Hranově a uzlově ohodnocenØ grafy

Def: Nechť V, H_V, H_E jsou množiny, $E \subseteq V^2$, $h_V : V \to H_V$, $h_E : E \to H_E$. Čtveřici (V, E, h_V, h_E) nazývÆme *uzlově a hranově ohodnocený orientovaný graf*.

VynechÆním hranovØho ohodnocenÍ h_E dostaneme uzlově ohodnocený orientovaný graf (V, E, h_V).

VynechÆním uzlovØho ohodnocenÍ h_V dostaneme hranově ohodnocený orientovaný graf (V, E, h_E).

Nepřipustíme-li v množině E dvojice tvaru (x, x), mÆme (ohodnocenØ) orientovanØ grafy bez smyček.

Def: Nechť V, H_V, H_E jsou množiny, E je nějakÁ množina dvouprvkových a jednoprvkových podmnožin množiny V, $h_V : V \to H_V$, $h_E : E \to H_E$. Čtveřici (V, E, h_V, h_E) nazývÁme *uzlově a hranově ohodnocený neorientovaný graf*.

VynechÁním hranovØho ohodnocení h_E dostaneme uzlově ohodnocený neorientovaný graf (V, E, h_V).

VynechÁním uzlovØho ohodnocení h_V dostaneme hranově ohodnocený neorientovaný graf (V, E, h_E).

Nepřipustíme-li v množině E jednoprvkovØ množiny, dostÁvÁme (ohodnocenØ) neorientovanØ grafy bez smyček.

PoznÆmka: Obecnější definice grafu připouští, aby množina hran E nebyla podmnožinou V^2 (resp. množiny všech dvouprvkových a jednoprvkových podmnožin množiny V), ale aby to byla libovolnÆ konečnÆ množina, a je navíc dÆno zobrazení $\iota : E \to V^2$, kterØ určuje, mezi kterými uzly danÆ hrana vede. Není-li zobrazení ι injektivní, hovoří se o tzv. *multigrafu*.

Implementace

Graf lze reprezentovat *maticí sousednosti*: neohodnocený graf $G = (V, E)$ na n uzlech je reprezentovÆn čtvercovou maticí $\bar{G} = [g_{i,j}]_{1 \leq i \leq n, 1 \leq j \leq n}$ nul a jedniček tak, že $g_{i,j} = 1$, prÆvě když $(i, j) \in E$.

Je-li graf hranově ohodnocený, $G = (V, E, h_E)$, $h_E : E \to H_E$, jsou v matici sousednosti přímo hodnoty hran: pro $(i, j) \in E$ je $g_{i,j} = h(i, j)$, pro $(i, j) \notin E$ je $g_{i,j} = v$, kde $v \notin H_E$.

Pro tzv. *řídkØ grafy*, tj. grafy, v nichž počet hran je v $o(n^2)$, je výhodnÆ reprezentace pomocí *množin sousedů*: neohodnocený graf $G = (V, E)$ na n uzlech je reprezentovÆn posloupností $[s_1, \ldots, s_n]$ množin nÆsledníků každØho uzlu. Jsou-li $v_1, \ldots, v_k$ všichni nÆsledníci uzlu u (tj. uzly, do nichž vede z u hrana), je $s_u = \{v_1, \ldots, v_k\}$.

Nejčastější reprezentace posloupnosti množin sousedů je polem seznamů. Je-li graf uzlově ohodnocený, obsahuje i-tý prvek pole $\bar{G}$ dvojici $(s_i, h_V(i))$. Je-li graf hranově ohodnocený, obsahuje j-tý prvek i-tØho seznamu s_i dvojici $(v_j, h_E(i, v_j))$.

ProchÆzení grafu

ProblØm: Projít systematicky všechny uzly grafu.

ProchÆzení grafu do hloubky

Algoritmus `dfs` (*depth-first search*)

$G = (V, E), \quad V = \{1, \ldots, n\}, \quad W \subseteq V$

Budeme označovat všechny navštívenØ uzly pomocí procedury`mark`.

Na začÆtku jsou všechny uzly nenavštívenØm(`marked`$(u) =$ `False`).

Pro každý uzel u je `Successors`(u) seznam uzlů – nÆsledníků uzlu$u$.

VychÆzíme z uzlů z množiny$W$ a postupujeme stÆle dÆl po hranÆch do dosud nenavštívených uzlů.

Cesty z označených do nově navštívených uzlů tvoří les s kořeny ve W.

ProchÆzení grafu do hloubky – imperativní pseudokód

```
procedure dfs (G:Graph; var W:Nodeset; var P:Nodearray);
  procedure dfs1 (u:Node);
   begin
     if not marked (u)
        then begin mark (u);
                   for v ∈ Successors (u) do
                     begin P[v] := u;
                           dfs1 (v)
                     end
             end
   end;
```

```
begin {dfs}
 for u ∈ V do begin unmark (u);
                    P[u] := empty
                end;
 for u ∈ W do
   if not marked (u) then dfs1 (u)
end;
```

Chceme-li projít do hloubky *všechny* uzly grafu $G = (V, E)$, položíme $W = V$.

Věta: Algoritmus `dfs` navštíví všechny uzly grafu dosažitelnØ z uzlů z množiny W.

Důkaz je zřejmý a plyne z toho, že se opakovaně označují všechny uzly, kterØ dosud označeny nebyly. Množina neoznačených uzlů se zmenšuje, takže algoritmus konverguje.

PoznÆmka: Algoritmus funguje stejně pro orientovanØ i neorientovanØ grafy.

PoznÆmka: Algoritmus obecně není deterministický, protože nemusí být (a nebývÆ) specifikovÆno pořadí uzlů v množinÆch `Successors(u)` ani v množině W. ProchÆzení grafu do hloubky tedy neurčuje pořadí navštívených uzlů jednoznačně.

Cvičení: Nechť $G = (V, E)$ je neorientovaný graf, $V = \{1, 2, 3, 4, 5, 6\}$, $E = \{\{1, 2\}, \{2, 3\}, \{4, 5\}, \{5, 6\}, \{1, 4\}, \{2, 5\}, \{3, 6\}\}$, $W = \{1\}$. JakÆ jsou všechna možnÆ pořadí navštívených uzlů při volÆní `dfs(`G`,`W`)`?

PoznÆmka: Významnou praktickou aplikací algoritmu `dfs` je nalezení tzv. *silně souvislých komponent* orientovanØho grafu.

Procházení grafu do šířky

Řešíme problém Projít systematicky všechny uzly grafu $G = (V, E)$ dosažitelné (orientovanou) cestou z některého z počátečních uzlů zadaných množinou $W \subseteq V$.

Algoritmus `bfs` (*breadth-first search*) prochází uzly v jiném pořadí než při prohledávání do hloubky: uzly, které leží blíže počátečním uzlům (uzlům, v nichž procházení začíná), jsou navštíveny dříve než uzly ležící dále od počátečních uzlů.

Pomocné datové struktury: fronta `Q` uzlů, les nejkratších cest (uložený v poli `P`)
Navštívené uzly se opět označují pomocí procedury `mark`,
uzel u bude označen, právě když $\texttt{marked}(u) = \texttt{True}$.

ProchÆzení grafu do šíky – imperativní pseudokód

```
procedure bfs (G:Graph; var W:Nodeset; var P:Nodearray);
  var Q:Queue;
  procedure bfs1 (q:Queue);
    begin while not (isempty(q)) do
            begin u := headq(q);
                  dequeue(q);
                  for v ∈ Successors (u) do
                    if not (marked(v))
                       then begin mark (v);
                                  enqueue (v,q);
                                  P[v] := u
                            end
            end
    end{bfs1};
```

```
begin {bfs}
   for u ∈ V do begin unmark (u);
                      P[u] := empty
                end;
   Q := emptyq;
   for u ∈ W do
     begin  mark (u);
            enqueue (u,Q)
     end;
   bfs1 (Q)
end {bfs}
```

Korektnost prochÆzení grafu do šíky

Def: *DØlkou sledu* (v hranově neohodnocenØm grafu) rozumíme počet hran na tomto sledu.

VzdÆlenost z uzlu u *do uzlu* v je dØlka minimÆlního sledu vedoucího z uzlu u do uzlu v, pokud takový sled existuje; pokud z uzlu u do uzlu v žÆdný sled neexistuje, pak vzdÆlenost položíme $+\infty$.

V neorientovanØm grafu mluvíme stručně o *vzdÆlenosti mezi uzly* u *a* v.

Věta: Nechť $G = (V, E)$, $s, u, v \in V$ a nechť vzdálenost z s do u je menší než vzdálenost z s do v. Pak uzel u bude při výpočtu `bfs`$(G, \{s\})$ označen dříve než uzel v.

Důkaz: Stačí ukázat, že do fronty Q jsou vždy přidávány uzly s aspoň tak velkou vzdáleností od s, než jakou měl poslední přidaný uzel. To však lehce plyne indukcí podle vzdálenosti od uzlu s.

Věta: Nechť $G = (V, E)$, $s \in V$. Pak po provedení výpočtu `bfs`$(G, \{s\})$ jsou navštíveny právě všechny uzly, do nichž vede z uzlu s (orientovaná) cesta.

Důkaz: Indukcí podle vzdálenosti od uzlu s.

Složitost prochÆzení grafu do šíky

Věta: Nechť $G = (V, E)$ je graf a $W \subseteq V$ je množina poÈátečních uzlů (vstupního stupně 0). Pak dØlka výpočtu `bfs`(G, W) v nejhorším případě je v $\Theta(|W| + |E|)$.

Důkaz věty: Každý uzel bude označen a zařazen do fronty a u všech uzlů z fronty se ptÆme na označenost jejich nÆsledníků. Tedy dØlka výpočtu bude jistě v $\Omega(|E|)$. Jelikož zařazujeme do fronty všechny uzly z W, je dØlka výpočtu takØ v $\Omega(|W|)$. Z těchto dvou faktů vyplývÆ, že je v $\Omega(|E|) \cap \Omega(|W|) = \Omega(|W| + |E|)$.

Do fronty zařazujeme jen uzly, kterØ až do tØ doby nebyly označenØ, a přitom je hned označíme. Odtud plyne, že každý uzel je do fronty zařazen jen jednou.
Tedy vnější cyklus while proběhne $O(|W| + |E|)$-krÆt. Test ve vnitřním cyklu for proběhne vždy tolikrÆt, kolik hran vychÆzí z uzlu u, dohromady tolikrÆt, kolik mÆ celý graf hran. Odtud opět dostÆvÆme, že dØlka výpočtu je v $O(|W| + |E| + |E|) = O(|W| + |E|)$.

Dohromady dostÆvÆme $\Theta(|W| + |E|)$, čímž je tvrzení dokÆzÆno.

MinimÆlní kostra grafu

Def: Nechť $G = (V, E, h_E)$, $h_E : E \to \mathbb{R}$, je hranově ohodnocený neorientovaný graf. *Faktor* grafu G je podgraf $F = (V, E', h_E|_{E'})$ grafu G.

Def: Takový faktor grafu G, který je strom, se nazývÆ *kostra* grafu G.

Def: Nechť G je souvislý neorientovaný graf s číselně ohodnocenými hranami. Kostra grafu G, kterÆ mÆ ze všech jeho koster nejmenší součet hranových ohodnocení, se nazývÆ *minimÆlní kostra* grafu G.

PoznÆmka: Kostry mají jen souvislØ grafy, protože kostra musí podle definice být strom. Tento požadavek však není zÆsadní a většinu úvah o kostrÆch lze přenØst i na nesouvislØ grafy: stačí uvažovat zvlÆšť kostru každØ komponenty.

Cvičení: MÆme neorientovaný graf $G = (V, E)$, $V = \{1, 2, 3, 4\}$, $E = \{\{1, 2\}, \{2, 3\}, \{3, 4\}, \{4, 1\}, \{1, 3\}\}$.
Kolik mÆ tento graf podgrafů, faktorů a koster? V každØm z těchto tří případů spočítejte, kolik je *všech* podgrafů (faktorů, koster), a kolik z nich je *navzÆjem neisomorfních*.

Generický algoritmus pro nalezení minimÆlní kostry

ProblØm: Je dÆn souvislý hranově ohodnocený neorientovaný graf $G = (V, E, h_E)$.

Úkolem je nalØzt jeho minimÆlní kostru.

```
type SpT = Set of Edges;
procedure genericMST (G:Graph; var A:SpT);
  begin
    A := ∅;
    while A ještě netvoří kostru do
      begin
        najdi hranu {u, v} bezpečnou vzhledem k A;
        A := A ∪ {{u, v}}
      end
  end
```

Def: Nechť G je souvislý hranově ohodnocený graf a A takový jeho podgraf, že A je podgrafem nějaké minimální kostry T grafu G. Pak každá hrana z $G - A$, která leží v minimální kostře T, se nazývá *bezpečná* vzhledem k A.

Def: Nechť $G = (V, E)$ je graf, $V_1 \subset V$, $V_2 \subset V$, $V_1 \neq \emptyset$, $V_2 \neq \emptyset$, $V_1 \cup V_2 = V$, $V_1 \cap V_2 = \emptyset$. Pak množině

$$C = \{\{x, y\} \in E \mid x \in V_1, y \in V_2\}$$

říkáme *řez* grafu G.

Je-li navíc $G' = (V', E', h')$ podgraf grafu G takový, že $V' \subseteq V_1$ nebo $V' \subseteq V_2$, říkáme, že řez C *respektuje* podgraf G'.

Věta: Nechť $G = (V, E, h_E)$ je souvislý hranově ohodnocený neorientovaný graf, $h_E : E \to \mathbb{R}$. Nechť množina hran $A \subseteq E$ je obsažena v nějaké minimální kostře grafu G, nechť C je řez respektující podgraf indukovaný hranami z A a nechť $\{u, v\} \in C$ je hrana s minimálním ohodnocením v C. Pak hrana $\{u, v\}$ je bezpečná vzhledem k A.

Důsledek: Nechť $G = (V, E, h_E)$ je souvislý hranově ohodnocený neorientovaný graf, množina hran $A \subseteq E$ je obsažena v nějaké minimální kostře grafu G a nechť T je nějaká souvislá komponenta v lese (V, A). Pak každá hrana spojující T s nějakou jinou komponentou v lese (V, A) a mající minimální ohodnocení je bezpečná vzhledem k A.

Kruskalův (Borůvkův-Kruskalův) algoritmus

Je dán hranově ohodnocený neorientovaný graf $G = (V, E, h_E)$, hledáme množinu hran $A \subseteq E$ tvořící minimální kostru $T = (V, A, h_E|_A)$.

```
type SpT = Set of Edges;
     Component = Set of Nodes; {„vhodně reprezentovaná“ množina}
procedure kruskal (G:Graph; var A:SpT);
  var W: Set of Component;
  begin A := ∅;
        W := ∅;
        for v ∈ V do W := W ∪ {{v}};
        seřadíme množinu E podle ohodnocení;
        for (u, v) ∈ E {v pořadí od nejníže ohodnocené} do
            if W_u ≠ W_v {tj. u, v leží v různých komponentách}
                then begin A := A ∪ {(u, v)};
                           sjednotíme obě tyto komponenty
                     end
   end
```

Korektnost Kruskalova algoritmu

Věta: Po skončení výpočtu `kruskal`(G, A) je v A množina hran grafu G tvořících jeho minimÁlní kostru.

Důkaz vyplývÁ z předchozího Důsledku.

Složitost Kruskalova algoritmu

Složitost algoritmu zÁvisí na datovÉ struktuře reprezentující množinu komponent W. Pro reprezentaci komponent lesa můžeme použít tzv. binomiÁlní haldy. Dotaz $W_u \neq W_v$ vyjÁdříme dotazem, zda halda, v níž se nachÁzí uzel u, je shodný s haldou, v níž se nachÁzí uzel v. Tento dotaz mÁ logaritmickou složitost.

Složitost takto implementovanÉho Kruskalova algoritmu je pak $\Theta(m \log m)$, kde $m = |E|$.

Primův algoritmus

Podobně jako Kruskalův algoritmus je Primův algoritmus speciÆlním případem generickØho algoritmu pro hledÆní minimÆlní kostry. Zatímco však u Kruskalova algoritmu tvoří množina hran A v každØm kroku les, v případě Primova algoritmu tvoří vždy strom (tj. *souvislý* les).

Odtud vyplývÆ, že na rozdíl od Kruskalova algoritmu požadavek souvislosti vstupního grafu je podstatný.

Protože strom začínÆme budovat v určitØm místě grafu, je součÆstí vstupních dat i nějaký pevně zvolený uzel r.

Vstup: souvislý hranově ohodnocený neorientovaný graf $G = (V, E, h_E)$, uzel $r \in V$.

Výstup: minimÆlní kostra $T = (V, A, h_E|_A)$ určenÆ vektorem předchůdců P.

```
type Node = 1..n;
     SpT = Array [Node] of 0..n;
procedure prim (G:Graph; r:Node; var P:SpT);
  var Q: Heap; {prioritní fronta uzlů, uspořÆdanÆ vzhledem k key}
      key: Array [Node] of Real;
      u,v: Node;
  begin Q := V; {všechny uzly grafu}
        for u ∈ Q do key[u] := ∞;
        key[r] := 0.0;   P[r] := 0;
        while not isempty(Q) do
           begin u := minH(Q);  Q := extractMinH(Q);
                 for v ∈ Successors(u) do
                    if member(v,Q) && h_E(u,v) < key[v]
                       then begin P[v] := u;
                                  Q := removeH(v,Q);
                                  key[v] := h_E(u,v);
                                  Q := insertH(v,Q)
                            end
           end
  end
```

Korektnost Primova algoritmu

Pro důkaz korektnosti algoritmu uvažujeme tento invariant:

Pro každý uzel v je `key[`v`]` minimum z ohodnocení všech hran, kterØ spojují uzel v s některým uzlem již vytvořenØčÆsti kostry (za minimum prÆzdnØ množiny považujeme ∞).

DatovÆ struktura Q je prioritní fronta uzlů (může být reprezentovanÆ např. binÆrní haldou) uspořÆdanÆ podle hodnot `key`.

Potom minimÆlní prvek z Q je koncový uzel hrany, kterÆ mÆ minimÆlní ohodnocení ze všech hran řezu oddělujícího už vytvořenou čÆst kostry od ostatních uzlů grafu. Tato hrana je tedy bezpečnÆ a může být přidÆna do budovanØ minimÆlní kostry.

Množina A hran hotovØčÆsti kostry je v Primově algoritmu jen implicitně a je určena polem `P`, což je pole předchůdců každØho uzlu v minimÆlní kostře.

Složitost Primova algoritmu

Nechť $n = |V|$, $m = |E|$. Je-li prioritní fronta reprezentovanÆ binÆrní haldou, lze inicializaci provØst včase $O(n \log n)$.

Cyklus while proběhne n-krÆt. To znamenÆ, že všechny extrakce minimÆlních prvků z prioritní fronty potrvají celkem $\Theta(n \log n)$.

Cyklus for proběhne dohromady (v celØm výpočtu) $\Theta(m)$-krÆt. Tělo cyklu for vyžaduje čas $\Theta(\log n)$ a v celØm výpočtu tedy spotřebuje čas $O(m \log n)$. Protože v nejhorším případě bude podmínka v těle cyklu for $\Theta(m)$-krÆt splněna, můžeme horní odhad $O(m \log n)$ nahradit přesným odhadem $\Theta(m \log n)$.

Dohromady dostÆvÆme složitost $\Theta(n \log n + n \log n + m \log n) = \Theta(m \log n)$, což je totØž jako u Kruskalova algoritmu.