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

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 ta- bulky <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> jednozna- čně určuje stok v síti. Pokud uvedený uzel v tabulkce <uzly.csv> neexis tuje, 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 návrato- vou 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| = <s>.“, kde <s> 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 odevzdáva- cího 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

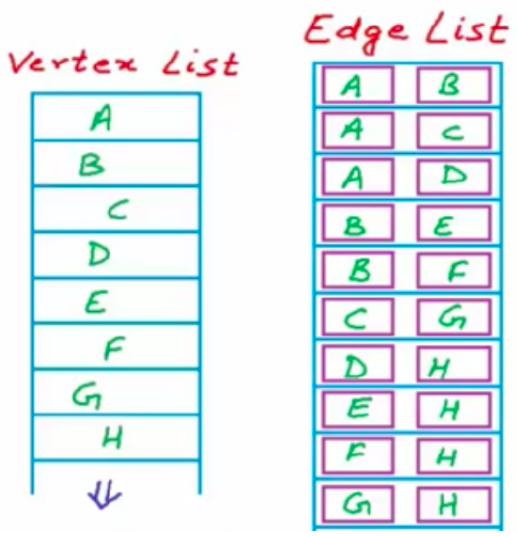
2.1 Analýza reprezentace grafu

V zadaní semestrální práce řešíme hledání maximální toku v grafu. Graf je v matematice definován jako dvojce vrcholů a hran. Grafy mohou být dvojího typu - orienované a neorientované. V naše případě je graf orientovaný a tedy jsou možné hledané cesty pouze jedním směrem a tedy z bodu jednoho bodu do druhého bodu.
Důležitou vlastností grafů je, zdali je graf ohodnocený nebo neohodnocený. Ohodnocení grafu znamená přiřazení vahy (ceny) hranám grafu. V naší semestrální práci budeme hrany ohodnocovat kapacitami získanými z CSV souboru.

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

2.1.1 Seznam vrcholů a hran

Nejjednoduší implementací grafu je pomocí dvouch seznamů. Jedním ze seznamů je seznam vrcholů a druhým je seznam hran.



Obrázek 2.1: Seznam vrcholů a hran

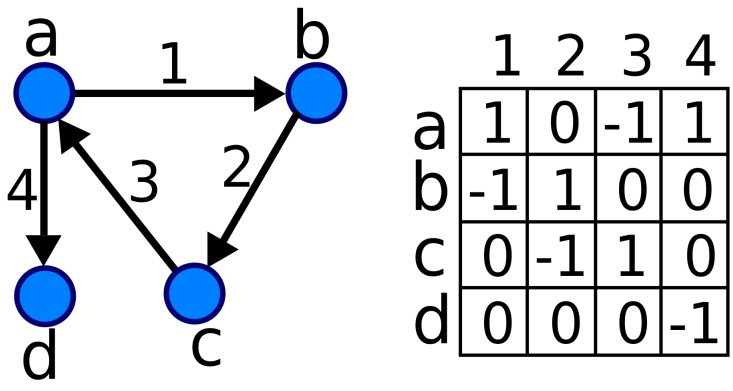
Na obrázku 2.1 můžete vidět grafické znázornění této reprezenatce. Vlevo vidíme seznam vrcholů a v pravo seznam hran, kde je ukázano odkud a kam dáná hrana vede.

Velikost paměti, která bude potřebná k tomuto seznamu vrcholů, bude odpovídat počet vrcholů grafu nebo 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 na 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 můžeme být zjištění, zdali dva vrcholy grafu jsou spolu propojeny hranou, či nejsou propojeny hranou. K takovéto operaci opět budeme potřebovat lineární prohledávání a časová náročnost této operace bude tedy opět O(E).

2.1.2 Matice sousednosti

Další způsobem reprezentace grafu je matice sousednoti.



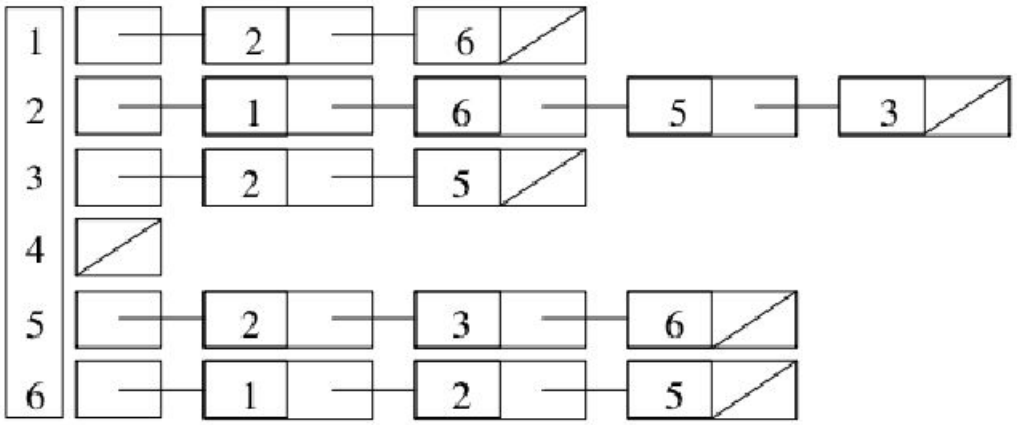
Obrázek 2.2: Matice sousednosti

Na obrázku 2.2 můžete vidět grafické znázornění reprezentace grafu pomocí matice sousednosti. Reprezentace je tvořena maticí tak, že v řádcích a sloupcích matice jsou zapsany vrcholy grafu. Jestli existuje mezi dvěma vrcholy hrana zjistíme na pozici 'i', 'j', kde 'i' je první vrchol a 'j' je druhý vrchol. Pokud mezi těmito vrcholy hrana existuje je na pozici 'i', 'j' v matici hodnota 1. 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 vyžívám matici sousednosti pro uložení kapacit hran, které také reprezentují spojení dvou vrcholu 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 výhodná. Pokud však graf bude řídký (bude obsahovat malé množství hran), budeme zbytečně zabírat velké množství paměti. Pokud by náš graf obsahoval 10 000 vrcholů a každé hraně by jsme rezervovali 4 byty paměti, potřebovali bychom pro matici sousednoti 400 000 000 bytů, neboli 400 MB paměti. Časová složitost základních operací bude naopak mnhohem 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 dvouch 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 gafu.

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

Časová složitost základních operacích bude táké velmi přívětivá. K zjištění, zdali mezi dvěmi vrcholy exostuje hrana vyžaduje pouze lineární prohledávání jednoho spojového seznamu. V takovém případě dostaneme časovou složtost O(V). Jelikož v našem grafu jsou vrcholy ohodnoceny unikátním ID, budeme moci spojový seznam seřadit podle velikosti a provést binární vyhledávání ve spojovém seznamu a dostali bychom časovou složitost O(logV). K nalezení všech připojených vrcholů nám opět bude stačit lieární 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 propostunosti - typckým příkladem je propustnost železniční, silniční nebo vodovoní sítě.

Definice

Síť definujeme jako ohodnocený graf $G = (V(G), E(G))$ s hodnotící funkcí $c : E(G) \to R_0^+$, která udává tzv. kapacitu každé hrany.
Na tomto grafu je množina vrcholů rozdělená na tři disjunktní 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 $t = E(G) \to R_0^+$. Aby mohla být výše uvedená funkce nazdaná tokem, musí splňovat následujíc podmínky:

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

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 fukci toku t tak, aby odtok ze zdrojů $\sum_{v \in V^+(G)} \sum_{e \in v \to} t(e) - \sum_{e \in \to v} t(e))$ byl maximální možný. Tato úloha může být různými způsobi modifikována.
Jednou z modifiakcí je převedení ulohy s mnoha zdroji a mnoha stoky na úlohu s jedním s 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' spojme hranou se všemi zdroji původního grafu a těmto hranám přiřadíme nekonečně velkou kapacitu, všechny stoky původního grafu spojme s vrcholem 's' a i těmto hranám přiřaďme 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 snado nahlednout, že maximální tok v tomto grafu zůžený na hrany původního grafu je maximální tokem v původním grafu.

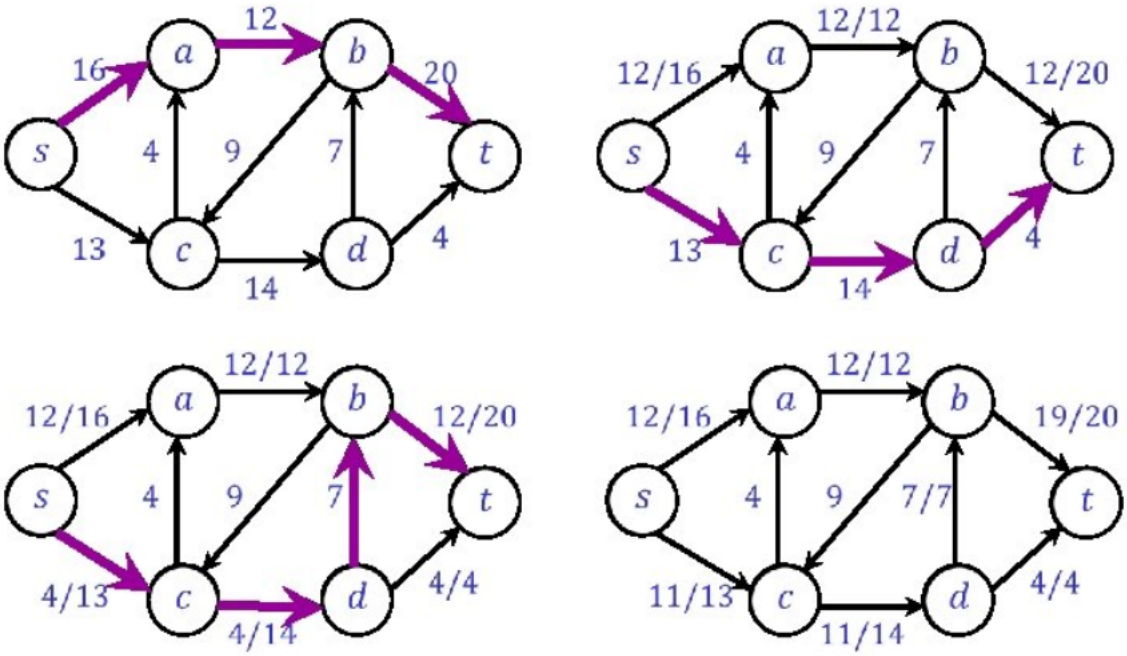
2.2.3 Edmondsův-Karpův algoritmus

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 graf je asyptotickz 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 algoritmem[1], s jediným rozdílem a to je definovaný 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 podstatn0 nasleduj9c 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 jí nalézt v čase $O(E)$.

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

[1] Fordův–Fulkersonův algoritmus



Obrázek 2.4: Edmondsův–Karpův algoritmus

3 Implementace

3.1 Implementace načtení dat ze souborů

Po spušetění program zkontroluje počet vstupních argumetů a zdali byli všechny povinné argumenty zadány. Pokud kontrola proběhla úspěšně voláme funkci *vertex_loader*(*char* input_vertex_file*), která příjmá jako vstupní argument funkce 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 vectoru načtených vrcholů existuje tak se netvoří istance načteného vrchalu. Rozdělené atributy jsou předány funkci *vertex_create*(), která příjmá jako argumenty načtené rozdělené atributy a funkce vrací pointer na dynamicky alokovanou paměť struktury vertex. Po načtení všech vrcholů grafu funkce vrátí ukazetel na dynamicky alokovanou paměť naplněného vectoru, který obsahuje ukazetele na strukturu vertex, prezentujíc vrchol grafu.

Jako další je potřeba načíst hrany, spojující příslušné vrcholy grafu. Načítaní hran zajištuje 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 ukazetele na dynamicky alokovanou struktury edge, reprezentující hranu grafu. Poslední krokem před tvorbou matice kapacit je kontrola zdali zadaný počátení a koncový vrchol se nachází v načteném grafu a zdali obě tyto hodnoty nejsou totožné.

3.1.1 Popis strutur pro uložení dat

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

```
typedef struct _vertex
{
    int id;
    char *WKT;
} vertex;
```

Zdrojový kód 3.1: Struktura `vertex`

Struktura vertex obsahuje tyto atributy:

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

Struktura edge reprezentuje hranu grafu.

```
typedef struct _edge
{
    int id;
    int source;
    int target;
    int capacity;
    char *isvalid;
    char *WKT;
}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 - Ukaztel na dynamicky alokovanou paměť pro uložení validity hrany (Textová reprezentace)
- WKT - ukazetel na dynamicky alokovanou paměť pro souřadnice hrany grafu (Textová reprezentace).

Struktura vector reprezentuje dynamicky reallcované pole.

```
typedef struct _vector_t {
    size_t count;
    size_t capacity;
    size_t item_size;
    void *data;
    vec_it_dealloc_t deallocator;
} vector_t;
```

Zdrojový kód 3.3: Struktura `vector`$_t$

Struktura vector_t 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 allokovaný 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, dný 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 úspěšně funkce začné číst další řádky, které obsahují příslušná data o vrcholech, které rozděli na příslušné atributy struktury vertex, a zavoláním funkce *vertex_create*(), které předá načtené atributy, získá ukazatel na dynamicky alokovanou paměť struktury vertex, který uloží do dynamicky alokované struktury vector_t představující dynamicky realocované 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 úspěšně funkce začné číst další řádky, které obsahují příslušná data o edge, které rozděli na příslušné atributy struktury edge, a zavoláním funkce *edge_create*(), které předá načtené atributy, získá ukazatel na dynamicky alokovanou paměť struktury edge, který uloží do dynamicky alokované struktury vector_t představující dynamicky realocované 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 a neuloží do pole uchovávající načítaná data.

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 řetezec, který vloží do knihovní funkce strcmp, kde porovna tento řetezec s definovaným řetezcem hlavičky a pokud jsou shodné vrátí hodnotu 0, pokud jsou odlišené 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 maximálí tok. Funkce prochazí celé předané pole a hledá shodu s předaným id počátečního nebo koncového vrcholu. Pokud je shoda nalezena vratíme 0, pokud se ale vrchol v grafu nenachází vratíme hodnotu 1, která identifikuje, že vrchol se v grafu nenachazí a funkce volající výšše uvedenou funkci program ukončí.

3.2 Implementace tvorby grafu

Jako strukturu pro tvorbu grafu jsem zvolil matici sousednosti, která velmi efektivně umožnuje 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 imaginárnách 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 vectorů přecházíme k samotné tvorbě matice kapacit. K vytvoření samotné matice kapacit využíváme metodu *capacity_matrix()*, funkce příjmá jako vstupní agumety námi vytvořená pomocná pole id a pole načtených hran. Funkce využívá pomocná pole id k určení velikosti matice a z příslušných vrcholů grafu získat kapacitu hrany hran, kterou následně nastavim 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 vectoru hran, vratíme ukazatel na dynamicky alokovanou paměť matice kapacit.

3.2.1 Popis strutur 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 příjmá jako vstupní argumetn načtené pole vrcholů. Funkce alokuje paměť pro strukturu vector_t, která v sobě bude ukládat číselnou hodnotu id hran, které byli načtny. Po úspěšné alokaci paměti pro vector_t uložím do vectoru číselnou načteného id. Po uložení všech id hran vratíme ukazatel na tuto dynamicky alokovanou paměť struktury vector_c, která má v sobě uložené hodnoty skutečných id hran.

get_image_vector()

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

capacity_matrix()

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

get_image_id_from_real()

Funkce příjmá 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 vratí z pole imaginárních id jeho imaginární id.

get_real_id_from_image()

Funkce příjmá 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 vratí z pole skutečnách id jeho skutečné id.

get_edge()

Funkce příjmá 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, vratíme 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 úspěšné přechazíme 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 vratí hodnotu NULL, hledání maximálnímu toku končí, vypíšeme nalezný maximální tok a podle jeho hodnoty vratíme hodnotu NULL, když je minimální tok roven nule a nebo ukazatel na dynamicky alokovanou strukturu vector_t, která uchovává hrany zjištěné při vyhledávaní 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),
    vector_count(vector_vertex), 0);
15      if(!residual) {
16          printf("Residual␣matrix␣allocation␣memory␣was␣not␣
    obtained.\n");
17          return NULL;
18      }
19
20      parents = (int *)malloc(vector_count(vector_vertex) *
    sizeof(int));
21      if(!parents) {
22          printf("Ancestor␣array␣allocation␣memory␣not␣retrieved
    .\n");
23          matrix_free(&residual);
24          return NULL;
25      }
26
27      minimal_cut = vector_create(sizeof(edge *), NULL);
28      if(!minimal_cut) {
29          printf("Memory␣for␣minimum␣slice␣vector␣allocation␣was␣
    not␣obtained.\n");
30          matrix_free(&residual);
31          free(parents);
32          return NULL;
33      }
34
35      for(i = 0; i < vector_count(vector_vertex); ++i) {
36          for(v = 0; v < vector_count(vector_vertex); ++v) {
37              matrix_set(residual, i, v, matrix_get(capacity, i,
    v));
38          }
39      }
40
41      maxFlow = 0;
42      parents = BFS(residual, source, target, parents,
    vector_vertex);
43
44      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          return NULL;
10     }
11
12     visiter = (int *)malloc(vector_count(vector_vertex) *
       sizeof(int));
13     if(!visiter) {
14         printf("The␣visiter␣array␣has␣not␣been␣created.\n");
15         return NULL;
16     }
17
18     for(i = 0; i < vector_count(vector_vertex); ++i) {
19         visiter[i] = 0;
20     }
21
22     q = queue_create();
23     if(!q) {
24         printf("The␣queue␣data␣structure␣allocation␣memory␣was␣
       not␣retrieved.\n");
25         free(visiter);
26         visiter = NULL;
```

```
    }

    q = enQueue(q, source);

    visiter[source] = 1;
    parents[source] = -1;

    while (q->front != NULL) {
        u = get_front_element(q);
        if(u == -1) {
            q = clean_queue(q);
            free(q);
            q = NULL;
            free(visiter);
            visiter = NULL;
            free(parents);
            parents = NULL;
            return NULL;
        }

        q = deQueue(q);

        for(i = 0; i < vector_count(vector_vertex); ++i) {
            if(visiter[i] == 0 &&  0 < matrix_get(residual, u,
    i)) {
                if(i == target) {
                    parents[i] = u;
                    q = clean_queue(q);
                    free(q);
                    q = NULL;
                    free(visiter);
                    visiter = NULL;
                    return parents;
                }

                q = enQueue(q, i);
                parents[i] = u;
                visiter[i] = 1;
            }
        }
    }
```

```
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 fronnty

```
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 přidáváná hodnota. Po úspěšné alokaci paměti a nastavení příslušné hodnoty do alokované struktury queue_item vratíme ukazaten 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 vratím 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 strukturuy 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 úspšném přidaní v metoda vratí 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 nasledující prvek ve frontě a uložený prvek v pomocné proměnné je uvolně a funkce vratí ukzatel na strukturu queue s odebraným prvek.

clean_queue()

Funkce pro odstranění zbylých prvků uložených ve frontě. Funkce přijmá 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, vratíme 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 prohledaní do šířky, který nalezne cestu grafem. Podrobnější popis algoritmu BFS, lze naleznout zde[1].

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. Fumkce 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 vectoru tak je daná hrana zapsána do souboru. Po skončení zápisu je souboru uzavřen a vracena hodnota 0, nastaneli nějaký problém v průběhu zapisu, bude vrácena hodnota 1.

[1] BFS

4 Uživatelský manuál

4.1 Spuštení 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 možé jí spustit. Aplikace se spouští příkazem „./flow.exe -v <soubor vrcholu> -e <soubor hran> -s <svtupní vrchol> -t<výstupní vrchol>“, dale 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 <svtupní 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 parametrù, či parametry nebudou pro spustění aplikace validní, aplikace vypíše příslušnou hlášku a ukončí program.

4.2 Průběh programu

Po zadání vstupních argumentů aplikace budou tyto argumenty zkontrolovany, pokud jsou validní načteme ze souboru vrcholy grafu a hrany grafu. Program provede potřebný výpočet a na konci 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 output.o main.o -o flow.exe -Wall -Wextra -pedantic -ansi -g

4.3.2 Spuštění pogramu a jeho výstup

D:\PC\SP_V4>flow.exe -v ./example/example_nodes.csv -e ./example/example_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 program bylo nalezení maximálního toku, který měl hodnotu 4 a ve výstupním souboru byli zaznámenány 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 date, 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 jednote 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 předavající se struktury vector_t.

6 Zdroje

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

2.1 Obrázek ukazjící reprezentaci grafu seznami vrcholů a hran:
https://www.youtube.com/watch?v=ZdY1Fp9dKzs
2.3 Obrázek ukazjící 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 rùzné problémy: https://stackoverflow.com