Semestrální práce

Největší klika v neorientovaném grafu

IAL

Lukáš Lev, 256660

November 27, 2024

1 Zadání

Náhradní projekt je určen pouze pro studenty, kteří v předmětu IFJ neřeší souběžný projekt (např. studenti FEKT nebo studenti opakující předmět). Tento projekt je týmový a řeší jej trojice nebo čtveřice studentů.

Zadání varianty *Klika grafu* je podgraf, který je úplným grafem (=kterýkoliv vrchol kliky je tedy spojen hranou se všemi ostatními vrcholy kliky).

Vytvořte program pro hledání největší kliky v neorientovaném grafu. Pokud existuje více řešení, nalezněte všechna. Výsledky prezentujte vhodným způsobem. Součástí projektu bude načítání grafů ze souboru a vhodné testovací grafy. V dokumentaci uveďte teoretickou složitost úlohy a porovnejte ji s experimentálními výsledky.

Všeobecné informace a pokyny k náhradním projektům
Řešení bude vypracováno v jazyce C a bude přeložitelné (pomocí příkazu make) na serveru eva.fit.vutbr.cz. Všechny zdrojové kódy, hlavičkové soubory, testovací data aj. budou logicky separovány a uloženy v příhodně pojmenovaných podadresářích. Použití nestandardních knihoven není dovoleno. Všechny části zadání varianty jsou nutnou součástí řešení.

Celkové hodnocení projektu sestává z následujících kategorií:

- funkčnost implementace (až 6 bodů),
- projektová dokumentace (až 4 body),
- obhajoba (až 5 bodů).

Řešení zabalené v jediném ZIP archivu je odevzdáváno pouze vedoucím týmu prostřednictvím STUDISu. Závazné pokyny pro vypracování projektové dokumentace a doporučení pro závěrečné obhajoby naleznete v Moodlu v sekci Projekty.

2 Abstrakt

Tento projekt byl proveden podle zadání z kapitoly **??**, jež bylo poskytnuto vyučujícím.

Předmětem tohoto projektu je hledání **největší kliky v neorientovaném grafu**, což je jeden z typických problémů v teorii grafů **TODO: citace**. Tato problematika je krátce popsána v kapitole **??**.

Pro hledání největší kliky v neorientovaném grafu byly navrženy dva algoritmy, a to sice

- algoritmus metodou hrubé síly (anglicky *brute force*),
- algoritmus zpětného vyhledávání (anglicky *backtracking*).

Pro každý z těchto algoritmů byla stanovena časová komplexita nejprve teoreticky a následně také experimentálně pomocí jednoduchého programu v jazyce C.

3 Úvod

3.1 Teorie

3.1.1 Zkoumané objekty

Graf je základní objekt teorie grafů. Skládá se z uzlů (vrcholů) a hran.[**?**]

dokumentace/pic/FIT_trans.png

Neorientovaný graf je graf, jehož všechny uzly jsou symetricky spojeny neorientovanou hranou.[**?**]

Úplný graf je neorientovaný graf, pro jehož každou dvojici vrcholů existuje právě jedna neorientovaná hrana.[**?**]

Klika (anglicky *clique*) je podmnožina vrcholů neorientovaného grafu. Tato podmnožina tvoří úplný graf.[**?**]

3.1.2 Teorie použitých algoritmů

4 Implementace v jazyce C

Podle zadání (kapitola **??**), byly oba algoritmy vypracovány v jazyce C. Další rysy implementace, které vycházejí ze zadání, se týkají zavedených datových struktur (kapitola **??**). Implementace obou algoritmů vykazují požadované chování a v případě výskytu několika klik o maximální velikosti nacházejí všechny tyto výsledky. Zároveň, nenachází-li se v grafu žádná hrana, jsou nalezeny všechny vrcholy jako největší kliky.

4.1 Reprezentace neorientovaného grafu

Podle zadání (kapitola **??**) je třeba vytvořit takovou datovou strukturu neorientovaného grafu, kterou lze snadno reprezentovat záznamem do jednoduchého souboru.

Jednou z možných variant je **matice sousednosti**. Tato matice je čtvercová a, jelikož v rámci implementace neuvažujeme hrany uzlů vedoucí na sebe sama, diagonála této matice je nulová. Pro ostatní prvky platí, že vyjadřují přítomnost hrany mezi uzly s indexem shodným s řádkem či sloupcem matice.

Následující rovnice uvádí názorný příklad matice sousednosti (grafická reprezentace této matice je na obr. **??**). Prvky této matice jsou hrany mezi vrcholy označenými indexem. Tedy například prvek $h_{04} = 1$ tvrdí, že mezi uzly 0 a 1 se nachází hrana, zatímco mezi uzly 0 a 1 nikoliv ($h_{01} = 0$).

$$\mathrm{M_s} = \begin{pmatrix} h_{00} & h_{01} & h_{02} & h_{03} & h_{04} \\ h_{10} & h_{11} & h_{12} & h_{13} & h_{14} \\ h_{20} & h_{21} & h_{22} & h_{23} & h_{24} \\ h_{30} & h_{31} & h_{32} & h_{33} & h_{34} \\ h_{40} & h_{41} & h_{42} & h_{43} & h_{44} \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix} \tag{1}$$

Zdrojový kód pro implementaci této matice se nachází v souboru `graph.c`. V něm je definována struktura `graph` popsaná níže.

```
1 typedef struct {
2     int size;  // velikost matice (odpovida poctu uzlu)
3     int** matrix;  // 2D pole pro ulozeni prvku matice
4 } graph;
```

Listing 1: Definice struktury neorientovaného grafu.

Soubor `graf.c` také obsahuje řadu funkcí vztahujících se k těmto grafům. Těmito funkcemi jsou:

- `graph* graph_init(int size)` pro inicializaci prázdného grafu s vhodnou velikostí,
- `void graph_delete(graph* g)` pro smazání grafu a uvolnění jeho paměti,
- `int graph_read_size(const char* filename)` pro přečtení velikosti grafu (počtu uzlů) v matici sousednosti ze souboru,

dokumentace/pic/FIT_trans.png

- `int graph_read(graph* g, const char* filename)` pro přečtení matice sousednosti ze souboru, vlastnosti grafu jsou uloženy do vstupní proměnné `g` (nevyužívá funkci `graph_read_size`,
- `int graph_write(graph* g, const char* filename)` pro zápis matice sousednosti do souboru,
- `void graph_print(graph* g)` pro vytištění grafu do terminálu

Soubory, které slouží pro ukládání grafu, jsou označeny příponou `.gh` a matici sousednosti uchovávají v následujícím formátu, kde první řádek reprezentuje velikost matice a ostatní řádky informace o jejich prvcích:

```
5
0 0 0 1 1
0 0 1 0 1
0 1 0 1 0
1 0 1 0 1
1 1 0 1 0
```

Listing 2: Zápis matice v `.gh` souboru.

Tato matice je reprezentována také graficky na obrázku **??**.

dokumentace/pic/graf_1.png

Figure 1: Grafická reprezentace neorientovaného grafu z kapitoly **??**.

Bližší popis implementovaného kódu je součástí komentářů v souboru `graph.c`.

4.2 Implementace algoritmu metody hrubou silou

Zdrojový kód pro algoritmus hledání největší kliky v neorientovaném grafu metodou hrubé síly (anglicky *brute force*) je obsahem souboru `algorithms/bruteforce.c`. Kód operuje se strukturou definovanou v kapitole **??**. Samotný soubor obsahuje podrobné komentáře, a tak je shrnutí implementace v této kapitole pouze zběžné.

Protože matice sousednosti obsahuje pouze prvky s hodnotou 1 nebo 0, byla zvolena reprezentace vybrané podmnožiny vrcholů pomocí binární masky. Na tu je referováno celým číslem `int subset`, které po převodu do binární soustavy určuje, které uzly jsou vybrány. Indexace se shoduje s indexací uzlů v matici.

dokumentace/pic/FIT_trans.png

Například pro matici definovanou v souboru v ukázce kódu **??** použijeme masku `subset` určenou číslem 10. Pro masku `subset` tak platí:

$$10_{(10)} = 01010_{(2)} \implies u_0u_1u_2u_3u_4 \tag{2}$$

Proto jsou maskou vybrány vrcholy na indexech 1 a 3 graficky reprezentovány na obrázku **??**. Pro vytvoření všech podgrafů vstupního grafu definovaného maticí sousednosti stačí iterativně inkrementovat hodnotu masky od nuly až do hodnoty $\mathbf{2^n}$, kde n je počet uzlů v grafu. Při iteraci těmito podgrafy stačí sledovat, zda jsou klikami, a pokud ano, pak také jejich velikost. Největší kliky jsou uchovávány v proměnné `int** largest_cliques` ve funkci `void bruteforce(graph* g)`.

dokumentace/pic/graf_1_13.png

Figure 2: Označení uzlů neorientovaného grafu maskou o hodnotě 10.

Soubor `algorithms/bruteforce.c` obsahuje následující funkce:

- `int is_clique_bruteforce(graph* g, int subset)` pro kontrolu, zda podgraf označený maskou `subset` je klikou (zda je matice, krom diagonály, naplněna hodnotami 1),
- `void bruteforce(graph* g)` pro iterativní hledání největší kliky. Tato funkce vytiskne všechny největší nalezené kliky do terminálu.

4.3 Implementace algoritmu metody zpětného vyhledávání

Zdrojový kód pro hledání největší kliky v neorientovaném grafu pomocí metody zpětného vyhledávání je obsažen v souboru `algorithms/backtracking.c`. V něm je pro tuto metodu použito následujících funkcí:

- `int is_clique_backtracking(graph* g, int* clique, int clique_size, int vertex)` pro kontrolu, zda přidání uzlu (neboli vrcholu, tedy `vertex`) zachová vlastnost kliky podgrafu `g`,
- `void find_clique_backtracking(graph* g, int* current_clique, int clique_size, int*** largest_cliques , int* max_size, int* clique_count, int start)` pro rekurzivní hledání největší kliky,
- `void backtracking(graph* g)` pro správu proměnných pro obě výše zmíněné funkce a jejich tisk do konzole.

dokumentace/pic/FIT_trans.png

Bližší informace o fungování kódu jsou uvedeny v komentářích souboru `algorithms/backtracking.c`. Následující popis je pouze stručné shrnutí.

Funkce `void backtracking(graph* g)`, jež je volána z vnějšího prostředí, spravuje proměnné `int* current_clique`, která je polem vrcholů tvořících právě analyzovanou kliku, `int** largest_cliques`, která je polem ukazatelů na pole vrcholů tvořících největší nalezené kliky, `int max_size`, která je velikost největší nalezené kliky, a `int clique_count`, která slouží jako počítadlo největších nalezených klik.

Tyto proměnné jsou použity při volání `find_clique_backtracking(g, current_clique, 0, &largest_cliques, &max_size, &clique_count, 0)`. V této funkci se po ověření vzniku nové kliky (funkcí `is_clique_backtracking`) rekurzivně volá opět funkce `find_clique_backtracking` s aktualizovanými parametry. Pro každou iteraci je ověřeno, zda právě analyzovaná klika má být uložena jako největší.

Princip zpětného vyhledávání je zde zaopatřen vynořením z rekurze a přepisem dříve zkoumaných `current_clique`. Díky tomu je na konci implementace zapotřebí uvolňovat pouze paměť alokovanou pro všechna nalezená řešení a `current_clique`.

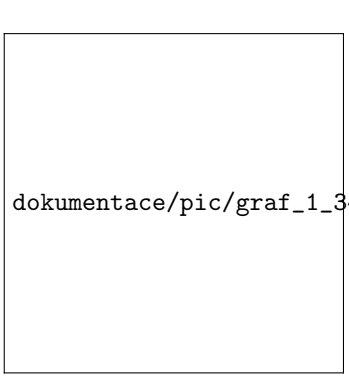
```
1  void find_clique_backtracking(graph* g, int* current_clique, int clique_size, int
       *** largest_cliques, int* max_size, int* clique_count, int start) {
2      if (clique_size > *max_size) {
3          *max_size = clique_size;
4          for (int i = 0; i < *clique_count; i++) {
5              free((*largest_cliques)[i]);
6          }
7          free(*largest_cliques);
8          *largest_cliques = NULL;
9          *clique_count = 0;
10     }
11     if (clique_size == *max_size) {
12         *largest_cliques = realloc(*largest_cliques, (*clique_count + 1) * sizeof(
               int*));
13         (*largest_cliques)[*clique_count] = malloc(clique_size * sizeof(int));
14         for (int i = 0; i < clique_size; i++) {
15             (*largest_cliques)[*clique_count][i] = current_clique[i];
16         }
17         (*clique_count)++;
18     }
19     for (int i = start; i < g->size; i++) {
20         if (is_clique_backtracking(g, current_clique, clique_size, i)) {
21             current_clique[clique_size] = i;
22             find_clique_backtracking(g, current_clique, clique_size + 1,
                   largest_cliques, max_size, clique_count, i + 1);
23         }
24     }
25 }
```

Listing 3: Zjednodušený kód pro rekurzivní funkci využitou při implementace metody zpětným vyhledáváním.

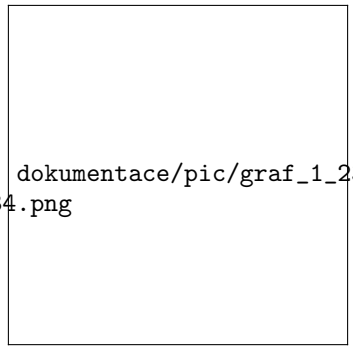
Jednoduchá grafická reprezentace přepisu prvků podgrafu v proměnné `current_clique` je znázorněno na obrázku **??**. Důležitou poznámkou však je, že pořadí zpracování indexů se nemusí shodovat s pořadím v programu.

5 Experiment

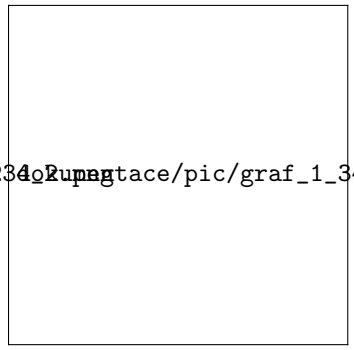
`dokumentace/pic/FIT_trans.png`



(a) Podgraf s uzly 3 a 4.



(b) Přidání uzlu 2 nezachová vlastnost kliky podgrafu.



(c) Přepis falešného uzlu kliky dalším podezřelým.

Figure 3: Ilustrace k přepisu uzlů v proměnné `current_clique` pro hledání zpětným vyhledáváním (pořádí uzlů nemusí odpovídat pořadí v programu).