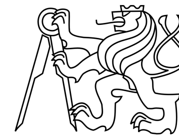
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMA NÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁ SKÉ PRÁCE

Název: Automatová knihovna - Grafy a grafové algoritmy
Student: Jan Brož
Vedoucí: Ing. Radomír Polách
Studijní program: Informatika
Studijní obor: Teoretická informatika
Katedra: Katedra teoretické informatiky
Platnost zadání: Do konce letního semestru 2016/17

Pokyny pro vypracování

Nastudujte grafy a grafové algoritmy maximálního toku, minimálního ezu a minimální kostry pro orientované a neorientované grafy.
Zrevidujte aktuální stav implementace graf v Automatové knihovn , která je vyvíjena na kated e teoretické informatiky, a rozši te ji tak, aby bylo možné vhodn implementovat výše zmín né grafové algoritmy.
Implementujte výše zmín né algoritmy.
Navrhn te vhodné testování implementovaných algoritm a prove te ho.

Seznam odborné literatury

Dodá vedoucí práce.

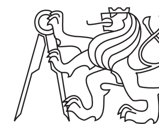
L.S.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
d kan

V Praze dne 18. února 2016

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky



Bakalářská práce

Automatová knihovna - Grafy a grafové algoritmy

Jan Brož

Vedoucí práce: Ing. Radomír Polách

17. května 2016

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 17. května 2016 ....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2016 Jan Brož. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Brož, Jan. *Automatová knihovna - Grafy a grafové algoritmy.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Tato práce se zabývá algoritmy z teorie grafů a reprezentací grafů v paměti počítače. Cílem je nastudovat a implementovat vybrané grafové algoritmy a integrovat je do knihovny zvané Automata Library (zkráceně ALib), vyvíjené na katedře. Jmenovitě jde o algoritmy nalezení minimální kostry, maximálního toku a minimálního řezu, a to jak pro neorientované, tak pro orientované grafy. Pro studium těchto algoritmů byla použita skripta pro ČVUT a publikace dostupné online. V souladu s knihovnou ALib byly implementovány v programovacím jazyce C++ a jejich správnost byla ověřena navržením a provedením automatizovaných testů. Vytvořené řešení poskytuje referenční implementaci grafových struktur a výše zmíněných algoritmů a lze použít pro doplnění jejich výuky.

Klíčová slova grafy, grafové algoritmy, implementace, knihovna, datové struktury, tok v síti, řez grafu, kostra grafu, C++

Abstract

This thesis deals with algorithms from the graph theory and representation of graphs in a computer memory. The goal is to study and implement selected graph algorithms and integrate them into a library called Automata Library (shortened as ALib), being developed at the department. Namely it is about algorithms for finding a minimum spanning tree, maximum flow and minimum cut in both undirected and directed graphs. To study these algorithms, textbooks of CTU and publications available online were used. In compliance with ALib, algorithms were implemented in programming language C++ and their correctness was verified by designing and executing automated tests. The solution provides a reference implementation of graph structures and above mentioned algorithms and can be used to support teaching them.

Keywords graphs, graph algorithms, implementation, library, data structures, flow in a network, cut of a graph, spanning tree, C++

Obsah

Úvod **3**
Význam oblasti 3
Zařazení do souvislostí 3
Cíl práce 3
Struktura práce 4

1 Definice pojmů **5**

2 Existující řešení **9**
2.1 Algoritmy 9
2.2 Grafové knihovny 9

3 Reprezentace grafů **13**
3.1 Známé reprezentace 13
3.2 Původní řešení v ALib 16
3.3 Provedené změny 17

4 Grafové algoritmy **19**
4.1 Nalezení minimální kostry neorientovaného grafu 19
4.2 Nalezení minimální kostry orientovaného grafu 22
4.3 Nalezení maximálního toku v síti 25
4.4 Nalezení minimálního řezu v síti 27
4.5 Nalezení maximálního toku v neorientované síti 29
4.6 Nalezení minimálního řezu v neorientované síti 31

5 Testování **33**
5.1 Test Kruskalova algoritmu 33
5.2 Test Edmondsova algoritmu 34
5.3 Test Ford/Fulkersonova algoritmu pro tok sítě 34
5.4 Test Ford/Fulkersonova algoritmu pro řez sítě 34

5.5 Test upravených algoritmů pro tok a řez neorientované sítě . . 34

Závěr **35**

Literatura **37**

A Obsah přiloženého CD **39**

Seznam obrázků

3.1 Příklad neorientovaného grafu 13
3.2 Příklad orientovaného grafu 13
3.3 Seznam hran neorientovaného grafu 3.1 14
3.4 Seznam hran orientovaného grafu 3.2 14
3.5 Seznamy sousedů neorientovaného grafu 3.1 14
3.6 Seznamy následníků orientovaného grafu 3.2 14
3.7 Matice sousednosti neorientovaného grafu 3.1 15
3.8 Matice sousednosti orientovaného grafu 3.2 15
3.9 Matice incidence neorientovaného grafu 3.1 15
3.10 Matice incidence orientovaného grafu 3.2 15
3.11 Předchozí vztahy tříd (podobně i pro orientovaný graf) 17
3.12 Nové vztahy tříd (podobně i pro orientovaný graf) 17

4.1 Příklad neexistence orientované kostry 22

Seznam tabulek

3.1 Srovnání složitostí reprezentací 16

Seznam algoritmů

1 Kruskalův algoritmus 22
2 Chu/Liu/Edmondsův algoritmus 26
3 Ford/Fulkersonův algoritmus 28
4 Nalezení minimálního řezu pomocí maximálního toku 29
5 Ford/Fulkersonův algoritmus pro neorientovanou síť 30
6 Nalezení minimálního řezu v neorientované síti 31

Úvod

Význam oblasti

Teorie grafů je velmi užitečná v mnoha oborech informačních technologií, jako například počítačové sítě, GPS navigace, vyhledávání textu nebo vztahy mezi uživateli sociální sítě. Dává návod, jak efektivně provádět operace nad strukturami, které lze chápat jako množinu bodů a spojnic mezi nimi. Referenční implementace algoritmů z grafové teorie se může hodit například při jejich výuce, nebo může sloužit jako základ pro upravené optimalizované verze řešící skutečné problémy z praxe.

Zařazení do souvislostí

Tato práce je součástí knihovny Automata Library nebo-li Automatová knihovna (dále jen zkráceně ALib). ALib je programová knihovna implementující matematické modely automatů, gramatik a dalších struktur, současně s algoritmy pracujícími nad těmito strukturami. Knihovna je vyvíjena na Katedře teoretické informatiky a přispívají do ní jak zaměstnanci katedry, tak studenti v rámci závěrečných prací. Účelem knihovny, a tedy i této práce, je poskytnout referenční přehlednou implementaci datových struktur a algoritmů používaných v teoretické informatice.

Dále práce částečně navazuje na bakalářskou práci Davida Roscy o isomorfismu planárních grafů [1]. Zdrojové kódy z jeho práce v repozitáři ALibu budou použity jako výchozí a budou dále upraveny tak, aby lépe vyhovovaly novým algoritmům.

Cíl práce

Cílem práce je vytvořit referenční implementaci několika algoritmů z grafové teorie a integrovat je do knihovny ALib. Jmenovitě jde o algoritmy nalezení

minimální kostry, maximálního toku a minimálního řezu, a to jak pro neorientované, tak pro orientované grafy. Pro splnění tohoto cíle bude nejprve zapotřebí revidovat stávající datové struktury reprezentující grafy a upravit je tak, aby splňovaly požadavky nových algoritmů, případně upravit kód pro lepší čitelnost. Následně bude třeba nastudovat vybrané algoritmy a implementovat je v jazyce C++ tak, aby byly dodrženy zvyklosti a rozhraní knihovny ALib. Na závěr se budou muset pro dané algoritmy navrhnout a provést testy, aby byla ověřena jejich správnost. Cílem není vytvořit optimalizovaný a rychlý kód, větší důraz bude kladen na jeho přehlednost a aby co nejvíce odpovídal matematickým zápisům a definicím.

Struktura práce

Práce má 5 části. Nejdříve jsou uvedeny základní definice z teorie grafů, které jsou potřeba pro pochopení algoritmů. Druhá část uvádí některé existující implementace grafů a grafových algoritmů. Ve třetí části je probrána současná implementace grafů v knihovně ALib a potřebné úpravy a vylepšení. Čtvrtá část se věnuje samotným algoritmům, jejich principům a problémům při jejich implementaci. V poslední části je popsán způsob testování algoritmů.

Kapitola 1

Definice pojmů

Zde jsou definovány pojmy z teorie grafů, které se v práci používají. Většina z nich je převzata z [2] aby nedocházelo k nekonzistencím.

Definice 1 (neorientovaný graf)**. Neorientovaný graf** je uspořádaná trojice $G = (H, U, \varrho)$, kde H je množina **hran**, U je množina **uzlů** a ϱ je zobrazení přiřazující každé hraně neuspořádanou dvojici uzlů. Toto zobrazení nazýváme **incidence grafu**. Říkáme, že **hrana** h **inciduje** s uzly u a v, když $\varrho(h) = \{u, v\}$

Poznámka 1. Některé publikace používají zjednodušenou definici grafu, jako uspořádanou dvojici $G = (H, U)$, kde H je přímo množina dvojic uzlů nazývaných hrany. Tato definice v mnoha případech stačí, ale je méně obecná a neumožňuje například multigrafy (bude vysvětleno později).

Definice 2 (orientovaný graf)**. Orientovaný graf** je uspořádaná trojice $G = (H, U, \sigma)$, kde H je množina **hran**, U je množina **uzlů** a σ je zobrazení přiřazující každé hraně uspořádanou dvojici uzlů. Toto zobrazení nazýváme **incidence grafu**. Říkáme, že **hrana** h **inciduje** s uzly u a v, když $\varrho(h) = (u, v)$

Poznámka 2. Neorientovaný a orientovaný graf se liší pouze v tom, že incidence jednou přiřazuje neuspořádanou a podruhé uspořádanou dvojici uzlů.

Definice 3 (podgraf)**.** Graf $G' = (H', U', \varrho')$ nazýváme **podgrafem** grafu $G = (H, U, \varrho)$ (zapisujeme $G' \subseteq G$), jestliže platí

$$(H' \subseteq H) \wedge (U' \subseteq U) \wedge (\forall h \in H' : \varrho'(h) = \varrho(h)).$$

Definice 4 (rovnoběžná hrana)**.** Hrany $h1$ a $h2$ nazýváme **rovnoběžné**, pokud $\varrho(h1) = \varrho(h2)$ (mají stejné krajní uzly).

Definice 5 (smyčka)**.** Hranu h nazýváme smyčkou, pokud $\varrho(h) = \{u, u\}$ pro nějaký uzel $u \in U$.

Definice 6 (prostý graf)**.** Graf se nazývá **prostý**, pokud neobsahuje žádné rovnoběžné hrany (zobrazení ϱ je prosté).

Definice 7 (obyčejný graf)**.** **Obyčejným grafem** nazýváme graf prostý a bez smyček.

Definice 8 (multigraf)**.** Graf se nazývá **multigraf**, pokud obsahuje alespoň jednu dvojici rovnoběžných hran (není prostý).

Definice 9 (hranově ohodnocený graf)**.** Graf nazýváme **hranově ohodnocený**, pokud existuje zobrazení přiřazující každé jeho hraně reálné číslo.

Definice 10 (sled)**.** Pro danou dvojici uzlů u a v grafu $G = (H, U, \varrho)$ nazýváme **sledem** posloupnost uzlů a hran

$$S = (u_0, h_1, u_1, h_2, \ldots, u_{n-1}, h_n, u_n)$$

kde $u_0 = u$ a $u_n = v$, $u_i \in U$, $h_i \in H$, $\varrho(h_i) = \{u_{i-1}, u_i\}$. Sled s alespoň jednou hranou, v němž jsou uzly u a v shodné, nazýváme **uzavřeným**, ostatní sledy nazýváme **otevřenými**.

Definice 11 (tah)**.** **Tahem grafu** G nazýváme takový jeho sled, v němž jsou všechny hrany různé.

Definice 12 (cesta)**.** **Cestou grafu** G nazýváme takový jeho tah, v němž každý uzel inciduje nejvýše se dvěma hranami tohoto tahu.

Definice 13 (kružnice)**.** **Kružnicí grafu** G nazýváme jeho uzavřenou cestu.

Definice 14 (souvislý graf)**.** **Souvislým grafem** nazýváme takový graf, mezi jehož libovolnými dvěma uzly existuje sled. **Komponentou grafu** nazýváme každý jeho maximální souvislý podgraf.

Definice 15 (strom)**.** **Stromem** nazýváme souvislý graf, který neobsahuje kružnice.

Definice 16 (kostra grafu)**.** **Kostrou grafu** G rozumíme takový jeho podgraf, který obsahuje všechny uzly z G a je stromem.

Definice 17 (síť)**. Sítí** nazýváme čtveřici $S = (G, q, s, t)$, kde $G = (H, U)$ je obyčejný orientovaný graf, $s \in U$ je **zdroj sítě** S, $t \in U$ je **spotřebič sítě** S a $q : H \mapsto R^+$ je nezáporné ohodnocení hran nazývané **kapacita sítě** S. Pro každou hranu h nazýváme příslušnou hodnotu $q(h)$ **kapacitou hrany** h.

Definice 18 (tok v síti)**. Tokem v síti** $S = (G, q, s, t)$ nazýváme hodnocení hran $f : H \mapsto R^+$ splňující následující podmínky:

- pro každou hranu $(u, v) \in H$ platí $0 \leq f(u, v) \leq q(u, v)$
- pro každý uzel u sítě S různý od s a t platí

$$\sum_{(u,v)\in H} f(u, v) = \sum_{(w,u)\in H} f(w, u).$$

Velikost toku f je dána rozdílem

$$|f| = \sum_{(s,v)\in H} f(s, v) - \sum_{(u,s)\in H} f(u, s) = \sum_{(u,t)\in H} f(u, t) - \sum_{(t,v)\in H} f(t, v).$$

Definice 19 (řez sítě)**. Řezem sítě** nazýváme množinu hran, po jejichž odebrání se zdroj a spotřebič nacházejí v různých komponentách.

Kapitola 2

Existující řešení

2.1 Algoritmy

Algoritmů pro nalezení minimální kostry neorientovaného grafu a nalezení maximálního toku a minimálního řezu v síti je více a většina z nich je známá a publikovaná na mnoha místech, například [2], [3], [4] nebo [5]. Existují k nim slovní popisy, pseudokódy i zdrojové kódy v různých jazycích dostupné pod určitou licencí. Jejich vlastní implementace by tedy neměla být problém.

Nalezení kostry v orientovaném grafu je již méně známý problém. Na jeho řešení přišli nezávisle na sobě Yoeng-Jin Chu a Tseng-Hong Liu (1965) a následně Jack Edmonds (1967). Jejich algoritmus je značně komplikovaný a existuje o něm jen několik článků v zahraniční literatuře (např. [6]). Na vině je nejspíše jeho menší využití. Jeho podoba se v průběhu let měnila. Objevily se články pokoušejících se o jeho upřesnění, navrhujících jeho konkrétní implementaci [7] nebo zpochybňujících implementace předchozí [8]. Dodnes je jeho popis stále poněkud matoucí. Tento algoritmus zatím není implementován v žádné veřejně dostupné C++ knihovně. Povede-li se ho zprovoznit v knihovně ALib, může to být pro akademickou sféru přínosem.

Algoritmy maximálního toku a minimálního řezu pracující na neorientovaných grafech nikde oficiálně publikované nejsou. Teorie grafů totiž nezná pojem „neorientovaná síť“. V praxi se ale tyto problémy mohou vyskytnout a proto je výhodné se těmito algoritmy zabývat. Další důvod, proč o nich neexistuje dílo, může být fakt, že algoritmy jsou poměrně jednoduchými modifikacemi verzí pro orientované grafy, které se dají po chvíli vymyslet.

2.2 Grafové knihovny

Grafových knihoven existuje mnoho. Jsou vytvářeny pro různé programovací jazyky a s různou specializací. Některé slouží k vizualizaci grafů, jiné se zaměřují na operace s grafy a jsou optimalizovány pro maximální výkon, a jiné

slouží spíše k výukovým účelům. Uvedu zde jen pár nejznámějších s podobným zaměřením jako má tato práce:

BGL

The Boost Graph Library často zkracovaná autory na BGL je asi nejrozšířenější grafová knihovna pro C++ a je součástí oblíbené sady knihoven Boost. Klade si za cíl být co nejvíc generická a znovupoužitelná, proto hojně využívá šablony a celá implementace je v hlavičkových souborech. Tento požadavek však vede na poněkud obtížné používání a špatnou čitelnost kódu. Zdrojové kódy jsou veřejně dostupné a šiřitelné pod vlastní licencí Boost Software License.

BGL implementuje 2 reprezentace grafu (seznamy sousedů a matici incidence) a několik základních algoritmů, například

- prohledávání do šířky,
- prohledávání do hloubky,
- Dijkstrův algoritmus nejkratší cesty,
- Bellman-Fordův algoritmus nejkratší cesty,
- Johnsonsonův algoritmus nejkratší cesty,
- Kruskalův algoritmus nejmenší kostry,
- Primův algoritmus nejmenší kostry,
- topologické uspořádání,

postrádá však například algoritmy nad sítěmi.

LEMON

Library for Efficient Modeling and Optimization in Networks (zkráceně LEMON) je šablonová knihovna pro C++ poskytující datové struktury a algoritmy zaměřené kombinatorické optimalizační úlohy související s grafy a sítěmi. Je součástí projektu COIN-OR, komunitou vyvíjený otevřený matematický software, a dá se používat pro nekomerční i komerční účely pod licencí Boost Software License.

LEMON implementuje neorientovaný i orientovaný graf v několika variantách podle dostupných funkcí a míry efektivity (čím více funkcionalit, tím méně efektivní). Také obsahuje poměrně bohatou sadu algoritmů a datových struktur z následujících oblastí:

- prohledávání grafu,
- nejkratší cesty,

- kostry grafu,
- souvislost a komponenty grafu,
- toky a řezy v sítích,
- párování grafu,
- planární grafy.

LEDA

Library of Efficient Data types and Algorithms (zkráceně LEDA) je — jak již z názvu patrno — kolekce datových struktur a algoritmů pro jazyk C++, jejíž součástí jsou i grafy a grafové algoritmy. Knihovna je proprietární a podle vybrané edice (Free Edition, Professional Edition a Research Edition) poskytuje různě velké sady funkcí. Obsahuje poměrně velké množství grafových algoritmů, ty ale nejsou dostupné ve Free edici.

SNAP

Stanford Network Analysis Platform je víceúčelová vysokovýkonnostní C++ knihovna pro analýzu a manipulaci s velkými grafy a sítěmi. Poskytuje třídy pro neorientovaný graf, orientovaný graf a orientovaný multi-graf. Tato knihovna však nemá žádné grafové algoritmy. Zdrojové kódy jsou dostupné pod licencí BSD.

NGraph

Jednoduchá C++ knihovna pro práci s malými až středními grafy. Má jednoduché rozhraní a snadno se používá výměnou za méně funkcí a podprůměrný výkon. Knihovna je navržena spíše pro výukové účely než masivní nasazení v praxi, nejvíce se tedy podobá záměrům této práce.

JGraphT

Grafová knihovna pro Javu, která podporuje rozmanité typy grafů (grafy neorientované, orientované, vážené, nevážené, prosté grafy, multigrafy a další) a obsahuje základní grafové algoritmy. Dá se využít v kombinaci s knihovnou JGraph, která umí grafy vizualizovat a modifikovat pomocí grafického uživatelského rozhraní.

NetworkX

Grafová knihovna pro Python umožňující vytváření, manipulaci a vizualizaci grafů a multigrafů. Obsahuje také velkou škálu algoritmů. Je to jediná ve-

řejně dostupná knihovna, která implementuje Edmondsův algoritmus minimální kostry, ale kód je celý v Pythonu.

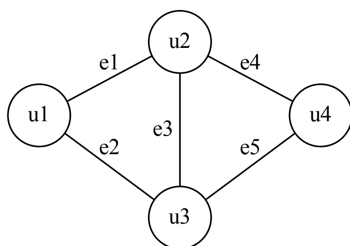
Kapitola **3**

Reprezentace grafů

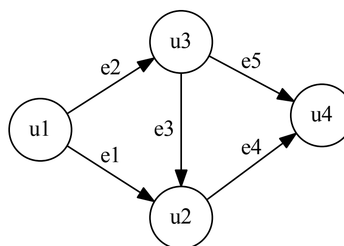
Reprezentace grafu je způsob, jakým jsou data grafu (uzly, hrany, váhy, ...) uloženy v paměti počítače a jakým jsou prováděny požadované operace (přidat uzel/hranu, odebrat uzel/hranu, projít sousedy uzlu, otestovat sousednost uzlů, ...). Různé reprezentace se liší v tom, kolik zaberou paměti a jak dlouho trvají operace v závislosti na počtu uzlů a hran.

3.1 Známé reprezentace

V literatuře (bude upřesněno u každého případu) lze nalézt následující možné reprezentace grafu. Pro jejich demonstraci budou použity grafy na obrázcích 3.1 a 3.2.



Obrázek 3.1: Příklad neorientovaného grafu



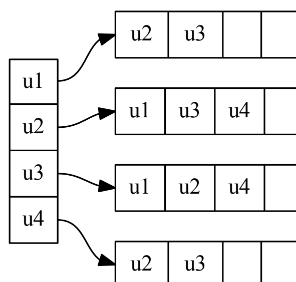
Obrázek 3.2: Příklad orientovaného grafu

nodes: u1, u2, u3, u4
u1 $\leftrightarrow$ u2
u1 $\leftrightarrow$ u3
u2 $\leftrightarrow$ u3
u2 $\leftrightarrow$ u4
u3 $\leftrightarrow$ u4

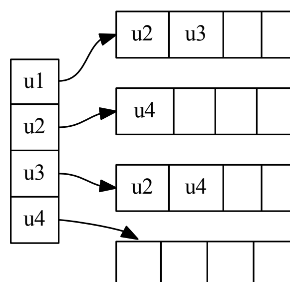
Obrázek 3.3: Seznam hran neorientovaného grafu 3.1

nodes: u1, u2, u3, u4
u1 $\rightarrow$ u2
u1 $\rightarrow$ u3
u3 $\rightarrow$ u2
u2 $\rightarrow$ u4
u3 $\rightarrow$ u4

Obrázek 3.4: Seznam hran orientovaného grafu 3.2



Obrázek 3.5: Seznamy sousedů neorientovaného grafu 3.1



Obrázek 3.6: Seznamy následníků orientovaného grafu 3.2

Seznam hran

Tato reprezentace uchovává seznam hran jako dvojic uzlů. Většinou se používá pro zadávání grafu na vstup nebo pro jeho uchovávání na disku. [3]

Příklad je na obrázku 3.3 a 3.4.

Seznamy sousedů

Tato reprezentace si pro každý uzel uchovává seznam jeho sousedů v neorientovaném grafu nebo seznam následníků (případně i předchůdců) v orientovaném grafu. Obecně platí, že je výhodná pro řídké grafy (grafy, ve kterých je každý uzel spojen hranou jen s několika uzly) a situace, kdy graf jednou vytvoříme a pak ho již nemodifikujeme, pouze prohledáváme. [3]

Reprezentace je znázorněna na obrázku 3.5 a 3.6.

Matice sousednosti

Matice sousednosti je čtvercová matice s rozměrem $|U|$. V neorientovaných grafech se v buňce $[u][v]$ nachází buď 1, když spolu uzly u a v sousedí, nebo 0 v opačném případě. V orientovaných grafech může být v buňce $[u][v]$ 1, pokud

	u1	**u2**	**u3**	**u4**
u1	0	1	1	0
u2	1	0	1	1
u3	1	1	0	1
u4	0	1	1	0

Obrázek 3.7: Matice sousednosti neorientovaného grafu 3.1

	u1	**u2**	**u3**	**u4**
u1	0	1	1	0
u2	0	0	0	1
u3	0	1	0	1
u4	0	0	0	0

Obrázek 3.8: Matice sousednosti orientovaného grafu 3.2

	e1	**e2**	**e3**	**e4**	**e5**
u1	1	1	0	0	0
u2	1	0	1	1	0
u3	0	1	1	0	1
u4	0	0	0	1	1

Obrázek 3.9: Matice incidence neorientovaného grafu 3.1

	e1	**e2**	**e3**	**e4**	**e5**
u1	1	1	0	0	0
u2	-1	0	-1	1	0
u3	0	-1	1	0	1
u4	0	0	0	-1	-1

Obrázek 3.10: Matice incidence orientovaného grafu 3.2

vede hrana z u do v, -1, pokud hrana vede z v do u, nebo 0, pokud žádná hrana mezi u a v není. Obecně se hodí spíše na husté grafy, nebo pro algoritmy, ve kterých v grafu často provádíme změny. [2]

Příklad najdete na obrázku 3.7 a 3.8.

Matice incidence

Matice incidence je obdélníková matice o $|U|$ řádcích a $|H|$ sloupcích. Pro orientovaný graf do buňky $[u][h]$ zapíšeme 1, pokud hrana h inciduje s uzlem u, nebo 0 v opačném případě. V orientovaném grafu píšeme 1, pokud hrana h vychází z uzlu u, -1, pokud do něj vstupuje, 0 jinak. [2]

Viz obrázek 3.9 a 3.10.

Srovnání

Tabulka 3.1 zachycuje paměťové náročnosti a časové složitosti nejčastěji používaných operací pro jednotlivé reprezentace. Uvedené časové složitosti neberou

Tabulka 3.1: Srovnání složitostí reprezentací

reprezentace	přidání hrany	test sousednosti	nalezení sousedů	paměťová náročnost
seznam hran	$\mathcal{O}(1)$	$\mathcal{O}(\|U\|)$	$\mathcal{O}(\|U\|)$	$\mathcal{O}(\|H\|)$
seznamy sousedů	$\mathcal{O}(1)$	$\mathcal{O}(adj(u))$	$\mathcal{O}(adj(u))$	$\mathcal{O}(\|U\| \cdot \|U\|)$
matice sousednosti	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\|U\|)$	$\mathcal{O}(\|U\|+\|H\|)$
matice incidence	$\mathcal{O}(1)$	$\mathcal{O}(\|H\|)$	$\mathcal{O}(adj(u) \cdot \|U\|)$	$\mathcal{O}(\|U\| \cdot \|V\|)$

v úvahu režii spojenou se zvětšováním polí. Fakt, že operace závisí na počtu sousedů konkrétního uzlu, je vyjádřen zápisem $\mathcal{O}(adj(u))$.

3.2 Původní řešení v ALib

V současné době jsou v knihovně implementovány seznamy sousedů a matice sousednosti. Vše je vytvořeno objektovým stylem s využitím dědění a skládání. David Rosca v [1] uvádí: „*Základní strukturou je třída Graph, od které dědí třídy DirectedGraph (pro orientované grafy) a UndirectedGraph (pro neorientované grafy). Třídy Node a DirectedEdge, UndirectedEdge představují uzly a hrany (orientované, resp. neorientované) v grafu.*“ Třídy `UndirectedGraph` a `DirectedGraph` obsahují ukazatel na objekt reprezentace. Ten může být pro neorientovaný graf instancí některé ze tříd

- `AdjacencyListUndirectedGraph`
- `AdjacencyMatrixUndirectedGraph`

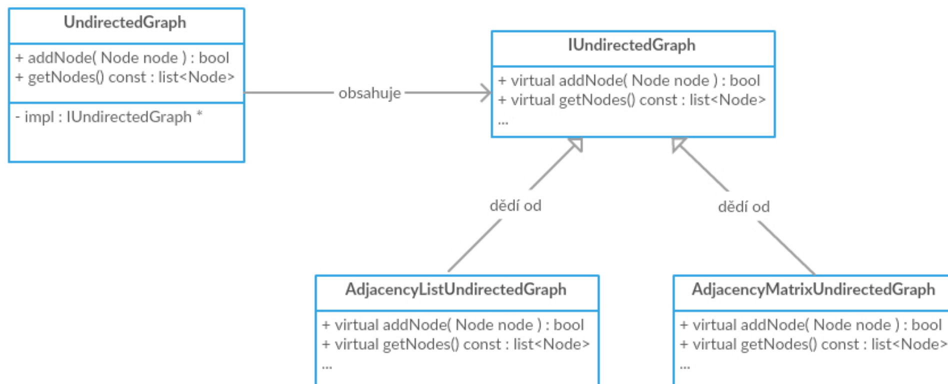
a pro orientovaný graf

- `AdjacencyListDirectedGraph`
- `AdjacencyMatrixDirectedGraph`

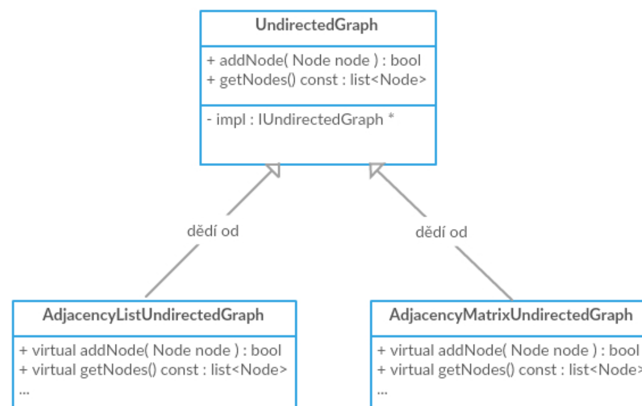
Tyto reprezentace jsou potomky abstraktních tříd `IUndirectedGraph` nebo `IDirectedGraph`. Situace je znázorněna na obrázku 3.11

Ohodnocení grafu

Ohodnocení je řešeno tak, že umožňuje přiřadit celočíselnou hodnotu libovolnému uzlu nebo hraně. Tento způsob má své výhody i nevýhody. Výhodou je, že zachovává jednoduchost a jistou univerzálnost návrhu. Nevýhody spočívají v tom, že algoritmus není schopen typem grafu říci, zda-li na vstupu požaduje graf s nějakým ohodnocením. Například algoritmus hledání nejkratší cesty požaduje, aby vstupní graf měl ohodnoceny všechny hrany. Toto se nyní musí kontrolovat za běhu programu – musí se ověřit, jestli každá hrana má přiřazenou hodnotu, a hlásit chybu, pokud nemá.



Obrázek 3.11: Předchozí vztahy tříd (podobně i pro orientovaný graf)



Obrázek 3.12: Nové vztahy tříd (podobně i pro orientovaný graf)

3.3 Provedené změny

Zjednodušení struktury

V původním řešením jsou vlastní reprezentace grafu obaleny další třídou `UndirectedGraph` nebo `DirectedGraph`, které nedělají skoro nic jiného, než delegují veškeré požadavky na konkrétní reprezentaci. Například takto:

```
bool UndirectedGraph::hasNode(const Node &node) const
{
    return impl->hasNode(node);
}
```

Toto obalení je zde zbytečné, proto se nově třídy reprezentací stávají potomky přímo tříd `UndirectedGraph`, `DirectedGraph`, jak je znázorněno na 3.12

Tato změna s sebou nese množství dalších drobných změn souvisejících s vnitřními mechanismy knihovny a s tím, že třídy `DirectedGraph` a `UndirectedGraph` se stávají abstraktními. Tyto změny nebudu dále rozebírat.

Ohodnocení grafu

Byl zde nápad zavést nové třídy pro následující typy grafu podle jeho vážení

- nevážený graf
- hranově vážený graf
- uzlově vážený graf
- plně vážený graf (vážené hrany i uzly)

ale následně bylo od tohoto záměru upuštěno, protože by značně zesložitil a znepřehlednil návrh, zejména proto, že každá varianta je potřeba pro neorientovaný i orientovaný graf. Nicméně není vyloučeno, že v budoucnu bude toto rozhodnutí přehodnoceno a v knihovně ALib se něco podobného objeví.

Místo toho byly přidány metody pro snadné zjištění, jestli jsou všechny hrany nebo všechny uzly ohodnocené.

Následníci a předchůdci

Doposud se v reprezentaci orientovaného grafu pomocí seznamu sousedů ukládali pouze následníci uzlu. Některé algoritmy (např. nalezení maximálního toku) však vyžadují efektivně projít i předchůdce uzlu. Z tohoto důvodu byla reprezentace předělána tak, aby každý uzel měl navíc seznam předchůdců, a metoda `getNeighbours` byla nahrazena dvojicí metod `getSuccessors` a `getPredecessors`.

Automatické odebírání uzlů

V původní verzi se nachází funkcionalita, která automaticky odebere uzel v případě, že odebereme poslední hranu s ním incidující. To neodpovídá obecnému pojetí grafu, ve kterém je možné mít i izolované uzly. Po dohodě s vedoucím byla tato funkce odstraněna.

Kapitola 4

Grafové algoritmy

Tato kapitola vysvětluje jednotlivé grafové problémy ze zadání a popisuje způsoby jejich řešení.

4.1 Nalezení minimální kostry neorientovaného grafu

Nalezení minimální kostry v neorientovaném grafu je velmi známý a v praxi se často vyskytující problém. Narazíme na něj například, pokud chceme propojit daná síťová zařízení (routery, switche, servery) co nejmenším množstvím kabelů.

Pro tento problém byla vynalezena celá řada algoritmů. Jako první objevil způsob řešení Otakar Borůvka. Na začátku algoritmu jsou všechny uzly samostatnou komponentou. V každé iteraci vybereme pro každou komponentu nejlevnější hranu spojující ji s jinou komponentou, a všechny je naráz přidáme do kostry. Postupně spojujeme komponenty, dokud nezůstane jediná. Nevýhodou algoritmu je, že funguje správně pouze na grafech, ve kterých lze minimální kostra určit jednoznačně, výhodou naopak to, že lze snadno paralelizovat. [3]

Dalším jednoduchým algoritmem je Primův/Jarníkův. Na rozdíl od Borůvkova algoritmu si udržuje pouze jednu komponentu souvislosti, která na začátku obsahuje jeden vybraný počáteční uzel. Tu v každém kroku rozšíří o nejlevější hranu vedoucí z komponenty do dalšího uzlu, dokud nepokryje všechny. [4]

Borůvkův algoritmus se do knihovny příliš nehodí kvůli jeho omezení a Jarníkův/Primův je již v ALib implementován. Dále tedy bude rozebrán a naprogramován třetí algoritmus – Kruskalův. Aby ho bylo možné implementovat efektivně, musíme se nejdříve podívat na jeden související problém – tzv. Union-Find problém.

Union-Find problém

Problému se také někdy říká dynamické udržování komponent souvislosti nebo množinový rozklad. Jde o to, jak efektivně určit, zda-li se 2 uzly nacházejí ve stejné komponentě, a případně sloučit 2 komponenty do jedné. Každé komponentě přiřadíme jeden uzel, kterému říkáme reprezentant komponenty. Řešení realizujeme dvojicí operací:

- `FIND`(u) vrátí reprezentanta komponenty obsahující uzel u.
- `UNION`(u, v) sjednotí dvě komponenty obsahující uzly u a v.

V [3] je popsáno několik řešení tohoto problému od jednoduchého po sofistikovanější.

Triviální řešení

Oindexujeme uzly čísly $1..n$, kde n je počet uzlů. Použijeme pole R o velikosti n a v i-té buňce udržujeme index reprezentanta komponenty s uzlem u_i. Operace `FIND`(u_i) jen vrátí $R[i]$, má tedy složitost $\mathcal{O}(1)$. Operace `UNION`(u_i, v_j) musí projít celé pole a všem uzlům s reprezentantem $R[v_j]$ ho změnit na hodnotu $R[u_i]$, to zabere $\mathcal{O}(n)$.

Často dostačující řešení

Předchozí řešení obohatíme o pole $size$, které pro daného reprezentanta r obsahuje v $size[r]$ velikost jeho komponenty, a pole $next$, které pro daný uzel obsahuje index dalšího uzlu ve stejné komponentě. Druhé pole simuluje spojové seznamy uzlů v jednotlivých komponentách ukončené nulou a říká nám, které položky máme projít při přepisování reprezentantů v `UNION`. Zároveň použijeme $size$ pro přepsání prvků v té menší komponentě. V nejhorším případě bude `UNION` trvat $\mathcal{O}(n)$, ale amortizovaně pouze $\mathcal{O}(\log n)$.

Přepojování stromů

Pro každou komponentu si budujeme orientovaný strom jejích uzlů, kde kořen je reprezentantem a hrany vedou od dětí k rodičům. Nejjednodušší je opět použít pole o velikosti n, kde hodnota buňky je index rodiče daného uzlu. Operace `FIND` projde stromem od uzlu ke kořeni a vrátí jeho index. Operace `UNION`(u, v) nejprve najde reprezentanty u a v a následně kořen jednoho stromu napojí pod kořen druhého stromu. Aby nedocházelo k nevyváženému růstu stromu do hloubky, pamatujeme si ještě u každého kořene hloubku jeho stromu $rank$, podle té vždy připojíme menší pod větší. Složitost obou operací tak shora omezíme na $\mathcal{O}(\log n)$.

Přepojování stromů se zkracováním cest

Předchozí řešení vylepšíme tak, že při každém provedení operace `FIND` napojíme všechny navštívené uzly stromu rovnou na jejich kořen. Při opakovaném volání operace na stejný uzel pak již dostáváme konstantní složitost.

Zvolené řešení a implementace

V knihovně bylo nakonec realizováno poslední z uvedených řešení. Jednak proto, že není nijak zvlášť komplikované a poskytuje dobrou složitost, a také proto, že se tento způsob učí v předmětu BI-GRA. Pro tento problém byla vytvořena třída `Components` s metodami realizující výše popsané operace a dodatečnou metodou `MakeSet`(u), která označí daný uzel za jednoprvkovou komponentu.

Kruskalův algoritmus

Popis

Kruskalův algoritmus spadá do kategorie tzv. hladových algoritmů. To znamená, že každém kroku vybírá aktuálně nejlepší možnost, tzv. lokální minimum, bez ohledu na to, co přijde později. Správnost tohoto přístupu pro hledání minimální kostry je dokázána v [2].

Na začátku každý uzel tvoří samostatnou komponentu. Následně prochází hrany v pořadí rostoucích vah a pokud jejím přidáním nevznikne kružnice, přidá se do kostry. Pro testování vzniku kružnice nám poslouží řešení Union-Find problému. Pokud se totiž oba krajní uzly hrany nacházejí ve stejné komponentě, pak nutně kružnice musí vzniknout. Naopak pokud jsou uzly z jiných komponent, pak se tyto komponenty přidáním hrany spojí v jednu.

Složitost algoritmu je dána složitostí řazení hran a operací `FIND` a `UNION`. Po seřazení musíme $2 \cdot |H|$ krát provézt nalezení komponenty a maximálně $|H|$ krát je sjednotit s amortizovanou složitostí $\mathcal{O}(\log |U|)$ Pro obecné řadící algoritmy se složitostí $\mathcal{O}(|H| \log |H|)$ jako např. QuickSort a výše vybrané řešení množinového rozkladu tedy vychází $\mathcal{O}(|H| \log |H|)$.

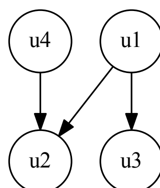
Pseudokód 1 Kruskalova algoritmu převzatý z [2].

Implementace

Pro efektivní procházení hran podle vah je nutné hrany nejprve seřadit. Na seřazení lze použít jakýkoliv efektivní řadící algoritmu, například QuickSort dostupný ve standardní knihovně C++ jako funkce `std::sort` z hlavičkového souboru `<algorithm>`. Zbytek je pouze přepis pseudokódu do syntace C++.

Algorithm 1 Kruskalův algoritmus

function Kruskal(G)
  $A \leftarrow \emptyset$
  for all $u \in U$ **do**
    MAKE_SET(u)
  seřaď H vzestupně podle vah
  for all hrana $\{u, v\} \in$ seřazené hrany **do**
    if FIND_SET(u) $\neq$ FIND_SET(v) **then**
      $A \leftarrow A \cap \{u, v\}$
      UNION(u, v);
  return A



Obrázek 4.1: Příklad neexistence orientované kostry

4.2 Nalezení minimální kostry orientovaného grafu

Kostra orientovaného grafu je pojem poněkud zavádějící. V anglické literatuře se místo toho používají pojmenování „arborescence“ a „branching“, které zatím nemají v českém jazyce ekvivalent. Jack Edmonds je v [6] definuje takto:

Definice 20. Branching je les (množina disjunktních stromů), jehož hran jsou orientovány tak, aby každá směřovala k jinému uzlu.

Definice 21. Arborescence je souvislý *branching*.

Z předchozího plyne, že *branching* je sjednocení disjunktních *arborescencí*. Optimální *branching* je potom takový, který má nejmenší možný součet vah jednotlivých hran. Tento problém je mnohem složitější než minimální kostra neorientovaného grafu a jeho řešení nemusí existovat pro každý orientovaný graf. Příklad, kdy nelze sestrojit *branching*, který pokryje celý graf, je na obrázku 4.1

Chu/Liu/Edmondsův algoritmus

Tento algoritmus byl nezávisle objeven několika lidmi v různých částech světa a je to jediný algoritmus řešící minimální kostru orientovaného grafu. Je značně

komplikovaný a byl postupně publikován v několika verzích.

Původní popis od Edmondse

Jack Edmonds v [6] popisuje svůj algoritmus velmi abstraktně, s pomocí čistě matematických úvah a zápisů, bez souvislostí s počítači a výpočetní náročností. V jeho popisu se vyskytuje několik datových struktur označovaných velkými písmeny, které nazývá jednoduše „bucket“, neboli kbelík. Jeho postup by se dal stručně shrnout takto:

Vytvoříme množinu kořenových komponent, do kterých zatím nevede žádná hrana. V této množině jsou na začátku všechny uzly. Postupně z ní vybíráme uzly a ke každému hledáme nejkratší hranu do něj vedoucí. Jestliže taková hrana neexistuje, uzel je kořen aktuálního stromu. Pokud nalezená hrana společně s předchozími nalezenými hranami vytvoří nový cyklus, zapamatujeme si hrany a uzly tvořící cyklus a nahradíme ho jediným uzlem. Takto pokračujeme, dokud není žádná komponenta s neprozkoumanými vstupními hranami. Následně se vrátíme ke vzniklým cyklům a vybíráme z nich hrany tak, aby splňovaly podmínky „branching“ a jejich váhy byly maximální.

Tarjanova implementace

O několik let později v [7] přichází Robert Tarjan s návrhem, jak implementovat Edmondsův algoritmus v počítačích, jaké použít datové struktury a jak konkrétně provést některé kroky. Podle něj je pro efektivní běh potřeba uchovávat:

- slabé komponenty pomocí tzv. *disjoint set* použité pro Union-Find problém,
- silné komponenty opět pomocí *disjoint set*,
- neprozkoumané hrany vstupující do každé silné komponenty pomocí slučovatelné prioritní fronty,
- kořenové komponenty, které mohou mít vstupující hrany,
- kořenové komponenty, které už nemají žádné vstupující hrany,
- pole obsahující pro každou silnou komponentu hranu, která do ní vstupuje,
- pole obsahující pro každou kořenovou komponentu její kořenový uzel

Dále tvrdí, že po proběhnutí algoritmu s výše popsanými strukturami lze finální *branching* získat prohledáním do hloubky z kořenových uzlů komponent, které nemají vstupující hrany. Tato implementace proběhne na daném grafu za $\mathcal{O}(|H| \log |U|)$.

V jeho díle také najdeme pseudokód podobný syntaxi jazyka Algol, který už se velmi blíží použitelné verzi algoritmu. Velká část pseudokódu 2 pochází z něj.

Camerinniho připomínky

O 2 roky později P. M. Cameriny ukázal v [8], že Tarjanův předpoklad existence jednoznačné cesty z uzlu kořenové komponenty do jiného uzlu nadřazené slabé komponenty byl chybný. Tím pádem nelze na závěr použít prohledání do hloubky, protože výsledek by nemusel být optimální.

Místo toho navrhuje použití lesa hran vstupujících do kořenových komponent. Optimální *branching* lze nakonec z tohoto lesa získat identifikací cest mezi listem a příslušným kořenem v lese. Konkrétní podoba postupu je vidět v pseudokódu 2.

Pseudokód a finální implementace

Pseudokód 2 a finální implementace v knihovně ALib jsou postaveny na poslední Tarjanově verzi se zahrnutými Camerinniho připomínkami. Původní pseudokód od Tarjana sice hledá kostru maximální, to lze ale jednoduše změnit na minimální obrácením podmínky haldy. Ta používá následující pojmenované datové struktury:

- **roots** uchovává kořenové komponenty, které mohou mít vstupující hrany,
- **rset** uchovává kořenové komponenty, které nemají vstupující hrany,
- **enter** má pro každé i hranu vstupující do komponenty i,
- **min** má pro každé i kořenový uzel komponenty i,
- **F** je les hran z Cameriniho připomínek,

a následující operace:

- WFIND(x) vrátí slabou komponentu obsahující uzel x,
- WUNION(a, b) spojí slabé komponenty obsahující uzly a, b,
- SFIND(x) vrátí silnou komponentu obsahující uzel x,
- SUNION(a, b) spojí silné komponenty obsahující uzly a, b,
- INIT(C, L) inicializuje frontu C aby obsahovala prvky z L,
- ADD(a, C) přičte hodnotu a k hodnotám všech prvků ve frontě C,
- MAX(C) smaže a vrátí prvek s největší hodnotou z fronty C,

- QUNION(C, D) přidá prvky z fronty D do fronty C a vyprázdní frontu D.
- $v(u, v)$ vrátí původní váhu hrany (u, v)
- $c(u, v)$ vrátí váhu hrany (u, v) upravenou pomocí ADD

Pro efektivní rozpoznání jak slabých tak silných komponent lze použít řešení Union-Find problému z Kruskalova algoritmu. Slučovatelná prioritní fronta je potřeba realizovat Binomiální nebo Fibonacciho haldou. V práci byly implementovány obě jako třídy `BinomialHeap` a `FibonacciHeap`, a to s pomocí popisů z [5], [9] a [10].

4.3 Nalezení maximálního toku v síti

Toky v sítích jsou v praxi jednou z velmi využívaných oblastí teorie grafů. Graf zde modeluje síť (např. dopravní, vodovodní, naftové, plynové, apod.), po níž se přepravuje určité médium. Přepravní kapacita je nějakým způsobem omezena a tím jsou omezeny i celkové přepravní možnosti.

V takovýchto sítích většinou chceme najít tok maximální, to jest takový tok, který má maximální možný součet toků v jednotlivých hranách.

Ford/Fulkersonův algoritmus

Popis

Algoritmus pracuje hledáním tzv. zlepšujících cest, to jsou neorientované cesty ze zdroje do spotřebiče, podél kterých lze upravit tok tak, aby výsledná velikost toku byla větší než předchozí. Na začátku nastavíme libovolný přípustný tok, například nulový. Dokud se daří nacházet zlepšující cesty, pak se zvyšuje tok v hranách těchto cest. Algoritmus končí poté, co není nalezena žádná další zlepšující cesta.

Hledání zlepšujících je speciální verzí prohledávání grafu od zdroje ke spotřebiči, které prochází následníky i předchůdce. Předchůdce je nutné procházet proto, že zlepšující cestu lze najít i tak, že se ubere tok v protisměru orientované hrany (u, v). Tím se zaručí zachování druhé podmínky toku pro uzel v a otevře se nová možnost, kudy vést cestu z uzlu u. Algoritmus nespecifikuje, v jakém pořadí navštěvovat otevřené uzly, stačí vybrat jakýkoliv otevřený uzel. Je tedy možné prohledávat do šířky, do hloubky i jakkoliv jinak.

Následně se ve všech hranách, které jsou součástí nalezené cesty, zvýší tok o stejnou hodnotu, která se určí jako minimum z rozdílů $q(u, v) - f(u, v)$ pro hrany (u, v) podél cesty. Tuto hodnotu buď určíme až těsně před zvyšováním toku projitím hran cesty nebo se průběžně počítá a uchovává v pomocném poli d při hledání cesty. V pseudokódu a finálním zdrojovém kódu je použita možnost druhá.

Algorithm 2 Chu/Liu/Edmondsův algoritmus

```
function OPTIMUM_BRANCHING(G)
    roots ← ∅
    for i ← 1 until n do INIT(i,I(i))
        Vytvoř S-množinu obsahující i jako jediný prvek
        Vytvoř W-množinu obsahující i jako jediný prvek
        enter[i] ← (0, 0)
        roots ← roots ∪ {i}
        min[i] ← i
    H ← ∅
    rset ← ∅
    while roots ≠ ∅ do
        Vyjmi některý kořen k z roots
        hrana (i, j) ←MAX(k)
        if v(i, j) ≤ 0 then rset ← rset ∪ {k}
        else if SFIND(i)= k then roots ← roots ∪ {k}
        else
            H ← H ∪ {(i, j)}
            if WFIND(i)≠WFIND(j) then WUNION(i,j)
                enter[k] ← (i, j)
            else
                val ← inf
                hrana (x, y) ← (i, j)
                while (x, y) ≠ (0, 0) do
                    if c(x, y) < val then
                        val ← c(x, y)
                        vertex ←SFIND(y)
                    (x, y) ← enter[SFIND(x)]
                ADD(val − c(i, j),k)
                min[k] ← min(vertex)
                (x, y) ← enter[SFIND(i)]
                while (x, y) ≠ (0, 0) do
                    ADD(val − c(x, y), SFIND(y))
                    QUNION(k, SFIND(y))
                    SUNION(k, SFIND(y))
                    (x, y) ← enter[SFIND(y)]
                roots ← roots ∪ {k}
    R ← ∀min[i] : i ∈ rset
    B ← ∅
    N ← kořenové uzly lesa F
    repeat
        if R ≠ ∅ then
            Smaž kořen v z R
        else
            Vezmi kterýkoliv kořen (w, v) ∈ N a přidej ho do B
        Najdi cestu P v lese F vedoucí z kořene do listu (u, v)
        Smaž z F všechny uzly a hrany z P a aktualizuj N
    until R = N = ∅ return B
```

Složitost této verze algoritmu nelze určit jednoznačně. Nalezení zlepšující cesty trvá $\mathcal{O}(|U|+|H|)$ a zvýšení toku $\mathcal{O}(|U|)$. Není ale možné určit, kolikrát se tyto kroky musí provést. Na iracionálních číslech algoritmus dokonce nemusí skončit nikdy.

Pseudokód 3 je převzatý z [2].

Implementace

Pro hledání zlepšující cesty bylo zvoleno prohledávání do šířky, pro procházení uzlů je tedy potřeba použít fronta otevřených uzlů. Pro jednoduchost lze vzít `std::queue` z C++ STL, i když rychlostně výhodnější by bylo napsat vlastní jednodušší třídu s jednorázovou alokací pole, bez kruhového posunování indexu začátku a konce a bez kontrol podtečení.

Dále se pseudokód snaží poměrně nepřehledným způsobem uchovat naráz 2 informace v poli p, a to index předchůdce uzlu a znaménko změny toku v hraně. V implementaci bylo toto pole rozděleno na dvě, jedno ukládající předchůdce a druhé pro znaménko.

4.4 Nalezení minimálního řezu v síti

Minimální řez počítáme tehdy, když chceme zjistit, jak co nejjednodušeji nebo nejlevněji oddělit 2 elementy v síti.

Algoritmus využívající maximální tok

Popis

Minimální řez v síti úzce souvisí s maximálním tokem. Tento vztah popisuje Ford-Fulkersonův teorém:

Věta 1. *Velikost maximálního toku v síti je rovna kapacitě jejího minimálního řezu.*

Z důkazu této věty v [2] lze vyvodit následující postup. Nejdříve necháme proběhnout algoritmus Forda/Fulkersona pro maximální tok. Potom ještě jednou provedeme prohledání grafu stejným způsobem jako při hledání zlepšující cesty a označíme všechny uzly, do kterých jsme se dostali ze zdroje. Následně přidáme do řezu ty hrany, které spojují tyto označené uzly s neoznačenými.

Viz pseudokód 4.

Implementace

I zde je možné projít komponentu obsahující zdroj jakýmkoliv prohledáváním, opět jsem zvolil prohledávání do šířky. Odebrané hrany jsou ukládány do `std::vector` a návratová hodnota obsahující hrany minimálního řezu je realizována pomocí `std::unordered_set`.

Algorithm 3 Ford/Fulkersonův algoritmus

```
function FORD_FULKERSON(S)
    for all hrana (u, v) ∈ H do
        f(u, v) ← 0
    while NAJDI_CESTU(S) do
        ZVYŠ_TOK(S)
    return f

function NAJDI_CESTU(S)
    for all uzel u ∈ U do
        stav[u] ← FRESH
    p[s] ← +s
    d[s] ← inf
    stav[s] ← OPEN
    repeat
        u ← libovolný otevřený uzel
        stav[u] ← CLOSED
        for all v ∈ následníci uzlu u do
            if stav[v] = FRESH ∧ f(u, v) < q(u, v) then
                stav[v] ← OPEN
                p[v] ← +u
                d[v] ← MIN(d[u], q(u, v) − f(u, v))
        for all v ∈ předchůdci uzlu u do
            if stav[v] = FRESH ∧ f(v, u) > 0 then
                stav[v] ← OPEN
                p[v] ← −u
                d[v] ← MIN(d[u], f(v, u))
    until neexistuje otevřený uzel ∨ u = t

function ZVYŠ_TOK(S)
    v ← t
    δ ← d[t]
    repeat
        u ← abs(p[v])
        if p[v] = u then
            f(u, v) ← f(u, v) + δ
        else
            f(v, u) ← f(v, u) − δ
        v ← u
    until v = s
```

Algorithm 4 Nalezení minimálního řezu pomocí maximálního toku

```
function Najdi_řez(S)
    C ← ∅
    f ← Ford_Fulkerson(S)
    for all uzel u ∈ U do
        stav[u] ← FRESH
    stav[s] ← OPEN
    repeat
        u ← libovolný otevřený uzel
        for all v ∈ následníci uzluu do
            if stav[v] = FRESH ∧ f(u, v) < q(u, v) then
                stav[v] ← OPEN
        for all v ∈ předchůdci uzlu u do
            if stav[v] = FRESH ∧ f(v, u) > 0 then
                stav[v] ← OPEN
    until neexistuje otevřený uzel
    for all hrana (u, v) ∈ H do
        if (stav[u] ≠ FRESH ∧ stav[v] = FRESH) ∨ (stav[u] = FRESH ∧
stav[v] ≠ FRESH) then
            C ← C ∪ (u, v)
    return C
```

4.5 Nalezení maximálního toku v neorientované síti

Tento problém je poněkud netradiční. V literatuře nelze nalézt oficiální algoritmus na jeho řešení. Teorie grafů totiž nezná pojem „neorientovaná síť“. V praxi ale můžeme chtít toky počítat i v neorientovaných grafech, proto se vyplatí se tímto problémem zabývat. Neorientovanou síť si zavedeme takto:

Definice 22 (neorientovaná síť)**. Neorientovanou sítí** nazýváme čtveřici $S = (G, q, s, t)$, kde $G = (H, U)$ je obyčejný neorientovaný graf, $s \in U$ je **zdroj sítě** S, $t \in U$ je **spotřebič sítě** S a $q : H \mapsto R^+$ je nezáporné ohodnocení hran nazývané **kapacita sítě** S. Pro každou hranu h nazýváme příslušnou hodnotu $q(h)$ **kapacitou hrany** h.

Upravený algoritmus Forda-Fulkersona

Popis

Algoritmus je jednoduchou modifikací Ford-Fulkersonova algoritmu pro orientovanou variantu. Nejdříve se každá neorientovaná hrana nahradí dvojicí

protichůdných orientovaných hran. Následně ve fázi aktualizace toků podél zlepšující cesty se vždy aktualizují obě hrany opačnou hodnotou. [3]

Algorithm 5 Ford/Fulkersonův algoritmus pro neorientovanou síť

```
function FORD_FULKERSON(S)
    for all hrana (u, v) ∈ H do
        f(u, v) ← 0
    while NAJDI_CESTU(S) do
        ZVYŠ_TOK(S)
    return f

function NAJDI_CESTU(S)
    for all uzel u ∈ U do
        stav[u] ← FRESH
    p[s] ← s
    d[s] ← inf
    stav[s] ← OPEN
    repeat
        u ← libovolný otevřený uzel
        stav[u] ← CLOSED
        for all v ∈ sousedé uzlu u do
            if stav[v] = FRESH ∧ f(u, v) < q(u, v) then
                stav[v] ← OPEN
                p[v] ← u
                d[v] ← MIN(d[u], q(u, v) − f(u, v))
    until neexistuje otevřený uzel ∨ u = t

function ZVYŠ_TOK(S)
    v ← t
    δ ← d[t]
    repeat
        u ← p[v]
        f(u, v) ← f(u, v) + δ
        f(v, u) ← f(v, u) − δ
        v ← u
    until v = s
```

Implementace

Pro převod neorientovaného grafu na orientovaný byl třídě `DirectedGraph` přidán konstruktor s parametrem `UndirectedGraph`. Zbytek je stejný jako u orientované verze.

4.6 Nalezení minimálního řezu v neorientované síti

Povaha tohoto problému je stejná jako problému předchozího. Abychom zprovoznili algoritmus pro neorientované sítě, musíme provést několik úprav.

Upravený algoritmus minimálního řezu v síti

Opět vyjdeme ze vztahu s maximálním tokem, ten platí i zde (podle [3]). Graf upravíme stejným způsobem jako u upraveného Ford/Fulkersonovo algoritmu a ten necháme proběhnout. Dále postupujeme stejně jako u orientované verze.

Algorithm 6 Nalezení minimálního řezu v neorientované síti

```
function NAJDI_ŘEZ(S)
    C ← ∅
    f ← FORD_FULKERSON(S)
    for all uzel u ∈ U do
        stav[u] ← FRESH
    stav[s] ← OPEN
    repeat
        u ← libovolný otevřený uzel
        for all v ∈ následníci uzluu do
            if stav[v] = FRESH ∧ f(u, v) < q(u, v) then
                stav[v] ← OPEN
    until neexistuje otevřený uzel
    for all hrana {u, v} ∈ H do
        if (stav[u] ≠ FRESH ∧ stav[v] = FRESH) ∨ (stav[u] = FRESH ∧ stav[v] ≠ FRESH) then
            C ← C ∪ {u, v}
    return C
```

Kapitola 5

Testování

Testování algoritmů není na rozdíl od běžného softwaru ve světě nijak standardizováno. Je potřeba si poradit „ad hoc“ pro každý případ zvlášť.

Typicky se vymyslí několik jednoduchých případů, které lze ověřit ručně na papíře, a výstup algoritmu se porovná pevně s zadanými hodnotami. Je dobré do těchto testů zahrnout několik případů, které jsou pro daný algoritmus považovány za krajní, například řadícímu algoritmu dáme zcela seřazené pole nebo naopak pole s přesně opačným pořadím.

Pokud máme k dispozici referenční řešení, o kterém víme, že je správně, potom můžeme testovat velké náhodné vstupy a výsledky nového algoritmu porovnat s tím referenčním. V tomto případě taková možnost není, vyžadovalo by to instalaci knihovny třetí strany, která by mohla porušit licenci knihovny ALib. Náhodnými vstupy bude tedy otestováno pouze, zda-li algoritmus skončí za daný čas a zda-li doběhne bez chyby.

V knihovně ALib se pro testování tříd a různých funkcionalit používá nástroj Cpp Unit. Jedná se o sadu maker pro zjednodušení psaní testů a systém pro spouštění testů a odchytávání chyb. I v této práci byl Cpp Unit použit.

Aby se pro každý testovaný graf nemusel opakovat kód, který přidává uzly, hrany a váhy, volá algoritmus a následně položku po položce kontroluje jeho výstup, je pro každý z algoritmů napsána jedna funkce pro test zadaného grafu a data grafu jsou specifikována formou staticky inicializovaných polí.

5.1 Test Kruskalova algoritmu

Výsledkem Kruskalova algoritmu je kostra. Vstupem i výstupem je tedy graf. Bohužel kostra neorientovaného grafu nemusí být vždy jednoznačná, proto nelze použít již hotový Primův algoritmus a porovnávat výsledky s ním. Je potřeba výsledky vypočítat ručně a zadat je do programu. Kontroluje se velikost množiny hran a existence jednotlivých hran ze zadaného očekávaného výsledku.

5.2 Test Edmondsova algoritmu

Výsledkem je opět kostra, což je graf. Testování probíhá stejným způsobem jako algoritmu předchozího.

5.3 Test Ford/Fulkersonova algoritmu pro tok sítě

Tok sítě jsou hodnoty přiřazené jednotlivým hranám. Z funkce algoritmu přijdou jako dvourozměrná tabulka hodnot pro dvojice uzlů. Správné řešení je zadáno jako pole hran a jejich hodnot. Pro každou hranu a její krajní uzly se ověří, jestli je hodnota z tabulky správná.

5.4 Test Ford/Fulkersonova algoritmu pro řez sítě

Řez je množina hran. Musí se tedy ověřit, jestli obsahuje všechny hrany, které má mít, a žádné navíc. Správné řešení je zadáno jako pole hran. Testuje se tedy, jestli se ve výstupu vyskytuje každá hrana z pole, a jestli je počet prvků řešení stejný jako velikost pole.

5.5 Test upravených algoritmů pro tok a řez neorientované sítě

Algoritmy pro tok a řez byly vytvořeny tak, že existují jen 2 třídy, jedna pro tok a druhá pro řez. Ty mají přetížené metody pro orientovaný a neorientovaný graf. Aby byl dodržen systém testování v knihovně ALib, kde každý soubor s třídou má jeden test s odpovídajícím názvem, testují se orientované i neorientované verze zároveň. Zadá se jenom jeden graf a k němu dvě řešení pro každou verzi algoritmu. Zároveň se tím zkrátí kód.

Závěr

Cílem bylo rozšířit knihovnu ALib o sadu algoritmů z teorie grafů, zajistit, aby datové struktury reprezentující grafy vyhovovaly těmto algoritmům, a ověřit správnost implementace vhodnými testy. Tento cíl byl splněn. Algoritmy byly úspěšně implementovány, integrovány do knihovny a na základě testů byly opraveny vzniklé chyby.

Co se týče grafových struktur, jejich implementace tak, aby byly univerzální, snadno použitelné a zároveň rychlé, je obecně velmi složitý problém. Objevuje se zde mnoho požadavků, které jsou často protichůdné, což potvrzuje i různorodost existujících řešení. Rozhodně je zde prostor k dalšímu vylepšování. Na toto téma by se možná nechala vypracovat celá závěrečná práce.

Momentálně se v knihovně nachází několik základních a 2 složitější grafové algoritmy. Nic nebrání tomu, aby byly přidány další.

Literatura

[1] ROSCA, David. *Automatová knihovna - isomorfismus planárních grafů.* Praha, 2015. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií.

[2] KOLÁŘ, Josef. *Teoretická informatika.* V Praze: České vysoké učení technické, 2009. ISBN 978-80-01-04331-8.

[3] ČERNÝ, Jakub. *Základní grafové algoritmy* [online]. Praha, 2010 [cit. 2016-05-05]. Dostupné z: `http://kam.mff.cuni.cz/~kuba/ka/ka.pdf`. MFF UK.

[4] MAREŠ, Martin. *Krajinou grafových algoritmů: průvodce pro středně pokročilé.* Praha: ITI, 2007. ISBN 978-80-239-9049-2.

[5] CORMEN, Thomas H. *Introduction to algorithms.* 3nd ed. Cambridge, Mass.: MIT Press, c2001. ISBN 0-07-013151-1.

[6] EDMONDS, Jack. Optimum Branchings. *Jour. of Research of The National Bureau of Standards.* 1967, č. 71B, str. 233–240

[7] TARJAN, Robert. Finding Optimum Branchings. *Networks.* 1977, č. 7, str. 25–35.

[8] CAMERINI, P. M., FRATTA, L., MAFFIOLI, F. A Note on Finding Optimum Branchings. *Networks.* 1979, č. 9, str. 309–312.

[9] TVRDÍK, Pavel. *Pokročilé haldy 1* [online]. 2015 [cit. 2016-04-7]. Dostupné z: `https://edux.fit.cvut.cz/courses/BI-EFA/_media/lectures/bi-efa2015prednaska6-heaps2-anim.pdf`

[10] TVRDÍK, Pavel. *Pokročilé haldy 2 a Úvod do vyhledávání* [online]. 2015 [cit. 2016-04-7]. Dostupné z: `https://edux.fit.cvut.cz/courses/BI-EFA/_media/lectures/bi-efa2015prednaska6-heaps2-anim.pdf`

Příloha **A**

Obsah přiloženého CD

```
readme.txt ........................ stručný popis obsahu CD
automata-library ..... zdrojové kódy implementace vyňaté z repozitáře
    alib2algo ........................ grafové algoritmy
    alib2data ............. datové struktury reprezentující grafy
text ................................. text práce
    obrazky ........... obrázky použité v práci a jejich zdrojové kódy
    thesis.pdf ................. text práce ve formátu PDF
    thesis.tex .......... zdrojová forma textu ve formátu LaTeX
```