VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

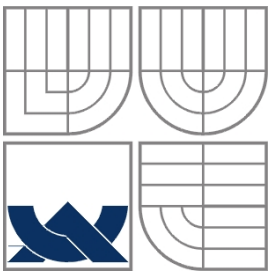
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

HLEDÁNÍ NEJKRATŠÍCH CEST GRAFEM

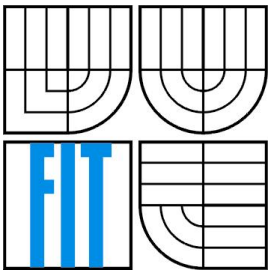
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE PETR JÁGR
AUTHOR

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

HLEDÁNÍ NEJKRATŠÍCH CEST GRAFEM

FINDING THE SHORTEST PATHS IN A GRAPH

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE
AUTHOR

PETR JÁGR

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JIŘÍ JAROŠ

BRNO 2007

Abstrakt

Předmětem této bakalářské práce je hledání, porovnání, úprava a implementace vhodných grafových algoritmů vedoucích k nalezení všech nejkratších cest mezi všemi dvojicemi vrcholů v neorientovaných grafech. Pro tento účel jsou využity modifikace již existujících algoritmů a jejich fragmentů tak, aby bylo docíleno co možná nejnižší časové náročnosti výpočtu. Porovnáme si Dijkstrův, Floyd-Warshallův a Bellman-Fordův algoritmus.

Klíčová slova

algoritmus, asymptotické vyjádření složitosti, Bellman-Fordův algoritmus, cesta, Dijkstrův algoritmus, dynamické programování, Floyd-Warshallův algoritmus, genetické algoritmy, heuristické algoritmy, hladové algoritmy, Java, nejkratší cesta, neorientovaný graf, ohodnocený graf, Omega, Omikron, orientovaný graf, paralelní algoritmy, plánování trasy, pravidelný graf, rekurzivní algoritmy, rozděl a panuj, sled, smyčka, souvislý graf, stupeň vrcholu, tah, teorie grafů, Theta

Abstract

The aim of this thesis is finding, comparing and implementation of algorithms for finding the shortest paths between each of pairs of nodes in a graph. For this task I use modifications of existing algorithms to achive the lowest time consumption of the computation. Modifications are established on Dijkstra's and Floyd-Warshall's algorithm. We also familiarize with Bellman-Ford algorithm.

Keywords

algorithm, asymptotic notation, Bellman-Ford algorithm, complete graph, degree of a vertex, Dijkstra's algorithm, directed graph, divide and conquer algorithms, dynamic programming, Floyd–Warshall algorithm, genetic algorithms, graph teory, greedy algorithms, heuristic algorithms, Java, loop, move, Omega, Omicron, paralel algorithms, path, path planning, recursive algorithms, regular graph, sequence, the shortest path, Theta, undirected graph, weighted graph

Citace

Petr Jágr: Hledání nejkratších cest grafem, bakalářská práce, Brno, FIT VUT v Brně, 2007

Hledání nejkratších cest grafem

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Jiřího Jaroše
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jméno Příjmení
Datum

Poděkování

Děkuji vedoucímu své práce, panu Jiřímu Jarošovi, za cenné rady, připomínky a materiály, které mi při psaní práce poskytl.

Obsah

1 Úvod

Jak již sám název této bakalářské práce napovídá, budu se dále v textu zabývat oblastí matematiky věnující se teorii grafů. Grafem tedy není myšleno grafické vyjádření funkce. Předmět v bakalářském studijním programu Informační technologie na FIT VUT Brno věnující se této oblasti se jmenuje Diskrétní matematika a z jeho studijních materiálů [1] budu čerpat většinu teoretického základu.

Nejkratší cesty v grafu najdou využití v mnoha oblastech. Namátkou jmenujme např. optimální řízení provozu v počítačové síti, v geografických informačních systémech (GIS) pro plánování trasy nebo také pro optimální rozvržení součástek elektrického obvodu na plošném spoji. Hlavní přínos tedy směřuje do oblasti logistiky a dalších s ní příbuzných oblastí.

Úkolem práce je vytvořit program pro nalezení všech nejkratších cest v grafu mezi všemi vrcholy s co možná nejnižší časovou náročností. Pokusím se poskytnout ucelený přehled nad touto problematikou a prezentovat několik vhodných algoritmů pro její řešení.

V následující kapitole si blíže definujeme pojmy související s grafy. Jaké druhy grafů známe a podle jakých kritérií je dělíme. Blíže rozvedeme jejich vlastnosti a na několika příkladech si ilustrujeme rozdíly.

Třetí kapitola bude v úvodu pojednávat o vzniku termínu algoritmus a dále o složitosti a klasifikaci algoritmů. Podrobně se budeme zabývat Dijkstrovým algoritmem, u kterého si rozebereme jeho časovou a paměťovou složitost. Aplikujeme tento algoritmus na jednoduchý hranově ohodnocený graf a budeme si na ilustracích vysvětlovat průběh. Poté rozvedeme úvahu o úpravách algoritmu potřebných pro splnění zadání. Dalším zajímavým algoritmem, který si přiblížíme, je Floyd-Warshallův. Opět se ho pokusíme vhodným způsobem modifikovat, aby vyhovoval požadavkům zadání a nalezl všechny nejkratší cesty. Na konci kapitoly prodiskutujeme dosažená řešení.

Čtvrtá kapitola bude na svém začátku pojednávat o vhodném jazyce pro implementaci a o důvodech, které mě vedly k výběru jazyka Java. Po letmém seznámení se s jazykem Java budou podrobněji rozebrány třídy programu a budou navrženy možné optimalizace běhu.

Závěrečná pátá část shrne dosažené výsledky a pravděpodobný budoucí rozvoj v této oblasti. Budou zde diskutovány problémy, se kterými jsem se potýkal při implementaci, a jakým způsobem se podařilo dosáhnout vytyčených cílů.

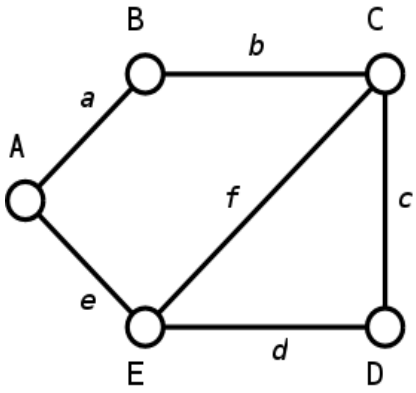
2 Grafy

Grafy najdou uplatnění v mnoha oblastech, od fyziky, elektrotechniky a chemie až třeba po ekonomii a lingvistiku. Nejprve se seznámíme s pojmy s nimi souvisejícími, jejichž pochopení je nutné pro další rozvoj myšlenek zabývajících se předmětem práce. Většina potřebných definic je převzata z [1] a z [3].

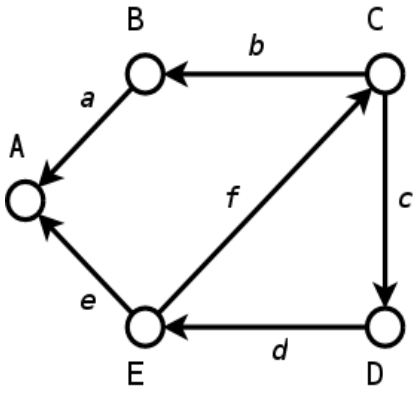
2.1 Základní pojmy

Neorientovaný graf G se skládá z množiny V vrcholů (uzlů) a množiny H hran tak, že každá hrana $h \in H$ je přiřazena neuspořádané dvojici (tj. dvouprvkové množině) vrcholů $u, v \in V$. Existuje-li jediná hrana $h \in H$ přiřazená dvojici vrcholů $u, v \in V$, píšeme $h \equiv \{u, v\}$. Obecně může být jedné dvojici vrcholů přiřazeno více hran, tyto hrany se nazývají násobné.

Podobně, *orientovaný graf G* se skládá z množiny V vrcholů a množiny H hran tak, že každé hraně $h \in H$ je přiřazena uspořádaná dvojice $(u, v) \in V \times V$ vrcholů $u, v \in V$. Existuje-li jediná hrana $h \in H$ přiřazená dvojici (u, v) vrcholů $u, v \in V$, píšeme $h \equiv (u, v)$. Hranu, u níž jsou prvky uspořádané nebo neuspořádané dvojice totožné, označujeme *smyčkou*.



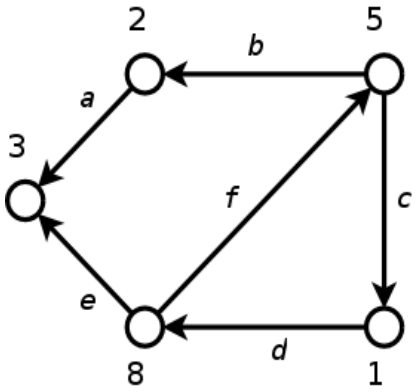
Obr. 1 – Neorientovaný graf



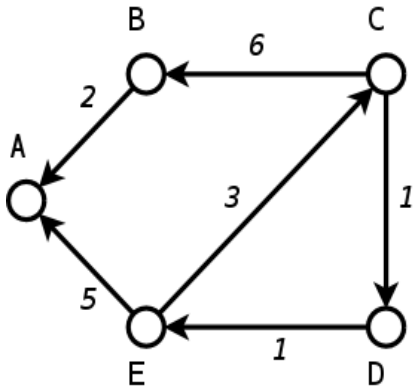
Obr. 2 – Orientovaný graf

Množiny vrcholů grafů na obrázcích 1 a 2 tedy obsahují $V = \{A, B, C, D, E\}$ a množiny hran $H = \{a, b, c, d, e, f\}$. U neorientovaného grafu existují hrany např. $a \equiv \{A, B\}$, $b \equiv \{B, C\}$ nebo $f \equiv \{E, C\}$. U orientovaného grafu existují hrany např. $a \equiv (B, A)$, $b \equiv (C, B)$ nebo $f \equiv (E, C)$.

Buď G graf s množinou vrcholů V a množinou hran H. Nechť $f: V \to R$ a $g: H \to R$ jsou zobrazení. Pak f se nazývá *vrcholovým ohodnocením* grafu G, g se nazývá *hranovým ohodnocením* grafu G. Dvojice (G, f), resp. (G, g) se nazývá *vrcholově*, resp. *hranově ohodnocený graf*.



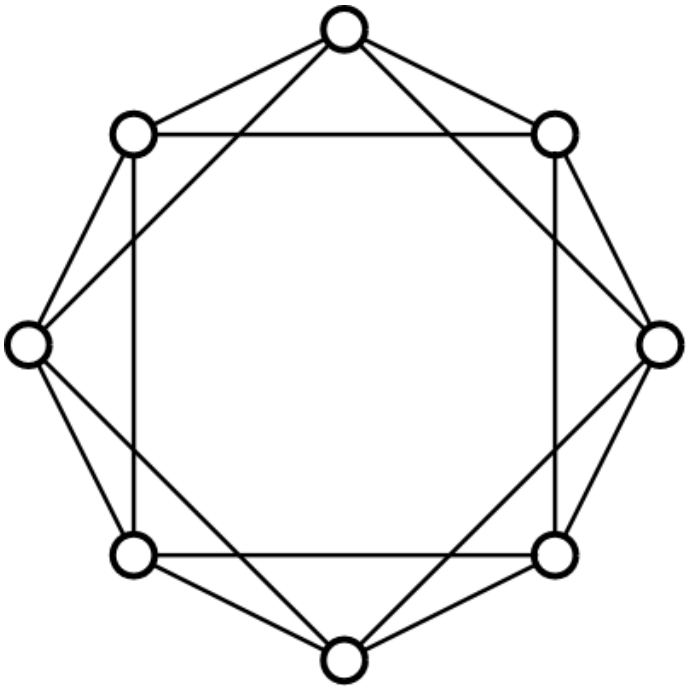
Obr. 3 – Vrcholově ohodnocený graf



Obr. 4 – Hranově ohodnocený graf

Hranová ohodnocení se dají využít např. pro vyjádření délky hrany nebo její kapacity. Vrcholové ohodnocení může vyjadřovat např. náročnost úkolu spojeného s průchodem vrcholu.

Vrcholy v_1 a v_2 nazýváme *sousedy*, právě když existuje hrana, která je spojuje. *Stupněm vrcholu* v neorientovaném grafu nazýváme počet cest vedoucích do/z vrcholu. Pokud všechny vrcholy grafu jsou téhož stupně, nazýváme graf *pravidelným grafem (regulárním)*.



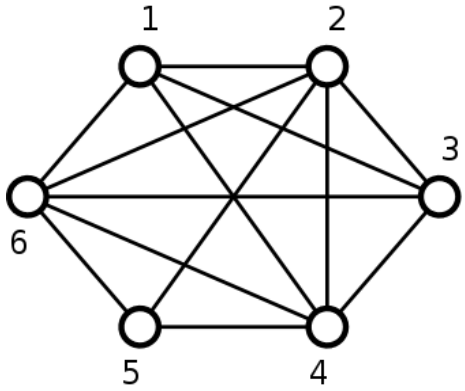
Obr. 5 – Neorientovaný pravidelný graf

Jak si můžeme všimnout, graf na obrázku 5 je pravidelný a všechny jeho vrcholy jsou stejného, čtvrtého, stupně.

Nechť G je graf. *Sledem délky n* v grafu G nazýváme posloupnost vrcholů v_i a hran h_j grafu G tvaru

$$v_0\, h_1\, v_1\, h_2\, v_3\, \ldots\, v_{n-1}\, h_n\, v_n,$$

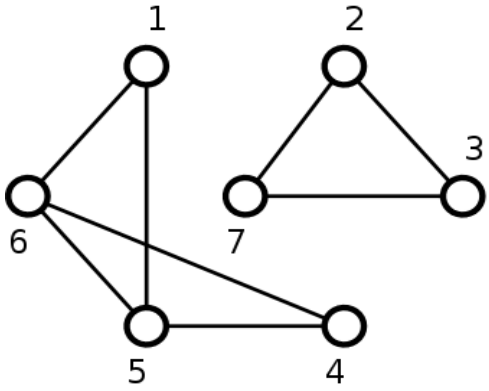
kde hraně h_i je přiřazena dvojice vrcholů $\{v_{i-1}, v_i\}$ (resp. (u, v) v orientovaném grafu). *Tahem* v grafu G nazýváme sled, v němž se každá hrana grafu vyskytuje nejvýše jednou. *Cestou* v grafu G nazýváme sled, v němž se každý vrchol grafu vyskytuje nejvýše jednou. Sled se nazývá *uzavřený*, jestliže $v_0 = v_n$ (počáteční vrchol je stejný jako koncový). Uzavřený sled se nazývá *uzavřeným tahem*, vyskytuje-li se v něm každá hrana grafu nejvýše jednou. Uzavřený sled se nazývá *uzavřenou cestou*, vyskytuje-li se v něm každý vrchol grafu, s výjimkou počátečního vrcholu, který je současně i koncový, nejvýše jednou. Na obrázku 6 si tyto pojmy blíže vysvětlíme.



Obr. 6 – Vrcholově ohodnocený graf

- 6{6, 3}3{3, 6}6{6, 2}2{2, 4}4 je sled délky 4, ale není to tah ani cesta (např. vrchol 6 a hrana {6, 3} se vyskytují dvakrát) .
- 6{6, 2}2{2, 4}4{4, 6}6{6, 3}3 je tah délky 4, ale ne cesta (vrchol 6 je zde dvakrát).
- 2{2, 4}4{4, 6}6{6, 3}3 je cesta délky 3 (a zároveň také tah).

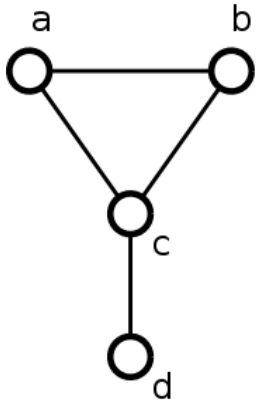
Graf G označíme za *cyklický*, když obsahuje uzavřenou cestu. Graf G se nazývá *souvislý*, jestliže mezi libovolnými dvěma vrcholy existuje sled. Pokud sled neexistuje je graf *nesouvislý*.



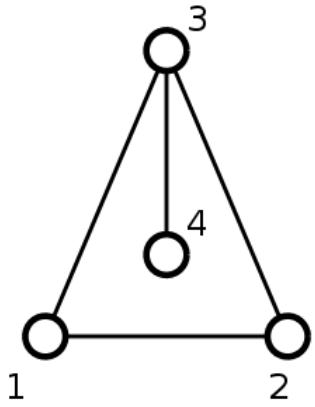
Obr. 7 – Vrcholově ohodnocený nesouvislý graf

Množina vrcholů grafu na obrázku 7 je $V = \{1, 2, 3, 4, 5, 6, 7\}$ a můžeme si povšimnout, že některé vrcholy opravdu nelze sledem propojit.

Grafy $G_1 = (V_1, H_1)$ a $G_2 = (V_2, H_2)$ se nazývají *izomorfní*, jestliže existují bijekce f: $V_1 \to V_2$ a g: $H_1 \to H_2$ takové, že libovolné hraně $h \in H_1$ jsou přiřazeny vrcholy $x, y \in V_1 \Leftrightarrow$ hraně $g(h) \in H_2$ jsou přiřazeny vrcholy $f(x), f(y) \in V_2$.



Obr. 8 – Graf G_1



Obr. 9 – Graf G_2

Příslušné bijekce f a g lze definovat takto:

$$f(\mathrm{a}) = 1, f(\mathrm{b}) = 2, f(\mathrm{c}) = 3, f(\mathrm{d}) = 4;$$

$$g(\{\mathrm{a}, \mathrm{b}\}) = \{1, 2\}, g(\{\mathrm{b}, \mathrm{c}\}) = \{2, 3\}, g(\{\mathrm{c}, \mathrm{d}\}) = \{3, 4\}, g(\{\mathrm{a}, \mathrm{c}\}) = \{1, 3\}.$$

3 Algoritmy

S algoritmy se v každodenním životě setkáváme velice často. Příkladem může být např. recept na vaření, podle kterého se připravuje pokrm. Algoritmem je tedy označován přesný návod nebo postup, kterým lze vyřešit určitý problém. V programování se občas zaměňuje pojem algoritmus s implementací algoritmu v konkrétním programovacím jazyce, což není správné. Algoritmus z jeho implementace často pochopit lze, ale je tu riziko neuvědomění si všech jeho vlastností.

Slovo algoritmus pochází ze jména díla významného matematika žijícího v oblasti dnešního Íránu (dřívější Persie) v období první poloviny 9. století. Jmenoval se **Abú Abd Alláh Muhammad Ibn Músá al-Chórezmí Abú Dža'far** (doslovný překlad „Otec Abdulláha, Mohameda, syn Mojžíšův, pocházející z města Chórézm“).

Obr. 1 - Al-Chórezmí

Mezi jeho počiny lze zařadit vytvoření systému arabských číslic a metody pro řešení lineárních a kvadratických rovnic. Překlad slova vyjadřující postup výpočtu do latiny zněl *algorismus*, a původně vyjadřoval „*provádění aritmetiky pomocí arabských číslic*“. Postupem času se název zkomolil na tvar *algorithmus* a nakonec se v některých jazycích zaměnilo *th* s *t*. V Českém jazyce používáme slovo *algoritmus* zhruba od začátku 20. století.

3.1 Vlastnosti algoritmu

Každý algoritmus má jistý vstup, který není podmínkou, a výstup. Vstupem se myslí hodnoty různých veličin zadávané před započetím provádění nebo v průběhu jeho činnosti. Avšak existují i algoritmy bez vstupu. Např. při výpočtu prvočísel není potřeba žádný vstup. Výstup není nutný, ale algoritmus bez výstupních veličin pozbývá významu. Problém pomocí něho vyřešíme, ale řešení nám není zpřístupněno. Algoritmus se skládá z elementárních (atomárních), dále nedělitelných, částí, kterým říkáme *kroky*. Provedení jednoho kroku se nazývá *iterace*. Algoritmus s konečným počtem kroků se nazývá *konečný*. Naopak *nekonečný algoritmus* se skládá z nekonečného počtu kroků. Sem můžeme zařadit již vzpomínaný výpočet prvočísel. O algoritmu řekneme že je *deterministický*, když v každém

stavu, v jakém se může nacházet, existuje jeho přesně popsané jednoznačné chování. V praxi často hledáme algoritmy, které jsou jistým způsobem kvalitní a vyhovují našim požadavkům. Kritérium pro hledání může být např. počet kroků algoritmu, aby výpočet skončil v rozumném čase, nebo jednoduchost aby se snížilo riziko chyby v implementaci. Další významné kritérium jsou požadavky na paměťový prostor pro algoritmus a data, nad kterými pracuje. Běžný jazyk často nezachytí všechny aspekty algoritmu a proto je potřeba při jeho slovní formulaci věnovat zvýšenou pozornost úplné a jednoznačné definici.

3.1.1 Složitost algoritmu

Konečnost algoritmu nám říká, že výpočet algoritmu někdy určitě skončí. Neznamená to ale, že jsme schopni se na dnešních počítačích dobrat výsledku v rozumném čase. Jako příklad můžeme uvést algoritmus řešící šachovou partii za hráče, který je na tahu, a snažící se o vynucení jeho výhry. Tento algoritmus není příliš složitý a je konečný, avšak dopočítání partie do konce tak aby se výsledek dal využít, je nad síly dnešních počítačů resp. výpočet by trval neúměrně dlouho a zabral by na dnešní dobu ohromné množství paměti. I přes významný technologický pokrok a stále se zvětšující paměti počítačů budou vždy existovat algoritmy, pro které bude paměti málo.

Aby se daly algoritmy mezi sebou porovnávat, zavedlo se několik hodnotících kritérií. Nejpoužívanější je asymptotické vyjádření složitosti popsané v [4]. Čas i prostor potřebný pro výpočet algoritmu závisí na velikosti vstupních dat. Proto má složitost nejčastěji podobu funkce velikosti dat, udávané počtem položek *N*. U asymptotické časové složitosti se toto N blíží nekonečnu. Porovnání má podobu tří různých složitostí.

- O Omikron (velké O, Ό, big O) - vyjadřuje horní hranici chování
- Ω Omega - vyjadřuje dolní hranici chování
- Θ Theta - vyjadřuje třídu chování

V praxi nejvyužívanější je složitost Omikron. Vyjadřuje *horní hranici* časového chování algoritmu a tedy dobu, do které algoritmus určitě skončí. Toto však platí od určité hranice konstanty N_0. Před ní může být časová složitost algoritmu výrazně rozkolísaná a může být problematické toto chování matematicky popsat.

Složitost Omega je definovaná podobně jako Omikron. Pouze s tím rozdílem, že nevyjadřuje horní hranici časového chování algoritmu, ale *dolní hranici* časového chování algoritmu. Od určitého N_0 výše můžeme o algoritmu říci, že neskončí dříve než je tato doba dolní hranice.

Složitost Theta vyjadřuje třídu časového chování algoritmu. Tzn. určuje meze (horní a dolní hranici) časového chování, mezi kterými se od určitého N_0 pohybuje doba provádění algoritmu. Pro zajímavost si můžeme uvést několik tříd spolu s algoritmy:

- *Theta(1)* – Takto označujeme algoritmy s *konstantní* časovou složitostí.
- *Theta(log (n))* – Takto označujeme algoritmy s *logaritmickou* časovou složitostí. Základ logaritmu není podstatný, protože hodnoty logaritmu pro různé základy se liší pouze konstantou vzájemného převodu. Do této kategorie patří např. rychlé vyhledávací algoritmy (binární vyhledávání v seřazeném poli).
- *Theta(n)* – Takto označujeme algoritmy s *lineární* časovou složitostí. Patří sem běžné vyhledávací algoritmy a několik algoritmů sekvenčně zpracovávajících datové struktury (sekvenční vyhledávání v poli).
- *Theta(n*log(n))* – Takto označujeme algoritmy s *linearitmickou* časovou složitostí. Tuto časovou složitost mají např. rychlé řadicí algoritmy.
- *Theta(n^2)* – Takto označujeme algoritmy s *kvadratickou* časovou složitostí. Sem patří algoritmy obsahující dva cykly čítající do N. Jedná se např. o bublinové řazení nebo sekvenční vyhledávání v dvourozměrném poli.
- *Theta(n^3)* – Takto označujeme algoritmy s *kubickou* časovou složitostí. Používáme je převážně pro jednoduché problémy. U složitých problémů vzrůstá náročnost výpočtu nepříjemně rychle a snižuje se možné uplatnění výsledků výpočtu.
- *Theta(k^n)* – Takto označujeme algoritmy s *exponenciální* (pro k=2 *binomickou*) časovou složitostí. Existuje několik použitelných algoritmů s touto třídou složitosti, avšak pro náročnější problémy jsou nepoužitelné. Do této skupiny patří algoritmy pro lámání hesel hrubou silou (brute-force) nebo grafové algoritmy zabývající se problémem obchodního cestujícího.

3.1.2 Klasifikace algoritmů podle principu fungování

Algoritmy nemusíme dělit pouze pomocí asymptotické složitosti, ale dají se dělit i podle principu fungování jaký používají. Jeden algoritmus můžeme zařadit i do více kategorií.

- *Rekurzivní algoritmy* – Využívají pro výpočet volání sama sebe. Dají se pomocí nich řešit i vcelku složité problémy poměrně elegantně a přehledně, ale za cenu vyšší režie oproti algoritmům, které rekurzi nevyužívají.
- *Hladové algoritmy* – K výsledku se propracovávají po jednotlivých rozhodnutích, která, jakmile jsou jednou učiněna, už nejsou dále revidována. Nezaručují úspěšné nalezení řešení.

- *Rozděl a panuj algoritmy* – Dělí problém na podproblémy. Po vyřešení podproblémů se na jejich základě řeší hlavní problém. Nelze aplikovat na problémy, jež se nedají rozumně rozložit na podproblémy.
- *Pravděpodobnostní algoritmy* – Provádějí některá rozhodnutí závisle na náhodě. Jsou nedeterministické a nezaručují přesné a včasné vyřešení problému.
- *Algoritmy dynamického programování* – Řeší části problému postupně od jednodušších po složitější. Složitější části řeší na základě už vyřešených jednodušších.
- *Paralelní algoritmy (vícevláknové)* – Jsou navrženy pro současný běh na více výpočetních jednotkách a proto mohou dosahovat kratší doby výpočtu. Jsou náročnější na implementaci, ale tam, kde jednovláknové algoritmy nejsou vhodné (jsou pomalé), není jiná možnost. Nutno podotknout, že ne všechny problémy se dají řešit paralelním programováním.
- *Evoluční algoritmy* – Snaží se napodobit chování biologických evolučních procesů. Vybírají nejlepší dosažená řešení a snaží se navzájem skloubit jejich odlišné postupy výpočtu. Tím se snaží co nejvíce přiblížit optimálnímu řešení.
- *Heuristické algoritmy* – Jsou navrženy pro nalezení řešení s určitou přesností. Často se pomocí nich řeší problémy, u kterých by nalezení přesného řešení bylo nepřijatelně časově náročné, nebo ty problém, kde algoritmus pro přesné řešení není znám.

3.2 Dijkstrův algoritmus

Autorem tohoto algoritmu uveřejněného v roce 1959 je nizozemský informatik Edsger Wybe Dijkstra, který byl roku 1972 oceněn Turingovou cenou za svůj přínos informatice.

Jedná se o konečný algoritmus pracující nad hranově ohodnocenými souvislými grafy, pomocí kterého jsme schopni nalézt délku nejkratší cesty z počátečního vrcholu do všech ostatních. Hrany grafu musí být ohodnoceny nezápornými hodnotami. Jestliže hledáme cestu do konkrétního vrcholu, výpočet můžeme ukončit dříve při označení tohoto vrcholu jako trvalého (viz. dále), a v dalších iteracích algoritmu již nepokračujeme. Pokud projdeme graf celý a hledáme délku cesty z počátečního vrcholu do všech ostatních, je třída asymptotické časové složitost algoritmu kvadratická (O N^2). Pokud hledáme délky nejkratších cest mezi všemi dvojicemi vrcholů, musíme algoritmus použít tolikrát, kolik je v grafu vrcholů (pro každý různý počáteční vrchol). Tím se složitost násobí počtem vrcholů a stává se složitostí kubickou (O N^3).

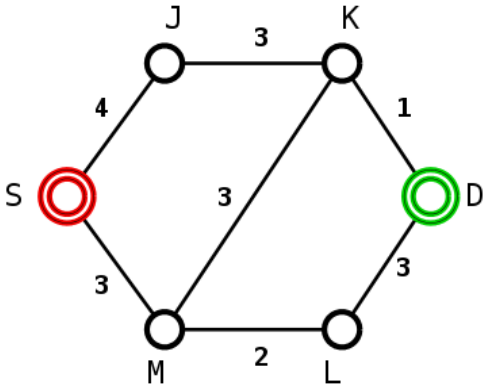
Vstupem je *abstraktní datová struktura graf*, *počáteční vrchol* a *koncový vrchol*, mezi kterými chceme nalézt nejkratší cestu. Dijkstrův algoritmus uchovává u každého vrcholu grafu hodnotu vyjadřující nejkratší nalezenou délku cesty k počátku a jeho stav, který určuje jestli je nalezená cesta dočasná nebo již trvalá. Pro pozdější rekonstrukci cesty je nutné uchovávat u každého vrcholu ještě odkaz na jeho předchůdce. U vrcholu počátečního je uchováván odkaz na sebe sama.

Slovní popis algoritmu:

1. Nastavíme u všech vrcholů, kromě počátečního, jejich nejkratší cestu k počátku na maximální hodnotu nebo jinou unikátní. U počátečního vrcholu nastavíme vzdálenost nulovou.
2. Nastavíme stav každého vrcholu na dočasný.
3. Z vrcholů označených jako dočasné vybereme vrchol v, který má nejnižší hodnotu délky cesty k počátku. V prvním cyklu to bude počáteční vrchol.
4. Prohlásíme vrchol v za trvalý a nastavíme u něj příslušný příznak.
5. Aktualizujeme ohodnocení všech dočasných sousedů s_i vrcholu v. Porovnáme aktuální ohodnocení s_i se součtem ohodnocení v a ohodnocení hrany $h \equiv \{v, s_i\}$. Pokud je aktuální ohodnocení s_i větší, přiřadíme mu druhou z porovnávaných hodnot a nastavíme předchůdce uzlu s_i na vrchol v. Pokud není aktuální ohodnocení s_i větší, pokračujeme dál v aktualizování ohodnocení všech sousedů uzlu v.
6. Pokud nejsou všechny vrcholy grafu nastaveny jako trvalé a zároveň není koncový vrchol označen za trvalý, pokračujeme bodem 3.

Předešlý postup výpočtu si zkusíme aplikovat na jednoduchý graf zobrazený na obrázku 1. Po průchodu jedním cyklem algoritmu si zobrazíme aktuální stav a rozvedeme změny oproti předchozímu stavu. Počátečním vrcholem hledané cesty je červený vrchol S a koncovým je zelený vrchol D.

- **1. cyklus algoritmu**



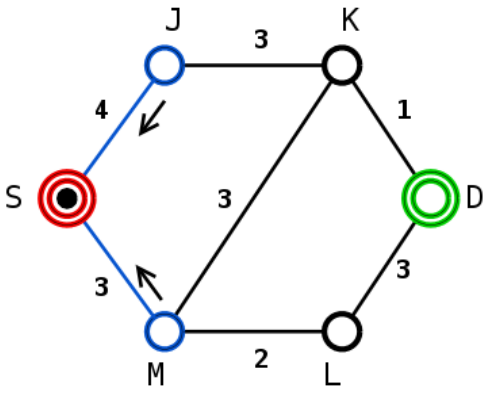
Obr. 1 – Hranově ohodnocený graf

Nastavíme hodnoty nejkratších cest z vrcholů k počátku na maximální možnou (MaxInt), u počátečního vrcholu *S* nastavíme hodnotu 0 a všechny vrcholy označíme jako dočasné. Tím splníme první dva body algoritmu.

Poté následuje zbylá část algoritmu, která se opakuje do doby, než je označen koncový vrchol cesty za trvalý, nebo dokud nejsou všechny vrcholy označeny za trvalé.

Vybereme z dočasných vrcholů ten co má nejnižší ohodnocení (má hodnotu délky cesty k počátku nejnižší) a označíme ho za trvalý. Nebudeme uvažovat případ nesouvislého grafu a budeme pro jednoduchost vybírat z vrcholů, které jsou dočasné a *nemají* ohodnocení rovné MaxInt. V prvním cyklu je vždy vybrán počáteční vrchol z hledané cesty, protože má na začátku vždy nejnižší ohodnocení. V našem případě se tedy jedná o vrchol *S*. Každému jeho dočasnému sousedovi, jak vidíme jedná se o vrcholy *J* a *M*, přiřadíme nižší hodnotu z dvou následujících. První hodnota je součet délky cesty z *S* k počátku, zde hodnota 0, s hodnotou hrany vedoucí z *S* k sousedovi. Druhá z porovnávaných hodnot je délka cesty k počátku, která je v sousedovi již uložena. Pokud se hodnota souseda změní, tzn. je mu přiřazena nižší hodnota než již nese, je nastaven jeho předchůdce na vrchol, přes který se k němu algoritmus propracoval. Toto je nutné pro pozdější rekonstrukci nejkratší cesty. U sousedního vrcholu *J* se porovnává nalezená hodnota (0+4) s hodnotou, která je uzlu přiřazená tj. MaxInt. Nižší je hodnota 4 a tu přiřadíme vrcholu *J*. Předchůdce vrcholu *J* je nastaven na vrchol *S*. U sousedního vrcholu *M* se porovnávají hodnoty (0+3) taktéž s hodnotou MaxInt. Nižší je hodnota 3 a tu přiřadíme vrcholu *M*. Předchůdce vrcholu *M* je nastaven na vrchol *S*. Obrázek 2 ilustruje v jakém stádiu výpočtu se nacházíme po prvním průchodu algoritmem. Zatím jsme se tedy jen „podívali“ na sousedy počátečního vrcholu.

Černá tečka uvnitř vrcholu *S* symbolizuje jeho označení jako trvalý. Modrou barvou jsou znázorněny hrany a vrcholy, které jsme prošli pomocí Dijkstrova algoritmu. Odkaz na předchůdce vrcholu je znázorněn černou šipkou.

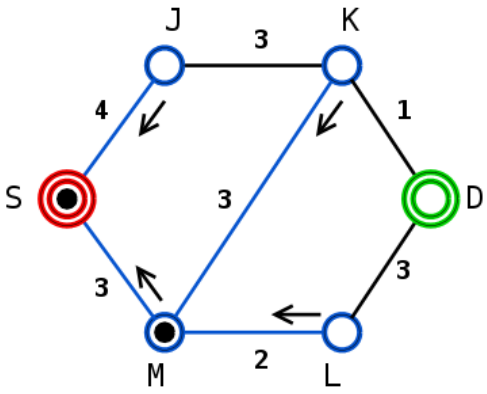


Obr. 2 - Průběh po prvním cyklu

Vrchol *S* nese hodnotu cesty 0 a je již trvalý. Vrchol *J* nese hodnotu cesty 4. Vrchol *M* nese hodnotu cesty 3. Vrcholům *J* a *M* byl přiřazen stejný předchůdce, vrchol *S*, a jsou stále označeny jako dočasné. Nyní můžeme přistoupit k dalšímu cyklu algoritmu, protože koncový vrchol cesty není označen jako trvalý a stále existují vrcholy označené jako dočasné.

- **2. cyklus algoritmu**

Opět vybereme z dočasných vrcholů ten co má nejnižší ohodnocení a označíme ho za trvalý. V tomto případě máme na výběr již z více vrcholů než jen z počátečního. Vrcholy s hodnotou MaxInt nebereme v potaz. Vrchol *J* nese hodnotu 4 a vrchol *M* hodnotu 3. Vybereme proto vrchol *M*, protože má ohodnocení nižší. Označíme ho za trvalý a všem jeho dočasným sousedům, jak vidíme jedná se o vrcholy *K* a *L*, opět přiřadíme nižší ze dvou následujících hodnot. První hodnota u vrcholu *K* je součet cesty z *M* k počátku a hrany z *M* do *K*, tedy 3+3. Druhá z porovnávaných hodnot je MaxInt uložená v *K*. Nižší je hodnota 6 a tu přiřadíme vrcholu *K*. Předchůdce vrcholu *K* nastavíme na vrchol *M*. U vrcholu *L* porovnáváme hodnoty 3+2 taktéž s MaxInt. Nižší je hodnota 5 a tu přiřadíme vrcholu *L*. Předchůdce vrcholu *L* je nastaven na vrchol *M*. Postup výpočtu máme graficky znázorníme na obrázku 3.



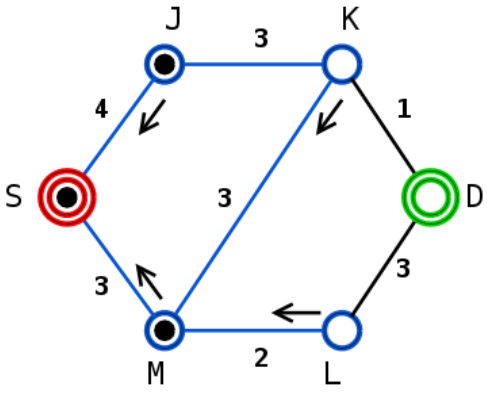
Obr. 3 – Průběh po druhém cyklu

Zde vidíme pokračující průchod Dijkstrova algoritmu grafem. Za trvalý byl označen vrchol *M* a hodnota nejkratší cesty k počátku se u něho již měnit nebude. Tato délka cesty je rovna nejkratší existující cestě a může být pomocí odkazů na předchůdce rekonstruována.

Oproti předchozímu cyklu se změnilo následující. Vrchol *M* nese hodnotu cesty 3 a stal se trvalým. Vrchol *K* nese hodnotu cesty 6. Vrchol *L* nese hodnotu cesty 5. Vrcholům *K* a *L* byl přiřazen stejný předchůdce, vrchol *M*, a jsou stále označeny jako dočasné. Nyní můžeme přistoupit k dalšímu cyklu algoritmu, protože koncový vrchol cesty není označen jako trvalý a stále existují vrcholy označené jako dočasné.

- **3. cyklus algoritmu**

Znova vybereme z dočasných vrcholů ten co má nejnižší ohodnocení a označíme ho za trvalý. Opět máme na výběr z více vrcholů. Vrchol *J* nese hodnotu 4, vrchol *K* hodnotu 6 a vrchol *L* hodnotu 5. Vybereme proto vrchol *J*, který nese nejnižší hodnotu. Označíme ho za trvalý a všem jeho dočasným sousedům, jak vidíme jedná se pouze o vrchol *K*, přiřadíme nižší hodnotu z dvou následujících. První hodnota je součet délky cesty z *J* k počátku, zde hodnota 4, s hodnotou hrany vedoucí z *J* do *K*, tedy 3. Druhá z porovnávaných hodnot je délka cesty k počátku, která je ve vrcholu *K* již uložena. Porovnáváme hodnoty (4+3) s již přiřazenou hodnotou 6. Nižší je hodnota 6, která je již přiřazená a proto ji není potřeba aktualizovat. Předchůdce vrcholu *K* zůstává nezměněn a je nastaven na vrchol *M* kudy vede nejkratší cesta. Přes vrchol *J* do počátku cesta vede, ale ne už nejkratší. Nyní si postup výpočtu graficky znázorníme na obrázku 4.



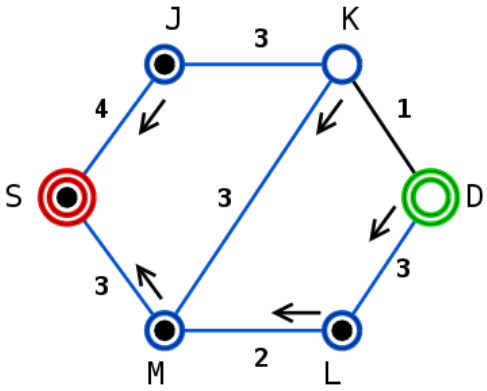
Obr. 4 – Průběh po třetím cyklu

Za trvalý byl označen vrchol *J* a hodnota nejkratší cesty k počátku se u něho již měnit nebude. Odkaz na předchůdce vrcholu *K* se nezměnil, protože jsme neaktualizovali jeho hodnotu nejkratší cesty k počátku. Nejkratší cesta k počátku vede stále přes vrchol *M*.

Oproti předchozímu cyklu se změnilo následující. Vrchol *J* nese hodnotu cesty 4 a stal trvalým. Vrchol *K* nese hodnotu cesty 6. Vrchol *L* nese hodnotu cesty 5. Vrcholy *K* a *L* mají stále přiřazeného stejného předchůdce, vrchol *M*. Nyní můžeme přistoupit k dalšímu cyklu algoritmu, protože koncový vrchol cesty není označen jako trvalý a stále existují vrcholy označené jako dočasné.

- **4. cyklus algoritmu**

Nyní se již pomalu blížíme k závěru. Opakujeme znova výběr z dočasných vrcholů. Vybereme vrchol s nejnižším ohodnocením a označíme ho za trvalý. V tomto případě máme na výběr z vrcholů *K* a *L*. Vrchol *K* nese hodnotu 6 a vrchol *L* hodnotu 5. Vybereme proto vrchol *L*. Označíme ho za trvalý a všem jeho dočasným sousedům, jak vidíme jedná se pouze o cílový vrchol *D*, přiřadíme nižší hodnotu z následujících dvou. První hodnota je součet délky cesty z *L* k počátku, zde hodnota 5, s hodnotou hrany vedoucí z *L* do *D*, tedy 3. Druhá z porovnávaných hodnot je délka cesty k počátku, která je ve vrcholu *D* již uložena. Porovnávají se hodnoty (5+3) s MaxInt. Nižší je hodnota 8 a tu přiřadíme vrcholu *D*. Předchůdce vrcholu *D* nastavíme na vrchol *L* kudy vede prozatím nejkratší nalezená cesta. Protože vrchol *D* ještě nebyl označen za trvalý, nemusí tato cesta být absolutně nejkratší. Nyní si postup výpočtu graficky znázorníme na obrázku 5.



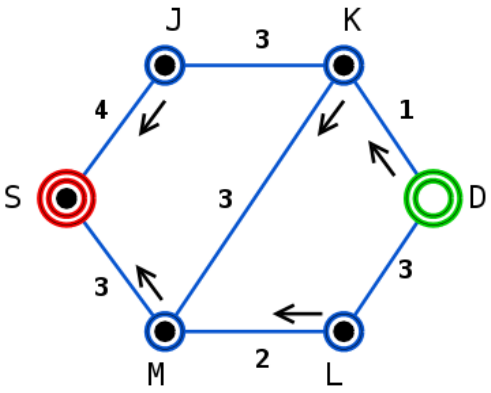
Obr. 5 – Průběh po čtvrtém cyklu

Za trvalý byl označen vrchol *L* a vede přes něj nejkratší nalezená cesta z koncového vrcholu *D* do počátku. Označit tuto cestu jako nejkratší a ukončit tím provádění algoritmu by bylo ale předčasné. Koncový vrchol *D* ještě není označen jako trvalý.

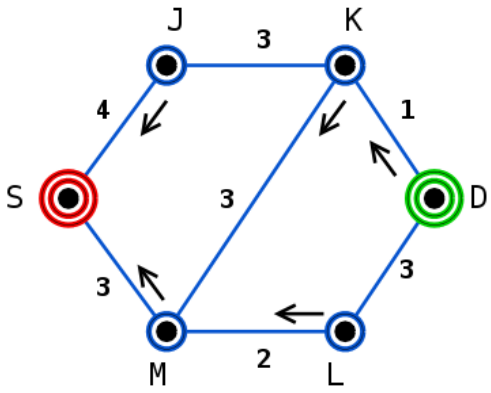
Vrchol *L* byl označen jako trvalý a nese hodnotu cesty 5. Vrcholu *D* byl přiřazen předchůdce v podobě vrcholu *L*. Nyní můžeme přistoupit k dalšímu, a lze odhadnout že již předposlednímu, cyklu algoritmu. Počet cyklů Dijkstrova algoritmu se totiž rovná počtu vrcholů v souvislém grafu.

- **5. a 6. cyklus algoritmu**

Vybereme vrchol s nejnižším ohodnocením a označíme ho za trvalý. V tomto případě máme na výběr z *K* a *D*. Vrchol *K* nese hodnotu 6 a vrchol *D* hodnotu 8. Vybereme proto vrchol *K*. Označíme ho za trvalý a všem jeho dočasným sousedům, jak vidíme jedná se pouze o cílový vrchol *D*, přiřadíme nižší hodnotu. První je součet délky cesty z *K* k počátku, zde hodnota 6, s hodnotou hrany vedoucí z *K* do *D*, tedy hodnota 1. Druhá z porovnávaných hodnot je délka cesty k počátku, která je ve vrcholu *D* již uložena. *A* to je hodnota 8. Porovnávají se hodnoty (6+1) s hodnotou 8. Nižší je hodnota 7 a tu přiřadíme vrcholu *D*. Předchůdce vrcholu *D* musíme změnit na vrchol *K*, protože jsme aktualizovali nejkratší nalezenou cestu. Nyní si postup výpočtu graficky znázorníme na obrázku 6.



Obr. 6 – Průběh po pátém cyklu



Obr. 7 – Konečný stav

Vrchol *K* nese hodnotu 6 a byl označen jako trvalý. Vrcholu *D* byla aktualizována hodnota na 7 a byl změněn předchůdce. Do množiny dočasných vrcholů patří pouze vrchol *D*, který je současně i vrcholem koncovým. V posledním cyklu algoritmu, vybereme a označíme vrchol *D* jako trvalý. Tím výpočet nejkratší cesty mezi vrcholy *S* a *D* končí. Mohli bychom pokračovat ve výpočtu kdyby existovaly ještě nějaké dočasné vrcholy, ale šlo nám o nalezení cesty z *S* do *D* a další výpočty by byly zbytečné. Konečný stav je znázorněn na obrázku 7. Cestu získáme postupným průchodem zpět k počátečnímu vrcholu po odkazech na předchůdce vrcholu. U našeho příkladu je nejkratší nalezená cesta *S*{S, M}*M*{M, K}*K*{K, D}*D* a má délku 7. Můžeme si všimnout, že přestože hledáme nejkratší cestu pouze mezi dvěma vrcholy, algoritmus najde nejkratší cestu mezi počátečním vrcholem a všemi co jsou označeny za trvalé. Pro nalezení nejkratších cest mezi všemi dvojicemi vrcholů stačí aplikovat Dijkstrův algoritmus pro každý vrchol grafu.

3.3 Floyd-Warshallův algoritmus

Podle [3] je autorem dnes již Emeritní Profesor na Pařížské univerzitě Dauphine Bernard Roy, který ho popsal a uveřejnil v roce 1959. K návrhu algoritmu přispíval i významný vědec zabývající se informatikou a později oceněný Turingovou cenou Robert Floyd. Algoritmus je proto někdy uváděn jako Roy-Floydův. Důkaz o správnosti algoritmu přinesl Stephen Warshall. Dnes se díky tomu většinou používá název Floyd-Warshallův. K důkazu se pojí historka o jeho vzniku. Bylo to prý díky sázce s jedním z kolegů ve společnosti Technical Operations, kde Stephen Warshall v té době pracoval. Vsadili se o to, kdo tento algoritmus dříve matematicky dokáže. Druhý den přišel Stephen do práce s jedním popsaným listem papíru, kde byl tento důkaz učiněn. Vyhrál tím sázku o jednu láhev Rumu.

Jedná se o algoritmus sloužící k nalezení nejkratších cest mezi všemi dvojicemi vrcholů v hranově ohodnoceném neorientovaném grafu. Toto zajišťuje jediný průchod grafem. Pracuje se třemi vnořenými cykly čítajícími do počtu vrcholů v grafu a proto je horní hranice asymptotické

časové složitosti kubická (O N^3). Horní hranice asymptotické paměťové složitosti je kubická (O N^2). Tento algoritmus můžeme zařadit do kategorie algoritmů využívajících techniku dynamického programování. Technika používání výsledků z jednodušších částí problému pro výpočet složitějších částí je zde uplatněna velmi významně.

Před začátkem výpočtu si do čtvercové matice sousednosti uložíme vzdálenosti mezi vrcholy. Matice sousednosti je dvourozměrné čtvercové pole o délce strany odpovídající počtu vrcholů grafu. Řádková a sloupcová souřadnice jednoznačně určuje dvojici vrcholů grafu. V prvcích matice je uložena délka cesty mezi vrcholy. Tam, kde hrana neexistuje, je uložena hodnota značící nekonečno. Na hlavní diagonále jsou uloženy cesty o nulové délce. Nad takovouto maticí probíhá výpočet a je v ní udržována nejkratší nalezená délka cesty pro každou dvojici.

Nyní můžeme přistoupit k samotnému algoritmu. Máme graf *G* o vrcholech 1 až N. Čtvercová matice sousednosti D[i][j] o rozměrech N*N nese ohodnocení hran grafu. Algoritmus se skládá z N fází. Na konci k-té fáze budeme porovnávat délku cesty uloženou v D[i][j], která může obsahovat vrcholy 1 až k-1, s délku cesty, která může navíc obsahovat vrchol k. Porovnáme hodnotu D[i][j] s hodnotou D[i][k] + D[k][j]. Kratší z nich je do prvku D[i][j] uložena. Po absolvování všech N fází je algoritmus u konce a můžeme z matice sousednosti zjistit délky nejkratších cest mezi všemi dvojicemi vrcholů v grafu.

S rekonstrukcí cest algoritmus nepočítá, ale o tuto funkcionalitu se dá jednoduše rozšířit. Jde o podobný princip jako odkazy na předchozí uzly v nejkratší cestě u Dijkstrova algoritmu. Vytvoříme si proto rekurzivní metodu. Ta vybere ze sousedů počátečního vrcholu takové, kteří mají nejkratší cestu ke koncovému vrcholu, a zařadí je do zásobníku. V dalším kroku vybere sousedy prvního vrcholu v zásobníku, kteří mají nejkratší cestu ke koncovému vrcholu, a také je zařadí do zásobníku. Po dosažení koncového vrcholu je cesta vypsána a ze zásobníku je vybrán další vrchol a takto až do doby než je zásobník prázdný. Jedná se o způsob prohledávání do hloubky (DFS).

Protože je algoritmus vcelku jednoduchý, nebudeme si graficky demonstrovat jeho průběh a pouze si ukážeme symbolický zápis.

```
n:=pocet uzlu grafu G (rozmer matice D)
for k:=1 to n do
   for i:=1 to n do
      for j:=1 to n do
         D[i,j]:=min (D[i,j], D[i,k]+D[k,j])
      end for
   end for
end for
```

Jednoduchost a elegantnost tohoto algoritmu pravděpodobně žádný jiný algoritmus řešící stejný problém nepředčí.

3.4 Bellman-Fordův algoritmus

Bellman-Fordův algoritmus hledá délku nejkratší cesty v ohodnoceném grafu. Některé hrany grafu mohou nést i záporné ohodnocení. Tím se liší od Dijkstrova algoritmu, který vyžaduje nezáporné ohodnocení u všech hran. Rychlost výpočtu je však nižší a proto se Bellman-Fordův algoritmus snažíme používat jen tam, kde má své opodstatnění.

Jako praktický příklad využití tohoto algoritmu můžeme uvést implementaci ve směrovacím protokolu RIP (Routing Information Protocol), který slouží ke komunikaci směrovačů propojených v síti. Bellman-Fordův algoritmus je využit k nalezení nejkratších cest mezi prvky sítě a přibližně do 30 sekund po změně topologie sítě je aktualizována směrovací tabulka směrovačů.

Opět je zde využito techniky dynamického programování jako u Floyd-Warshallova algoritmu. Uvedeme si nástin možné implementace, kterou jsem však kvůli vyšší časové složitosti Bellman-Fordova algoritmu oproti předchozím algoritmům nerealizoval.

Do polí A[i], B[i] a T[i] si uložíme počáteční vrchol hrany, koncový vrchol hrany a délku hrany pro každou hranu grafu. Dále si budeme ukládat délky sledů do dvourozměrného pole D[l][i], kde délka bude brána od vrcholu 1 k vrcholu i a ta bude sestrojena z l hran. V prvcích pole D[0][i] se pro všechny vrcholy nachází hodnota vyjadřující nekonečno. Výjimkou je pouze vrchol i=1, kde je opravdu možné se dostat k vrcholu 1 bez použití hrany. Při předpokladu, že známe D[l-1][i] pro všechna i a pro nějaké l>0, můžeme dopočítat D[l][i] pro jakékoliv i. Při výpočtu projdeme všechny hrany a hledáme minima pro všechny vrcholy. Výsledkem je minimum z D[l][N] pro l od 1 do N-1. V nejkratší cestě o l hranách můžeme poslední vynechat a zůstane cesta o l-1 hranách, která končí na začátku hrany l. Pak stačí zkontrolovat všechny možnosti pro poslední hranu. Když bude existovat cesta mezi vrcholy 1 a N, bude se skládat z l-1 hran. Toto by nemuselo platit, kdyby se v grafu nacházela uzavřená cesta.

Algoritmus hledá nejkratší cesty a proto hrozí jeho zacyklení v záporně ohodnocených uzavřených cestách. Pro jednoduchost můžeme předpokládat neexistenci záporných cyklů v grafu, jinak bychom museli implementovat mechanizmus, který je odhalí. To by v důsledku dále zvýšilo časovou náročnost provádění algoritmu.

3.5 Porovnání algoritmů

Všechny tři představené algoritmy najdou významné uplatnění v různých oblastech. Každý má svá specifika, která je předurčující k řešení rozdílných typů úloh. Od každého algoritmu lze v literatuře, ale i na internetu, nalézt velké množství modifikací.

Pokud jako hlavní kritérium pro srovnání bereme jednoduchost zápisu, je na tom Floyd-Warshallův algoritmus nejlépe. Je jednodušší na zápis a dalo by se říci i přímočařejší než Dijkstrův algoritmus. Bellman-Fordův algoritmus je ze všech třech nejsložitější a jeho provádění zabere nejvíce času. U Floyd-Warshallova algoritmu nesmíme přehlédnout, že pracuje nad maticí sousednosti a tu je, při obvyklé reprezentaci grafu dynamickými strukturami, nutné vytvořit.

Co se týče asymptotické časové složitosti při nalezení nejkratších cest mezi všemi vrcholy, jsou na tom první dva algoritmy stejně. Jejich náročnost vychází pouze z počtu vrcholů grafu. Bellman-Fordův algoritmus pracuje se složitostí O V*H, kde V je množina vrcholů a H množina hran. Pokud přesáhne počet hran grafu počet jeho vrcholů, a to je u souvislých grafů velmi často, je algoritmus složitější.

Pro řešení úlohy bakalářské práce se nejvíce hodí Dijkstrův a Floyd-Warshallův algoritmus. Není nutné se zabývat grafy se záporným ohodnocením a proto Bellman-Fordův algoritmus do implementace nezahrnu.

4 Implementace

Úkolem práce je vytvořit program pro hledání všech nejkratších cest v grafu mezi všemi dvojicemi jeho vrcholů. Na tuto úlohu je vhodných hned několik programovacích jazyků, avšak já jsem se rozhodl pro implementaci v jazyce Java. Pro dosažení nejefektivnějšího běhu programu by bylo pravděpodobně nejlepší použít jeden z nízkoúrovňových procedurálních jazyků typu C nebo Pascal, avšak k použití Javy mě vedlo několik důvodů. Prvním byl objektový model programování, který umožňuje jednodušší abstraktní pohled na problém. Druhým byla velmi široká základna jak uživatelů, tak aktivních vývojářů, a s tím spojený příslib do budoucna, že se Java nestane mrtvým jazykem. Nedávné uvolnění zdrojových kódů Java JDK pod licencí GPL tento příslib jen umocňuje. Důsledkem rozšířenosti je i snadná dostupnost literatury. Další z důvod proč jsem zvolil Javu je téměř bezproblémová přenositelnost vhodně napsaného programu mezi operačními systémy. V implementaci se budu snažit nepoužívat platformě závislé funkce.

4.1 Jazyk Java

První práce na tomto jazyce započaly v roce 1990, kdy byl James Gosling a jeho tým ve společnosti Sun Microsystems pověřeni projektem Oak dávající si za cíl vytvoření nového programovacího jazyka a virtuálního stroje pro běh programu. Jiný programovací jazyk s názvem Oak v té době již existoval a proto byl zvolen oficiální název Java. První verze Java 1.0 byla veřejnosti představena 23. května 1995. Java je objektově orientovaný interpretovaný multiplatformní programovací jazyk se syntaxí vycházející ze zaběhlých jazyků C a C++. Při návrhu jazyka Java se pracovalo se zkušenostmi z objektového C++. Tým projektu Oak se snažil vyvarovat některých typických neduhů jazyka C++. Např. byla zavedena silná typová kontrola, odstranění vícenásobné dědičnosti a nemožnost používat ukazatele. Byl kladen důraz na jednoduchost, jednoznačnost a přehlednost zápisu. Z dalších zvláštností jazyka můžeme uvést např. nemožnost použití příkazu goto, interní uložení všech textových dat ve formátu Unicode a zamezení používání globálních proměnných a funkcí.

Program napsaný v Javě je kompilován do mezikódu (anglicky bytecode), kde je stále zachována přenositelnost. Mezikód je možné spouštět ve virtuálních strojích (JVM – Java Virtual Machine) na různých platformách. Implementace virtuálního stroje platformě závislá je. O správu paměti se nestará program samotný, ale virtuální stroj (JVM). Aplikace si o paměť pouze žádají. Pokud je nějaká volná k dispozici, virtuální stroj ji poskytne programu. Pokud je jí nedostatek, je použit tzv. Garbage collector k uvolnění již nepoužívaných paměťových bloků. Tímto přístupem k alokaci paměti jsou odstraněny nepříjemné problémy, když v programu zapomeneme uvolnit

nepotřebné rezervované místo (anglicky memory leak). Hlavní výhodou oproti staticky kompilovaným jazykům je multiplatformnost aplikací.

Abych jenom nechválil, musím zmínit i některé nevýhody. Spuštění programu je vždy podmíněno spuštěním JVM a to má za následek jeho celkově pomalejší start. Interpretovanost sebou přináší pomalejší běh programu, a proto se zavedla technika JIT (Just In Time) kompilátoru. Jedná se o kompilaci často prováděných částí programu do nativního kódu a tím jejich zrychlení.

Javu se Sun Microsystems pokusil v roce 1997 standardizovat u organizace ISO/IEC, ale splnit tento záměr se ukázalo být náročnější než se čekalo, a později od něho upustil. Díky velké snaze vývojářů nejenom ze Sunu se daří udržovat Javu maximálně kompatibilní a stala se průmyslovým standardem. Licence, pod kterou byla Java šířena, si vynucovala kompatibilitu v implementaci a zabraňovala roztříštění aplikačního programového rozhraní. Dokonce se v tomto rozhořel soudní spor s firmou Microsoft o to, že implementace Javy dodávaná k jejím operačním systémům nese záměrně nekompatibilní úpravy a tím znemožňuje přenositelnost. Soud dal zapravdu žalobci, firmě Sun, a nařídil tyto nekompatibility odstranit. Od té doby Microsoft nepokračuje ve vývoji vlastní implementace Javy a ani ke svým systémům nedodává implementaci třetích stran. Licenční politika firmy Sun ohledně Javy byla často kritizována za svoji přísnost a nesvobodnost. Díky licenci nebyla začleněna do mnoha svobodných operačních systémů pod licencí GNU GPL. Později byla Java vydána duálně i pod licencí Operating System Distributor's License, která toto již umožnila. I přesto u mnoha vývojářů převládal názor, že úplné uvolnění Javy jako svobodného software by bylo lepší řešení. K tomuto závěru došli i vedoucí pracovníci Sun Microsystems a 12. listopadu 2006 započalo uvolňování první části zdrojových kódů pod svobodnou licencí GNU GPL verze 2. Proces uvolňování skončil 18. května 2007. Jak se tento odvážný krok projeví v budoucnu je těžké odhadnout, ale převládá názor, že se sice roztříští kompatibilita mezi některými implementacemi, ale jako referenční bude stále brána ta od Sunu, tedy projektu OpenJDK.org.

4.1.1 Dělení podle platforem

Implementace jazyka Java se dělí na platformy podle způsobu využití. Pro různé specifické účely je výhodné použít upravené virtuální běhové prostředí (JVM) s upraveným rozhraním pro programování aplikací (API). Je tak možné dosáhnout nižší náročnosti na hardware v oblasti mobilních zařízení nebo optimalizovat běh informačních systémů v podnikovém prostředí.

4.1.1.1 Java Card

Umožňuje běh jednoduchých aplikací (nazývaných aplety) na zařízeních s velmi omezenou kapacitou paměti. Těmi mohou být „chytré“ bankovní karty, SIM karty, karty používající se v hromadné dopravě a pod.. Je zde zajištěna nezávislost na standardu karty a tím přenositelnost programů. Díky

Java Card technologii se vytváření, udržování a úpravy softwaru v těchto zařízeních velmi zjednodušily a významně se tím snížily náklady na vývoj. Obsahuje ji celosvětově asi 90% všech nově vyrobených chytrých karet. Jedná se o Java platformu určenou nejmenším vestavěným (embedded) zařízením.

4.1.1.2 Java ME (Micro Edition)

Dříve označovaná jako *Java 2 Micro Edition*. Je určená pro složitější zařízení než je technologie Java Card, avšak stále s omezenou pamětí a omezeným výpočetním výkonem. Těmito zařízeními mohou být mobilní telefony, osobní digitální asistenti (PDA), hudební přehrávače apod.. Velkou hnací silou pro tuto platformu je komerční úspěch mnoha aplikací (hlavně her pro mobilní telefony). Používá se podle odhadů ve více než 1,5 miliardách přístrojů.

4.1.1.3 Java SE (Standard edition)

Toto je hlavní vývojová větev Javy, od které byly později odvozeny ostatní. Začala se vyvíjet už v roce 1991 v rámci projektu Oak. Až po odštěpení ostatních platforem se zavedl název Java SE. Používá se na běžných osobních počítačích (desktop i server), kde nejsme tak výrazně omezeni pamětí a výpočetním výkonem jako na mobilních telefonech nebo PDA. Slouží jako základ platformy Java EE

4.1.1.4 Java EE (Enterprise Edition)

Platforma určená pro vývoj přenositelných, robustních, škálovatelných a bezpečných serverových aplikací. Jako základ si bere Java SE, kterou rozšiřuje o podporu tvorby webových aplikací, webových služeb a distribuovaných vícevrstvých aplikací. Cílem Java EE je poskytnou infrastrukturu pro usnadnění vývoje.

4.2 Program

Implementace programu probíhala ve vývojovém prostředí Eclipse, které představuje jednotnou platformu pro návrh softwaru. Lze v ní tvořit projekty i pomocí jiných jazyků než je Java. Projekt Eclipse započala firma IBM, která pro jeho podporu vyvinula framework SWT. Frameworkem se označuje balík zahrnující podpůrný software pro vývoj programů. Díky SWT získává Eclipse stejné vlastnosti a vzhled jako nativní aplikace na konkrétním systému. Podpora zásuvných modulů je významným rysem této platformy a rozšiřuje její funkčnost. Existuje na několik stovek modulů avšak já jsem žádný z nich nepoužil, protože základní možnosti Eclipse byly pro mě dostačující.

Pro návrh bylo použito vývojové prostředí Eclipse 3.2.2 a jazyk Java 1.6.0.00 pod operačním systémem GNU/Linux.

Program můžeme spustit z terminálu jako jakýkoliv jiný program psaný v Javě. Při nesprávnosti vstupních parametrů program vypíše způsob jeho použití. Požaduje se uvedení parametru odkazujícího na vstupní soubor se strukturou grafu a parametru určujícího typ algoritmu, kterým se má úloha řešit. Volitelný je pak parametr pro potlačení výstupních dat. Toho využijeme při porovnávání rychlostí výpočtu, kdy nechceme aby byla doba běhu programu ovlivněna výpisy cest.

4.2.1 Vlastnosti tříd

V následujících podkapitolách si popíšeme jakou roli hrají třídy v programu a rozebereme si jejich úlohu.

4.2.1.1 PathFinder.java

Jedná se o hlavní třídu programu. Jejím úkolem je volání konkrétních algoritmů pro výpočet nejkratších cest a výpis nejkratších nalezených cest.

4.2.1.2 GraphAlgorithm.java

Tento soubor obsahuje rozhraní přes které přistupujeme k metodám programu. Je tím vytvořen jednotný způsob volání metod.

4.2.1.3 FileParser.java

Obsahuje jedinou metodu sloužící k načítání vstupních dat. Podle rozměrů uvedených ve vstupním souboru vytvoří matici sousednosti, naplní ji daty a předá jako svůj výstupní parametr.

4.2.1.4 Dijkstra.java

Sjednocuje obsluhu výpočtu Dijkstrovým algoritmem.

4.2.1.5 FloydWarshall.java

Řeší problém nalezení všech nejkratších cest pomocí Floyd-Warshallova algoritmu. Součástí je i rekurzivní metoda sloužící k sestavení nejkratších cest.

4.2.1.6 dijkstra/Algorithm.java

V tomto souboru je implementováno hledání všech nejkratších cest Dijkstrovým algoritmem. Je rozšířen o dříve zmiňované pole předchůdců a nemá vysoké nároky na paměť. Pro výpis cest se používá rekurzivní metoda.

4.2.1.7 dijkstra/Graph.java

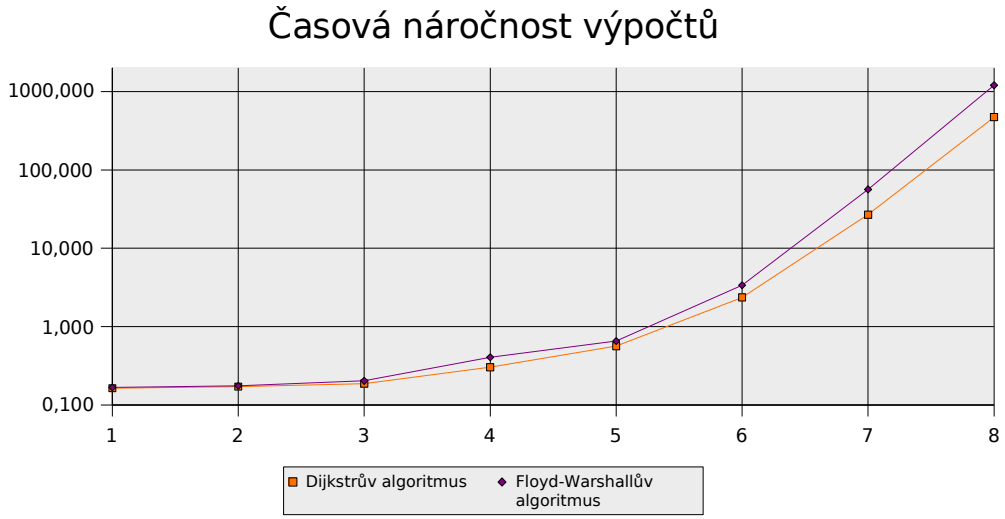
Obsahuje metody pro práci s datovými strukturami potřebnými pro výpočet pomocí Dijkstrova algoritmu.

4.2.2 Měření časové náročnosti výpočtů

V této kapitole uvedu několik výsledků experimentálních měření Dijkstrova a Floyd-Warshallova algoritmu a pokusím se je objasnit. Budu pozorovat závislost složitosti grafu na době výpočtu nejkratších cest. Pro zaznamenávání doby běhu programu použiji konzolovou utilitu time obsaženou v OS GNU/Linux, která byla za tímto účelem vyvinuta. Při spouštění programu použiji parametr -s aby se do doby výpočtu nezapočítávala doba výpisu cest. Rychlost běhu také závisí na rychlosti používaného osobního počítače (zejména CPU), velikosti dostupné operační paměti a použitého softwarového vybavení. Konfigurace testovacího počítače je následující:

- CPU Intel Pentium M (jádro Banias) 600MHz
- RAM 1280MB
- OS GNU/Linux (jádro 2.6.20)
- Java 1.6.0.00

Měření budu provádět na grafech s topologií n-rozměrných krychlí, přičemž n bude nabývat hodnot popořadě 1, 2, 3, 4, 5, 6, 7 a 8. Počet vrcholů grafů bude popořadě 2, 4, 8, 16, 32, 64, 128 a 256. Měření provedu opakovaně a z výsledků vytvořím průměrnou hodnotu, kterou vynesu do grafu.



Na grafu můžeme vidět průběh zvyšující se časové náročnosti výpočtu. Na vodorovné ose je vyznačen rozměr krychle a na svislé je v logaritmickém měřítku vyznačena doba běhu v sekundách.

Při pohledu na graf můžeme odhadnout jak bude křivka dále pokračovat, ale nelze přesně určit jaké matematické funkci se blíží. Teoreticky by měly mít oba algoritmy kubickou časovou složitost a s tímto tvrzením by se dalo podle grafu nejspíše souhlasit. Taky je patrné, že Dijkstrův algoritmus je implementován lépe než Floyd-Warshallův, protože se zvyšující složitostí grafu se křivky mírně rozjíždějí.

5 Závěr

Problém hledání všech nejkratších cest se na první pohled nezdá být příliš složitý. Pokud se však nad ním zamyslíme hlouběji a zhodnotíme existující algoritmy, dospějeme k závěru, že upravit je tak, aby měly třídu asymptotické časové složitosti nižší než kubickou, pravděpodobně není možné.

Dijkstrův algoritmus je navržen pro hledání nejkratších cest v hranově ohodnocených grafech. Avšak v zadání bylo požadováno hledání v neohodnocených grafech a nalezení všech nejkratších cest mezi všemi vrcholy. Abychom nám algoritmus pomohl nalézt nejkratší cestu i v hranově neohodnoceném grafu, můžeme využít následujícího postupu. Vytvoříme si hranově ohodnocený graf izomorfní s grafem neohodnoceným. Každá hrana ohodnoceného grafu ponese stejnou hodnotu (např. 1). Na takovýto graf Dijkstrův algoritmus lze aplikovat a nalezne nám nejkratší vzdálenosti spolu s jednou cestou. To ovšem nevyhovuje zadání, které hovořilo o nalezení všech nejkratších cest mezi dvěma vrcholy. Proto jsem byl nucen algoritmus upravit. Nejkratší cesty se ukládají pomocí odkazu na předchůdce v cestě. Rekonstrukce cesty je zpětným průchodem od koncového vrcholu k jeho předchůdci, dále od předchůdce k jeho předchůdci až nakonec dojdeme k počátečnímu vrcholu. Úprava spočívá v nahrazení odkazu na předchůdce za pole odkazů na předchůdce. Pokud jsme se do bodu C dostali z bodu B stejně dlouhou cestou jako již dříve z bodu A, nezměněný algoritmus toto vyhodnotí jako případ, kdy není nalezena kratší cesta než aktuální a nebere ji v úvahu. Pozměněný ji v úvahu bere a uloží odkaz na vrchol B do pole předchůdců vedle odkazu na vrchol A. Takto se dají velmi efektivně uložit všechny nejkratší cesty mezi počátečním vrcholem a všemi ostatními.

Další změnou oproti běžné implementaci Dijkstrova algoritmu je vytvoření rekurzivní funkce starající se o vypsání nejkratších cest. Ukázalo se, že výpis cest na standardní výstup je řádově časově náročnější než samotný výpočet. Pro grafy o vyšší složitosti (zhruba nad 256 vrcholů se stupněm vrcholu 8) je pravděpodobně lepší výsledky ukládat přímo do souboru a nevypisovat je na standardní výstup. Přesměrování výstupu do souboru je pomalejší než přímý zápis.

U Floyd-Warshallova algoritmu jsem v původní implementaci ukládal do matice sousednosti pro každé dvojice vrcholů cesty, které jsem mezi nimi nalezl. To se ukázalo jako krok špatným směrem, protože počet nejkratších cest u složitějších grafů výrazně stoupá. Bylo nutné najít jiné řešení. Pak jsem si uvědomil, že není potřeba uchovávat všechny cesty, ale pouze délky nejkratších cest. Způsob jakým bychom mohli získat z matice sousednosti, ve které jsou zaznamenány délky nejkratších cest, vrcholy obsažené v cestách je následující. Vytvoříme si rekurzivní metodu, která vybere ze sousedů počátečního vrcholu takové, kteří mají nejkratší cestu ke koncovému vrcholu, a zařadí je do zásobníku. V dalším kroku vybere sousedy prvního vrcholu v zásobníku, kteří mají

nejkratší cestu ke koncovému vrcholu, a také je zařadí do zásobníku. Po dosažení koncového vrcholu je cesta vypsána a ze zásobníku je vybrán další vrchol a takto až do doby než je zásobník prázdný. Jedná se o způsob prohledávání do hloubky (DFS).

Paměťová náročnost se po úpravě algoritmu, aby hledal jen hodnoty nejkratších cest a nezabýval se ukládáním cest samotných, významně snížila. Původně implementovaný Floyd-Warshallův algoritmus pracovat s přiměřenými paměťovými nároky u grafů přibližně o 32 vrcholech. Druhá upravená implementace Floyd-Warshallova algoritmu je na množství paměti mnohem méně závislá. U té nastává stejná situace jako u Dijkstrova algoritmu, kdy jsme schopni nalézt všechny nejkratší cesty poměrně rychle, ale problematické je jejich předání na výstup kvůli jejich množství.

U Dijkstrova algoritmu jsem se pokoušel o jeho vícevláknovou implementaci, ale kvůli časové tísni a malým zkušenostem v této oblasti jsem nebyl úspěšný. Nicméně bylo by zajímavé sledovat jaký vliv na rychlost má přizpůsobení algoritmu na více výpočetních jednotek. V tomto by mohl pokračovat další vývoj projektu.

Bellman-Fordův algoritmus jsem se implementovat nesnažil. Oproti Dijkstrovu algoritmu se liší schopností pracovat na grafech se záporně ohodnocenými hranami bez záporných smyček. Bylo proto zbytečné využívat tento pomalejší algoritmus, když Dijkstrův nebo Floyd-Warshallův algoritmus zvládnou zadanou úlohu v kratším čase.

V budoucnu budou aplikace pro hledání nejkratších cest v grafu pravděpodobně stále založené na jednom z těchto tří algoritmů, ale budou implementovány s ohledem na paralelní zpracování.

Literatura

[1] Kovár, M. Diskrétní matematika, Brno, 2002-2003, s. 87-113.

[2] Herout, P. Učebnice jazyka Java, 2. vydání, České Budějovice, Kopp 2006.

[3] Wikipedia.org, Dokument dostupný na http://wikipedia.org/.

[4] Honzík, J. M., Hruška, T., Máčel, M. Vybrané kapitoly z programovacích technik, Vysoké učení technické v Brně, Brno, 1991. ISBN 80-214-0345-4.

Seznam příloh

Příloha 1. CD s elektronickou verzí této práce, zdrojovými kódy vytvořeného programu a několika vstupními soubory, ve kterých jsou uloženy data o struktuře grafu.