VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

# DEMONSTRACE GRAFOVÝCH ALGORITMŮ

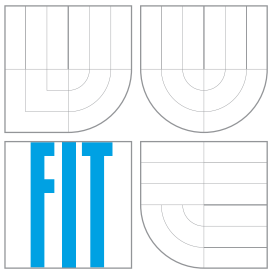
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAKUB VARADINEK

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

# DEMONSTRACE GRAFOVÝCH ALGORITMŮ

DEMONSTRATION OF GRAPH ALGORITHMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JAKUB VARADINEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. ZBYNĚK KŘIVKA, Ph.D.
SUPERVISOR

BRNO 2013

# Abstrakt

Tato bakalářská práce se zabývá vývojem aplikace pro demonstraci a vizualizaci některých grafových algoritmů. Aplikace uživateli umožňuje vytvořit graf, ohodnotit hrany nebo pojmenovat a rozmístit vrcholy. Nad takto vytvořeným grafem je možné nechat provádět jednotlivé algoritmy a vizuálně sledovat, jak algoritmus pracuje. K dispozici je taktéž možnost krokování algoritmu a interaktivní režim, kdy postup volí uživatel a aplikace provádí kontrolu správnosti zvolených kroků.



# Abstract

This bachelor thesis deals with the development of the application for demonstration and visualization of several graph algorithms. The application allows the user to create a graph, rate edges or name and layout vertices. The individual algorithms can be performed in created graph for visual observation how the algorithm works. There is also the possibility of stepping through the chosen algorithm and an interactive mode where next steps are selected by the user and the application checks the correctness of these steps.



# Klíčová slova

grafy, grafové algoritmy, DFS, BFS, Bellmanův-Fordův algoritmus, Dijkstrův algoritmus, algoritmus topologického uspořádání, JGraphX

# Keywords

Graphs, Graph Algorithms, DFS, BFS, Bellman-Ford algorithm, Dijkstra's algorithm, Topological sort, JGraphX

# Citace

Jakub Varadinek: Demonstrace grafových algoritmů, bakalářská práce, Brno, FIT VUT v Brně, 2013

# Demonstrace grafových algoritmů

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Zbyňka Křivky, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jakub Varadinek
13. května 2013

## Poděkování

Rád bych poděkoval vedoucímu mé práce Ing. Zbyňku Křivkovi, Ph.D. za cenné rady, konzultace a náměty pro další vylepšení aplikace.

# Obsah


**1 Úvod** **3**
1.1 Struktura práce . . . 4

**2 Analýza požadavků a volba grafových algoritmů** **5**
2.1 Existující nástroje . . . 5
2.2 Technické požadavky a omezení . . . 6
2.3 Volba algoritmů pro demonstraci . . . 6
2.4 Možnosti pro uživatele . . . 7

**3 Návrh aplikace** **8**
3.1 Volba knihovny pro práci s grafy . . . 8
3.1.1 Základní kritéria . . . 8
3.1.2 Možné varianty . . . 9
3.2 Popis knihovny JGraphX . . . 9
3.2.1 Reprezentace a vizualizace grafů . . . 10
3.2.2 Ukládání a načítání grafů . . . 10
3.3 Obecný popis simulátoru . . . 11
3.3.1 Vizuální stránka simulátoru . . . 11
3.3.2 Ovládání simulátoru uživatelem . . . 11
3.4 Interakce uživatele . . . 12
3.4.1 Prohledávání do šířky . . . 12
3.4.2 Prohledávání do hloubky . . . 12
3.4.3 Topologické uspořádání uzlů . . . 13
3.4.4 Hledání silně souvislých komponent . . . 13
3.4.5 Dijkstrův algoritmus . . . 14
3.4.6 Bellmanův-Fordův algoritmus . . . 14
3.5 Editor grafů . . . 14
3.6 Shrnutí návrhové části . . . 15

**4 Implementace aplikace** **16**
4.1 Uživatelské rozhraní editoru grafů . . . 16
4.2 Možnosti v rozhraní editoru . . . 17
4.3 Implementace editoru grafů . . . 18
4.4 Uživatelské rozhraní simulátoru . . . 19
4.4.1 Panel s pseudokódem . . . 20
4.4.2 Vizualizační panel . . . 20
4.4.3 Ostatní panely . . . 21
4.5 Reprezentace algoritmů v aplikaci . . . 21

4.5.1 Pseudokód algoritmu . . . 21
4.5.2 Instrukce algoritmu . . . 22
4.5.3 Uživatelské objekty v grafu . . . 22

**5 Řídicí jednotka simulátoru** **25**
5.1 Implementace řídicí jednotky . . . 25
5.2 Instrukce s řízením interaktivity . . . 27
5.3 Realizace krokování . . . 27
5.3.1 Výhody a omezení krokování . . . 28

**6 Testování aplikace** **29**
6.1 Testování jednotlivých algoritmů . . . 29
6.1.1 Testování algoritmu BFS . . . 29
6.1.2 Testování algoritmu DFS . . . 31
6.1.3 Testování algoritmu SCC . . . 32
6.1.4 Testování algoritmu topologického uspořádání . . . 33
6.1.5 Testování Bellman-Fordova algoritmu . . . 34
6.1.6 Testování Dijkstrova algoritmu . . . 36
6.2 Testování interakce . . . 37
6.3 Zhodnocení aplikace . . . 37

**7 Závěr** **39**

**A Obsah CD** **41**

**B Manuál** **42**

**C Pseudokódy algoritmů** **45**

**D Styly pro graf** **49**

# Kapitola 1

# Úvod

V této bakalářské práci budou popsány grafové algoritmy a vývoj aplikace, která demonstruje jejich činnost. Cílem je, aby výsledný program byl co nejvíce vizuálně názorný a poskytoval přehledně veškeré důležité informace a obsahy proměnných daného algoritmu. Uživateli jsou umožněny základní akce, jako je běh se zvolenou rychlostí, krokování (dopředu i dozadu), restart aktuální simulace a možnost zadání bodů, kde se má simulace zastavit. Pro každý krok (případně sérii kroků) aplikace zobrazuje jejich popis v textové podobě. Celý program je realizován v jazyce Java za použití knihovny **JGraphX** pro práci s grafy. Uživatelské rozhraní je realizováno pomocí standardní knihovny Swing.

Při výběru algoritmů se vycházelo z předmětu Grafové algoritmy (dále pod zkratkou **GAL**), který je vyučován v magisterském studiu na FIT VUT. Protože program je určen právě pro účely předmětu **GAL**, tak i pseudokódy algoritmů jsou převzaty z materiálů tohoto předmětu [1]. Pro tuto práci byly zvoleny algoritmy prohledávání stavového prostoru, **prohledávání do šířky** i **prohledávání do hloubky**. Dále to je algoritmus **topologického uspořádání** a algoritmus pro **hledání silně souvislých komponent**. Druhou skupinou jsou algoritmy pro hledání nejkratší cesty z jednoho uzlu do všech ostatních uzlů. Z této skupiny byly vybrány **Dijkstrův** a **Bellman-Fordův** algoritmus, které se liší ve schopnosti zpracování grafu, ve kterém se vyskytují záporně ohodnocené hrany.

Kromě standardní vizualizace průchodu algoritmu, program také nabízí interaktivní mód. V tomto módu je v klíčových bodech při provádění algoritmu uživateli předáno řízení, aby sám doplnil, jak bude algoritmus pokračovat. Program kontroluje, zda uživatel provedl tyto akce správně. V určitých případech (např. při vytváření seznamu sousedů) existuje větší množství správných variant, lišící se pouze pořadím zpracování. Standardně by se algoritmus řídil určitou vnitřní reprezentací grafu, zde ovšem uživatel má přímou možnost ovlivnit toto pořadí podle sebe. Tak může docílit jiného průchodu grafem, případně dosáhnout řešení rychleji, např. u **Bellman-Fordova** algoritmu vhodným výběrem pořadí relaxace hran.

Aplikace pro vizualizaci algoritmů jsou přístupné na internetu ve velkém počtu. Často však mají určitá omezení v podobě použití pouze předpřipraveného vstupu, případně neumožňují provádět krokování či detailně sledovat obsahy jednotlivých proměnných. Hlavní vlastnost, kterou se výsledná aplikace z této bakalářské práce odlišuje, je interaktivní mód popsaný v předchozím odstavci. Toto rozšíření ke standardní simulační části poskytuje nové možnosti, díky kterým si uživatel může přímo ověřit, zda správně porozumněl algoritmu.

## 1.1 Struktura práce

Práce je členěna do jednotlivých kapitol v pořadí stejném, jako probíhal i vývoj aplikace. Na začátku byla provedena základní analýza požadavků na aplikaci, volba algoritmů a hledání podobných existujících nástrojů. Provedená analýza je popsána v kapitole 2. Návrhem aplikace, volbou knihovny pro práci s grafy a soupisu základních podporovaných funkcí se zabývá kapitola 3. V této kapitole je také popsáno, jak bude probíhat interakce u jednotlivých algoritmů.

Další kapitoly jsou zaměřeny více prakticky. Implementace aplikace je popsána v kapitole 4, kde jsou k nalezení i snímky uživatelského rozhraní. Kapitola 5 popisuje hlavní část simulátoru – řídicí jednotku. V kapitole 6 se testuje deset různých grafů.

# Kapitola 2

# Analýza požadavků a volba grafových algoritmů

Tato kapitola rozebírá požadavky kladené na výslednou aplikaci. Zejména se jedná o možnosti, které má uživatel při používání aplikace, a o technické požadavky pro její používání. Je zde obecný popis existujících aplikací, které poskytují stejnou nebo podobnou funkcionalitu. Cílem této kapitoly je určit základní vlastnosti, vybrat vhodné algoritmy ke zpracování a předem odhadnout možné technické problémy a omezení. Zároveň v rámci analýzy jsem vyhledal nástroje podobného zaměření. Žádný z těchto nástrojů neobsahoval prvky interaktivity tak, jak požaduje zadání této práce.

## 2.1 Existující nástroje

Hledání existujících nástrojů je první fáze při vypracovávání této práce. Cílem bylo nalezení nástrojů, které poskytují stejné možnosti jako aplikace vytvářená v této práci. Nalezl jsem velké množství programů převážně ve formě Java Appletů a v dalších technologiích zaměřených na web. Sledované parametry aplikací byly následující:

- Možnost zadání uživatelem vytvořeného grafu.
- Podporované algoritmy.
- Způsob jakým je algoritmus demonstrován.
- Možnosti v ovládání, podpora uživatelské interakce. Interakcí je zde myšlena možnost uživatelem aktivně určovat další kroky při provádění algoritmu a zpětná kontrola od aplikace, zda uživatel postupuje správně.
- Typ aplikace (webová nebo klasická) a podporovaný operační systém.

Nalezené aplikace byly v nabízených možnostech od velmi jednoduchých, které obsahovaly pouze základní prvky, až po pokročilé. Část byla velmi úzce zaměřená pouze na jeden konkrétní algoritmus a předpřipravené grafy, ale s maximálním důrazem na vizuální efekty. Několik nalezených aplikací splňovalo základní požadavky tak, jak jsou definovány pro aplikaci vytvářenou v rámci této práce, kromě požadavku interaktivity. Program, který by umožňoval interaktivitu a zároveň podporoval uživatelem vytvořené grafy, jsem nenalezl.

Jednoduchou aplikaci provádějící algoritmus prohledávání do šířky je možné vidět na webové stránce[1]. Tato aplikace neposkytuje žádné možnosti v úpravě ani ve vytvoření grafu, nezobrazuje pseudokód a umožňuje uživateli pouze vybrat počáteční a hledaný uzel. Stejný algoritmus provádí i pokročilejší aplikace na stránce[2]. Druhá aplikace má již propracované ovládání algoritmu a dovoluje vytvořit si vlastní graf. Pro prezentaci algoritmu **topologického uspořádání** je vytvořena aplikace např. na stránce[3]. Neposkytuje sice mnoho možností, ale je více zaměřena na samotnou vizualizaci. Graf je zde možné zobrazit i v podobě seznamů sousedů a matice sousednosti.

Jednu z nejvíce propracovaných aplikací je možné naleznout na stránce[4]. Má implementované všechny možnosti kromě interaktivity. Uživatel si zde může vytvořit vlastní graf, při simulaci je zobrazen pseudokód a aplikace poskytuje možnosti krokování. Je také podporováno více algoritmů a jejich vizualizace je propracovaná. Za výhodu je možné označit českou lokalizaci aplikace.

## 2.2 Technické požadavky a omezení

Při návrhu bylo nutné vzít v potaz prostředí, kde bude aplikace nasazena, a některá omezení z toho plynoucí. Požadavek také je, aby se aplikace dala zobrazit na projektoru, který má nižší rozlišení. Tomu bylo nutné uzpůsobit i rozložení uživatelského rozhraní. Zásadní je využití komponent, které umožňují, aby uživatel mohl sám v uživatelském rozhraní u hlavních prvků určit, kolik mají zabírat místa.

K dalším vlastnostem, které by měla výsledná aplikace splňovat, patří možnost spuštění více instancí simulátoru soubežně. Jednotlivé instance se budou spouštět z hlavní části programu. Každá taková instance může mít jiný vstupní graf a provádět jiný algoritmus.

## 2.3 Volba algoritmů pro demonstraci

Volba algoritmů probíhala ve spolupráci s vedoucím práce. Cílem bylo vybrat několik algoritmů, které se vyučují v předmětu **GAL** a jsou vhodné pro vizuální demonstraci s uživatelskou interakcí. Algoritmy mohly být z oblastí např. prohledávání stavového prostoru, barvení grafů, toky v síti, Eulerovské tahy a algoritmy pro hledání nejkratších cest z jednoho uzlu do všech ostatních uzlů. Vybrané algoritmy je možné nalézt v následujícím seznamu.

- **prohledávání do šířky** (dále pod zkratkou **BFS** z anglického breadth-first search)
- **prohledávání do hloubky** (dále pod zkratkou **DFS** z anglického depth-first search)
- **hledání silně souvislých komponent** (dále pod zkratkou **SCC** z anglického strongly connected component)
- **Topologické uspořádání**
- **Bellman-Fordův algoritmus** a **Dijkstrův algoritmus** pro hledání nejkratších cest ze zadaného zdrojového vrcholu.

[1] http://www.javacoffeebreak.com/tutorials/aisearch/chapter6.html
[2] http://www.lupinho.net/lupinho.de/gishur/html/BFSApplet.html
[3] http://www.cs.usfca.edu/ galles/visualization/TopoSortDFS.html
[4] http://pavel.roblandservis.cz/bpfinal/plain/applet.html

Každému zvolenému algoritmu bude vytvořena vizualizace i interakce individuálně. V jednotlivých algoritmech budou zvoleny klíčové části algoritmu, kde bude probíhat interakce. Tyto klíčové části jsem vybíral po konzultacích s vedoucím práce a také podle popisu a vysvětlení algoritmů z knihy [2]. Zvolení těchto bodů a návrh interakce je nutné provést tak, aby se nejednalo, pokud je to možné, o nevýznamné části nebo triviální úkony, jako např. zápis inkrementované hodnoty.

## 2.4 Možnosti pro uživatele

Uživatel by měl mít veškeré možnosti, aby mohl aplikaci využít pro názornou výuku grafových algoritmů. Základem je možnost vytvoření si vlastního grafu jako vstupu do algoritmu. K tomu budou sloužit editor grafů umožňující vytvořit libovolný graf. V simulaci algoritmu bude základní ovládání a taktéž možnost aktivace interaktivního módu.

V části interakce uživatel přímo pracuje se vstupním grafem. Podle povahy aktuální části je možné po uživateli vyžadovat správný výběr jednoho nebo více uzlů, textové zadání nové hodnoty a případně určení typu hrany.

# Kapitola 3

# Návrh aplikace

Návrhová část dává kompletní přehled o klíčových vlastnostech aplikace. Návrh je udělán pro platformu Java SE (z anglického Standard Edition) s uživatelským rozhraním realizovaným za pomoci knihovny Swing. Tato platforma byla vybrána, protože umožňuje rychlý vývoj a snadnou přenositelnost.

V této části bude také rozebrána volba knihovny, která poskytuje možnosti pro vizualizaci a interakci s grafy. Popis vybrané knihovny bude umístěn již v této kapitole z toho důvodu, že její vlastnosti a možnosti, které poskytuje, jsou klíčové pro vytvoření funkčního návrhu.

## 3.1 Volba knihovny pro práci s grafy

Cílem bylo najít knihovnu pro zvolenou platformu, která bude zajišťovat kompletní práci s grafy. Knihovna měla primárně poskytovat vizualizaci a interakci s grafem. Také bylo nutné dát důraz na možnost uzpůsobení a rozšíření knihovnou poskytovaných tříd a metod pro konkrétní požadavky vytvářené aplikace. Výhodou je, pokud jsou reprezentace uzlů a hran navrženy tak, že mohou obsahovat uživatelem vytvořený objekt, který bude možno i společně s grafem uložit a načíst.

### 3.1.1 Základní kritéria

Z pohledu vizualizace se jedná o možnost změny vizuální podoby jednotlivých uzlů a hran při provádění algoritmu, a to nezávisle na ostatních buňkách. Důležité jsou především vlastnosti pro zvýraznění jako např. změna barvy a šířky obrysové čáry. Vítané jsou potom další vizualizační možnosti, které mohou přispět k lepšímu grafickému podání, jako např. změna geometrického tvaru uzlu. U hran je zejména důležitá schopnost přehledně zobrazit více hran mezi dvěma uzly, podpora orientovaných i neorientovaných hran a smyček (hrana, kde je cílový uzel totožný se zdrojovým). Důležitým aspektem je také nutnost zobrazení textového popisu konkrétního uzlu a hrany. Jedná se zejména o zobrazení jména a aktuálního stavu uzlu a u hrany její váhu.

V interakci je nutné dát uživateli možnost jednoduchého výběru uzlu a hrany. Protože uživatel bude muset zadávat i některé hodnoty, je žádoucí, aby knihovna tuto možnost měla uživatelsky přívětivě vyřešenou. Vstupy jsou však značně individuální podle povahy zpracovávaného algoritmu. Předpokladem je tedy možnost rozšířit/změnit funkce zajišťující interakci, aby jejich chování bylo vyřešeno individuálně pro daný algoritmus.

Důležité jsou i další aspekty, nesouvisející s technickým provedením knihovny. Jedná se zejména o licenční podmínky, kdy je pro účely této práce nutné, aby knihovna byla k volnému užití se zveřejněnými zdrojovými kódy. Při výběru je také nutné se zaměřit na kvalitu a rozsah dokumentace, na ukázkové příklady a celkovou podporu v komunitě. Výhodou je také, probíhá-li stále vývoj knihovny.

### 3.1.2 Možné varianty

Při hledání vhodných knihoven jsem narazil na několik možných variant dostupných pro platformu Java Swing. Všechny jsou pod licencemi Berkeley Software Distribution (dále pod zkratkou **BSD**). První variantou je knihovna **Prefuse**[1], která poskytuje mnoho možností, ale není již několik let v aktivním vývoji. Hlavní problém se jevil ve špatné kompatibilitě s nejnovější verzí použité platformy Java Swing.

Druhou variantou byla knihovna Java Universal Network/Graph[2] (dále pod zkratkou **JUNG**), která určena zejména pro grafy s velkým množstvím uzlů a hran. Nabízí ze všech knihoven pravděpodobně nejpokročilejší algoritmy pro automatické rozvržení grafu. Návrh celé knihovny je koncipován tak, že je umožněna maximální rozšiřitelnost. Pro knihovnu **JUNG** už nevycházejí aktualizace a její vývoj byl pravděpodobně zastaven. Původně měla vytvořená aplikace z této práce využívat knihovnu **JUNG**. Po skončení návrhové stránky a začátku vývoje se ukázalo, že knihovna je hodně orientovaná na práci s grafy, ale ztrácí v ostatních vlastnostech. U komponent pro vizualizaci a interakci nejsou implementovány některé funkce, které by bylo nutné doprogramovat. Řešit by se musel také problém při ukládání uživatelského objektu společně s grafem.

Pro práci s grafy byla zvolena knihovna **JGraphX**[3], která nahradila původně vybranou knihovnu **JUNG**. Změna knihovny v době dokončeného návrhu aplikace byla možná, protože obě knihovny poskytují stejné klíčové vlastnosti při práci s grafy. Knihovna **JGraphX** je nepřetržitě vyvíjena od roku 2001 a komerčně prodávána ve verzi pro jazyk Javascript. Oproti knihovně **JUNG** poskytuje kvalitnější dokumentaci, vlastní fórum s podporou a také pokročilejší ukázkové příklady. Má implementovaný převod modelu grafu do XML včetně vloženého uživatelského objektu. Není však vždy možné jednoduše modifikovat chování některých vnitřních objektů. Nevýhody v rozšiřitelnosti jsou vyváženy celou škálou možných nastavení jak pro samotný graf, tak i pro vizualizační komponentu.

## 3.2 Popis knihovny JGraphX

Knihovna **JGraphX** tvoří základ vytvořené aplikace. Poskytuje rozsáhlé možnosti pro reprezentaci a vizualizaci grafů. Obsahuje komponenty pro snadnou integraci s prostředím Java Swing. V následujících sekcích budou rozebrány základní vlastnosti této knihovny a uvedena některá rozšíření vytvořená pro účely této práce.

Pro pochopení a získání znalostí, jak s knihovnou pracovat, byla využita příručka od autorů knihovny [3]. Příručka popisuje pouze základní funkce a možnosti, ale nezabývá se návrhem knihovny ani způsobem, jak některé funkce upravit. Pro získání i těchto pokročilých znalostí jsem procházel zdrojové kódy knihovny. Knihovna využívá návrhové vzory a obecné zásady pro správný návrh. V orientaci ve zdrojových kódech a pro jejich pochopení jsem využil získaných znalostí z knihy o návrhových vzorech od Freemana [4]

[1] http://prefuse.org/
[2] http://jung.sourceforge.net/
[3] http://www.jgraph.com/

### 3.2.1 Reprezentace a vizualizace grafů

Graf je reprezentován samostatnou třídou, která poskytuje velké množství možností nastavení. Graf se v základu skládá z modelu a tabulky obsahující seznam předdefinovaných stylů. Model grafu slouží jako reprezentace struktury v podobě uzlů a hran. Uzly a hrany jsou v grafu reprezentovány objektem (dále pod označením buňka) stejného typu, který obsahuje příznak určující, zda jde o hranu či uzel. Každá taková buňka drží veškeré nastavení z geometrického pohledu (velikost a umístění), z pohledu abstraktního nastavení vizuální podoby a také informace o případném kontaktu s okolím (např. zdrojový a cílový uzel u hrany). Do každé takové buňky lze také vložit jako hodnotu libovolný objekt. U takto vloženého objektu je nutné zajistit, aby mohl být serializován do XML podoby.

Vizuální nastavení je reprezentováno v buňce výčtem požadovaných vlastností nebo pouze názvem jednoho z předdefinovaných stylů. O samotné vykreslení grafu a všech jeho buněk se stará vizualizační komponenta, kterou je možné přímo propojit s komponenty rozhraní Java Swing. Každá buňka je vykreslena přesně podle zadaných vizualizačních a geometrických parametrů. Hodnota vložená do buňky se standardně vykresluje převodem na řetězec (v Java standardní metoda *toString*), programátor má možnost specifikovat, jakým způsobem se má konkrétní vložený objekt zobrazit.

Vizualizační komponenta taktéž umožňuje interakci s uživatelem a nastavení limitujících omezení při interakci (např. zákaz vymazání buňky). Uživatel může provádět veškeré standardní akce jako například přibližování a oddalování, přesouvání jedné buňky i skupiny buněk naráz, změnu velikosti buněk a další akce. Důležitou akcí je možnost editace hodnoty vložené do buňky. Zde uživatel zadává novou hodnotu vždy ve formě řetězce a následný převod takto zadaného řetězce je již v režii programátora.

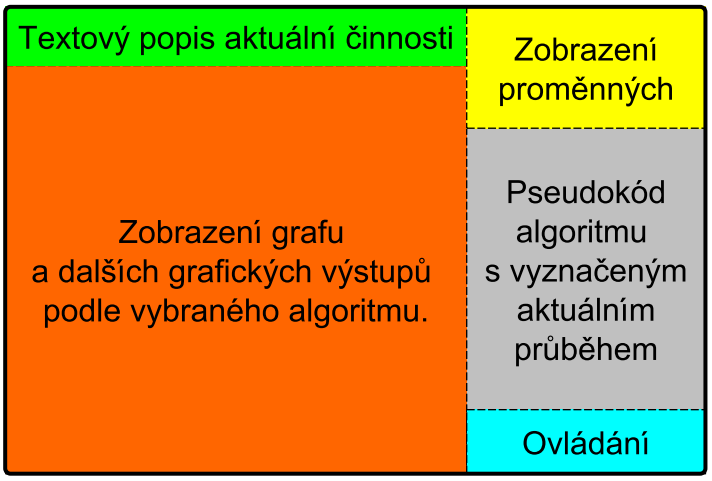
### 3.2.2 Ukládání a načítání grafů

Knihovna **JGraphX** umožňuje ukládání grafů s podporou v uložení v grafickém formátu a značkovacím jazyce XML. Z grafických formátů to jsou pro rastrovou grafiku např. formáty PNG, JPEG, GIF a pro vektorovou grafiku formát SVG. Při uložení do grafického formátu dojde ke ztrátě dat o struktuře, propojení a datovém obsahu grafu, nelze proto graf znovu načíst. Pro účely znovunačtení je nutné využít uložení v podobě XML. Při ukládání do XML se z grafu standardně ukládá pouze model, neukládá se nastavení možností ani tabulka předdefinovaných stylů. Model obsahuje pouze výčet všech uzlů a hran včetně jejich vizuálního nastavení, geometrických informací a případně uživatelem vložené hodnoty.

První problém s takovým uložením nastává v případě využívání vlastní tabulky předdefinovaných stylů. Uložený model obsahuje u buněk pouze odkazy do této tabulky (skrze jména stylů), tabulka se však neukládá a při znovunačtení grafu ze souboru se všechny buňky zobrazují pouze v základním stylu. Tento problém jsem vyřešil vytvořením rozšířeného ukládání a načítání, kde společně s modelem grafu se vždy ukládá i tabulka stylů.

Další problém představovalo ukládání vloženého uživatelského objektu společně s každou buňkou. Zde se ukládají datové položky vloženého objektu i název třídy pro daný objekt. V samotném simulátoru algoritmů se používají objekty, které odpovídají přesně potřebám algoritmu a mají rozšíření, která jsou používaná pouze za běhu, a není potřebné je ukládat. Vytvořil jsem proto třídy, mající pouze atributy, které se ukládají, a rozšířil ukládání i o převod na objekty tohoto typu.

Všechny objekty vložené do grafu musí být serializovatelné (v Java pomocí implementovaného rozhraní *Serializable*). Knihovna vnitřně pracuje tak, že při změnách provede kopii buňky a původní buňku z grafu odstraní. Na nově vytvořenou buňku jsou poté aplikovány




Obrázek 3.1: Návrh rozložení uživatelského rozhraní simulátoru

provedené změny. Kopie se provádí pomocí serializace buňky. Není tedy možné při serializaci některé atributy přeskočit (v Java pomocí anotace *transient*). Pokud by k takovému přeskočení atributů došlo, neprovádělo by se správně kopírování buněk.

## 3.3 Obecný popis simulátoru

Pro účely této aplikace je simulátor část, ve které uživatel může ovládat a sledovat průběh vybraného algoritmu. Simulátor se bude otevírat s vybraným grafem a algoritmem v nezávislém okně. Uživatel bude mít možnost mít otevřeno více simulací současně a s každou pracovat nezávisle na ostatních.

### 3.3.1 Vizuální stránka simulátoru

Simulátor z pohledu vizuální stránky musí zobrazovat všechny důležité informace při vykonávání algoritmu. Zobrazování těchto informací se dá rozdělit do pěti částí. Rozdělení jednotlivých částí je možné vidět na obrázku 3.1. Nejvíce dominantní z celého pohledu na simulátor je část, kde je zobrazen graf. Podle varianty algoritmu se tu navíc mohou zobrazovat i další pomocné části, např. další graf představující výsledek algoritmu.

Druhou část tvoří panel, kde je zobrazen pseudokód algoritmu. Tento panel uživateli zobrazuje pomocí zvýraznění, který řádek se právě provádí. Pro větší přehlednost tato část navíc poskytuje možnost vertikálního centrování, kdy aktivní řádek je vždy zobrazen uprostřed panelu. Simulátor také poskytuje zobrazení proměnných a textového popisu činnosti, kterou algoritmus právě vykonává.

### 3.3.2 Ovládání simulátoru uživatelem

V této části bude popsáno základní ovládání simulátoru, popis interakce v průběhu algoritmu bude rozebrán v samostatné sekci. Simulátor umožnuje uživateli libovolně měnit pozice uzlů i hran. Strukturu grafu a některé vložené hodnoty jako názvy uzlů a váhy hran měnit v simulátoru nelze. Pro samotné ovládání je poskytnut speciální panel s tlačítky

a posuvníkem pro rychlost. Uživatel může spustit algoritmus a bez dalších zásahů sledovat jeho postup, případně může upravit rychlost provádění, pozastavit provádění nebo ho restartovat. V jakékoliv fázi je možné algoritmus začít krokovat.

## 3.4 Interakce uživatele

Interakce s uživatelem je jednou z hlavních částí, kterou se bude aplikace odlišovat od podobných nástrojů. Cílem je umožnit uživateli volit kroky algoritmu a poté kontrolovat, zda je provádí správně. V této sekci je v bodech popsána interakce u jednotlivých algoritmů, jak bude implementována. U každého algoritmu jsou vyžadovány trochu jiné akce od uživatele a důraz je kladen na hlavní část konkrétního algoritmu. Pseudokódy jednotlivých algoritmů jsou k nalezení v příloze C.

### 3.4.1 Prohledávání do šířky

Důležitým prvkem v **BFS** je FIFO fronta $Q$ a vkládání prvků do této fronty. Dále je zde u každého uzlu proměnná, která představuje počet hran na cestě do tohoto uzlu od počátečního uzlu.

1. Graf je připraven do stavu, kdy jsou vidět jednotlivé pojmenované uzly. Uživatel vybere, který uzel bude jako počáteční. Po zvolení počátečního uzlu se do všech uzlů doplní aktuální vzdálenosti (nula pro počáteční, jinak nekonečno). Počáteční uzel se vloží do fronty $Q$.

2. Pokud fronta $Q$ není prázdná, zvolí uživatel další uzel, který bude z $Q$ fronty odebrán. Prázdná fronta značí konec algoritmu.

3. Uživatel je vyzván, aby vybral všechny uzly, které se mají jako další vložit do fronty. Pokud je výběr správný, doplní se automaticky u těchto uzlů nové hodnoty vzdáleností (uživatel není nucen je zadávat). Při zvolení chybného uzlu je upozorněn, že při výběru chyboval. Do fronty $Q$ se uzly vkládají v pořadí, v jakém jsou vybírány, uživatel tedy přímo ovlivňuje pořadí, v jakém bude algoritmus postupovat.

### 3.4.2 Prohledávání do hloubky

U tohoto algoritmu je výsledkem graf, kde jsou jednotlivé uzly označeny časovou známkou (pořadím), kdy začalo a kdy skončilo jejich zpracování. Důležitým prvkem je zde zásobník $S$, který v pseudokódu nahrazuje zápis pomocí rekurze. Další částí v průběhu vykonávání algoritmu je označení hran podle toho, zda se jedná o zpětnou **B** (anglicky Back), dopřednou **F** (anglicky Forward) nebo křížící **C** (anglicky Cross).

1. Uživatel je vyzván, aby zvolil další dosud nezpracovaný uzel. Do takto vybraného uzlu je doplněna první časová známka a je označen jako aktivní. Tento bod se poprvé vykoná po proběhnutí inicializace. Pokud jsou již všechny uzly zpracovány, je algoritmus dokončen.

2. K aktivnímu uzlu uživatel vytvoří postupným označováním seznam sousedních uzlů. V seznamu se musí nacházet všechny sousední uzly, jinak je uživatel upozorněn na chybu. Pořadí uzlů v tomto seznamu ovlivňuje další průchod algoritmu grafem.

3. Pokud je seznam sousedů prázdný, pokračuje se bodem 6. Jinak algoritmus odebere další uzel ze seznamu sousedů a pokračuje následujícím bodem.

4. Pokud odebraný uzel byl již v minulosti navštíven, pokračuje se následujícím bodem pro výběr typu hrany. V případě, že uzel jestě navštíven nebyl, stane se aktivním uzlem a doplní se první časová známka. Poté se provede vytvoření seznamu sousedů v bodě 2.

5. Uživatel je vyzván, aby u hrany vedoucí do vybraného uzlu zvolil její typ (**B**, **F**, **C**). Je kontrolována správnost zadaného typu hrany. Poté se znovu pokračuje bodem 3.

6. Do uzlu se automaticky doplní druhá časová známka a stane se uzavřeným. Aktivním se stává předcházející uzel podle zásobníku $S$. V případě, že je zásobník $S$, prázdný pokračuje se výběrem dalšího uzlu v bodě 1.

### 3.4.3 Topologické uspořádání uzlů

U **topologického uspořádání** uzlů je cílem vytvořit seznam uzlů $L$. Tento seznam bude obsahovat uzly vstupního grafu seřazené sestupně podle časové známky dokončení zpracování uzlu algoritmem **DFS**. Seznam uzlů $L$ je zde realizován jako graf, kde jsou všechny uzly vedle sebe v jedné linii.

1. Stejně jako v případě **DFS**, uživatel nejprve vybere počáteční uzel. Uzlu je automaticky doplněna první časová známka (hodnota vždy 1).

2. Uživatel vybírá další uzly tak, jak jsou zpracovávány v **DFS**. Časové známky se doplňují automaticky. Uzly musí být vybrány v souladu s algoritmem **DFS**, pokud je více možností, tak je přípustná libovolná z nich.

3. Znovuvybrání stejného uzlu se považuje za požadavek na uzavření tohoto uzlu. Uzavřený uzel se umístí do seznamu $L$. Pro větší uživatelský komfort dojde k uzavření i všech předcházejících uzlů, z kterých už nejde pokračovat dále. Uživatel není nucen uzly jeden po druhém odklikávat. Opět se pokračuje krokem 2, dokud nejsou všechny uzly v seznamu $L$.

### 3.4.4 Hledání silně souvislých komponent

Algoritmus **SCC** je specifický tím, že volá **DFS** dvakrát. Také potřebuje transponovaný vstupní graf. Cílem je postupně získat silně souvislé komponenty a tím vytvořit graf komponent. Při vizualizaci algoritmu je možnost vidět nezávisle původní graf, transponovaný graf a graf komponent.

1. První dva kroky algoritmu (dle pseudokódu) provede program sám, uživatel pouze zvolí počáteční uzel. Po zvolení počátečního uzlu se zavolá na vstupní graf **DFS** a do uzlů se zapíší časové známky. Poté se provede transponování grafu a ten se zobrazí.

2. Uživatel nyní v transponovaném grafu postupně vybere uzly tvořící další silně souvislou komponentu. Jedná se o komponentu, která bude jako další nalezená algoritmem **SCC**. Uživatel musí vybrat všechny uzly dané komponenty, nezáleží však na pořadí výběru uzlů.

3. Jakmile je dokončena jedna komponenta, tak se v grafu komponent objeví uzel, který ji představuje. Graf komponent se tvoří postupně z vytvořených silných komponent a doplňují se jednotlivá propojení (hrany). Zároveň jsou uzly komponenty barevně odlišeny v transponovaném grafu. Uživatel se znovu vrací do bodu 2 a vytváří další silně souvislou komponentu.

### 3.4.5 Dijkstrův algoritmus

**Dijkstrův** algoritmus vytváří množinu zpracovaných uzlů $S$ a využívá prioritní frontu $Q$, seřazenou vzestupně podle odhadu nejkratší cesty. Vybraný uzel z fronty $Q$ se vždy umisťuje do množiny $S$. Výsledkem by měl být graf, zobrazující nejkratší cesty od počátečního uzlu.

1. Uživatel vybere, z kterého uzlu se budou vzdálenosti a nejkratší cesty počítat. Do tohoto uzlu program doplní hodnotu vzdálenosti nula. Ostatní uzly mají nastavenou vzdálenost na nekonečno.

2. V tomto kroku uživatel určí, který uzel bude jako další zpracovávaný. Pořadí je dáno prioritní frontou a program kontroluje, zda je zvolený uzel správný.

3. Uživatel doplní nové hodnoty vzdáleností do okolních uzlů (relaxace hran) a vrací se na krok 2, dokud nejsou zpracovány všechny uzly. Zapsané hodnoty po relaxaci hrany jsou kontrolovány a uživatel je musí zapsat správně.

### 3.4.6 Bellmanův-Fordův algoritmus

Narozdíl od předchozího **Dijkstrova** algoritmu je zde větší množství iterací a provádění relaxace hran. Protože dochází k opakovanému vyhodnocování, není zde zastoupena žádná množina dokončených uzlů. Cílem je mít správně určené délky cest i s ohledem na záporné hrany a detekovat případný záporný cyklus. Uživatel může ovlivnit rychlost algoritmu správným výběrem hran.

1. Uživatel zvolí počáteční uzel. Postupně vybírá jednotlivé hrany a provádí jejich relaxaci. Program si pořadí hran zapamatuje pro další iterace.

2. V tomto kroku se již jednotlivé hrany vyznačují v pořadí, v jakém je uživatel vybral v prvním kroku. Pro každou hranu uživatel provede relaxaci hrany a případně zapíše nové hodnoty vzdálenosti (pouze menší než aktuální). Program kontroluje správnost zadaných hodnot. Tento krok se opakuje, dokud dochází ke změnám.

3. Po dokončení výpočtu vzdáleností se ještě jednou provede iterace nad hranami. Uživatel nyní rozhoduje, zda daná hrana porušuje kontrolní podmínku a v grafu se tak vyskytuje záporný cyklus.

## 3.5 Editor grafů

Editor grafů není přímo požadován v zadání této práce, ale od počátku je s ním počítáno jako s rozšířením. Editor by měl dát uživateli jednoduchou možnost, jak si vytvořit nový graf popřípadě upravit stávající. Uživatel může přidávat nové uzly i hrany a zadávat do nich základní informace jako jméno uzlu a váhu hrany. Navíc uživatel může dopředu určit počáteční uzel, od kterého bude algoritmus poté v tomto grafu začínat.

Editor taktéž umožní nastavit parametry, jak se se mají jednotlivé buňky vykreslovat. Jedná se zejména o vizualizační parametry typu barvy výplně, geometrický tvar, velikost písma, šířku a barvu obrysové čáry a další. Je možné zadat různé vizualizační parametry pro jednotlivé stavy uzlů a hran (např. uzel po inicializaci, stromová hrana, ...).

Z celkového pohledu bude editor sloužit jako hlavní okno programu. Editor společně s hlavním menu bude otevřen po spuštění programu. Samotná část editoru bude poskytovat grafické zobrazení grafu i jeho náhled. Viditelný bude také textový zápis grafu pomocí jednotlivých uzlů a seznamu jeho sousedů. U hlavního menu jsou důležité funkce pro načtení a uložení grafu, vytvoření nového grafu a výběr algoritmu, pro který bude spuštěna simulace.

## 3.6 Shrnutí návrhové části

V této kapitole byly detailně rozebrány jednotlivé prvky návrhu aplikace. Aplikace bude realizována za pomoci platformy Java SE a knihovny Swing pro uživatelské rozhraní. Klíčové pro celý návrh i další implementaci bylo vybrání knihovny pro práci s grafy. Zvolená knihovna pro práci s grafy se jmenuje **JGraphX** a má všechny požadované vlastnosti. Rozhodující pro výběr byla zvláště dobrá podpora této knihovny.

Druhá část návrhu popisovala rozložení a základní možnosti simulátoru. Je zde navrženo rozložení uživatelského rozhraní, podle kterého proběhne implementace. Rozebrána také byla interaktivita pro jednotlivé algoritmy. Pro každý algoritmus byly zvoleny klíčové části, kde by měla interaktivita probíhat. Cílem bylo, aby se po uživateli nechtěly úplně triviální úkony. V návrhu je také zahrnuto rozšíření v podobě editoru grafů.

# Kapitola 4

# Implementace aplikace

Implementace celé aplikace se skládá ze dvou hlavních bodů. Prvním bodem je vytvoření editoru grafů, který bude sloužit i jako hlavní část programu. Druhým bodem je vytvoření simulátorů pro jednotlivé algoritmy. Tato kapitola rozebírá oba body z pohledu uživatelského rozhraní i vnitřní implementace.

Implementace je provedena v programovacím jazyku Java, který poskytuje rozsáhlé možnosti ve využití již hotových řešení. Pro poznání těchto možností, jsem využíval knihu od Sierrové [5]. Při vytváření uživatelského rozhraní byla velmi nápomocná příručka od autorů knihovny Swing [6].

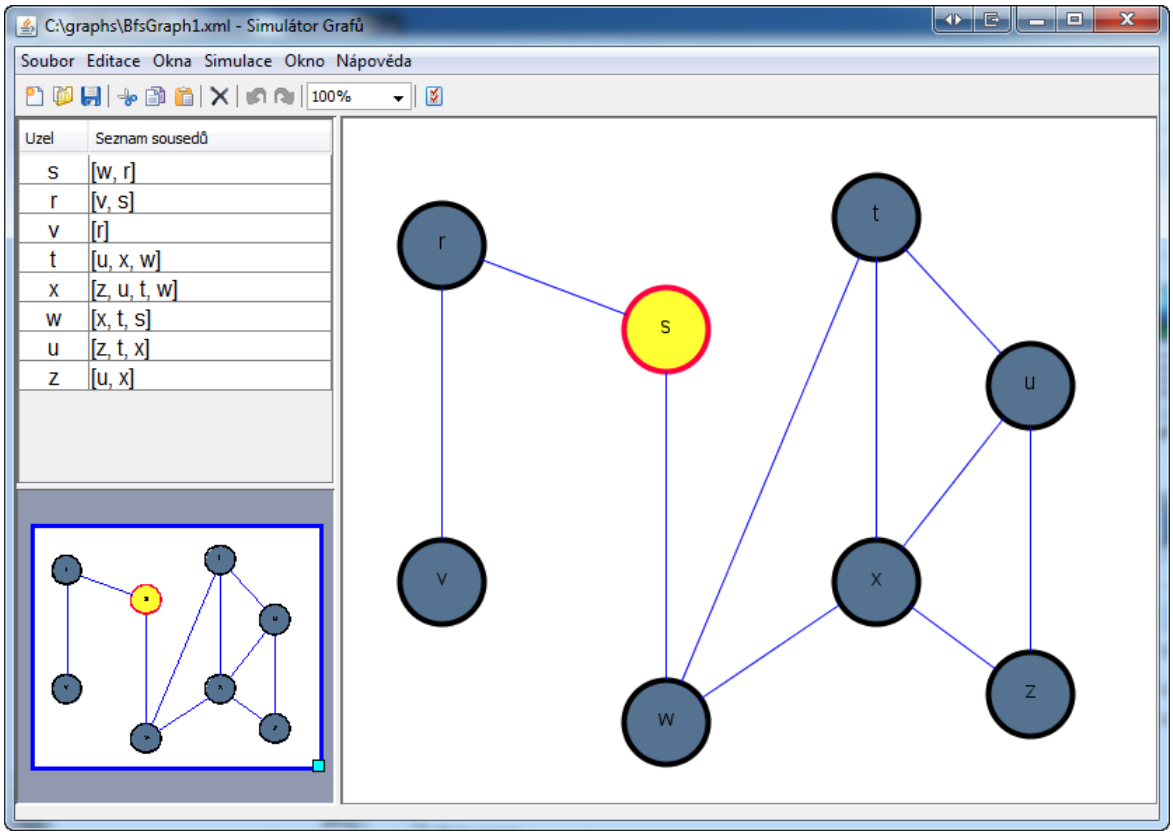
## 4.1 Uživatelské rozhraní editoru grafů

Uživatelské rozhraní editoru grafů je možné vidět na obrázku 4.1 a slouží zároveň jako hlavní rozhraní programu. Uživatel zde provádí načítání i ukládání vytvořených grafů a spouští simulátory jednotlivých algoritmů. Je zde možné taktéž zobrazit nápovědu a základní informace o aplikaci. Uživatel si může vybrat, ve kterém motivu knihovny Swing bude celé rozhraní zobrazeno. Obrázky použité pro tlačítka byly převzaty přímo z knihovny **JGraphX**.

Z pohledu editoru grafů je nabízen panel, kde je zobrazen graf a uživatel zde může přidávat nové uzly a hrany. Je zde možnost si každý uzel libovolně zvětšit či zmenšit a upravit jeho polohu. Při přidávání uzlu se mu implicitně vygeneruje nové jméno. Jména se generují jako velká písmena s počátečním písmenem *A*, postupně podle anglické abecedy. Pokud jsou již všechny písmena v grafu obsažena, začíná se od začátku písmenem *A* ke kterému je přidáno číslo podle toho kolikrát již proběhla rotace. Uzly jsou tedy přidávány v posloupnosti *A-Z*, *A1-Z1*, *A2-Z2*, ... Posloupnost jmen vždy platí pro aktuální graf. Pokud uživatel některé uzly vymaže nebo změní jejich jméno, nemá to vliv na pojmenování nových uzlů. Pokud dojde k načtení, případně vytvoření nového grafu, začínají jména uzlů znovu od začátku.

V levé části editoru je zobrazena podoba grafu v náhledovém okně a také v tabulkovém zápise. V náhledovém okně je možné provést přiblížení, oddálení a určení, která část grafu má být v aktuální editaci vidět. Všechny tyto akce se zároveň promítají do komponenty zobrazující graf a také do panelu nástrojů.

Nad náhledem je zápis pomocí tabulky, kde v prvním sloupci je jméno uzlu a ve druhém výpis seznamu sousedů. Pořadí uzlů v tabulce i v seznamu sousedů odpovídá vnitřní struktuře grafu a algoritmy budou graf v tomto pořadí také procházet. Toto pořadí může být



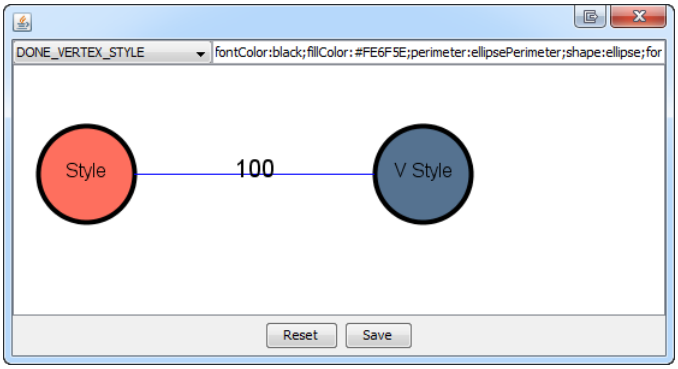

Obrázek 4.1: Uživatelské rozhraní editoru grafů a hlavního menu programu

změněno pomocí interakce v simulátoru a také označením uzlu jako počátečního v editoru grafů. Počáteční uzel je možné na obrázku rozhraní 4.1 vidět jako žlutý s červenou obrysovou čárou. Tabulka je spojena s komponentou vykreslující graf a vybraný uzel v tabulce se zároveň označí i v grafu.

## 4.2 Možnosti v rozhraní editoru

Editor poskytuje základní možnosti v úpravě grafu a jeho vizuálních stylů pro simulátor.

- Načítání, ukládání a vytváření grafů. Graf se načítá i ukládá v podobě XML ve kterém je zapsána jeho struktura a vizuální styly. Využívá se zde rozšíření popsané v sekci 3.2.2 a implementované ve třídě *GraphIO*.
- Vrácení se zpět a znovu provedení akce. Knihovna **JGraphX** poskytuje přímo třídy a metody, kterými se tato funkcionalita realizuje.
- Možnosti kopírování, vkládání, vymazání uzlů a hran. Částečná implementace je v knihovně **JGraphX**. Vytvořil jsem rozšíření, aby při těchto akcích nebyla porušena konzistence grafu. V případě vkládání uzlu se stejným jménem, jako má uzel již v grafu obsažený, dojde k automatickému přejmenování podle pravidel popsaných v předchozí sekci.



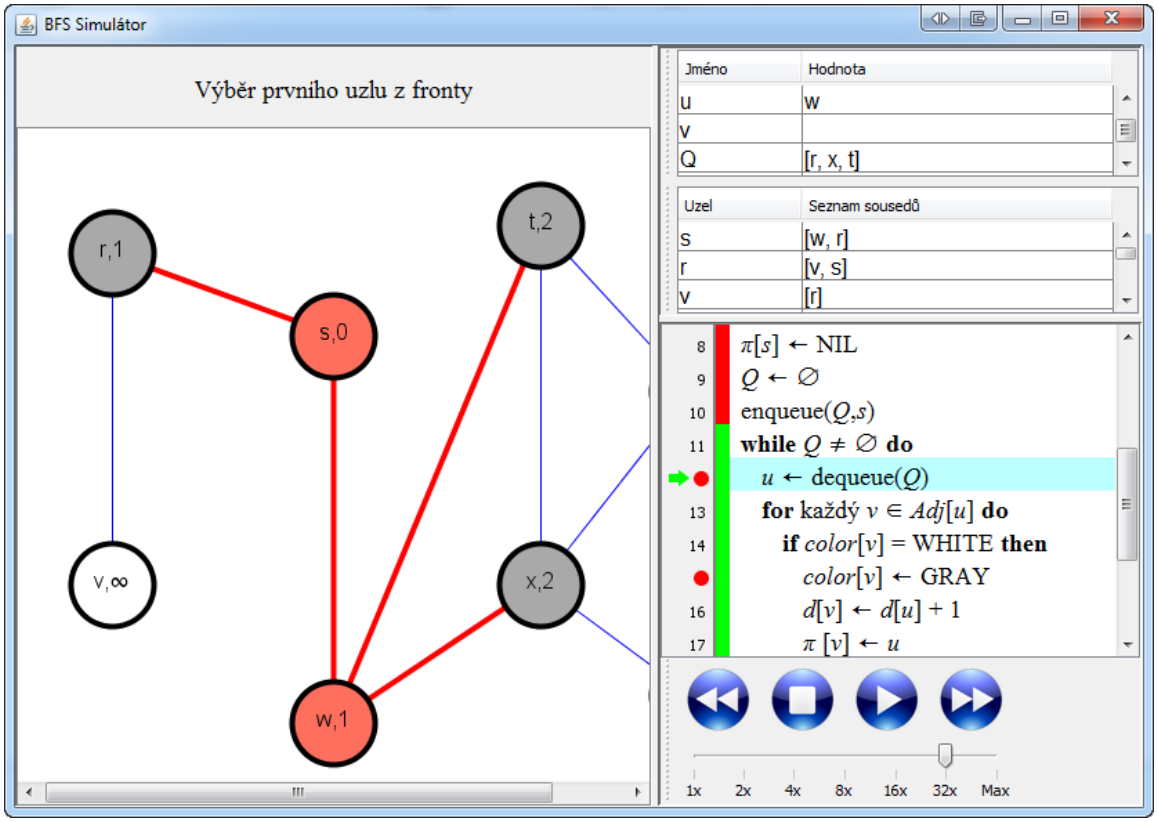

Obrázek 4.2: Dialog pro zadávání jednotlivých stylů pro graf

- Výběry skupiny z grafu, např. všech hran, všech uzlů, všech uzlů i hran. Implementace je v knihovně **JGraphX**.
- Nastavení počátečního uzlu. Takto označený uzel může být pouze jeden. Simulátor algoritmu potom zahajuje procházení od takto označeného uzlu.
- Nastavení stylů. Uživatel může nastavit u grafu kompletně styly, které budou použity v simulátoru. Z kontextového menu si vybere styl, který chce změnit, a upraví příslušné parametry tohoto stylu. Na obrázku 4.2 je možné vidět nastavení vizuálního stylu pro uzavřený uzel při simulaci. Příloha D popisuje základní možnosti pro styly.
- Přiblížení a oddálení. Možnost zvolit z předpřipravené procentuální hodnoty nebo zadat vlastní. Komponenta pro vizualizace grafu poskytuje možnost nastavit hodnotu přiblížení, není ovšem propojena s myší. Propojení jsem realizoval pomocí standardních možností, které poskytuje knihovna Swing a rozšířil ho i pro náhledový panel.
- Výběr motivu prostředí pro uživatelské rozhraní. Standardně se použije nativní pro danou platformu.
- Spuštění simulátoru jednotlivých algoritmů v novém okně.
- Nápovědu a informace o programu. Zobrazení je v podobě samostatného dialogového okna. Vnitřně je nápověda ve formě HTML a zobrazena je v textovém panelu.

V části pro editaci grafu je umístěno kontextové menu nabízející možnost skrytí popisků uzlů a hran. Navíc je nabízena možnost překreslení paralelních hran, aby nebyly vykresleny přes sebe.

## 4.3 Implementace editoru grafů

Při implementaci editoru grafů se vycházelo z již existujícího řešení v podobě ukázkového příkladu knihovny **JGraphX**. Toto řešení bylo nutné přizpůsobit a upravit pro potřeby řešené aplikace. V původním editoru bylo realizováno velké množství ukázkových funkcí a samotných možností, které knihovna poskytuje. Informace uložené do uzlů a hran jsou pouze v textové podobě, bez vnitřní struktury.




Obrázek 4.3: Uživatelské rozhraní simulátoru

Editor vytvořený v této práci má odebrané některé funkce, které nebyly důležité, případně nemohly být využity při vytváření grafu pro simulátor. Ponechány byly pouze základní funkce, které jsem z velké části upravil a rozšířil. Převzal jsem koncept návrhu pomocí akcí, který umožňuje oddělit prvky uživatelského rozhraní od výkonné části kódu.

## 4.4 Uživatelské rozhraní simulátoru

Rozhraní simulátoru je hlavní část, které jsem věnoval největší pozornost. Bylo nutné určit rozložení, rozhraní a také vytvořit komponenty, které mají jisté nestandardní vlastnosti. Značné problémy představovalo omezení v podobě nutnosti běhu v nízkém rozlišení. Při simulaci se muselo dopředu počítat s variantou, kdy pro uživatele není viditelný kompletní pseudokód ani vstupní graf.

Uživatelské rozhraní je z velké části poskládáno z komponent (např. *JSplitPane*), které dovolují změnit velikost jednotlivých částí rozhraní. Uživatel může také panely ovládání a proměnných nechat přejít do formy dialogového okna, jejich místo potom zabere panel s pseudokódem. Na obrázku 4.3 je možné vidět okno simulátoru pro algoritmus **BFS**. Je zde zachycen ve fázi vybírání uzlu z fronty $Q$ a jsou zde zadány dva body pro pozastavení na řádku 12 a 15.

### 4.4.1 Panel s pseudokódem

Panel s pseudokódem je v této aplikaci nejvíce propracovanou komponentou uživatelského rozhraní. Je postaven na textovém panelu (třída *TextLineNumber*) od Roba Camicka[1], který navíc vykresluje čísla řádků. Pro účely této aplikace bylo nutné realizovat velké množství rozšíření a úprav. V původní verzi se zobrazovala pouze čísla řádků bez dalších možností. Předělána je zejména metoda *paintComponent*, která vykresluje panel.

Z hlediska číslování je komponenta uzpůsobena tak, aby čísla řádků vždy začínala od začátku pro každou metodu v algoritmu. Jeden řádek z hlediska pseudokódu může zabírat i více řádků v textové komponentě, číslování je tomu uzpůsobeno. Reálný počet řádků, které bude jeden řádek pseudokódu zabírat, je dán velikostí volného místa. Reprezentace pseudokódu algoritmu a realizace číslování je popsána v sekci 4.5.1. Komponenta automaticky text formátuje tak, aby vždy byl vidět text kompletní (pokud to je technicky možné) a uživatel nemusel používat horizontální posuvník.

Druhým rozšířením je možnost nastavit v programu, který řádek má být označen jako aktivní (metoda *setCurrentLine*). Aktivní řádek se vykresluje označený u čísla řádku zelenou šipkou a text tohoto řádku má světle modré podbarvení. Pro podbarvování řádků jsem realizoval třídu *LineHighlighter*, která implementuje rozhraní *HighlightPainter* z knihovny Swing. Zároveň je možné zapnout zobrazení aktivního řádku tak, aby byl automaticky uprostřed panelu pseudokódu. Řádky umístěné na horním nebo dolním okraji se do středu neposouvají. Centrování aktivního řádku má výhody hlavně v tom, že uživatel vždy vidí řádky před i po řádku, který se právě provádí. Celý proces vizualizace díky tomuto rozšíření působí mnohem lépe a přehledněji.

V pseudokódu je také možné nastavit libovolné zvýraznění zadaných částí textu (metoda *highlightLine*). V aplikaci z této práce se toho využívá pro označení začátku řádků. Červené označení značí část kódu, kterou algoritmus už provádět nebude, zeleně jsou potom označeny řádky, které je třeba ještě vykonat. Zvýraznění řádků je stejně jako podbarvení aktivního řádku realizováno ve třídě *LineHighlighter*.

Posledním rozšířením jsou body pro zastavení. Tyto body jsou uživatelem zadávány pomocí dvojkliku myši na příslušný řádek, kde má být bod umístěn. V panelu pseudokódu se místo čísla řádku vykreslí označení v podobě kolečka červené barvy. Implementace je provedena pomocí zachytávání zpráv od myši v metodě *mouseClicked* a převodu souřadnice kliknutí na číslo řádku. Čísla vybraných řádků jsou uloženy ve vlastním seznamu. Tento seznam je poté k dispozici v programu při řízení simulace.

Nad panelem pseudokódu je navíc realizováno kontextové menu pro zapínání interaktivního režimu a centrování. Pro realizaci menu využívám třídy *JLayer* a *LayerUI* přidané v Java 1.7 do knihovny Swing. Tyto třídy umožňují zachytávat zprávy pro libovolnou komponentu uživatelského rozhraní. Není tedy nutné nijak zasahovat do již vytvořeného kódu. Menu je umístěno ve třídě *AbstractSimulatorPanel*.

### 4.4.2 Vizualizační panel

V tomto panelu jsou umístěny prvky, které vizuálně znázorňují průběh algoritmu. Podle konkrétního typu algoritmu může tato část obsahovat různé prvky. Například u algoritmu pro **topologické uspořádání** je kromě vstupního grafu zobrazen navíc i seznam $L$ ve formě druhého grafu. U algoritmu **SCC** jsou navíc zobrazeny transponovaný graf a graf komponent.

[1] K nalezení na adrese: http://tips4java.wordpress.com/2009/05/23/text-component-line-number/

Grafy jsou umístěny v komponentě pro zobrazení a práci s grafem, kterou poskytuje přímo knihovna **JGraphX**. V této komponentě je možné měnit rozložení grafu a také aktivovat některé pokročilé prvky v případě interaktivního módu. Mohou být také přidány některé rozšiřující možnosti, pokud je to vhodné pro konkrétní algoritmus. Například v případě algoritmu **DFS** v interaktivním módu se typ hrany vybírá z kontextového menu přímo u hrany.

### 4.4.3 Ostatní panely

Kromě hlavních prvků uživatelského rozhraní popsaných v předcházejících sekcích jsou zde i další části. První z nich je panel, kde se vyskytuje textový popis prováděného kroku v algoritmu. Tento popisek, jak je možné vidět na obrázku 4.3, se nachází nad vizualizačním panelem. Je vytvořen pomocí třídy *JLabel* a vnitřně uložen v podobě HTML.

Vedle popisku je na pravé straně umístěn panel pro proměnné. Realizace je provedena pomocí modifikace třídy *JToolBar*, aby panel mohl přejít do podoby dialogu. Jsou zde zobrazeny všechny důležité proměnné, které se v grafu vyskytují, včetně těch, které nejsou přímo v pseudokódu, např. zásobník $S$ v případě algoritmu **DFS** nebo iterační seznam cyklu. U každé proměnné je v podobě textového řetězce zobrazena její aktuální hodnota v daném kroku. Ukládání proměnných a převod na textový řetězec provádí řídicí jednotka popsaná v kapitole 5.

Poslední dosud nepopsanou částí uživatelského rozhraní je ovládání simulátoru. Uživatel má k dispozici celkem čtyři tlačítka a posuvník rychlosti. Dvě z nich slouží pro krokování dopředu a dozadu, další je tlačítko pro kontinuální běh zadanou rychlostí na posuvníku. Běh kontinuální rychlostí je realizován pomocí třídy *Timer* (z knihovny Swing), která periodicky vyvolává akce se zadanou prodlevou. Poslední tlačítko slouží pro restart algoritmu. Stejně jako panel proměnných je i panel ovládání realizován jako modifikace třídy *JToolBar*.

## 4.5 Reprezentace algoritmů v aplikaci

Způsob reprezentace algoritmu určuje i možnosti, které simulátor může nabízet. Pro účely této aplikace je celková reprezentace realizována ve třech částech. Jedná se o pseudokód ve formě dokumentu, algoritmus zapsaný jako posloupnost instrukcí a data, nad kterými algoritmus operuje, v podobě uzlů a hran grafu a vložených uživatelských objektů.

### 4.5.1 Pseudokód algoritmu

Pseudokód algoritmu je realizován jako vstupní dokument do textového panelu. Je možné zadat individuální styl a mít různé formátovaní např. pro klíčová slova, proměnné, normální text, ... Tyto styly je možné později jednoduše upravit, případně přidat nové.

Každý takový zápis algoritmu je v programátorské podobě tvořen polem objektů představujícím řádek. Každý řádek je potom tvořen polem objektů představujících jeden textový úsek daného řádku. Každému textovému úseku je možné přidat jeho styl. Pro účely tohoto programu jsem realizoval třídu, která obsahuje základní formátovací styly. Protože textový panel pracuje s textem ve vnitřní reprezentaci jako se speciálním objektem – dokumentem, je nutné reprezentaci algoritmu na tento dokument převést. Vytvořil jsem třídu, která tento převod ze zápisu pseudokódu na dokument realizuje.

Zápis pseudokódu algoritmu bez dalších pomocných informací by bylo komplikované dále zpracovat. Hlavním problémem jsou čísla řádku, která nejsou ve standardní podobě,

ale každá část algoritmu je číslována od začátku počínaje číslem jedna. Navíc je výsledný dokument ve své vnitřní struktuře, se kterou panel pseudokódu pracuje, rozšířen o pomocné řádky pro realizaci vertikálních mezer. Z těchto důvodů jsem vytvořenou třídu pro převod na dokument rozšířil tak, aby vypočítala čísla řádků pro zobrazení i čísla řádku reálná (bez vertikálních mezer). Tyto informace jsou formou atributu vloženy do každého řádku v dokumentu.

Tento přístup usnadňuje vytváření dalších pomocných komponent pro zobrazení pseudokódu. Značně vylepšuje efektivitu při překreslování panelu. Pokud by toto rozšíření nebylo realizováno, nestačilo by pro vykreslení viditelné oblasti projít pouze část dokumentu, která je zobrazena. Vždy by bylo nutné procházet celý dokument od začátku až po konec oblasti, která se vykresluje, a dopočítat čísla řádků.

### 4.5.2 Instrukce algoritmu

Algoritmus je zapsán pomocí posloupnosti instrukcí, které se vykonávají. Každá instrukce je ve formě objektu a poskytuje metodu, jejímž zavoláním dojde k vykonání instrukce. Instrukce poskytují možnosti v nastavování a ovládání všech částí z uživatelského rozhraní simulátoru.

V algoritmu se vyskytují dva základní typy instrukcí. První typ je klasická instrukce, která se vykonává bez vstupu uživatele. Druhým typem je instrukce interaktivní, která pozastaví vykonávání algoritmu a požaduje po uživateli zadání vstupu (často ve formě výběru z grafu). Interaktivní instrukce simulátor provádí pouze, pokud je aktivní interaktivní mód, jinak jsou přeskočeny. Přesný popis řízení intepretace a interaktivních instrukcí bude rozebrán v samostatné kapitole 5 o řídicí jednotce simulátoru.

Speciálními případy jsou instrukce, které realizují skoky a ohraničují určitou sérii instrukcí (jeden krok v simulaci). Instrukce pro skok dovoluje zadat podmínku a návěští, kam se má v případě splnění podmínky skočit. Pro ohraničení jednoho kroku se používá speciální instrukce, která zároveň provede zapsání všech dosud neuložených změn.

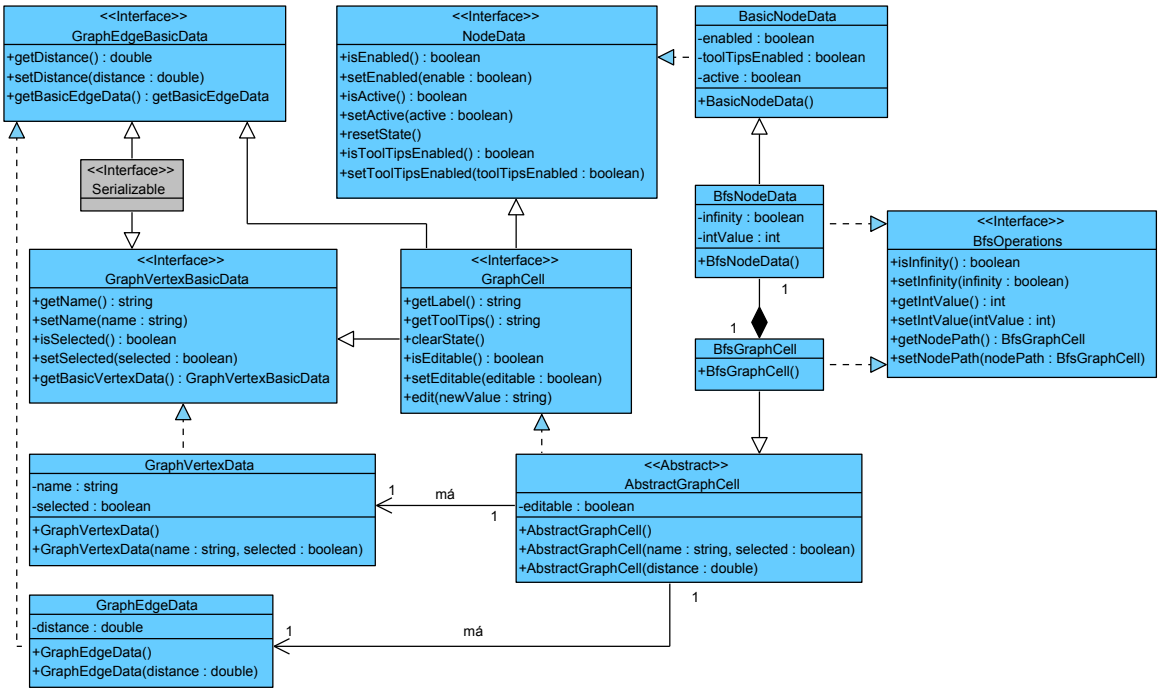
### 4.5.3 Uživatelské objekty v grafu

Při provádění algoritmu se standardně pracuje s objekty, které představují uzly a hrany v grafu. Každý algoritmus si navíc potřebuje ukládat různorodá data při svém běhu. Mohou to být nové hodnoty nejkratších cest, časové známky, případně odkaz na jiný uzel. Navíc je nutné ukládat příznaky, které určují, jakým způsobem bude provedena vizualizace daného uzlu nebo hrany.

Vytvořil jsem třídy, jejichž objekty jsou schopny tyto informace uchovat. Při vytváření bylo využito dědičnosti, aby nevznikal redundantní kód. Třídy jsou v postupné hierarchii od těch, které mají schopnost uložit jen základní informace, až po třídy, které implementují uložení dat podle potřeb konkrétního algoritmu. Využil jsem možnosti, kterou nabízí jazyk Java v podobě realizace rozhraní pro tyto třídy. Odkazování se na objekty pomocí jejich rozhraní umožňuje v budoucnosti snadnější úpravu stávajícího kódu. Výhody tohoto přístupu jsou shrnuty v knížce od Blocha [7].

Diagram tříd je možné vidět na obrázku 4.4. Nejsou zde ovšem znázorněny konkrétní třídy a rozhraní pro všechny algoritmy, ale pouze pro algoritmus **BFS**. Realizace tříd pro ostatní algoritmy je velmi podobná.

Základem jsou objekty, které se ukládají společně s grafem. Ty jsou univerzální a obsahují pouze atributy, které je nutné ukládat. Zvlášť jsem vytvořil třídu a rozhraní pro uzly a zvlášť pro hrany. Tento přístup je nutný z toho důvodu, že při ukládání dochází k převodu




Obrázek 4.4: Diagram tříd uživatelských objektů v grafu pro algoritmus **BFS**

do XML. V XML jsou poté uloženy všechny atributy objektu a navíc také jméno třídy. Při načtení se vytvoří objekt dané třídy s obsahem stejným jako při uložení.

Druhou částí bylo vytvoření tříd, které budou používány pro uložení všech informací za běhu simulátoru. Vytvoření bylo realizováno s inspirací v knihovně **JGraphX**, kde slouží jedna třída (*mxCell*) pro uložení dat uzlu i hrany. Vytvořena je tedy univerzální abstraktní třída *AbstractGraphCell* i rozhraní *GraphCell*. Od této abstraktní třídy dědí ostatní třídy, které poskytují metody a atributy nutné pro konkrétní algoritmus. Každá taková třída poskytuje metody vracející řetězce, které budou zobrazeny ve standardním a kontextovém popisku. Pokud to algoritmus vyžaduje, je zde vytvořena také metoda, která umí načíst textový vstup z grafu. Popis vytvořených tříd je uveden postupně v následujícím seznamu.

- Rozhraní (*GraphEdgeBasicData*, *GraphVertexBasicData*) a třídy (*GraphEdgeData*, *GraphVertexData*), jejichž objekty se ukládají společně s grafem. Obsahují pouze data, která se mají uložit. Standardně jde o jméno uzlu a příznak, zda je uzel počáteční, u hrany je to její váha.

- Jednotné rozhraní (*GraphCell*) a abstraktní třída (*AbstractGraphCell*) pro používání za běhu simulátoru. Jsou zde převážně metody, které jsou využívány v grafu při vizualizaci a interakci.

- Rozhraní (*NodeData*) a třída (*BasicNodeData*) pro uložení základních příznaků uzlů. Konkrétní algoritmus využije této základní implementace a rozšíří ji o atributy a operace, které potřebuje.

- Rozhraní operací (*BfsOperations*) a třídy (*BfsGraphCell*, *BfsNodeData*) pro algoritmus **BFS**. Uzel může mít zadanou celočíselnou hodnotu určující počet hran od počátečního uzlu. Přítomný je i odkaz na předchozí uzel.

- Společné třídy (*DfsGraphEdge*, *DfsGraphVertex*, *DfsNodeData*) a rozhraní (*DfsOperations*) pro algoritmy **DFS**, **SCC** a algoritmus **topologického uspořádání**. Třída pro uzel (*DfsGraphVertex*) má navíc možnosti pro potřeby algoritmu nastavit odkaz na předchozí uzel a první i druhou časovou známku. Vracený textový řetězec pro popis uzlu se skládá ze jména uzlu, první časové známky i druhé časové známky. Třída pro hranu (*DfsGraphEdge*) obsahuje informace o typu hrany.
- Třída (*SccGraphCell*) pro algoritmus **SCC** určená pouze pro graf komponent. Drží si informace o uzlech, které tvoří komponentu.
- Společné třídy (*BellmanFordGraphEdge*, *BellmanFordGraphVertex*, *BellmanFordNodeData*) a rozhraní (*BellmanFordOperations*) pro algoritmy **Bellman-Fordův** a **Dijkstrův**. Je zde plně implementována metoda *edit*, která umí načíst uživatelský vstup pro potřeby interaktivního módu, kdy uživatel zadává novou hodnotu vzdálenosti.

# Kapitola 5

# Řídicí jednotka simulátoru

Řídicí jednotku simulátoru jsem realizoval pro každý algoritmus přesně na míru. Obsahuje v sobě zápis algoritmu pomocí instrukcí uložených v pásce instrukcí. Jednotlivé instrukce slouží k samotnému vykonávání algoritmu a také k ovládání komponent uživatelského rozhraní. Na obrázku 5.1 je možné vidět základní přehled, jaké důležité možnosti jednotlivé části poskytují z pohledu řídicí jednotky. Zároveň je zde také ukázáno, jaký typ informací se předává z komponent do řídicí jednotky a naopak.

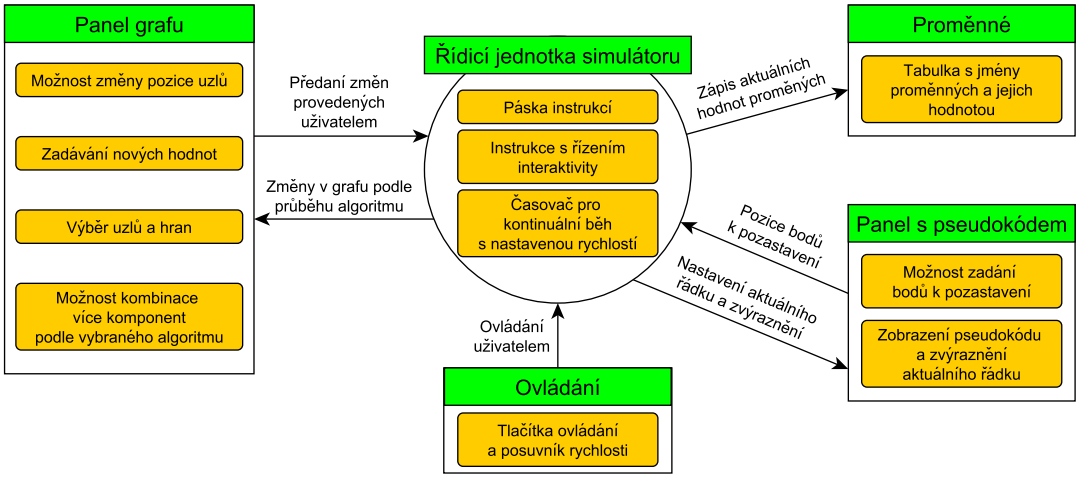
## 5.1 Implementace řídicí jednotky

Implementace je rozdělena do několika tříd, aby nedocházelo k zbytečně velké redundanci kódu. Diagram tříd je možné vidět na obrázku 5.2, kde jsou zachyceny základní třídy a třída pro algoritmus **BFS**. Třídy realizující řídicí jednotky ostatních algoritmů jsou vytvořeny podobným způsobem. Dále je nutné, aby řídicí jednotka mohla ovládat panel proměnných (třída *VariablesPanel*) a panel pseudokódu (třída *VisualizationPseudoCodePanel*). Každá řídicí jednotka navíc může ovládat jeden nebo více panelů pro vizualizaci grafu (třída *AlgorithmGraphComponent*).

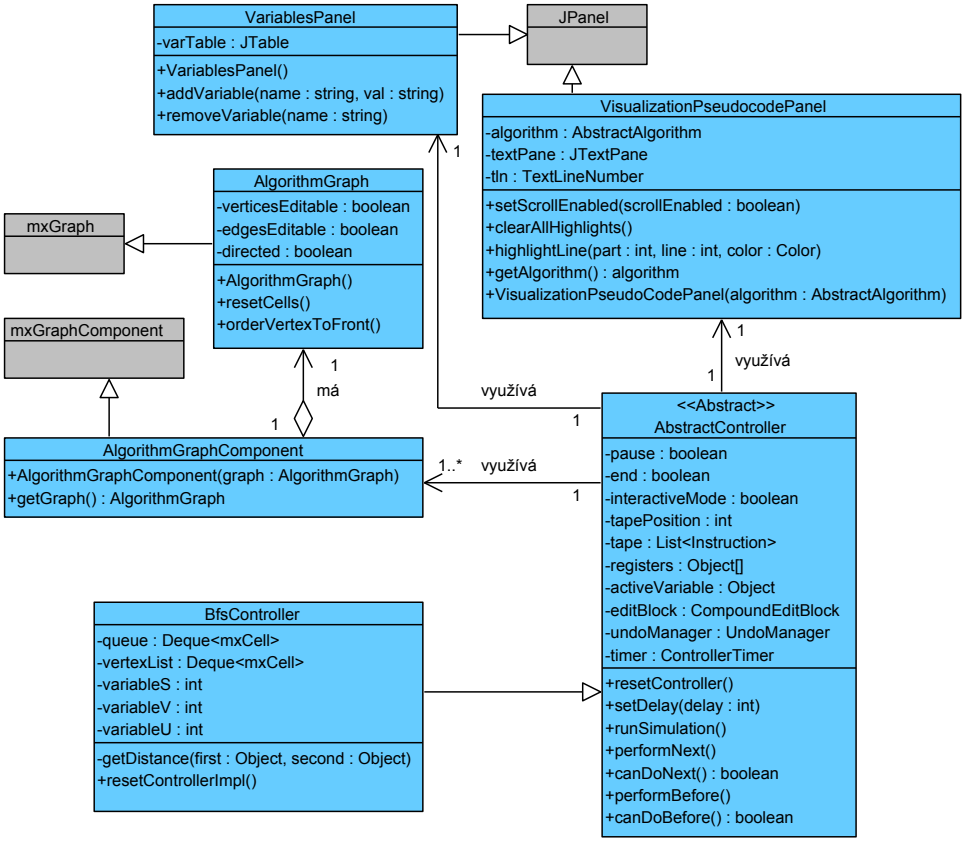
Na vrcholu hierarchie dědičnosti stojí abstraktní třída (*AbstractController*) řídicí jednotky. Tato abstraktní třída obsahuje realizaci základních instrukcí a důležitých metod. V simulátoru se na řídicí jednotku už odkazuje pouze přes rozhraní této hlavní třídy. Jsou zde také implementovány registry (atribut *registers*) v podobě pole objektů, které slouží pro uložení hodnot proměnných algoritmu. Důležitou částí je také páska instrukcí (atribut *tape*), kterou plní až třídy konkrétních algoritmů.

Druhým stupněm jsou abstraktní třídy, které poskytují instrukce a hotové řešení pro konkrétní kategorii algoritmů, např. abstraktní třída algoritmů pro hledání nejkratší cesty z jednoho uzlu do ostatních uzlů. Je možné realizovat libovolné množství tříd tohoto stupně, aby se minimalizovalo množství redundantního kódu. Všechny třídy tohoto stupně by měly být abstraktní.

Poslední stupeň jsou pak již konkrétní třídy pro jednotlivé algoritmy (v diagramu 5.2 například třída *BfsController*). Každá taková třída musí naplnit pásku instrukcí a seznam proměnných. Na proměnné se odkazuje pouze indexem do registrů. Zde již může docházet k určité redundanci, která je cenou za to, že každý algoritmus má různé instrukce interaktivity a trochu jinak řešené určité části. Nejčastěji jsou v těchto třídách implementovány instrukce interaktivity pro daný algoritmus. Instance tříd tohoto stupně se používají jako řídicí jednotka simulátoru.




Obrázek 5.1: Předávané informace a možnosti řídicí jednotky




Obrázek 5.2: Diagram tříd řídicí jednotky pro algoritmus **BFS**

## 5.2 Instrukce s řízením interaktivity

Hlavní činnost interaktivních instrukcí spočívá v pozastavení řídicí jednotky a aktivaci možnosti uživatelského vstupu. Tento typ instrukcí se aktivuje pouze v interaktivním módu, jinak jsou přeskočeny. Vypnutím interaktivního módu je možné přeskočit právě aktivní interaktivní instrukci a pokračovat v simulaci.

Každá interaktivní instrukce je unikátní a realizuje danou část interakce tak, jak je popsána v návrhu. Ve většině případů se jedná o vybrání správného uzlu nebo hrany respektive skupiny uzlů nebo skupiny hran. Pro tyto účely jsem naprogramoval třídu (*SelectCellsMouseAdapter*) pro vybírání v grafu, která umožňuje výběry různých typů prvků. Je možné instanci této třídy nastavit tak, aby povolila pouze výběr uzlů nebo pouze hran popřípadě obojí. Dále umožňuje nastavit i maximální počet vybraných položek v grafu.

Řídicí jednotka pracuje stejným způsobem s běžnou i interaktivní instrukcí. Po převzetí kontroly nad simulací pracuje interaktivní instrukce ve dvou stavech. Pokud je v neaktivním stavu, tak čeká, až bude řídicí jednotkou aktivována. Při aktivaci dochází k některým akcím jako např. spojení instance pro vybírání uzlů a hran s grafem, výpočet očekávaných výsledků a zablokování posunu pásky. Jakmile jsou tyto akce provedeny, přechází instrukce do aktivního stavu, kde porovnává zadaný vstup uživatele s očekávaným (správným) vstupem. Dokud uživatel nezadá akceptovatelný vstup, setrvává instrukce v aktivním stavu. Pokud je zadán vstup od uživatele správně, instrukce deaktivuje všechny prvky interakce, povoluje posun pásky a přechází zpátky do neaktivního stavu.

Chování popsané v předcházejícím odstavci způsobuje problém, pokud uživatel bude chtít v době aktivní interakce provést restart simulátoru nebo vypnutí interaktivního módu. V obou případech musí řídicí jednotka provést kontrolu, zda není právě aktivovaná interaktivní instrukce, a případně zavolat metodu, která ji deaktivuje.

## 5.3 Realizace krokování

Rozdělení algoritmů do kroků jsem implementoval pomocí speciální instrukce (vnořená třída *PauseInstruction* ve třídě *AbstractController*). Tato instrukce je na pásce vždy vložena po sérii instrukcí představujících jeden krok. Její vykonání způsobí uložení všech dosud neuložených akcí a nastaví indikaci pro řídicí jednotku, že byl dokončen krok.

Uložení všech změn je realizováno přes správce editačních akcí (standardně v angličtině jako *UndoManager*), které obsahují jednu metodu pro provedení změn a druhou metodu pro vrácení provedených změn. Každá instrukce, u které je nutné provedené změnit vrátit, vytvoří při svém vykonávání objekt editační akce. Tento objekt potom vykoná danou akci a přidá se do seskupení neuložených akcí.

Seskupení neuložených akcí (atribut *editBlock* ve třídě *AbstractController*)je realizováno jako objekt, který má v sobě uložených více editačních akcí a sám implementuje rozhraní pro editační akci. Pokud je u takového objektu volána metoda pro provedení změn nebo vrácení změn, tak objekt zavolá stejnou metodu u všech v něm uložených editačních akcí. Při příchodu speciální instrukce pro ukončení kroku, dojde k přidání seskupení neuložených akcí do správce editačních akcí a vytvoření nového objektu pro seskupení neuložených akcí.

Jeden krok je potom uložen ve správci editačních akcí jako jedna akce. Každý krok zpět je tedy v řídicí jednotce simulátoru pouze volání jedné metody pro vrácení změn. Realizace kroku dopředu je stejná, pouze se volá metoda pro provedení změn. Pokud jsou ve správci editačních akcí všechny akce provedeny, tak se pokračuje na pásce instrukcí,

dokud se nenarazí na instrukci konce algoritmu (vnořená třída *EndInstruction* ve třídě *AbstractController*).

### 5.3.1 Výhody a omezení krokování

Výhoda výše popsaného přístupu je v rychlosti a v jednoduchosti řešení. Kroky, které již simulátor provedl, se při posunu dopředu nemusí znovu opakovat a pouze se vykoná dříve vytvořená editační akce. Zároveň jsou uložené i všechny interakce, které uživatel vykonal. Je tedy možné vracet se opakovaně dopředu i dozadu bez nutnosti znovu požadovat po uživateli vstupy. Bez této funkcionality by uživateli nestačilo pouhé vypnutí interaktivního módu, protože některé interaktivní akce ovlivňují i průchod algoritmu grafem.

Za nevýhodu je možné označit předem nastavené omezení v podobě maximálního počtu kroků, o které se uživatel může vrátit. Počet těchto kroků přímo odpovídá maximálnímu množství uložených editačních akcí ve správci. Ten problém se projeví, pokud vstupní graf pro demonstraci je relativně velký a algoritmus potřebuje značné množství kroků, než graf zpracuje. Při testování grafů rozsahem vhodných pro demonstrační účely se problémy s limitem počtu kroků neprojevily.

# Kapitola 6

# Testování aplikace

Pro testování jsem vytvořil několik grafů a také využil příklady z předmětu **GAL**. U testovaných algoritmů se mohou výsledky lišit od výsledků uvedených v předmětu **GAL**. Tento rozdíl je způsoben vnitřní strukturou grafu, která způsobuje procházení uzlů v jiném pořadí.

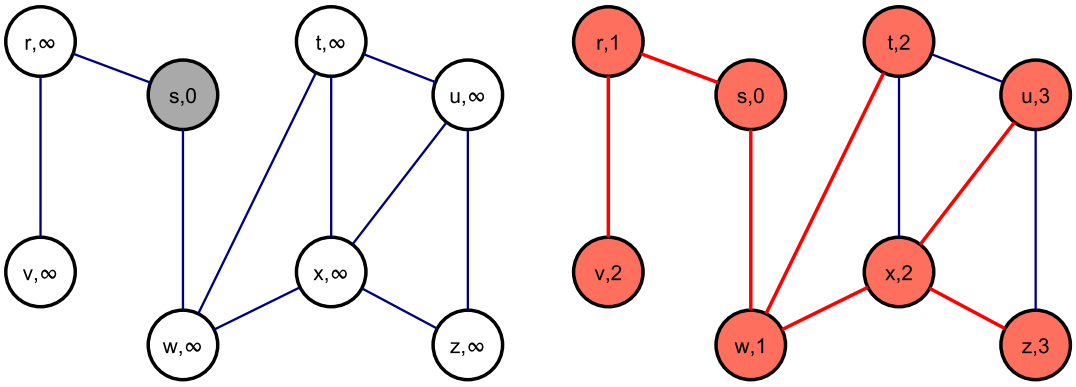
Další nesrovnalost může nastat v důsledku toho, že simulátor dovoluje použít jako vstupní graf i takový graf, který nesplňuje podmínky algoritmu. V takovém případě nemusí algoritmus správně fungovat a produkuje chybné výsledky. Před spuštěním simulace se provádí kontrola grafu a pokud graf podmínky algoritmu nesplňuje je zobrazeno upozornění na potenciální nevalidnost výsledku. Uživatel si může vyzkoušet, jak se algoritmus v takovém případě chová, a sledovat, kde vzniká chyba.

## 6.1 Testování jednotlivých algoritmů

V této sekci jsou pro každý test vždy umístěny obrázky grafů a tabulka. Na obrázcích jsou zobrazeny grafy v takové vizuální podobě, jak je produkuje simulátor. Určité úpravy pro lepší názornost na obrázku byly pouze v případě umístění popisů u hran. V tabulce je zapsána vnitřní struktura grafu a to tím způsobem, že pořadí uzlů v prvním sloupci odpovídá pořadí, v jakém jsou uzly seřazeny pro procházení. Ve druhém sloupci je umístěn seznam sousedů pro uzel z prvního sloupce. Pořadí uzlů v každém seznamu odpovídá také pořadí, v jakém jsou tyto uzly potom navštěvovány.

### 6.1.1 Testování algoritmu BFS

Na obrázcích 6.1 a 6.2 je možné vidět grafy zpracované algoritmem **BFS**. Do uzlů v grafu se doplňuje číselná hodnota představující počet hran na cestě od počátečního uzlu a červeně jsou vyznačeny stromové hrany. Primárně se v tomto algoritmu zpracovávají neorientované grafy, aby vždy došlo k průchodu celým grafem. Na obrázku 6.1 je znázorněn neorientovaný graf a jeho kompletní zpracování. Situaci, kde je použitý orientovaný graf a dva uzly nebyly algoritmem vůbec navštíveny, je možné vidět na druhém obrázku 6.2.

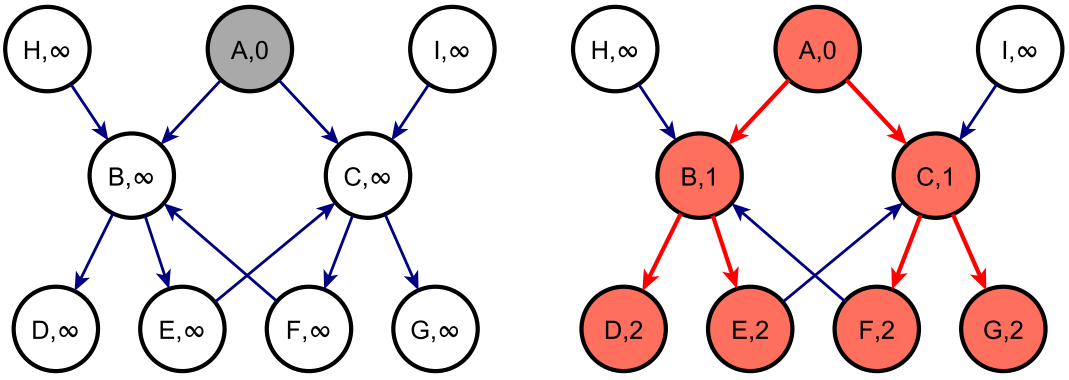



(a) Graf po inicializaci (b) Graf po zpracování

Obrázek 6.1: Neorientovaný graf na počátku a konci průchodu algoritmem **BFS**

| **Uzel** | **Seznam sousedů** |
|---|---|
| s | [w, r] |
| r | [v, s] |
| v | [r] |
| t | [u, x, w] |
| x | [z, u, t, w] |
| w | [x, t, s] |
| u | [z, t, x] |
| z | [u, x] |

Tabulka 6.1: Zápis grafu z obrázku 6.1 v tabulce




(a) Graf po inicializaci (b) Graf po zpracování

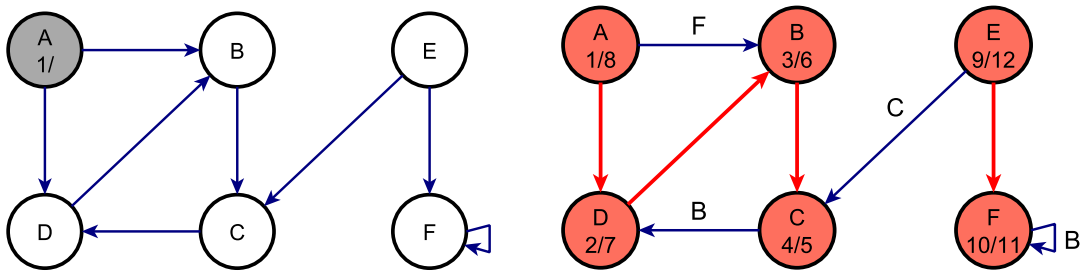
Obrázek 6.2: Orientovaný graf na počátku a konci průchodu algoritmem **BFS**

| Uzel | Seznam sousedů |
|---|---|
| A | [C, B] |
| B | [E, D] |
| C | [G, F] |
| D | [] |
| E | [C] |
| F | [B] |
| G | [] |
| H | [B] |
| I | [C] |

Tabulka 6.2: Zápis grafu z obrázku 6.2 v tabulce

### 6.1.2 Testování algoritmu DFS

Testování algoritmu **DFS** je podobné testování u algoritmu **BFS**. Na obrázku 6.3 je možné vidět standardní vstup **DFS** v podobě orientovaného grafu. Algoritmus **DFS** prochází vždy kompletně celý graf a navštíví všechny uzly. V grafu jsou stromové hrany vyznačeny červeně, ostatní hrany potom mají zapsán svůj typ viz popis v sekci 3.4.2. Pro neorientovaný graf u **DFS** vždy platí, že všechny hrany kromě stromových jsou zpětné, jak je možné vidět i na obrázku 6.4.



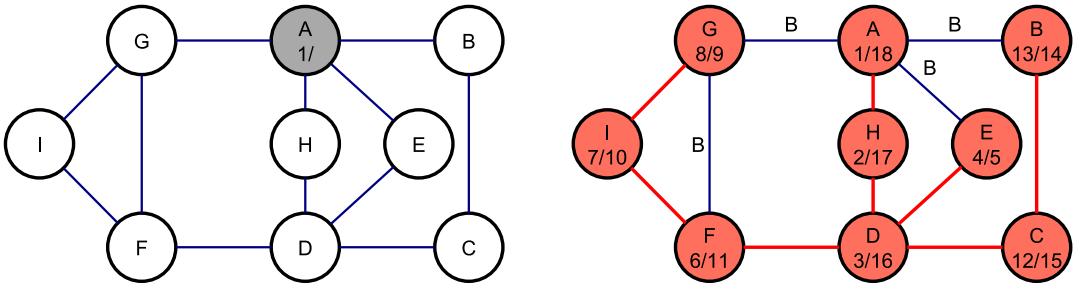

(a) Graf po inicializaci a navštívení prvního uzlu

(b) Graf po zpracování

Obrázek 6.3: Orientovaný graf na počátku a konci průchodu algoritmem **DFS**

| Uzel | Seznam sousedů |
|---|---|
| A | [D, B] |
| B | [C] |
| C | [D] |
| D | [B] |
| E | [F, C] |
| F | [F] |

Tabulka 6.3: Zápis grafu z obrázku 6.3 v tabulce




(a) Graf po inicializaci a navštívení prvního uzlu

(b) Graf po zpracování

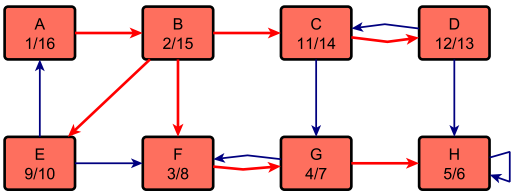
Obrázek 6.4: Neorientovaný graf na počátku a konci průchodu algoritmem **DFS**

| **Uzel** | **Seznam sousedů** |
|---|---|
| A | [H, E, G, B] |
| B | [C, A] |
| C | [D, B] |
| D | [E, F, H, C] |
| E | [A, D] |
| F | [I, G, D] |
| G | [I, A, F] |
| H | [A, D] |
| I | [G, F] |

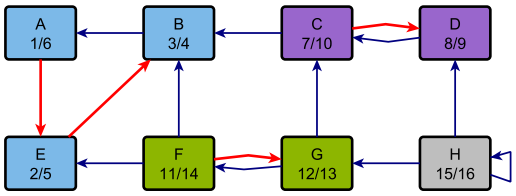
Tabulka 6.4: Zápis grafu z obrázku 6.4 v tabulce

### 6.1.3 Testování algoritmu SCC

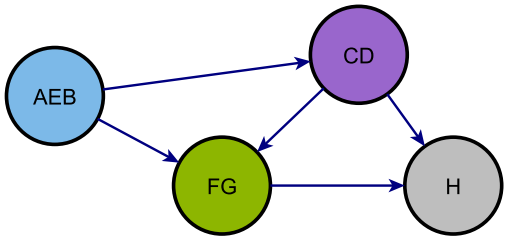
Při testování algoritmu **SCC** bylo nutné kontrolovat i průběhy algoritmu **DFS** na původním a transponovaném grafu. Na obrázcích 6.5a a 6.5b je možné vidět, oba grafy po zpracování algoritmem **DFS**. Jako počáteční uzel je zde vybrán uzel $A$. V transponovaném grafu je vždy počáteční uzel ten, který má největší druhou časovou známku, zde se jedná znovu o uzel $A$. Poslední obrázek je graf komponent 6.5c, stejné barevné rozlišení komponent je možné vidět, už i na obrázku transponovaného grafu 6.5b.




(a) Graf po dokončení pvního volání **DFS**




(b) Transponovaný graf po dokončení algoritmu **SCC**




(c) Výsledný graf komponent

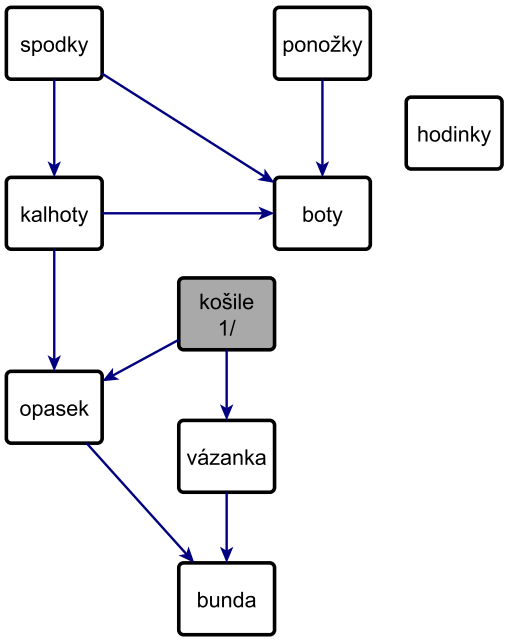
Obrázek 6.5: Ukázka zpracování grafu algoritmem **SCC**

| **Uzel** | **Seznam sousedů** |
|---|---|
| A | [B] |
| B | [F, E, C] |
| C | [G, D] |
| D | [H, C] |
| E | [F, A] |
| F | [G] |
| G | [H, F] |
| H | [H] |

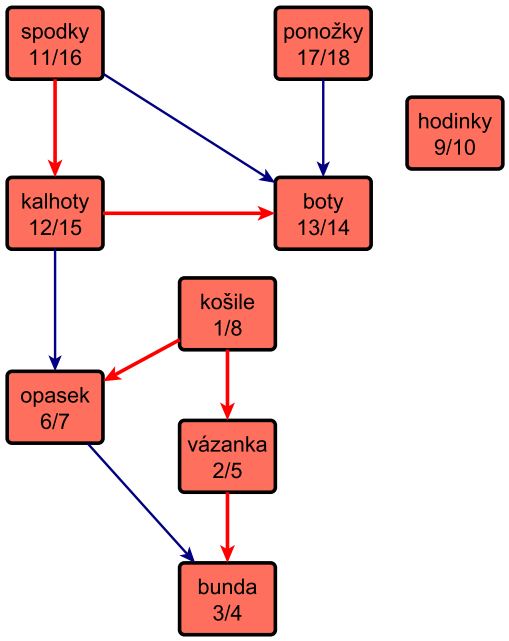
Tabulka 6.5: Zápis grafu z obrázku 6.5 v tabulce

### 6.1.4 Testování algoritmu topologického uspořádání

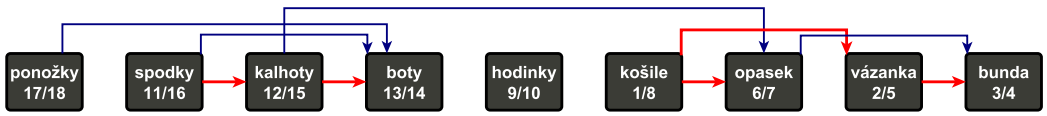
Výsledkem algoritmu **topologického uspořádání** je seznam $L$. Stejně jako v případě algoritmu **SCC** je i zde využíván algoritmus **DFS**. Obrázky 6.6a a 6.6b ukazují graf před a po zpracování algoritmem **DFS**. Na obrázku 6.6c je potom možné vidět seznam $L$ v podobě grafu, který má uzly v jedné linii. Stejně jako v původním grafu zpracovaném pomocí **DFS**, tak i v seznamu $L$ jsou vyznačeny stromové hrany červeně.




(a) Graf po inicializaci a navštívení prvního uzlu




(b) Graf po zpracování algoritmem **topologického uspořádání**




(c) Výsledek v podobě seznamu $L$

Obrázek 6.6: Ukázka zpracování grafu algoritmem **topologického uspořádání**

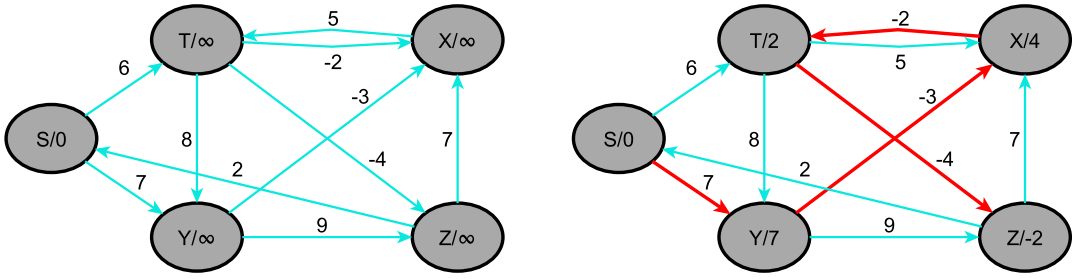
| **Uzel** | **Seznam sousedů** |
|---|---|
| košile | [vázanka, opasek] |
| vázanka | [bunda] |
| bunda | [] |
| opasek | [bunda] |
| hodinky | [] |
| spodky | [kalhoty, boty] |
| kalhoty | [opasek, boty] |
| boty | [] |
| ponožky | [boty] |

Tabulka 6.6: Zápis grafu z obrázku 6.6 v tabulce

### 6.1.5 Testování Bellman-Fordova algoritmu

**Bellman-Fordův** algoritmus dokáže kromě hledání nejkratších cest i detekovat záporný cyklus. Zpracování standardního grafu je možné vidět na obrázku 6.7. Nejkratší cesty do jednotlivých uzlů jsou vyznačeny červěně a u každého uzlu je zapsána délka cesty. Druhý

obrázek 6.8 ukazuje graf, kde se vyskytuje záporný cyklus. Kontrolní podmínku, v případě tohoto grafu, porušuje více hran, zeleně je zvýrazněna pouze první, na kterou algoritmus narazí.

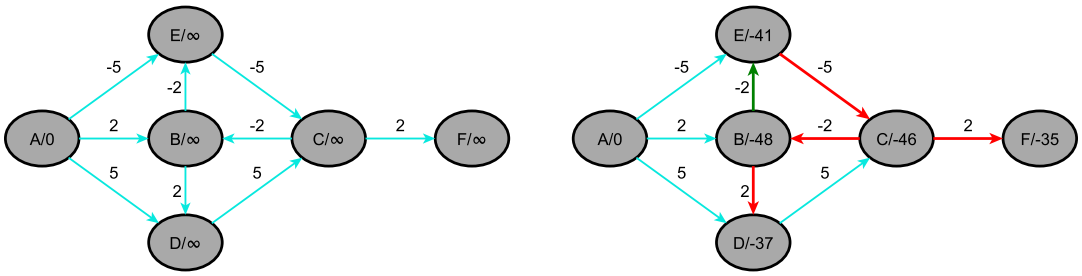



(a) Graf po inicializaci (b) Graf po zpracování

Obrázek 6.7: Ukázka hledání nejkratších cest pomocí **Bellman-Fordova** algoritmu

| **Uzel** | **Seznam sousedů** |
|---|---|
| S | [Y, T] |
| T | [Z, X, Y] |
| Z | [X, S] |
| X | [T] |
| Y | [X, Z] |

Tabulka 6.7: Zápis grafu z obrázku 6.7 v tabulce




(a) Graf po inicializaci (b) Graf po zpracování

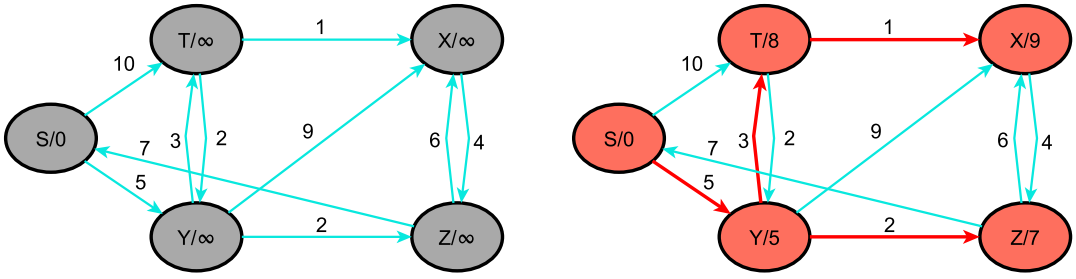
Obrázek 6.8: Ukázka detekce záporného cyklu v grafu pomocí **Bellman-Fordova** algoritmu

| Uzel | Seznam sousedů |
|---|---|
| A | [E, D, B] |
| B | [D, E] |
| C | [B, F] |
| D | [C] |
| E | [C] |
| F | [] |

Tabulka 6.8: Zápis grafu z obrázku 6.8 v tabulce

### 6.1.6 Testování Dijkstrova algoritmu

**Dijkstrův** algoritmus nedisponuje schopností detekovat záporný cyklus, jako je tomu u **Bellman-Fordova** algoritmu. Vznik chyby při hledání nejkratších cest je možný, i pokud vstupní graf obsahuje záporné hrany. Standardní průběh **Dijkstrova** algoritmu lze vidět na obrázku 6.9. Nejkratší cesty jsou zvýrazněny červeně a uzlům se také nastavuje červené zabarvení, pokud jsou přidány do množiny dokončených uzlů $S$. Druhý obrázek 6.10 ukazuje chybu algoritmu při zpracování grafu, kde jsou záporně ohodnocené hrany. Z uzlu $F$ je možné pokračovat do uzlu $B$ s cenou cesty 5 místo nalezené cesty s cenou 7.



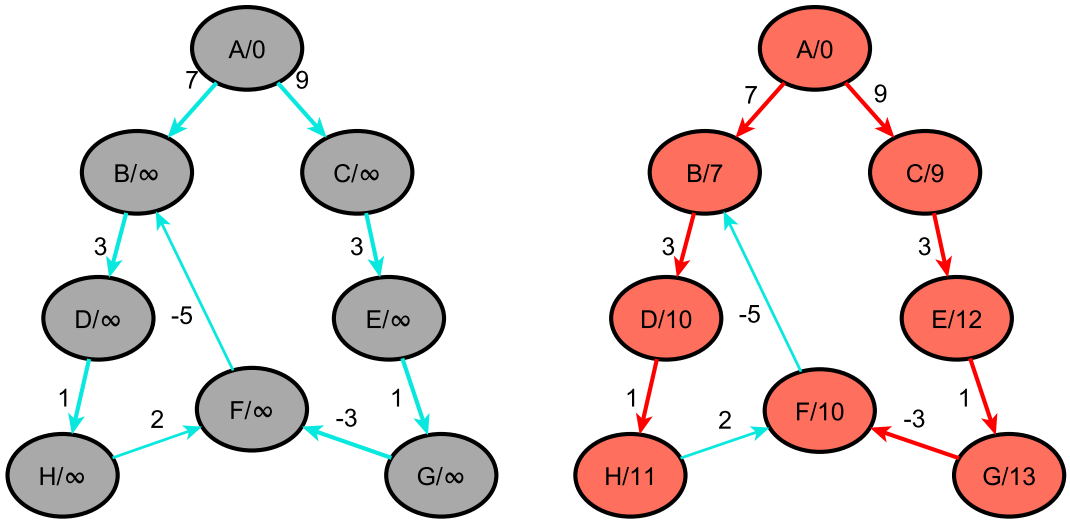

(a) Graf po inicializaci

(b) Graf po zpracování

Obrázek 6.9: Ukázka hledání nejkratších cest pomocí **Dijkstrova** algoritmu

| Uzel | Seznam sousedů |
|---|---|
| S | [Y, T] |
| T | [X, Y] |
| Z | [X, S] |
| X | [Z] |
| Y | [X, Z, T] |

Tabulka 6.9: Zápis grafu z obrázku 6.9 v tabulce




(a) Graf po inicializaci

(b) Graf po zpracování

Obrázek 6.10: Úkázka chyby pří hledání nejkratších cest pomocí **Dijkstrova** algoritmu

| **Uzel** | **Seznam sousedů** |
|---|---|
| A | [C, B] |
| B | [D] |
| C | [E] |
| D | [H] |
| E | [G] |
| F | [B] |
| G | [F] |
| H | [F] |

Tabulka 6.10: Zápis grafu z obrázku 6.10 v tabulce

## 6.2 Testování interakce

Testování interaktivního režimu probíhalo na grafech uvedených v této kapitole. Tyto grafy jsem také upravoval a mírně měnil jejich strukturu. Problém při testování tohoto režimu je ve změně průchodu algoritmu grafem. Dosahuje se potom trochu jiných výsledků a je nutné individuálně kontrolovat jejich správnost. Tento problém se vyskytuje v největším rozsahu u algoritmů **BFS**, **DFS** a **topologického uspořádání**. Při testování jsem zkoušel různé možnosti průchodů algoritmem a odhalil několik problémů a chyb. Nalezené chyby jsem postupně opravil a znovu provedl ověření práce simulátoru.

## 6.3 Zhodnocení aplikace

Cílem vytvořené aplikace bylo didakticky demonstrovat průběh grafových algoritmů s možností interaktivního režimu. Na testovaných grafech se potvrdilo, že aplikace produkuje očekávané výsledky. Není ovšem možné otestovat simulátor pro všechny varianty grafů z toho

důvodu, že je uživateli dána možnost vytvořit si vlastní graf. Simulátor navíc umožňuje použít vstupní graf nesplňující podmínky daného algoritmu.

Při vizualizaci algoritmu, se osvědčil panel pseudokódu. Rozšiřování jednotlivých řádků podle velikosti panelu a pozice aktuálního řádku vždy uprostřed značně vylepšilo přehlednost v pseudokódu. Rozšíření panelu o body pro pozastavení umožnilo nechat zastavit algoritmus přesně na vybraném řádku. Bylo by možné také vytvořit rozšíření, kde by se zadávali i podmínky za jakých se má simulátor zastavit, např. podle hodnoty dané proměnné.

Rychlost simulace je možné zadávat přes posuvník z předvolených rychlostí. Z nabídky rychlostí si uživatel může vybrat takovou rychlost, která odpovídá jeho požadavkům. V simulátoru schází možnost přeskočit přímo na požadovanou část. Pro rychlé přeskočení na konkrétní řádek je možné využít bod pro pozastavení a spustit simulaci maximální rychlostí.

Interaktivní režim byl implementován podle návrhu v sekci 3.4. V testování se ukázalo, že je možné ho plně využít u všech algoritmů. Pozitivní vlastností pro uživatele je možnost vybrat si, v jakém pořadí bude graf algoritmem procházen. Uživatel se také může mnohem lépe seznámit s daným algoritmem a pochopit významy jednotlivých proměnných. Interakce by mohla být také rozšířena tak, aby uživatel prováděl všechny kroky algoritmu. V takovém případě by bylo vhodné uživateli umožnit výběr, jestli chce provádět všechny akce nebo pouze ty významné.

Editor grafů umožňuje uživateli si jednoduše připravit graf pro simulátor. Poskytnuté možnosti v editoru jsou pouze základní, které dostačují pro vytvoření a editaci grafu. Zadávání stylů u grafu vyžaduje od uživatele znalost jednotlivých možností, které zde může zadat.

# Kapitola 7

# Závěr

V této bakalářské práci byla provedena analýza, návrh, implementace a testování aplikace pro demonstraci grafových algoritmů. Tvorba aplikace probíhala v jazyce Java a pro práci s grafy využívá knihovnu **JGraphX**. Celkem je podporována simulace šesti různých algoritmů a u každého z nich je plně funkční interaktivní režim. Uživateli je umožněno spouštět více simulátorů souběžně. Součástí je také rozšíření v podobě editoru grafů, který umožňuje návrh grafu i zadání vizuálních stylů pro simulaci. Testování probíhalo na více než deseti různých grafech s kontrolou, zda simulátor podává správné výsledky.

Jedinečnost aplikace spočívá ve vytvoření interaktivního režimu. Uživateli je dána možnost v klíčových okamžicích průběhu algoritmu převzít kontrolu a vybrat, jak má algoritmus dále pokračovat. Uživatelem provedené akce kontroluje simulátor, zda jsou učiněny v souladu s průběhem algoritmu. V interaktivním režimu je také možné změnit průchod algoritmu grafem oproti běžnému průchodu pouze podle vnitřní reprezentace grafu.

Aplikaci je možné využít pro výuku grafových algoritmů. Přednášejícímu je dána možnost připravit si graf a vysvětlovat průběh algoritmu souběžně s jeho vizualizací. Student potom může aplikaci využít na vlastním počítači a ověřit si své znalosti. Díky uložení grafů v podobě XML souboru je umožněno jednoduché předávání již připravených grafů. Reálné nasazení aplikace je plánováno pro předmět **GAL**.

V budoucím vývoji by bylo možné pokračovat rozšířením podpory pro více grafových algoritmů. Případně do editoru grafů implementovat další funkce a uživatelsky příjemnější zadávání stylů. Vhodným rozšířením může také být implementace ukládání stavu simulace. Pokud by aplikace měla být využitelná i pro uživatele neovládající češtinu, je nutné provést překlad programu a pseudokódů minimálně do anglického jazyka.

# Literatura


[1] KŘIVKA, Zbyněk a Tomáš MASOPUST. *Grafové algoritmy* [online]. 2012 [cit. 2012-12-04]. Dostupné z: https://www.fit.vutbr.cz/study/courses/GAL/public/gal-handouts.pdf

[2] CORMEN, Thomas H. *Introduction to algorithms*. Vyd. 3. Cambridge: MIT Press, 2009, xix, 1292 s. ISBN 978-0-262-03384-8.

[3] JGraphX (JGraph 6) User Manual. JGRAPH LTD. *Support for JGraphX and mxGraph* [online]. 2013 [cit. 2013-02-26]. Dostupné z: http://jgraph.github.io/mxgraph/docs/manual_javavis.html

[4] FREEMAN, Eric a Elisabeth FREEMAN. *Head first design patterns*. Vyd. 1. Sebastopol: O´Reilly Media, 2004, xxxvi, 638 s. ISBN 05-960-0712-4.

[5] SIERRA, Kathy. *Head first Java*. Vyd. 2. Sebastopol: O´Reilly, 2005, xxxii, 688 s. ISBN 05-960-0920-8.

[6] The Swing Tutorial. ORACLE. *Oracle Documentation* [online]. 2012 [cit. 2012-12-27]. Dostupné z: http://docs.oracle.com/javase/tutorial/uiswing/

[7] BLOCH, Joshua. *Effective java*. Vyd. 2. Upper Saddle River: Addison-Wesley, 2008, xxi, 346 s. ISBN 03-213-5668-3.

# Příloha A

# Obsah CD

Jako příloha je k práci přidáno CD, které obsahuje položky uvedené v následujícím seznamu:

- Textová část bakalářské práce v souboru **BP.pdf**.
- Programová dokumentace vygenerovaná ze zdrojových souborů pomocí nástroje javadoc ve složce **doc**.
- Zdrojové soubory textové práce ve složce **latex**.
- Zdrojové soubory aplikace, včetně souboru build.xml pro překlad, ve složce **sources**.
- Přeložená aplikace ve spustitelné formě, včetně spouštěcího souboru pro operační systém MS Windows, ve složce **GraphSimulator**. Součástí jsou také připravené ukázkové grafy.

# Příloha B

# Manuál

Tato příloha obsahuje základní instrukce, jak spustit a používat vytvořenou aplikaci.

## Překlad a spuštění programu

Pro překlad je vyžadována instalace Java Development Kit ve verzi 1.7. Ke zdrojovým souborům aplikace je přidán Ant[1] skript build.xml. Pomocí nástroje Ant je možné provést překlad a vytvořit výsledný jar soubor (příkaz *Ant jar*). Poskytnuta je také možnost vygenerovat programovou dokumentaci ze zdrojových souborů (příkaz *Ant doc*).

Aplikace je plně funkční pod operačním systémem MS Windows 7 Service Pack 1. Na jiných platformách nemusí správně fungovat. Spuštění aplikace vyžaduje minimálně instalaci Java Runtime Enviroment ve verzi 1.7. Skript build.xml poskytuje možnost spustit výsledný jar soubor (příkaz *Ant run*). Pro uživatele operačního systému MS Windows je navíc připraven spustitelný exe soubor vytvořený pomocí nástroje Launch4j[2].

## Editor

Editor slouží zároveň jako hlavní část programu, kde je umístěno menu.

- **Vytváření, načítání a ukládání grafu** – možnosti se nacházejí v menu pod nabídkou *Soubor*. Graf se vždy ukládá i načítá jako XML soubor.
- **Práce s grafem**
  - V grafu je možné přibližovat a oddalovat použitím kolečka u myši a současným držením klávesy Ctrl.
  - U vrcholů se mění pozice a velikost pomocí myši.
  - Změnu hodnoty u hrany i vrcholu je možné provést dvojklikem myši a zadáním nové hodnoty.
  - Vkládání nového vrcholu se děje pomocí kombinace klávesy levý Alt a klikem pravého tlačítka myši. Jméno každého vrcholu musí být v grafu unikátní a neprázdné.
  - Hrana se přidává tažením myši od středu jednoho vrcholu do druhého vrcholu. Editor nepovoluje volné hrany nebo hrany nespojující dva vrcholy.

[1] http://ant.apache.org/
[2] http://launch4j.sourceforge.net/

  - Vymazání vrcholu nebo hrany je možné pomocí klávesy Delete.
  - Uzel je možné pro účely následující simulace označit jako počáteční z nabídky *Editace*. Takový uzel je zobrazen žlutě s červeným obrysem.
  - Je možné používat standardní klávesové zkratky pro kopírování a vložení.

- **Nastavení stylu**
  - Styl se nastavuje v nabídce *Editace* → *Nastavit Styl*
  - Z nabídky se vybírá styl, který se má změnit, a poté se textově upraví. Standardně je možné zadávat hodnoty jako barvu, tloušťku čar a také tvar. Všechny možnosti pro styly jsou k nalezení v příloze D.
  - Při zadávání změn je zobrazen náhled, kde jsou dva vrcholy a jedna hrana viz obrázek 4.2. Vždy se při změně stylu mění pouze hrana a levý vrchol. Pravý vrchol je nastaven na standardní zobrazení uzlu pro tento graf.

- **Změna motivu**
  - Je možné z nabídky *Okno* změnit aktuální motiv pro uživatelské rozhraní. Standardní funkčnost je testována na operačním systému MS Windows. Ostatní motivy slouží jako možnost pro uživatele, pokud by se na jeho konfiguraci zobrazovaly lépe.

## Simulátor

Simulátory jednotlivých algoritmů se spouštějí z hlavního menu pod nabídkou *Simulace*. Jako vstupní graf simulátor použije graf načtený v editoru a spustí simulaci ve vlastním okně.

- **Úpravy a interakce u grafu**
  - Úpravy grafu, přibližování a oddalování jsou stejné jako v případě editoru grafů. V simulační části jsou ovšem vypnuty možnosti pro změnu struktury grafu. Provedené úpravy platí pouze pro aktuální běh simulátoru a nedají se uložit.
  - V interaktivní části se vrcholy a hrany vybírají kliknutím myši na příslušné místo v grafu. Zadávání hodnot se děje pomocí dvojkliku myši. Speciální případ je u algoritmu **DFS**, kde se typ hrany vybírá z kontextového menu přímo u hrany.

- **Panel proměnných**
  - Tažením myši v levé části přejde panel do dialogu. Zavřením dialogu se vrátí do původní polohy.

- **Panel pseudokódu**
  - Pravým tlačítkem myši se vyvolá kontextové menu
  - Body pro pozastavení se zadávají dvojklikem myši do oblasti čísla řádku.
  - Centrování grafu a pseudokódu lze vypnout nebo zapnout z menu.
  - Interaktivní mód se zapíná z menu. Je možné opakované vypnutí a zapnutí v průběhu simulace.

  - Možnost vrátit panely proměnných zpátky do jejich původní polohy je taktéž přítomna v kontextovém menu panelu.

- **Ovládání**
  - K dispozici jsou tlačítka: krok dozadu, restart, běh podle rychlosti a krok dopředu.
  - Pod tlačítky je posuvník rychlosti, kterým lze nastavit prodlevu mezi kroky simulátoru.

# Příloha C

# Pseudokódy algoritmů

**Algoritmus 1** Prohledávání do šířky - BFS($G$, $s$)

Vstupní graf $G$, kde jsou uzly a hrany.
Určený počáteční uzel $s$ v grafu.

```
1: for každý uzel u ∈ V − {s} do
2:     color[u] ← WHITE
3:     d[u] ← ∞
4:     π[u] ← NIL
5: end for
6: color[s] ← GRAY
7: d[s] ← 0
8: π[s] ← ∅
9: ENQUEUE(Q,s)
10: while Q ≠ ∅ do
11:     u ←DEQUEUE(Q)
12:     for každý uzel v ∈ Adj[u] do
13:         if color[v] = WHITE then
14:             color[v] ← GRAY
15:             d[v] ← d[u] + 1
16:             π[v] ← u
17:             ENQUEUE(Q,v)
18:         end if
19:     end for
20:     color[v] ← BLACK
21: end while
```

**Algoritmus 2** Prohledávání do hloubky - DFS($G$)

Vstupní graf $G$, kde jsou uzly a hrany.

1: **for** každý uzel $u \in V$ **do**
2: $color[u] \leftarrow$ WHITE
3: $\pi[u] \leftarrow$ NIL
4: **end for**
5: $time \leftarrow 0$
6: **for** každý uzel $u \in V$ **do**
7: **if** $color[u] =$ WHITE **then**
8: DFS-VISIT($u$)
9: **end if**
10: **end for**

**Algoritmus 3** DFS-VISIT($u$)

Vstupní uzel $u$.

1: $color[u] \leftarrow$ GRAY
2: $time \leftarrow time + 1$
3: $d[u] \leftarrow time$
4: **for** každý uzel $v \in Adj[u]$ **do**
5: **if** $color[v] =$ WHITE **then**
6: $\pi[v] \leftarrow u$
7: DFS-VISIT($v$)
8: **end if**
9: **end for**
10: $color[u] \leftarrow$ BLACK
11: $f[u] \leftarrow time \leftarrow time + 1$

**Algoritmus 4** TOPOLOGICAL-SORT($G$)

Vstupní graf $G$, kde jsou uzly a hrany.

1: zavolej DFS($G$) pro výpočet hodnot $f[v]$
2: každý dokončený uzel zařaď na začátek seznamu uzlů L
3: **return** $L$

**Algoritmus 5** Silně souvislé komponenty - SCC($G$)

Vstupní graf $G$, kde jsou uzly a hrany.

1: zavolej DFS($G$) pro výpočet hodnot $f[u]$
2: vypočítej $G^T$
3: zavolej DFS($G^T$), ale v hlavním cyklu uvažuj uzly v klesajícím pořadí podle hodnoty $f[u]$
4: na výstup dej uzly každého stromu z **DFS** lesa, určeného na řádku 3, jako samostatnou silně souvislou komponentu.

**Algoritmus 6** Inicializace - Initialize-Single-Source($G$, $s$)

Vstupní graf $G$, kde jsou uzly a hrany.
Určený počáteční uzel $s$ v grafu.

1: **for** každý uzel $v \in V$ **do**
2: $d[v] \leftarrow \infty$
3: $\pi[v] \leftarrow$ NIL
4: **end for**
5: $d[s] \leftarrow 0$

**Algoritmus 7** Relaxace hran - Relax($u$, $v$, $w$)

Uzel $u$ na jedné straně hrany.
Uzel $v$ na druhé straně hrany.
Funkce $w$ pro určení délky mezi uzly.

1: **if** $d[v] > d[u] + w(u, v)$ **then**
2: $d[v] \leftarrow d[u] + w(u, v)$
3: $\pi[v] \leftarrow u$
4: **end if**

**Algoritmus 8** Dijkstra($G$,$w$,$s$)

Vstupní graf $G$, kde jsou uzly a hrany.
Funkce $w$ pro určení délky mezi uzly.
Určený počáteční uzel $s$ v grafu.

1: INITIALIZE-SINGLE-SOURCE($G$,$s$)
2: $S \leftarrow \varnothing$
3: $Q \leftarrow V$
4: **while** $Q \neq \varnothing$ **do**
5: $u \leftarrow$ EXTRACT-MIN($Q$)
6: $S \leftarrow S \cup \{u\}$
7: **for** každý uzel $v \in Adj[u]$ **do**
8: RELAX($u$, $v$, $w$)
9: **end for**
10: **end while**

**Algoritmus 9** Bellman-Ford($G$,$w$,$s$)

Vstupní graf $G$, kde jsou uzly a hrany.
Funkce $w$ pro určení délky mezi uzly.
Určený počáteční uzel $s$ v grafu.

```
1: INITIALIZE-SINGLE-SOURCE(G,s)
2: for i ← 1 to n − 1 do
3:     for každou hranu (u, v) ∈ E do
4:         RELAX(u,v,w)
5:     end for
6: end for
7: for každou hranu (u, v) ∈ E do
8:     if d[v] > d[u] + w(u, v) then
9:         return FALSE
10:    end if
11: end for
12: return TRUE
```

# Příloha D

# Styly pro graf

Každý styl se do buňky zapisuje pomocí jména a hodnoty stylu oddělené dvojtečkou. Mezi jednotlivé styly se vkládá středník.
Příklad: `shape:rectangle;fillColor:red`

| **Název stylu** | **Možné hodnoty** | **Popis** |
|---|---|---|
| shape | rectangle, ellipse, doubleEllipse, rhombus, hexagon, triangle, cloud, actor, swimlane, cylinder, label, connector (pro hranu) | tvar vykreslení buňky |
| perimeter | ellipsePerimeter, rectanglePerimeter, rhombusPerimeter, trianglePerimeter, hexagonPerimeter | hranice buňky |
| fillColor, strokeColor, fontColor | zápis je možný jménem barvy v angličtině nebo šestnáctkovým RGB | barvy pro výplň vrcholu, hrany a textu |
| fontFamily | jméno fontu | font pro text v buňce |
| fontSize | celočíselná nezáporná hodnota pro velikost textu | velikost textu pro buňku |
| fontStyle | 1 (bold), 2 (italic), 4 (underline), 8 (shadow), více stylů současně se zadává součtem příslušných hodnot | styl textu v buňce |
| opacity, textOpacity | 0 až 100 | průhlednost buňky a textu |
| rotation | 0 až 360 | rotace buňky |
| align, labelPosition | left, center, right | horizontální zarovnání |
| verticalAlign, verticalLabelPosition | bottom, middle, top | vertikální zarovnání |
| endArrow, startArrow | none, classic, block, open, oval, diamond | tvar šipky u hrany |
| dashed | 0 nebo 1 | přerušovaná čára hrany |
| rounded | 0 nebo 1 | zakulacené rohy u vrcholu |
| direction | north, south, east, west | stanovuje směr vykreslení tvaru buňky |

Tabulka D.1: Základní styly pro buňky v grafu