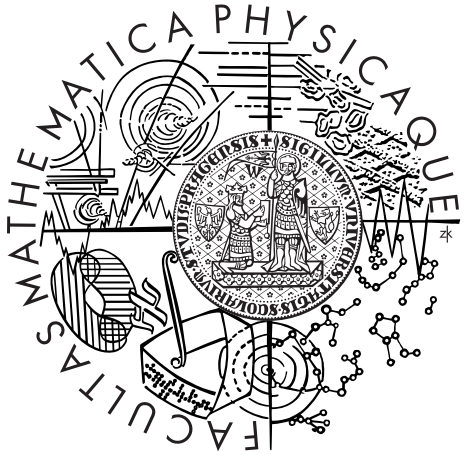
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Jana Rapavá

Algoritmy založené na omezené expanzi - implementace a vyhodnocení

Informatický ústav Univerzity Karlovy

Vedoucí bakalářské práce: doc. Mgr. Zdeněk Dvořák, Ph.D.
Studijní program: Informatika
Studijní obor: obecná informatika

Praha 2014

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Název práce: Algoritmy založené na omezené expanzi - implementace a vyhodnocení

Autor: Jana Rapavá

Ústav: Informatický ústav Univerzity Karlovy

Vedoucí bakalářské práce: doc. Mgr. Zdeněk Dvořák, Ph.D, Informatický ústav Univerzity Karlovy

Abstrakt: Tato bakalářská práce navazuje na větu, která říká, že mnoho tříd grafů - konkrétně třídy s omezenou expanzí - má vlastnosti zjednodušující rozhodování grafových problémů definovatelných v logice prvního řádu. Důležitým příkladem takového problému je izomorfismus podgrafů. Cílem této práce je implementovat a otestovat navrhnutý algoritmus pro tento problém (který má lineární časovou složitost vzhledem k velikosti grafu, ve kterém hledáme podgraf).

Klíčová slova: izomorfismus podgrafů, omezená expanze, stromová dekompozice

Title: Algorithms based on bounded expansion - implementation and evaluation

Author: Jana Rapavá

Department: Computer Science Institute of Charles University

Supervisor: doc. Mgr. Zdeněk Dvořák, Ph.D, Computer Science Institute of Charles University

Abstract: This thesis builds upon the result that a lot of graph classes - namely classes with bounded expansion - have properties which make deciding graph problems definable in first-order logic easier. One very important example of such a problem is subgraph isomorphism. The purpose of this work is to implement and test proposed algorithm for this problem (which has a linear time complexity in relation to the size of graph we are looking for the subgraph in).

Keywords: subgraph isomorphism, bounded expansion, tree decomposition

Názov práce: Algoritmy založené na obmedzenej expanzii - implementácia a vyhodnotenie

Autor: Jana Rapavá

Ústav: Ústav informatiky Karlovej univerzity

Vedúci bakalárskej práce: doc. Mgr. Zdeněk Dvořák, Ph.D, Ústav informatiky Karlovej univerzity

Abstrakt: Táto bakalárska práca stavia na výsledku, ktorý nám vraví, že veľa tried grafov - konkrétne triedy s obmedzenou expanziou - má vlastnosti zjednodušujúce rozhodovanie grafových problémov definovateľných v logike prvého rádu. Dôležitým príkladom takého problému je izomorfizmus podgrafov. Cieľom tejto práce je implementovať a otestovať navrhnutý algoritmus pro tento problém (ktorý má lineárnu časovú zložitosť vzhledem k veľkosti grafu, v ktorom hľadáme podgraf).

Kľúčové slová: izomorfizmus podgrafov, obmedzená expanzia, stromová dekompozícia

Obsah

Úvod

V praxi sa často stretávame s modelmi skutočnosti, ktoré používajú grafy. Konkrétny problém, ktorý potrebujeme vyriešiť, potom často môžme previesť na nejakú zo štandardných grafových úloh. Jednou z takýchto úloh je zistenie, či daný graf obsahuje daný podgraf - problém izomorfizmu podgrafov.

Tento problém má bohaté aplikácie v rozličných oblastiach, od modelovania molekúl, cez strojové učenie a rozpoznávanie vzorov, až po algoritmy počítačového videnia. [1]

Cieľom tejto bakalárskej práce je implementovať a otestovať algoritmus pre rozpoznávanie izomorfných podgrafov založený na stromových dekompozíciach.

Prvá kapitola sa zaoberá pojmami a výsledkami z teórie grafov, ktoré pre tento algoritmus potrebujeme.

Druhá kapitola popisuje samotnú implementáciu.

Záver poskytuje zhrnutie získaných výsledkov a naznačuje možné smery pre ďalší výskum v tejto oblasti.

1. Prehľad potrebnej teórie

V tejto kapitole sa nachádza stručný zoznam definícii a viet, ktoré budeme potrebovať v ďalších častiach práce.

Problém izomorfizmu podgrafov je definovaný nasledovne:

Definícia. *[2]*
***Inštancia**: Grafy $G = (V_1, E_1)$, $H = (V_2, E_2)$.*
***Zadanie**: Obsahuje G podgraf izomorfný H? Inými slovami, existujú $V \subseteq V_1$, $E \subseteq E_1$ a bijekcia $f : V_2 \to V$, pre ktoré platí, že $|V| = |V_2|$, $|E| = |E_2|$ a $\{u, v\} \in E_2 \Leftrightarrow \{f(u), f(v)\} \in E$?*

Ako vidíme z [2], tento problém je NP-úplný; pravdepodobne teda pre jeho všeobecnú inštanciu neexistuje algoritmus s polynomiálnou časovou zložitosťou.

Súčasné algoritmy pre problém izomorfizmu podgrafov môžme rozdeliť do troch skupín. V prvej skupine sa nachádzajú algoritmy založené na modifikácii a vylepšeniach backtrackingu, ako napríklad klasický algoritmus Ullmana [3], alebo techniky prerezávania stromu predstavené v [4]. V druhej skupine sú stratégie poskytujúce približné, ale často dostatočne dobré riešenia - neurónové siete, simulované žíhanie, alebo genetické algoritmy. [1] V neposlednom rade vieme efektívne riešiť niektoré špeciálne inštancie tohto problému - prípady, keď je G les a H strom [2], alebo keď má H špecifický tvar [5].

Nešetřil a Mendez vo svojich článkoch [6] a [7] zaviedli pojem tried grafov s obmedzenou expanziou a poskytli algoritmus pre hľadanie izomorfných podgrafov v grafoch, ktoré patria do takýchto tried. Ich algoritmus je založený na stromovej dekompozícii grafu G. Aby sme sa mohli zaoberať touto problematikou, potrebujeme najskôr definovať nasledovné pojmy:

Definícia. Vzdialenosť $d(x, y)$ *medzi vrcholmi x a y v grafe je dĺžka najkratšej cesty spájajúcej x a y, alebo ∞, ak x a y patria do rozličných komponent súvislosti.*

Definícia ([7])**.** Polomer *súvislého grafu G je definovaný ako:*

$$\rho(G) = \min_{r \in V(G)} \max_{x \in V(G)} d(x, r).$$

Polomer nesúvislého grafu je maximum z polomerov jeho komponent súvislosti.

Definícia ([7])**.** *H je* minor *grafu G, ak je H možné získať z G postupnosťou kontrakcií hrán, odstránení hrán a odstránení vrcholov; značíme $H < G$.* Hĺbka minoru *H grafu G depth(H,G) je najmenší polomer časti G, ktorú musíme kontrahovať, aby sme dostali H.*

Definícia ([6])**.** *Buď $f : \mathcal{N} \to \mathcal{R}$. G má* expanziu obmedzenú *f, ak pre každý $G' < G$ získaný kontrakciou disjunktného zjednotenia súvislých grafov s polomerom $\leq r$ a odstraňovaním vrcholov platí, že hranová hustota G' je zhora ohraničená $f(r)$.*

Definícia ([7])**.** Grad *grafu G hodnosti r je*

$$\nabla_r(G) = \max_{\substack{H<G, \\ depth(H,G) \leq r}} \frac{|E(H)|}{|V(H)|}.$$

Definícia ([6]). *Buď $\mathcal{C}$ trieda grafov. $\mathcal{C}$ má* obmedzenú expanziu, *ak $\exists f : \mathcal{N} \to \mathcal{R}$ t.ž. $\forall G \in \mathcal{C}\ \forall r : \nabla_r(G) \leq f(r)$.*

Teraz si predstavíme definície a výsledky z niekoľkých ďalších potrebných oblastí.

1.1 Acyklické orientácie grafov

Definícia. *Buď $G = (V, E)$ neorientovaný graf.* Orientácia *grafu G je orientovaný graf $G' = (V, E')$ t.ž. $(u, v) \in E \Leftrightarrow (u, v) \in E' \vee (v, u) \in E'$.*

Veta 1 ([7]). *Buď $G = (V, E)$, $|V| = n$, $|E| = m$. G má orientáciu s maximálnym vstupným stupňom k práve vtedy, keď $k \geq \nabla_0(G)$. Navyše existuje algoritmus, ktorý nájde acyklickú orientáciu G s maximálnym vstupným stupňom $\lfloor 2\nabla_0(G) \rfloor$ v čase $O(n+m)$.*

Algoritmus funguje nasledovne: nájde v aktuálnom grafe vrchol so stupňom $\leq 2\nabla_0(G)$, všetkým hranám, ktoré sú s ním incidentné, priradí orientáciu do tohto vrcholu, odstráni vrchol a zavolá sa rekurzívne na zvyšok grafu.

1.2 Tranzitívne fraternálne augmentácie

Definícia ([7]). *Buď $G = (V, E)$ orientovaný graf.* Striktná 1-tranzitívna fraternálna augmentácia *je orientovaný graf $H = (V, E')$, kde E' obsahuje všetky hrany z G a navyše spĺňa nasledovné podmienky:*

- $(x, z), (z, y) \in E \Rightarrow (x, y) \in E'$
- $(x, z), (y, z) \in E \Rightarrow (x, y), (y, x)$ *alebo oboje ležia v E'*
- *E' neobsahuje nijaké ďalšie hrany.*

Definícia ([7]). Striktná tranzitívna fraternálna augmentácia *orientovaného grafu G je postupnosť $G = G_1 \subseteq G_2 \subseteq \ldots G_i \subseteq G_{i+1} \subseteq \ldots$, pre ktorú platí, že G_{i+1} je striktná 1-tranzitívna augmentácia G_i pre $i \geq 1$.*

Tranzitívna fraternálna augmentácia grafu sa dá spočítať v lineárnom čase [7].

1.3 Centrované ofarbenia

Definícia ([6]). *Buď $p \in \mathcal{N}$. Buď H ľubovoľný súvislý podgraf G. Potom* p-centrované ofarbenie *grafu G je ofarbenie grafu G, v ktorom má každé H jednu z nasledovných vlastností:*

- *v H sa nejaká farba vyskytuje práve raz*
- *v H sa vyskytuje aspoň p rozličných farieb.*

Ofarbenie grafu, ktoré je p-centrované, sa dá nájsť v lineárnom čase pomocou štandardného žravého algoritmu [7].

1.4 Stromové dekompozície

Definícia ([7])**.** Stromová dekompozícia *grafu G je dvojica (T, λ), kde T je strom a λ je funkcia z vrcholov T do podmnožín $V(G)$ s nasledovnými vlastnosťami:*

- $\forall v \in V(G) : \{x \in V(T) : v \in \lambda(x)\}$ *indukuje podstrom T*
- $\forall \{v, w\} \in E(G)\, \exists x \in V(T) : \{v, w\} \subseteq \lambda(x)$.

Definícia. Uzáver *clos(F) zakoreneného lesa F je graf s množinou vrcholov $V(F)$ a množinou hrán $\{\{x, y\}$: x je predok y v $F \wedge x \neq y\}$*

Definícia. Stromová hĺbka *súvislého grafu G je minimálna výška zakoreneného stromu, ktorého uzáver obsahuje G ako podgraf.*

Myšlienka, že centrované ofarbenie grafu sa dá použiť na konštrukciu jeho stromovej dekompozície, pochádza z článku [7]. Algoritmus uvedený v tomto článku ale nefunguje napríklad v prípade cesty na troch vrcholoch. Tu však použijeme jeho základnú ideu: ak máme centrované ofarbenie grafu G p farbami, vieme z neho skonštruovať zakorenený les výšky p, ktorý obsahuje G vo svojom uzávere.

Jeden z možných postupov je znázornený v 1.

Ak je G súvislý, algoritmus vráti zakorenený strom Y. Z neho vieme určiť stromovú dekompozíciu G: položíme $T = Y$, $\lambda(x) = \{v \leq_Y x\}$ (kde $\leq_Y$ je čiastočné usporiadanie definované clos(Y)).

Ak je G navyše z triedy grafov s obmedzenou expanziou, jeho stromová dekompozícia nám dáva rozklad G, v ktorom každých $i \leq p$ častí indukuje podgraf so stromovou hĺbkou $\leq i$. [6]

1.5 Hľadanie izomorfných podgrafov

Jednou z možností využitia stromovej dekompozície je zistenie, či je graf H izomorfný s nejakým podgrafom grafu G, a to v lineárnom čase vzhľadom k veľkosti grafu G. [7]

2 predstavuje jeden z možných algoritmov pre tento problém.

Veta 2. *2 má časovú zložitosť $O(2^{|V(H)|}|V(G)|)$.*

Algoritmus 1 Hľadanie zakoreneného lesa obsahujúceho G

Vstup: c: centrované ofarbenie grafu G p farbami

Výstup: F: zakorenený les výšky p, ktorý má G vo svojom uzávere

```
Buď F = nil; root, prevRoot ∈ V(G).
for all G_i komponenta súvislosti G do
  used ← new Boolean[p].
  root ← null.
  prevRoot ← null.
  rootColor ← 0.
  prevRootColor ← 0.
  for all n ∈ V(G_i) do {hľadáme vrchol, ktorého farba sa vyskytuje v G práve raz}
    if n nie je navštívený then
      if used[c(n)] then
        if c(n) = rootColor then
          root ← prevRoot.
          rootColor ← prevRootColor.
        end if
      else
        prevRoot ← root.
        prevRootColor ← rootColor.
        root ← n.
        rootColor ← c.
      end if
      used[c(n)] ← true.
    end if
  end for
  Zavolaj sa rekurzívne na G\{root}, rekurzívne volanie vráti zakorenený les F'.
  Pridaj k F strom, ktorý má ako koreň root a synovia koreňa sú stromy z F'.
end for
```

Algoritmus 2 Detekcia podgrafu grafu G, ktorý je izomorfný H

Vstup: T stromová dekompozícia grafu G, H neorientovaný graf.
Výstup: **true** ak G má indukovaný podgraf izomorfný s H, **false** inak

```
for all P cesta z koreňa do vrcholu pri DFS prechode T do
  sg' ← new Boolean[][][].
  y ← posledný vrchol P.
  for all H' ≤ H do
    for all S ⊆ V(H') do
      for all f:S → V(P) prostá funkcia do
        if (backtrack) then
          if y ∈ f(S) then
            n ← vzor y v f.
            if hrany v G nezodpovedajú hranám medzi y a zvyškom H then
              sg'[H'][S][f] ← false.
            end if
          else
            sg'[H'][S][f] ← sg[H' \ {n}][S \{n}][f \{(n → y)}].
          end if
        else
          sg'[H'][S][f] ← sg[H'][S][F].
        else
          sg'[H'][S][f] ← sg[H'][S][f] or ⋁_{v∈V(H')\S} sg[H'][S ∪{v}][f ∪{v → y}].
        end if
      end for
    end for
  end for
  for all w do
    if (sg'[length1][length2][w]) then
      return true
    end if
  end for
end for
return false
```

2. Implementácia

2.1 Výber programovacieho jazyka

Pretože ide o bakalársku prácu zameraná na algoritmy, je rozumné vybrať si vysokoúrovňový programovací jazyk, ktorý programátorovi umožní nestrácať čas implementačnými detailami. Okrem toho to musel byť jeden z programovacích jazykov, v ktorých som už písala väčšie projekty.

Toto kritérium zúžilo množinu použiteľných jazykov na Javu a Haskell. Podrobný pohľad na štruktúru potrebných algoritmov nám odhalí, že kvôli časovej zložitosti budeme často potrebovať pracovať s poľami. Pretože práca s poľami v Haskelli sa mi nejavila ako dostatočne jednoduchá a prirodzená, táto bakalárska práca je nakoniec implementovaná v Jave.

Pre beh tohto programu je potrebná Java 7.

2.2 Návrh programu

Program je možné intuitívne rozdeliť na dve časti. Prvá implementuje algoritmus pre stromovú dekompozíciu, druhá časť je tvorená samotným algoritmom pre detekciu izomorfných podgrafov.

2.3 Užívateľské rozhranie

Program je distribuovaný ako JAR archív a spúšťa sa z príkazového riadka. Pri spustení prijíma štyri argumenty - súbor obsahujúci prvý graf, súbor obsahujúci druhý graf, požadovaný počet augmentácií a počet farieb, ktoré je možné použiť pri hľadaní p-centrovaného ofarbenia. Formát vstupného súboru je podrobne popísaný v 2.4.4. Výsledok výpočtu sa vypisuje na štandardný výstup.

2.4 Detaily implementácie

Každá podkapitola sa zaoberá jedným zo zdrojových súborov programu a vysvetľuje jeho obsah.

2.4.1 DegreeQueue.java

Dva z použitých algoritmov rekurzívne vyberajú v grafe vrchol s najmenším stupňom. Trieda *DegreeQueue* implementuje dátovú štruktúru, ktorá toto umožňuje. Štruktúra je implementovaná ako pole stupňov, kde každá položka obsahuje odkaz na zoznam vrcholov daného stupňa.

Trieda QueueItem slúži len ako wrapper, pretože Java nevie vytvoriť pole zoznamov špecifického typu, a to ani pomocou reflexie. Generické triedy sú v Jave implementované spôsobom, ktorý neumožňuje získať odkaz na triedu LinkedList<InputNode> [8].

Je síce možné vytvoriť pole zoznamov nešpecifikovaného typu [8], to však pre moje potreby nie je dostatočne typovo bezpečné.

Na vytvorenie *DegreeQueue* potrebujeme plne inicializovaný graf.

Na objektoch tejto triedy je možné volať metódu *update()*, ktorá aktualizuje frontu po zmazaní vrcholu. Vrcholy, ktorých stupeň sa zmenil, pridávame na koniec pôvodného zoznamu, ale mohli sme si rovnako dobre vybrať aj začiatok. Keď chceme odobrať vrchol s najmenším stupňom, odoberáme ho zo začiatku zoznamu.

Tento spôsob nám vyhovuje, pretože príslušným algoritmom záleží len na stupni a fungujú správne s ľubovolným vrcholom najmenšieho stupňa.

(To si môžeme ukázať na príklade algoritmu, ktorý hľadá orientáciu grafu: Predpokladajme, že máme dva vrcholy stupňa minDeg, čo je najnižší stupeň, aký sa v grafe vyskytuje. Potom tieto vrcholy môžem vybrať v ľubovoľnom poradí - buď spolu susedia a jeden z nich musí teda mať vo vzniknutej orientácii vstupný stupeň minDeg, alebo spolu nesusedia a jeden z nich musím vybrať ako prvý.)

Časová zložitosť: Potrebujeme, aby sme za celú dobu behu algoritmu strávili vo volaniach tejto metódy len lineárny čas. Pretože odstraňujeme vrcholy jeden po druhom, až kým nie je graf prázdny, potrebujeme, aby každé volanie *update()* stálo amortizovane konštantný čas.

Na dosiahnutie takejto časovej zložitosti si potrebujeme pamätať najnižší stupeň vrcholu, ktorý je momentálne vo fronte. Ten označme k. Keď z grafu odstránime vrchol a zavoláme na príslušnej *DegreeQueue* metódu *update()*, nový najmenší stupeň vo fronte musí byť z intervalu $[k-1, n-1]$ (kde n je aktuálny počet vrcholov).

Nasleduje rozbor prípadov:

- **stupeň je k-1:** Pretože sme zmazali vrchol s najmenším stupňom, stupeň $k-1$ môže mať len bývalý sused odstráneného vrcholu. Stačí teda skontrolovať všetkých susedov zmazaného vrcholu.

- **stupeň je k:** Stačí skontrolovať, či je príslušný index v *DegreeQueue* neprázdny.

- **stupeň sa zvýšil:** Musíme nájsť najbližší neprázdny stupeň; označme l.

  Ak sa najbližší neprázdny stupeň nachádza nízko, prejdeme malú vzdialenosť.

  Ak sa najbližší neprázdny stupeň nachádza vysoko, stupne všetkých vrcholov grafu prislúchajúceho *DegreeQueue* musia byť z intervalu $[l, n-1]$. V tomto úseku teda nebudú skoro žiadne prázdne priehradky. Nutne teda musí nasledovať postupnosť krokov, kedy prejdeme len malú vzdialenosť, alebo nový najnižší stupeň zistíme triviálne.

Po spísaní príslušných rovností zistíme, že celé pole počas behu algoritmu prejdeme len konštantný počet krát. Z toho vyplýva, že časová zložitosť jednej operácie *update()* je amortizovane konštanta.

2.4.2 Edges.java

Tento súbor obsahuje triedu *Edge*, ktorá implementuje dátovú štruktúru reprezentujúcu hranu, a triedu *Edges*, ktorá je knižnicou statických metód pre prácu so zoznamami hrán.

Okrem nich obsahuje aj prevodné funkcie z orientovaného grafu na zoznam hrán *digraph2edges* a zo zoznamu hrán na neorientovaný graf *edges2graph* (ktorá sa do triedy Graph nehodí, pretože nemá generický typ).

Mať prevodné funkcie takto asymetrické môže vyzerať zvláštne, ale opačný smer prevodu som počas implementácie nepotrebovala.

2.4.3 Graph.java

Táto trieda obsahuje dátové štruktúry pre neorientované a orientované grafy.

Prvý návrh obsahoval jednu triedu pre oba typy grafov; ohľadom internej reprezentácie som sa rozhodovala medzi zoznamom hrán (do ktorého by sa jednoduchšie pridávali hrany) a zoznamom susedov (z ktorej je možné rýchlo odstraňovať vrcholy). Uvedená tabuľka popisuje požadované operácie a ich časové zložitosti. (Používame konvenciu, kde n značí počet vrcholov, m počet hrán a m_1 počet pridaných hrán.)

Operácia	LinkedList<Pair<Integer, Integer>>	Graph
Konštrukcia	$\mathcal{O}(m)$	$\mathcal{O}(m+n)$
Odstránenie vrcholu	$\mathcal{O}(m)$	$\mathcal{O}(n)$
Uniq	$\mathcal{O}(m)$	$\mathcal{O}(m+n)$
Merge	$\mathcal{O}(m+m_1)$	$\mathcal{O}(n+m+m_1)$

Nakoniec som sa namiesto odstraňovania vrcholov rozhodla označovať si už spracované vrcholy. Riešenie problému s násobnými hranami je popísané v časti 2.4.8.

Finálna reprezentácia neorientovaných grafov je pomerne komplikovaná. Objekty predstavujúce vrcholy grafu sú uložené do zoznamu. Každý takýto vrchol obsahuje svoj názov, stupeň, stav navštívenosti a odkaz na zoznam hrán, ktoré z neho vedú. Každá hrana sa zároveň odkazuje na tú istú hranu v opačnom smere.

Vzťah "byť hranou v opačnom smere" je reprezentovaný triedou *LinkEdge*. Tá ešte obsahuje odkaz na vrchol na druhom konci hrany, aby sme vždy, keď označíme súčasný vrchol ako zmazaný, vedeli efektívne znížiť stupeň všetkých jeho susedov. To je tiež dôvod, prečo je tento odkaz reprezentovaný typom *LinkEdge* a nie typom *Edge*.

Táto dátová štruktúra má tú nevýhodu, že ju nie je možné úplne inicializovať už pri načítaní vstupu - na jej vytvorenie je potrebné prejsť vstupný graf dvakrát. Druhý prechod má na starosti metóda *postInitNodes*.

Trieda *Graph* ešte implementuje pomocné metódy *empty()*, *isLastVertex()* a *getMinDeg()* a rozklad na komponenty súvislosti.

Orientované grafy sú nakoniec reprezentované samostatnou triedou *Digraph*, pretože potrebujú implementovať iné metódy ako neorientované grafy - menovite pridanie hrany a zahodenie orientácie. Interne sú reprezentované poľom zoznamov predchodcov, čo nám umožňuje nájsť TFA v lineárnom čase.

2.4.4 Main.java

Tento súbor obsahuje okrem metódy *main()* aj funkcie na parsovanie vstupu.

Formát vstupných grafov je popísaný nasledovnou gramatikou:

Graph = Node\n |Node\n Graph
Node = Integer–List
List = [] |[Int(,Int)*]

Pokiaľ vstupné grafy nezodpovedajú tomuto formátu, program vyhadzuje výnimku.

2.4.5 Node.java

Tento súbor obsahuje všetky triedy, ktoré sa dajú použiť ako prvky grafu orientovaného alebo neorientovaného grafu. Každý potenciálny uzol v neorientovanom grafe dedí od abstraktnej triedy *Node* a implementuje rozhranie *NodeInterface*. Navyše má každá z nich má svoju vlastnú položku nbs, ktorá zatieňuje tie zdedené z nadradených tried.

2.4.6 NodeInterface.java

NodeInterface obsahuje metódy, ktoré umožnia implementujúcim triedam byť prvkami grafu. Ide o prístup k jednotlivým dátovým položkám, iteráciu na susedoch a porovnávanie s uzlom toho istého typu.

Ide o generický interface, pretože ak mám v Jave triedu, ktorá implementuje iterátor, trieda, ktorá od nej dedí, nemôže implementovať iterátor nad iným typom.

2.4.7 Tree.java

Tento súbor obsahuje implementáciu n-árneho stromu, ktorú používa algoritmus pre stromovú dekompozíciu.

2.4.8 TreeDecomposition.java

findOrientation()

Táto metóda implementuje algoritmus, ktorý nájde acyklickú orientáciu grafu s nízkym vstupným stupňom. *findOrientation()* prijíma ako parameter pomocnú štruktúru typu *Digraph*, do ktorej len dopĺňa predchodcov príslušných vrcholov (tým zaručí správne poradie vrcholov).

findAugmentation()

Táto metóda počíta striktnú TFA zadaného grafu tak, že cyklicky hľadá a pridáva do grafu tranzitívne a fraternálne hrany (pozri 1.2).

Aby sme zabránili pridávaniu hrán, ktoré už existujú, vytvorený graf prevedieme na zoznam hrán, ten utriedime priehradkovým triedením, odstránime duplicity a výsledný zoznam prevedieme na graf. Na tieto operácie používame funkcie z triedy *Edges*.

Pri testovaní funkčnosti *findAugmentation()* som použila nasledovný rozbor prípadov:

Buď u a v vrcholy grafu G (bez slučiek a násobných hrán), medzi ktorými po iterácii *findAugmentation()* majú viesť hrany $u \to v$ a $v \to u$. Tieto dve hrany môžu byť typu:

- **tranzitívna-tranzitívna** Aby nastal tento prípad, v orientovanom grafe, s ktorým pracujeme, by sa musel vyskytovať orientovaný cyklus. *findOrientation()* ale vracia len acyklické orientácie, tento prípad teda nemôže nastať.

- **fraternálna-fraternálna** V tomto prípade sa pridá len jedna hrana v smere, ktorý určí *findOrientation()*.

- **tranzitívna-fraternálna** Pridá sa len jedna hrana v smere určenom tranzitívnou hranou.

- **pôvodná-fraternálna** Pridá sa nová fraternálna hrana.

- **pôvodná-tranzitívna** Tento prípad si vyžaduje cyklickú orientáciu, čo nemôže nastať.

Výpočet TFA by bolo možné zrýchliť tak, že si každý vrchol bude pamätať poslednú iteráciu, v ktorej sa mu zvýšil stupeň, a dôjde k prepočítavaniu len v prípade, kedy sa v predchádzajúcej iterácii zmenil stupeň aspoň jedného zo zúčastnených vrcholov. Ďalej nie je potrebné, aby algoritmus, ktorý dopĺňa tranzitívne hrany, testoval vrcholy, ktoré už majú stupeň $n - 1$.

findColoring()

Táto metóda hľadá ofarbenie grafu p farbami. Štruktúra rekurzie je podobná ako vo funkcii *findOrientation()*.

Pri implementácii metódy *findColoring* som sa rozhodovala medzi grafom ofarbených uzlov (čo si vyžadovalo generickú triedu *Graph*) a zoznamom dvojíc <popiska_vrcholu, farba>. Aby som mohla jednotlivé uzly priamo usporiadať do dekompozičného stromu, rozhodla som sa pre graf. Toto rozhodnutie si však vyžiadalo neelegantný kompromis v metóde *Graph.postInitNodes*, ktorá potrebuje presný typ grafu, na ktorý je zavolaná (inak by nemohla fungovať v lineárnom čase).

findColoring() prijíma ako parameter pomocnú štruktúru typu *Graph<ColoredNode>*, do ktorej pridáva farby príslušných vrcholov. To nám zaručuje, že výstupný graf bude mať vrcholy v očakávanom poradí.

V prípade, kedy graf nie je možné ofarbiť daným počtom farieb, metóda vyhadzuje výnimku.

findFullDecomposition

Táto metóda rozloží graf na komponenty súvislosti a potom nájde stromovú dekompozíciu každej z nich.

Stromová dekompozícia sa konštruuje z centrovaného ofarbenia pomocou 1 - vždy nájdeme v grafe vrchol, ktorého farba sa vyskytuje len raz, odstránime ho z grafu, zvyšok grafu rozložíme na komponenty súvislosti a zavoláme sa na každú z nich rekurzívne.

Mohlo by sa zdať, že tieto opakované rozklady na komponenty súvislosti počas celého behu algoritmu zaberú viac ako lineárny čas, ale to sa nestane, pretože uvažované grafy sú z tried s obmedzenou expanziou [6].

2.4.9 DFS.java

Algoritmus pre hľadanie izomorfných podgrafov je založený na prehľadávaní do hĺbky (DFS). Z dôvodu prehľadnosti som sa rozhodla štruktúru prehľadávania do hĺbky oddeliť od výkonného jadra algoritmu a umiestniť ju sem.

Tento súbor obsahuje triedu *State*, ktorá implementuje dátovú štruktúru reprezentujúcu stav v danom vrchole DFS stromu. Položke *state* by teoreticky stačili rozmery $2^{|V(H)|} \times 2^{|V(H)|} \times p!$, ale nepodarilo sa mi nájsť spôsob, ako správne prepočítať tretiu súradnicu pri prechode DFS stromom, takže v implementácii jej veľkosť závisí aj na počte vrcholov grafu G (čo samozrejme zhoršuje časovú zložitosť).

Ďalej obsahuje triedu *DFS*. Konštruktor objektov tejto triedy prijíma ako parameter strom, ktorý sa má prejsť. Každé zavolanie metódy *stepDFS()* potom vráti cestu z koreňa stromu do vrcholu, v ktorom sa práve nachádzame počas prehľadávania do hĺbky.

2.4.10 Subsets.java

Tento súbor obsahuje triedu *Subset*, ktorá má ako dátové položky podmnožinu nosnej množiny a jej index. *SubsetGenerator* dostane v konštruktore ako parametre nosnú množinu a pôvodnú množinu (tú potrebujeme kvôli indexovaniu). Vygenerovaná podmnožina je interne reprezentovaná ako kombinácia indexov prvkov z nosnej množiny.

Existujú dva základné algoritmy pre generovanie všetkých podmnožín danej množiny. Prvý postupne inkrementuje čítač a v každom kroku vyberie do podmnožiny prvky, ktoré zodpovedajú bitom nastaveným na 1. Druhý vracia podmnožiny v lexikografickom poradí kombinácií prvkov nosnej množiny - postupne generuje kombinácie rádu $1 \ldots |S|$ [9].

Ďalšiu výzvu predstavuje fakt, že daná podmnožina musí mať počas celého behu algoritmu stále ten istý index (bez ohľadu na to, akú nosnú množinu práve skúmame). Kvôli tomuto problému som sa rozhodla nakoniec implementovať druhý algoritmus, ale ukázalo sa, že trpí rovnakými problémami ako prvý. Posledným riešením je generovať reťazce priradené podmnožinám, ukladať ich do hešovacej tabuľky a v nej vždy vyhľadať index práve skonštruovanej podmnožiny. Toto riešenie nie je veľmi efektívne z hľadiska pamäťovej zložitosti, ale pretože veľkosť nosnej množiny je už obmedzená inými faktormi, ide o dostatočne dobré riešenie.

2.4.11 Subgraphs.java

Tento súbor obsahuje triedu *Subgraph*, ktorá reprezentuje podgraf nosného grafu aj s jeho indexom, a triedu *SubgraphGenerator*, ktorá využíva služby triedy *SubsetGenerator*. Využívame fakt, že podgraf je vlastne podmnožina množiny vrcholov, z ktorej sme odstránili všetky hrany vedúce mimo podmnožinu.

2.4.12 Permutations.java

Tento súbor obsahuje triedy *Permutation*, ktorá zapuzdruje permutáciu a jej index do jedného celku, a *PermutationGenerator*, ktorá generuje všetky permutácie

z prvkov 0, ..., n-1 v lexikografickom poradí. Hľadanie nasledujúcej permutácie je založené na zdrojovom kóde z [10].

2.4.13 Functions.java

Tento súbor obsahuje triedy *InjectiveFunction*, ktorá zodpovedá prostej funkcii, a *FunctionGenerator*, ktorá postupne generuje všetky prosté funkcie z danej množiny do danej množiny.

Prostá funkcia je reprezentovaná dvojicou <permutácia indexov oboru hodnôt, počet relevantných položiek>. Tento spôsob nám zaručuje, že ak budeme iterovať cez všetky permutácie, z každej vezmeme len príslušný počet prvých položiek a preskočíme tie, ktoré sme už vygenerovali, prejdeme v konečnom dôsledku všetky variácie daného rádu. O iteráciu cez všetky permutácie sa stará trieda *PermutationGenerator*.

Podrobnejší rozbor tejto techniky sa nachádza napríklad v [11].

Funkcia ďalej obsahuje svoj definičný obor, svoj obor hodnôt a zoznam dvojíc prvkov, ktoré na seba zobrazuje. Tie určuje príslušná variácia oboru hodnôt var: i-tému prvku definičného oboru je priradený prvok z oboru hodnôt, ktorý sa nachádza na indexe $var[i]$.

Označme si veľkosť definičného oboru k a veľkosť oboru hodnôt n. Poradie, v akom sú funkcie generované, potom zodpovedá lexikografickému usporiadaniu na variáciách z n prvkov rádu k.

2.4.14 Combinatorics.java

Táto trieda obsahuje statické metódy pre výpočet počtu permutácií, variácií a kombinácií daného rádu z daného počtu prvkov.

2.4.15 SubgraphIsomorphism.java

Tento súbor obsahuje výkonné jadro algoritmu, ktorý slúži na detekciu izomorfných podgrafov.

Môžem predpokladať, že graf H je súvislý (ak nie, rozložím na komponenty súvislosti a pracujem s každou samostatne). Metóda *findIsomorphicSubgraph()* teda prijme medzi svojimi parametrami grafy G a H a nájde stromovú dekompozíciu grafu G. Potom postupne vezme stromovú dekompozíciu každej komponenty súvislosti G a zavolá na ňu *findIsomorphicSubgraphInComponent()*. Táto metóda je presnou implementáciou algoritmu 2.

Záver

Vzhľadom na to, že implementácia algoritmu sa zatiaľ nejaví ako stopercentne správna, je ťažké urobiť jednoznačný záver. Je však evidentné, že Java sa nakoniec neukázala ako vhodný programovací jazyk pre túto úlohu.

Veľkosť vstupného grafu H je obmedzená rozsahom dátového typu int, pretože ostatné dátové typy v Jave nie je možné použiť na indexovanie polí. Pri implementácii som často musela obchádzať aj iné nedokonalosti a úskalia tohto jazyka.

Celková myšlienka algoritmu však vyzerá rozumne a použiteľne, a jeho realizácia v inom programovacom jazyku (prípadne s niekoľkými optimalizáciami spomenutými v kapitole 2) by sa mohla ukázať ako prakticky úspešná.

Zoznam použitej literatúry

[1] MESSMER, Bruno T., a Horst BUNKE. *Efficient subgraph isomorphism detection: A decomposition approach*. IEEE Transactions on Knowledge and Data Engineering 12 (2) (2000) 307-323 [online]. Dostupné z: IEEExplore. DOI 10.1109/69.842269.

[2] GAREY, Michael R., a David S. JOHNSON. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979. ISBN 0-7167-1045-5.

[3] ULLMAN, J.R. *An Algorithm for Subgraph Isomorphism*. Journal of the ACM 23 (1) (1976) 31-42 [online]. Dostupné z: ACM DL. DOI 10.1145/321921.321925.

[4] HARALICK, R.M. a G.L. ELLIOT. *Increasing Tree Search Efficiency for Constraint Satisfaction Problems*. Artificial Intelligence 14 (3) (1980) 263-313 [online]. Dostupné z: ScienceDirect. DOI 10.1016/0004-3702(80)90051-X.

[5] ALON, N., R. YUSTER a U. ZWICK. *Color-coding* Journal of the ACM 42 (4) (1995) 844-856 [online]. Dostupné z: ACM DL. DOI 10.1145/210332.210337.

[6] NEŠETŘIL, J., a P. Ossona de MENDEZ. *Grad and classes with bounded expansion I. Decompositions*. European Journal of Combinatorics 29 (3) (2008) 760-776 [online]. Dostupné z: ScienceDirect. DOI 10.1016/j.ejc.2006.07.013.

[7] NEŠETŘIL, J., a P. Ossona de MENDEZ. *Grad and classes with bounded expansion II. Algorithmic Aspects*. European Journal of Combinatorics 29 (2008) 777-791 [online]. Dostupné z: ScienceDirect. DOI 10.1016/j.ejc.2006.07.014.

[8] LANGER, Angelika. *Java Generics FAQs* Dostupné z: http://www.angelikalanger.com/GenericsFAQ/FAQSections/ParameterizedTypes.html. Posledná aktualizácia: 10.05.2013.

[9] McCAFFREY, James. *Generating the mth Lexicographical Element of a Mathematical Combination* [online]. Dostupné z: http://msdn.microsoft.com/en-us/library/aa289166%28v=vs.71%29.aspx Vydané v júli 2004.

[10] ISRAEL, Alistair. *JCombinatorics - SepaPnkIterator* [online]. GitHub repozitár. Dostupné z: https://github.com/aisrael/jcombinatorics/blob/master/src/main/java/jcombinatorics/permutations/SepaPnkIterator.java. Vydané dňa: 02.09.2009.

[11] ISRAEL, Alistair. *A Simple, Efficient P(n,k) Algorithm* [online]. Dostupné z: https://alistairisrael.wordpress.com/2009/09/22/simple-efficient-pnk-algorithm/. Vydané dňa: 22.09.2009.