Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Cestová a stromová šírka kubických grafov

Bakalárska práca

2019
Filip Husár

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Cestová a stromová šírka kubických grafov

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	doc. RNDr. Robert Lukoťka, PhD.

Bratislava, 2019
Filip Husár

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

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

Názov: Cestová a stromová šírka kubických grafov
Pathwidth and treewidth of cubic graphs

Anotácia: Cestová a stromová šírka sú parametre, ktoré sa bežne vyskytujú v parametrizovanej zložitosti. Za predpokladu ohraničenej cestovej/stromovej šírky možno efektívne riešiť mnohé problémy, ktoré sú vo všeobecnosti NP-ťažké. Fomin a Hoie [F. V. Formin, K. Hoie: Pathwidth of cubic graphs and exact algorithms, Information Processing Letters 97 (2006), 191-196] skonštruovali triedu algoritmov, pomocou ktorej je možné pre každé epsilon nájsť cestovú dekompozíciu kubického grafu so šírkou nanajvýš (1/6+epsilon).|V(G)|. Tento výsledok je možné použiť na nájdenie pomerne rýchlych algoritmov na riešenie NP-ťažkých problémov na kubických grafoch. Nie je však známa žiadna trieda kubických grafov, ktorej cestová šírka by sa približovala 1/6.|V(G)|. Cieľom práce je preto hľadať nové prístupy a heuristiky na nájdenie cestového/stromového rozkladu kubického grafu s malou šírkou.

Vedúci: doc. RNDr. Robert Lukoťka, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 25.09.2018

Dátum schválenia: 06.11.2018 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Týmto by som chcel poďakovať svojmu školiteľovi doc. RNDr. Robertovi Lukoťkovi, PhD. za cenné rady poskytnuté pri riešení problémov tejto práce a jej písaní.

Abstrakt

Cestová a stromová dekompozícia je rozčlenenie vrcholov grafu do množín vriec pričom platí, že každá hrana sa nachádza v niektorom vreci a každý vrchol je v súvislom úseku vriec. Napokon, vrecia tvoria cestu, prípadne strom, podľa dekompozície ktorá je vykonaná. Budeme sa zaoberať algoritmom cestovej dekompozície kubických grafov, ktorý bol publikovaný v článku F.V.Fomina a K.Høieho. K jeho realizácii použijeme niekoľko algoritmov bisekcie kubického grafu z článku B.Moniena a R.Preisa. Vďaka dekompozíciám s malou šírkou rozkladu vieme riešiť mnoho $\mathcal{NP}$-ťažkých problémov na grafoch efektívne.

V tejto práci implementujeme spomínané algoritmy na výpočet približnej bisekcie a cestovej dekompozície, ktorej šírka rozkladu bude vo väčšine prípadov približne $(1/6)|V(G)|$. Implementácia bude z dôvodu jej rýchlosti oproti článkom odlišná v niektorých detailoch. Použijeme pritom knižnicu ba-graph v jazyku C++.

Kľúčové slová: grafy, cestová dekompozícia, cestová šírka, približná bisekcia grafu

Abstract

Path and tree decomposition is (non-disjunct) division of vertices into bags such that every edge is included in any bag and furthemore every vertex is included in continuous section of bags. Finally, bags form a path or tree, according to which decomposition was made. We will take a look on algorithm for path decomposition of 3-regular graphs in article published by F.V.Fomin and K.Høie. But at first, we must use several algorithms for bisection from article published by B.Monien and R.Preis. Decompositions with low decomposition width help us solve many $\mathcal{NP}$-hard problems on graphs effectively.

In this paper we implement these algorithms for compute rough bisection and path decomposition with decomposition width which obtains in most cases $(1/6)|V(G)|$. Implementation will differ from articles in a few details which will save non-trivial time. We will use ba-graph library in C++ programming language.

Keywords: graphs, path decomposition, pathwidth, rough bisection

Obsah

Úvod **1**

1 Pojmy a definície **3**
1.1 Základné definície teórie grafov 3
1.2 Stromová dekompozícia 4
1.3 Cestová dekompozícia 5
1.4 Stromová a cestová šírka 6
1.5 Rez grafu 8

2 Známe horné ohraničenia **11**
2.1 Šírka bisekcie 11
2.2 Cestová šírka 19

3 Implementácia optimálneho rezu **23**
3.1 Lema o červených a čiernych hranách 23
3.2 Lema o 1-pomocných množinách 29
3.3 Optimálny rez 31

4 Implementácia cestovej dekompozície **33**
4.1 Lema o cestovej dekompozícii 33
4.2 Veta o ohraničenej šírke rozkladu 36

Záver **39**

Úvod

Za posledné dekády rokov sa oblasť počítačových technológií posúva stále výraznejšími krokmi dopredu. Úlohy na spracovanie počítačmi sú čoraz komplikovanejšie, náročnejšie na výpočty a obvody už atakujú fyzikálne zákony určujúce ich teoretickú rýchlosť. Na to, aby mohol vývoj počítačov ísť ruka v ruke so stále narastajúcimi požiadavkami ľudstva, je potreba zameriavať svoju pozornosť na algoritmické problémy.

Teória grafov je oblasť matematiky ktorej sa vo významnej miere venuje aj informatika. Grafové štruktúry sú súčasťou veľkého počtu programovacích jazykov. S rôznymi problematikami z oblasti teórie grafov sa môžeme stretnúť aj pri architektúre hardvérových komponentov počítaču, ako napr. VLSI [3](Very-Large-Scale Integration), ktorý pri navrhovaní intergovaných obvodov kombinuje milióny tranzistorov tak, aby výkonnosť bola čo najvyššia. Podobne aj pri kompilovaní programov vo vysokoúrovňových jazykoch [4], kedy všetky medzivýpočty možno uchovávať buď v registroch alebo v hlavnej pamäti, rieši túto problematiku teória grafov. V obidvoch spomenutých problémoch zohráva dôležitú úlohu cestová alebo stormová dekompozícia.

Cestová dekompozícia grafu je sekvencia množín vrcholov daného grafu taká, že každý vrchol grafu sa vyskytuje v súvislej sekvencii množín a každá hrana grafu sa vyskytuje v niektorej z množín (je reprezentovaná dvomi koncovými vrcholmi) Cieľom cestovej dekompozície je, aby množiny neboli príliš veľké, nech algoritmy, v ktorých cestová dekompozícia zohráva dôležitú rolu, boli čo najefektívnejšie. Inak povedané, chceme aby najväčšia z množín dekompozície mala čo najmenej vrcholov. Tento atribút nazývame šírka rozkladu. O dekompozíciach ktoré sa v tomto smere už nedajú zlepšiť hovoríme, že dosiahli cestovú šírku [4].

Podobnou problematikou sa zaoberá aj stromová dekompozícia. Od cestovej dekompozície sa líši tým, že množiny vytvárajú stormovú štruktúru. V tejto práci sa budeme zaoberať implementáciou algoritmov približnej bisekcie a cestovej dekompozície, ktorej šírka rozkladu je pre väčšinu vstupov $(\frac{1}{6} + \epsilon)n$. Využijeme dva už známe články zaoberajúce sa horným ohraničením šírky bisekcie kubických grafov [1], na čo bude nadväzovať článok o hornom ohraničení cestovej dekompozície [2]. Dôkazy týchto článkov sa následne pokúsime vhodne implementovať do programu, ktorý vďaka dynamickému programovaniu nájde v pomerne dobrom čase pre ľubovoľný kubický graf cestovú dekompozíciu s dobrou šírkou rozkladu.

Kapitola 1

Pojmy a definície

V tejto kapitole si vymenujeme základné prostriedky, ktoré budeme potrebovať na prácu v oblasti cestovej a stromovej šírky. Cestová a stromová šírka je jednou z novších oblastí teórie grafov, ktorá zohráva veľmi dôležitú rolu v algoritmických problémoch na grafoch. Hovorí nám o tom, ako dobrú cestovú alebo stromovú dekompozíciu vieme na danom grafe spraviť.

1.1 Základné definície teórie grafov

Pod pojmom graf budeme v tejto práci rozumieť množinu vrcholov a multimnožinu hrán ktoré ich spájajú. Takýmito grafmi možno zjednodušene znázorniť rôzne situácie z reálneho života. Vrcholy môžu znázorňovať ľudí a hrany priateľstvá medzi nimi. Vrcholy takisto môžu znázorňovať rohy ulíc a hrany medzi nimi cesty.

> **Definícia 1:** [7] *Neorientovaný konečný multigraf* (ďalej len graf) G je usporiadaná trojica $G = (V, E, f)$, kde V je neprázdna konečná množina vrcholov, E je konečná množina hrán a incidenčná funkcia $f : E \implies \{\{x, y\} : x, y \in V\}$ priraďuje každej hrane neusporiadanú dvojprvkovú multimnožinu.

Ak je graf z kontextu daný, pre $|V| = n, |E| = m$ budeme vrcholy označovať ako $v_1, ..., v_n$ a hrany $e_1, ..., e_m$, pričom $e_i = \{v_j, v_k\}, 1 \leq i \leq m$ je hrana spájajúca vrcholy v_j a v_k pre nejaké $j, k \leq n$. V prípade, že hrán spájajúcich vrcholy v_j a v_k je viac, deterministicky vyberieme ľubovoľnú z nich.

Ako si môžeme v uvedenej definícii všimnúť, v grafe budeme povoľovať hrany začínajúce aj končiace v tom istom vrchole, nazývané aj slučky. Definíciou tiež povoľujeme aj viacnásobné hrany, teda pre ľubovoľné dva vrcholy môže existovať viac prvkov z množiny E takých, že ony sami dané vrcholy spájajú.

Cestová a stromová dekompozícia zoskupuje vrcholy grafu do množín, ktoré sú navzájom spojené hranami tak, že množiny tvoria líniu, resp. strom. Tieto množiny zvykneme nazývať aj *vrecia*.

Súvislosť grafu hovorí o tom, či je graf zložený z jedného alebo viac komponentov. Nech máme graf $G = (V, E)$. Množina vrcholov $U \subseteq V$ spolu s hranami, ktoré spájajú vrcholy z tejto množiny, je komponent práve vtedy pokiaľ existuje cesta medzi ľubovoľnými dvoma vrcholmi z tejto množiny a zároveň pridaním nového vrchola do tejto množiny predošlá podmienka prestáva platiť. Pokiaľ má graf práve jeden komponent, je súvislý. Pokiaľ má graf viac komponentov, je nesúvislý. V grafe G sa kružnica nachádza práve vtedy, keď existuje dvojica vrcholov $v_1, v_2 \in V(G)$, že medzi vrcholmi v_1, v_2 existujú aspoň dve hranovo disjunktné cesty.

1.2 Stromová dekompozícia

Vyslovením predošlých tvrdení môžeme definovať pojem stromu [7]. Graf G nazývame *strom* práve vtedy keď je súvislý a neobsahuje kružnicu. Graf, ktorý je strom, má niektoré veľmi zaujímavé a praktické vlastnosti [8], ktoré nám môžu pomôcť k lepšej efektivite algoritmov na grafoch. Jedna zo základných vlastností stromu je tá, že medzi každými dvomi vrcholmi existuje práve jedna cesta. Z nej vieme odvodiť ďalšie známe vlastnosti stromov ako napr. minimálnu spojitosť grafu či maximálnu acyklickosť grafu.

Na to, aby sme mohli používať pojmy ako stromová či cestová šírka, si potrebujeme najprv zadefinovať stromovú (resp. cestovú) dekompozíciu grafu. Označenia $V(G)$ a $E(G)$ v nasledujúcich definíciach odkazujú na vrcholy, resp. hrany grafu G.

Definícia 2: [8] Majme graf $G = (V, E)$, strom T a zobrazenie $\nu : V(T) \to 2^{|V|}$. *Stromová dekompozícia* je taká dvojica *(T, ν)*, ktorá spĺňa nasledujúce podmienky:

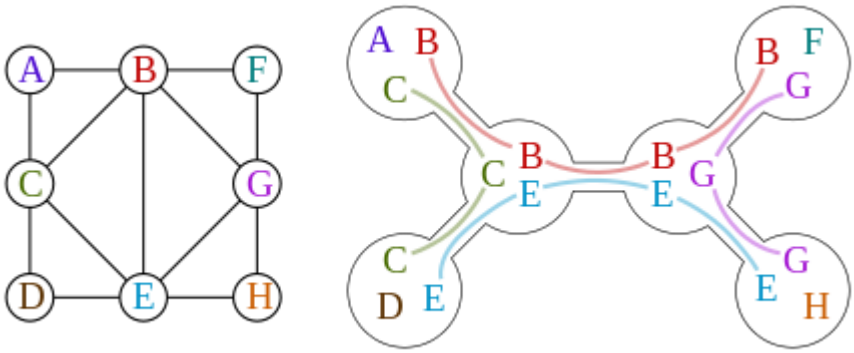
i. $V = \bigcup_{t \in T} V_t$

ii. pre každú hranu $e \in E$ existuje také $t \in T$, že oba koncové vrcholy hrany e ležia v V_t

iii. $V_{t_1} \bigcap V_{t_3} \subseteq V_{t_2}$ platí pre všetky $t_1, t_2, t_3 \in T$ pokiaľ t_2 leží na ceste spájajúcej t_1 s t_3

Podmienky i. a ii. hovoria o tom, že G je zjednotením podgrafov indukovaných vrcholmi množín $V_1, ..., V_n$.

Podmienka iii. hovorí o tom, že pre každý vrchol existuje na stromovej štruktúre cesta, že v každom vreci patriacom do tejto cesty sa nachádza daný vrchol.



Obr. 1.1: Príklad stromovej dekompozície

Na uvedenom obrázku 1.1 môžeme spozorovať, že každá oblasť grafu je v danej stromovej dekompozícii reprezentovaná jedným vrecom.

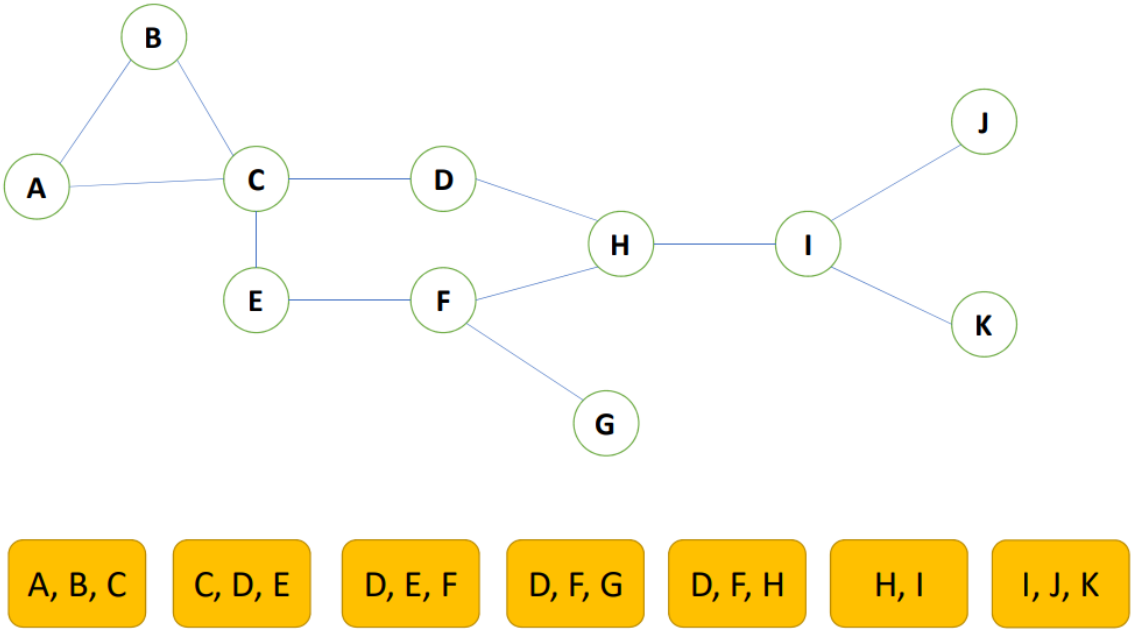
1.3 Cestová dekompozícia

Cestová dekompozícia sa od stromovej dekompozície líši iba v malom detaile. Vrecia sú usporiadané do postupnosti - sekvencie, teda každé vrece má práve dvoch susedov až na prvé a posledné vrece v postupnosti. Jednoducho povedané, v stromovej dekompozícii vrecia tvoria strom, v cestovej dekompozícii vrecia tvoria cestu.

> **Definícia 3:** [4] Majme graf $G = (V, E)$, cestu $P \in \{0, 1, ..., k\}$ pre nejaké $k \in \mathbb{N}$ a množinu množín vrcholov $\xi = \{V_p\}_{p \in P}$ takých, že $V_p \subseteq V$ pre $p = 0, 1, ..., |E(P)|$. *Cestová dekompozícia* je taká dvojica *(P, ξ)*, ktorá spĺňa nasledujúce podmienky:
>
> 1. $V = \bigcup_{p \in P} V_P$
> 2. pre každú hranu $e \in E$ existuje také $p \in P$, že oba koncové vrcholy hrany e ležia v V_p
> 3. pre všetky $i, j, k \in P$ platí $i < j < k \implies V_i \bigcap V_k \subseteq V_j$

Lineárnu dekompozíciu konečného grafu G nazývame *cestová dekompozícia*. Pokiaľ bude cestová dekompozícia z kontextu jasná, kvôli lepeju prehľadnosti zavedieme pre i-te vrece dekompozície značenie V_i.

V práci sa budeme zaoberať výlučne konečnými grafmi, preto je pre nás v tejto chvíli zaujímavá práve cestová dekompozícia.



Obr. 1.2: Príklad cestovej dekompozície

1.4 Stromová a cestová šírka

Nie každá dekompozícia môže byť pre nás dostatočne uspokojivá. Podstatným atribútom, či už stromovej alebo cestovej dekompozície, je šírka rozkladu. Tá je odvodená od počtu vrcholov, ktoré sa nachádzajú v jej najväčšom vreci.

Najprv si definujeme šírku rozkladu stromovej dekompozície.

Definícia 4: [4] Majme graf G. Nech (T, ν) je stromová dekompozícia G a nech (P, ξ) je cestová dekompozícia G. *Šírka rozkladu* je číslo

$$tdw(T, \nu) = max_{B \in V(T)} |\nu(B)| - 1$$
$$\text{resp. } pdw(P, \xi) = max_{B \in V(P)} |\xi(B)| - 1$$

Podľa definícií 2 a 3, T a P označuje strom, resp. cestu tvorenú vrecami dekompozície.

Definícia 5:[4] *Stromová šírka tw* grafu G je najmenšia šírka rozkladu spomedzi všetkých stromových dekompozícií grafu G.

$$tw(G) = min\{tdw(D) : D \text{ je stromová dekompozícia grafu } G\}$$

Ako si môžeme všimnúť, šírka rozkladu a tiež aj stromová šírka grafu na obrázku 1.1 je 2. Na základe predchádzajúcej definície o stromovej šírke poznamenávame, že cieľom stromovej dekompozície je dosiahnuť stromovú šírku, čo je ekvivalentné tvrdeniu, že stromová dekompozícia má za cieľ mať šírku rozkladu čo najmenšiu.

Cestová šírka je definovaná analogicky: množiny $V_t \in T$ nahradíme množinami $V_p \in P$ definovanými v cestovej dekompozícii.

> **Definícia 6:** [4] *Cestová šírka pw* grafu G je najmenšia šírka rozkladu spomedzi všetkých cestových dekompozícií grafu G.

$$pw(G) = min\{pdw(D) : D \text{ je cestová dekompozícia grafu } G\}$$

Pozn.:Šírka rozkladu a tiež aj cestová šírka grafu na obrázku 1.2 je 3

Z definície stromovej šírky vidíme, že existuje trieda grafov s nulovou šírkou rozkladu. Pre túto triedu grafov platí, že neobasujú hrany. Ak by obsahovali hrany, dekompozícia by mala kvôli podmienke ii. v niektorom vreci dva vrcholy, teda stromová šírka by bola aspoň 1.

Ďalej môžeme nahliadnuť že pre triedu grafov so šírkou rozkladu $tw = 1$ platí, že pôvodné grafy sú lesy [5]. Vyplýva to z toho, že každá množina obsahuje najviac dva vrcholy, teda pôvodný graf nemôže obsahovať cyklus, lebo ak by obsahoval, neexistovala by taká dekompozícia, ktorá by tento cyklus rozbila na množiny s najviac dvomi vrcholmi.

Existuje mnoho ďalších takýchto tried grafov so zaujímavými vlastnosťami, avšak v našej práci sa budeme zaoberať hlavne cestovou dekompozíciou s malou šírkou rozkladu, resp. blížiacou sa k cestovej šírke, preto sa ďalej pozrime na to, prečo je cestová šírka taká podstatná v efektívnych algoritmoch.

Využitie v efektívnych algoritmoch

Na začiatku 70-tych rokov bolo spozorované, že veľká trieda optimalizačných problémov by mohla byť v prípade ohraničenosti dimenzie grafov efektívne vyriešená dynamickým programovaním [6]. Dimenzia grafu je pojem blízky stromovej šírke a síce hovorí o tom, že daný graf vieme vnoriť pri klasickej reprezentácii do euklidovského priestoru s určitou dimenziou.

Bolo ukázané, že pomocou dekompozície s cestovou, resp. stromovou šírkou vieme na grafoch pomocou dynmického programovania vyriešiť efektívnejšie množstvo problémov, ktoré sú $\mathcal{NP}$-ťažké.

Príklady:

- Jedným z takýchto problémov na grafoch je farbenie grafu. Vďaka dynamickému programovaniu poznáme algoritmy, ktoré pre graf s n vrcholmi a so stromovou šírkou k nájdu farbenie grafu v čase $O(k^{k+O(1)} \cdot n)$.[9]

- Z menej známych problémov spomenieme hľadanie celkového počtu nezávislých množín v grafe či hľadanie celkového počtu párení v grafe. Wan a kol. [10] vytvoril dva algoritmy, ktoré na základe stromovej šírky veľkosti k nájdu v n-vrcholovom grafe celkový počet nezávislých množín v čase $O(2^k \cdot kn^3)$ a počet párení v grafe v čase $O(2^{2k} \cdot kn^3)$.

1.5 Rez grafu

V kapitole 2 si uvedieme vetu o hornom ohraničení cestovej šírky spolu s dôkazom. Na jej dokázanie využijeme dôkaz vety o hornom ohraničení bisekcie 3-regulárnych grafov.

Na to aby sme vedeli čo to je bisekcia grafu, najprv si definujeme *rez grafu*

Definícia 7: [8] *Rez* $\pi = (V_0, V_1)$ grafu G je rozdelenie vrcholov grafu $V(G)$ do dvoch disjunktných množín V_0 a V_1.

Rez, ktorý rozdelí vrcholy grafu $V(G)$ na množiny V_0, V_1 ktorých veľkosť sa líši nanajvýš o jeden vrchol, sa nazýva *bisekcia*.

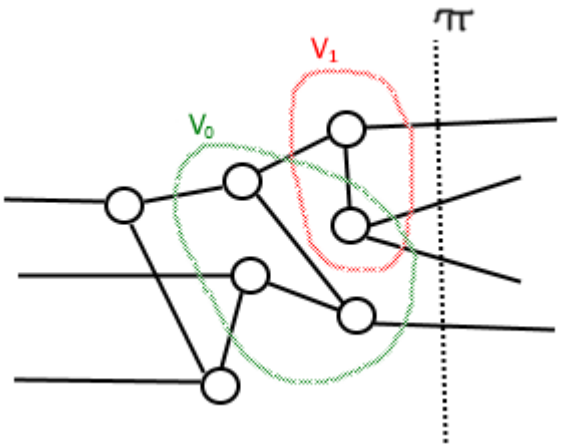
Definícia 8: [8] *Veľkosť rezu* $\pi = (V_0, V_1)$ grafu G s neohodnotenými hranami je počet hrán $e = \{v_j, v_k\}$ takých, ktoré majú jeden koncový vrchol v množine V_0 a druhý koncový vrchol v množine V_1.

Nasledovné dve definície využijeme v dôkaze lemy o 1-pomocných množinách.

Definícia 9: [1] Nech π je rez grafu $G = (V, E)$. Pre $S \subset V_p(\pi)$, pre nejaké $p \in \{0, 1\}$ označme

$$H(S) = |\{\{v, w\} \in E; v \in S, w \in V \setminus V_p(\pi)\}| - |\{\{v, w\} \in E; v \in S, w \in V_p(\pi) - S\}|$$

veľkosť zmenšenia (ak je $H(S) < 0$ tak zväčšenia) šírky rezu po presunutí množiny vrcholov S do druhej partície. Množinu S nazývame vzhľadom na rez π $H(S)$-pomocná.



Obr. 1.3: Množiny V_0 a V_1, ktoré sú vzhľadom na rez π 1-pomocné

Príklady: (na obrázku 1.3)

- Množina V_0 je 2-pomocná. Tri hrany patria do rezu π, jedna hrana vychádzajúca z množiny V_0 nepatrí do rezu π.
- Množina V_1 je (-2)-pomocná. Takže po jej presunutí do druhej partície sa veľkosť rezu zväčší o dve hrany.
- Vrcholy mimo množín V_0 a V_1 sú samé o sebe (-3)-pomocné

Kapitola 2

Známe horné ohraničenia

V tejto kapitole si spravíme prehľad doteraz známych výsledkov v oblasti cestovej a stromovej šírky. Ukážeme si aký vzťah má cestová a stromová šírka k bisekcii grafu a ako vieme bisekciu grafu premostiť k zaujímavému výsledku na cestovej dekompozícii.

Nakoľko sú pre nás v tejto práci zaujímavé iba konštrukčné dôkazy, budeme z článkov vynechávať dôkazy o existencii jednotlivých entít v grafe. Väčšinu dôležitých konštrukčných krokov treba v tejto kapitole spomenúť, nakoľko ich budeme viac rozoberať v ďaĺších implementačných kapitolách 3 a 4.

2.1 Šírka bisekcie

Poznáme širokú škálu problémov na grafoch, ktoré sa zaoberajú partíciami grafov. Úlohou je rozdeliť vrcholy grafu rovnomerne do určitého počtu partícií tak, aby počet hrán spájajúcich vrcholy z rôznych partícií bol čo najmenší. Počet takýchto hrán zvykneme označovať ako *veľkosť rezu*.

Špeciálnym prípadom partície grafu je *bisekcia*, kedy vrcholy grafu máme rovnomerne rozdelené práve do dvoch množín. Veľkosť rezu bisekcie zvykneme označovať ako *šírka bisekcie*. Začiatkom tohto tisícročia prišiel B. Monien s R. Preisom k výsledkom [1] ktoré ukázali, že so zvyšujúcim sa počtom vrcholov kubických grafov sa horné ohraničenie šírky bisekcie týchto grafov blíži k $(\frac{1}{6})|V(G)|$ (zhora)

Vzťah bisekcie grafu a cestovej šírky

O pár rokov neskôr na základe predchádzajúcich výsledkov o bisekcii kubických grafov F.V.Fomin s K.Hoiem skonštruovali triedu algoritmov [2], ktoré pre ľubovoľné $\epsilon > 0$ priradia $n_\epsilon \in \mathbb{N}$ a nájdu cestovú šírku grafu veľkú nanajvýš $(\frac{1}{6} + \epsilon)|V|$ pre všetky kubické grafy s počtom vrcholov $n > n_\epsilon$

Vzťahy medzi šírkou bisekcie a cestovej šírky však nie sú natoľko pevné. Ako vstupný parameter pre algoritmus pre cestovú dekompozíciu nepotrebujeme bisekciu,

stačí nám dostatočne malý rez grafu, ktorý rozdeľuje vrcholy do množín ktoré sa veľkosťou nelíšia až príliš. Takýto rez budeme nazývať približná bisekcia. V kapitole 3 implementácia si približnú bisekciu definujeme presnejšie. Zatiaľ ju budeme v tejto kapitole používať ako abstraktný pojem.

Vo zvyšku kapitoly si predstavíme dve lemy a vetu o veľkosti bisekcie na kubických grafoch spolu aj s ich konštrukčnými dôkazmi.

Lema o červených a čiernych hranách

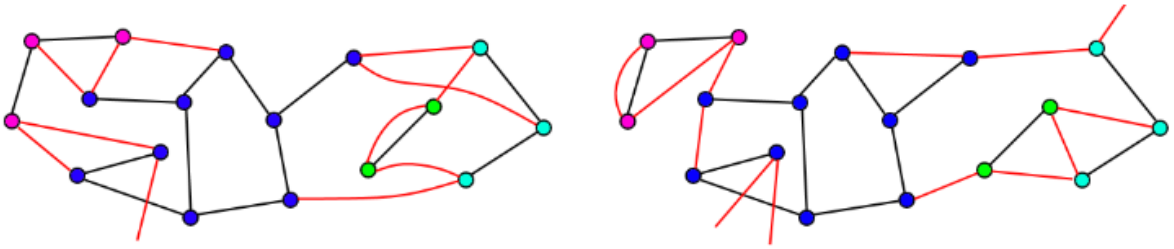
Lema o červených a čiernych hranách sa so svojím konštrukčným dôkazom používa na nájdenie aspoň 1-pomocnej množiny spĺňajúcej lemu 2.1. Napriek tomu že jej použitie je v rámci dôkazu lemy 2.1 len jeden malý krok, zo všetkých dôkazov v tejto práci je tento najkomplikovanejší. Uvedieme si ho až na niektoré menej podstatné detaily v plnom znení.

> **Lema 1:** [1] Nech $G = (V, E), E = B \uplus R$ je kubický graf s čiernymi hranami B a červenými hranami R. Nech je každý vrchol susedný s aspoň jednou čiernou hranou. Nech $|R| > (\frac{1}{2} + \epsilon)|V|$ pre $\epsilon > 0$. Potom existuje množina $S \subset V$ veľkosti $O(\frac{1}{\epsilon})$ taká, že počet červených hrán medzi vrcholmi množiny S je väčší ako počet čiernych hrán medzi vrcholmi množiny S a vrcholmi množiny $V \setminus S$

Nech $S \subseteq V(G)$. Potom množinou $R_{int}(S)$ budeme označovať všetky červené hrany medzi vrcholmi S. Množinou $B_{ext}(S \subseteq V(G))$ budeme označovať všetky čierne hrany medzi medzi ľubovoľným vrcholom z množiny S a vrcholom z množiny $V(G) \setminus S$. Pokiaľ je graf G alebo množina S z kontextu jasná, budeme ich vynechávať.

Dôkaz: [1]

Uvažujme graf $G_{black} = (V, B)$ pozostávajúci z množiny vrcholov a čiernych hrán pôvodného grafu G. Nech F je rodina komponentov grafu G_{black}. Komponent $I \in F$ považujeme za *malý*, pokiaľ je počet jeho vrcholov nanajvýš $\frac{1+2\epsilon}{\epsilon}$, inak ho považujeme za *veľký*. Počet červených hrán medzi množinou S a malými čiernymi komponentmi budeme označovať $\alpha(S)$. Označme $s(S)$ ako veľkosť zjednotenia množiny S s týmito komponentmi. Pokiaľ $s(I) \leq \frac{(1+2\epsilon)\cdot(\alpha(I)+1)}{\epsilon}$, čierny komponent I považujeme za *tenký*, inak za *hrubý*.



Obr. 2.1: Príklad tenkého a príklad hrubého čierneho komponentu pre $\epsilon = 0,2$

Na obrázku 2.1 vľavo môžeme vidieť čierny komponent (tvorený modrými vrcholmi), ktorý je s malými komponentami spojený ôsmimi červenými hranami a veľkosť jeho zjednotenia s týmito komponentami je 15. Modrý komponent pri ohodnotení $\epsilon = 0,2$ považujeme za tenký.

Vpravo na obrázku môžeme vidieť čierny komponent (tvorený modrými vrcholmi), ktorý je s malými komponentami spojený tromi červenými hranami a veľkosť jeho zjednotenia s týmito komponentami je 17. Modrý komponent pri ohodnotení $\epsilon = 0,2$ považujeme za hrubý.

Pokiaľ existuje malý komponent $I \in F$ taký, že v grafe G existuje červená hrana medzi vrcholmi tohto komponentu, daný komponent spĺňa lemu, keďže je dostatočne malý a má aspoň jednu internú (medzi vrcholmi komponentu) červenú hranu. Kvôli tomu že je to čierny komponent, neexistuje čierna hrana spájajúca I s $V(G) \setminus I$, teda červených interných hrán je viac ako čiernych externých.

Pokiaľ existujú dva malé komponenty $I, J \in F$ spojené červenou hranou, zjednotenie týchto komponentov spĺňa lemu, keďže je dostatočne malé a je tam minimálne jedna interná červená hrana a žiadna externá.

Pokiaľ sme doposiaľ zostali neúspešní, z F budeme vyhadzovať nasledovné komponenty:

1. Vyhoďme všetky malé komponenty $I \in F$ ktoré obsahujú čierne hrany tvoriace cyklus

2. Pokým existuje veľký a hrubý komponent $I \in F$, vyhoď I z F aj so všetkými malými komponentami, ktoré sú s ním spojené červenou hranou

pozn: Počas kroku 2 sa môžu tenké komponenty zmeniť na hrubé a naopak.

Po vykonaných úpravách všetky zvyšné komponenty v F sú

1. veľké a tenké, alebo

2. malé, a navyše čierne hrany tvoria strom

Pokiaľ sme doteraz nenašli množinu vrcholov spĺňajúcu lemu, v rodine komponentov F sa s istotou nachádza veľký a tenký komponent (dôkaz o jeho existencii vynecháme). Nech $I_{vt} \in F$ je veľký a tenký komponent. Nech $G_{I_{vt}}$ je graf indukovaný čiernymi hranami komponentu I_{vt}. Daný graf teraz v dvoch krokoch zredukujeme.

Z grafu $G_{I_{vt}}$ postupne zmažeme všetky vrcholy stupňa 1 až žiaden taký nezostane. Po ich zmazaní nahradíme v grafe $G_{I_{vt}}$ všetky súvislé úseky vrcholov stupňa 2 jednou hranou (spájajúcou vrcholy stupňa 3). Každá pridaná hrana e medzi vrcholy stupňa 3 nahrádza vymazané vrcholy, ktoré tvoria strom. Označme túto množinu ako $T(e)$. Analogicky s tenkosťou, resp. hrubosťou množiny vrcholov budeme novopridané hrany označovať ako netučné, resp. tučné.(opäť kontrolujeme pomer medzi počtom červených hrán spájajúcich $T(e)$ s malými komponentmi a veľkosťou zjednotenia $T(e)$ s týmito množinami). V grafe $G_{I_{vt}}$ existuje vrchol v, že počet červených hrán, ktoré spájajú $T(e)$ s malým komponentom, pričom e je netučná hrana susediaca s vrcholom v, je aspoň 4 (dôkaz o existencii vrchola vynechávame). Toto číslo označme ako $n(v)$.

- Pokiaľ pre vrchol v platí $n(v) = 4$, tak zjednotenie vrcholu v s vymazanými stromami na ktoré odkazujú susedné netučné hrany vrchola v, spolu s malými komponentami spojenými štyrmi červenými hranami, spĺňa lemu, keďže tam sú aspoň štyri interné červené hrany (spájajúce malé komponenty) a nanajvýš tri externé čierne hrany (netučné hrany môžu byť tri, pričom so zvyškom grafu $G_{I_{vt}}$ sú spojené čiernou hranou)

- Pokiaľ $n(v) \geq 5$ a vrchol v je sused s dvomi netučnými hranami, kde každá z nich odkazuje na vymazaný strom ktorý spájajú dve červené hrany s malými komponentmi, potom zjednotenie vrcholu v s vymazanými stromami daných dvoch netučných hrán spolu s malými komponentmi spojenými štyrmi červenými hranami, spĺňa lemu, keďže tam sú aspoň štyri interné červené hrany (spájajúce malé komponenty) a tri externé čierne hrany (netučné hrany sú dve, pričom so zvyškom grafu $G_{I_{vt}}$ sú spojené čiernou hranou, susedná hrana vrchola v ktorá nebola spomenutá je treťou externou čiernou hranou)

Pokiaľ sme stále nenašli množinu spĺňajúcu lemu, niektorá zo susedných hrán e vrchola v je netučná a $T(e)$ spájajú s malými komponentmi aspoň tri červené hrany. Ukážeme, ako vieme nájsť v $T(e)$ množinu vrcholov takú, že spolu s malými komponentami spojenými cez červenú hranu táto množina spĺňa lemu.

Vrcholy spojené s $T(e)$ čiernou hranou označme v_1, v_2. Majme množinu $K = T(e) \cup \{v_1, v_2\}$. K je strom. Zoberme jeden z vrcholov v_1, v_2 ako koreň a hrany nech sú orientované smerom od koreňa k listom. Označme v strome vrcholy ktoré spĺňajú nasledujúce podmienky ako kandidujúce:

- Vrchol v vo svojom podstrome neobsahuje žiaden z vrcholov v_1, v_2
- Podstrom vrchola je spojený s malými komponentmi dvomi červenými hranami
- Podstrom vrchola neobsahuje žiaden iný vrchol ktorý spĺňa predošlé dve podmienky, t.j. vrchol je čo najďalej od koreňa

Pokiaľ pre niektorý z vrcholov čo spĺňa vyššie uvedené podmienky nie je zjednotenie jeho podstromu s malými komponentami spojenými dvomi červenými hranami príliš veľké, toto zjednotenie spĺňa lemu (dve interné červené hrany spájajúce malé komponenty, jedna čierna externá hrana od vrcholu v smerom k rodičovi v strome K)

Pokiaľ je však pre všetky vrcholy spĺňajúce podmienky spomínané zjednotenie príliš veľké, vytvorme nový graf $\overline{G}$ pozostávajúci z vrcholov v_1, v_2, kandidujúcich vrcholov a všetkými cestami (vrcholmi stupňa 2) medzi nimi. Spomínané cesty v grafe $\overline{G}$ korešpondujú s tučnými/netučnými hranami. Pre cestu P označme $T(P) = \bigcup_{v \in P} v \cup T(w)$, kde $w \in K \setminus \overline{G}$ je sused vrchola v. Ak taký nie je, $T(w)$ je prázdna množina. $T(P)$ teda obsahuje všetky vrcholy na ceste P a všetky podstromy týchto vrcholov nepatriace grafu $\overline{G}$. Pre vrcholy v stupňa 3 v grafe $\overline{G}$ označíme $n(v)$ počet červených hrán spájajúcich $T(P)$ s malými komponentami červenou hranou pre všetky P ktoré susedia s vrcholom v. V grafe existuje vrchol v pre ktorý $n(v) \geq 4$. Dôkaz o jeho existencii vynecháme. Cesty P susediace s vrcholom v korešpondujú s netučnými hranami.

Budeme postupovať analogicky ako v predošlom prípade. Ak $n(v) = 4$, vieme nájsť množinu vrcholov spĺňajúcu lemu. Ak $n(v) \geq 5$ a pre dve z ciest P_1, P_2 platí, že $T(P_{1,2})$ spájajú s malými komponentami dve červené hrany, vieme nájsť množinu vrcholov spĺňajúcich lemu. Pokiaľ sme stále nenašli takú množinu, pre niektorú cestu P susediacu s vrcholom v platí, že $T(P)$ spájajú s malými komponentami aspoň tri červené hrany. Vrchol $v \in P$ označme ako červený, pokiaľ má suseda $w \in K \setminus \overline{G}$ takého, že $T(w)$ je spojené s malým komponentom cez červenú hranu.

- Ak je $T(P)$ spojené s malými komponentami najviac cez štyri červené hrany, zjednotenie cesty P a týchto malých komponentov spojených tromi červenými hranami spĺňa lemu. Sú tam aspoň tri interné červené hrany spájajúce malé komponenty, dve externé čierne hrany spájajúce okraje cesty P so zvyškom grafu $\overline{G}$.
- Pokiaľ je $T(P)$ spojené s malými komponentami aspoň cez päť červených hrán, nájdeme takú trojicu $\{v_1, v_2, v_3\}$ červených vrcholov na ceste P, že zjednotenie $T(w_{1,2,3})$ ciest medzi vrcholmi v_1, v_2, v_3 a malých komponentov aspoň cez tri červené hrany spĺňa lemu. Sú tam aspoň tri interné červené hrany spájajúce malé komponenty, dve externé čierne hrany spájajúce okraje cesty P so zvyškom $\overline{G}$.

Dôkaz, že zjednotenia sú dostatočne malé, vynecháme. Týmto sme ukázali ako má vyzerať algoritmus na nájdenie množiny vrcholov spĺňajúcich lemu o červených a čiernych hranách.

Lema o 1-pomocných množinách

V tejto časti si ukážeme návod, ako nájsť množinu vrcholov takú, ktorej presunutím do druhej partície zmenčím veľkosť rezu.

Majme rez $\pi = (V_0, V_1)$ grafu G. Vrcholy množiny $V(G)$ budeme klasifikovať podľa toho ako ďaleko sa nachádzajú od rezu, t.j. akú má dĺžku najkratšia cesta z daného vrcholu (napr. patriaceho do V_0) do ľubovoľného vrcholu v druhej množine rezu (V_1). V množine C sa budú nachádzať všetky vrcholy vo vzdialenosti 1 od rezu, t.j. vrcholy ktoré majú suseda v množine V_1 (V_0). V množine D sa budú nachádzať vrcholy vo vzdialenosti 2 od rezu, v množine E sa budú nachádzať vrcholy so vzdialenosťou aspoň 3 od rezu. Navyše, v množine $D_i, i \in \{1, 2, 3\}$ sa budú nachádzať všetky vrcholy z množiny D, ktoré susedia s i vrcholmi z množiny C. Teda $V_0 = C \uplus D_3 \uplus D_2 \uplus D_1 \uplus E$. Číslami $c(X), d_3(X), d_2(X), d_1(X), e(X)$ budeme označovať počet vrcholov daného typu v množine $X \subseteq V$.

> **Lema 2:** [1] Nech $\pi = (V_0, V_1)$ je rez grafu $G = (V, E), V = V_0 \uplus V_1$. Pokiaľ je šírka rezu väčšia ako $(\frac{1}{3} + 2\epsilon)|V_0|, \epsilon > 0$, vieme nájsť aspoň 1-pomocnú množinu $S \in V_0$ veľkosti $O(\frac{1}{\epsilon})$.

Teraz si v skratke predstavíme hlavné myšlienky dôkazu.

Dôkaz: [1]

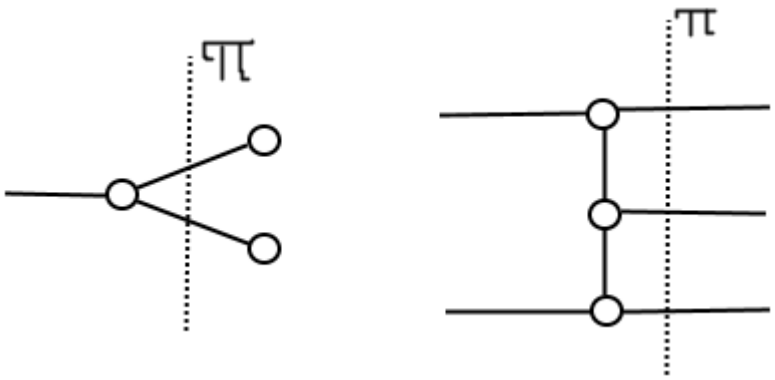
V množine V_0 sa môžu nachádzať situácie, ktoré priamo vedú k malej 1-pomocnej množine ktorá spĺňa lemu.

1. Pokiaľ má niektorý z C-vrcholov aspoň dvoch susedov v V_1, tento vrchol sám o sebe tvorí aspoň 1-pomocnú množinu.
2. Množina troch spojených C-vrcholov tvorí 1-pomocnú množinu.
3. Nech vrchol v má C-suseda, ktorý je susedom ďalšieho C alebo D_3 vrchola. Pokiaľ zvyšní dvaja susedia vrchola v sú z množiny $C \cup D_2 \cup D_3$, zjednotenie všetkých spomenutých vrcholov a ich C-susedov tvorí aspoň 1-pomocnú množinu veľkosti nanajvýš 11 vrcholov.

Ak sme doteraz nenašli žiadnu štruktúru spĺňajúcu lemu, na grafe G_{V_0} spravíme transformácie, po ktorých v grafe nezostanú žiadne D_3 vrcholy a taktiež ani hrany medzi C-vrcholmi.

Transformácia 1:

Pôvodná situácia: $\alpha, \beta \in C, \gamma \in D_1 \cup D_2, \delta \in D_1 \cup E, \{\alpha, \beta\}, \{\beta, \gamma\}, \{\gamma, \delta\} \in E(G_{V_0})$. V transformovanom grafe $\overline{G_{V_0}}$ nahradíme hrany $\{\alpha, \beta\}, \{\gamma, \delta\}$ hranami $\{\alpha, \gamma\}, \{\beta, \delta\}$

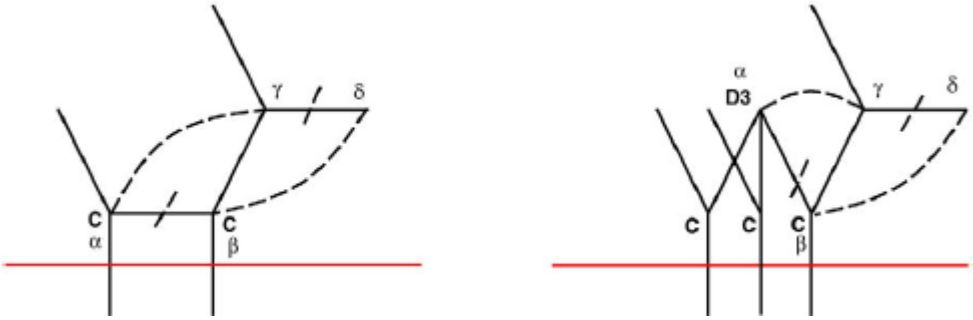


Obr. 2.2: Jednoduché príklady 1-pomocných množín typu 1. a 2.

Transformácia 2:

Pôvodná situácia: $\alpha \in D_3, \beta \in C, \gamma \in D_1 \cup D_2, \delta \in D_1 \cup E, \{\alpha, \beta\}, \{\beta, \gamma\}, \{\gamma, \delta\} \in E(G_{V_0})$. V transformovanom grafe $\overline{G_{V_0}}$ nahradíme hrany $\{\alpha, \beta\}, \{\gamma, \delta\}$ hranami $\{\alpha, \gamma\}, \{\beta, \delta\}$

Po vykonaní spomenutých transformácií použijeme na grafe $\overline{G_{V_0}}$ lemu o červených a čiernych hranách 2.1 na nájdenie 1-pomocnej množiny.



Obr. 2.3: Transformácia 1 a transformácia 2

Vytvorím nový graf K, ktorý bude pozostávať z $D \cup E$ vrcholov. Všetky hrany medzi vrcholmi $D \cup E$ v grafe $\overline{G_{V_0}}$ budú v grafe K čierne. Červené hrany budú medzi každými dvomi vrcholmi, ktoré sú v $\overline{G_{V_0}}$ spojené so spoločným C-vrcholom.

K spĺňa predpoklady pre lemu 1 pre $\overline{\epsilon} = 3\epsilon$. Použitím lemy 1 nájdeme 1-pomocnú množinu $\overline{S}$, ktorá spolu so susednými C-vrcholmi spĺňa lemu pre graf $\overline{G_{V_0}}$.

Zostáva nám ukázať ako nájdenú 1-pomocnú množinu na transformovanom grafe $\overline{G_{V_0}}$ upravíme tak, aby to bola 1-pomocná množina v pôvodnom grafe G_{V_0}.

Všetky tranformácie ktoré sme vykonali na grafe G_{V_0} teraz budeme inverzne simulovať (teda aj vykonávať v opačnom poradí - najprv inverzné transformácie 2, potom inverzné transformácie 1). V skutočnosti nebudeme pridávať ani odoberať hrany, ale iba upravovať 1-pomocnú množinu S v závislosti od konfigurácie vrcholov v grafe $\overline{S}$.

Pre vykonávanú inverznú tranformáciu si skontrolujeme či sa $\alpha, \beta, \gamma, \delta$ nachádzajú v 1-pomocnej množine.

Inverzná transformácia 2:

1. $\alpha, \gamma \in \overline{S}, \beta, \delta \notin \overline{S} \implies S = \overline{S} \cup \{\beta\}$
2. $\alpha, \gamma \notin \overline{S}, \beta, \delta \in \overline{S} \implies S = \overline{S} \cup \{\alpha, \gamma\} \cup \{v \in C; \{v, \alpha\} \in E\}$

Inverzná transformácia 1:

1. $\alpha, \gamma \in \overline{S}, \beta, \delta \notin \overline{S} \implies S = \overline{S} \cup \{\beta\}$
2. $\alpha, \gamma \notin \overline{S}, \beta, \delta \in \overline{S} \implies S = \overline{S} \cup \{\alpha, \gamma\}$

Obidve inverzné transformácie zväčšia 1-pomocnú množinu S prinajhoršom o konštantu, teda S spĺňa lemu.

Lema o 1-pomocnej množine nám hovorí o tom, že pokiaľ je šírka rezu približnej bisekcie väčšia ako $(\frac{1}{3} + 2\epsilon)|V_0|, \epsilon > 0$ kde $|V_0| \geq |V_1|$, vieme vo V_0 nájsť dostatočne malú 1-pomocnú množinu. Inak povedané, vždy tento algoritmus aplikujeme na väčšiu z množín V_0, V_1, tým pádom sa snažíme dostať čo najbližšie k bisekcii. Niekedy by sa však mohlo stať, že po presunutí vrcholov medzi množinami sa ich pomer ešte zväčší, avšak vyriešenie tohto problému si bližšie ukážeme v kapitole 3 implementácia optimálneho rezu.

2.2 Cestová šírka

V dôkaze o dekompozícii kubického grafu budeme potrebovať vetu o dekompozícii stromu, ktorú si teraz uvedieme bez dôkazu.

Veta a dekompozícii stromu:[11] Pre ľubovoľný strom T s počtom vrcholov $n \geq 3$ platí, že cestová šírka stromu T je nanajvýš $\log_3 n$

Lema o cestovej dekompozícii

V tejto časti si ukážeme možný návod, ako nájsť cestovú dekompozíciu s dostatočne malou šírkou rozkladu.

Lema 3: [2] Nech G je graf s vrcholmi stupňa nanajvýš 3. Pre každú množinu $X \subseteq V(G)$ existuje cestová dekompozícia $(X_1, X_2, ..., X_r)$ grafu G so šírkou nanajvýš $max\{|X|, \lfloor n/3 \rfloor\} + (2/3)\log_3 n + 1$, pričom $X = X_r$

Dôkaz: [2]

Lemu budeme dokazovať matematickou indukciou. Na grafe s jedným vrcholom je tvrdenie lemy triviálne. Predpokladajme, že tvrdenie platí pre všetky grafy s počtom vrcholov menším ako n pre nejaké $n > 1$

Majme graf G s n vrcholmi a množinu $X \subseteq V(G)$. Môžu nastať viaceré situácie:

Prípad 1: Nájdeme vrchol $v \in X$ pre ktorý $N(v) \setminus X = \emptyset$, t.j. v nemá žiadneho suseda mimo množiny X.

Podľa indukčného predpokladu existuje cestová dekompozícia $(X_1, X_2, ..., X_r)$ grafu $G \setminus \{v\}$ so šírkou rozkladu nanajvýš $max\{|X| - 1, \lfloor (n-1)/3 \rfloor\} + (2/3)\log_3 n - 1 + 1$, pričom $X = X_r$. Pridaním vrchola v do množiny X_r nadobúda cestová dekompozícia šírku rozkladu nanajvýš $max\{|X|, \lfloor n/3 \rfloor\} + (2/3)\log_3 n + 1$

Prípad 2: Nájdeme vrchol $v \in X$ pre ktorý $|N(v) \setminus X| = 1$, t.j. vrchol v má mimo množiny X práve jedného suseda. Označme tento susedný vrchol u.

Podľa indukčného predpokladu pre $G \setminus \{v\}$ a $X \setminus \{v\} \cup \{u\}$ existuje cestová dekompozícia $P' = (X_1, X_2, ..., X_r)$ grafu $G \setminus \{v\}$ so šírkou rozkladu nanajvýš

$max\{|X|, \lfloor (n-1)/3 \rfloor\} + (2/3)\log_3 n - 1 + 1$, pričom $X \setminus \{v\} \cup \{u\} = X_r$. Vytvoríme novú cestovú dekompozíciu P z P' pridaním množín $X_{r+1} = X \cup \{u\}, X_{r+2} = X$, takže novovzniknutá cestová dekompozícia bude mať tvar $P = (X_1, X_2, ..., X_r, X_{r+1}, X_{r+2})$ a jej šírka rozkladu bude nanajvýš $max\{|X|, \lfloor n/3 \rfloor\} + (2/3)\log_3 n + 1$.

Prípad 3: Pre každý vrchol $v \in X$ platí $|N(v) \setminus X| \geq 2$, t.j. vrchol má aspoň dvoch susedov mimo množiny X. V takejto situácii rozlišujeme dva podprípady, podľa toho či je množina X veľká alebo malá. (detailnejší postup je ukázaný v pôvodnom článku)

Podprípad 3.A: pokiaľ je množina X vzhľadom na graf G veľká, vieme povedať, že v $G \setminus X$ sa nachádza komponent T, ktorý je strom. Podľa vety o dekompozícii stromov 2.2 existuje cestová dekompozícia $P^1 = (X_1, X_2, ..., X_r)$ grafu T so šírkou rozkladu nanajvýš $(2/3)\log_3 n$. Podľa indukčného predpokladu existuje cestová dekompozícia

$P^2 = (Y_1, Y_2, ..., Y_t = X)$ grafu $G \setminus V(T)$ so šírkou rozkladu najviac $|X| + 1$. Kombináciou týchto dvoch cestových dekompozícií dostaneme dekompozíciu

$P = (Y_1, Y_2, ..., Y_t = X, X_1 \cup X, X_2 \cup X, ..., X_r \cup X, X)$ so šírkou rozkladu nanajvýš $max\{|X|, \lfloor n/3 \rfloor\} + (2/3)\log_3 n + 1$

Podprípad 3.B: pokiaľ je množina X vzhľadom na graf G malá, pridáme do množiny X taký počet vrcholov z $V(G) \setminus X$ aby bola množina vzhľadom na graf G veľká. Po pridaní vrcholov do množiny X sa budeme nachádzať v niektorom z vyššie uvedených prípadov.

Matematickou indukciou sme dokázali tvrdenie lemy.

Veta o ohraničenej šírke rozkladu

Grafom $G[X]$ budeme rozumieť graf generovaný vrcholmi množiny $X \subseteq V(G)$. Okrem vrcholov X doň patria aj hrany medzi týmito vrcholmi a nič ďalšie.

Veta 4: Pre ľubovoľné $\epsilon > 0$ vieme nájsť celé kladné číslo n_ϵ, že pre každý 3-regulárny graf $G, |V(G)| > n_\epsilon$, existuje cestová dekompozícia so šírkou rozkladu nanajvýš $(\frac{1}{6} + \epsilon)|V(G)|$

Dôkaz:

Majme graf G. Vďaka lemám o červených hranách 2.1 a 1-pomocných množinách 2.1 vieme nájsť približnú bisekciu so šírkou rezu nanajvýš $(\frac{1}{6} + \epsilon)|V(G)|$. Približná bisekcia nám bude k požadovanému výsledku cestovej dekompozície stačiť. Predpokladajme, že pokiaľ je šírka rezu π grafu G väčšia ako $(\frac{1}{3} + 2 \cdot \epsilon)|V_0|$ pre niektorú z rezových množín V_0, vieme tento rez zmenšiť presunutím niektorých vrcholov z tejto množiny. Predpokladajme, že menšia z množín má $\frac{|V(G)|}{2}$ vrcholov. Po presunutí $O(\frac{1}{\epsilon})$ bude mať teda ľubovoľná z množín nanajvýš $(\frac{1}{3} + 2 \cdot \epsilon)(|\frac{|V(G)|}{2}| + O(\frac{1}{\epsilon})) = (\frac{1}{6} + \epsilon)(|V(G)| + O(\frac{1}{\epsilon}))$, čo pri vhodne zvolenom epsilon môže znamenať, že približná bisekcia nám zaručuje, aby sa nasledovný algoritmus cestovej dekompozície svojou šírkou blížil k $\frac{1}{6}|V(G)|$.

Označme partície V_0, V_1. C-vrcholy vo V_0 označme $\Delta(V_0)$, C-vrcholy v V_1 označme $\Delta(V_1)$. Pomocou lemy o dekompozícii 2.2 vieme vytvoriť cestové dekompozície $P_0 = (X_1, X_2, ..., X_r = \Delta(V_0), P_1 = (Y_1, Y_2, ..., Y_t = \Delta(V_1)$podgrafov $G[V_0], G[V_1]$. Tieto dekompozície majú veľkosť nanajvýš $max\{(\frac{1}{6} + \epsilon)n, \lfloor n/6 \rfloor + 1\} + (2/3)\log_3 n + 1$

Teraz vytvoríme dekompozíciu $P_x = (Z_1, Z_2, ..., Z_s)$. Množina Z_1 bude obsahovať vrcholy $\Delta(V_0)$, Množina Z_s bude obsahovať vrcholy $\Delta(V_1)$. Pre Z_i, kde i je nepárne, vyberieme vrchol $v \in Z_i \setminus Z_s$ pričom ďalšie dve množiny konfigurujeme následovne: $Z_{i+1} = Z_i \cup N(v) \cap Z_s, Z_{i+2} = Z_{i+1} \setminus \{v\}$

Keďže v každom kroku vyhodím jeden vrchol z V_0 a nahradím ho jeho susedmi V_1, po konečnom počte krokov sa dostaneme k úprave množiny Z_s, ktorá obsahuje vrcholy z $\Delta(V_1)$.

Konečná dekompozícia grafu G sa skladá zo spomenutých troch čiastkových dekompozícií a má tvar $P = (X_1, X_2, ..., X_r, Z_1, Z_2, ..., Z_s, Y_t, Y_{t-1}, ..., Y_1)$

Kapitola 3

Implementácia optimálneho rezu

V tejto kapitole si ukážeme, ako sme pri implementácii približnej bisekcie riešili technické problémy rôzneho druhu. Implementáciou dôkazov budeme prechádzať v chronologickom poradí ako v kapitole 2, preto sa budeme odvolávať na časti dôkazov iba keď to bude vyslovene potrebné.

V práci budeme používať knižnicu *ba_graph*, aj kvôli ktorej sme používali programovací jazyk *C++*. Na začiatok si teda predstavíme ktoré prostriedky tejto knižnice budeme v práci využívať. Ako každá iná grafová knižnica, tak aj táto obsahuje základné triedy ako *Vertex* alebo *Edge*, teda poskytuje rozhranie pre vrcholy a hrany. My však budeme vo väčšine implementácie pristupovať k vrcholom cez triedu *Number*, ktorá má v sebe uložené číslo vrchola. To nám, ako uvidíme neskôr, pre problematiku cestovej šírky postačuje. Vrcholy grafu budeme mať pre rýchlosť prístupu uložené v dátovej štruktúre *set*.

K hranám grafu budeme pristupovať cez štruktúru *multiset*, v ktorej budem mať uložené dvojice vrcholov, teda $\{Number, Number\}$. Dôvodom ukladania hrán do *multiset* je prípustnosť viacnásobných hrán, čo *multiset* povoľuje. Každý vrchol v grafe má priradené autentické číslo, preto nám na viacerých miestach v programe bude postačovať prístup k vrcholu cez triedu *Number*.

3.1 Lema o červených a čiernych hranách

Cieľom lemy 1 je nájsť takú množinu vrcholov, v rámci ktorej je viac červených hrán medzi vrcholmi ako čiernych hrán vychádzajúcich z množiny vrcholov mimo množinu.

Základnou funkciou pre konštrukciu tejto lemy je funkcia *redBlackEdges*

```
set<Number> redBlackEdges (Graph &g, multiset<pair<Number, Number>>
&blackEdges, float epsilon)
```

V prípade že epsilon nebude zadané ako parameter, použije sa preň hodnota $\frac{|R|}{|V|} - \frac{1}{2.01}$ vychádzajúca z nerovnice $\epsilon < \frac{|R|}{|V|} - \frac{1}{2}$.

Aby sme mali všetky potrebné informácie o grafe, potrebujeme okrem počtu vrcholov a zozname hrán (reprezentovaných v premennej g) vedieť aj ktoré hrany sú červené a ktoré sú čierne. Preto budeme vyžadovať multimnožinu čiernych hrán *blackEdges*. Hrany v grafe g nepatriace multimnožine *blackEdges* sú červené. Nakoniec už len potrebujeme hodnotu ϵ, čo získame ako paramter z lemy 2, ktorú si ukážeme neskôr.

Keďže však už vrámci funkcie $oneHelpfulSet$ rozhodujeme ktoré hrany budú červené a čierne, popri rozhodovaní rovno vytvárame graf indukovaný čiernymi hranami *blackGraph* - ten budeme tiež využívať vrámci tejto lemy. Aj keď je teda parameter *blackGraph* (spolu aj s Factory) redundantný, vrámci interných výpočtov budeme používať nasledovnú funkciu:

```
set<Number> redBlackEdges (Graph &g, multiset<pair<Number, Number>>
&blackEdges, float epsilon, Graph &blackGraph, Factory &f)
```

V prípade, že sa nachádza v grafe vrchol ktorý má červenú slučku, tento vrchol spĺňa lemu, keďže slučka je považovaná za dve hrany vychádzajúce z vrchola, teda vrchol má dve interné červené hrany a jednu externú čiernu (tretia hrana musí byť čierna kvôli podmienke v leme).

Na to aby sme zistili či je množina vrcholov I tenká alebo hrubá potrebujeme vedieť koľko červených hrán je takých, že spája danú množinu s malými komponentami. Označením SC budeme chápať zjednotenie všetkých malých čiernych komponentov.

Ukážeme si tri rôzne prístupy:

- Prvá možnosť je prechádzať všetkými červenými hranami a zisťovať či jeden z jej koncových vrcholov je v I a druhý patrí do niektorého malého komponentu. Na to aby sme zistili, či je druhý v niektorej malej množine, musíme iteratívne prejsť všetky malé komponenty. Preto časová náročnosť tohto prístupu je $O(|R|\mu|SC|)$, čo pre nás nie je uspokojivé.

- Ďalší spôsob je namiesto červených hrán prejsť všetkými vrcholmi I, no opäť sa dostávame do podobného problému, kedy zistiť, či susedný vrchol cez červenú hranu patrí do malého komponentu je časovo náročné. Tento prípad je ale predsa o niečo lepší a trvá $O(|I|\mu|SC|)$.

- Keď vymeníme poradie iterácií a najprv prejdeme všetky vrcholy SC a overíme, či nie ich sused patrí do množiny I, celkový postup nás bude stáť $O(|SC|)$ času

Veľkosť zjednotenia množiny I so všetkými malými komponentami spojenými červenou hranou budeme riešiť analogicky. Čo však môžeme oproti predchádzajúcemu

postupu vylepšiť je, že vždy keď pre daný malý komponent nájdeme červenú hranu spájajúcu komponent s I, zväčším zjednotenie o veľkosť tohto malého komponentu a preruším for cyklus pre vrcholy daného malého komponentu.

Keď už máme základné funkcie prichystané, uložíme si komponenty indukované čiernymi hranami do F vo formáte $set(set < Number >)$. To spravíme jednoduchým prehľadávaním s využitím zaznačenia už navštívených vrcholov. Z daného vrcholu sa môžem pohybovať iba po čiernych hranách a do nenavštívených susedov, v ktorých budem postup rekurzívne opakovať. Po návrate z rekurzí funkcia uloží navštívené vrcholy. Keďže je graf konečný, táto funkcia tiež skončí.

Podľa dôkazu 2.1 potrebujeme overiť, či sa nenachádza v F taký malý komponent I, ktorý by bol pozitívny, resp. by bol spojený s iným malým komponentom červenou hranou.

Prvú možnosť vieme skontrolovať triviálne, pomocou už vytvorenej funkcie *isPositive*, pre druhý prípad použijeme funkciu *smallSetsViaRedEdge*, ktorá nájde všetky malé komponenty spojené s I - pre každý malý čierny komponent zistí, či neexistuje červená hrana z jeho vrcholov do nejakého vrchola z I.

```
set<Number> smallSetsViaRedEdge(set<Number> vertices, Graph &g,
multiset<pair<Number, Number>> blackEdges, float epsilon,
set<set<Number>> &connectedComponents, bool addOnlyOneSet)
```

Funkcia *smallSetsViaRedEdge* má parametricky dané, či chceme nájsť všetky malé komponenty alebo stačí jeden taký. V kroku spomenutom vyššie chceme práve jeden komponent (*addOnlyOneSet* je nastavené na *true*), preto po jeho nájdení funkcia skončí. Táto funkcia bude v svojom plnom rozsahu využitá ešte neskôr.

V konštrukčnom dôkaze tejto lemy vrcholy aj hrany mažeme. Aby sme neprišli o dôležité údaje ktoré budeme neskôr potrebovať, nebudeme vrcholy ani hrany v skutočnosti mazať, ale iba ich budeme označovať za vymazané. Na druhú stranu si to však vyžaduje, aby sme vždy pri prechádzaní susedov vrcholov kontrolovali, či nie je daný sused vymazaný.

Pokiaľ sa nenachádza v F žiaden komponent spĺňajúci lemu, funkciami *step1, step2* upravíme množinu komponentov F podľa dôkazu.

- V kroku 1 využijeme fakt, že v grafe s vrcholmi stupňa ≤ 3 sa kružnica nachádza práve vtedy, keď je v ňom počet 1-stupňových vrcholov nanajvýš taký ako počet 3-stupňových. [8]

```
(1) for auto comp: smallBlackComponents:
(2)     b1, b3 = 0
(3)     for auto n: comp
(4)         numOfBlackEdges = 0
(5)         for auto edge: n
(6)             if edge is black internal
                then: numOfBlackEdges++;
(7)         if numOfBlackEdges==1
            then b1++;
(8)         if numOfBlackEdges==3
            then b3++;
(9)     if b1<=b3
        then erase component
```

- Počas druhého kroku sa, narozdiel od prvého, môžu dodatočne vyskytnúť niektoré konfigurácie ktoré treba vymazať. Druhý krok ukončím až v momente, ako som spravil jednu celú iteráciu elementami z F bez toho aby som niektorý z nich vyhodil.

```
(1) while (true)
(2)     erased = false
(3)     for auto comp: large thick black components
(4)         erase all connected small componen, finally erase comp
(5)         erased = true;
(6)     if !erased
        then break
```

Redukciu niektorého veľkého tenkého komponentu riešim v nasledujúcich dvoch krokoch:

1. *reduceStep1*: Pre každý vrchol komponentu na redukovanie sa pozriem koľko má nevymazaných susedov spojených čiernou hranou. Pokiaľ má iba jedného takého suseda, označím vrchol za vymazaný a zavolám mazaciu funkciu *removeLine* na susedný vrchol. Mazacia funkcia sa zastaví keď niektorý vrchol bude mať dvoch nevymazaných čiernych susedov. Keďže sa removeLine volá funkciou reduceStep1, ktorá prejde postupne všetkými vrcholmi, po redukcii nezostane žiaden vrchol stupňa 1.

2. *reduceStep2*: Vymazanie všetkých ciest tvorených vrcholmi stupňa 2 nie je problém. Treba si však zaznačovať vrcholy ktoré vymažeme a potom ich vymazať naraz. Postupné mazanie by zapríčinilo zníženie stupňa vrcholov a mohlo by sa stať, že by sme takto vymazali všetky vrcholy v komponente.

V reduceStep2 nepridávam hrany medzi vrcholy stupňa 3, spravím tak až vo funkcii *edgeRef*, ktorá ako parameter okrem iného vyžaduje zoznam vymazaných vrcholov. Pole vymazaných vrcholov znegujeme, teda vrcholy komponentu po redukcii budú označené ako vymazané. Potom zavoláme funkciu connectedComponents, ktorou získame zredukované stromy. Pre každý strom zistíme ktorý vrchol mimo stromu je spojený čiernou hranou. Takéto vrcholy sú dva a medzi nimi bude hrana odkazujúca na tento vymazaný strom.

Funkciu *isFat* vytvoríme analogicky ako funkciu *isThin*.

Vo funkcii *n(v)*, ktorá pre netučné hrany e incidentné s vrcholom v spočíta počet červených hrán spájajúcich $T(e)$ s malým komponentom, skombinujeme už vytvorené funkcie *isFat* a *alfa*, ktorá počíta červené hrany spájajúce množinu s malým komponentom.

V prípade ak pre niektorý vrchol v platí $n(v) = 4$, funkciou *smallSetsViaRedEdge* dostaneme požadovanú množinu spĺňajúcu lemu. Všimnime si, že v tomto prípade nechceme iba jeden komponent ako na začiatku.

Podobne postupujeme aj v prípade $n \geq 5$

V situácii, kedy vo vymazanom strome označujeme vrcholy ako kandidujúce, hlavne využívame funkcie *designateNodes* a *undesignateAbove*. Prvá z funkcií overí prvé dve podmienky pre kandidujúci vrchol.

Tretia podmienka hovorí o tom, že kandidujúci vrchol nemôže mať v podstrome žiaden iný kandidujúci vrchol. Inak povedané, medzi každou dvojicou *koreň-kandidujúci vrchol* nemôže byť iný kandidujúci vrchol, teda vždy po označení vrchola ako kandidujúceho funkcia *undesignateAbove* všetky vrcholy medzi koreňom a kandidujúcim vrcholom označí ako nekandidujúce.

Konštrukciu grafu $\overline{G}$ pozostávajúcu zo všetkých ciest medzi kandidujúcimi vrcholmi a koreňom vytvoríme pomerne jednoducho: pre každý kandidujúci vrchol pridám do množiny všetkých jeho predkov až po koreň. Pokiaľ som na ceste narazil na vrchol ktorý už je pridaný, pridávanie vrcholov do grafu ukončím.

Keďže vrcholy grafu máme uložené v dátovej štruktúre *set*, o duplicitu vrcholov sa nemusíme obávať.

Podobne získame všetky cesty medzi kandidujúcimi vrcholmi. Funkcie *getAllPaths* a *getPath* prechádzajú od každého kandidujúceho vrchola smerom hore.

- Ak daný vrchol nie je koreň, over či jeho rodič má iba jedno dieťa. Ak áno, pridaj ho do cesty, ak nie, je to vrchol stupňa 3, keďže má dve deti a ako nekoreňový vrchol aj rodiča. V takom prípade ho do cesty nepridaj, cestu pridaj do množiny ciest a getPath spusti na 3-stupňovom rodičovi.

- Ak daný vrchol je koreň, doterajšiu cestu nahraďme novou, ktorú získame z funkcie *pathOfChildren*, ktorá vytvorí cestu z koreňa a vrcholov s jedným dieťaťom.

Nasledujúci úsek s funkciou $n(v)$ riešime analogicky ako v predošlom prípade.

V prípade, že $N(v) \geq 5$ a zároveň pre niektorú cestu P z vrchola v platí, že $3 \leq \alpha(T(P)) \leq 4$, požadovanú množinu nájdeme ľahko. Pokiaľ $\alpha(T(P)) \geq 5$, zistíme, ktoré vrcholy na ceste P sú červené, t.j. pre ktoré $v \in P$ platí $\alpha(T(v)) = 1$.

Aby sme mohli nájsť množinu spĺňajúcu lemu, potrebujeme si červené vrcholy dať do poradia *sequence* v akom nasledujú na ceste P. To spravíme tak, že z vrchola v zistíme, ktorý sused patrí do cesty P. Tento sused je na jednom jej kraji. Postupne budeme prechádzať na druhý koniec cesty a vždy pri prechádzaní červeným vrcholom tento vrchol pridáme do *sequence*. Aby sme si prvky uchovávali v lineárnom poradí, *sequence* musí byť vector.

Podobne v poradí budeme mať uložené aj cesty medzi červenými vrcholmi. Pre jednoduchosť algoritmu treba spomenúť, že prvý element v zozname týchto ciest nie je cesta medzi dvomi červenými vrcholmi.

Stačí nám už len prejsť všetky trojice červených vrcholov v_i, v_{i+1}, v_{i+2} a dvoch ciest P_{i+1}, P_{i+2} medzi nimi. Každú takúto päticu skontrolujeme, či je zjednotenie $T(V_i) \cup T(v_{i+1}) \cup T(v_{i+2}) \cup T(P_{i+1}) \cup T(P_{i+2})$ ako doteraz najmenšie nájdené. Ak áno, označme túto päticu ako doteraz najmenšiu nájdenú.

Po $|P|-2$ krokoch nájdeme päticu, ktorá spolu s malými komponentami cez červené hrany spĺňa lemu.

3.2 Lema o 1-pomocných množinách

V tejto leme budú veľkú úlohu zohrávať transformácie - pridávanie a odstraňovanie hrán.

Podobne ako v predchádzajúcej leme, ani v tejto nebudeme pridávať či odstraňovať hrany na vstupnom grafe. Potrebujeme si ho zachovať pre ďalšie použitie. Transformované hrany si budeme uchovávať nasledovne.

V *neighbors* si budeme ukladať ktorý vrchol má akých susedov v pôvodnom grafe. V *transormedNeighbors* budeme mať uložených susedov daného vrchola po vykonaných transformáciách. Pre prípady viacnásobných hrán či slučiek máme susedov vrchola uložených v *multiset*.

Základnou funkciou tejto lemy je *getHelpfulSet*, kde $v0$ obsahuje vrcholy jednej z dvoch množín ktoré vznikli rezom. Vrcholy grafu &*graph* ktoré nepatria do $v0$ patria do $v1$.

```
set<Number> getHelpfulSet(Graph &graph, set<Number> v0, float epsilon)
```

Rovnomenná funkcia getHelpfulSet so štvrtým parametrom `f (Factory)` je využívná pri hľadaní rezu grafu.

Na začiatku si prejdeme vrcholy z $v0$ a zistíme, ktoré hrany majú opačný vrchol mimo tejto množiny. Tieto hrany tvoria rez. Následne si overíme prítomnosť jednoduchých konfigurácií vrcholov, ktoré by priamo viedli ku k-pomocnej množine; $k > 0$. Oproti dôkazu z článku sme medzi jednoduché konfigurácie pridali aj dvojicu spojených C-vrcholov so spoločným D_2-vrcholom, čo tvorí 1-pomocný množinu. Pokiaľ sme žiadne knfigurácie v grafe nenašli, vykonáme transformácie podľa dôkazu.

Vykonané transformácie si budeme ukladať do *vector*, keďže po nájdení 1-pomocnej množiny na transformovanom grafe budeme vykonávať inverzné transformácie v reverznom poradí, takže ich potrebujeme mať uložené v správnom poradí.

Pri transformácii 1 hľadáme štyri vrcholy priamo. Začneme od ľubovoľného C-vrcholu a zistím či nemá C-suseda. Ak áno, nájdem ďalšie vrcholy podľa dôkazu.

Je dôležité si zapamätať, kde sa nachádzal tretí a štvrtý vrchol. Aby sme vedeli, či γ treba vymazať z D_1 alebo z D_2 a pridať do D_2, resp. D_3. Podobne aj pri vrchole δ.

Aj pri transformácii 2 hľadáme požadované vrcholy priamo cez susedov. Tu si však nepotrebujeme pamätať do ktorej množiny patrí γ, keďže pridaná ani odobratá hrana z vrcholu γ ho nespájala s C-vrcholom, teda vrchol γ zostane v množine D_i v ktorej sa nachádza.

Vytvoríme graf s červeno-čiernymi hranami podľa dôkazu a nájdeme 1-pomocnú množinu *derivedSetS*. V reverznom poradí vykonáme tranformácie typu 2, následne aj transformácie typu 1. Transformácie vykonávame pomocou iterátorov rbegin() a rend() na množine executedTransformations2 a executedTransformations1. V opačnom poradí

prechádzame vykonanými transformáciami a sledujeme či spĺňajú podmienky v leme o 1-pomocných množinách 2.1 na pridanie vrcholov do derivedSetS.

Upravená 1-pomocná množina spolu aj so susednými C-vrcholmi spĺňa lemu. Pokiaľ je však množina príliš veľká, že po jej presune do V_1 bude $|V_0| < \frac{|V(G)|}{3}$, vrátime prázdnu množinu, čo autorita volajúca túto funkciu bude chápať ako informáciu aby ukončila hľadanie optimálneho rezu.

```
if (v0.size()-vertices.size() >= graph.order() / 3.0) {
    return vertices;
} else {
    vertices.clear();
    return vertices;
}
```

Zvolené čísla zaručujú, že žiadna z množín V_0, V_1 nebude nikdy väčšia od druhej 2-násobne alebo viac. Príklad: Pri 20-vrcholovom grafe pripúšťame, že po presunutí pomocnej množiny bude mať jedna z množín V_0, V_1 nanajvýš 13 vrcholov.

3.3 Optimálny rez

Ako vstupné parametre funkcie *cutSizeBisection* dostaneme graf a hodnotu ϵ. Vytvoríme si množiny vrcholov V_0, V_1, ktoré sa budú líšiť veľkosťou najviac o jeden vrchol.

Aby následné hľadanie optimálneho rezu netrvalo dlho, nebudeme do V_0, resp. V_1 pridávať vrcholy náhodne. Na začiatok pridáme jeden vrchol do V_1 a všetky ostatné do V_0. V premennej *cut* budem mať zoznam rezových hrán, teda z množiny V_0 do V_1.

Pokým $|V_1| < |V_0|$, v konštantnom čase zvolíme jednu rezovú hranu, jej koncový vrchol $v \in V_0$ vyhodíme z V_0 a pridáme do V_1. Po aktualizovaní partícií prejdeme hrany presunutého vrchola. Hrany ktoré sú vo V_1 vyhodíme z *cut*, hrany ktoré sú vo V_0 pridáme do *cut*. Treba si však uvedomiť, že to môžeme spraviť až dodatočne, keď sa skončí *for auto* cyklus, keďže pridaním alebo odstránením elementu z *cut* by iterátor stratil svoju pozíciu a program by padol. Nakoniec aktualizujeme *cut*.

Môžeme si všimnúť, že po takomto inicializovaní partícií neexistuje vo V_1 vrchol, ktorý by bol vzhľadom na rez 3-pomocný, keďže každý vrchol má aspoň jedného suseda vnútri svojej partície, teda jeho presunutím rez zmenšíme nanajvýš o 1.

Vo while cykle budeme hľadať pomocnú množinu vo väčšej z množín V_0, V_1. Tento while cyklus ukončuje situácia, kedy pomocnú množinu nevieme nájsť, resp. kedy je pomocná množina taká veľká, že po jej presunutí by jedna z množín bola oproti druhej príliš malá. Túto skutočnosť kontroluje už ukázaná funkcia *getHelpfulSet*.

Po ukončení while cyklu vrátime ako výsledok optimálneho rezu zoznam vrcholov v oboch množinách V_0, V_1.

Ukážka kódu: while cyklus a pomocné množiny

```
while (true) {
    if (v1.size() > v0.size()) {
        set<Number> setToMove = getHelpfulSet(graph, v1, epsilon, f);
        if (setToMove.empty()) break;
        for (auto &n: setToMove) {
            v1.erase(n);
            v0.insert(n);
        }
    } else {
        set<Number> setToMove = getHelpfulSet(graph, v0, epsilon, f);
        if (setToMove.empty()) break;
        for (auto &n: setToMove) {
            v0.erase(n);
            v1.insert(n);
}   }   }
```

Ukážka kódu: prvotné naplnenie množín V_0, V_1.

```
set<Number> v0, v1, allVerticesOfGraph;
for (int i = 1; i<graph.order(); i++)
    v0.insert(graph[i].n());
v1.insert(graph[0].n());
multiset<pair<Number, Number>> cut;
for (auto &n: v1)
    for (auto &n2: graph[n])
        if (v1.find(n2.n2())==v1.end())
            cut.insert(pair(min(n, n2.n2()), max(n, n2.n2())));
while (v1.size()<v0.size()){
    multiset<pair<Number, Number>> toErase, toAdd;
    multiset<pair<Number, Number>> toErase, toAdd;
    pair<Number, Number> e = cut.begin().operator*();
    if (v1.find(e.first)!=v1.end()){
        v0.erase(e.second);
        v1.insert(e.second);
        for (auto &n: graph[e.second])
            if (v1.find(n.n2())!=v1.end()){
                toErase.insert(pair(n.n2(), e.second));
            } else {
                toAdd.insert(pair(n.n2(), e.second));
            }
    } else {
        v0.erase(e.first);
        v1.insert(e.first);
        for (auto &n: graph[e.first])
            if (v1.find(n.n2())!=v1.end()){
                toErase.insert(pair(n.n2(), e.first));
            } else {
                toAdd.insert(pair(n.n2(), e.first));
            }
    }
    for (auto &a: toAdd)
       cut.insert(pair(min(a.first, a.second), max(a.first, a.second)));
    for (auto &r: toErase)
        cut.erase(pair(min(r.first, r.second), max(r.first, r.second)));
}
```

Kapitola 4

Implementácia cestovej dekompozície

4.1 Lema o cestovej dekompozícii

V tejto kapitole si ukážeme ako implementovať dôkaz o cestovej dekompozícii s horným ohraničením cestovej šírky. Rozhranie, cez ktoré pristupuje užívateľ programu obsahuje iba Graph.

Keďže je lema 3 dokázaná matematickou indukciou, *pathDecomposition* bude volať rekurzívnu funkciu *makeDecomposition*.

```
vector<set<Number>> pathDecomposition(Graph &graph, int epsilonExp)
vector<set<Number>> makeDecomposition(Graph &graph, vector<set<Number>>
&decomposition, set<Number> &currentSet, set<Number> &verticesToDecompose)
```

Funkcia bude upravovať dekompozíciu, teda vektor množín vrcholov, ktorú si bude posúvať ako parameter do vnoreného volania. Množinu X, teda tú ktorá predstavuje momentálne posledné vrece v dekompozícii, budeme nazývať *currentSet*. Keď v currentSet nie je žiaden vrchol, znamená to, že sme na konci dekompozície jednej z množín V_0, V_1. Ak je množina currentSet neprázdna, môžeme sa dostať do štyroch rôznych situácií. Tri z nich vieme podľa dôkazu lemy pomerne priamo implementovať.

Ukážka kódu: prípad 1

```
verticesToDecompose.erase(n);
set<Number> newSet;
for (auto &v: currentSet)
    newSet.insert(v);
newSet.erase(n);
decomposition = makeDecomposition(graph, decomposition, newSet,
                    verticesToDecompose);
return decomposition;
```

Ukážka kódu: prípad 2

```
Number neighbor;
for (auto &n2: graph[n])
    if((currentSet.find(n2.n2()) == currentSet.end()) &&
            (verticesToDecompose.find(n2.n2()) != verticesToDecompose.end()))
        neighbor = n2.n2();
set<Number> newSet;
for (auto &v: currentSet)
    newSet.insert(v);
newSet.insert(neighbor);
decomposition.push_back(newSet);
newSet.erase(n);
verticesToDecompose.erase(n);
decomposition = makeDecomposition(graph, decomposition, newSet,
                    verticesToDecompose);
return decomposition;
```

Zostáva nám pozrieť sa na prípad *3*. Pokiaľ je v *currentSet* málo vrcholov, vykonáme dekompozíciu stromu.

Dekompozíciu stromu zrealizujeme tiež rekurzívne. Okrem aktuálneho vrcholu *vertex* budeme ako parameter posielať aj predošlý vrchol *parent*. Na začiatku si niektorý vrchol stupňa ≤ 2 označíme ako *parent* a dekompozíciu pustíme na jeho susedoch.

Idea je nasledovná:

1. Pokiaľ nemáš žiadneho suseda okrem *parent*, do dekompozície pridaj množinu s vrcholom emphvertex
2. Pre každého suseda okrem *parent* rekurzívne zavolaj túto funkciu, následne do každej pridanej množiny v dekompozícii pridaj aj vrchol *vertex*

Šírka rozkladu síce nie je na úrovni ako prezentovali Ellis a spol. [11], avšak stále je logaritmická, čo je pre nás uspokojivé.

Pseudokód dekompozície stromu:

```
treeDecomposition(graph, treeVertices, vertex, parent){
    decomposition, subdecomposition;
    numberOfNeighbors = 0;
    for (children of vertex in tree){
        numOfNeighbors++;
        subdecomposition = treeDecomposition(graph, treeVertices, child, vertex);
        for (bags of subdecomposition)
```

```
            add bag to decomposition
    }
    if vertex is leaf, add empty bag
    for (bags of decomposition)
        add vertex to all bags
    return decomposition;
}
```

Pokiaľ je v *currentSet* málo vrcholov, nachádzame sa v podprípade *3.B*, kedy do *currentSet* pridáme podľa lemy 3 2.2 nejaký počet vrcholov aby sme sa ocitli v jednom z predchádzajúcich vyriešených prípadov.

Ukážka z kódu: prípad 3.B

```
bool addedVertex = false;
for (auto &n: currentSet) {
    for (auto &inc: graph[n]) {
        if (currentSet.find(inc.n2()) == currentSet.end()) {
            currentSet.insert(inc.n2());
            addedVertex = true;
            break;
        }
    }
    if (addedVertex) break;
}
decomposition = makeDecomposition(graph, decomposition,
                    currentSet, verticesToDecompose);
    return decomposition;
```

Ako si môžeme všimnúť v kóde, nepostupujeme úplne podľa návodu z lemy. Do vreca *currentSet* vždy práve jeden vrchol - susedný s niektorým z tejto množiny. Daný krok nám v drvivej väčšine prípadov postačuje k tomu, aby sme sa ocitli v prípade 2, resp. 1.

Môže sa však stať, že sa opäť ocitneme v prípade 3.B, pokiaľ pridávam suseda takého vrchola, ktorý má až 3 susedov mimo množiny *currentSet* a pridaný vrchol má zvyšných dvoch susedov tiež mimo tejto množiny. To nám ale nevadí, rekurzívne sa vnoríme do nového problému a takýto prípad vyriešime ako predtým.

4.2 Veta o ohraničenej šírke rozkladu

V tejto podkapitole si zhrnieme postup ako demonštrovať cestovú dekompozíciu s ohraničenou šírkou rozkladu.

Ako vstup programu dostaneme ľubovoľný kubický graf. Na ňom nájdeme podľa implementácie v kapitole 3 optimálny rez.

Otázka ale znie: akú hodnotu má mať ϵ? Veta v článku o bisekcii je formulovaná takto

"...pre každé ϵ existuje také n že všetky grafy s aspoň n vrcholmi..."

My však riešime problém z opačnej strany. Dostaneme graf a snažíme sa nájsť také ϵ že výsledná dekompozícia bude pre nás prijateľná. Keďže nevieme nijako predpovedať akú hodnotu ϵ bude mať, musíme skúšať. Na začiatku priradíme epsilonu hodnotu $\frac{1}{2}$, spravíme dekompozíciu, zistíme šírku rozkladu a následne tento postup zopakujeme.

Používateľ pri vstupe dostane možnosť nastaviť ako dlho/presne bude program fungovať. Okrem grafu preto zadá aj číslo od 1 do 15. Toto číslo bude predstavovať exponent pre $\frac{1}{10}$ a takto upravené číslo bude predstavovať limit, pod ktorý sa bude musieť dostať ϵ. Čím bude exponent väčší, tým bude limit menší a program bude trvať dlhšie, za to ale môže potenciálne priniesť lepšie výsledky.

Na množinách V_0 a V_1 vykonáme podľa implementácie v kapitole 4 cestové dekompozície P_0, P_1, ktorých krajné množiny zodpovedajú $cut \cap V_0$, resp. $cut \cap V_1$. Zostáva nám spraviť dekompozíciu spájajúcu P_0 a P_1. Implementácia z dôkazu tohto postupu je priama.

Idea dekompozície rezových vrcholov:

```
(1)     vytvor vrece vrcholov z množiny v0 a zároveň patriacich rezu
(2)     označ niektorý vrchol z posledného vreca
(3)     vytvor kópiu posledného vreca a suseda vrchola z (2) cez rezovú
        hranu pridaj do tohto vreca. (takýto sused je práve jeden kvôli
        postupu kedy sme rez zmenšovali pokiaľ sa dalo)
(4)     vytvor ďalšie nové vrece, kde sused nahradí označený vrchol
(5)     kroky (2)-(4) opakuj kým je vo vreci nejaký vrchol z v0
```

Nakoniec, aby výsledná dekompozícia bol prehľadnejšia, spravíme menšiu redukciu: každú množinu v dekompozícii, ktorá je podmnožinou susednej, vyhodíme. K tomu nám bude stačiť jednoduchý for cyklus a funkcia *includes(x, y)*, ktorá zistí či y je podmnožinou x.

Keďže nadmnožiny týchto množín sú susedné a zahrňujú všetky hrany čo tieto množiny, podmienky regulárnej cestovej dekompozície zostanú neporušené.

Nakoniec bude môcť používateľ otestovať korektnosť vykonanej dekompozície.

Ukážka funkcie testujúcej korektnosť dekompozície

```
bool decompositionTest(Graph &graph_calculate,
            vector<set<Number>> decomposition){
    bool goodDecomposition = true;
    int num_of_checked = 0;
    map<int,bool>checked;
    for (auto &rot:graph_calculate)
        for (auto &edge:rot)
            checked[edge.e().to_int()] = false;
    for (auto &bag: decomposition)
        for (auto &v: bag){
            for(auto &neighbor: graph_calculate[v]){
                if ((bag.find(neighbor.n2())!=bag.end())&&
                    (!checked[neighbor.e().to_int()])){
                    checked[neighbor.e().to_int()] = true;
                    num_of_checked++;
                }
            }
        }
    if (num_of_checked!=graph_calculate.size())
        goodDecomposition = false;
    for (auto &rot: graph_calculate){
        int q = 0;
        for (auto &bag: decomposition){
            if ((bag.find(rot.n())!=bag.end())&&(q==0))
                q++;
            else if ((bag.find(rot.n())==bag.end())&&(q==1))
                q++;
            else if ((bag.find(rot.n())!=bag.end())&&(q==2))
                goodDecomposition = false;
        }
    }
    return goodDecomposition;
}
```

Testovanie programu

Program ktorý je prílohou k tejto práci obsahuje aj spustiteľný program `test.cpp`, ktorý obsahuje najmä testovanie väčších úsekov.

1. redBlackEdgesTest: spĺňa výsledok implementovanej funkcie požiadavky lemy 1, t.j. má množina vrcholov viac interných červených hrán ako externých čiernych?
2. oneHelpfulSetTest: je výsledok implementovanej funkcie 1-pomocná množina?
3. bisectionTest: sú množiny rezu disjunktné? Sú rezové množiny podmnožinou množín rezu?
4. pathDecompositionTest: spĺňa výsledok dekompozície podmienky dekompozície?

Testy bisectionTest a pathDecompositionTest boli spustené pre tri rôzne $epsilonExp = \{1, 8, 15\}$ na desiatich rôznych grafoch. Sedem z nich je definovaných v priečinku *graph_examples* vrámci prílohy a sú malej veľkosti - od desať vrcholov po 36 vrcholov. Zvyšné tri grafy boli vygenerované funkciou *random_regular_multigraph* z knižnice *ba-graph* - 100-vrcholový, 500-vrcholový a 1000-vrcholový. Testy redBlackEdges a oneHelpfulSet sú náročnejšie na nastavenie vstupných hodnôt, preto boli vykonané iba na siedmych menších grafoch. Okrem týchto štyroch bol pridaný test pre transformácie grafu vrámci hľadanie 1-pomocnej množiny. Tento test sa nachádza naspodku `test.cpp`. Keďže je testovanie približnej bisekcie aj cestovej dekompozície grafov s veľkým počtom vrcholov časovo náročné, okrem test.cpp bol vytvorený súbor `simple_test.cpp`, ktorý testuje všetko okrem 500-vrcholového a 1000-vrcholového grafu a iba pre $epsilonExp = 15$, za účelom zbehnutia v priebehu pár sekúnd.

Zhodnotenie implementácie

Program sme spustili na grafoch rozličnej veľkosti. Na obrázku 4.1 môžeme vidieť, že približne už od 100-vrcholových grafoch dosahuje implementovaný algoritmus šírku dekompozície veľkosti $\frac{1}{6}(V(G))$. Napriek tomu že je na niektorých grafoch možné spraviť oveľa lepšiu cestovú dekompozíciu, algoritmus aj kvôli rýchlosti a menšej zložitosti končí pri dosiahnutí horného ohraničenia.

Počet vrcholov	10	50	100	500	1000	2000	3000
Šírka dekompozície	4	11	17	83	167	334	500
(1/6)|V(G)|	1,66	8,33	16,66	83,33	166,66	333,33	500

Obr. 4.1: Šírka rozkladu cestovej dekompozície pre grafy s rôznym počtom vrcholov

Záver

Cestová dekompozícia grafov je veľmi dôležitým faktorom efektívnych algoritmov na grafoch, ale aj mnohých iných oblastiach matematiky a informatiky. Ako sme mohli vidieť, nadobudnúť dobrú šírku cestovej dekompozície je náročné a jej nadobudnutie si vyžaduje veľa študovania a skúmania.

Implementáciou cestovej dekompozície sme vytvorili vhodnú platformu na skúmanie súvislostí medzi grafmi s nižšou, ale aj vyššou cestovou šírkou, na ktorej vieme skúmať túto oblasť ešte do väčšej hĺbky.

Algoritmus v tejto práci spraví cestovú dekompozíciu grafu, ktorej šírka rozkladu grafov s rastúcim počtom vrcholov sa dostáva pod hranicu $\frac{1}{6}|V(G)|$.

Stromová šírka je podobne ako cestová šírka dôležitý a využívaný pojem v algoritmoch na grafoch. Keďže však o nej nie sú známe natoľko dobré výsledky, zameriavali sme sa prevažne na cestovú šírku.

Literatúra

[1] Burkhard Monien - Robert Preis, 2006, Upper bounds on the bisection width of 3- and 4-regular graphs, Journal of Discrete Algorithms 4, 475s.-491s.

[2] Fedor V. Fomin - Kjartan Høie, 2005, Pathwidth of cubic graphs and exact algorithms, Information Processing Letters 97, 191s.-196s.

[3] Jianmin Li - John Lillis - Chung-Kuan Cheng, Linear Decomposition Algorithm for VLSI Design Applications, Dept. of Computer Sci. & Engr., University of California, San Diego

[4] Wikipedia, Pathwidth, použité 24.1.2019, [online] Dostupné na internete: <https://en.wikipedia.org/wiki/Pathwidth>

[5] Wikipedia, Treewidth, použité 24.1.2019, [online] Dostupné na internete: <https://en.wikipedia.org/wiki/Treewidth>

[6] Wikipedia, Tree decomposition, použité 24.1.2019 [online] Dostupné na internete: <https://en.wikipedia.org/wiki/Tree_decomposition>

[7] Keijo Ruohonen, Graph Theory, 2013

[8] Reinhard Diestel, Graph Theory, 1997

[9] Shuji Isobe - Xiao Zhou - Takao Nishizeki, A Polynomial-Time Algorithm for Finding Total Colorings of Partial k-Trees, Graduate School of Information Sciences, Tohoku University

[10] Pengfei Wan - Jianhua Tu - Shenggui Zhang - Binlong Li, Computing the numbers of independent sets and matchings of all sizes for graphs with bounded treewidth, 2018, 42s.-47s.

[11] J.A. Ellis - I.H. Sudborough - J.S. Turner, The vertex separation and search number of a graph, Inform. and Comput. 113, 1994, 50s.–79s.