Sem vložte zadanie Vašej práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra Softwarového inženýrství

Bakalárska práca

Rozšiřující zásuvný modul pro GraphBrowser

Matúš Tóth

Vedúci práce: Ing. Ivan Šimeček, Ph.D.

11. mája 2015

Poďakovanie

Chcel by som poďakovať Ing. Ivanovi Šimečkovi, Ph.D. za odbornú pomoc a rady k tejto práci. Ďalej by som chcel poďakovať rodine a priateľom za prejavenú podporu.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval(a) samostatne a že som uviedol(uviedla) všetky informačné zdroje v súlade s Metodickým pokynom o etickej príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov, a skutočnosť, že České vysoké učení technické v Praze má právo na uzavrenie licenčnej zmluvy o použití tejto práce ako školského diela podľa § 60 odst. 1 autorského zákona.

V Prahe 11. mája 2015 .....

Odkaz na túto prácu

Tóth, Matúš. *Rozšiřující zásuvný modul pro GraphBrowser*. Bakalárska práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Táto práca popisuje analýzu grafového editoru napísaného v programovacom jazyku Java s využitím architektúry MVC, ktorý umožňuje vytvárať grafy a overovať ich vlastnosti. Popisuje aj vývoj rozšírení daného grafového editoru o ďalšie vlastnosti a algoritmy ako či graf obsahuje uzavretý Eulerovský ťah, či je graf Hamiltonovský, alebo nájdenie minimálnej kostry grafu.

Kľúčové slová Grafový editor, vlastnosti grafov, farebnosť, planarita, Hamiltonovský graf, minimálna kostra, Eulerovský ťah, Java, rozšírenie

Abstract

This thesis describes the analysis of graph editor written in Java programming language using the MVC architecture, which allows you to create graphs and verify their properties It also describes the development of the extensions of the graph editor with further properties and algorithms eg whether a graph contains a closed Eulerian trail, whether the graph is Hamiltonian, or finding the minimal spanning tree.

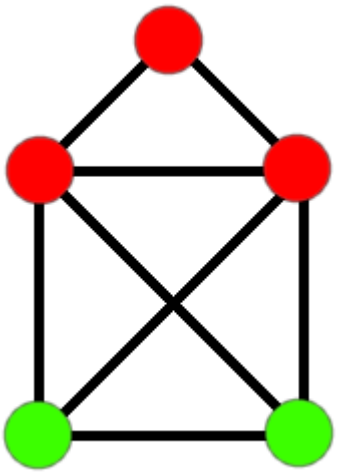
Keywords Graph editor, properties of graph, grahp coloring, planarity, Hamiltonian graphs, minimum spanning tree, Eulerian trail, Java, extension

Obsah

Zoznam obrázkov

Úvod

Dá sa domček nakresliť jedným ťahom? Mnohým sa to určite neraz podarilo. Ale prečo to tak je? Aj týmto sa zaoberá teória grafov. Teória grafov, ako je poznať už z názvu, je veda zaoberajúca sa grafmi. Grafy v tomto ponímaní budeme brať ako uzly a hrany medzi nimi. Načrtnutá vlastnosť je úzko spätá s Eulerovými grafmi a hľadaním ťahov. Skúste začať svoj ťah v jednom z uzlov, z ktorého vedie nepárny počet hrán. Vyšlo to?



Obr. 0.1: Domček

Zaručene áno. Ľudia sa v minulosti veľa zaujímali o teóriu grafov a tým poskytli živnú pôdu pre mnoho moderných disciplín, či už počítačových, alebo matematických. Za zakladateľa tejto teórie sa považuje Leonard Euler, ktorý v roku 1736 prezentoval svoj problém siedmych mostov mesta Královce [1]. Tento problém bol nadmieru podobný tomu, ktorý som načrtol prvou vetou: kresbe domčeka jedným ťahom. Narozdiel od nášho problému, ten jeho na-

ozaj riešenie nemal. V súčasnej dobe ale určite neriešime problém siedmych mostov. Čo taká dopravná sieť v New Yorku? Na to by sme si len s ľudskými silami určite nevystačili. A tu prichádzajú na rad počítače a algoritmy a moja bakalárska práca.

Vo svojej bakalárskej práci sa zaoberám už existujúcim grafovým editorom, GraphBrowserom [2], umožňujúcim vytvárať a ukladať grafy. Nezaoberá sa však takým širokým spektrom algoritmov a nezisťuje toľko vlastností ako by som si predstavoval od dokonalého grafového editoru/analyzátora. Preto som sa rozhodol rozšíriť tento editor o niektoré zaujímavé a názorné vlastnosti.

Moja práca je delená do štyroch kapitol:

- Kapitola **Teoretický základ** približuje základné pojmy teórie grafov, teórie zložitosti a samotné algoritmy potrebné na rozšírenie daného editora.
- Kapitola **Analýza** zahŕňa analýzu požiadaviek na túto prácu, popisuje grafový editor GraphBrowser, jeho možnosti a najdôležitejšie triedy, s ktorými pracujem aj ja.
- V kapitole **Návrh** je koncept môjho rozšírenia, popis nástrojov použitých pri jeho implementácii.
- Kapitola **Realizácia** obsahuje konkrétne úpravy v grafovom editore a tiež zakomponovanie rozšírenia do tohoto editoru.

Kapitola 1

Teoretický základ

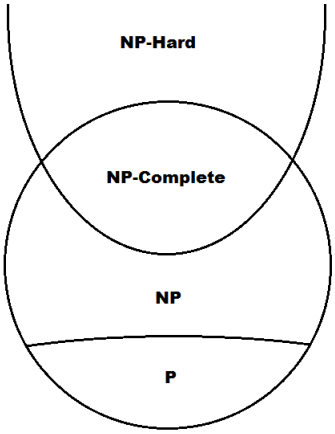
V tejto sekcii sa budem venovať teoretickému základu či už práce, na ktorú naväzujem, alebo mojej práce ako takej.

1.1 Teória zložitosti

Teória zložitosti (podrobnejšie viď [3]) je matematicko-informatickým odvetvím, ktoré sa zaoberá klasifikáciou výpočetných problémov. Tieto výpočetné problémy budeme chápať ako úlohy vyriešiteľné algoritmicky. Klasifikácia prebieha podľa zložitosti, ktorá sa delí na časovú a pamäťovú zložitosť. Časová zložitosť sa dá intuitívne chápať ako počet krokov algoritmu v závislosti na veľkosti vstupu. Vhodným príkladom je sčítanie dvoch matíc s n riadkami a $2n$ stĺpcami. Časová zložitosť takejto procedúry je $O(n^2)$, keďže v každom z n riadkov musíme sčítať $2n$ hodnôt. Toto by mohlo zavádzať, že časová zložitosť je $O(2n^2)$, ale, pri časovej zložitosti sa zanedbávajú konštanty. Takže $O(kn^2)$, kde $k \in R$ je konštanta, značíme ako $O(n^2)$.

Turingov stroj (TS) je teoretický model počítača opísaný matematikom Alanom Turingom. Skladá sa z procesorovej jednotky, tvorenej konečným automatom, programu v tvare pravidiel prechodovej funkcie a pravostrannej nekonečnej pásky pre zápis medzivýsledkov. Turingov stroj môže byť deterministický, alebo nedeterministický podľa toho, či jeho prechodová funkcia má pre každú kombináciu vstupu a stavu najviac jeden výsledný stav (deterministický), alebo množinu stavov (nedeterministický). Podľa časovej zložitosti delíme problémy na:

- P - je trieda zložitosti, ktorá obsahuje všetky problémy riešiteľné deterministickým Turingovovým strojom v polynomiálnom čase. Patria sem teda problémy riešiteľné so zložitosťou $O(n^k)$, kde $k \in R$ je konštanta.
- NP - je trieda zložitosti, ktorá obsahuje všetky problémy riešiteľné nedeterministickým Turingovovým strojom v polynomiálnom čase. Jej podmnožinou je teda aj P. [4]



Obr. 1.1: Triedy zložitosti

- NP-ťažké - je trieda zložitosti, ktorá obsahuje problémy, ktoré sú „aspoň také ťažké ako najťažší problémy v NP“. Problém je NP-ťažký keď každý problém v NP je naň redukovateľný v polynomiálnom čase.
- NP-úplné sú prienikom problémov NP a NP-ťažkých, čiže reprezentujú najťažšie problémy v NP a zároveň sa všetky problémy z NP dajú na ne v polynomiálnom čase redukovať. [5]

Obrázok 1.1 ukazuje rozloženie týchto problémov.

1.2 Teória grafov

Je to časť diskrétnej matematiky ktorá sa zaoberá grafmi, vlastnosťami a algoritmami (podrobnejšie viď [6]). Tieto algoritmy a vlastnosti sú však na teoretickej úrovni. Všetky ale pracujú s grafmi. Neorientovaným grafom nazývame usporiadanú trojicu $G =< H, U, i >$, prvky množiny H sú hranami grafu G, prvky množiny U uzly grafu G a zobrazenie incidenciou grafu G. Úloha incidencie grafu spočíva v tom, že priraďuje každej jeho hrane neusporiadanú dvojicu uzlov.

Ak je $(h) = [u, v]$, nazývame uzly u, v krajnými uzly hrany h, o hrane h hovoríme, že inciduje s uzlami u, v. Zvláštne prípady medzi hranami predstavujú tzv. slučky, pre ktoré je $(h) = [u, u]$. Stupňom uzlu rozumieme počet hrán s týmto uzlom incidentných. Izolovaným uzlom grafu nazývame taký uzol v, s ktorým neinciduje žiadna hrana, t.j. jeho stupeň $deg(v) = 0$

Takto definovaný graf však umožňuje existenciu takzvaných rovnobežných hrán, t.j. pre nejakú dvojicu hrán $h1$, $h2$ platí, že $(h1) = (h2) = [u, v]$. Graf, ktorý umožňuje existenciu takýchto hrán nazývame multigraf, naopak graf, ktorý ich neumožňuje nazývame prostým grafom. Ďalší druh grafov možno získať tak, že zakážeme existenciu slučiek - prosté grafy bez slučiek budeme

nazývať obyčajnými grafy.

Graf sa nazýva hranovo ohodnoteným grafom, keď $\forall h \in H$ je priradené nejaké číslo $w \in R$ (váha hrany). Spomínaný editor pred rozšírením určoval len dve vlastnosti a to planaritu a ofarbenie grafu. Tieto vlastnosti, ako aj zistenie či graf obsahuje Eulerový ťah, vyžadujú obyčajný neorientovaný graf. Algoritmy na získanie minimálnej kostry grafu a nájdenie optimálnej cesty pre obchodného cestujúceho (*travelling salesman problem*) však vyžadujú hranovo ohodnotený graf.

1.3 Vlastnosti grafov a grafové algoritmy

V tejto sekcii popíšem vlastnosti grafov a grafové algoritmy, ktoré sú v práci, na ktorú naväzujem implementované a aj tie, ktoré plánujem implementovať ja.

1.3.1 Farbenie grafu

Farbenie grafu je priradzovanie farieb (najčastejšie reprezentovaných ako prirodzené čísla) jednotlivým elementom grafu. Farbenie uzlov znamená priradiť uzlom grafu farby tak, aby žiadne dva susedné uzly nemali rovnakú farbu.

Analogicky, farbenie hrán je priradzovanie farieb jednotlivým hranám tak, aby žiadne dve susedné hrany nemali rovnakú farbu.

V bakalárskej práci, na ktorú naväzujem sa pracuje s farbením uzlov. Farbenie, ktoré priradí uzlom najviac k farieb nazývame k-farbenie grafu. Najmenší počet farieb potrebný k zafarbeniu grafu G nazývame chromatické číslo a značíme $\chi(G)$.

Uzly rovnakej farby tvoria farebnú triedu. Každá takáto trieda tvorí nezávislú množinu. To znamená, že k-farebnosť delí graf a jeho vrcholy tak isto ako keby bol rozdelený na k partícií a pojmy k-partitný a k-farebný graf majú rovnaký význam. Stanoviť chromatické číslo grafu je NP-úplný problém. [7]

Existuje celý rad algoritmov pre výpočet chromatického čísla, ale niektoré len odhadujú hornú hranicu, ktorú chromatické číslo neprekročí, a len niektoré počas jeho behu zafarbia graf tak, aby bol zafarbený správne a len daným počtom farieb.

V praxi sa potom často používajú algoritmy, ktoré kladú určité podmienky na graf, alebo algoritmy založené na heuristike. Ich cieľom nie je nájsť presné riešenie, ale iba riešenie dostatočne dobré. Všeobecne sa používajú v situáciách, keď dostupné zdroje nestačia na využitie exaktných algoritmov (čas) [8].

Jednou skupinou algoritmov založených na heuristike, ktoré riešia tento problém, sú algoritmy sekvenčné. Vyberú podľa daných kritérií nezafarbený uzol a priradí mu prvú doposiaľ použitú farbu, ktorú nemajú jeho susedia. Ak žiadna taká farba nie je, uzlu sa priradí nová farba. Takto sa pokračuje až kým nie sú zafarbené všetky uzly [8].

Algorithm 1 Sequence algorithm

```
procedure algorithmSequence(graph)
    list ← graph.vertices
    color ← 0
    while list.size ≠ 0 do
        v ← list[0]
        for j ← 1, list.size − 1 do
            if list[j].coloredNeighbours > v.coloredNeighbours then
                u ← list[j]
            else if list[j].coloredNeighbours = v.coloredNeighbours then
                if list[j].notColoredNeigh > v.notColoredNeigh then
                    v ← list[j]
        for j ← 0, color do
            if ∄ vertex x ∈ v.neighbours, x.color = j then
                v.color ← j
                break
        if v.color not set then
            v.color ← color + 1
            color ← color + 1
        list ← list − v
    return color
```

Algoritmus Welsh-Powel [9] je heuristický algoritmus, presnejšie *greedy coloring algoritmus* a chromatické číslo len odhaduje.

Algorithm 2 Welsh-Powell Algorithm

```
procedure algorithmWelshPowell(graph)
    list ← graph.nodes
    i ← 1                                   ▷ number of current color
    while list.size ≠ 0 do                  ▷ while there are uncolored vertices
        sort list by vertex degree
        list[0].color ← i
        for j ← 1, list.size − 1 do
            if vertex list[0] ∉ list[j].neighbours then
                list[j].color ← i
        list delete colored vertices
        i ← i + 1
    return i                    ▷ i = upper estimate of the chromatic number
```

Algoritmy 2 aj 1 sú oba implementované v práci, na ktorú navädzujem, ale iba ukázaný sekvenčný algoritmus sa reálne využíva na ofarbenie uzlov a zároveň určenie chromatického čísla.

1.3.2 Planarita

Rovinný graf (alebo tiež planárny) je graf, ktorý možno v rovine znázorniť takým spôsobom, že žiadne dve hrany sa nekrížia. O planarite nám tiež hovorí Kuratowského veta, ktorá sa používa v niektorých algoritmoch. „Graf je rovinný práve vtedy, ak neobsahuje žiadny podgraf homeomorfný s grafom K_5 ani žiadny podgraf homeomorfný s grafom $K_{3,3}$“. [1]

Zistiť či je graf planárny sa dá v časovej zložitosti $O(n)$, kde n je počet uzlov daného grafu. Slúži k tomu napr. algoritmus navrhnutý Boyerom a Myrvoldovou. [10] Tento algoritmus alebo tiež algoritmus, ktorého autormi sú Fraysseix, Mendez a Rosenstieh [11], patrí do skupiny algoritmov *Edge addition method*, ktoré iteratívne pridávajú do novo tvoreného grafu hrany z grafu pôvodného.

Ďalšími skupinami algoritmov sú *Path addition method* a *Vertex addition method*. Algoritmy patriace do prvej skupiny vyhľadajú cesty, ktoré pridávajú do nového grafu. Do tejto skupiny patrí napr. Hopcroft a Tarjan [12]. *Vertex addition method* je skupina, ktorá opäť postupne pridáva, ale uzly (vrcholy), ako už názov napovedá. Algoritmy, ktoré postupne pridávajú uzly, sú napr. algoritmus od Lempela, Evena a Cederbauma [13] alebo algoritmus, ktorého autormi sú Booth a Lueke. Ďalšími algoritmmami sú napr. Demoucron, Malgrange a Pertuiset algoritmus [14] alebo Auslander-Parter algoritmus [15].

Algoritmus 3 je použitý pri testovaní planarity v GraphBrowseri.

1.3.3 Sled, ťah, Eulerovký ťah, Eulerovský graf

Sled je taká postupnosť uzlov, že medzi každými dvoma po sebe idúcimi uzlami existuje hrana. Ťah je sled, v ktorom sa neopakujú hrany. Uzavretý ťah je taký ťah, v ktorom začiatočný uzol je zároveň aj koncovým uzlom. Cesta je taký ťah, v ktorom sa neopakujú uzly. Eulerovský ťah je ťah, ktorý obsahuje všetky hrany grafu. Graf obsahuje otvorený Eulerovský ťah práve vtedy, keď je spojitý (spája všetky uzly, z každého uzlu existuje cesta do ostatných uzlov, stupeň žiadneho uzlu nie je 0) a stupeň najviac dvoch uzlov je nepárny (ostatné sú párne). Uzavretý Eulerovský ťah je uzavretý ťah, ktorý obsahuje všetky hrany grafu. Eulerovský graf je graf, ktorý obsahuje uzavretý Eulerovský ťah. Je to práve vtedy, keď je spojitý a má všetky stupne uzlov párne. [16]

Algoritmus na nájdenie takéhoto uzavretého ťahu je takmer tak isto priamočiary ako tieto podmienky. Predpokladajme teda spojitý graf s uzlami párneho stupňa. Napríklad Hierholzerov algoritmus [17] z roku 1873 ukazuje jednoduchosť riešenia. Jeho podstata spočíva vo vybratí akéhokoľvek uzlu, keďže sa jedná o uzavretý ťah a musíme sa doň aj vrátiť. z tohoto vybratého uzlu u vedieme ťah (je jedno ako ťaháme, vždy sa vieme dostať do tohoto uzlu naspäť). Po niekoľkých krokoch sme sa ocitli zase v začiatočnom uzle. Týmto sme získali uzavretý ťah $< u, h_1, \cdots, h_n, u >$. Nie je však ešte zaručené, že sme pokryli všetky hrany a teda sa nemusí jednať o uzavretý Eulerovský ťah.

Algorithm 3 Algoritmus Demoucron, Malgrange a Pertuiset

```
procedure algorithmDemoucron(graph)
    c ← findCircuit(graph)
    if c = null then
        return true
    else
        faces[0] ← faces[1] ← c
        graph ← graph − c
        components ← getComponents(graph)
        while components.velikost ≠ 0 do
            face ← null
            component ← null
            for i ← 0, component.size − 1 do
                component ← components[i]
                component.faceCount ← 0
                for j ← 0, faces.size − 1 do
                    face ← faces[j]
                    if canBeDrawn(component, face) then
                        component.faceCount ← component.faceCount + 1
                if component.faceCount = 0 then
                    return false
                else if component.faceCount = 1 then
                    break
            path ← findPath(component, face)
            faces ← faces − face + divideFace(face, path)
            components ← components − component + getComponents(component − path)
        return true
```

Skúsme sa teda pozrieť na tento ťah ešte raz. Ak narazíme v danom ťahu na nejaký uzol v ($< u, h_1, \cdots, v, \cdots, h_n, u >$), z ktorého ešte vedú nepokryté hrany, vykonáme tento algoritmus z tohoto uzlu ignorujúc už pokryté hrany. Získali sme ďalší uzavretý ťah $< v, h_k, \cdots, h_l, v >$. Keďže v je uzlom aj nášho pôvodného ťahu, aj ťahu nového, môžeme dané ťahy spojiť a to spôsobom $< u, h_1, \cdots, v, h_k, \cdots, h_l, v, \cdots, h_n, u >$. Takto pokračujeme, až budú všetky hrany pokryté. Výsledný ťah je Eulerovský ťah.

1.3.4 Hamiltonovská cesta, Hamiltonovský graf

Kružnica je taká cesta, ktorá začína aj končí v rovnakom uzle. Hamiltonovská cesta je taká cesta, ktorá každý uzol grafu navštívi práve raz. Hamiltonovská kružnica je taká Hamiltonovská cesta, ktorá je kružnica. Hamiltonovský graf je taký graf, ktorý obsahuje Hamiltonovskú kružnicu. Problém Hamiltonovskej

cesty a problém Hamiltonovskej kružnice sú problémy zaoberajúce sa tým, či daný graf obsahuje daný prvok. V roku 1857 Sir William Rowan Hamilton uviedol matematický hlavolam a to; či je možné viesť po hranách pravidelného dvanásťstena takú kružnicu, aby obsahovala každý vrchol práve raz. Na pohľad podobný problém ako s nájdením Eulerovského ťahu také rozhodne nie sú. Oba tieto problémy patria do skupiny NP-úplných problémov [18]. Existuje ale množstvo postačujúcich podmienok pre to, aby bol graf Hamiltonovský. Spomeniem dve najznámejšie:

1. ORE Nech G je graf, $|V| = n, n \geq 3$. Ak pre každú dvojicu vrcholov, ktoré nie sú spojené hranou je súčet ich stupňov aspoň n, tak G je Hamiltonovský. [19]

2. DIRAC Nech G je graf, $|V| = n, n \geq 3$. Ak má každý vrchol stupeň aspoň $n/2$, tak graf je Hamiltonovský. [20]

Keďže sa ale jedná o NP-úplný problém znamená to, že so vzrastajúcim počtom vrcholov a hrán rastie čas riešenia exponenciálne. Preto sa k algoritmom zisťujúcim, či daný graf obsahuje Hamiltonovskú kružnicu/cestu pristupuje tromi základnými spôsobmi:

- algoritmy pre presné nájdenie riešenia (pre problémy s malým počtom vrcholov)

- suboptimálne a heuristické algoritmy (osvedčené v dostatočnom počte prípadov, nedokázateľné, nemusia zaručovať optimálne riešenia ani hromadnosť)

- nájdenie špeciálneho typu grafu, pre ktorý vieme určiť presné riešenie

Medzi presné algoritmy nájdenia takejto kružnice/cesty patria napríklad algoritmy

- Brute force search, ktorý v grafe o n uzloch skúsi všetky poradia týchto uzlov, čiže jeho časová zložitosť je $O(n!)$

- Frank Rubinov algoritmus [21], ktorý rozdelí hrany grafu do troch skupín:

  1. Hrany, ktoré musia byť v danej kružnici/ceste
  2. Hrany, ktoré v nej byť naopak nemôžu
  3. Nerozhodné hrany

  Ako algoritmus ide, rozhoduje o tretej kategórii a či zastaviť hľadanie, alebo v ňom pokračovať. Algoritmus rozdeľuje graf na komponenty, ktoré môžu byť vyriešené oddelene.

- Bellman-Held-Karp algoritmus [22] je algoritmus využívajúci dynamické programovanie s časovou zložitosťou $O(n^2 2^n)$. Pracuje spôsobom, že pre každú množinu uzlov a pre každý uzol v v tejto množine skúma, či existuje cesta pokrývajúca všetky uzly v tejto množine a zároveň končiaca v uzle v.

- Backtracking. Jedná sa o algoritmus na nájdenie Hamiltonovskej kružnice. Pridá do výslednej kružnice jeden uzol. Keďže ide o kružnicu vedúcu cez všetky uzly, je jedno ktorý uzol pridáme. Potom sa rekurzívne snaží nadpájať ďalšie uzly. Ak sú už všetky uzly zahrnuté a zároveň z posledného uzlu vedie hrana do počiatočného uzlu kružnice, algoritmus končí, ak sa dostane do nejakého stavu, z ktorého sa nedá v kružnici pokračovať, vynorí sa z rekurzie a skúša ďalšie možnosti.

Úzko súvisiacim problémom s Hamiltonovským grafom je tiež problém obchodného cestujúceho (*travelling salesman problem*), ktorý sa zaoberá nájdením optimálnej Hamiltonovskej kružnice, t.j. takej, že je súčet ohodnotení jednotlivých hrán je minimálny. Je zaujímavý hlavne kvôli jeho praktickej využiteľnosti. Na pohľad triviálna myšlienka je však tiež klasifikovaná ako NP-úplný problém.

1.3.5 Kostra, minimálna kostra

Podgraf grafu G je taký graf, ktorý vznikne z pôvodného grafu G vymazaním vybraných hrán, respektíve uzlov a zároveň aj všetkých hrán vedúcich z týchto uzlov. Kostra je taký podgraf, ktorý obsahuje všetky uzly pôvodného grafu, je stromom a spája všetky uzly (medzi každými dvoma uzlami existuje práve jedna cesta). Pojem minimálna kostra pracuje s hranovo ohodnoteným grafom. Je to taká kostra, ktorej súčet váh hrán je minimálny spomedzi všetkých kostier.

Nájdením minimálnej kostry sa zaoberajú aj tri najznámejšie algoritmy.

Prvý algoritmus pre nájdenie minimálnej kostry bola vyvinutý českým vedcom Otakarom Borúvkom v roku 1926. Borůvkův algoritmus [23][24] je algoritmus pre nájdenie minimálnej kostry v grafe, ktorého všetky váhy hrán sú odlišné. Bol prvýkrát publikovaný v roku 1926 Otakarom Borůvkom pre nájdenie efektívneho spôsobu výstavby elektrického vedenia na Morave. Algoritmus bol znovuobjavený Choquetom [25] v roku 1938. Opäť Florekom, Łukasiewiczom, Perkálom, Steinhausom, a Zubrzyckim [26] v roku 1951; a znovu Sollinom [27] v roku 1965. Vzhľadom k tomu, že Sollin bol jediný počítačový vedec žijúci v anglicky hovoriacej krajine v tomto zozname, tento algoritmus je často nazývaný Sollinov algoritmus.

Algoritmus začína tým, že najprv skúma každý vrchol a pridá do minimálnej kostry najlacnejšiu hranu z tohto vrcholu a to bez ohľadu na už pridané hrany. Týmto algoritmus tvorí celky. Pokračuje spájaním týchto zoskupení

v podobným spôsobom (pridaním vždy najlacnejšej hrany), kým kostra spája všetky vrcholy.

Nasledúci pseudokód znázorňuje beh Borůvkovho algoritmu.

Algorithm 4 Borůvkův algoritmus

```
1: function getMinimumSpanningTree(OriginalGraph)
2:     spanningTree ← all vertices of OriginalGraph
3:     while spanningTree not connected do
4:         for each component C in spanningTree do
5:             edge ← cheapest edge from C to an outside vertex
6:             add edge to spanningTree
       return spanningTree
```

Časová zložitosť tohoto algoritmu je $O(E \log V)$, kde$E$ je počet hrán a V počet uzlov. Jeho nevýhodou je však potreba grafu, kde žiadne dve hrany nie sú rovnako ohodnotené.

Jarník-Primov algoritmus bol vyvinutý v roku 1930 českým matematikom Vojtěchom Jarníkom [28] a v roku 1957 nezávisle na ňom počítačovým vedcom Robertom C. Primom [29]. Taktiež ho znovuobjavil Edsger Dijkstra v roku 1959.

Algoritmus pracuje tak, že najskôr si vyberie jeden uzol, pridá ho do minimálnej kostry. Iteratívne sa potom snaží rozširovať už existujúcu kostru o najnižšie ohodnotené hrany vedúce do uzlov, ktoré ešte v kostre nie sú zahrnuté.

Algorithm 5 Jarník-Prim algoritmus

```
function getMinimumSpanningTree(OriginalGraph)
    spanningTree ← one vertex of OriginalGraph
    while spanningTree not contains all vertices do
        edge ← cheapest edge from spanningTree to outsideVertex
        add outsideVertex to spanningTree
        add edge to spanningTree
    return spanningTree
```

Tento algoritmus už nepredpokladá rozdielne ohodnotenia hrán a stále jeho časová zložitosť je $O(E \log V)$.

Tretím dôležitým algoritmom a zároveň aj algoritmom použitým v tejto práci je Kruskalov algoritmus [30] nájde hranu s najmenšou hmotnosťou, ktorá spája akékoľvek dva stromy v lese [31]. Tento algoritmus sa prvýkrát objavil v roku 1956 a bol napísaný Josephom Kruskalom.

Aj tento algoritmus pracuje v čase $O(E \log V)$.

Algorithm 6 Kruskal algoritmus

```
function getMinimumSpanningTree(OriginalGraph)
    spanningTree ← all vertices of OriginalGraph
    while spanningTree not connected do
        edge ← another cheapest edge from spanningTree
        if edge connects 2 trees then
            add edge to spanningTree
    return spanningTree
```

1.4 Programovací jazyk Java

Je to programovací jazyk [32] ktorého syntax vychádza z C/C++. Výhodou výberu Javy je hlavne prenositeľnosť a nezávislosť od operačného systému „*write once, run anywhere*“ (WORA), vďaka kompilácii do bytecodu, ktorý preloží Java Virtual Machine (JVM) na akejkoľvek architektúre počítača. Ďalšou výhodou je jednoduchá alokácia prostriedkov a taktiež jednoduchá dealokácia - žiadna. Garbage collector sa stará o akékoľvek uvolnenie pamäte, aj keď sa väčšinou jedná o uvoľňovanie až po dokončení behu programu. Medzi výhody ďalej patria knižnice, ktoré jednoducho vieme importovať a tak využívať mnohé užitočné funkcie.

Samozrejme Java má aj svoje nevýhody. Spomínaný Garbage collector má pri komplikovanejších programoch veľké problémy s dealokáciou. Tiež knižnice majú mnohokrát nespočetné množstvo chýb/bugov, hlavne v grafických knižniciach (ktoré budú dôležité akurát v mojej práci). Samotný Java Virtual Machine má niekoľko známych bugov, takže ani beh samotnej Javy nie je bezchybný. Nevýhodou oproti C++ je napríklad neexistencia referencií, ktoré sú nedostatočne nahradené nezameniteľnými typmi (*Immutable types*).

1.5 Architektúra MVC

Model-View-Controller (MVC) [33] je softvérový architektonický vzor pre realizáciu užívateľských rozhraní. Rozdeľuje softvérové aplikácie do troch vzájomne prepojených častí, aby sa oddelili reprezentácie informácií od spôsobu, akým sú informácie prezentované alebo prijaté od užívateľa.

Rovnako ako u iných softvérových vzorov MVC vyjadruje „jadro riešenia“ problému a zároveň umožňuje, aby bola špecificky upravená pre každý systém.

1.5.1 Komponenty

Ústredným prvkom v MVC je model a zachytáva správanie aplikácie v zmysle problémovej domény, nezávisle na užívateľskom rozhraní. Tento model priamo riadi dáta a logiku aplikácie. View môže byť ľubovoľný výstup reprezentá-

cie informácií, ako je graf, diagram. Tretia časť, controller, prijíma vstup a konvertuje ho na príkazy pre model alebo view.

1.5.2 Interakcie

Okrem rozdelenia aplikácie na tri druhy komponent, koncept Model-View-Controller definuje interakcie medzi nimi. Controller môže odosielať príkazy modelu na aktualizáciu informácií o jeho stave. Môže tiež odosielať príkazy jemu pridruženému viewu na zmenu prezentácie modelu v danom zobrazení. Model oznámi svojim pridruženým viewom a controllerom, keď došlo k zmene jej stavu. Toto oznámenie umožňuje viewom aktualizovať výstup a controllerom zmeniť sadu príkazov, ktoré sú k dispozícii. View žiada informácie z modelu a ďalej ich používa pre generovanie výstupu pre užívateľa.

1.6 Knižnice na prácu s grafmi

Keďže je GraphBrowser napísaný v programovacom jazyku Java, naskytuje sa možnosť využiť niektorú z množstva knižníc, vytvorených na prácu s grafmi. Medzi najznámejšie patria napríklad knižnice:

- JGraphT [34] je grafová knižnica napísaná v Jave, ktorá poskytuje rôzne objekty a algoritmy z teórie grafov. JGraphT podporuje rôzne typy grafov, vrátane:

  Orientované aj neorientované grafy. Grafy s váženými/neváženými/označenými alebo ľubovoľnými užívateľom špecifikovanými hranami. Rôzne možnosti multiplicít hrán vrátane: obyčaných grafov, jednoduchých grafov, multigrafov, pseudografov. Listenable grafy, ktoré umožňujú externým listenerom sledovať udalosti.

  Aj keď je silným nástrojom, JGraphT je navrhnutý tak, aby bol jednoduchý a type-safe. Napríklad, vrcholy grafu môžu byť akékoľvek objekty.

- JUNG [35] - Java Universal Network/Graph Framework - je softvérová knižnica, ktorá poskytuje spoločný jazyk pre modelovanie, analýzu a vizualizáciu dát, ktoré môžu byť reprezentované vo forme grafu alebo siete. Je napísaná v programovacom jazyku Java, ktorý umožňuje aplikáciám založených na JUNGu využiť rozsiahle vstavané schopnosti Java API, rovnako ako aj ostatné existujúce Java knižnice tretích strán.

  JUNG architektúra je navrhnutá tak, aby podporu rôznych reprezentácií entít a ich vzťahov, ako sú napríklad orientované a neorientované grafy, multi-modálne grafy, grafy s rovnobežnými hranami a hypergrafy. Uľahčuje vytváranie analytických nástrojov pre komplexné data sety, ktoré môžu skúmať vzťahy medzi subjektmi. Aktuálna distribúcia JUNG

zahŕňa implementáciu niekoľkých algoritmov z teórie grafov, data miningu a analýzy sociálnych sietí, ako je napríklad clustering, dekompozícia, optimalizácia, generovanie náhodného grafu, štatistické analýzy a výpočet vzdialeností v sieťach, tokoch. Okrem toho, poskytuje mechanizmy na filtrovanie, ktoré umožňujú užívateľom zamerať svoju pozornosť, alebo ich algoritmy, na špecifickú časť grafu. Ako open source knižnica, JUNG stanovuje spoločný framework pre grafovú/ sieťovú analýzu a vizualizáciu.

- GMGraphLib [36] je grafová knižnica naprogramovaná v Jave. Poskytuje programátorovi rozhranie na vytvorenie vyhľadávania, ako BFS a DFS, detekovanie komponent a cyklov v grafe. Pracuje ako s orientovanými aj neorientovanými grafy, či už s ohodnotenými alebo neohodnotenými (aj hranami aj uzlami). Okrem toho poskytuje aj radu grafových algoritmov zahŕňajúce algoritmy ako Dijkstra a Prim. Taktiež ponúka algoritmus, ktorý poskytuje približne optimálne riešenie na problém obchodného cestujúceho.

1.7 Načítanie a ukladanie grafu

Práca, na ktorú naväzujem umožňuje ukladanie a opätovné načítanie rôznych formátov grafov. Medzi tieto formáty patria TGF, DOT, GML.

Formát TGF umožňuje zaznamenať čistú grafovú štruktúru, ktorej elementy (uzly a hrany) môžu mať iba atribút popisok. Každý riadok prináleží práve jednému elementu. Najprv sa zapíšu uzly, kde riadok s uzlom obsahuje id uzla a medzerou oddelený popisok. Potom sa vloží symbol mriežka, ktorý oddeľuje uzly od hrán. Nakoniec sa zapíšu hrany, kde riadok s hranou obsahuje id zdrojového uzla, medzerou oddelené id cieľového uzla a prípadne medzerou oddelený popisok hrany. V práci, na ktorú naväzujem je tento formát rozšírený na formát TGFC, ktorý je základným formátom pre aplikáciu. Formáte TGF chýbajú súradnice pre jednotlivé uzly, TGFC ich naopak obsahuje. Súradnice uzlov sú vložené medzi id uzla a jeho popisok, výsledný riadok s uzlom potom obsahuje id uzla, x-ovú súradnicu, y-ovú súradnicu a prípadne popisok, jednotlivé atribúty sú opäť oddelené medzerou.

DOT je grafový formát, ktorý umožňuje vytvárať orientované aj neorientované grafy s rôznymi atribútmi. Štruktúra tohto formátu je pomerne zložitejšia oproti TGF, ale pretože sa stále jedná o obyčajný text, používateľ v ňom môže ľahko čítať a editovať informácie o grafe.

GML je ďalším grafovým formátom, ktorého kľúčovými vlastnosťami sú prenositeľnosť, jednoduchá syntax, rozšíriteľnosť a flexibilita. Tento formát obsahuje hierarchický zoznam, ktorého prvky sú dvojice kľúč (meno atribútu) a hodnota.

Kapitola 2

Analýza

V tejto kapitole sa budem venovať analýze požiadaviek na moju prácu a zároveň analýzou GraphBrowseru.

Analýza je jeden zo základných stavebných kameňov pri vývoji rozšírení, aj aplikácií ako takých. Jedná sa o popis toho, čo má výsledná aplikácia robiť. Kvalitne spracovanou analýzou je možné predísť chybám, ktoré by vznikli pri implementácii a bolo by náročné ich odstrániť po ich vzniku.

2.1 Analýza požiadaviek

Analýza požiadaviek ukazuje funkčné a nefunkčné požiadavky na výslednú aplikáciu. Funkčné požiadavky popisujú možnosti interakcie užívateľa s aplikáciou, kým nefunkčné požiadavky určujú nároky na jeho implementáciu alebo výkonnosť aplikácie.

2.1.1 Funkčné požiadavky

- Pridanie váh hrán
- Pridanie zistenia, či je graf Eulerovský
- Pridanie zistenia, či je graf Hamiltonovský
- Pridanie nájdenia minimálnej kostry
- Zahrnutie týchto vlastností do užívateľského rozhrania

2.1.2 Nefunkčné požiadavky

- Použitie programovacieho jazyku Java
- Pokračovanie v architektúre MVC s návrhovým vzorom Observer

2.2 Splnenie bodov zadania

Táto sekcia jasne určuje ciele práce tak, aby bolo splnené zadanie.

2.2.1 Pridanie váh hrán

Keďže niektoré vlastnosti, ktoré zo zadania musí táto práca zahŕňať vyžadujú aby bol graf hranovo ohodnotený, užívateľ bude môcť tvoriť hranovo ohodnotené grafy a upravovať hodnoty týchto hrán.

2.2.2 Pridanie zistenia, či je graf Eulerovský

Aplikácia zistí, či graf, ktorý momentálne v GraphBrowseri editujeme je Eulerovský.

2.2.3 Pridanie zistenia, či je graf Hamiltonovský

Aplikácia zistí, či graf, ktorý momentálne v GraphBrowseri editujeme je Hamiltonovský.

2.2.4 Pridanie nájdenia minimálnej kostry

Rozšírenie umožní užívateľovi nájsť a zobraziť minimálnu kostru grafu, ktorý momentálne v GraphBrowseri editujeme .

2.2.5 Zahrnutie vlastností do užívateľského rozhrania

Rozšírenie tiež bude tieto všetky algoritmy a vlastnosti zahŕňať v grafickom rozhraní aplikácie, tak aby grafické rozhranie zostalo čo najprívetivejšie.

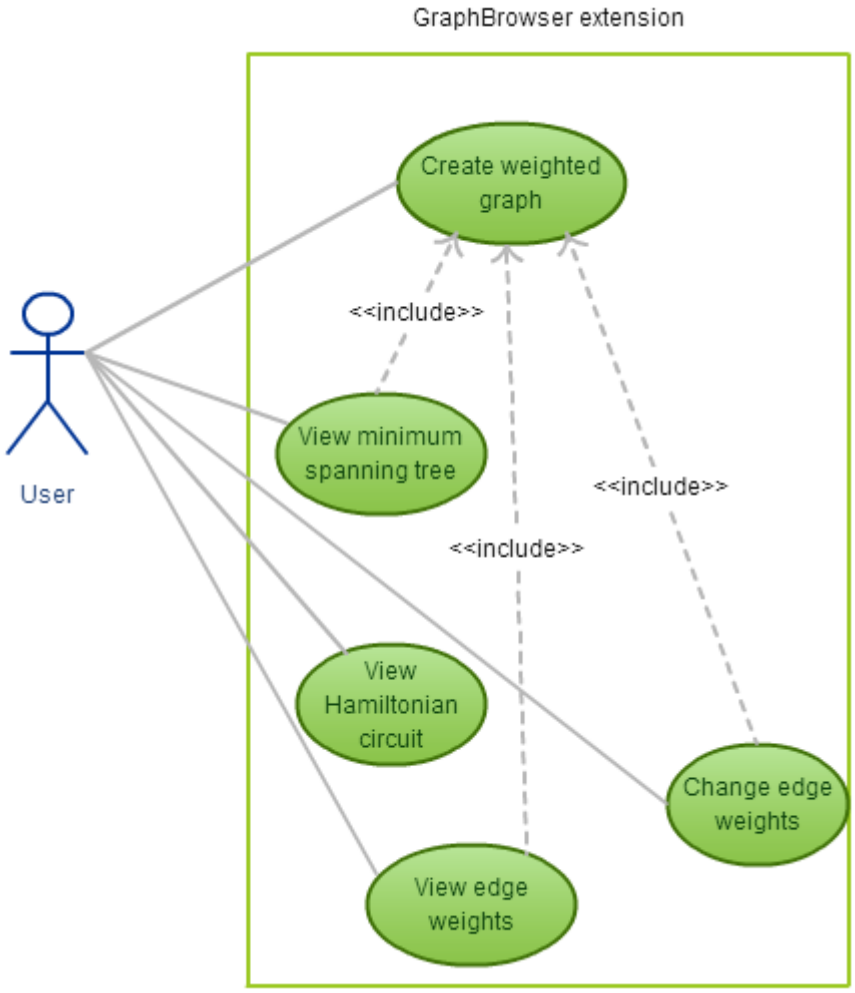
Na základe týchto požiadaviek som vytvoril use case diagram 2.1.

2.2.6 Použitie programovacieho jazyku Java s architektúrou MVC s návrhovým vzorom Observer

Implementácia rozšírenia bude prebiehať v programovacom jazyku Java, využitá architektúra pri tvorbe užívateľského rozhrania zase bude MVC s návrhovým vzorom Observer.

2.3 GraphBrowser

V tejto sekcii sa venujem analýze už existujúceho programu, GraphBrowseru [2], nástrojom použitým na jeho tvorbu, ako aj jeho súčastiam. Je to grafový editor, ktorý umožňuje užívateľovi vytvárať grafy, overovať ich vlastnosti a následne ukladať vo viacerých formátoch (taktiež ich opätovné načítať). Zmienenými vlastnosťami sú farebnosť a planarita. Grafový editor je desktopová

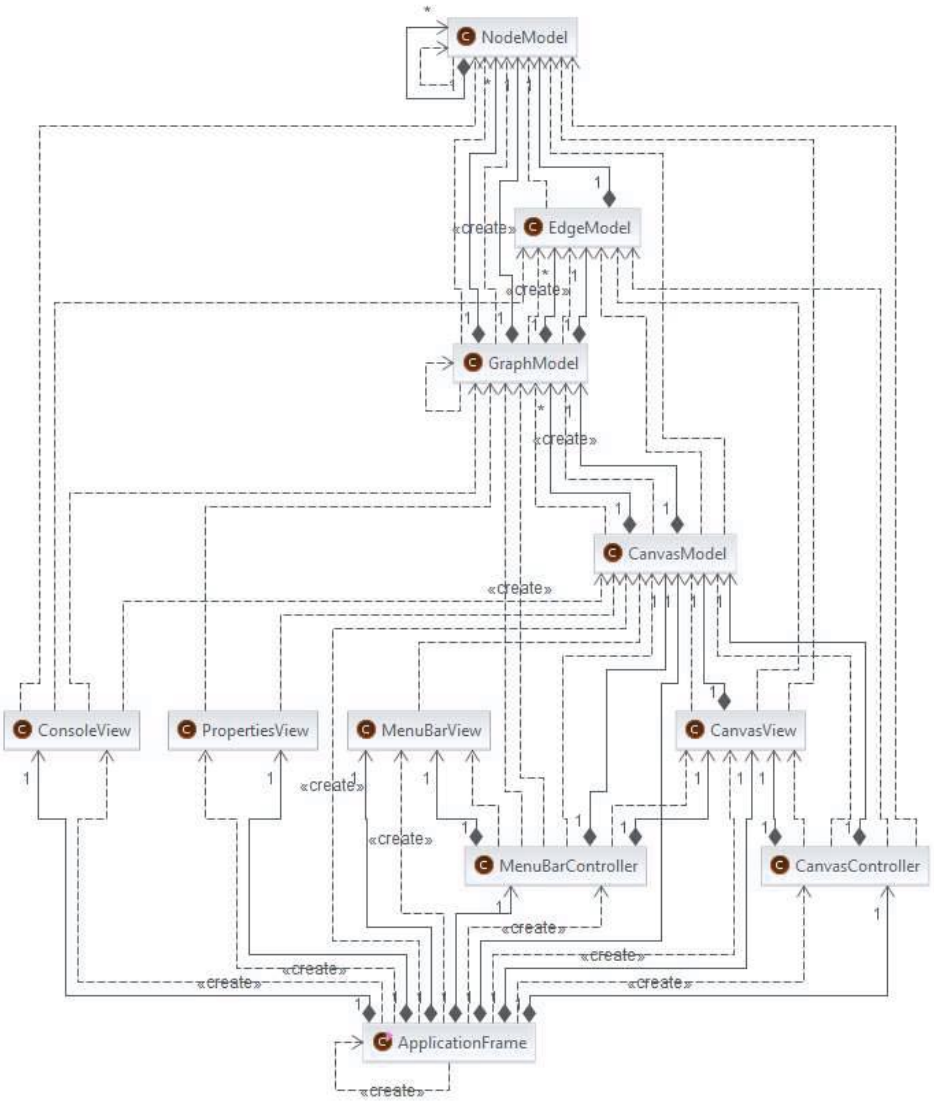


Obr. 2.1: Use case diagram pre rozšírenie

aplikácia písaná v programovacom jazyku Java s využitím architektúry MVC. Tento grafový editor je jedinečný z hľadiska výpočtu planarity.

Jedným z nedostatkov GraphBrowseru je pre mňa nekomplexnosť tohoto programu. z nespočetného množstva vlastností sa spracovávajú len dve (z hľadiska zložitosti problémov sú zaujímavé, ale použiteľnosť týchto vlastností nie je taká veľká, aká bola náročnosť).

Ďalej sa budem venovať opisu tried GraphBrowseru, ktoré sú pre toto rozšírenie nevyhnutné.



Obr. 2.2: Doménový model jadra GraphBrowseru

2.3.1 GraphModel

Trieda *GraphModel* je jadrom celej aplikácie keďže uchováva všetky informácie o grafe. Obsahuje dva atribúty množinového typu nesúce informácie o jeho hranách a uzloch a to *edges* a *nodes*. Bezparametrický konštruktor vytvorí obe množiny a tieto množiny ďalej upravujú rôzne metódy. Prvou takouto metódou je *addNode*, ktorá podľa parametrov vytvorí uzol a zároveň ho pridá do *nodes*. Ďalšou metódou je *addEdge*, ktorá analogicky vytvorí hranu medzi dvoma uzlami. Zároveň však zasahuje do susednosti jednotlivých uzlov. Keďže sa jedná o graf neorientovaný, pridá do množiny susedov počiatočného uzlu koncový uzol a naopak do množiny susedov koncového uzlu počiatočný uzol. Taktiež metódami upravujúcimi obsah týchto množín sú metódy *setEdges* a *setNodes*, ktoré podľa parametru nastavia celú množinu, či už uzlov alebo hrán na tento parameter.

Táto trieda tiež ukladá informácie o tom, či je graf planárny, či je nakreslený planárne (keďže planarita grafu nezávisí od toho ako je nakreslený, nemusia byť tieto atribúty rovnaké) a tiež chromatické číslo grafu. Všetky tieto atribúty sa nastavujú pomocou metódy *computeProperties*, ktorá využíva ďalšie triedy *ColoringService* a *PlanarityService*.

Keďže GraphBrowser ponúka aj možnosť ukladania grafu, je tu atribút *filePath*, ktorý obsahuje cestu k súboru, do ktorého ukladáme. Trieda samozrejme obsahuje aj setre a getre a zároveň metódu *toString* pre výpis a zjednodušené ukladanie grafu.

2.3.2 NodeModel

Trieda *NodeModel* obsahuje všetky informácie potrebné pre reprezentáciu uzlov. Obsahuje celočíselný identifikátor uzlu *id*, popisok *label*, zoznam susedov *neighbors*. Tiež zaznamenáva polohu uzlu na plátne použitím triedy Point. Polohu teda zaznamenáva *Point center*. Ďalšou dôležitou informáciou, ktorú uchováva *NodeModel* je farba uzlu. Táto farba sa nastavuje počas metódy *computeProperties* (spomenutej v sekcii 2.3.1). Metódy triedy *NodeModel* zahŕňajú konštruktor, nastavujúci *id, center, label* na hodnotu jednotlivých parametrov a farbu nastaví na 0.

Ďalšími zaujímavosťami ohľadom *NodeModel* sú metódy *NodeComparatorById* a *NodeComparatorByDegree*, ktoré implementujú rozhranie Comparator<NodeModel>. *NodeComparatorByDegree* sa využíva pri algoritme Welsh-Powell slúžiacom na horný odhad chromatického čísla grafu. Druhá metóda *NodeComparatorById* sa zase využíva pri pridaní nového uzlu do grafu po kliknutí na plátno.

Trieda tiež obsahuje setre a getre pre lepší prístup a metódu *toString* využívanú pri výpise grafu, alebo na získanie informácií o uzle.

2.3.3 EdgeModel

Trieda *EdgeModel* je trieda reprezentujúca hrany. Obsahuje informácie o jej počiatočnom uzle, koncovom uzle v dvoch atribútoch *NodeModel* source a *NodeModel target*. Aj keď sú dané atribúty pomenované *target* a *source* jedná sa o neorientovanú hranu.

Trieda *EdgeModel* obsahuje konštruktor, ktorý dané atribúty nastaví podľa parametrov. Keďže ale nie je možné v GraphBrowseri hrany upravovať takým spôsobom, že by sa niektoré z atribútov zmenili, táto metóda neobsahuje setre, ale iba getre. Samozrejmosťou je metóda *toString* zase využívanou či už pri ukladaní grafu, alebo zistení informácií o hranách.

2.3.4 Canvas

V tejto sekcii sa budem venovať plátnu a teda triedam, ktoré o ňom uchovávajú informácie, či už po vizuálnej, alebo modelovej stránke.

2.3.4.1 CanvasController

Trieda *CanvasController* je trieda identifikujúca vstupy od užívateľa z *CanvasView*. Keďže je treba identifikovať rôzne typy interakcie myši (klik ľavým tlačítkom, klik pravým tlačítkom, ťahanie), trieda dedí od *MouseAdapteru*. Metódami, ktoré teda bolo treba upraviť sú *mouseClicked()*, *mouseReleased()*, *mouseWheelMoved()*, *mouseDragged()* a *mouseMoved()*.

Metóda *mouseClicked()* sa zavolá, keď užívateľ klikne myšou. Táto metóda sa kvôli rôznorodosti jednotlivých operácii delí na *leftMouseButtonClicked()* a *rightMouseButtonClicked()*.

2.3.4.2 CanvasModel

Táto trieda reprezentuje celé plátno, uchováva graf aj rôzne nastavenia. Ako jediná trieda z GraphBrowseru dedí od triedy *observed*, čo jej umožňuje pridať si observery a notifikovať ich, ak sa graf alebo čokoľvek iné zmení.

Hneď štyri atribúty sú typu *GraphModel*, tými sú *graph*, *marked*, *unmarked* a *clipboard*. *Graph* uchováva celý graf, ktorý používateľ práve tvorí. *Marked* a *unmarked* sú navzájom disjunktné, sú podgrafmi práve tvoreného grafu, kde *marked* obsahuje všetky hrany a uzly, ktoré sú označené, a *unmarked* neoznačené, teda zvyšok. *Clipboard* funguje ako schránka a udržiava skopírovanú alebo vystrihnutú časť grafu.

Ďalej je tu skupina atribútov typu *boolean*, ktoré slúžia ako príznaky vnútorného stavu plátna. Atribút *isColoring* je príznak zapnutia či vypnutia farbenia uzlov podľa chromatického čísla. Ďalšími sú príznaky ťahanie - *isEdgeDragged* je *true*, ak je tvorená hrana, *isNodeDragged* je *true*, ak dochádza k posunu časti grafu, *isRectSelectionDragged* je *true*, ak používateľ ťahá obdĺžnik pre výber a *isScreenDragged* je *true*, ak používateľ pohybuje plátnom.

Zostávajúce príznaky sú *isNodeLabeling* a *isEdgeLabeling*, ktoré zapínajú a vypínajú popisky uzlov a hrán, a príznak *isSpaceEnough*, ktorý je *true*, ak je dostatok miesta pre vloženie nového uzlu.

Ďalej sú tu dôležité atribúty týkajúce sa polohy plátna. Atribút *cursor* typu *Point*, reprezentuje polohu kurzora. Ďalšími atribútmi tohto typu sú *point* a *shift*. *Point* slúži ako pomocný bod napr. pri ťahaní obdĺžnika pre výber a *shift* reprezentuje horizontálny a vertikálny posun plátna. Ďalej je tu *zoom* typu *double*, ktorý uchováva priblíženie plátna.

Čo sa týka metód, *CanvasModel* ich obsahuje zo všetkých najviac. Okrem konšštruktoru, setrov a geterov, obsahuje metódy, ktoré umožňujú všetky akcie dostupné užívateľovi.

Metóda *addGraphToCenter()* umožňuje pridať graf do existujúceho grafu. Táto metóda je využívaná napr. pri generovaní alebo kopírovaní grafu. Metóda musí zabezpečiť, že *id* jednotlivých uzlov pridávaného grafe nebudú kolidovať s *id* uzlov existujúceho grafu, preto metóda posunie všetky *id* pridávaného grafu o počet uzlov existujúceho grafu, teda na koniec. Ďalej metóda musí zabezpečiť, aby sa pridávaný graf zobrazil na stred plátna, inak by sa mohlo stať, že by užívateľ pridal graf niekde mimo zobrazenú časť a nevedel o tom. Potom je potrebné označiť celý pridávaný graf, aby s ním užívateľ mohol ihneď pohybovať na miesto, kam sa mu to hodí. Nakoniec je potrebné prepočítať vlastnosti novovytvoreného grafu.

2.3.4.3 CanvasView

CanvasView dedí od triedy *JPanel*, ktorý sa sám o sebe zobrazuje ako prázdna plocha, ak sú mu dané rozmery. Pomocou *override* metódy *paint()* sa prekresľuje grafické zobrazenie panela. Metóda *paint()* je potom volaná z funkcie *update()* zakaždým, keď sa dáta plátna zmenia.

Vnútri metódy *paint()* sa najskôr zavolá rodičovská metóda *paint()*, ktorá zaistí vykreslenie samotného panelu. Potom pomocou grafickej triedy *Graphics*, ktorej inštancia je daná parametrom, sa môžu do panela vykresľovať jednotlivé objekty. Ale pred tým, než dochádza k vykresľovaniu jednotlivých objektov, sa pretypovaním z *Graphics* urobí *Graphics2D*.

Vykresľovanie objektov na plátno prebieha v tomto poradí: neoznačené hrany, označené hrany, neoznačené uzly, označené uzly, spojovacie hrany a obdĺžnik pre výber. Postupnosť je dôležitá, pretože objekt vykreslený skôr je prekreslený objektom, ktorý je vykreslený neskôr. pre pochopenie tejto dôležitosti je potrebné povedať, ako sa dané objekty vykresľujú.

Hrana, ktorá je reprezentovaná triedou *EdgeModel*, sa vykresľuje ako čiara, ktorej koncovými bodmi sú stredy uzlov danej hrany. Pokiaľ sa jedná o označenú hranu, čiara bude mať väčšiu hrúbku. Popisok sa vykreslí vo vodorovnej polohe a jeho stred leží v strede hrany.

Uzol, ktorý je reprezentovaný triedou *NodeModel*, sa vykresľuje o niečo zložitejšie. Najskôr sa vykreslí kruh, ktorý má farbu podľa farebnosť uzla.

Potom sa vykreslí kružnica čiernej farby s rovnakým stredom a s rovnakým polomerom ako kruh, ktorá má slúžiť ako ohraničenie. Pokiaľ sa jedná o označený uzol, hrúbka čiary kružnice je väčšia. Nakoniec sa opäť vykreslí popisok vo vodorovnej polohe so stredom ležiacim v strede uzlu.

Spojovacie hrany sa vykresľujú, ak metóda *CanvasModelu isEdgeDragged()* vráti *true*. zo stredu každého označeného uzla povedie hrana do aktuálnej pozície kurzora. Tieto hrany nemajú popisky a vykresľujú sa ako označené hrany, ale ich čiary sú prerušované, aby sa odlíšili od ostatných.

2.3.5 MenuBar

V tejto sekcii sa budem venovať opisu lišty menu.

2.3.5.1 MenuBarController

MenuBarController sa stará o akcie spojené s *menuBarView*, respektíve ak používateľ klikne na nejakú možnosť z menu lišty, controller tento vstup odchytí a vykoná príslušnú akciu. Aby controller mohol odchytávať tieto akcie, opäť musí implementovať rozhranie *ActionListener* a s ním metódu *actionPerformed()*.

Vnútri metódy *actionPerformed()* sa porovnáva objekt, ktorý vyvolal túto metódu, s jednotlivými komponentmi (inštanciami triedy *JMenuItem*) *menubarView*. Toto porovnanie prebieha pomocou niekoľkonásobného *else-ifu* (ak prvá podmienka neplatí, vyskúša sa ďalšia podmienka, atď.), pretože *switch* sa nedá použiť na rovnosť objektov. Potom sa zavolá metóda z *CanvasModel*u, ktorá zodpovedá vybranej možnosti.

2.3.5.2 MenuBarView

Trieda *menuBarView* zobrazuje lištu s menu. *MenuBarView* dedí od *JMenuBar*, čo mu poskytuje veľkú funkčnosť, ako napr. nastaviť ho pomocou metódy *setJMenuBar()* ako lištu pre *ApllicationFrame* bez ďalších starostí. Jednotlivé menu sú potom inštanciami triedy *JMenu* a jednotlivé položky menu sú typu *JMenuItem*.

Menu, ktoré nás bude zaujímať je *View*, ktoré umožňuje približovať a oddaľovať plátno ako koliesko myši, zapnúť a vypnúť popisky hrán a uzlov (na to opäť slúži *JCheckBoxMenuItem*) a možnosť nechať si vycentrovať plátno na stred grafu, keby sa stalo , že by sa užívateľ stratil.

2.3.6 ConsoleView

ConsoleView sa stará o zobrazenie panela s *JEditorPane*, ktorý umožňuje formátovať text vnútri seba. Do *JEditorPane* sú vypísané uzly a hrany. Spolu s *PropertiesView* je súčasťou bočného panela.

2.3.7 PropertiesView

PropertiesView je postavené rovnako ako *ConsoleView* s tým rozdielom, že do *JEditorPanu* sa vypisujú informácie o grafe. Týmito informáciami sú počet uzlov, počet hrán, chromatické číslo, či je graf vykreslený planárne a či je graf planárny. Ak je graf prázdny, užívateľovi sa nezobrazia žiadne informácie.

2.3.8 FileService

FileService je trieda, ktorá obsahuje rôzne metódy pre uloženie a nahranie grafu zo súboru. Tieto metódy sa dajú rozdeliť do dvoch skupín - metódy, ktoré sa starajú o samotné uloženie a nahranie grafu v danom formáte a metódy, ktoré tvoria nadstavbu nad týmito metódami.

Kapitola 3

Návrh

V tejto kapitole sa budem venovať návrhu svojho rozšírenia do spomínaného GraphBrowseru.

3.1 Použitie knižníc na prácu s grafmi

Všetky knižnice spomenuté v sekcii 1.6 poskytujú algoritmy potrebné na získanie minimálnej kostry grafu. Jedine JGraphT však poskytuje funkcie na zistenie, či je graf Eulerovský. Keďže nájdenie Hamiltonovskej cesty alebo kružnice je NP-úplný problém a určenie, či je graf Hamiltonovský potrebujeme presne a nie heuristicky, ani jedna z týchto knižníc nespĺňa túto podmienku.

Napriek tomu, JGraphT ponúka najbohatšie možnosti na rozšírenie GraphBrowseru a to preto, lebo nemá presne daný formát grafu a jeho hrany aj uzly môžu byť nielen predom dané triedy ako *integer* v GMGraphLib a JUNG, alebo *DefaultEdge*. Môžeme teda použiť už existujúce triedy *EdgeModel* a *NodeModel* pre lepšiu prepojenosť grafu s touto knižnicou. Preto ju aj vo svojej práci použijem.

Keďže bolo potrebné vymyslieť aj to, akým spôsobom bude komunikovať knižnica s grafom, ktorý máme v GraphBrowseri vytvorený, rozhodol som sa pre jednoduchosť a tiež ľahkú manipuláciu zahrnúť reprezentáciu aktuálneho grafu priamo do triedy *GraphModel*.

3.1.1 Eulerovský graf

Ako zo zadania vyplýva, moje rozšírenie má zahŕňať aj zistenie, či je graf Eulerovský (vlastnosť opísaná v kapitole 1.3.3). Keďže knižnica JGraphT ponúka aj triedu *EulerianCircuit*, ktorá už obsahuje metódu na nájdenie uzavretého Eulerovského ťahu, rozhodol som sa ju použiť. Jej využitie bude pomocou novej triedy, ktorá bude poskytovať rozhranie pre komunikáciu *GraphModel*u a *EulerianCircuit*.

3.1.2 Hamiltonovský graf

Ďalšou vyžadovanou vlastnosťou je zistenie, či je graf Hamiltonovský (vlastnosť opísaná v sekcii 1.3.4). JGraphT síce obsahuje triedu *HamiltonianCycle*, ale táto trieda obsahuje len jednu metódu a to *getApproximateOptimalForCompleteGraph*. Keďže ani ostatné knižnice neposkytujú (respektíve ich integrácia by vyžadovala príliš veľké úsilie) triedy ani metódy na zistenie tejto vlastnosti, rozhodol som sa zistenie tejto vlastnosti naprogramovať sám.

Využijem pri tom algoritmus backtrackingu a podobne ako v sekcii 3.1.1, ho zahrniem do samostatnej triedy. Pôvodne som plánoval zahrnúť aj spomínanú metódu *getApproximateOptimalForCompleteGraph()* na nájdenie riešenia problému obchodného cestujúceho, ale tento problém je NP-úplný a daný algoritmus neposkytuje presné riešenie, ale len riešenie nájdené heuristicky, čiže suboptimálne. Za vstup používa úplný graf, v ktorom hľadá toto riešenie. Aj napriek tomuto faktu som predpokladal, že toto riešenie je dostatočne dobré a na grafoch, ktoré sme schopní s GraphBrowserom spracovávať.

Rozšíril som teda graf, ktorý práve v GraphBrowseri máme na úplný graf, použil metódu *getApproximateOptimalForCompleteGraph()* a následne zisťoval, že táto metóda mnohokrát označuje hrany, ktoré vznikli doplnením na úplný graf (pridané s váhou, ktorá im zaručuje, že vo výslednom riešení byť určite nemôžu). Rozumné riešenia som dostával len pri grafoch, ktoré mali menej ako štyri uzly. Namiesto toho som sa teda rozhodol označovať hrany, ktoré patria do Hamiltonovskej kružnice, ktorú nájdem backtrackingom.

3.1.3 Minimálna kostra

Posledným prvkom rozšírenia vyplývajúcim zo zadania je nájdenie minimálnej kostry grafu (podrobnejšie popísané v kapitole 1.3.5). JGraphT obsahuje aj rozhranie *MinimumSpanningTree*, ktoré implementujú triedy *KruskalMinimumSpanningTree*, používajúca Kruskalov algoritmus a *PrimMinimumSpanningTree*, používajúca algoritmus Jarník-Prim. Algoritmus Jarník-Prim však vyžaduje, aby bol graf, ktorého kostru hľadáme spojitý, tak som sa rozhodol pre Kruskalov algoritmus. Využitie triedy *KruskalMinimumSpanningTree* a jej komunikáciu s *GraphModel*om bude zabezpečovať ďalšia trieda.

3.2 Grafické rozhranie

Spomínané vlastnosti je potrebné zahrnúť do grafického rozhrania. Informáciu o tom, či je graf Hamiltonovský alebo Eulerovský je možné zachytiť pomocou pridania dvoch riadkov do stavového panelu. Minimálnu kostru a Hamiltonovskú kružnicu však musíme zahrnúť priamo do vykresľovania. Preto som uvážil, že hrany patriace do Hamiltonovskej kružnice budú zelené a hrany patriace do minimálnej kostry budú červené. Je však možné, že nastane situácia, keď

jedna hrana bude súčasťou ako kružnice tak aj kostry. Je preto potrebné pridať možnosť výberu, ktorý z týchto prvkov sa zobrazuje.

3.3 Testovanie

Testovanie bolo navrhnuté rôznymi spôsobmi. Keďže sa jedná o aplikáciu, ktorá vyžaduje interakciu s užívateľským rozhraním, rozdelil som toto testovanie do troch kategórií.

Prvou kategóriou je testovanie naivným užívateľom. To zase prebieha v troch krokoch.

1. Predloženie programu užívateľovi, ktorý nepozná ovládanie GraphBrowseru a požadovať od neho vytvorenie grafu.

2. Na už existujúcom grafe zobraziť váh hrán a zmeniť váhy špecifických hrán.

3. Zobrazenie minimálnej kostry, alebo Hamiltonovskej kružnice.

Druhou kategóriou testov je testovanie užívateľov informovaných o ovládaní GraphBrowseru a tiež jeho implementácii. Jedná sa o testovanie, ktoré by odhalilo prípadné chyby v implementácii či už mojej, alebo knižnici JGraphT.

Poslednou kategóriou navrhovaných testov sú záťažové testy, ktoré sa sústredia na rýchlosť programu pri grafe, ktorý obsahuje veľké množstvo uzlov a hrán (a zároveň aj skúmanie, koľko uzlov a hrán je „príliš veľké množstvo“).

Kapitola 4

Realizácia

V tejto kapitole je popísaná realizácia môjho rozšírenia GraphBrowseru podľa návrhu v kapitole 3.

4.1 GraphModel

Keďže mojou úlohou je vytvoriť rozšírenie, ktoré zahŕňa vlastnosti, na výpočet ktorých je treba pracovať s grafom, ktorého hrany sú ohodnotené. JGraphT poskytuje množstvo rôznych grafov, ktoré spája jednotné rozhranie *Graph<V,E>*. Keďže v GraphBrowseri sa nepredpokladá s multihranami ani slučkami a zároveň plánujeme implementáciu váh hrán, rozhodol som sa použiť existujúcu triedu *SimpleWeightedGraph<V,E>*. Využitie tejto triedy má však isté podmienky. *SimpleWeightedGraph<V,E>* požaduje určenie V,E. V je trieda pre uzly a E trieda pre hrany. Napriek tomu, že JGraphT poskytuje aj štandardné triedy ako *DefaultEdge* a *DefaultWeightedEdge*, rozhodol som sa namiesto nich použiť triedy vlastné. Ako uzly teda použijeme už existujúcu triedu *NodeModel*. Ako hrany ale nemôžeme použiť priamo *EdgeModel* bez úprav, pretože *SimpleWeightedGraph* vyžaduje, aby trieda pre hrany bola buď priamo *DefaultWeightedEdge*, alebo od nej dedila.

4.1.1 EdgeModel

Rozšírenie *EdgeModel* na *DefaultWeightedEdge* je veľmi prosté a to pridaním atribútu double *weight* a preťažením metódy *getWeight()*.

Máme teda graf *SimpleWeightedGraph<NodeModel,EdgeModel>*. Tento graf som pridal do už existujúcej triedy GraphModel z GraphBrowseru ako atribút *jgraphtRepresentation*. Tým pádom bolo potrebné upraviť aj pridávanie hrán a uzlov (metódy *addEdge()* a *addNode()*) tak, aby sa pridávaním do *edges* a *nodes* pridávali zároveň do *jgraphtRepresentation*. Keďže trieda *SimpleWeightedGraph<V,E>* zahŕňa metódy *addEdge()* a *addVertex()* pridanie daných prvkov je už triviálne.

Keďže je v GraphBrowseri aj možnosť hrany a uzly odstraňovať, je treba ich vymazávať aj z *jgraphtRepresentation*. Pôvodne sa o mazanie starala trieda *CanvasModel* metódou *deleteMarked()*, ktorá z grafu spojeného s týmto plátnom vymazala všetky aktuálne označené prvky. Ja som zodpovednosť mazania presunul do *GraphModel*u, pretože siahať takýmto spôsobom na jednotlivé prvky grafu by sa podľa môjho názoru nemalo a zároveň *jgraphtRepresentation* by mal byť určite private atribútom, takže by nastával problém ako mazať zároveň aj odtiaľ, aby nepracoval s neaktuálnymi dátami.

Takto som do *GraphModel* pridal metódu *deleteMarked()*, kde je ako parameter označená časť grafu, t.j. uzly a hrany ktoré majú byť vymazané. Táto metóda je sčasti pôvodná metóda z *CanvasModel* rozšírená o príkazy ktoré zaručia vymazanie aj z *jgraphRepresentation*.

4.2 SpanningTreeService

Trieda *SpanningTreeService* sa stará o nájdenie minimálnej kostry grafu. Knižnica JGraphT obsahuje aj triedu *KruskalMinimumSpanningTree*, ktorá, ako je aj z jej názvu poznať, používa Kruskalov algoritmus (spomenutý v sekcii 1.3.5) na vytvorenie minimálnej kostry. Táto trieda potrebuje ako parameter konštruktoru *SimpleWeightedGraph<V,E>*. Pomocou metódy *getEdgeSet()* poskytuje množinu hrán obsiahnutých v minimálnej kostre. Keďže však potrebujeme tieto hrany aj nejakým spôsobom označiť, že patria do tejto kostry pridal som do *EdgeModel* atribút boolean *spanningTreeIncluded*, ktorý spočiatku všetkým hranám nastavíme na false (ani jedna hrana nie je súčasťou kostry) a následne všetkým hranám z množiny *getEdgeSet()* nastavíme na true.

4.3 HamiltonianService

Táto trieda sa stará o zistenie, či je graf Hamiltonovský (vlastnosť popísaná v kapitole 1.3.4). Pomocou metódy *isHamiltonian()*, kde je ako parameter náš graf. Využíva metódu backtrackingu začínajúc metódou boolean *hasCycle()*, kde nastavíme počiatočný uzol a skúšame vytvoriť kružnicu cez všetky uzly metódou *addAnotherNode()*. Využívame tiež metódu *canBeAdded()*, ktorá skúma, či môže byť uzol pridaný (už sa v kružnici nachádza, neexistuje hrana z aktuálneho uzlu do uzlu nového). Pôvodne táto trieda tiež určovala, ktoré hrany patria do riešenia problému obchodného cestujúceho pomocou knižnice JGraphT.

4.4 EulerianCircuitService

Trieda *EulerianCircuitService* zabezpečuje zistenie, či je graf Eulerovský (vlastnosť popísaná v kapitole 1.3.3). Obsahuje metódu *isEulerian()*, kde je ako

parameter graf, ktorý na existenciu uzavretého Eulerovského ťahu testujeme. V tejto metóde je zase použitá knižnica JGraphT, špecificky jej trieda *EulerianCircuit* s metódou boolean *isEulerian()*. Tento zásah do GraphBrowseru taktiež vyžadoval pridanie atribútu boolean *eulerian* do triedy *GraphModel*.

4.5 FileService

GraphBrowser tiež ponúka možnosť uložiť a načítať graf v štyroch rôznych formátoch (spomenutých v sekcii 1.7)

- TGF - Hrany som rozšíril o váhu, ale do formátu TGF ju aj tak zahrnúť nemôžeme. Preto sa ukladanie grafu do TGF nezmení, ale pri načítavaní každej hrane dáme prednastavenú váhu 1,0.

- TGFC - Keďže TGFC nemá žiaden predom určený formát, zahrnúť tam váhu hrán nie je problém. Metóda *saveToTGFC()* využíva aj triedu EdgeModel a jej metódu *toString()* ako prednastavený formát pre uloženie hrany. Preto som do tejto metódy pridal vypísanie váhy hrany. Takto máme každú hranu uloženú nasledujúcim spôsobom:
  id počiatočného uzla id koncového uzla váha popisok
  Teraz už ostalo len upraviť metódu *openFromTGFC* a váhu naspäť načítať. Pre jednoduchosť som vytvoril pre *EdgeModel* druhý konštruktor, ktorý sa líši od pôvodného o to, že okrem uzlov a popisku ako parametrov má ako parameter aj váhu hrany.

- DOT disponuje aj atribútom weight, ktorý odpovedá váhe hrany. Uloženie teda vyžaduje jedine pridanie „weight k“, kde k je váha danej hrany. Načítanie teda bolo potrebné upraviť pridaním ďalšej možnosti do *switch*u vo funkcii *openFromGML()* a to po načítaní "weight"načítať zo súboru číslo vo formáte *double*, ktoré sa následne nastaví ak ováha hrany.

- GML taktiež ponúka bohaté možnosti zakódovania rôznych atribútov a medzi nimi zase aj weight. Postup je teda analogický ako pri ukladaní a načítaní DOT.

Keďže ostatné vlastnosti grafu vypočítame a zistíme z už uložených dát, netreba ukladať žiadne iné podrobnosti o grafe.

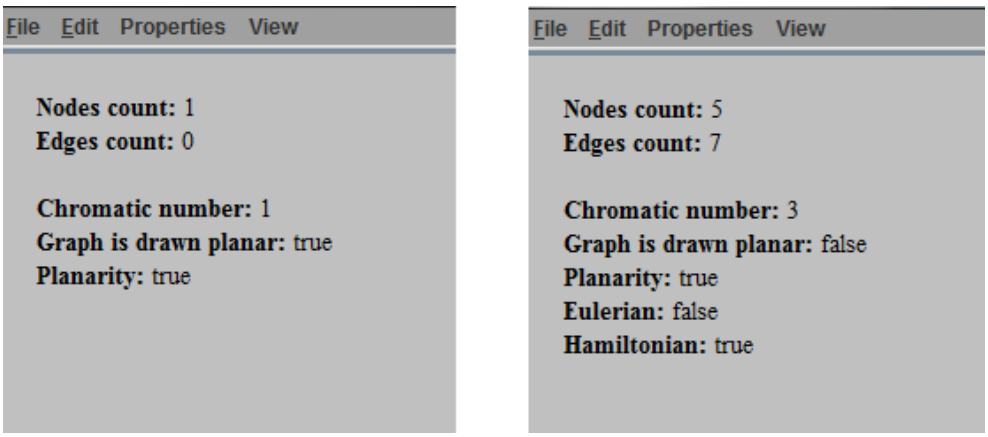
4.6 GUI

Implementované vlastnosti tiež treba nejako vhodne zahrnúť aj do GUI.

4.6.1 PropertiesView

Trieda *PropertiesView* sa stará o zobrazené informácie na postrannom paneli. Obsahuje informácie o počte hrán, uzlov, chromatické číslo a zároveň informáciu o tom, či je graf planárny. Táto informácia môže nadobúdať dve hodnoty a to true alebo false, podľa toho či planárny je alebo nie. Taktiež informácia o tom, či je graf Eulerovský alebo či je Hamiltonovský obsahujú len dve hodnoty a tak som sa rozhodol tieto informácie zahrnúť sem a to pridaním dvoch riadkov

- Eulerian: true/false
- Hamiltonian: true/false



Obr. 4.1: PropertiesView

4.6.2 MenuBar

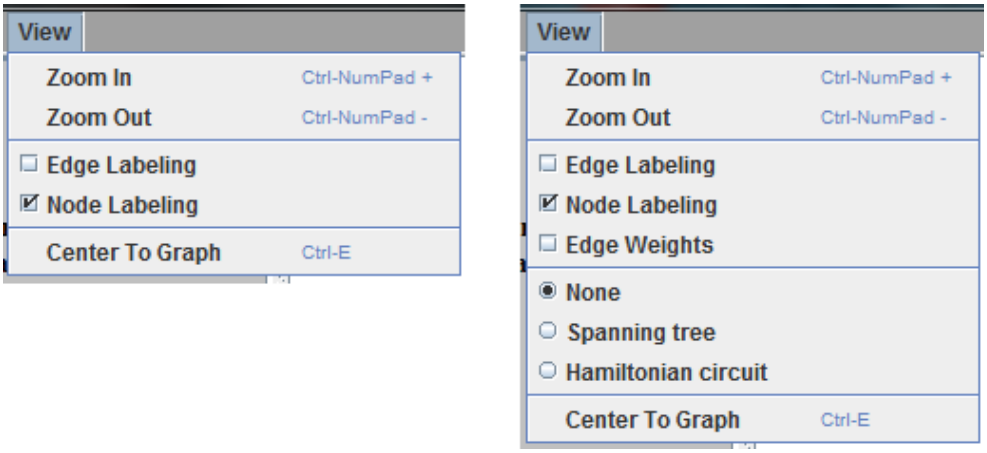
Niektoré implementované vlastnosti vyžadujú aj iné zásahy do užívateľského rozhrania. Ako je už existujúca možnosť skryť alebo zobraziť popisky uzlov a hrán, pridal som možnosť zobrazenia váhy hrán a to pridaním ďalšieho checkboxu do hlavného menu do kolonky view. Minimálnu kostru grafu by bolo vhodné nejako zakomponovať priamo do grafu. Najskôr je však potrebné vytvoriť možnosť výberu, či minimálnu kostru znázorniť. Pôvodne som plánoval možnosť zobrazenia ako minimálnej kostry, tak aj riešenia problému obchodného cestujúceho, na čo by obyčajné checkboxy nestačili, keďže niektoré hrany grafu môžu byť ako v minimálnej kostre, tak aj v riešení problému. Preto som sem zaviedol radiobox s troma možnosťami výberu:

1. None zobrazuje všetky hrany pôvodnou farbou

2. Spanning tree zobrazuje hrany patriace do minimálnej kostry grafu červenou farbou

3. Hamiltonian circuit zobrazuje hrany patriace do Hamiltonovskej kružnice.

Posledná vlastnosť nie je riešením problému obchodného cestujúceho, ale obyčajnou Hamiltonovskou kružnicou kvôli problémom s algoritmom z knižnice JGraphT (spomenuté v kapitole 4.3).



Obr. 4.2: ViewMenu

4.6.3 Canvas

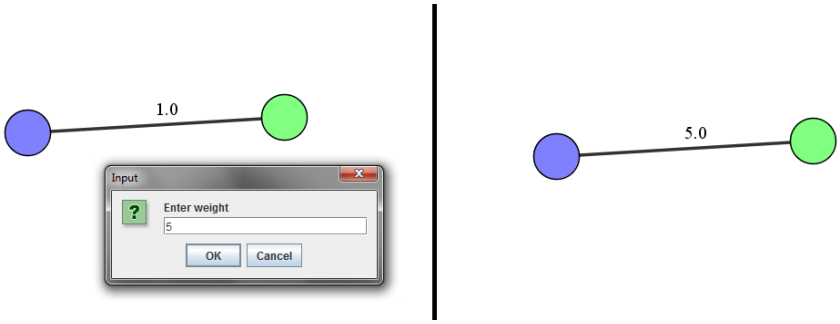
Zmeny sa samozrejme týkajú aj plátna.

4.6.3.1 CanvasController

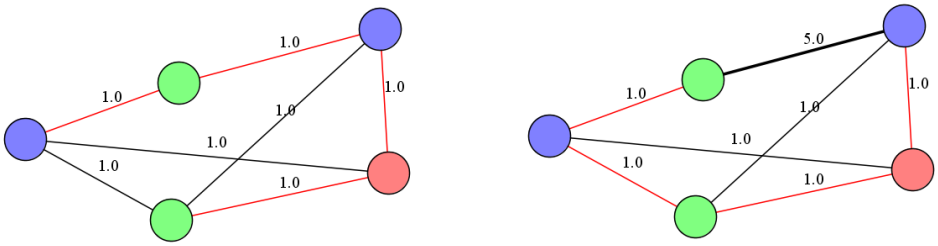
Je to trieda starajúca sa o plátno, jeho interakciu s užívateľom, rozpoznávanie jednotlivých akcií. Sú s ním spojené dve triedy a to *CanvasModel* a *CanvasView*, ktoré controller riadi na základe týchto akcií. Obsahuje metódu *leftMouseButtonClicked(MouseEvent e)*, ktorá rozpoznáva užívateľom vykonané akcie zahŕňajúce klik ľavým tlačítkom myši. Keďže som do editoru pridal váhu hrany, je taktiež potrebné vytvoriť možnosť túto váhu upraviť, inak by strácala zmysel. Rozhodol som sa preto, že váhu upravíme dvojklikom na hranu, ktorej váhu upraviť chceme (obrázok 4.3). Táto akcia môže mať za následok zmenu minimálnej kostry grafu, tak bolo tiež potrebné preskúmať túto kostru (ukázané na obrázku 4.4).

4.6.3.2 CanvasModel

CanvasModel predovšetkým ukladá informácie o plátne. Obsahuje informácie napríklad o tom, či zobrazovať popisky hrán alebo uzlov. Preto som sem pridal aj informácie o tom, či zobrazovať váhy hrán, minimálnu kostru aj Hamiltonovskú kružnicu (ak existuje) a to v podobe boolean atribútov.



Obr. 4.3: Zmena váhy hrany



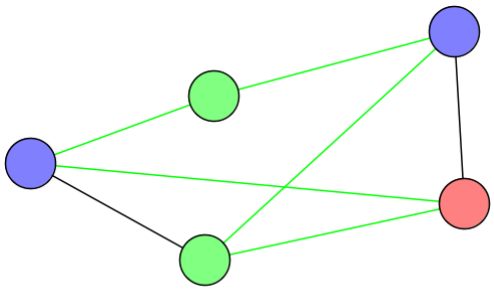
Obr. 4.4: Zmena minimálnej kostry

4.6.3.3 CanvasView

Tieto zmeny zasiahnu aj *CanvasView* a konkrétne metódu *paintEdge*. Treba ju rozšíriť o tri vlastnosti. Vykreslenie na zeleno, vykreslenie na červeno a možnosť vykresliť hrane váhu. Keďže sme triedu *EdgeModel* upravili tak, aby niesla všetky tieto potrebné informácie a zároveň vieme, s ktorým *CanvasModel*om je tento view spojený, vieme teda zistiť aj to, ktorú možnosť máme v radioboxe (informácia uložená v *CanvasModel* ako boolean atribút) zaškrtnutú a tým pádom aj ktorú farbu použiť pri vykresľovaní hrany patriacej aj do kružnice, aj do minimálnej kostry a tiež či máme vykresliť aj váhu hrany. Obrázok 4.5 ukazuje zobrazenie Hamiltonovskej kružnice.

4.6.4 GenerateService

Miernou zmenou musela podstúpiť aj táto trieda. Keďže *GraphModel* nemá žiadnu kontrolu pri pridávaní hrany, či sa niektorý z krajných uzlov sa reálne v grafe nachádza. Takáto možnosť pri tvorení grafu klikaním nemá šancu nastať a ani pri generovaní grafu by nemala byť problematická, keďže tam daný uzol aj tak nakoniec pridáme. *SimpleWeightedGraph* však dedí takúto kontrolu a hranu môžeme pridať len medzi uzly, ktoré už v grafe sú. Bolo preto potrebné prispôsobiť poradie pridávania tejto kontrole.



Obr. 4.5: Hamiltonovská kružnica

4.7 Testovanie

Aplikácia bola pôvodne vyvíjaná v Jave verzia 1.8, ale keďže nevyužívam žiadne prvky, ktoré sú novozavedené v tejto verzii a spätná kompatibilita (spúšťanie programu vyvíjaného s JDK 1.8 pomocou JDK 1.7 alebo nižšieho) nie je na dostatočnej úrovni, respektíve na žiadnej úrovni, rozhodol som sa aj pri vývoji použiť JDK 1.7. Testovanie teda tiež prebiehalo v Jave verzia 1.7 a testy sú rozvrhnuté tak, ako bolo navrhnuté v sekcii 3.3.

4.7.1 Testovanie korektnosti algoritmov

Toto testovanie zase prebiehalo prvotným vytvorením grafu a následným vymazávaním, generovaním častí grafu, prekresľovaním a zmenou váh hrán. Toto testovanie odhalilo nedostatky v implementácii JGraphT, konkrétne v triede *HamiltonianCycle*, čo malo za následok zmenu zobrazenia riešenia problému obchodného cestujúceho na obyčajnú Hamiltonovskú kružnicu. Inak testovanie prebiehalo hladko a bez problémov.

4.7.2 Záťažové testovanie

Keďže sa plátno prekresľuje niekedy aj niekoľkokrát za sekundu, pri veľkých grafoch nastávajú problémy s rýchlosťou, pretože vykreslenie (spomenuté v sekcii 2.3.4.3) prechádza všetky hrany aj uzly.Tieto problémy nastávajú až pri grafoch obsahujúcich nad 100 uzlov. Keďže predpokladané využitie GraphBrowseru aj s jeho rozšírením je na vzdelávacie účely, alebo na overovanie vlastností grafov pre nás zaujímavých a grafy nad 100 uzlov už rozhodne nespadajú ani do jednej z týchto kategórií, môžeme povedať, že aplikácia spĺňa účely, či už rýchlostne, alebo funkčne.

4.7.3 Užívateľské testovanie

Testovanie prebiehalo štyrmi testermi a testovacími operačnými systémami boli Windows 7, Ubuntu 14.10, Ubuntu 14.04. Notebook s Windows 7 a Ubuntu 14.10 má 4 GB RAM a Intel procesor rady i7 s dvoma jadrami. Ďalší notebook s Windows 7 má 10 GB RAM a Intel procesor rady i7 so štyrmi jadrami. Posledný notebook, s operačným systémom Ubuntu 14.04, na ktorom bola aplikácia testovaná má 8 GB RAM, Intel procesor rady i7 so štyrmi jadrami.
Navrhnutý užívateľský test sa skladal z troch úloh pre testerov:

1. Vytvorenie grafu
2. Zobrazenie váh hrán a zmeniť váhu špecifickej hrany
3. Zobrazenie minimálnej kostry alebo Hamiltonovskej kružince

4.7.4 Vyhodnotenie a dotazník

Po dokončení testovania boli testerom položené 4 otázky.

- **Podarilo sa Vám splniť všetky úlohy?**
  Každý užívateľ dokázal vytvoriť uzly, keďže táto úloha vyžadovala iba kliknutie na plátno. U väčšiny však následne nastali problémy s pridávaním hrán, ktoré sa pokúšali pridávať kliknutím pravého tlačítka myši na uzol, alebo kliknutím ľavého tlačítka myši na uzol a následným ťahaním. So zobrazením hrán nemal žiadny užívateľ problém, ale následná zmena váhy hrany už zase spôsobovala problémy. Väčšine testerov sa však úspešne podarilo zmeniť váhu.

  Záver: Z tohoto testovania vyplynul záver, že ovládanie plátna síce spĺňa svoj účel, ale zároveň by sa malo stať viac intuitívnym. Na zmenu ovládania však nebol dostatok času, túto zmenu v budúcnosti plánujem uskutočniť.

- **Čo sa Vám na aplikácii páčilo?**
  Všetkým testerom sa páčila grafická stránka aplikácie, vyššia funkčnosť aj zaujímavé zakomponovanie daných vlastností do grafu a grafického rozhrania.
- **Čo sa Vám na aplikácii nepáčilo?**
  Jedinou nevýhodou, ktorú testeri označili, bola neintuitívnosť ovládania. Inak žiadny tester nemal výhrady.
- **Čo by ste na aplikácii vylepšili alebo zmenili?**
  Okrem ovládania boli testeri spokojní so všetkým. Navrhované zmeny zahŕňali implementáciu ďalších vlastností a algoritmov.

4.8 Popis splnenia zadania

Nasledujúca tabuľka 4.6 ukazuje splnenie základných podmienok.

Analyzujte kód GraphBrowseru	
✓	Bod je splnený keďže analýza je popísaná v sekcii 2.3
Navrhnite jeho rozšírenie	
✓	Návrh spomínaného rozšírenia v kapitole 3

Obr. 4.6: Splnenie základných požiadaviek

Nasledujúca tabuľka 4.7 ukazuje splnenie algoritmických podmienok a podmienok na zisťovanie ďalších vlastností.

Rozšírte ho o zistenie, či je graf Eulerovský	
✓	Teoretická časť v sekcii 1.3.3, návrhová časť v sekcii 3.1.1, realizácia v kapitole 4.4
Zistenie, či je graf Hamiltonovský	
✓	Teoretická časť problému v sekcii 1.3.4, návrh v kapitole 3.1.2,realizácia v sekcii 4.3
Pridanie nájdenia minimálnej kostry grafu	
✓	Teoretická časť rozobraná v sekcii 1.3.5, návrhová časť v kapitole 3.1.3 a realizácia v sekcii 4.2
Vhodné zakomponovanie spomenutých vlastností do grafického rozhrania	
✓	Návrh v sekcii 3.2, realizácia opísaná v kapitole 4.6

Obr. 4.7: Splnenie požiadaviek na rozšírenie

Z vyššieuvedených tabuliek vidno, že všetky podmienky pre túto prácu boli splnené.

Záver

Cieľom mojej práce bolo vytvoriť rozšírenie pre GraphBrowser, ktoré mu dovolí zisťovať ďalšie vlastnosti grafu, ktorý sme si pomocou neho vytvorili, či už vygenerovaním, alebo vlastnoručným naklikaním jednotlivých prvkov grafu, ale aj ukladať grafy do rôznych formátov a opätovne ich načítavať. Toto rozšírenie zahŕňa pridanie váhy hrán a samozrejme aj možnosť váhy upravovať.

Pridanie váhy hrán ponúklo možnosť pridať nájdenie minimálnej kostry a jej následné vykreslenie červenou farbou. Ďalej je schopné z daného grafu zistiť, či obsahuje Hamiltonovskú kružnicu a tým pádom, či je graf Hamiltonovský a zároveň možnosť vykreslenia tejto kružnice zelenou farbou. Ďalšou možnosťou, ktorú moje rozšírenie poskytuje, je zistenie, či graf obsahuje uzavretý Eulerovský ťah, teda či je náš graf Eulerovský. Vytýčený cieľ som úspešne splnil.

Počas implementácie som narážal na množstvo problémov, ale zároveň som objavoval rôzne možnosti ďalšieho rozšírenia. Keďže ale nebol dostatok času na zahrnutie týchto rozšírení, rozhodol som sa ich implementovať v budúcnosti, či už vo voľnom čase, alebo ako súčasť diplomovej práce. Medzi tieto rozšírenia patria väčšinou ďalšie vlastnosti ako napríklad nájdenie dominujúcej a nezávislej podmnožiny a iné, ale aj zásahy do užívateľského rozhrania, ako upravenie ovládania (keďže pri testovaní aplikácie naivným užívateľom súčasné ovládanie nebolo dostatočne intuitívne).

S výsledkom svojej práce aj práce Tomáša Ondreja som spokojný, z GraphBrowseru sa stala aplikácia, ktorá je čím ďalej tým bližšie ku komplexnosti potrebnej na výuku grafových algoritmov, alebo aj overenie vlastností grafov pre nás zaujímavých.

Literatúra

[1] Šišma, P.: *Teorie grafů 1736-1963*. Praha: Prometheus, první vydání, 1997, ISBN 80-719-6065-9.

[2] Ondrej, T.: *Ověření vlastností grafu*. Bakalářská práce, České vysoké učení technické v Praze, Katedra teoretické informatiky, Praha, 2014.

[3] Arora, S.; Barak, B.: *Computational Complexity: A Modern Approach*. New York, NY, USA: Cambridge University Press, první vydání, 2009, ISBN 0521424267, 9780521424264.

[4] Alsuwaiyel, M. H.: *Algorithms: Design Techniques and Analysis*. World Scientific Publishing Company, 1999, ISBN 9810237405.

[5] van Leeuwen, J.: *Handbook of Theoretical Computer Science. Vol. A, Algorithms and complexity*. Amsterdam: Elsevier, 1994, ISBN 0262720140.

[6] Biggs, N.; Lloyd, E. K.; Wilson, R. J.: *Graph Theory, 1736-1936*. New York, NY, USA: Clarendon Press, 1986, ISBN 0-198-53916-9.

[7] Večerka, A.: Grafy a grafové algoritmy. 2007. Dostupné z: `http://phoenix.inf.upol.cz/esf/ucebni/Grafy_a_grafove_algoritmy.pdf`

[8] Hyblerová, B.: *Algoritmy pro barvení grafů*. Diplomová práce, Západočeská univerzita, Katedra matematiky, Plzeň, 2010.

[9] Hordějčuk, V.: Algoritmus Welsh-Powell. [online], [cit. 2014-04-21]. Dostupné z: `http://voho.cz/wiki/informatika/algoritmus/graf/welsh-powell/`

[10] Boyer, J. M.; Myrvold, W. J.: On the Cutting Edge: Simplified O(n) Planarity by Edge Addition. *Journal of Graph Algorithms and Applications*, ročník 8, 2004: s. 241—-273.

[11] de Fraysseix, H.: Trémaux Trees and Planarity. *Eur. J. Comb.*, ročník 33, č. 3, duben 2012: s. 279–293, ISSN 0195-6698.

[12] Hopcroft, J.; Tarjan, R.: Efficient Planarity Testing. *J. ACM*, ročník 21, č. 4, říjen 1974: s. 549–568, ISSN 0004-5411.

[13] Lempel, A.; Even, S.; Cederbaum, I.: An algorithm for planarity testing of graphs. *Theory of Graphs*, 1967: s. 215—-232.

[14] Kohnert, A.: Algorithm of Demoucron, Malgrange, Pertuiset. prosinec 2004. Dostupné z: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.2680&rep=rep1&type=pdf`

[15] Patrignani, M.: Planarity Testing and Embedding. 2004. Dostupné z: `http://www.win.tue.nl/~nikhil/courses/2012/2WO08/planarity.pdf`

[16] Mallows, C. L.; Sloane, N. J. A.: Two-Graphs, Switching Classes and Euler Graphs are Equal in Number. *SIAM Journal on Applied Mathematics*, ročník 28, č. 4, 1975: s. 876–880, doi:10.1137/0128070.

[17] Fleischner, H.: X.1 Algorithms for Eulerian Trails. *Eulerian Graphs and Related Topics: Part 1 (Annals of Discrete Mathematics)*, ročník 2, 1991: s. 1–13.

[18] Michael R. Garey, D. S. J.: *Computers and intractability : a guide to the theory of NP-completeness*. W.H. Freeman, 1979, ISBN 0-7167-1045-5, 199–200 s.

[19] Ore, O.: Note on Hamilton Circuits. *The American Mathematical Monthly*, ročník 67, č. 1, január 1960.

[20] Dirac, G. A.: Some Theorems on Abstract Graphs. ročník s3-2, č. 1, 1952: s. 69–81, doi:10.1112/plms/s3-2.1.69.

[21] Rubin, F.: A Search Procedure for Hamilton Paths and Circuits. *Journal of the ACM*, ročník 21, č. 4, október 1974: s. 576–580.

[22] Bellman, R.: Dynamic Programming Treatment of the Travelling Salesman Problem. *Journal of the ACM*, ročník 9, č. 1, január 1962: s. 61–63.

[23] Borůvka, O.: O jistém problému minimálním. *Práce Moravské přírodovědecké společnosti*, ročník 3, 1926: s. 37–58.

[24] Borůvka, O.: Příspěvek k řešení otázky ekonomické stavby elektrovodních sítí. *Elektronický Obzor*, ročník 15, 1926: s. 153–154.

[25] Choquet, G.: Étude de certains réseaux de routes. *Comptes-rendus de l'Académie des Sciences*, ročník 206, 1938: s. 310–313.

[26] Florek, K.: Sur la liaison et la division des points d'un ensemble fini. *Colloquium Mathematicum*, ročník 2, 1951: s. 282–285.

[27] Sollin, M.: Le tracé de canalisation. *Programming, Games, and Transportation Networks*, 1965.

[28] Jarník, V.: O jistém problému minimálním. *Práce Moravské Přírodovědecké Společnosti*, ročník 6, 1930: s. 57–63.

[29] Prim, R. C.: Shortest connection networks and some generalizations. *Bell System Technical Journal*, ročník 36, 1957: s. 1389–1401.

[30] Kruskal, J. B.: On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, ročník 7, 1956: s. 48–50.

[31] Trudeau, R. J.: *Introduction to Graph Theory*. New York: Dover Pub, 1993, ISBN 978-0-486-67870-2.

[32] Oracle Corp.: Java. [online]. Dostupné z: `http://www.oracle.com/cz/technologies/java/features`

[33] Eckstein, R.: Java SE Application Design With MVC. *Oracle Technology Network*, březen 1977. Dostupné z: `http://www.oracle.com/technetwork/articles/javase/index-142890.html`

[34] Naveh, B.: JGraphT. [online]. Dostupné z: `http://jgrapht.org/`

[35] JUNG Framework Development Team: JUNG. [online]. Dostupné z: `http://jung.sourceforge.net/`

[36] gabrielgmendonca: gmgraphlib. [online]. Dostupné z: `https://www.openhub.net/p/gmgraphlib`

Dodatok **A**

Zoznam použitých skratiek

DOT Graph Description Language

GML Graph Modeling Language

GUI Graphical user interface

IDE Integrated Development Environment

JDK Java Development Kit

JRE Java Runtime Environment

JVM Java Virtual Machine

MVC Model-View-Controller

TGF Trivial Graph Format

TGFC Trivial Graph Format with Coordinates

XML Extensible markup language

Dodatok **B**

Obsah priloženého CD

- `Assignment.pdf` ..... zadanie práce
- `readme.txt` ..... stručný popis obsahu CD
- `Project` ..... adresár so zdrojovými kódmi aplikácie
- `GraphBrowser.jar` ..... spustiteľný súbor aplikácie
- `Thesis`
  - `BP_Toth_Matus_2015.pdf` ..... text práce vo formáte PDF
  - `BP_Toth_Matus_2015.tex` ..... text práce vo formáte LaTeX