FAKULTA
APLIKOVANÝCH VĚD
ZÁPADOČESKÉ
UNIVERZITY
V PLZNI

KIV/PRO

KOMPONENTY ORIENTOVANÝCH A NEORIENTOVANÝCH GRAFŮ

Martin Hamet
hamet@students.zcu.cz

5. prosince 2017

Obsah

1 Zadání

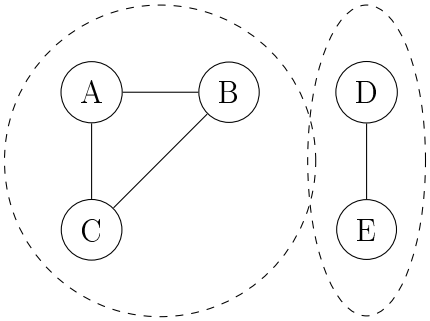
Nalezněte komponenty zadaného grafu. Vstupem algoritmu bude (ne)orientovaný graf a výstupem by měl být seznam komponent a vrcholů grafu které k nim přísluší.

2 Analýza

Komponentou grafu rozumíme soubor vrcholů ve kterém existuje cesta mezi každou dvojicí vrcholů z tohoto souboru.

2.1 Neorientovaný graf

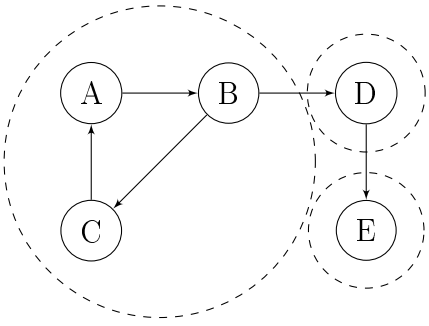
V neorientovaném grafu se problém redukuje pouze na hledání všech dostupných vrcholů z náhodně zvoleného počátečního vrcholu. Všechny nalezené vrcholy budou součástí jedné souvislé komponenty viz obr.1 kde jsou naznačeny dvě souvislé komponenty.



Obrázek 1: Neorientovaný graf.

2.2 Orientovaný graf

V orientovaném grafu musíme rozlišovat mezi silně a slabě souvislými komponentami. Silně souvislá komponenta odpovídá definici viz 2. Je tedy nutné aby existovala cesta mezi každými dvěma vrcholy komponenty jako je znázorněno na obr. 2, kde jsou naznačeny 3 silně souvislé komponenty.



Obrázek 2: Orientovaný graf.

Naopak pokud nás zajímá slabě souvislá komponenta stačí pokud existuje cesta alespoň jedním směrem. V případě obr. 2 by celý graf byl jednou slabě souvislou komponentou. Pro hledání slabě souvislých komponent můžeme využít přístupu jako pro neorientovaný graf tím že budeme ignorovat směry cest mezi jednotlivými uzly.

3 Existující metody

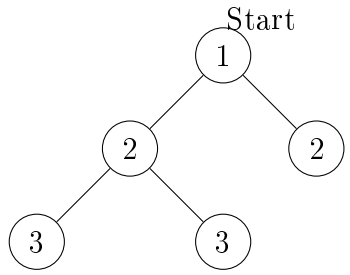
3.1 Neorientovaný graf

Pro nalezení souvislých komponent neorientovaného grafu lze využít metod, které zajistí uplný průchod grafem jako *Depth Fist Search* (DFS) a *Breadth First Search* (BFS) jak je zmíněno v knize *The Algorithm Design Manual[2]*.

Pro obě metody potřebujeme seznam všech vrcholů grafu včetně jejich hran. Algoritmus proběhne následujícím způsobem:

1. Nastavíme čítač komponent `k=1`.
2. Vybereme nenavštívený vrchol `V` ze seznamu vrcholů. Pokud jsou již všechny vrcholy označené jako navštívené ukončíme algoritmus.
3. Označíme `V` jako navštívený a jako součást komponenty číslo `k`.
4. Nový vrchol `V` získáme ze zvolené metody průchodu `DFS/BFS` a pokračujeme bodem `3`, pokud průchod dle `BFS/DFS` skončil zvýšíme čítač komponent `k=k+1` a pokračujeme bodem `2`.

3.1.1 Průchod BFS



Obrázek 3: Průchod BFS.

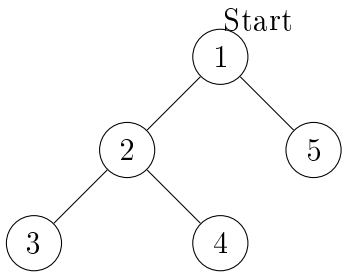
Pro průchod typu `BFS` využijeme fronty `Q` do které budeme vkládat nově nalezené vrcholy. Budeme tedy grafem postupovat "ve vlnách".

1. Zvolíme počáteční vrchol, označíme ho jako navštívený a vložíme ho do fronty `Q`.
2. Vybereme prvek `V` z fronty `Q`.
3. Všechny nenavštívené sousedy vrcholu `V` označíme jako navštívené a vložíme je do fronty `Q`.
4. Pokud je `Q` prázdná ukončíme průchod. Jinak pokračujeme bodem `2`.

Příklad průchodu je znázorněn na obr. 3, kde čísla jednotlivých vrcholů označují pořadí jejich nalezení.

3.1.2 Průchod DFS

Pro průchod typu `DFS` využijeme zásobníku `Z` do kterého budeme vkládat nově nalezené vrcholy. Budeme tedy grafem postupovat tak, že se nejprve "vnoříme"do co možná největší hloubky a postupně se vracíme k vrcholům které jsme vynechali po cestě.



Obrázek 4: Průchod DFS.

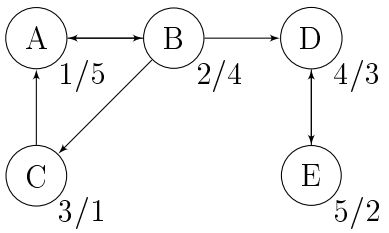
1. Zvolíme počáteční vrchol, označíme ho jako navštívený a vložíme ho do zásobníku `Z`.
2. Vybereme prvek `V` ze zásobníku `Z`. Pokud je `Z` prázdný ukončíme průchod.
3. Všechny nenavštívené sousedy vrcholu `V` označíme jako navštívené a vložíme je do zásobníku `Z` a pokračujeme bodem `2`.

3.2 Orientovaný graf

V orientovaných grafech nás budou zajímat pouze silně spojené komponenty. Pro slabě spojené komponenty lze využít postupů pro neorientované grafy (viz 3.1).

3.2.1 Kosaraju's algorythm[1]

Tento přístup vyžaduje dva modifikované `DFS` průchody. Postup lze rozdělit do dvou fází dopředný a zpětný průchod. Využijeme dvou zásobníků Z_1 [1] a Z_2. Do zásobníku Z_1 budeme ukládat nalezené vrcholy čekající na ukončení a do Z_2 ukončené vrcholy.



Obrázek 5: Dopředný průchod grafem.

značení (C_1/C_2):
C_1 - pořadí přidání vrcholu do Z_1
C_2 - pořadí přidání vrcholu do Z_2

[1] Zásobník Z_1 lze nahradit rekurzivním průchodem `DFS`.

Dopředný průchod

1. Zvolíme nenavštívený vrchol ze seznamu všech vrcholů, označíme ho jako navštívený a vložíme ho do Z_1.
2. Vybereme jednoho nenavštíveného souseda prvku V_{z1}, označíme ho jako navštíveného a vložíme ho do Z_1, pokud V_{z1}[2] nemá žádné další nenavštívené sousedy, odstraníme ho ze zásobníku Z_1 a vložíme ho do Z_2.
3. Opakujeme krok `2` dokud není Z_1 prázdný.
4. Pokračujeme krokem `1` dokud nejsou všechny vrcholy grafu označené jako navštívené.

Příklad V grafu (viz obr.5) jsme začali z vrcholu `A` a sousedy jsme nacházeli v pořadí v pořadí `B`, `C`. Vrchol `C` již nemá dalšího nenavštíveného souseda odebereme ho tedy ze zásobníku Z_1 a přidáme do Z_2. Dále pokračujeme sousedy vrcholu `B`. Vrchol `E` bude tedy jako druhý přidaný do Z_2.

Obsah Z_2 po dopředném průchodu bude v pořadí od vrcholu zásobníku *C, E, D, B, A*.

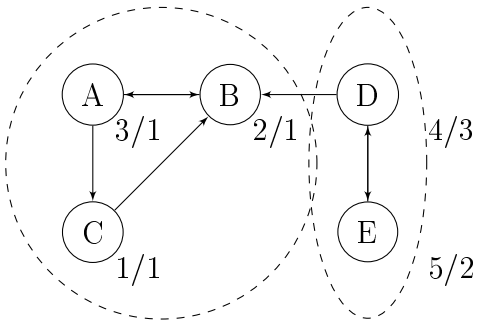
Zpětný průchod Nejprve je třeba vytvořit transponovaný graf k původnímu. Tedy otočit směr všech hran v grafu.

1. Nastavíme počítadlo komponent `k=1`.
2. Vybereme prvek V_{z2}[3] , označíme ho jako navštívený[4] a označíme ho jako součást komponenty `k`.
3. Z vybraného vrcholu provedeme průchod grafu (`DFS` viz 3.1.2 nebo `BFS` viz 3.1.1), nové vrcholy označujeme jako nalezené a přiřazujeme je komponentě `k`.
4. Po skončení průchodu vybereme další V_{z2} ze zásobníku a zvýšíme čítač komponent `k=k+1`.
   - Pokud je vybraný vrchol již označený jako nalezený opakujeme výběr.

[2] Vrchol V_{z1} je prvek na vrcholu zásobníku Z_1.
[3] Vrchol V_{z2} je prvek na vrcholu zásobníku Z_2.
[4] Označení nalezených vrcholů se nepřenáší z dopředného průchodu.

– Pokud je vrchol neoznačený označíme ho jako nalezený a pokračujeme krokem `3`.
– Pokud je Z_2 prázdný ukončíme algoritmus.

Příklad V zásobníku Z_2 se po dopředném průchodu nacházejí prvky *C, E, D, B, A*. Vybereme V_{z2} (C), označíme ho jako nalezený a jako součást komponenty `k`. Průchodem transponovaného grafu (viz obr.6) z vybraného vrcholu C dle `DFS` nalezneme vrcholy *A, B*, které označíme za nalezené a jako součásti komponenty `k`.



Obrázek 6: Zpětný průchod grafem.

značka (C_1/C_2):
C_1 - pořadí přidání nalezení vrcholu
C_2 - číslo přiřazené komponenty

Zvýšíme čítač komponent `k=2`. Vybereme další prvek ze zásobníku Z_2 (E). Označíme vrchol jako nalezený a jako součást komponenty `k`. Provedeme průchod grafu z vrcholu E přes všechny nenalezené prvky. Jediný nenalezený prvek je D, který obdobně označíme. V Z_2 zbývají prvky *D, B, A*, které postupně odstraňujeme, protože již byli označeny jako nalezené v předchozích krocích.

Výsledkem jsou dvě silně souvislé komponenty `k=1`(*A, B, C)* a `k=2` (*D, E*).

3.2.2 Tarjan's algorythm[3]

Tento přístup oproti *Kosaraju's algorythm* uvedeného v sekci 3.2.1 urychlí zpracování grafu, protože vyžaduje pouze jeden průchod do hloubky (`DFS` viz 3.1.2). Při průchodu grafem je objeveným vrcholům přiřazena hodnota nalezení `v.dsc` a jsou označeny jako zpracovávané a jsou přidány do zásobníku. Pokud je v průběhu nalezena cesta k vrcholu, který je zpracováván, jedná se o smyčku a všechny zúčastněné vrcholy jsou jedné komponenty. Při průchodu se snažíme najít co největší smyčku. Ukládáme si tedy hodnotu o nejstaršího dosažitelného vrcholu `v.low`.

Po prozkoumání všech následníků vrcholu `v`, kterému zůstali přiřazené stejné hodnoty objevení a nejstaršího následníka (`v.low` a `v.dsc`), všichni jeho ještě zpracovávaní následníci jsou součástí jedné komponenty. Tyto vrcholy jsou odebrány ze zásobníku a jsou označeny jako zpracované.

```
   Data: Graph(V, E)
   /* Množina vrcholů V a hran.                                  */
   Result: V, K
   /* Množina vrcholů V s číslem komponenty K.                   */
1  time=1;
   /* čítač času nalezení vrcholu                                */
2  k=0;
   /* čítač komponent                                            */
   /* Čas nalezení vrcholu                                       */
3  foreach v in V do                  /* pro každý vrchol grafu */
4  |  if v.dsc = 0 then          /* pokud v ještě nebyl nalezený */
5  |  |  Scc(u);
6  |  end
7  end

8  Function Scc(v):
9  |  v.dsc = time; /* čas objevení vrcholu                      */
10 |  v.low = time; /* inicializace nejstaršího následovníka    */
11 |  time = time + 1;
12 |  stack.push(v);
13 |  v.prg = true; /* označení vrcholu jako zpracovávaný       */
   |  /* zpracování následovníků                                 */
14 |  foreach n in v do                 /* pro každého souseda v */
15 |  |  if n.dsc=0 then                        /* neobjevený v */
16 |  |  |  Scc(n);
17 |  |  |  v.low = min(v.low, n.low);
   |  |  |  /* výběr nejstaršího následovníka                   */
18 |  |  else if n.prg=true then     /* n je již zpracovávaný */
19 |  |  |  v.low = min(v.low, n.dsc);
20 |  |  end
21 |  end
22 |  if v.low=v.dsc then       /* nejstarší předek komponenty */
23 |  |  k=k+1;
24 |  |  repeat
25 |  |  |  x=stack.pop();
26 |  |  |  x.prg=false;
27 |  |  |  přidáme x do komponenty č.k
28 |  |  until x!=v;
   |  |  /* komponenta k je hotová a je možné jí vypsat        */
29 |  end
   |  /* návrat k předkovi                                      */
30 |  return;
```

Algorithm 1: Tarjan's algorythm

4 Zvolené řešení

Pro řešení problému jsme zvolili *Tarajan's algorithm* (viz 3.2.2) především kvůli rychlejšímu zpracování. První zmíněnou metodu zpomalují dva potřebné průchody grafu.

Originální algoritmus jsme upravili pouze přidáním dalšího vlastního zásobníku pro nahrazení rekurze, která vyžaduje značnou velikost programového zásobníku což značně ovlivňovalo možnou velikost zpracovávaných dat.

4.1 Implementace

V této sekci jsou popsány hlavní moduly programu a jejich vzájemná funkce.

4.1.1 `Program`

Hlavní třída programu zodpovědná za zpracování předaných argumentů programu a následné spuštění zvolených algoritmů.

4.1.2 `GraphGen`

Třída určená pro generování grafu podle zadaných parametrů. Graf je reprezentován maticí sousednosti.

`getMatrix()` Vygeneruje a vrátí požadovanou matici grafu.

`getReport()` Vygeneruje informace o posledním grafu a vrátí je v textové podobě.

4.1.3 `SCCI`

Interface pro jednotlivé implementace algoritmů pro získání silně souvislých komponent.

`WriteComponents()` Metoda provede algoritmus aktuální zvolené implementace a jednotlivé komponenty vypíše do konzole.

4.1.4 Implementace algoritmů

Jednotlivé třídy testovaných algoritmů implementující rozhraní `SCCI`.

`Tarjan` Originální *Tarjan's algorithm* implementovaný s využitím rekurze.

`TarjanNonRec` *Tarjan's algorithm* implementovaný s bez využití rekurze.

`Kosaraju` Originální *Kosaraju's algorythm* implementovaný bez využití rekurze.

5 Uživatelská příručka

Program vyžaduje nainstalovaný `Microsoft.NET Framework`.

5.1 Program

Program je koncipován jako konzolová aplikace. Argumenty programu jsou popsány níže. Samotný program nevyžaduje žádné další soubory. Všechna potřebná vstupní data jsou generovány programem samotným podle zadaných parametrů.

5.2 Spuštění programu

Program se spouští z příkazové řádky s následujícími parametry
(bez znaků `[` a `]`).

`Scc.exe [`n_1`] [`n_2`] [-s_<`n_3`>] [-m] [-c] [-d] [-r] [-k]`

Defaultní nastavení programu

- Program generuje neorientované grafy.
- Generátor využívá systémového času v milisekundách pro generování náhodných čísel.
- Pro výpočet je použitá nerekurzivní implementace *Tarjan's algorithm.*
- Ve výsledném výpisu jsou u jednotlivých komponent pouze čísla, označující kolik uzlů bylo přiřazeno dané komponentě.

Příklad spuštění: `Scc.exe 15 20 -d -s 123 -k -c -m`

5.2.1 Atributy

Povinné atributy

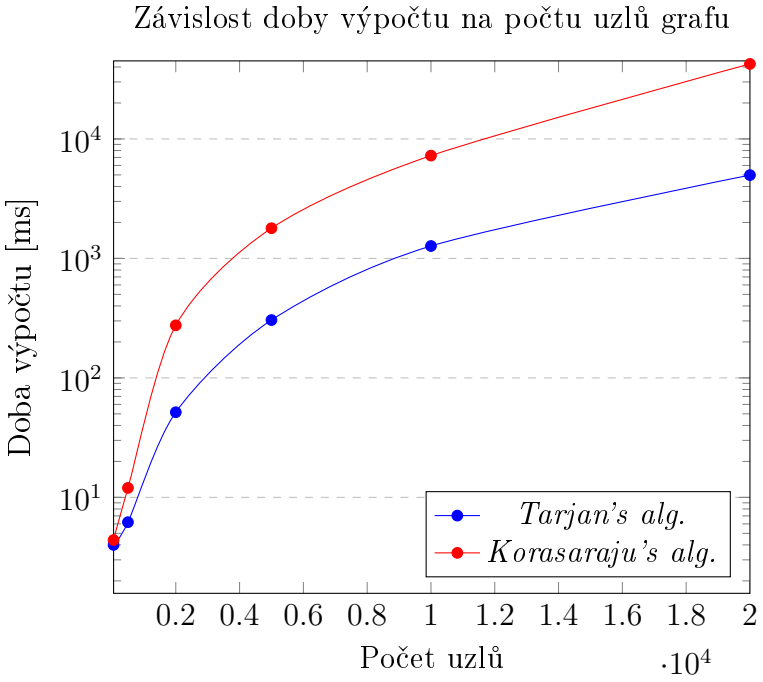
- [n_1] Počet uzlů grafu.
- [n_2] Maximální počet hran v grafu.

Nepovinné atributy

- [-s <n_3>] Použití deterministického generátoru pro graf, kde n_3 je použitý jako počáteční stav ("seed").
- [-m] Vypíše do konzole matici sousednosti vygenerovaného grafu.
- [-c] Vypíše konkrétní čísla uzlů přiřazených ke komponentám.
- [-d] Generátor grafu bude generovat orientovaný graf.
- [-r] V případě *Tarjan's algorithm* jako výpočetního algoritmu použije jeho rekurzivní implementaci.
- [-k] Použije *Kosaraju's algorithm* jako způsob nalezení komponent.

6 Experimenty a výsledky

Pro testování byly vždy generovány neorientované grafy z důvodu vzniku více souvislých komponent, oproti generování mnoha jednoprvkových. Každé měření bylo provedeno 10× a časy běhu jsme získali jako průměr jednotlivých měření.



Obrázek 7: Graf k tabulce 1.

		Čas běhu		
Uzlů	Hran	Tnr	Knr	Parametry programu
50	50	$4.0ms$	$4.4ms$	`50 50 -s 244`
500	500	$6.2ms$	$12.0ms$	`500 500 -s 292`
2×10^3	2×10^3	$51.6ms$	$275.5ms$	`2000 2000 -s 179`
5×10^3	5×10^3	$304.8ms$	$1.79s$	`5000 5000 -s 186`
1×10^4	1×10^4	$1.27s$	$7.25s$	`5000 5000 -s 322`
2×10^4	2×10^4	$4.98s$	$42.45s$	`5000 5000 -s 614`

Tabulka 1: Porovnání algoritmů

Tnr - *Tarjan's algorithm* bez rekurze
Knr - *Kosaraju's algorithm bez rekurze*

7 Závěr

Řešení problému pro neorientované grafy se dá snadno nalézt pomocí úplného průchodu grafu. Pro orientované grafy jsme testovali dvě metody (*Kosaraju's algorithm* a *Tarjan's algorithm*. Obě metody naleznou řešení vždy ovšem rozdíly jsou patrné při větším počtu uzlů a hran jak je znázorněno na grafu obr. 7. Při větším počtu uzlů se značně projeví lineární složitost druhého zmíněného algoritmu.

Při testování vznikl problém s omezenou velikostí programového zásobníku, proto jsme implementovali verzi *Tarajn's algorithm* bez použití rekurze. Tato verze umožnila zpracování grafů až do řádů desítek tisíc uzlů. Další navyšování velikosti grafu by bylo možné po dalších úpravách jeho reprezentace.

Reference

[1] Jeffrey D. Ullman Alfred V. Aho, John E. Hopcroft. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[2] Steven S. Skiena. *The Algorithm Design Manual.* Springer Publishing Company, 2008.

[3] R. E Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.