Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Hľadanie súvislých podgrafov v grafe

Bakalárska práca

2024
Mikalai Halavachenka

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Hľadanie súvislých podgrafov v grafe

Bakalárska práca

Študijný program: Aplikovaná informatika
Študijný odbor: Informatika
Školiace pracovisko: Katedra aplikovanej informatiky
Školiteľ: Mgr. Dominika Mihálová

Bratislava, 2024
Mikalai Halavachenka

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Mikalai Halavachenka
Študijný program: aplikovaná informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Hľadanie súvislých podgrafov v grafe
Finding connected subgraphs of a graph

Anotácia: Cieľom bakalárskej práce je implementovať efektívny algoritmus pre hľadanie všetkých podgrafov v grafe. Pri hľadaní podgrafov bude možné zadať parametre ako napr. počet vrcholov, priemer, veľkosť stupňa vrcholov a iné.

Vedúci: Mgr. Dominika Mihálová
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: doc. RNDr. Tatiana Jajcayová, PhD.

Dátum zadania: 04.10.2023

Dátum schválenia: 05.10.2023

doc. RNDr. Damas Gruska, PhD.
garant študijného programu

....

študent vedúci práce

Poďakovanie: Tu môžete poďakovať školiteľovi, prípadne ďalším osobám, ktoré vám s prácou nejako pomohli, poradili, poskytli dáta a podobne.

Abstrakt

Slovenský abstrakt v rozsahu 100-500 slov, jeden odstavec. Abstrakt stručne sumarizuje výsledky práce. Mal by byť pochopiteľný pre bežného informatika. Nemal by teda využívať skratky, termíny alebo označenie zavedené v práci, okrem tých, ktoré sú všeobecne známe.

Kľúčové slová: jedno, druhé, tretie (prípadne štvrté, piate)

Abstract

Abstract in the English language (translation of the abstract in the Slovak language).

Keywords:

Obsah

Úvod **1**

1 Východiská práce **3**

1.1 Terminológia a definície teórie grafov 3

1.1.1 Graf 3

1.1.2 Podgraf 3

1.1.3 Stupeň vrcholu 4

1.1.4 Cesta 4

1.1.5 Súvislé grafy 4

1.2 Použité technológie 5

1.2.1 C++ 5

2 Algoritmy vyhľadávania podgrafov **7**

2.1 ConSubg 7

2.1.1 CombinationsFromTree 7

2.1.2 BuildTree 9

2.1.3 CombinationTree 10

2.1.4 CombinationsWithV 10

2.1.5 ConSubg 11

2.1.6 Testovanie algoritmu 11

2.2 Reverse Search 12

2.2.1 isValidExtension 12

2.2.2 EnumerateCIS 13

2.2.3 ReverseSearch 13

2.2.4 Testovanie algoritmu 14

2.3 Simple Forward 14

2.3.1 simpleForward 14

2.3.2 getSubgraphs 15

2.3.3 Testovanie algoritmu 15

3 Návrh **17**
3.1 Analýza požiadaviek 17
3.1.1 Popis konzolovej aplikácie 17
3.1.2 Funkcionálne požiadavky 18
3.1.3 Nefunkcionálne požiadavky 18

Záver **19**

Príloha A **23**

Príloha B **25**

Zoznam obrázkov

1.1 Graf 4
1.2 Podgrafy veľkosti 3 grafa z príkladu 1.1.1 4
1.3 Cesty 5
1.4 Súvislý a nesúvislé grafy 5

Zoznam tabuliek

2.1 Tabuľka popísaných algoritmov 7
2.2 Experimenty na náhodných grafoch 12
2.3 Funkcie algoritmu Reverse Search 12

Úvod

todo

Kapitola 1

Východiská práce

Táto kapitola je rozdelená na n častí: vysvetlenie pojmov z teórie grafov, predstavenie algoritmov na vyhľadávanie podgrafov...

1.1 Terminológia a definície teórie grafov

V tejto časti uvádzame definície z teórie grafov, ktoré súvisia s našou prácou.

1.1.1 Graf

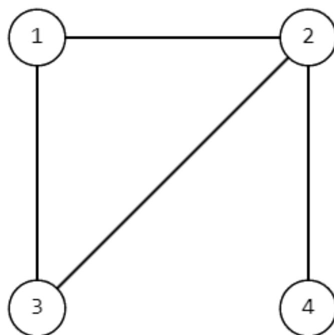
Graf je matematická abstrakcia reálneho systému akejkoľvek podstaty, ktorej objekty majú párové vzťahy. Grafy majú veľmi široké využitie, napríklad schémy leteckých spoločností, chemické štruktúry, genealogický strom, železničné schémy a mnoho ďalších príkladov. Najjednoduchšou abstrakciou systémov, ktorých prvky majú párové vzťahy, je neorientovaný graf.

Definícia 1.1.1 *Graf alebo neorientovany graf G je usporiadaná dvojica množín $G = (V, E)$ takých, že $E \subseteq V \times V$. Kde E je množina neusporiadaných dvojíc prvkov neprázdnej množiny V. Množina V je množina vrcholov a E je množina hrán. Ak je G graf, potom $V = V(G)$ je množina vrcholov G a $E = E(G)$ je množina hrán. Hrana $\{x, y\}$ spája vrcholy x a y a označuje sa xy. Teda xy a yx označujú tú istú hranu; vrcholy x a y sú konečné vrcholy tejto hrany. Ak $xy \in E(G)$, potom x a y sú susedné vrcholy G a vrcholy x a y sú incidentné s hranou xy[2,* **?***].*

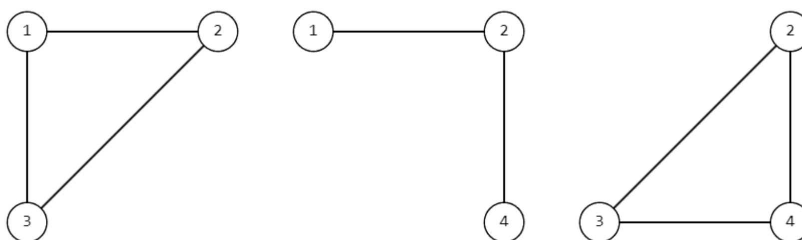
Príklad 1.1.1 *Na obrázku 1.1 je príklad grafu s množinami $V = \{1, 2, 3, 4\}$, $E = \{12, 23, 31, 24\}$.*

1.1.2 Podgraf

Definícia 1.1.2 *Graf H je podgrafom grafu G, ak $V(H) \subseteq V(G)$ a $E(H) \subseteq E(G)$[8,* **?***].*



Obr. 1.1: Graf



Obr. 1.2: Podgrafy veľkosti 3 grafa z príkladu 1.1.1

Príklad 1.1.2 *Na obrázku 1.2 sú podgrafy veľkosti 3 grafa z príkladu 1.1.1.*

1.1.3 Stupeň vrcholu

Definícia 1.1.3 *Stupeň $d_G(v) = d(v)$ vrcholu v je počet hrán s vrcholom v, tento počet sa rovná počtu susedov vrcholu v. Vrchol stupňa 0 sa nazýva izolovaný vrchol [3, **?**].*

Príklad 1.1.3 *Stupeň vrcholu 1 z príkladu 1.1.1 je $d_G(1) = 2$, vrcholu 2 je $d_G(2) = 3$.*

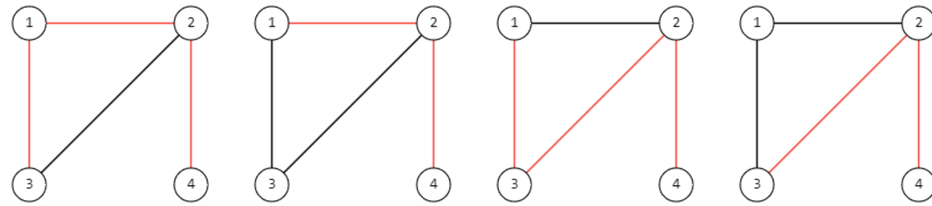
1.1.4 Cesta

Definícia 1.1.4 *Cesta je neprázdny graf $P = (V, E)$, kde $V = v_1, v_2, \dots, v_k$, $E = v_1v_2, v_2v_3, \dots, v_{k-1}v_k$. Cestu často označujeme prirodzenou postupnosťou jej vrcholov $P = v_1v_2 \dots v_k$ a voláme P cestou od v_1 po v_k. Dĺžka cesty je počet hranou v ňou [3, **?**].*

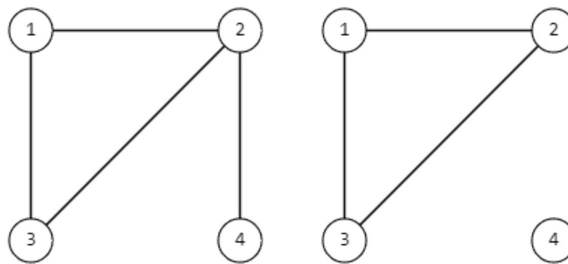
Príklad 1.1.4 *V grafe môže existovať niekoľko ciest z vrcholu x do vrcholu y. Na obrázku 1.3 sú zobrazené 2 cesty z vrcholu 1 do vrcholu 4 a 2 cesty z vrcholu 3 do vrcholu 4.*

1.1.5 Súvislé grafy

Definícia 1.1.5 *Graf G sa nazýva súvislý, ak sú ľubovoľné dva jeho vrcholy spojené cestou v grafe G[3, **?**].*



Obr. 1.3: Cesty



Obr. 1.4: Súvislý a nesúvislé grafy

To znamená, že prechádzajúc grafom sa dá dostať do ktoréhokoľvek vrcholu alebo že v grafe neexistujú izolované vrcholy.

Príklad 1.1.5 *Na obrázku 1.4 sú zobrazené 2 grafy. Prvý graf je súvislý, druhý nie. Vrchol 4 druhého grafu je izolovaný.*

1.2 Použité technológie

1.2.1 C++

Jazyk C++ navrhol a implementoval Bjarne Strustrup v laboratóriách AT&T Bell Laboratories s cieľom spojiť organizačné a konštrukčné výhody jazyka Simula so systémovými programovými možnosťami jazyka C. Pôvodná verzia jazyka C++ s názvom "C with Classes"(C s triedami) bola prvýkrát použitá v roku 1980; Podporovala tradičné techniky systémového programovania a abstrakciu údajov. Základné nástroje objektovo orientovaného programovania boli pridané v roku 1983 a do komunity C++ sa postupne zaviedli techniky objektovo orientovaného návrhu a programovania. Jazyk bol prvýkrát komerčne dostupný v roku 1985. V rokoch 1987 až 1989 boli do jazyka pridané všeobecné programovacie prostriedky[6].

Programovací jazyk C++ poskytuje pamäťový a výpočtový model, ktorý sa presne zhoduje s modelom väčšiny počítačov. Okrem toho poskytuje výkonné a flexibilné mechanizmy abstrakcie, t. j. jazykové konštrukcie, ktoré umožňujú programátorovi zavádzať a používať nové typy objektov, ktoré zodpovedajú konceptom aplikácie. Jazyk C++ teda podporuje štýly programovania, ktoré sa spoliehajú na pomerne priamu

manipuláciu s hardvérovými prostriedkami, ktoré poskytujú vysoký stupeň efektívnosti, ako aj programovacie štýly vyššej úrovne, ktoré sa spoliehajú na používateľom definované typy, ktoré poskytujú model údajov a výpočtov, ktorý je bližší ľudskému pohľadu na úlohy vykonávané počítačom. Tieto štýly programovania vyššej úrovne sa často označujú ako abstrakcia údajov, objektovo orientované programovanie a generické programovanie[7].

Kapitola 2

Algoritmy vyhľadávania podgrafov

2.1 ConSubg

Algoritmus ConSubg(k, G), ktorý predložili Karakashian, Choueiry a Hartke, je určený na nájdenie všetkých spojených podgrafov veľkosti k v grafe G. Autori poukazujú na to, že algoritmus využíva štruktúru grafu na zabránenie vzniku nesúvislých podgrafov, a preto je výhodný najmä na vyhľadávanie vo veľkých riedkych grafoch. Algoritmus má také vlastnosti, ako je vytvorenie kombinačného stromu a operátor $\otimes_t$, ktorý umožňuje generovať podgrafy bez duplicít[4].

Tabuľka 2.1: Tabuľka popísaných algoritmov

Algoritmus	Volá algoritmus	Pseudokód
ConSubg	CombinationsWithV	Algoritmus 1
CombinationsWithV	CombinationTree CombinationsFromTree	Algoritmus 2
CombinationTree	BuildTree	Algoritmus 3
BuildTree	BuildTree	Algoritmus 2
CombinationsFromTree	CombinationsFromTree	Algoritmus 1

ConSubg sa skladá z 5 menších algoritmov. V Tabuľke 2.3 sú uvedené názvy týchto algoritmov, názvy algoritmov, ktoré interne volajú a označenie pseudokódu. Predstavenie algoritmov začneme od konca, aby algoritmy, ktoré v sebe volajú ďalšie algoritmy, už boli popísané predtým.

2.1.1 CombinationsFromTree

Algoritmus 1 je CombinationsFromTree(root, k), ktorý ako vstup prijíma koreň kombinačného stromu a celé číslo k. Na začiatku sa inicializujú premenné ako t, ktorá

Algorithm 1 CombinationsFromTree

```
1:  t ← root
2:  lnodesets ← ∅
3:  if k = 1 then return {{t}}
4:  end if
5:  for i ← 1 to min(|child(t)|, (k − 1)) do
6:      kCombs ← kCombinations(i, child(t))
7:      kCompos ← kCompositions(i, (k − 1))
8:      for all NodeComb ∈ kCombs do
9:          for all StrComp ∈ kCompos do
10:             fail ← false
11:             for pos ← 1 to i do
12:                 stRoot ← NodeComb[pos − 1]
13:                 size ← StrComp[pos − 1]
14:                 S[pos − 1] ← combinationsFromTree(stRoot, size)
15:                 if S[pos − 1] = ∅ then fail ← true and break
16:                 end if
17:             end for
18:             if fail is true then continue
19:             end if
20:             combProduct ← S[0]
21:             for j ← 1 to i do
22:                 combProduct ← combProduct ⊗_t S[i]
23:             end for
24:             lnodesets ← lnodesets ∪ combProduct
25:         end for
26:     end for
27: end for
28: return lnodesets
```

uchováva uzol stromu, a *lnodesets*, ktorá uchováva množiny uzlov stromu obsahujúce koreň stromu. Ak $k = 1$, vráti sa t. Potom sa vykoná cyklus pre každé i od 1 do $min(|Childs(t)|, (k-1))$. V ktorom pomocou metód kCombinations a kCompositions vytvorí všetky kombinácie prvkov z množiny $Childs(t)$ veľkosti i a všetky reťazce dĺžky i na intervale celých čísel $[1, (k-i+1)]$ tak, že súčet prvkov reťazca sa rovná k. Potom pomocou získaných kombinácií a reťazcov v cycle metoda zavolá samu seba a vrátený výsledok uloží do $S[pos]$. Ak výsledok je , tak sa cyklus presunie na ďalšiu iteráciu. Ak nie, tak potom sa spustí metóda unionProduct(S_1, S_2) pre každy prvok S od 1 do i, ktorá vráti množiny uzlov získané binárnou operáciou $\otimes_t$ medzi množinami množín S_1 a S_2:

$$S_1 \otimes_t S_2 = \begin{cases} \emptyset, ak\ S_1 = \emptyset \\ S_1, ak\ S_2 = \emptyset \\ \{x | (x = s_1\ s_2) \wedge (s_1 \in S_1) \wedge (s_2 \in S_2) \\ \wedge(\forall i \in s_1, j \in s_2, Vrchol(i) \neq Vrchol(j)) \wedge ((\exists j \in s_2 \text{Vrchol(j) je novy}) \\ \vee(\forall i \in s_1, j \in s_2, l \in Child(i), Vrchol(l) \neq Vrchol(j))), inak\} \end{cases}$$

Výsledok uloží do lnodesets.
Časová zložitosť uvedená autormi pre operáciu $S_1 \otimes_t S_2$ je $O(|S_1| * |S_2| * |s_1| * |s_2|)$, kde $|s_1|$ a $|s_2|$ sú rozmery najväčších prvkov S_1 a S_2 resp.

2.1.2 BuildTree

Algorithm 2 BuildTree

```
1: list[depth] ← list[depth − 1]
2: for all v ∈ Neighbors(node.vertex) do
3:     if v ∉ list[depth] then
4:         newnode.vertex ← v
5:         Child(node) ← {newnode} ∪ Child(node)
6:         list[depth] ← list[depth] ∪ {newnode}
7:         if v ∉ visited then visited ← visited ∪ {v}
8:         elsenewnode.new ← false
9:         end if
10:        if depth + 1 < k then BuildTree(newnode, depth+1, g, k)
11:        end if
12:    end if
13: end for
```

Algoritmus 2 BuildTree(node, depth, G, k). Na vstupe prijíma uzol kombinačného stromu *node*, celé čísla *depth* a *k* a graf *G*. Najprv inicializuje prvok s indexom *depth*

v *list* tak, že tam uloží $list[depth-1]$. Potom sa spustí cyklus, ktorý skontroluje každého suseda v vrcholu grafu zodpovedajúceho uzlu *node*, či v môže byť pridaný do stromu ako potomok *node*. Na začiatku cyklu sa kontroluje, či $list[depth]$ obsahuje v, ak nie, potom sú ďalšie kroky nasledovné. Najprv sa vytvorí uzol kombinačného stromu *newnode*, ktorý zodpovedá vrcholu v a je potomkom *node*. Potom sa tento vrchol pridá do $list[depth]$. Ďalej ak vrchol v nebol doteraz navštívený, označí sa ako navštívený, ak bol doteraz navštívený, *newnode* sa označí ako videný. Na konci, ak je $depth < k$, algoritmus rekurzívne zavolá BuildTree(newnode, depth+1, G, k). Časová zložitosť uvedená autormi pre algoritmy CombinationTree a BuildTree spolu je $O\left(d^{(k-1)}\right)$, kde d je maximálny stupeň G.

2.1.3 CombinationTree

Algorithm 3 CombinationTree

1: $root.vertex \leftarrow v$
2: $list[0] \leftarrow \{v\}$
3: BuildTree(root, 1, g, k)
4: **return** *root*

Algoritmus 3 je CombinationTree(v, k, G). Algoritmus prijíma ako vstup vrchol v, celé číslo k a graf G. CombinationTree na začiatku inicializuje základné premenné: *root* je koreň kombinačného stromu zodpovedajúci vrcholu v a *list* je zoznam veľkosti k. Potom sa spustí algoritmus BuildTree(root, 1, G, k), ktorý vytvorí kombinačný strom s koreňom *root*. Na konci CombinationTree vráti *root*.

2.1.4 CombinationsWithV

Algorithm 4 CombinationsWithV

1: $root \leftarrow CombinationTree(v, k, G)$
2: $allCombs \leftarrow CombinationsFromTree(root, k)$
3: **return** $Label(allCombs)$

Algoritmus 4 CombinationsWithV(v, k, G), ktorý ako vstup prijíma vrchol v, celé číslo k a graf G. Na začiatku algoritmu sa použije algoritmus CombinationTree(v, k, G) na inicializáciu premennej *tree*, ktorá uchováva kombinačný strom. Potom sa spustí algoritmus CombinationsFromTree(tree, k), ktorého výsledok sa uloží do premennej *allCombs*. V premennej *allCombs* je uložený zoznam uzlov kombinačného stromu - sú to podgrafy obsahujúce vrchol v. Na konci algoritmus vráti zoznamy vrcholov grafu, ktoré zodpovedajú uzlom kombinačného stromu.

2.1.5 ConSubg

Algorithm 5 ConSubg

```
1: allConnSubg ← ∅
2: for all v ∈ V(G) do
3:     allConnSubg ← allConnSubg ∪ CombinationsWithV(v, k, G)
4:     G.deleteVertex(v)
5: end for
6: return allConnSubg
```

Algoritmus 5, ktorý je zároveň hlavným algoritmom, je ConSubg(k, G). Na vstupe má celé číslo k a graf G. Na začiatku algoritmu sa inicializuje premenná $allConnSubg$, do ktorej sa uložia všetky nájdené podgrafy a ktorá je zároveň návratovou hodnotou funkcie. Potom nasleduje cyklus, v ktorom sa pre každý vrchol $v \in V(G)$ zavolá algoritmus CombinationsWithV(v, k, G) a výsledok sa uloží do $allConnSubg$. Na konci cyklu sa vrchol v odstráni z grafu G. Cyklus pokračuje, kým počet vrcholov v grafe nie je menší ako veľkosť podgrafov k.

2.1.6 Testovanie algoritmu

Autori vykonali 3 experimenty so svojím algoritmom, v ktorých ho porovnali s primitívnejšími algoritmami. Prvý experiment sa týkal grafov, kde $|V| = 100$ a stupne vrcholov boli 10 a 40. Pre grafy so stupňom vrcholov rovným 10 algoritmus fungoval dobre, pri veľkosti podgrafov od 3 do 5 bol čas hľadania blízky 1, pri 6 približne 20 sekúnd, pri 7 približne 400 sekúnd. Ostatné algoritmy boli pomalšie a pri veľkosti 7 nedosiahli vôbec žiadne výsledky. Keď je však stupeň vrcholov grafu 40, algoritmus funguje horšie ako ostatné pri veľkosti podgrafu 5 a pri 6 a viac nefunguje vôbec. V druhom experimente ponechali veľkosť podgrafov 4, stupne vrcholov grafu 10 a 40 a zväčšili veľkosť grafu z $|V| = 100$ na $|V| = 900$. Tu algoritmus funguje lepšie ako ostatné v oboch prípadoch, keď je stupeň vrcholov 10 a keď je 40. Algoritmy, s ktorými sa ConSubg porovnáva, si nevedia poradiť s grafmi, ktorých veľkosť presahuje 650. Aj ConSubg vykazuje dobrú rýchlosť vyhľadávania, v prípade stupňa 10 sa čas behu s rastúcou veľkosťou grafu výrazne nemení a zostáva okolo 1, v prípade stupňa 40 sa čas behu rovnomerne zvyšuje a dosiahnutie veľkosti 900 bude okolo 200 s. V treťom experimente autori zvyšovali stupeň vrcholov grafu. Tento experiment ukázal, že na grafe $|V| = 100$ sa od stupňa 25 algoritmus stáva pomalším ako ostatné. Na grafoch $|V| = 300$ a $|V| = 400$ je však algoritmus vo všetkých prípadoch oveľa rýchlejší ako ostatné. Tieto experimenty ukazujú, že tento algoritmus je vhodný na hľadanie podgrafov vo veľkých grafoch, pretože v tomto výrazne prekonáva algoritmy, s ktorými ho autori porovnávali.

Tabuľka 2.2: Experimenty na náhodných grafoch

Experiment	Veľkosť grafu	Stupeň vrcholov	Veľkosť podgrafu
1	100	10 40	3, ...,7 3, ...,6
2	100,150,...,900	10 40	4
3	100 300 400	{5,10, ...,40 }	4

2.2 Reverse Search

Algoritmus ReverseSearch(G, k), ktorý predstavili Alokshiya, Salem a Abed, vytvára stromovu cestu prechádzania grafu bez stromu v pamäti. Na tento účel používa štyri zoznamy. U je zoznam, do ktorého sa ukladá cesta prechádzania. C - zoznam susedov všetkých vrcholov v U. P - zoznam rodičov pre každý vrchol. D - zoznam, v ktorom je uložená vzdialenosť od počiatočného vrcholu[1].

Algoritmus je rozdelený na 3 funkcie.

Tabuľka 2.3: Funkcie algoritmu Reverse Search

Funkcia	Volá funkcie	Pseudokód
ReverseSearch	EnumerateCIS	Algoritmus 8
EnumerateCIS	EnumerateCIS isValidExtension	Algoritmus 7
isValidExtension	-	Algoritmus 6

2.2.1 isValidExtension

Algorithm 6 isValidExtension(U, D, v)

1: $anchor \leftarrow U.first$
2: **if** $v < anchor$ **then** return false
3: **end if**
4: **if** $D[v] > D[last]$ **then** return true
5: **end if**
6: return $D[v] = D[last]$ *and* $v > last$

Tretia funkcia isValidExtension(U, D, v) kontroluje, či zoznam U možno rozšíriť o vrchol v. Najprv inicializuje premenné *anchor* - počiatočný vrchol v U a *last* - posledný vrchol. Potom skontroluje, či je *anchor* lexigraficky väčší akov, ak áno, vráti $false$. Potom vráti *true*, ak je vzdialenosť medzi v a *anchor* väčšia ako vzdialenosť medzi *last* a *anchor*. Nakoniec funkcia vráti *true*, ak sa vzdialenosť medzi v a *anchor* rovná vzdialenosti medzi *anchor* a *last* a v je lexigraficky väčší ako *last*, inak vráti $false$. Zložitosť isValidExtension špecifikovaná autormi je $O(1)$. Potom, ak už nie je k dispozícii žiadne riešenie, zložitosť celého algoritmu je $O(|V|)$.

2.2.2 EnumerateCIS

Algorithm 7 EnumerateCIS(U, P, D, C)

1: **for all** $v \in C$ **do**
2: $P[v] \leftarrow U.last$
3: $D[v] \leftarrow D[U.last] + 1$
4: **if** $U.size = k$ **then** $subgraphs \leftarrow subgraphs \cup U$ continue
5: **end if**
6: **if** isValidExtension(U, D, v) **then**
7: $EnumerateCIS(U + [v], P, D, C \cup Neighbors(v))$
8: **end if**
9: **end for**

Druhá funkcia, ktorá je základom algoritmu, EnumerateCIS(U, P, D, C) počíta podgrafy. Na vstupe prijíma zoznamy U, P, D a C. Na začiatku funkcie sa spustí cyklus, ktorý prechádza vrcholy v z C. V cykle sa inicializuje vzdialenosť $D[v]$ a rodičovský vrchol $P[v]$. Potom nasleduje kontrola v tvare veľkosť $U = k$. Ak áno, vráti sa zoznam U, ktorý je podgrafom. Ak nie, algoritmus skontroluje, či je možné cestu U rozšíriť o vrchol v pomocou funkcie isValidExtension(U, D, v). Ak môže, pridá vrchol v do U a tiež aktualizuje množinu C pridaním všetkých susedov v a potom zavolá samu seba.

2.2.3 ReverseSearch

Prvou funkciou je ReverseSearch(G, k). Na vstupe prijíma graf G a celé číslo k. Potom ide cyklus, ktorý prechádza každý vrchol $v \in V(G)$. Na začiatku cyklu inicializuje zoznamy U, D a P. Pridá vrchol v do U, $D[v] = 0$, pretože vzdialenosť vrcholu od seba je 0, a $P[v] = -1$, pretože v "strome"je v koreňovým uzlom. Potom pre vrchol v spustí funkciu EnumerateCIS(U, P, D, Neighbors(v)), ktorá nájde podgrafy veľkosti k obsahujúce vrchol v. Na konci funkcia vráti zoznam podgrafov.

Algorithm 8 ReverseSearch(U, P, D, C)

1: $subGraphs \leftarrow \emptyset$
2: **for all** $v \in V(G)$ **do**
3: $U \leftarrow v$
4: $P[v] \leftarrow -1$
5: $D[v] \leftarrow 0$
6: $enumerateCIS(U, P, D, graph.getNeighbors(v))$
7: **end for**
8: $returnsubGraphs$

2.2.4 Testovanie algoritmu

Autori porovnali tento algoritmus s inými algoritmami na náhodne vygenerovaných grafoch s rôznou veľkosťou a hustotou. Napríklad na grafe s hustotou 0,6 bol čas behu algoritmu 1 s, kde $|V| = 25$. Pre $|V|$ od 26 do 29 bol čas behu do 10 sekúnd, od 29 do 32 do 100 sekúnd, od 32 do 34 vrátane bol čas behu medzi 100 a 1000 sekundami. V porovnaní s algoritmami ako TGE a BDDE bol rýchlejší a prešiel všetkými testami, napríklad BDDE si nevedel poradiť s grafmi väčšími ako 27. Autori testovali Reverse-Search aj na skutočných chemických grafoch, kde sa ukázal byť niekoľkokrát rýchlejší ako TGE.

2.3 Simple Forward

Algoritmus simpleForward, ktorý predstavili Komusiewicz a Sommer, je rozdelený na 2 algoritmy, prvý hlavný simpleForward(P, X) hľadá podgrafy s vrcholom v, druhý getSubgraphs(G, k) spustí simpleForward(P, X) pre vrcholy $v \in V(G)$[5].

2.3.1 simpleForward

Základný algoritmus simpleForward(P, X) prijíma ako vstup P - kde bude uložený podgraf a X - množinu susedov. Ak je veľkosť P = k, uloží podgraf P a vráti true. Potom sa inicializuje premenná hasIntLeaf s počiatočnou hodnotou false. Ďalej sa spustí cyklus, ktorý beží, kým X nie je prázdna množina. Na začiatku cyklu sa inicializuje premenná u, do ktorej sa uloží prvý vrchol z X. Tento vrchol z X sa vymaže. Potom sa inicializuje množina newX, do ktorej sa pridajú všetci susedia u okrem tých, ktorí už sú v P, a množina newP, ktorá je priesečníkom množín P a u. Na konci cyklu sa funkcia zavolá sama seba, ak vráti true, hasIntLeaf = true a cyklus pokračuje, ak nie, vráti hasIntLeaf. Na konci funkcia vráti hasIntLeaf.

Algorithm 9 simpleForward(P, X)

1: **if** $P.size = k$ **then** $subGraphs \leftarrow subGraphs \cup P$
2: **end if**
3: $hasIntLeaf \leftarrow false$
4: **while** $X \neq \emptyset$ **do**
5: $u \leftarrow X.begin$
6: $X \leftarrow X \backslash u$
7: $newX \leftarrow u.neighbors \backslash P.neighbors$
8: $newP \leftarrow P \cup u$
9: **if** $simpleForward(newP, newX)$ **then** $hasIntLeaf \leftarrow true$
10: **else** $returnhasIntLeaf$
11: **end if**
12: **end while**

2.3.2 getSubgraphs

Algorithm 10 getSubgraphs(G, k)

1: $subGraphs \leftarrow \emptyset$
2: **while** $G.size >= k$ **do**
3: $v \leftarrow V(G).first$
4: $P \leftarrow \{v\}$
5: $X \leftarrow v.neighbors$
6: $simpleForward(P, X)$
7: $G.deleteVertex(v)$
8: **end while**

Na vstupe je graf G a celé číslo k. Potom spustí cyklus, ktorý beží, kým je počet vrcholov grafu väčší alebo rovný k. Na začiatku cyklu sa vyberie prvý vrchol v grafu G. Potom inicializuje dve množiny P - kde bude uložený podgraf a X - množinu susedov. Vrchol v sa pridá do množiny P a množina X sa naplní susedmi tohto vrcholu. Potom sa spustí metóda simpleForward(P, X), ktorá hľadá podgrafy. Na konci cyklu sa vrchol v odstráni z grafu.

2.3.3 Testovanie algoritmu

Autori porovnali SimpleForward s algoritmami BDDE, Kavosh, RwD, RwP. Porovnania boli vykonané na grafoch rôznych veľkostí a pre rôzne k, kde autori uvádzajú, koľko percent riešení algoritmy našli v danom časovom intervale. V prvom porovnaní pre $k \in \{3, \ldots, 10\}$ SimpleForward našiel viac ako 10% riešení v prvých sekundách a viac ako 30 % za 500 sekúnd, čo je niekoľkokrát rýchlejšie ako RwD a RwP, BDDE a Kavosh

ukázali približne rovnaký výsledok. V druhom porovnaní $k \in \{n_c - 1, n_c - 2, n_c - 3\}$, kde n_c je poradie najväčšej súvislej komponenty grafu. Tu si SimpleForward viedol lepšie a našiel viac ako 20 % riešení v prvých 5 sekundách, približne rovnaký výsledok mal aj Kavosh. Za 500 sekúnd našiel SimpleForward viac ako 40 % riešení, zatiaľ čo Kavosh našiel len približne 30 %, ostatné algoritmy nenašli ani 10 %.

Kapitola 3

Návrh

3.1 Analýza požiadaviek

3.1.1 Popis konzolovej aplikácie

Konzolová aplikácia poskytuje používateľovi možnosť vyhľadávať podgrafy zadanej používateľom veľkosti v grafe zadanom používateľom. Podgrafy sa vyhľadávajú pomocou jedného z troch algoritmov, ktoré si tiež vyberá používateľ. Vstupné grafy a výstupné podgrafy sa uložia do súboru formatu txt. Interakcia s aplikáciou sa realizuje pomocou príkazov, ktoré používateľ zadáva do konzoly. Bude pozostávať zo 4 krokov.

Prvý krok umožní používateľovi zvoliť spôsob inicializácie grafu. Prvým spôsobom je zoznam hrán, druhým spôsobom je zoznam susedov. Ak používateľ zadá nesprávny príkaz, aplikácia ho upozorní a vyzve ho, aby si vybral znova.

V druhom kroku používateľ zadá názov súboru, v ktorom je uložený graf na vstup. Po zadaní aplikácia skontroluje, či takýto súbor existuje a či je graf v súbore správny. V prípade, že súbor neexistuje alebo je graf nesprávny, aplikácia vypíše chybu a ponúkne opätovné zadávanie.

V treťom kroku používateľ vyberie jeden z navrhovaných algoritmov, ktorý sa použije na vyhľadávanie podgrafov. Ak používateľ zadá nesprávny príkaz, aplikácia ho upozorní a vyzve na opätovný výber. V poslednom štvrtom kroku používateľ zadá veľkosť podgrafov, ktoré sa majú vyhľadať. Potom aplikácia spustí vyhľadávací algoritmus a zapíše podgrafy do súboru formatu txt.

Vytvorená aplikácia je ľahko pochopiteľná a použiteľná. Rozhranie je textové menu, v ktorom je každá činnosť a príkaz jasne popísaný, takže používateľ nemá pri práci s aplikáciou žiadne problémy.

3.1.2 Funkcionálne požiadavky

Funkcionálne požiadavky definujú, ako sa má aplikácia správať, aké vlastnosti a funkcie má poskytovať používateľovi.

Funkcionálne požiadavky:

1. Aplikácia umožní používateľovi vyhľadávať podgrafy v grafe.
2. Aplikácia umožní používateľovi definovať graf pomocou txt súboru.
3. Aplikácia umožní používateľovi nastaviť typ inicializácie grafu.
4. Aplikácia umožní používateľovi zadať algoritmus na vyhľadávanie podgrafov.
5. Aplikácia umožní používateľovi určiť veľkosť podgrafov, ktoré sa majú vyhľadať.
6. Aplikácia uloží nájdené podgrafy do txt súboru.

3.1.3 Nefunkcionálne požiadavky

Nefunkcionálne požiadavky opisujú vlastnosti konzolovej aplikácie.

Nefunkcionálne požiadavky:

1. Aplikácia bude konzolová.
2. Aplikácia bude implementovaná v jazyku C++.
3. Na zostavenie programu použijeme make a gcc.

Záver

todo

Literatúra

[1] Salem S. Alokshiya M. and F Abed. A linear delay algorithm for enumerating all connected induced subgraphs. *BMC Bioinformatics*, 20, 2019.

[2] B. Bollobas. *Modern Graph Theory*. Graduate Texts in Mathematics. Springer New York, 2013.

[3] R. Diestel. *Graph Theory*. Electronic library of mathematics. Springer, 2005.

[4] Choueiry B.Y. Karakashian S and Hartke S.G. An algorithm for generating all connected subgraphs with k vertices of a graph. Technical Report UNL-CSE-2013-0005, University of Nebraska-Lincoln, 2013.

[5] Christian Komusiewicz and Frank Sommer. Enumerating connected induced subgraphs: Improved delay and experimental comparison. *Discrete Applied Mathematics*, 303:262–282, 2021.

[6] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[7] Bjarne Stroustrup. An overview of the c++ programming language. *Handbook of object technology*, page 72, 1999.

[8] R.J. Trudeau. *Introduction to Graph Theory*. Dover Books on Mathematics. Dover Publications, 2013.

Príloha A: obsah elektronickej prílohy

V elektronickej prílohe priloženej k práci sa nachádza zdrojový kód programu a súbory s výsledkami experimentov. Zdrojový kód je zverejnený aj na stránke `http://mojadresa.com/`.

Ak uznáte za vhodné, môžete tu aj podrobnejšie rozpísať obsah tejto prílohy, prípadne poskytnúť návod na inštaláciu programu. Alternatívou je tieto informácie zahrnúť do samotnej prílohy, alebo ich uviesť na obidvoch miestach.

Príloha B: Používateľská príručka

V tejto prílohe uvádzame používateľskú príručku k nášmu softvéru. Tu by ďalej pokračoval text príručky. V práci nie je potrebné uvádzať používateľskú príručku, pokiaľ je používanie softvéru intuitívne alebo ak výsledkom práce nie je ucelený softvér určený pre používateľov.

V prílohách môžete uviesť aj ďalšie materiály, ktoré by mohli pôsobiť rušivo v hlavnom texte, ako napríklad rozsiahle tabuľky a podobne. Materiály, ktoré sú príliš dlhé na ich tlač, odovzdajte len v electronickej prílohe.