Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Generovanie kubických grafov zo 7-regulárnych grafov

Diplomová práca

2021
Bc. Juraj Žitňanský

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Generovanie kubických grafov zo 7-regulárnych grafov

Diplomová práca

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

Bratislava, 2021
Bc. Juraj Žitňanský

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Juraj Žitňanský
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Generovanie kubických grafov zo 7-regulárnych grafov
Generating cubic graphs from 7-regular graphs

Anotácia: Jaegerova hypotéza hovorí, že každý cyklicky 7-súvislý kubický graf je hranovo 3-zafarbiteľný. Na druhej strane, bezmostový kubický graf je hranovo 3-zafarbiteľný práve vtedy, ak má 2-faktor pozostávajúci iba z párnych cyklov. Je preto prirodzené že každý potenciálny kontrapríklad na jaegerovu hypotézu musí mať veľa nepárnych cyklov, najjednoduchšie 7-cyklov. Jeden zo spôsobov ako takéto grafy generovať je zobrať 7-regulárny graf a nahradiť vrcholy tohoto grafy cyklami dĺžky 7. Kľúčom k efektívnemu generovaniu takýchto grafov je efektívne detekovanie izomorfných štruktúr vytvorených počas generovania.

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: 31.10.2019

Dátum schválenia: 12.12.2019 prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie: Rád by som sa poďakoval môjmu školiteľovi doc. RNDr. Robertovi Lukoťkovi, PhD. za poskytnutie cenných rád a pripomienok, za pomoc a trpezlivosť. Ďalej ďakujem svojej rodine a kamarátom za pomoc a trpezlivosť.

Abstrakt

Hľavným cieľom diplomovej práce bol návrh a implementácia algoritmov, alebo programov určených na vygenerovanie neizomorfných kubických grafov s cyklickou súvislosťou 7 a následné overenie zafarbiteľnosti daných grafov, čiže zisťovanie či daný graf je snark alebo nie. V práci uvádzame spôsoby ako obmedziť nutnosť overovania izomorfizmu za pomoci globálneho zoznamu neizomorfných grafov. Podarilo sa nám vytvoriť viacero programov, ktoré sú schopné generovať hľadané grafy a taktiež program na zisťovanie zafarbiteľnosti, respektíve existencie nikde-nulového $\mathbb{Z}_2$x$\mathbb{Z}_2$ toku. Všetky tieto programy využívajú knižnice ba-graph a nauty.

Kľúčové slová: kubický graf, snark, zafarbiteľnosť, izomorfizmus, automorfizmus, nikde-nulový tok

Abstract

The main goal of diploma thesis was the design and implementation of algorithms or programs designed to generate nonisomorphic cubic graphs with a cyclic connectivity 7 and subsequent verification of the colorability of the graphs, which means determing wheter the graph is snark or not. In this thesis we present ways to reduce the need to check isomorphism by using a global lost of non-isomorphic graphs. We managed to create few programs that are capable of generating searched graphs and also a program for determining colorability and also existence of nowhere-zero $\mathbb{Z}_2$x$\mathbb{Z}_2$ flow. All these programs are using ba-graph and nauty library.

Keywords: cubic graph, snark, colorabilty, isomorphism, automorphism, nowhere-zero flow

Obsah

Zoznam obrázkov

Zoznam tabuliek

Úvod

V našej diplomovej práci sme sa zaoberali algoritmami na vygenerovanie vhodnej podtriedy kubických grafov. Konkrétne takej, medzi ktorými sa môžu nájsť kontrapríklady na Jaegerovu hypotézu. Jaegerova hypotéza hovorí, že každý cyklicky 7-súvislý kubický graf je hranovo 3-zafarbiteľný. Na druhej strane, bezmostový kubický graf je hranovo 3-zafarbiteľný práve vtedy, ak má 2-faktor pozostávajúci iba z párnych cyklov [9]. Hlavným cieľom diplomovej práce je teda efektívne vygenerovať kompletnú množinu takýchto grafov a následne overiť, či sa medzi nimi nenachádza nejaký graf, ktorý by danú hypotézu vyvrátil. Overovanie tejto vlastnosti sme uskutočnovali za pomoci úpravy programu na zisťovanie zafarbiteľnosti. Finálne programy na generovanie grafov a na zisťovanie zafarbiteľnosti majú byť následne implementované do univerzitnej knižnice ba-graph.

Prácu je rozdelená do piatich kapitol. V prvej kapitole popíšeme potrebné pojmy a definície, ktoré následne budeme využívať v priebehu diplmovej práce. Popíšeme o dôležitej podtriede kubických grafov, konkrétne snarkoch. Taktiež opíšeme izomirfizmus a spôsoby akými ho budeme hľadať.

V druhej kapitole opíšeme dva súčasne používané generovacie algoritmy a to konkrétne generovacie algoritmy pre kubické grafy a snarky.

V ďalšej kapitole sa budeme venovať využívaným knižniciam, ktoré poslúžia pri tvorbe programov na generovanie a zafarbovanie grafov.

Štvrtá kapitola sa bude zaoberať našimi spôsobmi generovania. Popíšeme si rôzne verzie postupov a ako dané riešenia boli efektívne.

V poslednej kapitole podrobne popíšeme implementáciu nášho programu na zafarbovanie grafov a na zisťovanie existencie nikde-nulového $\mathbb{Z}_2$x$\mathbb{Z}_2$ toku v grafe $K8$ s piatimi nahradenými vrcholmi cyklami dĺžky 7. Vysvetlíme spôsob akým zafarbujeme grafy, uvedieme časti zdrojového kódu a vysvetlíme jeho význam.

Diplomová práca z veľkej časti nadväzuje na moju bakalársku prácu preto niektoré časti práce sú viac menej identické s danou prácou [7]. A na základe konzultácie s mojim školiteľom sme usúdili, že by bolo zbytočné písať o tom istom 2 krát len inou formou.

Kapitola 1

Základné pojmy a definície

V diplomovej práci riešime problém zaoberajúci sa kubickými grafmi a snarkami. Pre lepšie pochopenie tejto podtriedy grafov si v tejto kapitole uvedieme základné definície týkajúce sa týchto grafov a ich vybraných vlastností.

1.1 Grafy a základné definície

Pod pojmom graf budeme mať na mysli jednoduchý neorientovaný graf bez násobnych hrán a slučiek. Ak nejaký graf bude obsahovať násobné hrany alebo slučky, tak takýto graf budeme nazývať multigraf.

Označením $V(G)$ respektíve $E(G)$ označujeme množinu vrcholov respektíve množinu hrán grafu G.

Ak je v jeden z vrcholov hrany e, potom je vrchol v incidentný s hranou e. Vrchol v_i nazývame *susedným* s vrcholom v_j práve vtedy, ak oba vrcholy sú incidentné s tou istou hranou.

Počet hrán incidentných s vrcholom v_i označujeme ako *stupeň vrcholu* v_i, označujeme to symbolom $deg(v_i)$. *Minimálny stupeň* vrcholu v grafe G označujeme $\delta(G) = min\{deg(v_i)|v_i \in V\}$. *Maximálny stupeň* vrcholu v grafe G označujeme $\Delta(G) = \max\{deg(v_i)|v_i \in V\}$.

Rozšírením grafu budeme v našej diplomovej práci definovať nejakú modifkáciu menších grafov, za pomoci ktorej budeme generovať väčsie grafy

Kružnica alebo *cyklus* v teórii grafov označuje graf, ktorý sa skladá z uzavretej postupnosti prepojených vrcholov.

Sieťou budeme nazyvať orientovaný graf G, ktorého množinu vrcholov budeme označovať $V(G)$ a množina jeho hrán budeme označovať $V(E)$. Každá hrana e má priradenú nezápornú celočíselnú hodnotu, túto hodnotu budeme nazývať *kapacita* označovať to budeme $c(e)$. Ak navyše označíme dva vrcholy, jeden ako začiatočný a jeden ako koncový, tak môžeme túto sieť nazývať *prietokovou sieťou*.

Tok v sieti G je funkcia $\phi : E(G) \rightarrow\rightarrow R$, ktorá spĺňa nasledujúce vlastnosti [10]:

1. $0 \leq \phi(e) \leq c(e)$ pre každú orientovanú hranu e,
2. pre každý vnútorný vrchol v, t.j. vrchol rôzny od zdroja a ústia platí *podmienka kontinuity*,

Podmienka kontinuity : $\sum_{e \in E-(v)} \phi(e) = \sum_{e \in E+(v)} \phi(e)$

Väčšinou sa požaduje $\phi(e) \neq 0$ pre všetky hrany e, taký tok sa volá *nikde-nulový tok*. Nikde-nulový k-tok je tok s hodnotami v množine $\{\pm 1, \pm 2, \ldots, \pm (k\text{-}1)\}$.

Mostom grafu nazývame takú hranu grafu G, ktorá nepatrí do žiadnej kružnice grafu G. Ak takúto hranu vynecháme, tak takto vytvorený podgraf bude obsahovať o 1 komponent viac ako graf G.

Chromatický index grafu je minimálny počet farieb, ktoré musíme použiť na zafarbenie hrán grafu, tak aby všetky susediace hrany mali rôzne farby. *Chromatické číslo grafu* je minimálny počet farieb, ktoré musíme použiť na zafarbenie vrcholov grafu tak, aby všetky susediace vrcholy mali rôzne farby.

Ak všetky vrcholy grafu majú rovnaký stupeň k , alebo ak $\delta(G) = \Delta(G) = k$, tak potom graf G nazývame *k-regulárny* graf. Trojregulárne grafy nazývame tiež aj *kubické grafy*.

Kubické grafy, ktoré majú chromatický index 3 budeme nazývať *zafarbiteľné* a tie čo majú chromatický index 4 budeme nazývať *nezafarbiteľné*.

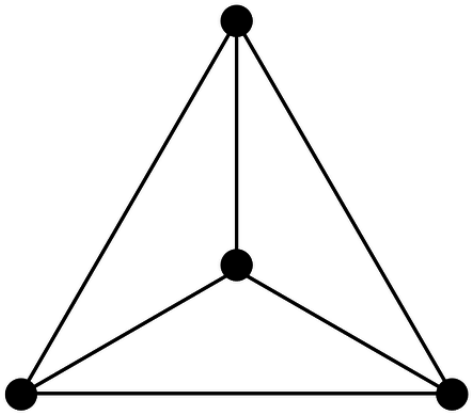
1.2 Snarky

Predtým, než zadefinujeme pojem snarku, musíme vysvetliť jednu dôležitú vlastnosť, ktorú použijeme pri charakterizácií snarkov. Táto dôležitá vlastnosť je *súvislosť (prepojenosť)*, presnejšie *hranová súvislosť*. Avšak, keďže, každý kubický graf je najviac hranovo 3-súvislý, potrebujeme lepšie rozlíšiť súvislosť kubických grafov. Graf G je k-hranovo cyklicky súvislý, ak odstránenie menej ako k hrán z grafu G nevytvorí dva podgrafy, z ktorých oba obsahujú aspoň jeden cyklus. Najväčšie celé číslo k, pre ktoré platí, že graf G je k-hranovo cyklicky súvislý sa nazýva cyklická hranová súvislosť grafu G a označuje sa ako $\lambda_c(G)$.

Ak graf G neobsahuje žiadny cyklický rez, tak potom $\lambda_c(G) = \infty$ ako je to napríklad pri grafe K_4.

Keďže hranová cyklická súvislosť a vrcholová cyklická súvislosť kubického grafu je rovnaká [11], môžme si dovoliť nepoužívať slovíčko hranová a ďalej budeme uvádzať len *cyklická súvislosť* a k-cyklicky súvislý graf.

Vizingova veta : Pre každý jednoduchý graf G platí, že chromatický index grafu G $\leq \Delta(G) + 1$



Obr. 1.1: Kubický graf K4

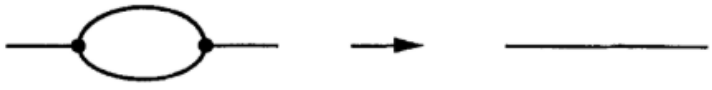
Vizingova veta prirodzene rozdeľuje jednoduché grafy na dve množiny a to konktrétne na grafy, na ktorých hranové zafarbenie stačí $\Delta(G)$ farieb, tieto grafy nazývame aj *grafy 1. triedy* a na grafy, na ktorých hranové zafarbenie treba a stačí $\Delta(G)$ +1 farieb, tieto grafy nazývame aj *grafy 2. triedy*.

Tvrdenie Vizingovej vety neplatí pre grafy s násobnými hranami s výnimkou kubických grafov s násobnými hranami.

Snark je súvislý bezmostový kubický graf, ktorý nie je hranovo 3-zafarbiteľný, jeho chromatický index je 4, čiže je nezafarbitľný. Alebo môžeme povedať, že je to bezmostový kubický graf 2. triedy.

Snarky v mohých prípadoch slúžia ako protipríklady, často v definíciach o snarkoch sa vyskytuje slovíčko *netriviálny (non-trivial)*. Toto slovíčko pri grafoch väčšinou znamená, že graf neobsahuje most. Okrem obsahovania mostu, snarky obsahujú aj ďalšie vlastnosti, ktoré môžu pre daný graf znamenať, že je triviálny. A to vtedy ak je snark triviálnou variáciou iného snarku.

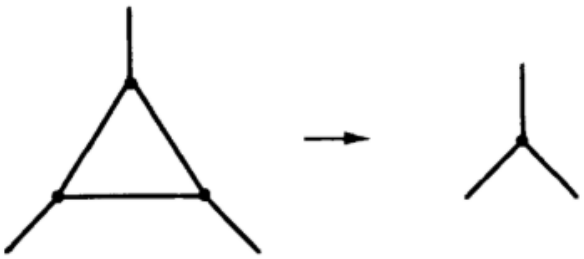
Snark považujeme za triviálny ak obsahuje *digon*, trojuholník alebo cyklus dĺžky 4. Ak snark G obsahuje digon, môžeme ho z grafu jednoducho odstrániť a nahradiť novou hranou, čím nám vznikne nový graf G', ako môžeme vidieť na obrázku 1.2.



Obr. 1.2: Nahradenie digonu

Ak snark G obsahuje trojuholník, môžeme ho nahradiť jedným vrcholom, výsledkom tohoto ako možeme vidieť na obrázku 1.3 bude nový graf G'

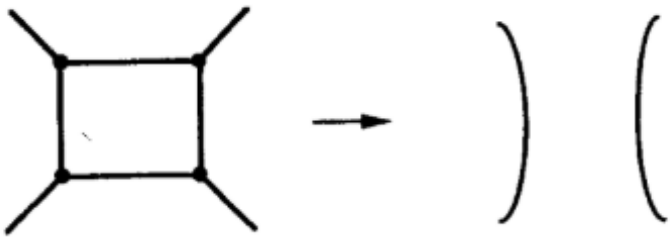
Pri týchto dvoch operáciach je ľahko nahliadnuteľne, že graf G je zafarbiteľný vtedy a len vtedy ak aj graf G' je zafarbiteľný. Pridávaním digonu alebo trojuholníkov do



Obr. 1.3: Nahradenie trojuholníka vrcholom

grafu takýmto spôsobom, nám dáva možnosť vykonštruovať nekonečne veľa snarkov z jedného pôvodného snarku, ale všetky snarky vytvorené takýmto spôsobom budú len triviálnou modifikáciou pôvodného.

A nakoniec, ak snark G obsahuje kružnicu dĺžky 4, možeme ju nahradiť dvoma paralelnými hranami ako môžeme vidieť na obrázku 1.4, čo bude viesť k vytvoreniu nového grafu G'.



Obr. 1.4: Nahradenie kružnice dĺžky 4 dvoma paralelnými hranami

Zase je ľahko nahliadnuteľné, že graf G' je zafarbiteľný tak aj graf G je zafarbiteľný. Toto však naopak vo všeobecnosti neplatí.

Taktiež sa stalo zaužívaným vylúčiť snarky, ktoré su hranovo 2 alebo 3-súvislé . Je ľahko nahliadnuteľné, že pri použití takých minimálnych rezov, môže byť snark rozdelený do dvoch menších kubických grafov, z ktorých aspoň jeden je snark. Inými slovami, snark s nezávislým dvoj-rezom alebo troj-rezom môže byť zredukovaný do menšieho snarku alebo rozdelený do dvoch menších snarkov. V tomto zmysle snarky s malými cyklovo-rozdelovacími hranovými rezmi sa tiež stanú triviálnou modifikáciou tých s menej vrcholmi. Preto netriviálne snarky by maly byť najmenej cyklicky 4 súvislé.

Z týchto prípadov nám teda vyplýva, že netriviálne snarky by mali mať obvod aspoň 5 a ich cyklická súvisloť by mala byť aspoň 4. Z tohoto dostávame nasledujúce definície: *Netriviálny snark* je 4-hranovo cyklicky súvislý nezafarbiteľný kubický graf s obvodom aspoň 5. Ostatné snarky sa nazývajú *triviálne*. Pre dlhšiu diskusiu o netrivialite snarkov Detailnejšiu a dlhšiu diskusiu o netrivialite snarkov nájdete v nasledujúcom článku [12]

1.3 Izomorfizmus grafov

Izomorfizmus grafov G a H sú bijektívne zobrazenia vrcholov $f_v : V(G) \to V(H)$ a hrán $f_e : E(G) \to E(H)$, pre ktoré platí, že každý vrchol $v \in V(G)$ je incidentný s hranou $e \in E(G)$ vtedy a len vtedy ak je vrchol $f_v(v) \in V(H)$ incidentný s hranou $f_e(e) \in E(H)$. Grafy G a H nazývame *izomorfné grafy* ak medzi nimi existuje izomorfizmus.

Izomorfné grafy majú spoločné vlastnosti, ktoré nazývame invariantami grafov. Výpočet invriantu grafov môže byť založené na relatívne jednoduchých pravidlách s polynomiálnymi zložitosťami. Formálne invariantom grafu nazývame takú funkciu γ, ktorá pre izomorfné grafy G a H nadobúda zhodné hodnoty, t.j. $\gamma(G) = \gamma(H)$.

Invarianty môžeme rozdeliť podľa rozmeru na výslednej funkcií. Pozornosť budeme venovať číselným a vektorovým invariantom.

Reprezentantom číselných invariantov sú:

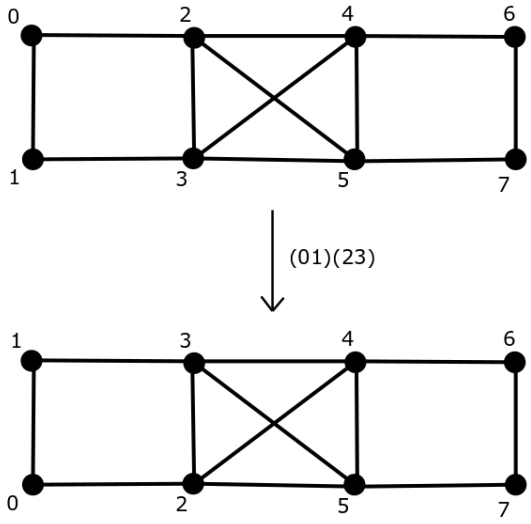
- počet vrcholov *($n = |V|$)*,
- počet hrán *($m = |E|$)*,
- najvyšší a najnižší stupeň ($\Delta(G)$ a $\delta(G)$),
- Priemerný stupeň – *$d(G)$*, platí: $\Delta(G) \geq d(G) \geq \delta(G)$,

Reprezentantom vektorových invariantov sú:

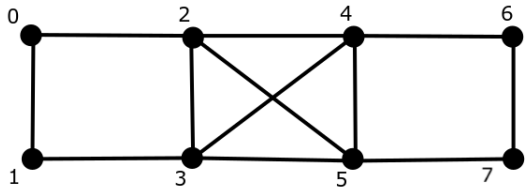
- Skóre grafu. Neklesajúcu postupnosť stupňov vrcholov grafu G označujeme $D = (\deg(v_i, v_i \in V(G))$ nazývame *skóre grafu*. Dva grafy, ktoré nemajú zhodné skóre nemôžu byť izomorfné.

Automorfizmus je špeciála verzia izomorfizmu. Automorfizmus je izomorfné zobrazenie objektu do seba samého. Automorfizmus grafu je permutácia vrcholov taká, že množiná hrán ostáva bez zmeny 1.5.

Na obrázku 1.5 sme zamenili vrcholy 0 a 1 a zároveň 2 a 3. Ako môžeme vidieť množina hrán ostáva rovnaká a preto permutácia vrcholov (0 1) (2 3) je automorfizmus. Aplikáciou dvoch autmorfizmov, jeden po druhom dostávame taktiež automorfizmus. Všetky automofizmy grafov spolu s binárnou operáciou skladania vytvárajú grupu automorfizmov grafu [4]. Ľahko sa dá overiť, že to je naozaj grupa, pretože pre ňu platia základné vlastnosti grupy, ako napríklad, že operácia skladania je asociativna. Ku každému prvku z grupy existuje inverzný prvok. A existuje aj jej neutrálny prvok. Pretože počet automorfizmov môže byť pomerne veľký, je oveľa efektívnejšie pracovať s množinou *generátorov grupy automorfizmov*. Množina generátorov grupy je podmnožina grupy, taká, kde každý prvok grupy môže byť vyjadrený kombináciou konečného počtu prvkov podmnožiny a ich inverzami. Každý prvok z danej množiny sa nazýva generátorom grupy. Čiže množina generátorov grúp automorfizmov je špeciálna množina automorfizmov, taká, že každý automorfizmus z grupy sa dá vyjadriť kombináciou



Obr. 1.5: Automorfizmus



Obr. 1.6: Graf-príklad

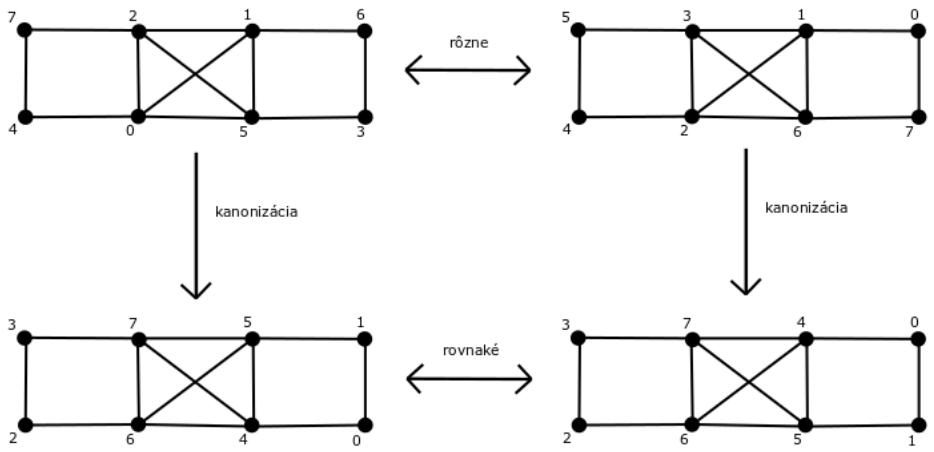
automorfizmov tejto množiny. Automorfizmus taktiež definuje aj vzťahy ekvivalencie medzi vrcholmi v grafe. Dva vrcholy sú ekvivalentné, ak existuje automorfizmus z jedného vrchola do druhého. Množiny ekvivalentných vrcholov sa nazývajú orbity. V grafe 1.6 sú orbity : {0, 1, 6, 7} a {2, 3, 4, 5}.

1.4 Kanonická forma grafu

Pracujeme s funkciou *Canon* z knižnice Nauty, ktorá pre graf na vstupe vygeneruje jeho kanonickú formu. Pre takto vytvorený označený graf pomocou funkcie *Canon* platia určité tvrdenia :

- graf G a $Canon(G)$ sú izomorfné
- ak $Canon(G) = Canon(H) \Leftrightarrow$ grafy G a H sú izomorfné
- ak $Canon(G) \neq Canon(H) \Leftrightarrow$ grafy G a H sú neizomorfné

Tieto tvrdenia nám pomáhajú práve pri hľadaní izomorfizmu medzi dvoma grafmi a to tak, že pokiaľ chceme zistiť, či sú dva grafy G a H izomorfné, musíme previesť oba grafy na ich kanonickú formu, čiže dostaneme grafy $Canon(G)$ a $Canon(H)$. Následne tieto 2 grafy porovnáme a pokiaľ sú totožné, je zaručené, že sú oba grafy G a H vzájomne izomorfné. A zase naopak, ak by sme zistili, že $Canon(G)$ a $Canon(H)$ nie sú totožné, môžeme s istotou tvrdiť, že grafy G a H sú neizomorfné. Aj keď hľadanie kanonickej formy je náročnejšie ako hľadanie samotného izomorfizmu, tento spôsob je výhodný použiť ak máme veľké množstvo grafov. Pretože nemusíme zakaždým zisťovať izomorfizmus medzi každou dvojicou grafov, ale raz si vypočítame kanonickú formu daného grafu a následne budeme porovnávať len tie a to je jednoduché, pretože tie môžeme mať uložené aj vo forme stringového reťazca. Pojem *kanonizácia* označuje prevedenie grafu G na jeho kanonickú formu. 1.7



Obr. 1.7: 2 grafy pred a po prevedení na ich kanonickú formu

Kanonickou redukciou budeme nazývať špeciálne zobrazenie z grafu G' do grafu G, kde graf G' vznikol rozšírením grafu G. Zároveň toto zobrazenie musí rešpektovať izomorfizmus. Ak platí, že graf G vznikol rozšírením grafu H a graf G' je izomorfný s grafom G, tak potom G' vznikol nejakým rozšírením grafu H', ktorý je izomorfný s grafom H. Kanonická reduckia je potom zobrazenie, kroré grafu G priraďuje graf H tak, že G je rozšírením grafu H, navyše kanonické redukcie izomorfných grafov sú si izomorfné. Práve aplikácia vhodnej kanonickej redukcie je ďalším z postupov, ktorými budeme zabezpečovať hľadanie izomofizmu grafov.

Kapitola 2

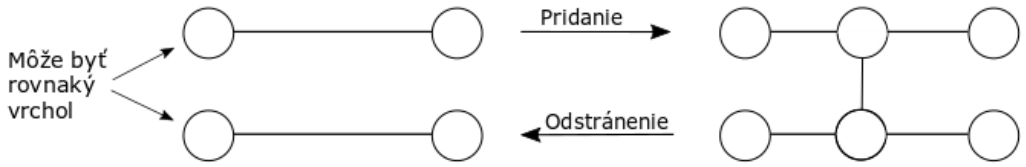
Generovanie

Hlavným cielom tejto kapitoly je ukázať, aké techniky sa reálne používaju pri generovaní grafov. Existuje mnoho spôsobov ako efektívne generovať rôzne typy grafov. V našej dipolmovej práci sa pozrieme len na algoritmy týkajúce sa kubických grafov. Kubické grafy často slúžia ako najmenšie a potenciálne najľahšie proti príklady pre rôzne otvorené problémy v teórií grafov, preto je pre nás práve trieda kubických grafov taká zaujímavá.

2.1 Generovanie kubických grafov

Aj v tomto prípade existuje mnoho algoritmov ako generovať rôzne podmnožiny kubických grafov. Ako napríklad algoritmus na generovanie snarkov bez malých kružníc a mnohé ďalšie. My si konkrétne ukážeme algoritmus, ktorý umožnuje generovanie kubických grafov. Tento algoritmus bol publikovaný v článku [3], ktorý vyšiel v roku 2011 a jeho autormi sú Gunnar Brinkmann, Jan Goedgebeur a Brendan D. Mckay. Základná myšlienka tohoto algoritmu je, že vytvoríme kubický graf *G'* s n vrcholmi z kubického grafu G, ktorý mal *n-2* vrcholov. A to tak, že dve ľubovoľné hrany pôvodného kubického grafu G rozdelíme dvomi novými vrcholmi v_{n-1}, v_n a súčasne medzi nimi vytvoríme hranu. Grafické znázornenie môžeme vidieť na obrázku 2.1 [3]. Opačná operácia k tejto je odoberanie dvoch susedných vrcholov v_{n-1}, v_n a hrán, ktoré vychádzali z týchto vrcholov a následné pridanie hrany medzi dva vrcholy iné ako v_n, ktoré boli susedné s vrcholom v_{n-1} a pridanie hrany medzi dva vrcholy iné ako v_{n-1}, ktoré boli susedné s vrcholom v_n. Takúto operáciu nazývame odoberanie hrany a v prípade že takto vytvorený graf je súvislý kubický graf nazývame takúto hranu medzi vrcholmiv_{n-1}, v_n **odstrániteľná**, inak je takáto hrana **neodstrániteľná**. Kubický graf, ktorý neobsahuje žiadne odstrániteľné hrany nazývame **primárny graf**. Podľa definícií, môže byť každý súvislý kubický graf skonštruovaný z primárneho grafu rekurzívnym opakovaním operácie pridanie hrany. Čiže našou hlavnou úlohou bude zistiť, ako vytvoriť primárne

grafy.

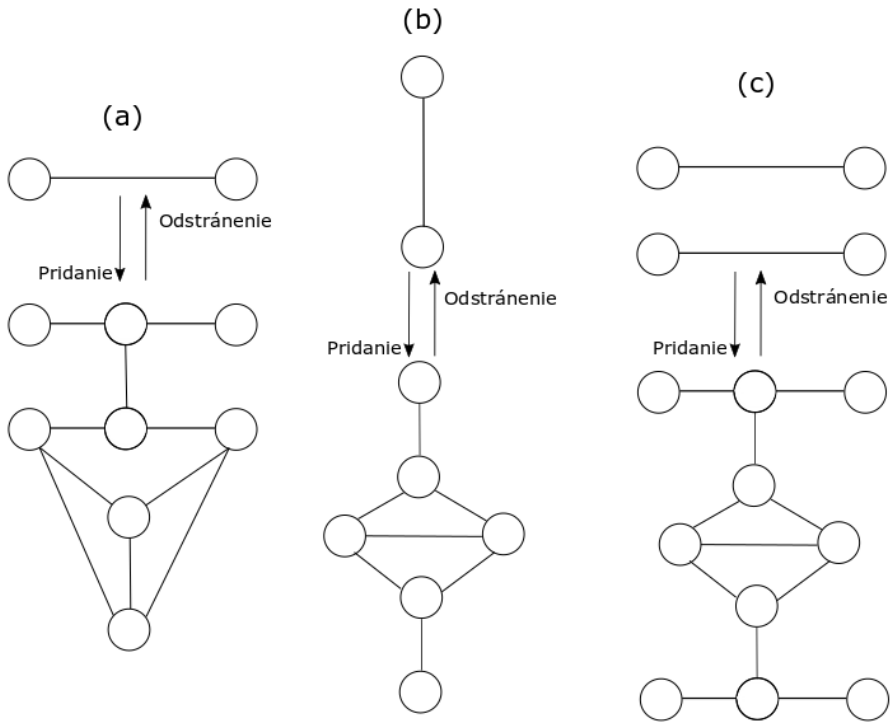


Obr. 2.1: Grafická ukážka pridávania a odoberania hrany

Známy spôsob ako generovať primárne grafy je využiť graf K_4 redukovaný o hranu, takýto graf budeme označovať ako K_4^-. Takýto graf má dva vrcholy stupňa 3 a dva vrcholy stupňa 2. Vrcholy stupňa 2 využijeme pre pripojenie k existujúcej hrane (obrázok 2.3 prípad (b)), alebo k dvom existujúcim hranám nemajúcim spoločný vrchol (obrázok 2.3 prípad (c)). Pripojenie grafu do jednej hrany znamená zrušenie tejto hrany a jej nahradenie hranami medzi koncovými vrcholmi pôvodnej hrany a vrcholmi stupňa 2, grafu K_4^-. V prípade, že vloženie grafu K_4^- realizujeme do dvoch hrán grafu, tieto hrany rozdelíme novými vrcholmi, z ktorých vedieme hranu k vrcholom stupňa 2. V oboch prípadoch novo vzniknuté hrany vytvorili na svojich koncových vrcholoch, vrcholy stupňa 3.

Ďalším spôsobom je využiť graf K_4^+, ktorý vznikne z grafu K_4 vložením vrcholu do jednej z hrán. Tento vrchol bude jediným vrcholom so stupňom 2. Vrcholy stupňa 2 využijeme pre pripojenie k existujúcej hrane (obrázok 2.3 prípad (a))) tak, že hranu rozdelíme novým vrcholom, ktorý spojíme hranou s vrcholom stupňa 2 v grafe K_4^+.

Vyššie uvedený postup generovania grafu pomocou K_4^- a K_4^+ je návodom pre dôkaz tvrdenia: Trieda primárnych grafov môže byť vygenerovaná z K_4 rekurzívnym aplikovaním konštrukčných operácií znázornených na obrázku 2.3. [3]

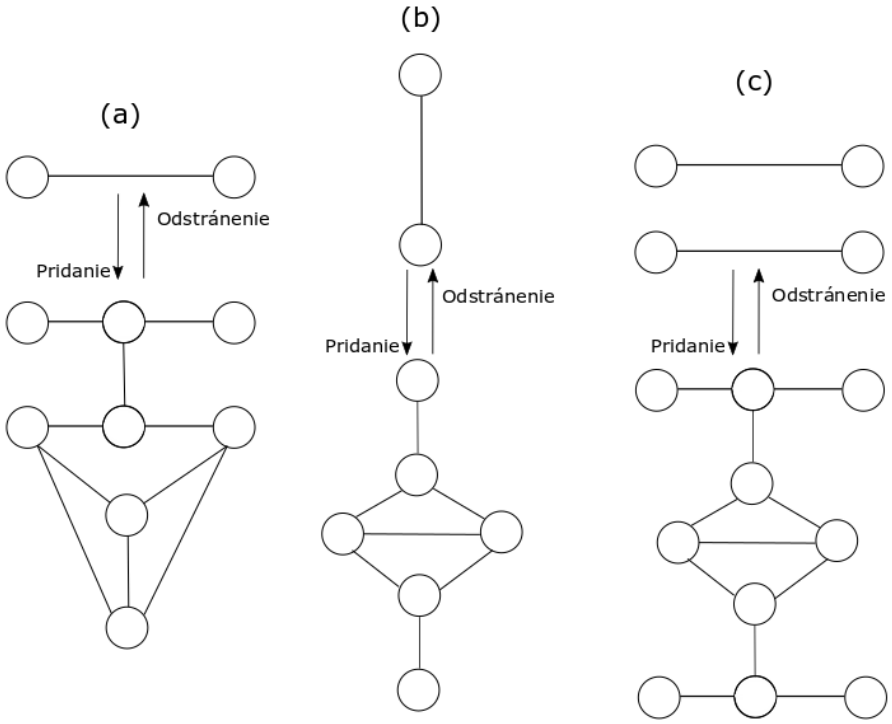


Obr. 2.2: Konštrukčné operácie pre primárne grafy

Známy spôsob ako generovať primárne grafy je využiť graf K_4 redukovaný o hranu, takýto graf budeme označovať ako K_4^-. Takýto graf má dva vrcholy stupňa 3 a dva vrcholy stupňa 2. Vrcholy stupňa 2 využijeme pre pripojenie k existujúcej hrane (obrázok 2.3 prípad (b)), alebo k dvom existujúcim hranám nemajúcim spoločný vrchol (obrázok 2.3 prípad (c)). Pripojenie grafu do jednej hrany znamená zrušenie tejto hrany a jej nahradenie hranami medzi koncovými vrcholmi pôvodnej hrany a vrcholmi stupňa 2, grafu K_4^-. V prípade, že vloženie grafu K_4^- realizujeme do dvoch hrán grafu, tieto hrany rozdelíme novými vrcholmi, z ktorých vedieme hranu k vrcholom stupňa 2. V oboch prípadoch novo vzniknuté hrany vytvorili na svojich koncových vrcholoch, vrcholy stupňa 3.

Ďalším spôsobom je využiť graf K_4^+, ktorý vznikne z grafu K_4 vložením vrcholu do jednej z hrán. Tento vrchol bude jediným vrcholom so stupňom 2. Vrcholy stupňa 2 využijeme pre pripojenie k existujúcej hrane (obrázok 2.3 prípad (a))) tak, že hranu rozdelíme novým vrcholom, ktorý spojíme hranou s vrcholom stupňa 2 v grafe K_4^+.

Vyššie uvedený postup generovania grafu pomocou K_4^- a K_4^+ je návodom pre dôkaz tvrdenia: Trieda primárnych grafov môže byť vygenerovaná z K_4 rekurzívnym aplikovaním konštrukčných operácií znázornených na obrázku 2.3. [3]



Obr. 2.3: Konštrukčné operácie pre primárne grafy

Kubické grafy teda vieme generovať v dvoch krokoch :

K_4

⇓ (operácie typu (a) (b) (c) z obrázku 2.3)

Primárne grafy

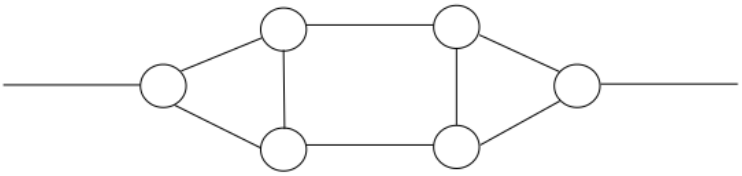
⇓ (operácia pridanie hrany z obrázku 2.1)

Všetky kubické grafy

Generovanie grafov s redukovateľnými trojuholníkmi

Všimnime si, že operáciu pridania hrán aplikovanú na dve hrany, ktoré majú spoločný koncový bod je možné vidieť ako nahradenie spoločného koncového bodu trojuholníkom a výsledok závisí len od toho, aký bol ten spoločný koncový bod a nie, ktoré dve hrany boli použité. Takúto operáciu budeme nazývať vkladanie trojuholníka. Podobne aj opačnú operáciu ku vkladaniu trojuholníka, ktorú nazývame odoberanie trojuholníka, možno vidieť ako nahradenie redukovateľného trojuholníka jedným vrcholom. Odobratiu trojuholníka dávame prioritu dávame oproti iným odobratiam hrán v tom zmysle, že ak má graf redukovateľné trojuholníky, požadujeme aby boli vytvorené trojuholníkovým vkladaním. To nám umožňuje spájať trojuholníkové operácie. Myšlienkou je súčasne odobrať každý redukovateľný trojuholník, je tu ale aj výnimka a to taká, že dva trojuholníky v podgrafe ext(K_4^-), znázornený na obrázku 2.4, v ktorom nemôžu byť tieto trojuholníky odstránené súčasne. Avšak odstránenie ktoréhokoľvek z dvoch

trojuholníkov v ext(K_4^-) spôsobí vytvorenie rovnakého, menšieho grafu, takže našu redukciu zlúčených trojuholníkov definujeme ako odstránenie jedného trojuholníka z každého ext(K_4^-) a každého ďalšieho redukovateľného trojuholníka. To poskytuje (až do izomorfizmu) jedinečného predka pre každý súvislý kubický graf, ktorý má redukovateľné trojuholníky.



Obr. 2.4: Podgraf ext(K_4^-) s dvoma redukovateľnými trojuholníkmi, ktoré nemôžu byť odstránené súčasne

Ďalej identifikujeme opačnú operáciu k redukcií zlúčených trojuholníkov. Pre graf $G = (V, E)$ nazývame množinu $S \subseteq V$ rozšíriteľnú, ak aspoň jeden vrchol každého redukovateľného trojuholníka G je obsiahnutý v S. Potom pridanie zlúčených trojuholníkov je vloženie trojuholníka do každého z vrcholov v S. Je ľahké vidieť, že toto je operácia opačná k redukcií zlúčených trojuholníkov. Po pridaní zlúčených trojuholníkov, všetky ext(K_4^-) podgrafy a ostatné redukovateľné trojuholníky boli vytvorené v tejto operácií. Použitie pridanie zlúčených trojuholníkov do odlišných rozšíriteľných množín grafu G by mohlo poskytnúť izomorfné grafy. Aby sme tomu predišli, definujeme vzťah ekvivalencie na rozšíriteľných množinách a aplikujeme pridanie zlúčených trojuholníkov iba do jednej množiny v každej triede ekvivalencie.

Vzťah ekvivalencie $\equiv$ na rozšíriteľných množinách je generovaný nasledujúcimi dvoma ekvivalenciami:

(a) Ak existuje automorfizmus γ G s γ $(S) = S'$, potom $S \equiv S'$

(b) Ak $|S| = |S'|$ a $(S \backslash S') \cup (S' \backslash S)$ je množina vrcholov K_4^- so stupňom dva v grafe G tak, že každý z S, S' obsahuje presne jeden vrchol v tomto K_4^-, potom $S \equiv S'$

Uvažujme graf G a dve rozšíriteľné množiny S, S' grafu G. Nech $T(G, S)$, resp. $T(G, S')$, označujú grafy získané aplikáciou pridania trojuholníka na S, resp. S', Potom $T(G, S)$ a $T(G, S')$ sú izomorfné, len ak $S \equiv S'$. [3]

Pretože pre každý graf s redukovateľnými trojuholníkmi je graf vyplývajúci z redukcie zlúčených trojuholníkov jednoznačne určený, dostaneme nasledujúce tvrdenie: Ak presne jeden zástupca každej triedy izomorfizmu pre súvislé kubické grafy až do $n - 2$ vrcholov je daný, potom aplikovanie pridania zlúčených trojuholníkov na jedného člena každej triedy ekvivalencie rozšíriteľných množín, ktoré vedú ku kubickému súvislému grafu s n vrcholmi, generuje presne jedného zástupcu pre každú triedu izomorfizmu

súvislých kubických grafov na n vrcholoch, ktoré obsahujú redukovateľné trojuholníky. [3]

Generovanie neprimárnych grafov bez redukovateľných trojuholníkov

V princípe sú neprimárne súvislé kubické grafy bez redukovateľných trojuholníkov na n vrcholoch generovateľné aplikovaním operácie pridanie hrany do každej dvojice vrcholov v grafe s $n-2$ vrcholmi čo garantuje, že vo výslednom grafe sa nenachádzajú žiadne redukovateľné trojuholníky. Takúto dvojicu hrán nazývame rozšíriteľnou dvojicou hrán. To však môže viesť k vytvoreniu izomorfných kópií. Preto teraz popíšeme ako si môžeme byť istý, že zoznam vygenerovaných grafov obsahuje len neizomorfné grafy.

Prvou úlohou je definovať pre každý neprimárny graf bez redukovateľných trojuholníkov kanonické odstraňovanie hrany, ktorá je jedinečná pre izomorfizmus. Výsledkom vykonávania kanonického odstraňovania hrán bude graf G', jednoznačne určený grafom G, z ktorého graf G môže byť vytvorený pridaním hrany. Graf G budeme akceptovať vtedy a len vtedy, ak je vyrobený z grafu G' opačnou operáciou kanonického odstránenia hrán, inak graf G zamietame.

2.2 Generovanie snarkov

Detailnejšie nájdete tento algoritmus popísaný v článku [2]. Generovací algoritmus pre snarky je založený na generovacom algoritme pre kubické grafy. Počas generovania sa vytvárajú trojuholníky a keďže slabé snarky majú obvod najmenšej kružnice aspoň 4, posledná operácia musí byť vždy poupravená operácia vkladania hrany a to tak, že vloženie novej hrany bude možné iba medzi hrany, ktoré nemajú spoločný vrchol. Tento algoritmus na generovanie snarkov, ktorý opisujeme implementuje pohľad dopredu, ktorý môže často rozhodnúť, že operácia vkladanie hrany povedie k vytvoreniu zafarbiteľného grafu a tým pádom sa mu možno vyhnúť. S narastajúcim počtom vrcholov rýchlo narastá aj počet kubických grafov, preto aj také vyhnutie sa vloženiu poslednej hrany poskytuje značné zrýchlenie algoritmu.

Algoritmus využíva známe tvrdenia :

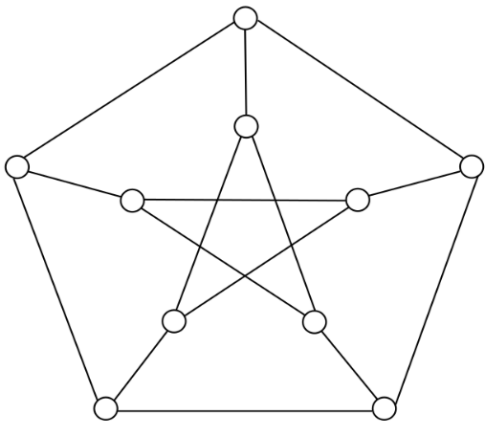
1. Kubický graf je zafarbiteľný vtedy a len vtedy ak má 2-faktor všetkých cyklov párnej dĺžky, taktiež sa to nazýva párny 2-faktor. [2]

2. Nech F je párny 2-faktor kubického grafu G. Potom všetky grafy G získané aplikovaním operácie vloženia hrany medzi dve hrany e a e', ktoré sú súčasťou rovnakého cyklu v F, budú zafarbiteľné. [2]

3. Nech kubický graf G má chromatické číslo. Ak dve hrany e, e' patria do rovnakého cyklu 2-faktoru, ktorý je indukovaný dvoma rôznymi farbami, graf G získaný

aplikovaním operácie vloženia hrany medzi hrany e a e' bude zafarbiteľný. [2]

4. Nech kubický graf G je zafarbiteľný, potom všetky grafy G' vytvorené z grafu G aplikovaním operácie vloženia hrany, tak, že vrcholy tejto novo pridanej hrany sú súčasťou kružnice s obvodom 4 v G' sú zafarbiteľné. [2]



Obr. 2.5: Úkážka snarku - Petersenov graf

Ak teda chceme generovať všetky snarky s n vrcholmi, nemusíme aplikovať operáciu vloženia hrany do grafov s $n-2$ vrcholmi, sú zafarbiteľné, ak vložená hrana bude súčasťou štvorca. Na prvý pohľad vyzerá tvrdenie 4 zaujímavo len pre operácie, ktoré nevytvárajú trojuholníky, pretože v poslednom kroku nikdy trojuholníky nevyrábame. Ale v skutočnosti to pre trojuholníky umožňuje skoršie vyhľadávanie. Dôležité je, že posledná vložená hrana je vždy hrana v cykle s najmenším obvodom.

Takže predpokladajme, že máme graf G s $n-2$ vrcholmi a aspoň s jedným trojuholníkom a že chceme vytvoriť slabé snarky s n vrcholmi. Z takéhoto grafu, môžeme získať iba grafy s obvodom najviac 4. Takže posledná vložená hrana bude v kružnici s obvodom 4. Ak by graf G bol zafarbiteľný, potom aj všetci potomkovia by boli zafarbiteľný, takže nemusíme konštruovať zafarbiteľné grafy s trojuholníkmi veľkosti $n-2$. Takže pre trojuholníky udáva tvrdenie 4 hraničné kritérium, ktoré sa môže použiť na úrovni $n-4$.

Kapitola 3

Nauty, ba-graph

V tejto kapitole si popíšeme knižnice, ktoré sme použili pri generovaní grafov. Popíšeme reprezentáciu grafu v danej knižnici, základné operácie s grafom. Tiež ukážeme alebo vysvetlíme veci, ktoré budeme potrebovať k tomu, aby sme vedeli pracovať s danou knižnicou.

3.1 Nauty

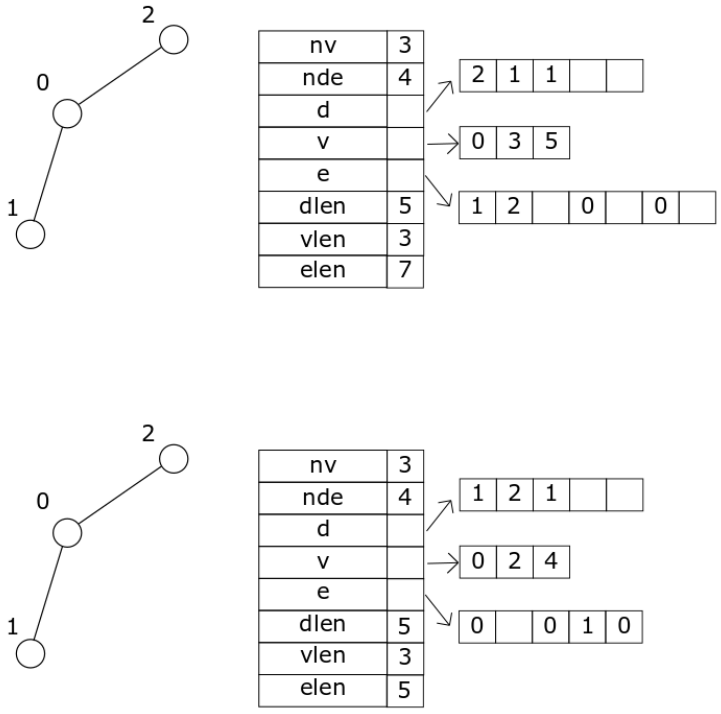
Nauty je grafová knižnica napísaná v jazyku C, ktorá slúži primárne na zisťovanie grúp automorfizmov pre grafy a na testovanie izomorfizmu medzi grafmi. Tieto funkcie knižnica poskytuje s využitím kanonickej formy, ktorej vytvorenie predstavuje jednu z kľúčových operácií nad grafom.

Knižnica podporuje 2 typy grafov, a to dense grafy (husté grafy) a sparse grafy (riedke grafy). V ďalšom texte budeme popisovať len vlastnosti implementácie spojené s riedkymi grafmi. Popíšeme reprezentáciu grafu rádu n. Vrcholy v tomto grafe sú očíslované $0, 1, ..., n-1$. Tieto grafy využívajú typ sparsegraph, ktorý je ukladaný ako štruktúra s nasledujúcimi parametrami [1]:

- **int nv:** počet vrcholov grafu
- **site_t nde:** počet orientovaných hrán, kde slučka sa ráta ako 1 a ostatné neorientované hrany ako 2
- **site_t *v:** pointer na pole veľkosti aspoň nv
- **int *d:** pointer na pole veľkosti aspoň nv
- **int *e:** pointer na pole veľkosti aspoň nde
- **site_t vlen, dlen, elen:** reálne veľkosti polí v, d a e.

Pre každý vrchol $i = 0, ..., n-1$, $d[i]$ reprezentuje stupeň daného vrcholu v grafe, $v[i]$ reprezentuje index v poli e a to tak, že $e[v[i]], e[v[i]+1], ..., e[v[i]+d[i]-1]$ sú vrcholy ku ktorým je vrchol i spojený hranou.

Predtým ako môže byť použitá štruktúra sparsegraph použitá, je potrebné aby bola inicializovaná. Polia d, v, e by mali byť nastavené na $NULL$. Atribúty $dlen$, $vlen$ a $elen$ by mali byť nastavené na 0. Po inicializácií, sa veľkosti polí automaticky nastavia na potrebné hodnoty.



Obr. 3.1: Reprezentácia sparsegraphu [1]

3.1.1 Volanie nauty

Volanie do nauty má tvar :

$nauty(g, lab, ptn, active, orbits, options, stats, workspace, worksize, m, n, canong)$

Význam jednotlivých parametrov:

graf alebo sparsegraph *g: Vstupný graf. Iba na čítanie.

int *lab, *ptn: Dve polia veľkosti n. Ich použite závisí od rôznych options. Ak $options.defaultptn = TRUE$ tak hodnoty v týchto poliach sú ignorované, v opačnom prípade rozhodujú o ofarbení grafu. Ak $options.getcanon = TRUE$, v tom prípade je hodnota lab na výstupe kanonické označenie grafu, pretože uvádza vrcholy grafu g v takom poradí v akom musia byť pre **canong**.

set *active: Tento argument sa využíva len výnimočne, nauty bude fungovať správne aj keď namiesto neho dáme $NULL$.

int *orbits: Výstupný parameter, ktorý si nauty knižnica vyráta pri transformácií grafu na sparsegraf. Je to pole veľkosti n, ktoré obsahuje orbity grupy automorfizmu.

optionblk *options: Štruktúra, ktorá obsahuje nastavenia (options) k procedúre.

statsblk *stats: Štruktúra, používaná nauty knižnicou na poskytovanie nejakých štatistík ohľadom toho, čo to robí.

setword *workspace, worksize: Adresa a dĺžka poľa používaného pre pracovné skladovanie. Odporúčaná hodnota worksize je $\geq 50m$.

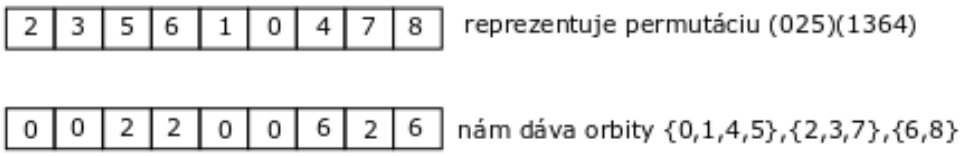
int m: Celé číslo m, také aby platilo $WORDSIZE * m \geq n$. Štandardne je $WORDSIZE$ 32.

int n: Počet vrcholov v grafe.

graph alebo sparsegraph *canong: Kanonická forma grafu g, ktorú vyrobí nauty.

3.1.2 Užitočné možnosti nauty knižnice

Za pomoci Nauty knižnice si vieme pomerne ľahko zistiť orbity grafu. Permutácia vrcholov grafu je reprezentovaná ako pole celých čísel veľkosti n, kde n reprezentuje počet vrcholov grafu. i-ta položka v poli nám udáva obraz i pri danej permutácií vrcholov. Za pomoci daného pola permutácií si vieme vyrátať aj orbity vrcholov grafu. Orbity sú taktiež reprezentované poľom celý čísel veľkosti n. Kde hodnota i-teho prvku reprezentuje najmenšie číslo vrcholu v tej istej orbite ako vrchol i.



Obr. 3.2: Reprezentácia polí permutácie a orbít [1]

Rôzne možnosti sú poskytované nauty pomocou možností štruktúr. Typ optionblk sa používa pre nauty. Vo všetkých prípadoch sa dôrazne odporúča, aby sa hodnoty najprv nastavili na predvolené hodnoty pomocou jedného z poskytnutých makier a keďže my využívame riedke grafy, zvolíme si $DEFAULTOPTIONSSPARSEGRAPH$, ktoré slúži pre neorientované riedke grafy v nauty.

Teraz opíšeme nejaké options, ktoré my budeme používať:

- **boolean getcanon:** Ak je to TRUE, potom sa vytvorí graf v kanonickom tvare. Prednastavená hodnota je FALSE.

- **boolean defaultptn:** Ak je to TRUE, predpokladá sa, že všetky vrcholy v grafe majú rovnakú farbu, čiže počiatočné hodnoty parametrov *lab* a *ptn* sú ignorované. Ak je to FALSE, potom je počiatočné zafarbenie vrcholov určené podľa polí *lab* a *ptn*. Defaultne je to nastavené na TRUE.

- **boolean writeautoms:** Ak je to TRUE, do outfilu sa vypíšu generátory grúp automorfizmu. Ak nie je definovaný outfile za pomoci FILE *outfile: tak v tom prípade je to nastavené na NULL čo je ekvovalentné stdout. Formát výpisu ďalej záleží od nastavenia options cartesian a linelength. Defaultne je to nastavené na FALSE.

- **boolean userautomproc:** Slúži na volanie nami vytvorenej funkcie zakaždým keď sa nájde nový generátor grúp automorfizmu. Žiadne volania nebudú uskutočnené ak je hodnota NULL. Defaultne je to nastavené na NULL.

3.2 Ba-graph

Ba-graph je tak isto ako nauty grafová knižnica, ktorá je neustále aktualizovaná. Ponúka viacero možností, čo sa tam dá robiť v súvislosti s grafmi. Je písaná v jazyku C++. Grafy sa v tejto knižnici vytvárajú pomerne jednoducho a je tam viac spôsobov ako nejaký graf vygenerovať. Existujú v nej aj rôzne funkcie ako nejaký typ grafu vygenerovať, ako príklad môžeme uviesť funkcie, ktoré slúžia na vygenerovanie prázdneho alebo kompletného grafu. Na to aby sme vedeli pracovať s touto knižnicou, treba sa oboznámiť so základnými dátovými typmi, ktoré táto knižnica ponúka.

- **Graph:** reprezentuje graf.

- **Vertex:** reprezentuje vrchol, novo vytvorený vrchol nemusí byť nutne naviazaný na nejaký graf

- **Edge:** reprezentuje hranu, ktorá spája 2 vrcholy, opäť ako pri vrchole, hrana nemusí byť naviazaná na graf

- **Id:** každý vrchol a každá hrana má svoje Id

- **Number:** keď pridávame vrchol do grafu, musíme tomuto vrcholu priradiť unikátne označenie v rámci grafu

- **Location:** reprezentuje hranu grafu, definovanú pomocou unikátnych identifikátorov vrcholov grafu

- **Incidence :** reprezentuje hranu grafu, definovanú pomocou vrcholov priradených grafu

- **Rotation:** reprezentuje priradenie vrcholu do grafu, štruktúra spája objekt typu Vertex s objektom typu Number, súčasne obsahuje informáciu o incidenciách vrcholu s ostatnými vrcholmi.

Taktiež si ešte popíšeme viaceré funkcie, ktoré budeme využívať. Pre nás najdôležitejšia funkcia, ktorú budeme v programe využívať, a ktorú knižnica ba-graph ponúka sa nazýva **canonical_sparse6**. Je to funkcia, ktorá na vstupe dostáva graf vo formáte ba-graph. Táto funkcia je schopná za pomoci ďalších pomocných funkcií, ako napríklad:

- **write_sparse6:** funkcia, ktorá pretransformuje graf formátu ba-graph na stringový reťazec
- **stringtosparsegraph:** funkcia, ktorá zo stringového reťazca dokáže vytvoriť graf typu sparsegraph

prerobiť graf z formátu ba-graph do formátu typu sparsegraph. Následne vytvorí kanonickú formu tohto novovzniknutého grafu. Pomocou funkcie **sgtos6** pretransformuje kanonickú formu sparsegrafu do formátu stringového reťazca, ktorý aj posiela na výstup.

Ďalšia významná funkcia je **automorfizmus**. Funkciu sme implementovali do súboru **automorphism_group.hpp**. Táto funkcia na vstupe dostane graf vo formáte ba-graph a za pomoci nauty knižnice si na výstup vrátime vektor generátorov grupy automorfizmov pre daný graf na vstupe. Posledná funkcia, ktorú tu spomenieme je funkcia **Orbity**. Ako nám už názov napovedá, táto funkcia dostane na vstup graf formátu ba-graph a na výstup pošle vektor orbít pre daný graf.

3.2.1 Colorizer

V knižnici Ba-graph je implementovaná verzia programu na ofarbovanie kubických grafov. Pomocou tohoto programu vieme povedať, či nejaký kubický graf je zafarbiteľný alebo nie. Túto funkcionalitu nám zabezpečuje funckia **is_colorable**. Daná funkcia pri farbení vrcholov G pracuje s využitím algoritmu [13]. Pre vrcholy používa farby 0,1 a 2. Vstupom algoritmu je nejaká vhodná cestová dekompozícia grafu G, ktorá predstavuje bijektívny obraz hrán grafu E(G). Konštrukciu dekompozície pre potreby nášho algoritmu popíšeme v kapitole 5. Algoritmus pri svojej práci využíva a na výstupe poskytuje dve dátové štruktúry. Prvou dátovou štruktúrou je usporiadané pole $Order$, ktoré reprezentuje poradie vrcholov grafu v druhej časti výstupnej štruktúry. Druhá štruktúra predstavuje špeciálne konštruované bitové pole $ColoringBitArray$, kde jednotlivý bit predstavuje existenciu farbenia zodpovedajúcemu indexu poľa. Dĺžka tohto bitového poľa je mocninou čísla 3. Ak máme v poli Order k vrcholov, potom veľkosť bitového poľa je 3^k. Indexom tohoto bitového poľa sú reprezentácie jednotlivých farbení vrcholov poľa $Order$, ktorý predstavuje postupnosť $a_1a_2...a_{k-1}a_k$, kde pre každé

$i \in \{1, ..., k\}$ nadobúda hodnoty $a_i \in 0, 1, 2$. Táto postupnosť predstavuje číslo zapísané v trojkovej sústave. Číselná reprezentácia tohto indexu $\sum_{i=1}^{k} a_i * 3^{i-1}$. Algoritmus z pohľadu pamäťovej zložitosti využíva optimalizované uloženie ColoringBitArray, ktoré spočíva v uložení poľa do 3^{k-3} slov obsahujúcich slová dĺžky 3^3. Pre získanie informácie o existencii farbenia vrcholov pre konkrétny index t je potom potrebné použiť funkciu **ColoringBitArray::index(t/27,t % 27)**.

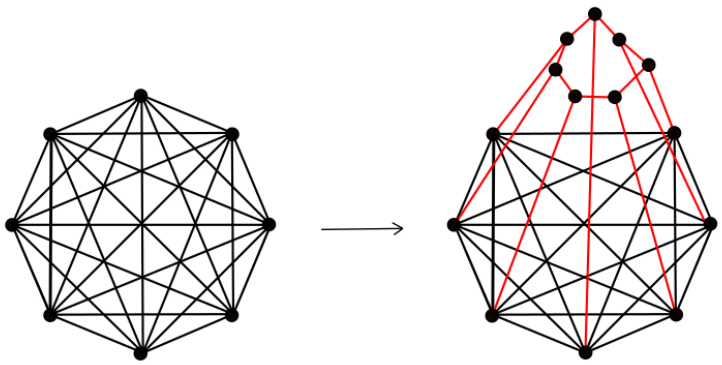
Kapitola 4

Naše generovania + výsledky

Ako sme už spomínali v predchádzajucej kapitole existuje viacero algoritmov na to ako generovať kubické grafy a aj na to ako následne generovať snarky, my sa však týmito algoritmami nebudeme zaoberať. Kľúčom k efektívnemu generovaniu grafov je efektívne detegovanie izomorfných štruktúr vytvorených počas generovania. To zabezpečujeme za pomoci viacerých postupov, ako napríklad generovania grupy automorfizmov alebo kanonickej redukcie. V priebehu tvorby diplomovej práce sme sa dopracovali k viacerým verziám programu a v tejto kapitole ich postupne popíšeme.

4.1 Prvá verzia

Každý potenciálny kontrapríklad na Jaegerovu hypotézu musí mať veľa nepárnych cyklov, najjednoduchšie 7-cyklov. Jeden zo spôsobov ako takéto grafy generovať, je zobrať 7-regulárny graf a nahradiť vrcholy tohoto grafu cyklami dĺžky 7. Ako také postupné generovanie vyzerá môžeme vidieť na obrázku 4.1, na začiatku máme graf K_8, následne nahradíme jeden z jeho vrcholov, kružnicou dĺžky 7. Postupne takto nahradíme každý vrchol pôvodného grafu K_8 až nakoniec dostaneme kubický graf.



Obr. 4.1: Nahradenie vrcholu z K8 cyklom dĺžky 7

Pri prvej verzií programu sme postupovali presne tak ako, popisujeme v úvode tejto podkapitoly. Vybrali sme si jeden vrchol pôvodného grafu K_8 a ten sme nahradili kružnicou dĺžky 7. Po odstránení pôvodného vrcholu a jeho nahradením kružnicou nám už ostávalo iba skúsiť všetky možnosti, ktorými môžme túto kružnicu do nášho pôvodného grafu zapojiť. Tých možnosti je pre každú novú kružnicu presne 7!, s využitím automorfizmov kružnice je možné redukovať počet porovnaní na 6!/2. Následne sme pre každú jednu možnosť nášho novovzniknutého grafu vyrobili za pomoci funkcie **canonical_sparse6** jeho reprezentáciu kanonickej formy v stringovej podobe a postupne sme si ukladali neizomorfné možnosti do globálneho zoznamu grafov. To, že či nejaký novovzniknutý graf je alebo nie je izomorfný už s nejakým existujúcim grafom sme jednoducho overili nahliadnutím do nášho globálneho zoznamu. Ak sa kanonická forma daného grafu nenachádza v zozname, tak náš novovzniknutý graf pridáme do globálneho zoznamu a ideme overovať ďalšiu možnosť. Výsledky generovania týmto algoritmom sú uvedené v nasledujúcej tabuľke 4.1:

Tabuľka 4.1: Počet vygenerovaných grafov pri nahradení jedného vrchola

Počet nahradených vrcholov	Počet grafov	Čas
1	1	0,2 sec
2	115	0,5 sec
3	26 649	30 sec

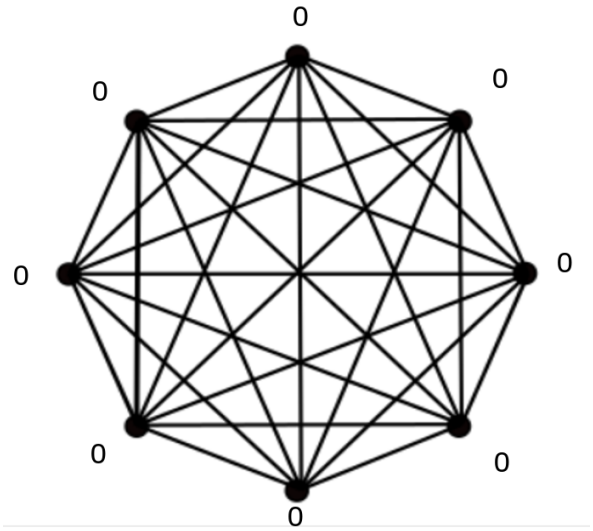
Neskôr sme však zistili, že tieto výsledky nie sú úplne správne a to preto, lebo sme skúšali vždy nahradiť iba jeden vrchol, legikograficky najbližší k už nahradeným kružniciam. Vrcholy sme nahradzovali od najmenšieho po najvačší. A ako sa ukázalo nahradením iného vrcholu nám mohli vzniknúť rôzne grafy. Preto sme sa rozhodli na každej úrovni skúsiť nahradiť postupne všetky vrcholy. Výsledky generovania sú uvedené v následujúcej tabuľke 4.2:

Tabuľka 4.2: Počet vygenerovaných grafov pri nahrádzaní každého vrchola

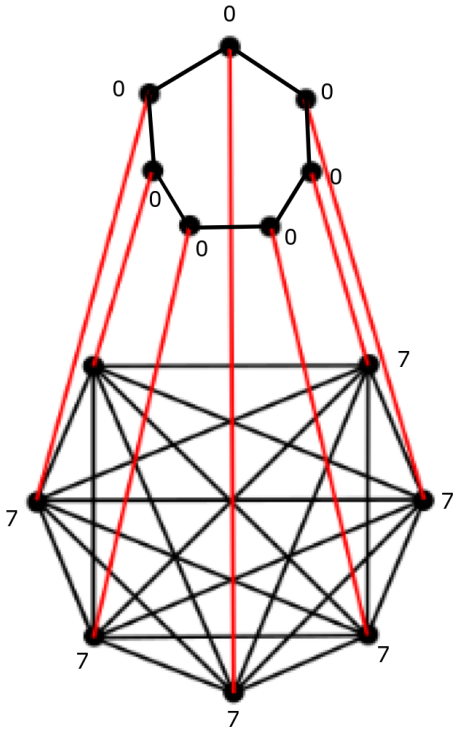
Počet nahradených vrcholov	Počet grafov	Čas
1	1	1,2 sec
2	115	3 sec
3	65 361	2 min 30 sec

Toto riešenie je časovo náročné, preto sme museli nájsť spôsob, ktorým by sme zabezpečili to, že budeme nahrádzať len vrcholy, pre ktoré to má zmysel. Vyriešili sme to tak, že ešte predtým, než sme začali nahrádzať vrcholy, pozreli sme sa na orbity vrcholov

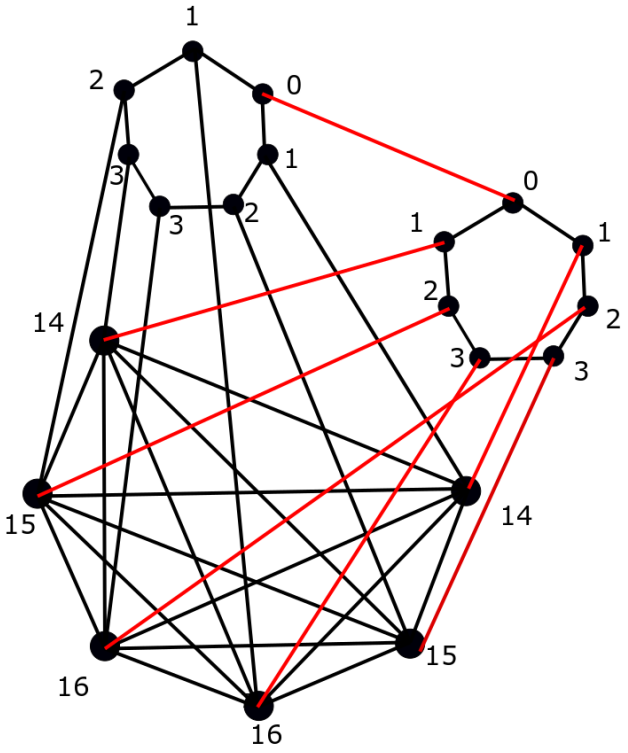
pôvodného grafu a následne sme nahrádzali len vrcholy, ktoré sa líšia svojou hodnotou v poli orbít. Ako môžete vidieť na obrázku 4.2 a 4.3, tam nám postačí vybrať si len jeden vrchol, ale ako náhle už budeme chcieť nahrádzať 3. vrchol, tak rôzne vrcholy majú rôzne orbity. 4.3



Obr. 4.2: Hodnoty orbít vrcholov grafu K8



Obr. 4.3: Hodnoty orbít vrcholov grafu K8 s jedným nahradeným vrcholom



Obr. 4.4: Hodnoty orbít vrcholov grafu K8 s dvomi nahradenými vrcholomi

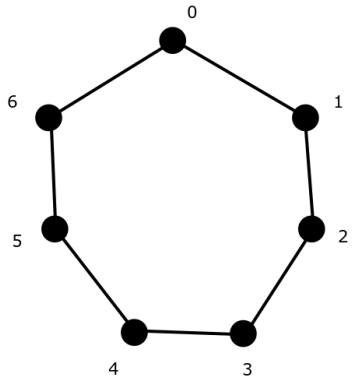
Výsledky generovania za pomoci tejto verzie sú uvedené v nasledujúcej tabulke 4.3:

Tabuľka 4.3: Počet vygenerovaných grafov pri nahradení vrcholov s rozdielnou orbitovou hodnotou

Počet nahradených vrcholov	Počet grafov	Čas
1	1	0,5 sec
2	115	1,2 sec
3	65 361	2 min 7 sec

4.2 Druhá verzia

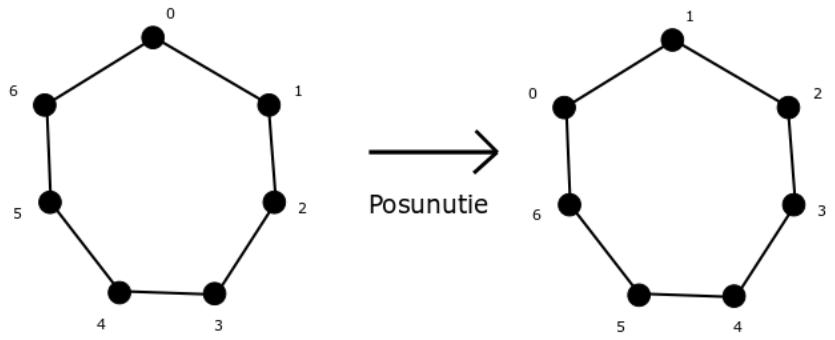
Pri druhej verzii programu sme sa začali zamýšlať nad spôsobom generovania, ktorý by nepotrebol uchovávať si globálny zoznam neizomorfných grafov. Inak povedané, chceli sme zabezpečiť, aby každý vygenerovaný graf sám o sebe vedel, či je dobrý alebo nie. Tento problém sme sa pokúsili vyriešiť už spomínanými postupmi a to konkrétne generátorom grupy automorfizmov a kanonickou redukciou, ktorú popíšeme neskôr. Pri tejto verzií programu taktiež upravujeme nahrádzanie vrcholov grafu $K8$. Vrchol grafu $K8$ nenahrádzame kružnicou dĺžky 7, ale siedmimi vrcholmi. Následne budeme zisťovať všetky možnosti akými môžme pospájať týchto 7 vrcholov do kružnice bez toho, aby sme vygenerovali nejaký izomorfný graf. Na úplnom začiatku, ešte predtým než spustíme program na generovanie, ktorý voláme funkciou **generateGraphs** si vytvoríme takzvaný pomocný graf. Tento graf nám bude slúžiť na to, aby sme zistili, ako môžeme pospájať novopridané vrcholy do kružnice bez toho, aby nám tam vznikol nejaký automorfizmus. Náš pomocný graf bude na začiatku obsahovať 7! vrcholov. To sa rovná počtu spôsobov akým vieme pospájať vrcholy do kružnice dĺžky 7. Každý jeden vrchol nášho pomocného grafu bude reprezentovať možnosť, ako môžeme pospajať vrcholy v kružnici dĺžky 7, ináč povedané vrchol pomocného grafu reprezentuje permutáciu vrcholov kružnice. Kružnica na obrázku 4.5 reprezentuje permutáciu vrcholov {0123456}.



Obr. 4.5: Kružnica dĺžky 7

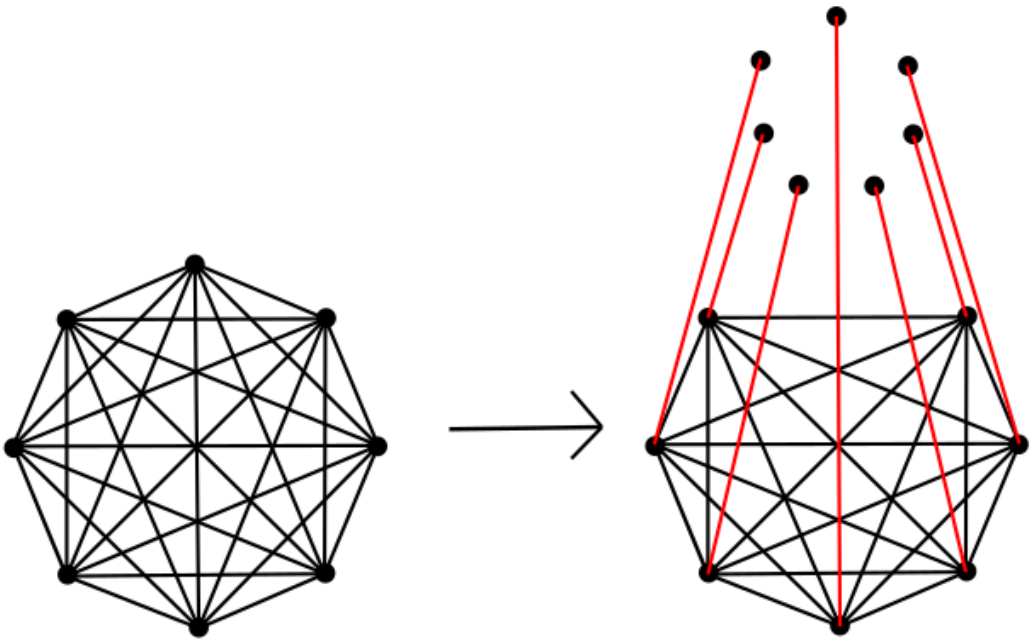
Ďalej chceme dopredu vylúčiť možnosti, o ktorých vieme, že budú automorfné. Samotná kružnica dĺžky 7 obsahuje vždy presne 14 automorfizmov. Týchto 14 automorfizmov reprezentujú posunutia a reverz vrcholov. Na obrázku 4.6 môžeme vidieť, ako také posunutie vrcholov vyzerá.

Odstránenie týchto možnosti vykonávame nasledovným posupom. Postupne budeme iterovať cez všetky vrcholy nášho pomocného grafu. Ak sa za pomoci kombinácií posunutí a reverzov vieme dostať na nejaký iný vrchol pomocného grafu, tak tento vrchol z grafu vymažeme. Čiže, napríklad z prvého vrcholu pomocného grafu, konkrétne



Obr. 4.6: Posunutie vrcholov

z vrcholu 0123456, ktorý reprezentuje nasledovné pospájanie vrcholov v kružnici, 4.5, sa vieme za pomoci posunutia dostať na vrchol 1234560, pretože pre tieto dva vrcholy platí to, že jeden vznikne posunutím druhého. A ako už vieme, skladaním automorfizmov dostaneme opäť automorfizmus. Vďaka tomu vieme povedať, že napríklad aj vrchol 5432106 musíme ostrániť, pretože tento vrchol vznikol viacnásobnou kombináciou aplikovania posunutia na vrchol 0123456. Po správnom odstránení všetkých takýchto vrcholov, dostaneme graf, v ktorom sa nachádza už iba 7!/14 čiže 360 vrcholov. Po úspešnom vytvorení a zredukovaní pomocného grafu, môžeme začať s generovaním grafov. Na začiatku dostaneme graf a v ňom si vyberieme jeden vrchol, ktorý nahradíme siedmimi vrcholmi. Na obrázku 4.7 môžeme vidieť ako takéto nahradenie vyzerá.



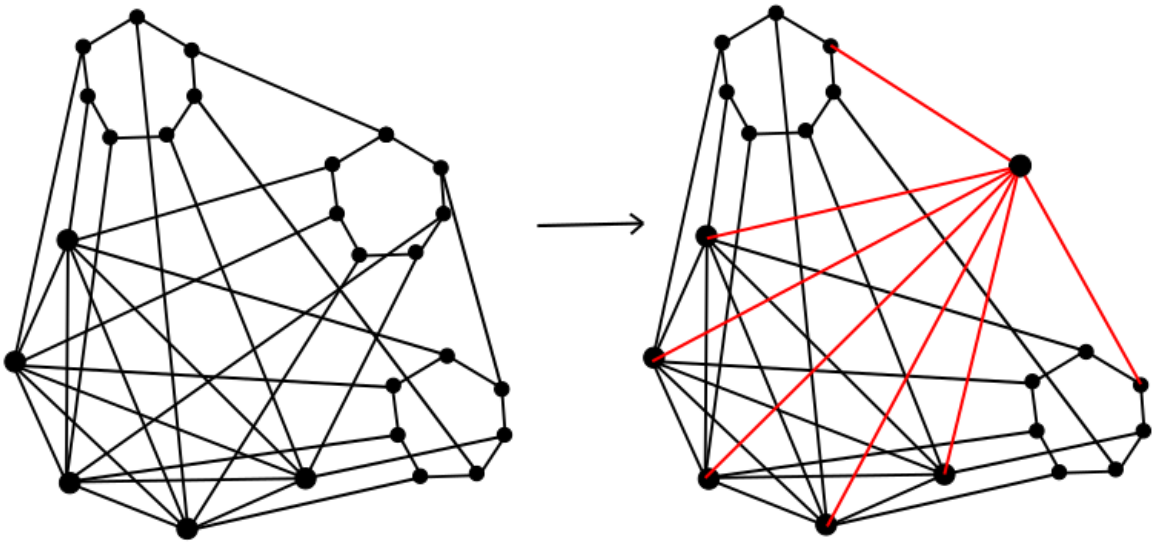
Obr. 4.7: Nahradenie vrcholu z K8 siedmimi vrcholmi

Tentoraz nemusíme skúšať všetky spôsoby, akými môžeme pospájať vrcholy grafu s novopridanými siedmimi vrcholmi, pretože budeme riešiť všetky možnosti akými môžeme pospájať dané vrcholy do kružnice. V ďalšom kroku si pre tento graf zistíme jeho generátory grupy automorfizmov. To zabezpečujeme nami implementovanou funkciou za pomoci nauty knižnice v súbore **automorphism_group.hpp**. Ak zistíme, že nami vytvorený graf obsahuje nejaké generátory, tak budeme postupovať nasledovne. Prekopírujeme si náš globálny pomocný graf do lokálne novovytvoreného pomocného grafu. Ďalej, keď budeme spomínať pomocný graf, budeme mať na mysli už iba lokálnu verziu. Ako budeme ďalej postupovať ukážeme v nasledujúcej časti kódu.

```
std::vector<int> verticesPokus;
std::vector<int> verticesPermuted;
int vrcholKam, pom;
std::vector <std::vector<int>> generatory = automorfizmus(G3);
for (const auto &permutaciaPokus : generatory) {
    for (auto &r : pomocnyGraf) {
         verticesPokus = mapa[r.n().to_int()];
         verticesPermuted.clear();
         for (auto v : verticesPokus) {
            verticesPermuted.push_back(permutaciaPokus[v + (vrchol *
               7)] - (vrchol * 7)  );
          }
         vrcholKam = mapaRev[verticesPermuted];
         pom = 0;
         for (auto &i : pomocnyGraf[r.n()]) {
             if (i.r2().n().to_int() == vrcholKam) {
                 pom = 1;
              }
          }
         if (pom == 0 && vrcholKam != r.n().to_int()) {
             addE(pomocnyGraf, Location(r.n().to_int(), vrcholKam));
          }
    }
 }
```

Postupne budeme iterovať cez každý generátor. V cykle budeme iterovať cez všetky vrcholy nášho pomocného grafu. Vypočítame si automorfný obraz vrcholu a ak medzni nimi neexistuje hrana, tak tieto 2 vrcholy spojíme hranou. To nám opäť zníži počet možností, ktorými môžeme pospájať vrcholy v kružnici. Po aplikovaní všetkých generátorov, si na náš pomocný graf zavoláme funkciu **ConnectedComponents**, ktorá pracuje na princípe prehľadávania do hĺbky. Na začiatku si v tejto funckií inicializu-

jeme všetky vrcholy pomocného grafu ako nenavštívené. Následne budeme iterovať cez všetky vrcholy grafu. Ak je nenavštívený, tak tento vrchol si vložíme do výstupného zoznamu vrcholov. Ďalej si z daného vrcholu spustíme prehľadávanie do hĺbky a všetky vrcholy, do ktorých sa dostaneme označíme za navštívené. Výstupom funkcie bude zoznam vrcholov, ktoré reprezentujú nesúvislé komponenty pomocného grafu. V tomto výslednom zozname sa nachádzajú vrcholy, ktoré reprezentujú všetky možné spôsoby akým môžeme pospájať vrcholy do kružnice tak, aby nami novovzniknutý graf nebol automorfný a tým pádom aj izomorfný s nejakým iným grafom, ktorý vygenerujeme z toho istého pôvodného grafu. Následne sme sa pokúsili aplikovať metódu kanonickej redukcie. Skúsili sme metódu, pri ktorej sme si pre náš vytvorený graf vypočítali hodnoty orbít vrcholov jeho kružníc. Pokiaľ nami posledná pridaná kružnica nebola reprezentovaná lexikograficky najmenšou množinou orbít vrcholov kružnice, tak takýto graf by sme zahadzovali, lebo by to znamenalo, že taký graf má vzniknúť z iného grafu. Táto metóda nezafungovala úplne správne, pretože pri nahradzovaní tretej kružnice došlo k chybe a zahodil sa jeden graf, ktorý sa mal vytvoriť a pridali sa 2 grafy, ktoré boli izomorfné. Správne implementovanie tejto metódy spočiva v tom, že by sme pre daný graf s $k + 1$ nahradenými vrcholmi kružnicami dĺžky 7 v cykle postupne vygenerovali $k + 1$ grafov s k nahradenými vrcholmi. Jednu kružnicu týchto grafov by sme skontrahovali naspať do vrcholu. Ako vyzerá jedna možnosť vzniknutého grafu pri kanonickej redukcii môžeme vidieť na obrázku 4.8.



Obr. 4.8: Graf po skontrahovaní jednej kružnice späť do vrcholu

Pre každý takýto graf by sme následne museli vyrobiť jeho kanonickú formu, ktorú by sme si zapamätali pre jednoduchosť porovnávania v jeho stringovej forme. Ak by sme po skontrahovaní našej poslednej pridanej kružnice nedostali lexikograficky najmenšiu stringovú hodnotu, tak takýto graf by sme vyhlásili za zlý a zahodili by sme ho. Bohužial toto riešenie pri tejto forme generovania nie je časovo efektívne, pretože by sa tam veľa krát volala funkcia **canonical_sparse6** na vytvorenie kanonickej formy, ktorá nie je úplne triviálna a výpočet stojí nejaký čas. Preto sme sa rozhodli túto myšlienku v tomto prípade zahodiť a tým pádom sme boli nútení ponechať si globálny zoznam neizomorfných grafov na kontrolu. Týmto spôsobom generovania sme sa dostali až na nahradenie 4. vrcholu. Nahradenie 5. vrcholu týmto spôsobom nevyzerá ďalej reálne, najmä kvôli veľkému počtu neizomorfných grafov, ktoré by sme museli mať uložené v pamäti a eventuálne by sme presiahli limit vnútornej pamäte počitača a generovanie by sa buď výrazne spomalilo alebo by predčasne skončilo a ak by sa nám aj podarilo tento problém s globálnym zoznamom odstrániť kanonickou redukciou za cenu výrazného spomalenia, generovanie by trvalo príliš dlho. Výsledky generovanie pomocou tejto verzie sú uvedené v nasledujúcej tabuľke 4.4:

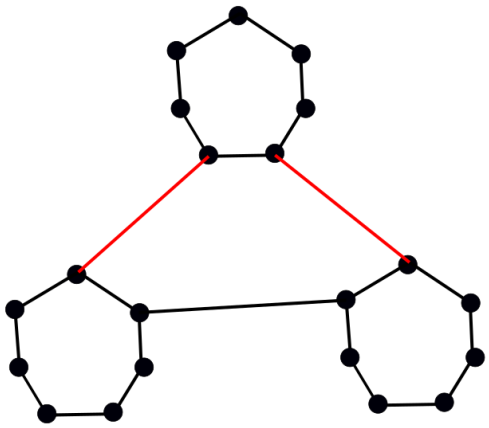
Tabuľka 4.4: Počet vygenerovaných grafov s použitím generátorov grúp automorfizmov

Počet nahradených vrcholov	Počet grafov	Čas
1	1	0,1 sec
2	115	0,3 sec
3	65 361	50 sec
4	29 173 290	18 hod

4.3 Tretia verzia

Po zistení výsledkov z predchádzajúcej verzie programu pri nahradní 4. vrcholov sme usúdili, že bude najlepšie pozmeniť náš prístup ku generovaniu grafov a ako dosiahnuť naš cieľ overenia Jaegerovej hypotézy. Zvolili sme si teda celkom nový postup generovania. Generovanie tentokrát nebudeme začínať s grafom $K8$, ale s obyčajnou kružnicou dĺžky 7 ku ktorej postupne budeme pripájať dalšie takéto kružnice. Dokonca môžeme začínať rovno aj dvomi kružnicami, ktoré sú spojené hranou, pretože vďaka automorfizmom kružníc vieme, že existuje len jeden neizomorfný graf takéhoto typu. Ako teda može vyzerať krok v našom generovaní, môžeme vidieť na obrázku 4.9 .

Výzvou pri tomto postupe generovania bolo nájsť spôsob ako efektívne zistiť všetky možné kombinácie, akým možme zapojiť novopridanú kružnicú k pôvodnému grafu. Výhodou pri tomto type generovania je to, že zo začiatku nám počet grafov bude rásť



Obr. 4.9: Jedna z možností ako zapojiť kružnicu k dvom kružniciam

oveľa pomalšie ako pri predchádzajúcom postupe. Eventuálne ale, ak by sme sa dostali až na zapojenie ôsmej kružnice a vygenerovali by sme všetky neizomorfné grafy, dostali by sme rovnakú množinu grafov, ako keby sme sa pri predchádzajúcom postupe dostali po nahradenie všetkých vrcholov grafu $K8$. Keďže pri predchádzajúcej verzií, náš postup s využitím generátorov grupy automorfizmov zafungoval a dosiahli sme vďaka nemu výrazné zrýchlenie, rozhodli sme sa tento postup využiť znova. Na začiatku si pre graf na vstupe musíme vytvoriť nový pomocný graf veľkosti 7^n, kde n = počet už zapojených kružníc, ktorý reprezentuje usporiadané k-tice vrcholov jednotlivých kružníc 1 až k, ku ktorým budeme pripájať vrcholy novej kružnice $k+1$. Pre jednoduchosť, tieto usporiadané k-tice reprezentujeme ako čislo zapísanné v 7-čkovej sústave. Z tohto pomocného grafu musíme vylúčiť vrcholy, ktoré už sú použité pri spojení kružníc 1 až k. Ako také niečo robíme si ukážeme v nasledovnom kóde.

```
Graph pomocnyGraf(empty_graph(power(7, stupen)));
int vrchol;
int pomoc;
std::vector<int> pole;
std::vector<int> pomocnePole;
    for (auto ii : G.list(RP::all(), IP::primary())) {
         if (ii->n1().to_int() / 7 != ii->n2().to_int() / 7) {
             pomocnePole.push_back(ii->n1().to_int());
             pomocnePole.push_back(ii->n2().to_int());
            }
        }
    for (auto &v : pomocnePole) {
          pomoc = 1;
          for (int i = 0; i < v / 7; i++) {
                pomoc = pomoc * 7;
            }
           for (auto &r : pomocnyGraf) {
                if ((r.n().to_int() / pomoc) \% 7 == v \% 7) {
                    pole.push_back(r.n().to_int());
                }
            }
            for (auto &i : pole) {
                deleteV(pomocnyGraf, i);
            }
            pole.clear();
        }
```

Nasledujúcim krokom je očistenie vrcholov pomocného grafu o ich automorfné obrazy. Implementácia je analogická k čisteniu automorfizmov uvedenému v predchádzajúcej verzií. Rovnako ako v predchádzajúcej verzií, po aplikácií všetkých automorfizmov si na náš pomocný graf zavoláme funkciu **ConnectedComponents**. Táto funkcia nám opäť vráti zoznam vrcholov, ktoré budú reprezentovať nezávislé komponenty pomocného grafu. Tento istý algoritmus aplikujeme aj pre samostatnú kružnicu, ktorú budeme do grafu pridávať. Následne keď už budeme mať vypočítané komponenty pomocného grafu pre graf G a komponenty pomocného grafu pre kružnicu, vieme vyskúšať všetky možnosti ako zapojiť kružnicu do nášho grafu G, bez toho aby nám z rovnakého grafu G vznikli 2 izomorfné grafy G'. Následne si pre každý graf vytvoríme jeho kanonickú formu a za pomoci globáleho zoznamu neizomorfných grafov si overíme, či sme takýto už náhodou nevygenerovali. Počas priebehu programu, ktorý mal za úlohu vygenerovať všetky grafy, ktoré vzniknú zapojením piatich kružníc sme si všimli, že generátory grupy automorfizmov sa skoro nevyskytujú, preto sme usúdili, že ak by sme chceli gene-

rovať grafy s väčším počtom zapojených kružníc ako 5, tak táto časť by nám už časovo nijako nepomáha a preto by sme ju chceli vypnúť. Výsledky generovania pomocou tejto verzie sú uvedené v nasledujúcej tabulke 4.5:

Tabuľka 4.5: Počet vygenerovaných grafov s použitím generátorov grúp automorfizmov

Počet zapojených kružníc	Počet grafov	Čas
2	1	0 sec
3	10	0,2 sec
4	2 175	3,3 sec
5	6 485 412	2 hod 30 min

4.4 Štvrtá verzia

Cieľom tejto verzie bolo, zbaviť sa globálneho zoznamu, pretože opäť by nastal rovnaký problém ako vo verzií 2 a to teda, že pri velkom počte grafom by sa nám náš globálny zoznam grafov nemusel zmestiť do pamäte. Preto sme sa rozhodli zakomponovať do tejto verzie postup s kanonickou redukciou, ktorý je popísaný ďalej v tejto podkapitole. Program bude síce pomalší, ale umožní nám to sa dostať na zapojenie aj viac ako 5 kružníc. Problém, že program bude pomalší sme sa pokúsili vyriešiť paralelizáciou. Spravili sme si teda dva nové programy a to konkrétne **test_BezK8v2.cpp** a **test_bezK8citanieZoSuborov.cpp**. Prvý je rovnaký ako program vo verzií 3, s tým rozdielom, že neizomorfné grafy len hľadanej veľkosti zapisuje do textového súboru. Ukladáme neizomorofné grafy v ich sparse forme, ktorú vieme jednoducho uložiť ako string. Takto si vieme vygenerovať textové súbory so všetkými grafmi, ktoré obsahujú menej ako 6 zapojených kružníc. Druhý program pracuje nasledovne. Postupne načítava grafy z intervalu definovanom parametrami **from** a **to** v kroku podľa parametra **step** z nami určeného textového súboru parametrom **vstupnySubor**, kde je vygenerovaný zoznam grafov. Tento program slúži na vygenerovanie všetkých grafov, ktoré budu mať o jednu kružnicu viac ako majú grafy v zadanom textovom súbore. Čiže, ak by sme tomuto programu podhodili textový súbor, ktorý by obsahol všetkých 10 grafov s tromi zapojenými kružnicami. Výsledkom programu bude súbor, ktorý bude obsahovať grafy so štyrmi zapojenými kružnicami. Po načítaní nejakého grafu z textového súboru skúsime všetky možnosti, ktorými môžeme zapojiť novú kružnicu do nášho pôvodného grafu. Ďalej si za pomoci kanonickej redukcie overíme, či tento náš novovzniknutý graf je správny alebo nie. Kanonickú redukciu v tomto prípade budeme realizovať nasledovane. Z nášho novo vytvoreného grafu, ktorý obsahoval $k+1$ kružníc si v cykle skúsime odstrániť postupne každú kružnicu. Čiže dostaneme $k+1$ grafov s k

kružnicami. Následne si pre každý takýto graf za pomoci funkcie **canonical_sparse6** vyrobíme jeho kanonickú formu uloženú vo formáte stringu. To nám umožnuje ľahké porovnávanie daných kanonických foriem. V následujúcom kóde si ukážeme ako takúto operáciu vykonávame.

```
for(int i=0; i<stupen; i++){
      Graph G2(copy_identical(G1));
      pomoc = 0;
      for(int j = 0; j < 7; j++ ){
          deleteV(G2,i*7+j);
       }
       std::string ret1 = canonical_sparse6(G2);
        if(ret1 < povodnyGraf){
        pomoc++;
         break;
         }
}
```

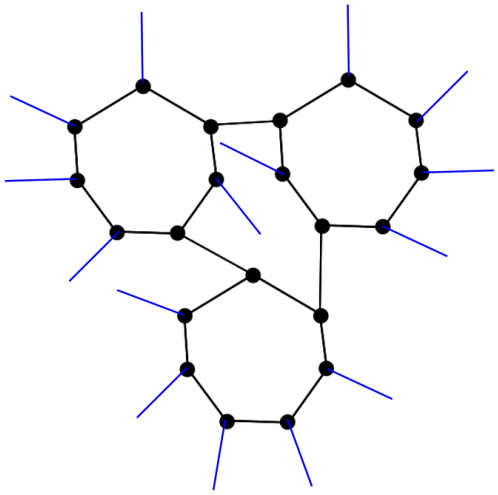
Ak náš pôvodný graf nebol lexikograficky najväčší, tak túto možnosť neakceptujeme. Táto vlastnosť kanonickej redukcie nám zabezpečí, že graf, ktorý by mohol vzniknúť z iného grafu G' neakceptujeme. Kanonická redukcia nám však negarantuje, že z grafu s k kružnicami nemôže vzniknúť viacej vzájomne izomorfných grafov s $k+1$ kružnicami a preto si musíme pamätať lokálny zoznam neizomorfných grafov pre aktuálne rozširovaný graf na vstupe. Lokálne zoznamy v tomto prípade sú malé a pamäťovo nepredstavujú vážny problém. Z dôvodu toho, že tento program je paralelizovaný neudávam tu tabuľku s výsledkami.

Kapitola 5

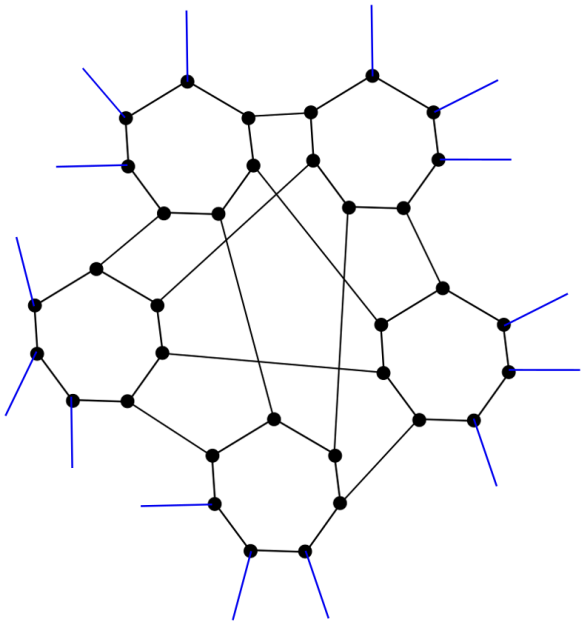
Zafarbiteľnosť

V tejto kapitole si popíšeme akým spôsbom sme vyriešili problém s ofarbovaním grafov aj keď sa nám nepodarilo vygenerovať množinu všetkých grafov, ktoré by sme dosiahli nahradením vrcholov grafu K8 kružnicami dĺžky 7. Keby sa nám to poradilo a do nejakého textového súboru by sme si odložili všetky vygenerované grafy, tak jednoducho by sme použili ofarbovač, ktorý už je implementovaný v knižnici Ba-graph. Tento ofarbovač vie povedať o kubickom grafe, či je zafarbiteľný alebo nie. A ak by sa ukázalo, že nejaký graf nie je zafarbiteľný, tým pádom by sme našli proti príklad na Jaegerovu hypotézu. Spôsob akým sme to riešili my je taký, že zobrali sme si množinu všetkých vygenerovaných grafov s piatimi zapojenými kružnicami. Tých, ako už vieme, je 6 485 112. A množinu všetkých grafov s tromi zapojenými kružnicami, tých je presne 10. Najprv si nájdeme všetky možné ofarbenia grafu. Tieto ofarbenia si nájdeme vďaka malej úprave funkcie **is_colorable** z colorizeru. Upravenú funkciu si nazveme **is_colored** a namiesto toho, aby nám funckia povedala, či je graf zafarbiteľný alebo nie, vrátime si ColoringBitArray, bitové pole všetkých ofarbení grafu. Keďže colorizer modifikuje pre svoje potreby uspriadanie vrcholov pre získanie správneho poradnia vrcholov v ColoringBitArray pracujeme aj s poľom order, ktorý generuje colorizer. Toto bitové pole bude veľkosti 3 na velkosť vektora order, ktorý obsahuje vrcholy so stupňom 1 alebo 2. V našom prípade aj grafy s piatimi zapojenými kružnicami aj s tromi zapojenými kružnicami majú rovnaký počet vrcholov stupňa 2 a to konkrétne 15. 5.1 5.2

Našou upravenou funkciou sme teda dostali bitové pole velkosti 3^{15}, ktoré reprezentuje všetky ofarbenia grafu. V tomto poli dĺžky 3^{15} sa nachádza veľa zbytočných kombinácií farbenia grafu. Tieto farbenia grafu je vhodné pre ďalši výpočet odstrániť. Pole si vieme skrátiť tým, že odstránime také farbenia, pre ktoré neplatí paritná lema. Keďže máme nepárny počet vrcholol stupňa 2, konkrétne 15, tak vieme povedať, že všetky farby sa v jednom zafarbení musia vyskytovať nepárny počet krát. Týmto skrátime naše pole približne na štvrtinovú veľkosť. Následne odstránime ešte všetky permutácie zafarbení. Čiže ak máme zafarbenie napríklad 0 1 1 2 1. Tak môžeme odstrániť



Obr. 5.1: Graf s 3 zapojenými kružnicami, kde hrany zafarbené na modro reprezentujú hrany ktorými sa môžme napojiť



Obr. 5.2: Graf s 5 zapojenými kružnicami, kde hrany zafarbené na modro reprezentujú hrany ktorými sa môžme napojiť

zafarbenia 0 2 2 1 2, 1 0 0 2 0, 1 2 2 0 1, 2 0 0 1 0 a 2 1 1 0 1. Z dôvodu optimalizácie pri testovaní naplnenia paritnej lemy si predpočítame menšiu množinu vo veľkosti 3^5, ktorú následne použijeme vo výpočte plnej množiny indexov ColoringBitArray. Zadeklarujeme si mapu **std::map<int, std::vector<unsigned char» spravneTrojice**, ktorá bude slúžiť na uchovanie iba správnych ofarbení grafov. Pod množinou správnych ofarbení budeme rozumieť množinu ofarbení redukovanú o permutácie farieb ofarbenia.

Vytvoríme si pomocné 2 rozmerné pole **poleTrojic** typu unsigned char. V nasledujúcom kóde si ukážeme ako toto pole budeme inicializovať.

```
unsigned char poleTrojic [243][8];
    for(int i = 0; i < 243; i++) {
        for (int j = 0; j < 3; j++) {
            poleTrojic[i][j] = 0;
        }
        int pom = i;
        for (int j = 0; j < 5; j++) {
            poleTrojic[i][7 - j] = pom \% 3;
            poleTrojic[i][pom \% 3]++;
            pom /= 3;
        }
    }
```

Týmto spôsobom máme v našom 2 rozmernom poli poleTrojic uložené všetky ofarbenia grafov, ktoré majú 5 vrcholov stupňa 2 alebo 1. Kombináciou troch takýchto 2 rozmerných polí poleTrojic získame všetky možné kombinácie ofarbenia grafov, ktoré maju 15 vrcholov stupňa 2 alebo 1. Pri spájaní polí robíme parity check, čiže vyhodíme všetky ofarbenia, ktoré nespĺňajú paritnú lemu. V mape spravneTrojice ukladáme vektor ofarbenia jednotlivých vrcholov. Tieto vektory následne použijeme pri identifikácií správneho indexu do ColoringBitArray pre permutované poradie vrcholov.

```
std::vector<unsgined char> hodnota(15);
for(int i=0; i<243; i++){
    for(int j=0; j<243; j++){
        for(int k=0; k<243; k++){
 if(((poleTrojic[i][0]+poleTrojic[j][0]+poleTrojic[k][0]) & 1) == 1 &&
    ((poleTrojic[i][1]+poleTrojic[j][1]+poleTrojic[k][1]) & 1) == 1 &&
    ((poleTrojic[i][2]+poleTrojic[j][2]+poleTrojic[k][2]) & 1) ==1){
                for(int l=0; l<5; l++){
                    hodnota[l] = poleTrojic[k][7-l];
                    hodnota[l+5] = poleTrojic[j][7-l];
                    hodnota[l+10] = poleTrojic[i][7-l];
                }
                spravneTrojice[((i*243 + j)*243 + k)] = hodnota; }}}}
```

V mape **spravneTrojice** teraz máme na správných pozíciach iba farbenia, ktoré spĺňajú paritnú lemu. Ďalej budeme chcieť odstrániť všetky permutácie ofarbení. Ako sme to zabezpečili si ukážeme v následujúcom kóde. Na začiatku si vytvoríme pomocné pole typu unsigned char, ktoré si naplníme všetkymi permutáciami farieb.

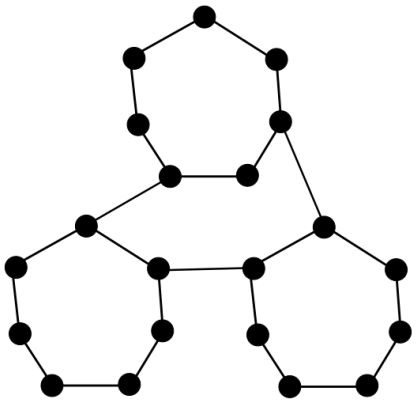
```
unsigned char pomocnePole [6][3] =
   {{0,1,2},{0,2,1},{1,0,2},{1,2,0},{2,0,1},{2,1,0}};

for(auto const& i : spravneTrojice){
    for(int j=1; j<6; j++){
         int pom=0;
         for (int k=0; k<15; k++){
                pom = (pom << 1) + pom;
                pom = pom + pomocnePole[j][i.second[14-k]];
            }
            spravneTrojice.erase(pom);
        }
    }
```

V mape spravneTrojice teraz máme namapované len správne farbenia a nevyskytujú sa nám tam žiadne zbytočné kombinácie farbení. Počet týchto farbení je 597871 čo je zhruba 1/24 z celkového počtu 3^{15} farbení. Z ohľadom na skutočnosť, že každé farbenie grafu s pätnástimi vrcholmi stupňa 2 alebo 1 vracia rovnaké bitové pole reprezentujúce všetky kombinácie farbenia vrcholov, je možné redukciu reprezentovanú mapou **spravneTrojice** vypočitať len raz a uchovať pre opakované použitie.

Pri uvažovaní o možnostiach overenia zafarbiteľnosti spojených grafov s piatimi zapojenými kružnicami a tromi zapojenými kružnicami prichádzalo do úvahy niekoľko možností implementácia. Prvá úvaha spočívala v prístupe nájsť všetky kombinácie spojení vrcholov stupňa 2 na jednej a na druhej strane. Týchto kombinácií je $3!^5$ * $5!^3$ čo predstavuje viac ako 13 miliard možností, akými môžeme 2 grafy z týchto množín spojiť. Nakoľko výpočet takéhoto objemu je nad hranicami našich výpočtových možností, rozhodli sme sa pre overenie špecifického prípadu. Pôvodne riešený problém sme transformovali na problém zisťovania $\mathbb{Z}_2$x$\mathbb{Z}_2$ nikde-nulového toku pre graf $K8$ s piatimi nahradenými vrcholmi cyklami dĺžky 7.

5.1 Overenie $\mathbb{Z}_2$x$\mathbb{Z}_2$ nikde-nulového toku

V tejto časti nebudeme pokračovať v skúmaní hranovej 3-zafarbiteľnosti, ale pozornosť venujeme $\mathbb{Z}_2$x$\mathbb{Z}_2$ nikde-nulovému toku pre graf $K8$ s piatimi nahradenými vrcholmi cyklami dĺžky 7. Nakoľko náš ofarbovač nefunguje na vrcholoch vyššieho stupňa rozhodli sme sa problém transformovať na problém, kde každý z vrcholy so stupňom 7 nahradíme kružnicou so siedmimi vrcholmi. Na základe tohto predpokladu, môžeme využiť implementáciu vychádzajúcu zo predvypočítaného zoznamu grafov s piatimi kružnicami na jednej strane a zoznamom grafov s tromi zapojenými kružnicami na druhej strane. Pre overenie $\mathbb{Z}_2$x$\mathbb{Z}_2$ nikde-nulového toku nám postačuje vyskúšať jeden vhodne vybraný graf s tromi zapojenými kružnicami. Pre naše overenie sme si vybrali ten z grafov, ktorý obsahuje 7 cyklus (v našom súbore graf na riadku 2). 5.3



Obr. 5.3: Vybraný graf s tromi zapojenými kružnicami, ktorý obsahuje 7 cyklus

Toky sa zachovávajú vzhľadom na kontrakciu, čiže ak máme tok a zkontrahujeme cyklus do vrholu, tak budeme mať znova tok. Výsledkom nášho skúmania môžu byť dva výsledky, prvým výsledkom, v prípade, že ak nenájdeme hranovú 3-zafarbiteľnosť pre nejakú možnosť spojenia grafov, tak sme našli graf, ktorý nie je 3-zafarbiteľný a tým pádom sme vyriešili pôvodný problém. Druhým výsledkom je, že v prípade, ak pre každý graf s piatimi kružnicami nájdeme hranovú 3-zafarbiteľnosť, tak sme dokázali, že pôvodný graf $K8$ s piatimi nahradenými vrcholmi kružnicami dĺžky 7 má nikde-nulový $\mathbb{Z}_2$x$\mathbb{Z}_2$ tok.

Pre identifikáciu spojení pre grafy s piatimi zapojenými kružnicami a tromi zapojenými kružnicamisme sme na oboch stranách zaviedli očíslovanie vrcholov stupňa 2 od 0 po 14. Zmysluplné kombinácie na jednotlivých stranách grafu spočívajú v prepojení kružnice s rôznymi kružnicami na druhej strane. Týmto sme zistili, že optimálnejšie pre nájdenie všetkých kombinácií bude permutovať pripojenia do kružníc s tromi vrcholmi stupňa 2, ktorých je $(3!)^5$. Skutočnosť, že testujeme rádovo 8 tisíc možností permutácií očíslovania vrcholov 0 až 14 nám dáva priestor predpočítať všetky permutácie

zafarbenia grafu s tromi zapojenými kružnicami. Predpokladaná pamäťová náročnosť je: počet permutácií * 3^{15} / 24 bitov pre jeden graf. Vzhľadom na uloženie bitového pola do 64 bitových slov, to predstavuje 7776 * 9432 * 8bytov. Predpočítanie týchto všetkých ofarbení pre všetky permutácie umožní následné sekvenčné alebo paralelné overovanie grafov s piatimi zapojenými kružnicami, pre ktoré už žiadne permutácie nebudeme musieť počítať.

Pre úspešnú aplikáciu vyššie uvedených predpokladov aplikujeme pre graf pozostávajuci z piatich kružníc očíslovanie vrcholov 0,1,2 pre vrcholy prvej kružnice grafu 3,4,5 pre vrcholy druhej kružnice grafu ... po vrcholy 12,13,14 pre vrcholy piatej kružnice (realizuje to funkcia graphToOrder). Pre graf pozostávajúci z troch kružníc aplikujeme očíslovanie vrcholov 0,3,6,9,12 pre vrcholy prvej kružnice, 1,4,7,10,13 pre vrcholy druhej kružnice a 2,5,8,11,14 pre vrcholy tretej kružnice (realizuje graphToOrder2).

```
std::vector<std::pair<int, int>> graph_to_order (Graph &G){

    std::vector<std::pair<int, int>> order;
    std::map<int,int> prechodovaMapa;
    int vrcholVkruznici = 0;
    int pocet3 =15;
    for (auto &v : G){
        if(v.degree() == 2){
            prechodovaMapa[v.n().to_int()] = vrcholVkruznici;
            vrcholVkruznici ++;
        } else {
            prechodovaMapa[v.n().to_int()] = pocet3;
            pocet3++;
        }
    }
    for (auto ii : G.list(RP::all(), IP::primary())) {

        order.push_back(std::pair( prechodovaMapa[ii->n1().to_int()],
           prechodovaMapa[ii->n2().to_int()]));
    }
    return order;
}
```

```
std::vector<std::pair<int, int>> graph_to_order2 (Graph &G){
    std::vector<std::pair<int, int>> order;
    std::map<int,int> prechodovaMapa;
    int vrcholVkruznici = 0;
    int pocet3 =15;
    for (auto &v : G){
        if(v.degree() == 2){
            prechodovaMapa[v.n().to_int()] = vrcholVkruznici/5 + (
               vrcholVkruznici\%5)*3;
            vrcholVkruznici ++;
        } else {
            prechodovaMapa[v.n().to_int()] = pocet3;
            pocet3++;
        }
    }
    for (auto ii : G.list(RP::all(), IP::primary())) {

        order.push_back(std::pair( prechodovaMapa[ii->n1().to_int()],
           prechodovaMapa[ii->n2().to_int()]));
    }
    return order;
}
```

Zmysluplné permutácie vrcholov 0 až 14 pre napojenie grafov s piatimi a tromi kružnicami je možné predpočítať a uchovať v pamäti na začiatku procesu podobne ako mapu **spravneTrojice**. Pre generovanie využijeme už vytvorené **pomocnePole** z predchádzajúcej časti programu. V následujúcom kóde môžeme vidieť ako také niečo uskutočňujeme.

```
int pomocnePolePermutacie[7776][15];

for(unsigned char i1=0; i1<6; i1++){
    int j1 = i1*6;
    for(unsigned char i2=0; i2<6; i2++){
        int j2 = (j1+i2)*6;
        for(unsigned char i3=0; i3<6; i3++){
            int j3 = (j2+i3)*6;
            for(unsigned char i4=0; i4<6; i4++){
                int j4 = (j3+i4)*6;
                for(unsigned char i5=0; i5<6; i5++){
 pomocnePolePermutacie[j4+i5][0] = pomocnePole[i1][0];
 pomocnePolePermutacie[j4+i5][1] = pomocnePole[i1][1];
 pomocnePolePermutacie[j4+i5][2] = pomocnePole[i1][2];
```

```
 pomocnePolePermutacie[j4+i5][3] = pomocnePole[i2][0]+3;
 pomocnePolePermutacie[j4+i5][4] = pomocnePole[i2][1]+3;
                ...
                ...
                ...
 pomocnePolePermutacie[j4+i5][13] = pomocnePole[i5][1]+12;
 pomocnePolePermutacie[j4+i5][14] = pomocnePole[i5][2]+12;
                    }
                }
            }
        }
    }
```

S využitím takto predpripravených dátových štruktúr pristúpime k načítaniu grafov s tromi kružnicami. Mali sme možnosť zvoliť si 2 prístupy k riešeniu. Prvý spočíval v permutovaní grafu a následnom výpočte farbenia. To predstavuje 7776 výpočtov ColoringBitArray. Druhý prístup spočíva vo výpočte ColoringBitArray pre graf na vstupe. A následné permutovanie vektoru ColoringBitArray. V tomto prípade realizujeme 1 výpočet ColoringBitArray. Výpočet ColoringBitArray vyžaduje vyššie zmienené prečíslovanie vrcholov grafu na vstupe a následné premapovanie ColoringBitArray aplikovaním návratového poradia vrcholov v poli Order na výstupe funkcie **is_Colored**. Výpočet permutácie ColoringBitArray v tomto prípade predstavuje len dalšie premapovanie pola **Order** z využitím poľa permutovaných vrcholov **pomocnePolePermutacie**. V nasledujúcom kóde je možné vidieť ako takéto niečo robíme.

```
int pocet = 0;
std::map<int,int> mapOrder;
for (auto i : pokus2.second){
      mapOrder[i.first] = pocet;
      pocet++;
       }

int poleOrder[15];

for(int i=0; i< power(6,5); i++){
    for(int j =0; j<15; j++){
        poleOrder[j] = mapOrder[pomocnePolePermutacie[i][j]];
     }
     finalneFarbenia.push_back(convertCBA(pokus2.first, spravneTrojice
        , poleOrder,spravnaMapa));
}
```

Následne si za pomoci funckie **convertCBA** premapujeme všetky ofarbenia grafu do podoby 64 bitových slov. Pre premapovanie využijeme redukovanú mapu farbení spravneTrojice. Proces premapovanie spočíva vo výpočte indexu do ColoringBitArray. Premapovanie indexu sa realizuje funkciou **premapovanieOrderu**, do ktorej ako parametre vstupujú: pole farbení pri lexikograficky najmenšom usporiadaní vrcholov, t.j v poradí 0 až 14 a mapovanie poradia vrcholov grafu na vrcholy 0 až 14. Ako vyzerá naša funkcia **premapovanieOrderu** môžeme vidieť v nasledujúcej časti kódu.

```
//power3 je pole mocnin 3
int premapovanieOrderu( std::vector<unsigned char> pole, int
    poleOrder [] ){
     int pom=0;
     for (int i=0; i<15; i++){
         pom = pom + pole[i] * polePower3[poleOrder[i]];
     }
     return pom;
}
```

Takto získame pre každé farbenie rovnaký výsledný počet 64 bitových slov (9342), ktoré reprezentujú farebenie permutovaného grafu, transformované do usporiadania vrcholov 0 až 14, ktorý uložíme do globálneho zoznamu finálnych farebení. Po kompletnom vygenerovaní globálneho zoznamu finálnych farbení môžeme začať spracovávať textový súbor s grafmi s piatimi zapojenými kružnicami. Pre každý načítaný graf, vypočítame za pomoci funckie **is_Colored** jeho ColoringBitArray, ktorý skonvertujeme rovnakým postupom funkciou ConvertCBA na 64 bitových slov farbení. Takto získané pole 64 bitových slov porovnáme so všetkými poliami 64 bitových slov vo vektore finálnych farbení **finalneFarbenie**. Za zafarbiteľné grafy považujeme tie dvojice polí 64 bitových slov, kde existuje aspoň na jednom bite súčasne hodnota 1, čo znamená, že existuje rovnaké zafarbenie vrcholov grafu s piatimi kružnicami a grafu s tromi kružnicami. Túto funkcionalitu nám zabezpečujú dve funkcie a to konkrétne **jeZafarbitelny** a **suZafarbitelne**. Ako vyzerá implementácia týchto 2 funkcií môžeme vidieť v nasledujúcih kódoch.

```
bool suZafarbitelne (unsigned long long int graf1[],  unsigned long
    long int graf2 []){
     unsigned long long int pom;
     for (int i=0; i<velkostSpravnychTrojic;i++) {
         pom = graf1[i] & graf2[i];
         if (pom != 0) {
             break;
         }
     }
     return (pom != 0);
}

std::vector<bool> jeZafarbitelny (unsigned long long int graf1 [], std
    ::vector<unsigned long long int*> &zoznamGrafov2){
     std::vector<bool> vysledok;
     for (auto graf2: zoznamGrafov2){
         vysledok.push_back(suZafarbitelne(graf1,graf2));
     }
     return vysledok;
}
```

Ak by sa v porovnávaných poliach nenašlo žiadne zhodné farbenie, znamenalo by to, že graf vytvorený spojením daných grafov s piatimi a tromi kružnicami by bol nezafariteľný a tým pádom by sme našli kontrapríklad na Jaegerovu hypotézu.

Navrhovaný a implementovaný algoritmus spracovania vyžaduje pre svoju činnosť podľa spustených testov okolo 1 GB RAM. Vytvorenie pomocných štruktúr algoritmu predstavuje približne 10s. Napočítanie 7776 redukovaných vektorov farbení pre jeden graf osciluje okolo 250s pre jeden graf s tromi kružnicami na vstupe. Nasledujúce sekvenčné spracovanie grafov s piatimi kružnicami na vzorke 10000 grafov dosahovalo priemernú rýchlosť 31s/100 grafov. Pri žiadnom z testov program nebežal ako jediný na testovacom PC. Pri orientačnom prepočte na množinu by spracovanie za predpokladu trvalo 6500000/100 * 31 (čo predstavuje okolo 24 dní). S prihliadnutím na tieto výsledky bolo nevyhnutné do algoritmu zaviesť paralelizmus. Paralelizmus je implementovaný z dvoch uhlov pohľadu. Prvý je na princípe hrubej sily, kedy rozdelíme spracovanie na viacero počítačov prostredníctvom parametrov spustenia. Tento prístup má jednu podstatnú nevýhodu a to opakovaný výpočet prípravnej fázy a množiny farbení pre 7776 grafov, čo predstavuje odhadovanú stratu 250s * počet paralelných spustení. Druhý spôsob je na princípe multithreadingu v rámci jedného počítača. V tomto prípade jednotlivé thready procesu využívajú predpočítané dáta z prípravnej fázy.

Pre implementáciu multithreadingu bola využitá konštrukcia **std::for_each(std::execution::par, aplikovaná na zoznam grafov na vstupe)**. Do tohto procesu bola implementovaná parametrizácia špecifikujúca maximálnu veľkosť fragmentu zoznamu grafov vstupujúcich do multithread výpočtu.

Parametre spustenia v poradí:

- **from3:** Začiatok intervalu spracovávaných grafov s tromi kružnicami
- **to3:** Koniec intervalu spracovávaných grafov s tromi kružnicami
- **step3:** Krok spracovania grafov s tromi kružnicami
- **vstupnySubor3:** Názov vstupného súboru s grafmi s tromi kružnicami
- **from5:** Začiatok intervalu spracovávaných grafov s piatimi kružnicami
- **to5:** Koniec intervalu spracovávaných grafov s piatimi kružnicami
- **step5 :** Krok spracovania grafov s piatimi kružnicami
- **vstupnySubor5:** Názov vstupného súboru s grafmi s piatimi kružnicam
- **paralelyzedFragment5:** Maximálna veľkosť fragmentu zoznamu grafov s piatimi kružnicami

Záver

V úvode sme si stanovili cieľ, vytvoriť algoritmus, ktorý by bol schopný generovať vhodnú podskupinu kubických grafov. A aby sme vedeli o každom nami vygenerovanom grafe povedať, či náhodou nie je kontrapríkladom na Jaegerovu hypotézu.

V diplomovej práci sa nám podarilo vyrobiť 2 algoritmy, ktoré by boli schopné vygenerovať hľadanú podmnožinu grafov. Naneštastie, tých grafov je až príliš veľa a vygenerovanie všetkých takých grafov, by bolo časovo a taktiež pamäťovo nerealizovateľné. Úspešne sa nám však podarilo upraviť program na ofarbovanie grafov, ktorý nám zabezpečuje, teoretickú možnosť vygenerovať všetky možné farbenia hľadanej množiny grafov, a tým pádom overiť Jaegerovu hypotézu. Následne sme sa rozhodli doplniť overenie špecifického prípadu súvisiaceho s riešeným problémom. Pôvodne riešený problém sme transformovali na problém zisťovania $\mathbb{Z}_2$x$\mathbb{Z}_2$ nikde-nulového toku pre graf $K8$ s piatimi nahradenými vrcholmi cyklami dĺžky 7.

Literatúra

[1] Brendan D. McKay, Adolfo Piperno. nauty and traces user's guide (version 2.6), 2016. Dostupné na `http://pallini.di.uniroma1.it/Guide.html`.

[2] Gunnar Brinkmann, Jan Goedgebeur, Jonas Hägglund, and Klas Markström. Generation and properties of snarks. *Journal of Combinatorial Theory, Series B*, 103(Issue 4):468–488, 2013.

[3] Gunnar Brinkmann, Jan Goedgebeur, and Brendan D. Mckay. Generation of cubic graphs. *Discrete Mathematics and Theoretical Computer Science*, 13(2):69–79, 2011.

[4] Chalmovianska. Automorfizmy euklidovskej roviny, 2011. `https://flurry.dg.fmph.uniba.sk/webog/SuboryOG/chalmovianska/g2/automorfizmy.pdf`.

[5] Miquel A Fiol, Giuseppe Mazzuoccolo, and Eckhard Steffen. On measures of edge-uncolorability of cubic graphs: A brief survey and some new results. *arXiv preprint arXiv:1702.07156*, 2017.

[6] Ian Holyer. The np-completeness of edge-coloring. *SIAM Journal on computing*, 10(4):718–720, 1981.

[7] Juraj Žitňanský. Návrh a implementácia vysokoúrovňového interfacu pre generovanie grafov, 2019.

[8] Martin Kochol. Snarks without small cycles. *journal of combinatorial theory, Series B*, 67(1):34–47, 1996.

[9] Edita Máčajová and Martin Škoviera. Irreducible snarks of given order and cyclic connectivity. *Discrete mathematics*, 306(8-9):779–791, 2006.

[10] Edita Máčajová. Prednášky z teórie grafov, 2021. `http://www.dcs.fmph.uniba.sk/~macajova/TG/pozn20_7.pdf`.

[11] Roman Nedela and Martin Škoviera. Atoms of cyclic connectivity in cubic graphs. *Mathematica Slovaca*, 45(5):481–499, 1995.

[12] Roman Nedela and Martin Ŝkoviera. Decompositions and reductions of snarks. *Journal of Graph Theory*, 22(3):253–279, 1996.

[13] Jakub Tetek Robert Lukoťka. A 3-edge-coloring algorithm, 2019. `https://bgw.labri.fr/2019/booklet.pdf`.

[14] Eduard Toman. Enumerácia diskrétnych štruktúri, 2011. `https://new.dcs.fmph.uniba.sk/files/texty/eds.pdf`.