Univerzita Pardubice
Fakulta elektrotechniky a informatiky

Vizualizace hledání nejkratší cesty na grafech

Bc. Tomáš Havránek

Diplomová práce
2010

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2009/2010**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Tomáš HAVRÁNEK**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Vizualizace hledání nejkratší cesty na grafech**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování:

V úvodní části práce je nutné provést přehled základních pojmů teorie grafů. Popis hledání minimální cesty na grafech (orientovaných i neorientovaných), popis algortimů - Dijkstrův, Floydův a Fordovy metody. Cílem diplomové práce je vytvoření editoru, který umožní grafickou realizaci zadávaného grafu: Editor umožní vypsat jeho vlastnosti - skóre grafu, počet hran, počet vrcholů, zda se jedná o strom, zda jde o Eulerův graf, počet komponent. Průběh jednotlivých algoritmů bude zobrazen po krocích včetně zobrazení dalších hodnot v tabulce. Nedílnou součástí bude i Help a uživatelská příručka.

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

Demel, J., Grafy a jejich aplikace , Academia 2002 Nešetřil, J., Teorie grafů , SNTL 1979 Sedláček, J., Kombinatorika v teorii a praxi , Nakladatelství ČSAV 1964 Nečas, J., Grafy a jejich použití , Polytechnická knižnice, SNTL 1978

Vedoucí diplomové práce: **Ing. Soňa Neradová**
Katedra softwarových technologií

Datum zadání diplomové práce: **30. října 2009**

Termín odevzdání diplomové práce: **21. května 2010**

L.S.

prof. Ing. Simeon Karamazov, Dr.
děkan

doc. Ing. Antonín Kavička, Ph.D.
vedoucí katedry

V Pardubicích dne 10. listopadu 2009

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury. Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 20. 8. 2010

Bc. Tomáš Havránek

ANOTACE

Tato diplomová práce se zabývá tvorbou aplikace pro vizualizaci hledání nejkratší cesty na grafech několika různými algoritmy. V teoretické části jsou shrnuty pojmy teorie grafů a popsány základní algoritmy hledání nejkratších cest na grafech. V další části jsou analyzovány požadavky na aplikaci. Následuje část věnující se implementaci a popisu programu.

KLÍČOVÁ SLOVA

Teorie grafů, graf, nejkratší cesta, Dijkstrův, Floydův, Bellmanův-Fordův, algoritmus, hrana, vrchol

TITLE

Visualisation of shortest path finding on the graphs

ANNOTATION

This thesis deals with making of application for visualisation of shorthest path finding on the graps by several different algorithms. In the theoretical part summarizes terms of graph theory and describes basic algorithms of shorthest path finding. In next section are analyzed requirements for application. Followed by section dedicated to implemetation and description of application.

KEYWORDS

Graphs theory, graph, shortest path, Dijkstra, Floyd, Bellman-Ford, algorithm, edge, vertex

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM POUŽITÝCH ZKRATEK

BMP	Windows Bitmap (formát pro ukládání rastrové grafiky)
DS	Datová struktura
GDI	Graphics Device Interface (aplikační programovací rozhraní)
GPS	Global Positioning System (satelitní navigační systém)
GXL	Graph eXchange Language (jazyk pro popis grafů)
JPEG	Joint Photographic Expert Group (formát pro ukládání rastrové grafiky)
OOP	Objektově Orientované Programování (metodika programování software)
OS	Operační Systém
PNG	Portable Network Graphics (formát pro ukládání rastrové grafiky)
UML	Unified Modeling Language (jazyk pro vizuální modelování)

1 ÚVOD

Hlavním cílem diplomové práce je vytvoření aplikace pro vizualizaci hledání nejkratších cest na orientovaných i neorientovaných grafech několika algoritmy.

V teoretické části práce bude čtenář seznámen se základními pojmy teorie grafů a vybranými algoritmy hledání nejkratších cest na grafech.

V další části diplomové práce bude provedena analýza dostupných řešení pro vizualizaci algoritmů hledání nejkratších cest a analýza požadavků na aplikaci. Na analýzu bude navazovat část věnovaná implementaci, v níž popíši nejzajímavější části aplikace a třídy a datové struktury, jež tyto části používají. Na konci implementační části bude ověřeno splnění požadavků z analytické části práce. V poslední kapitole bude popsáno uživatelské prostředí a ovládání aplikace.

2 ZÁKLADNÍ POJMY TEORIE GRAFŮ

V úvodní kapitole této diplomové práce budou uvedeny základní pojmy z oblasti teorie grafů. Výčet pojmů nebude z daleka kompletní, popíši hlavně ty pojmy, které jsou důležité pro tuto práci a jsou používány v jejích dalších kapitolách. Pro podrobnější seznámení se s teorií grafů odkazuji čtenáře na literaturu [Demel, 2002] a [Volek, 2005], která byla hlavním zdrojem pro tuto kapitolu.

2.1 GRAF

Graf je složen z množin vrcholů a hran, kde každá hrana spojuje dva vrcholy, ovšem ne každý vrchol musí být nutně spojen s jiným. Hrany mohou být orientované nebo neorientované, pokud je hrana orientovaná, jeden z jejích vrcholů je počáteční a druhý koncový. Z počátečního vrcholu hrana vychází a do koncového vstupuje. Neorientované hrany nerozlišují počáteční a koncový vrchol.

Grafy, ve kterých jsou všechny hrany orientované, nazýváme orientovanými grafy. Naopak grafy s neorientovanými hranami nazýváme neorientované grafy. Existují také grafy, jež obsahují oba druhy hran, ty jsou označeny jako smíšené grafy.

2.1.1 Neorientovaný graf

Neorientovaný graf je uspořádaná trojice $G = (V, E, \varepsilon)$. Prvky množiny V se nazývají *vrcholy* grafu G, prvky množiny E se nazývají *hrany* grafu G a ε je zobrazením, které nazýváme incidencí grafu G. *Incidence* ε grafu přiřazuje každé hraně jedno až dvouprvkovou množinu vrcholů. Tyto vrcholy nazýváme *krajní vrcholy hrany* e a říkáme o nich, že s hranou *incidují* nebo že jsou s hranou *incidentní*. Naopak o hraně e tvrdíme, že je s vrcholy *incidentní*, či že je *spojuje*.

Pokud hrana e inciduje jen s jedním vrcholem nazýváme ji *smyčkou*. Může také nastat případ kdy je několik různých hran incidentních se stejnými vrcholy, tedy pro hrany e_1, e_2 platí $\varepsilon(e_1) = \varepsilon(e_2)$. Takové hrany nazýváme *násobné* nebo *rovnoběžné*.

Vrchol, který není incidentní s žádnou hranou, nazýváme *izolovaný vrchol*. Graf může obsahovat prázdnou množinu hran E.

2.1.2 Orientovaný graf

Orientovaný graf je tvořený uspořádanou trojicí $G = (V, E, \varepsilon)$, kde prvky množiny V nazýváme vrcholy, prvky množiny E orientované hrany a zobrazení $\varepsilon: E \rightarrow V^2$, které nazýváme vztahem incidence. Zobrazení přiřazuje každé hraně e grafu G uspořádanou dvojici vrcholů (v_1, v_2). První vrchol z této dvojice v_1 nazýváme počáteční vrchol hrany e a označujeme jej $Pv(e)$. Druhý vrchol v_2 nazýváme koncový vrchol hrany e a značíme $Kv(e)$.

O *orientované hraně* tvrdíme, že vede z vrcholu v_1 do vrcholu v_2, nebo stejně jako u neorientovaného grafu, že vrcholy spojuje. O vrcholech v_1, v_2 říkáme, že jsou krajními vrcholy a jsou incidentní s hranou e a naopak hrana e je incidentní s vrcholy v_1 a v_2

I u orientovaného grafu se můžeme setkat s hranami, jejichž počáteční a koncový vrchol je totožný tj. smyčkami a také rovnoběžnými (násobnými) hranami e_1, e_2 pro něž platí rovnost počátečních a koncových vrcholů, tedy $Pv(e_1) = Pv(e_2)$ a $Kv(e_1) = Kv(e_2)$. Stejně tak s izolovanými vrcholy - vrcholy neincidujícími s žádnou hranou.

2.1.3 Multigraf, prostý graf.

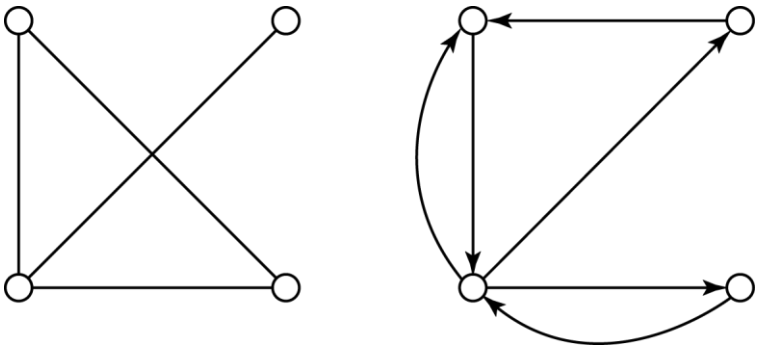
Graf, který obsahuje rovnoběžné nebo násobné hrany nazýváme *multigraf*. Naopak graf, ve kterém nejsou násobné hrany přípustné nebo násobnost každé hrany je rovná nanejvýš jedné, nazýváme *prostý graf*.

2.1.4 Ohodnocený graf

Ohodnoceným grafem rozumíme graf, jehož prvky (hrany, vrcholy) jsou nějakým způsobem nejčastěji číselně ohodnoceny. Toto ohodnocení přidává do grafu informace, které by šlo jinak popsat jen těžce, nebo zbytečně složitě. Například náklady, jež je nutné vynaložit na průchod hranou nebo vrcholem, délku potrubí, propustnost hrany, důležitost komunikace aj. Takovýto graf nazýváme ohodnocený graf. Ohodnocené grafy můžeme ještě dělit na hranově ohodnocený, vrcholově ohodnocený, nebo hranově a vrcholově ohodnocený graf.

2.1.5 Zobrazování grafů

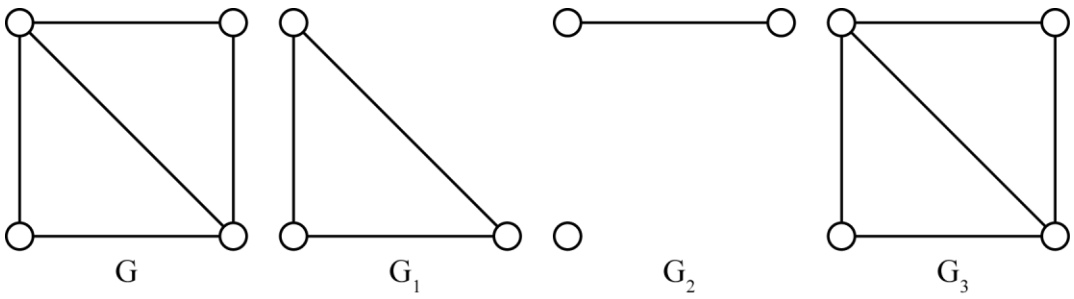
Grafy zobrazujeme kreslením na dvou rozměrnou plochu, není to ale podmínkou existují i grafy, jež se kreslí v trojrozměrném prostoru. Vrcholy kreslíme jako kroužky nebo kolečka a hrany jako úsečky (křivky pokud je hran mezi dvěma vrcholy více, nebo se jedná o smyčku) spojující dva incidentní vrcholy. Úsečky (křivky) navíc podle typu grafu doplňujeme o šipku ve směru od počátečního ke koncovému vrcholu označující orientaci hrany. Příklady kreslení grafů jsou na obrázku 1.



Obrázek 1: Kreslení grafů – zleva neorientovaný prostý graf, orientovaný multigraf

2.1.6 Podgraf

Podgraf je G' grafu G je graf, který vznikne vynecháním některých nebo žádných vrcholů a/nebo hran grafu G. Matematicky zapsáno podgraf grafu $G = (V, E, \varepsilon)$ je graf $G' = (V', E', \varepsilon')$ pro jehož množinu vrcholů platí $V' \subseteq V$, pro množinu hran platí $E' \subseteq E$ a pro všechny hrany $\varepsilon'(e) = \varepsilon(e)$. Zmiňme také, že graf samotný je svým vlastním podgrafem, jak je možné vidět na obrázku 2, kde je nakreslen graf G a grafy G_1, G_2 a G_3 jsou příklady jeho podgrafů.



Obrázek 2: Graf G a příklady jeho podgrafů

2.2 SLEDY

Neorientovaným sledem je střídavá posloupnost vrcholů a hran $v_0, e_1, v_1, e_2, v_2, \dots, e_k, v_k$, kde je každá hrana e_i incidentní s vrcholy v_{i-1} a v_i. Sled nazýváme *orientovaným*, pokud všechny hrany v jeho posloupnosti $v_0, e_1, v_1, e_2, v_2, \dots, e_k, v_k$ jsou orientované. Pro hrany orientovaného sledu platí podmínka, že každá hrana e_i musí být orientována ve směru sledu tedy, že $Pv(e_i) = v_{i-1}$ a $Kv(e_i) = v_i$. U neorientovaného sledu na orientaci hran nezáleží, proto jej lze také označit jako neorientovaný sled.

2.2.1 Tah, cesta

Orientovaný (neorientovaný) tah je orientovaný (neorientovaný) sled, pro který platí, že se v něm žádná hrana neopakuje. *Orientovanou (neorientovanou) cestou* nazýváme orientovaný (neorientovaný) sled (tah), ve kterém se neopakuje žádný vrchol.

2.2.2 Uzavřený sled, tah, cesta

Sled nazveme *uzavřeným*, pokud jsou jeho počáteční a koncový vrchol totožné. Obdobně *uzavřený tah* je nazýván uzavřený sled, ve kterém se neopakují hrany. *Uzavřená cesta* je uzavřený sled, v němž se neopakují hrany a mimo počátečního a koncového žádné vrcholy. Všechny výše jmenované sledy mohou být orientované nebo neorientované. Pro uzavřenou neorientovanou cestu se používá pojem *kružnice* a pro uzavřenou orientovanou cestu pojem *cyklus*.

2.2.3 Eulerovský tah

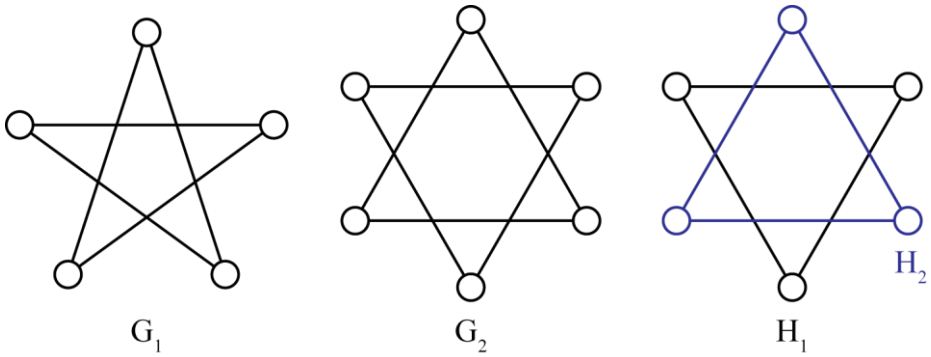
Eulerovský tah je tah, který obsahuje všechny hrany grafu. Eulerovské tahy dělíme podle orientace hran na orientované a neorientované a podle totožnosti výchozího a koncového vrcholu na otevřené a uzavřené. *Uzavřený eulerovský* tah začíná a končí v témže vrcholu, naopak *otevřený eulerovský* tah začíná a končí v různých vrcholech.

2.2.4 Souvislost grafu, komponenta souvislosti

Graf je *souvislý*, pokud mezi každou dvojicí jeho vrcholů existuje neorientovaná cesta. Příklad souvislého grafu G_1 a nesouvislého grafu G_2 je na obrázku č. 3. U orientovaných grafů ještě rozlišujeme tzv. silnou souvislost. Orientovaný graf je *silně souvislý* v případě,

že mezi všemi dvojicemi jeho vrcholů v_1, v_2 existuje orientovaná cesta v obou směrech, tedy z v_1 do v_2 i zpět.

Komponentou souvislosti grafu nazýváme podgraf H grafu G, který je maximálně souvislý, tj. neexistuje další větší část grafu, se kterou by byl souvislý. Například na obrázku číslo 3 jsou zobrazeny komponenty souvislosti grafu G_2 černě komponenta (maximálně souvislý podgraf) H_1 a modře komponenta H_2.



Obrázek 3: Souvislý gr. G_1, nesouvislý gr. G_2, komponenty souvislosti H_1 a H_2 grafu G_2

2.3 DALŠÍ POJMY TEORIE GRAFŮ

V tomto oddíle budou popsány další pojmy z teorie grafů, které nemohly být popsány dříve, než byly definovány pojmy z oddílů předcházejících.

2.3.1 Stupeň vrcholu

Stupeň vrcholu $d_G(v)$ je počet hran incidujících s vrcholem v, pokud s vrcholem incidují smyčky tak se započítávají dvakrát. U orientovaného grafu lze stupeň vrcholu také definovat jako součet *výstupního* stupně vrcholu $d_G^+(v)$ a vstupního stupně vrcholu $d_G^-(v)$, tj. $d_G(v) = d_G^-(v) + d_G^+(v)$. *Výstupní stupeň vrcholu* $d_G^+(v)$ je počet hran vystupujících z vrcholu v a *vstupní stupeň vrcholu* $d_G^+(v)$ je počet hran vstupujících do vrcholu v.

2.3.2 Skóre grafu

Termínem *skóre grafu* nazýváme libovolně řazenou posloupnost stupňů vrcholů grafu. Skóre zapisujeme do kulatých závorek takto: $D_G = \left(d_G(v_1), d_G(v_2), \ldots, d_G(v_n)\right)$, např. skóre grafu G z obrázku 2 může být zapsáno několika způsoby a stále se jedná o stejné skóre: $D_G = (2,3,2,3)$, $D_G = (2,2,3,3)$. Platí, že dvě skóre považujeme za stejná, pokud jedno dokážeme převést na druhé přerovnáním (permutací) pořadí čísel. [Wiki01]

2.3.3 Strom

Strom je graf, který neobsahuje kružnici a je souvislý. Pro stromy platí, že mezi každou dvojicí vrcholů existuje právě jedna cesta. Dále má pro stromy platnost takzvaný Eulerův vzorec pro počty hran a vrcholů: $|E| = |V| - 1$, kde $|E|$ je mohutnost množiny (počet) hran grafu a $|V|$ mohutnost množiny (počet) vrcholů grafu. Z Eulerova vzorce pro stromy tedy vyplývá, že počet hran stromu je roven počtu vrcholů stromu minus jedna. [Jirovský, 2008]

2.3.4 Eulerovský graf

Neorientovaným eulerovským grafem nazýváme neorientovaný graf, ve kterém existuje neorientovaný uzavřený eulerovský tah. Ten v neorientovaném grafu G existuje právě tehdy, pokud je graf G souvislý a každý jeho vrchol má sudý stupeň. Orientovaný graf, v němž existuje orientovaný uzavřený eulerovský graf nazýváme *orientovaným eulerovským grafem*. Orientovaný uzavřený eulerovský tah v grafu existuje právě tehdy, když je graf G souvislý a pro každý jeho vrchol v_i platí rovnost výstupního a vstupního stupně $d_G^+(v_i) = d_G^-(v_i)$. Eulerovské grafy je možné nakreslit jedním tahem, aniž bychom kteroukoliv hranu grafu prošli dvakrát a nakonec se vrátit do původního vrcholu.

3 HLEDÁNÍ NEJKRATŠÍCH CEST

Jedním z nejčastějších problémů, na jehož řešení se aplikuje teorie grafů, je hledání nejkratší cesty. Algoritmy nalezení nejkratší cesty se využívají v mnoha oborech lidské činnosti, jako příklad uvedu plánování nejrychlejší nebo nejkratší trasy mezi dvěma městy v plánovači cest na internetu nebo v GPS navigaci, vyhledávání jízdních řádů, směrování paketů v počítačových sítích a jiných dalších oblastech. Vybrané algoritmy hledání nejkratších cest budou popsány v této kapitole.

Před popisem vybraných algoritmů pro hledání nejkratší cesty je nutné definovat několik pojmů. Popisované algoritmy hledání nejkratší cesty pracují s hranově ohodnoceným grafem, ve kterém ohodnocení hrany e je reálné číslo a značíme jej $a(e)$. Ohodnocení hrany nazýváme *délkou hrany*. *Délka cesty* je součet délek (ohodnocení) hran, které se nacházejí na cestě. *Nejkratší cestou* mezi vrcholy v_p a v_k nazýváme cestu s nejmenší délkou. Délka nejkratší cesty mezi vrcholy v_p a v_k se rovná hodnotě *vzdálenosti* $u(v_p, v_k)$ vrcholů v_p, v_k. Pokud mezi vrcholy v_p a v_k neexistuje žádná cesta, definujeme vzdálenost $u(v_p, v_k) = \infty$. V případě hledání nejkratší cesty na neohodnoceném grafu, je délka cesty reprezentovaná počtem hran na cestě, nebo převedením grafu na ohodnocený, ve kterém každá hrana má délku přesně 1. Pro použití v algoritmech si ještě definujeme *délku aktuálně nejkratší nalezené cesty* $U(v)$ mezi počátečním vrcholem v_p a vrcholem v. Pokud mezi vrcholy v_p a v žádná cesta neexistuje, nebo jsme žádnou nenalezli je $U(v) = \infty$.

Hledat nejkratší cesty lze i v obecně hranově ohodnoceném grafu, jehož hrany mohou být ohodnocené i záporně. Představa záporné délky hrany vypadá nesmyslně, ale nesmysl to rozhodně není. Ohodnocení hran můžeme považovat za náklady nutné na průchod hranou. V tomto případě bude po průchodu hranou se záporným ohodnocením naopak zaplaceno nám. Jen některé z dále uvedených algoritmů lze pro hledání nejkratších cest s obecně ohodnoceným grafem použít. Zda ano bude uvedeno u každého algoritmu.

Pokud graf obsahuje cyklus se zápornou délkou neboli záporný cyklus nelze nejkratší (nejlevnější) cestu dále popsanými algoritmy nalézt, protože průchod cyklem bude zlepšovat délku „cenu“ a to při každém průchodu. [Černý, 2008]

3.1 DIJKSTRŮV ALGORITMUS

Algoritmus představil v roce 1959 nizozemský informatik Edsger Dijkstra. Je určený pro hledání nejkratších cest v grafech s nezáporným ohodnocením hran. Algoritmus najde všechny nejkratší cesty mezi daným počátečním vrcholem v_p a všemi ostatními dosažitelnými vrcholy v grafu G. Může být také použit pro hledání nejkratší cesty mezi dvěma konkrétními vrcholy grafu v_p a v_k. V tom případě se průběh algoritmu přeruší ve chvíli, kdy najde nejkratší cestu do zadaného cílového vrcholu v_k a další cesty již nehledá. Pokud hledáme všechny nejkratší cesty z jednoho počátečního vrcholu, má algoritmus odhad časové složitosti $O(|V|^2 + |E|)$. V případě, že hledáme nejkratší cesty mezi všemi vrcholy grafu, musíme algoritmus použít pro každý vrchol grafu a odhad časové složitosti vzroste na $O(|V|^3 + |E|)$. Dijkstrova algoritmu se využívá například v síťových směrovacích protokolech Open Shortest Path First (OSPF) a Intermediate system to Intermediate System (IS-IS). [Wiki02]

3.1.1 Popis Dijkstrova algoritmu

Vstupem algoritmu je počáteční vrchol v_p, graf G pro který platí, že každá hrana $e \in E$ má ohodnocení $a(e) \geq 0$. Výstupem algoritmu je pro každý vrchol v hodnota $U(v)$ a $ODKUD(v)$, která uchová informaci o předchůdci vrcholu v na cestě z vrcholu v_p. Jako hodnotu předchůdce $ODKUD(v)$ vrcholu v můžeme uchovávat buď incidentní hranu, po které vede do vrcholu nejkratší cesta, nebo předcházející vrchol na nejkratší cestě z vrcholu v_p. Pomocnou proměnnou algoritmu je množina D definitivně ohodnocených vrcholů, tedy vrcholů, jejichž hodnota $U(v)$ se již měnit nebude.

Krok 1 – inicializace. Při inicializaci se pro všechny vrcholy $v \in V$ grafu G nastaví hodnota $U(v) = \infty$, mimo počátečního vrcholu v_p, kterému se nastaví $U(v_p) = 0$ a množina D se inicializuje jako prázdná $D = \emptyset$.

Krok 2 – výběr vrcholu. Z množiny $V \backslash D$ vybereme vrchol v s nejmenší hodnotou $U(v)$. Pokud hodnota $U(v) = \infty$ algoritmus přerušíme, protože do dalších vrcholů již cesta neexistuje. Jinak vrchol v zařadíme do množiny definitivně ohodnocených vrcholů D a postoupíme ke kroku 3.

Krok 3 – relaxace hran vrcholu v. Pro každou hranu $e \in E(v)$ vrcholu v v případě neorientovaného grafu, resp. každou hranu $e \in E^+(v)$ vystupující z vrcholu v pokud se jedná o orientovaný graf, aplikujeme krok 3a a po zpracování všech hran algoritmus pokračuje znovu krokem 2.

Krok 3a. Zkoumáme platnost $U(v_e) > U(v) + a(e)$. Pokud tvrzení platí, změníme hodnotu $U(v_e) = U(v) + a(e)$ a nastavíme předchůdce vrcholu $ODKUD(v_e) = e$ nebo $ODKUD(v_e) = v$ podle toho, jak jsme se rozhodli informace o předchůdci uchovávat. Zkoumaný vrchol v_e je u orientovaného grafu $v_e = Kv(e)$ a neorientovaného grafu $v_e = \varepsilon(e)\backslash\{v\}$.

V případě, kdy hledáme nejkratší cestu mezi dvěma konkrétními vrcholy v_p a v_k můžeme algoritmus ukončit ve druhém kroku ve chvíli, kdy vybereme vrchol v_k a zařadíme jej do množiny definitivně ohodnocených vrcholů D.

Pokud je algoritmus implementován s využitím datové struktury halda jako prioritní fronty pro uchování a výběr vrcholů s nedefinitivním ohodnocením, lze snížit jeho odhad časové složitosti až na $O((|E| + |V|)\log|V|)$ v případě použití binární haldy nebo $O(|E| + |V|\log|V|)$ pokud je použita Fibonacciho halda. [Wiki02]

Následuje příklad implementace algoritmu v pseudokódu s využitím DS halda:

```
funkce DijkstraAlg(G, vp)
start
     pro všechny vrcholy v uvnitř G:
          U(v):= nekonečno;
          ODKUD(v):= nic;
          HALDA.PRIDEJ(v);
     U(vp):= 0;
     dokud HALDA není prázdná
          v:= HALDA.ODEBER_VRCHOL_S_NEJMENŠÍM_U();
          když U(v)= nekonečno:
               přeruš_cyklus;
          pro všechny hrany e vrcholu v:
               ve:= KONCOVÝ_VRCHOL(e);
               když U(ve) > U(v) + a(e):
                    U(ve):= U(v) + a(e);
                    ODKUD(ve):= v;
     vrať U, ODKUD;
konec
```

3.1.2 Rekonstrukce nejkratší cesty u Dijkstrova algoritmu

Když máme vypočítány nejkratší cesty z vrcholu v_p, bude nás pravděpodobně také zajímat, kudy tyto cesty vedou. U Dijkstrova algoritmu lze orientovaný sled zrekonstruovat snadno, protože u každého vrcholu disponujeme informací, který vrchol je jeho předchůdcem resp. přes kterou hranu do něj vede cesta. Jako vstup pro rekonstrukci cesty tedy potřebujeme znát hodnoty předchůdců $ODKUD(v)$ pro všechny vrcholy a cílový vrchol v_c do kterého chceme nejkratší cestu zrekonstruovat. Rekonstrukce poté probíhá následovně:

Krok 1 – inicializace. Pokud $U(v_k) = \infty$, ukončíme rekonstrukci, protože mezi vrcholy v_p a v_k cesta neexistuje. Jinak nastavíme pomocnou hodnotu aktuálního vrcholu $v = v_k$ a inicializujeme posloupnost vrcholů na cestě $c = \{v\}$.

Krok 2 – rekonstrukce. Aktualizujeme hodnotu vrcholu $v = ODKUD(v)$ a na první místo posloupnosti vrcholů c zařadíme aktuální vrchol v. Jestliže platí $v \neq v_p$, opakujeme krok 2.

Po rekonstrukci máme posloupnost vrcholů c v pořadí nejkratší cesty. V případě, že bychom jako hodnoty $ODKUD(v)$ uchovávali hrany, ze kterých se do vrcholu přišlo, musíme do posloupnosti zařazovat počáteční vrcholy $v = Pv(e)$ hran $e = ODKUD(v)$. Do posloupnosti c je možné zařadit i hrany a získat tak kompletní cestu.

3.2 FLOYDŮV ALGORITMUS

Algoritmus je známý také jako Floydův-Warshallův, Royův-Floydův nebo algoritmus WFI. Ve své současné formě byl představen Robertem Floydem v roce 1962. Název algoritmu je spjat se třemi jmény, protože se jedná o v podstatě stejný algoritmus pro hledání tranzitivního uzávěru binární relace, který nezávisle na sobě představili v roce 1958 Bernard Roy a v roce 1962 Stephen Warshall. [Weiss, 2010]

Algoritmus slouží k nalezení nejkratších cest mezi všemi dvojicemi vrcholů v_i, v_j v hranově ohodnoceném grafu G, který neobsahuje záporné cykly. To algoritmus zvládne při jediném průchodu. Na rozdíl od Dijkstrova jej lze použít i na grafy jejichž hrany mají záporné ohodnocení. Odhad časové složitosti algoritmu je $O(|V|^3)$.

3.2.1 Popis Floydova algoritmu

Algoritmus pracuje tak, že porovnává všechny možné cesty v grafu pro každou dvojici vrcholů grafu. Vstupem algoritmu je *matice délek hran* A a výstupem *matice vzdáleností* U. Matice délek hran A je čtvercová matice o rozměrech $|V| \times |V|$, prvky matice $a(i,j)$ mají hodnotu délky nejkratší hrany mezi vrcholy v_i a v_j. Pokud mezi vrcholy v_i, v_j žádná hrana neexistuje je $a(i,j) = \infty$. Délka nejkratší hrany je rovna nule pokud $i = j$, tedy $a(i,i) = 0$. Z této matice poté algoritmus počítá posloupnost matic $U_0, U_1, U_2, \dots, U_n$, ve které $n = |V|$. V této posloupnosti každá matice U_k, má vlastnost $u_k(i,j)$ pro kterou platí, že je rovna nejkratší cestě z v_i do v_j procházející přes vrcholy $v_1, v_2, v_3, \dots, v_k$ a v_i, v_j. Matice $U_0 = A$ to znamená, že obsahuje nejkratší délky cest, které neprochází přes jiný než počáteční a cílový vrchol. Naopak v matici U_n jsou délky nejkratších cest, které procházejí přes libovolný vrchol grafu, a proto se jedná o matici vzdáleností U, která je výstupem tohoto algoritmu. Pro výpočet prvků matice U_k platí následující vztah:

$$u_k(i,j) = \min\{u_{k-1}(i,j), u_{k-1}(i,k) + u_{k-1}(k,j)\}.$$

Je tedy nutné znát předchozí matici U_{k-1}. Při výpočtu prvků matice se zkoumá, zda mezi vrcholy v_i a v_j nevede kratší cesta přes vrchol v_j. Pokud ano nastaví se délka aktuálně nejkratší cesty z vrcholu v_i do v_j jako součet délek aktuálně nejkratších cest mezi vrcholy v_i a v_k a vrcholy v_k a v_j, tedy $u_{k-1}(i,k) + u_{k-1}(k,j)$.

Pro ruční výpočet je jistě vhodné a přehlednější vypisovat každou matici U_0 až U_n zvlášť, ale pro programové zpracování by to bylo nanejvýš neoptimální. Asymptotická paměťová složitost algoritmu při uchovávání každé matice je $O(|V|^3)$, ale v případě, kdy budeme pracovat pouze s jednou maticí a hodnoty jejích vlastností budeme v průběhu algoritmu aktualizovat, zmenší se asymptotická paměťová složitost na $O(|V|^2)$. Pracovat pouze s jedinou maticí v počítači lze, protože při transformaci matice U_{k-1} na matici U_k se hodnoty prvků v k řádku a k sloupci nemění a aktualizace ostatních prvků probíhá nezávisle. Algoritmu tedy stačí předat matici A, která bude v průběhu jeho výpočtu přetransformována na matici U.

Ukázka Floydova algoritmu napsaného v pseudokódu s využitím jedné transformované matice, která je reprezentována polem a proměnná n je rovna počtu vrcholů ($n = |V|$):

```
funkce FloydAlg(A)
start
     pro k:= 1 až n:
          pro i:= 1 až n:
               pro j:= 1 až n:
                    pokud A[i][j] > A[i][k] + A[k][j]:
                         A[i][j]:= A[i][k] + A[k][j];
     vrať A;
konec
```

Bohužel Floydův algoritmus ve své ryzí formě poskytuje jako výsledek pouze matici vzdáleností, ze které není možné nalezené cesty zrekonstruovat. Pro přidání možnosti rekonstrukce nejkratších cest naštěstí stačí jen malá modifikace algoritmu. Algoritmus bude pracovat navíc s jednou maticí o rozměrech $|V| \times |V|$, kterou nazveme *matice následníku* a označíme jí Q. Prvky $q(i,j)$ matice následníků Q budou obsahovat informace o vrcholu v, který je prvním následníkem vrcholu v_i na nejkratší cestě mezi vrcholy v_i a v_j. Matice Q bude inicializována hodnotami $q(i,j) = v_j$ a aktualizovat se bude společně s maticí U podle předpisu $q(i,j) = q(i,k)$, tedy vrchol, který je prvním následníkem vrcholu v_i na cestě z vrcholu v_i do vrcholu v_k se stane prvním následníkem vrcholu v_i na cestě mezi vrcholy v_i a v_j.

Následující přiklad je ukázkou modifikovaného Floydova algoritmu v pseudokódu:

```
funkce FloydModAlg(A)
start
     pro i:= 1 až n:
          pro j:= 1 až n:
               Q[i][j]:= V(j);
     pro k:= 1 až n:
          pro i:= 1 až n:
               pro j:= 1 až n:
                    pokud A[i][j] > A[i][k] + A[k][j]:
                         A[i][j]:= A[i][k] + A[k][j];
                         Q[i][j]:= Q[i][k];
     vrať A, Q;
konec
```

3.2.2 Rekonstrukce nejkratší cesty u modifikovaného Floydova algoritmu

Z výstupu modifikovaného Floydova algoritmu můžeme jednoduše nejkratší cestu zrekonstruovat. Pokud cesta existuje, můžeme projít matici následníků Q podle následujícího algoritmu:

Krok 1 – inicializace. Nastavíme pomocnou proměnnou $v_i = v_p$ a posloupnost vrcholů na cestě $c = \{v_p\}$.

Krok 2 – rekonstrukce. Aktualizujeme proměnou $v_i = q(i,k)$ a její hodnotu zařadíme na konec posloupnosti vrcholů v cestě c. Pokud platí $v_i = v_k$, pak ukončíme rekonstrukci, jinak pokračujeme krokem 2.

Takto jsme získali posloupnost vrcholů c ležících na nejkratší cestě mezi vrcholy v_p a v_k.

3.2.3 Detekce cyklů se zápornou délkou

Jak již bylo popsáno výše, Floydův algoritmus může pracovat i s grafem, který má záporně ohodnocené hrany. Na takovém grafu se teoreticky může vyskytnout záporný cyklus a pokud je v grafu cyklus se zápornou délkou Floydův algoritmus nepracuje správně. Zda v grafu existuje záporný cyklus lze pomocí Floydova algoritmu zjistit jednoduše. Graf obsahuje záporný cyklus, pokud se na diagonále matice vzdáleností A objeví záporná hodnota $a(i,j) < 0$. Přítomnost záporného cyklu lze testovat již v průběhu algoritmu nebo až po jeho skončení.

3.3 BELLMANŮV-FORDŮV ALGORITMUS

Je algoritmus, který vyvinuli američtí matematici Richard E. Bellman a Lester R. Ford. Tento algoritmu dokáže nalézt nejkratší cesty z daného výchozího vrcholu v_p do všech ostatních dosažitelných vrcholů v v hranově ohodnoceném grafu G, jehož hranové ohodnocení může obsahovat i záporné hodnoty. Stejně jako Floydův algoritmus nedokáže najít nejkratší cesty v grafu s cyklem záporné délky, ale na konci svého běhu umí přítomnost záporného cyklu v grafu detekovat. Podobně jako Dijkstrův algoritmus používá metodu relaxace hran, ale na rozdíl od něj nevybírá vrchol s nejmenším nedefinitivním ohodnocením v a nerelaxuje jen hrany $e \in E^+(v)$ z něj vycházející, ale relaxuje všechny $e \in E$ hrany grafu a to přesně $|V| - 1$ krát. Všechny hrany relaxuje $|V| - 1$ krát z důvodu,

že maximální možný počet hran v cestě je roven právě tomuto číslu. Složitost Bellmanova-Fordova algoritmu je $O(|V| \cdot |E|)$. Distribuovanou variantu tohoto algoritmu používá směrovací protokol Routing Information Protocol (RIP). [Wiki04]

3.3.1 Popis Bellmanova-Fordova algoritmu

Jako vstup je algoritmu předán obecně hranově ohodnocený graf G a počáteční vrchol v_p, z něhož bude algoritmus hledat nejkratší cesty do všech ostatních dosažitelných vrcholů grafu. Výstupy algoritmu jsou stejné jako u Dijkstrova algoritmu, tedy pro každý vrchol v ohodnocení $U(v)$, které vyjadřuje délku nejkratší nalezené cesty mezi počátečním vrcholem v_p a vrcholem v. Dalším výstupem je pro každý vrchol v hodnota $ODKUD(v)$ uchovávající nejbližší předcházející vrchol (hranu) vrcholu v na nejkratší cestě z v_p.

Krok 1 – inicializace. Vrcholu v_p nastavíme hodnotu $U(v_p) = 0$ všem ostatním vrcholům v grafu G hodnota $U(v) = \infty$. Dále $ODKUD(v) = nic$ pro každý vrchol v a nastavíme čítač i na hodnotu $i = 0$.

Krok 2 – relaxace hran. Čítač i zvýšíme o jedna $i = i + 1$. Pokud je $i < |V| - 1$, tak s každou hranu $e \in E$ provede krok 2a, jinak přejdeme na krok 3.

Krok 2a. Položíme $v_{pe} = Pv(e)$ a $v_{ke} = Kv(e)$. Pokud platí $U(v_{ke}) > U(v_{pe}) + a(e)$, nastavíme délku nejkratší cesty do koncového vrcholu v_{ke} hrany e hodnotou $U(v_{ke}) = U(v_{pe}) + a(e)$ a předchůdce $ODKUD(v_{ke}) = v_{pe}$.

Krok 3 – detekce záporného cyklu. Pokud pro kteroukoliv hranu $e \in E$ platí $U(v_{ke}) > U(v_{pe}) + a(e)$, kde $v_{pe} = Pv(e)$ a $v_{ke} = Kv(e)$, přerušíme algoritmus, protože graf obsahuje záporný cyklus a výsledky nebudou správné.

Příklad Bellmanova-Fordova algoritmu v pseudokódu:

```
funkce BellmanFordAlg(G, vp)
start
     pro všechny vrcholy v uvnitř G:
          U(v):= nekonečno;
          ODKUD(v):= nic;
     pro i:= 1 až n-1:
          pro všechny hrany e uvnitř G:
               vep:= POČÁTEČNÍ_VRCHOL(E);
               vek:= KONCOVÝ_VRCHOL(E);
               pokud U(vek) > U(vek) + a(e):
                    U(vek):= U(vek) + a(e);
     pro všechny hrany e uvnitř G:
               vep:= POČÁTEČNÍ_VRCHOL(E);
               vek:= KONCOVÝ_VRCHOL(E);
               pokud U(vek) > U(vek) + a(e):
                    ukončit_chyba(„Detekován záporný cyklus!“);
     vrat U, ODKUD;
konec
```

Protože v mnoha grafech obsahují i nejdelší z nejkratších cest méně než $|V| - 1$ hran, lze malou modifikací výpočet algoritmu urychlit. Stačí, když si budeme pamatovat, zda byla během kroku 2 algoritmu některá kontrolovaná hrana nekorektní, tedy zda jsme měnili délku nejkratší cesty do koncového vrcholu této hrany. Pokud během některého průchodu krokem 2 budou všechny hrany korektní, je jisté, že ve všech dalších iteracích budou také korektní, proto můžeme algoritmus ukončit. [Černý, 2008]

Ukázka modifikovaného Bellmanova-Fordova algoritmu v pseudokódu:

```
funkce BellmanFordAlg(G, vp)
start
     pro všechny vrcholy v uvnitř G:
          U(v):= nekonečno;
          ODKUD(v):= nic;
     pro i:= 1 až n-1:
          doslo_k_ralxaci:= false;
          pro všechny hrany e uvnitř G:
               vep:= POČÁTEČNÍ_VRCHOL(E);
               vek:= KONCOVÝ_VRCHOL(E);
               pokud U(vek) > U(vek) + a(e):
                    U(vek):= U(vek) + a(e);
                    doslo_k_ralxaci:= true;
          pokud doslo_k_relaxaci = false:
               vrat U, ODKUD;       //pokračování na další straně
```

```
     pro všechny hrany e uvnitř G:
               vep:= POČÁTEČNÍ_VRCHOL(E);
               vek:= KONCOVÝ_VRCHOL(E);
               pokud U(vek) > U(vek) + a(e):
                    ukončit_chyba(„Detekován záporný cyklus!“);
     vrat U, ODKUD;
konec
```

3.3.2 Rekonstrukce nejkratší cesty u Bellmanova-Fordova algoritmu

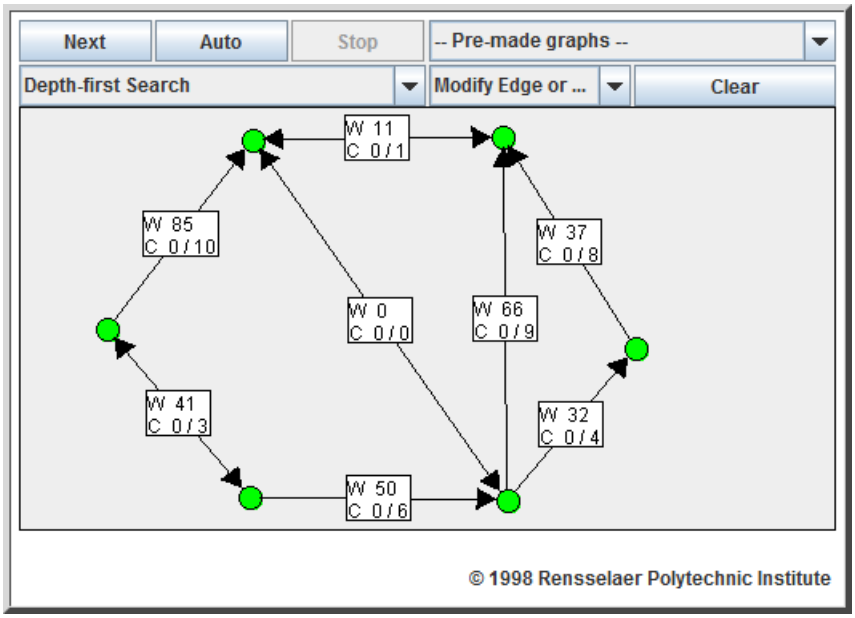
Rekonstrukce nejkratší cesty pro cesty nalezené Bellanovým-Fordovým algoritmem probíhá totožným způsobem jako rekonstrukce cest nalezených Dijkstrovým algoritmem, která je popsána v oddílu 3.1.2 na straně číslo 23.

4 ANALÝZA

V této kapitole práce budou nejprve popsány a zhodnoceny některé dostupné aplikace pro vizualizaci grafových algoritmů, dále bude uveden soupis požadavků na aplikaci a podrobnější rozbor každého z nich. Na konci kapitoly uvedu, který programovací prostředí a jazyk jsem pro implementaci zvolil.

4.1 DOSTUPNÉ APLIKACE

Aplikací pro vizualizaci průběhu grafových algoritmů je na Internetu k nalezení velké množství. Většinou jsou to aplikace vytvořené jako Java applety, které běží na webových stránkách. Bohužel velké množství těchto aplikací je jen jednostranně zaměřená. Mnoho z nich dokáže vizualizovat pouze jediný algoritmus nebo jeho průběh zobrazuje na předem definovaném grafu, který navíc není možné upravit. A pokud již je možné graf editovat, nelze jej uložit pro opětovné použití nebo využít standardních funkcí „Zpět“ a „Vpřed“ pro vrácení a opětovné vykonání změn při editaci grafu.



Obrázek 4: Applet - Graph theory applet

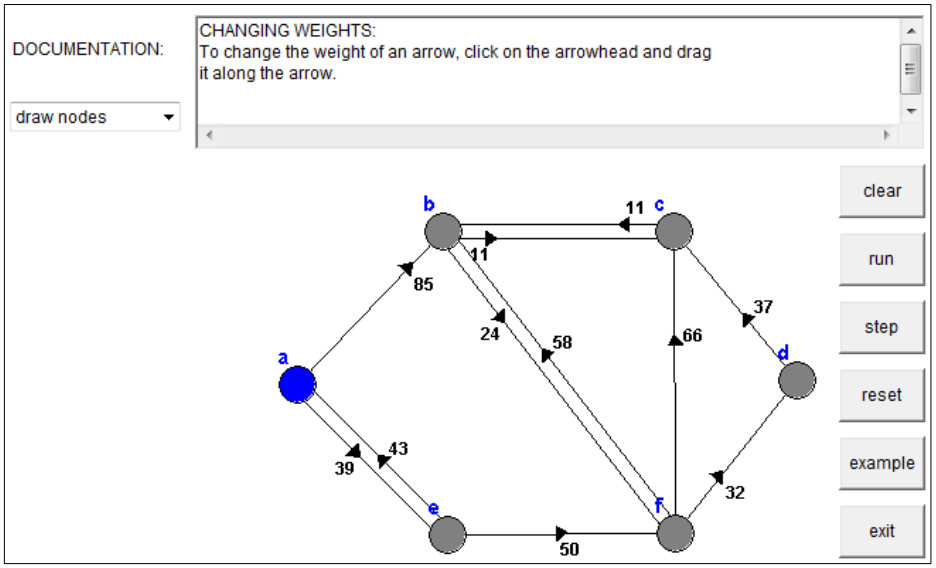
4.1.1 Applet – Graph theory applet

Jak je již z názvu patrné jedná se o Java applet, který je dostupný na webové adrese http://links.math.rpi.edu/applets/appindex/graphtheory.html. Screenshot z aplikace je zobrazen na obrázku č. 4. Tento applet umí zobrazit průběh šesti algoritmů teorie grafů

konkrétně Forův-Fulkersonův algoritmus pro hledání maximálního toku v síti, Dijkstrův, Floydův a Bellmanův-Fordův algoritmus pro hledání nejkratších cest v grafu, Kruskalův algoritmus pro nalezení minimální kostry grafu a algoritmus prohledávání grafu do hloubky. Applet umožňuje vytvoření hranově ohodnoceného grafu, do kterého lze přidávat orientované i neorientované hrany, ovšem nelze zadat multihrany. Průběh algoritmu je velmi stručně popisován a lze jej krokovat ručně nebo automaticky. Export ani import není podporován stejně tak funkce historie. Práce s appletem je velice přirozená, není třeba se s ním zdlouhavě seznamovat, je to dáno hlavně přepínačem funkcí editoru grafu, díky kterému vždy pracujeme pouze v jednom modu např. přidávání hran, odebírání vrcholů.

4.1.2 Applet – Dijkstra's Shortest Path Algorithm

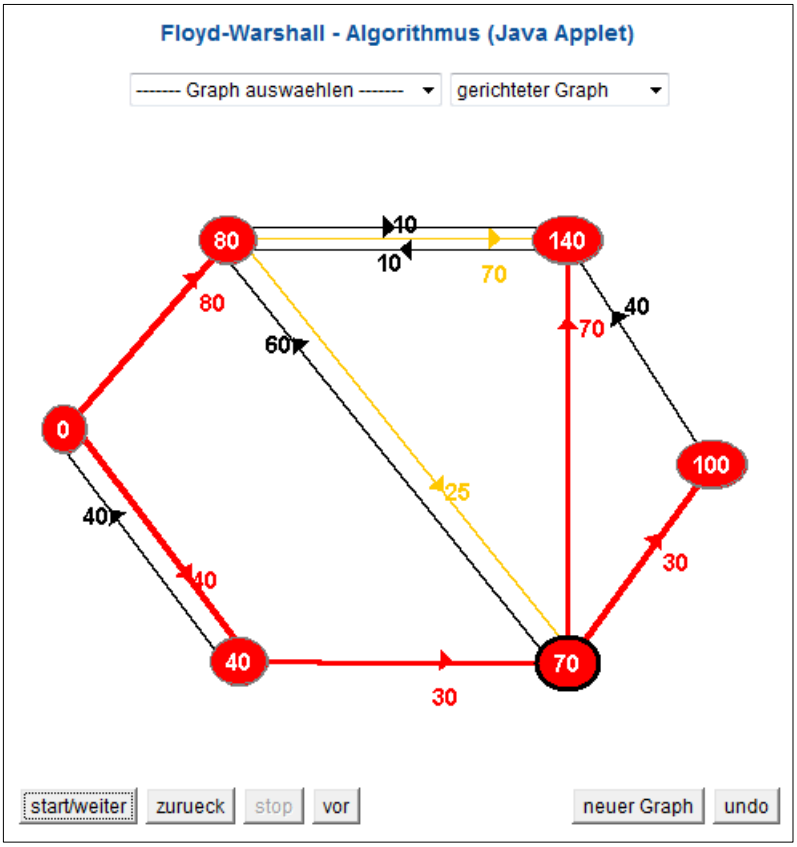
Tento Java applet pro vizualizaci Dijkstrova algoritmu dostupný na webové stránce http://www.dgp.toronto.edu/people/JamesStewart/270/9798s/Laffra/DijkstraApplet.html. Aplikace umožňuje vytvoření a editaci vlastního hranově ohodnoceného orientovaného grafu. Na vytvořeném grafu lze následně zobrazit průběh hledání nejkratší cesty, včetně poměrně rozsáhlého popisu jednotlivých kroků. Průběh algoritmu lze krokovat ručně i automaticky. Bohužel aplikace neumožňuje export ani import grafů ani nejsou zahrnuty funkce historie „Zpět" a „Vpřed". Editace a vytváření grafů v aplikaci je jednoduché, navíc aplikace obsahuje rychlou nápovědu, která ulehčí osvojení ovládání aplikace. Jak aplikace vypadá, je možné vidět na obrázku číslo 5.



Obrázek 5: Applet – Dijkstra's Shortest Path Algorithm

4.1.3 Applet – FloydWarshallApplet

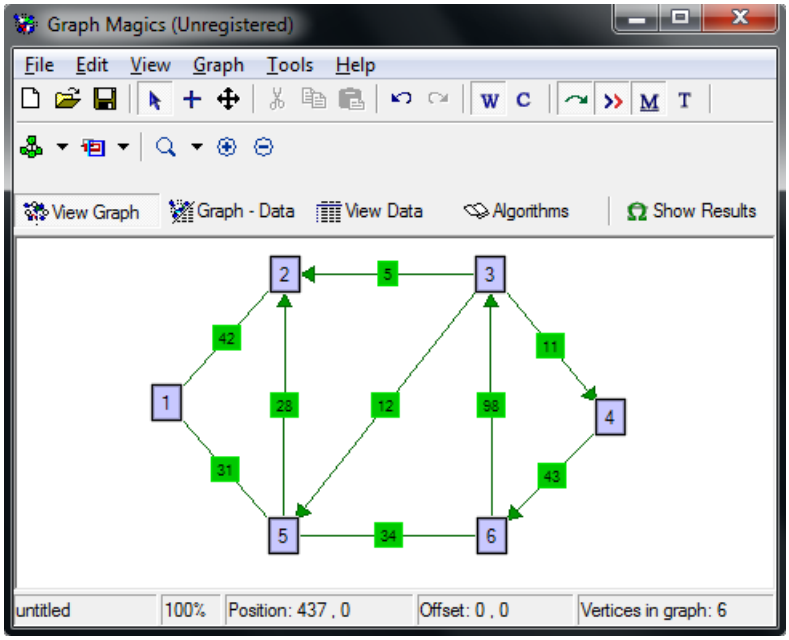
Opět se jedná Javovský applet, který je možné vyzkoušet na webové stránce http://www-m9.ma.tum.de/Allgemeines/FloydWarshallApplet. Aplikace je určena pro hledání nejkratší cesty Floydovým algoritmem v orientovaném nebo neorientovaném obecně ohodnoceném grafu. Applet nabízí několik předpřipravených grafů, na kterých je možné průběh algoritmu vyzkoušet, nebo je možné vytvořit vlastní graf. Krokovat algoritmus lze automaticky stejně tak ručně. Jako jediná aplikace z doposud popsaných implementuje funkci zpět pro vrácení změn provedených při editaci, ale chybí zde funkce vpřed a funkce pro import a export grafu pro opětovné použití. Dalším záporem této aplikace je absence výpisu nebo zobrazení transformace matice délek hran A na matici vzdáleností U, která je hlavním pracovním prvkem vizualizovaného algoritmu a navíc matice vzdáleností U jeho výstupem. Tvorba a úprava grafů v tomto appletu je téměř shodná jako v předchozím – práce s ním není nijak těžká ani složitá. Obrázek číslo 6 zobrazuje aplikaci po skončení algoritmu.



Obrázek 6: Applet – FloydWarshallApplet

4.1.4 Graph Magics 2.1

Graph Magics je aplikace určená pro operační systém Windows, jejíž 30-ti denní zkušební verze je ke stažení z domovské stránky www.graph-magics.com. V Graphs Magics je možné vytvářet a editovat orientované ohodnocené grafy bez multihran a smyček. Aplikace implementuje také několik základních grafových algoritmů, u kterých je možné vizualizovat jejich průběh, z pohledu zaměření této práce nás budou zajímat jen dva a to hledání nejkratší cesty mezi dvěma vrcholy Dijkstrovým algoritmem a hledání všech nejkratších cest z výchozího vrcholu Bellman-Fordovým algoritmem. Průběh algoritmu nelze krokovat ručně, nýbrž pouze automaticky. Program umožňuje ukládání vytvořených grafů a samozřejmě také jejich opětovné načtení, také je možné grafy exportovat do obrázků několika formátů. Dále nechybí implementace funkcí pro práci s historií a schránkou. Nejslabší stránkou aplikace je její ovládání, které je nepřirozené a složité. Aby bylo možné ovládání programu zvládnout, je nutné několikrát nahlédnout do nápovědy, která je naštěstí dobře zpracovaná. Na obrázku 7 je screenshot aplikace Graph Magics.



Obrázek 7: Graph Magics

4.1.5 Porovnání vlastností vybraných aplikací

Na předchozích stránkách bylo představeno několik aplikací pro vytváření grafů a vizualizaci průběhu grafových algoritmů na nich. Nyní bude uvedena přehledná tabulka porovnání vlastností popsaných aplikací.

Tabulka 1: Porovnání vlastností vybraných aplikací

	Graph theory applet	**Dijkstra's Shortest Path Algorithm**	**FloydWarshall-Applet**	**Graph Magics 2.1**
Ovládání	Nejjednodušší	Jednoduché	Jednoduché	Obtížné
Funkce historie	Chybí	Chybí	Pouze zpět	Ano
Export a import	Chybí	Chybí	Chybí	Ano
Export do obrázku	Chybí	Chybí	Chybí	Ano
Krokování algoritmů	Ručně i automaticky	Ručně i automaticky	Ručně i automaticky	Pouze automaticky
Multihrany	Ne	Ano	Ano	Ne
Popis průběhu algoritmu	Ano, stručný	Ano, rozsáhlý	Ne	Ne
Nápověda	Ne	Ano	Ano, FAQ	Ano, rozsáhlá

Analýzou popsaných i některých dalších aplikací jsem se snažil získat přehled o tom, jak aplikace pro vizualizaci grafových algoritmů obecně vypadají, jakým způsobem pracují, jaké možnosti uživateli nabízejí. Z této analýzy jsem vycházel při vytváření a kompletaci požadavků na aplikaci, která má být výsledkem této diplomové práce.

4.2 ANALÝZA POŽADAVKŮ

Ze zadání diplomové práce vyplývají hlavní požadavky na zpracování aplikace, těmito požadavky jsou:

- grafické zadávání a editace grafu,
- výpis vlastností grafu,
- zobrazování průběhu algoritmů po krocích,
- zobrazení případných dalších hodnot v tabulkách,
- uživatelská příručka.

Mimo těchto požadavků jsem v průběhu fází analýzy a implementace přidal k původním ještě několik dalších požadavků, které jsem začlenil z důvodu zjednodušení práce

s aplikací, rozšíření jejích možností a zvýšení komfortu práce s aplikací. Mezi tyto požadavky patří:

- export a import grafů,
- popis jednotlivých kroků probíhajícího algoritmu,
- implementace funkcí práce se schránkou,
- implementace funkcí historie,
- možnost práce s několika grafy najednou,
- přívětivé uživatelské prostředí,
- modulárnost - rozšiřitelnost aplikace,
- využívání principů OOP.

V dalších pododdílech této kapitoly budou popsány jednotlivé definované požadavky a ožnosti jakými lze tyto požadavky vyřešit.

4.2.1 Grafické zadávání a editace grafu

Pro uživatele je nejjednodušší graf reprezentovat nakreslením, jak bylo popsáno v oddílu 2.1.5. Proto i zadávaní vrcholů a hran grafu, by mělo být v editoru možné graficky s pomocí kurzoru myši. Kurzor myši by měl interagovat s kreslící plochou a prvky grafu na ploše nakreslenými. Uživatel by měl mít možnost i grafického přizpůsobení grafu jako např. úprava tloušťky hran nebo velikosti vrcholů, dále by byla užitečná možnost graf přiblížit nebo oddálit.

Tento rozvedený požadavek jasně definuje množinu použitelných programovacích jazyků. Tyto jazyky musí podporovat grafický výstup na obrazovku a vstup polohovacího zařízení myši.

4.2.2 Výpis vlastností grafu

Dalším požadavkem, který byl definován již v zadání diplomové práce, je schopnost aplikace přehledně poskytnout některé vlastnosti grafu jako jsou počet vrcholů, hran a komponent, skóre grafu, informace zda se jedná o strom nebo Eulerův graf.

Je to obecný požadavek, který nás nijak neomezuje ve výběru prostředků pro tvorbu aplikace.

4.2.3 Zobrazování průběhu algoritmů po krocích

Pod tímto požadavkem si můžeme představit zvýrazňování některých částí grafu, se kterými aktuálně algoritmus pracuje. Toto zvýraznění může být dosaženo například obarvením prvků grafu barvami různými od barev používaných pro prosté zobrazení grafu.

Z tohoto požadavku nám znovu vyplývá použití programovacího prostředí, které podporuje grafický výstup.

4.2.4 Zobrazení případných dalších hodnot v tabulkách

Některé algoritmy pracují s dalšími hodnotami mimo nakreslený graf, které by bylo vhodné nějakým způsobem zobrazovat. V případě Floydova algoritmu je jistě zajímavé sledovat, jakým způsobem probíhá transformace matice délek hran A na matici vzdáleností U. Data by se mohla zobrazovat v tabulkách s pojmenovanými sloupci a případně i řádky.

Tento požadavek je znovu zaměřen na grafický výstup aplikace a jeho podporu ve vývojovém prostředí.

4.2.5 Uživatelská příručka

Jak jsem zmínil v oddíle 4.1, některé aplikace nemají úplně jednoduché ovládání nebo některé funkce nemusejí být lehce pochopitelné. Protože se jedná o aplikaci odborně zaměřenou, je velmi vhodné její ovládání popsat v uživatelské příručce.

Požadavek na uživatelskou příručku je velmi obecný a neovlivňuje nám výběr jazyka ani vývojového prostředí. Jedná se o velmi důležitou součást finální aplikace, která má uživatele s aplikací seznámit a dostatečně podrobně pospat veškeré funkce a možnosti aplikace. Pokud uživatelská příručka nebude dostupná v elektronické podobě přímo z aplikace, bylo by dobré, aby aplikace obsahovala alespoň rychlou nápovědu.

4.2.6 Export a import grafů

U většiny komerčních programů se jedná o samozřejmou funkci aplikace. Ovšem jak bylo popsáno v předchozím oddíle, mnoho aplikací pro vizualizaci grafových algoritmů toto neumožňuje. Bylo by vhodné export a import implementovat podle zažitých zvyklostí,

tedy, že po výběru jedné z těchto funkcí aplikace vyvolá dialog pro otevření nebo uložení souboru grafu.

Jedná se o obecný požadavek, který nemusí výběr programovacího jazyka ovlivnit. Ale je pravdou, že výběr vhodného jazyka nám muže splnění tohoto požadavku ulehčit, protože některé jazyky implementují metody pro serializaci a deserializaci datových objektů a disponují třídami dialogů pro otevření a uložení souborů.

4.2.7 Popis jednotlivých kroků probíhajícího algoritmu

Průběh jednotlivých algoritmů je vhodné mimo grafické vizualizace také slovně popisovat. Popis aktuálního kroku totiž velice zpřehlední průběh algoritmu. Navíc pokud uživatel není s algoritmem plně seznámen nebo přesně nechápe některé jeho funkce, může slovní popis přispět k pochopení algoritmu.

Tento požadavek konkretizuje, jak má vypadat vizualizace a je tedy obecný a neovlivní výběr programovacího jazyka.

4.2.8 Implementace funkcí práce se schránkou

V současné době hojně používané funkce „Kopírovat“, „Vložit“ a „Vyjmout“, mohou velmi zrychlit tvorbu nebo editaci grafu. Jejich implementace by měla obsahovat spolupráci s kurzorem myši, kdy uživatel může pomocí myši přes prvky grafu nakreslit obdélník výběru a označit ty prvky, které obdélník ohraničuje. Takto vybrané vrcholy případně i hrany může uživatel kopírovat do schránky a poté libovolně vložit do grafu pod kurzor myši.

Pro tento požadavek je nutné programovací prostředí, jež umožňuje zachytávat vstup myši a klávesnice pro označení prvků grafu a také prostředí, které poskytuje grafický výstup pro kreslení obdélníku výběru. Pro přístup ke schránce je nutné buď použít funkce operačního systému nebo funkce poskytující programovací jazyk.

4.2.9 Implementace funkcí historie

Podobně jako funkce pro práci se schránkou, mohou funkce historie „Zpět“ a „Vpřed“ zjednodušit práci při tvorbě nebo editaci grafu. Například pokud uživatel omylem smaže část nebo posune část grafu.

Jedná se o obecný požadavek neovlivňující vývojové prostředí. Je jen nutné se zamyslet nad tím, jaké změny bude editor evidovat, jaké množství změn se bude v historii uchovávat a jakým způsobem.

4.2.10 Možnost práce s několika grafy najednou

Práce s několika okny najednou je v dnešní době velmi populární, vždyť každý aktuální webový prohlížeč umožňuje otevření více panelů, každý s jinou stránkou. To uživatel může využít pro porovnání výsledků algoritmů mezi jednotlivými grafy nebo v kombinaci s funkcemi pro práci se schránkou pro dočasné uložení části grafu na jiné místo.

Požadavek na práci s více grafy najednou ovlivňuje volbu programovacího prostředí, to totiž musí umožnit zobrazit několik podoken programu nebo nezávislé spuštění více instancí programu.

4.2.11 Přívětivé uživatelské prostředí

Důležitým prvkem aplikace je mimo její funkčnosti i její vzhled. Jedná se hlavně o ergonomii aplikace, tedy aby tlačítka a nástroje byly přehledně uspořádané, dostatečně rychle dostupné, výstižně popsané, v případě grafických tlačítek nebo kurzorů, aby jejich obrázky odpovídaly funkci a použití, aby je nebylo lehké zaměnit. Je dobré, když aplikace s uživatelem dostatečně často komunikuje, např. ve chvíli ukončení činnosti algoritmu je vhodné to uživateli oznámit, například zobrazením okna se zprávou nebo výpisem na obrazovku.

Požadavek na ergonomii aplikace nemusí ovlivnit výběr programovacího jazyka. Co ale rozhodně ovlivní je typ aplikace. Z hlediska ergonomie bude rozhodně mnohem vhodnější formulářová grafická aplikace, než konzolová textová. Mimo to, konzolová aplikace by byla v rozporu s některými ostatními požadavky.

4.2.12 Modulárnost - rozšiřitelnost aplikace

Modulárnost je požadavek, na nějž je v dnešní době kladen velký důraz. Pokud je aplikace vyvinuta v souladu s tímto požadavkem, lze ji v budoucnu snadněji obohatit o nové funkčnosti. Komponenty aplikace mezi sebou mohou sdílet části kódu, tím se kód zjednodušuje a snižuje se i možnost zanesení chyb a jejich duplicita. Vytvořené komponenty aplikace mohou být znovu použitelné.

Z tohoto požadavku plyne nutnost využití programovacího jazyka, který podporuje objektově orientované programování a vylučuje to všechny jazyky pro strukturované programování. Mimoto požadavek klade důraz na vhodné navržení struktury programu a používaných tříd.

4.2.13 Využívání principů OOP

Tento požadavek souvisí s předchozím požadavkem na modulárnost a rozšiřitelnost aplikace. Objektově orientované programovaní (OOP) je moderní metoda programování, která využívá takzvaných *objektů*, datových struktur tvořených z datových složek – *atributů* a složek reprezentujících chování objektů – *metod*. Objekty jsou instancemi takzvaných tříd, které popisují jejich obecné chování. OOP zpřehledňuje strukturu programu, umožňuje opakovaného použití kódu, využití zapouzdření, abstrakce, dědičnosti a polymorfizmu. [Wiki05]

Požadavek na využívání principů OOP redukuje množinu použitelných programovacích jazyků jen na ty, které objektově orientované programování umožňují.

4.3 PROGRAMOVACÍ JAZYK A VÝVOJOVÉ PROSTŘEDÍ

Po zanalyzování všech požadavků, jsem došel k závěru, že programovací jazyk, který bude použit při implementaci, musí být objektově orientovaný, měl by obsahovat knihovnu nebo podporu pro práci s grafikou, dále musí nabízet podporu práce se soubory a vstupem z myši a klávesnice. Tyto nároky splňuje hned několik programovacích jazyků jako například Java, C++ nebo C#.

Z výše jmenovaných jazyků jsem vybral jazyk C#. Je to objektově orientovaný jazyk, který díky provázanosti s Microsoft .NET Frameworkem poskytuje velké množství knihoven. Jednou z knihoven je knihovna GDI+, která umožňuje kreslení na komponenty aplikace. Další věcí, jež ulehčí práci je velké množství vytvořených dialogů, které mohou být do aplikace zakomponovány, jako například dialogy otevření a uložení souboru, výběru barvy nebo dialog pro tisk. Velkou výhodou je podpora serializace a deserializace objektů. Pro programování aplikace v jazyce C# jsem se mimo jiné rozhodl také proto, že s ním mám nejvíce zkušeností. [Wiki06]

Jako vývojové prostředí jsem zvolil Microsoft Visual Studio 2008, které je díky programu MSDN Academic Alliance pro studenty fakult a vysokých škol do něj zapojených, zdarma k dispozici. Od volby vývojového prostředí se odvíjí verze jazyka C# použitá pro programování aplikací. V prostředí Visual Studio 2008 je možné programovat ve verzích C# 2.0, která se pojí s .NET Frameworkem verze 2.0 a C# 3.0, jež využívá .NET Framework verze 3.5. Z důvodu využití některých funkcionalit, které jsou obsaženy pouze v .NET 3.5 je aplikace naprogramována ve verzi 3.0 jazyka C#. [Wiki07]

Aplikaci jsem se rozhodl vytvořit jako standardní formulářovou aplikaci určenou pro operační systémy Microsoft Windows s nainstalovaným balíkem .NET Framework 3.5. Přestože je aplikace primárně určena pro použití na operačních systémech Windows mělo by být možné ji zkompilovat a provozovat i na unixových OS s instalovaným projektem MONO.

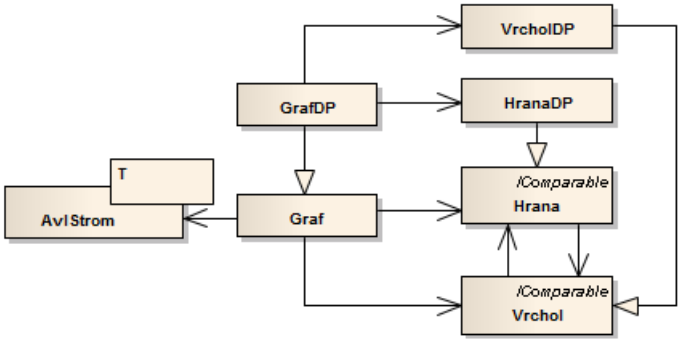
5 IMPLEMENTACE

V první části této kapitoly popíši vybrané datové struktury použité při implementaci aplikace, dále se zaměřím na některé třídy a popíši jejich funkci.

Při programování aplikace jsem kladl důraz na využívání možností, které nabízí OOP a často využíval například dědičnost a polymorfismus. Díky tomu jsem dosáhl vysokého stupně znovu použitelnosti některých vytvořených komponent a maximalizoval možnosti pro rozšíření aplikace o další funkčnosti.

5.1 DATOVÁ STRUKTURA PRO REPREZENTACI GRAFU

Datová struktura graf je jedním z hlavních prvků aplikace. Tato struktura uchovává informace o hranách a vrcholech grafu a umožňuje ostatním částem aplikace přístup k nim. Na obrázku 8 je UML diagram tříd, které datová struktura graf využívá.



Obrázek 8: UML diagram tříd datové struktury graf

Aplikace využívá implementace abstraktní datové struktury graf ve třídě *Graf*, která umožňuje přímý přístup k vrcholům i hranám grafu a využívá křížové reprezentace. Třída *Graf* je implementována s využitím datové struktury AVL strom, pro rychlé vyhledání a přístup k prvkům grafu. Každý vrchol reprezentovaný třídou *Vrchol* obsahuje pole s odkazy na incidentní hrany reprezentované třídou *Hrana*. Stejně tak hrany obsahují odkazy na incidující vrcholy. Díky této křížové reprezentaci je možné jednoduše a rychle přistupovat k následníkům i předchůdcům vrcholů a hran. Aplikace sama pracuje s třídami *GrafDP*, *VrcholDP* a *HranaDP*, které jsou potomky tříd *Graf*, *Vrchol* a *Hrana* a jsou konkretizovány pro její potřeby.

5.1.1 Třída Vrchol

Třída *Vrchol* reprezentuje obecný vrchol s informacemi o jeho incidenci. Datové složky třídy *Vrchol* jsou:

- **Hrana[] vstupujiciHrany, vystupujícíHrany** – pole incidujících hran vstupujících (končících) do vrcholu, vystupujících (začínajících) ve vrcholu

Vlastnosti třídy:

- **Hrana[] IncidentniHrany** – vrací pole vytvořené sloučením polí *vstupujiciHrany* a *vystupujiciHrany*
- **Hrana[] VstupujiciHrany, VystupujiciHrany** – vrací pole *vstupujiciHrany*, *vystupujiciHrany*

Třída Vrchol osahuje následující metody:

- **int CompareTo(object obj)** – porovnává instanci třídy *Vrchol* s jinou instancí třídy, vrací celočíselnou hodnotu rovnou nule, pokud se jedná o stejné vrcholy
- **Hrana OdeberHranu(Hrana hrana)**, **OdeberVstupujiciHranu(Hrana hrana), OdeberVystupujiciHranu(Hrana hrana)** – metody odeberou hranu z polí hran a vrátí ji navrátovou hodnotou
- **void PridejHranu(Hrana hrana), PridejVstupujiciHranu(Hrana hrana), PridejVystupujiciHranu(Hrana hrana)** – tyto metody přidají do jednoho z polí *vstupujiciHrany* a *vystupujiciHrany* hranu předanou parametrem metod

5.1.2 Třída Hrana

Tato třída představuje obecnou hranu grafu a uchovává informace o incidenci. Třída *Hrana* obsahuje tyto datové složky:

- **Vrchol pocatecniVrchol, koncovyVrchol** – instance třídy *Vrchol*, ze kterého hrana vychází, do kterého vstupuje

Vlastnosti třídy *Hrana*:

- **Vrchol PocatecniVrchol, KoncovyVrchol** – vrací nebo nastavuje počáteční, koncový vrchol hrany

Metody třídy:

- **int CompareTo(object obj)** – slouží k porovnání dvou instancí třídy
- **Vrchol[] DejIncidentniVrcholy** – metoda návratovou hodnotou vrací pole incidentních vrcholů

5.1.3 Třída Graf

Třída *Graf* je abstraktní datovou strukturou představující graf a umožňující přistup k jeho prvkům. Datové složky třídy jsou:

- **AvlStrom<Hrana> hrany** – tabulka vytvořená na AVL stromu, která obsahuje všechny hrany grafu a díky které je možné hrany procházet
- **bool orientovanyGraf** – logická hodnota značící zda se jedná o orientovaný nebo neorientovaný graf
- **AvlStrom<Vrchol> vrcholy** – tabulka obsahující vrcholy grafu

Třída má následující vlastnosti:

- **bool OrientovanyGraf** – vrací nebo nastavuje logickou hodnotu určující, zda se jedná o orientovaný graf

Metody třídy *Graf*:

- **Hrana OdeberHranu(Hrana hrana)** – odebere hranu z grafu a zajistí i její odebrání z incidujících vrcholů, odebranou hranu vrací návratovou hodnotou
- **Vrchol OdeberVrchol(Vrchol vrchol)** – odebere vrchol z grafu včetně odebrání hran s ním incidujících a vrátí jej návratovou hodnotou

- **void PridejHranu(Hrana hrana),**
  PridejHranu(Vrchol pocatecniVrchol, Vrchol koncovyVrchol) – přidá do grafu již vytvořenou hranu předanou parametrem metody, nebo vytvoří hranu mezi zadanými vrcholy a přidá ji do grafu
- **void PridejVrchol(Vrchol)** – vloží do grafu vrchol z parametru metody

5.1.4 Třída VrcholDP

VrcholDP je potomkem třídy *Vrchol* a rozšiřuje její zodpovědnosti, proto také obsahuje stejné datové složky, vlastnosti a metody jako rodičovská třída a ty zde nebudu znovu popisovat. Oproti rodičovské třídě má tyto datové složky:

- **string nazev** – uchovává název vrcholu
- **double ohodnocení** – hodnota ohodnocení vrcholu, pokud je ohodnocen
- **Bod poloha** – instance třídy *Bod*, která obsahuje informace o poloze vrcholu na grafu
- **TimeDate vytvořen** – přesný čas vytvoření vrcholu, použitý pro porovnávání vrcholů v metodě *CompareTo*

Třída má následující přidané vlastnosti:

- **string Nazev** – vrací nebo nastavuje název vrcholu
- **double Ohodnocení** – vrací nebo nastavuje ohodnocení vrcholu
- **Bod Poloha** – vrací nebo nastavuje instanci třídy Bod určující polohu vrcholu
- **int Stupeň** – vrací stupeň vrcholu

Metody třídy:

- **string ToString()** – vrací řetězec s názvem vrcholu

5.1.5 Třída HranaDP

Tato třída je potomek třídy Hrana, která ji konkretizuje pro potřeby aplikace. Datové složky, vlastnosti a metody zděděné z třídy původní nebudou popisovány. Následují datové složky třídy:

- **string nazev** – název hrany

- **double ohodnoceni** – váha, délka nebo ohodnocení hrany
- **DateTime vytvořena** – přesný čas vytvoření hrany, pro potřeby porovnávání metodou *CompareTo*

Vlastnosti třídy *HranaDP*:

- **bool JeSmycka** – vrací logickou hodnotu indikující, zda hrana je smyčkou
- **string Nazev** – vrací nebo nastavuje název hrany
- **double Ohodnocení** – nastavuje nebo vrací ohodnocení hrany

Metda třídy:

- **void ZmenOrientaciHrany()** – změní orientaci hrany

5.1.6 Třída GrafDP

Třída *GrafDP* dědí většinu svých zodpovědností ze třídy *Graf*. Je to datová struktura grafu, s níž pracují ostatní komponenty grafu jako například třídy algoritmů nebo třídy vizualizační. Dále budou popsány složky třídy, které má navíc proti rodičovské třídě. Datové složky třídy *GrafDP* jsou:

- **string nazev** – řetězec obsahující pojmenování grafu

Vlastnosti třídy:

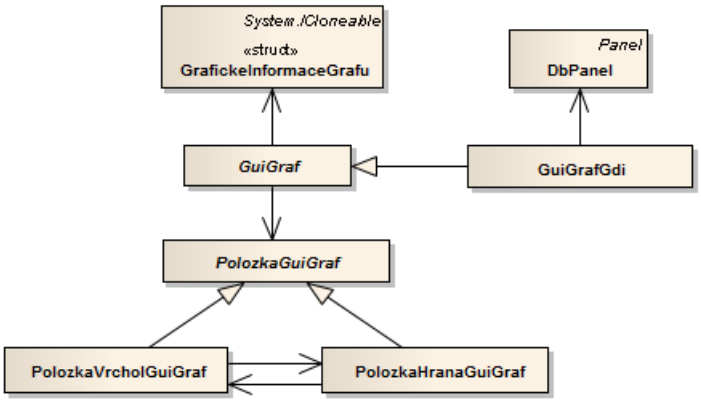
- **IEnumerable Hrany** – vrací iterator pro procházení všech hran grafu
- **string Nazev** – nastavuje a vrací řetězec názvu grafu
- **int PocetHran** – vrací počet hran, které graf obsahuje
- **int PocetVrcholů** – vrací počet vrcholů grafu
- **IEnumerable Vrcholy** – vrací iterator, jež umožní procházení tabulky vrcholů grafu

Metody třídy GrafDP:

- **HranaDP[] DejHranyMeziVrcholy(VrcholDP vrchol1, VrcholDP vrchol2)** – metoda vrátí pole všech hran, které spojují vrcholy zadané parametry metody

5.2 TŘÍDY PRO VYKRESLOVÁNÍ GRAFU

Dalším důležitým prvek aplikace je „modul“ vykreslování grafu. Tento modul má na starost mimo vykreslování grafiky grafu i ošetření vstupu uživatele provedeného nad vykresleným grafem, zvýrazňování určených částí grafu. Právě vykreslený graf je hlavním ovládaným elementem z pohledu uživatele, a proto je velmi důležité, aby jeho vykreslování při zadávání, posunu jeho částí nebo zvýrazňování jeho prvků bylo dostatečně rychlé a neobtěžovalo uživatele například blikáním nebo pomalou odezvou. Hlavní třídou, která má na starost vykreslování je abstraktní třída *GuiGraf*, resp. její potomek třída *GuiGrafGdi*. Tyto třídy spolupracují s dalšími pomocnými třídami, které je možné vidět v UML diagramu na obrázku číslo 9. Jednotlivé třídy a struktura budou důkladněji popsány na následujících stránkách.



Obrázek 9: UML diagram tříd pro vykreslování grafu

5.2.1 Struktura GrafickeInformaceGrafu

Tato struktura je navržena aby uchovávala informace o barvách, tloušťkách čar, velikosti a řezu písma a velikosti kružnic vrcholů. Datovými složkami jsou:

- **System.Drawing.Pen AlgoritmusHranaPen, AlgoritmusVrcholObrysPen, HranaPen, VrcholObrysPen, ZvyraznenaHranaPen, ZvyraznenyVrcholObrysPen** – pera grafické knihovny GDI+ pro různé druhy vykreslování hran a obrysu vrcholů
- **System.Drawing.Brush AlgoritmusVrcholVyplnBrush, PopiskyBrush, VrcholVyplnBrush, ZvyraznenyVrcholBrush** – štětce grafické knihovny GDI+ pro vyplňování ploch kruhů vrcholů a kreslení textu popisků

- **System.Drawing.Font PopiskyFont** – písmo používané pro kreslení popisků prvků grafu
- **float PrumerObrysKruhu** – průměr kružnice pro kreslení vrcholů

Metody struktury:

- **object Clone()** – vytvoří kopii struktury a všech jejích datových složek

5.2.2 Abstraktní třída PolozkaGuiGraf

Je abstraktní třída zapouzdřující základní chování vykreslovaných položek grafu, tedy hran a vrcholů a uchovává informace o jejich vizualizaci. Konkrétní chování je implementováno v potomcích třídy *PolozkaVrcholGuiGraf* a *PolozkaHranaGuiGraf*. Má tyto datové složky:

- **HranicniObalka hranicniObalka** – reprezentuje obdélníkovou obálku prvku grafu, která se používá pro urychlení určování polohy kurzoru vůči prvku
- **System.Drawing.Bitmap popisekBitmap** – bitmapa obsahující text popisku položky, pro rychlejší vykreslování na plátno grafu
- **float priblizeni** – hodnota aktuálního přiblížení plátna používaná při změně zoomu plátna pro přepočet datových složek s informacemi o bodech nutných pro kreslení položek

Třída má tyto vlastnosti:

- **HranicniObalka HranicniObalka** – vrací hraniční obálku položky grafu
- **abstract object Polozka** – abstraktní vlastnost, vrací objekt třídy VrcholDP, nebo HranaDP
- **System.Drawing.Bitmap PopisekBitmap** – vrací bitmap s popiskem položky

Metody třídy:

- **abstract bool LeziBodVeVyberuPolozky(Bod bod)** – abstraktní metoda, která vrací logickou hodnotu *true* pokud bod předaný parametrem leží v prostoru výběru položky, jinak vrací *false*
- **abstract void PosunPolozku(Bod posun)** – abstraktní metoda, posune položku o hodnotu zadanou vektorem parametru

- **abstract void PrepoctiHranicniObalku(float priblizeni, GrafickeInformaceGrafu gif, float magnetismus)** – abstraktní metoda, přepočítá body hraniční obálky položky pro změněné hodnoty parametrů
- **void VytvorPopisek(GrafickeInformaceGrafu)** – vykreslí do bitmapy *popisekBitmap* popisek položky

5.2.3 Třída PolozkaVrcholGuiGraf

Tato třída je potomkem třídy *PolozkaGuiGraf*. Každá instance uchovává vizuální vlastnosti jednoho konkrétního vrcholu grafu. Obsahuje všechny metody rodičovské třídy. Abstraktní metody a vlastnosti jsou přeprogramovány, aby správně fungovaly pro vizualizaci vrcholů grafu. Třída *PolozkaVrcholGuiGraf* rozšiřuje rodičovskou třídu o tyto datové složky:

- **PolozkaHranaGuiGraf[] incHrany** – pole instancí třídy *PolozkaHranaGuiGraf*, které uchovávají informace o vizualizaci hran incidujících s vrcholem, jehož vizualizační data uchovává instance této třídy
- **float magnetismusOdStredu** – vzdálenost od středu vykresleného vrcholu, do které kurzor myši zvýrazní vrchol a umožní jeho editaci nebo manipulaci s ním
- **Bod polohaPriblizeni** – poloha středu vykreslovaného vrcholu přepočtená na souřadnice plátna podle aktuálního měřítka (přiblížení plátna)
- **VrcholDP vrchol** – instance třídy *VrcholDP*, ke které patří vizualizační data uchovaná v objektu této třídy

Vlastnosti třídy:

- **PolozkaHranaGuiGraf[] IncidentniHrany** – vrátí pole s vizualizačními daty hran incidujících s vrcholem
- **Bod Poloha** – vrací nebo nastavuje polohu středu vrcholu na plátně

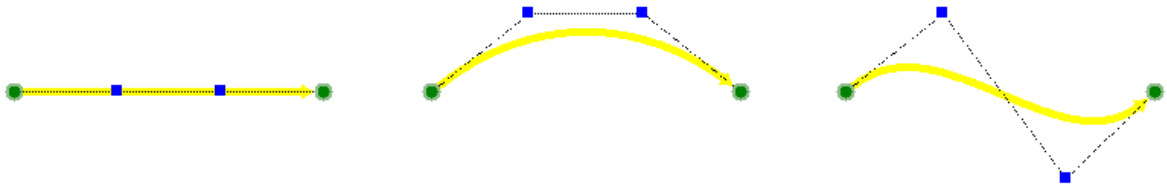
Třída má tyto metody:

- **void OdeberHrany(PolozkaHranaGuiGraf[] hrany)** – odstraní z pole *incHrany* odkazy na vizualizace hran předaných parametrem metody

- **void PridejHranu(PolozkaHranaGuiGraf hrana)** – přidá do pole *incHrany* odkaz na instanci třídy vizualizace hrany *PolozkaHranaGuiGraf*

5.2.4 Třída PolozkaHranaGuiGraf

Tato třída je také potomkem PolozkaGuiGraf, ze které přebírá definované funkčnosti a ozšiřuje je o další nutné k vizualizaci hran grafu. Hrany grafu je možné kreslit jako úsečky nebo Bézierovy křivky. Zakřivení hran lze měnit pomocí dvou editačních bodů, jak je možné vidět na obrázku číslo 10. Pokud je hrana přidávána mezi vrcholy, které mezi sebou již hranu obsahují, bude hrana vizualizovaná jako křivka, jinak jako úsečka. Úsečku lze editací kontrolních bodů změnit na křivku.



Obrázek 10: Editace zakřivení hrany

Datové složky třídy *PolozkaHranaGuiGraf* jsou:

- **Bod[] bodyKrivka** – pole kontrolních bodů pro kreslení křivky
- **HranaDP hrana** – instance hrany grafu, jež je vizualizovaná objektem této třídy
- **PolozkaVrcholGuiGraf[] incVrcholy** – pole odkazů na instance položek vizualizace vrcholů incidentích s hranu, jež je vizualizovaná instancí této třídy
- **bool krivka** – logická hodnota nabývá *true* pokud je hrana vizualizovaná jako křivka, *false* pokud jako úsečka
- **float magnetismusSCarou** – maximální vzdálenost od křivky, do které kurzor myši zvýrazní hranu a je umožněna interakce s ní
- **System.Drawing.Bitmap oblastVyberu** – obsahuje bitmapu s vykreslenou křivkou hrany pro potřeby zjištění vzdálenosti bodu od křivky
- **float vzdalenostKonceSipekOdStreduVrcholu** – vzdálenost mezi středem vykresleného vrcholu a jeho obrysem, používá se k výpočtu bodů, ve kterých začíná a končí vykreslování hrany

Vlastnosti třídy:

- **Bod[] BodyKrivka** – vrací pole kontrolních bodů pro kreslení křivky
- **PolozkaVrcholGuiGraf[] IncidentniVrcholy** – vrací pole odkazů na instance tříd vizualizace vrcholů incidujích s hranou
- **bool Krivka** – vrací logickou hodnotu *true*, pokud je hrana vizualizovaná jako křivka, *false* pokud není

Metody třídy:

- **void nastavHranicniObalkuAVykresliOblastVyberu()** – přepočítá hraniční obálku hrany a vykreslí křivku do bitmapy *oblastVyberu*
- **void NastavKontrolniBody(Bod bod2, Bod bod3)** – nastaví druhý a třetí kontrolní bod Bézierovy křivky, kterou je hrana vykreslována
- **int pocetHranMeziVrcholy()** – vrátí počet hran mezi vrcholy incidentními hrany
- **void PrepoctiPosunVrcholu()** – přepočítá počáteční a koncový bod pro kreslení křivky nebo úsečky, metoda je využívána při posunu inicdujícího vrcholu
- **void ZmenOrientaci()** – změní orientaci vizualizované hrany

5.2.5 Abstraktní třída GuiGraf

Abstraktní třída *GuiGraf*, v sobě zapouzdřuje chování společné pro své potomky, kteří by měli rozšířit její funkčnost o vykreslování grafu. Třídu *GuiGraf* jsem navrhl jako abstraktní pro zachování možnosti přidat do aplikace dalšího potomka, který by mohl implementovat kreslení například pomocí některé grafické knihovny podporující hardwarovou akceleraci, jakými jsou OpenGL nebo Microsoft DirectX. Implementace potomka vykreslujícího graf pomocí těchto knihoven by mohla zvýšit rychlost aplikace na počítačích se slabším procesorem, protože potomek *GuiGrafGdi* využívá pro kreslení knihovnu GDI+, která není kompletně akcelerovaná a některé grafické výpočty provádí procesor. [Wiki08]

Třída implementuje funkčnosti nezávisející na použité grafické knihovně, například uložení vybraných položek do schránky systému, načtení položek ze schránky do grafu a definuje abstraktní funkce, které jsou implementovány až v konkrétních potomcích.

Abstraktní třída *GuiGraf* zapouzdřuje tyto datové složky:

- **byte bodEditaceHrany** – označuje, který kontrolní bod křivky hrany je právě editován
- **Bod[] bodyPosun** – pole bodů, které je užíváno pro výpočet maximálního možného posunu prvků grafu vzhledem k rozměrům plátna při jejich posunu nebo vkládání do grafu
- **Timer casovac** – časovač pro zobrazení vlastností prvků grafu
- **GrafickeInformaceGrafu gif** – struktura s informacemi o grafické reprezentaci graf
- **GrafDP graf** – instance grafu, který je třídou potomka zobrazován a manipulován
- **Cursor gumaCursor, hranyCursor, vrcholyCursor** – kurzory myši používané pro různé nástroje aplikace
- **float magnetismus** – hodnota magnetismu prvků grafu (vzdálenost kurzoru od prvku grafu, ve které spolu interagují)
- **PolozkaGuiGraf[] oznacenePolozky, oznacenePolozkyAlgoritmus** – pole s instancemi tříd vizualizace položek, které jsou označeny nebo označeny algoritmem a mají být při vykreslování zvýrazněny
- **PolozkaGuiGraf polozkaPodMysi** – položka grafu, která leží v oblasti interakce s kurzorem myši a bude při vykreslení zvýrazněna
- **PolozkyGuiGraf[] polozky** – pole vizualizací položek grafu
- **bool popisky, zobrazitOhodnoceniHran, zobrazitOhodnoceniVrcholu** – logické hodnoty ovlivňující, zda se při vykreslování grafu budou zobrazovat také popisky a ohodnocení hran a vrcholů
- **PracovniNastroj pracovniNastroj** – výčtový typ, který určuje jaký pracovní nástroj aplikace je právě aktivní
- **float priblizeni** - hodnota aktuálně nastaveného přiblížení kreslícího plátna
- **bool probihaAlgoritmus** – logická hodnota značící, zda nad grafem probíhá některý algoritmus

- **Dictionary<object, PolozkaGuiGraf> prvkyGrafuPolozkyGuiGraf** – slovník uchovávající dvojice klíč a hodnota, kde klíč je instance třídy VrcholDP nebo HranaDP a hodnotou je odpovídající instance třídy PolozkaVrcholGuiGraf nebo PolozkaHranaGuiGraf, pro potřeby rychlé identifikace odpovídajících instancí
- **Form rodicForm** – rodičovský formulář, na kterém se zobrazuje graf
- **Bod rozmeryPlatna, rozmeryPlatnaPriblizeni** – rozměry plátna a rozměry plátna po aplikaci hodnoty přiblížení

Vlastnosti abstraktní třídy *GuiGraf*:

- **GrafDP Graf** – vrací instanci grafu, se kterým třída pracuje
- **GrafickeInformaceGrafu GrafickeInformaceGrafu** – vrací a nastavuje grafické informace vykreslování grafu
- **Cursor GumaCursor, HranyCursor, VrcholyCursor** – nastavuje kurzory myši pro použití s různými nástroji aplikace
- **float Magnetismus** – nastavuje a vrací hodnotu magnetismu
- **abstract System.Drawing.Bitmap ObrazekGrafu** – abstraktní vlastnost, která vrací bitmapu s vykresleným grafem
- **bool Popisky, ZobrazitOhodnoceniHran, ZobrazitOhodnoceniVrcholu** – vrací a nastavuje logickou hodnotu zobrazování popisků a ohodnocení hran a vrcholů
- **abstract PracovniNastroj PracovniNastroj** – abstraktní vlastnost, nastavuje pracovní nástroj aplikace
- **abstract float Priblizeni** – nastavuje a vrací hodnotu přiblížení plátna
- **bool ProbihaAlgoritmus** – nastavuje a vrací logickou hodnotu, zda probíhá algoritmus nad grafem
- **abstract Bod RozmeryPlochy** – vrací a nastavuje rozměr kreslícího plátna, abstraktní vlastnost

Třída obsahuje tyto metody:

- **Bod dejMoznyPosun(Bod posun)** – metoda ověří, zda je možné posunout označené prvky grafu ve směru a délce vektoru daného parametrem metody,

pokud ano vrátí vektor rovný parametru, pokud ne vrátí vektor stejného směru jako parametr, ale v délce maximálně možné

- **PolozkaGuiGraf dejPolozkuUBodu(Bod bod)** – vrátí položku, v jejíž oblasti výběru leží bod předaný parametrem, upřednostní položku vrcholu před hranou
- **PolozkaGuiGraf[] DejPolozky()** – vrátí všechny položky vizualizace prvků grafu
- **PolozkaGuiGraf[] KopirujOznacenePolozky()** – metoda vrací kopie všech položek vizualizace prvků grafu, které obsahuje pole *oznacenePolozky*
- **PolozkaGuiGraf[] kopirujPolozky(PolozkaGuiGraf[] kopirovanePolozky)** – vrací kopie položek předaných parametrem
- **void nastavBodyPosun(PolozkaGuiGraf[] polozky)** – naplní pole *bodyPosun* body z položek předaných parametrem, které je nutné kontrolovat při posunu položek
- **void NastavPolozky(PolozkaGuiGraf[] novePolozky)** – odstraní z pole *polozky* všechny položky vizualizace a nahradí je položkami předanými parametrem metody
- **void NastavPolozkyAlgoritmus(VrcholDP[] zvyrazneneVrcholy, VrcholDP[] zvyrazneneVrcholyAlgoritmus, HranaDP[] zvyrazneneHrany, HranaDP[] zvyrazneneHranyAlgoritmus)** – naplní pole *oznacenePolozky* a *oznacenePolozkyAlgoritmus* polozkami vizualizace prvků grafu předaných parametry metody
- **abstract void NastavStav(GrafDP graf, PolozkaGuiGraf[] polozky, double sirka, double vyska, GrafickeInformaceGrafu gif)** – nastaví instanci do stavu daného parametry abstraktní metody
- **void OdeberOznacenePolozky()** – metoda odstraní z pole *polozky* položky vizualizace, které jsou v poli *oznacenePolozky* odstraní prvky odpovídající těmto položkám z grafu a vymaže pole *oznacenePolozky*
- **void OdeberPolozku(PolozkaGuiGraf polozka)** – odstraní položku předanou parametrem z pole polozky a odpovídající hranu nebo vrchol z grafu
- **void onGrafAlgoritmusMouseDown(GrafAlgoritmusMouseDownEventArgs e)** – vyvolá událost *GrafAlgoritmusMouseDown*

- **void onGrafMouseMove(GrafMouseMoveEventArgs e)** – vyvolá událost GrafMouseMove
- **void onZmenaGrafu(EventArgs e)** – vyvolá událost ZmenaGrafu
- **abstract void OznacVsechnyPolozky()** – abstraktní metoda označí všechny položky grafu (zkopíruje je do pole *oznacenePolozky*)
- **abstract void prekresliGraf()** – abstraktní metoda, která překreslí graf
- **void prepocitejHranicniObalky()** – metoda zajistí přepočet hraničních obálek všech položek vizualizace
- **void pridejPolozku(PolozkaGuiGraf polozka)** – přidá do pole *polozky* položku předanou parametrem a do grafu jí odpovídající hranu, nebo vrchol
- **void VlozPolozky(PolozkaGuiGraf[] polozky)** – vloží do grafu hrany a vrcholy odpovídající položkám vizualizace předaným parametrem metody a přidá položky z parametru do pole *polozky*
- **bool zmensiGrafPokudJeNutne(PolozkaGuiGraf[] polozky)** – upraví pozice předaných položek, pokud se nevejdou na plátno grafu, vrací *true* pokud došlo k úpravě
- **void ZrusZvyraznenePolozkyAlgoritmus()** – zruší zvýrazňování položek, které byly zvýrazněny algoritmem

5.2.6 Třída GuiGrafGdi

Tato třída je potomkem abstraktní třídy *GuiGraf* a je hlavní třídou pro vizualizaci grafu, se kterou se uživatel setká. Pro vykreslování grafu využívá funkcí grafické knihovny GDI+, která je součástí .NET framework. Třída kreslí na povrch instance třídy *DbPanel*, která je vytvořena povrchu rodičovského formuláře jako jeho komponenta. Třída také reaguje na aktivity uživatele nad vykresleným grafem a podle aktivního nástroje, polohy kurzoru a stisknutého tlačítka myši provádí různé manipulace s grafem.

Pro vykreslování grafů využívá třída principu double bufferingu. Pokud něco vykreslujeme bez double bufferingu, kreslíme přímo na výstupní zařízení, v případě této aplikace bychom kreslili přímo na povrch komponenty *DbPanel* a to způsobuje blikání obrazu, které je velmi nepříjemné. Proto jsem v této třídě implementoval vykreslování na principu double bufferingu. Při generování obrazu s pomocí double bufferingu máme v paměti uchovány dva obrazové buffery. Tyto dva buffery se střídavě vykreslují na obrazovku,

a zatímco je obsah jednoho bufferu zobrazen na výstupním zařízení, do druhého bufferu v paměti se vykreslují data dalšího snímku. Ve chvíli, kdy je snímek do nezobrazeného bufferu dokreslen, tento se zobrazí na výstupní zařízení a do bufferu, jež byl předím zobrazen se začne kreslit další snímek animace. Díky tomu, že je vždy zobrazen celý dokreslený snímek, nedochází k blikání obrazu. V této třídě jsem double buffering implementoval pomocí bitmapy, do které vždy kreslím aktuální data a ve chvíli, kdy jsou vykreslena, tuto bitmapu překreslím na povrch komponenty *DbPanel*. Tím je odstraněno blikání, které by vznikalo, kdybych data kreslil přímo na *DbPanel*. Pro další urychlení vykreslování, třída nepřekresluje při každém novém snímku celý graf, ale vykresluje pouze jeho změněné části. Třída obsahuje následující datové složky:

- **System.Drawing.Bitmap frameBufferBmp** – bitmap používaný jako obrazový buffer
- **System.Drawing.Graphics frameBufferGraphics** – instance třídy zapouzdřující povrch, na který se dá kreslit, v tomto případě povrch bitmapy obrazového bufferu
- **int hranaCislo, vrcholCislo** – hodnoty aktuálního čísla přidávaného vrcholu a hrany pro vytváření jedinečných názvů vrcholů a hran při jejich přidávání do grafu
- **DbPanel platnoDbPanel** – instance komponenty *DbPanel* na jejíž povrch se zobrazuje graf
- **System.Drawing.Pen pozadiPen, System.Drawing.Brush pozadiBrush** – pero a štětec používaný při invalidaci nakreslených prvků grafu
- **System.Drawing.Bitmap pozadiPlatnoBitmap** – bitmapa, na které je vykresleno pozadí, využívá se při invalidaci vykreslených položek grafu
- **Bod predchoziPolohaKurzoru** – poloha kurzoru získaná v předchozím průchodu metody *platnoOnMouseMove*, využívá se například při posouvání částí grafu pro výpočet vektoru posunu
- **Bod prvniBodVyberu** – poloha kurzoru myši při stisknutí tlačítka, využívá se například při kreslení obdélníku výběru
- **Bod velikostInvalidaceInfo** – rozměr plochy, kterou je třeba invalidovat pro odstranění vykresleného informačního obdélníku

- **System.Drawing.Rectangle viditelnaOblast** – obdélník s polohou a rozměry viditelné části povrchu kreslícího plátna
- **System.Drawing.Pen vyberPen** – pero pro kreslení obdélníku výběru

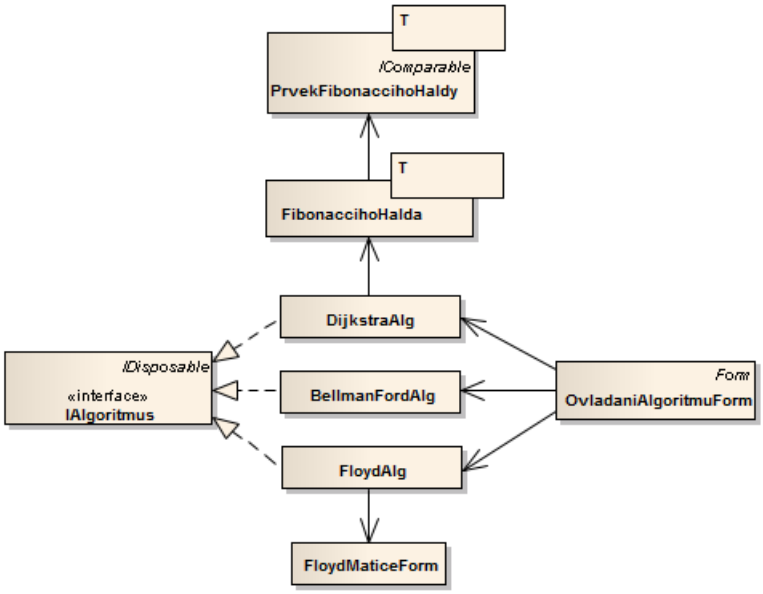
Metody třídy:

- **void casovacOnTick(object sender, EventArgs e)** – metoda ošetří událost OnTick časovače pro zobrazení informací o zvýrazněném prvku grafu
- **Bod dejBodUprostredBeziery(Bod[] bodyBezierovyKrivky)** – metoda vrací bod, který leží uprostřed Bézierovy křivky dané kontrolními body z parametru metody
- **System.Drawing.Rectangle dejViditelnouOblast()** – vrací obdélník viditelné oblasti povrchu komponenty platnoDbPanel
- **void invalidujInfoDoFrameBufferu(), invalidujObdelnikVyberuDoFrameBufferu(), invalidujPodrobnouHranuDoFrameBufferu(), invalidujPolozkyDoFrameBufferu(), invalidujPridavanouHranuDoFrameBufferu()** – všechny metody provádí invalidaci různých prvků do obrazového bufferu (odstranění vykreslených prvků z plátna tím, že je překreslí pozadím), zrychlí se tím vykreslování jednotlivých snímků, protože se nevykresluje znovu celý graf
- **void invalidujPolozku(PolozkaGuiGraf polozka, Graphics gr)** – invaliduje zadanou položku na zadaném povrchu
- **void nastavFrameBuffer()** – vytvoří prázdný obrazový buffer
- **void platnoOnMouseDown(object sender, MouseEventArgs), platnoOnMouseMove(object sender, MouseEventArgs), platnoOnMouseUp(object sender, MouseEventArgs)** – metody ošetřující interakci (stisk tlačítka, pohyb kurzoru, uvolnění tlačítka) myši s povrchem kreslícího plátna
- **void prekresliGrafBezPolozek(PolozkaGuiGraf[] nevykreslovanePolozky, bool nastavitPozadi)** – překreslí graf bez položek předaných parametrem

- **void rodicFormOnResize(object sender, EventArgs e), rodicFormOnScroll(object sender, ScrollEventArgs e)** – metody ošetřující události (změna velikosti, posunutí) vyvolané rodičovským formulářem
- **void vykresliFrameBuffer()** – vykreslí obrazový buffer na povrch plátna
- **void vykresliHranu(PolozkaHranaGuiGraf hrana, Graphics gr, Pen pero, bool zvyraznit), vykresliVrchol(PolozkaVrcholGuiGraf vrchol, Graphics gr, Pen obrysPen, Brush vyplnBrush, bool zvyraznit)** – vykreslí položku grafu na určený povrch podle zadaných parametrů
- **void vykresliInfoDoFrameBufferu(), vykresliObdelnikDoFrameBufferu(Bod polohaKurzor), vykresliPodrobnouHranuDoFrameBufferu(), vykresliPridavanouHranuDoFrameBuferu(Bod polohaKurzor)** – metody vykreslí různé prvky do obrazového bufferu
- **void vykresliPolozku(PolozkaGuiGraf polozka, Graphics gr)** – vykreslí položku předanou parametrem metody na daný grafický povrch
- **void vymazFrameBuffer()** – vymaže obrazový buffer
- **void zvyrazniManipulacniBodKrivkyDoFrameBufferu(bool prvniBod), zvyrazniPolozkyDoFrameBufferu()** – metody zvýrazní manipulační bod křivky nebo položku grafu do obrazového bufferu
- **void zvyrazniPolozku(PolozkaGuiGraf polozka, Grapgics gr), zvyrazniPolozkuAlgoritmus(PolozkaGuiGraf polozka, Graphics gr)** – vykreslí položku zvýrazněně na zadaný grafický povrch

5.3 TŘÍDY GRAFOVÝCH ALGORITMŮ

Poslední částí aplikace, jejíž třídy budu popisovat, je implementace algoritmů a jejich ovládání. Na obrázku číslo 11 je UML diagram těchto tříd. V aplikaci jsou k dispozici tři algoritmy hledání nejkratší cesty Bellmanův-Fordův, Dijskstrův a Floydův. Pro ovládání jejich průběhu aplikace obsahuje formulář ovladače. Na tomto ovladači je možné zvolit, jak průběh algoritmu zobrazit, zda jej chceme krokovat ručně či automaticky nebo zvolit mezi kterými vrcholy chceme hledat cestu. Ovladač obsahuje textové pole, kde jsou vypisovány texty popisující aktuálně probíhající kroky algoritmu.



Obrázek 11:UML diagram třídagoritmů hledání nejkratších cest

5.3.1 Rozhraní IAlgoritmus

Rozhraní IAlgoritmus musí implementovat všechny třídy algoritmů, které mají být v aplikaci použity pro jejich provádění a vizualizaci. Toto rozhraní je předpisem určujícím, které metody, vlastnosti a události musí mít třídy algoritmů implementovány, aby je mohla aplikace používat. To umožňuje velmi jednoduché přidání dalších algoritmů do aplikace a přispívá k možnostem její rozšiřitelnosti. Rozhraní IAlgoritmus definuje tuto vlastnost a událost:

- **string Nazev** – vrací název algoritmu
- **EventHandler Dokonceno** – událost, která je vyvolána při ukončení práce algoritmu

Rozhraní předepisuje metody:

- **string Dalsi()** – metoda, která vykoná další krok algoritmu a vrací řetězec se zprávou o něm
- **void MinimalizujOkna** – volání metody zapříčiní minimalizaci všech formulářových oken, která otevřela instance třídy algoritmu
- **string Start(GuiGraf guiGraf, GrafDP graf, VrcholDP zVrchol, VrcholDP doVrchol)** – inicializuje algoritmus pro hledání nejkratší cesty a vrátí textovou zprávu o průběhu inicializace
- **string Stop()** – ukončí algoritmus a vrátí o tom zprávu

5.3.2 Třída BellmanFordAlg

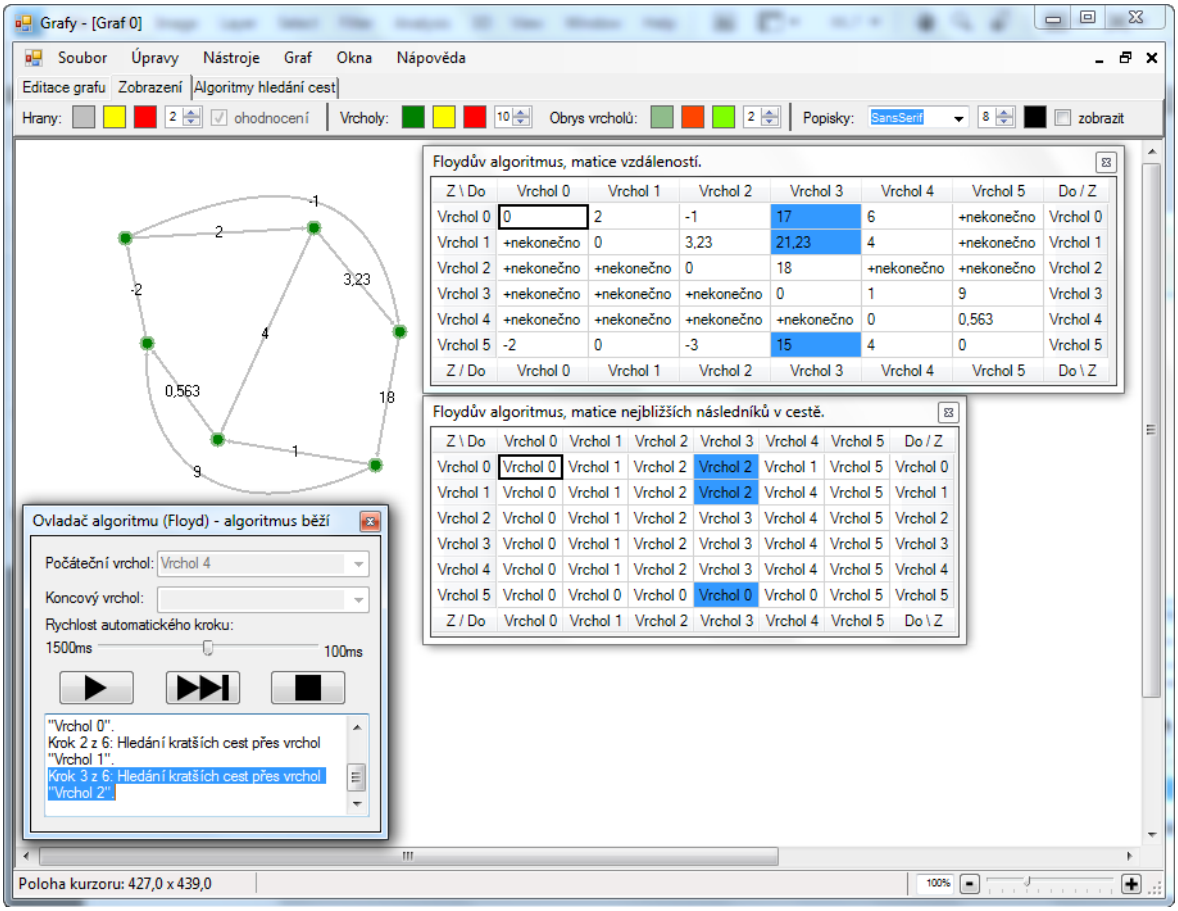
Třída implementuje Bellmanův-Fordův algoritmus hledání nejkratších cest v grafu. Dědí všechny metody, vlastnost a událost z rozhraní *IAlgoritmus* a některé další metody, ty zde ale nebudu popisovat. Průběh algoritmu je zobrazován přímo na vizualizaci grafu.

5.3.3 Třída DijkstraAlg

Instance této třídy provádí nad daným grafem hledání nejkratší cesty a vizualizaci Dijkstrova algoritmu. Je implementací rozhraní *IAlgoritmus*. Pro uchování nedefinitivně ohodnocených vrcholů a výběr vrcholu s nejmenším ohodnocením používá Fibonacciho haldu, která je naprogramovaná v třídě *FibonaccihoHalda*. Vizualizace hledání nejkratší cesty Dijkstrovým algoritmem probíhá stejně jako Belmannova-Fordova algoritmu - přímo na vykresleném grafu.

5.3.4 Třída FloydAlg

Tato třída zapouzdřuje chování Floydova algoritmu pro účely této aplikace. Je samozřejmé, že implementuje rozhraní *IAlgoritmus*. Pro zobrazení průběhu algoritmu používá dva formuláře s tabulkami (třída *FloydMaticeForm*), ve kterých jsou vepsány matice vzdáleností a matice nejbližších následníků. Tyto formuláře je možné vidět na obrázku číslo 12.



Obrázek 12: Formuláře s vypsanými maticemi Floydova algoritmu a ovladač algoritmu

5.3.5 Třída OvladaniAlgoritmuForm

Tato třída je používána pro ovládání průběhu zvoleného algoritmu. Jak vypadá okno ovladače, je možné vidět vlevo dole na obrázku číslo 12. Třída *OvladaniAlgoritmuForm* má tyto datové složky:

- **IAlgoritmus algoritmus** – instance algoritmu, který je třídou ovládán
- **Timer casovac** – časovač pro automatické krokování algoritmu
- **bool dokonceno** – logická hodnota indikující zda činnost algoritmu skončila
- **int posledniPozicetextBox** – číslo pozice posledního znaku v textovém poli
- **bool puvodniZobrazeniOhodnoceniHran** – logická hodnota značící, zda před spuštěním algoritmu bylo zobrazeno ohodnocení hran
- **bool vyberZComboBox** – logická hodnota sloužící pro určení, ve kterém roletovém poli pro výběr počátečního a koncového vrcholu se má označit vrchol, na který bylo kliknuto

Třída obsahuje tuto vlastnost:

- **IAlgoritmus Algoritmus** – slouží k nastavení algoritmu ovládaného třídou

Metody třídy:

- **void algoritmusOnDoknceno(object sender, EventArgs e)** – metoda ošetřuje událost *Dokonceno* algoritmu
- **void dalsi()** – volá metodu aktuálního algoritmu pro vykonání dalšího kroku
- **void dalsiButton_Click(object sender, EventArgs e), stopButton_Click(object sender, EventArgs e), startPauseCheckBox_CheckedChange(object sender, EventArgs e)** – metody ošetřují události kliknutí na jednotlivá tlačítka ovladače
- **void Minimalizuj()** – minimalizuje okno ovladače a okna algoritmu, pokud jsou nějaká zobrazena
- **void nastavStavTextBox(string text) -** připíše na konec textového pole text předaný parametrem a označí jej
- **void OnClosed(EventArgs e), OnClosing(ClosingEventArgs e)** – ošetřuje události vyvolané při zavírání a po zavření formuláře ovladače
- **void OnGrafAlgoritmusMouseDown(object sender, GrafAlgoritmusMouseDownEventArgs e) -** ošetřuje událost klinutí na plátno grafu, když je otevřené okno ovladače, používá se například pro výběr počátečního a koncového vrcholu
- **void onTimerTick(object sender, EventArgs e)** – ošetřuje událost tiknutí časovače pro automatický přechod na další krok algoritmu
- **void rychlostTrackBar_ValueChanged(object sender, EventArgs e)** – metoda ošetřuje událost změna hodnoty na posuvníku nastavení času automatického krokování
- **void scrollujNaKonecTextBoxu(TextBox textBox)** – zobrazí poslední řádky textového pole
- **void Show(GrafDP graf)** – zobrazí formulář ovladače
- **void Start()** – spustí vizualizaci algoritmu

5.4 OVĚŘENÍ SPLNĚNÍ POŽADAVKŮ NA APLIKACI

Na následujících řádcích bych chtěl provést ověření a zhodnocení splnění požadavků kladených na aplikaci.

5.4.1 Grafické zadávání a editace grafu

Tento požadavek byl splněn. Aplikace umožňuje zadávání a úpravy grafu grafickou metodou velmi podobnou kreslení ve značně jednoduchém programu pro tvorbu grafiky.

5.4.2 Výpis vlastností grafu

U zadaného grafu je možné zobrazit okno, ve kterém jsou vypsány všechny informace o grafu, jež byly obsaženy v zadání. Z toho plyne, že tento požadavek byl splněn.

5.4.3 Zobrazování průběhu algoritmů po krocích

Jak bylo popsáno výše, ovládání jednotlivých algoritmů umožňuje jejich průběh krokovat. Krokovat algoritmy je možné ručně i automaticky v určených intervalech. Průběh algoritmu je možné kdykoliv přerušit a spustit jej od začátku. Požadavek byl splněn.

5.4.4 Zobrazení případných dalších hodnot v tabulkách

U Floydova algoritmu, který pracuje s maticí nejbližších následníku v cestě a maticí délek hran, kterou transformuje na matici vzdáleností, se tyto matice zobrazují ve vlastních oknech. U zbylých dvou algoritmů, jsem nepovažoval za nutné nějaké další hodnoty zobrazovat. Proto i tento požadavek pokládám za splněný.

5.4.5 Uživatelská příručka

Uživatelskou příručku k aplikaci jsem vytvořil, čímž jsem splnil i požadavek na ni.

5.4.6 Export a import grafů

Grafy vytvořené v aplikaci je možné uložit do zvoleného souboru i je následně ze souboru načíst. Navíc aplikace také nabízí možnost exportovat obrázek grafu do souborů několika grafických formátů, konkrétně BMP, JPEG, PNG. Požadavek byl splněn.

5.4.7 Popis jednotlivých kroků probíhajícího algoritmu

V okně ovladače algoritmu je textové pole, do kterého se popisují kroky probíhajícího algoritmu včetně zvýraznění aktuálního popisku. I tento požadavek byl splněn.

5.4.8 Implementace funkcí práce se schránkou

Požadavek na práci se schránkou byl splněn. Aplikace umí vyjmout i kopírovat označené části grafu a následně je vkládat na jiné místo stejného, nebo jiného grafu. Volání těchto funkcí je umožněno i pomocí klávesových zkratek Ctrl+X, Ctrl+C a Ctrl+V.

5.4.9 Implementace funkcí historie

Aplikace uchovává až 30 kroků historie změn. Historii je možné libovolně procházet vzad i vpřed, kliknutím na položku v menu nebo pomocí klávesových zkratek Ctrl+Z a Ctrl+Y. Požadavek byl splněn.

5.4.10 Možnost práce s několika grafy najednou

Aplikace je naprogramovaná tak, aby bylo možné otevřít nebo vytvořit více grafů najednou a pracovat s nimi. Tento požadavek byl splněn.

5.4.11 Přívětivé uživatelské prostředí

Při tvorbě aplikace jsem kladl důraz, aby vzhled všech prvků aplikace byl co nejvíce popisný a výstižný. Navíc téměř každý prvek aplikace obsahuje textová popisek, který se zobrazí, pokud nad prvkem chvíli setrvá kurzor myši. Také jsem se snažil vytvořit jednotlivé panely aplikace co nejvíce přehledné a ovládaní vytvořit co nejjednodušší. Proto si myslím, že i tento požadavek se mi podařilo splnit, pokud ne kompletně tak alespoň z větší části.

5.4.12 Modulárnost - rozšiřitelnost aplikace

Už z popisu jednotlivých tříd plyne, že splnit tento požadavek jsem se opravdu snažil a myslím si, že se mi to i podařilo a požadavek jsem splnil

5.4.13 Využívání principů OOP

Dodržování a využívání principů objektově orientovaného programování je z velké části zaručeno použití čistě objektového jazyka C#. Také z implementace tříd popsaných

v předchozím oddíle je možné vidět použití různých principů OOP jako dědičnost, polymorfismus. Tento požadavek byl splněn.

5.5 NÁVRHY NA VYLEPŠENÍ APLIKACE

Myslím si, že by aplikaci slušelo, kdyby uživateli byl umožněn výběr grafické knihovny pro vykreslování grafů. Určitě by stačilo do aplikace přidat k naprogramované vizualizaci pomocí knihovny GDI+ vizualizaci vytvořenou jednou z akcelerovaných knihoven OpenGL a DirectX. A to hlavně z důvodu, že na strojích se slabším procesorem může být vykreslování grafů pomalejší. Což by mohlo být při použití zmiňovaných knihoven odstraněno.

Dalším zajímavým vylepšením by mohlo být rozšíření aplikace o vizualizaci dalších grafových algoritmů. Tím by se z ní stal velmi univerzální nástroj pro seznámení uživatelů s principy grafových algoritmů.

V neposlední řadě by aplikace mohla umět mimo exportu a importu do vlastního formátu i export a import do obecných grafových formát, které byly vyvinuty pro ukládání grafů. Mezi tyto formáty patří například GraphML nebo GXL.

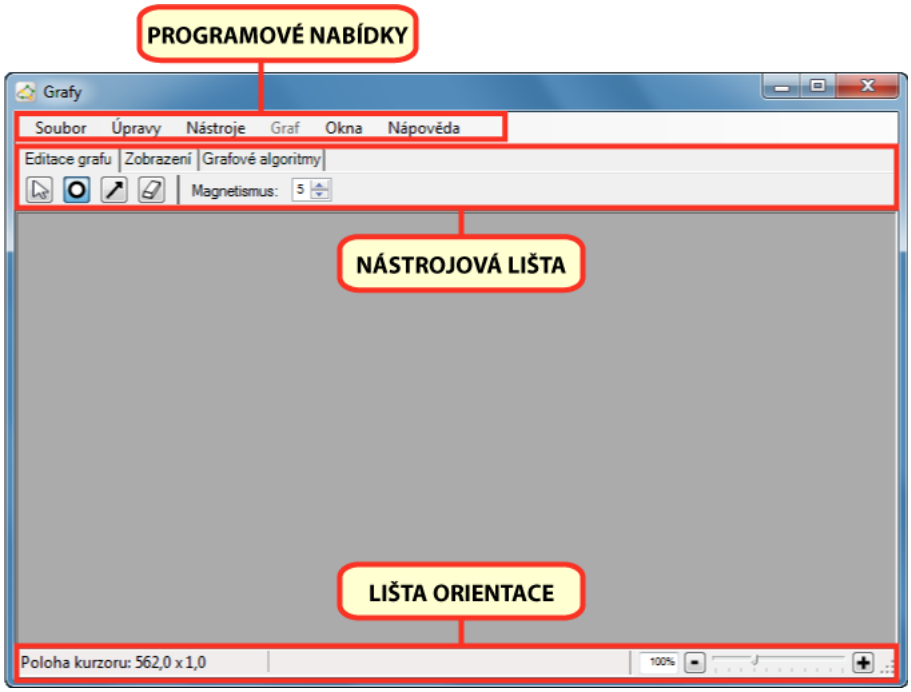
6 POPIS A OVLÁDÁNÍ APLIKACE

V této kapitole budou popsány prvky uživatelského rozhraní aplikace a dále bude popsáno její ovládání.

6.1 HLAVNÍ OKNO APLIKACE

Po spuštění aplikace se na obrazovce objeví hlavní okno, to je vidět na obrázku číslo 13. Z hlavního okna aplikace jsou dostupné všechny její funkčnosti. Okno se skládá ze tří hlavních prvků:

- lišty programových nabídek,
- nástrojové lišty
- a spodní lišty orientace.



Obrázek 13: Hlavní okno aplikace

6.1.1 Lišta programové nabídky

Tento ovládací panel obsahuje roletková menu, ze kterých je možná spouštět nebo přepínat různé funkce aplikace. Jsou zde zastoupeny jak standardní funkce typu otevřít soubor, uložit soubor či zpět a znovu, tak například položky pro výběr nástroje editace grafu nebo zobrazení vlastností grafu.

6.1.2 Nástrojová lišta

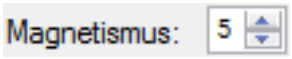
Lišta obsahuje tři záložky s nástroji pro práci s aktuálně otevřeným grafem. První je záložka *Editace grafu*, která obsahuje tlačítka pro zvolení nástroje pro editaci grafu a posuvník pro určení hodnoty magnetismu a je zobrazena na obrázku číslo 13. Mezi čtyři nástroje editace grafu, které aplikace obsahuje, patří nástroj *Výběr*, nástroj *Editace vrcholů*, nástroj *Editace hran* a nástroj *Guma*. Nástroje lze také aktivovat z menu *Nástroje* programové lišty.

 **Výběr** – Slouží k označování položek grafu a jejich přemísťování a úpravu vlastností. Nástroj lze také zvolit pomocí klávesy **A**.

 **Editace vrcholů** – Nástroj se používá pro přidávání vrcholů do grafu a úpravu jejich vlastností. Klávesová zkratka nástroje editace vrcholů je **V**.

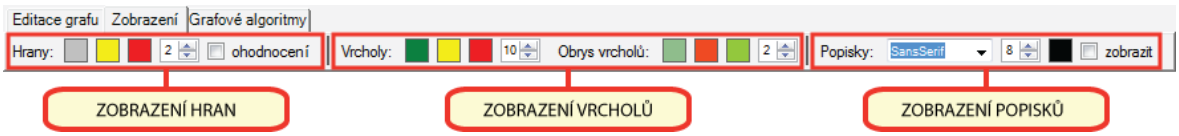
 **Editace hran** – S nástrojem je možné přidávat hrany, upravovat průběh křivek pokud jsou jako křivky vykresleny a měnit vlastnosti hran. Klávesová zkratka **H**.

 **Guma** – Používá se pro mazání prvků grafu. Pro volbu lze použít klávesu **E**.



Magnetismus – Hodnota nastavuje vzdálenost mezi prvkem grafu a kurzorem, při které dojde ke zvýraznění prvku, který je poté možné označit.

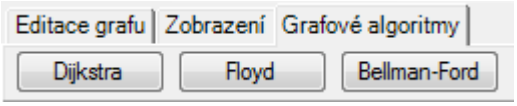
Další záložkou je záložka Zobrazení, která slouží pro úpravu vizuálních vlastností grafu. Záložka je rozdělena do třech částí, jak je vidět na obrázku 14. V první části této záložky je možné nastavit barvy pro vykreslování hran, tloušťku čar jakou budou hrany vykreslovány a zvolit zda se má zobrazovat ohodnocení hran. V prostřední části se nastavují barvy výplně a obrysu vrcholů, včetně průměru vykresleného vrcholu a tloušťky čáry obrysu. Poslední část této záložky umožňuje zvolit, zda zobrazovat popisky prvků grafu, řez písma popisků a jeho výšku.



Obrázek 14: Záložka Zobrazení

Třetí záložka na panelu nástrojů umožňuje spuštění vizualizace vybraných grafových algoritmů. Záložka obsahuje tři tlačítka. První je tlačítko Dijkstra, kterým spustíme

vizualizaci Dijkstrova algoritmu pro hledání nejkratších cest, druhé tlačítko s popiskem Floyd umožňuje spustit vizualizaci Floydova algoritmu pro nalezení matice vzdáleností a poslední tlačítko Bellman-Ford spustí hledání nejkratších cest Bellmanovým-Fordovým algoritmem. Záložku je možné vidět na obrázku číslo 15.



Obrázek 15: Záložka Grafové algoritmy

6.1.3 Lišta orientace

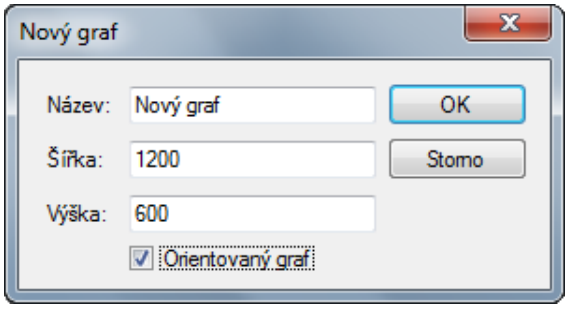
Na liště orientace můžeme najít dva prvky. Na levé straně lišty se nachází text informující o aktuální poloze kurzoru myši nad plátnem grafu. Informace o poloze kurzoru jsou dostupné pouze, pokud je v aplikaci zobrazen nějaký graf. Na pravé straně lišty je ovládací panel úrovně přiblížení plátna grafu. Pomocí něho lze zjistit aktuální hodnotu přiblížení a přiblížit nebo oddálit právě zobrazený graf. To je možné buď zapsáním hodnoty přiblížení v procentech do textového pole, změnou polohy posuvníku na panelu nebo stiskem tlačítek po stranách posuvníku. Panel úrovně přiblížení je na obrázku číslo 16.

Obrázek 16: Panel úrovně přiblížení

6.2 OVLÁDÁNÍ APLIKACE

6.2.1 Vytvoření, otevření a uložení grafu

Vytvoření nového grafu – Nový graf vytvoříme kliknutím na menu **Soubor | Nový** nebo použitím klávesové zkratky **Ctrl+N**, vyplněním položek dialogu *Nový graf* a potvrzením tlačítkem OK. V dialogu *Nový graf* (obrázek číslo 17) je nutné vyplnit název grafu, rozměry plátna, na které se bude graf vykreslovat a určit, zda se jedná o orientovaný graf.



Obrázek 17: Dialog Nový graf

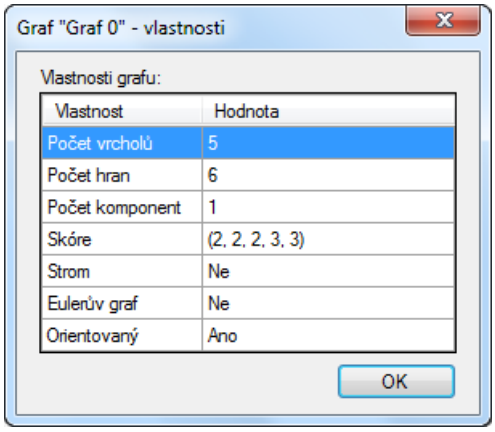
Otevření grafu ze souboru – Pro otevření grafu klikneme na menu **Soubor | Otevřít** a v dialogovém okně výběru souboru zvolíme soubor s grafem, který chceme otevřít. Pro vyvolání dialogového okna otevření souboru lze použít klávesovou zkratku **Ctrl+O**.

Uložení grafu do souboru – Graf uložíme do souboru kliknutím na menu **Soubor | Uložit**, stisknutím klávesové zkratky **Ctrl+S**, nebo kliknutím na menu **Soubor | Uložit** jako a v dialogovém okně uložení do souboru vybereme umístění a název nového souboru nebo vybereme soubor, který chceme přepsat.

Export do obrázku – Graf můžeme kliknutím na menu **Soubor | Export** exportovat do obrazového souboru, jehož typ zvolíme v dialogu uložení do souboru.

6.2.2 Vlastnosti grafu

Zobrazit vlastnosti grafu je možné kliknutím na menu **Graf | Vlastnosti** nebo použitím klávesové zkratky **Alt+Enter**. Vlastnosti grafu ze zobrazí v okně, jako je možné vidět na obrázku číslo 18.



Obrázek 18: Okno vlastnosti grafu

6.2.3 Editace vrcholů grafu

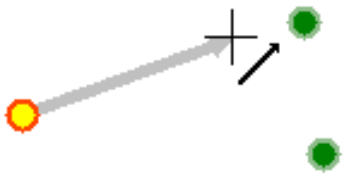
Přidání nového vrcholu grafu – Pro přidávání vrcholů musíme mít aktivní nástroj Editace vrcholů. Nástroj aktivujeme stisknutím na tlačítka na panelu nástrojů, kliknutím na menu **Nástroje | Editace vrcholů** nebo stisknutím klávesy **V**. Když je nástroj *Editace vrcholů* aktivní stačí pro přidání vrcholu do grafu jen kliknout na plátno grafu a vyplnit název grafu.

Změna názvu vrcholu grafu – Název vrcholu lze změnit, pokud je aktivní nástroj *Editace vrcholů* nebo *Výběr*. Název změníme v dialogovém okně, které se objeví, když kurzorem myši najedeme nad vrchol, tak aby se zvýraznil, a poté klikneme pravým tlačítkem myši.

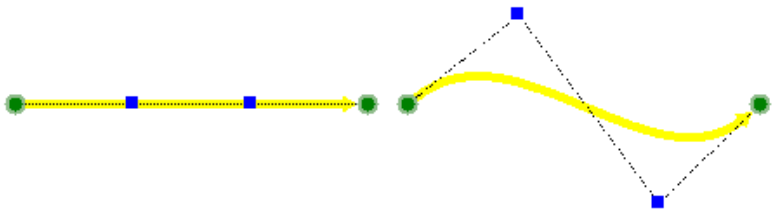
6.2.4 Editace hran grafu

Přidání nové hrany – Novou hranu je možné přidat pouze, pokud je aktivní nástroj *Editace hran*. Aktivaci nástroje Editace hran můžeme provést stisknutím tlačítka  , kliknutím na menu **Nástroje | Editace hran**, případně pomocí klávesy **H.** Hranu lze přidat jen mezi dva vrcholy, které v grafu již existují. Přidání provedeme tak, že najedeme kurzorem myši nad vrchol grafu, ze kterého má hrana vycházet, a stiskneme levé tlačítko myši. Se stisknutým tlačítkem myši přetáhneme její kurzor nad vrchol, do kterého má hrana vstupovat, a tlačítko uvolníme. Vyplníme údaje o hraně a potvrdíme dialog. Hrana se během tažení myší bude vykreslovat z počátečního vrcholu do místa pod kurzorem myši jako je vidět na obrázku číslo 19.

Změna vlastností hrany – Změnit vlastnosti hrany jako je název a ohodnocení lze, pokud na ni klikneme pravým tlačítkem myši a je aktivní nástroj *Výběr* nebo *Editace hran*. Editovat průběh křivky hrany lze jen s aktivním nástrojem *Editace hran* a to tak, že na hranu klikneme levým tlačítkem myši. Hrana se zobrazí v režimu editace křivky, jak je vidět na obrázku číslo 20. V tomto režimu můžeme manipulovat se 2 body určujícími průběh křivky stisknutím levého tlačítka nad jedním z nich a tažením na nové místo.



Obrázek 19: Vytváření Nové Hrany

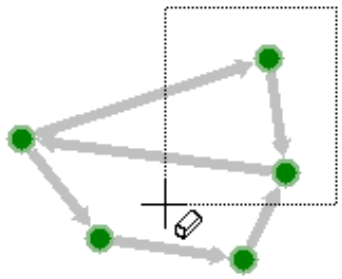


Obrázek 20: Hrana v režimu editace křivky

6.2.5 Mazání prvků grafu

Mazat vrcholy a hrany grafu lze pomocí nástroje *Guma*, který aktivujeme stisknutím tlačítka , kliknutím na menu **Nástroje | Guma** nebo stiskem klávesy **E**. Nástrojem *Guma* můžeme mazat dvěma způsoby, první je umístění kurzoru nad některý prvek a stisknutí levého tlačítka myši, tím dojde k odstranění tohoto prvku z grafu.

Druhým způsobem je stisknutí levého tlačítka myši a tažení kurzoru myši přes prvky tak, aby zobrazovaný obdélník obsahoval všechny prvky, které chceme smazat, ukázka je na obrázku číslo 21. Prvky budou smazány ve chvíli, kdy tlačítko uvolníme. Mazat prvky lze i bez aktivovaného nástroje *Guma*, a to tak, že jeden nebo více prvků označíme a stiskneme klávesu **DELETE** nebo menu **Úpravy | Smazat.**



Obrázek 21: Mazání výběrem pomocí nástroje guma

6.2.6 Označování, posun, kopírování a vkládání prvků grafu

Označení prvků grafu - Prvky grafu je možné označit nástrojem *Výběr*. Nástroj *Výběr* se aktivuje stiskem tlačítka , kliknutím na **Nástroje | Výběr** nebo stisknutím klávesy **A.** Hrany a vrcholy můžeme označovat jednotlivě kliknutím na ně. Pokud chceme označit více vrcholů a hran najednou, ohraničíme je obdélníkem výběru podobně jako u mazání více prvků najednou nástrojem. Přidat další prvky k aktuálnímu výběru můžeme jejich označením se stisknutou klávesou **SHIFT**. Pokud chceme označit všechny prvky grafu najednou, klikneme na menu **Úpravy | Označit vše** nebo stiskneme klávesovou zkratku **Ctrl+A**.

Posun prvků grafu – Označené prvky grafu můžeme nástrojem *Výběr* posouvat po plátně. Posunutí provedeme tak, že umístíme kurzor myši nad některou označenou položku, stiskneme levé tlačítko myši a se stlačeným tlačítkem přesuneme kurzor na místo, kam chceme položky přesunout a zde tlačítko uvolníme.

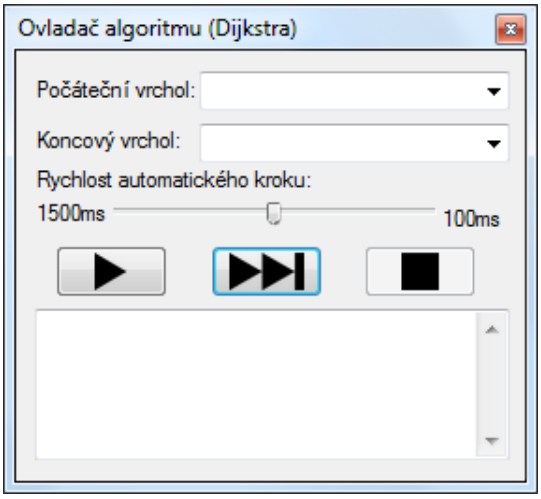
Kopírování prvků do schránky – Pro zkopírování označených položek grafu do schránky OS klikneme na menu **Úpravy | Kopírovat** nebo použijeme klávesovou zkratku **Ctrl+C**.

Vkládání prvků grafu ze schránky – Pokud jsou ve schránce zkopírované nějaké hrany a vrcholy, můžeme je vložit do otevřeného grafu kliknutím na menu **Úpravy | Vložit** nebo stisknutím klávesové zkratky **Ctrl+V**. Prvky ze schránky se vloží pod kurzor myši.

6.2.7 Vizualizace grafových algoritmů

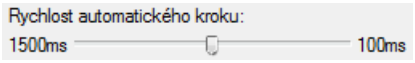
Výběr algoritmu – Na záložce *Grafové algoritmy Nástrojové lišty* klikneme na tlačítko algoritmu, který chceme vizualizovat. Po kliknutí na tlačítko zvoleného algoritmu se objeví okno ovladače algoritmů. Toto okno můžeme vidět na obrázku číslo 22.

Příprava před spuštěním algoritmu – Pokud to algoritmus vyžaduje pro svou funkci, je nutné vybrat počáteční vrchol, případně i koncový. Výběr vrcholů se provádí zvolením řádku s názvem vrcholu v jednom z rozbalovacích seznamů, které se nachází v horní části ovladače, nebo umístěním kurzoru nad vrchol a kliknutím na něj. Při prvním kliknutí se vrchol pod kurzorem myši vybere v rozbalovacím seznamu pro výběr počátečního vrcholu, při následujícím kliknutí se vrchol vybere v rozbalovacím seznamu koncového vrcholu.



Obrázek 22: Okno ovladače algoritmu

Ovládání algoritmu – K ovládání běhu algoritmu jsou na ovladači připravena tři tlačítka a jeden posuvník.



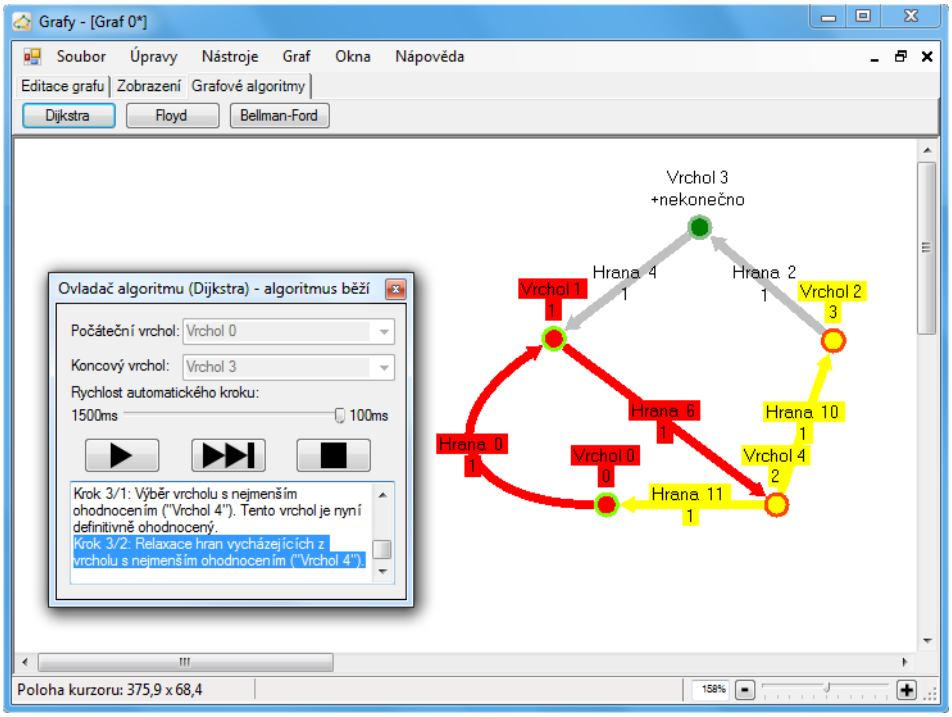
Rychlost automatického kroku – Slouží k nastavení časového intervalu mezi jednotlivými kroky automatického krokování. Interval je možné nastavit v rozmezí 0,1s – 1,5s.

Spustit utomatické krokování algoritmu – Spustí vizualizaci algoritmu s automatickým posunem na další krok v zadaném intervalu.

Další krok algoritmu – Přejde na další krok spuštěného algoritmu. Pokud algoritmus neběží spustí jej v režimu ručního krokování.

Pozastavit automatické krokování algoritmu – Pozastaví automatické krokování algoritmu. Pokud je běh algoritmu pozastaven, lze dál krokovat ručně nebo v automatickém krokování pokračovat.

Zastavit algoritmus – Ukončí probíhající algoritmus.



Obrázek 23: Aplikace s probíhajícím Dijkstrovým algoritmem na malém grafu

Výstup algoritmů – Algoritmy zobrazují svůj průběh přímo na grafu zvýrazňováním vrcholů a hran, se kterými právě pracují, a dále do textového pole na dolní straně ovladače

vypisují slovní popis aktuálních kroků. Některé algoritmy zobrazují navíc další výstup do vlastních oken. Na obrázku 23 je zobrazen jeden z kroků Dijkstrova algoritmu na jednoduchém grafu.

7 ZÁVĚR

V teoretické části práce byly shrnuty poznatky z oblasti teorie grafů. Dále byly podrobně rozebrány tři základní algoritmy hledání nejkratších cest v grafech, konkrétně Bellmanův-Fordův algoritmus, Dijkstrův algoritmus a Floydův algoritmus. Poznatky z této části jsem využil při implementaci aplikace. Teoretická část práce společně s vytvořenou aplikací by měla čtenáři pomoci tyto algoritmy pochopit.

Hlavním cílem diplomové práce bylo vytvořit aplikaci, která umožní vytvářet grafy, zobrazovat jejich vlastnosti a především dokáže vizualizovat průběh algoritmů hledání nejkratších cest na grafech. Tento cíl byl splněn. Aplikace umožňuje vytvoření a editaci grafů velmi intuitivně s pomocí myši, dále umožňuje zjišťování vlastností grafů a dokáže vizualizovat průběh Bellmanova-Fordova, Dijkstrova a Floydova algoritmu. Zobrazovat jednotlivé kroky algoritmů umí aplikace ve dvou režimech, automaticky v daných intervalech nebo je může ručně ovládat uživatel. To vše aplikace dokáže v jednoduchém a přehledném uživatelském prostředí.

Protože byla aplikace navržena s důrazem kladeným na možnost budoucího rozšíření, může být jednoduše obohacena o možnost vizualizovat i další zajímavé algoritmy teorie grafů a jiné funkčnosti.

BIBLIOGRAFIE

[Demel, 2002] DEMEL, Jiří. *Grafy a jejich aplikace*. Praha : Academia, 2002. 257 s. ISBN 80-200-0990-6.

[Volek, 2005] VOLEK, Josef. *Operační výzkum I*. Pardubice : Univerzita Pardubice, 2005. 111 s. ISBN 80-7194-410-6.

[Wiki01] Skóre grafu. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, [cit. 2010-08-02]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Sk%C3%B3re_grafu>.

[Jirovský, 2008] JIROVSKÝ, Lukáš. *Teorie grafů* [online]. 2008, 28. 5. 2008 [cit. 2010-08-03]. Dostupné z WWW: <http://teorie-grafu.elfineer.cz/>.

[Černý, 2008] ČERNÝ, Jakub. *Základní grafové algoritmy* [online]. [s.l.] : [s.n.], 2008 [cit. 2010-08-03]. Dostupné z WWW: <http://kam.mff.cuni.cz/~kuba/ka/ka.pdf>.

[Wiki02] Dijkstra's algorithm. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 11:41, 1 March 2002, last modified on 05:21, 5 August 2010 [cit. 2010-08-05]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm>.

[Weiss, 2010] WEISSTEIN, Eric W. *MathWorld : A Wolfram Web Resource* [online]. c2010 [cit. 2010-08-05]. Floyd-Warshall Algorithm. Dostupné z WWW: <http://mathworld.wolfram.com/Floyd-WarshallAlgorithm.html>.

[Wiki03] Floyd-Warshall algorithm. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 08:52, 20 May 2003, last modified on 22:41, 31 July 2010 [cit. 2010-08-05]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm>.

[Wiki04] Bellman-Ford algorithm. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 12:31, 4 May 2003, last modified on 02:14, 3 August 2010 [cit. 2010-08-05]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm>.

[Wiki05] Object-oriented programming. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 21:45, 25 October 2001, last modified on 02:52, 10 August 2010 [cit. 2010-08-10]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Object-oriented_programming>.

[Wiki06] C Sharp (programming language). In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 20:45, 25 October 2001, last modified on 20:43, 10 August 2010 [cit. 2010-08-11]. Dostupné z WWW: <http://en.wikipedia.org/wiki/C_Sharp_%28programming_language%29>.

[Wiki07] .NET Framework. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 18. 11. 2005, 15:57, last modified on 18. 11. 2005, 15:57 [cit. 2010-08-11]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/.NET_Framework>.

[Wiki08] Graphics Device Interface. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 12:36, 6 November 2002, last modified on 03:23, 9 July 2010 [cit. 2010-08-14]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Graphics_Device_Interface>.