Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Farebné množiny kubických multipólov

Bakalárska práca

2019
Miroslav Mrózek

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Farebné množiny kubických multipólov

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	RNDr. Ján Mazák, PhD.

Bratislava, 2019
Miroslav Mrózek

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

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

Názov: Farebné množiny kubických multipólov
Colouring sets of cubic multipoles

Anotácia: Náplňou práce je preskúmať, aké farby môžu mať trčiace hrany kubických multipólov. Práca bude nadväzovať na existujúci výskum v tejto oblasti.

Cieľ: Ciele práce:
1. na základe existujúceho výskumu pre rezy s veľkosťou 1 až 6 navrhnúť všeobecný kanonický tvar pre farebnú množinu multipólu
2. navrhnúť a implementovať dátový typ pre multipól a jeho farebnú množinu v jazyku C++ a doplniť ho do existujúcej grafovej knižnice ba_graph
3. vytvoriť úplnú databázu k-pólov s danou farebnou množinou, danou cyklickou súvislosťou a minimálnym počtom vrcholov pre k do 5 a čiastočnú databázu pre k = 6 a k = 7

Literatúra:

Vedúci: RNDr. Ján Mazák, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Dátum zadania: 31.10.2018

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

.....
študent

.....
vedúci práce

Poďakovanie: Ďakujem svojmu školitelovi za vedenie a rady, a svojej rodine za podporu.

Abstrakt

Po aplikovaní operácie hranového rezu na snark, dostaneme 2 grafy, ktorým keď na miesta kde stratili hranu pridáme trčiacu hranu, ktorá na jednom konci má vrchol a na druhom nie, získame 2 kubické multipóly. Pre multipól nás nezaujíma farbenie celého grafu, ale iba aké farbenia sú pripustiteľné pre trčiace hrany, značené spolu ako farebná množina, ktorá vychádza z usporiadania trčiacich hrán multipólu, čo ale spôsobuje, že jednému multipólu prislúcha viacero farebných množín podľa poradia trčiacich hrán, kvôli čomu vznikla potreba pre kanonickú formu farebnej množiny, ktorá už nezávisí na poradí trčiacich hrán. V tejto práci riešime aj počítanie farebnej množiny multipólu pre dané poradie hrán, ako aj v kanonickom tvare. Taktiež vznikla potreba pre databázu multipólov s malým počtom vrcholov, na ktorých bude prebiehať výskum, preto v rámci tejto práce boli vygenerované kubické multipóly veľkosti do 20 vrcholov pomocou existujúceho generátora Pregraph.

Kľúčové slová: multipól, snark, farbenie

Abstract

After aplying edge-cut operation on a snark, we get 2 graphs, to which if we add a semiedge, which is an edge with one end with a vertex and second without, to the place they lost an edge, we get 2 cubic multipoles. For multipoles, we do not care for the colouring of the whole graph, obly the satisfiable colourings of the semiedges, named colouring set, which is determined by the order of semiedges, which results in one multipole having several colouring sets depending on the order of semiedges, because of which has arisen a need for a canonical form for a colouring set, which no longer depends on the order of semiedges. In this thesis we focus on computing a colouring set of a multipole for given order of semiedges as well as a colouring set in canonical form. We also needed a database of multipoles with a small amount of vertices, so in the scope of this thesis we also generated cubic multipoles with up to 20 vertices with existing generator Pregraph.

Keywords: multipole, snark, colouring

Obsah

Zoznam obrázkov

Zoznam tabuliek

Úvod

V tejto práci sa budeme zaoberať kubickými multipólmi a farebnou množinou multipólou. Hlavným cieľom práce je definovať a implementovať kanonický tvar farebnej množiny do knižnice $ba - graph$.

Telo tejto práce je rodelené následovne.

V prvej položíme teoretické základy tejto práce. Definujeme v nej dôležité pojmy a prejdeme niektoré známe poznatky.

V druhej kapitole sú rozoberáme implementáciu rozhrania na počítanie farebnej množiny a rôzne rozhodnutia, ktoré sme počas tejto práce urobili.

V tretej kapitole sa pozeráme na príklady použitia daných funkcií implementovaného rozhrania, ktoré boli vybraté z testov, spolu s ukážkovými programami, ktoré boli takisto vyvinuté ako súčasť tejto práce, ale neboli súčasťou algoritmu počítania farebnej množiny ukázaného v druhej kapitole

Vo štvrtej kapitole hovoríme o programe $Pregraph$, ktorý slúži na generovanie súvislých kubických grafov s určitými vlastnosťami, o formáte takto vygenerovaných grafoch, a o multipóloch, ktoré sme s jeho pomocou vygenerovali.

Kapitola 1

Multipóly

1.1 Multipóly

V tejto kapitole si objasníme mnohé pojmy, týkajúce sa multipólov a ich farbenia. Budeme využívať štandardné definície z teórie grafov, rozšírené na multipóly, ako napríklad:

- *V(M)* je označenie množiny vrcholov multipólu M
- *E(M)* je označenie množiny hrán multipólu M

Definícia 1 *Multipól $M = (V, E)$ sa skladá z množiny vrcholov V a množiny hrán E. Každá hrana $e \in E$ má dva konce a každý môže, ale nemusí, byť incidentný s vrcholom[2].*

Hrany multipólu môžu byť jedným zo štyroch typov.

- *Súvislá hrana* má dva konce e a f, e je incidentný s vrcholom E a f je incidentný s vrcholom F, pričom $e \neq f$, $E \neq F$.
- *Slučka* je hrana, ktorej oba konce sú incidentné s rovnakým vrcholom s rovnakým vrcholom.
- *Trčiaca hrana* je hrana, ktorá má len jeden koniec incidentný s vrcholom..
- *Izolovaná hrana* je hrana, ktorej žiaden koniec nie je incidentný s vrcholom.

Voľný koniec je koniec hrany, ktorý nie je incidentný so žiadnym vrcholom. Množinu voľných koncov multipólu M budeme označovať $S(M)$. *Polohrana* je jeden koniec hrany. Tieto pojmy sú vysvetlené na obrázku 1.1.

K-pól je multipól s práve k trčiacimi hranami. 0-pól je kubický graf. Ak majú trčiace hrany k-pólu M priradené usporiadanie $s_1, s_2, \ldots, s_k$, potom $M(s_1, s_2, \ldots, s_k)$ sa nazýva *usporiadaný k-pól*.

Je praktické rozdeliť si množinu voľných koncov $S(M)$ do disjunktých podmnožín $S_1, S_2, \ldots, S_n$, ktoré nazývame *konektory*. Každý konektor má dané lineárne usporiadanie voľných koncov. Voľný koniec, ktorý nie je obsiahnutý v žiadnom konektore, sa nazýva *zvyškový voľný koniec*. Množinu všetkých zvyškových voľných koncov multipólu M označujeme $Res(M)$. Ak konektor S obsahuje iba jeden voľný koniec s, namiesto toho môžeme písať namiesto S iba samotný voľný koniec s. Rozdelenie voľných koncov do konektorov je užitočné vtom, že niektoré vlastnosti multipólov by sa bez nich ťažko opisovali. Príkladom je istý typ 5-pólu s názvom *negátor*, ktorý má 2 konektory s dvomi voľnými koncami a 1 zvyškový voľný koniec. Je zaujímavý vtom, že ak dva konce z jedného konektora majú rovnakú farbu, potom voľné konce v druhom konektore musia mať rozdielne farby.

Počet vrcholov v multipóle M značíme $|M|$. V tejto práci budeme uvažovať o kubických multipóloch, teda každý vrchol bude incidentný s tromi hranami.

1.2 Hranové farbenie multipólov

V tejto práci sa budeme zaoberať farbením multipólov, teda aj vytŕčajúce hrany môžu byť zafarbené. Ako neskôr ukážeme, množina nenulových prvkov grupy $(Z_2 \times Z_2, +)$, má dobré vlastnosti. Túto množinu budeme označovať K.

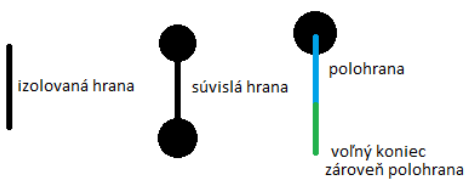
Nech M je multipól, K je množina farieb a $\varphi : E(M) \to K$ je zobrazenie, ktoré priraďuje každej hrane z M farbu z K. Potom φ sa nazýva *3-hranové farbenie*, alebo jednoduchšie, farbenie multipólu M, ak pre každý vrchol $v \in V(M)$ majú hrany s ním incidentné po dvoch rozdielne farby.

Ak existuje farbenie multipólu M, vravíme, že je zafarbiteľný.

Použitím farieb z množiny K, môžeme využiť sčítanie v grupe $Z_2 \times Z_2$, aby sme analyzovali správanie zobrazenia $\varphi : E(M) \to K$. Ak označíme $\delta(v)$ množinu hrán incidentných s vrcholom $v \in V(M)$, potom je zjavne vidno, že φ je farbením vtedy a len vtedy, ak

$$\sum_{e \in \delta(v)} \varphi(e) = 0 \tag{1.1}$$

pre každý vrchol v. Táto rovnica je Kirchhoffov zákon o prúdoch v grafoch. Vďaka tomu, že každý prvok v K je sám so sebou inverzný, nemusíme rozlišovať orientáciu



Obr. 1.1: Príklad multipólu

toku. Farbenie pomocou množiny K sa taktiež nazýva *Taitovo farbenie.*

Ak sa na farbenie φ multipólu pozrieme ako na tok, tak ľahko môžeme pozorovať z vlastností toku, že

$$\sum_{e \in S(M)} \varphi(e) = 0. \tag{1.2}$$

Lema 1 (Paritná lema) *Nech φ je 3-hranové farbenie k-pólu P. Potom počet voľných koncov P ktoré sú vo φ rovnako zafarbené každou danou farbou má rovnakú paritu ako k.*

Túto lemu vieme vyjadriť aj pomocou grupy $(Z_2 \times Z_2, +)$.

Lema 2 *Nech M je k-pól a k_1, k_2 a k_3 sú množiny hrán vyfarbených farbami $(0,1)$, $(1,0)$ a $(1,1)$. Potom*

$$k_1 \equiv k_2 \equiv k_3 \equiv k \pmod 2. \tag{1.3}$$

1.3 Snarky

Snarky sú ešte stále zväčša neprebádanou oblasťou teórie grafov, napriek vyše sto rokom výskumu. Ten začal v roku 1880, kedy Peter Guthrie Tait dokázal ekvivalenciu problému 4 farieb a tvrdenia, že planárne bezmostové grafy sú 3-hranovo zafarbiteľné[3]. Existuje viacero podobných definícií, z ktorých väčšina vyžaduje nielen to, aby bol graf kubický a nebol 3-hranovo zafarbiteľný, ale aj aby bol *netriviálny.* Ale netriviálny snark je pojem, ktorý má viacero definícií. Najčastejšie je požadované, (okrem toho aby bol graf kubický a 3-hranovo nezafarbiteľný) aby neobsahoval most, teda hranu, po ktorej odstránení sa zvýši počet komponentov grafu, alebo/aj aby minimálny cyklus v danom grafe mal dĺžku minimálne 5. Preto sme zvolili nasledujúcu definíciu.

Definícia 2 *Snark je kubický graf bez mosta, ktorý nie je 3-hranovo zafarbiteľný. Netriviálny snark je snark, ktorý navyše neobsahuje cyklus dĺžky 4 alebo menej. Každý snark, ktorý nie je netriviálny, je triviálny.*

Petersen objavil prvý snark, známy ako Petersenov graf (obr. 1.2), ktorý je najmenším snarkom. Dokonca existuje hypotéza, ktorá hovorí, že z každého snarku je možné vytvoriť Petersenov graf konečným množstvom kontrakcií vrcholov.

1.4 Farebné množiny

Každé 3-hranové zafarbenie usporiadaného, kubického k-pólu $P = P(x_1, x_2, \ldots, x_k)$ jednoznačne určuje farebný vektor $a = a_1 a_2 \ldots a_k$, kde $a_i \in 0, 1, 2$ $a_i = j$ znamená, že x_i je zafarbený farbou $j \in \{0, 1, 2\}$ v 3-hranovom zafarbení. Permutovaním farieb

dostávame nanajvýš 6 farebných vektorov, zodpovedajúcich *rovnakému farbeniu*. Z pomedzi nich vyberáme lexikálne minimálny farebný vektor z triedy ekvivalencie a, ktorý budeme nazývať *farebná skupina*[2]. Vravíme, že farebný typ dĺžky k je *stupňa* k. Prirodzene všetky farebné skupiny spĺňajú Paritnú lemu.

Definícia 3 (Farebná množina) *[2] Pre daný usporiadaný k-pól $P = P(x_1, x_2, \ldots, x_k)$ definujeme farebnú množinu $C(P)$ pre P ako množinu farebných skupín stupňa k, určených množinou všetkých 3-hranových farbení P. Vo všeobecnosti definujeme farebnú množinu ako podmnožinu množiny farebných skupín stupňa k. Farebná množina $\mathcal{P}$ je uskutočniteľná, ak existuje multipól P, pre ktorý platí $\mathcal{P} = Col(P)$.*

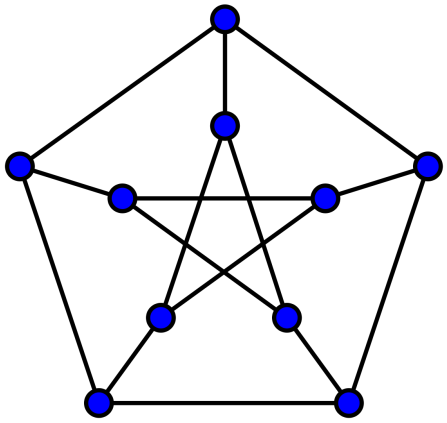
1.4.1 Kanonický tvar farebnej množiny

Keďže farebná množina je daná usporiadaním trčiacich hrán multipólu, tak viacero rozdielnych farebných množín zodpovedá jednému multipólu po preusporiadaní trčiacich hrán. Z toho vznikla potreba pre kanonický tvar.

Definícia 4 *Nech k-pól $P = P(x_1, x_2, \ldots, x_k)$. Pre všetky premutácie trčiacich hrán vypočítame ich farebné množiny. Lexikograficky najmenšia z nich je v kanonickom tvare.*

1.5 Kempeho reťazec

Kempeho reťazec je veľmi užitočným nástrojom pre hranové farbenie. Kempeho reťazce striedajúce dve farby boli taktiež dôležité pri dôkazoch 4-, 5- aj 6-dekompozície snarkov[2]. Mi budeme pracovať s operáciou *Kempeho prepínača*, definovanú pre multipóly.



Obr. 1.2: Petersenov graf

Definícia 5 (Kempeho prepínač) *[2] Nech M je multipól s daným farbením a danou nerozšíriteľnou cestou, striedajúcou 2 farby, ktorá začína aj končí v trčiacej hrane. Kempeho prepínač je operácia, ktorá na tejto ceste vymení jednu farbu za druhú.*

Nech M je k-pól a γ je farbenie P, ktoré určuje farebnú skupinu a. Každý Kempeho prepínač aplikovaný na P má za následok nové farbenie γ' s korešpondujúcou farebnou skupinou b.

Kapitola 2

Implementácia

V tejto kapitole budeme rozoberať implementačné detaily a samotné funkcie ktoré tvoria rozhranie na prácu s multipólmi.

Taktiež by som rád spomenul, že všetky funkcie sú testované aspoň v základných scenároch.

2.0.1 Multipól

Multipóly sa implementačne veľmi nelýšia od grafov. Trčiace hrany iba majú na svojej voľnej polohrane vrchol stupňa jedna, ktorý slúži na označenie polohrany (z implementačných dôvodou), čo je možné, pretože pracujeme iba s kubickými multipólmi (teda v nich niesú žiadne skutočné vrchloly stupňa jedna). Tieto vrcholy nazývame *terminály*. Vďaka tomu, že nie sú špeciálnou triedou, na nich vieme použiť (niekedy po miernej úprave) mnohé z už naprogramovaných funkcií existujúcich v knižnici $ba - graph$. Teda nie je potrebné prepisovať väčšinu funkcií na podporu špeciálnej triedy, čo by bolo mimo rozsah bakalárskej práce. Multipóly sú uložené vo vlastných súboroch, ale na to, aby ich programátor správne použil s existujúcimi funkciami si musí dať pozor sám.

2.0.2 Farbenie

Na nájdenie všetkých možných farbení využívame už existujúcu triedu $PDColorizer$, ktorá spracuje graf s vrcholmi stupňa 2 a 3, a vrátí objekt $ColouringBitArray$.

$ColouringBitArray$ obsahuje 3^n bitov, kde n je počet vrcholov stupňa 2, a predstavuje tak všetky farbenia, z ktorých splniteľné sú reprezentované kladným bitom.

2.1 Hlavné funkcie

Pre jednoduchšie čítanie som zaviedol alias pre farbenie trčiacich hrán aj pre farebnú množinu.

```
using Colouring = std::vector<int>;
using ColourSet = std::vector<Colouring>;
```

2.1.1 getCanonicalColourSet

Toto se hlavná rodina funkcií tohto rozhrania (používame preťaženie mien funkcií), slúžiaca na získanie kanonického tvaru farebnej množiny multipólu. Hlavná idea spočíva vtom, že zoberieme pole predstavujúce poradie trčiacich hrán a *ColouringBitArray*. Oboje permutujeme, následne zo všetkých permutácií *ColouringBitArray* vyberieme lexikograficky najmenšiu a k nej prislúchajúcu premutáciu trčiacich hrán, a obe vraciame.

```
std::pair<std::vector<int>, ColourSet> getCanonicalColourSet(Graph &g) {
  ColouringBitArray cba = getAllColourings(g);
  if (cba.all_false()) {
    throw std::runtime_error("No colouring.");
  }
  auto terminals = internal::getTerminals(g);
  std::vector<int> terminalIds;
  for (auto &vertex : terminals) {
    terminalIds.push_back(vertex.to_int());
  }

  return getCanonicalColourSet(terminalIds, cba);
}
```

Nasledujúca funkcia sa mierne líši tým, že ako parameter vieme nastaviť podmnožinu terminálov, vďaka čomu vo výsledku dostaneme menšiu farebnú množinu iba pre tie trčiace hrany, ktoré nás zaujímajú.

```
std::pair<std::vector<int>, ColourSet>
getCanonicalColourSet(std::vector<int> &terminalIds, Graph &g) {
  ColouringBitArray cba = getAllColourings(g);
  if (cba.all_false()) {
    throw std::runtime_error("No colouring.");
  }
```

```
  return getCanonicalColourSet(terminalIds, cba);
}
```

```
std::pair<std::vector<int>, ColourSet>
getCanonicalColourSet(std::vector<int> &terminalIds, ColouringBitArray cba) {
  cba = internal::minimalColouringBitArray(cba);
  auto permutations = internal::permutations(terminalIds, cba);

  std::vector<ColouringBitArray> bitArrayVector;
  for (auto &c : permutations.second) {
    bitArrayVector.emplace_back(c);
  }
  sort(bitArrayVector.begin(), bitArrayVector.end());

  if (bitArrayVector.size() == 0) {
    throw std::runtime_error("No colouring.");
  }
  auto &res = bitArrayVector[0];
  auto it =
      std::find(permutations.second.begin(), permutations.second.end(), res);
  int id = it - permutations.second.begin();

  ColourSet canonicalColourSet = colouringBitArrayToColourSet(res);
  return std::pair(permutations.first[id], canonicalColourSet);
}
```

2.1.2 getColourSet

Tieto funkcie slúžia na získanie farebnej množiny, pre graf s danou trčacimi hranami v danom usporiadaní. Sú súčasťou verejného rozhrania pre prípady, kedy nepotrebujeme farebnú množinu v kanonickom tvare.

Využívajú internú funkciu *getAllColourings* a niektoré ďalšie potrebné na prípravu.

```
ColourSet getColourSet(Graph &g) {
  internal::throwForSmallWidthMultipoles(g);

  auto edges = internal::setup(g);
  auto terminals = internal::getTerminals(g);
  auto multiedges = internal::addMultiedges(g, terminals);
  edges.insert(edges.end(), multiedges.begin(), multiedges.end());
```

```
  auto res = internal::getColourSet(g, edges);

  internal::removeMultiedges(g);
  return res;
}

ColourSet getColourSet(Graph &g, std::vector<Vertex> selectedTerminals) {
  internal::throwForSmallWidthMultipoles(g);

  auto edges = internal::setup(g);
  auto multiedges = internal::addMultiedges(g, selectedTerminals);
  edges.insert(edges.end(), multiedges.begin(), multiedges.end());

  auto res = internal::getColourSet(g, edges);

  internal::removeMultiedges(g);
  return res;
}
```

2.1.3 getAllColourings

Táto funkcia vracia pre daný multipól farebnú množinu, ale v tvare *ColouringBitArray*.

```
ColouringBitArray getAllColourings(Graph &g) {
  auto colourSet = getColourSet(g);
  auto cba = colourSetToBitArray(colourSet);
  auto res = internal::minimalColouringBitArray(cba);
  return res;
}
```

2.1.4 Prevádzacie funkcie

Tu sú zaradné dve funkcie, ktoré konvertujú farebné množiny medzi tvarmi *ColouringBitArray* a *ColourSet*.

```
ColourSet colouringBitArrayToColourSet(ColouringBitArray &cba) {
  ColourSet res;
  int colour = 0;
  int size = cba.size().to_int64();
  int length = 0;
```

```
  while (size > 1) {
    length++;
    size = size / 3;
  }
  for (auto i = ColouringBitArray::Index(0, 0); i < cba.size(); i++) {
    if (cba.get(i)) {
      Colouring colouring = internal::getColouring(colour, length);
      res.push_back(internal::renumberColouring(colouring));
    }
    colour++;
  }

  return internal::filterColourSet(res);
}

ColouringBitArray colourSetToBitArray(ColourSet &colourSet) {
  if (colourSet.size() < 1)
    throw std::runtime_error("Empty ColourSet");
  auto cba = ColouringBitArray(power(3, colourSet[0].size()), false);
  for (auto &colouring : colourSet) {
    uint_fast64_t num = internal::colouringToDecimal(colouring);
    auto i = ColouringBitArray::Index::to_index(num);
    cba.set(i, true);
  }
  return std::move(cba);
}
```

2.2 Interné funkcie

V tejto sekcii sa nachádza kód k interným funkciám rozhrania.

renumberColouring

Keďže nezáleží na farbe jednotlivých hrán, ale na tom, že farba množín hrán je rovnaká/rozdielna, tak máme funkciu, ktorá prečísluje farbenie, ktoré získame z funkcie *getColour* na lexikograficky najmenšie s rovnakým významom, teda napr. z farbenia 201 by sme dostali farbenie 012.

```
Colouring renumberColouring(Colouring colouring) {
  int n[3];
  std::fill_n(n, 3, -1);
```

```
  for (int &i : colouring) {
    for (int j = 0; j < 3; j++) {
      if (n[j] == -1)
        n[j] = i;
      if (n[j] == i) {
        i = j;
        break;
      }
    }
  }
  return colouring;
}
```

2.2.1 getColouring

Funkcia *getColouring* berie ako argumenty číslo v desiatkovej sústave a počet číslic, ktorý má mať výsledné farbenie, a vráti vektor číslic v trojkovej sústave, čo predstavuje farbenie trčiacich hrán.

```
Colouring getColouring(int colour, int length) {
  Colouring res;
  res.resize(length);
  for (int i = length - 1; i >= 0; i--) {
    int x = colour / (power(3, i));
    res[i] = x;
    colour = colour % (power(3, i));
  }
  return res;
}
```

2.2.2 colouringToDecimal

Farbenie je uložené vektor čísel v trojkovej sústave, táto funkcia ho prevádza z trojkovej sústavy do desiatkovej.

```
int colouringToDecimal(Colouring &colouring) {
  uint_fast64_t num = 0;
  for (int i = colouring.size() - 1; i >= 0; i--) {
    num += colouring[i] * power(3, i);
  }
  return num;
```

```
}
```

2.2.3 filterColourSet

V *colouringBitArray* sa nachádzajú duplikáty, teda farbenia s rovnakým významom, no iným zápisom (napr. 012 a 201). Funkcia $filterColourSet$ dostane ako argument vector všetkých splniteľných farbení už v lexikograficky minimálnom zápise. To znamená, že sa v ňom nachádza viacero identických farbení. Vďaka objektu *std* :: *set* sa ich vieme zbaviť prakticky zadarmo ešte aj s bonusom, že budú zoradené od lexikograficky najmenšieho farbenia po najväčšie.

```
ColourSet filterColourSet(ColourSet &colourings) {
  std::set<Colouring> s(colourings.begin(), colourings.end());
  ColourSet res(s.begin(), s.end());
  return res;
}
```

2.2.4 getTerminals

$getTerminals$ je jednoduchá funkcia na získanie vrcholov stupňa 1, ktoré predstavujú trčiace hrany.

```
std::vector<Vertex> getTerminals(Graph &g) {
  std::vector<Vertex> terminals;
  for (auto &rotation : g) {
    if (rotation.degree() == 1) {
      terminals.push_back(rotation.v());
    }
  }
  return terminals;
}
```

2.2.5

Táto funkcia slúži na získanie daného $ColouringBitArray$ v lexikograficky minimálnom tvare. Využíva nato funkciu $colouringBitArrayToColourSet$, ktorá vo svojom tele volá $filterColourSet$.

```
ColouringBitArray minimalColouringBitArray(ColouringBitArray &cba) {
  auto temp = colouringBitArrayToColourSet(cba);
```

```
  auto t = colourSetToBitArray(temp);
  return t;
}
```

2.2.6 getColourSet

getColourSet je hlavná interná funkcia, ktorá využíva *PDColorizer* na získanie všetkých uspokojiteľnych farbení v objektoch *ColouringBitArray*. Následne iteruje cez všetky *ColouringBitArray* a tie ktoré majú aspoň jedno uspokojiteľné farbenie uloží do vektoru *allColourings*. Tie následne prekonvertuje na *ColourSet*, spojí do jedného, vyfiltruje duplikáty pomocou funkcie *filterColourSet* a vráti výsledok.

```
ColourSet getColourSet(Graph &g, std::vector<Edge> &pathDecomposition) {
  PDColorizer pathDecompositionColorizer;
  pathDecompositionColorizer.initialize(g);
  for (auto edge : pathDecomposition)
    pathDecompositionColorizer.process_state(edge.v1(), edge.v2());
  auto &state = pathDecompositionColorizer.state;
  std::vector<ColouringBitArray> allColourings;
  for (unsigned int i = 0; i < state.size(); i++)
    if (!state[i].all_false())
      allColourings.emplace_back(state[i]);

  ColourSet res;
  for (auto &c : allColourings) {
    auto x = colouringBitArrayToColourSet(c);
    res.insert(res.end(), x.begin(), x.end());
  }

  return internal::filterColourSet(res);
}
```

2.2.7 throwForSmallWidthMultipoles

throwForSmallWidthMultipoles rieši 0 a 1-póli, ktorých spracovanie by inak viedlo k nedokumentovanému správaniu, kvôli internej implementácii.

```
void throwForSmallWidthMultipoles(Graph &g) {
  auto terminals = g.list(RP::degree(1), RT::n());
  if (terminals.size() == 0)
    throw std::runtime_error("Graph has no semiedge");
```

```
  if (terminals.size() == 1) {
    throw std::runtime_error("1-poles have no colouring");
  }
}
```

2.2.8 setup

Setup je funkcia, ktorá mierne upraví daný multipól, využije $shortest_path_heuristic$ na nájdenie zoradenia hrán grafu *pathDecomposition*, vráti multipól do pôvodného stavu a vráti daný *pathDecomposition*.

```
std::vector<Edge> setup(Graph &g) {
  auto createdEdges = internal::addCycleToTerminals(g);

  if (has_cut_edge(g)) {
    throw std::runtime_error(
        "Current implementation does not work with bridges.");
  }

  auto pathDecomposition = shortest_path_heuristic(g, g[0][0].e());
  internal::removeCycleFromTerminals(g, pathDecomposition.ordered_edges,
                                     createdEdges);
  return pathDecomposition.ordered_edges;
}
```

2.2.9 Permutácie

Kvôli počítaniu kanonického tvaru

```
// takes terminalIds and ColouringBitArray
// returns all pair of all terminalIds and cba permutations,
// so that at index i, terminalPermutations[i] would have
// ColouringBitArray cbaPermutations[i]
std::pair<std::vector<std::vector<int>>, std::vector<ColouringBitArray>>
permutations(std::vector<int> terminalIds, ColouringBitArray cba) {
  std::vector<ColouringBitArray> cbaPermutations;
  std::vector<std::vector<int>> terminalPermutations;
  permutationsRecursive(terminalIds, cba, 0, terminalPermutations,
                        cbaPermutations);
  return std::pair(terminalPermutations, cbaPermutations);
}
```

```
// takes terminalIds, ColouringBitArray, id of element to permute,
// and terminalPermutations and cbaPermutations to store return values
// calculates all permutations for terminals and ColouringBitArrays
// so that at index i, terminalPermutations[i] would have
// ColouringBitArray cbaPermutations[i]
void permutationsRecursive(std::vector<int> &terminalIds,
                           ColouringBitArray &cba, int id,
                           std::vector<std::vector<int>> &terminalPermutations,
                           std::vector<ColouringBitArray> &cbaPermutations) {
  if (id == terminalIds.size() - 1) {
    terminalPermutations.emplace_back(terminalIds);
    nextPermutation(terminalIds, id);
    cbaPermutations.emplace_back(cba);
    nextPermutation(cba, id);
  } else {
    for (int i = 0; i < terminalIds.size() - id; i++) {
      permutationsRecursive(terminalIds, cba, id + 1, terminalPermutations,
                            cbaPermutations);
      nextPermutation(terminalIds, id);
      nextPermutation(cba, id);
    }
  }
}
```

2.2.10 Nespomenuté funkcie

Vyššie nie sú rozoberané niektoré funkcie, ktoré iba riešia implementačné detaily. Preto ich aspoň menom spomenieme tu.

Sú nimi: $addMultiedges, removeMultiedges, addCycleToSemiedges, removeCycleFromSemiedges$.

Kapitola 3

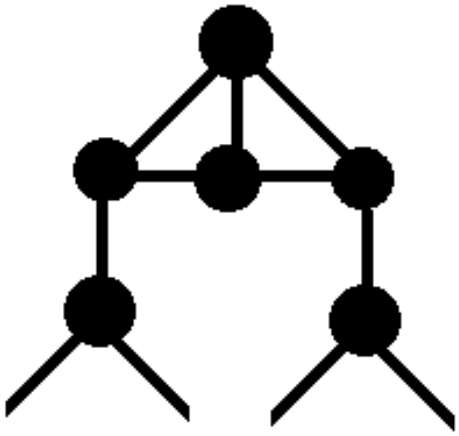
Použitie

V tejto kapitole sa pozrieme na niektoré úryvky kódu vybraného z testov, ako aj menšie programy na ilustráciu .

3.1 Príklad z testov

3.1.1 getCanonicalColourSet

V tomto teste vytvoríme multipól ako na obr. 3.1 a testujeme, či dostaneme správnu farebnú množinu v kanonickom tvare a k nej správne poradie hrán.



Obr. 3.1: 4-pól

```
void test_getCanonicalColourSet() {
  Graph g(createG());
  addMultipleV(g, 10);
  for (int i = 1; i < 4; i++) {
    addE(g, Location(0, i));
  }
```

```
  addE(g, Location(1, 2));
  addE(g, Location(2, 3));
  addE(g, Location(1, 4));
  addE(g, Location(3, 5));
  addE(g, Location(4, 6));
  addE(g, Location(4, 7));
  addE(g, Location(5, 8));
  addE(g, Location(5, 9));

  auto terminalsAndColourSet = getCanonicalColourSet(g);
  auto terminals = internal::getTerminals(g);
  std::vector<int> terminalIds;
  for (auto &vertex : terminals) {
    terminalIds.push_back(vertex.to_int());
  }
  auto temp = terminalIds[0];
  terminalIds.erase(terminalIds.begin());
  terminalIds.push_back(temp);

  assert(terminalIds == terminalsAndColourSet.first);
  ColourSet res = {{0, 0, 1, 1}, {0, 1, 0, 1}};
  assert(terminalsAndColourSet.second == res);
}
```

3.2 Programy

3.2.1 Spájanie multipólov

V rámci skúšania funkcionality boli naprogramované aj funkcie na zistenie, či vieme 2 multipóly spojiť do snarku.

Ak zoberieme 2 multipóly s daným usporiadaním trčiacich hrán, ktoré určujú farebnú množinu, vieme ich spojiť do snarku v danej konfigurácii (i-ta trčiaca hrana prvého multipólu s i-tou trčiacou hranou druhého multipólu) ak ich farebné množiny majú aspoň jedno spoločné farbenie.

To, že či sa dajú 2 multipóly spojiť do snarku v nejakom usporiadaní trčiacich hrán vieme zistiť tak, že pre prvý multipól vypočítame všetky permutácie trčiacich hrán a k nim farebné množiny. Pre tieto farebné množiny skontrolujeme, či majú aspoň jedno farbenie spoločné s farebnou množinou druhého multipólu a ak nemajú, je možné ich spojiť do snarku.

```
std::pair<std::vector<int>, bool>
canCreateSnark(std::vector<int> &terminalIdsFirst, ColouringBitArray &first,
               ColouringBitArray &second) {
  if (first.size() != second.size()) {
    throw std::runtime_error("ColouringBitArrays are not same size");
  }
  if (first.all_false() || second.all_false()) {
    return std::pair(terminalIdsFirst, true);
  }
  auto perm = internal::permutations(terminalIdsFirst, first);

  auto secondMin = internal::minimalColouringBitArray(second);
  for (int j = 0; j < perm.second.size(); j++) {
    auto cba = internal::minimalColouringBitArray(perm.second[j]);
    bool res = true;
    for (ColouringBitArray::Index i(0, 0); i < cba.size(); i++) {
      if (cba[i] && secondMin[i]) {
        res = false;
        break;
      }
    }
    if (res) {
      return std::pair(perm.first[j], true);
    }
  }
  return std::pair(terminalIdsFirst, false);
}

// takes 2 graphs
// returns pair: bool => if they can be joined into a snark
// vector<int> => terminalIds of first graph in order in which it is
    possible to
// create a snark, or original order if impossible
std::pair<std::vector<int>, bool> canCreateSnark(Graph &first, Graph
    &second) {
  // get cba for both multipoles and terminal ids for first
  auto cbaFirst = getAllColourings(first);
  auto cbaSecond = getAllColourings(second);

  auto terminals = internal::getTerminals(first);
  std::vector<int> terminalIds;
```

```
  for (auto &v : terminals) {
    terminalIds.push_back(v.to_int());
  }

  return canCreateSnark(terminalIds, cbaFirst, cbaSecond);
}
```

Toto je kód programu, ktorý berie na vstupe *std* :: *cin* multipóly vo formáte *graph*6 a vypisuje tie páry, ktoré sa dajú spojiť do snarku.

```
...
int count = 0;

void callback(std::string s, Graph &g, Factory &f,
              std::vector<std::pair<int, ColouringBitArray>> *bitArrays) {
  try {
    auto terminals = g.list(RP::degree(1), RT::n());
    if (terminals.size() < 1)
      return;
    auto cba = getAllColourings(g);
    if (!cba.all_false()) {
      (*bitArrays).emplace_back(std::pair(count, cba));
    }
  } catch (std::runtime_error &e) {
  }
  count++;
}

int main() {
  std::vector<std::pair<int, ColouringBitArray>> bitArrays;
  read_graph6_stream<std::vector<std::pair<int, ColouringBitArray>>>(
      std::cin, callback, &bitArrays);

  std::vector<int> terminalIds;
  for (int i = 0;
       i < colouringBitArrayToColourSet(bitArrays[0].second)[0].size(); i++) {
    terminalIds.emplace_back(i);
  }
  for (int i = 0; i < bitArrays.size(); i++) {
    auto &first = bitArrays[i];
    for (int j = i; j < bitArrays.size(); j++) {
      auto &second = bitArrays[j];
```

```
      auto res = canCreateSnark(terminalIds, first.second, second.second);
      if (res.second) {
        std::cout
            << "Graphs number " << first.first << " and " << second.first
            << " can create a snark, with first having terminals in order:"
            << std::endl
            << res.first << std::endl;
      }
    }
  }

  return 0;
}
```

3.2.2 Farebné množiny

Ďalším užitočným programom je *colourSets*, ktorý berie na vstupe *std* :: *cin* multipóly, vypočíta pre ne farebnú množinu v kanonickom tvare a nakoniec vypíše všetky rôzne ktoré našiel.

```
...
using namespace ba_graph;

// Reads from stream and outputs unique colouringSets

u_int32_t count = 0;

void callback(std::string s, Graph &g, Factory &f,
              std::set<ColourSet> *colourSets) {
  try {
    auto terminals = g.list(RP::degree(1), RT::n());
    if (terminals.size() < 1)
      return;
    auto res = getCanonicalColourSet(g);
    if (res.second.size() > 0) {
      (*colourSets).emplace(res.second);
      count++;
    }
  } catch (std::runtime_error &e) {
  }
}
```

```
// Pipe in all k-poles to get all found k-colour-sets
int main() {
  std::set<ColourSet> colourSets;
  read_graph6_stream<std::set<ColourSet>>(std::cin, callback, &colourSets);

  std::cout << "Multipoles: " << count << std::endl;
  std::cout << "Number of unique ColourSets in canonical form: "
            << colourSets.size() << std::endl;
  for (auto &colourSet : colourSets) {
    std::cout << colourSet << std::endl;
  }

  return 0;
}
```

Kapitola 4

Pregraph

Spolu s novými funkciami na prácu s multipólmi vznikla potreba pre multipóly. Preto využívame Pregraph.

Pregraph [1] je program vydaný pod GNU licenciou, slúžiaci na rýchle generovanie súvislých, kubických grafov s rôznymi vlastnosťami.

Umožňuje nám jednoduché a rýchle vytvorenie veľkého množstva multipólov s malým počtom vrcholov, na ktorých už vieme ďalej pracovať.

4.0.1 Formát pregaph

program *Pregraph* ukladá grafy vo svojom vlastnom binárnom formáte. Každý súbor má hlavičku s, podľa ktorej vieme zistiť či používa veľký, či malý endian. Potom nasledujú grafy. Graf začína počtom hrán, potom postupne vymenúva ku každej hrane k nej pripojené vrcholy a zakončí nulou. Pre multipóly je ešte jeden vrchol naviac, s ktorým boli spojené trčiace hrany, kvôli jednoduchosti zápisu. Po vymenovaní poslednej hrany okamžite začína ďalší graf, ak existuje.

Bolo potrebné dopísať do knižnice funkcionalitu na načítanie tohto formátu, a pomocné funkcie. Všetky funkcie nepracujú priamo so súborom, ale namiesto toho berú ako argument prúd znakov, čo znamená, že je ich možne použiť aj na reťazce písmen, čo sa dá využiť pri testovaní, kde namiesto pomalého načítavania zo súboru, máme reťazec znakov zadrôtovaný v teste, už pripravený na načítanie.

Na načítanie je možné použiť napríklad funkciu $read_pregraph_file$, ktorá ako jeden zo svojich parametrov berie aj funkciu, ktorú zavolá na graf ihneď po jeho načítaní, teda nie je potrebné čakať kým sa všetky grafy načítajú a hlavne ich nie je potrebné mať všetky v pamäti.

Aj napriek tomu, že nemáme v úmysle zapisovať v tomto formáte, boli vytvorené aj funkcie pre zapisovanie, najmä kvôli kontrole správneho načítania.

Tabuľka 4.1: Počty multipólov podľa počtu trčiacich hrán s menej ako 16 vrcholmi

k	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
	8595	9538	72306	40134	211157	75968	311942	79873	272272
k	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	
	51261	150376	20589	53125	4901	11289	552	1132	

4.0.2 Rozdelenie multipólov

Boli vygenerované kubické multipóly veľkosti do 20 vrcholov. Po vygenerovaní multipólov ich bolo treba rozdeliť do viacerých súborov. Rozdelili sme ich podľa počtu trčiacich hrán, počtu vrcholov a obvodu. Chceli sme ich rozdeliť aj podľa cyklickej súvislosti, ale nemáme algoritmus, ktorý by ju počítal pre nekompletné grafy, čo sme nechali ako otvorený problém.

Záver

Definovali a implementovali sme kanonický tvar pre farebnú množinu multipólu do knižnice $ba - graph$. Taktiež sme vytvorili menšie programy, ktoré používajú túto novú funkcionalitu a vygenerovali mnohé grafy s malým počtom vrcholov, kvôli ktorým sme museli naprogramovať podporu pre nový formát uloženia grafu. Dokonca sme napísali aj testy, ktoré pokrývajú aspoň základnú funkctionalitu funkcií nového rozhrania.

Prirodzene, stále je čo vylepšovať, stále je možné upraviť kód pre lepšiu čitateľnosť či rýchlosť, napísať viacej testov, spísať viacej nápomocných komentárov k funkciám, alebo ďaľšie menšie programy na demonštráciu. Napriek tomu máme dôveru, že sme splnili zadanie.

Literatúra

[1] T. Pisanski G. Brinkmann, N. Cleemput. Pregraph. `http://caagt.ugent.be/pregraphs/`.

[2] M. Škoviera R. Nedela. Decompositions and reductions of snarks. *J. Graph Theory*, pages 253–291, 1996.

[3] Peter Guthri Tait. Remarks on the colourings of maps. *Proceedings of the Royal Society of Edinburgh*, 1880.