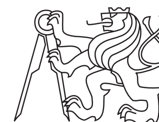
Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky



Bakalářská práce

Automatová knihovna - isomorfismus planárních grafů

David Rosca

Vedoucí práce: Ing. Radomír Polách

11. ledna 2016

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 11. ledna 2016 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2016 David Rosca. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Rosca, David. *Automatová knihovna - isomorfismus planárních grafů*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Předmětem této práce je problém isomorfismu planárních grafů. Práce v první části popisuje implementaci základních grafových struktur a algoritmů. Dále je popsána implementace algoritmu Hopcroft–Wong pro řešení problému isomorfismu planárních grafů v lineárním čase. V závěru práce je porovnán výkon tohoto algoritmu s naivním algoritmem podle definice isomorfismu. Práce je součástí projektu Automatová knihovna.

Klíčová slova isomorfismus planárních grafů, Hopcroft–Wong, automatová knihovna

Abstract

Topic of this thesis is a problem of planar graph isomorphism. First part of the thesis describes implementation of basic graph structures and algorithms. Following is the description of implementation of Hopcroft–Wong algorithm solving planar graph isomorphism in linear time. In the end, there is a comparison of performance of this algorithm and naive algorithm from the definition of isomorphism. This thesis is part of Automata Library.

Keywords planar graph isomorphism, Hopcroft–Wong, automata library

Obsah

Úvod **1**
Cíle práce 1
Struktura práce 1

1 Definice a pojmy **3**

2 Analýza a návrh **7**
2.1 Existující řešení 7
2.2 Automatová knihovna 8

3 Grafový modul Automatové knihovny **11**
3.1 Rozhraní Node 11
3.2 Rozhraní Edge 11
3.3 Rozhraní Graph 12
3.4 Vnitřní reprezentace grafu 14
3.5 Grafové algoritmy 16

4 Hopcroft–Wong algoritmus **19**
4.1 Reprezentace grafu 19
4.2 Popis algoritmu 21
4.3 Pomocné algoritmy 21
4.4 Redukce 23
4.5 Shrnutí 30

5 Testování **31**
5.1 Automatické testy 31
5.2 Srovnání výkonu 31

Závěr **33**

A Seznam použitých zkratek **35**

Literatura **37**

B Obsah přiloženého CD **39**

Seznam obrázků

3.1 Diagram rozhraní Node 11
3.2 Diagram rozhraní DirectedEdge 12
3.3 Diagram rozhraní UndirectedEdge 12
3.4 Diagram rozhraní Graph 13
3.5 Neorientovaný graf 15
3.6 Spojová reprezentace grafu 3.5 15
3.7 Matice sousednosti grafu 3.5 16
3.8 Matice incidence grafu 3.5 16

4.1 Ilustrační příklad interní reprezentace grafu 20
4.2 Celkový koncept Hopcorft–Wong algoritmu 21
4.3 Smyčka 24
4.4 Skein 26
4.5 Nově přidané hrany (červeně) 27
4.6 Operace odstranění význačné hrany 28
4.7 Redukce výjimky uzlu stupně 4 29
4.8 Redukce druhé výjimky uzlu stupně 4 30

5.1 Srovnání výkonu algoritmů 32

Seznam tabulek

3.1 Srovnání časové složitosti operací 14

5.1 Výsledky testování výkonu algoritmů 32

Úvod

Problém isomorfismu grafů patří do skupiny NP problémů, nevíme však zda je v P či NP-úplný. Takové problémy nazýváme NP-přechodné. Obecně se ale věří, že isomorfismus grafů spadá do skupiny P, čemuž nasvědčuje i nejnovější vývoj v oboru.

Z tohoto důvodu se tedy v případě isomorfismu grafů zaměřujeme na specifické typy grafů. Díky tomuto omezení je možné uvažovat algoritmy se složitostí nižší než polynomiální. Jedním z těchto algoritmů je také Hopcroft–Wong algoritmus, rozhodující isomorfismus planárních grafů v lineárním čase.

Cíle práce

Hlavním tématem této práce je naimplementovat Hopcroft–Wong algoritmus do *Automatové knihovny* [1] a porovnat jeho výkon s naivním algoritmem pro isomorfismus grafů podle definice.

Ke splnění tohoto cíle bude nejdříve potřeba přidat podporu pro grafy do *Automatové knihovny.* Cílem není efektivní implementace, naopak důležitější je přehledný kód odpovídající matematickým zápisům algoritmů.

Nakonec bude potřeba implementovat generátor náhodných grafů k testování implementovaných algoritmů.

Struktura práce

V první kapitole jsou definovány základní grafové pojmy. V druhé kapitole je popsána analýza existujících řešení a návrh implementace do *Automatové knihovny.* Kapitola 3 popisuje implementaci datových struktur a základních grafových algoritmů v *Automatové knihovně.* Implementace Hopcroft–Wong algoritmu je popsána v kapitole 5. V poslední kapitole je popsáno testování implementace algoritmů a uvedeny výsledky.

Kapitola 1

Definice a pojmy

Tato kapitola definuje základní grafové pojmy, které se vyskytují v této práci. Definice jsou z velké části převzaty z [9].

Definice 1.1. *Graf* je uspořádaná trojice $G = \langle H, U, \varrho \rangle$, kde:

- H je množina hran grafu G,
- U je množina uzlů grafu G,
- $\varrho : H \to U \times U$ je množina dvojic, určujících krajní uzly hran. Pokud je ϱ množina orientovaných dvojic, nazýváme takový graf *orientovaný*, resp. *neorientovaný*.

Definice 1.2. Graf $G' = \langle H', U', \varrho' \rangle$ nazýváme *podgrafem* grafu $G = \langle H, U, \varrho \rangle$ (zapisujeme $G' \subseteq G$), jestliže platí

$$(H' \subseteq H) \wedge (U' \subseteq U) \wedge \forall h \in H' (\varrho'(h) = \varrho(h)).$$

Podgraf $G' = \langle H', U', \varrho' \rangle$, jehož množina uzlů je shodná s množinou uzlů grafu G, nazýváme *faktorem* grafu G.

Definice 1.3. Hrana h_1 v grafu G je *rovnoběžná*, právě když existuje taková hrana h_2 v grafu G, že $h_1 \neq h_2 \wedge \varrho(h_1) = \varrho(h_2)$.

Definice 1.4. Hranu nazýváme *smyčkou*, právě když oba krajní uzly této hrany jsou stejné.

Definice 1.5. *Prostý* graf je graf bez rovnoběžných hran.

Definice 1.6. *Obyčejný* graf je prostý graf bez smyček.

Definice 1.7. Nechť $G_1 = \langle H_1, U_1, \varrho_1 \rangle$ a $G_2 = \langle H_2, U_2, \varrho_2 \rangle$ jsou dva grafy a φ bijekce množiny $H_1 \cup U_1$ na množinu $H_2 \cup U_2$ taková, že

- zúžené zobrazení $\varphi | H_1$ je bijekce H_1 na H_2 ($\varphi : H_1 \leftrightarrow H_2$)

- zúžené zobrazení $\varphi|U_1$ je bijekce U_1 na U_2 ($\varphi : U_1 \leftrightarrow U_2$)
- φ zachovává incidenci, tzn. pro libovolné $h \in H$ platí

$$\varrho_1(h) = [u, v] \implies \varrho_2(\varphi(h)) = [\varphi(u), \varphi(v)].$$

Zobrazení φ se pak nazývá *isomorfismus* mezi grafy G_1 a G_2 a grafy G_1, G_2, pro které tento isomorfismus lze nalézt, se nazývají *isomorfní grafy* (zapisujeme $G_1 \cong G_2$).

Definice 1.8. Nechť $G = \langle H, U, \varphi\rangle$ je graf, $u \in U$ libovolný uzel a $A \subseteq U$ libovolná podmnožina uzlů. *Množinou sousedů* $\Gamma(u)$ *uzlu* u nazýváme podmnožinu uzlů definovanou vztahem

$$\Gamma(u) = \{v \ \in U \ : \exists h \in H(\varphi(h) = [u, v\ ])\}.$$

Množinu sousedů $\Gamma(A)$ *podmnožiny* A definujeme vztahem $\Gamma(A) = \bigcup_{v\in A} \Gamma(v)$. *Stupněm* $\delta_G(u)$ uzlu u v grafu G nazýváme počet hran s ním incidujících. Symboly $\delta(G)$ a $\Delta(G)$ označujeme minimální, resp. maximální stupeň uzlu v grafu G.

Definice 1.9. Nechť pro danou dvojici uzlů u a v v grafu $G = \langle H, U, \varrho\rangle$ existuje posloupnost uzlů a hran

$$S \ = \langle u_0, h_1, u_1, h_2, \ldots, u_{n-1}, h_n, u_n\rangle$$

kde $h_i \in H, \varrho(h_i) = [u_{i-1}, u_i]$ pro $i = 1, 2, \ldots, n$, $u_i \in U$ pro $i = 0, 1, \ldots, n$, $u_0 = u, u_n = v$. Pak tuto posloupnost nazýváme *sledem* grafu G mezi uzly u a v. Uzly u, v jsou *krajní uzly* sledu S (u je *počáteční*, v *koncový uzel*), uzly $u_1, u_2, \ldots, u_{n-1}$ jsou *vnitřní uzly* sledu S. Číslo $n(\geq 0)$ nazýváme *délkou sledu* S a značíme $d(S)$. Sled s alespoň jednou hranou, v němž jsou uzly u a v shodné, nazýváme *uzavřeným*, ostatní sledy (včetně sledů nulové délky) nazýváme *otevřenými*.

Definice 1.10. *Tahem grafu* G nazýváme takový jeho sled, v němž jsou všechny hrany různé. *Cestou grafu* G nazýváme takový jeho tah, v němž každý uzel inciduje nejvýše se dvěma hranami tohoto uzlu. *Kružnicí* nazýváme uzavřenou cestu.

Definice 1.11. *Stromem* nazýváme souvislý graf neobsahující kružnice.

Definice 1.12. *Souvislým grafem* nazýváme takový graf, mezi jehož libovolnými dvěma uzly existuje sled. *Komponentou grafu* nazýváme každý jeho maximální souvislý podgraf.

Definice 1.13. Graf nazýváme *k–souvislý*, pokud při odebrání libovolných $k - 1$ hran zůstává stále souvislý.

Definice 1.14. Graf G nazveme *planárním*, pokud lze sestrojit jeho diagram v rovině tak, že žádné dvě hrany nemají kromě krajních uzlů žádné společné body.

Planární graf rozčleňuje rovinu do několika disjunktních oblastí (pokud obsahuje alespoň jednu kružnici), a ty nazýváme *stěnami* planárního grafu. *Vnější stěnou* rozumíme tu z nich, která je neomezená.

Definice 1.15. Graf, jehož všechny uzly mají stejný stupeň, se nazývá *regulární*. Pokud je tento stupeň roven k, nazveme tento graf *k–regulární*.

Kapitola 2

Analýza a návrh

Tato kapitola se zabývá existujícím řešením v nejznámějších grafových knihovnách. Dále je v této kapitole popsán návrh implementace do *Automatové knihovny*.

2.1 Existující řešení

Vzhledem k tomu, že isomorfismus grafů patří mezí podstatně složitější grafové problémy, a také díky tomu, že neexistuje žádný praktický algoritmus pro obecné grafy, zdaleka ne všechny grafové knihovny obsahují implementaci některého algoritmu. Důvodem také je, že problém isomorfismu grafů není v praxi (kromě některých specifických problémů) velmi využitelný.

Pokud už ale obsahují funkce k řešení isomorfismu, zpravidla se jedná o heuristický algoritmus *VF2* [10, 11] s časovou složitostí $\mathcal{O}(|V^2|)$ v nejlepším případě a $\mathcal{O}(|V! \cdot V|)$ v nejhorším případě, kde V je maximální počet uzlů z obou grafů.

Žádná z dále popsaných knihoven neobsahuje efektivní implementaci algoritmu, který by se specializoval pouze na jeden určitý typ grafů. Stejně také neexistuje volně dostupná implementace Hopcroft–Wong algoritmu, s největší pravděpodobností nebyl tento algoritmus ještě nikdy implementován.

2.1.1 Boost Graph Library

Boost Graph Library [2] (dále jen *BGL*) je *C++* knihovna, která je součástí velice populární sady knihoven *Boost*. *BGL* je pouze hlavičková knihovna, což znamená, že veškerá implementace se nachází v hlavičkových souborech a k použití knihovny není potřeba slinkovat program se sdílenou knihovnou. *BGL* implementuje rozhraní grafu pomocí dvou reprezentací (spojová reprezentace a matice sousednosti) a obsahuje několik základních grafových algoritmů.

Pro isomorfismus grafů *BGL* nabízí dvě funkce `bool isomorphism()` a `bool vf2_subgraph_iso()`. První zmíněná funkce je implementována pomocí jed-

noduchého algoritmu vycházejícího z definice isomorfismu, který je navíc rozšířen o backtracking, časová složitost v nejhorším případě však stále zůstává $\mathcal{O}(|V!|)$. Druhá zmíněná funkce, jak již název napovídá, je implementována pomocí *VF2* algoritmu (viz 2.1) a také řeší problém isomorfismu podgrafů.

BGL je dostupná na všech hlavních platformách, jedinou podmínkou je adekvátní *C++ kompilátor*. Je distribuována pod vlastní svobodnou open–source licencí *Boost Software License* (kompatibilní s *GNU GPL*).

2.1.2 Library of Efficient Data types and Algorithms

Library of Efficient Data types and Algorithms [3] (dále jen *LEDA*) je proprietární C++ knihovna, kterou vyvíjí *Algorithmic Solutions Software GmbH*. Je dostupná ve třech edicích: Free, Professional a Research. Free edice obsahuje pouze datové struktury, ostatní edice obsahují také implementace algoritmů. *LEDA* není pouze grafová knihovna, obsahuje i jiné datové struktury.

Pro isomorfismus grafů *LEDA* nabízí funkci `bool find_iso()` která implementuje algoritmus *VF2*.

LEDA je dostupná pro platformy MS Windows, Linux a Mac OS X a za další poplatek také ve zdrojové podobě. Pro individuální použití není kvůli vysoké ceně vhodná.

2.1.3 JGraphT

JGraphT [4] je volně dostupná open–source Java knihovna. Podpora algoritmů isomorfismu grafů je v současné době experimentální a opět se jedná o implementaci algoritmu *VF2*.

JGraphT je distribuována dle volby pod licencí *GNU LGPL* nebo *Eclipse Public License*.

2.1.4 NetworkX

NetworkX [5] je Python grafová knihovna. Obsahuje velké množství algoritmů včetně funkce `is_isomorphic` implementované pomocí algoritmu *VF2*.

NetworkX je distribuována pod open–source *BSD* licencí.

2.2 Automatová knihovna

Automatová knihovna neobsahuje v současné době žádné grafové struktury, které by bylo možné využít pro účely této práce. Z *Automatové knihovny* bude použita pouze primitivní struktura pro reprezentaci pojmenování hran a uzlů (`Label`), rozhraní pro vstup/výstup a rozhraní pro testování implementovaných algoritmů.

Dále budou použity datové struktury a základní algoritmy ze standardní C++ knihovny *Standard Template Library (STL)*. *Automatová knihovna* vy-

žaduje kompilátor s podporou standardu *C++11*, který přidává mimo jiné i implementaci hashovací tabulky (`std::unordered_map`), čehož bude v implementaci využito.

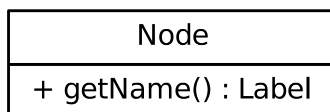
Kapitola 3

Grafový modul Automatové knihovny

Tato práce přidává do *Automatové knihovny* nový modul pro práci s grafy. Základní strukturou je třída `Graph`, od které dědí třídy `DirectedGraph` (pro orientované grafy) a `UndirectedGraph` (pro neorientovné grafy). Třídy `Node` a `DirectedEdge, UndirectedEdge` představují uzly a hrany (orientované, resp. neorientované) v grafu.

3.1 Rozhraní Node

Třída `Node` představuje uzel v orientovaném a neorientovaném grafu.



Obrázek 3.1: Diagram rozhraní Node

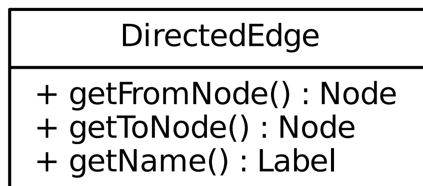
Každý uzel v grafu je unikátně identifikovatelný svým jménem (funkce `getName()`). Stejný uzel může být použit ve více grafech, samotný uzel není svázán s jediným grafem (stejně také nemá funkci, která by vracela tento graf).

3.2 Rozhraní Edge

Třída `DirectedEdge` představuje hranu v orientovaném grafu a třída `UndirectedEdge` hranu v neorientovaném grafu.

Také jedna hrana může být použita ve více grafech [1]. Ačkoliv implementace dovoluje sdílení hran a uzlů mezi více grafy, je to pouze implementační detail a dále toho nebude využito.

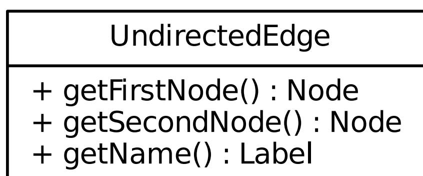
3.2.1 DirectedEdge



Obrázek 3.2: Diagram rozhraní DirectedEdge

Každá orientovaná hrana je unikátně identifikovatelná svým jménem (funkce `getName()`), zdrojovým uzlem (funkce `getFromNode()`) a cílovým uzlem (funkce `getToNode()`). Pokud graf neobsahuje rovnoběžné hrany, není nutné pojmenovat hrany.

3.2.2 UndirectedEdge



Obrázek 3.3: Diagram rozhraní UndirectedEdge

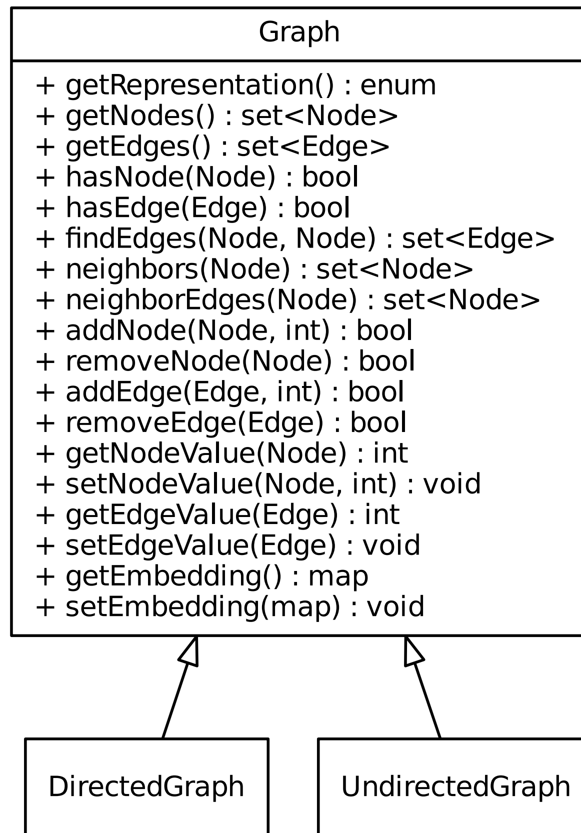
Každá neorientovaná hrana je unikátně identifikovatelná svým jménem (funkce `getName()`) a incidujícími uzly bez ohledu na pořadí (funkce `getFirstNode()` a `getSecondNode()`).

3.3 Rozhraní Graph

Třída `DirectedGraph` představuje orientovaný graf a třída `UndirectedGraph` neorientovaný graf. Obě třídy mají stejné funkce, proto zde bude uveden pouze společný diagram (a případné odlišnosti vysvětleny).

[1] I v případě, že tyto grafy neobsahují krajní uzly hrany - chybějící uzly budou přidány.

Graf je vnitřně implementován jednou ze tří reprezentací (spojová reprezentace, matice sousednosti a matice incidence). Jako výchozí reprezentace je použita spojová reprezentace, která je nejobvyklejší a ve většině případů také nejefektivnější.



Obrázek 3.4: Diagram rozhraní Graph

- `getRepresentation()`: vrací použitou vnitřní reprezentaci grafu.
- `getNodes()`: vrací množinu všech uzlů grafu.
- `getEdges()`: vrací množinu všech hran grafu.
- `hasNode(Node n)`: vrací `true` pokud graf obsahuje uzel `n`.
- `hasEdge(Edge e)`: vrací `true` pokud graf obsahuje hranu `e`.

- `findEdges(Node n1, Node n2)`: vrací množinu všech hran mezi uzly `n1` a `n2`. V případě orientovaného grafu je `n1` chápán jako zdrojový uzel a `n2` jako cílový uzel.
- `neighbors(Node n)`: vrací množinu všech uzlů sousedních s uzlem `n`.
- `neighborEdges(Node n)`: vrací množinu všech hran sousedních s uzlem `n`.
- `addNode(Node n, int v)`: přidá uzel `n` do grafu. Volitelný parametr `v` udává hodnotu uzlu.
- `removeNode(Node n)`: odstraní uzel `n` a všechny jeho sousední hrany z grafu.
- `addEdge(Edge e, int v)`: přidá ihranu `e` do grafu. Volitelný parametr `v` udává hodnotu hrany.
- `removeEdge(Edge e)`: odstraní hranu `e` z grafu. Pokud se z koncových uzlů stanou izolované uzly, jsou odstraněny.

Tato struktura umožňuje reprezentovat obyčejné grafy, grafy s rovnoběžnými hranami i grafy se smyčkami. K uzlům a hranám jsou také volitelně ukládané jejich hodnoty (funkce `setNodeValue()` a `setEdgeValue`), které mohou být použity pro ukládání vah ve váženém grafu. Pokud je potřeba pracovat s ohodnoceným váženým grafem, je nutné ukládat váhy mimo graf (např. do hashovací tabulky).

Pro planární grafy je tu funkce `setEmbedding`, pomocí které lze určit rovinné nakreslení (pořadí sousedních hran pro každý uzel).

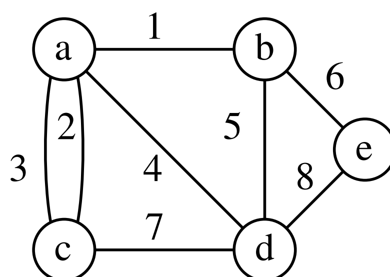
3.4 Vnitřní reprezentace grafu

Jak již bylo zmíněno, vnitřní reprezentace grafu je implementována třemi způsoby. V tabulce 3.1 je srovnána časová složitost základních operací pro jednotlivé reprezentace.

	Spojová rep.	**Matice sousednosti**	**Matice incidence**
addEdge	$\mathcal{O}(1)$	$\mathcal{O}(1)$ amort.	$\mathcal{O}(\lvert V\rvert)$
removeEdge	$\mathcal{O}(\lvert V\rvert)$	$\mathcal{O}(1)$ amort.	$\mathcal{O}(\lvert V\rvert)$
neighbors	$\mathcal{O}(1)$	$\mathcal{O}(\lvert V\rvert)$	$\mathcal{O}(\lvert V\rvert \cdot \lvert E\rvert)$

Tabulka 3.1: Srovnání časové složitosti operací

U matice sousednosti je uvedena amortizovaná složitost, kde je zanedbána operace zvětšení a zmenšení matice. Pro velmi řídké grafy je avšak složitost *addEdge* a *removeEdge* operací rovna $\mathcal{O}(|V|)$.



Obrázek 3.5: Neorientovaný graf

3.4.1 Spojová reprezentace

Spojová reprezentace je ve většině případů (algoritmy, které především procházejí graf a provádějí pouze minimální modifikace grafu) nejvhodnější reprezentace grafu. Nevýhodou může být paměťová složitost $\mathcal{O}(|V| \cdot |E|)$ pro velmi husté (až úplné) grafy. Kromě této nevýhody v paměťově omezeném prostředí se však jedná o nejlepší volbu, a proto je také zvolena jako výchozí reprezentace.

a → b d c c

b → a d e

c → a a d

d → c a b e

e → b d

Obrázek 3.6: Spojová reprezentace grafu 3.5

3.4.2 Matice sousednosti

Matice sousednosti je vhodná v případě časté modifikace grafu. Další výhodou matice sousednosti je také paměťová složitost pro velmi husté grafy, která je $\mathcal{O}(|V|^2)$. Naopak složitost průchodu grafem je nezanedbatelná a může se negativně promítnout do výsledné složitosti použitého algoritmu.

3.4.3 Matice incidence

Matice incidence je obdélníková matice, která má v každém sloupci vždy dvě nenulová čísla (1 a -1 pro orientovaný graf, 1 a 1 pro neorientovaný). Z hle-

	a	**b**	**c**	**d**	**e**
a	0	1	0	1	2
b	1	0	1	1	0
c	0	1	0	1	0
d	1	1	1	0	1
e	2	0	0	1	0

Obrázek 3.7: Matice sousednosti grafu 3.5

diska časové a paměťové složitosti nevyniká v žádné operaci, a proto je vždy lepší použít buď spojovou reprezentaci nebo matici sousednosti. Do *Automatové knihovny* byla implementována zejména proto, že se vyučuje v rámci předmětu *BI-GRA (Grafové algoritmy)*, pro praktické využití se zde ale nehodí.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
a	1	1	1	1	0	0	0	0
b	1	0	0	0	1	1	0	0
c	0	0	0	0	0	1	0	1
d	0	0	0	1	1	0	1	1
e	0	1	1	0	0	0	1	0

Obrázek 3.8: Matice incidence grafu 3.5

3.5 Grafové algoritmy

V rámci této práce byly také implementovány některé základní grafové algoritmy. Jedná se o:

- Prohledávání grafu do hloubky (*DFS*) a do šířky (*BFS*)
- Hledání nejkratší cesty (*Dijkstra*, *Bellman–Ford* a *Floyd–Warshall*)
- Hledání minimální kostry grafu (*Jarník–Prim*)
- Topologické uspořádání
- Isomorfismus obecných grafů (dle definice)

3.5.1 Generování planárních grafů

K účelům testování je potřeba generovat náhodné planární grafy. Implementovaný generátor je velmi jednoduchý, využívá množinu předem zvolených planárních grafů které vzájemně propojuje tak, aby se zachovala planarita.

3.5.2 Rovinné nakreslení planárních grafů

Pro rovinné nakreslení planárních grafů byl zvolen *Hopcroft–Tarjan* [7] algoritmus. Jedná se o lineární algoritmus, který na vstup přijímá 2–souvislý obyčejný planární graf. Algoritmus také dokáže zjistit, zda daný graf je nebo není planární.

Byla použita C++ implementace z knihovny *LEDA* [8], upravená pro struktury v *Automatové knihovně.*

Kapitola 4

Hopcroft–Wong algoritmus

Tato kapitola se zabývá popisem a implementací *Hopcroft–Wong* [6] algoritmu pro testování isomorfismu planárních grafů v čase $\mathcal{O}(|V|)$ lineárním s počtem uzlů v grafu.

Tento algoritmus byl publikován v roce 1972. V té době se jednalo o první publikovaný lineární algoritmus pro řešení isomorfismu planárních grafů. Ačkoliv je tento algoritmus lineární, není autory považován za efektivní z důvodu vysoké konstanty.

Algoritmus rozhoduje o isomorfismu 3–souvislých planárních grafů. Vzhledem k použité funkci pro rovinné nakreslení grafu (viz 3.5.2), která přijímá 2–souvislé obyčejné planární grafy, se v rámci této práce omezíme pouze na obyčejné 3–souvislé planární grafy. Algoritmus (ani jeho implementace v rámci práce) ale takové omezení nemá.

Poznámka: Z důvodu obtížného překladu do češtiny budou některé pojmy dále použity v anglickém jazyce.

4.1 Reprezentace grafu

Algoritmus pracuje s vlastní reprezentací grafu, do které musí být graf nejdříve převeden.

Graf je reprezentován jako pole uzlů, hran a stěn. Každá hrana je rozdělena podle krajního uzlu na dvě části (konce hran či edge-ends).

4.1.1 Struktura Vertex

Struktura `Vertex` představuje uzel v grafu a obsahuje členské proměnné:

- `int vlabel`: číselné označení uzlu
- `int degree`: stupeň uzlu

- `EdgeEnd* vedge`: edge-end hrany vycházející z tohoto uzlu

4.1.2 Struktura EdgeEnd

Struktura `EdgeEnd` představuje jeden konec hrany v grafu a obsahuje členské proměnné:

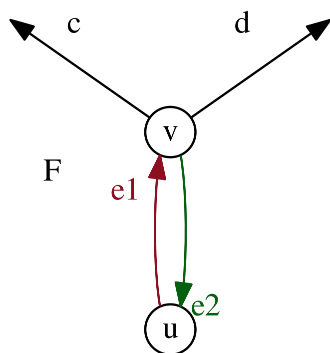
- `int elabel`: číselné označení konce hrany
- `Vertex* vertex`: uzel incidující s tímto koncem hrany
- `Face* ccwface`: stěna proti směru hodinových ručiček
- `EdgeEnd* ccwedge`: hrana proti směru hodinových ručiček
- `EdgeEnd* cwedge`: hrana ve směru hodinových ručiček
- `EdgeEnd* otheredge`: opačný konec hrany

4.1.3 Struktura Face

Struktura `Face` představuje stěnu grafu a obsahuje členské proměnné:

- `int edges`: počet hran stěny
- `EdgeEnd* fedge`: edge-end patřící této stěně

Z tohoto popisu nemusí být některé detaily jednoznačné, proto je zde uveden obrázek 4.1, který názorně ilustruje vztahy mezi uzly, konci hran a stěnami.



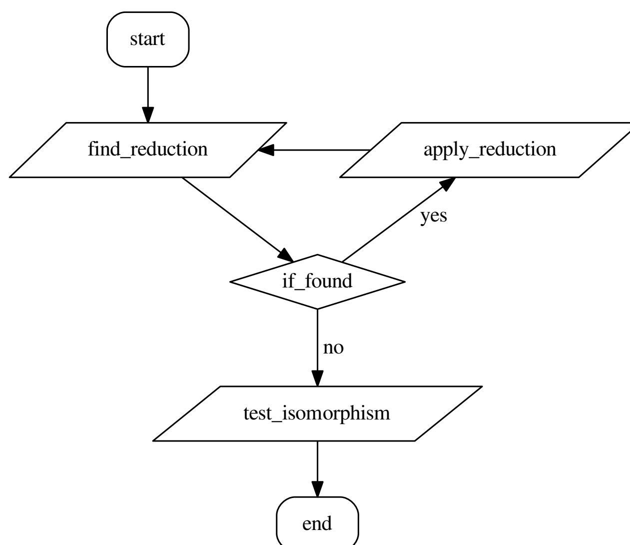

```
e1.vertex = v
e2.vertex = u
e1.cwedge = d
e1.ccwedge = c
e1.otheredge = e2
e1.ccwface = F

v.vedge = c (d, e2)
F.fedge = e1 (c)
d.otheredge.ccwedge = e2
c.otheredge.cwedge = e2
```

Obrázek 4.1: Ilustrační příklad interní reprezentace grafu

Na obrázku je hrana e zobrazena jako její dva konce $e1$ a $e2$, jedná se však pouze o jedinou hranu. Konce hran jsou znázorněny jako orientované hrany, podle toho k jakému uzlu směřují. `vedge` a `fedge` u uzlů a stěn může být libovolná hrana náležící ke stěně, příp. uzlu, na obrázku je toto znázorněno ostatními možnostmi v závorce.

4.2 Popis algoritmu

Algoritmus nejdříve přiřadí číselné označení všem uzlům a hranám v grafu. Uzly jsou označeny číslem 1 a hrany číslem 2. Poté algoritmus opakovaně aplikuje dále specifikované redukce na oba grafy, dokud je to možné. Nakonec jsou výsledné grafy testovány na isomorfismus podle definice. [2]



Obrázek 4.2: Celkový koncept Hopcorft–Wong algoritmu

4.3 Pomocné algoritmy

Algoritmus používá dva pomocné algoritmy k zakódování informace do označení uzlů a hran.

[2] Testování všech možných kombinací mapování uzlů z prvního na druhý graf. V této fázi se již jedná o grafy se shora omezeným počtem uzlů, proto můžeme tento krok provést v konstantním čase.

4.3.1 Rozdělení vektorů podle skupin ekvivalence

Tento algoritmus rozdělí množinu *n–tic* různé délky tak, že $(a_1, \dots, a_m)$ a $(b_1, \dots, b_n)$ jsou umístěny ve stejné třídě ekvivalence, právě když

$$m = n$$

$$a_i = b_i,\ 1 \leq i \leq n$$

Algoritmus 4.1 Vector partitioning

1: **while** an n–tuple with an unpigeonholed component exists **do**
2: **for** each class with an unpigeonholed component **do**
3: **for** each n–tuple in this class **do**
4: pigeonhole the n–tuple on the first unpigeonholed component if such a component exists, otherwise place into a special pigeonhole, putting location of each new pigeonhole used on a stack;
5: **end for**
6: **while** stack of occupied pigeonholes $\neq \emptyset$ **do**
7: empty the n–tuples placed into the pigeonhole referenced by the top entry on the stack into a new equivavelnce class;
8: pop top entry from stack
9: **end while**
10: **end for**
11: **end while**

Algoritmus 4.1 je použit k „zakódování“ n–tic do jediného čísla, použitého při změně označení uzlů a hran v grafu. Každé nově vytvořené třídě ekvivalence se přiřadí číslo pomocí funkce `newInteger()`, která každým voláním inkrementuje svou návratovou hodnotu.

Pokud je $(a_1, \dots, a_m)$ umístěna v třídě ekvivalence „pojmenované“ n rozdělením P, pak značíme

$$ENCODE_P(a_1, \dots, a_m) = n$$

$$DECODE(n) = (a_1, \dots, a_m)$$

4.3.2 Isomorfismus kruhu

Uvažujme druhou relaci ekvivalence mezi n–ticemi různé délky takovou, že $(a_0, \dots, a_{m-1})$ je ekvivalentní s $(b_0, \dots, b_{n-1})$ právě když pro nějaké k platí

$$m = n$$

$$a_i = b_{(i+k) \bmod n},\ 0 \leq i \leq n-1$$

n–tici pod touto ekvivalencí nazveme *n–kruh.* V n–kruhu $(a_0, \dots, a_{n-1})$, je $a_{(i+1) \bmod n})$ sousední *proti hodinovým ručičkám* k $a_i,\ 0 \leq i \leq n-1$. a_i

Algoritmus 4.2 Circle isomorphism

1: **while** there exist two distinct integers in some circle **do**
2: **for** each occurence of an ordered pair a, b where a is clockwise adjacent to b in some circle and $a \neq b$ **do**
3: construct a 2–tuple (a, b) associated with this occurence;
4: **end for**
5: call this partitioning P;
6: **for** all classes in order of their names **do**
7: replace the occurences of the two components (if they both still exist), associated with (a, b) by a single component whose value is $ENCODE_P(a, b)$; ▷ this results in shrinking the circle containing the occurences by one for each such replacement;
8: **end for**
9: **end while**
10: **if** both circles' dimensions are not the same or the value common to all components of one circle is not the same as that common to all components of the other **then**
11: stop with no isomorphism
12: **else**
13: stop with isomorphism
14: **end if**

je sousední *po směru hodinových ručiček* k a_j právě když a_j je sousední proti směru hodinových ručiček k a_i.

Algoritmus 4.2 rozhodne o isomorfismu dvou n–kruhů v čase $\mathcal{O}(n)$. Jednoduchou modifikací získáme algoritmus, který rozdělí množinu n–kruhů do tříd „isomorfních" kruhů. Stejně jako v případě n–tic, tento algoritmus použijeme k „zakódování" n–kruhů do jednoho čísla.

4.4 Redukce

Redukcí se rozumí taková modifikace grafu, při které dojde ke snížení počtu uzlů nebo hran. Algoritmus postupně aplikuje všechny možné redukce, podle jejich priority. Po aplikaci každé redukce může dojít k nesrovnalostem mezi oběma grafy, např. pokud tato redukce ovlivnila jiný počet uzlů v prvním a ve druhém grafu, je jasné, že tyto dva grafy nejsou isomorfní. V rámci všech redukcí také dochází ke změně označení uzlů a hran. Redukce budou dále popsány v pořadí podle jejich priority.

Důležitou vlastností redukcí je, aby jejich výsledek byl stejný, jako kdyby se prováděli naprosto stejně na všech podobných podgrafech. Změna označení proto také musí být provedena současně pro oba grafy najednou.

Když už není možné aplikovat žádnou redukci, zbývající graf bude buď jeden z pěti regulárních polyhedrálních grafů nebo graf o jednom uzlu. Tyto

grafy jsou následně testovány na isomorfismus podle definice v konstantním čase.

Algoritmus 4.3 popisuje obecnou šablonu pro funkci implementující redukci. Funkce `findReduction` ukládá označení z obou grafů do jediného pole L, tím je zaručena následná správná změna označení ve funkci `applyReduction`, bez ohledu na pořadí uzlů a hran v obou grafech.

Algoritmus 4.3 Šablona pro redukce

1: array of labels L;
2: findReduction(G1);
3: findReduction(G2); ▷ findReduction finds all elements to apply reduction on in a graph and stores the labels in L
4: **if** number of elements to apply reduction on in G1 $\neq$ G2 **then**
5: end with indication of not isomorphic graphs
6: **end if**
7: partition(L); ▷ vector or circle partitioning
8: applyReduction(G1);
9: applyReduction(G2);

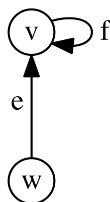
4.4.1 Odstranění smyček a uzlů stupně jedna

Definice 4.1. *Loop (smyčka)* je taková hrana, kde oba konce hrany incidují se stejným uzlem v a oba konce hrany jsou navzájem sousední. Uzel v nazýváme *loop vertex*.

Pokud v má hrany, které nejsou smyčkou a počet smyček je větší než jedna, graf sestávající z uzlu v a hran incidujících s v nazýváme *corolla*. Pokud v má hrany, které nejsou smyčkou a pouze jednu smyčku, graf je nazýván *knot*.

4.4.1.1 Odstranění smyček

První redukce spočívá v odstranění smyček z uzlů mající i hrany, které nejsou smyčkami.



Obrázek 4.3: Smyčka

Uvažujme trojici e, f, v (viz 4.3), kde f je smyčka, v je uzel smyčky a e je sousední hrana proti směru hodinových ručiček k hraně f. Pro každou takovou trojici je do pole L přidána trojice: [3]

$$\left(label(w \to v),\ label(f \to f)_1,\ label(f \to f)_2\right)$$

První $label(f \to f)$ označuje konec hrany, který je sousední ve směru hodinových ručiček k hraně e, druhý pak k druhému konci.

Následně je pole L rozděleno pomocí circle partitioning algoritmu, $label(e \to v)$ je změněn na hodnotu třídy ekvivalence výše uvedené trojice a smyčka f je z grafu odstraněna.

4.4.1.2 Odstranění uzlů corolla

Pro každý uzel corolla v sestrojíme n–tici, tak že do této n–tice přidáme pro každou hranu uzlu v trojici: [4]

$$\left(label(v),\ label(v \to v)_1,\ label(v \to v)_2\right)$$

Pro každý uzel je tedy do pole L přidána n–tice, kde n je rovno trojnásobku počtu hran uzlu.

Následně je pole L rozděleno pomocí circle partitioning algoritmu, $label(v)$ je změněn na hodnotu třídy ekvivalence n–tice patřící tomuto uzlu a všechny hrany jsou odstraněny.

4.4.1.3 Odstranění uzlů knot

Pro každý uzel knot v je do pole L přidána šestice:

$$\left(label(v),\ label(v \to v)_1,\ label(v \to v)_2,\ label(v),\ label(v \to v)_2,\ label(v \to v)_1\right)$$

Následně je redukce analogická s odstraněním uzlů corolla (viz výše).

4.4.1.4 Odstranění uzlů stupně jedna

Definice 4.2. *Spoke* je uzel stupně jedna a jeho hrana. *Spoke center* je uzel incidující se spoke. Pokud má spoke center pouze incidentní hrany, které jsou spoke a počet těchto hran je větší než jedna, jedná se o *star (hvězdu)*. Pokud má takovouto hranu pouze jednu, nazveme tento uzel *dumbell*.

Na spoke center s hranami, které nejsou spoke je aplikována redukce analogická s 4.4.1, na hvězdy je aplikována redukce analogická s 4.4.1.1 a na dumbell je aplikována redukce analogická s 4.4.1.2.

[3] $label(e \to v)$ se rozumí jako označení konce hrany mezi uzly e a v, který míří k uzlu v.
[4] $label(v)$ se rozumí jako označení uzlu v.

4.4.2 Nahrazení shluků

Definice 4.3. *Clump (shluk)* je maximální množina hran $e_1, \dots, e_k,\ k > 1$, spojující dva různé uzly v a w taková, že alespoň jeden z uzlů v a w jsou sousední s uzlem jiným než v a w, a

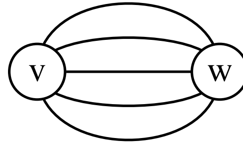
$$e_i CCW_v e_{i+1}\ e_i CW_w e_{i+1},\ 1 \leq i < k$$

$e_1 CW_v e_2$ značí, že hrana e_1 je sousední po směru hodinových ručiček k hraně e_2 u uzlu v. V případě CCW se jedná o sousednost proti směru hodinových ručiček.

Clump vektor uzlu v je vektor $\big(label(e_1 \to v), \dots, label(e_k \to v)\big)$ a clump vektor uzlu w je vektor $\big(label(e_k \to w), \dots, label(e_1 \to w)\big)$.

Pro každý shluk jsou do pole L přidány oba clump vektory. Následně je pole L rozděleno pomocí vector partitioning algoritmu. Všechny clump hrany jsou poté nahrazeny novou hranou a její konce jsou označeny hodnotami tříd ekvivalence vzniklé z clump vektorů.

Definice 4.4. *Skein* je graf skládající se ze dvou uzlů u a w a k hran $(k > 1)$, každá hrana incidující s uzly u a w. Uzly u a w názýváme *skein uzly* a hrany *skein hrany.*



Obrázek 4.4: Skein

Pro každý skein se skein uzly v a w a skein hranou f jsou do pole L přidány dva vektory:

$$\big(label(v),\ label(w \to v),\ label(v \to w),\ label(w)\big)$$

$$\big(label(w),\ label(v \to w),\ label(w \to v),\ label(v)\big)$$

Následně je pole L rozděleno pomocí circle partitioning algoritmu a clump je nahrazen jediným uzlem označeným menší hodnotou třídy ekvivalence z těchto dvou vektorů.

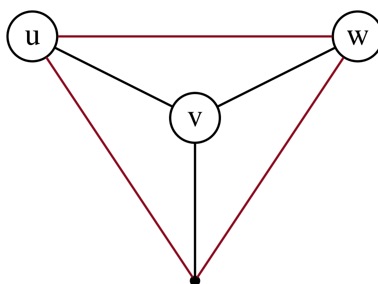
4.4.3 Čtyři obecné a dva speciální případy

Definice 4.5. Uzel v stupně $d = 2, 3, 4, 5$ budeme nazývat *uzel nízkého stupně.*

4.4.3.1 Odstranění izolovaných uzlů nízkého stupně

Definice 4.6. Uzel v stupně d, který není sousední s žádným uzlem stejného stupně d nazveme *izolovaným uzlem.*

Pro každý izolovaný uzel v stupně d provedeme následující. Pro každou dvojici sousedních hran (v, u) a (v, w), pro které platí $(v, u)CCW_v(v, w)$, přidáme novou hranu (u, w) takovou, že platí $(u, w)CCW_u(u, v)$ a $(v, w)CCW_w(u, w)$. Hrany (u, w), (u, v) a (v, w) tedy vytvoří trojúhelníkovou stěnu v grafu.



Obrázek 4.5: Nově přidané hrany (červeně)

Pro každou takovou nově vytvořenou hranu jsou do pole L přidány dva vektory:

$$\big(label(v),\ label(u \to v),\ label(v \to u)\big)$$

$$\big(label(v \to w),\ label(w \to v),\ label(v)\big)$$

Následně je pole L rozděleno pomocí vector partitioning algoritmu, konce nově vytvořených hran jsou označeny hodnotou třídy ekvivalence jejich vektorů a uzel v s incidujícími hranami je odstraněn.

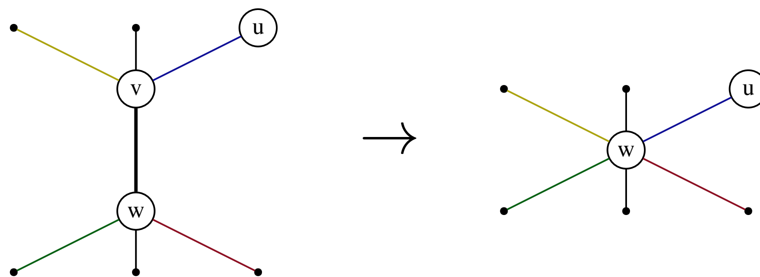
Tato redukce je provedena pro $d = 2, 3, 4, 5$ v tomto pořadí.

4.4.3.2 Neregulární grafy bez izolovaných uzlů nízkého stupně

Tato redukce se týká grafů s uzlem stupně $d = 2, 3, 4, 5$, který má nejméně jeden sousední uzel se stupněm d a nejméně jeden sousední uzel se stupněm x, $x \neq d$. Je tu také jedna výjimka, v případě kdy $d = 4$ a sousední uzly jsou stupně $4, x, 4, y$ (redukce pro tento případ je popsána dále).

Definice 4.7. Uvažujme uzel v stupně d s nejméně jedním sousedním uzlem stupně d a nejméně jedním sousedním uzlem stupně $x \neq d$ (a v nespadá do výjimky popsané výše). Uvažujme d–kruh C, jehož komponenty jsou stupně

sousedních uzlů proti směru hodinových ručiček uzlu v tak, že každá komponenta nerovnající se d je změněna na X [5]. C může být pouze jedním z konečného počtu d–kruhů. Pro uzel v nazveme incidentní hranu, která také inciduje s uzlem stupně jiným než d, *význačnou* hranou, pokud isomorfismus G_1 na G_2 mapuje v na w a význačnou hranu u v na význačnou hranu u w.



Obrázek 4.6: Operace odstranění význačné hrany

Redukce pro uzel stupně d je následující: Nechť v je uzel stupně d, (v, w) je význačná hrana a (u, v) je sousední hrana proti směru hodinových ručiček k hraně (v, w) u uzlu v. Pro každý uzel v je do pole L přidán vektor:

$$(label(v \to w),\ label(w \to v),\ label(v),\ label(u \to v))$$

Následně je pole L rozděleno pomocí vector partitioning algoritmu a $label(u \to v)$ je změněn na hodnotu třídy ekvivalence odpovídajícího vektoru. Nechť (w, w_1) je sousední hrana po směru hodinových ručiček k hraně (v, w) u uzlu w. Hrana (v, w) a uzel v jsou odstraněny z grafu. Hrany dříve incidentní s uzlem v, v pořadím začínající s (u, v) jsou vloženy jako sousední hrany k uzlu w mezi hrany (w, w_1) a (w, w_k).

Tato redukce je provedena pro uzly stupně $d = 2, 3, 4, 5$ v tomto pořadí.

4.4.3.3 První výjimka

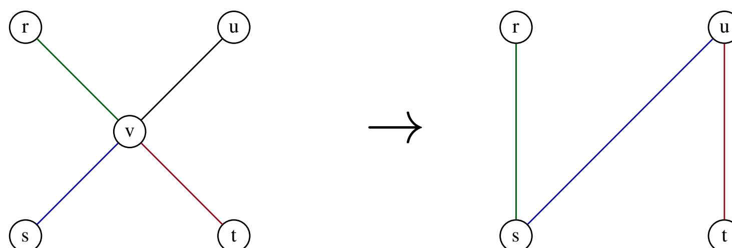
Speciální případ pro uzly stupně 4, které mají sousední uzly stupně $(4, x, 4, y)$.

Pro každý takový uzel v, označme sousední uzly odpovídající $(4, x, 4, y)$ jako (r, s, t, u). Hrany (r, v) a (t, v) se u uzlu v rozpojí a přepojí k uzlům s a u. Uzel v se následně z grafu odstraní a hrany (s, v) a (v, u) se spojí do jediné hrany (s, u). Do pole L jsou následně přidány dva vektory:

$$(label(v),\ label(s \to v),\ label(r \to v))$$

$$(label(v),\ label(u \to v),\ label(t \to v))$$

[5] X uvažujeme jako speciální hodnotu



Obrázek 4.7: Redukce výjimky uzlu stupně 4

Následně je pole L rozděleno pomocí vector partitioning algoritmu a $label(r \to s)$ a $label(t \to u)$ jsou změněny na hodnoty třídy ekvivalence jejich vektorů.

4.4.3.4 Regulární grafy s izolovanými stěnami

Definice 4.8. Uvažujme označení stěn v grafu podle toho, kolik hran tato stěna má. Stěnu F nazveme *izolovanou* vzhledem k uzlu v, pokud je to jediná stěna se svým označením u uzlu nízkého stupně v. Uvažujme hrany e a f sousední u uzlu v, pro které platí $eCCW_v f$. Hranu e nazveme *význačnou* hranu stěny F u uzlu v. Tyto význačné hrany mají unikátní izolované stěny F a jejich uzly v.

Tato redukce je provedena pro izolované stěny s označením $d = 3, 4, 5$. (2–regulární grafy jsou přímo testovány na isomorfismus pomocí circle isomorphism algoritmu). Pro každý uzel v s význačnou hranou a izolovanou stěnou označenou d, redukce odstraní tuto význačnou hranu a uzel v stejně jako v 4.4.3.2. Je tu ovšem jedna komplikace, a to, že u jednoho uzlu může být více význačných hran.

Všechny tyto význačné hrany tvoří v grafu buď stromy nebo kružnice. V případě stromů jsou postupně odstraňovány listy a změna označení je identická s 4.4.3.2. V případě kružnice jsou hrany postupně odstraněny, tedy stěna je zredukována na jediný uzel, a tomuto uzlu je změněno označení na jednotnou hodnotu vrácenou funkcí `newInteger()` (tzn. v případě více kružnic v rámci redukce se všechny zbývající uzly označí stejnou hodnotou).

4.4.3.5 Poslední obecná redukce

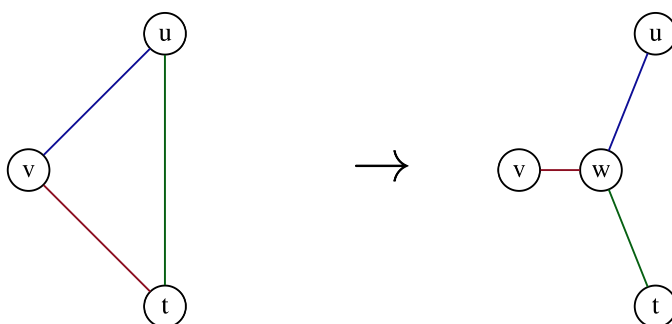
V této chvíli zbývá pouze redukce pro grafy s uzlem v, který má incidentní stěny označeny (podle definice výše) přesně dvěma hodnotami a každá tato hodnota se objevuje nejméně dvakrát. Z toho plyne, že se můžeme omezit pouze na regulární grafy stupně čtyři a pět. Tyto grafy musí mít podle Eulerovy formule trojúhelníkové stěny.

Definice 4.9. Označme stěny s více než třemi hranami x a trojúhelníkové stěny 3. Uvažujme uzly s trojúhelníkovými stěnami. Kromě speciálního případu s kruhem $(3, x, 3, x)$, kruh tvořený označením stěn okolo těcho uzlů může být jedním z pěti různých isomorfních tříd (čtyři stupně pět a jeden stupně čtyři). V každé této třídě je buď unikátní dvojice sousedních hodnot 3 nebo hodnot x (nebo obojí). Hranu, která odděluje dvě stěny z této dvojice, nazýváme *význačnou*.

Redukce spočívá v odstranění význačných hran stejně jako v 4.4.3.2, ale může nastat situace, kdy je hrana význačnou pro oba uzly této hrany. V tomto případě se tato hrana odstraní stejně jako v 4.4.3.2 a zbývajícímu uzlu se jednotně změní označení pomocí funkce `newInteger()` (podobně jako v 4.4.3.4).

4.4.3.6 Druhá výjimka

Speciální případ pro uzly s označením stěn $(3, x, 3, x)$.



Obrázek 4.8: Redukce druhé výjimky uzlu stupně 4

Redukce je následující: Do každé trojúhelníkové stěny přidáme nový uzel a přepojíme hrany ohraničující tuto stěnu k nově přidanému uzlu. V této redukci, jako jediné, nedochází k žádné změně označení.

4.5 Shrnutí

Výše jsou popsány všechny redukce. Po provedení všech možných redukci algoritmus otestuje zbylé grafy na isomorfismus podle definice. Jak již bylo zmíněno, všechny možné zbylé grafy jsou shora omezené počtem uzlů, a proto můžeme časovou složitost této fáze algoritmu označit jako konstantní.

Úpravou algoritmu lze také získat pro isomorfní grafy i mapování uzlů z prvního grafu na první. Toto ovšem nebylo cílem této práce a tedy byl implementován pouze algoritmus v původní publikované formě.

Kapitola **5**

Testování

V této kapitole jsou popsány metody a výsledky testování implementovaných algoritmů.

5.1 Automatické testy

V průběhu implementace Hopcroft–Wong algoritmu bylo přidáno množství automatických testů. Tyto testy byly použity k ověření správnosti implementace reprezentace grafu, základních operací s grafem (např. odstranění hrany) a jednotlivých redukcí.

Dále také byl k testování použit generátor náhodných grafů (viz 3.5.1). V rámci testování bylo nalezeno množství chyb, které byly následně opraveny.

5.2 Srovnání výkonu

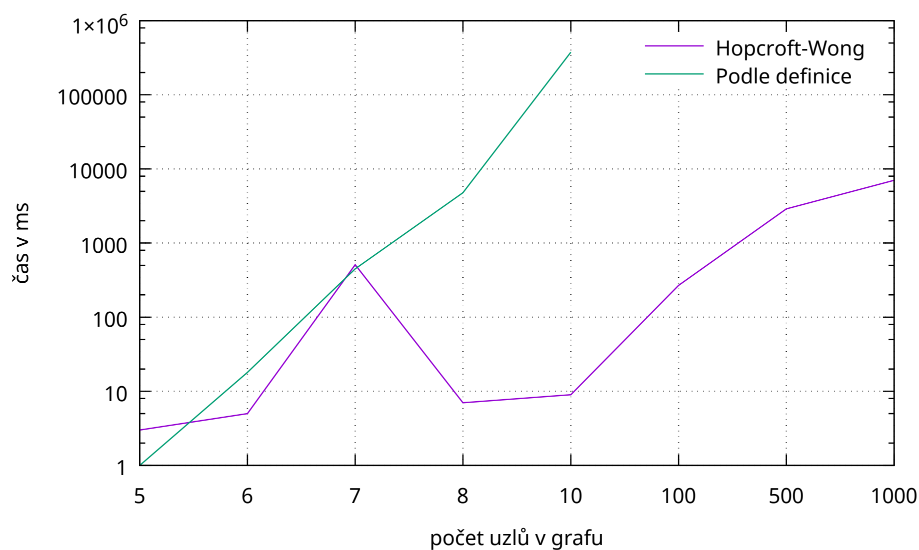
Algoritmus Hopcroft–Wong byl výkonově porovnán s naivním algoritmem dle definice isomorfismu. Vzhledem k tomu, že algoritmus dle definice má časovou složitost $\mathcal{O}(|V|!)$, nejsou výsledky nijak překvapující (pro grafy s více než osmi uzly je tento algoritmus již prakticky nepoužitelný).

Algoritmy byly testovány na náhodných 3–souvislých planárních grafech pomocí programu `agraphbench2`. Výsledky jsou průměrem z pěti spuštění testu. Do času Hopcroft–Wong algoritmu není započítána doba běhu Hopcroft–Tarjan algoritmu (rovinné nakreslení grafu), pouze doba běhu samotného algoritmu isomorfismu.

U výsledků testování (viz 5.1) je zajímavý případ u grafu se sedmi uzly. Čas Hopcroft–Wong algoritmu je o mnoho větší než pro podstatně větší grafy. Důvodem je, že u takto malých grafů se do doby vykonávání nejvíce promítne testování zbylých grafů (po aplikování redukcí) algoritmem podle definice a v tomto případě zbyl značně větší graf, než v ostatních případech. Nejedná se o chybu měření.

	Hopcroft–Wong			Podle definice		
	min	*avg*	*max*	*min*	*avg*	*max*
5 uzlů	3 ms	3 ms	3 ms	0,7 ms	0,7 ms	0,8 ms
6 uzlů	4 ms	5 ms	7 ms	15 ms	18 ms	20 ms
7 uzlů	504 ms	510 ms	516 ms	443 ms	448 ms	450 ms
8 uzlů	7 ms	7 ms	8 ms	4,700 s	4,783 s	4,873 s
10 uzlů	9 ms	9 ms	10 ms	364,670 s	375,385 s	426,265 s
100 uzlů	267 ms	269 ms	272 ms	–	–	–
500 uzlů	2,824 s	2,888 s	2,902 s	–	–	–
1000 uzlů	6,903 s	7,015 s	7,218 s	–	–	–

Tabulka 5.1: Výsledky testování výkonu algoritmů



Obrázek 5.1: Srovnání výkonu algoritmů

Pro algoritmus podle definice také nejsou uvedeny hodnoty pro grafy větší jak s deseti uzly. Důvodem je velmi dlouhá doba běhu, pro ilustrační účely stačí výsledky na malých grafech.

Závěr

Cílem této práce byla implementace Hopcroft–Wong algoritmu a přidání grafového modulu do *Automatové knihovny.*

V práci se relativně úspěšně podařilo tento algoritmus implementovat, pro velké množství grafů funguje zcela správně. Při testování bylo ovšem nalezeno také několik grafů, pro které implementace vrací špatné výsledky. Vzhledem k některým nejasnostem v popisu algoritmu se tyto nedostatky bohužel nepodařilo odstranit.

Tento algoritmus není s největší pravděpodobností v současné době nikde používán a také neexistuje žádná jeho veřejně dostupná (a nejspíše ani neveřejná) implementace. Podle samotných autorů tohoto algoritmu se také nejedná o praktický algoritmus, díky jeho komplikovanosti a ačkoliv lineární složitosti, velké časové konstantě. Z těchto důvodů výsledek této práce předčil počáteční očekávání.

V implementaci jsou také některé operace provedeny s neoptimální složitostí. Možné zlepšení, kromě samotné opravy funkčnosti, může být také optimální implementace těchto operací. V rámci *Automatové knihovny* se do budoucna mohou implementovat i další grafové algoritmy. Další užitečnou funkcí je vizualizace grafů v grafické podobě (např. pomocí programu *Graphviz*), která nebyla v rámci této práce implementována.

Příloha **A**

Seznam použitých zkratek

BGL Boost Graph Library

LEDA Library of Efficient Data types and Algorithms

STL Standard Template Library

DFS Depth–First Search

BFS Breadth–First Search

BI-GRA Grafové algoritmy

Literatura

[1] TRÁVNÍČEK, Jan. *Ing. Jan Trávníček / Automata library / GitLab* [online]. [cit. 2015-12-19] Dostupné z: https://gitlab.fit.cvut.cz/travnja3/automata-library

[2] SIEK, Jeremy, Lie-Quan LEE a Andrew LUMSDAINE. *The Boost Graph Library - 1.57.0* [online]. [cit. 2015-12-19]. Dostupné z: http://www.boost.org/doc/libs/1_57_0/libs/graph/doc/index.html

[3] Algorithmic Solutions Software GmbH. *Algorithmic Solutions Software GmbH* [online]. [cit. 2015-12-19]. Dostupné z: http://www.algorithmic-solutions.com/leda/index.htm

[4] NAVEH, Barak and Contributors. *JGraphT - a free Java Graph Library* [online]. [cit. 2015-12-19]. Dostupné z: http://jgrapht.org/

[5] NetworkX developer team. *NetworkX: Python software for complex networks* [online]. [cit. 2015-12-19]. Dostupné z: https://networkx.github.io

[6] HOPCROFT, J.E, J.K. WONG. Linear time algorithm for isomorphism of planar graphs (Preliminary Report). In *STOC '74 Proceedings of the sixth annual ACM symposium on Theory of computing.* New York: ACM, 1974, s. 172-184.

[7] HOPCROFT, J.E, R.E. TARJAN. Isomorphism of planar graphs (working paper). In *Complexity of Computer Computations.* New York: Plenum Press, 1972, s. 143-150.

[8] MEHLHORN, Kurt, Petra MUTZEL, Stefan NÄHER An implementation of the Hopcroft and Tarjan planarity test and embedding algorithm. Research Report MPI-I-93-151, Max-Planck-Institut für Informatik, Im Stadtwald, D-6612. Saarbrücken, Germany, 1993.

[9] KOLÁŘ, Josef. *Teoretická informatika.* Vyd. 1. V Praze: České Vysoké Učení Technické, 2009, 206 s. ISBN 978-80-01-04331-8.

[10] CORDELLA, L.P, P. FOGGIA, C. SANSONE, M. VENTO. An improved algorithm for matching large graphs. In: *3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition.* Cuen, 2001. s. 149-159.

[11] CORDELLA, L.P, P. FOGGIA, C. SANSONE, M. VENTO. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence.* 2004. s. 1367-1372.

Příloha B

Obsah přiloženého CD

```
readme.txt .................................. stručný popis obsahu CD
src
  automata-library ................... projekt Automatová knihovna
  thesis ...................... zdrojová forma práce ve formátu LaTeX
text
  BP_Rosca_David_2016.pdf ............. text práce ve formátu PDF
```