UNIVERZITA MATEJA BELA V BANSKEJ BYSTRICI
FAKULTA PRÍRODNÝCH VIED

VÝPOČTOVÝ SYSTÉM PRE ŠTRUKTURÁLNU ANALÝZU GRAFOV S PODPOROU PARALELIZMU

Diplomová práca

383d4a02-425c-46cc-9d24-34186d953a38

Študijný program:	Aplikovaná informatika
Študijný odbor:	9.2.9. Aplikovaná informatika
Pracovisko:	Katedra informatiky
Vedúci práce:	doc. Mgr. Ján Karabáš, PhD.
Konzultant:	RNDr. Miroslav Melicherčík, PhD.

Banská Bystrica 2019 **Bc. Jakub Strmeň**

Univerzita Mateja Bela v Banskej Bystrici
Fakulta prírodných vied

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Jakub Strmeň
Študijný program: aplikovaná informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: aplikovaná informatika
Typ záverečnej práce: Magisterská záverečná práca
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Výpočtový systém pre štrukturálnu analýzu grafov s podporou paralelizmu

Anotácia: Snark je cyklicky 4-súvislý kubický graf obvodu aspoň 5, ktorého hrany nie je možné regulárne hranovo ofarbiť tromi farbami. Záujem o snarky je spôsobený tým, že mnoho dôležitých a ťažkých problémov diskrétnej matematiky je možné redukovať na otázky o snarkoch. V poslednom období sa pri štúdiu snarkov čoraz viac uplatňujú počítačové experimenty. Okrem iného bola počítačom vygenerovaná databáza všetkých snarkov po 36 vrcholov.

Počítačové programy sa prirodzene používajú na overovanie vybraných invariantov snarkov. Napriek relatívne veľkému počtu vygenerovaných grafov, v súčasnosti nepoznáme dostatočne dobre charakteristické vlastnosti členov triedy ani vlastnosti triedy grafov ako takej. Skôr naopak, pri štúdiu invariantov určujúcich tieto vlastnosti je nutné databázu vygenerovaných grafov prehľadávať, grafy s určenými vlastnosťami separovať podľa zadaných kritérií a ďalej študovať.

Keďže algoritmy na štrukturálnu analýzu snarkov vo svojej podstate využívajú ako podprocedúru algoritmus na určenie regulárneho hranového ofarbenia (sub)kubického grafu, ktorý je exponenciálnej zložitosti, zistenie, či zadaný graf spĺňa predložené kritériá je zvyčajne ťažký problém. Vzhľadom na počet inštancií grafu je prirodzenou voľbou použitie paralelného systému -- clustra, gridu, distribuovanej siete (bot).

Chceme preskúmať možnosti návrhu prototypov aplikácií pre paralelné počítanie vlastností: filtrov, selektorov a de/multiplexerov. Vedomosti získané z analýzy takýchto výpočtových systémov chceme využiť pri návrhu a implementácii aplikácie, ktorá bude testovať vzorku snarkov s veľkosťou viac ako 65 miliónov členov. Cieľom práce je definovať a analyzovať prototypy paralelných a/alebo distribuovaných aplikácií na riešenie zadaných problémov z teórie grafov. V ďalšom rade je dôležité implementovať aplikácie odvodené od prototypov s pracovnými názvami filter, selektor a de/multiplexer. Implementované aplikácie chceme využiť pri teste známych snarkov na tzv. "subkritické vlastnosti", čo považujeme za ďalší výstup záverečnej práce.

Vedúci: doc. Mgr. Ján Karabáš, PhD.
Katedra: KIN FPV - Katedra informatiky
Vedúci katedry: RNDr. Miroslav Melicherčík, PhD.

Univerzita Mateja Bela v Banskej Bystrici
Fakulta prírodných vied

Dátum zadania: 08.03.2018

Dátum schválenia smerovej katedry: 06.04.2018 prof. RNDr. Roman Nedela, DrSc.
garant študijného programu

Vyhlásenie

Vyhlasujem, že táto diplomová práca je mojím pôvodným autorským dielom, ktoré som vypracoval samostatne. Všetky zdroje, pramene a literatúru, ktoré som pri vypracovaní používal alebo z nich čerpal, v práci riadne citujem s uvedením úplného odkazu na príslušný zdroj.

Jakub Strmeň

Poďakovanie

V prvom rade by som chcel poďakovať svojmu vedúcemu práce doc. Mgr. Jánovi Karabášovi, PhD. za všetky rady, literatúru a starostlivé vedenie počas celého vývoja diplomovej práce. Tiež by som chcel poďakovať svojmu konzultantovi RNDr. Miroslavovi Melicherčíkovi, PhD. za pomoc pri riešení problémov aj nad rámec jeho povinností. V neposlednom rade by som chcel poďakovať svojim rodičom za trpezlivosť s mojím štúdiom a svojej budúcej manželke za podporu a pochopenie počas ukončenia štúdia a písania práce.

(Part of the) Computing was performed in the High Performance Computing Center of the Matej Bel University in Banska Bystrica using the HPC infrastructure acquired in project ITMS 26230120002 and 26210120002 (Slovak infrastructure for high-performance computing) supported by the Research & Development Operational Programme funded by the ERDF.

Práca bola čiastočne podporená z grantu VEGA 01/0487/17 Ministerstva školstva, vedy, výskumu a športu SR.

Abstrakt

Bc. Jakub Strmeň, Výpočtový systém pre štrukturálnu analýzu grafov s podporou paralelizmu, Univerzita Mateja Bela v Banskej Bystrici, Fakulta prírodných vied, Katedra informatiky, vedúci práce doc. Mgr. Ján Karabáš, PhD., Diplomová práca, Banská Bystrica, 2019, 105 strán.

Cieľom tejto práce je navrhnúť a implementovať výpočtový systém pre analýzu vlastností kubických hranovo 3-regulárne neofarbiteľných grafov s podporou paralelizmu. Pomocou implementovaného systému sme preskúmali vybrané chromatické vlastnosti na všetkých snarkoch do veľkosti 36 vrcholov. Ďalšou vzorkou pre analýzu chromatických vlastností boli snarky nepárnosti štyri o veľkosti 44 vrcholov. Výsledkom tejto práce je návrh a implementácia daného systému v jazyku C++ spolu s výsledkami analýzy chromatických vlastností vybraných grafov.

Kľúčové slová: *graf, snark, invariant, paralelný systém, chromatický index grafu, chromatické invarianty snarku.*

Abstract

Bc. Jakub Strmeň, Computational system for structural graph analysis with support of parallelism, Matej Bel University, Faculty of Natural Sciences, Department of Computer Science, supervisor doc. Mgr. Ján Karabáš, PhD., Master thesis, Banská Bystrica, 2019, 105 pages.

Main objective of the master thesis is to design and implement computational system for analysis of cubic 3-edge non colorable graphs with emphasis on efficiency using parallel computing. Using the implemented system we analyzed a set of chromatic properties of all existing snarks up to 36 vertices. We also analyzed this properties on the sample of 31 snarks of oddness four and order 44. Main outcome of the thesis is implemented system in C++ language along with results of computational analysis of mentioned graphs.

Keywords: *graph, snark, invariant, parallel computing, chromatic index of graph, chromatic invariant of snark.*

Predhovor

Grafy, ktoré sa nedajú regulárne hranovo ofarbiť tromi farbami a neobsahujú most nazývame snarky. V roku 1880 P. G. Tait nepredpokladal, že takéto grafy existujú [58], no v roku 1891 J. Petersen objavil prvý snark [53]. V predkladanej práci pojednávame o vybraných chromatických vlastnostiach snarkov, ktoré vyjadrujú mieru neofarbiteľnosti grafu [51]. Pre všetky existujúce snarky veľkosti menej ako 38 vrcholov spolu s vybranou skupinou snarkov so 44 vrcholmi budeme zisťovať danú množinu vlastností. Pri výskume týchto vlastností a pre náročnosť ich zisťovania sme nútení navrhnúť výpočtový systém, ktorý implementujeme v jazyku C++. Pre paralelizáciu výpočtov si zvolíme knižnicu MPI tiež implementovanú v jazyku C a C++. V závere práce uvádzame výsledky analýzy vlastností, ktoré sme získali.

Obsah

Zoznam obrázkov

Zoznam algoritmov

Zoznam tabuliek

Úvod

Teória grafov sa zaoberá množstvom problémov. Mnohé z nich sú vo všeobecnosti riešiteľné práve vtedy, ak sa dajú vyriešiť pre kubické grafy. Kubické grafy sú zvyčajne vzhľadom na stupeň grafu prvou netriviálnou inštanciou pre tieto problémy, ktorou sa má zmysel zaoberať. Pre ďalšie zjednodušenie môžeme pri mnohých takýchto problémoch ohraničiť triedu skúmaných grafov na kubické, ktoré nemožno hranovo regulárne ofarbiť tromi farbami. Napríklad pre mnohé problémy týkajúce sa cyklov a párovania je postačujúce dokázať ich pre snarky [17].

Tu sa dostávame k problematike, o ktorú sa matematici zaujímajú už takmer jeden a pol storočia. Ide o problém hranového farbenia kubických grafov a s ním súvisiacich taitovsky neofarbiteľných grafov. Najväčší záujem o takéto grafy vzbudil Tait práve v roku 1880, keď poukázal na ekvivalenciu existencie planárneho snarku s neplatnosťou známeho problému štyroch farieb [58]. Problém štyroch farieb spočíva v tvrdení, že regióny každej planárnej mapy vieme ofarbiť nanajvýš štyrmi farbami. Ako píše Gardner, jednoduchším spôsobom ako hľadať protipríklad pre 4-ofarbiteľnosť mapy by mohlo byť hľadanie grafu s určitými vlastnosťami. Takýto graf je práve planárny snark, ktorého definíciu uvedieme neskôr v práci [35]. Keďže veta o štyroch farbách bola pomocou počítačov dokázaná v roku 1976 vieme, že planárny snark neexistuje [13]. Keď však vypustíme planárnosť, dostaneme neprázdnu množinu grafov, snarkov, ktoré sú objektom nášho záujmu.

V prvej časti práce uvedieme stručnú motiváciu pre výskum v danej oblasti a základnú terminológiu z teórie grafov. Následne zadefinujeme pojem snark a niektoré jeho vlastnosti, popíšeme stručnú históriu a uvedieme niekoľko konkrétnych grafov danej kategórie. Ústrednou témou práce budú chromatické vlastnosti snarkov a ich výskum.

V druhej kapitole predstavíme požiadavky, potrebné algoritmy, návrh a implementáciu výpočtového systému pre analýzu vlastností snarkov, ktorý tvorí jadro práce. Systém bude nevyhnutne využívať procedúru pre hľadanie regulárneho hranového ofarbenia grafu. Hľadanie takéhoto ofarbenia je však známy NP-úplný problém, a preto sa budeme snažiť navrhnúť systém s podporou čo najväčšej miery paralelizácie výpočtov.

V závere práce opíšeme výsledky týkajúce sa vybraných chromatických vlastností snarkov dosiahnuté pomocou implementovaného systému.

Kapitola 1

Vlastnosti snarkov

1.1 Motivácia

Okrem už spomenutej a dokázanej vety o štyroch farbách existuje množstvo ďalších dôvodov, prečo sa o problematiku farbenia kubických aj všeobecných grafov a ich vlastnosti zaujímať.

Hľadanie regulárneho hranového ofarbenia grafu je však spolu s vrcholovým farbením a mnohými inými známa NP-úplná úloha [38, 42]. To znamená, že riešenie nášho problému je mimoriadne náročné, no tiež aj to, že každý úspech v riešení hranového farbenia sa môže odzrkadliť v množstve iných problémov.

Úlohou hranového ofarbenia grafu vo všeobecnosti vieme modelovať aj viaceré praktické problémy. Napríklad zápas každý s každým a minimalizácia počtu odohraných kôl [18], rozvrhnutie spojení v sieťových komunikačných protokoloch [34], rozvrhnutie svetelných frekvencií pri sieťovej komunikácii používajúcej optické vlákna [30] alebo plánovanie výrobného procesu [60]. Podrobnejšie si opíšeme nasledujúce dva:

Problém plánovania výrobného procesu je optimalizačný problém, kde sa dá využiť hranové ofarbenie grafu. V tomto prípade potrebujeme nastaviť výrobný proces čo najefektívnejšie s tým, že máme súbor predmetov, kde na výrobu každého z nich je potrebné vykonať určité operácie. Každá z týchto operácií sa môže vykonať na určitom špecifickom stroji a my sa snažíme nájsť plán výroby, kde budú stroje maximálne vyťažené a čas bude čo najkratší. Tu sa dá problém formulovať ako úloha hranovej 3-ofarbiteľnosti grafu tak, že každý výrobok bude reprezentovaný vrcholom z prvej množiny a každý stroj bude reprezentovaný vrcholom druhej množiny. Teraz už len stačí prepojiť hranou každý výrobok so strojom na ktorom je nutné vykonať operáciu pre daný výrobok. Následne hľadáme hranové ofarbenie s minimálnym počtom farieb (opäť - počet farieb = počet nutných časových/výrobných krokov). Inak povedané hľadáme hranové ofarbenie v bipartitnom grafe [60].

Rozvrhnutie spojení v sieťových protokoloch TDMA (time division multiple access) sa tiež dá uvažovať ako problém ofarbenia hrán grafu. Hľadáme rozvrhnutie spojení tak, aby každý uzol siete mohol komunikovať so svojimi susedmi bez toho aby sa rušili. Tu však autori riešenia namiesto obyčajného neorientovaného grafu používajú orientovaný graf, kde každú neorientovanú hranu (uv) nahradia dvoma orientovanými hranami (uv, vu). Následne sa pridajú dodatočné podmienky ofarbenia a pomocou heuristiky a ďalších nadväzujúcich krokov sa hľadá riešenie [34].

1.2 Základné pojmy

Keďže sa v práci budeme zaoberať problémami z teórie grafov, v úvode si zadefinujeme základné pojmy, s ktorými sa v texte budeme stretávať.

Definícia 1.1. *Graf $G = (V, E)$ je dvojica množín, kde V je neprázdna, konečná množina vrcholov a $E \subseteq [V]^2$ je množina hrán [25, 37, 52].*

Prvky množiny E sú neusporiadané dvojprvkové podmnožiny $\{u, v\}$ množiny V. Z tejto definície vyplýva, že graf je jednoduchý a neorientovaný, takže neobsahuje slučky, orientované hrany ani násobné hrany. Pre vylúčenie nejasností vždy predpokladáme $V \cap E = \emptyset$. Počet vrcholov grafu nazývame rád a označujeme $|G|$.

Vrchol v je *incidentný* s hranou e ak $v \in e$ [25]. Hrana $e = \{u, v\}$ má dva koncové vrcholy a to u a v. Dva vrcholy u a v nazveme susedné ak v grafe existuje hrana $e = \{u, v\}$. Dve hrany e_1 a e_2 nazveme susedné ak obe tieto hrany e_1, e_2 majú spoločný koncový vrchol [25]. Takúto susednosť budeme značiť $e_1 \sim e_2$. Stupňom vrcholu v označíme počet hrán, ktoré sú s týmto vrcholom v incidentné. Stupeň vrcholu značíme $deg(v)$ [37]. Maximálnym stupňom grafu G označíme maximum z $deg(v)$ všetkých vrcholov grafu G.

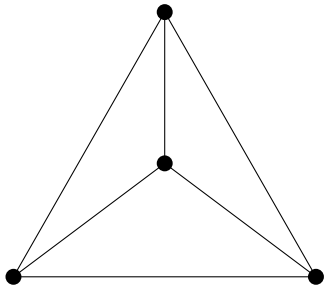
Definícia 1.2. *Kubický graf G=(V,E) je graf, v ktorom pre každý vrchol v je $deg(v) = 3$.*

V práci nás však bude zaujímať len špecifický typ grafov a to kubický graf. Kubický graf G (trivalentný, 3-regulárny, 3-graf) je teda súvislý, konečný, jednoduchý graf, kde každý vrchol je incidentný práve s tromi hranami.

Diagramom grafu G myslíme jeho grafickú reprezentáciu v ploche P pozostávajúcu z množiny bodov B a množiny čiar S, kde každý bod $b \in B$ reprezentuje vrchol $v \in V$ a každá čiara $s \in S$ reprezentuje hranu $e \in E$ [52]. Planárny graf G je taký, ktorého diagram v dvojrozmernom priestore (rovine) neobsahuje žiadne pretínajúce sa hrany. Diagram najmenšieho planárneho kubického grafu môžeme vidieť na Obr. 1.

Majme graf G=(V,E) a graf G'=(V',E'). Ak V' $\subseteq$ V a E' $\subseteq$ E, potom graf G' nazveme podgrafom grafu G. Neskôr v texte budeme tiež pracovať s grafom, ktorý je

takmer kubický. Pre každý vrchol v takéhoto grafu G=(V,E) platí $deg(v) \leq 3$ a nazveme ho *subkubický*.



Obr. 1: Príklad jednoduchého kubického grafu.

Definícia 1.3. *Párovanie M v grafe G, je množiná nezávislých hrán.*

Dve hrany *e=(u,v)* a *f=(x,y)* nazveme nezávislé práve vtedy, keď nemajú spoločný vrchol.

Definícia 1.4. *Perfektné párovanie M v grafe G je také párovanie, kde množina vrcholov $W(M)$ obsahuje všetky vrcholy $V(G)$.*

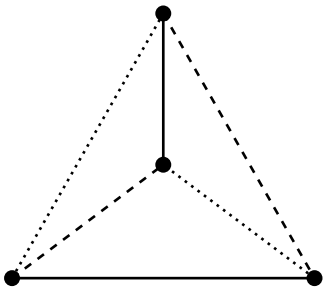
Problém hranového ofarbenia

Keď máme definovaný graf a s ním súvisiace pojmy, môžeme si postupne zadefinovať regulárne hranové ofarbenie grafu. Takéto ofarbenie je zásadnou vlastnosťou grafu a budeme sa s ním stretávať naprieč celou prácou.

Definícia 1.5. *Majme graf $G = (V, E)$ a neprázdnu konečnú množinu farieb C. Hranové ofarbenie je priradenie (zobrazenie) $\gamma : E \to C$, teda priradenie farieb hranám. Ak pre $\forall v \in V$ platí, že zafarbenie $\gamma(e_i)$ všetkých hrán e_i, kde $i = 1, ..., n$, susediacich s vrcholom v je navzájom rôzne (resp. $\forall\{e_1, e_2\} : e_1 \in E, e_2 \in E, e_1 \sim e_2 : \gamma(e_1) \neq \gamma(e_2)$), takéto zafarbenie nazveme regulárne. [37, 25]*

Regulárne hranové farbenie grafu G, kde množina farieb C obsahuje práve k-prvkov nazývame k-regulárne. Ak je graf možné hranovo 3-regulárne ofarbiť, takéto farbenie tiež nazývame *taitovské*. Príklad 3-regulárneho hranového ofarbenia kubického grafu môžeme vidieť na Obr. 2.

Definícia 1.6. *Chromatický index grafu G resp. hranové chromatické číslo je najnižšie prirodzené číslo k, pre ktoré je možné graf G k-regulárne ofarbiť [25].*



Obr. 2: Príklad regulárneho hranového ofarbenia 3-farbami (plná, čiarkovaná, bodkovaná).

Chromatický index grafu G je teda najnižší možný počet farieb, ktoré postačujú na regulárne zafarbenie grafu G. Chromatické číslo je ekvivalent chromatického indexu pre vrcholové ofarbenie grafu.

Nájdenie regulárneho hranového ofarbenia sa dá vnímať buď ako optimalizačný problém, alebo ako rozhodovací problém. V prvom prípade sa snažíme nájsť chromatický index grafu a v druhom prípade sa pre dané prirodzené číslo k rozhodujeme, či daný graf je k-regulárne ofarbiteľný.

V roku 1964 ukrajinský matematik V.G. Vizing dokázal, že na hranové ofarbenie jednoduchého neorientovaného grafu s maximálnym stupňom Δ zaručene stačí Δ+1 farieb [59]. Rozhodnúť však, či na takéto ofarbenie grafu stačí Δ farieb alebo nie, je NP-úplný problém [38, 43]. To znamená, že nepoznáme žiadny polynomiálny algoritmus na rozhodnutie tejto otázky v grafe (a s prihliadnutím na predpklad P $\neq$ NP takýto algoritmus ani neexistuje).

Definícia 1.7. *NP-úplný problém $\mathcal{P}$ je problém, pre ktorý platí:*

1. *$\mathcal{P} \in NP$ a teda, že problém je NP-ťažký (tzn. vieme v polynomiálnom čase určiť, či x je riešením $\mathcal{P}$).*
2. *ľubovoľný iný NP-ťažký problém vieme v polynomiálnom čase redukovať na problém $\mathcal{P}$ [50].*

NP-úplnosť problému nájdenia chromatického indexu kubického grafu dokázal Holyer. Dôkaz bol formulovaný pomocou polynomiálnej redukcie problému *3SAT* na taitovské ofarbovanie [38]. *3SAT* je problém splniteľnosti boolovskej formuly v 3-konjunktívnom normálnom tvare.

V našej práci nás bude zaujímať len 3-regulárne hranové ofarbenie grafu. Preto, keď budeme tvrdiť o grafe, že je alebo nie je ofarbiteľný, budeme mať vždy na mysli práve takúto ofarbiteľnosť.

Predtým ako definujeme samotný *snark*, potrebujeme ešte definovať niektoré ďalšie vlastnosti grafu G na to aby sme o G mohli povedať, že je snarkom.

Definícia 1.8. *Obvod grafu G je dĺžka (počet hrán) najkratšieho cyklu v grafe.*

Konektivitou alebo súvislosťou grafu G nazveme minimálny počet hrán alebo vrcholov po ktorých odstránení sa zvyšok grafu G rozpadne na viac ako jeden súvislý podgraf.

Definícia 1.9. *Cyklická hranová súvislosť grafu G je definovaná ako minimálny počet hrán $\{u, v\}$ grafu G takých, že po ich odstránení z grafu sa graf rozpadne na súvislé nezávislé podgrafy (komponenty) pričom aspoň dva z nich obsahujú cyklus [26, 58]. Cyklickú súvislosť označujeme ako λ_C*

Cyklickú súvislosť prvýkrát definoval už Tait v roku 1880. Je to spolu s obvodom grafu jeden z najdôležitejších parametrov snarku. Všeobecne je v grafoch miera konektivity grafu väčšinou vyjadrená hranovou súvislosťou. Hranovou súvislosťou nazveme minimálny počet hrán, ktorých odstránenie spôsobí rozpad grafu na nezávislé komponenty. Takáto súvislosť je však pri kubických grafoch vždy rovná trom, a preto má zmysel uvažovať o spomenutej cyklickej hranovej súvislosti.

1.3 Snark

Definícia 1.10. *Snark je definovaný ako súvislý kubický graf s obvodom aspoň 4 a cyklickou súvislosťou aspoň 4, ktorý sa nedá 3-regulárne hranovo ofarbiť [35, 39, 58].*

Pojem snark v súvislosti s teóriou grafov prvýkrát použil M. Gardner [35] aby sme sa vyhli zdĺhavému pojmu „nontrivial uncolorable trivalent“. Inšpirácia pre názov *snark* pochádza z básne Lewisa Carrolla - The Hunting of the Snark. V nej Carroll slovom „Snark“ označuje fiktívneho, ťakžko nájditeľného zvieracieho tvora. Ešte v roku 1975 panoval názor, že snarky sú veľmi vzácne a ťažko objaviteľné. Ako píše Isaacs, ktokoľvek, kto ich bude hľadať „will be vividly impressed with the maddening difficulty of finding a 3-graph he cannot color“, teda bude mimoriadne ohúrený až šialenou náročnosťou nejaký nájsť [39]. V roku 1976 Gardner píše: „vieme, že snarky sa ťažko hľadajú“, čo naznačuje, že práve táto podobnosť medzi „Snarkom“ a taitovsky neofarbiteľným kubickým grafom je hlavný dôvod pre toto pomenovanie. Toto presvedčenie nakoniec nebolo ďaleko od pravdy, ako ukazuje G. Brinkmann v [17], z celkového počtu kubických grafov o veľkosti 28 vrcholov a obvodom aspoň 4 je len 0.00015% snarkov. Percento početnosti snarkov voči všetkým kubickým grafom dokonca klesá s narastajúcim počtom vrcholov.

Konkrétna definícia vlastností snarku však bola v priebehu času diskutovaná a odpoveď na otázku čo je a čo nie je triviálny snark bola pomerne dlho nejasná (pozri

[51, 19, 21, 39, 41]). Nedela a Škoviera v [51] uvádzajú, že netriviálny snark by mal mať obvod aspoň 5. Brinkmann dokonca definuje snark ako graf s cyklickou súvislosťou aspoň 5 a ak je cyklická súvislosť nižšia, jedná sa o „slabý“ snark [17].

Upresnenie týchto vlastností je podstatné z toho dôvodu, že napríklad obvod grafu nižší ako päť umožňuje výskyt triviálnych prvkov grafu ako napríklad trojuholník alebo štvorec, ktoré sa dajú jednoducho nahradiť vrcholom, resp. dvoma hranami. Alebo naopak, každý vrchol snarku sa dá nahradiť trojuholníkom a vznikne tak väčší snark s obvodom tri. Takéto snarky sú len jednoduchými variáciami menších snarkov, z ktorých vznikli [51, 35].

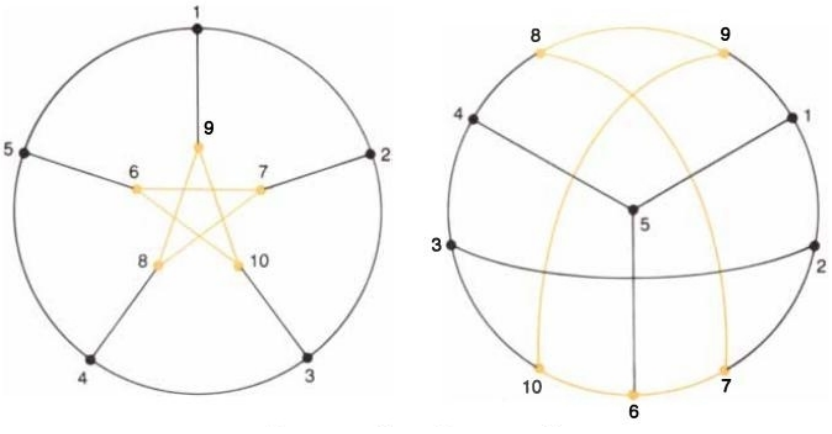
1.3.1 Známe snarky

Prvý známy snark, Petersenov graf, bol objavený v roku 1891 a má 10 vrcholov. Na ďalší snark sa čakalo vyše 50 rokov. V roku 1946 D. Blanuša objavil dva 18-vrcholové snarky, dnes pomenované po ňom a v roku 1948 B. Descartes objavil ďalší. Po nich sa nasledujúci objavil až v roku 1973, nájdený G. Szekeresom. V roku 1975 Isaacs pomocou vyššie uvedených, dovtedy jediných známych snarkov zostrojil konštrukciu pre dve nekonečné rodiny snarkov [39].

Do roku 1975, boli známe len nasledujúce 4 snarky [39].

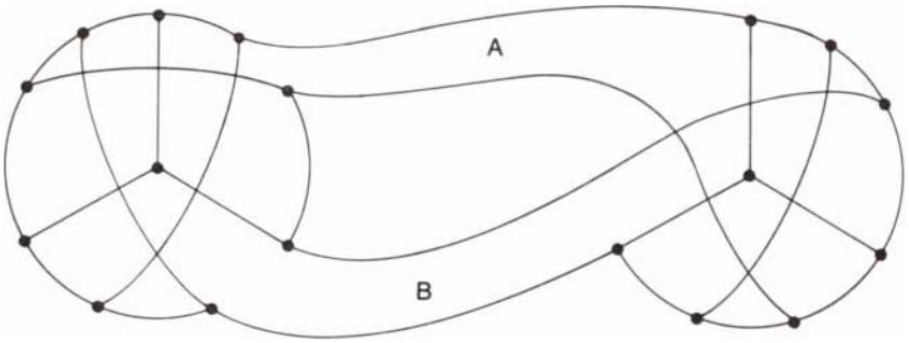
- Petersen 1891 10 vrcholov [53]
- Blanuša 1946 18 vrcholov [15]
- Descartes 1948 210 vrcholov [24]
- Szekeres 1973 50 vrcholov [57]

Ako sa neskôr ukázalo, Petersenov graf o veľkosti 10 vrcholov ukázaný na Obr. 3 je najmenším možným snarkom.

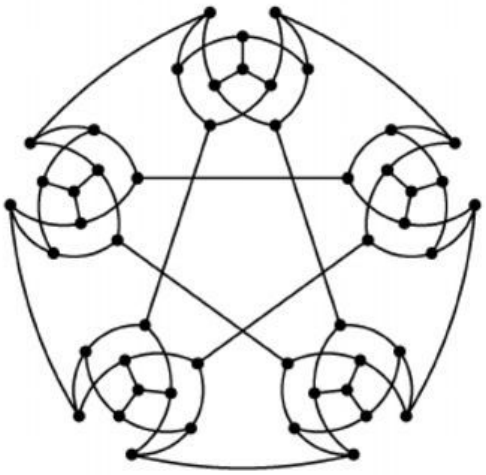


Obr. 3: Dva rôzne diagramy Petersenovho grafu [35].

Ďalší, Blanuša snark, vidíme na Obr. 4. Hoci si to Blanuša neuvedomoval, dá sa zostrojiť z dvoch Petersenových grafov [35]. Szekerešov snark môžeme vidieť na Obr. 5.



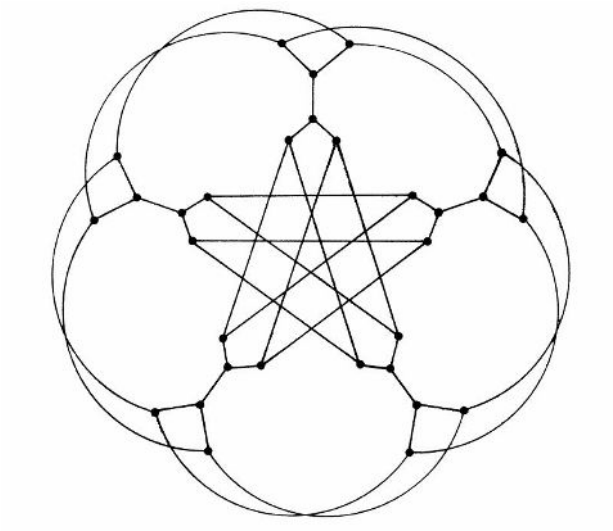
Obr. 4: Blanuša snark zostrojený z 2 Petersenových grafov.[35].



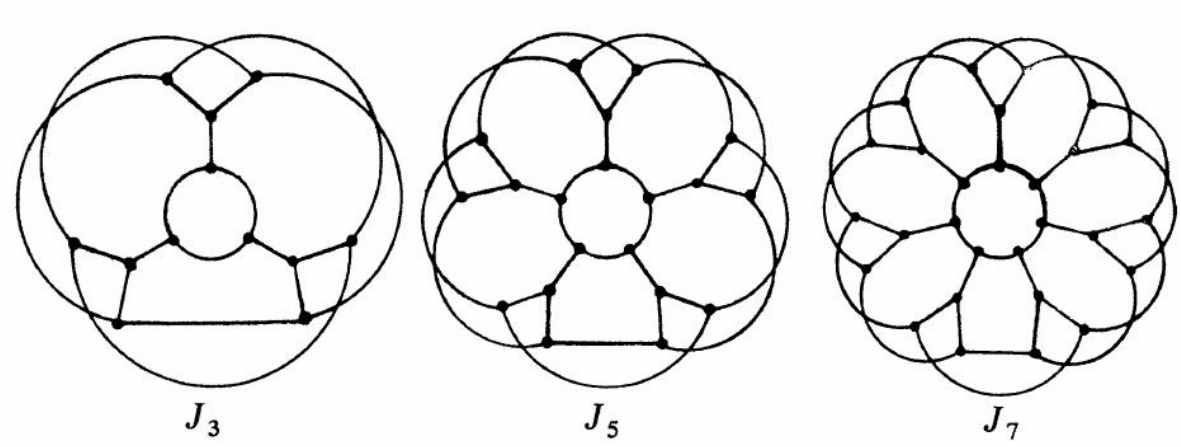
Obr. 5: Szekeres snark.

Ďalší známy snark, *dvojhviezdu*, môžeme vidieť na Obr. 6.

Rodiny snarkov. Po prvých, veľmi sporadicky vyskytujúcich sa snarkoch, objavil v roku 1975 Rufus Isaacs obsiahlu nekonečnú množinu snarkov. On sám tento objav označuje za neuveriteľný. Túto triedu grafov pomenoval po autoroch predchádzajúcich prvých snarkov a teda BDS, ako skratka pre Blanuša, Descartes, Szekeres. Práve týmito grafmi bolo nájdenie nekonečnej triedy inšpirované [39]. Prvé tri snarky z rodiny BDS vidíme na Obr. 7. Poznáme však aj ďalšie rodiny snarkov, ako napríklad Goldbergove snarky, zovšeobecnené Blanušove a Szekeresove snarky alebo zovšeobecnené Celminsove-Swartove snarky [63].



Obr. 6: Double-star snark [39].



Obr. 7: Flower snarky J3, J5 a J7.[39].

1.3.2 Počet snarkov

Ako už bolo spomenuté, do roku 1975 boli známe len 4 snarky. S nástupom výpočtovej techniky na pole matematiky a teoretickej informatiky sa postupne podarilo vygenerovať všetky snarky do veľkosti 36 vrcholov. Do veľkosti 30 vrcholov boli všetky testované vlastnosti na týchto grafoch zhodne overené nezávisle vo Švédsku a Belgicku [17]. Aktuálny stručný prehľad počtov snarkov uvádzame v Tabuľke 1. Všetky uvedené skupiny grafov aj tabuľka samotná sú k dispozícii na [5].

počet vrcholov	obvod ≥ 4	obvod ≥ 5	obvod ≥ 6	$\lambda_C \geq 5$
10	1	1	0	1
12, 14, 16	0	0	0	0
18	2	2	0	0
20	6	6	0	1
22	31	20	0	2
24	155	38	0	2
26	1 297	280	0	10
28	12 517	2 900	1	75
30	139 854	28 399	1	509
32	1 764 950	293 059	0	2 953
34	25 286 953	3 833 587	0	19 935
36	404 899 916	60 167 732	1	180 612

Tabuľka 1: Počet známych snarkov do veľkosti 36 vrcholov.

1.4 Chromatické vlastnosti snarkov

V nasledujúcej podkapitole sa dostávame k hlavnej téme práce. Z množstva vlastností, ktoré snark môže mať nás budú najviac zaujímať práve také, ktoré určitým spôsobom vyjadrujú mieru jeho neofarbiteľnosti. Práve tieto vlastnosti nám totižto indikujú, ako veľmi je snark náchylný k ofarbiteľnosti alebo naopak ako silne je neofarbiteľný. Najprv si definujeme niektoré operácie na grafe, ktoré budeme neskôr potrebovať. V nasledujúcich definíciách vždy predpokladáme, že graf je snark.

Pri definíciách chromatických vlastností grafu G budeme často uvažovať o určitom podgrafe G' grafu G, pričom G' vytvoríme odstránením niektorých vrcholov alebo hrán z grafu G. Preto odstránením hrany budeme chápať jej rozpojenie. Keď teda z grafu G odstránime hranu $e = (u, v) \in E$ znamená to, že vrcholy {u,v} už naďalej nebudú susedné, teda nebudú spojené hranou. Takéto odstránenie budeme značiť $G - (u, v)$. Ďalej pod odstránením vrcholu $v \in V$ z grafu $G = (V, E)$ budeme chápať odstránenie všetkých s ním susediacich hrán $e = (v, w) \in E$ a následné odstránenie vrcholu v z množiny V. Takéto odstránenie označujeme $G - \{v\}$. Toto odstraňovanie hrán alebo vrcholov z grafu G môže ovplyvňovať jeho ofarbiteľnosť. Ak by sme odstránili len jednu hranu či vrchol, ofarbiteľnosť grafu to nezmení. Ak však z grafu odstránime aspoň dve hrany, dvojicu vrcholov alebo kombináciu hrana vrchol, graf sa môže stať ofarbiteľný. Preto si definujeme odstrániteľné a neodstrániteľné dvojice hrán a vrcholov.

Definícia 1.11. *Dvojicu vrcholov {u,v} nazveme odstrániteľnou práve vtedy, keď graf $G - \{u, v\}$ zostane neofarbiteľný. Naopak neodstrániteľnou dvojicou vrcholov {u,v} na-*

zveme takú dvojicu, pre ktorú platí, že $G - \{u, v\}$ je ofarbiteľný. [51]

Definícia 1.12. *Dvojicu hrán $\{e, f\} \in E$ nazveme odstrániteľnou práve vtedy, keď graf $G - \{e, f\}$ zostane neofarbiteľný. Naopak neodstrániteľnou dvojicou hrán $\{u,v\}$ nazveme takú dvojicu, pre ktorú platí, že $G - \{u, v\}$ je ofarbiteľný. [51]*

Takéto odstrániteľné, resp. neodstrániteľné množiny hrán a/alebo vrcholov môžu úzko súvisieť aj s redukciami a konštrukciami snarkov (viď Prílohu E) [51, 55]. Keď máme definované potrebné pojmy, môžeme prejsť k definícii jednotlivých chromatických vlastností.

Definícia 1.13. *Rezistenciou grafu G nazveme minimálny počet hrán/vrcholov, po ktorých odstránení graf G zostane ofarbiteľný.*

Vieme, že hranová a vrcholová rezistencia sa rovnajú [63]. Preto ich v definícii nerozlišujeme.

Definícia 1.14. *Rezistibilitou hrany $e_0 \in E$ grafu G(V, E) nazveme minimálny počet hrán $\{e_1, ..., e_r\} \in E$ takých, že $G - \{e_0, e_1, ..., e_r\}$ zostane ofarbiteľný.*

Definícia 1.15. *Rezistibilitou vrcholu $v_0 \in V$ grafu G(V, E) nazveme minimálny počet vrcholov $\{v_1, ..., v_r\} \in E$ takých, že $G - \{v_0, v_1, ..., v_r\}$ zostane ofarbiteľný.*

Pre zjednodušenie si zavedieme aj pojem *index hranovej rezistibility* grafu G, ktorý bude vyjadrovať počet hrán grafu G, ktorých rezistibilita je vyššia ako rezistencia grafu G. Tento index budeme označovať e_{ri} (pre „edge resistability index“). Rovnako si takýto pojem zavedieme aj pre vyjadrenie počtu vrcholov s vyššou rezistibilitou ako je rezistencia grafu a teda *index vrcholovej rezistibility* grafu G. Tento index budeme označovať v_{ri}.

Kritické vlastnosti vyjadrujú náchylnosť grafu k ofarbiteľnosti.

Definícia 1.16. *Snark nazveme kritický práve vtedy, keď pre každú dvojicu susedných vrcholov $\{u, v\}$ platí, že $\{u, v\}$ je neodstrániteľná dvojica.*

Definícia 1.17. *Snark nazveme kokritický práve vtedy, keď pre každú dvojicu nesusedných vrcholov $\{u, v\}$ platí, že $\{u, v\}$ je neodstrániteľná dvojica.*

Keďže graf môže byť zároveň kritický aj kokritický, takýto graf nazveme *bikritický* resp. *ireducibilný* snark. Ak je však graf len kritický, môžeme ho pomenovať *striktne kritický* a naopak ak je len kokritický môžeme povedať, že je *striktne kokritický*. Každá z týchto skupín grafov sa môže vyznačovať určitými špecifickými vlastnosťami a má zmysel sa nimi zaoberať aj samostatne. Ak pri kritických vlastnostiach trochu uvoľníme podmienky, dostaneme sa k „subkritickým vlastnostiam“. Tieto vlastnosti tiež určitým spôsobom vyjadrujú náchylnosť grafu k ofarbiteľnosti.

Definícia 1.18. *Snark G=(V,E) nazveme vrcholovo subkritický práve vtedy, keď pre každý vrchol $v \in V$ existuje vrchol $u \in V$ taký, že $\{u, v\}$ je neodstrániteľná dvojica. Teda, že $G - \{u, v\}$ je ofarbiteľný.*

Definícia 1.19. *Snark G=(V,E) nazveme hranovo subkritický práve vtedy, keď pre každú hranu $e \in E$ existuje hrana $f \in E$ taká, že $\{e, f\}$ je neodstrániteľná dvojica hrán.*

Pre vrcholovo subkritický snark teda platí, že rezistibilita každého vrchola je rovná dvom. Pre hranovo subkritický graf to obdobne platí pre hrany. Ak je graf G vrcholovo subkritický no nie je hranovo subkritický, takýto graf nazveme *striktne vrcholovo subkritický*. Vzhľadom k tomu, že ako dokážeme neskôr, hranová subkritickosť implikuje vrcholovú subkritickosť, striktnú hranovú subkritickosť by nemalo zmysel definovať obdobne ako vrcholovú. Preto nazveme graf G *striktne hranovo subkritický* vtedy, ak G je hranovo subkritický no nie je kritický.

Ako ďalšie definujeme vlastnosti, ktoré sú akýmsi opakom kritických. Stabilné vlastnosti naznačujú silnú neofarbiteľnosť grafu.

Definícia 1.20. *Snark nazveme stabilný práve vtedy, keď pre každú dvojicu susedných vrcholov $\{u, v\}$ platí, že $\{u, v\}$ je odstrániteľná dvojica.*

Definícia 1.21. *Snark nazveme kostabilný práve vtedy, keď pre každú dvojicu nesusedných vrcholov $\{u, v\}$ platí, že $\{u, v\}$ je odstrániteľná dvojica.*

Ak je graf G stabilný a zároveň kostabilný G nazveme *bistabilný*. Ak je však G stabilný no nie je kostabilný, G nazveme *striktne stabilný*. Ak je však G kostabilný znamená to, že je bistabilný.

Vieme, že niektoré tieto vlastnosti navzájom súvisia. Tieto súvislosti by nám mohli neskôr pomôcť aj pri definovaní algoritmov pre zisťovanie daných vlastností. Dá sa pomerne jednoducho dokázať, že ak je graf kritický, znamená to, že je aj hranovo aj vrcholovo subkritický a tiež, ak je graf hranovo kritický je zároveň aj vrcholovo kritický.

Lema 1.1. *Každý kritický graf G=(V,E) je vrcholovo subkritický.*

Dôkaz. Ak je graf kritický, potom pre každý vrchol $v \in V$ existujú aspoň 3 vrcholy $\{u_1, u_2, u_3\} \in V$ také, že $(u_i, v) \in E$ a zároveň $\{u_i, v\}$ pre $i = 1, 2, 3$ tvorí neodstrániteľnú dvojicu vrcholov. Potom prirodzene pre každý vrchol $v \in V$ existuje aspoň jeden $u \in V$ taký, že $\{u, v\}$ je neodstrániteľná dvojica. □

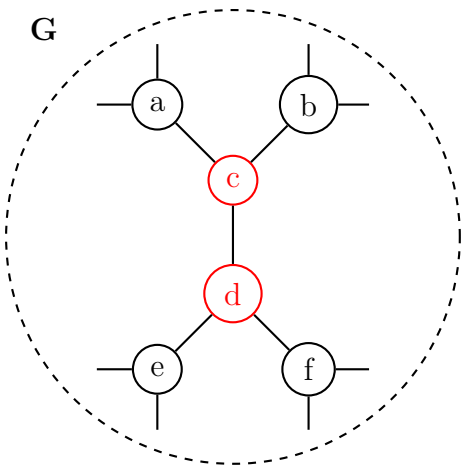
Obdobne by sme mohli dokázať, že každý kokritický graf je aj vrcholovo subkritický.

Lema 1.2. *Každý kokritický graf G=(V,E) je vrcholovo subkritický.*

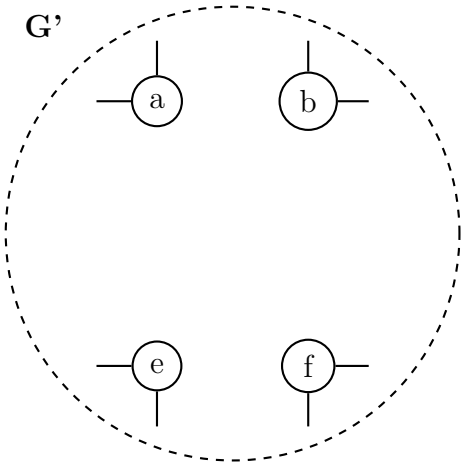
Dôkaz. Ak je graf G=(V,E) kokritický, potom pre každý vrchol $v \in V$ existuje neprázdna množina vrcholov $\{u_1, ..., u_i\} \in V$ takých, že $(u_j, v) \notin E$, kde $j = 1, ..., i$ a zároveň $\{u_i, v\}$ tvorí neodstrániteľnú dvojicu vrcholov. Potom prirodzene pre každý vrchol $v \in V$ existuje aspoň jeden $u \in V$ taký, že $\{u, v\}$ je neodstrániteľná dvojica. Spomenutá množina vrcholov $\{u_1, ..., u_i\} \in V$ grafu G=(V,E) je neprázdna preto, že G je snark a teda kubický graf o veľkosti najmenej desať vrcholov. Z toho vyplýva, že pre každý vrchol v existuje v grafe aspoň jeden vrchol u, ktorý s v nie je susedný. □

Lema 1.3. *Každý kritický graf G=(V,E) je hranovo subkritický.*

Dôkaz. Máme graf G=(V,E), ktorý je kritický snark. To znamená, že pre $\forall\{c, d\} \subset V$, ktoré susedia platí, že tvoria neodstrániteľnú dvojicu. Teda po odstránení $\{c, d\}$ sa graf G stane ofarbiteľný (viď. Obr. 8 a 9). To znamená, že na danom mieste vznikal konflikt ofarbiteľnosti. Graf, kde po odstránení takejto dvojice vrcholov zostanú vo výreze 4 vrcholy stupňa 2 nazveme G'. Do zobrazeného výrezu teraz môžeme pridať podgraf pozostávajúci z 2 vrcholov $\{c', d'\}$ spojených hranou (c',d') a ďalších dvoch hrán. Tieto ďalšie dve hrany môžu byť ľubovoľná kombinácia spĺňajúca $\{(a, c') \vee (b, c')\} \wedge \{(e, d') \vee (f, d')\}$. Týmto vložením vytvoríme graf G". Príklad takéhoto vloženého podgrafu môžeme vidieť na Obr. 10. Daný podgraf zaručene zachová ofarbiteľnosť celého grafu G". Totižto, nech by predchádzajúci graf G' bol ofarbený akokoľvek, vloženú cestu dĺžky tri vieme vždy hranovo ofarbiť tromi farbami bez toho aby sme vytvorili konflikt ofarbenia.



Obr. 8: Výrez kritického snarku, miesto kde vzniká konflikt ofarbiteľnosti (dva susediace vrcholy).

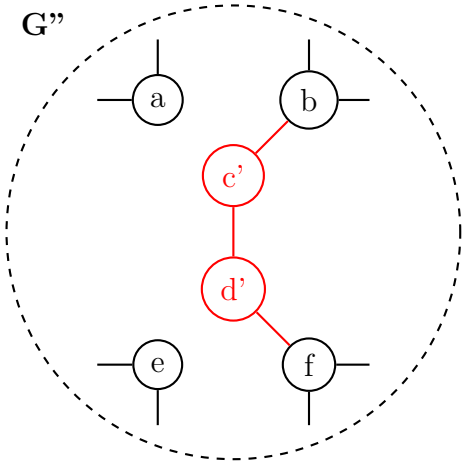


Obr. 9: Výrez grafu po odstránení dvoch susedných vrcholov. Zvyšok grafu je teraz zafarbiteľný - vyplýva z vlastnosti kritickosti.

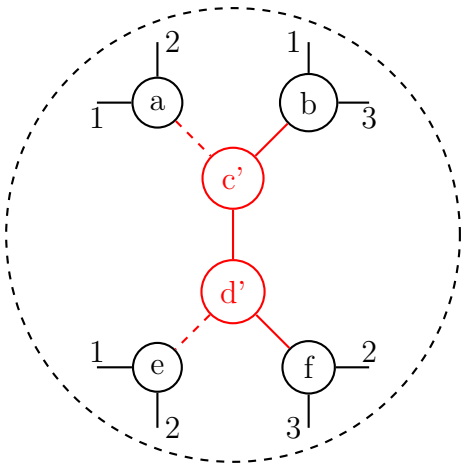
Keď však do takéhoto grafu G" pridáme zvyšné dve hrany doplňujúce graf G" na pôvodný graf G, graf sa stane neofarbiteľným. Z toho ale vyplýva, že konflikt ofarbiteľnosti spôsobujú práve tieto dve pridané hrany (viď na Obr. 11). Vzhľadom na to,

že dopĺňaný podgraf vytvárame v súlade s podmienkou, že bude vždy obsahovať práve jednu ľubovoľnú z hrán $(a, c') \vee (b, c')$ a práve jednu z hrán $(e, d') \vee (f, d')$ vyplýva, že vždy pre tú hranu z prvej dvojice, ktorá v podgrafe chýba vytvára neodstrániteľnú dvojicu hrana z druhej dvojice, ktorá v podgrafe chýba. Napríklad na Obr. 11) hrana *(a,c')* tvorí neodstrániteľnú dvojicu s hranou *(e,d')*. Keby sme doplnený podgraf vytvorili z hrán (a,c') a (e,d'), neodstrániteľnú dvojicu hrán by tvoril pár (b,c') a (f,d') a podobne.

Zostáva nám už len hrana *(c,d)*, s touto hranou je to ale tiež jednoduché. Vzhľadom na to, že kritickosť znamená neodstrániteľnosť *každých* dvoch susedných vrcholov, môžeme obdobne odstrániť namiesto vrcholov c a d napríklad vrcholy a a c a tým pádom sa aj hrana *(c,d)* dostáva do situácie kde predtým bola hrana (e,d).



Obr. 10: Pridané hrany a vrcholy nevytvárajú konflikt ofarbiteľnosti grafu z Obr. 9 za žiadnych okolností.



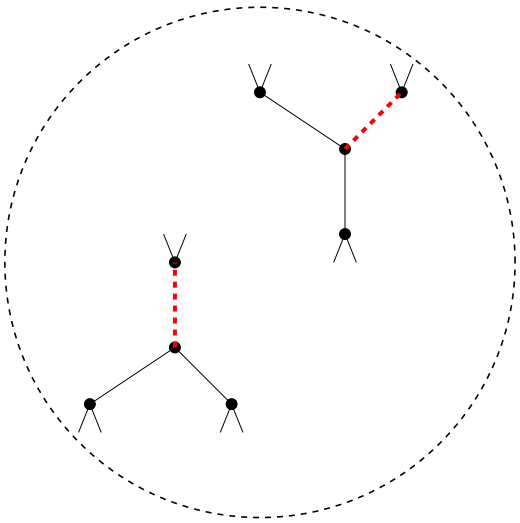
Obr. 11: Príklad odstránenia konfliktu ofarbiteľnosti odstránením dvoch (čiarkovaných) hrán.

Vzhľadom na to, že kritickosť znamená neodstrániteľnosť každých dvoch susedných vrcholov, aj náš dôkaž môžeme aplikovať na každú dvojicu susedných vrcholov. Z toho vyplýva, že pre každú hranu $(u, v) \in E$ existuje hrana $(x, y) \in E$ taká, že $\{(u, v), (x, y)\}$ tvoria neodstrániteľnú dvojicu hrán. □

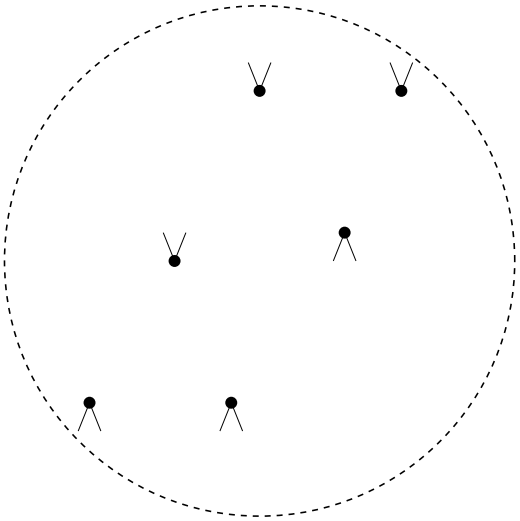
Lema 1.4. *Každý hranovo subkritický graf je aj vrcholovo subkritický.*

Dôkaz. Tu je dôkaz taktiež intuitívny. Z hranovej subkritickosti vyplýva, že pre každú hranu $(u, v) \in E$ existuje hrana $(x, y) \in E$ taká, že $\{(u, v), (x, y)\}$ tvorí neodstrániteľnú dvojicu. To znamená, že po odstránení napríklad dvojice vrcholov $\{u, x\}$ odstránime aj hrany (u,v) a (x,y) a graf sa stane ofarbiteľný. Teda dvojica $\{u, x\}$ je neodstrániteľná. Keďže v grafe G pre každú hranu (u,v) vieme nájsť takúto hranu (x,y), tak aj pre každý vrchol u vieme nájsť vrchol x taký, že $G - \{u, x\}$ je ofarbiteľný. Teda graf G je vrcholovo subkritický.

Grafické znázornenie môžeme vidieť na Obr. 12 a 13. □



Obr. 12: Dve hrany grafu, vytvárajúce konflikt ofarbiteľnosti.



Obr. 13: Po odstránení ľubovoľného vrcholu susediaceho s hranou z Obr. 12 sa odstráni aj hrana vytvárajúca konflikt.

Poznámka. Pri definovaní chromatických vlastností sa však ešte môžeme pozastaviť a opýtať sa, či existujú snarky, pre ktoré neplatí ani jedna z kritických vlastností? Teda snark, ktorý by nebol kritický, kokritický, a ani vrcholovo či hranovo subkritický. Takýto graf nazveme „akritický“ snark.

Definícia 1.22. *Snark G=(V,E) nazveme akritický práve vtedy, keď nie je vrcholovo subkritický.*

Akritický snark teda nemá žiadnu z kritických vlastností. Pre takýto akritický snark $G = (V, E)$ platia nasledovné tvrdenia:

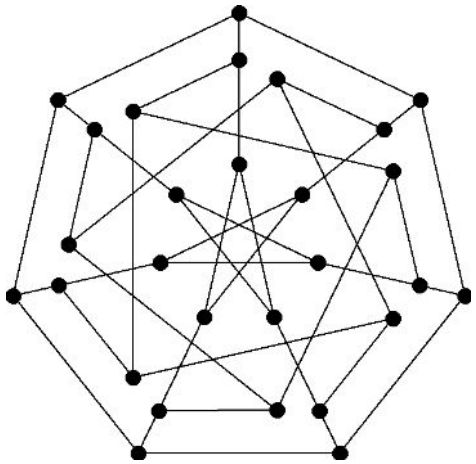
- $\exists\{u, v\} \in V$, kde {u,v} je susedná dvojica a $G - \{u, v\}$ je ofarbiteľný
- $\exists\{u, v\} \in V$, kde {u,v} je nesusedná dvojica a $G - \{u, v\}$ je ofarbiteľný
- $\exists\{u, v\} \in V$, kde {u,v} je susedná dvojica a $G - \{u, v\}$ je neofarbiteľný
- $\exists\{u, v\} \in V$, kde {u,v} je nesusedná dvojica a $G - \{u, v\}$ je neofarbiteľný
- $\exists v \in V$ taký, že $\nexists u \in V$, kde $u \neq v \wedge G - \{u, v\}$ je ofarbiteľný
- $\exists e \in E$ taká, že $\nexists f \in E$, kde $f \neq e \wedge G - \{e, f\}$ je ofarbiteľný

Ďalšie vlastnosti grafov

Nepárnosť grafu G, ktorú označujeme $\omega(G)$, je minimálny počet nepárnych cyklov v 2-faktore grafu G [51]. 2-faktorom grafu G(V, E) nazveme taký 2-regulárny podgraf grafu G, ktorý obsahuje všetky vrcholy V grafu G.

Hamiltonovskosť. Ak graf G obsahuje Hamiltonovskú kružnicu, G je ofarbiteľný. Naopak to však nemusí platiť. Poznáme mnoho ofarbiteľných kubických grafov, ktoré

Hamiltonovskú kružnicu nemajú [39]. Napr. Coxeterov graf o veľkosti 28 vrcholov zobrazený na Obr. 14.



Obr. 14: Coxeterov graf.

Pokrytie pármi. Perfect matching index alebo index perfektného párovania vyjadruje koľko perfektných párovaní vrcholov je potrebných pre pokrytie všetkých hrán grafu. Každý snark potrebuje na pokrytie všetkých hrán minimálne 4 perfektné párenia. Berge-Fulkersonova hypotéza tvrdí, že na pokrytie hrán každého snarku stačí 5 perfektných párení.

Perfektný snark je snark, ktorý sa nedá získať I-extenziou z menších snarkov [55]. Príklad takejto I-extenzie môžeme vidieť v prílohe E.

Keď máme definované potrebné pojmy a uviedli sme čitateľa do podrobností snarku a jeho vlastností, môžeme prejsť na metódy a prostriedky ich skúmania, ktorými budeme k danej problematike v práci pristupovať.

Kapitola 2

Metódy a prostriedky výskumu

Ako už bolo spomínané, jedným z hlavných cieľov práce je uľahčiť získavanie poznatkov potrebných pre výskum v oblasti snarkov. V tejto kapitole navrhneme výpočtový systém, ktorý by toto uľahčenie priniesol. V úvode špecifikujeme požiadavky na takýto systém. Následne na základe týchto požiadaviek navrhneme alebo opíšeme algoritmy, ktoré budú potrebné a potom navrhneme model systému. Na základe navrhnutého modelu budeme postupne systém implementovať. Na záver kapitoly uvedieme merania implementovaného systému a spomenieme možný vývoj systému v budúcnosti.

2.1 Analýza požiadaviek

Hneď v úvodnom náhľade môžeme vidieť dve základné požiadavky. Prvou je prirodzene aby bol systém schopný efektívne analyzovať chromatické vlastnosti, potenciálne aj veľkej vstupnej množiny snarkov (rádovo stovky miliónov grafov). Z hľadiska dobrého návrhu výpočtového systému je prirodzenou druhou požiadavkou čo najväčšia miera znovupoužiteľnosti systému alebo jeho prvkov. Inými slovami by sme mohli rozumným návrhom položiť základ pre komplexný modulárny systém pre podporu výskumu snarkov. Do tohto systému by neskôr malo byť možné doimplementovať algoritmy pre hromadné testovanie aj iných ako len chromatických vlastností snarkov. Tiež by sme pri návrhu mali myslieť na to, aby jednotlivé prvky systému boli od seba čo najmenej závislé a aby teda boli, napríklad ako knižnice a moduly, ľahko použiteľné aj mimo systém. Takýto prístup k návrhu je známy už desiatky rokov, viď napríklad [33].

Keď sa pokúsime požiadavky rozmeniť na drobnejšie, môžeme aspoň väčšinu z nich zhrnúť do nasledujúcich bodov. Systém by teda mal byť schopný:

- načítať graf z vopred určených formátov
- rozumne držať v pamäti inštanciu grafu (podporovať aj viaceré spôsoby)

- testovať taitovskú ofarbiteľnosť grafu
- overiť, či daný graf je súvislý a kubický
- zistiť obvod grafu
- zistiť cyklickú hranovú súvislosť grafu
- overiť, či graf je snark
- odstraňovať hrany a vrcholy grafu
- testovať chromatické vlastnosti snarkov a to najmä:
  - kritickosť
  - kokritickosť
  - vrcholovú subkritickosť
  - hranovú subkritickosť
  - stabilitu
  - kostabilitu
  - rezistenciu
  - rezistibilitu hrán
  - rezistibilitu vrcholov
- na základe zistených vlastností rozdeliť snarky do určených skupín
- zapísať výsledky testov do vopred určeného formátu

Ďalej by sme pre rozšíriteľnosť a znovu použiteľnosť systému mali pri návrhu dbať, aby:

- bolo možné pridať formát, z ktorého je systém schopný graf načítať
- bolo možné pridať triedu reprezentujúcu graf
- bolo možné pridať novú implementáciu ofarbovacieho algoritmu
- implementované funkcie a testy (ako napríklad test chromatických vlastností) neboli závislé na jednotlivej implementácii triedy graf alebo ofarbovacieho algoritmu
- bolo možné pridať nové algoritmy testujúce vlastnosti grafov
- bolo možné pridať nové funkcie a operácie aplikovateľné na grafy

Všetky nové funkcionality a pridané prvky by sme mali byť schopný efektívne použiť pri práci s veľkou množinou grafov, podobne ako pri teste chromatických vlastností. Systém by tiež mal byť dobre dokumentovaný.

V skratke teda budeme potrebovať algoritmy pre zisťovanie jednotlivých vlastností grafu, definovať podporované vstupy a požadované výstupy výpočtov, zamyslieť sa nad efektivitou výpočtov a nakoniec navrhnúť model systému pre implementáciu.

2.2 Algoritmy a špecifikácie

V tejto podkapitole si najprv opíšeme algoritmy, ktoré rozhodujú o ofarbiteľnosti grafu a tiež spomenieme niektoré ďalšie algoritmy potrebné pre systém. Ďalej navrhneme algoritmy pre testovanie chromatických vlastností a tiež rezistencie a rezistibility grafu. Potom sa pokúsime zamyslieť nad spôsobom, akým budeme dané vlastnosti pomocou navrhnutých algoritmov za rôznych okolností efektívne zisťovať. Napokon definujeme podporovaný vstupný formát súborov a špecifikujeme ako by mal vyzerať výstup.

Algoritmus pre obvod grafu. V implementácii systému budeme tiež potrebovať algoritmus pre zistenie obvodu grafu. Tento algoritmus je len jednoduchou modifikáciou prehľadávania grafu do šírky. Podrobnejší opis tohto algoritmu nájdeme v Prílohe C.

Algoritmus pre cyklickú hranovú súvislosť grafu. Pre zisťovanie cyklickej hranovej súvislosti grafu máme viacero možností viď. [26, 46]. My si pre implementáciu zvolíme algoritmus z [26].

2.2.1 Algoritmy pre hľadanie regulárneho hranového ofarbenia grafu

Teraz predstavíme stručný prehľad niektorých známych algoritmov pre hranové ofarbovanie grafu. K jednotlivým algoritmom uvedieme aj ich teoretickú výpočtovú zložitosť.

Rekurzívny algoritmus. Ideovo najjednoduchší a najintuitívnejší algoritmus pre hranové ofarbovanie grafu. Je založený na prehľadávaní grafu do šírky (BFS, Breadth First Search). Na začiatku vložíme do rady štartovací vrchol, potom spustíme rekurzívnu funkciu, ktorej výstup bude našim výsledkom. Náčrt rekurzívnej funkcie vidíme v 1.

Asymptotická zložitosť tohto algoritmu je $O(2^n)$ kde n je počet vrcholov grafu.

Kowalikov algoritmus [61] využíva tvrdenie, že graf G je 3-hranovo ofarbiteľný práve vtedy, keď obsahuje tzv. vhodné párovanie. V skratke tento algoritmus funguje tak, že rekurzívne generuje semi-kubické grafy pomocou vetvenia. Tieto grafy sú také, že ak niektorý z nich obsahuje vhodné párovanie, potom je pôvodný graf G 3-hranovo

Algorithm 1: OfarbiRekurzívne

Input: kubický graf G, rada vrcholov q

Output: true ak je graf taitovsky zafarbiteľný, false inak

```
1  v ← vyber vrchol z q
2  for ∀ permutácia farieb do
3      resetuj G a q do vstupného stavu
4      for ∀ hrana e = {u, v} susediaca s v do
5          priraď hrane e prislúchajúcu farbu z permutácie
6          if nastal konflikt ofarbenia then
7              pokračuj na ďalšiu permutáciu
8          q ← q + u
9      if OfarbiRekurzívne(G, q) then
10         return true
11 return false
```

ofarbiteľný. Týchto grafov môže byť exponenciálne veľa a testujú sa na prítomnosť vhodného párovania.

Definícia 2.1. *Vhodné párovanie alebo „fitting matching“. Párovanie M v grafe G nazveme vhodným ak každý súvislý komponent v podgrafe $G - M$ tvorí cestu alebo cyklus párnej dĺžky. [61]*

Definícia 2.2. *Semi-kubický graf Graf G nazveme semi-kubický, keď nemá žiadne vrcholy stupňa 1 a každé dva vrcholy stupňa 2 sú od seba vo vzdialenosti aspoň 3.*

Aj keď teoretická zložitosť tohto algoritmu je veľmi dobrá a to $O(1.344^n)$ kde n je počet vrcholov grafu, algoritmus má dosť vysokú réžiu a v praxi zrejme neohúri.

CVD heuristika [32] (skratka pre „conflict vertex displacement“) funguje pomocou vylepšovania náhodného ofarbenia grafu zmenou konfliktných vrcholov. Ak je graf ofarbiteľný, heuristika dá efektívne korektnú odpoveď. Na úvode náhodne predpočítame ofarbenie grafu 3 farbami. Potom hľadáme vrcholy, kde nastáva konflikt ofarbenia. Ďalej aplikujeme tzv. Kempeho výmenu. Z jednej z konfliktných hrán odštartujeme alternujúcu cestu až po vrchol, kde jedna z farieb cesty vytvára ďalší konflikt. Potom tieto farby hrán cesty navzájom vymeníme. Tento proces opakujeme vopred určený počet krát. Predpokladaný čas behu algoritmu je $O(n^2)$.

Redukcia na CNF-SAT. Taitovské ofarbenie pomocou redukcie na SAT a následné riešenie SAT formuly je tiež jednou z možností. Ukazuje sa, že dostupné implementácie

pre rozhodnutie SAT formuly umožňujú efektívne rozhodnúť taitovskú ofarbiteľnosť aj pre omnoho väčšie grafy ako ostatné spomínané prístupy. Pre redukciu problému taitovskej ofarbiteľnosti na rozhodnuteľnosť SAT formuly v konjunktívnom normálnom tvare je potrebné vykonať nasledovné kroky. Do SAT formuly budeme pridávať klauzuly, ktoré budú obsahovať premenné (literály). Premenná bude v tvare $x_{(ij)k}$ a bude nadobúdať platnú (true, 1) hodnotu práve vtedy, keď bude hrana (i, j) ofarbená farbou k.

1. pre $\forall$ hranu $(i, j) \in E$ pridáme do SAT formuly nasledovné klauzuly:
   - $(x_{(ij)1} \vee x_{(ij)2} \vee x_{(ij)3})$
   - $(\neg x_{(ij)1} \vee \neg x_{(ij)2} \vee x_{(ij)3})$
   - $(\neg x_{(ij)1} \vee x_{(ij)2} \vee \neg x_{(ij)3})$
   - $(x_{(ij)1} \vee \neg x_{(ij)2} \vee \neg x_{(ij)3})$
   - $(\neg x_{(ij)1} \vee \neg x_{(ij)2} \vee \neg x_{(ij)3})$
2. pre $\forall$ dvojicu susedných hrán $(i, j), (k, l) \in E$, kde $i = k \vee l$ alebo $j = k \vee l$, pridáme nasledovné klauzuly:
   - $(\neg x_{(ij)1} \vee \neg x_{(kl)1}) \wedge (\neg x_{(ij)2} \vee \neg x_{(kl)2}) \wedge (\neg x_{(ij)3} \vee \neg x_{(kl)3})$

V prvom bode teda pridávame pre každú hranu klauzuly, ktoré vyjadrujú, že každá hrana musí byť ofarbená práve jednou farbou. V druhom bode potom pridávame klauzuly, ktoré zabezpečujú, že žiadne dve susedné hrany nebudú mať rovnakú farbu. Do SAT formuly by sa dali pridať ešte ďalšie klauzuly, ktoré by potenciálne mohli urýchliť nájdenie rozporu. Tým sa však v našej práci zaoberať nebudeme.

Ďalšie možnosti. Z ďalších prístupov k ofarbovaniu grafu poznáme napríklad FFC algoritmus, ktorý rozhoduje ofarbiteľnosť pomocou hľadania toku na fundamentálnych cykloch. Tiež môžeme problém hranovej ofarbiteľnosti redukovať na niektorý iný problém z triedy NP. Napríklad môžeme taitovskú ofarbiteľnosť grafu G rozhodnúť pomocou vrcholovej 3-ofarbiteľnosti tzv. „line grafu“ H pôvodného grafu G [43].

2.2.2 Algoritmus pre testovanie chromatických vlastností

Pri testovaní chromatických vlastností sa dá postupovať viacerými spôsobmi v závislosti od množiny vlastností, ktoré chceme otestovať. V nasledujúcej časti ukážeme jeden z prístupov, ktorý bude testovať vybrané chromatické vlastnosti vrátane subkritických vlastností daného grafu. Druhý prístup uvádzame v Prílohe D. Oba algoritmy na vstupe predpokladajú kubický graf a ako podprocedúru využívajú algoritmus pre rozhodnutie taitovskej ofarbiteľnosti.

Ak chceme zistiť vybrané chromatické vlastnosti (menovite kritickosť, kokritickosť, vrcholovú a hranovú subkritickosť, stabilitu a kostabilitu) grafu, použijeme nasledujúci algoritmus. Ten najprv zisťuje všetky vlastnosti, ktoré sa týkajú vrcholovej kritickosti, resp. stability. Ak sa po tomto teste zistí, že graf nie je kritický, pokračuje sa na test hranovej subkritickosti opísanom v Algoritme 5.

Algorithm 2: TestChromaticProperties

Input: Cubic *graph*

Output: Property of criticality and both of subcriticality of given graph

```
1 edgeSubcritical ← false
2 TestVertexChromaticProperties(graph)
3 if graph is critical then
4 |   edgeSubcritical ← true
5 else
6 |   edgeSubcritical ← TestEdgeSubcriticality(simpleGraph)
7 return
```

Test vrcholových chromatických vlastností. Na rozdiel od testu subkritických vlastností, v tomto Algoritme 3, si priebežné výsledky odstrániteľnosti dvojice vrcholov ukladáme do matice M o veľkosti n^2, pričom n = počet vrcholov grafu. Hodnoty matice $M[i][j]$ ukazujú, či graf po odstránení dvojice vrcholov $\{i, j\}$ je ofarbiteľný alebo nie. Po otestovaní každého riadku matice (teda vrcholu i na prítomnosť neodstrániteľných dvojíc s ostatnými vrcholmi j) sa skontroluje, či nastala zmena v niektorej z vlastností.

Ak sme získali všetky výsledky, nie je nutné dopočítať zostávajúce položky matice a výpočet sa ukončí. Kontrola vlastností po otestovaní každého vrchola je opísaná v Algoritme 4.

Pri kontrole zmeny vlastností grafu po testovaní vrcholu v vezmeme v a pre všetky ostatné vrcholy u_i kontrolujeme, či $\{v, u_i\}$ sú susedné a či tvoria odstrániteľnú dvojicu. Ak sú $\{v, u_i\}$ susedné a neodstrániteľné, graf nie je stabilný a v zachováva vrcholovú subkritickosť. To znamená, že pre daný vrchol v sme už jeho partnera u_i pre neodstrániteľnú dvojicu našli. Ak však $\{v, u_i\}$ tvorí odstrániteľnú dvojicu, graf nie je kritický. Ak naopak $\{v, u_i\}$ nie sú susedné a tvoria neodstrániteľnú dvojicu, graf nie je kostabilný a v tiež zachováva vrcholovú subkritickosť. Ak tvoria odstrániteľnú dvojicu, graf nie je kokritický.

Algorithm 3: TestVertexChromaticProperties

Input: Cubic *graph*

Output: Criticality, cocriticality, stability, costability and vertex subcriticality of given graph

```
1  critical ← true
2  cocritical ← true
3  stable ← true
4  costable ← true
5  vertexSubcritical ← true
6  matrixOfColourings[graphSize][graphSize] ← fillallwithfalse
7  foreach vertex of graph do
8      sg′ ← removeVertex(graph, vertex)
9      foreach vertex′ of sg′ do
10         if matrixOfColourings[vertex][vertex′]nottestedyet then
11             sg″ ← removeVertex(sg', vertex')
12             colourable ← 3EdgeColour(sg")
13             matrixOfColourings[vertex][vertex'] = colourable
14     checkResultsOfVertex(matrixOfColourings, vertex)
15     if resultsobtained then
16         return          /* we already obtained all wanted results */
17
18 return
```

Test hranovej subkritickosti. Ak sa ukázalo, že daný graf je kritický, tento test nie je nutné spustiť. Ak však graf kritický nie je, musíme testovať aj jeho hranovú subkritickosť a teda pre každú dvojicu hrán overiť, či je odstrániteľná alebo nie. Tento test vidíme v Algoritme 5.

Časová zložitosť

Test vybraných chromatických vlastností sa skladá z dvoch hlavných častí. Test vrcholových chromatických vlastností beží v dvoch vnorených cykloch dĺžky maximálne n, kde n je počet vrcholov daného grafu. Táto časť má teda maximálne $O(n^2)$ iterácií. Druhá časť je test hranovej subkritickosti, kde sú dva vnorené cykly dĺžky maximálne m, kde m je počet hrán daného grafu. Počet iterácií tejto časti je $O(n^2)$ keďže počet hrán m v kubickom grafe je práve $\frac{3}{2}n$ pre n vrcholov a $\frac{3}{2}$ berieme ako konštantu.

Podobne ako to bolo v prípade testu vybraných chromatických vlastností aj test

Algorithm 4: CheckResultsOfVertex

Input: matrix of colurings, vertex v to check results of

Output: change in criticality, cocriticality, stability, costability and vertex subcriticality of given graph, also answer, if all results obtained

```
1  vertexSubcriticalFlag ← false
2  foreach vertex u_i of graph do
3      if v ∼ u_i then
4          if G − {v, u_i} is colourable then
5              vertexSubcriticalFlag ← true
6              stable ← false
7          else
8              critical ← false
9      else
10         if G − {v, u_i} is colourable then
11             vertexSubcriticalFlag ← true
12             costable ← false
13         else
14             cocritical ← false
15 if vertexSubcriticalFlag is false then
16     vertexSubcritical ← false
17 if all properties are false then
18     resultsObtained ← true
19 return
```

subkritických vlastností uvedený v prílohe D, sa skladá z dvoch vnorených cyklov pre test vrcholovej subkritickosti o počte iterácií $O(n^2)$ a následného testu hranovej subkritickosti o maximálnom počte iterácií $O(n^2)$.

Oba algoritmy však vo vnorených cykloch (teda pri zisťovaní vrcholových vlastností alebo hranovej subkritickosti) spúšťajú test 3-regulárnej hranovej ofarbiteľnosti, ktorý má vo všeobecnosti exponenciálnu časovú zložitosť. Zložitosť oboch predstavených algoritmov je teda rovnaká a to $O(n^2 * 2^n)$.

Algorithm 5: TestEdgeSubcriticality

Input: Cubic graph *graph*

Output: Edge subcriticality property of given graph

```
1  edgeSubcritical ← true
2  foreach edge of graph do
3      sg′ ← removeEdge(graph, edge)
4      foreach edge′ of sg′ do
5          sg″ ← removeEdge(sg', edge')
6          if 3EdgeColour(sg″) = true then
7              edgeSubcritical ← true
8              break
9          edgeSubcritical ← false
10     if edgeSubcritical = false then
11         return false
12 return true
```

2.2.3 Algoritmus pre rezistenciu a rezistibilitu

Prvým prístupom k testovaniu rezistencie grafu, ktorý sme skúsili, bolo rekurzívne zisťovanie rezistibility hrán pomocou prehľadávania do hĺbky. Implementácia takého algoritmu sa však ukázala ako vysoko neefektívna. Algoritmus zbytočne pri množstve hrán zachádzal do príliš veľkej hĺbky grafu, a tak aj zistenie rezistibility jednej hrany pri grafe do 30 vrcholov bolo časovo neprijateľné. Druhým pokusom bol algoritmus postupujúci po úrovniach, ktorý nezisťuje postupne rezistibilitu všetkých hrán, ale postupne sa pýta, či je graf G i-rezistentný pre *i = 1, ..., m*, kde *m* je počet hrán grafu.

Algoritmus pre hranovú rezistenciu grafu je opísaný v Algoritme 6. Postupne sa pre každé prirodzené číslo *i = 1, ..., m*, kde $m = |E|$ budeme pýtať, či je toto číslo *i* rovné rezistencii grafu. Na túto otázku nám bude odpovedať algoritmus 7, ktorý bude postupne testovať všetky *i-tice* hrán $\{e_1, ..., e_i\}$, či $G - \{e_1, ..., e_i\}$ je ofarbiteľný. Ak áno, práve aktuálne *i* bude rovné rezistencii grafu G.

Test rezistibility hrán bude prebiehať tak, že pre každú hranu *e* grafu G vytvoríme graf $G' = G - e$ a budeme testovať rezistenciu grafu G'. Práve rezistencia grafu G' je totiž zároveň rezistibilitou hrany e. Tento postup vidíme v Algoritme 8.

Algoritmus pre vrcholovú rezistenciu a rezistibilitu grafu odvodíme z predchádzajúceho algoritmu pre testovanie hranovej rezistencie a rezistibility. Stačí, keď v Algoritme 6 vymeníme podmienku hlavného *while* cyklu za $i \leq |V| of graph$ a v Algoritme 7 budeme

Algorithm 6: EdgeResistance

Input: graph

Output: edge resistance of given graph

```
1 i ← 1
2 while i ≤ |E|ofgraph do
3     result ← EdgeResistanceRecursive(graph, i)
4     if result is true then
5         return i
6     i++
```

vo *foreach* cykle namiesto hrán prechádzať cez všetky vrcholy daného grafu.

Pre zistenie rezistibility vrcholov potom vo *foreach* cykle Algoritmu 8 budeme podobne prechádzať namiesto všetých hrán grafu cez všetky jeho vrholy.

Algorithm 7: EdgeResistanceRecursive

Input: graph, maxNesting

Output: true if edge resistance of given graph equals to maxNesting

```
1  if maxNesting = 0 then
2      if graph is colourable then
3          return true
4      else
5          return false
6  foreach edge ∈ edges of graph do
7      graph′ ← remove edge from graph
8      result ← EdgeResistanceRecursive(graph', maxNesting-1)
9      if result is true then
10         return true
11 return false
```

Časová zložitosť

Algoritmus pre hranovú rezistenciu a rezistibilitu má zložitosť $O(k * m^k)$, kde m je počet hrán grafu a k je maximálna možná rezistencia grafu. Ak by sme každému vrcholu grafu odstránili jednu hranu, z grafu by sa stal 2-regulárny graf, ktorý je určite ofarbiteľný 3 farbami a teda počet týchto hrán môže byť našou hornou hranicou k. V tomto prípade by k bolo rovné $\frac{n}{2}$.

Algorithm 8: EdgeResistability

Input: graph

Output: set of edges with its resistability

```
1 results ← ∅
2 foreach edge ∈ edges of graph do
3     graph' ← remove edge from graph
4     resistability ← EdgeResistance(graph')
5     results ← results + {edge, resistability}
6 return results
```

Algoritmus pre vrcholovú rezistenciu a rezistibilitu má zložitosť $O(k * n^k)$, kde n je počet vrcholov grafu a k je maximálna možná rezistencia grafu.

Algoritmy testu chromatických vlastností z 2.2.2 a test rezistencie grafu by sa dali ľahko skombinovať, čo by ušetrilo výpočtový čas potrebný na zistenie všetkých daných vlastností.

2.2.4 Paralelizmus

Ako už bolo spomenuté, našim primárnym cieľom bude do daného výpočtového systému implementovať algoritmy pre testovanie vybraných chromatických vlastností a rezistencie grafu opísané v predchádzajúcej časti. Tu je dôležité všimnúť si teoretickú časovú zložitosť navrhnutých algoritmov. Tá je pri teste vybraných chromatických vlastností pre jeden graf $O(n^2 * 2^n)$, kde n je počet vrcholov grafu. Už táto zložitosť sama o sebe nie je lichotivá, no pri dobrej implementácii ofarbovacieho algoritmu by výpočet stále mohol prebehnúť v prijateľnom čase (pre graf do sto vrcholov rádovo v sekundách). Ak si však predstavíme, že chceme testovať tieto vlastnosti nie pre jeden graf, ale pre celý súbor grafov, musíme k zložitosti pridať k určujúce počet grafov. Takto dostávame zložitosť $O(k * n^2 * 2^n)$. Číslo k v tomto prípade však nemôžeme brať ako konštantu. Pri príliš veľkom súbore grafov by ich počet k mohol dokonca zásadne ovplyvniť čas behu výpočtu v neprijateľnej miere. Rovnako je to pri teste rezistencie grafu a rezistibility hrán a vrcholov. Tu je dokonca možné, že v prípade väčšieho grafu systém nebude schopný v prijateľnom čase otestovať ani jeden takýto graf.

Predstavme si, že testovanie jedného grafu o veľkosti 36 vrcholov na všetky chromatické vlastnosti by trvalo napríklad jednu sekundu. Pri testovaní všetkých grafov o veľkosti 36 vrcholov, ktorých je vyše 404 miliónov, by to však bolo vyše 112 222 hodín, čo je takmer 13 rokov. To by samozrejme bolo v rozpore s jednou zo základných požiadaviek na systém a to efektivitou výpočtov. V tomto momente prichádza otázka,

ako aj v tejto situácii zabezpečiť prijateľný čas behu výpočtov. V rámci optimalizácie jednotlivých krokov sa najmä vzhľadom na NP-úplnosť ofarbovania už veľa ušetriť nedá. Sekvenčný výpočet takejto škály má svoje ohraničenia. V dnešnej dobe je už ale prirodzenou odpoveďou paralelizmus výpočtu.

Paralelizácia výpočtov je v dnešnej dobe bežnou praxou. Poznáme množstvo paralelných algoritmov, ktoré riešia známe problémy ako triedenie, násobenie matíc, dopytovanie v databázach, hľadanie najkratších ciest v grafe a podobne [20, 22, 45]. Rovnako poznáme aj množstvo techník ako sa dajú problémy paralelizovať [16]. Práve otázka, ako problém prispôsobiť pre paralelizáciu je v tomto prípade kľúčová. Zásadná je dekompozícia problému na menšie časti. Ak problém vieme rozdeliť na menšie časti, ktoré môžu byť vykonávané súbežne, potom je náš problém ideálnym kandidátom pre tento prístup. V najlepšom prípade nám N výpočtových jednotiek prinesie N-násobné zrýchlenie. Takéto zrýchlenie sa však len málokedy dá dosiahnuť, pretože paralelizácia výpočtu si vo veľkej väčšine vyžaduje určitú réžiu a mnohé problémy sa nedajú úplne dekomponovať na nezávislé podproblémy. Pri paralelizácii problému môžeme brať do úvahy aj rozsah v akom chceme problém paralelne riešiť. Po technickej stránke je tento rozsah ohraničený paralelným výpočtovým strojom, na ktorom algoritmus bude bežať.

Prvým a najbežnejším prípadom môže byť použitie dnes bežného viac jadrového procesoru, tzv. **CPU paralelizmus**. V tomto prípade je výpočet vykonávaný vo viacerých vláknach, ktoré sú na sebe viac menej nezávislé - tzv. *multithreading*. Celý výpočet má spoločnú *cache* pamäť aj operačnú pamäť. Komunikácia prebieha len lokálne prostredníctvom rýchlych zberníc. Takýto viacjadrový CPU ale aj v dnešnej dobe poskytuje len určitý počet výpočtových jadier. Pre ilustráciu môžeme uviesť napríklad v dnešných dňoch vysokovýkonný profesionálny CPU „AMD RYZEN Threadripper 2990WX“ poskytuje 32 jadier [1]. Pre bežné použitie pri menej náročných úlohách je tento prístup ideálny. Ak by sme však potrebovali masívnejší paralelizmus museli by sme sa pozrieť niekam inam.

Ak by sme teda chceli úlohu paralelizovať masívnejšie, vhodným spôsobom by bolo využitie niektorého typu vysokovýkonného počítania, teda **HPC** (pre „high performace computing“). Ako takýto typ počítača môžeme vnímať napríklad *superpočítač, klaster, grid* alebo *cloud*. Každý z nich má svoje výhody a nevýhody [44, 56]. Najmä z dôvodu dostupnosti sa v práci sa zameriame len na jeden typ HPC počítača a to *klaster*.

Klaster vo všeobecnosti znamená zhluk susediacich objektov. V informatike ide o množinu počítačov (výpočtových jednotiek), ktoré spolupracujú a dajú sa vnímať ako jeden systém [23]. Ide teda o paralelný-distribuovaný systém, ktorý pozostáva z množstva výpočtových strojov, uzlov. Ako jeden uzol budeme vnímať jeden samostatný výpočtový stroj, napríklad jeden CPU s vlastnou operačnou pamäťou. Tieto výpočtové

uzly sú navzájom prepojené určitým typom siete, komunikujú spolu a dokážu navonok vystupovať ako jeden ucelený systém. Takýto klaster môže obsahovať stovky, tisíce a teoreticky až neobmedzené množstvo uzlov. V takom rozsahu a pri dobrej paralelizácii problému môže klaster výrazne znížiť časovú náročnosť daného výpočtu. Po teoretickej stránke pôjde vždy ale len o polynomiálne zlepšenie.

Podstatným krokom pre využitie paralelizmu aj v našom prípade by bola už spomínaná dekompozícia úlohy. V práci sú pre nás zaujímavé hlavne nasledujúce problémy:

- testovanie chromatických vlastností grafu
- testovanie chromatických vlastností na množine grafov
- testovanie vrcholovej a hranovej rezistencie a resistability hrán a vrcholov grafu
- testovanie rezistencie a rezistibility množiny grafov

Náročnosť každého zo spomenutých problémov je závislá na veľkosti grafu a tiež na množstve testovaných grafov. Už z predchádzajúcej kapitoly o analýze zložitosti teda vychádza, že každá z týchto úloh si môže vyžadovať rôznu granularitu dekompozície. Ak chceme testovať chromatické vlastnosti jedného malého grafu, nemusíme vôbec uvažovať o paralelizácii daného výpočtu. Ak však chceme testovať tieto vlastnosti na väčšom grafe, sekvenčný algoritmus by mohol trvať neprípustne dlho (rádovo v minútach až hodinách). Tu by sme mohli uvažovať o rozdelení problému na úrovni jedného grafu. Napríklad by každé vlákno mohlo samostatne pracovať s vopred určenou podmnožinou dvojích vrcholov resp. hrán, pre ktoré by určovala ich (ne)odstrániteľnosť. Takýmto spôsobom by sa testovanie celého grafu mohlo rozdistribuovať na maximálne n^2 vlákien. Testovanie chromatických vlastností na množine grafov nám ponúka ďalšiu možnosť dekompozície. Úplne najjednoduchší prístup by v tomto prípade bol priradiť každému vláknu jeden alebo podmnožinu množiny grafov, ktoré samostatne otestuje a potom zhromaždiť výsledky. Pri testovaní rezistencie a rezistibility môžeme použiť podobnú stratégiu.

Paralelizácia či už na úrovni viacjadrového procesora alebo na úrovni klastra sa teda v našom prípade zdá byť rozumnou voľbou. Ak však budeme vyvíjať viacero rôznych paralelných algoritmov pre rôznu granularitu rovnakej úlohy, tieto algoritmy by mali mať čo najviac spoločného. Implementácia týchto algoritmov by teda mala byť schopná v čo najväčšej miere využívať spoločné prvky a líšiť sa len v tom najnutnejšom. Tiež budeme musieť nájsť vhodnú technológiu poskytujúcu prostredie pre paralelné a distribuované výpočty.

2.2.5 Formát vstupných a výstupných súborov

Vstupný súbor by mal obsahovať jeden alebo množinu grafov v podporovanom formáte. V prvotnom návrhu bude podporovaný len *graph6* formát grafu. Tento formát dokáže reprezentovať jednoduchý neorientovaný graf do veľkosti 68719476735 vrcholov. Zvolili sme si ho z dvoch dôvodov. Prvým je, že súbory všetkých vygenerovaných snarkov do veľkosti 36 vrcholov sú dostupné práve v ňom. Druhým dôvodom je relatívne nízka pamäťová náročnosť na uloženie grafu v takomto formáte. Ešte úspornejšie by bolo uloženie snarkov do formátu *sparse6*, ktorý je vhodný pre riedke grafy. Úplná a detailná špecifikácia oboch týchto kódovaní grafu sa dá nájsť na [3]. Súbor obsahujúci šesť grafov zapísaných graph6 kódovaním tak obsahuje 6 riadkov, kde každý riadok zodpovedá práve jednému grafu. Pre ilustráciu uvádzame obsah súboru, ktorý pozostáva zo všetkých snarkov o veľkosti 20 vrcholov na Obr. 15.

```
S?hW@eOGG??B_?O@g???C?a???wC@??Oc
S?hW@eOGGC?A_A?@gO??GO???GW?AO?@c
S?GW@E?GG?GB_AO_g_?CP_??P?G‘??O?S
S?‘W@e?GG?GA_A??g?GCP?_O?KHO??‘?C
S?hW@eOGG?GB_A?_G?GC???_??w?O_?AS
S?gQ@eOOGC?AP??BO@@?GB????o?E???[
```

Obr. 15: Súbor so všetkými snarkami o veľkosti 20 vrcholov v graph6 formáte.

Výstup by mal obsahovať počet otestovaných grafov. Rovnako tak aj sumár obsahujúci počty grafov s jednotlivými testovanými vlastnosťami, ktoré sa nachádzajú vo vstupnej množine. Ďalej by výstup mohol obsahovať vybrané skupiny záznamov, podľa určitých vopred špecifikovaných vlastností. Takýto záznam by obsahoval graf v graph6 formáte a jednotlivé vlastnosti tohto grafu. Pre ďalšie spracovanie, napríklad do nerelačnej databázy by bolo rozumné tieto záznamy a skupiny záznamov zapisovať vo formáte JSON [9, 27]. Ilustračný príklad výsledného sumáru k testovanej vzorke grafov uvádzame na Obr. 16 a príklad k skupine záznamov s určitými vlastnosťami vo formáte JSON na Obr. 17.

2.3 Návrh logického modelu

Na základe požiadaviek špecifikovaných v podkapitole 2.1 teraz navrhneme logický model systému. Ten by mal byť schopný paralelne (na jednom CPU ale aj na klastri) efektívne vykonať vybrané úlohy na kubických grafoch.

```
# BICRITICAL: 115 (irreducible)
# CRITICAL: 115
# STRICTLY CRITICAL: 0
# COCRITICAL: 115
# STRICTLY COCRITICAL: 0
# VERTEX SUBCRITICAL: 28391
# STRICTLY VERTEX SUBCRITICAL: 21
# EDGE SUBCRITICAL: 28370
# STRICTLY EDGE SUBCRITICAL: 28255
# NONE: 8

# STABLE: 0
# STRICTLY STABLE: 0
# COSTABLE: 0
# STRICTLY COSTABLE: 0
# BISTABLE: 0

# NUMBER OF RESULTS: 28399
```

Obr. 16: Takto by mohol vyzerať sumár výsledkov po testovaní súboru grafov na chromatické vlastnosti.

Skôr ako začneme priamo modelovať takýto systém, budeme potrebovať špecifikovať niekoľko základných prvkov a to:

- vývojové prostredie, pre ktoré bude model určený
- technológiu pre implementáciu paralelného algoritmu
- paralelný algoritmus pre vykonávanie vybraných procedúr
- algoritmy pre vybrané procedúry

2.3.1 Vývojové prostredie a paralelné technológie

Systém budeme modelovať a implementovať objektovo orientovaným spôsobom. Pre implementáciu si zvolíme programovací jazyk C++. Jazyk C++ v štandarde C++14 a C++17 je moderný objektovo orientovaný jazyk s dôrazom na efektivitu výslednej aplikácie [40]. Poskytuje dobrú podporu pre generické programovanie a dnešné kompilátory sú schopné výraznej optimalizácie zdrojového kódu. Ďalším dôvodom pre výber tohto vývojového prostredia je natívna podpora paralelných knižníc ako *thread*, *openMP* či *MPI*. Knižnica thread je priamou súčasťou C++ od štandardu C++11 a spolu s knižnicou openMP poskytujú podporu pre viacvláknové programovanie v rámci

```
{
  "STABLE_GRAPHS": [
    {
      "bicritical": false,
      "bistable": false,
      "cocritical": false,
      "costable": false,
      "critical": false,
      "cyclicConnectivity": 0,
      "edgeSubcritical": false,
      "girth": 5,
      "graphInG6": "S?hW@eOGG??B_?O@g???C?a???wC@??Oc",
      "none": false,
      "numberOfVertices": 20,
      "snark": true,
      "stable": true,
      "vertexSubCritical": false
    },
    {
      ... d'alší záznam o grafe ...
    }
  ]
}
```


Obr. 17: Takto by mohol vyzerať JSON záznam grafov spolu s ich vlastnosťami.

jedného procesora. Knižnica MPI (skratka pre „message passing interface“), ktorá nie je priamou súčasťou C++ implementuje rozhranie pre komunikáciu uzlov viac procesorového systému, v našom prípade klastra [48, 54].

Ďalšou možnosťou pre implementáciu nášho systému by mohol byť napríklad jazyk Go, ktorý je inšpirovaný jazykom C a je orientovaný na paralelné programovanie s dôrazom na efektivitu. Hlavným dôvodom prečo sme uprednostnili jazyk C++ je podpora komunity a dostupnosť implementovaných knižníc. Tiež by sme mohli uvažovať o použití jazyka Java. Tento jazyk je však interpretovaný a spracovávaný pomocou virtuálneho medzistupňa a využíva dynamickú správu pamäti (tzv. *garbage collector*) na základe čoho predpokladáme nižší výsledný výkon [7, 36]. Ďalším dôvodom je napríklad dostupnosť implementovaných ofarbovacích algoritmov alebo knižníc pre rozhodnutie riešiteľnosti SAT formuly, ktoré zásadne ovplyvňujú výsledný výkon nášho

systému. Najrýchlejšie dostupné *ofarbovače* ako aj *SAT solvery* sú implementované práve v jazyku C++ [2, 12].

2.3.2 Paralelný algoritmus

Ďalším krokom pred samotným návrhom modelu celého systému je návrh paralelného algoritmu, ktorý by umožnil spracovávanie zadanej úlohy na viacerých vláknach resp. uzloch súčasne. Chceme, aby tento algoritmus bol využiteľný pre rôzne úlohy a preto sa ho pokúsime do istej miery zovšeobecniť.

Pri úplne všeobecnom pohľade môžeme predpokladať, že algoritmus dostane vstupný súbor inštancií a spôsobom akým vykonať všetky podúlohy pre každú inštanciu. Takýto úplne zovšeobecnený algoritmus (viď. 9) nám však pri konkrétnej implementácii veľa nepomôže a nijakým spôsobom nereflektuje zvolenú technológiu.

Algorithm 9: SimpleParallelAlgorithm

Input: input set of instances

Output: results

```
1 for subjob of instance do in parallel
2     do subjob
3     add result to results
```

Ďalej sa teda pokúsime navrhnúť špecifickejší algoritmus, ktorý bude využívať MPI rozhranie. Pri implementácii paralelných algoritmov je bežným postupom určenie role *hlavného* uzla a *pracovného* uzla. Pri takomto rozdelení hlavný uzol „moderuje“ celý beh programu a pracovné uzly vykonávajú úlohy, ktoré im hlavný uzol pridelí. Ďalej v práci môžeme hlavný uzol nazývať „master“ a pracovné uzly „slave“.

Hlavný uzol bude v cykle prijímať výsledky a posielať úlohy kým nedostane výsledky zo všetkých rozposlaných úloh. Môže sa zdať, že sme si pomýlili poradie úkonov v hlavnom cykle. Najprv však hlavný uzol prijme správu od niektorého z pracovných uzlov, ktorá môže, ale aj nemusí obsahovať už vypočítané výsledky. Na základe tejto správy vie, že daný uzol je voľný. Pripraví mu teda úlohu a dáta a pošle ich v správe. Po odoslaní správy nasleduje ďalšia iterácia cyklu. Týmto spôsobom je zaručené čiastočné vyváženie vyťaženia pracovných uzlov. Vždy sa nová úloha priradí uzlu, ktorý sa prvý ohlásil ako voľný. Formuláciu môžeme vidieť v algoritme 10.

Každá odoslaná správa má svoj identifikátor. Vedúci uzol si uchováva zoznam práve spracovávaných správ. Takéto správy boli odoslané, ale ešte na ne nebola prijatá odpoveď. Vo fáze prijímania správy môžeme pomocou neblokujúcej komunikácie

Algorithm 10: MasterNode

Input: $inputFile$ - file with instances, $outputFile$ - file for results

```
1 openFile(inputFile)
2 while not received all results do
3     Receive(dataIn, ..., fromSlaveID)
4     processResults(dataIn)
5     prepareData(dataOut)
6     Send(dataOut, ..., fromSlaveID)
7 shut down all slaves
8 write results to outputFile
```

MPI_Irecv sledovať ako dlho uzol čaká na prijatie odpovede. Ak tento čas čakania presiahne stanovený limit, môžeme správu odoslať znovu niektorému z voľných uzlov a počkať na novú odpoveď. Týmto spôsobom môžeme predísť nekonečnému čakaniu na niektorý z výsledkov, ktorý od konkrétneho pracovného uzlu z rôznych dôvodov nemusí nikdy prísť. Dôvodom môže byť napríklad neočakávaná chyba a pád aplikácie na niektorom z uzlov alebo hardvérový problém niektorého uzlu ako napríklad výpadok elektrickej energie alebo odpojenie zo siete.

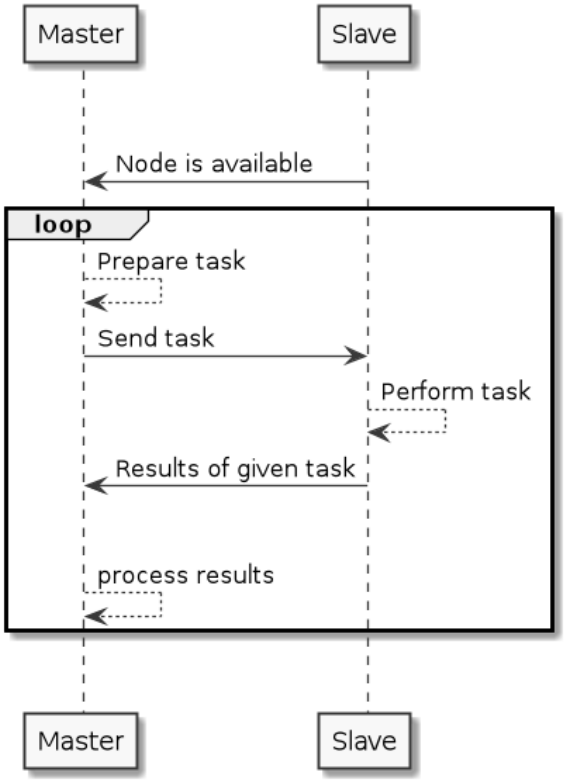
Pracovný uzol na začiatku svojej činnosti pošle správu hlavnému uzlu o tom, že je pripravený. Následne na to uzol vstupuje do cyklu, kde prijíma úlohy od hlavného uzla, spracuje ich a výsledky posiela naspäť a nasleduje ďalšia iterácia. Formuláciu môžeme vidieť v algoritme 11.

Algorithm 11: SlaveNode

```
1 Send(null, ..., toMaster)
2 while not shutdown signal do
3     Receive(inData, ..., fromMaster)
4     result ← performGivenFunction(inData)
5     Send(outputData, ..., toMaster)
```

Jednoduchú ilustráciu komunikácie a priebehu výpočtu takéhoto algoritmu môžeme vidieť na Obr. 18.

Predstavený paralelný algoritmus pozostávajúci z dvoch samostatných algoritmov pre hlavný a pracovný uzol je takto schopný postarať sa o paralelizáciu výpočtu do istej miery všeobecnej úlohy. Môžeme povedať, že ide určitým spôsobom o šablónu



Obr. 18: Priebeh MPI výpočtu a komunikácie.

pre konkrétny paralelný algoritmus. Pre túto šablónu si teraz môžeme vytvoriť viacero špecifikácií. Pre každú špecifikáciu bude nutné definovať metódy objavujúce sa vo vyššie uvedených algoritmoch, a ktoré doteraz mali len neurčitú funkciu. Pre hlavný uzol by išlo o funkcie *processResults* a *prepareData* a pre pracovný uzol ide o funkciu *performGivenFunction*. Tiež je dôležité uviesť formu a účel ich parametrov *dataIn a dataOut*, teda správ medzi hlavným a pracovným uzlom.

Paralelný test chromatických vlastností súboru grafov. V tomto prípade by sme mohli jednotlivé funkcie použité v týchto algoritmoch definovať nasledovne. Funkcia *prepareData(dataOut)* by do odchádzajúcej správy načítala riadky zo vstupného súboru. Každý riadok vstupného súboru by predstavoval graf vo formáte g6. Funkcia *processResults(dataIn)* by spracovala vypočítané chromatické vlastnosti daných grafov. Podľa týchto vlastností by grafy mohla napríklad roztriediť do rôznych výstupných množín. Pre pracovný uzol by funkcia *performGivenFunction(inData)* pre každý riadok prijatých dát načítala graf z g6 formátu. Následne by graf otestovala na všetky chromatické vlastnosti a tieto výsledky by pripojila do správy pre odoslanie.

Paralelný test rezistencie a rezistibility súboru grafov. Aj tu by sme pre hlavný uzol definovali funkcie rovnako. Pre pracovný uzol by funkcia *processResults(dataIn)*

tiež načítala graf z g6 formátu a následne by ho otestovala na rezistenciu a rezistibilitu.

Paralelný test rezistencie a rezistibility jedného grafu. V prípade, že by sme potrebovali otestovať len jeden ale veľký graf, mohli by sme funkcie upraviť nasledovne. Funkcia *prepareData(dataOut)* by do odchádzajúcej správy pridala vždy ten istý graf a k nemu hranu (alebo vrchol), ktorej rezistibilitu chceme vypočítať. Takto by sme postupovali pre všetky hrany (vrcholy) grafu. Funkcia *processResults(dataIn)* by prijala výslednú rezistibilitu a pridala tento výsledok k zodpovedajúcej hrane (vrcholu). Pre pracovný uzol by funkcia *processResults(dataIn)* načítala graf z g6 formátu a následne by ho otestovala na rezistibilitu hrany uvedenej v prijatej správe.

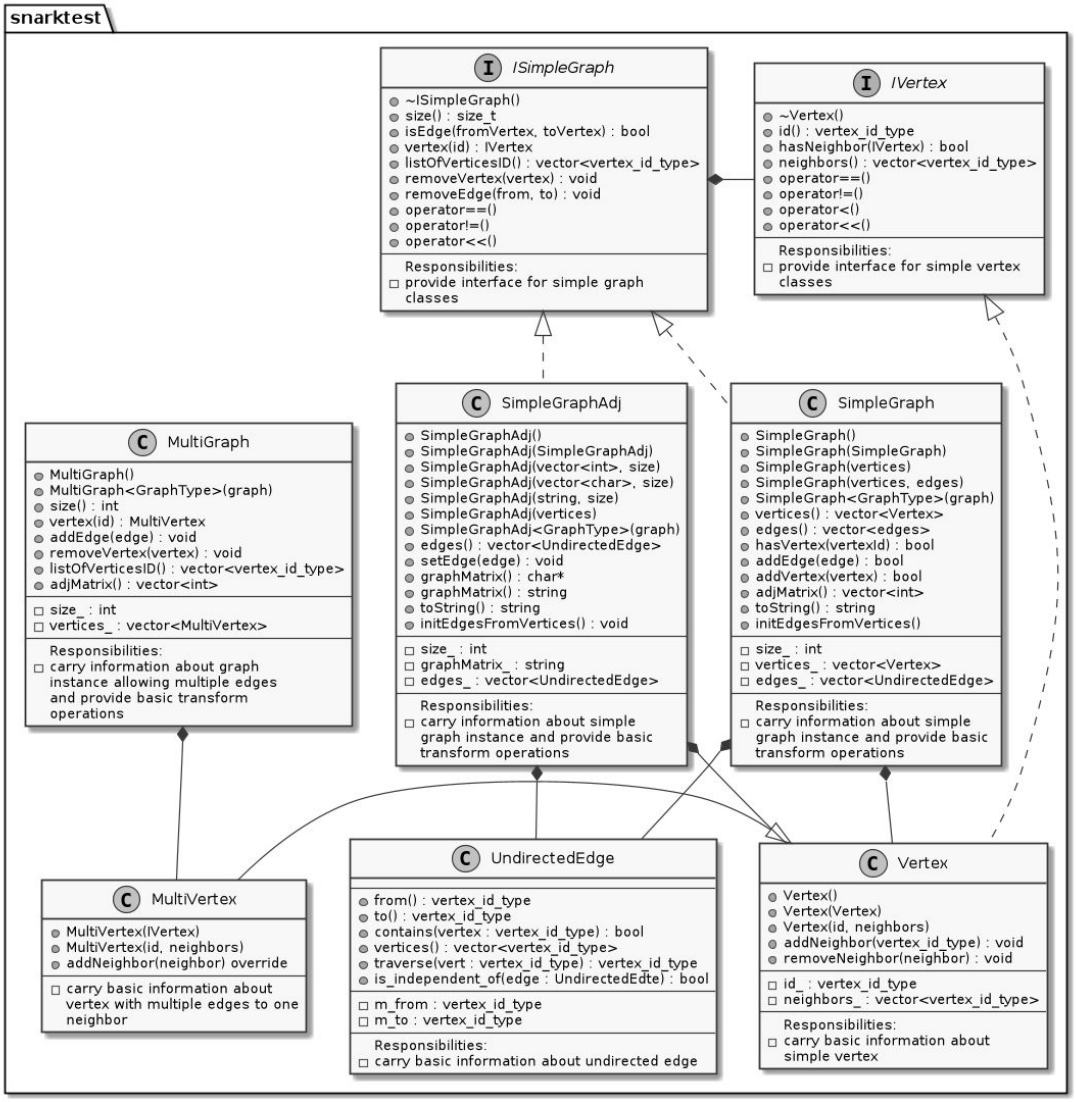
Keď máme definované vývojové prostredie s potrebnými technológiami podporujúce paralelizmus a aj samotné algoritmy, ktoré bude potrebné implementovať, môžeme postúpiť k návrhu tried a modelu systému ako celku.

2.3.3 Diagram tried

Návrh implementácie systému budeme modelovať pomocou jazyka UML a diagramov tried. Pri návrhu tried systému budeme postupovať od najmenších stavebných prvkov až po triedy implementujúce algoritmy a ich kombinácie. Tiež sa budeme snažiť nasledovať *SOLID* princípy návrhu [47]. Keďže budeme systém implementovať v jazyku C++, pre prehľadnosť a nekonfliktnosť všetky triedy obalíme do *namespace snartest*.

Graf. Základnou stavebnou jednotkou celého systému bude trieda reprezentujúca graf. Najprv definujeme rozhranie *ISimpleGraph*, ktoré takáto trieda musí spĺňať. Ďalej tu budeme mať implementácie s názvom *SimpleGraph* a *SimpleGraphAdj*. Tie sa skladajú z tried *Vertex*, ktorá implementuje rozhranie *IVertex* a reprezentuje vrchol grafu a *UndirectedEdge*, ktorá reprezentuje hranu grafu. Trieda *SimpleGraph* bude mať graf interne reprezentovaný ako zoznam susednosti, trieda *SimpleGraphAdj* ho bude mať reprezentovaný ako maticu susednosti. Pre algoritmy ako cyklická hranová súvislosť grafu budeme potrebovať aj graf umožňujúci násobné hrany. Preto definujeme aj triedu *MultiGraph*, ktorá bude reprezentovať práve takýto typ grafu. Diagram týchto tried vidíme na Obr. 19.

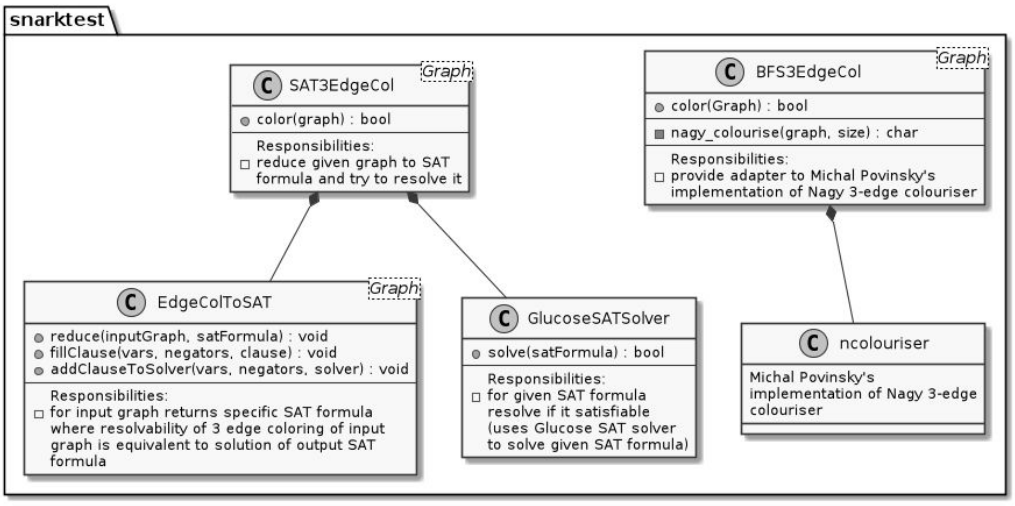
Ofarbovače. Ďalšou základnou a nevyhnutnou časťou systému budú triedy nesúce implementácie ofarbovacích algoritmov. Pre účely práce budeme venovať pozornosť dvom implementáciám ofarbovacích algoritmov. Jednou z nich bude rekurzívny ofarbovač na základe práce Karola Nagya [50], ktorý implementoval Michal Povinský [2].



Obr. 19: Rozhranie tried nesúcich informácie o grafe (resp. graf samotný).

Tento ofarbovač dokáže pracovať len s grafom reprezentovaným maticou susednosti vo forme ukazovateľa na pole znakov. Preto ho obalíme triedou pre kompatibilitu s triedami implementujúcimi rozhranie *ISimpleGraph*. Druhou implementáciou je algoritmus využívajúci redukciu na SAT formulu a jej riešenie spomenutý v Sekcii 2.2. Riešenie SAT formuly bude zabezpečovať implementácia *Glucose SAT solver* z [14], ktorá je založená na *MiniSAT* implementácii z [28]. Diagram týchto tried vidíme na Obr. 20.

Čítanie, zápis a transformácie. Keď máme definované triedy nesúce graf, potrebujeme byť schopní dostať inštanciu grafu z externého zdroja. Napríklad extrahovať graf vo formáte g6, ktorý dostaneme vo vstupnom súbore. Rovnako budeme neskôr potrebovať graf zakódovať naspäť do tohto formátu. O tieto úkony sa starajú triedy implementujúce rozhrania *IReader a IWriter*, ktoré môžeme vidieť na Obr. 21. Tiež tu uvádzame triedu, ktorá realizuje základné transformácie grafu ako odstránenie hrany



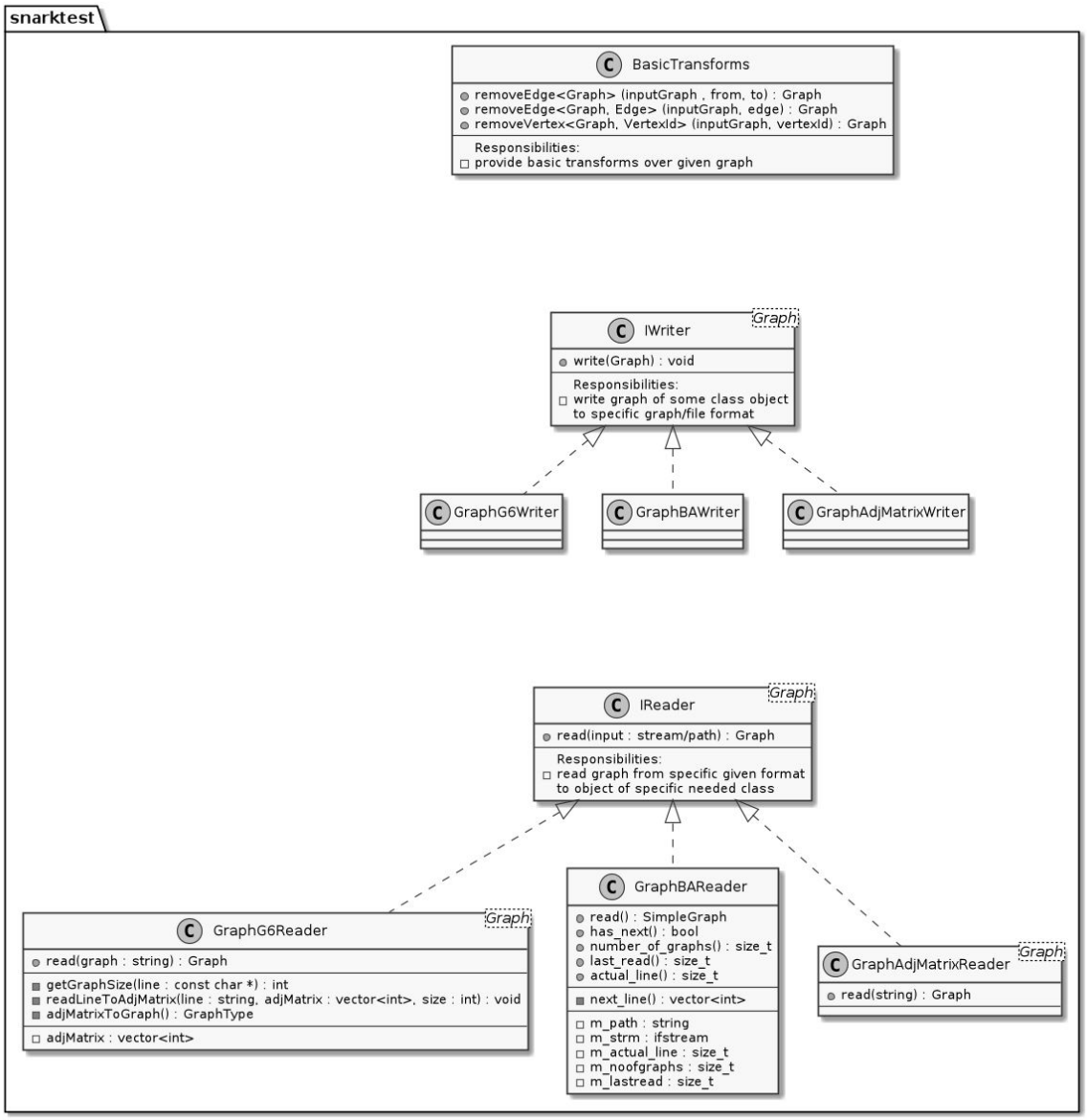
Obr. 20: Hranové ofarbovanie grafu.

alebo vrchola. Táto trieda vždy vráti nový graf, ktorý neobsahuje špecifikovanú hranu alebo vrchol.

Testy vlastností. Jadro celého systému pozostáva z tried, ktoré implementujú algoritmy pre testovanie vlastností kubických grafov. Nosné sú triedy pre testovanie hranovej rezistencie *EdgeResistance*, testovanie obvodu *Girth* a zrejme najdôležitejšia hierarchia tried pre testovanie chromatických vlastností *SubcriticalProperties, CriticalProperties a ChromaticProperties*. Ako môžeme vidieť na Obr. 22 prvá zo spomenutých tried testuje len kritickosť a vrcholovú a hranovú subkritickosť. Ďalšia trieda *CriticalProperties* dokáže testovať aj kokritickosť a posledná trieda *ChromaticProperties* testuje všetky vybrané chromatické vlastnosti vrátane stability a kostability. Trieda *CyclicEdgeConnectivity* bude obsahovať implementáciu algoritmu pre testovanie cyklickej hranovej súvislosti z [26]. Tento algoritmus si vyžaduje tiež algoritmus pre hľadanie maximálneho toku v grafe, ktorý bude v triede *MaxFlow*. Posledná trieda *TestForSetOfGraphs*, ktorá je priamo použitá aj v ďalšej nadväzujúcej vrstve má jednoduchú úlohu. V cykle pre každý riadok vstupného prúdu načíta graf, otestuje na ňom potrebné vlastnosti a zapíše ich do výstupného objektu s výsledkami.

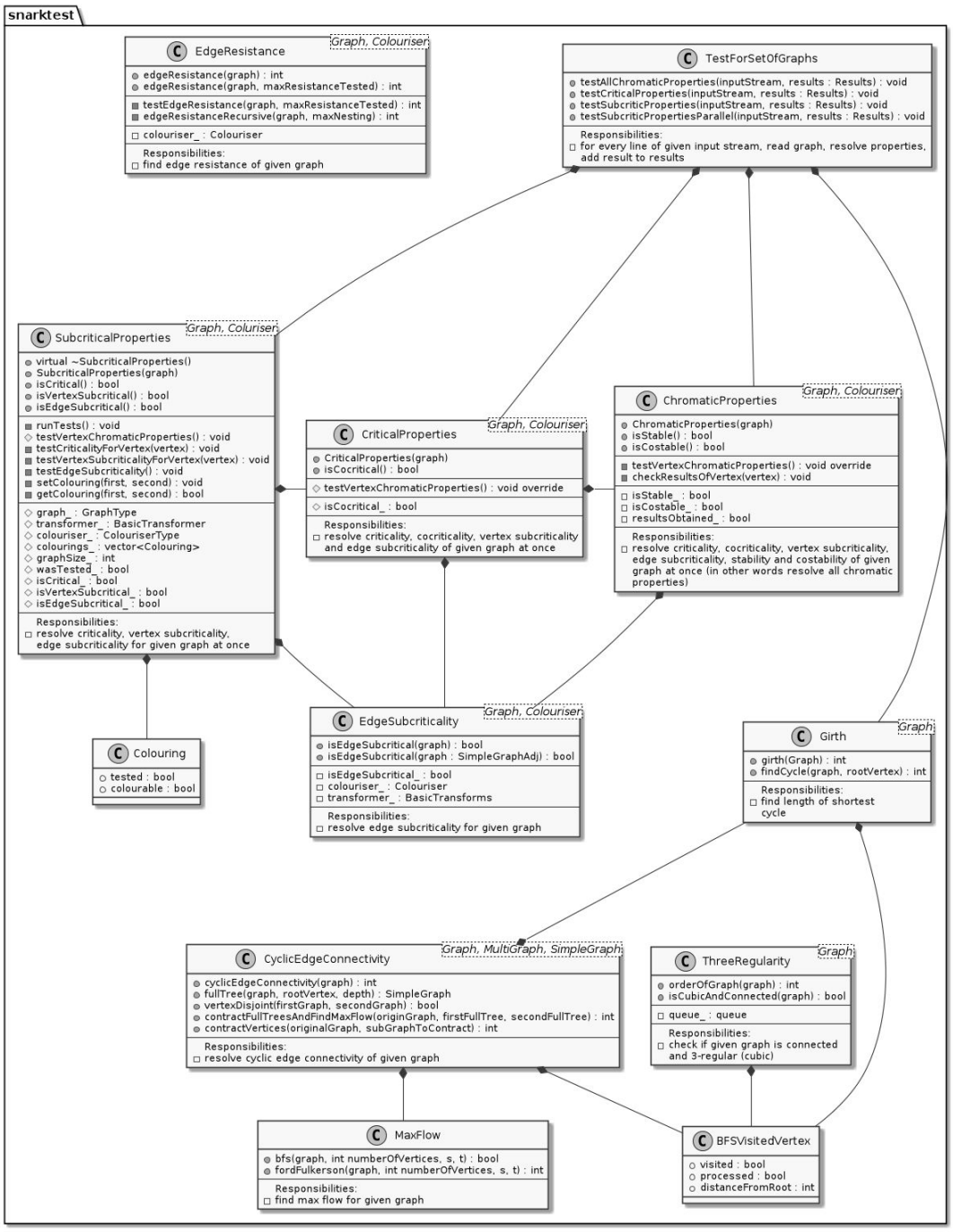
Väčšina spomínaných tried bude mať parameter pre generické programovanie. Túto generiku budeme v implementácii realizovať prostredníctvom *template* tried a metód. Takýto postup sme zvolili jednak pre väčšiu použiteľnosť s neskôr definovanými triedami a tiež aby sme sa v rámci optimality vyhli virtuálnym volaniam metód, ktoré by vyžadoval polymorfizmus.

Výsledky testov. Po vykonaní testov vlastností grafu budeme potrebovať nejakým spôsobom spracovať a uchovať výsledky. Trieda *GraphProperties* bude uchovávať všetky



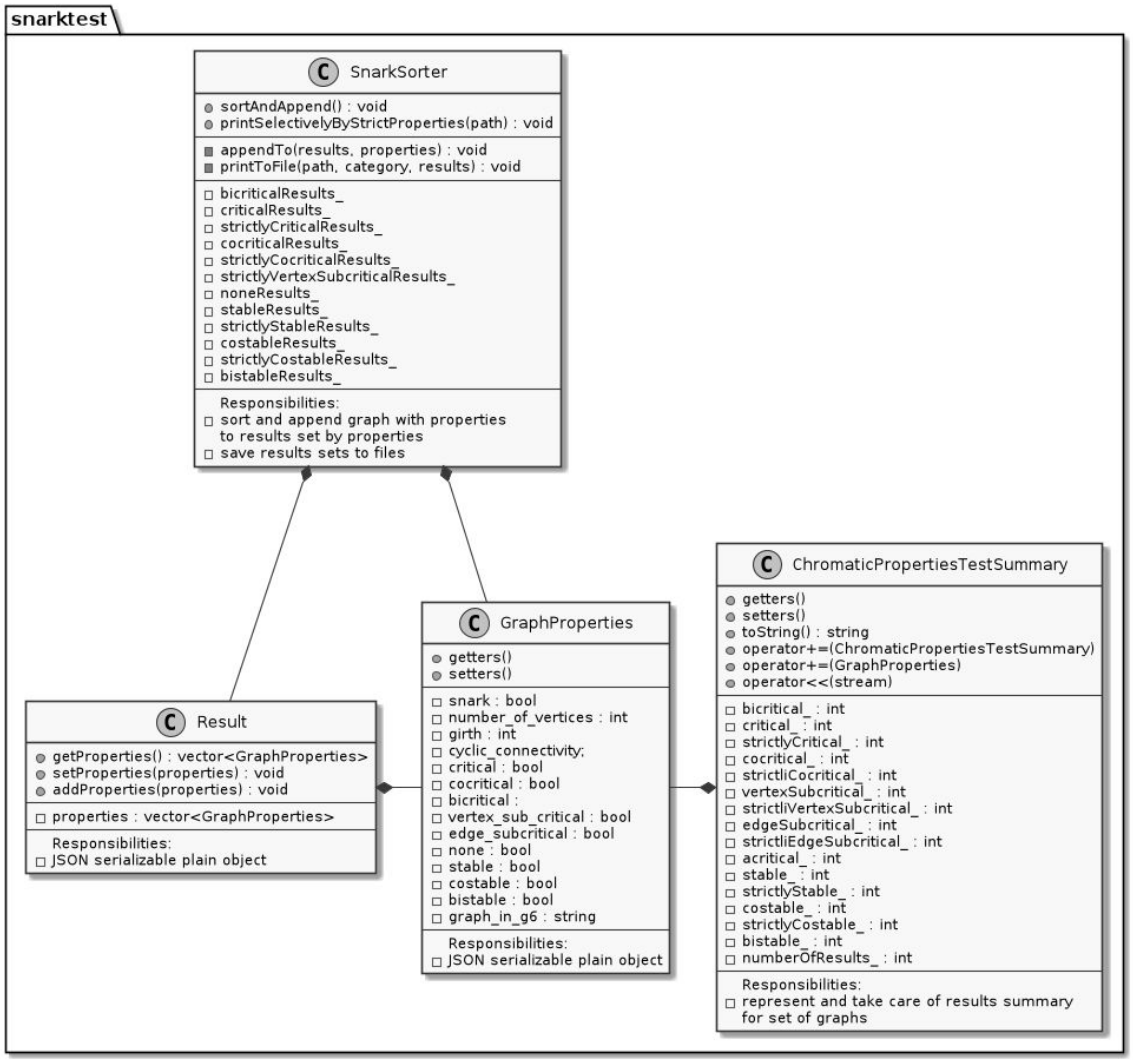
Obr. 21: Transformácie grafu, čítanie grafu zo súboru, zápis grafu do súboru.

chromatické a niektoré ďalšie vlastnosti o grafe ako aj graf samotný v g6 formáte. Objekty tejto triedy bude možné serializovať do formátu JSON. Túto funkcionalitu využijeme hneď dvakrát, pri posielaní výsledkov pomocou MPI rozhrania a tiež v budúcnosti pri ukladaní výsledkov do nerelačnej databázy. Trieda *ChromaticProperties-TestSummary* bude reprezentovať sumár testu množiny grafov. V podstate bude len nosiť informáciu o početnosti grafov s jednotlivými vlastnosťami a tiež celkový počet testovaných grafov. Posledná trieda *SnarkSorter* bude slúžiť na triedenie a zhromaždenie vybraných skupín grafov podľa určených vlastností. Tiež bude možné tieto skupiny grafov zapísať do samostatných súborov. Diagram týchto tried môžeme vidieť na Obr. 23.



Obr. 22: Testovanie vlastností grafu.

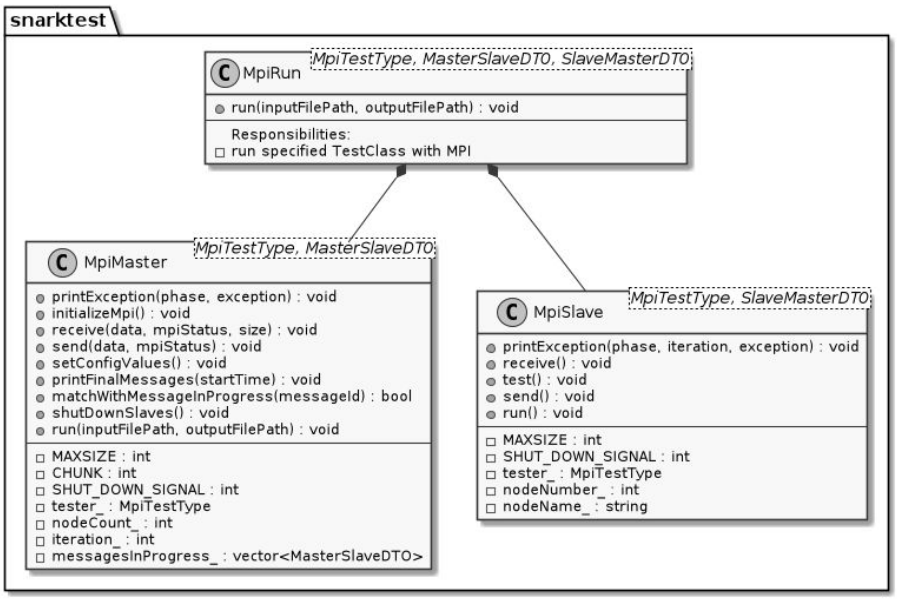
MPI algoritmus. Na Obr. 24 môžeme vidieť triedy implementujúce zovšeobecnený paralelný algoritmus popísaný v predchádzajúcej časti 2.3.2. Táto implementácia využíva MPI rozhranie pre komunikáciu jednotlivých uzlov výpočtového prostredia. Trieda *MpiMaster* implementuje Algoritmus 10 a zabezpečuje moderovanie celého výpočtu. Trieda *MpiSlave* obdobne implementuje Algoritmus 11 a stará sa o vykonanie zadaných úloh. Celý paralelný výpočet sa spúšťa použitím triedy *MpiRun*, ktorá obsahuje len jednu funkciu. Táto funkcia zabezpečí inicializáciu MPI prostredia, rozdelenie rolí uzlom a



Obr. 23: Triedy pre výsledky testov chromatických vlastností.

ukončenie prostredia. Pre túto triedu je potrebné zadať ako parameter triedu, ktorá špecifikuje spôsob vykonania paralelného výpočtu, podobne ako to bolo načrtnuté v Sekcii 2.3. Rovnako treba pre použitie triedy *MpiRun* definovať formu správ, ktoré budú posielané z hlavného uzlu pracovnému uzlu a aj naopak. Forma týchto správ bude reprezentovaná triedami s názvom *MasterSlaveDTO* a *SlaveMasterDTO* (DTO pre „data transfer object“). Príklady takýchto tried si ukážeme na Obr. 25.

MPI špecializácie. Špecializáciu všeobecného MPI paralelného algoritmu uskutočníme pomocou tried, ktoré musia implementovať metódy definované v rozhraní *IMpiTest*. Práve danými metódami určíme akým spôsobom bude hlavný uzol rozdeľovať úlohy a spracovávať ich výsledky (metódy *masterPreProcess, masterReadData, masterProcessReceivedMessage, masterPostProcess*). Tiež pomocou nich určíme, aký úkon má vykonať pracovný uzol s prijatou správou (metóda *slaveDo*). Ďalšie metódy (*mas-*

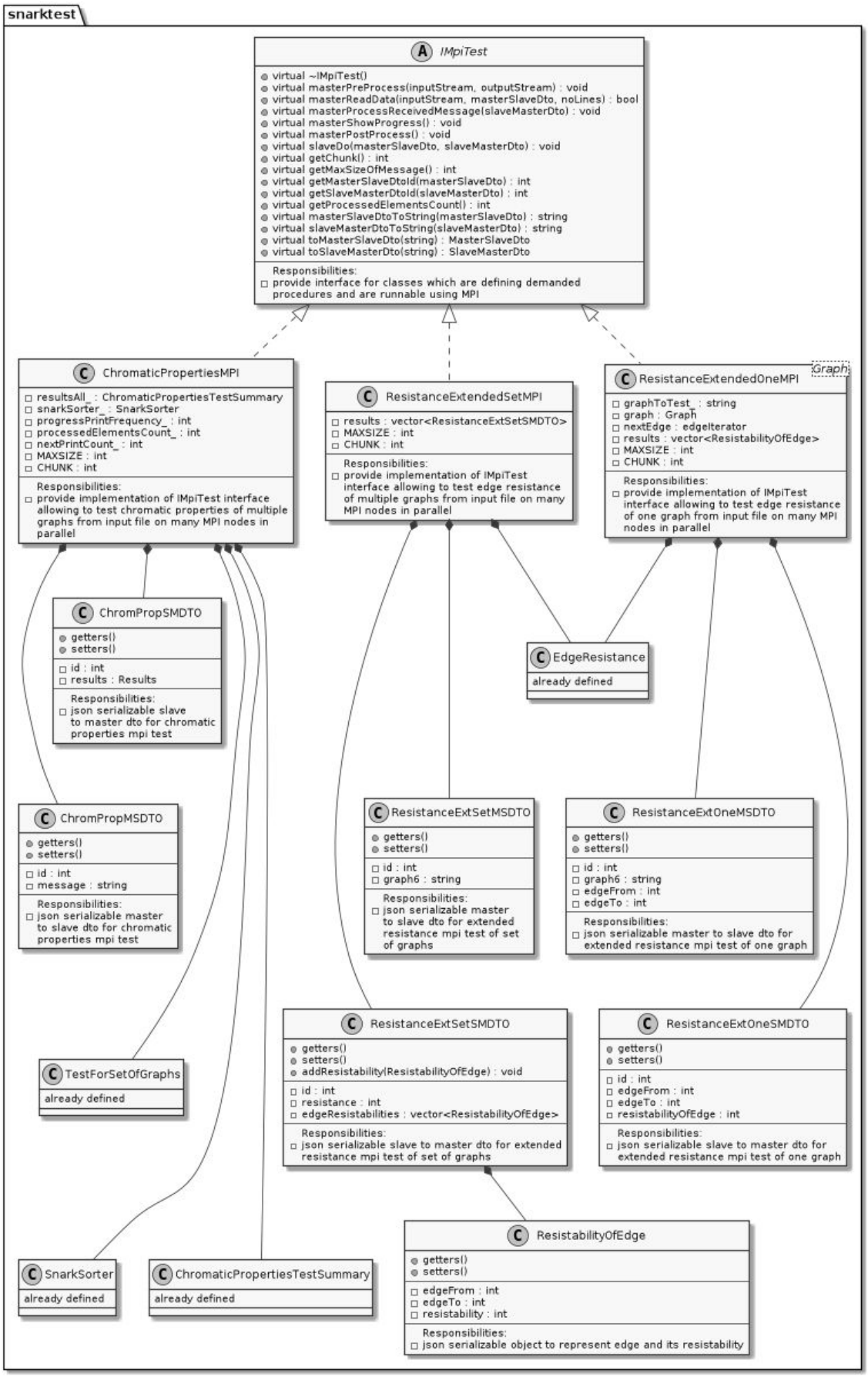


Obr. 24: MPI role a spúšťanie procedúr.

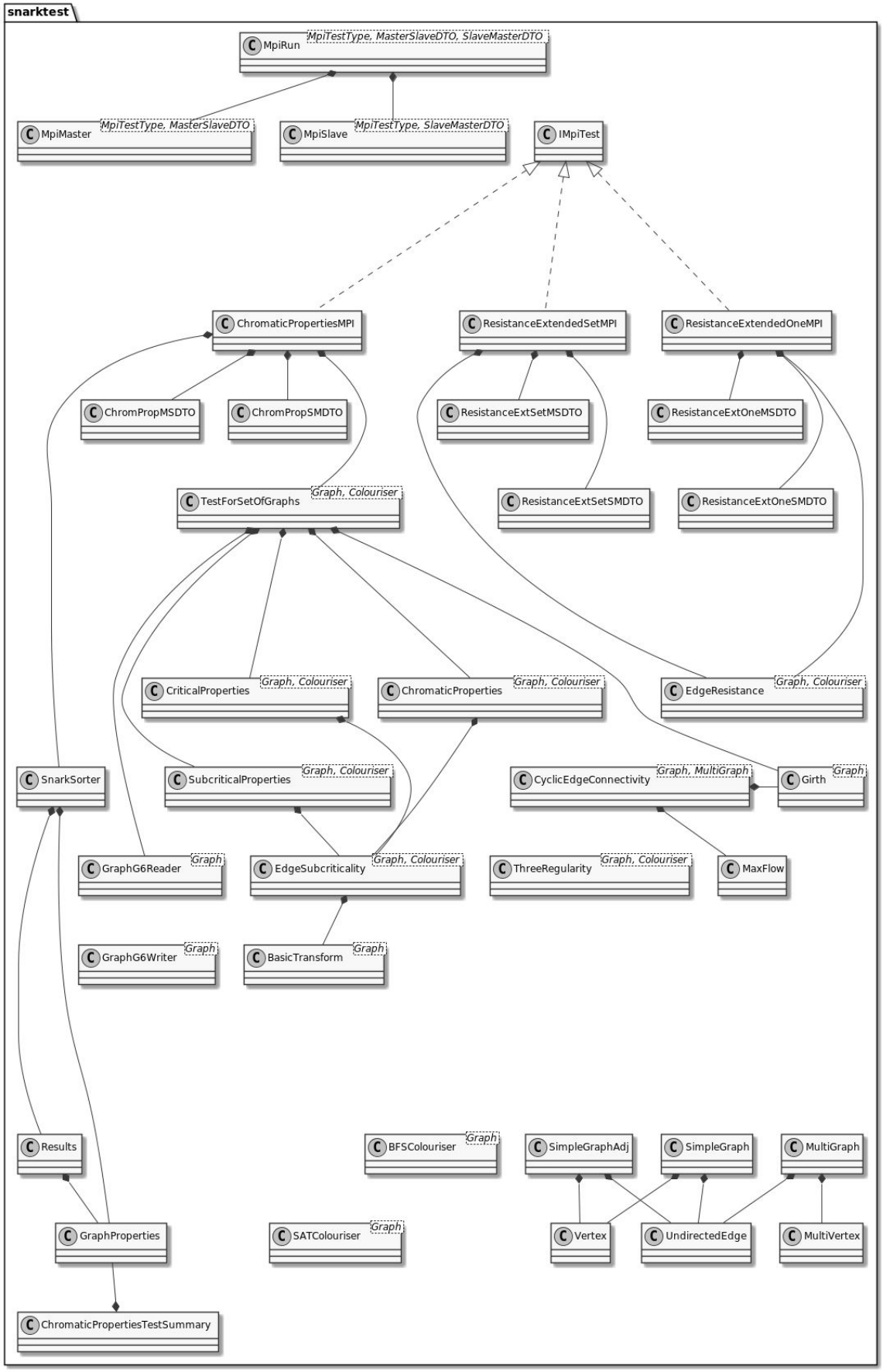
terSlaveDtoToString, slaveMasterDtoToString, toMasterSlaveDto, toSlaveMasterDto) určujú, akým spôsobom sa majú transformovať DTO objekty do reťazca znakov, ktorý je možné pomocou MPI poslať inému uzlu v správe. Alebo naopak, ako z prijatého reťazca znakov získame žiadaný DTO objekt. Zvyšnými metódami (napr. *getChunk, getMaxSizeOfMessage*) vieme MPI algoritmu poskytnúť dôležité alebo podporné informácie, ktorými môžeme ovplyvniť beh výpočtu. Napríklad *getChunk* metóda by mala vracať počet prvkov, ktoré chceme načítať a odoslať v jednej správe jednému pracovnému uzlu a teda definuje veľkosť jednej úlohy. Metóda *getMaxSizeOfMessage* definuje maximálnu veľkosť posielanej správy. Tento atribút je nevyhnutný pre správne čítanie MPI správy prijímateľom.

Ďalej potrebujeme definovať už spomenuté DTO triedy reprezentujúce správy medzi týmito dvoma uzlami. Na Obr. 25 môžeme vidieť špecifikácie pre test chromatických vlastností množiny grafov, test rozšírenej hranovej rezistencie množiny grafov a test rozšírenej hranovej rezistencie jedného grafu. Všetky tieto špecifikačné triedy nasledujú ideu z návrhu paralelného algoritmu zo Sekcie 2.3.

Súhrn všetkých dôležitých tried a ich vzťahy môžeme vidieť na Obr. 26.



Obr. 25: Rozhranie pre MPI beh procedúr.



Obr. 26: Diagram tried - hlavné časti

2.4 Implementácia, testovanie, nasadenie systému a merania

Implementácia systému prebiehala postupne vo viacerých fázach. V prvej fáze sme implementovali jednotlivé triedy pre graf a ofarbovacie algoritmy. Až postupne sme dostali výslednú formu tried ako je zobrazená v predchádzajúcej podkapitole.

V ďalšej fáze sme implementovali test subkritických vlastností opísaný v Prílohe D. Následne sme implementovali prvú verziu MPI paralelného algoritmu a otestovali ho spolu s testom subkritických vlastností. Ďalej sme implementovali algoritmus pre test kritických vlastností a test všetkých chromatických vlastností uvedený v 2.2.2. Implementácie sme otestovali lokálne na jednom CPU. Následne, pre vyladenie MPI konfigurácie, sme ich testovali aj na cvičnom klastri pozostávajúcom z deviatich uzlov obsahujúcich dokopy 20 výpočtových jadier. Hlavné behy systému boli potom uskutočnené na klastri HPCC UMB [6]. Niektoré z meraní jednotlivých testovacích výpočtov uvádzame v Tabuľke 2.

Z nameraných časov trvania môžeme pozorovať nasledovné zistenia. Počet grafov obsiahnutý v jednej správe (jednej úlohe) odoslanej z hlavného uzlu pracujúcemu uzlu nezohráva povšimnuteľnú úlohu. Ďalej si môžeme všimnúť rozdiel trvania testu subkritických vlastností a testu chromatických vlastností. Zložitosť obidvoch algoritmov sme uviedli v 12. Aj keď oba algoritmy majú rovnakú teoretickú zložitosť, z nameraných časov môžeme usúdiť, že kritické vlastnosti (kritickosť, kokritickosť a subkritickosť) grafu sa zisťujú rýchlejšie ako stabilita a kostabilita.

Ďalšiu skutočnosť, ktorú si môžeme všimnúť je rozdiel pri jednotlivých ofarbovacích algoritmoch. Z našej skúsenosti sa ukazuje, že nami použitá implementácia rekurzívneho algoritmu (v tabuľke ako „bfs“) je rýchlejšia na menších grafoch. Ešte pri grafoch o veľkosti 40 vrcholov je tento algoritmus dvojnásobne rýchlejší ako „sat“. Ak však testujeme väčšie grafy, ukazuje sa, že naša implementácia algoritmu používajúceho redukciu na SAT (v tabuľke ako „sat“) získava prevahu (test ofarbiteľnosti grafu o 76 vrcholoch je viac ako 100-násobne rýchlejší pri sat ofarbovači).

Zdroj testovaných inštancií je dostupný na webovej adrese [5]. Technické parametre jednotlivých strojov, na ktorých sme systém testovali sú nasledovné:

- jednotkou „l“ označujeme jeden CPU Intel Core i5 1.9GHz so 4 vláknami
- jednotkou „ck“ označujeme cvičný klaster pozostávajúci z 9 CPU a spolu 20 jadrami
- jednotkou „kn“ označujeme jadrá procesorov Intel Xeon E5-2670 (2.6 - 3.3GHz) na klastri HPCC UMB

- jednotkou „ks“ označujeme jadrá starších procesorov Intel Xeon X5670 (2.93 - 3.33GHz) tiež na klastri HPCC UMB

Podrobnejšiu špecifikáciu hardvéru klastra HPCC UMB môžeme nájsť na [6].

Pri implementácii DTO objetov používaných pre posielanie správ v MpiMaster a MpiSlave algoritmoch sme na transformáciu objektov do JSON formátu použili C++ knižnicu nlohmann::json dostupnú na [8].

typ výpočtu	**o**	**cpu**	**k**	**n**	**t**
subkritické vlastnosti	sat	4 l	28 399	30	3 611s
subkritické vlastnosti - po 10 graf.	sat	20 ck	28 399	30	1 363.3s
subkritické vlastnosti - po 1 grafe	sat	20 ck	28 399	30	1 304s
subkritické vlastnosti	bfs	4 l	28 399	30	380.2s
subkritické vlastnosti + optim.	bfs	4 l	28 399	30	350.5s
kritické vlastnosti	bfs	4 l	28 399	30	373.5s
chromatické vlastnosti	bfs	4 l	28 399	30	910.4s
chromatické vlastnosti	bfs	1 l	25	40	4.4s
chromatické vlastnosti	sat	1 l	25	40	9.2s
test ofarbiteľnosti	bfs	1 l	1	76	0.858s
test ofarbiteľnosti	sat	1 l	1	76	0.006s
chromatické vlastnosti	bfs	112 kn	28 399	30	9s
chromatické vlastnosti	bfs	112 kn	139 854	30	47.6s
chromatické vlastnosti	bfs	112 kn	293 059	32	193.3s
chromatické vlastnosti	bfs	112 kn	$> 1.7mil$	32	1065s
chromatické vlastnosti	bfs	112 kn	$> 3.8mil$	34	1h 41m 14s
chromatické vlastnosti	bfs	-kn	$> 25mil$	34	4h 31m 35s
chromatické vlastnosti	bfs	208 kn	$> 60mil$	36	31h 23m 1s
chromatické vlastnosti	bfs	155 kn + 136 ks	$> 404mil.$	36	6d 2h 55m

Tabuľka 2: Merania testovacích výpočtov. Legenda: **o**=ofarbovací algoritmus, **cpu**=počet výpočtových jadier a ich špecifikácia, **k**=počet grafov, **n**=počet vrcholov grafu, **t**=čas.

Na záver sme implementovali test hranovej a vrcholovej rezistencie grafu s rezistibilitou hrán a vrcholov. Spolu s nimi sme implementovali aj súvisiace triedy pre paralelizáciu pomocou MPI. Merania z testov tejto funkcionality môžeme vidieť v Tabuľke 3.

Ukážku implementácie hlavnej funkcie *run* triedy *MpiMaster* môžeme vidieť na Obr. 27. Podobne implementáciu hlavnej funkcie triedy *MpiSlave* môžeme vidieť na Obr. 28.

typ výpočtu	o	cpu	k	n	t
hr. r+r	bfs	4 l	413	34	136.3s
vr. r+r	bfs	4 l	413	34	49s
hr. r+r	bfs	4 l	3702	36	2037.5s
vr. r+r	bfs	4 l	3702	36	703s
hranová rezistencia	sat	4 l	1	76	84.3s
hr. r+r po 1 grafe na uzol	bfs	31 kn	31	44	911.3s
vr. r+r po 1 grafe na uzol	bfs	31 kn	31	44	112.8s
hr. r+r po 1 hrane na uzol	sat	114 kn	1	76	125.6s
vr. r+r po 1 hrane na uzol	sat	114 kn	1	76	49.6s

Tabuľka 3: Merania testu hranovej rezistencie. Legenda zhodná s tabuľkou 2, **hr. r+r** pre hranovú a **vr. r+r** pre vrcholovú rezistenciu a rezistibilitu.

Ďalší vývoj systému

Pre komplexnosť by do systému neskôr mohli pribudnúť ďalšie testy a funkcie ako napríklad konštrukcie snarkov cez všetky uvedené možnosti uvedené v Prílohe E, test izomorfizmu, cyklická súvislosť s identifikáciou 4-rezov a 5-rezov a identifikácia 5-cyklov a klastrov 5-cyklov [63].

Tiež by systém neskôr mohol podporovať grafické rozhranie a vizualizáciu grafov. Pre tento účel by mohli byť nápomocné napríklad knižnice typu *OGDF (Open Graph Drawing Framework)* alebo vizualizačný softvér pre grafy *GraphViz* [4, 11, 29].

Ďalej by malo zmysel preskúmať aké percento ofarbovaných podgrafov, napríklad pri testovaní chromatických vlastností, je ofarbiteľných. Je pravdepodobné, že medzi podgrafmi bude veľmi vysoké percento práve ofarbiteľných grafov. Preto by zrejme malo význam implementovať ofarbovaciu heuristiku z [32], ktorá pri ofarbiteľných grafoch vie veľmi rýchlo rozhodnúť, že sú ofarbiteľné. Takýmto spôsobom by sa mohol celý výpočet urýchliť.

```
/** ============== MPI MASTER ============== */
void run(const std::string &inputFilePath,
         const std::string &outputFilePath) {
  // open input file
  std::ifstream inputFile(inputFilePath);
  if (!inputFile) throw std::invalid_argument("Input file not found!");
  // open output file
  std::ofstream outputFile(outputFilePath + "summary");
  if (!outputFile) throw std::invalid_argument("Output file path not valid!");

  tester.masterFirst(inputFile, outputFile);
  outputFile.close();

  init();

  MPI_Status stat;
  int size = -1;
  char data[MAXSIZE];
  int sentMessagesCount = 0;
  int receivedMessagesWithResults = 0;

  iteration_ = -1;
  while (true) {
    ++iteration_;

    //// READ DATA TO SEND phase
    MasterSlaveDTO dataOut{};
    try {
      if (!tester.readDataForMaster(inputFile, dataOut, CHUNK, iteration_))
        break;
    } catch (std::exception &exception) {
      printException("READ DATA TO SEND", exception);
      continue;
    }

    //// RECEIVE phase
    // receive message (with results) from available(free) node
    try {
      receive(data, stat, MAXSIZE);
    } catch (std::exception &exception) {
      printException("RECEIVE", exception);
      continue;
    }

    //// SEND phase
    send(dataOut, stat);

    receivedMessagesWithResults += stat.MPI_TAG;
    ++sentMessagesCount;

    tester.showProgressMaster(CHUNK);
  }
  std::cout << "MASTER --- end of file !!! \n";

  //// GET REST OF RESULTS phase
  while (!messagesInProgress_.empty()) {
    try {
      receive(data, stat, MAXSIZE);
    } catch (std::exception &exception) {
      printException("GET REST OF RESULTS", exception);
    }
  }

  shutDownSlaves();
  tester.masterFinal(outputFilePath);
  printFinalMessages();
}
```

Obr. 27: Implementácia hlavnej funkcie run() v triede MpiMaster.

```
/** ============== MPI SLAVE ============== */
void run() {
  MAXSIZE = tester.getMaxSizeOfMessage();
  MPI_Status stat;
  int nodeNameLength;

  MPI_Comm_rank(MPI_COMM_WORLD,
                &nodeNumber_); /* and this processes' rank is */
  MPI_Get_processor_name(nodeName_, &nodeNameLength);

  int size = 0;
  char data[MAXSIZE];

  // send that I am available
  MPI_Send(std::string{}.c_str(), 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD);

  int iteration = -1;
  while (true) {
    ++iteration;
    //// RECEIVE phase
    try {
      // receive new job
      MPI_Recv(&data, MAXSIZE, MPI_CHAR, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
               &stat);
      MPI_Get_count(&stat, MPI_CHAR, &size);

      if (stat.MPI_TAG == SHUT_DOWN_SIGNAL) break;
    } catch (std::exception &exception) {
      printExceptionAndSendMPIAvailableMessage("RECEIVE", iteration,
                                               exception);
      continue;
    }

    //// TEST phase
    SlaveMasterDTO results{};
    try {
      std::string received(data, static_cast<unsigned long>(size));
      tester.test(tester.toMasterSlaveDTO(received), results);
    } catch (std::exception &exception) {
      printExceptionAndSendMPIAvailableMessage("TEST", iteration, exception);
      continue;
    }

    //// SEND phase
    std::string message = tester.slaveMasterDTOtoString(results);
    if ((message.length()) > MAXSIZE) {
      std::cerr << message.length() << "\n";
      throw std::runtime_error("Message bigger than defined MAXSIZE");
    }

    try {
      MPI_Send(message.c_str(), static_cast<int>(message.size()), MPI_CHAR, 0,
               1, MPI_COMM_WORLD);
    } catch (std::exception &exception) {
      printExceptionAndSendMPIAvailableMessage("SEND", iteration, exception);
      continue;
    }
  }
}
```

Obr. 28: Implementácia hlavnej funkcie run() v triede mpiSlave.

Záver

Po implementácii navrhnutého systému, sme otestovali všetky existujúce snarky do veľkosti 36 vrcholov na chromatické vlastnosti opísané v Kapitole 1. Výber z výsledkov ktoré sme získali uvádzame v Tabuľke 4. Niektoré z nami získaných poznatkov už boli predtým známe [5, 17, 51]. Zo získaných výsledkov sú zrejmé nasledovné pozorovania.

Pozorovanie 2.1. *Všetky kritické aj kokritické snarky do rádu 36 majú obvod aspoň päť.*

Výsledky ďalej ukazujú existenciu akritických snarkov, ktoré sú mimoriadne vzácne. Napríklad v triede snarkov rádu 30 (z celkového počtu 139 854) existuje len 12 takýchto grafov.

Pozorovanie 2.2. *Najmenší akritický snark má 28 vrcholov.*

Akritické snarky rádu 28 sú len tri. Pomocou nástroja na zisťovanie izomorfizmu *nauty and Traces* [10] sme medzi týmito tromi grafmi identifikovali snark *5FLOW28* (neoficiálny názov). Tento snark skonštruovali E. Máčajová a A. Raspaud v [49], kde zohral významnú úlohu a môžeme ho vidieť na Obr. 29.

Tiež sme zistili, že do veľkosti 36 vrcholov neexistuje žiadny kostabilný snark a teda ani bistabilný.

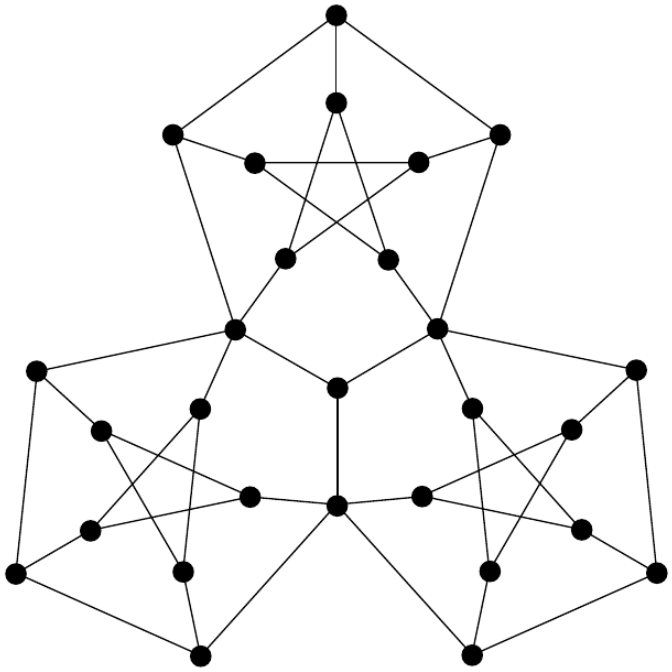
Problém 2.1. *Aký najmenší kostabilný snark existuje?*

V testovanej vzorke tiež môžeme pozorovať tendenciu početnosti bikritických grafov. Tá je pri grafoch, ktorých počet vrcholov $|V| \equiv 0 \, mod \, 4$ výrazne nižšia ako početnosť bikritických grafov, ktorých počet vrcholov $|V| \equiv 2 \, mod \, 4$. Bolo by zaujímavé sledovať, či sa táto tendencia zachová aj pri množinách väčších grafov.

Skúsme sa teraz viac zamerať na triedu akritických grafov.

Lema 2.1. *Akritický snark obsahuje aspoň jeden vrchol a jednu hranu s rezistibilitou aspoň tri.*

Dôkaz. Akritický snark G z definície nie je vrcholovo subkritický. To znamená, že musí existovať aspoň jeden vrchol v, pre ktorý neexistuje vrchol u taký, že $\{u,v\}$ tvorí



Obr. 29: Snark 5FLOW28 [31].

neodstrániteľnú dvojicu. Keďže vrcholová subkritickosť je implikovaná hranovou subkritickosťou, ak graf nie je vrcholovo subkritický, nemôže byť ani hranovo subkritický. Ak graf nie je hranovo subkritický, obsahuje aspoň jednu hranu e, pre ktorú neexistuje hrana f taká, že $\{e,f\}$ tvorí neodstrániteľnú dvojicu. □

Problém 2.2. *Aká je rezistencia a index hranovej a vrcholovej rezistibility akritických grafov?*

Po identifikovaní všetkých akritických snarkov do rádu 36 sme skúmali aj rezistenciu a indexy hranovej a vrcholovej rezistibility týchto grafov. Zhrnutie našich zistení uvádzame v Tabuľke 5. Ukázalo sa, že všetky akritické snarky do rádu 36 majú rezistenciu dva a index hranovej rezistibility každého z nich je aspoň tri.

Pozorovanie 2.3. *Všetky hrany aj vrcholy všetkých akritických snarkov do rádu 36 majú rezistibilitu menšiu ako štyri.*

Zaujímavý je aj vzťah akritickosti a všetkých stabilných snarkov našej vzorky, ktorých je spolu 75.

Pozorovanie 2.4. *Všetky stabilné snarky do rádu 36 sú akritické.*

Miera rezistencie väčšiny týchto stabilných snarkov je však ku podivu veľmi nízka.

Pozorovanie 2.5. *Všetky stabilné snarky rádu 34 a až 54 z 58 stabilných snarkov rádu 36 majú* $v_{ri} = 1$ *a* $e_{ri} = 3$. *Jeden stabilný snark rádu 36 má* $v_{ri} = 4$ *a* $e_{ri} = 14$.

n	g	k	bikrit	skrit	skokrit	svsubk	akrit	st	kost
10	≥ 5	1	1	0	0	0	0	0	0
18	≥ 5	2	2	0	0	0	0	0	0
20	≥ 5	6	1	0	0	0	0	0	0
22	≥ 4	31	2	0	0	0	0	0	0
22	≥ 5	20	2	0	0	0	0	0	0
24	≥ 4	155	0	0	2	0	0	0	0
24	≥ 5	38	0	2	0	0	0	0	0
26	≥ 4	1 297	111	0	2	2	0	0	0
26	≥ 5	280	111	0	2	2	0	0	0
28	≥ 4	12 517	33	0	2	6	3	0	0
28	≥ 5	2 900	33	0	2	6	3	0	0
30	≥ 4	139 854	115	0	0	35	12	0	0
30	≥ 5	28 399	115	0	0	21	8	0	0
32	≥ 4	1 764 950	13	16	340	196	70	0	0
32	≥ 5	293 059	13	16	340	45	30	0	0
34	≥ 4	25 286 953	40328	2	429	1887	413	7	0
34	≥ 5	3 833 587	40328	2	429	484	66	7	0
36	≥ 4	404 899 916	13720	828	364	19316	3702	58	0
36	≥ 5	60 167 732	13720	828	364	5485	953	25	0

Tabuľka 4: Početnosť snarkov podľa vybraných chromatický vlastností. Legenda: **n** = počet vrcholov, **g** = obvod grafu, **k** = počet všetkých snarkov, **skrit** = striktne kritické snarky, **skokrit** = striktne kokritické snarky, **svsubk** = striktne vrcholovo subkritické snarky, **akrit** = akritické snarky, **st** = stabilné snarky, **kost** = kostabilné snarky

Zvyšné stabilné grafy sú však naopak mimoriadne rezistentné a majú najvyšší index vrcholovej aj hranovej rezistibility zo všetkých akritických grafov.

Pozorovanie 2.6. *Tri stabilné snarky rádu 36 majú* $v_{ri} = 20$ *a* $e_{ri} = 34$.

Pri zisťovaní rezistencie a rezistibility sme spracovali aj vzorku 31 grafov o veľkosti 44 vrcholov. Tieto grafy majú nepárnosť štyri a sú najmenšími snarkami s rezistenciou tri a cyklickou súvislosťou štyri [62]. Výsledky testov pre tieto grafy môžeme vidieť v Tabuľke 6.

Pozorovanie 2.7. *Napriek predpokladu všetkých 31 testovaných grafov rádu 44 obsahuje aj štvor-rezistibilné hrany no nie všetky obsahujú aj štvor-rezistibilné vrcholy.*

Posledný graf, ktorého rezistenciu a rezistibilitu sme skúmali je nám doposiaľ najmenší známy snark s cyklickou súvislosťou päť a nepárnosťou štyri [62]. Tento graf má 76 vrcholov.

početnosť podľa	**n = 28**	**n = 30**	**n = 32**	**n = 34**	**n = 36**
akritickosti	3	12	70	420	3760
r = 2	3	12	70	420	3760
r = 3	0	0	0	0	0
$v_{ri} = 1$	1	2	13	75	649
$v_{ri} = 2$	2	6	35	195	2011
$v_{ri} = 3$	0	0	0	0	44
$v_{ri} = 4$	0	4	12	70	410
$v_{ri} = 5$	0	0	0	0	0
$v_{ri} = 6$	0	0	10	36	203
$v_{ri} = 8$	0	0	0	44	180
$v_{ri} = 10$	0	0	0	0	260
$v_{ri} = 20$	0	0	0	0	3
$e_{ri} = 3$	1	2	13	75	616
$e_{ri} = 4$	0	0	0	0	33
$e_{ri} = 5$	2	6	35	195	1734
$e_{ri} = 6$	0	0	0	0	107
$e_{ri} = 8$	0	4	12	70	568
$e_{ri} = 10$	0	0	0	0	36
$e_{ri} = 11$	0	0	10	36	203
$e_{ri} = 12$	0	0	0	0	19
$e_{ri} = 14$	0	0	0	44	181
$e_{ri} = 17$	0	0	0	0	260
$e_{ri} = 34$	0	0	0	0	3

Tabuľka 5: Početnosť snarkov podľa veľkosti grafu a počtu rezistibilnejších hrán/vrcholov. Legenda: **r** = rezistibilita grafu, v_{ri} = index vrcholovej rezistablitity, e_{ri} = index hranovej rezistibility, **n** = počet vrcholov grafu

Pozorovanie 2.8. *Rezistencia skúmaného snarku rádu 76 je tri, pričom 12 zo 114 hrán grafu má rezistibilitu štyri no len jeden zo 76 vrcholov má rezistibilitu vyššiu ako je rezistencia grafu.*

Pre overenie výsledkov momentálne neexistuje nezávislá (ani žiadna iná) implementácia. Čiastočnú správnosť výpočtov sme sa však pokúsili overiť porovnaním dostupných informácií z iných zdrojov. Na [5] sú dostupné všetky kritické aj bikritické grafy do veľkosti 36. Množiny nami zistených kritických aj bikritických grafov sa úplne zhodujú s danými množinami z [5].

v_{ri}	0	1	2	3	4	-	-	-	-
počet grafov	6	18	4	0	3	-	-	-	-
e_{ri}	2	3	4	5	6	7	8	9	14
počet grafov	3	2	2	1	9	7	2	2	3

Tabuľka 6: Rezistibilita hrán a vrcholov vzorky grafov o veľkosti 44 vrcholov.

Resumé

V práci sme definovali snarky ako špeciálnu triedu kubických grafov a ich chromatické vlastnosti, o ktoré sa zaujímame. Postupne sme navrhli a implemetnovali systém, ktorý nám umožnil získať informácie o daných vlastnostiach z celej množiny snarkov rádu maximálne 36. Po analýze vybraných chromatických vlastností na danej vzorke grafov, sme poukázali na existenciu tzv. *akritických* snarkov. Identifikovali sme všetky tieto grafy z danej množiny a ďalej sme hlbšie skúmali ich rezistenčné vlastnosti. Tu sa ukázalo, že všetky akritické grafy do rádu 36 majú rezistenciu dva, no rezistibilita ich vrcholov a hrán nikdy nie je viac ako tri. Počet tri-rezistibilných hrán resp. vrcholov akritických grafov sa môže pomerne výrazne líšiť a väčšie grafy umožňujú väčšiu početnosť takýchto hrán/vrcholov.

Ďalej sme zistili, že všetky vybrané 44 vrcholové grafy s rezistenciou tri a nepárnosťou štyri obsahujú aj štvor-rezistibilné hrany, no nie všetky obsahujú aj štvor-rezistibilné vrcholy. Tiež sme zistili, že rezistencia jedného vybraného snarku rádu 76 s nepárnosťou štyri je tri a identifikovali sme rezistibilitu jeho hrán a vrcholov.

Dúfame, že dosiahnuté výsledky ako aj systém samotný napomôžu v ďalšom výskume snarkov v budúcnosti.

Zoznam použitej literatúry

[1] AMD Ryzen Threadripper specifications. https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-2990wx. Accessed: 2019-04-17.

[2] C implementation of Karol Nagy's bfs 3-edge couriser by Michal Povinsky. https://github.com/jkarabas. Accessed: 2019-04-17.

[3] Graph6 format description. https://users.cecs.anu.edu.au/~bdm/data/formats.html. Accessed: 2019-04-11.

[4] GraphViz software. https://www.graphviz.org/. Accessed: 2019-04-11.

[5] House of graphs. https://hog.grinvin.org/Snarks. Accessed: 2019-04-11.

[6] HPCC UMB. http://www.hpcc.umb.sk/sk/systemy.html. Accessed: 2019-04-11.

[7] Java and C++ comparison. https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/java-gpp.html. Accessed: 2019-04-17.

[8] JSON C++ library. https://github.com/nlohmann/json. Accessed: 2019-04-11.

[9] JSON website. https://www.json.org/. Accessed: 2019-04-17.

[10] nauty and Traces. http://pallini.di.uniroma1.it/. Accessed: 2019-04-25.

[11] Open Graph Drawing Framework. http://www.ogdf.net/doku.php. Accessed: 2019-04-17.

[12] SAT solvers competition. https://www.satcompetition.org/. Accessed: 2019-04-17.

[13] Appel, K., and Haken, W. Every planar map is four colorable. part i: Discharging. *Illinois J. Math. 21*, 3 (09 1977), 429–490.

[14] Audemard, G., and Simon, L. Glucose 2.1: Aggressive - but reactive - clause database management, dynamic restarts. In *Proc. of International Workshop of Pragmatics of SAT (Affiliated to SAT)* (Trento, Italie, Jun 2012).

[15] Blanuša, D. Problem četiriju boja [the four-color problem]. *Glasnik Mat. Fiz. Astr. Ser. II* (1946), 31–42.

[16] Blelloch, G. E., and Maggs, B. M. Algorithms and theory of computation handbook. Chapman & Hall/CRC, 2010, ch. Parallel Algorithms, pp. 25–25.

[17] Brinkmann, G., Goedgebeur, J., Hägglund, J., and Markström, K. Generation and properties of snarks. *Journal of Combinatorial Theory, Series B 103*, 4 (2013), 468 – 488.

[18] Burke E.; De Werra, D. K. J. *5.6.5 Sports Timetabling.* CRC Press, 2004. ISBN: 978-1-58488-090-5.

[19] Cameron, P. J., Chetwynd, A. G., and Watkins, J. J. Decomposition of snarks. *Journal of Graph Theory 11*, 1 (1987), 13–19.

[20] Casanova, H., Legrand, A., and Robert, Y. *Parallel Algorithms.* 01 2008.

[21] Chetwynd, A. G., and Wilson, R. J. Snarks and supersnarks. *The Theory and Application of Graphs* (1987), 215–241.

[22] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[23] Coulouris, G., Dollimore, J., and Kindberg, T. *Distributed Systems. Concepts and Design. 3rd edition.* 01 2000.

[24] Descartes, B. Network-colourings. *The Mathematical Gazette 32*, 299 (1948), 67–69.

[25] Diestel, R. *Graphentheory; 2nd ed.* Springer, Heidelberg, 2000. Record from the Electronic Library of Mathematics/European Mathematical Society.

[26] Dvořák, Z., Kára, J., Král', D., and Pangrác, O. An algorithm for cyclic edge connectivity of cubic graphs. In *Algorithm Theory - SWAT 2004* (Berlin, Heidelberg, 2004), T. Hagerup and J. Katajainen, Eds., Springer Berlin Heidelberg, pp. 236–247.

[27] Ecma International. The json data interchange format. Standard ECMA-404, October 2013.

[28] Eén, N., and Sörensson, N. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing* (Berlin, Heidelberg, 2004), E. Giunchiglia and A. Tacchella, Eds., Springer Berlin Heidelberg, pp. 502–518.

[29] Ellson, J., Gansner, E., Koutsofios, L., North, S., Woodhull, G., Description, S., and Technologies, L. Graphviz — open source graph drawing tools. In *Lecture Notes in Computer Science* (2001), Springer-Verlag, pp. 483–484.

[30] Erlebach, T., and Jansen, K. The complexity of path coloring and call scheduling. *Theor. Comput. Sci. 255*, 1-2 (Mar. 2001), 33–50.

[31] Esperet, L., Mazzuoccolo, G., and Tarsi, M. The structure of graphs with Circular flow number 5 or more, and the complexity of their recognition problem. *arXiv e-prints* (Jan 2015), arXiv:1501.03774.

[32] Fiol, M., and Vilaltella Castanyer, J. A simple and fast heuristic algorithm for edge-coloring of graphs. *AKCE Int. J. Graphs Combin. 10* (10 2013), 263–272.

[33] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[34] Gandham, S., Dawande, M., and Prakash, R. Link scheduling in sensor networks: distributed edge coloring revisited. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.* (March 2005), vol. 4, pp. 2492–2501 vol. 4.

[35] Gardner, M. Mathematical games. *Scientific American 234*, 126–130.

[36] Gherardi, L., Brugali, D., and Comotti, D. A java vs. c++ performance evaluation: A 3d modeling benchmark. vol. 7628.

[37] HLINĚNÝ, P. *Základy teorie grafů [online]*, 1. vyd. ed. Elportál. Masarykova univerzita, 2010 [cit. 2019-03-15].

[38] Holyer, I. The np-completeness of edge-coloring. *SIAM Journal on Computing 10*, 4 (1981), 718–720.

[39] Isaacs, R. Infinite families of nontrivial trivalent graphs which are not tait colorable. *The American Mathematical Monthly 82*, 3 (1975), 221–239.

[40] Programming languages – c++. Standard, International Organization for Standardization, Geneva, CH, 12 2017.

[41] J. Watkins, J., and J. Wilson, R. A survey of snarks.

[42] Kleinberg, J., and Tardos, E. *Algorithm Design*. Addison Wesley, 2006.

[43] Kosowski, A., and Manuszewski, K. Classical coloring of graphs. In *Graph colorings*, vol. 352 of *Contemp. Math.* Amer. Math. Soc., Providence, RI, 2004, pp. 1–19.

[44] Kumar, R., and Charu, S. Comparison between cloud computing, grid computing, cluster computing and virtualization. vol. 3.

[45] Kumar, V. *Introduction to Parallel Computing*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[46] Liang, J., Lou, D.-j., and Nie, R.-h. Algorithms for determining the cyclic edge connectivity of cubic graphs. pp. 149–154.

[47] Martin, R. C. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[48] Message Passing Interface Forum. Mpi: A message-passing interface standard, version 2.2. Specification, September 2009.

[49] Máčajová, E., and Raspaud, A. On the strong circular 5-flow conjecture. *Journal of Graph Theory 52*, 4 (2006), 307–316.

[50] Nagy, K. Kombinatorické vlastnosti snarkov. Master's thesis, Univerzita Komenského v Bratislave, 1995.

[51] Nedela, R., and Ŝkoviera, M. Decompositions and reductions of snarks. *Journal of Graph Theory 22*, 3 (1996), 253–279.

[52] Palúch, S. *Algoritmická teória grafov*. Žilinská univerzita, 2008.

[53] Petersen, J. Die theorie der regulären graphen. *Acta Math. 15* (1891), 193–220.

[54] Snir, M., Otto, S. W., Huss-Lederman, S., Walker, D. W., and Dongarra, J. *MPI: The complete reference*. MIT Press, Cambridge, MA, 1996.

[55] Steffen, E. Classifications and characterizations of snarks. *Discrete Mathematics 188*, 1 (1998), 183 – 203.

[56] Sterling, T., Lusk, E., and Gropp, W., Eds. *Beowulf Cluster Computing with Linux*, 2 ed. MIT Press, Cambridge, MA, USA, 2003.

[57] Szekeres, G. Polyhedral decompositions of cubic graphs. *Bull. Austra. Math. Soc. 8* (1973), 367–387.

[58] Tait, P. G. Remarks on the colouring of maps. *Proceedings of the Royal Society of Edinburgh 10*, 4 (1880), 501–503.

[59] Vizing, V. G. On an estimate of the chromatic class of a p-graph. *Melody Diskret. Analiz 3* (1964), 25–30.

[60] Williamson, D. P., Hall, L. A., Hoogeveen, J. A., Hurkens, C. A. J., Lenstra, J. K., Sevast janov, S. V., and Shmoys, D. B. Short shop schedules. *Oper. Res. 45*, 2 (1997), 288–294.

[61] Łukasz Kowalik. Improved edge-coloring with three colors. *Theoretical Computer Science 410*, 38 (2009), 3733 – 3742.

[62] Škoviera, M. personal communication.

[63] Škoviera, M. Poznámky k tvorbe systému pre testovanie snarkov. unpublished, November 2015.

Prílohy

V prílohách sa nachádzajú nasledujúce položky:

Príloha A: Používateľská príručka.

Príloha B: Systémová dokumentácia.

Príloha C: Algoritmus pre obvod grafu.

Príloha D: Algoritmus pre subkritické vlastnosti grafu.

Príloha E: Konštrukcie snarkov.

UNIVERZITA MATEJA BELA V BANSKEJ BYSTRICI
FAKULTA PRÍRODNÝCH VIED

VÝPOČTOVÝ SYSTÉM PRE ŠTRUKTURÁLNU ANALÝZU GRAFOV S PODPOROU PARALELIZMU

Používateľská príručka

Banská Bystrica, 2019 **Bc. Jakub Strmeň**

Príloha A

Používateľská príručka

Výstup implementácie navrhnutého systému je vo forme spustiteľných binárnych súborov, ktoré sa dajú spustiť pomocou príkazu *mpirun* na platforme *Linux*. Tieto súbory sú: *chromatic-properties-mpi*, *edgeResistanceX-one*, *edgeResistanceX-set*, *vertexResistanceX-one*, *vertexResistanceX-set*. Spúšťanie všetkých týchto súborov je takmer totožné. Súbor *chromatic-properties-mpi* môžeme spustiť príkazom: *„mpirun -np $pocet_uzlov –hostfile $subor_s_nazvami_uzlov ./chromatic-properties-mpi $vstupny_subor $vystupny_subor“*. Parameter súboru s názvami uzlov je nepovinný, no ak ho nezadáme, program sa spustí len na lokálnom stroji s využitím všetkých jeho jadier. Vstupný súbor v tomto prípade musí byť vo formáte *.g6* bez hlavičky. Pre spustenie súboru je však nutné mať nainštalovaný softvér podporujúci príkaz *mpirun*. Takýmto príkladom je aj balík *OpenMPI* dostupný na $https://www.open-mpi.org/$. Každý takýto skompilovaný binárny súbor je však citlivý na *MPI* knižnice, s ktorými bol kompilovaný, a preto spustenie v inom prostredí ako bolo kompilačné, bude pravdepodobne neúspešné. Preto je ideálne kompilovať zdrojový kód priamo na stroji, kde bude program používaný.

Kompiláciu môžeme spustiť pomocou príkazov cmake a následne make. Väčšinou je ideálne vytvoriť si v priečinku so zdrojovým kódom priečinok s názvom napr. *build* a v tomto priečinku použiť príkaz *„cmake ../“*. Potom v tomto priečinku spustíme príkaz *make all*, ktorý skompiluje zdrojový kód a vytvorí spustiteľné súbory spomenuté vyššie. Ak chceme vytvoriť len jeden z týchto súborov, napríklad *chromatic-properties-mpi*, stačí použiť príkaz *make chromatic-properties-mpi*.

Pre spúšťanie programu na klastri UMB sme používali systém Torque, ktorý manažuje obsadenosť a vyváženosť klastra. Pre spustenie úlohy v tomto systéme je potrebné použiť skript, kde pomocou #PBS inštrukcií korigujeme požadované zdroje pre beh programu. Ďalej v tomto skripte uvedieme vyššie spomenutý príkaz pre spustenie behu MPI programu. Takýmto skriptom potom pomocou príkazu *qsub* vložíme

danú úlohu do systému Torque, ktorý ju sám automaticky spustí, keď sa požadované zdroje uvoľnia a tieto zdroje bude pre potreby behu aplikácie manažovať.

Príklad skriptu pre Torque:

```
#!/bin/bash
#PBS -N nazov_aplikacie
#PBS -A identifikator_projektu
#PBS -r n
#PBS -q batch
#PBS -l nodes=7:ppn=32:ht #počet uzlov a ich typ
#PBS -v tpt=1
#PBS -l mem=40000m #požadovaná pamäť
#PBS -l walltime=96000:00:00 #predpokladaný maximálny čas behu programu
mpirun ...
```

Zdrojové súbory sú pomocou nástroja *cmake* definované ako knižnice *snarktest, sat* a *cluster-lib*. Pre pridanie knižníc do projektu by preto malo byť postačujúce pridať do súboru *CMakeLists.txt*, kde je definovaný cieľový projekt, tento riadok:
add_subdirectory(root_directory_of_my_project/SnarktestExtended)
Samozrejme, že zadaná cesta musí obsahovať celý projekt *SnarktestExtended* a všetky jeho zdrojové súbory. Potom už môžeme priradiť knižnicu pre ľubovoľný cieľ pridaním riadku:
„target_link_libraries(my-project snarktest sat cluster-lib)“

UNIVERZITA MATEJA BELA V BANSKEJ BYSTRICI
FAKULTA PRÍRODNÝCH VIED

VÝPOČTOVÝ SYSTÉM PRE ŠTRUKTURÁLNU ANALÝZU GRAFOV S PODPOROU PARALELIZMU

Systémová dokumentácia

Banská Bystrica, 2019 **Bc. Jakub Strmeň**

Príloha B

Systémová dokumentácia

Vzhľadom na rozsah systémovej dokumentácie ju v tlačenej verzii neuvádzame. Je však k dispozícii na priloženom CD nosiči.

Príloha C

Algoritmus pre obvod grafu

Algorithm 12: GirthOfGraph

Input: Cubic graph *simpleGraph*

Output: Girth of given graph

```
1  shortestCycleLength ← ∞
2  foreach vertex of simpleGraph do
3      next ← BFSLengthOfShortestCycle(simpleGraph, vertex)
4      if next < shortestCycleLength then
5          shortestCycleLength ← next
6  return shortestCycleLength
```

Algorithm 13: BFSLengthOfShortestCycle

Input: Cubic graph *simpleGraph*, vertex *vertex*

Output: length shortest cycle from given *vertex*

```
1  visited[simple.graphSize] ← (bool visited, int phase)
2  queue.push(vertex)
3  phase ← 0
4  while queue is not empty do
5      vertex ← top of queue
6      phase ← visited[vertex].phase
7      if vertex was visited then
8          return phase + visited[vertex].phase
9      else
10         foreach neighbor of vertex do
11             visited[neighbor].phase ← phase
12             queue.push(neighbor)
13 return Graph does not contain cycle
```

Príloha D

Test subkritických vlastností

V tomto algoritme budeme testovať len vrcholovú a hranovú subkritickosť grafu G. Pre optimalizáciu výpočtu však budeme najprv testovať aj kritickosť grafu. Ako sme totižto ukázali v 1.4 kritickosť implikuje hranovú aj vrcholovú subkritickosť. Ak bude graf kritický, nie je nutné pre každý vrchol v kontrolovať všetky ostatné vrcholy pre nájdenie neodstrániteľnej dvojice, stačí sa pozerať na susedov tohto vrcholu. Čo je však z hľadiska výpočtu ešte dôležitejšie, nie je potom nutné samostatne kontrolovať hranovú subkritickosť. Ako je teda opísané v algoritme 14, najprv testujeme kritickosť a vrcholovú subkritickosť a ak graf nie je kritický, pokračujeme v teste hranovej subkritickosti.

Algorithm 14: TestSubcriticalProperties

Input: Cubic graph *simpleGraph*

Output: Property of criticality and both of subcriticality of given graph

```
1 edgeSubcritical ← false
2 critical ← true
3 vertexSubcritical ← true
4 TestCriticalityAndVertexSubcriticality(simpleGraph)
5 if critical is true then
6 |   edgeSubcritical ← true
7 else
8 |   edgeSubcritical ← TestEdgeSubcriticality(simpleGraph)
9 return
```

Test kritickosti a vrcholovej subkritickosti. Algoritmus 15 testuje kritickosť grafu. V prípade, že pri určitom vrchole sa zistí, že daný graf nie je kritický (tzn. odstránenie tohto vrcholu s niektorým jeho susedom nespôsobí ofarbiteľnosť), od to-

hoto vrcholu sa pokračuje v testovaní vrcholovej subkritickosti. Tzn. že sa musia pre každý vrchol odstraňovať a následne graf testovať nielen jeho susedia ale všetky ostatné vrcholy.

Algorithm 15: TestCriticalityAndVertexSubcriticality

Input: Cubic graph *simpleGraph*

Output: Criticality and vertex subcriticality property

```
1  critical ← true
2  vertexSubcritical ← true
3  foreach vertex of simpleGraph do
4      if critical = true then
5          foreach neighbor of vertex do
6              sg′ ← removeVertex(simpleGraph, vertex)
7              sg″ ← removeVertex(sg', neighbor)
8              if 3EdgeColour(sg″) = false then
9                  critical ← false
10     if critical = false then
11         sg′ ← removeVertex(simpleGraph, vertex)
12         foreach vertex′ of sg′ do
13             sg″ ← removeVertex(sg', vertex')
14             if 3EdgeColour(sg″) = true then
15                 vertexSubcritical ← true
16                 break
17             vertexSubcritical ← false
18         if vertexSubcritical = false then
19             return     /* there exist a vertex of sg for which doesn't
                exist vertex′ of sg′ such that sg″ is colorable */
20
21 return
```

Test hranovej subkritickosti Ak sa ukázalo, že daný graf je kritický tento test nie je nutné spustiť. Ak však graf kritický nie je, musíme testovať aj jeho hranovú subkritickosť a teda pre každú dvojicu hrán tieto odstrániť a overiť, či graf po ich odstránení je ofarbiteľný alebo nie. Tento algoritmus vidíme v 5.

Algorithm 16: TestEdgeSubcriticality

Input: Cubic graph *simpleGraph*

Output: Edge subcriticality property of given graph

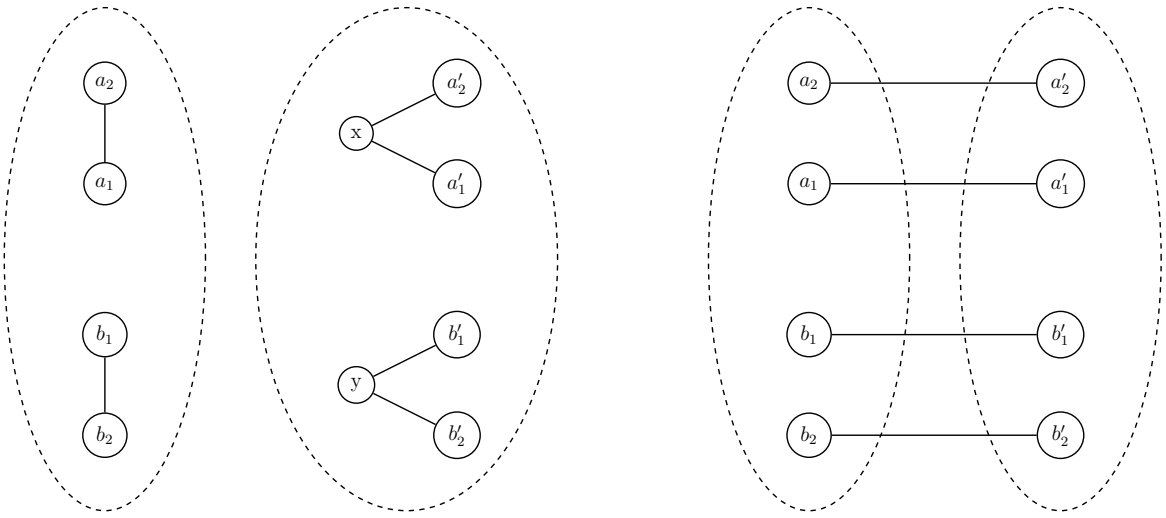
```
1  edgeSubcritical ← true
2  foreach edge of simpleGraph do
3      sg′ ← removeEdge(simpleGraph, edge)
4      foreach edge′ of sg′ do
5          sg″ ← removeEdge(sg', edge')
6          if 3EdgeColour(sg″) = true then
7              edgeSubcritical ← true
8              break
9          edgeSubcritical ← false
10     if edgeSubcritical = false then
11         return false
12 return true
```

Príloha E

Konštrukcie snarkov

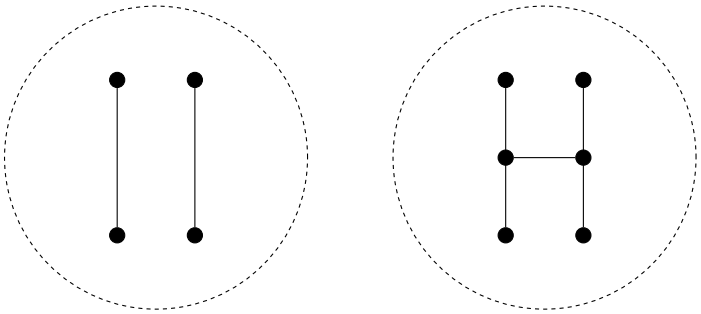
Zo snarkov vieme rôznymi spôsobmi skonštruovať väčšie snarky. Niektoré z týchto spôsobov si stručne ukážeme v nasledujúcej podkapitole. Pre väčšinu týchto konštrukcií sú zásadné odstrániteľné hrany a vrcholy, ich dvojice, resp. kombinácie. Práve z tohto dôvodu môže testovanie chromatických vlastností úzko súvisieť aj s konštrukciami nových snarkov. Napríklad vieme, že každá odstrániteľná dvojica hrán umožňuje I-extenziu snarku.

4-súčin (alebo dot-produkt) dvoch snarkov G a H vznikne nasledovne. Vezmeme dve nesusedné hrany $e = (a_1, a_2)$ a $f = (b_1, b_2)$ a odstránime ich z grafu G. Potom zoberieme dvojicu vrcholov $\{x, y\}$ grafu H takú, že x susedí s vrcholmi $\{a'_1, a'_2\}$ a y susedí s vrcholmi $\{b'_1, b'_2\}$. Vrcholy $\{x, y\}$ spolu s ich hranami odstránime z grafu H. Ďalej spojíme vrcholy $(a_1, a'_1)(a_2, a'_2)(b_1, b'_1)(b_2, b'_2)$. Graf, ktorý vznikol súčinom grafu G a H bude snark. [51]



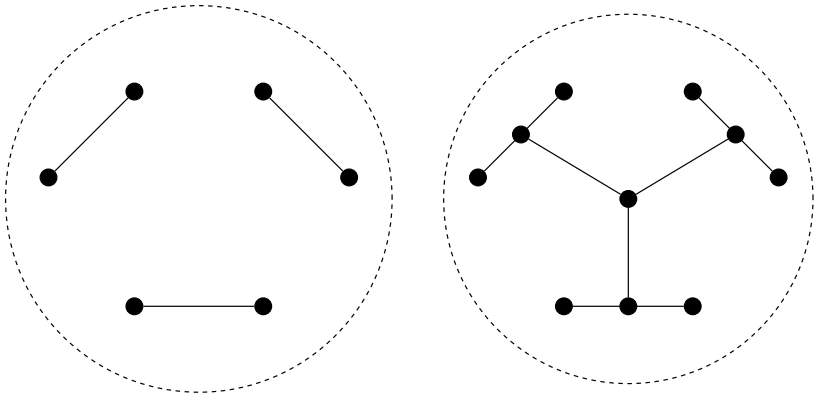
Obr. 30: Vľavo dvojica hrán v grafe G a dvojica vrcholov x,y v grafe H. Vpravo vidíme aplikovaný 4-súčin po odstránení dvoch hrán z grafu G a dvoch vrcholov z grafu H.

I-extenzia zväčší snark o dva vrcholy. Pre jej realizáciu potrebujeme dvojicu odstrániteľných hrán. Do stredu každej z týchto hrán vložíme nový vrchol a tieto dva novovzniknuté vrcholy spojíme hranou. Ilustrácia na obrázku 31.



Obr. 31: Vľavo dvojica odstrániteľných hrán v grafe G. Vpravo aplikovaná I-extenzia na pôvodnú dvojicu hrán.

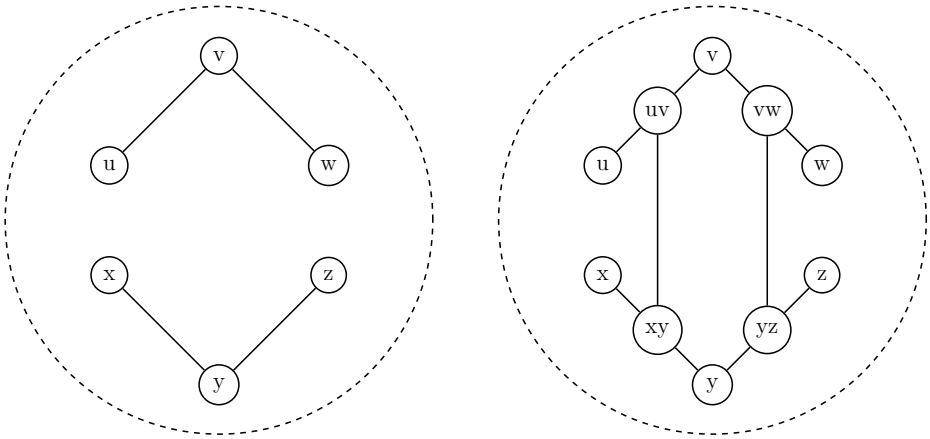
Y-extenzia zväčší snark o 4 vrcholy. Potrebujeme trojicu odstrániteľných hrán. Do stredu každej z nich vložíme vrchol. Do grafu pridáme ďalší, štvrtý vrchol a spojíme ho s tromi ostatnými novými vrcholmi, ktoré vznikli na daných hranách. Príklad Y-extenzie môžeme vidieť na obrázku č. 32.



Obr. 32: Vpravo aplikovaná Y-extenzia na pôvodnú trojicu hrán.

2I-extenzia Pre 2I extenziu potrebujeme dve dvojice susedných hrán (u,v), (v,w) a (x,y), (y,z). Do každej z hrán vložíme nový vrchol a hranami prepojíme novovzniknuté vrcholy (uv, xy) a (vw, yz). Príklad 2I extenzie môžeme vidieť na obrázku 33.

Ďalšie možnosti transformácie snarku sú napríklad aj 2-sum, 3-sum, O-extenzia, 5-súčin. Pri testovaní chromatických vlastností je dobré si všímať aj odstrániteľné nesusedné hrany, pretože pomocou nich cez I-extenziu vieme snark rozšíriť. Je ľahko dokázateľné, že odstrániteľná dvojica je zároveň aj rozšíriteľná.



Obr. 33: Vľavo dve dvojice susedných hrán v grafe G. Vpravo aplikovaná 2I-extenzia na vybrané hrany.