Daniel Kráľ; Martin Mareš; Milan Straka

Recepty z programátorské kuchařky Korespondenčního semináře z programování, 3. část

Rozhledy matematicko-fyzikální, Vol. 80 (2005), No. 4, 23–32

Persistent URL: http://dml.cz/dmlcz/146095

Recepty z programátorské kuchařky Korespondenčního semináře z programování, 3. část

Daniel Kráľ, Martin Mareš, Milan Straka, MFF UK Praha

Recepty z programátorské kuchařky jsou povídáním o algoritmech a datových strukturách, které připravili organizátoři Korespondenčního semináře z programování, který pro studenty středních škol pořádá MFF UK*). Tentokrát vysvětlíme, co je problém minimální kostry a jak ho řešit, a popíšeme datovou strukturu *Disjoint-Find-Union* (její název je často zkracován na DFU), která se nám bude při řešení tohoto problému hodit.

Grafy – některé termíny a vlastnosti

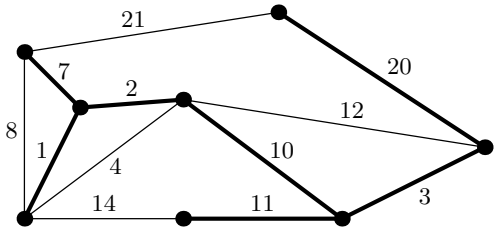
Co je *graf*, jsme vysvětlili v první z kuchařek. *Podgraf* nějakého grafu vznikne z původního grafu odebráním některých vrcholů a hran. *Cesta* v grafu je posloupnost vrcholů taková, že každý vrchol je s následujícím spojen hranou a vrcholy se na cestě neopakují. Pokud si graf představíme jako města spojená silnicemi, pak cesta je „cestovní plán“, jak se dostat z jednoho z měst do některého dalšího. *Kružnice* v grafu je posloupnost vrcholů, v níž jsou dvojice po sobě následujících vrcholů opět spojeny hranami, první vrchol je shodný s posledním a žádné jiné dva vrcholy nejsou totožné.

Pokud v grafu mezi každými dvěma vrcholy existuje cesta, řekneme, že graf je *souvislý*. Pokud graf souvislý není, můžeme ho rozdělit na podgrafy, které již souvislé jsou a nevedou mezi nimi žádné hrany. Takovým podgrafům budeme říkat *komponenty souvislosti*. Komponenta souvislosti je tedy maximální podgraf grafu, který je ještě souvislý. Souvislý graf má jednu komponentu souvislosti (celý graf).

*) Více se o semináři můžete dozvědět na jeho webové stránce `http://ksp.mff.cuni.cz/`.

Strom je graf, který je souvislý a zároveň *acyklický*, tj. neobsahuje žádnou kružnici. *List* stromu je takový vrchol, ze kterého vede jen jedna hrana. Není těžké si uvědomit, že každý strom (s alespoň dvěma vrcholy) má alespoň dva listy. O něco těžší je nahlédnout, že strom s n vrcholy má právě $n-1$ hran. Dokážeme to matematickou indukcí aplikovanou na počet vrcholů stromu. Strom s jedním vrcholem neobsahuje žádnou hranu. Pokud máme strom s n vrcholy, kde $n > 1$, potom libovolný list z tohoto stromu odebereme. Tím získáme opět strom (souvislost jsme porušit nemohli a kružnici jsme také nevytvořili), přičemž počet jeho vrcholů je o 1 menší. Podle indukčního předpokladu má tento strom o jednu hranu méně než vrcholů. Nyní list „přilepíme“ zpět, čímž zvýšíme počet vrcholů i hran o 1. Počet vrcholů uvažovaného stromu je tedy rovněž o 1 větší než počet hran.

V dalším se omezíme jen na souvislé grafy. *Kostra* souvislého grafu je jeho podgraf, který je stromem a obsahuje všechny vrcholy původního grafu. Jinými slovy, kostra je podgraf, který obsahuje všechny vrcholy a nejmenší počet hran, takový, že mezi každými dvěma vrcholy existuje cesta. Pokud každou hranu grafu ohodnotíme nějakou *vahou*, což v našem případě bude vždy kladné číslo, dostaneme *ohodnocený graf*. *Váha kostry* je součet vah hran, které obsahuje, a *minimální kostra* je kostra s nejmenší vahou (obr. 1). Graf může mít více minimálních koster, např. jestliže všechny hrany grafu mají ohodnocení 1, pak váha každé kostry je $n-1$, kde n je počet vrcholů grafu. Pokud si graf představíme jako města spojená silnicemi, pak problém nalezení minimální kostry můžeme vidět takto: Chceme určit silnice, které se budou v zimě udržovat sjízdné, a to tak, aby součet délek těchto silnic byl co nejmenší a zároveň aby bylo možné se po nich přepravit mezi každými dvěma městy.



Obr. 1. Příklad minimální kostry. Body reprezentují vrcholy grafu a úsečky hrany mezi nimi. Váhy hran jsou uvedeny čísly a minimální kostra je vyznačena tučně.

Algoritmus hledání minimální kostry

Algoritmus na hledání minimální kostry, který předvedeme, je typickou ukázkou tzv. *hladového algoritmu*. Nejprve setřídíme hrany vzestupně podle jejich váhy. Kostru budeme postupně vytvářet přidáváním hran od té s nejmenší vahou tak, že hranu do kostry přidáme právě tehdy, pokud spojuje dvě (prozatím) různé komponenty souvislosti vytvořeného podgrafu. Jinak řečeno, hranu do vytvářené kostry přidáme, pokud v ní zatím neexistuje cesta mezi vrcholy, které tato hrana spojuje.

Je zřejmé, že tímto postupem získáme kostru, tj. souvislý acyklický podgraf (pokud je vstupní graf souvislý, což mlčky předpokládáme). Než ukážeme, že nalezená kostra je opravdu minimální, podívejme se na časovou složitost našeho algoritmu: Pokud má vstupní graf N vrcholů a M hran, vyžaduje úvodní setřídění hran čas $O(M \log M)$ (použijeme některý z rychlých třídicích algoritmů popsaných v minulé části kuchařky). Po setřídění hran se pokusíme každou z M hran přidat. V dalším textu ukážeme datovou strukturu, s jejíž pomocí bude M testů toho, zda mezi dvěma vrcholy vede cesta, trvat nejvýše $O(M \log N)$. Celková časová složitost našeho algoritmu je tedy $O(M \log N)$ (platí totiž $\log M \leqq \log N^2 = 2 \log N$). Paměťová složitost je lineární vzhledem k počtu hran, tj. $O(M)$.

Důkaz správnosti hladového algoritmu

Zbývá dokázat, že nalezená kostra vstupního grafu je minimální. Bez újmy na obecnosti můžeme předpokládat, že váhy všech hran grafu jsou vesměs různé. Pokud tomu tak není, přičteme k některým z hran, jejichž váhy jsou duplicitní, velmi malá kladná čísla tak, aby pořadí hran nalezené naším třídicím algoritmem zůstalo zachováno. Tím se kostra nalezená naším algoritmem nezmění a pokud bude tato kostra minimální s modifikovanými váhami, bude minimální i pro původní zadání.

Označme T_{alg} kostru nalezenou naším algoritmem a $T_{\min}$ minimální kostru. Nechť $e_1, \ldots, e_{N-1}$ jsou hrany kostry T_{alg} v pořadí, v jakém byly naším algoritmem přidány do kostry. Pokud jsou kostry T_{alg} a $T_{\min}$ různé, pak existuje hrana e_i, která není obsažena v kostře $T_{\min}$. Zvolme nejmenší takové i, tedy kostra $T_{\min}$ obsahuje všechny hrany $e_1, \ldots, e_{i-1}$, ale neobsahuje hranu e_i. Váha každé hrany kostry $T_{\min}$,

kromě $i-1$ hran $e_1, \ldots, e_{i-1}$, je větší než váha hrany e_i. Kdyby tomu tak nebylo, pak by kostra $T_{\min}$ obsahovala hranu f s menší vahou než e_i a náš algoritmus by hranu f použil jakožto jednu z hran $e_1, \ldots, e_{i-1}$, což se nestalo.

Přidejme nyní hranu e_i ke kostře $T_{\min}$. Takto vzniklý podgraf vstupního grafu obsahuje kružnici, neboť před přidáním hrany e_i existovala v kostře $T_{\min}$ cesta mezi koncovými vrcholy hrany e_i. Tuto kružnici označme C. Protože T_{alg} neobsahuje žádnou kružnici, obsahuje kružnice C alespoň jednu hranu e', která není v kostře T_{alg}. Protože hrana e' není žádnou z hran $e_1, \ldots, e_{i-1}$, je její váha větší než váha hrany e_i. Odstraňme nyní hranu e' a označme T' výsledný podgraf, tj. graf získaný z kostry $T_{\min}$ záměnou hrany e' za hranu e_i. Protože hrany e' a e_i ležely na společné kružnici C, je T' souvislý podgraf. Součet vah hran kostry T' je menší než součet vah hran kostry $T_{\min}$, což není možné, neboť $T_{\min}$ je minimální kostra.

Předpoklad, že kostry T_{alg} a $T_{\min}$ jsou různé, tedy vede ke sporu. Proto je kostra T_{alg} nalezená naším algoritmem skutečně minimální.

Disjoint-Find-Union

Datová struktura DFU slouží k udržování rozkladu množiny na několik disjunktních podmnožin. To znamená, že pomocí této struktury můžeme pro každé dva z uložených prvků říci, zda patří či nepatří do stejné podmnožiny rozkladu. V algoritmu hledání minimální kostry budou prvky v DFU vrcholy daného grafu a budou náležet do stejné podmnožiny rozkladu, pokud mezi nimi v již vytvořené části kostry existuje cesta. Jinými slovy, podmnožiny v DFU budou odpovídat komponentám souvislosti vytvářené kostry.

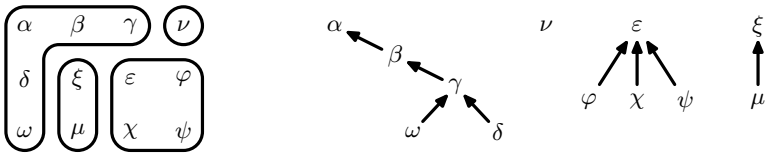
S reprezentovaným rozkladem umožňuje datová struktura DFU provádět tyto dvě operace:

find Test, zda dva prvky leží ve stejné podmnožině rozkladu. Tato operace bude v případě našeho algoritmu odpovídat testu, zda dva vrcholy leží ve stejné komponentě souvislosti.

union Sloučení dvou podmnožin do jedné. Tuto operaci v našem algoritmu na hledání kostry provedeme vždy, když do vytvářené kostry přidáme hranu (tehdy spojíme dvě komponenty souvislosti dohromady).

Nejdříve vysvětlíme, jak budeme jednotlivé podmnožiny v DFU reprezentovat. Prvky obsažené v jedné podmnožině budou tvořit zakořeněný strom. Připomeňme, že *zakořeněný strom* je datová struktura, kterou si můžeme představit jako orientovaný graf*) s jedním význačným vrcholem, *kořenem*, ze kterého vede právě jedna orientovaná cesta do každého jiného vrcholu. Kdybychom „zapomněli“ na orientaci hran, bude tento graf stromem ve smyslu definice stromu na začátku článku. Pokud z vrcholu u vede hrana do vrcholu v, řekneme, že vrchol u je *otcem* vrcholu v a vrchol v *synem* vrcholu u. Kořen stromu je tedy jediný vrchol, který nemá otce. Vrcholy, které nemají syny, se nazývají *listy*. Pokud v programu používáme zakořeněné stromy, odpovídají jejich vrcholy obvykle datovým položkám a orientované hrany mezi nimi představují ukazatele.

V datové struktuře DFU vedou ukazatele mezi prvky trochu nezvykle, od listů ke kořeni (obr. 2). Operaci `find` lze jednoduše implementovat tak, že pro dané dva prvky nejprve nalezneme kořeny jejich stromů. Jsou-li tyto kořeny stejné, jsou prvky ve stejném stromě, a tedy i ve stejné podmnožině rozkladu. Naopak, jsou-li kořeny různé, jsou dané prvky v různých stromech, a tedy i v různých podmnožinách reprezentovaného rozkladu. Operaci `union` provedeme tak, že mezi kořeny stromů reprezentujících slučované podmnožiny přidáme ukazatel, a tím tyto stromy spojíme dohromady.



Obr. 2. Rozklad množiny $\{\alpha, \beta, \gamma, \delta, \varepsilon, \varphi, \psi, \chi, \xi, \mu, \nu, \omega\}$ a jeho možná reprezentace pomocí datové struktury DFU.

Následuje implementace dvou právě popsaných operací. Množina, jejíž rozklad reprezentujeme, bude pro jednoduchost množina přirozených čísel od 1 do N. Ukazatele ve stromě si pamatujeme v poli `parent`, kde 0 znamená, že prvek nemá otce, tj. že je kořenem svého stromu. Funkce `root(v)` vrací kořen stromu, který obsahuje prvek v.

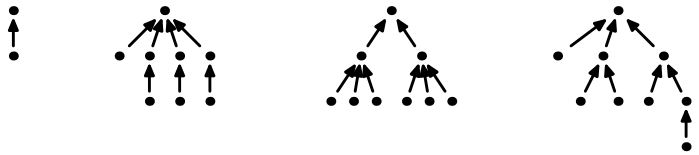
*) S orientovanými grafy jsme se setkali v první z našich Kuchařek.

```
var parent:array[1..N] of integer;

procedure init;
  var i:integer;
  begin
    for i:=1 to N do parent[i]:=0;
  end;

function root(v: integer):integer;
  begin
    if parent[v]=0 then root:=v
    else root:=root(parent[v]);
  end;

function find(v,w:integer):boolean;
  begin
    find:=(root(v)=root(w));
  end;

procedure union(v,w:integer);
  begin
    v:=root(v);
    w:=root(w);
    if v<>w then parent[v]:=w;
  end;
```

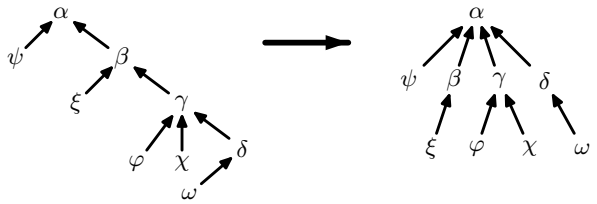
S předvedenou implementací operací `find` a `union` by se mohlo stát, že by stromy odpovídající některým podmnožinám vypadaly jako „hadi“; pokud by takový strom obsahoval N prvků, byl by na nalezení jeho kořene potřebný čas $O(N)$. Ke zrychlení práce DFU se používají dvě jednoduchá vylepšení:

union by rank Každý prvek má přiřazen `rank`. Na začátku jsou ranky všech prvků rovny nule. Při provádění operace `union` propojíme dva stromy s kořeny různých ranků tak, že ukazatel vede od kořene s menším rankem ke kořeni s větším rankem. Ranky kořenů (a ani ostatních prvků obou stromů) se v tomto případě nemění. Pokud kořeny obou stromů mají stejný rank, propojíme je libovolně, ale rank kořene výsledného stromu zvětšíme o 1 (obr. 3).

path compression Ve funkci `root(v)` přepojíme všechny prvky na cestě od prvku v ke kořeni přímo na kořen, tj. otce těchto prvků změníme na kořen příslušného stromu (obr. 4).



Obr. 3. Příklady stromů v datové struktuře DFU, jejichž kořeny mají postupně ranky 1, 2, 2 a 3.



Obr. 4. Provedení operace *path compression* při volání funkce `root` pro prvek δ.

Než obě metody blíže rozebereme, podívejme se, jak se změní implementace funkcí `root` a `union`:

```
var parent:array[1..N] of integer;
    rank:array[1..N] of integer;

procedure init;
  var i:integer;
  begin
    for i:=1 to N do
      begin
        parent[i]:=0;
        rank[i]:=0;
      end;
  end;
```

```
function root(v: integer):integer;
  { path compression }
  begin
    if parent[v]=0 then root:=v
    else begin
      parent[v]:=root(parent[v]);
      root:=parent[v];
    end;
  end;

function find(v,w:integer):boolean;
  begin
    find:=(root(v)=root(w));
  end;

procedure union(v,w:integer);
  { union by rank }
  begin
    v:=root(v);
    w:=root(w);
    if v=w then exit;
    if rank[v]=rank[w] then
      begin
        parent[v]:=w;
        rank[w]:=rank[w]+1;
      end
    else
      if rank[v]<rank[w] then parent[v]:=w
      else parent[w]:=v;
  end;
```

Zaměřme se nyní blíže na metodu *union by rank*.

Předně platí: Pokud je prvek v s rankem r kořenem stromu v datové struktuře DFU, pak tento strom obsahuje alespoň 2^r prvků. Dokážeme to indukcí podle r. Pro $r = 0$ tvrzení zřejmě platí. Předpokládejme nyní, že $r > 0$ a že tvrzení platí pro $r - 1$. V okamžiku, kdy se rank prvku v změnil z $r - 1$ na r, jsme sloučili dva stromy, jejichž kořeny měly rank $r - 1$. Každý z těchto dvou stromů měl dle indukčního před-

pokladu alespoň 2^{r-1} prvků, proto má výsledný strom alespoň 2^r prvků, což jsme měli dokázat.

Z dokázaného tvrzení plyne, že rank každého prvku je nejvýše $\log_2 N$ a že prvků s rankem r je nejvýše $N/2^r$ (všimněme si, že rank prvku v DFU se nemění od okamžiku, kdy tento prvek přestane být kořenem nějakého stromu).

Když provádíme jen *union by rank*, je *hloubka* každého stromu v DFU (tj. největší počet hran mezi vrcholem tohoto stromu a jeho kořenem) rovna ranku jeho kořene, protože rank kořene se mění právě tehdy, když hloubku stromu zvětšujeme o 1. A protože rank každého prvku je nanejvýš $\log_2 N$, je hloubka každého stromu v DFU také nanejvýš $\log_2 N$. V uvažovaném případě tedy procedura `root` spotřebuje čas nejvýše $O(\log N)$, a proto také operace `find` a `union` budou vyžadovat čas $O(\log N)$. Z toho vyplývá, že na vykonání M operací `find` a `union` budeme potřebovat čas $O(M \log N)$. To nám dává odhad potřebný v důkazu časové složitosti našeho algoritmu na hledání minimální kostry.

Disjoint-Find-Union s metodou path compression

Nyní popíšeme, jak se datová struktura DFU chová, pokud místo techniky `union by rank` použijeme `path compression`, a co se stane, když použijeme obě techniky současně.

Podívejme se nejprve na případ, kdy používáme jen techniku `path compression`. Začneme jednoduchým příkladem. Pokud datová struktura DFU obsahuje N jednoprvkových množin, jejichž prvky jsou, řekněme, čísla od 1 do N, může se stát, že postupným provedením operace `union` na dvojice prvků i a $i+1$, $i = 1, \ldots, N-1$, vytvoříme jeden strom tvořený orientovanou cestou s hranami z 1 do 2, z 2 do 3 atd. Pokud nyní zavoláme operaci `find` na číslo 1, budeme na nalezení kořene potřebovat čas lineární v N. Určitě tedy neplatí, že každé použití operace `find` nebo `union` vyžaduje čas $O(\log N)$. Lze však ukázat toto: Máme-li strukturu DFU tvořenou N jednoprvkovými stromy a provedeme-li M operací `union` a `find`, bude všech M operací dohromady potřebovat čas nejvýše $O(M \log N)$. V našem příkladě vyžadovalo prvních $N-1$ operací `union` čas $O(1)$ a poté jedna operace spotřebovala čas $O(N)$. Dohromady těchto $M = N$ operací potřebovalo čas $O(N)$.

Nemáme sice dobrý časový odhad toho, kolik času spotřebuje jedna operace `union` nebo `find`, ale máme zaručeno, že M těchto operací ne-

překročí čas $O(M \log N)$. Tento odhad je dostatečně dobrý pro použití v analýze algoritmu hledání minimální kostry.

Podívejme se nyní, co se stane, když budeme používat obě techniky současně. Ani v tomto případě neexistuje dobrý odhad času potřebného k provedení jedné operace `union` nebo `find`, ale lze ukázat, že na provedení M takových operací potřebujeme čas $O\big((N+M)\cdot \log^* N\big)$. Funkce $\log^* N$ je tzv. *iterovaný logaritmus*, jehož definice následuje. Nejprve definujeme funkci $2 \uparrow k$ rekurzivním předpisem

$$2 \uparrow 0 = 1, \qquad 2 \uparrow k = 2^{2\uparrow(k-1)}.$$

Je tedy $2 \uparrow 1 = 2,\ 2 \uparrow 2 = 2^2 = 4,\ 2 \uparrow 3 = 2^4 = 16,\ 2 \uparrow 4 = 2^{16} = 65\,536,$ $2 \uparrow 5 = 2^{65\,536}$ atd. Iterovaný logaritmus $\log^* N$ čísla N je pak nejmenší přirozené číslo k takové, že $N \leqq 2 \uparrow k$. Jiná (ekvivalentní) definice iterovaného logaritmu je ta, že $\log^* N$ je nejmenší počet, kolikrát musíme číslo N opakovaně zlogaritmovat, než dostaneme hodnotu menší nebo rovnu jedné. Vidíme tedy, že funkce $\log^* N$ je ve všech praktických aplikacích shora omezena číslem 5. Ve skutečnosti lze ukázat ještě lepší časový odhad $O\big(M\alpha(N)\big)$, kde $\alpha(N)$ je tzv. inverzní Ackermannova funkce. Definici této funkce, ani technicky poněkud náročnější důkazy uvedených časových odhadů předvádět nebudeme; čtenář je může nalézt v některém z pramenů uvedených v literatuře.

Literatura:

[1] Aho, A. V., Hopcroft, J. E., Ullman, J. D.: *The design and analysis of computer algorithms.* Addison-Wesley, Reading, MA, 1974

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.: *Introduction to algorithms, Chapter 21.* MIT Press, Cambridge, MA, 2001

[3] Tarjan, R. E., van Leeuwen, J.: *Worst-case analysis of set union algorithms.* J. ACM 31(2) (1984) 245–281

[4] Tarjan, R. E.: *Efficiency of a good but not linear set union algorithm.* J. Assoc. Comput. Mach. 22 (1975), 215–215