



Semestrální práce z KIV/PC

Hledání maximálního toku v grafu silniční sítě

Daniel Cifka
A19B0021P
dcifka20@students.zcu.cz

5. 1. 2022

Obsah

1	Zadání	2
2	Analýza úlohy	4
2.1	Analýza reprezentace grafu	4
2.1.1	Seznam vrcholů a hran	4
2.1.2	Matice sousednosti	5
2.1.3	Spojový seznam	7
2.2	Maximální tok	8
2.2.1	Motivace	8
2.2.2	Úloha a její řešení	8
2.2.3	Edmondsův-Karpův algoritmus	9
3	Implementace	11
3.1	Implementace načtení dat ze souborů	11
3.1.1	Popis struktur pro uložení dat	12
3.1.2	Popis důležitých funkcí pro uložení dat	13
3.2	Implementace tvorby grafu	15
3.2.1	Popis struktur pro tvorbu matice kapacit	15
3.2.2	Popis důležitých funkcí tvorbu matice kapacit	16
3.3	Implementace nalezení maximálního toku	17
3.3.1	Popis struktur pro výpočet maximálního toku	22
3.3.2	Popis důležitých funkcí pro výpočet maximálního toku	23
3.4	Implementace zápisu do souboru	24
3.4.1	Popis důležitých funkcí pro zápis do souboru	24
4	Uživatelský manuál	25
4.1	Spuštění programu	25
4.2	Průběh programu	25
4.3	Ukázka běhu programu	26
4.3.1	Překlad programu	26
4.3.2	Spuštění programu a jeho výstup	26
5	Závěr	27
6	Zdroje	28

1 Zadání

Naprogramujte v jazyce ANSI C přenositelnou¹ konzolovou aplikaci, jejímž účelem bude nalezení maximálního toku v síti G s jedním zdrojem z a jedním stokem s .

Vaše vstupní i výstupní soubory typu csv je možné vizualizovat pomocí aplikace QGIS.

Program se bude spouštět příkazem `flow.exe` s následujícími přepínači:

- `-v <uzly.csv>`: Parametr specifikuje vstupní soubor typu csv, který obsahuje informace o uzlech sítě. Při zadání nevalidní tabulky program vypíše chybovou hlášku „Invalid vertex file.“ a skončí s návratovou hodnotou 1. Zaručte, aby načtení uzlů grafu bylo vždy první operací.
- `-e <hrany.csv>`: Parametr určuje vstupní soubor typu csv, jehož řádky popisují hrany sítě. V případě zadání nevalidní tabulky program vypíše chybovou hlášku „Invalid edge file.“ a skončí s návratovou hodnotou 2.
- `-s <source id>` Parametr jednoznačně určuje zdroj v síti díky primárnímu klíči `id` tabulky `<uzly.csv>`. Pokud zadaný uzel v tabulce `<uzly.csv>` neexistuje, program vypíše chybovou hlášku „Invalid source vertex.“ a skončí s návratovou hodnotou 3.
- `-t <target id>` Parametr na základě primárního klíče `id` v tabulce `<uzly.csv>` jednoznačně určuje stok v síti. Pokud uvedený uzel v tabulce `<uzly.csv>` neexistuje, nebo je shodný se zdrojem `<source id>`, program vypíše chybovou hlášku „Invalid sink vertex.“ a skončí s návratovou hodnotou 4.
- `-out <out.csv>` V případě úspěšného nalezení toku nenulové velikosti jsou do csv souboru daného tímto nepovinným parametrem uloženy hrany minimálního řezu, jež odděluje zadaný zdroj a stok. Výstupní tabulka má stejnou strukturu jako vstupní (`<hrany.csv>`). Hrany jsou v této tabulce seřazeny dle jejich identifikátoru `id` vzestupně. Pokud je zadané umístění neplatné, program vypíše chybovou hlášku „Invalid output file.“ a skončí s návratovou hodnotou 5. Pokud soubor již existuje, program jej přepíše.

- -a Jedná se o nepovinný přepínač, při jehož uvedení bude program pracovat i s hranami, které mají příznak isvalid nastavený na hodnotu False. Při spuštění bez parametru -a tedy program uvažuje pouze hrany s příznakem isvalid s hodnotou True.

Není-li některý z povinných parametrů uveden, aplikace končí příslušnou chybovou hláškou a návratovou hodnotou. Po úspěšném dokončení algoritmu hledání maximálního toku program vypíše hlášku

„Max network flow is $|x| = \langle s \rangle$.“, kde $\langle s \rangle$ je velikost nalezeného toku, tj. propustnost minimálního řezu.

V případě, že v síti neexistuje tok nenulové velikosti, program skončí s návratovou hodnotou 6. Platí-li opačné tvrzení a zároveň uživatel použil parametr -out, pak je vytvořen výstupní soubor `<out.csv>`, do kterého jsou zapsány hrany odpovídajícího minimálního řezu. Program následně skončí s návratovou hodnotou EXIT SUCCESS.

Váš program může být během testování spuštěn například takto:

„flow.exe -e edgs.csv -v vertcs file.csv -a -s 43 -t 81“

Hotovou práci odevzdejte v jediném archivu typu ZIP prostřednictvím automatického odevzdávacího a validačního systému. Postupujte podle instrukcí uvedených na webu předmětu. Archiv nechť obsahuje všechny zdrojové soubory potřebné k přeložení programu, makefile pro Windows i Linux (pro překlad v Linuxu připravte soubor pojmenovaný makefile a pro Windows makefile.win) a dokumentaci ve formátu PDF vytvořenou v typografickém systému TEX (LATEX). Bude-li některá z částí chybět, kontrolní skript Vaši práci odmítne.

2 Analýza úlohy

2.1 Analýza reprezentace grafu

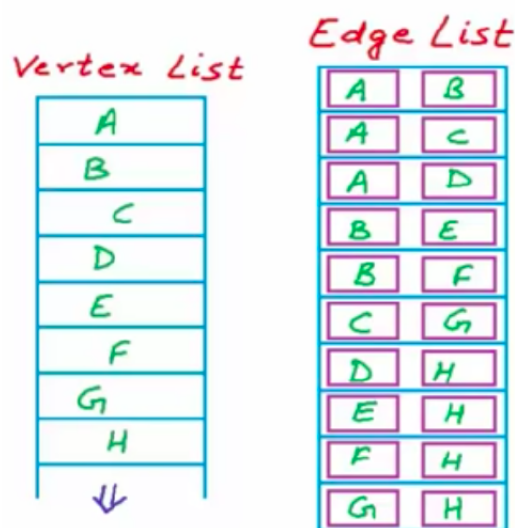
V zadání semestrální práce řešíme hledání maximálního toku v grafu. Graf je v matematice definován jako dvojice vrcholů a hran. Grafy mohou být dvojího typu - orientované a neorientované. V našem případě je graf orientovaný a tedy je možné hledání cesty pouze jedním směrem a to z bodu jednoho do bodu druhého.

Důležitou vlastností grafů je, zdali je graf ohodnocený nebo neohodnocený. Ohodnocení grafu znamená přiřazení váhy (ceny) hranám grafu. V naší semestrální práci budeme hrany ohodnocovat kapacitami získanými z CSV souboru.

Graf můžeme programově reprezentovat třemi možnými způsoby. U každého způsobu budeme pozorovat paměťovou náročnost struktury a časovou náročnost základních operací.

2.1.1 Seznam vrcholů a hran

Nejjednodušší implementací grafu je pomocí dvou seznamů. Jedním ze seznamů je seznam vrcholů a druhým je seznam hran.



Obrázek 2.1: Seznam vrcholů a hran

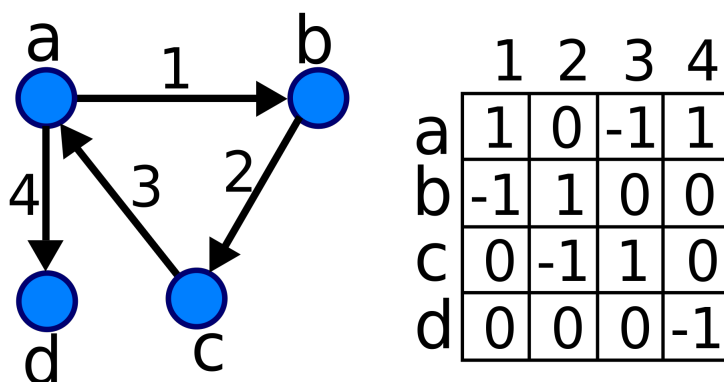
Na obrázku 2.1 můžeme vidět grafické znázornění této reprezentace. Vlevo vidíme seznam vrcholů a vpravo seznam hran, kde je ukázáno odkud a kam daná hrana vede.

Velikost paměti, která bude potřebná k tomuto seznamu vrcholů, bude odpovídat počtu vrcholů grafu neboli $O(V)$. Stejná velikost paměti bude také potřeba pro seznam odpovídajících hran grafu, neboli $O(E)$. Paměťová náročnost celého vytvořeného grafu odpovídá paměti $O(V + E)$.

Nyní se podíváme na časovou náročnost základních operací, které bychom nad grafem mohli potřebovat. K vyhledání všech vrcholů připojených k hledanému vrcholu je potřeba lineární prohledání seznamu hran. Počet operací by odpovídal počtu hran a tedy časová náročnost operace by byla $O(E)$. Další potřebnou operací nad grafem může být zjištění, zdali dva vrcholy grafu jsou spolu propojeny hranou, či nejsou propojeny. K takovéto operaci opět budeme potřebovat lineární prohledávání. Časová náročnost této operace bude tedy opět $O(E)$.

2.1.2 Matice sousednosti

Dalším způsobem reprezentace grafu je matice sousednosti.



Obrázek 2.2: Matice sousednosti

Na obrázku 2.2 můžeme vidět grafické znázornění reprezentace grafu pomocí matice sousednosti. Reprezentace je tvořena maticí tak, že v řádcích a sloupcích matice jsou zapsány vrcholy grafu. Jestliže existuje mezi dvěma vrcholy hrana zjistíme tak, že na pozici 'i', 'j' bude hodnota 1. Kde 'i' je první vrchol a 'j' je druhý vrchol. Pokud mezi těmito vrcholy hrana neexistuje je na pozici 'i', 'j' v matici hodnota 0.

Pokud by se jednalo o neorientovaný graf je matice sousednosti symetrická a tedy na pozici 'i', 'j' je stejná hodnota jako na pozici 'j', 'i'.

V mé semestrální práci využívám matici sousednosti pro uložení kapacit hran, které také reprezentují spojení dvou vrcholů mezi sebou. Vrcholy, které spolu nejsou spojeny mají v matici kapacit hodnotu 0.

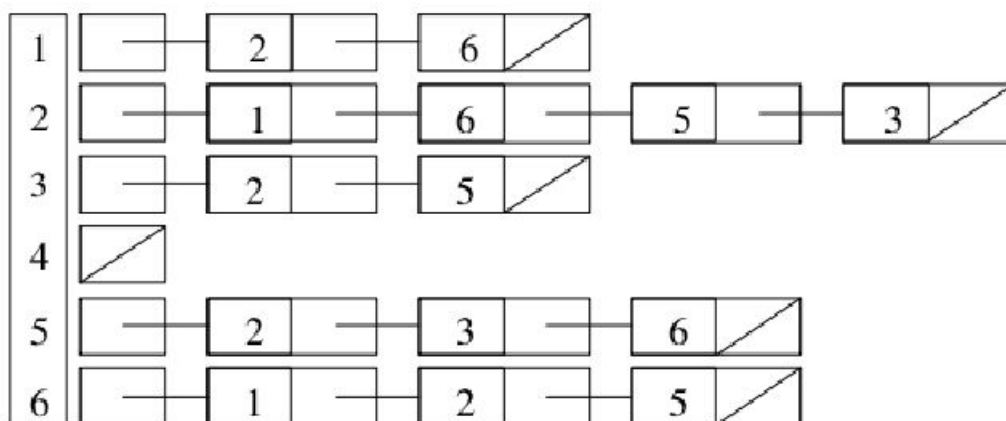
Velikost paměti potřebné k této reprezentaci bude o poznání horší. Narozdíl od seznamu hran. Bude potřeba k uložení matice sousednosti potřeba přesně V^2 paměti - $O(V^2)$.

Velkou nevýhodou této reprezentace je, že kromě existujících hran ukládáme i neexistující hrany, což je přebytečná informace. Pokud bude graf obsahovat velké množství hran (počet hran se bude blížit druhé mocnině počtu vrcholů), byla by tato reprezentace velmi nevýhodná. Pokud však graf bude řídký (bude obsahovat malé množství hran). Pokud by náš graf obsahoval 10 000 vrcholů a každé hraně by jsme rezervovali 4 byty paměti. Potřebovali bychom pro matici sousednosti 400 000 000 bytů, neboli 400 MB paměti. Časová složitost základních operací bude naopak mnohem výhodnější než u předchozí reprezentace grafu. Pro nalezení všech připojených vrcholů k jednomu vrcholu museli bychom lineárně prohledat pouze jeden řádek matice. Z tohoto důvodu dostáváme časovou složitost $O(V)$. Pokud bychom chtěli zjistit, zdali jsou dva vrcholy propojeny hranou a znali bychom jejich index v matici sousednosti, dostaneme časovou složitost $O(1)$. Tohoto přístupu bude potřeba v řešení problému jelikož potřebujeme vždy od dvou konkrétních vrcholů zjistit kapacitu hrany, která je propojuje. Časová složitost této operace odpovídá $O(1)$.

Jelikož Edmondův-Karpův algoritmus, který bude použit pro výpočet maximálního toku požaduje jako vstupní argument matici kapacit, která je nejlépe reprezentována maticí sousednosti, tak byl zvolen tento přístup reprezentace grafu.

2.1.3 Spojový seznam

Posledním ze způsobů reprezentace abstraktní datové struktury graf, je reprezentace pomocí spojvého seznamu.



Obrázek 2.3: Spojový seznam

Na obrázku 2.3 můžete vidět grafické znázornění reprezentace grafu pomocí spojvého seznamu. Hlavní myšlenkou spojvého seznamu je, že každý vrchol má odkazy na vrcholy, které jsou k němu připojené. Programově tuto strukturu většinou vytváříme tzv. *next*, kde *next* je odkaz na další připojený vrchol. Tímto způsobem odstraníme přebytečné informace, které byli zanesené do matici sousednoti, kterými byli informace o neexistujících hranách. Velikost potřebné paměti k uložení abstraktní datové struktury graf bude dána počtem hran, tudíž opět $O(E)$. Do struktury nejsou přičemž zanesené žádné přebytečné informace, jakým jsou neexistující hrany grafu.

Časová složitost základních operacích bude také velmi přívětivá. K zjištění, zdali mezi dvěma vrcholy existuje hrana vyžaduje pouze lineární prohledávání jednoho spojvého seznamu. V takovém případě dostaneme časovou složitost $O(V)$. Jelikož v našem grafu jsou vrcholy ohodnoceny unikátním ID, budeme moci spojvý seznam seřadit podle velikosti a provést binární vyhledávání ve spojovém seznamu a dostali bychom časovou složitost $O(\log V)$. K nalezení všech připojených vrcholů nám opět bude stačit lineární prohledávání s časovou složitostí $O(V)$.

2.2 Maximální tok

2.2.1 Motivace

Problémy řešené v rámci zkoumání toků v sítích jsou motivovány praktickými úlohami o propustnosti - typickým příkladem je propustnost železniční, silniční nebo vodovodní sítě.

Definice

Sít definujeme jako ohodnocený graf $G = (V(G), E(G))$ s hodnotící funkcí $c : E(G) \rightarrow R_0^+$, která udává tzv. kapacitu každé hrany.

Na tomto grafu je množina vrcholů rozdělena na tři disjunktí množiny: **zdroje, stoky, vnitřní vrcholy** $V(G) = V^+(G) \cup V^-(G) \cup V^0(G)$.

Na každé hraně grafu můžeme pak definovat tzv. tok, kladnou veličinu určenou funkcí $t : E(G) \rightarrow R_0^+$. Aby mohla být výše uvedená funkce nazdaná tokem, musí splňovat následující podmínky:

- Tok je omezený kapacitou hrany: $\forall e \in E(G) : t(e) \leq c(e)$
- Ve vnitřních vrcholech se musí vstupní a výstupní tok rovnat
 $\forall v \in V^0(G) : \sum_{e \in \rightarrow v} t(e) = \sum_{e \in v \rightarrow} t(e)$
- Ve zdrojích nesmí být přítok větší než odtok
 $\forall v \in V^+(G) : \sum_{e \in \rightarrow v} t(e) \leq \sum_{e \in v \rightarrow} t(e)$
- Ve stocích nesmí být odtok větší než přítok
 $\forall v \in V^-(G) : \sum_{e \in v \rightarrow} t(e) \leq \sum_{e \in \rightarrow v} t(e)$

2.2.2 Úloha a její řešení

Základní úlohou je najít maximální tok v grafu. Maximalitou se v tomto kontextu myslí "co nejvíce využít propustnost", to znamená navrhnout funkci toku t tak, aby odtok ze zdrojů $\sum_{v \in V^+(G)} \sum_{e \in v \rightarrow} t(e) - \sum_{e \in \rightarrow v} t(e)$ byl maximální možný. Tato úloha může být různými způsoby modifikována.

Jednou z modifikací je převedení úlohy s mnoha zdroji a mnoha stoky na úlohu s jedním zdrojem a jedním stokem.

Tuto zajímavou a jednoduchou modifikaci nyní popíši:

- Zavedeme si do grafu dva nové vrcholy 'z' a 's', které označíme za virtuální zdroj a virtuální stok. Vrchol 'z' spojíme hranou se všemi zdroji původního grafu a těmto hranám přiřadíme nekonečně velkou kapacitu, všechny stoky původního grafu spojíme s vrcholem 's' a i těmto hranám přiřadíme nekonečně velkou kapacitu. Na závěr všechny původní zdroje a stoky označíme jako vnitřní vrcholy nově vzniklého grafu, takže tento nově vzniklý graf bude mít $V^+ = \{z\}$ a $V^- = \{s\}$. V nově vytvořeném grafu lze velmi snadno nahlednout, že maximální tok v tomto grafu zůžený na hrany původního grafu je maximální tokem v původním grafu.

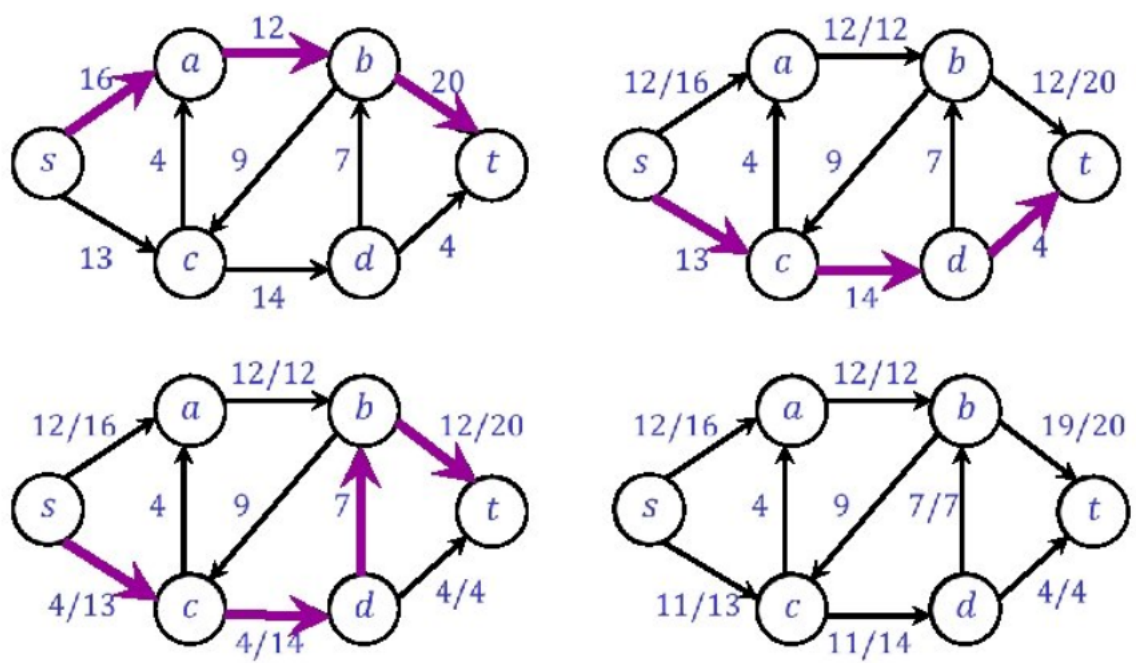
2.2.3 Edmondsův-Karpův algoritmus

Jedná se o algoritmus, který je v informatice a teorii grafů implementací Fordova-Fulkersonova algoritmu pro výpočet maximálního toku v síti. Tento algoritmus dosahuje časové složitosti $O(VE^3)$. Tento algoritmus je asymptoticky pomalejší než Goldbergův algoritmus, který má časovou složitost $O(V^3)$, ale v praxi je rychlejší Edmondsův-Karpův algoritmus pro řídké grafy. Algoritmus je totožný s Fordovým-Fulkersonovým algoritmem¹, s jediným rozdílem a to je definované pořadí výběru zlepšující cesty v případě existence většího počtu zlepšujících cest. Vybraná zlepšující cesta musí být vždy nejkratší možná. Pro důkaz korektnosti a časové složitosti $O(VE^2)$ jsou podstatné následující vlastnosti:

- Délka nalezené zlepšující cesty v průběhu algoritmu nikdy neklesá.
- Je-li v aktuálním kroku algoritmu měněn tok hranou jejíž tok byl změněn v některém z předchozích kroků, pak je délka zlepšující cesty v aktuálním kroku ostře větší než v příslušném kroku předchozím.
- Cesta ze zdrojového vrcholu do koncového vrcholu je nejvýše V dlouhá a lze ji nalézt v čase $O(E)$.

Na obrázku 2.4 můžeme vidět jak Edmondsův-Karpův algoritmus prochází graf a hledá maximální tok.

¹Fordův-Fulkersonův algoritmus



Obrázek 2.4: Edmondsův–Karpův algoritmus

3 Implementace

3.1 Implementace načtení dat ze souborů

Po spuštění program zkontroluje počet vstupních argumentů a zdali byli všechny povinné argumenty zadány. Pokud kontrola proběhla úspěšně voláme funkci *vertex_loader(char*input_vertex_file)*, která přijímá jako vstupní argument cestu k souboru, který má být načtený. Funkce načte soubor, každý řádek rozdělí na příslušné atributy struktury vertex, která reprezentuje vrchol grafu. Pokud je nalezen vrchol který již ve vektoru načtených vrcholů existuje tak se vytvoří instance načteného vrcholu. Rozdělené atributy jsou předány funkci *vertex_create()*, která přijímá jako argumenty načtené rozdělené atributy a funkce vrací ukazatel na dynamicky alokovanou paměť struktury vertex. Po načtení všech vrcholů grafu funkce vrátí ukazatel na dynamicky alokovanou paměť naplněného vektoru, který obsahuje ukazatele na strukturu vertex, prezentující vrchol grafu.

Jako další je potřeba načíst hrany, spojující příslušné vrcholy grafu. Načítání hran zajišťuje funkce *edge_loader(char*input_edge_file)*, která obdobně jako funkce *vertex_loader(char*input_vertex_file)* načítá data, které následně ukládá do dynamicky alokované struktury vector, která uchovává ukazatele na dynamicky alokovanou strukturu edge, reprezentující hranu grafu. Posledním krokem před tvorbou matice kapacit je kontrola zdali zadaný počátek a koncový vrchol se nachází v načteném grafu a obě tyto hodnoty nejsou totožné.

3.1.1 Popis struktur pro uložení dat

Struktury pro uložení získaných dat jsou 3. Struktura vertex, edge, vector
Struktura vertex reprezentuje vrchol grafu.

```
1 typedef struct _vertex
2 {
3     int id;
4     char *WKT;
5 } vertex;
```

Zdrojový kód 3.1: Struktura vertex

Struktura vertex obsahuje tyto atributy:

- id - unikátní identifikátor vrcholu grafu
- WKT - ukazatel na dynamicky alokovanou paměť pro souřadnice vrcholu grafu (Textová reprezentace).

Struktura edge reprezentuje hranu grafu.

```
1 typedef struct _edge
2 {
3     int id;
4     int source;
5     int target;
6     int capacity;
7     char *isvalid;
8     char *WKT;
9 } edge;
```

Zdrojový kód 3.2: Struktura edge

Struktura edge obsahuje tyto atributy:

- id - unikátní identifikátor hrany grafu
- source - počáteční vrchol hrany
- target - koncový vrchol hrany
- capacity - kapacita hrany
- isvalid - Ukazatel na dynamicky alokovanou paměť pro uložení validity hrany (Textová reprezentace)
- WKT - ukazatel na dynamicky alokovanou paměť pro souřadnice hrany grafu (Textová reprezentace).

Struktura `vector` reprezentuje dynamicky realkované pole.

```
1 typedef struct _vector_t {  
2     size_t count;  
3     size_t capacity;  
4     size_t item_size;  
5     void *data;  
6     vec_it_dealloc_t deallocator;  
7 } vector_t;
```

Zdrojový kód 3.3: Struktura `vectort`

Struktura `vectort` obsahuje tyto atributy:

- `count` - aktuální počet prvku v poli
- `capacity` - capacity maximální možná počet prvku v poli
- `item_size` - velikost přidávané struktury do pole
- `data` - ukazatel na dynamicky alokovaný prvek pole
- `deallocator` - Ukazatel na funkci uvolňující přidávané prvky

3.1.2 Popis důležitých funkcí pro uložení dat

`vertex_loader()`

Funkce, která podle vstupního argumentu cesty k souboru, daný soubor otevře. Přečte první řádku souboru csv, která obsahuje hlavičku csv, tu předá funkci `header_check()`, která pouze porovná načtenou hlavičku s nadefinovanou hlavičkou a pokud je shodná vrátí hodnotu 0. Pokud porovnání hlaviček proběhne úspěšně funkce začne číst další řádky, které obsahují příslušná data o vrcholech, které rozdělí na příslušné atributy struktury `vertex`, a zavoláním funkce `vertex_create()`, předá načtené atributy. Funkce vrátí ukazatel na dynamicky alokovanou paměť struktury `vertex`, který uloží do dynamicky alokované struktury `vectort` představující dynamicky realokované pole.

`edge_loader()`

Funkce, která podle vstupního argumentu cesty k souboru, daný soubor otevře. Přečte první řádku souboru csv, která obsahuje hlavičku csv, kterou předá funkci `header_check()`, která pouze porovná načtenou hlavičku s nadefinovanou hlavičkou a pokud je shodná vrátí hodnotu 0. Pokud porovnání hlaviček proběhne úspěšně funkce začne číst další řádky, které obsahují

příslušná data o edge, které rozdělí na příslušné atributy struktury edge, a zavoláním funkce `edge_create()`, předá načtené atributy. Funkce vrátí ukazatel na dynamicky alokovanou paměť struktury edge, který uloží do dynamicky alokované struktury `vector_t` představující dynamicky realokované pole.

`duplicity_check_edge() \ vertex()`

Funkce požaduje jako vstupní argumenty pole vrcholů \ hran a id právě načítaného prvku. Funkce prochází celé pole a pokud najde shodu zadaného id s id v poli uloženém je vrácena hodnota 0 a příslušná načítací funkce dat tento načtený prvek přeskočí.

`header_check()`

Funkce požaduje jako vstupní argument načtenou hlavičku csv souboru a id podle kterého funkce vybere se kterou hlavičkou má načtenou hlavičku kontrolovat. Funkce vezme načtený řetězec, který vloží do knihovni funkce `strcmp`, kde porovná tento řetězec s definovaným řetězcem hlavičky a pokud jsou shodné vrátí hodnotu 0, pokud jsou odlišné vrátí hodnotu -1, což v načítací funkci identifikuje, že hlavička není správná a soubor nelze načíst.

`check_vertex_in_graph()`

Funkce požaduje jako vstupní argumenty pole načtených vrcholů a id počátečního nebo cílového vrchol, pro který hledáme maximální tok. Funkce prochází celé předané pole a hledá shodu s předaným id. Pokud je shoda nalezena vrátíme 0, pokud se ale vrchol v grafu nenachází vrátíme hodnotu 1, která identifikuje, že vrchol se v grafu nenachází a funkce volající výše uvedenou funkci program ukončí.

3.2 Implementace tvorby grafu

Jako strukturu pro tvorbu grafu jsem zvolil matici sousednosti, která velmi efektivně umožňuje přístup ke kapacitám hran grafu potřebných pro výpočet maximálního toku. Matice sousednosti je implementována pomocí struktury `matrix` implementovanou na cvičeních.

Program po načtení potřebných dat ze souborů zavolá funkce `get_real_vector()` a `get_image_vector()`, které požadují jako vstupní parametr pole načtených vrcholů. Obě tyto funkce vrací ukazatel na dynamicky alokovanou paměť pole, kde ukazatel funkce `get_real_vector()` obsahuje adresu na pole skutečných načtených id hran, naopak ukazatel funkce `get_image_vector()` obsahuje adresu na pole imaginárních id hran. Pole imaginárních id, pouze pomáhá, aby nebylo alokováno mnoho paměti, jelikož některé skutečné hrany mají velmi vysoké id a matice kapacit, by v tomto případě alokovala velké množství paměti, což by bylo zbytečné. Po úspěšné tvorbě pomocných vektorů přecházíme k samotné tvorbě matice kapacit. K vytvoření samotné matice kapacit využíváme funkci `capacity_matrix()`, funkce přijímá jako vstupní argumenty námi vytvořená pomocná pole id a pole načtených hran. Funkce využívá pomocná pole id k určení velikosti matice a z příslušných vrcholů grafu získá z pole kapacit kapacitu hrany, kterou následně nastaví v matici kapacit na příslušné pozici určené imaginárním id, které převedlo skutečný id na nižší hodnotu id. Když nastavíme příslušné kapacity hranám, které jsou uvedeny ve vektoru hran, vrátíme ukazatel na dynamicky alokovanou paměť matice kapacit.

3.2.1 Popis struktur pro tvorbu matice kapacit

Struktura pro uložení matice kapacit: `matrix`.

```
1 typedef struct _matrix {  
2     size_t rows;  
3     size_t cols;  
4     mat_num_t *items;  
5 } matrix;
```

Zdrojový kód 3.4: Struktura `matrix`

Struktura `matrix` obsahuje tyto atributy:

- `row` - počet řádek matice
- `col` - počet sloupců matice
- `*items` - reálná hodnota

3.2.2 Popis důležitých funkcí tvorbu matice kapacit

get_real_vector()

Funkce přijímá jako vstupní argument načtené pole vrcholů. Funkce alokuje paměť pro strukturu `vector_t`, která v sobě bude ukládat číselnou hodnotu id hran, které byli načteny. Po úspěšné alokaci paměti pro `vector_t` uloží do vectoru číselnou načteného id. Po uložení všech id hran vrátíme ukazatel na tuto dynamicky alokovanou paměť struktury `vector_c`, která má v sobě uložené hodnoty skutečných id hran.

get_image_vector()

Funkce přijímá jako vstupní argument načtené pole vrcholů. Funkce alokuje paměť pro strukturu `vector_t`, která v sobě bude ukládat číselnou hodnotu imaginárních id hran, které byly tvořeny cyklem `for`. Po úspěšné alokaci paměti pro `vector_t` uloží do vectoru číselnou získanou z cyklu `for`. Po uložení všech id získaných cyklem `for` vrátíme ukazatel na tuto dynamicky alokovanou paměť struktury `vector_c`, která má v sobě uložené hodnoty imaginárních id hran.

capacity_matrix()

Funkce přijímá jako vstupní argumenty načtené pole hrana, vytvořená pomocná pole id hran. Alokuje pro strukturu `matrix` paměť. Do matice kapacit nastavíme příslušné kapacity hran, které získáme z předaného vectoru hran. Po úspěšném nastavení kapacit v matici kapacit, vrátíme ukazatel na dynamicky alokovanou strukturu `matrix`, která obsahuje nastavené kapacity hran.

get_image_id_from_real()

Funkce přijímá jako vstupní argumenty vytvořená pomocná pole id hran a skutečné id hrany. Funkce projde cyklem `for` pole skutečných id pokud nalezne shodu skutečného id s hledaným id, tak vrátí z pole imaginárních id jeho imaginární id.

get_real_id_from_image()

Funkce přijímá jako vstupní argumenty vytvořená pomocná pole id hran a skutečné id hrany. Funkce projde cyklem `for` pole imaginárních id, pokud nalezne shodu imaginárního id s hledaným id, tak vrátí z pole skutečných id jeho skutečné id.

get__edge()

Funkce přijímá jako vstupní argumenty načtené pole hrana, počáteční vrchol hrany, koncový vrchol hrany a kapacitu hrany. Funkce projde pomocí cyklu for pole načtených hran a pokud nalezne hranu splňující předané argumenty počátečního vrcholu, koncového vrcholu a kapacity, vrátíme ukazatel na nalezenou hranu.

3.3 Implementace nalezení maximálního toku

Před začátkem vyhledávání maximálního toku v grafu, alokujeme paměť pro strukturu matrix, která bude představovat měnící se matici kapacit, dále alokujeme paměť pro dynamické pole celých čísel, které bude uchovávat id vrcholu, který předchází aktuálnímu vrcholu a nakonec alokujeme paměť pro strukturu vector_t, která bude uchovávat hrany minimálního řezu. Jakmile jsou všechny alokace úspěšně přecházíme k vykonání Edmonds-Karpova algoritmu, který ke svému provedení potřebuje algoritmus BFS, který projde celý graf a nalezne nejkratší cestu do koncového vrcholu. Pokud BFS nalezne cestu, nalezneme minimální tok, který přičteme k doposud nalezenému maximálnímu toku a znovu zavoláme BFS, pokud BFS vrátí hodnotu NULL, hledání maximálního toku končí, vypíšeme nalezný maximální tok a podle jeho hodnoty vrátíme hodnotu NULL, když je minimální tok roven nule a nebo ukazatel na dynamicky alokovanou strukturu vector_t, která uchovává hrany zjištěné při vyhledávání maximálního toku.

```
1 vector_t *max_flow(matrix *capacity, const vector_t *
    real_vector, const vector_t *image_vector, const vector_t *
    vector_vertex, const vector_t *vector_edge, const size_t
    source, const size_t target) {
2     matrix *residual = NULL;
3     vector_t *minimal_cut = NULL;
4     edge *minimal_edge = NULL, *help_edge = NULL;
5     size_t i = 0, u = 0, v = 0;
6     int path = 0, maxFlow = 0, residual_capacity = 0, *parents
    = NULL, help_capacity = 0;
7
8
9     if(!capacity || !real_vector || !vector_vertex || !
    vector_edge) {
10        printf("Not all required input arguments are specified.
    ");
```

```

11         return NULL;
12     }
13
14     residual = matrix_create(vector_count(vector_vertex),
15 vector_count(vector_vertex), 0);
16     if(!residual) {
17         printf("Residual_matrix_allocation_memory_was_not_
18 obtained.\n");
19         return NULL;
20     }
21
22     parents = (int *)malloc(vector_count(vector_vertex) *
23 sizeof(int));
24     if(!parents) {
25         printf("Ancestor_array_allocation_memory_not_retrieved
26 .\n");
27         matrix_free(&residual);
28         return NULL;
29     }
30
31     minimal_cut = vector_create(sizeof(edge *), NULL);
32     if(!minimal_cut) {
33         printf("Memory_for_minimum_slice_vector_allocation_was_
34 not_obtained.\n");
35         matrix_free(&residual);
36         free(parents);
37         return NULL;
38     }
39
40     for(i = 0; i < vector_count(vector_vertex); ++i) {
41         for(v = 0; v < vector_count(vector_vertex); ++v) {
42             matrix_set(residual, i, v, matrix_get(capacity, i,
43 v));
44         }
45     }
46
47     maxFlow = 0;
48     parents = BFS(residual, source, target, parents,
49 vector_vertex);
50
51     while (parents != NULL) {

```

```

45     path = __INT_MAX__;
46     help_capacity = __INT_MAX__;
47     for(i = target; i != source; i = parents[i]) {
48         u = parents[i];
49         residual_capacity = matrix_get(residual, u, i);
50         path = min_flow(path, residual_capacity);
51         if(path == -1) {
52             matrix_free(&residual);
53             free(parents);
54             vector_destroy(&minimal_cut);
55             return NULL;
56         }
57
58         if(residual_capacity == path) {
59             help_edge = get_edge(vector_edge,
get_real_id_from_image(real_vector, image_vector, u),
get_real_id_from_image(real_vector, image_vector, i), path)
;
60             if(residual_capacity <= help_capacity &&
help_edge != NULL) {
61                 minimal_edge = help_edge;
62                 help_capacity = minimal_edge->capacity;
63             }
64         }
65     }
66
67     for (i = target; i != source; i = parents[i]) {
68         u = parents[i];
69         matrix_set(residual, u, i, matrix_get(residual, u,
i) - path);
70         matrix_set(residual, i, u, matrix_get(residual, i,
u) + path);
71     }
72     maxFlow += path;
73     vector_push_back(minimal_cut, &minimal_edge);
74     parents = BFS(residual, source, target, parents,
vector_vertex);
75 }
76
77 printf("Max_network_flow_is_|x|=_%d.\n", maxFlow);
78 if(maxFlow == 0) {

```

```

79     matrix_free(&residual);
80     vector_destroy(&minimal_cut);
81     return NULL;
82 }
83 matrix_free(&residual);
84 return minimal_cut;
85 }
86 }

```

Zdrojový kód 3.5: Funkce max_flow

```

1 int *BFS (matrix *residual, const size_t source, const size_t
    target, int *parents, const vector_t *vector_vertex) {
2     size_t i = 0;
3     int u = 0;
4     int *visiter = NULL;
5     queue *q = NULL;
6
7     if(!residual || !parents || !vector_vertex) {
8         printf("Not all required input arguments are specified.
9         ");
10        return NULL;
11    }
12
13    visiter = (int *)malloc(vector_count(vector_vertex) *
14    sizeof(int));
15    if(!visiter) {
16        printf("The visiter array has not been created.\n");
17        return NULL;
18    }
19
20    for(i = 0; i < vector_count(vector_vertex); ++i) {
21        visiter[i] = 0;
22    }
23
24    q = queue_create();
25    if(!q) {
26        printf("The queue data structure allocation memory was
27        not retrieved.\n");
28        free(visiter);
29        visiter = NULL;
30    }
31
32    return q;
33 }

```

```

27     }
28
29     q = enqueue(q, source);
30
31     visiter[source] = 1;
32     parents[source] = -1;
33
34     while (q->front != NULL) {
35         u = get_front_element(q);
36         if(u == -1) {
37             q = clean_queue(q);
38             free(q);
39             q = NULL;
40             free(visiter);
41             visiter = NULL;
42             free(parents);
43             parents = NULL;
44             return NULL;
45         }
46
47         q = dequeue(q);
48
49         for(i = 0; i < vector_count(vector_vertex); ++i) {
50             if(visiter[i] == 0 && 0 < matrix_get(residual, u,
51 i)) {
52                 if(i == target) {
53                     parents[i] = u;
54                     q = clean_queue(q);
55                     free(q);
56                     q = NULL;
57                     free(visiter);
58                     visiter = NULL;
59                     return parents;
60                 }
61
62                 q = enqueue(q, i);
63                 parents[i] = u;
64                 visiter[i] = 1;
65             }
66         }

```

```

67
68     q = clean_queue(q);
69     free(q);
70     q = NULL;
71     free(visiter);
72     visiter = NULL;
73     free(parents);
74     parents = NULL;
75     return NULL;
76 }

```

Zdrojový kód 3.6: Funkce BFS

3.3.1 Popis struktur pro výpočet maximálního toku

Struktury pro výpočet maximálního toku: queue_item, queue Struktura queue_item reprezentuje přidávaný prvek do fronty

```

1 typedef struct _queue_item
2 {
3     int data;
4     struct _queue_item* next;
5 }queue_item;

```

Zdrojový kód 3.7: Struktura queue_item

Struktura queue_item obsahuje tyto atributy:

- data - číselná hodnota přidávaného prvku
- *next - ukazatel na další prvek ve frontě

Struktura queue reprezentuje abstraktní datovou strukturu frontu

```

1 typedef struct _queue
2 {
3     queue_item *front, *rear;
4 } queue;

```

Zdrojový kód 3.8: Struktura queue_item

Struktura queue_item obsahuje tyto atributy:

- *front - ukazatel na prvek čela fronty
- *rear - ukazatel na prvek konce fronty

3.3.2 Popis důležitých funkcí pro výpočet maximálního toku

new_item()

Funkce pro tvorbu prvku přidávaného do fronty. Vstupními argumenty funkce je číselná hodnota, která bude přidána do fronty. Funkce alokuje paměť pro strukturu `queue_item`, kde je uchována přidávaná hodnota. Po úspěšné alokaci paměti a nastavení příslušné hodnoty do alokované struktury `queue_item` vrátíme ukazatel na tuto alokovanou paměť.

queue_create()

Funkce pro tvorbu abstraktní datové struktury fronty. Funkce pouze alokuje paměť pro strukturu `queue`, která představuje abstraktní datovou strukturu fronta. Po úspěšné alokaci paměti vrátím ukazatel na dynamicky alokovanou strukturu `queue`.

enQueue()

Funkce pro přidání prvku do fronty. Vstupními argumenty funkce je číselná hodnota, která bude přidána do fronty a ukazatel na dynamicky alokovanou paměť struktury `queue`. Funkce vezme vstupní číselnou hodnotu, kterou předá funkci `new_item()`, která vrátí ukazatel na nově vytvořený prvek který přidáme předané struktury `queue`, na kterou odkazuje předaný vstupní ukazatel. Po úspěšném přidání ve funkci vrátí ukazatel na strukturu `queue`, s nově přidaným prvkem.

deQueue()

Funkce pro odebrání prvku z fronty. Vstupním argumentem funkce je ukazatel na dynamicky alokovanou paměť struktury `queue`. Funkce získá prvek z čela fronty, uloží ho do pomocné proměnné typu `queue_item` čelní prvek se nastaví na následující prvek ve frontě a uložený prvek v pomocné proměnné je uvolně a funkce vrátí ukazatel na strukturu `queue` s odebraným prvkem.

clean_queue()

Funkce pro odstranění zbylých prvků uložených ve frontě. Funkce přijímá jako vstupní argument ukazatel na dynamicky alokovanou strukturu `queue`. Funkce volá funkci `deQueue()`, která odebere zbývající prvky uložené ve frontě. Jakmile jsou všechny prvky odebrány z fronty, vrátíme ukazatel na dynamicky alokovanou strukturu `queue`, která je nyní prázdná.

BFS()

Funkce která prohledává graf do šířky a hledá možné cesty grafem. Vstupní argumenty funkce jsou měnící se matice kapacit, počáteční vrchol, koncový vrchol, ukazatel na dynamicky alokované pole rodičovských vrcholů a ukazatel na dynamicky alokovanou strukturu `vector_t`. Funkce představuje algoritmus prohledání do šířky, který nalezne cestu grafem. Podrobnější popis algoritmu BFS, lze naléznout zde¹.

3.4 Implementace zápisu do souboru

3.4.1 Popis důležitých funkcí pro zápis do souboru

`write_file()`

Funkce pro zápis hran minimálního řezu do souboru. Funkce požaduje jako vstupní argumenty cestu ke vstupnímu souboru, a ukazatel na dynamicky alokovanou paměť struktury `vector_t`, která obsahuje hrany minimálního řezu. Funkce dynamicky alokuje pole celých čísel, kam se uloží id hran minimálního řezu. Pole id hran se předá knihovní funkci `qsort`, která seřadí pole id vzestupně. Toto pole id je následně použito pro vypis srovnaných hran, pokud je v poli id rovno id vybraného elementu z předaného vektoru tak je daná hrana zapsána do souboru. Po skončení zápisu je souboru uzavřen a vracena hodnota 0, nastane-li nějaký problém v průběhu zápisu, bude vrácena hodnota 1.

¹BFS

4 Uživatelský manuál

4.1 Spuštění programu

Aplikace je pouze konzolová. Aplikaci je nejprve nutné přeložit. O přeložení aplikace se stará soubor makefile. Soubor makefile se spustí příkazem `make` pro linux a `mingw32-make` pro windows v konzoli. Po přeložení aplikace je možné ji spustit. Aplikace se spouští příkazem „`./flow.exe -v <soubor vrcholu> -e <soubor hran> -s <svstupní vrchol> -t <výstupní vrchol>`“, dále zde můžeme zadat ještě argumenty „`-a, -out <výstupní soubo>`“, které jsou nepovinné. Tento zápis je platný pouze pro operační systém linux, pro windows je příkaz tento „`flow.exe -v <soubor vrcholu> -e <soubor hran> -s <svstupní vrchol> -t <výstupní vrchol>`“ dále zde můžeme zadat ještě argumenty „`-a, -out <výstupní soubo>`“, které jsou nepovinné. Zadávané soubory `<soubor vrcholu>`, `<soubor hran>` a `<výstupní soubo>` musí být ve formátu .CSV. Pokud bude aplikace spuštěna s neplatným počtem parametrů, či parametry nebudou pro spuštění aplikace validní, aplikace vypíše příslušnou hlášku a ukončí program.

4.2 Průběh programu

Po zadání vstupních argumentů aplikace budou tyto argumenty zkontrolovány, pokud jsou validní načteme ze souboru vrcholy grafu a hrany grafu. Program provede potřebný výpočet maximálního toku a nakonci vypíše do konzole velikost maximálního toku a pokud je zadán nepovinný přepínač `-out`, tak jsou do souboru vypsány hrany minimálního řezu seřazené vzestupně podle id.

4.3 Ukázka běhu programu

4.3.1 Překlad programu

```
D:\PC\SP_V4>mingw32-make
gcc -c edge.c -Wall -Wextra -pedantic -ansi -g
gcc -c vertex.c -Wall -Wextra -pedantic -ansi -g
gcc -c vector.c -Wall -Wextra -pedantic -ansi -g
gcc -c input.c -Wall -Wextra -pedantic -ansi -g
gcc -c matrix.c -Wall -Wextra -pedantic -ansi -g
gcc -c queue.c -Wall -Wextra -pedantic -ansi -g
gcc -c graph.c -Wall -Wextra -pedantic -ansi -g
gcc -c maxflow.c -Wall -Wextra -pedantic -ansi -g
gcc -c output.c -Wall -Wextra -pedantic -ansi -g
gcc -c main.c -Wall -Wextra -pedantic -ansi -g
gcc edge.o vertex.o vector.o input.o matrix.o queue.o graph.o maxflow.o out-
put.o main.o -o flow.exe -Wall -Wextra -pedantic -ansi -g
```

4.3.2 Spuštění programu a jeho výstup

```
D:\PC\SP_V4>flow.exe -v ./example/example_nodes.csv -e ./example/exam-
ple_edges.csv -s 0 -t 6 -out ahoj2.csv
Max network flow is  $|x| = 4$ .
```

id	source	target	capacity	isvalid	WKT
2	0	3	1	True	"Geographical description of edge 2. A few commas , , , to make it a little more realistic."
3	1	4	1	True	"Geographical description of edge 3. A few commas , , , to make it a little more realistic."
4	2	4	1	True	"Geographical description of edge 4. A few commas , , , to make it a little more realistic."
9	5	6	1	True	"Geographical description of edge 9. A few commas , , , to make it a little more realistic."

Obrázek 4.1: Výstup ve formátu CSV souboru

5 Závěr

I přes všechny potíže se mi podařilo naprogramovat funkční aplikaci, která splňuje požadavky semestrální práce.

Při ladění programu byli nejdříve využity zkušební data, která obsahují pouze pár vstupních dat, ale bylo možné na nich ověřit všechny možné scénáře, které mohou nastat, při reálných datech. Program byl spuštěn na stroji Acer nitro 5 se zkušebními vstupními daty (`./example/example_nodes.csv`, `./example/example_edges.csv`, 0, 6, `myoutput.csv`) a výstupem programu bylo nalezení maximálního toku, který měl hodnotu 4 a ve výstupním souboru byli zaznamenány 4 hrany, které představovali minimální řez grafem. Čas testu nebylo možné zaznamenat, jelikož test proběhl velmi rychle. Dalším testem který byl proveden, byl test na reálných datech, který byl spuštěn na stroji Acer nitro 5 s daty (`./plzen/pilsen_nodes.csv`, `./plzen/pilsen_edges.csv`, 108, 109, `myoutput.csv`) a výstupem programu bylo nalezení maximálního toku, který měl hodnotu 2000 a ve výstupním souboru byl zaznamenán jediný zaznam, které představovali minimální řez grafem. Čas tohoto testu se pohyboval v řádech jednotek sekund.

V programu se jistě nacházejí konstrukce, které by bylo možné zjednodušit, ale tak jak byl program napsán, odpovídá normě ansi C (C89). V programu nejsou použity žádné nebezpečné konstrukce, i když v původním návrhu programu byla použita globální proměnná pro uchování imaginárních id, kterou jsem následně přepracoval do předávající se struktury `vector_t`.

6 Zdroje

Zde jsou odkazy na materiály, které jsem během řešení své semestrální práce využil.

2.1 Obrázek ukazující reprezentaci grafu seznamy vrcholů a hran:

<https://www.youtube.com/watch?v=ZdY1Fp9dKzs>

2.3 Obrázek ukazující reprezentaci grafu pomocí spojového seznamu:

<http://www.512.cz/images/b/bf/Graf-spojovy-seznam.jpg>

Fold-Fulkersonův algoritmus:

<https://www.teiresias.muni.cz/amalg/www/cs/adaptation/fold-fulkerson>

BFS: <https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>

Edmonds-Karpův algoritmus:

<https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/>

Pro různé problémy: <https://stackoverflow.com>