Documentace knihovny flows

Pavel Klavík

zápočtový program k NPRG030 ZS 2007/08

Obsah

1 Teoretický pohled na toky v síti

1.1 Zavedení maximálního toku

Představme si, že máme potrubní *síť*. Taková síť se skládá z jednosměrných trubek, kterými voda může téci jenom jedním směrem, a uzlů. Uzel je místo, kde se síť větví a setkávají se tam konce trubek. Každá trubka má svoji *průtokovou kapacitu*, tedy kolik litrů vody jí může protéct za určitý časový okamžik. Žádným způsobem nemůžeme tuto kapacitu překročit. Zároveň je síť dobře utěsněná, tedy když do libovolného uzlu přiteče určité množství vody, přesně tolik také odteče (podobně uvnitř trubky se nic nikde neztratí).

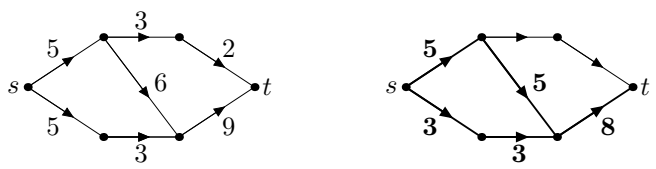
V síti budeme mít dva významné uzly, říkejme jim *zdroj* (označuje se s z anglického source) a *spotřebič* (t jako target). Zdroj je schopný do sítě vpustit libovolně velké množství vody a spotřebič je schopen naopak libovolné množství pohltit (spotřebovat). Aby naše síť byla zajímavá, nechť je zdroj různý od spotřebiče. Ukázka sítě je na obrázku 1.

Tok je ohodnocení trubek (hran), které říká, kolik litrů jimi protéká. Musí však splňovat dvě podmínky. Žádnou trubkou neteče více, než je její kapacita (a pochopitelně minimálně nula). Druhá podmínka říká, že co do vrcholu přiteče, to také odteče, ovšem s vyjímkou zdroje a spotřebiče. Velikost toku si lze definovat třeba jako počet litrů, které vytékají ze zdroje mínus počet litrů, které zpět přítékají.

Maximální tok je potom zjevně takový, jehož velikost je maximální. Na obrázku 1 vpravo je vyznačen maximální tok v síti. A právě účelem této knihovny je hledání maximálního toku v síti.

S toky v síti souvisí na první pohled dosti odlišný pojem *řez grafu*. Řez je věrný svému jménu a můžeme si ho představit tak, že graf rozřízneme podél některých hran. Tím rozdělíme vrcholy grafu na dvě skupiny, formálně je tedy řez nějaká podmnožina vrcholů grafu. Řez lze také popsat pomocí rozříznutých hran, to jsou ty hrany, které spojují vrcholy dvou různých skupin. V dalším textu, pokud nebude uvedeno jinak, vždy uvažujme řez takový, že zdroj je v jedné skupině (označme si ji A) a spotřebič ve druhé (A'). Takovému řezu se říká *tokový řez*. Hrany řezu také můžeme rozdělit na hrany vedoucí vlevo (to jsou ty ze skupiny A do skupiny A') a hrany vpravo (které vedou naopak z A' do A).

Kapacitou řezu se myslí součet kapacit hran vedoucích vlevo. *Minimální řez* je takový řez, jehož kapacita je minimální. Také se hodí zavést si *velikost toku*



Obrázek 1: Vlevo je síť a vpravo vyznačený maximální tok.

přes řez. Je to součet toho, co protéká směrem vlevo mínus to, co teče vpravo.

Z výše uvedených pojmů plynou dvě pěkná pozorování:

- **Velikost toku přes libovolný tokový řez je stejná.** Námi zavedená velikost toku je velikost toku přes řez, kde skupinu A tvoří jenom zdroj. Naopak velikost toku přes řez, který není tokový, je nulová.
- **Velikost toku přes libovolný řez je menší než kapacita libovolného řezu.** Nabízí se na první pohled netriviální otázka, zda někdy může nastat rovnost? Ano, pokud tok je maximální a řez minimální. Dokonce platí, že pokud nalezneme maximální tok, tak máme také minimální řez, tedy algoritmus na hledání maximálního toku je samoverifikační.

Hledání maximálního toku má velice pěkné aplikace a řada i velice odlišných úloh se na něj nechá převést. Například můžeme pracovat s železniční sítí. Uzly budou nádraží a trubky budou vlakové spoje. Kapacita trubky pak bude maximální počet pasažerů, který jsme schopni přepravit. Velikost maximálního toku pak bude počet pasažérů, které jsme schopni přepravit od zdroje ke spotřebiči. Pomocí hledání maximálního toku lze však řešit odlišnější úlohy, například maximální párování v bipartitním grafu. Více informací o aplikacích toků je v kapitole 4.

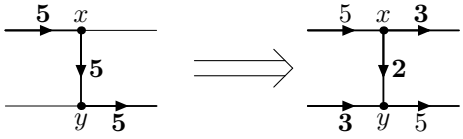
Zbývá ovšem vyřešit problém největší, tedy jak budeme maximální tok hledat. Hledání maximálního toku je problém řešitelný v polynomiálním čase i paměti. Popišme si bez důkazu hlavní myšlenky různě efektivních algoritmů.

1.2 Ford-Fulkersonův algoritmus

Zaveďme si nejprve několik pojmů. *Rezervou hrany* myslíme rozdíl její kapacity a toho, co skrz ní už protéká. Potom *zlepšující cesta* je cesta vedoucí ze zdroje do spotřebiče, jejíž všechny hrany mají kladnou rezervu. Po takové cestě tedy ještě můžeme poslat minimum z rezerv jednotek vody a vytvoříme tak lepší tok.

Co kdybychom maximální tok v síti zkusili hledat hladově? Tedy dokud existuje zlepšující cesta ze zdroje ke spotřebiči, pošleme po ní, co se dá. Pokud není, nalezli jsme maximální tok. Tento algoritmus však nefunguje a je jednoduché nalézt protipříklad. Stačí si vzít síť z obrázku 1 s kapacitami všech hran rovnými jedné. Pokud budeme mít smůlu a jako první blokující cestu si zvolíme tu využívající mezihranu směrem dolů, zablokujeme si obě cesty (horní i dolní) a nenalezneme tak maximální tok.

Ford-Fulkersonův algoritmus přichází s jednoduchou úpravou. Zlepšující cesta může vést i proti směru hrany. Rezerva takové hrany je pak rovna velikosti toku přes ní a pokud se rozhodneme přes ni poslat zlepšující cestu, tok přes ní naopak snižujeme. Pro lepší představu se podívejte na obrázek 2. Nejprve vede zlepšující cesta přes vrchol x do y a pošleme přes ní 5 jednotek. Poté jsme nalezli podle výše uvedeného pravidla zlepšující cestu přes y do x, po které můžeme poslat 3 jednotky. Avšak tok po hraně z x do y se nezvýší, ale naopak z x do y pošleme o 3 jednotky méně, tedy zbudou pouze 2. Všimněme si, že jsme tím neporušili ani jednu z podmínek toku. Vše nyní funguje správně.



Obrázek 2: Zlepšující cesta proti směru hrany.

Následuje několik pozorování o Ford-Fulkersonově algoritmu:

- **Pro racionální kapacity se Ford-Fulkersonův algoritmus zastaví.** S každou zlepšující cestou totiž zvětšíme velikost toku alespoň o jedničku (pro celočíselné kapacity, racionální prostě přenásobíme na celočíselné), tedy po konečně mnoha krocích nenajdeme žádnou zlepšující cestu. Tedy algoritmus doběhne v konečném čase.

- **Algoritmus nám současně najde minimální řez roven nalezenému toku,** tím je tedy dokázána jeho správnost. Nechť se Ford-Fulkersonův algoritmus zastavil. Podívejme se na řez tvořený všemi vrcholy, pro které existuje zlepšující cesta ze zdroje. Zjevně zdroj tam leží a naopak spotřebič ne (jinak by se algoritmus nezastavil). Potom hrany řezu mají všechny nulovou rezervu (a tedy velikost řezu je rovna velikosti toku), protože jinak by to byl spor se zvoleným řezem, šlo by tam vrchol přidat.

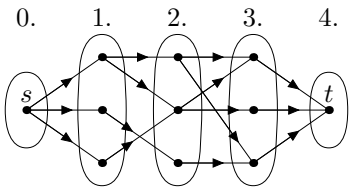
- **V síti s celočíslenými kapacitami** nám algoritmus nalezne celočíselný maximální tok, protože během výpočtu se nikdy neceločíselný tok nevytvoří.

A jak budeme hledat zlepšující cestu? Prostě projdeme graf do šířky. První nalezenou zlepšující cestu, která je současně nejkratší, použijeme. Tento algoritmus má speciální název *Edmonsův-Karpův algoritmus*. Lze dokázat, že při tomto postupu vybírání cest není časová složitost shora závislá na hodnotách kapacit hran (tedy můžeme si jenom pomoci). Asymptotická časová složitost algoritmu je $\mathcal{O}(NM^2)$, paměťová $\mathcal{O}(N+M)$, kde N je počet vrcholů a M je počet hran. Jeho implementace není obtížná.

1.3 Dinicův algoritmus a Algoritmus tří Indů

Dinicův algoritmus je založen na principech Ford-Fulkersonova algoritmu a přichází s následující myšlenkou. Nebudeme se snažit přidávat pouze zlepšující cesty, přidáme najednou celé zlepšující toky. Tím si určitě správnost algoritmu nepokazíme. Jaké toky však přidávat? Určitě by měly být lepší jak zlepšující cesta, ale zase to nemůže být maximální tok. Vhodným kompromisem je takzvaný *blokující tok*. Blokující tok je to, co bychom dostali při použití Ford-Fulkersonova algoritmu bez procházení hran v protisměru. Navíc aplikujeme tento postup na rozvrstvenou a začištěnou síť.

Co je to *rozvrstvená síť*? Vždy nalezneme průchodem do šířky nejkratší zlepšující cestu. Rozvrstvená síť bude síť, jejíž vrstvy budou tvořeny vrcholy, které



Obrázek 3: Rozvrstvená a začištěná síť – zde probíhá výpočet.

jsou stejně daleko do zdroje po nejkratší zlepšující cestě. A protože chceme v takové síti hledat zlepšující cestu hladově, potřebujeme si ji udržovat začištěnou. To znamená, že odmažeme všechny vrcholy nulového výstupního či vstupního stupně (a to nám může vytvořit nové, proto proces začišťování pokračuje, dokud takový vrchol v síti existuje. Vrcholy nulového vstupního stupně se vlastně ani nemusíme zabývat, do nich se nikdy při hladovém procházení sítí nemůžeme dostat. Při zvýšení toku o zlepšující cestu nám mohou hrany zaniknout (už jsme po nich poslali co šlo), proto musíme dále síť průběžně začištovat. Ukázku rozvrtvené a začištěné sítě naleznete na obrázku 3.

Správnost Dinicova algoritmu plyne ze správnosti Ford-Fulkersona. Navíc pro něj platí několik zajímavých věcí:

- **Začištování sítě lze výrazně zjednodušit.** Dokážeme ho provádět při procházení do hloubky. Prostě jdeme rovnou za nosem ke zdroji a když skončíme ve slepé uličce, tak se budeme vracet zpátky a současně cestu umazávat.

- **Délka nejkratší zlepšující cesty vzroste** po každém přidání zlepšujícího toku. Z toho plyne velice pěkná časová složitost algoritmu, asymptoticky $\mathcal{O}(N^2M)$ a pro řadu grafových tříd se chová mnohem lépe.

Algoritmus tří Indů je upravený Dinicův algoritmus, vymysleli ho Malhotra, Kumar a Maleshwari a umožňuje nám hledat blokující tok v čase $\mathcal{O}(N^2)$ (oproti výše uvedené implementaci Dinicova algoritmu v čase $\mathcal{O}(NM)$).

Pro každý vrchol rozvrstvené sítě si spočteme jeho *vstupní* a *výstupní rezervy* a v průběhu výpočtu si je budeme udržovat aktualizované. *Rezervou* vrcholu pak bude minimum z těchto dvou hodnot.

Blokující tok pak hledáme tak, že vždy vezmeme vrchol s nejmenší rezervou a protlačíme přes něj tuto rezervu od zdroje ke spotřebiči. Protlačení uděláme nejprve od zdroje k němu rekurzivně, že si vezmeme dostatek jeho předchůdců v nižší vrstvě a přes každý pošleme dost velký tok, aby součet dal onu rezervu. Takto postupujeme po vrstvách až ke zdroji, podobně to uděláme se spotřebičem. Protlačení projde právě proto, že protlačujeme minimální rezervu, tedy nemůžeme se v žádném vrcholu zaseknout. Na konci výpočtu pak vymažeme vrcholy, jejichž rezerva klesla na nulu, síť začistíme a pokračujeme, dokud existuje vrchol s kladnou rezervou. Výsledkem je nalezený blokující tok.

Časová složitost Dinicova algoritmu v kombinaci s Algoritmem tří Indů je pouze $\mathcal{O}(N^3)$.

1.4 Goldbergův algoritmus

Poslední algoritmus je mnohem jednodušší než Dinicův, přesto je nejrychlejší. Je založený na zcela odlišných principech. Snažíme se celý maximální tok postupně převádět od zdroje ke spotřebiči. Nejprve si budeme muset definovat několik pojmů, pak si popíšeme, jak algoritmus funguje.

Přebytek vrcholu je rozdíl toho, co do něj přiteče a toho, co z něj vyteče. V případě toku musí být přebytek ve všech vrcholech výjma zdroje a spotřebiče (označme takový vrchol jako *vnitřní*) nulový. Definujme si obecnější pojem než tok. *Vlna* je stejně jako tok ohodnocení hran. Stejně jako tok musí splňovat podmínku o kapacitě hran, ale stačí nám, aby přebytek v libovolném vnitřním vrcholu byl nezáporný. Vlna je tokem právě tehdy, když přebytek v libovolném vnitřním vrcholu je nulový.

V každém kroku výpočtu budeme pracovat s vlnou, na konci výpočtu nám z ní vznikne tok. Myšlenka je taková, že si vždy zvolíme nějaký vrchol s přebytkem a pokusíme se přebytek převést na jeho sousedy. K tomu, aby se nám výpočet nezacyklil, využíváme *výšky vrcholu*. Každému vrcholu přiřadíme jedno přirozené číslo, které bude jeho výškou. Obecný princip převádění přebytků je ten, že voda teče pouze z kopce, proto lze přebytek převádět pouze z vrcholu s vyšší výškou do vrcholu s nižší výškou. Proto, když se chceme zbavit přebytku v nějakém vrcholu, tak nalezneme nějakého níže položeného souseda s rezervou. Pokud takový není, zvýšíme výšku vrcholu o jedna.

Algoritmus funguje následovně. Na začátku nastavíme výšku všech vrcholů nulovou, zdroj umístíme do výšky N (kde N je počet vrcholů grafu) a po všech hranách ze zdroje pošleme, co se dá (čímž v sousedních vrcholech vytvoříme přebytky). V každém kroku vezmeme vrchol s přebytkem v, jehož výška je nejvyšší (to je heuristika, algoritmus funguje rychleji), a podíváme se na všechny z něj vedoucí hrany s kladnou rezervou (to jsou ale i na hrany do něj vedoucí, pokud už po nich něco teče). Pokud existuje hrana s kladnou rezervou, která vede z kopce (tedy vrchol v je výše než druhý vrchol této hrany), pak po ní pošleme minimum z její rezervy a přebytku ve vrcholu a příslušně poupravíme přebytky. Pokud taková hrana neexistuje, zvedneme výšku vrcholu v o jedna nad minimum z výšek sousedních vrcholů, do nichž vedou hrany s kladnou rezervou. Mohli bychom sice zvýšit výšku pouze o jedna, ale toto nám ušetří opakované zvedání výšky nejvyššího vrcholu a po každé operaci zvednutí výšky bude následovat operace poslání přebytku.

Následuje několik zajímavých, ne úplně triviálních, pozorování:

- **Kdykoliv posíláme přebytek po hraně, spád hrany je roven 1.** To nám umožnuje naprogramovat převedení přebytku v konstantním čase. Také nám to umožnuje dokázat následující pozorování.
- **Maximální výška vrcholu je rovna** $2N$**.** Pokud by tomu tak totiž nebylo, posílali bychom někdy přebytek po spádu větším jak 1.
- **Goldbergův algoritmus má nejlepší časovou složitost.** Pokud použijeme výše zmiňovanou heuristiku, kdy se zbavujeme přebytků vždy v nejvýše položeném vrcholu, potom časová složitost je $\mathcal{O}(N^2\sqrt{M})$, což pro

husté grafy nám dává stejně jako Algoritmus tří Indů $\mathcal{O}(N^3)$, ale pro řidké lepší časovou složitost $\mathcal{O}(N^{\frac{5}{2}})$.

- **Jeho implementace je velice jednoduchá** ve srovnání s ostatními algoritmy.

2 Vnější rozhraní knihovny

Knihovna slouží k hledání maximálního toku v síti. Všechny algoritmy mají to omezení, že kapacity hran musí být celočíselné. Knihovna předpokládá, že vstupní data budou v korektním formátu, nejsou dodatečně verifikována. Knihovna obsahuje vlastní datové typy a funkce. Všechny mají v názvu prefix `flows_`. Datová struktura `flows_graph` obsahuje všechny informace o grafu (počet vrcholů, hran, stupně jednotlivých vrcholů, popis jednotlivých hran, i informace o nalezeném toku). Funkce se dělí na dva typy: funkce pro práci s daty a vlastní funkce algoritmů pro hledání maximálního toku.

2.1 Napojení knihovny na program

Knihovna je psána v jazyku C a její integrování do programu je velice jednoduché. Všechny soubory knihovny nakopírujte do adresáře a do programu vložte následující řádek:

```
#include "flows.h"
```

To přilinkuje hlavičky funkcí a datové struktury knihovny. Dále je nutné při kompilaci zkompilovat soubory knihovny a připojit je ke kompilovánému programu. Ukázky využití knihovny jsou v kapitole 4.

2.2 Funkce pro práci s daty

Funkce pro práci s daty umožnují načítání informací o grafu do paměti, jejich uvolňování, vyprázdnění toku a získání velikostí toku přes jednotlivé hrany.

2.2.1 Funkce flows_load_graph

Na začátku programu je potřeba zavolat funkci `flows_load_graph`, která má následující syntaxi:

```
flows_graph* flows_load_graph(int N, int M, int* edges,
                                            int source, int target);
```

Funkce má následující parametry. Parametr `int N` je počet vrcholů grafu, parametr `int M` je počet hran, `int* edges` je ukazatel na první prvek pole délky $3M$ integerů, každá trojice (u, w, c) odpovídá jedné hraně v grafu, která vede z vrcholu u do vrcholu w a má kapacitu c. Poslední dva parametry jsou

číslo vrcholu, který je v síti zdrojem (`int source`), a číslo vrcholu, který je spotřebičem (`int target`). Vrcholy jsou číslovány od nuly do $N - 1$.

Funkce vrací ukazatel na datovou strukturu `flows_graph`, kterou v paměti alokovala. Při načítání nejprve z grafu odstraní multihrany pomocí matice sousednosti. Při tom se ztratí veškeré informace o nich, multihrany splynou do jedné hrany, jejíž kapacita bude rovna součtu kapacit všech hran. Poté projde graf do šířky od zdroje ke spotřebiči, zahodí všechny neperspektivní vrcholy a hrany. Pro takto začištěný graf spočítá stupně vrcholů, alokuje paměť a informace o grafu zapíše do struktury `flows_graph`, jejíž ukazatel vrátí. Paměťová i časová náročnost funkce je rovna $\mathcal{O}(N^2 + M)$.

2.2.2 Funkce flows_free_graph

Při ukončení práce s knihovnou je potřeba zavolat funkci `flows_free_graph`, která se stará o uvolnění datové struktury `flows_graph` z paměti. Tato funkce má následující syntaxi.

```
void flows_free_graph(flows_graph* graph);
```

Jejím jediným parametrem `flows_graph* graph` je ukazatel na strukturu `flows_graph`, kterou chceme uvolnit z paměti.

2.2.3 Funkce flows_get_flow_edges

Tato funkce se používá pro získání nalezeného toku, protože funkce na načtení grafu `flows_load_graph` ořezala vrcholy a hrany a hrany přečíslovala. Funkce má následující syntaxi.

```
int flows_get_flow_edges(flows_graph* graph, int* edges);
```

První parametr funkce `flows_graph* graph` je ukazatel na datovou strukturu `flows_graph`, kde jsou uložené informace o síti a toku. Druhý parametr `int* edges` je ukazatel na první prvek alokovaného pole velikosti M (kde M je původní velikost grafu, který byl předán na vstupu funkci `flows_load_graph`). Do tohoto pole se uloží velikostu toku přes jednotlivé hrany.

2.2.4 Funkce flows_clean_flow

Tato funkce se používá uvnitř algoritmů na hledání maximálního toku, pokud jejich parametr `int clean` je roven 1. Vynuluje všechny nalezené hodnoty toku. Někdy však může být užitečné nastavit nějaký počáteční tok, tehdy ale algoritmy nezaručí, že naleznou tok maximální. Syntaxe funkce je následující:

```
void flows_clean_flow(flows_graph* flow);
```

Její jediný parametr `flows_graph* flow` je ukazatel na datovou strukturu `flows_graph*`, jejíž hodnoty toku chceme vynulovat.

2.2.5 Funkce flows_verify_flow

Funkce ověří, zda nalezený tok splňuje podmínky toku. Každý tok musí splňovat: $0 \leq f \leq c$ pro každou hranu, součet toků na vstupujících hranách roven součtu toků na vystupujících hranách pro každý vrchol výjma zdroje a spotřebiče a velikost toku je správně určena, tedy tolik odchází ze zdroje a přichází do spotřebiče. Syntaxe je následující:

```
int flows_verify_flow(flows_graph* graph);
```

Jediný parametr `flows_graph* graph` je ukazatel na datovou strukturu se sítí a nalezeným tokem. Návratová hodnota je `1`, pokud tok splňuje všechny podmínky, jinak se na `stdout` vypíše chybová zpráva a návratová hodnota je rovna `0`.

2.2.6 Funkce flows_minimal_cut

Tato funkce nalezne v síti s nalezeným maximálním tokem příslušný minimální řez. Tato funkce je verifikační, kontroluje správnost algoritmů na hledání maximálního toku. V současné verzi předpokládá, že nalezený tok je korektní. Nalezne nám velikost řezu tvořeného vrcholy, do nichž existuje zlepšující cesta ze zdroje. Má tuto syntaxi:

```
int flows_minimal_cut(flows_graph* graph);
```

Jediný parametr `flows_graph* graph` je ukazatel na datovou strukturu se sítí a nalezeným tokem. Návratová hodnota je velikost minimálního řezu.

2.3 Funkce algoritmů

Všechny funkce algoritmů mají stejnou syntaxi, liší se pouze jménem. Obecná syntaxe algoritmu je:

```
void jmeno_algoritmu(flows_graph* graph, int clean);
```

První parametr funkce `flows_graph* graph` je ukazatel na datovou strukturu `flows_graph`, ve které jsou uloženy informace o síti. Druhý parametr `int clean` je většinou nastaven na hodnotu 1 a říká, zda se mají vynulovat hodnoty toků uvnitř struktury `flows_graph` před vlastním výpočtem maximálního toku.

V knihovně jsou implementovány tyto funkce algoritmů:

```
void flows_ford_fulkerson(flows_graph* graph, int clean);
void flows_dinic(flows_graph* graph, int clean);
void flows_three_indians(flows_graph* graph, int clean);
void flows_goldberg(flows_graph* graph, int clean);
```

Implementují Ford-Fulkersonův algoritmus, Dinicův algoritmus, Algoritmus tří Indů a Goldbergův algoritmus.

3 Vnitřní fungování knihovny

3.1 Datová struktura flows_graph a její podstruktury

Datová struktura `flows_graph` obsahuje veškeré informace o síti a toku v ní nalezeném. Její prvky jsou následující:

`int N`	počet vrcholů grafu
`int M`	počet hran osekaného grafu
`int sourceM`	počet hran grafu ze vstupu knihovny
`int s,t`	čísla vrcholu, který je zdroj a který je spotřebič
`int flowsize`	velikost nalezeného maximálního toku
`flows_edge* e`	ukazatel na první prvek pole hran
`flows_vertex* v`	ukazatel na první prvek pole vrcholů

Datová struktura `flows_edge` obsahuje informace o jedné hraně. Její prvky jsou následující:

`int u,w`	hrana vede z vrcholu `u` do vrcholu `w`
`int c`	kapacita hrany
`int f`	velikost toku přes hranu
`int snum`	číslo, pod kterým byla předána `flows_load_graph`

Datová struktura `flows_vertex` obsahuje všechny informace o vrcholu. Její prvky jsou následující:

`int nin`	počet vstupních hran
`int nout`	počet výstupních hran
`int* in`	pole s indexy vstupních hran v poli `flows_edge`
`int* out`	pole s indexy výstupních hran v `flows_edge`

Bližší informace naleznete v souboru `flows_data.h`.

3.2 Ford-Fulkersonův algoritmus

Implementace Ford-Fulkersonova algoritmu je poměrně přímočará a je umístěna v souboru `flows_ford_fulkerson.c`. Graf procházíme do šířky, pokud nalezneme zlepšující cestu, pošleme po ní, co se dá. Pokud ne, prohlásíme nalezený tok za maximální. Na závěr ještě spočteme jeho velikost. Procházení do šířky provádíme pomocí fronty uvnitř cyklu.

3.3 Dinicův algoritmus a Algoritmus tří Indů

Protože jsou oba algoritmy velice podobné (oba potřebují začištěnou rozvrstvenou síť), jejich implementace je prováděna stejnou funkcí a je umístěna v souboru `flows_dinic.c`. Tyto algoritmy jsou asi nejsložitější z celé knihovny a

jejich zdrojové kódy by bylo potřeba značně zjednodušit. Nejprve vyrobíme rozvrstvenou a začištěnou síť (pomocí funkce `net_layers`). Pro Dinicův algoritmus nám stačí jednosměrná na procházení od zdroje ke spotřebiči, pro tři Indy potřebujeme obousměrnou. Dále výpočet probíhá již odděleně.

V Dinicově algoritmu se o hledání zlepšujících cest stará rekurzivní funkce `find_path`. Ta nalezne zlepšující cestu, pošle po ní, co se dá a zároveň se stará o začišťování sítě. Pokud už není zlepšující cesta, máme zlepšující tok. Poté nalezneme novou rozvrstvenou síť.

Procházení rozvrstvené sítě je v Algoritmu tří Indů o trochu složitější a stará se o něj několik funkcí. Nejprve vždy vypočteme rezervy jednotlivých vrcholů. Poté vždy najdeme vrchol s nejmenší rezervou (to děláme tím nejhloupějším algoritmem, prostě projdeme všechny). Použijeme funkci `send_flow` a přeneseme rozvrstvenou sítí daný počet jednotek. Zároveň si průběžně dokládáme do fronty vrcholy k začištění, které provedeme na závěr ve funkci `clean_queue`. Implementace tohoto algoritmu je však příliš složitá, bylo by potřeba ji zjednodušit.

3.4 Goldbergův algoritmus

Tento algoritmus je ze všech nejjednodušší a jeho implementace je uložena v souboru `flows_goldberg.c`. Pro každý vrchol si zde pamatujeme navíc, v jaké je výšce a jaký je v něm přebytek (pole typu `overflow`) a seznam z něj vycházejících hran s přebytkem (pole typu `edge`). Dále si pro každou výškovou hladinu pamatujeme jednosměrný spojový seznam všech vrcholu s přebytkem, které v ní leží. Neprázdné výškové hladiny máme uspořádány opět ve spojovém seznamu. Díky tomu nalezneme vrchol v nejvyšší výšce v konstantním čase. Pak projdeme všechny jeho sousedy. Buď nalezneme takového, do kterého můžeme poslat část přebytku. Potom pošleme a případně vrchol umístíme do o jedna nižší výškové hladiny (pokud předtím neměl přebytek, tak v ní nebyl). To můžeme udělat, protože takový vrchol musí ležet o jedna níže, jinak by byl moc velký spád. Pokud takový nenalezneme, tak jsme si průběžně počítali nejmenší výšku jeho souseda s kladnou rezervou, tak vrchol pozvedneme ještě o jedna výše nad něj (abychom operaci zvedání výšky nemuseli opakovat vícekrát).

3.5 Co by chtělo předělat

Hlavní problém zdrojových kódu jednotlivých algoritmů je ten, že jsou zbytečně dlouhé. Správný krok by bylo nepochybně abstrahovat od detailů datové struktury `flows_graph`, tedy napsat sadu funkcí, jejichž používáním by se kód zkrátil a tím stal přehlednějším. Například velkým nedostatkem je dvojí implementace některých částí – jednou pro hrany výstupní a podruhé s drobnými změnami pro hrany vstupní. Ty se týkají hlavně rozdílného počítání rezerv hrany a že jsou uloženy cílové vrcholy v různých proměnných (jednou `u` a jednou `w`). Dalo by se napsat poměrně jednoduchou sadu funkcí, která by tyto věci dělala automaticky a v návrhu algoritmu bychom se o ně už nemuseli starat. Druhým řešením by bylo přepracovat datový návrh, nenapadlo mě však rozumější řešení.

Zdrojový kód Goldbergova algoritmu by bylo potřeba celý přepsat. Na některých grafech funguje velice pomalu, na jiných je srovnatelně rychlý s ostatními algoritmy. Nepodařilo se mi však objevit, v čem vězí problém.

Další část, na které by určitě chtělo zapracovat, jsou vstupy a výstupy knihovny. Nejsou úplně nejflexibilnější a nevýhodou také je, že není možné síť za chodu měnit, tedy přidávat či odebírat vrcholy a hrany. Odebírání by šlo implementovat, přidávání nikoliv. Zároveň načítání grafu funguje kvůli matici sousednosti ve špatném čase $\mathcal{O}(N^2)$, což pro řidké grafy může velice zhoršit časovou složitost. Řešením by bylo hrany nejprve utřídit přihrádkovým tříděním, poté začistit a nakonec vytvořit graf. Toto řešení by fungovalo v čase $\mathcal{O}(N+M)$.

Užitečné by bylo také přidat verifikace vstupních dat, případně oznámit uživateli nalezenou chybu. Zatím se knihovna chová podle způsobu `Garbage In Garbage Out` a přenechává veškerou verifikační práci na programátorovi.

4 Demonstrace použití knihovny

4.1 Hledání maximálního toku v uživatelem zadané síti

První z demonstrací knihovny hledá maximální tok v uživatelem zadané síti. Na vstupu načte počet vrcholů N, počet hran M, číslo zdroje s, číslo spotřebiče t a informace o jednotlivých hranách (odkud, kam, jaká je její kapacita). Vrcholy jsou číslovány od 0 do $N-1$. Výstupem jsou najité toky jednotlivými algoritmy. Zdrojový kód programu se nachází v souboru `maxflow.c`.

Nalezený tok je pokaždé zkontrolován a je k němu nalezena velikost příslušného minimální řez. Program lze výhodně kombinovat s programem `valid`, protože jeho výstup lze použít jako vstup.

4.2 Autoverifikace knihovny a porovnání rychlostí

Zdrojový kód programu se nachází v souboru `valid.c`. Tento program nejprve načte tři čísla ze `stdin`: počet vrcholů sítě, počet hran sítě a počet testů t. Následně provede t testů, v každém sestrojí náhodnou síť dané velikosti, poté zavolá jednotlivé algoritmy v pořadí Ford-Fulkersonův algoritmus, Dinicův algoritmus, Algoritmus tří Indů a Goldbergův algoritmus, u každého změří čas jeho běhu v sekundách. Také testuje výstupy, to jak do správnosti podmínek toku, tak je porovnává i navzájem a s nalezeným minimálním řezem k příslušnému toku. Pokud objeví chybu, oznámí to a je předčasně ukončen. Výstup, ve kterém došlo k chybě, je uložen do souboru `valid.out`. Pokud proběhne v pořádku všech t testů, program vypíše nalezené statistické výsledky. Tento program byl velice efektivní při odstraňování chyb ve zdrojových kódech algoritmů.

Následuje ukázka statistických dat z běhu algoritmu:

```
$ ./valid (N=1000, M=75000, počet=1000)
...
Stats:
Average N: 1000
```

```
Average M: 72037.7
Average flowsize: 35254.1
Loading of graph: total 51.94s, 1.07076% of total time
Ford-Fulkerson: total 162.46s, 3.34916% of total time
Dinic: total 13.13s, 0.270679% of total time
Three Inds: total 19.97s, 0.411687% of total time
Goldberg: total 4603.27s, 94.8977% of total time
```

4.3 Maximální párování v bipartitním grafu

Převedení problému maximálního párování na maximální toky v síti si předvedeme na jednoduchém příkladu.

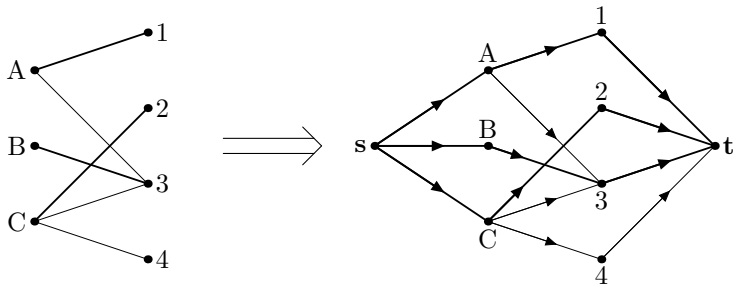
Úložka. V pohádkovém království měli šachovnici o stranách N a M a ne ledajakou, byla celá ze sýra. Jednoho dne se stala veliká pohroma, zlá myš v ní přes noc vykousala hromadu děr. Když to král viděl, napadla ho tato otázka: jak rozmístit na tuto šachovnici co nejvíc věží tak, aby se navzájem neohrožovaly. Věž ohrožuje všechna políčka ve stejném řádku a sloupci, pokud mezi věží a tímto políčkem není díra. A protože si král s tímto problémem nevěděl rady, přenechal jeho řešení královským informatikům. Dokážete jim pomoci?

Řešení. Úlohu vyřešíme pomocí maximálního párování v bipartitním grafu, které poté převedeme na hledání maximálního celočíselného toku v síti. Nejprve si popišme, co je to maximální párování v bipartitním grafu a jak ho lze převést na hledání maximálního celočíselného toku v síti.

Bipartitní graf je neorientovaný graf, jehož vrcholy jsou rozděleny do dvou skupin (podobně jako řez) a hrany vedou vždy z jedné skupiny do druhé. V dalším textu značme vrcholy jedné partity velkými písmeny a vrcholy druhé čísly. Dále *párování* je nějaká podmnožina hran grafu taková, že žádné dvě hrany nemají společný vrchol. *Maximální párování* je párování takové, jehož velikost je maximální. Na obrázku 4 vlevo je ukázka bipartitního grafu, tučně vyznačené hrany jsou párování, a to dokonce maximální.

A jak převedeme hledání maximálního párování na hledání celočíselného maximálního toku? Vytvoříme následující síť. Hrany zorientujeme z levé partity do pravé. Přidáme zdroj, který spojíme hranami se všemi vrcholy levé partity. Přidáme také spotřebič, do kterého vedou hrany ze všech vrcholů pravé partity. Kapacity všech hran budou jedničkové a hledáme celočíselný maximální tok. Hrany s kladným tokem pak budou tvořit párování. Vytvořená síť spolu se svým maximálním tokem je na obrázku 4.

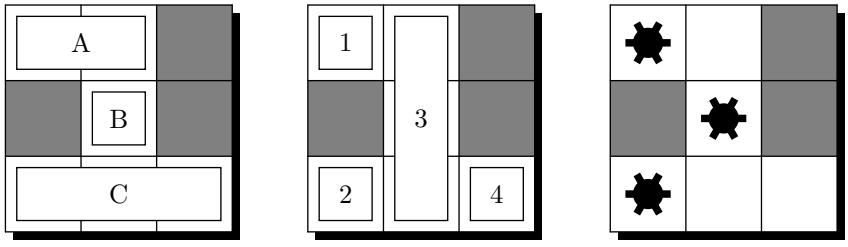
Na hledání maximálních toků máme efektivní polynomiální algoritmy, které se navíc pro podobné sítě chovají ještě lépe. Ještě je ale potřeba převést původní úložku na hledání maximálního párování v bipartitním grafu. Nejprve si vytvořme bipartitní graf. Vezmeme šachovnici, každý její sloupec je rozdělen na několik, alespoň jednu, částí. Tyto části budou tvořit vrcholy levé partity. Všimněme si, že když věž umístíme do libovolné z těchto částí, vůbec neohrožuje (vertikálně) části ostatní. Podobně vrcholy pravé partity budou tvořit části řádků. Pro lepší představu se podívejte na obrázek 5.



Obrázek 4: Převod problému maximálního párování na maximální tok.

Podívejme se nyní na libovolné políčko šachovnice. To bude příslušet k právě jednomu vrcholu levé a pravé partity, tedy každé políčko bude hrana v bipartitním grafu. Jestliže se na nějaké políčko rozhodneme umístit věž, potom už do dané části řádku a části sloupce nesmí přijít jiná věž. Tedy libovolné neohrožující postavení věží odpovídá párování v bipartitním grafu. A protože hledáme rozmístění co největší, potřebujeme nalézt maximální párování. K šachovnicím na obrázku 5 odpovídá bipartitní graf a z něj vytvořená síť na obrázku 4. A rozmístění věží na obrázku 5 odpovídá nalezenému maximálnímu párování na obrázku 4.

Tím jsme s teoretickým rozborem úlohy hotovy. Zdrojový kód demonstrujícího programu je uložen v souboru `chess.c`. Program využívá Dinicův algoritmus, který pro takovouto speciální síť funguje v čase $\mathcal{O}((NM-K)^{\frac{3}{2}})$, kde N a M je rozměr šachovnice a K je počet děr. Při dané implementaci nám nalezne celočíselný tok. Vstup programu tvoří tři celá čísla N (šířka), M (výška) a K a poté K dvojic souřadnic děr (číslováno od 1 do N, resp. M). Výstupem je počet věží a jejich rozmístění, jak pomocí souřadnic, tak i obrázek.



Obrázek 5: Rozdělení šachovnice na levou a pravou partitu a rozmístění věží.