UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

Porovnání algoritmů vyhledávání nejkratších cest

Lumír Gago

Bakalářská práce
2013

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2012/2013**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Lumír Gago**
Osobní číslo:	**I10045**
Studijní program:	**B2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Porovnání algoritmů vyhledávání nejkratších cest**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování:

Primárním cílem bakalářské práce je porovnání vyhledávání nejkratších cest v rámci hranově ohodnoceného (planárního) grafu zejména pomocí Dijkstrova algoritmu a algoritmu A*.
Reprezentace grafu je postavena nad vhodnou abstraktní datovou strukturou umožňující efektivní implementace výše zmíněných algoritmů.
Evoluce výpočtů algoritmů bude rovněž graficky ilustrována v rámci jednoduchého zobrazovacího prostředí.
Pro testování cílové aplikace se použije vybraný segment reálné dopravní sítě.

Rozsah grafických prací:

Rozsah pracovní zprávy:

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

Seznam odborné literatury:

CENEK, P. , KLIMA, V., JANÁČEK, J. Optimalizace dopravních a spojových procesů, Žilina, Univerzita Žilina, 1994.
VOLEK, J. Operační výzkum I., skripta DFJP UPa, Pardubice 2002
CORMEN a kol.: Introduction to algorithms, MIT Press, Cambridge, 2001

Vedoucí bakalářské práce: **prof. Ing. Antonín Kavička, Ph.D.**
Katedra softwarových technologií

Datum zadání bakalářské práce: **21. prosince 2012**
Termín odevzdání bakalářské práce: **10. května 2013**

L.S.

prof. Ing. Simeon Karamazov, Dr.
děkan

Ing. Lukáš Čegan, Ph.D.
vedoucí katedry

V Pardubicích dne 29. března 2013

Prohlášení autora

Prohlašuji, že jsem tuto práci 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 5. 5. 2013 Lumír Gago

Poděkování

Děkuji prof. Ing. Antonínu Kavičkovi, Ph.D. za přesné vedení bakalářské práce, cenné rady připomínky a veškerý věnovaný čas.

Anotace

Tato práce se zabývá implementací vyhledávacích algoritmů Dijkstrův algoritmus a algoritmus A* pro nalezení nejkratších cest v grafu, návrhem datových struktur pro implementaci planárního grafu a jeho grafické znázornění.

Klíčová slova

Algoritmus A*, Dijkstrův algoritmus, algoritmus, datové struktury, graf, grafika, matice.

Title

Comparison of search algorithms shortest paths

Annotation

Project is aimed to implementation searching algorithm Dijkstra's algorithm and A*'s algorithm that solves the single-source shortest path problem for a graph, design of a data structure for implementation planar graph and its graphic representation.

Keywords

A* algorithm, Dijkstra algorithm, algorithm, data structure, graph, graphic, matrix.

Obsah

1 Popis zadání

Hlavním cílem bakalářské práce je porovnání vyhledávání nejkratších cest v rámci hranově ohodnoceného (planárního) grafu zejména pomocí Dijkstrova algoritmu a algoritmu A*.

Reprezentace grafu je postavena nad vhodnou abstraktní datovou strukturou umožňující efektivní implementace výše zmíněných algoritmů.

Evoluce výpočtů algoritmů bude rovněž graficky ilustrována v rámci jednoduchého zobrazovacího prostředí.

Pro testování cílové aplikace bude použit vybraný segment reálné dopravní sítě.

2 Úkoly práce

Hlavním úkolem práce je implementovat Dijkstrův algoritmus a algoritmus A*. S tímto souvisí volba vhodných datových struktur pro realizaci grafu a četnost volání jednotlivých metod. Často volané metody by měly mít optimalizovanou asymptotickou výpočetní složitost.

Evoluce výpočtů jednotlivých verzí algoritmů bude pro menší rozsah dat zobrazena graficky. Zobrazené evoluce výpočtů algoritmů bude možné uložit do *JPEG* souboru a je možné pro oba algoritmy vypočítat matici vzdáleností.

Práce bude testována na malém rozsahu dat a na větším rozsahu dat. K tomuto účelu bude vygenerován graf ze souboru, kde každý řádek je reprezentován městem České nebo Slovenské republiky.

3 Teoretické předpoklady pro práci

Před samotnou implementací datových struktur a algoritmů je nutné stanovit teoretické předpoklady práce; pečlivě nastudovat jednotlivé kroky algoritmů a plně jim porozumět, určit jaké datové struktury jsou vhodné pro řešený problém.

V této práci je použito následující označení grafu:

$$G = (V, E, \varphi)$$

kde G je graf, V je množina vrcholů, E je množina hran a φ je reálná funkce definovaná na E a přiřazující každé hraně dvojici vrcholů.

3.1 Algoritmy

Algoritmus je schematický postup pro řešení určitého druhu problému, který je prováděn pomocí konečného množství přesně definovaných kroků. Obecná definice zní: „Konečná sada kroků pro dosažení daného cíle“.

3.1.1 Dijkstrův algoritmus

Dijkstrův algoritmus je algoritmus sloužící k hledání nejkratších cest na grafech s kladně ohodnocenými hranami.

Mějme graf G, v němž hledáme nekratší cestu. V je množina všech vrcholů z grafu a E je množina všech hran grafu G. Vrchol s z množiny V je výchozí vrchol a vrchol k z množiny V je koncový vrchol. Dijkstrův algoritmus si pro každý vrchol v z množiny V pamatuje délku nejkratší cesty z výchozího vrcholu do aktuálního vrcholu. Tuto hodnotu označíme jako $d[v]$. Dijkstrův algoritmus si také pamatuje množinu nenavštívených vrcholů, označme ji N, a množinu již navštívených vrcholů, označme ji Y.

Jednotlivé kroky Dijkstrova algoritmu jsou následující:

1. Inicializace, neboli určení výchozího a koncového vrcholu, nastavení $d[v]$ pro všechny vrcholy z V na ∞, mimo s. Pro s nastavíme $d[s]$ na 0. Pokud $d[v] = \infty$, poté k danému vrcholu není známa žádná cesta. V inicializačním kroku všechny vrcholy z množiny V náleží množině N.

2. Odebereme z N vrchol v_{min} s nejmenším ohodnocením $d[v_{min}]$ a zkontrolujeme ohodnocení všech sousedních vrcholů v_n s odebraným vrcholem v_{min}. Pokud je $d[v_{min}] + \varphi(v_{min}, v_n) < d[v_n]$, poté tedy $d[v_n] = d[v_{min}] + \varphi(v_{min}, v_n)$. Vrchol v_{min} je odebrán z množiny N a vložen do množiny Y.

3. Zkontrolujeme, zda je množina N prázdná. Pokud ne, opakujeme krok 2., pokud ano, pokračujeme krokem 4.

4. Zrekonstruujeme zpětně cestu z koncového do výchozího vrcholu. Začneme v koncovém vrcholu k, hledáme předchůdce p, pro kterého platí: $d[p] = d[k] - \varphi(k, p)$. Výsledná hodnota $d[k]$ je délka hledané cesty. Krok 4. Je opakován, dokud nedojdeme k vrcholu p.

V obecném případě je asymptotická výpočetní složitost Dijkstrova algoritmu rovna $O(|V|^2+|E|)$, kde $|V|$ je počet vrcholů grafu a $|E|$ je počet hran.

3.1.2 Algoritmus A*

A* algoritmus, je obdobně jako Dijkstrův algoritmus, také algoritmus, sloužící k hledání nejkratších cest na kladně orientovaných grafech. Využívá stejný princip jako Dijkstrův algoritmus, ale navíc je rozšířen o heuristickou funkci. Heuristická funkce odhaduje skutečnou vzdálenost z aktuálního vrcholu do koncového vrcholu podle určitých kritérií. Hodnota heuristické funkce se poté používá při výběru následujícího nezpracovaného vrcholu z množiny N.

Opět mějme graf G, v němž hledáme nekratší cestu. V je množina všech vrcholů z grafu a H je množina všech hran grafu G. Vrchol s z množiny V je výchozí vrchol a vrchol k z množiny V je koncový vrchol. A* algoritmus si pro každý vrchol v z množiny V pamatuje délku nejkratší cesty z výchozího vrcholu do aktuálního vrcholu, označme ji

d[v], a navíc hodnotu heuristické funkce, označme ji *h[v]*. A* algoritmus si také pamatuje množinu nenavštívených vrcholů, označme ji *N*, a již navštívených vrcholů, označme ji *Y*.

Jednotlivé kroky algoritmu A* jsou následující:

1. Inicializace, neboli určení výchozího a koncového vrcholu, nastavení *d[v]* pro všechny vrcholy z *V* na ∞ a vypočítáme hodnotu heuristické funkce pro každý *v* z *V*, mimo *s*. Pro *s* nastavíme *d[s]* na *0* a *h[s]* také na 0. Pokud $d[v] = \infty$, poté k danému vrcholu není známá žádná cesta. V inicializačním kroku všechny vrcholy z množiny *V* náleží množině *N*.

2. Odebereme z *N* vrchol v_{min} s nejmenším součtem $d[v_{min}] + h[v_{min}]$ a zkontrolujeme ohodnocení všech sousedních vrcholů v_n s odebraným vrcholem v_{min}. Pokud je $d[v_{min}] + \varphi(v_{min}, v_n) < d[v_n]$, poté tedy $d[v_n] = d[v_{min}] + \varphi(v_{min}, v_n)$. Vrchol v_{min} je odebrán z množiny *N* a vložen do množiny *Y*.

3. Zkontrolujeme, zda je množina *N* prázdná. Pokud ne, opakujeme krok 2., pokud ano, pokračujeme krokem 4.

4. Zrekonstruujeme zpětně cestu z koncového do výchozího vrcholu. Začneme v koncovém vrcholu *k,* a hledáme předchůdce *p,* pro kterého platí: $d[p] = d[k] - \varphi(k, p)$. Výsledná hodnota *d[k]* je délka hledané cesty. Krok 4. Je opakován, dokud nedojdeme k vrcholu *p*.

Časová složitost algoritmu závisí na použité heuristické funkci. V nejhorším případě je počet prozkoumaných uzlů exponenciální vzhledem k délce řešení. V optimálním případě je složitost polynomiální. Algoritmus A* z důvodu použití heuristické funkce nemusí najít vždy nejlepší cestu mezi dvěma vrcholy.

3.2 Datové struktury

Datová struktura slouží k uložení informací v paměti počítače. Požadavky na struktury jsou často odlišné a nelze tedy napsat jednu univerzální strukturu, kterou by bylo možné použít k uložení jakýchkoli dat. Z tohoto důvodu bylo vytvořeno více druhů datových struktur a jsou používány tam, kde se zdají být nejvhodnější. Nejčastějšími kritérii při návrhu datové struktury jsou efektivita a maximální úspora paměti.

3.2.1 ADT Graf

Nejvšeobecnější grafovou strukturu přestavuje pseudomigraf, jehož různými omezeními získáme další užší třídy grafových struktur. Pseudomigraf je uspořádaná trojice $G = (V, E, \varphi)$.

Pseudomigraf *G* může být s orientovanými, nebo neorientovanými hranami. Jestliže platí $\varphi(e) = (u, v)$, pro *u, v* z množiny *V*, pak hranu *e* nazýváme neorientovanou hranou. Platí-li pro hranu *e* z množiny *E*, že $\varphi(e) = [u, v]$, pro *u, v* z množiny *V*, pak o hraně *e* mluvíme

jako o orientované hraně. Má-li každý vrchol *v* v pseudomigrafu *G* přiřazenou *p-tici* reálných čísel a každá hrana *h* má přiřazenou *q-tici* reálných čísel, kde $p+q \geq 1$, potom říkáme pseudomigraf *G* je ohodnocený. V opačném případě se *G* nazývá neohodnocený pseudomigraf. Ohodnocený pseudomigraf *G*, pro který platí $p=0$, resp. $q=0$, nazýváme hranově ohodnocený, resp. vrcholově ohodnocený. Jsou-li vrcholy *u, v* z množiny *V* spojeny dvěma či více hranami, jedná se o rovnoběžné hrany. Rovnoběžné hrany se nazývají násobné, pokud jsou všechny orientované, nebo neorientované, a zároveň mají identický výchozí a koncový vrchol. Pokud je vrchol spojený sám se sebou, příslušnou hranu nazýváme smyčka. Mohutností rozumíme počet vrcholů grafu.

Z hlediska datových struktur lze ohodnocení chápat jako libovolná data spojená s příslušným vrcholem, resp. hranou.

Graf je planární právě tehdy, když lze sestrojit jeho diagram v rovině tak, že žádné dvě hrany se nepřekrývají.

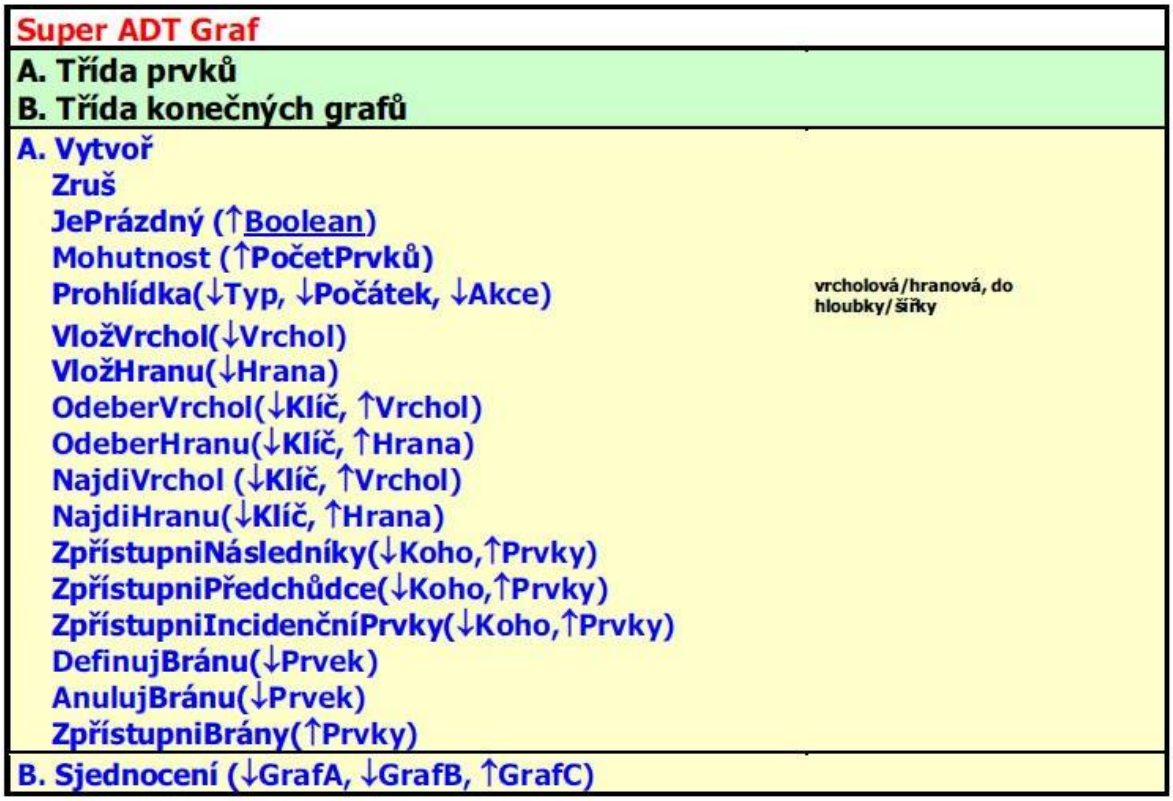
K prezentaci grafu se nejčastěji používají matice incidence, matice sousednosti, nebo spojová prezentace.

V matici incidence řádky, označme je *x*, prezentují jednotlivé vrcholy grafu a sloupce, označme je *y*, jednotlivé hrany. Souřadnice $[x, y] = 1$, pokud daná hrana z vrcholu vychází. $[x, y] = -1$, pokud je daný vrchol cílovým vrcholem dané hrany a $[x, y] = 0$ pokud hrana s vrcholem neinciduje.

Matice sousednosti je čtvercová matice o rozměrech mohutnosti grafu. Řádky, označme je *xx*, i sloupce, označme je *yy*, představují vrcholy grafu. Souřadnice $[xx, yy] = 1$, pokud z vrcholu na daném řádku vychází hrana do vrcholu daného sloupce. V opačném případě $[xx, yy] = 0$. Matice sousednosti pro neorientované grafy je symetrická podle hlavní diagonály.

Spojová prezentace představuje spojový seznam vrcholů. Každý vrchol (seznam) poté obsahuje ukazatele na všechny vrcholy, do kterých daný vrchol vede hranu.

Abstraktní datový typ Graf (odrážející binární relaci v množině) představuje heterogenní bipartití strukturu pracující se dvěma odlišnými třídami prvků – vrcholy a hranami.



Obrázek 1 - Přehled metod Super ADT Graf zdroj:[3]

Klasifikace abstraktních datových struktur vymezených na Super ADT Graf:

1. Vrcholově orientované struktury
   Operace *VložVrchol*, resp. *OdeberVrchol* nejsou implementovány, nebo jejich složitost není menší než $O(|V|)$.

2. Vrcholově dynamické struktury
   Vhodné pro interaktivní nebo dynamickou prácí s grafem zejména vzhledem k vrcholům.

3. Hranově statické struktury
   Operace *VložHranu*, resp. *OdeberHranu* nejsou implementovány, nebo jejich složitost není menší než $O(|E|)$.

4. Hranově dynamické struktury
   Vhodné pro interaktivní nebo dynamickou prácí s grafem zejména vzhledem k hranám.

Ve své práci uvádím teoretický předpoklad práce nad statickým grafem. Struktura grafu se v paměti počítače vytvoří při inicializaci a operace zajišťující změnu struktury jsou volány jen velmi zřídka.

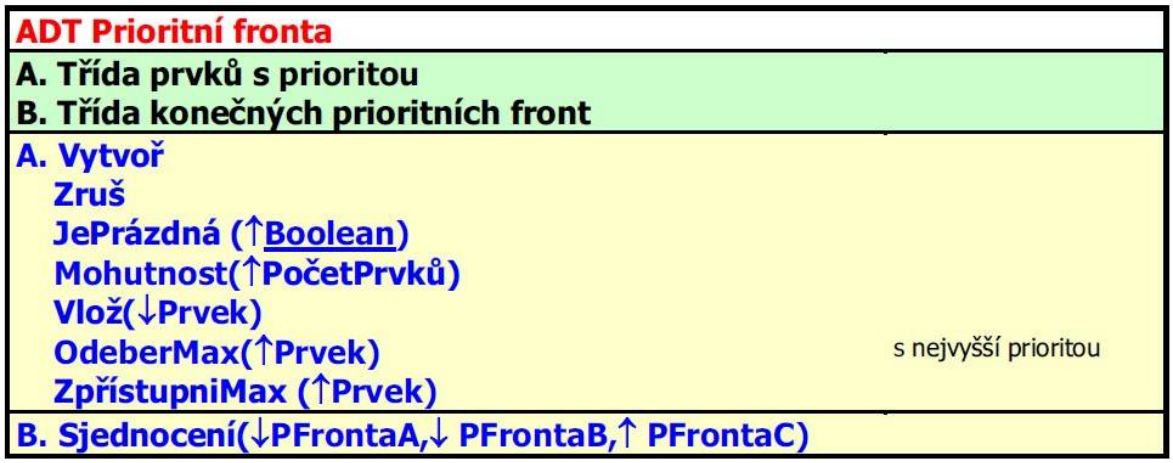
3.2.2 ADT Prioritní Fronta

Prioritní fronta je jednou ze základních datových struktur. Jedná se o množinu s lineárním uspořádáním, přičemž uspořádání je určováno prioritami prvků vzhledem k jejich pořadí odebírání ze struktury.

Speciálními případy prioritní fronty jsou:

1. Zásobník – datová struktura typu LIFO
   Priorita (čas vstupu) přiřazena vloženému prvku implicitně – nejvyšší prioritu má „časově nejmladší“ prvek.

2. Fronta – datová struktura typu FIFO
   Priorita (čas vstupu) přiřazena vloženému prvku implicitně – nejvyšší prioritu má „časově nejstarší“ prvek.



Obrázek 2 - Přehled metod ADT Prioritní fronta zdroj:[3]

Binární halda se chová jako prioritní fronta. V binární haldě platí, že každý potomek daného prvku má nižší prioritu, nebo stejnou prioritu, než rodič. Je-li poslední úroveň stromu nezaplněna zcela, jsou prvky ukládány do této nezaplněné úrovně stromu. Mluvíme-li o levostranné haldě, poté je nezaplněná úroveň stromu zaplňována z leva doprava. Vlastnost být haldou je rekurzivní, což znamená, že jakýkoli podstrom z haldy je také binární haldou. Tato vlastnost zajišťuje, že se binární halda chová jako prioritní fronta.

Můžeme rozlišiovat min-heap a max-heap. U min-heapu považujeme nižší hodnotu klíče za vyšší prioritu, kdežto u max-heapu naopak.

Hlavní výhodou binární haldy je, že asymptotická výpočetní složitost operace *Vlož* je rovna $O(\log_2 n)$. *OdeberMax* má $O(\log_2 n)$.

3.2.3 Možné implementace a efektivita

Stěžejním předpokladem při implementaci grafu je jeho použití. Záleží, zda budeme vytvářet statickou, nebo dynamickou datovou strukturu. S tímto souvisí složitost jednotlivých operací. Metody na reorganizaci struktury grafu budou ve statických

strukturách mnohem méně často volány, než v dynamických strukturách. Z tohoto důvodu můžeme statický a dynamický graf implementovat odlišně.

Dvěma základními přístupy při implementaci grafu jsou:

1. Vrcholově orientovaný přístup.
   Metody vyhledávání orientovány na zpřístupňování vrcholů, zpřístupňování hran je až druhotné.

2. Vrcholově orientovaný přístup.
   Metody vyhledávání orientovány na zpřístupňování hran, zpřístupňování vrcholů je až druhotné.

Pro implementaci se používají hvězdy nebo křížová reprezentace - tzv. řídké matice.

Hvězda představuje kompozitní datovou strukturu skládající se z prvotní datové struktury uchovávající vrcholy, resp. hrany a z druhotných datových struktur, které evidují relace vrcholů, resp. hran z prvotní datové struktury s ostatními vrcholy, resp. hranami z druhotných datových struktur, a ostatních vrcholů, resp. hran z druhotných datových struktur s vrcholy, resp. hranami z prvotní datové struktury.

Hvězdy můžeme implementovat jako dopředné, zpětné a dopředně-zpětné. Prvotní i druhotná datová struktura může být reprezentována polem, kde vrcholy jsou identifikovány pomocí prvních n přirozených čísel, nebo tabulkou, kde vrcholy jsou identifikovány klíčovou položkou, tzv. klíčem.

Hvězdy mohou být realizovány jako následující abstraktní typy:

1. Pole-pole
2. Pole-tabulka
3. Tabulka-pole
4. Tabulka-tabulka

Řídká matice je speciální typ matice, která ukládá do paměti jen nenulové prvky. Realizuje se pomocí dynamického lineárního seznamu, kde vrchol má referenci na položky seznamu, což jsou hrany, které v sobě uchovávají ukazatele na prvky, s nimiž hrana inciduje.

Řídké matice je možné realizovat jako následující abstraktní typy:

1. Pole-tabulka × pole-tabulka
2. Tabulka-tabulka × tabulka-tabulka

Pro dynamické grafy je výhodnější použít tabulkové reprezentace postavené na binárním stromu, jelikož reorganizační metody *Najdi*, *Vlož*, *Odeber* nad uspořádanou tabulkou mají asymptotickou výpočetní složitost $O(\log_2 n)$.

Prioritní frontu je možné implementovat pomocí:

1. Dynamický lineární seznam
2. Pole
3. Binární strom - halda

Haldy vždy vytváří vyvážený strom, což znamená, že aktuální úroveň stromu je vždy zcela zaplněna. Až po zaplnění aktuální úrovně je započato vkládání do následující úrovně. Speciálním případem je levostranná halda, kde aktuální úroveň zaplňujeme zleva doprava.

Prioritní fronta implementována pomocí binárního stromu má asymptotickou výpočetní složitost $O(log_2 n)$ pro metodu *Vlož* a $O(log_2 n)$ pro metodu *OdeberMax*.

Logaritmické asymptotické výpočetní složitosti lze docílit také nad polem. S výhodou se zde využívá metoda půlení intervalu.

3.3 Prostředí pro implementaci

Z mnoha programovacích jazyků jsem, si pro realizaci své práce, zvolil objektově orientovaný programovací jazyk Java, který vyvinula firma Sun Microsystems. Hlavní výhodou jazyku Java je nezávislost na platformě. Java nevytváří přímo strojový kód, ale tzv. mezikód. Programy napsané v jazyku Java poté mohou být spuštěny na libovolném počítači, který má k dispozici interpreter Javy, neboli Java Virtual Machine. Správa paměti v jazyce Java je realizována automaticky nástrojem Garbage collector.

4 Praktická část

Jsou-li určeny teoretické předpoklady práce, je možné přistoupit k samotné praktické části práce. Je uveden teoretický předpoklad práce nad statickým grafem. Tato kapitola je zaměřena na samotné implementace datových struktur, Dijkstrova algoritmu a algoritmu A*.

4.1 Algoritmy

Implementace Dijkstrova algoritmu a algoritmu A* jsou velmi podobné. A* algoritmus navíc oproti Dijkstrově algoritmu využívá heuristickou funkci.

Všechny vrcholy grafu mají při vytvoření instance vrcholu nastaveny hodnotu heuristické funkce $h[v] = 0$ a vzdálenost $d[v]$ je rovna maximální hodnotě datového typu *Integer*, což symbolizuje ∞. Každý vrchol také uchovává ukazatel na předchůdce, abychom mohli po dokončení výpočtu zpětně zrekonstruovat cestu.

4.1.1 Dijkstrův algoritmus

Dijkstrův algoritmus je v aplikaci implementován ve třídě *Dijkstra* a implementuje rozhraní *IDijkstra*. Dijkstrův algoritmus má reference na výchozí a koncový vrchol počítané cesty. Tyto body předáme instanci Dijkstrova algoritmu pomocí konstruktoru.

Dijkstrův algoritmus využívá prioritní frontu, která je implementována pomocí vnitřní datové struktury poskytované jazykem Java – *PriorityQueue*. *PriorityQueue* je prioritní fronta implementována nad binární haldou. Prioritní fronta uchovává všechny vrcholy, pro které platí podmínka $d[v_{min}] + \varphi(v_{min}, v_n) < d[v_n]$, kde $d[v_{min}]$ je vypočítaná cesta od výchozího vrcholu k vrcholu v_{min}, $\varphi(v_{min}, v_n)$ je ohodnocení hrany mezi vrcholy v_{min}, v_n a $d[v_n]$ je vypočítaná cesta od výchozího vrcholu k vrcholu v_n. Vrchol v_n je sousedem vrcholu v_{min}.

Prioritní fronta využívá metodu *compare(vrchola v_1, vrchol v_2)*, která setřídí vrcholy vzestupně dle vypočtené vzdálenosti $d[v_n]$.

Při vytvoření instance algoritmu se nastaví vzdálenost počátečního vrcholu $d[s] = 0$, předchůdce sama na sebe a vloží se do prioritní fronty.

Hlavními metodami Dijkstrova algoritmu jsou:

1. *getNejmensi()*
   Metoda odebere a vrátí vrchol z prioritní fronty.

2. *ohodnotNaslednika(vrchol v_1, vrchol v_2, délka hrany)*
   Metoda porovná vzdálenost vypočítané cesty vrcholu $d[v_2]$, zda není větší než $d[v_1]$ + *délka hrany*. Pokud je podmínka vyhodnocena kladně, je $d[v_2] = d[v_1]$ + *délka hrany* a jako předchůdce vrcholu v_2 je nastaven vrchol v_1. Vrchol v_2 je vložen do prioritní fronty.

V prioritní frontě mohou nastat duplicity. Pokud některý vrchol je sousedící s více vrcholy a více vrcholů postupně splnilo podmínku pro vložení do prioritní fronty, poté později vložený vrchol v_{n+1} s nižším ohodnocením, je v prioritní frontě před hůře ohodnoceným vrcholem v_n a platí $d[v_{n+1}] < d[v_n]$.

Odeberou-li se oba duplicitní vrcholy z prioritní fronty, tak se vždy pracuje s hodnotou lépe ohodnoceného vrcholu v_{n+1}. Hůře ohodnocený vrchol v_2 neovlivní celkový výpočet, jelikož podmínka v metodě *ohodnotNaslednika(vrchol v_1, vrchol v_2, délka hrany)* nemůže být vyhodnocena kladně.

Metody jsou nad grafem volány, dokud metoda *ohodnotNaslednika(vrchol v_1, vrchol v_2, délka hrany)* nevrátí hodnotu *false*, nebo pokud ohodnocený vrchol je zároveň koncovým vrcholem počítané cesty. Způsob ukončení záleží na zvolené variantě Dijkstrova algoritmu.

Ukázka třídy *Dijkstra* s důležitými metodami.

```
public class Dijkstra implements IDijkstra {

IAbstrPrioritniFronta<IBod> body;
IBod start, cil;

public Dijkstra(IBod start, IBod cil) {
 body = new AbstrPrioritniFronta<>(new ComparatorDleVzdalenosti());
 this.start = start;
```

```
 this.cil = cil;
 start.setVzdalenost(0); //nastavení vzdálenosti počátku na 0
 start.setPredchudce(start); //nastavení předchůdce sama na sebe
 body.vlozPrvek(start); //vložení startu do haldy
}

@Override
public IBod getNejmesi() {
 if (!body.jePrazdna()) {
  return body.odeberPrvek(); //odebere první prvek z haldy
 }
 return null;
}

@Override
public boolean ohodnotNaslednika(IBod start, IBod cil, double vzdalenost)
{
if (start != null) {
 if (cil.getVzdalenost() > start.getVzdalenost() + vzdalenost) {
  cil.setVzdalenost(start.getVzdalenost() + vzdalenost);
  cil.setPredchudce(start);
  body.vlozPrvek(cil);
  return true;
 }
 if (!body.jePrazdna()) {
  return true;
 }
}
return false;
}
```

4.1.2 Algoritmus A*

Algoritmus A* je v aplikaci implementován ve třídě *Astar* a implementuje rozhraní *IAstar*. Implementace algoritmu A* je téměř totožná s implementací Dijkstrova algoritmu. Algoritmus A* využívá navíc heuristickou funkci oproti Dijkstrovu algoritmu. Algoritmus A* má také reference na výchozí a koncový vrchol počítané cesty. Tyto vrcholy předáme instanci algoritmu opět pomocí konstruktoru.

A* algoritmus stejně jako Dijkstrův algoritmus využívá totožnou prioritní frontu, ale odlišnou metodu *compare(vrchol v_1, vrchol v_2)*, která setřídí vrcholy vzestupně dle součtu $d[v_n] + h[v_n]$.

Při vytvoření instance algoritmu se nastaví vzdálenost počátečního vrcholu $d[s] = 0$, hodnota heuristické funkce $h[s] = 0$, předchůdce sama na sebe a vloží se do prioritní fronty.

Hlavními metodami A* algoritmu jsou:

1. *getNejmensi()*
   Metoda odebere a vrátí vrchol z prioritní fronty.

2. *ohodnotNaslednika(vrchol v_1, vrchol v_2, délka hrany)*
   Metoda pracuje identicky jako stejnojmenná metoda v Dijkstrově algoritmu.

3. *vypoctiHeuristiku(Vrchol)*

   Metoda vypočítá přímou vzdálenost vzdušnou čarou od aktuálního vrcholu ke koncovému vrcholu.

Metody jsou nad grafem volány, dokud metoda *ohodnotNaslednika(vrchol v_1, vrchol v_2, délka hrany)* nevrátí hodnotu *false*, nebo pokud ohodnocený vrchol je zároveň koncovým vrcholem počítané cesty. Způsob ukončení záleží na zvolené variantě A* algoritmu.

Ukázka třídy *Astar* s důležitými metodami.

```
public class Astar implements IAstar {

IAbstrPrioritniFronta<IBod> body;
IBod start, cil;

public Astar(IBod start, IBod cil) {
 body = new AbstrPrioritniFronta<>(new ComparatorDleVzdalenostiHeuris());
 this.start = start;
 this.cil = cil;
 start.setVzdalenost(0); //nastavení vzdálenosti počátku na 0
 start.setHeuristika(0); //nastavení heuristiky počátku na 0
 start.setPredchudce(start);
 body.vlozPrvek(start);
}

@Override
public IBod getNejmesi()
 if (!body.jePrazdna()) {
  return body.odeberPrvek();
 }
 return null;
}

@Override
public boolean ohodnotNaslednika(IBod start, IBod cil, double vzdalenost)
{
if (start != null) {
 if (cil.getVzdalenost() > start.getVzdalenost() + vzdalenost) {
  cil.setVzdalenost(start.getVzdalenost() + vzdálenost
  cil.setPredchudce(start);
  body.vlozPrvek(cil);
  return true;
 }
 if (!body.jePrazdna()) {
  return true;
 }
}
return false;
}

@Override
public void vypoctiHeuristiku(IBod bod){
 Point s = bod.getPozice();
 Point k = this.cil.getPozice();
 bod.setHeuristika(Math.sqrt(((s.x - k.x) * (s.x - k.x)) + ((s.y - k.y) *
(s.y - k.y))));
}
}
```

4.1.3 Vytipování stěžejních operací

Stěžejními metodami obou algoritmů je metoda pro zpřístupnění vrcholu a zpřístupnění všech sousedů zpřístupněného vrcholu. Často volané jsou také metody pro vložení a odebrání z prioritní fronty

Ukázky implementací algoritmů. Hodnota nextStep je inicializována na true.

```
while (nextStep) {
 bod = dijkstra.getNejmesi();
 for (Iterator<Struktury.AbstrGraf.Hrana> it =
 graf.prohlidkaNasledniku(bod); it.hasNext();) {
  nextStep = dijkstra.ohodnotNaslednika(bod,
  (IBod) it.next().getCil().getData(),
  ((IHrana) it.next().getData()).getVzdalenost());
 }
}
```

```
while (nextStep) {
 bod = astar.getNejmesi();
 for (Iterator<Struktury.AbstrGraf.Hrana> it =
 graf.prohlidkaNasledniku(bod); it.hasNext();) {
  astar.vypoctiHeuristiku((IBod) it.next().getCil().getData());
  nextStep = astar.ohodnotNaslednika(bod,
  (IBod) it.next().getCil().getData(),
  ((IHrana) it.next().getData()).getVzdalenost());
 }
}
```

4.2 Graf

V teoretickém úvodu je uveden předpoklad práce nad statickým grafem. Pro implementaci grafu byla zvolena datová struktura typu tabulka-pole. Datová struktura graf uchovává ukazatele na výchozí a koncový vrchol počítané cesty.

Při vytvoření instance grafu jsou ukazatele pro výchozí a koncový vrchol počítané cesty nastaveny na *null* a je vytvořena prvotní datová struktura, která uchovává vrcholy grafu.

Prvotní datová struktura je implementována pomocí vnitřní datové struktury jazyku Java – *ArrayList*, která využívá datový typ *Uzel*. *Uzel* reprezentuje vrchol grafu. Prvotní datová struktura je utříděná dle klíče. Vyhledávání v prvotní datové struktuře je implementováno pomocí klíče nebo indexu v poli.

Každý *Uzel* vlastní referenci na data spojená s daným vrcholem a na druhotnou datovou strukturu implementovanou nad vnitřní datovou strukturou jazyku Java – *ArrayList*. Druhotná datová struktura využívá datový typ *Hrana*. *Hrana* vlastní referenci na data spojená s danou hranou a na cílový vrchol incidence.

Nejčastěji volanou metodou je *prohlídkaNasledniku(Uzel v)*. Jedná se o iterátor procházející všechny sousedy daného vrcholu. Je volána v každém kroku Dijkstrova algoritmu a A* algoritmu, pro ohodnocení sousedů. Asymptotická časová složitost této metody je *O(n)*, jelikož se prochází všechny hrany v druhotné datové struktuře.

Často volaná je také metoda *najdiUzel(Uzel v)*. Tato metoda je volána v metodě *prohlídkaNasledniku(Uzel v)* pro nalezení vrcholu *v* v prvotní datové struktuře podle klíče.

4.3 Aplikace

Práce s programem je velmi jednoduchá a intuitivní. Zadávání dat je prováděno pomocí myši a klávesových modifikátorů.

Graf je jednoduše graficky vykreslen. Grafické výstupy je možné jednoduše uložit do *JPEG* souborů. Vypočítané matice incidencí pro Dijkstrův algoritmus a A* algoritmus jsou uloženy do *TXT* souboru.

Dijkstrův algoritmus a algoritmus A* byly napsány v několika variantách. Algoritmy mohou projít všechny vrcholy grafu, nebo pokud dojdou při výpočtu ke koncovému vrcholu, tak se ukončí. Dalšími variantami jsou výpočet a zobrazení cesty okamžitě, sledování postupu výpočtu pomocí časovače a postup po jednotlivých krocích výpočtu manuálně.

5 Závěr

Hlavním cílem práce je implementovat Dijkstrův algoritmus a algoritmus A*, čehož je docíleno. Datové struktury jsou implementovány pro práci se statickým grafem. Metody grafu používané pro výpočet algoritmů mají optimalizovanou asymptotickou výpočetní složitost.

Funkčnost algoritmů je ověřena dvěma způsoby. Na malém rozsahu dat, který je graficky zobrazen. Druhým způsobem je vygenerování grafu z většího rozsahu dat. Z tohoto grafu je vygenerována matice vzdáleností pro oba algoritmy. Správnost výpočtu je určena symetričností matice vzdáleností podle hlavní diagonály.

6 Zdroje

[1] CENEK, Petr, KLIMA, Valent, JANÁČEK, Jaroslav. *Optimalizace dopravních a spojových procesů.* Žilina : Vysoká škola dopravy a spojov, 1994. 343 s. ISBN 80-7100-197-X.

[2] VOLEK, Josef. *Operační výzkum I.* 2. vyd. Pardubice : Univerzita Pardubice, 2008. 111 s. ISBN 978-80-7395-073-6.

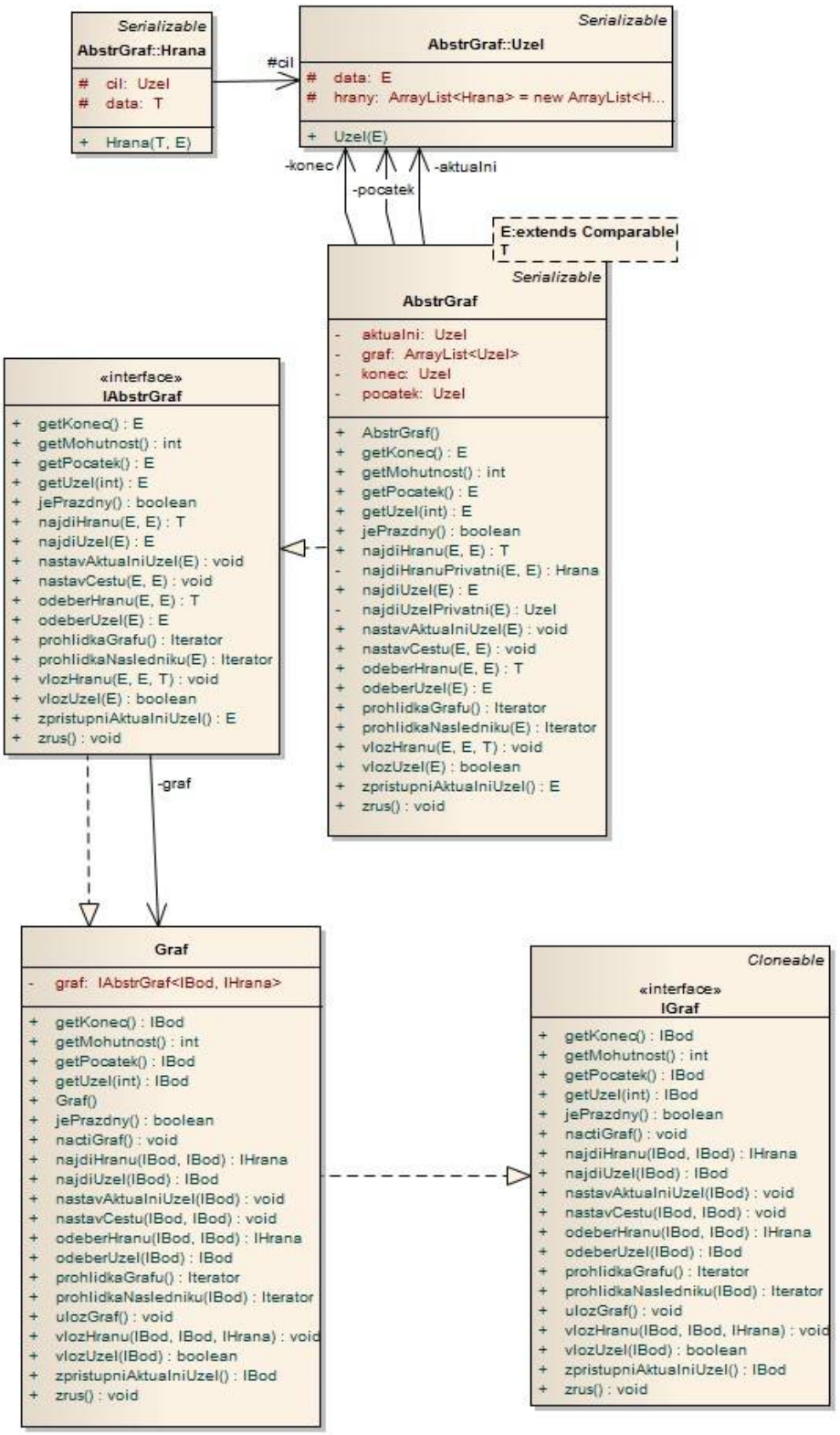
[3] KAVIČKA, Antonín. *Datové struktury.* Elektronické sylaby přednášek předmětu Datové struktury. 2011-2012.

[4] MIČKA, Pavel. *Algoritmy.net.* [online]. 2008 - 2013 [cit. 2013-04-14]. Dostupné z: <http://www.algoritmy.net>.

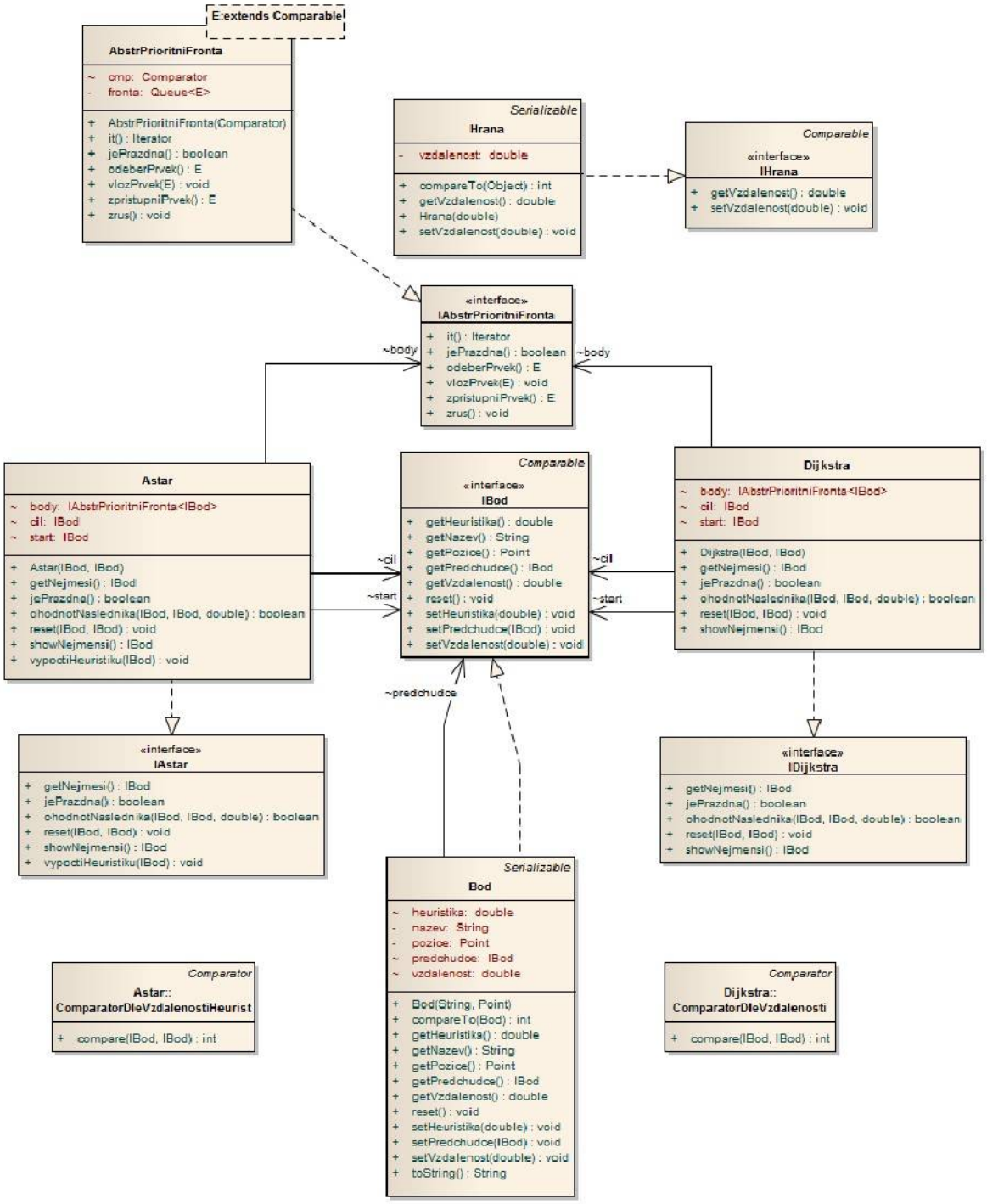
[5] ORACLE. *Java SE Documentation: Class PriorityQueue<E>*. [online]. 1993 - 2013 [cit. 2013-04-14]. Dostupné z: <http://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html>.

[6] ORACLE. *Java SE Documentation: Class ArrayList<E>*. [online]. 1993 - 2013 [cit. 2013-04-14]. Dostupné z: <http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html>.

Příloha A – UML diagram aplikace

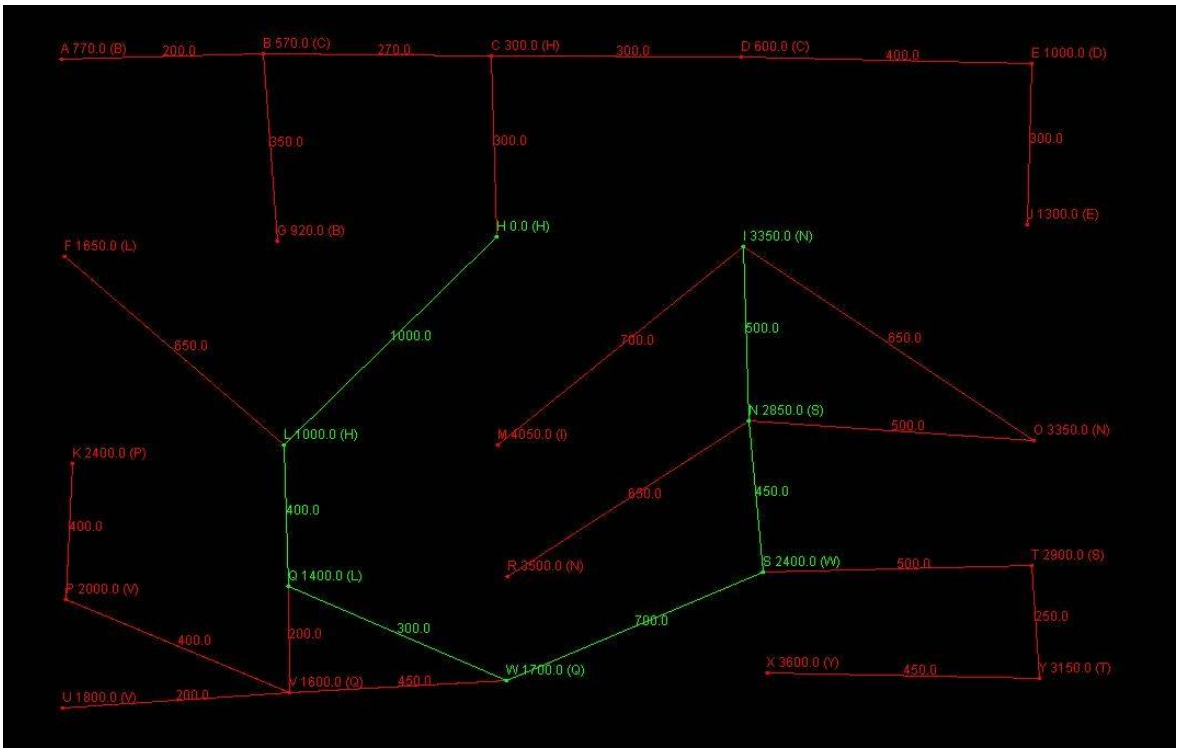


Obrázek 3 - UML Datových struktur zdroj:vlastní

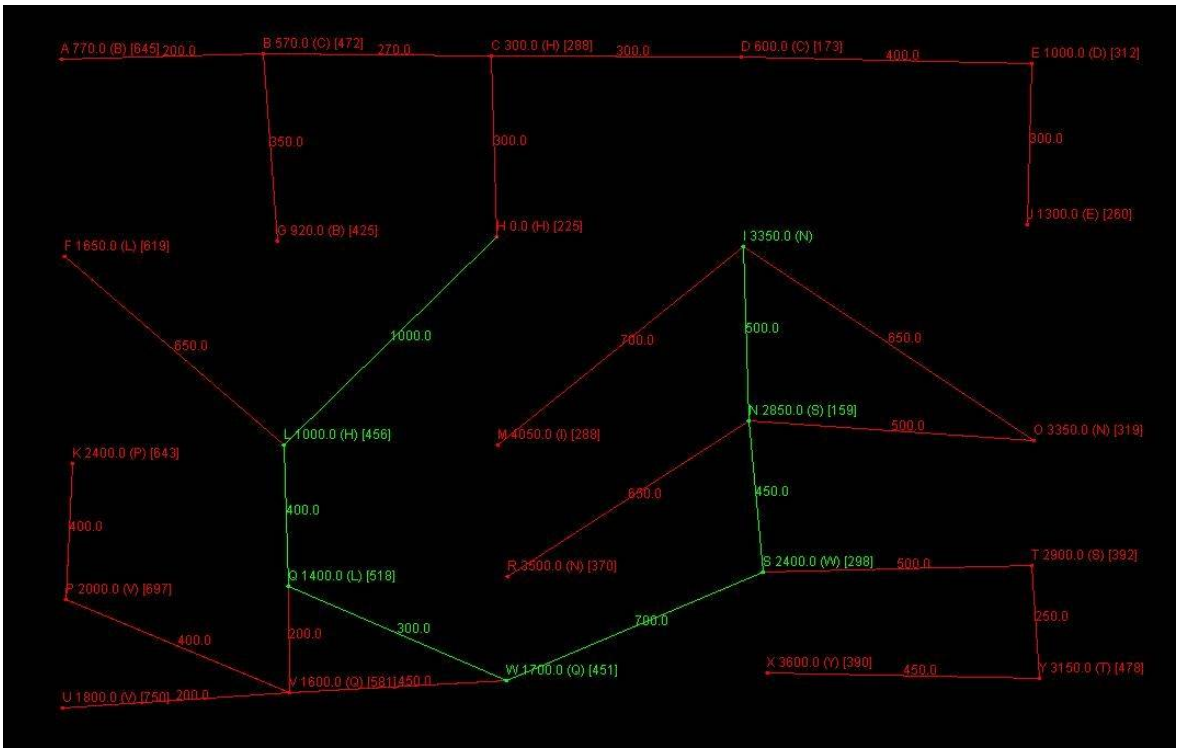


Obrázek 4 – UML Algoritmů zdroj:vlastní

Příloha B – Ilustrace výpočtu



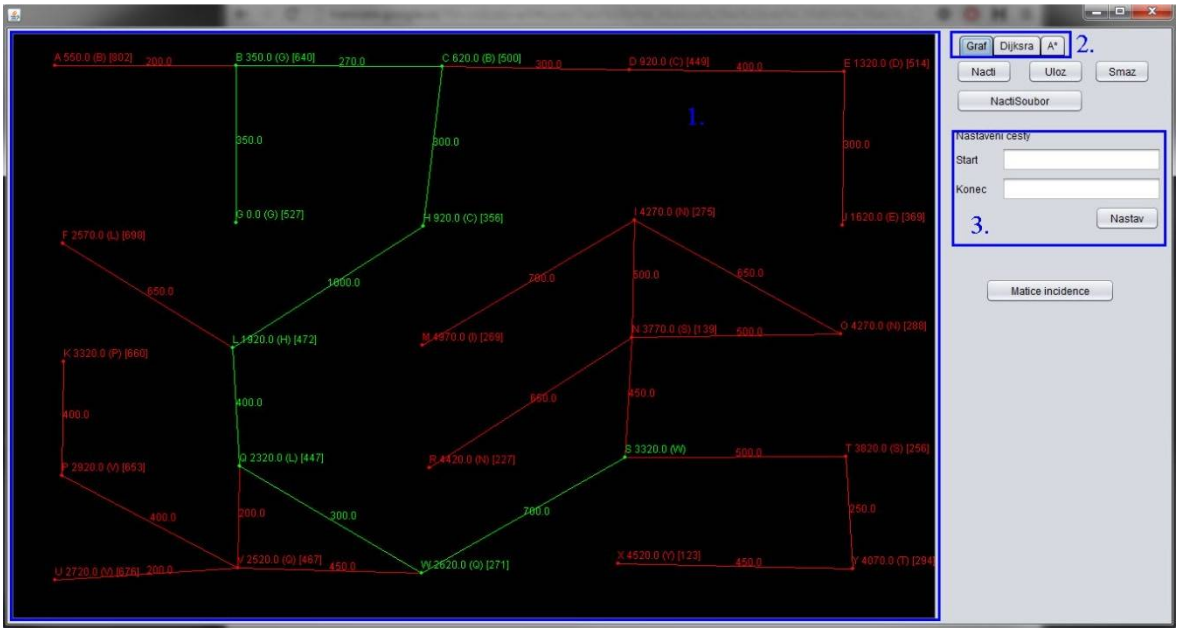
Obrázek 5 – Ilustrace výpočtu pomocí Dijkstrova algoritmu zdroj:vlastní



Obrázek 6 - Ilustrace výpočtu pomocí algoritmu A* zdroj:vlastní

Příloha C – Ovládání aplikace

Data je možné zadávat pomocí myši. Klikem levým tlačítkem myši na černý zobrazovací panel zadáte vrchol grafu. Po kliknutí se zobrazí input dialog pro zadání názvu vrcholu. Již vytvořený vrchol odstraníte pomocí levého tlačítka myši s modifikátorem *ALT*. Odstraněním vrcholu se odstraní i veškeré hrany incidující s tímto vrcholem.



Obrázek 7 - Hlavní okno aplikace zdroj:vlastní

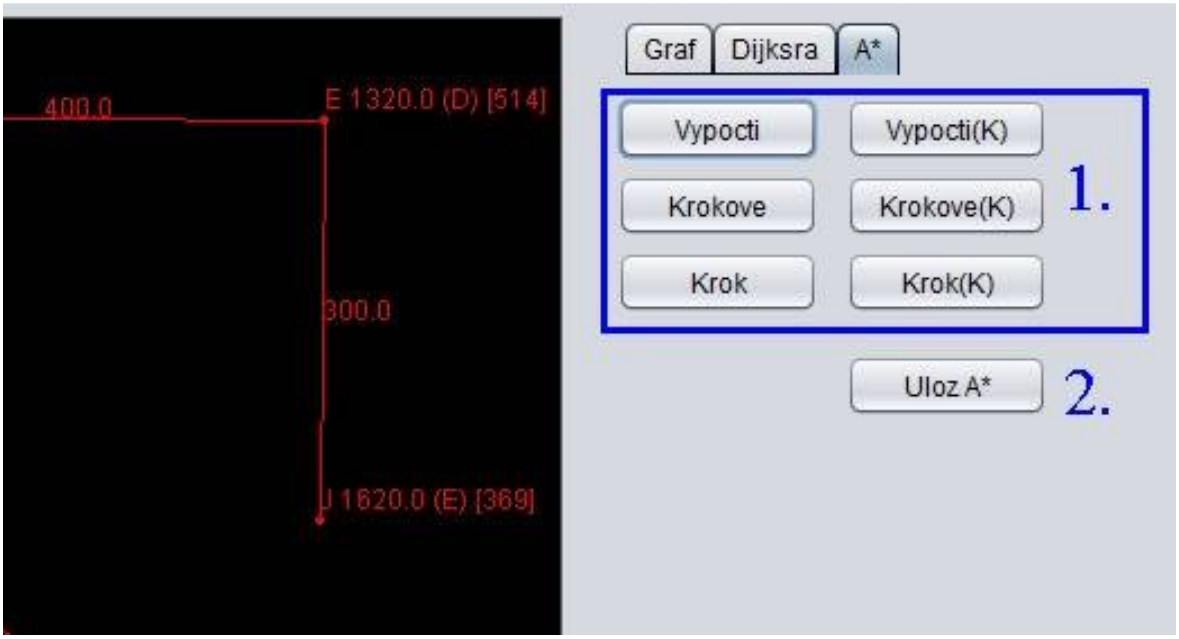
1. Zobrazovací panel
2. Záložky
3. Nastavení cesty

Táhnutím pravým tlačítkem myši z jednoho vrcholu do druhého zadáte hranu. Po uvolnění tlačítka se objeví input dialog na zadání hodnoty pro ohodnocení hrany. Tažením pravým tlačítkem mezi vrcholy s existující hranou s modifikátorem *ALT* hranu mezi danými vrcholy odstraníte.

Tažením pravým tlačítkem mezi dvěma vrcholy s modifikátorem *CTRL* nastavíte výchozí a koncový vrchol počítané cesty. Výchozí a koncový vrchol lze také zadat do text editu a nastavit tlačítkem *„Nastav“*.

Na záložce *Graf* dále najdete tlačítka pro uložení a nahrání instance grafu do souboru a resetování dané instance. Lze zde také načíst testovací soubor o velikosti 813 vrcholů. Nachází se zde také tlačítko pro výpočet matice vzdáleností pro oba algoritmy, které jsou uloženy do textových souborů.

Záložky *Dijkstra* a *A** umožňují počítat zadanou cestu různými variantami algoritmů a je možnost uložit výsledek zobrazeného výpočtu jako *JPEG* soubor. Abychom se k těmto záložkám dostali, musíme nejprve zadat výchozí a koncový vrchol počítané cesty.



Obrázek 8 - Záložka A* zdroj:vlastní

1. Varianty algoritmu
2. Uložení ilustrace výpočtu

Graf je zobrazen na černém panelu jako červená síť. Hodnota výpočtu v daném vrcholu je zobrazena za vrcholem. V kulatých závorkách je uveden předchůdce, z kterého se do daného bodu dostaneme. U algoritmu A* je navíc v hranatých závorkách zobrazena hodnota heuristické funkce.

Je-li mohutnost grafu větší než sto, tak se graf již graficky nezobrazuje. Pokud počítáme algoritmy krokově, tak modře je zobrazen aktuální vrchol, a zeleně zpracovávané okolí. Výpočet je možné uložit do *JPEG* souboru.

Příloha D – Obsah přiloženého CD

1. Bakálářská práce ve formátu PDF.
2. Projekt z vývojového prostředí NetBeans se zdrojovými kódy.
3. Složka s aplikací.