $v.d + h(v)$

Rank prvku

- množina A obsahující n vzájemně různých čísel
- číslo $x \in A$ má rank i právě když v A existuje přesně $i - 1$ čísel menších než x
- ke každému uzlu x přidáme atribut $x.size$ - počet vnitřních uzlů v podstromě s kořenem x , včetně x
- **$x.size = x.lef.size + x.right.size + 1$**

Vyhledání klíče s daným rankem

- začínáme od kořene x
- hledáme uzel s rankem i
- z definice atributu `.size` plyne, že počet uzlů v levém podstromu uzlu x navýšen o 1 (= r) je přesně rank klíče uloženého v x v podstromě s kořenem x
- když $i = r$, tak x je hledaný uzel
- když $i < r$, tak i -ty nejmenší klíč se nachází v levém podstromě uzlu x a je i -tým nejmenším klíčem v tomto podstromě
- když $i > r$, tak i -ty nejmenší klíč se nachází v pravém podstromě uzlu x a jeho pořadí v tomto podstromě je i snížení o počet uzlů levého podstromu
- složitost $O(\log n)$

```
findNodeWithRank(x, i) {
    r = x.left.size + 1;
    if (i == r){
        return x;
    } else {
        if(i < r)
            return findNodeWithRank(x.left, i);
        else
            return findNodeWithRank(x.right, i - r);
    }
}
```

Určení ranku daného prvku

- přidáme $x.\text{left.size}$
- sledujeme cestu od x do kořene
- jestliže uzel na cestě je levým synem, neměníme rank
- jestliže uzel na cestě je pravým synem, tak k rank přičtem jeho levý podstrom + 1

```
getRankOfNode(T, x){
    r = x.left.size + 1;
    y = x;
    while(y != T.root){
        if(y == y.p.right)
            r = r + y.p.left.size + 1;
        y = y.p;
    }
    return r;
}
```

Hašování

Hašovací funkce

Hašovací funkce musí být v ideálním případě:

- **jednoduchá uniformní**, tj. že pro každý prvek univerza je pravděpodobnost jeho zahašování na kterýkoliv index tabulky stejná
- **load factor** pro danou tabulku s m pozicemi, ve které je uložených n prvků,

definujeme faktor naplnění α předpisem $\alpha = n/m$

- v hašovací tabulce, ve které jsou kolize řešeny zřetězením a ve kterém se používá jednoduchá uniformní funkce, má operace úspěšného/neúspěšného vyhledávání prvku průměrnou časovou složitost **$O(1 + \alpha)$**

Řazení

- **stabilní řazení** zachovává vzájemné pořadí položek se stejným klíčem
- **in situ** algoritmy nepotřebují extrasekvenční prostor

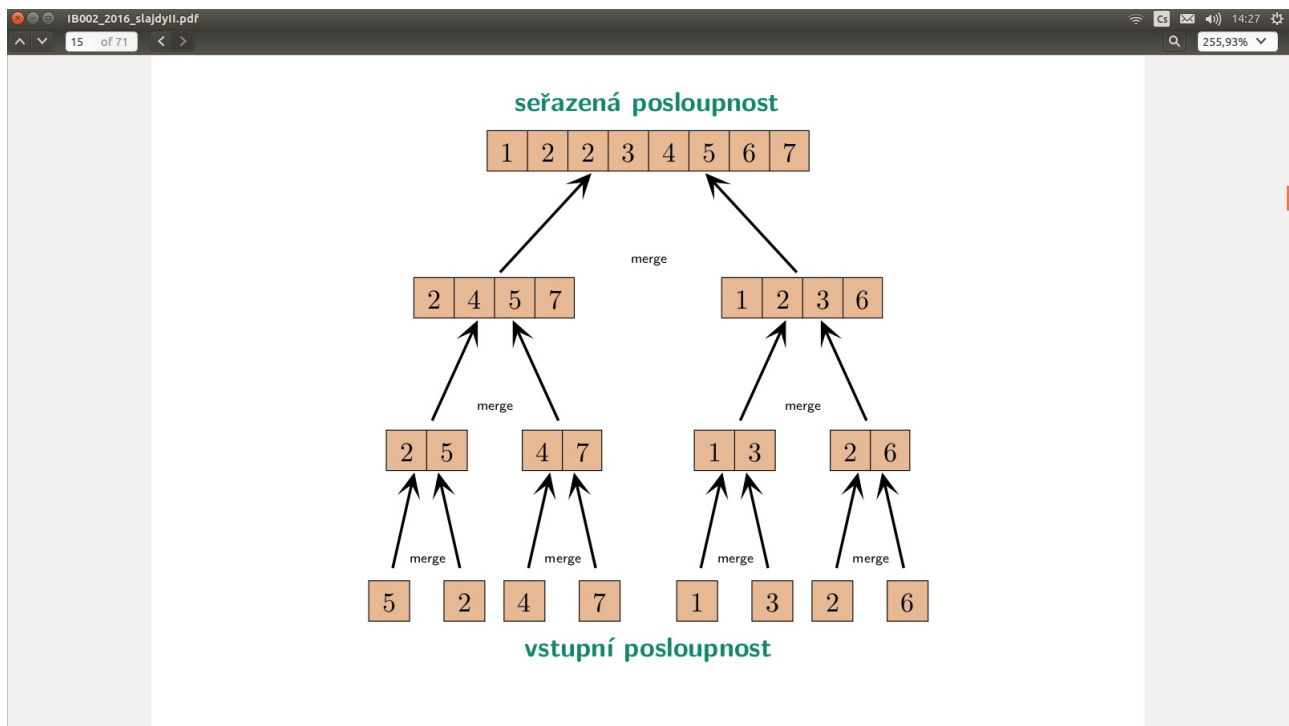
algoritmus	nejhorší případ	průměr
řazení vkládáním	$\Theta(n^2)$	$\Theta(n^2)$
řazení výběrem	$\Theta(n^2)$	$\Theta(n^2)$
řazení sléváním	$\Theta(n \log n)$	$\Theta(n \log n)$
řazení haldou	$\Theta(n \log n)$	$\Theta(n \log n)$
řazení rozdělováním	$\Theta(n^2)$	$\Theta(n \log n)$
řazení počítáním	$\Theta(k + n)$	$\Theta(k + n)$
číslicové řazení	$\Theta(d(n + k))$	$\Theta(d(n + k))$
přihrádkové řazení	$\Theta(n^2)$	$\Theta(n)$

Merge sort - řazení sléváním

- při slévání porovnáváme vedoucí prvky obou posloupností (ty jsou již seřazené)
- menší z porovnávaných prvků přesuneme do výsledné posloupnosti a zarážku v daném poli zvýšíme o 1
- po provedení je pole seřazeno
- složitost je **$\Theta(n \log n)$, $\Theta(n \log n)$**
- není in situ
- je stabilní

Pseudokód - Procedure Merge Sort

```
procedureMergeSort(A, p, r) {  
    if(p < r) {  
        q = (p + r) / 2;  
        procedureMergeSort(A, p, q);  
        procedureMergeSort(A, q + 1, r);  
        mergeSort(A, p, q, r);  
    }  
}
```



Příklady algoritmů aplikované na řešení problémů

Nejkratší cesta z jednoho vrcholu do všech ostatních vrcholů.

- je zobecněním hledání nejkratší cesty mezi dvěma vrcholy
- Dijkstrův nebo Bellmanův-Fordův algoritmus

Nejkratší cesta, která přechází přes konkrétní vrcholy v daném pořadí

- problém si rozbijeme na podproblémy
- pokud hledáme např. cestu z a do b přes c tak řešíme nejkratší cestu z a do c (na grafu bez b) a potom nejkratší cestu z c do b (na grafu bez a)
- výsledky spojíme

Nejkratší cesta ze všech vrcholů do jednoho vrcholu

- z cílu se stane počátek cesty
- otočí se orientace hran
- problém redukuje na podproblém hledání nejkratších cest z jednoho vrcholu do všech ostatních

Nalezení vrcholů jen do určité vzdálenosti

- využijeme Dijkstrův algoritmus
- hledání provádíme do té doby, dokud bude prioritní fronta obsahovat vrcholy v zadané vzdálenosti
- jakmile překročí vzdálenost, algoritmus ukončíme
- nalezené vrcholy jsou v zadaném okolí

Nalezení nejdelší cesty v acyklickém grafu

- použijeme algoritmus pro hledání nejkratší cesty, který umí pracovat se záporně ohodnocenými hranami
- záporné cykly nás neohroží, máme acyklický graf

- ohodnocení hran převedem na negaci, např. 7 -> -7
- necháme algoritmus hledat nejkratší cestu
- výstup znegujeme zpět

Nejkratší cesty mezi všemi dvojicemi vrcholů

- lze provést V-krát hledání nejkratších cest do všech vrcholů z jednoho vrcholu
- k tomu použijeme Bellman-Fordův algoritmus

Nejkratší cyklus přes všechny vrcholy

- známo jako problém obchodního cestujícího
- patří do třídy NP-těžkých problémů
- nejjednodušší řešení je vyzkoušet všechny možné cesty

Modifikujte Dijkstrův algoritmus tak aby hledal nejkratší cestu z více výchozích vrcholů do všech ostatních vrcholů v orientovaném grafu

- do grafu přidáme nový vrchol, který bude nový iniciální
- z něj zavedeme hrany délky 0 do všech výchozích vrcholů
- nakonec hledáme nejkratší cestu z nového vrcholu do všech ostatních vrcholů

Algoritmus k nalezení nejkratší cesty, která je rostoucí

- upravíme graf, tak že si ve vrcholu pamatují i příchozí hranu
- podle hodnoty příchozí hrany pak umažu výchozí hrany, jejichž ohodnocení je menší nebo rovno ohodnocení příchozí hrany

Odstranění duplicit z pole v čase $O(n \log n)$

```
eraseDup(a, n) {
    sort(a); // seřadíme pole v čase  $O(n \log n)$ , třeba merge sort
    index = 1;
    for(i = 2 to n) {
        if(a[index] != a[i]){
            index++;
            swap(a[index], a[i]);
        }
    }
    // duplicity zůstanou za indexem pole
    return a[1,...,index];
}
```

Algoritmus pro nalezení minimální kostry grafu

- všechny hrany si seřadíme podle velikosti vzestupně
- hranu s nejmenší váhou použijeme jako první hranu kostry
- pokud jsme tím už vytvořili kostru, tj. graf měl jen dva vrcholy, končíme. V opačném případě vezmeme hranu s druhou nejmenší váhou
- pokud by vznikla kružnice (cyklus), hranu nepoužijeme
- opakujeme poslední krok, dokud vznikající kostra nespojí všechny vrcholy grafu

Algoritmus pro nalezení maximální kostry

- využijeme algoritmus pro nalezení minimální kostry
- změníme ohodnocení hran z kladných na záporné a použijeme algoritmus pro minimální kostru
- druhou možností je upravit hladový algoritmus, tak že hrany seřadíme sestupně podle délky hrany

Složitost

O notace

$f(n) = O(g(n))$ právě tehdy když existují kladné konstanty n_0 a c takové, že pro všechny $n > n_0$ platí $f(n) < cg(n)$

Příklady

- $8n^2 - 88n + 888 = O(n^2)$, protože $8n^2 - 88n + 888 < 8n^2$ pro všechna $n > 11$
- $8n^2 - 88n + 888 = O(n^3)$, protože $8n^2 - 88n + 888 < 1n^2$ pro všechna $n > 10$
- $8n^2 - 88n + 888 \neq O(n)$, protože $cn < 8n^2 - 88n + 888$ pro $n > c$

Kuchařková metoda

- a, b a d jsou konstanty
- $T(n) = aT(n/b) + \Theta(n^c)$

Potom **platí**

- $\Theta(n^c)$, když $a < b^c$
- $\Theta((n^c) \log n^*)$, když $a = b^c$
- $\Theta(n^{\log_b(a)})$, když $a > b^c$

Binární stromy

- **binární strom** obsahuje uzly které mají nejvýše 2 syny
- **plný binární strom** každý vnitřní uzel má 2 syny
- **vyvážený binární strom** hloubka podstromu se od sebe liší maximálně o 1
- **úplný binární strom** vyvážený binární strom plněný zleva, jeden poslední vnitřní uzel nemusí být stupně k

Halda

- max heapify - garantuje vlastnost haldy, $O(\log n)$
- build max heap - vybuduje z pole haldu, $\Theta(n)$
- heapsort - seřadí prvky pole, $O(n \log n)$