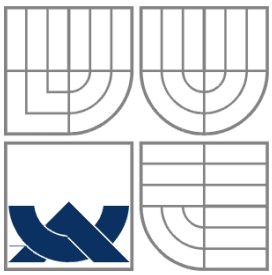
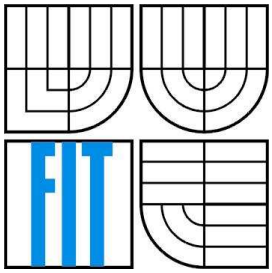


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

NEJKRATŠÍ CESTY V GRAFU

SHORTEST PATHS IN A GRAPH

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MICHAL KRAUTER

VEDOUCÍ PRÁCE
SUPERVISOR

RNDr. TOMÁŠ MASOPUST, Ph.D.

BRNO 2009

Abstrakt

Tato práce se zabývá problematikou nejkratších cest v grafu. Hledání těchto cest patří mezi základní problémy teorie grafů s četnými praktickými aplikacemi. Problém hledání nejkratších cest lze rozdělit na dvě skupiny. V první z nich hledáme nejkratší cesty z jednoho konkrétního uzlu do všech ostatních uzlů a v druhé hledáme nejkratší cesty mezi všemi páry vrcholů grafu. U každé skupiny jsou v textu uvedeny principy a algoritmy, které problém řeší. Studovány a popsány jsou jak klasické, tak i nové efektivnější metody. Z každé skupiny jsou vybrány, implementovány a experimentálně porovnány některé algoritmy pro hledání nejkratších cest v grafu.

Klíčová slova

Nejkratší cesta, grafový algoritmus, nejkratší cesty mezi všemi vrcholy grafu, Dijkstrův algoritmus, Bellmanův-Fordův algoritmus, Floydův-Warshallův algoritmus.

Abstract

This thesis deals with shortest paths problem in graphs. Shortest paths problem is the basic issue of graph theory with many pracitcal applications. We can divide this problem into two following generalizations: single-source shortest path problem and all-pairs shortest paths problem. This text introduces principles and algorithms for generalizations. We describe both classical and new more efficient methods. It contains information about how some of these algorithms were implemented and offers an experimental comparison of these algorithms.

Keywords

Shortest paths, graph algorithms, single-source shortest path problem, all-paris shortest path problem, Dijkstra’s algorithm, Bellman-Ford's algorithm, Floyd-Warshall’s algorithm.

Citace

Krauter Michal: Nejkratší cesty v grafu, diplomová práce, FIT VUT v Brně, Brno, 2009.

Nejkratší cesty v grafu

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením RNDr. Tomáše Masopusta, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Michal Krauter
2009-01-27

Poděkování

Děkuji tímto vedoucímu své diplomové práce, panu RNDr. Tomáši Masopustovi, Ph.D., za odborné vedení, užitečné rady, podnětné připomínky a konzultace při tvorbě této práce.

Obsah

1 Úvod

Informatika může jen těžko existovat bez pojmů a postupů teorie grafů, neboť často vyjadřujeme vztahy mezi nějakými objekty pomocí grafu. Grafy a grafové algoritmy prostupují nejen teoretickými základy oboru informatika, ale jejich používání tvoří neodmyslitelnou součást i tak prakticky zaměřených oblastí, jakou jsou např. programovací techniky nebo počítačové sítě. Formulací nějakého problému v pojmech teorie grafů totiž získáme velmi názorný matematický model, pro který je většinou možné posoudit existenci řešení problému a náročnost jeho výpočtu. Současně dává možnost využít k vlastnímu řešení některého z existujících grafových algoritmů nebo jeho vhodné modifikace. Nabízí se nepřeberné množství postupů při řešení problémů využívající právě grafy.

Tak třeba silniční síť lze znázornit pomocí grafu tak, že křižovatky odpovídají vrcholům grafu a silnice mezi nimi hranám. Každý úsek silnice odpovídající hraně má svou délku. Můžeme tedy položit následující otázky. Jaká je nejkratší cesta z Brna do Prahy? Kolik má kilometrů a kudy vede?

Problém hledání nejkratších cest je jedním z nejzákladnějších problémů síťové optimalizace. Zásadně se rozvíjí v mnoha síťových optimalizačních algoritmech a je již více než čtyři desetiletí studován. Problém hledání nejkratší cesty v grafu a jemu podobné problémy se vyskytují téměř všude: při směrování paketů na internetu, při hledání dopravního spojení mezi dvěma místy, při plánování pohybu robota, při směně mezi jednotlivými měnami, při operačním výzkumu, při aproximaci lineární funkce atd.

V následujících kapitolách prostudujeme základní algoritmy řešící problém hledání nejkratších cest a provedeme přehled nových a efektivnějších metod. Na téma porovnávání těchto algoritmů již bylo v minulosti provedeno několik studií – např. [12],[13],[21] a [41], ze kterých budeme taktéž čerpat.

Ve druhé kapitole se seznámíme s některými základními pojmy, které jsou nezbytné pro další studium dané problematiky.

Ve třetí kapitole probereme krátce možnosti reprezentace grafů v počítači. Probereme si u každé možné reprezentace výhody a nevýhody a stanovíme formát, který poté použijeme pro jednotlivé algoritmy.

Ve čtvrté kapitole si zavedeme pojem vzdálenosti na grafu.

V následující kapitole uvedeme problém nejkratších cest a provedeme jeho rozdělení na dvě hlavní skupiny.

V šesté kapitole ukážeme základní algoritmy řešící problém nejkratších cest z jednoho daného vrcholu do všech ostatních a vytvoříme přehled novějších a efektivnějších metod. Z nich pak tři algoritmy experimentálně porovnáme.

V sedmé kapitole se zaměříme na druhou skupinu algoritmů, a to na ty řešící problém nejkratších cest mezi všemi páry vrcholů. Rovněž popíšeme základní algoritmy a provedeme rešerši nových a efektivních metod. Ke konci kapitoly pak tři algoritmy experimentálně porovnáme.

Diplomová práce nenavazuje na bakalářskou práci ani na semestrální projekt, který byl na jiné téma obhájen před rokem.

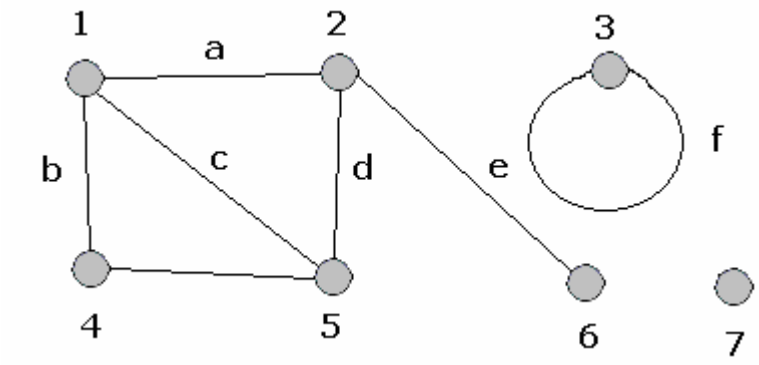
2 Základní pojmy

Předpokládám čtenářovu znalost základních matematických pojmů jako je množina, relace, kartézský součin, atd., které v textu již nebudu definovat a seznámit se s nimi lze ve většině učebnic z přehledu - např. [2],[4],[6] nebo [9]. V této kapitole si objasníme pár pojmů z teorie grafů, které jsou pro další studium problematiky nepostradatelné. Pokud by se čtenáři zdál výklad nedostatečný, lze se obrátit na některou z učebnic teorie grafů, např. na knihu J.Nešetřila [8].

2.1 (Neorientovaný) graf

V literatuře je možné se setkat s různými variantami definice pojmu graf. Neformálně se **graf** skládá z tzv. vrcholů a tzv. hran. Hrana vždy spojuje dva vrcholy a je buď orientovaná, nebo neorientovaná. U hran orientovaných rozlišujeme počáteční a koncový vrchol. Neorientované hrany chápeme jako symetrické spojení dvou vrcholů. Orientovaný graf má všechny hrany orientované, neorientovaný graf má všechny hrany neorientované. A teď trochu formálněji.

Neorientovaným grafem nazýváme uspořádanou trojici G = <H, U, ρ>, prvky konečné množiny H nazýváme **hranami** grafu G, prvky konečné množiny U **uzly** grafu G a zobrazení ρ **incidencí** grafu G. Role incidence ρ grafu spočívá v tom, že přiřazuje každé jeho hraně neuspořádanou dvojici uzlů: Je-li ρ(*h*) = [*u*, *v*], nazýváme uzly *u*, *v krajními uzly* hrany *h*. O hraně *h* říkáme, že *inciduje* s uzly *u*, *v* (spojuje uzly *u* a *v*). Zvláštní případ mezi hranami představují tzv. **smyčky**, pro které je ρ(h) = [*u*, *u*] (viz např. hrana f na obr. 2.1). **Izolovaným uzlem** grafu nazýváme takový uzel, s nímž neinciduje žádná hrana (např. uzel 7 na obr. 2.1).



Obrázek 2.1: Neorientovaný graf

Přívlastek „neorientovaný“ často vynecháváme, bude-li to z kontextu jasné. Někdy se setkáváme i s definicí grafu, v níž se vypouští incidence. Za hrany se v takové definici považují přímo neuspořádané dvojice uzlů. Každá dvojice uzlů pak může určovat pouze jedinou hranu. Toto omezení nebude na závadu u grafů, v nichž nepřipouštíme existenci tzv. **rovnoběžných hran** (tzn. hran se shodnou množinou krajních uzlů) - takové grafy se nazývají **prosté**. Naopak graf, ve kterém existuje alespoň jedna dvojice rovnoběžných hran, budeme nazývat **multigrafem**. U prostých grafů je tedy možné ztotožnit hrany s odpovídajícími neuspořádanými dvojicemi krajních uzlů - incidence je

potom zřejmá a není třeba ji v zápisu prostého grafu uvádět. Další druh grafů lze získat tak, že zakážeme existenci smyček - prosté grafy bez smyček budeme nazývat **obyčejnými grafy**.

Úplným grafem nazýváme obyčejný graf, který má *n* uzlů a právě (n nad 2) hran, neboť to je počet různých dvouprvkových podmnožin jeho uzlů, a každé dva různé uzly jsou v něm spojeny hranou. Přidáním libovolné hrany mezi dvěma různými uzly úplného grafu již tedy nutně dostaneme multigraf.

2.2 Podgraf

Graf G' = <H', U', ρ'> nazýváme **podgrafem** grafu G = <H, U, ρ> (zapisujeme G' $\subseteq$ G), jestliže platí:

$$(H' \subseteq H) \,\&\, (U' \subseteq U) \,\&\, \forall h \in H'(\rho'(h) = \rho(h)). \qquad \textbf{(2.2.1)}$$

2.3 Množina sousedů

Nechť G = <H, U, ρ> je graf, $u \in U$ libovolný uzel a A $\subseteq$ U libovolná podmnožina uzlů. **Množinou sousedů** Γ(*u*) uzlu *u* nazýváme podmnožinu uzlů definovanou vztahem:

$$\Gamma(u) = \{v \in V: \exists h \in H: \rho(h) = [u, v]\}. \qquad \textbf{(2.3.1)}$$

2.4 Stupeň uzlu

Nechť G = <H, U, ρ> je graf, $u \in U$ libovolný uzel. **Stupněm** $\delta_G(u)$ uzlu *u* v grafu G nazýváme počet hran s ním incidujících. Symboly δ(G) a Δ(G) označujeme minimální, resp. maximální stupeň uzlu v grafu G.

2.5 Sled

Představme si nyní následující procházku po nějakém grafu: vyjdeme z určitého uzlu grafu a po nějaké hraně s ním incidující přejdeme do uzlu sousedního, odtud po další hraně do nového uzlu atd., až se dostaneme k nějakému cílovému uzlu (popř. nazpět do uzlu výchozího). Posloupnosti hran a uzlů, po nichž budeme takto postupně přecházet, hrají v teorii grafů velmi důležitou roli, takže nyní podáme jejich přesnou definici.

Nechť pro danou dvojici uzlů u a v grafu G = <H, U, ρ> existuje posloupnost uzlů a hran:

$$S = \langle u_0, h_1, u_1, h_2, ..., u_{n-1}, h_n, u_n \rangle, \qquad \textbf{(2.5.1)}$$

kde $h_i \in H$, $\rho(h_i) = [u_{i-1}, u_i]$ pro i = 1, 2, ..., n, $u_i \in U$ pro i = 0, 1, ..., n, $u_0 = u$, $u_n = v$. Pak tuto posloupnost nazýváme **sledem** grafu G mezi uzly *u* a *v*. Uzly *u*, *v* jsou krajní uzly sledu S (*u* je počáteční, *v* koncový uzel), uzly $u_1, u_2, ..., u_{n-1}$ jsou vnitřní uzly sledu S. Číslo n (≥0) nazýváme **délkou sledu** S a značíme d(S). Sled s alespoň jednou hranou, v němž jsou uzly *u* a *v* shodné, nazýváme uzavřeným, ostatní sledy (včetně sledů nulové délky) nazýváme otevřenými. Při určení sledu často stačí zadat pouze příslušnou posloupnost hran $<h_1, h_2, ..., h_n>$, u prostého grafu můžeme naopak použít posloupnost uzlů $<u_0, u_1, u_2, ..., u_n>$.

Ze všech možných sledů mezi danými dvěma uzly jsou nejdůležitější takové, které neprocházejí opakovaně žádnou hranou grafu a popř. ani uzlem grafu - pro ně zavedeme zvláštní názvy následující definicí. Mezi nimi můžeme totiž např. hledat ty sledy, které mají nejmenší délku (při pevně zvolené dvojici krajních uzlů). Tak se dostáváme k otázce zavedení vzdálenosti na grafu, které se podrobně věnujeme v kapitole 4.

2.6 Tah, cesta a kružnice

Tahem grafu G nazýváme takový jeho sled, v němž jsou všechny hrany různé. **Cestou grafu** G nazýváme takový jeho tah, v němž každý uzel inciduje nejvýše se dvěma hranami tohoto tahu. **Kružnicí** nazýváme uzavřenou cestu.

Z každého otevřeného sledu $S = <h_1, h_2, ..., h_n>$ mezi uzly *u* a *v* neorientovaného grafu lze vybrat cestu spojující tytéž dva uzly. Pokud S není cestou, musí procházet některým ze svých vnitřních uzlů *x* opakovaně. Potom je ale možné vypustit z S všechny hrany nacházející se mezi prvním a druhým průchodem uzlem *x* - tak dostaneme sled S o menší délce, který spojuje stále tytéž uzly *u* a *v*. Je-li S cestou, jsme hotovi; jinak proces opakujeme od začátku. Po konečném počtu kroků nutně dospějeme k cestě, neboť délka sledu S se stále zmenšuje a sled délky 1 je vždy cestou.

Velmi důležitou vlastností grafů je propojitelnost různých dvojic uzlů pomocí sledů (a tedy pomocí cest). Pro vyjádření této vlastnosti zavedeme pojem souvislosti grafu. **Souvislým grafem** nazýváme takový neorientovaný graf, mezi jehož libovolnými dvěma uzly existuje sled.

2.7 Orientovaný graf

Hrany v neorientovaném grafu jsou obousměrné - můžeme jimi procházet v obou možných směrech. V některých aplikacích je tato vlastnost hran na závadu. Ve vývojovém diagramu musí být hrany orientovány, aby mohly jednoznačně vyjádřit časovou následnost jednotlivých operací. V elektrickém obvodu nám zase orientace hran naznačí předpokládaný směr napětí. Ve schématu silniční sítě nám orientace zachycuje přípustný směr průjezdu ulicí atd. Vhodným modelem je pak orientovaný graf.

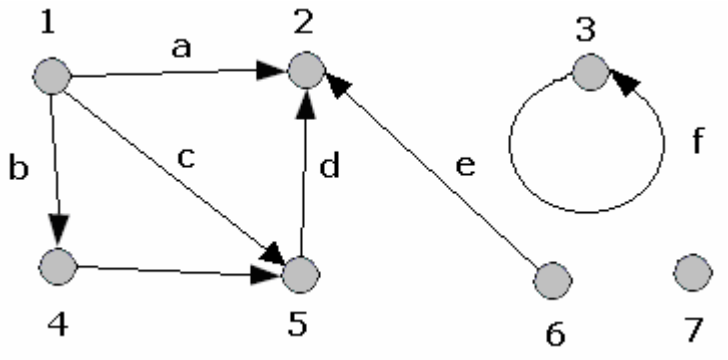
Ve znázornění grafu se orientace hran vyjádří pomocí šipek, ve formální definici představuje orientaci hran uspořádání krajních uzlů - hrana již neinciduje s neuspořádanou, ale s uspořádanou dvojicí uzlů. Doplnění orientace hran graf jistě obohacuje - oproti neorientovanému grafu je možno zavést mnoho nových pojmů, řešit nové druhy problémů, a rozšířit tak možnosti přímé aplikace teorie grafů.

Orientovaným grafem nazveme uspořádanou trojici $G = <H, U, \sigma>$, prvky konečné množiny H nazýváme **orientovanými hranami** grafu G, prvky konečné množiny U **uzly** grafu G a zobrazení $\sigma : H \rightarrow U \times U$, které nazýváme **incidencí** grafu G. Toto zobrazení přiřazuje každé hraně uspořádanou

dvojici vrcholů. Jestliže pro $h \in H$ je $\sigma(h) = (u, v)$, nazýváme uzel *u počátečním* a uzel *v koncovým uzlem* hrany *h*, rovněž říkáme, že hrana *h* je *orientována* od uzlu *u* k uzlu *v*. Existenci hrany z *u* do *v* vyjadřujeme také zápisem $u\Gamma v$, uzel *u* nazýváme předchůdcem uzlu *v* nebo obdobně uzel *v* následníkem uzlu *u*. Formálně lze relaci následování na množině uzlů orientovaného grafu zavést vztahem:

$$u\Gamma v \Leftrightarrow_{df} \exists h \in H: \sigma(h) = (u, v). \qquad \textbf{(2.7.1)}$$

Orientované hrany h_1, h_2 nazýváme **rovnoběžnými**, když platí $\sigma(h_1) = \sigma(h_2)$, neboli když h_1, h_2 mají stejné počáteční a stejné koncové uzly. Není-li třeba rozlišovat pořadí uzlů, které hrana *h* spojuje, nazýváme je jako u neorientovaných grafů krajními uzly hrany *h*. Je-li $\sigma(h) = (u, u)$, nazýváme hranu *h* **orientovanou smyčkou** (přívlastek „orientovaná“ u hran, smyček, apod. budeme dále vynechávat, bude-li z kontextu jasné, že se jedná o orientovaný graf).



Obrázek 2.2: Orientovaný graf

Podobně jako u neorientovaných grafů rozlišujeme podle výskytu rovnoběžných hran **orientované multigrafy** a **prosté orientované grafy**. U prostých orientovaných grafů lze opět za množinu hran považovat přímo odpovídající podmnožinu kartézského součinu $H \subseteq U \times U$, a incidenci σ můžeme pak z vyjádření grafu vypustit. V tomto případě se množina hran shoduje s relací následnosti Γ orientovaného grafu.

Úplným **symetricky orientovaným grafem** nazýváme prostý orientovaný graf $K_U = <U \times U, U>$, tzn. graf, v němž je každá uspořádaná dvojice uzlů spojena orientovanou hranou.

Abychom mohli co nejvíce využít analogických pojmů pro přenášení výsledků mezi oběma typy grafů, ukážeme nejprve základní možnost přechodu mezi orientovaným a neorientovaným grafem, při němž se fakticky nezmění struktura grafu. Z orientovaného grafu $G = <H, U, \sigma>$ můžeme získat odpovídající neorientovaný graf prostě tak, že zrušíme orientaci hran, podobně můžeme z neorientovaného grafu $G = <H, U, \rho>$ získat orientovaný graf tak, že každou z jeho hran určitým způsobem orientujeme. Těmto přechodům budeme říkat **zrušení**, resp. **zavedení orientace** v grafu a jejich přesnou definici zde uvádět nebudeme.

Jinou možností přechodu od neorientovaného grafu k orientovanému při zachování jeho charakteru s ohledem na propojitelnost uzlů je tzv. **symetrická orientace neorientovaného** grafu: Pro každou hranu *h*, $\rho(h) = [u, v]$, $u \neq v$ vytvoříme dvojici opačně orientovaných hran h' : $\sigma(h') = (u, v)$ a h'': $\sigma(h'') = (v, u)$. Případné smyčky pouze orientujeme, tzn. pro $\rho(h) = [u, u]$ vytvoříme jedinou hranu h' : $\sigma(h') = (u, u)$.

Orientovaný graf $G_2 = <H, U, \sigma_2>$ nazýváme **opačně orientovaným** vzhledem k orientovanému grafu $G_1 = <H, U, \sigma_1>$ (zapisujeme $G_2 = \bar{G_1}$), pokud pro incidence σ_1 a σ_2 platí vztah

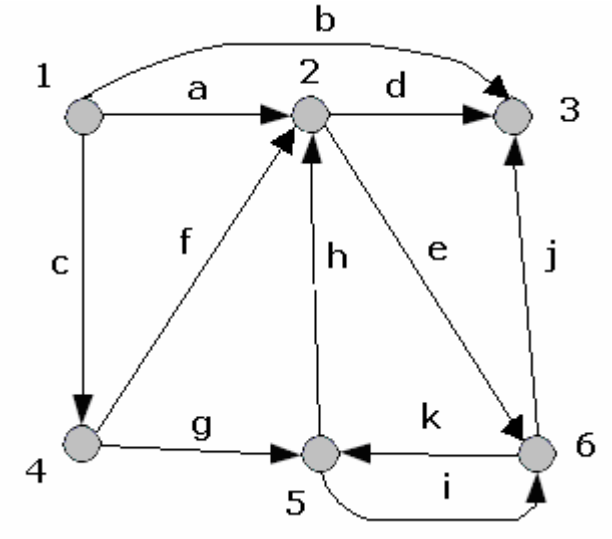
$$\sigma_1(h) = (u, v) \Leftrightarrow \sigma_2(h) = (v, u) \text{ pro každou hranu } h \in H. \qquad \textbf{(2.7.2)}$$

2.8 Orientovaný sled, spojení

Využijeme nyní možnosti jednoznačného přechodu od orientovanému grafu k neorientovanému prostřednictvím zrušení orientace k tomu, abychom zavedli pojmy vyjadřující propojitelnost uzlů bez ohledu na orientaci hran.

Sledem orientovaného grafu G nazýváme takovou posloupnost uzlů a hran, která je sledem v neorientovaném grafu G' vzniklém zrušením orientace grafu G.

Analogicky zavedeme i speciální druhy sledů - **uzavřený sled, tah, cesta** a **kružnice orientovaného grafu**. Tímto jsme pouze vytvořili předpoklady pro využití základních pojmů a poznatků o neorientovaných grafech pro grafy orientované. Nyní zavedeme a na grafu z obr. 2.3 ilustrujeme nové specifické pojmy, ve kterých se bude orientace hran odrážet.



Obrázek 2.3: Spojení a cykly

Nechť pro danou dvojici uzlů *u* a *v* orientovaného grafu G = <H, U, σ> existuje posloupnost uzlů a hran:

$$S = \langle u_0, h_1, u_1, h_2, ..., u_{n-1}, h_n, u_n \rangle, \qquad \textbf{(2.8.1)}$$

kde $h_i \in H$, $\sigma(h_i) = (u_{i-1}, u_i)$ pro i = 1, 2, ..., n, $u_i \in U$ pro i = 0, 1, ..., n, $u_0 = u$, $u_n = v$. Pak tuto posloupnost nazýváme **spojením grafu** G z uzlu *u* do uzlu *v*. Uzel *u* je počátečním, uzel *v* koncovým uzlem spojení S, číslo n (≥ 0) nazýváme délkou spojení. Skutečnost, že existuje orientované spojení z *u* do *v*, budeme též stručně vyjadřovat zápisem $u \rightarrow v$.

Rozdíl mezi sledem a spojením orientovaného grafu spočívá tedy v tom, že u spojení se požaduje souhlasná orientace všech jeho hran ve směru od počátečního ke koncovému uzlu. Pro orientovaný graf na obr. 2.3 např. sledy (opět zapisované jen jako posloupnosti hran) S_1 = <a, d, j> a S_2 = <c, g, k>, nejsou spojeními, zatímco sledy S_3 = <a, e, k, i> a S_4 = <c, g, i> jsou spojeními z uzlu 1 do uzlu 6.

2.9 Orientovaný tah, cesta a cyklus

Orientovaným tahem, resp. **orientovanou cestou** nazýváme takové spojení, které je tahem, resp. cestou po zrušení orientace. Uzavřenou orientovanou cestu nazýváme **cyklem**. Stejně jako u neorientovaných grafů považujeme i orientovanou cestu nulové délky za otevřenou, cyklus tedy musí obsahovat alespoň jednu hranu. V orientovaném grafu na obr. 2.3 jsou dříve uvedená spojení S_3 a S_4 orientovanými tahy, S_4 je dokonce orientovanou cestou. Cykly tohoto grafu jsou např. spojení S_5 = <e, k, h> a S_6 = <k,i>.

Je zřejmé, že z každého spojení z uzlu *u* do uzlu *v* lze vybrat orientovanou cestu mezi těmito uzly (obdobně jako u sledu v neorientovaném grafu). Mezi sledy a spojeními jsou však také dosti významné rozdíly: není např. možné tvrdit, že mezi dvěma uzly orientovaného grafu neexistuje buď vůbec žádné, nebo nekonečně mnoho spojení. Počet různých spojení v orientovaném grafu může totiž být konečný, přestože tento graf obsahuje nekonečně mnoho různých sledů. Propojitelnost uzlů pomocí spojení lze vyjádřit novým druhem souvislosti, specifickým pro orientované grafy. Orientovaný graf G nazýváme **silně souvislým**, jestliže pro libovolnou dvojici uzlů *u*, *v* existuje spojení z uzlu *u* do uzlu *v* a spojení z uzlu *v* do uzlu *u* (tedy $u \rightarrow v$ a $v \rightarrow u$). Graf znázorněný na obr. 2.3 je sice souvislý, ale není silně souvislý, neboť např. z uzlu 5 nevede žádné spojení do uzlu 4.

2.10 Strom

Z aplikačního hlediska patří stromy k nejpopulárnějším pojmům teorie grafů. **Stromem** je každý souvislý graf bez kružnic. Pro každé dva vrcholy stromu existuje právě jedna cesta, která je spojuje. V programování jsou stromy považovány za velmi důležitý druh datových struktur. Pomocí stromů můžeme také přehledně vyjádřit systematický postup probírání všech možností při řešení nějaké (grafové) úlohy. Např. strom všech maximálních cest grafu G , které vycházejí z uzlu *x*.

2.11 Kořenový strom

Za orientovaný strom obecně považujeme jakýkoliv orientovaný graf, z něhož vznikne zrušením orientace strom. Jako negativní příklad můžeme uvést tvrzení v neplatném orientovaném znění:

Graf G je orientovaným stromem právě tehdy, když mezi každými jeho dvěma uzly existuje právě jediná orientovaná cesta.

Abychom mohli požadovat existenci orientovaných cest, musíme jeden uzel orientovaného stromu odlišit od ostatních a spokojit se s existencí (jedinečné) orientované cesty z tohoto uzlu do libovolného jiného. Uzel, který byl zvolen za základ orientace, se nazývá **kořen**.

Kořenovým stromem s kořenem *u* nazýváme takový orientovaný strom T, ve kterém je každá cesta spojující uzel u s libovolným uzlem x orientovanou cestou. Délku cesty z kořene do uzlu x nazýváme **hloubkou uzlu** *x* v kořenovém stromu T a značíme $hl_T(x)$. **Hloubkou** hl(T) **stromu** T rozumíme maximální hloubku nějakého jeho uzlu.

Je zřejmé, že výběrem kořene *u* v neorientovaném stromu je jednoznačně určena odpovídající orientace hran v kořenovém stromu, a nemusíme ji proto na diagramech kořenových stromů znázorňovat. Pro vyjádření toho, že uzel *u* je kořenem kořenového stromu T, používáme označení T_u.

Každý uzel kořenového stromu kromě kořene má právě jednoho předchůdce, zatímco následníků může mít libovolný počet (včetně žádného). Tato vlastnost dokonce kořenové stromy plně charakterizuje. Vrcholy, které nemají následníka, se nazývají *koncové* nebo též *listy* a každý kořenový strom má alespoň jeden koncový uzel.

2.12 Orientované grafy a binární relace

Každý orientovaný graf určuje binární relaci následování Γ na množině svých uzlů. To lze ovšem chápat také obráceně - každá binární relace R na libovolné množině U může být vyjádřena pomocí prostého orientovaného grafu jako jeho relace následování. Stačí vzít přímo graf G_R = <R, U>, v němž jsou spojeny hranou ty uspořádané dvojice uzlů (*u*, *v*), které jsou v relaci R (tzn. pro něž platí *u*R*v*). Studium prostých orientovaných grafů se tedy kryje se studiem binárních relací. Ukážeme, jak lze pojmy týkající se binárních relací přenést na orientované grafy.

Vlastnosti binárních relací se na grafech projeví takto:

reflexivita - každý uzel má smyčku
tranzitivita - každá dvojice uzlů propojitelná spojením je spojena přímo hranou
symetrie - graf je symetricky orientován
antisymetrie - kromě případných smyček neexistují dvojice opačně orientovaných hran se stejnými krajními uzly
asymetrie - má-li graf hranu (*u*, *v*), pak již nesmí mít hranu (*v*, *u*)
ireflexivita - graf nemá žádnou smyčku

3 Reprezentace grafů

Řadu problémů v informatice formulujeme pomocí teorie grafů. Abychom tyto problémy mohli řešit na počítači, musíme mít k dispozici takovou reprezentaci grafu, která je vhodná pro počítačové zpracování. Mezi vhodné reprezentace z tohoto hlediska nemůžeme počítat znázornění grafu pomocí obrázku, které jsme dosud pro ilustraci pojmů teorie grafů používali. Struktura grafu je totiž zcela vyjádřena v jeho incidenci, nebo-li v určení krajních uzlů hran. Naproti tomu bitová mapa zachycující obrázek grafu je vedle dalších nevýhod i zbytečně paměťově náročná a pro reprezentaci jeho struktury nevhodná. Obsahuje totiž převážně neužitečná data o konkrétním tvaru a rozmístění grafických prvků vyjadřujících uzly a hrany grafu. Neznamená to však, že bychom při řešení grafových úloh nikdy neuvažovali vedle strukturní i případnou visuální podobu grafu. Tyto případy jsou však omezeny jen na speciální grafové aplikace. Visuální znázornění struktury grafu je především vhodné jako prostředek styku s uživatelem.

Existují dva druhy reprezentace grafu: **maticová a spojová**. Maticový popis grafů má spíše teoretický než-li praktický význam. Poskytuje totiž vazbu mezi teorií grafů a lineární algebrou, jež dává vedle možnosti charakterizovat vlastnosti grafů algebraickými prostředky i řadu zajímavých tvrzení. Různé varianty spojové reprezentace jsou standardním způsobem vyjádření grafu při realizaci grafových algoritmů. Oproti maticové formě mají jednak menší asymptotickou paměťovou složitost, ale především umožňují dosáhnout pro většinu řešených úloh i lepší časové složitosti algoritmů ve srovnání s maticovou reprezentací (více o možnostech reprezentace v [1]).

Maticovou reprezentaci neorientovaných grafů zde řešit nebudeme (lze najít v příslušné literatuře) a přejdeme rovnou na orientované grafy.

3.1 Matice orientovaných grafů

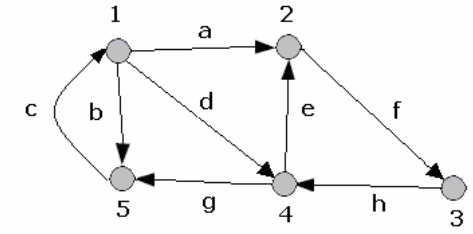
Mezi základní formy vyjádření struktury orientovaných grafů patří matice reprezentující buď incidenci σ, nebo relaci následování Γ grafu. Podáme nyní definici těchto matic pro orientované grafy a ukážeme jejich vlastnosti.

Nechť G = <H, U, σ> je orientovaný graf s množinou hran H = {h_1, h_2, ..., h_m} a množinou uzlů U = {u_1, u_2, ..., u_n}. **Maticí incidence** grafu G pak nazýváme celočíselnou matici **A** = [a_{ik}] typu (n, m), jejíž prvky jsou dány vztahy:

$$\begin{aligned} a_{ik} &= 1 \text{ pokud } \sigma(h_k) = (u_i, u_j) \text{ pro jisté } u_j \in U, \\ a_{ik} &= -1 \text{ pokud } \sigma(h_k) = (u_j, u_i) \text{ pro jisté } u_j \in U, \\ a_{ik} &= 0 \text{ v ostatních případech.} \end{aligned} \qquad \textbf{(3.1)}$$

Vytvoření matice incidence ilustrujeme na grafu z obr. 3.1. Matice **A** (v obrázku zapsaná schematicky formou tabulky) má tolik řádků, kolik má graf G uzlů, počet sloupců je roven počtu hran grafu G. Každý sloupec obsahuje jeden prvek 1 a jeden prvek -1, ostatní prvky jsou rovny nule, počet prvků 1 v i-tém řádku je roven $\delta_G^+(u_i)$, počet prvků -1 v i-tém řádku je roven $\delta_G^-(u_i)$. Každý sloupec matice **A** obsahuje právě dvě jedničky (jednu kladnou a jednu zápornou), neboť každá hrana inciduje se dvěma uzly (smyčky zde neuvažujeme). Tvar matice incidence záleží pochopitelně na pořadí, v jakém jsme očíslovali uzly a hrany grafu. Změna očíslování však způsobí pouze permutaci řádků

nebo sloupců matice **A**. Velmi snadno se z matice **A** orientovaného grafu získá matice **A** neorientovaného grafu vzniklého zrušením orientace: místo a_{ik} použijeme $|a_{ik}|$.



$A =$

	a	b	c	d	e	f	g	h
1	1	1	-1	1	0	0	0	0
2	-1	0	0	0	-1	1	0	0
3	0	0	0	0	0	-1	0	1
4	0	0	0	-1	1	0	1	-1
5	0	-1	1	0	0	0	-1	0

$V =$

	1	2	3	4	5
1	0	1	0	1	1
2	0	0	1	0	0
3	0	0	0	1	0
4	0	1	0	0	1
5	1	0	0	0	0

Obrázek 3.1: Matice incidence a sousednosti

Nechť G = <H, U, σ> je orientovaný graf s množinou uzlů U = $\{u_1, u_2, ..., u_n\}$. **Maticí sousednosti** grafu G nazýváme čtvercovou celočíselnou matici **V** = $[v_{ik}]$ řádu n, jejíž prvky jsou dány vztahem:

$$v_{ik} = |\sigma^{-1}(u_i, u_k)|. \qquad \textbf{(3.2)}$$

Prvek v_{ik} tedy určuje počet orientovaných hran vedoucích z u_i do u_k. V případě prostého grafu je matice **V** maticovým vyjádřením relace následnosti Γ grafu G, relaci předcházení Γ^{-1} vyjadřuje transponovaná matice $\mathbf{V}^T$. Matice sousednosti orientovaného grafu není obecně symetrická. Na obr. 3.1 je u grafu rovněž uvedena jeho matice sousednosti **V**. Na rozdíl od matice incidence nezachycuje matice sousednosti vzájemný vztah hran a uzlů grafu, dovoluje však vyjádřit případnou existenci smyček (nenulovými diagonálními prvky).

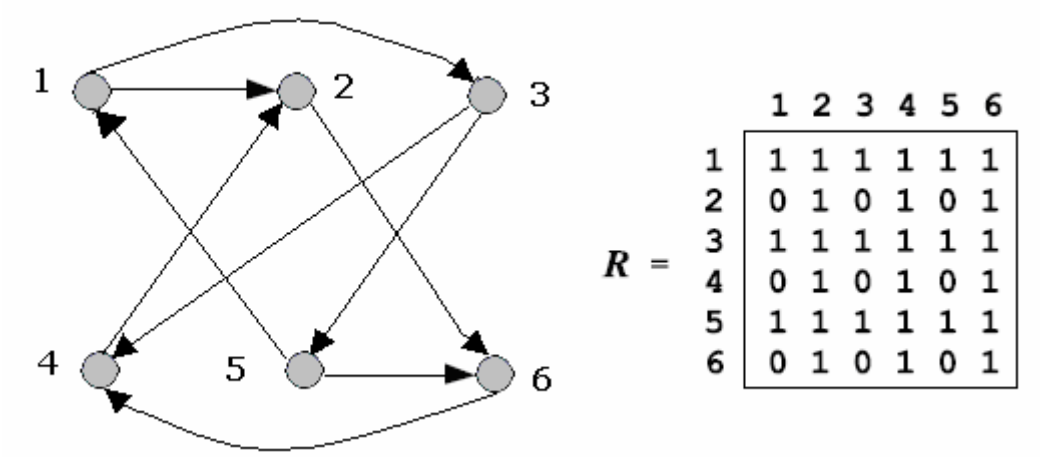
S použitím Booleových operací lze spočítat **matici dosažitelnosti R** jako maticovou reprezentaci reflexivně-tranzitivního uzávěru relace následnosti Γ. Pro prvky této matice platí:

$r_{ik} = 1$ pokud existuje spojení z uzlu u_i do u_k, **(3.3)**
$r_{ik} = 0$ v opačném případě.

Při výpočtu je nejlepší využít vztahu **R** = Γ* a počítat matici **R** jako matici reflexivně-tranzitivního uzávěru relace Γ pomocí Floydova-Warshallova algoritmu (uveden níže). Matice incidence **A** i matice sousednosti **V** poskytují o daném grafu téměř stejnou informaci, naproti tomu v matici dosažitelnosti **R** je obsažena pouze informace globálního charakteru, z níž se lokální vlastnosti grafu určit nedají.

Existují další druhy matic, které se používají k vyjádření struktury grafu. Více o nich lze nalézt v učebnicích teorie grafů nebo v [1].

Matice **V** a především pak matice **A** obsahuje pro grafy o větším počtu uzlů mnoho nulových prvků. Z toho plynou základní nevýhody maticového vyjádření grafu pro potřeby počítačového zpracování. Z velikosti matic **A** nebo **V** vyplývající paměťové nároky a především rozptýlený charakter informace, kterou je třeba používat při realizaci většiny operací a řešení typických úloh na grafech. Paměťové nároky je sice možné snížit na únosnou mez využitím binární povahy ukládaných údajů, tím se však dále komplikuje přístup k jednotlivým hodnotám (zato operace např. s celými řádky lze realizovat poměrně efektivně). Pro zadávání struktury grafu formou vstupních dat je maticové vyjádření rovněž nevhodné - je nepřehledné a nedovoluje snadnou kontrolu správnosti zadání.

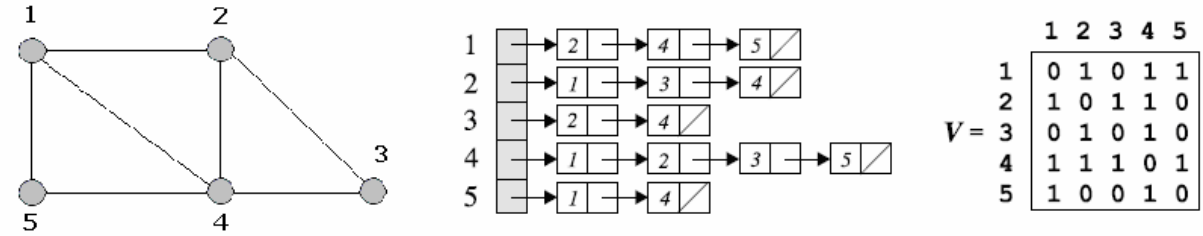


Obrázek 3.2: Matice dosažitelnosti

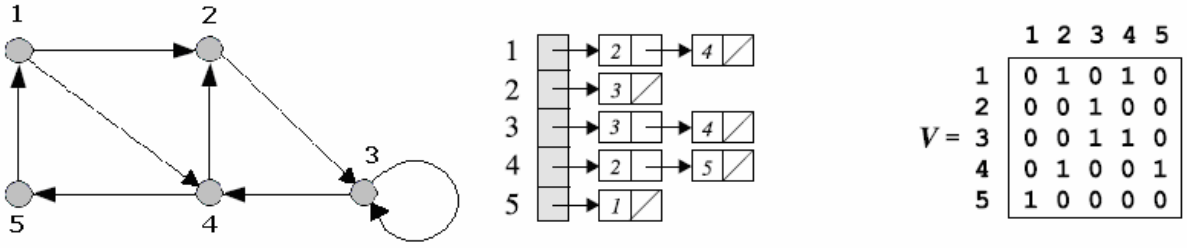
3.2 Spojová reprezentace grafu

Jako nevýhody maticové reprezentace grafu jsme uvedli paměťové nároky a rozptýlenost uložené informace. Z programovacích technik je dobře známa spojová metoda ukládání řídkých matic - jejími variantami jsou i spojové způsoby vyjadřování struktury grafu.

Základní forma spojové reprezentace neorientovaného grafu G = <H, U, ρ> se stává z pole *Adj* obsahujícího |U| odkazů (ukazatelů) na seznamy sousedů jednotlivých uzlů z množiny U. Pro každý uzel $u \in$ U obsahuje spojový seznam *Adj*[*u*] jeden záznam pro každou neorientovanou hranu *h* = [*u*, *v*] $\in$ H s druhým krajním uzlem *v*. Záznam typicky obsahuje identifikaci (číslo) uzlu *v*, případně identifikaci nebo doplňující údaje týkající se hrany *h*, pořadí záznamů je obecně libovolné.



Obrázek 3.3: Spojová reprezentace neorientovaného grafu



Obrázek 3.4: Spojová reprezentace orientovaného grafu

Na obr. 3.3 ukazujeme neorientovaný graf společně s jeho možnou spojovou reprezentací. Je vidět, že paměťová složitost spojové reprezentace je dána součtem délky pole ukazatelů (t.j. |U|) a celkovou délkou všech seznamů sousedů (t.j. 2|H|). V asymptotickém vyjádření to představuje $O(|U|+2|H|) = O(|U|+|H|) = O(\max(|U|, |H|))$.

Při přechodu k orientovaným grafům se na spojové reprezentaci téměř nic nezmění, záznamy ve spojovém seznamu však odpovídají orientovaným hranám vycházejícím z daného uzlu. To znamená, že pro každý uzel $u \in U$ obsahuje spojový seznam $Adj[u]$ jeden záznam pro každou orientovanou hranu $h = (u, v) \in H$ s koncovým uzlem v. Tento záznam může vedle identifikace uzlu v opět obsahovat i další údaje, např. délku hrany h pro hranově ohodnocené grafy, atd. Možnou spojovou reprezentaci orientovaného grafu ukazujeme na obr. 3.4. Každá hrana se nyní v seznamu následníků projeví právě jednou, což sice znamená menší skutečné paměťové nároky reprezentace, ale její asymptotická paměťová složitost bude stále $O(|U|+|H|)$ jako pro neorientované grafy (více viz [1]).

4 Vzdálenost na grafu

Co je to vzdálenost? Jak pro daný graf G a jeho dvojici uzlů *u*, *v* určíme vzdálenost d(*u*, *v*)? Právě těmto otázkám se budeme věnovat v této kapitole. Vzdálenost je matematicky popsána pojmem *metrika* (viz literatura). My si vystačíme se vzdáleností v grafech, tj. s grafovou metrikou. Začneme zobecněním samotného pojmu vzdálenosti, která bude definována prostřednictvím optimální (nejkratší) cesty mezi dvěma uzly, což značí, že se problematika vzdáleností kryje s určováním nejkratších cest.

Představme si dopravní plán města vyjádřený grafem. Jaká bude nejkratší trasa pro přejezd z místa A do místa B? Která část města je z daného místa nejobtížněji dostupná? Do kterého místa postavit požární zbrojnici, aby měla v případě požáru co nejvýhodnější umístění vůči všem částem města? Tyto a další podobné otázky vyúsťují zcela zákonitě v potřebu zavedení vzdálenosti na grafu (více viz [6]).

Vzdáleností uzlů *u* a *v* v neorientovaném souvislém grafu G = <H, U, ρ> nazýváme délku nejkratší cesty spojující tyto dva uzly; značíme ji $d_G(u, v)$. Většinou bude z kontextu jasné, v jakém grafu vzdálenost uvažujeme, takže budeme index G vynechávat a psát prostě d(*u*, *v*). Základní vlastnosti vzdálenosti shrnuje následující tvrzení.

Nechť G = <H, U, ρ> je neorientovaný souvislý graf. Potom pro libovolné jeho uzly *u*, *v*, *z* platí:

a) vzdálenost d(*u*, *v*) je celé nezáporné číslo;
b) d(*u*, *v*) ≥ 0; přitom d(*u*, *v*) = 0, právě když *u* = *v*; **(4.1)**
c) d(*u*, *v*) = d(*v*, *u*);
d) d(*u*, *v*) ≤ d(*u*, *z*) + d(*z*, *v*);
e) je-li d(*u*, *v*) > 1, pak platí: ∃z ∈ U(*u* ≠ *z* ≠ *v* & d(*u*, *v*) = d(*u*, *z*) + d(*z*, *v*)).

Vlastnosti b), c), d) jsou tzv. *axiómy metriky*, které by měla splňovat každá funkce určená k vyjadřování vztahů blízkosti a vzdálenosti na nějaké množině, vlastnosti a) a e) jsou specifické pro naši definici vzdálenosti na grafech (podrobněji v [6]).

4.1 Vzdálenost na orientovaných a ohodnocených grafech

Vzdálenost na orientovaných grafech má velmi mnoho společného se vzdáleností na grafech neorientovaných, je zde však nejméně jeden podstatný rozdíl - zřejmě není symetrickou funkcí svých argumentů. Při formulaci tvrzení obdobného jako 4.1 bychom tedy museli část c) vynechat. Splnění vlastnosti a) je podmíněno silnou souvislostí vyšetřovaného grafu. Zbývající vlastnosti b), d) a e) lze převzít beze změny.

Zavedeme pojem **vzdálenosti** d(*u*, *v*) z uzlu *u* do uzlu *v* v orientovaném grafu - definujeme ji opět jako délku (t.j. počet hran) nejkratší orientované cesty z *u* do *v*. Je zřejmé, že takto bude

vzdálenost definována pro všechny dvojice uzlů jen v případě silně souvislého grafu. Pokud v grafu žádná cesta z *u* do *v* neexistuje, můžeme považovat vzdálenost d(*u*, *v*) za nekonečnou.

Až dosud jsme při určování vzdálenosti považovali všechny hrany stejně dlouhé – takové pojetí je však pro mnoho aplikací nevyhovující. Je potřeba při zavedení vzdálenosti uvažovat možnost ohodnocení hran grafu reálnými čísly. Zavedeme tedy ještě další zobecnění vzdálenosti takto : Nechť G = <H, U, σ> je orientovaný graf s reálným ohodnocením hran *w*: H → **R**. Bez újmy na obecnosti budeme předpokládat, že G je prostý graf (jinak stačí vybrat z každé skupiny rovnoběžných hran tu, která je ohodnocena nejmenším číslem) a pro libovolné spojení S = <h_1, h_2, ..., h_k> grafu G definujeme jeho **w-délku** vztahem

$$d_w(S) = \sum_{i=1}^{k} w(h_i) \qquad \textbf{(4.1.1)}$$

W-vzdáleností $d_w(u, v)$ z uzlu *u* do uzlu *v* grafu G rozumíme nejmenší w-délku spojení z *u* do *v*, pokud takové spojení v grafu G existuje. Je-li uzel *v* z uzlu *u* nedostupný, pokládáme $d_w(u, v) = \infty$.

Takto pojaté chápání vzdálenosti nám dává větší aplikační možnosti. Ohodnocení hran w může mít v konkrétních aplikacích grafů různé interpretace: kromě skutečné délky nějaké křivky to může být např. časový údaj, cena, ztráta nebo jakákoliv jiná kvantita, jejíž úhrnné množství se získá součtem ohodnocení jednotlivých hran. Ve většině případů budou hodnoty w(*h*) nezáporné, apriori však nechceme vyloučit ani možnost záporného ohodnocení některých (nebo všech) hran grafu.

V takovém případě ale vzniká otázka, zda má předchozí definice vůbec smysl. Pokud má nějaká orientovaná cesta C z *u* do *v* neprázdný průnik s cyklem grafu G, který má zápornou w-délku, pak se s každým opakovaným zařazením tohoto cyklu do spojení s cestou C stále snižuje výsledná w-délka, takže minimální spojení neexistuje. V takových případech budeme pokládat $d_w(u, v) = -\infty$.

Všimněme si po tomto doplnění původních vlastností vzdálenosti (4.1) a zkusme, v jakých případech lze pro w-vzdálenost formulovat alespoň analogické vztahy, pokud by stejné vztahy neplatily.

a) **vzdálenost d(*u*, *v*) je celé nezáporné číslo** - tuto vlastnost lze zachovat jen při nezáporném celočíselném ohodnocení všech hran.

b) **d(*u*, *v*) ≥ 0; přitom d(*u*, *v*) = 0, právě když *u* = *v*** - tuto vlastnost lze zachovat jen při kladném ohodnocení všech hran.

c) **d(*u*, *v*) = d(*v*, *u*)** - se symetrií je třeba se u orientovaných grafů definitivně rozloučit.

d) **d(*u*, *v*) ≤ d(*u*, *z*) + d(*z*, *v*)** - trojúhelníková nerovnost platí beze změny i pro w-vzdálenosti, neboť je dána podmínkou minimality a součtovým charakterem určení w-délky cesty.

e) **je-li d(*u*, *v*) > 1, pak existuje uzel *z* ∈ U tak, že *u* ≠ *z* ≠ *v* a platí d(*u*, *v*) = d(*u*, *z*)+d(*z*, *v*)** - pro w-vzdálenost je možné vyslovit analogické tvrzení, je jen třeba upravit jeho předpoklad. Podmínka $d_w(u, v) > 1$ totiž nevyjadřuje to, co je pro důkaz zapotřebí, tedy existenci nějakého vnitřního uzlu na minimální cestě. Zobecněním vlastnosti e) dostaneme následující větu.

Nechť G = <H, U> je orientovaný graf s ohodnocením hran w : H → **R** a buď s jeho zvolený uzel. Potom pro každou hranu (*u*, *v*) ∈ H platí :

$$d_w(s, v) \le d_w(s, u) + w(u, v). \qquad \textbf{(4.1.2)}$$

5 Hledání nejkratší cesty

Nalezení nejkratší cesty je podproblém problému hledání optimální cesty, tj. nejlepší podle různých kritérií. Mezi další podproblémy patří hledání nejširší cesty, nejužší cesty, nejdelší cesty, nejspolehlivější a nejporuchovější cesty (popis jednotlivých problémů v [3]). Při řešení těchto úloh se používá tzv. Bellmanův princip optimálnosti, který říká, že každá část optimální cesty je optimální cestou (pokud vede z místa A do místa C optimální cesta přes místo B, musí být i cesta z A do B optimální). Optimální cesty jsou podmnožinou více obecné třídy problémů, a to problém obchodního cestujícího (TSP), vyžadující nalezení cesty přes každý vrchol v určitém pořadí tak, že začínáme a končíme v tom stejném uzlu, tak aby délka cesty byla minimální. Aplikace TSP zahrnují - prodavač navštěvující zákazníky, zásobování poboček z centrálního skladu, svoz odpadu, prohlídky budov bezpečnostní agenturou atd. a negeografické aplikace - od návrhu VLSI až po sekvenci DNA.

Hledáme-li nejkratší cesty v multigrafech, můžeme z každé množiny paralelních hran vynechat všechny kromě nejkratších z nich, aniž by to mělo vliv na délku nejkratší cesty. V kladně ohodnocených grafech lze vyloučit taktéž smyčky, neboť nemohou být obsaženy v žádné cestě.

V následujícím textu budeme převážně uvažovat orientované a ohodnocené grafy. Každá hrana bude ohodnocena nějakým reálným číslem. Nejlépe by bylo uvažovat jen nezáporně ohodnocené hrany, nicméně je vyloučit nemůžeme. Při určitých úlohách se nám záporně ohodnocené hrany hodí. Pojem nejkratší cesty lze brát jako zobecnění pojmu vzdálenosti. Za vzdálenost dvou uzlů považujeme délku nejkratší cesty mezi nimi ve smyslu součtu ohodnocení hran.

Pokud budou v grafu záporně ohodnocené hrany a z těch hran se někde podaří vytvořit záporně ohodnocený cyklus (v němž součet ohodnocení hran dá záporné číslo), vzniká problém, neboť průchodem cyklu se vzdálenost zmenšuje – což neodpovídá tomu co by jsme intuitivně předpokládali, neboť spojení přes cyklus lze libovolně zkrátit až do mínus nekonečna. Tuto situaci se budeme snažit detekovat.

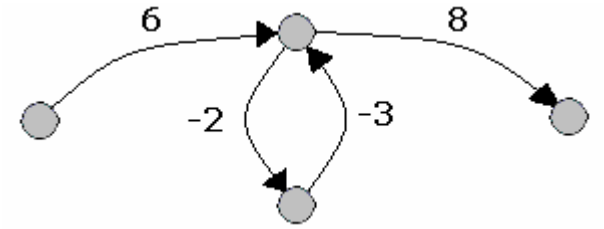
Až tak nám nebude vadit, pokud nějaké dva uzly spolu nebudou propojitelné. Pak jejich vzdálenost je $+\infty$. Jestliže z uzlu do uzlu neexistuje cesta, pak je lze považovat za nekonečně vzdálené. Naopak, když se nějaké spojení z *u* do *v* dotýká záporně ohodnoceného cyklu, tak jejich vzdálenost lze považovat za $-\infty$. Je třeba si definovat jak s těmi nekonečny budeme počítat.

$$\begin{aligned} &a + (\infty) = (\infty) + a = \infty;\ a \neq -\infty \quad a \text{ je libovolné konečné číslo} \\ &a + (-\infty) = (-\infty) + a = -\infty;\ a \neq \infty \\ &(\infty)+(-\infty) = \infty \end{aligned} \qquad \textbf{(5.1)}$$

5.1 Obecné ohodnocení hran

Jak jsme již naznačili v úvodu této kapitoly, při řešení problému nejkratších cest nám bude velké problémy způsobovat obecné ohodnocení hran. Některé algoritmy dávají správný výsledek pouze pro nezáporné ohodnocení hran. Proč tomu tak je? Buď to plyne z aplikace, kdy se v problému řešeném daným algoritmem záporné ohodnocení nevyskytuje, proto při vývoji algoritmů nebyla nutnost řešit záporné ohodnocení, nebo je to z důvodu rychlosti algoritmu. Ačkoliv víme o existenci záporně ohodnocených hran, nám jde především o rychlost výpočtu, pak se s tím musíme nějak vyrovnat - např. se nabízí možnost přehodnotit ohodnocení hran (viz další kapitoly).

Proč vůbec záporné ohodnocení, když to moc neodpovídá délce hran? Ohodnocení si lze představit i jako cenu, kterou musíme zaplatit za průchod hranou. Záporná cena znamená, že někdo zaplatí nám. Horší případ nastává, vyskytnou-li se v grafu záporné cykly. Průchodem tímto cyklem dokonce vyděláme a je výhodné po něm procházet stále dokola a postupně snižovat aktuální cenu spojení. Po nekonečně mnoha průchodech ji snížíme až na mínus nekonečno (nekonečně vyděláme). Pokud graf obsahuje záporný cyklus, tak optimální nejlevnější řešení neexistuje. Nalezneme sice nejkratší cestu, neboť v konečném grafu je jen konečně mnoho cest, nicméně taková cesta nebude nejlevnějším řešením dané úlohy.



Obrázek 5.1: Záporný cyklus

V průběhu výčtu algoritmů bude u každého uvedeno, zda lze nebo nelze použít pro záporně ohodnocené hrany.

5.2 Problémy hledání nejkratších cest

Máme celkem čtyři základní varianty problému minimálních cest.

a) Z jednoho konkrétního uzlu do jiného uzlu
b) Z jednoho konkrétního uzlu do všech ostatních uzlů
c) Ze všech uzlů do jednoho pevného uzlu
d) Ze všech uzlů do všech ostatních

Dá se ukázat, že první tři typy problémů jsou velice podobné, protože když hledám nejkratší cestu z jednoho uzlu do jiného konkrétního uzlu, tak musím nutně procházet přes nějaké další uzly a celkem se nám hodí nepřestat, když čirou náhodou najdu konkrétní cílový uzel a nestojí to už tak moc velkou námahu dojet a zjistit vzdálenosti a nejkratší cesty ke všem ostatním (samozřejmě nemusím, ale je to v zásadě stejný problém).

Následující text bude proto rozdělen na dvě části, v první budeme hledat nejkratší cesty z jednoho konkrétního uzlu do všech ostatních (s tím, že nám tato varianta řeší i problémy a) a c) a v kapitole 7 budeme hledat nejkratší cesty mezi všemi dvojicemi uzlů.

Mezi další varianty problému nejkratších cest patří např. určit nejkratší cestu procházející zadanými vnitřními uzly nebo nalézt prvních *k* nejkratších cest mezi dvěma uzly, atd. Někdy mohou být časové a paměťové nároky hledání nejkratší cesty vlivem složitosti grafu tak vysoké, že se spokojíme s mírně suboptimální cestou, která se nalezne rychle a s rozumnými nároky na paměť. Mezi postupy používané pro řešení takto formulovaných úloh patří algoritmy tzv. heuristického hledání, které tvoří součást metod umělé inteligence.

6 Nejkratší cesty z jednoho uzlu

Nejprve si provedeme určité zjednodušení. Od této chvíle budeme kvůli stručnosti označovat počet uzlů |U| grafu symbolem *n* a počet hran |H| grafu symbolem *m*.

Základní úlohou při určování vzdáleností v orientovaném grafu G = <H, U> je nalezení nejkratších cest z jednoho zadaného uzlu $s \in U$ do všech ostatních uzlů $v \in U$ grafu G. Algoritmus řešení této základní úlohy lze použít i v dalších variantách hledání nejkratších cest:

1) Problém nejkratších cest do jediného cílového uzlu spočívá v nalezení nejkratších cest ze všech uzlů $u \in U$ do pevně zadaného cílového uzlu *t*. Stačí zřejmě řešit základní úlohu v opačně orientovaném grafu G^-.

2) Problém jediné nejkratší cesty spočívá v určení nejkratší cesty pro jedinou zadanou (uspořádanou) dvojici uzlů $u, v \in U$. Pokud řešíme základní úlohu s výchozím uzlem *u*, vyřešíme tím současně i problém jediné nejkratší cesty. I když se může zdát, že se jedná o problém jednodušší než je základní úloha, není znám žádný algoritmus, který by měl asymptotickou složitost v nejhorším případě lepší než algoritmus pro základní úlohu.

3) Problém všech nejkratších cest znamená nalezení nejkratší cesty pro všechny (uspořádané) dvojice uzlů $u, v \in U$. Také tento problém lze řešit s použitím algoritmu pro základní úlohu tak, že jej použijeme postupně pro každý uzel grafu v roli výchozího uzlu. Existují však rychlejší postupy, kterým se věnujeme podrobněji v následující kapitole.

6.1 Nejkratší cesta v neohodnoceném grafu

Délka cesty v neohodnoceném grafu je rovna počtu hran, které cestu vytvářejí. Vzdálenost mezi vrcholy *u* a *v* je délka nejkratší cesty mezi *u* a *v*. Pokud neexistuje cesta mezi *u* a *v*, tak je vzdálenost nekonečná. Takto definovaná vzdálenost odpovídá vzdálenosti v ohodnoceném grafu, kde je každá hrana ohodnocená jedničkou. V tomto případě je možno nalézt řešení pomocí algoritmu prohledávání do šířky, jehož výpočet pro *n* vrcholů a *m* hran trvá ***O*(n+m)**. Prohledávání do šířky však může dát nesprávný výsledek u obecně ohodnoceného grafu, a to i v případě, že ohodnocení hran jsou nezáporná.

6.2 Prohledávání grafu do šířky (BFS)

Prohledání grafu, t.j. systematická prohlídka všech jeho uzlů i hran, patří mezi nejjednodušší a současně nejpotřebnější algoritmické postupy na grafech. Prohledávání do šířky (angl. breadth-first search) představuje základní variantu postupu prohledávání a z jeho hlavních myšlenek vycházejí další důležité grafové algoritmy - např. Dijkstrův algoritmus hledání nejkratší cesty (viz níže).

Pro zadaný graf G = <H, U> a vyznačený uzel *s* se prostřednictvím hledání do šířky systematicky prověřují hrany grafu G s cílem „nalézt“ každý uzel, který je dosažitelný z uzlu *s*. Prohledáním do šířky se současně stanoví vzdálenost od *s* ke každému dosažitelnému uzlu, a jako vedlejší produkt se tak získá **strom hledání do šířky -** nadále jej pro stručnost budeme označovat jako **BF-strom** - obsahující všechny z *s* dosažitelné uzly. Pro každý uzel v obsažený v tomto stromu je cesta z kořene *s* do uzlu *v* zároveň nejkratší cestou z *s* do *v* v grafu G. BF-strom je tedy současně **stromem nejkratších cest** z uzlu *s* do všech dosažitelných uzlů. V popsané podobě funguje algoritmus prohledávání do šířky pro neorientované i pro orientované grafy.

Název tohoto způsobu prohledávání vyjadřuje uniformitu postupu, při kterém se hranice mezi objevenými a (dosud) neobjevenými uzly rovnoměrně posouvá na celé své šířce. To má za následek, že algoritmus objeví všechny uzly ležící ve vzdálenosti k od uzlu *s* dříve, než první uzel ve vzdálenosti *k*+1. V průběhu prohledávání je každý algoritmem zpracovávaný uzel označen nejprve jako „nový“ (NEW), potom jako „otevřený“ (OPEN) a nakonec jako „uzavřený“ (CLOSED). Na začátku jsou všechny uzly nové. Jakmile algoritmus poprvé narazí při prohledávání na určitý uzel, označí jej jako otevřený. Jako uzavřený pak označí každý uzel, který už nepatří do hranice mezi objevenými a neobjevenými uzly - tedy uzel, který už zaručeně nemá žádného nového souseda.

BF-strom obsahuje na počátku pouze svůj kořen, kterým je uzel *s*. Jakmile se v průběhu procházení seznamu sousedů (následníků) nějakého uzlu u objeví nový uzel *v*, doplní se BFstrom o hranu (*u*, *v*) a uzel *v*. Uzel *u* se tak stane předchůdcem (rodičem) uzlu *v* v BFstromu. Každý uzel může být objeven nejvýše jednou, takže může mít také nejvýše jednoho předchůdce. Vedle základních relací vyjádřených pojmy předchůdce a následník budeme používat také reflexivně-tranzitivní uzávěry těchto relací pojmenované jako předek a potomek. Je tedy každý uzel *y* nacházející se na cestě z *s* do *x* (včetně uzlu *x* samotného) předkem uzlu *x* a naopak uzel *x* je potomkem každého takového uzlu *y*. Je důležité nezapomínat, že všechny zmiňované relace jsou odvozeny podle struktury BF-stromu, nikoliv podle výchozího grafu, takže relace „být předkem" nebo „být potomkem" mají charakter uspořádání.

Algoritmus 6.1 Prohledávání do šířky

BFS (G,s)

```
1     for každý uzel u ∈ U – {s}
2         do stav[u] := NEW
3            p[u] := NIL
4            d[u] := ∞
5     stav[s] := OPEN
6     p[s] := NIL
7     d[s] := 0
8     INITQUEUE; ENQUEUE(s)
9     while not EMPTYQUEUE
10        do u := QUEUE_FIRST
11           for každé v ∈ Adj[u] do
12              if stav[v] = NEW then stav[v] := OPEN
13                 p[v] := u
14                 d[v] := d[u] + 1
15                 ENQUEUE(v)
16           DEQUEUE
17           stav[u] := CLOSED
```

V algoritmu 6.1 hledání do šířky BFS předpokládáme, že vstupní (obyčejný) graf G = <H, U> je zadán pomocí spojové reprezentace. Algoritmus hledání dále používá několika pomocných údajů

vztahujících se ke každému uzlu a frontu, do níž ukládá otevřené uzly. Stav každého uzlu *u* zachycuje proměnná stav[*u*], předchůdce uzlu *u* je uložen v proměnné p[*u*]. Pokud uzel u nemá žádného předchůdce (je to např. uzel s nebo dosud neobjevený uzel), pak je p[*u*] = NIL. Vzdálenost od počátku *s* k uzlu u počítá algoritmus v proměnné d[*u*].

Všechny uzly mimo s označíme jako nové, zatím nedosažitelné z *s* a bez předchůdce. Samotný uzel *s* otevřeme s nulovou vzdáleností od *s* rovněž bez předchůdce a uložíme jej do fronty. Dokud není fronta prázdná, vybereme z ní první otevřený uzel a jeho sousedy, kteří jsou noví, otevřeme, určíme jim vzdálenost i předchůdce a uložíme je do fronty. Odebereme uzel *u* z fronty a zavřeme jej.

Kromě počáteční inicializace se nikdy nemůže stav uzlu změnit na NEW, takže test na řádce 12 zaručuje, že se žádný uzel nemůže uložit do (a tedy ani vybrat z) fronty více než jednou. Operace ukládání a vybírání mají pro frontu složitost $O(1)$, takže celkový čas operací s frontou je $O(n)$. Seznam sousedů uzlu se prochází pouze při vybírání uzlu z fronty, a to nastává pro každý uzel nejvýše jednou. Protože celková délka seznamů sousedů pro všechny uzly je $O(m)$, spotřebuje se procházením seznamů sousedů celkově nejvýše $O(m)$. Inicializační část má ovšem složitost $O(n)$, takže celková složitost bude $\boldsymbol{O(n+m)}$. Časová složitost algoritmu BFS je tedy stejná jako paměťová složitost spojové reprezentace grafu.

Nyní se budeme věnovat algoritmu BFS s ohledem na vlastnosti jím vytvořeného BF-stromu a spočtených hodnot d[*u*] pro uzly dosažitelné z uzlu *s*. Měli bychom potvrdit již dříve uvedené prohlášení, že pro každý (z uzlu *s* dosažitelný) uzel u je hodnota d[*u*] rovna vzdálenosti d(*s*, *u*), a BF-strom je tedy stromem minimálních cest z uzlu *s* do všech dosažitelných uzlů. Řádka 14 algoritmu se provádí pouze pro objevené - a tedy z *s* dosažitelné - uzly, takže se jenom pro ně počáteční nekonečná hodnota d[*v*] změní na konečnou. Nedosažitelným uzlům zůstane hodnota d[*v*] = ∞ = d(*s*, *v*).

Způsob, jakým se v algoritmu BFS zachycuje struktura BF-stromu je velmi úsporný: každý uzel *v* má prostřednictvím své hodnoty p[*v*] určeného svého (jediného) předchůdce, případně příznak, že žádného předchůdce nemá (p[*v*] = NIL). Toto vyjádření struktury je však možné pouze pro omezenou třídu grafů a navíc umožňuje efektivně realizovat jen některé operace. Odkazy na předchůdce v poli p[*v*] vyjadřují sice strukturu BF-stromu, jejich základní smysl je však určovat předchůdce uzlu v na nejkratší cestě z uzlu *s* do *v*.

Lze také použít algoritmu **prohledávání do hloubky** (angl. depth-first search), který probírá hrany vycházející z posledně nalezeného uzlu *v*, který má ještě nějaké neprobrané hrany. Když probere všechny jeho hrany, vrátí se zpátky k uzlu, z něhož se do uzlu *v* dostal, a z něho pokračuje po další dosud neprobrané hraně. Takovým způsobem se postupuje tak dlouho, dokud se neobjeví všechny uzly dosažitelné z prvního výchozího uzlu. Pokud zbývá nějaký neobjevený uzel, zvolí se jako další výchozí uzel, a prohledávání do hloubky pak pokračuje z něho. Tento postup se opakuje tak dlouho, až jsou objeveny všechny uzly.

Podobně jako při hledání do šířky uchovávají se i při hledání do hloubky o každém uzlu určité pomocné údaje, namísto fronty otevřených uzlů je zde ale (implicitní) zásobník zajišťující implementaci rekurze. Při objevení nového uzlu v jako následníka uzlu *u* se stejně jako u hledání do šířky přiřadí p[*v*] := *u*.

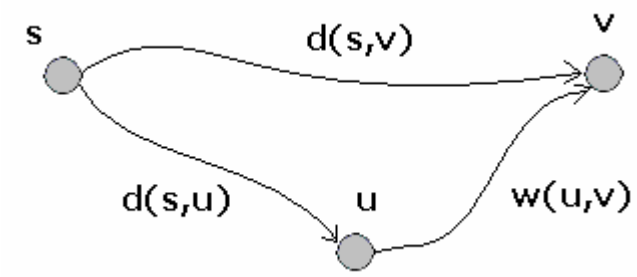
Algoritmus opět rozděluje uzly do tří skupin: nové (dosud neobjevené), otevřené a uzavřené. Otevřeným se uzel stane po svém prvním objevení, uzavřený je poté, co byl kompletně prozkoumán jeho seznam sousedů. Vedle toho se každému uzlu u přidělují dvě časové „značky“ uchovávané v poli d[*u*] a f[*u*]: značka d[*u*] odpovídá okamžiku objevení uzlu *u* (t.j. okamžiku jeho otevření), značka f[*u*] odpovídá jeho uzavření (t.j. okamžiku skončení průchodu jeho seznamu sousedů). Tyto značky vypovídají nejen o průběhu prohledávání, ale odrážejí i řadu vlastností grafu, a tak se často používají v dalších grafových algoritmech. Hodnotami značek jsou přirozená čísla od 1 do 2*n*, neboť pro každý uzel nastává právě jednou jeho objevení a právě jednou se uzavře. Po přidělení určité hodnoty se

značkovací čítač zvýší o 1, a tato hodnota bude přidělena příště. Výsledná složitost celého algoritmu DFS je ***O*(n+m)**. Pseudokód algoritmu DFS zde uveden nebude a lze jej nalézt v [1].

Ve většině aplikací mají hrany jinou než jednotkovou délku, neboť ohodnocení hrany například odpovídá délce úseku silnice. Předpokládejme, že délky hran jsou nezáporná celá čísla. Prohledávání do šířky, tak jak jsme ho použili výše, nebude fungovat, protože přímá hrana z *u* do *v* může být mnohem delší než cesta vedoucí po dvou krátkých hranách – úpravou algoritmu BFS se dostaneme k algoritmu Dijkstry (viz kapitola 6.4).

6.3 Relaxace hrany

Algoritmy řešící problém hledání nejkratší cesty z jednoho konkrétního uzlu do všech ostatních (neboli single-source shortest paths problem) jsou postaveny na následujícím zjištění. Zjistím vzdálenosti z uzlu *s* do všech ostatních uzlů a podívám se na jakoukoliv orientovanou hranu *h* v grafu. Pak musí platit trojúhelníková nerovnost, která je zachována díky tomu, že vždy bereme nejkratší cesty. Z uzlu *s* do uzlu *v* to nemůže být dál, než kolik to bude přes libovolný uzel *u* a přes hranu, která ho spojuje do uzlu *v*. Pokud by toto neplatilo, pak neplatí, že vzdálenost je délka nejkratší cesty (formálněji v [1]). Pro libovolnou hranu $(u, v) \in H$ a uzel $s \in V$ platí : $d_w(s, v) \leq d_w(s, u) + w(u, v)$. Tato nerovnost platí nejen pro konečné vzdálenosti, ale i pro ∞ a -∞.



Obrázek 6.2: Trojúhelníková nerovnost

A teď můžeme rozhodit na grafu počáteční vzdálenosti a ty upravovat pokaždé tak, že se podíváme na jednu hranu a zjistíme, zda je splněna podmínka nerovnosti. Pokud je, nic neděláme. Pokud není, pak to znamená, že přes tuto hranu h mi vede kratší cesta a tudíž tuto vzdálenost můžeme snížit. Toto je základní myšlenka pro operaci, které říkáme **relaxace hrany** *h*.

Ještě než přejdeme k vlastním algoritmům třeba se podívat jaké datové struktury potřebujeme. Algoritmy mají mnoho společného s prohledáváním grafu do šířky. Prostřednictvím odkazů na předchůdce zachycují strukturu stromu nejkratších cest. Pokud chceme cestu zpětně rekonstruovat, pak je nutné si uchovávat hodnotu p[*u*], která odkazuje uzel, který je předchůdcem na nejkratší cestě. Co třeba u každého uzlu uchovávat je aproximovaná vzdálenost od výchozího uzlu *s*. A nakonec ještě potřeba pomocné (prioritní) fronty, ve které budeme ukládat otevřené uzly pro další zpracování.

Většina algoritmů v této kapitole má dvě společné operace a ty si teď popíšeme, abychom je nemuseli u každého algoritmu opakovat. První z nich je **inicializace algoritmu**. Provádíme ji standardním způsobem. Všem uzlům nedefinuji předchůdce (tzn. NIL), všem uzlům dáme nekonečnou vzdálenost (základní aproximaci vzdálenosti) s výjimkou uzlu s, který inicializuji na počáteční vzdálenost 0.

Algoritmus 6.2 Inicializace algoritmu pro hledání cest

INIT_PATHS(G,s,w)
1 **for** každý uzel $u \in U$
2 **do** p[u] : = NIL
3 d[u] : = ∞
4 d[s] := 0

Na rozdíl od prohledávání do šířky je potřeba hodnotu d[u] průběžně upravovat, neboť první nalezení uzlu ještě nezaručuje, že se k němu dospělo nejkratší cestou. Každé následující „znovuobjevení“ tohoto uzlu u může způsobit snížení dosavadní hodnoty d[u], neboť při určení w-vzdálenosti nehraje roli počet hran minimální cesty, ale součet jejich ohodnocení.

Snižování hodnoty d[u] přitom vychází z následující základní úvahy: je-li d[u] horní mezí skutečné vzdálenosti $d_w(s, u)$, potom platí (podle [6]):

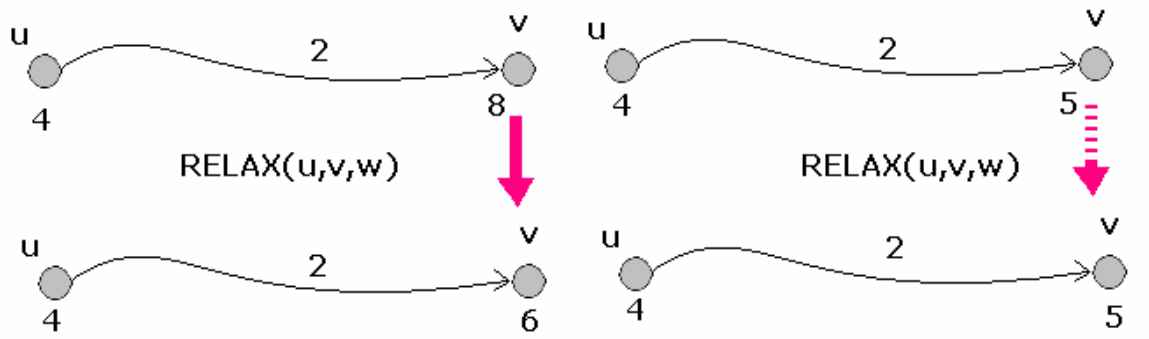
$$d[u] + w(u, v) \geq d_w(s, u) + w(u, v) \geq d_w(s, v), \qquad \textbf{(6.1)}$$

takže také hodnota d[u] + w(u, v) pro libovolnou hranu (u, v) je horní mezí vzdálenosti $d_w(s, v)$. Protože hodnotu horní meze je možné zpřesňovat pouze tak, že ji snižujeme, můžeme novou hodnotu d[v] určit jako d[u] + w(u, v), jakmile bude nová hodnota menší, než byla hodnota předchozí. A jsme zpátky u elementární operace **relaxace hrany** (u, v). Ta je onou druhou základní operací většiny následujících algoritmů,

Algoritmus 6.3 Relaxace hrany

RELAX(u,v,w)
1 **if** d[v] > d[u] + w(u, v)
2 **then** p[v] : = u
3 d[v] : = d[u] + w(u, v)

Lze-li d[v] zmenšit, tak ho zmenšíme a také upravíme předchůdce. Na obr. 6.2 jsou dvě rozdílné situace, na kterých ukazujeme účinek relaxace. V prvním případě je podmínka na řádce 1 operace RELAX splněna, takže po relaxaci je nová hodnota d[v] menší. Ve druhém případě se relaxací hodnota d[v] nezměnila. Jak je bezprostředně vidět, po provedení relaxace hrany (u, v) bude hodnota d[v] zaručeně splňovat nerovnost d[v] ≤ d[u] + w(u, v). Mohla ji splňovat již před relaxací, a pak se nic nezměnilo, nebo platila nerovnost opačná, a pak se při relaxaci dosadila do d[v] hodnota nová. Formální zdůvodnění a další tvrzení o operaci relaxace lze nalézt v [1].



Obrázek 6.2: Relaxace hrany

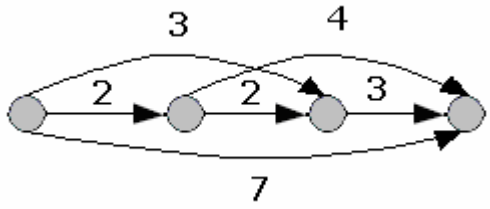
Stále nám ale chybí jednoduchý návod, jak hrany pro relaxaci vybírat, abychom dostali skutečně efektivní algoritmus hledání cest. Návrhu a analýze složitosti takových algoritmů se věnujeme v dalších odstavcích.

6.4 Acyklické grafy

Starosti se zápornými cykly zcela odpadají u acyklických grafů (neobsahují totiž žádný cyklus), pro které lze navrhnout výrazně zjednodušenou variantu se složitostí **Θ(n+m)**. Nejkratší cestu v nich nalezneme jednoduchým způsobem. Nalezneme topologické uspořádání uzlů a pak už jen procházíme vrcholy v nalezeném pořadí a relaxujeme hrany z nich vedoucí. Hrany mohou být ohodnoceny libovolně, tedy i záporně.

6.4.1 Topologické uspořádání

Topologické uspořádání umíme udělat pomocí algoritmu prohledávání do hloubky. Topologickým uspořádáním se rozumí takové úplné uspořádání uzlů, v němž je pro každou hranu (*u*, *v*) uzel *u* před uzlem *v*. Takové uspořádání lze zřejmě zavést pouze pro acyklické grafy, neboť libovolný cyklus (včetně smyčky) by takové uspořádání znemožnil. Lze jej vyjádřit tak, že uzly seřadíme do jediné horizontální úrovně, a orientace všech hran pak musí směřovat jedním směrem - např. zleva doprava. Topologické uspořádání grafu není obecně určeno jednoznačně. Časová složitost popsaného postupu je zřejmě ***O*(n+m)**, neboť prohledání do hloubky trvá *O*(n+m) a k přidání nového uzlu na začátek seznamu potřebujeme *O*(1) pro každý z *n* uzlů zpracovávaného grafu. Algoritmus topologického uspořádání acyklického orientovaného grafu lze najít v literatuře.



Obrázek 6.2: Acyklický graf v topologickém uspořádání

Pro testování acykličnosti a současně pro topologické uspořádání grafu se nabízí ještě další postupy. Zcela neefektivní by ovšem bylo např. systematicky procházet všechny orientované cesty grafu a testovat, zda některá z nich nelze uzavřít, a vytvořit tak cyklus. Rozumnější je hledat v grafu G podgraf splňující nějakou jednoduše ověřitelnou podmínku pro existenci cyklu. Takovou snadno testovatelnou podmínku nabízí následující tvrzení.

Nechť pro uzly neprázdného orientovaného grafu G = <H, U> platí $\delta_G^+(u) \geq 1$ pro všechna $u \in U$ (tzn. graf G nemá žádný list). Potom graf G obsahuje cyklus. Nebo-li neprázdný orientovaný graf obsahuje cyklus, jestliže nemá žádný list nebo žádný kořen (více v [1]). Pro existenci cyklu v grafu navíc stačí, aby uvedenou podmínku splňoval nějaký jeho podgraf - ukážeme teď velmi

jednoduchý postup, který takový podgraf určí nebo jeho existenci vyloučí. Základem postupu je následující tvrzení.

Orientovaný graf G je acyklický právě tehdy, je-li acyklický každý jeho podgraf G - {*u*}, kde *u* je libovolný kořen nebo list grafu G. Při zjišťování acykličnosti tedy stačí vypustit z grafu nějaký jeho kořen nebo list, potom jiný kořen nebo list vzniklého podgrafu atd., až dospějeme k prázdnému grafu nebo k podgrafu, který už žádný kořen ani list nemá. V souladu s rozšířeným zněním předchozího tvrzení ovšem stačí vypouštět např. pouze kořeny grafu. Algoritmus je rovněž uveden v [1] a jeho celková složitost je jako pro algoritmus využívající prohledávání do hloubky - tedy ***O*(n+m).**

6.4.2 Hledání nejkratší cesty v acyklickém grafu

Máme-li acyklický graf v topologickém uspořádání, pak už jen stačí procházet uzly v tomto pořadí a pro každý uzel relaxovat všechny z něj vycházející hrany.

Algoritmus 6.4 DAG algoritmus

DAG_PATHS(G,s,w)
1 topologické uspořádání uzlů grafu G
2 INIT_PATH(G,s)
3 **for** každý uzel *u* v pořadí jeho topologického uspořádání
4 **do for** každé v ∈ *Adj*[*u*]
5 **do** RELAX(u,v,w)

Zdůvodnění časové složitosti této varianty vychází z toho, že topologické uspořádání na řádce 1 lze provést v čase **Θ(n+m)**. Hlavní cyklus se provádí pro každý uzel právě jednou a výběr uzlu *u* trvá konstatní čas. Ve vnitřním cyklu se pak relaxuje každá hrana grafu právě jednou, a to opět v čase *O*(1).

6.5 Dijkstrův algoritmus

Dijkstrův algoritmus (Dijkstra tento algoritmus navrhl v [17]) je možné chápat jako určité zobecnění postupu prohledávání grafu do šířky (viz výše). Také při něm se počínaje výchozím uzlem s šíří ve směru k dosud vůbec nebo jen „oklikou“ objeveným částem grafu systematicky posouvaná vlna. Tato vlna pohlcuje jen takové uzly, k nimž byla již nalezena nejkratší cesta.

Důležitým předpokladem použitelnosti Dijkstrova algoritmu je, aby ohodnocení hran w mělo pouze nezáporné hodnoty, jinak algoritmus nebude dávat správné výsledky. Řešíme tedy nejkratší cestu z jednoho uzlu do všech ostatních na orientovaném grafu G = <H, U> s nezáporným ohodnocením hran $w : H \rightarrow \mathbf{R}^+$.

Základní kroky Dijkstrova algoritmu a údaje uchovávané o každém uzlu se kryjí s tím, co jsme popsali výše. Jeho specifičnost spočívá pouze ve způsobu výběru hran pro relaxaci. Algoritmus postupně rozšiřuje množinu uzlů S, jejichž aproximovaná vzdálenost d[*u*] se již shoduje se skutečnou w-vzdáleností $d_w(s, u)$. V analogii s prohledáváním do šířky bychom tyto uzly mohli označit jako uzavřené.

Zbývající uzly grafu G, tedy U - S, jsou umístěny v prioritní frontě Q seřazené vzestupně podle hodnot d[*u*]. Přestože se formálně nerozlišují otevřené a nové uzly, na začátku této fronty jsou vždy uzly otevřené, neboť jen ony mají přiřazenu konečnou hodnotu d[*u*]. Následující vyjádření Dijkstrova algoritmu předpokládá reprezentaci grafu pomocí seznamu následníků.

Algoritmus 6.5 Dijsktrův algoritmus

DIJKSTRA(G,s,w)

```
1     INIT_PATHS(G,s)
2     S := ∅
3     INIT_QUEUE(Q)
4     for každý uzel u ∈ U
5         do ENQUEUE(Q,u)
6     while not EMPTY(Q)
7         do u := EXTRACT_MIN(Q)
8             S := S ∪ {u}
9             for každý uzel v ∈ Adj[u]
10                do RELAX(u,v,w)
```

Při prvním průchodu cyklem while se jako nejbližší vybere právě uzel *s*, který má jako jediný přiřazenu konečnou (nulovou) hodnotu d[*u*]. Při relaxaci hrany (*u*, *v*) se v případě úpravy hodnoty d[*v*] předpokládá zařazení uzlu *v* na odpovídající místo ve frontě Q. Do této fronty se však žádný uzel neukládá opakovaně, takže z ní bude vybrán právě jednou a hlavní cyklus na řádcích 6 - 10 se bude opakovat *n*-krát.

Kritériem řazení do fronty je d[*u*] (horní) odhad w-vzdálenosti $d_w(s, u)$, tedy délky nějaké minimální cesty, a jeho výsledkem je strom minimálních cest z uzlu *s*.

V literatuře je často tento algoritmus popisován jako jedna z variant hladového algoritmu. I v případě hledání nejkratších cest je splněna jak podmínka optimálnosti podstruktury, tak i podmínka hladového výběru (lokálně optimální volba vede ke globálně optimálnímu řešení). Správnost Dijkstrova algoritmu a ověření všech tvrzení lze nalézt v literatuře např. [6] a není součástí této práce.

Algoritmus je konečný, protože v každé iteraci prohlásí jeden uzel za uzavřený. Vrcholů je *n* a z každého vede nejvýše *n* hran. V průběhu algoritmu lze testovat, jestli je hodnota *d*[*v*] vybraného vrcholu *v* rovno nekonečnu a případně skončit (k žádným změnám už by stejně nedošlo). Podobně pokud hledáme pouze nejkratší cestu do vrcholu *w* a ne do všech vrcholů, tak lze skončit v momentě, kdy bude *w* prohlášen za uzavřený. Pokud je však povoleno ohodnocení hran záporné, může být počet výběrů z fronty exponenciální – více v [11].

Není překvapením, že od jeho uvedení v roce 1959 byl algoritmus předmětem četných čistění a modifikací tak jako numerických experimentů. To způsobilo výrazné zlepšení výkonu algoritmu zvláště díky použití sofistikovaných datových struktur k implementaci prioritní fronty. Taková datová struktura musí podporovat následující operace na kolekci položek, z nichž každá je asociovaná klíčem, který určuje pořadí:

INSERT(i, k): přidej položku i do kolekce s klíčem k.
DELETE(i): smaž položku i z kolekce.
DEL-MIN(): vrať položku s nejmenším klíčem z kolekce a vymaž jí z kolekce.
DEC-KEY(i, k): nastav klíč položky i na hodnotu k; hodnota k nemůže být větší než aktuální hodnota klíče položky i. Tato operace je obvykle implementována jako DELETE následované INSERT.

Dijkstrův algoritmus běží na grafech s *n* uzly a *m* hranami, vykonává sekvenci *n* INSERT a *n* DEL-MIN a (hodně) *m* DEC-KEY operací. Pro všechny řídké grafy, kde *m* je asymptoticky větší než

n, bychom chtěli DEC-KEY operaci nejlépe v čase *O*(1). Rozlišíme dva druhy datových struktur v závislosti na tom jak je či není operace DEC-KEY nákladná.

a) **Prioritní fronta** (PQ) podporuje operace INSERT, DELETE a DEL-MIN a nepodporuje DEC-KEY významněji než DELETE následovanou INSERT. Atomická PQ podporuje INSERT, DELETE, DEL-MIN v *O*(1) a udržuje *m* klíčů.

b) **F-heap** podporuje všechny operace a DEC-KEY je významněji levnější než DELETE následované INSERT. INSERT a DEC-KEY lze v čase *O*(1) a DEL-MIN a DELETE v *O*(log n).

Algoritmus rozděluje uzly na 3 stavy : nové (v anglické literatuře označené jako unlabeled), otevřené (labeled) a uzavřené (scanner). Nové uzly mají d(*v*) nekonečno. Otevřené uzly s konečnou vzdáleností, jejíž minimální cena není dosud známá a u uzavřených uzlů známe jejich minimální cenu. Algoritmus opakuje následující kroky až jsou všechny vrcholy uzavřené.

- Vybere otevřený uzel *v* takový, který má d(*v*) minimální a změníme jeho stav na uzavřený.
- Pro každou hranu (*v*, *w*), je-li d(*v*) + w(*v*, *w*) < d(*w*), nahraď d(*w*) hodnotou d(*v*) + w(*v*, *w*) a deklaruj *w* jako otevřený byl-li nový.

Klíč k účinné implementaci DA je použití haldy jako prioritní fronty. Halda se skládá ze sady položek, každá s přidruženým klíčem reálné hodnoty, která umožňuje operace INSERT, DEL-MIN a DEC-KEY. V implementaci Dijkstrova algoritmu založeného na haldě, obsahuje halda *h* všechny otevřené vrcholy a klíč obsahuje aktuální odhad vzdálenosti. Máme *n* operací INSERT, *n* operací DEL-MIN a nanejvýš *m*-*n*+1 operací DEC-KEY (více v [7]).

Časová složitost Dijkstrova algoritmu závisí na implementaci prioritní fronty. Počáteční inicializace a cyklus na řádcích 4 - 5 budou trvat *O*(n). Hlavní cyklus se provede *n*-krát a v rámci každého průchodu proběhne jednou výběr uzlu a zpracování všech jeho následníků. Předpokládáme-li realizaci prioritní fronty **binární haldou**, lze výběr uzlu provést v čase *O*(log n), celkově tedy budou operace výběru trvat *O*(n log n). Relaxace hrany se provádí celkem *m*-krát a při změně hodnoty d[*u*] je třeba zařadit uzel *u* na odpovídající místo v prioritní frontě, což trvá *O*(log n). Dohromady tedy dostáváme asymptotickou časovou složitost *O*(n log n + m log n) = *O*((n + m) log n) = ***O*(m log n)**. Tento výsledek ukazuje, že binární haldu bude výhodné použít především pro řídké grafy, kdy je počet hran lineárně omezen počtem uzlů.

Pro některé třídy orientovaných grafů, např. grafy s omezeným ohodnocením hran, rovinné grafy a téměř acyklické grafy (obsahují poměrně velmi málo cyklů vzhledem k množství uzlů, které mají – viz [19]) zlepšíme časovou složitost Dijkstrova algoritmu při použití **Fibonacciho haldy** [18]. Při její implementaci lze snížit celkovou složitost pro operaci výběru uzlů na *O*(n log n) a na *O*(m) pro úpravy pořadí při relaxacích, neboť jedna operace zařazení uzlu se sníženou hodnotou d[*u*] se pak provádí v konstantním čase. Celkově bychom tedy v tomto případě měli složitost ***O*(n log n+m)**.

Pro husté grafy je ovšem možné zvážit i sekvenční implementaci fronty Q v poli. Jde pak o tzv. **naivní implementaci** Dijkstrova algoritmu, která hledá minimum mezi všemi kandidáty v neseřazeném poli. Výběr jednoho uzlu pak trvá *O*(n), celkově tedy *O*(n^2), naproti tomu snížení hodnoty *d*[*u*] lze zajistit v konstantním čase. Celková asymptotická časová složitost Dijkstrova algoritmu je v tomto případě *O*(n^2 + m) = ***O*(n^2)**.

Možností jak implementovat prioritní frontu v algoritmu je mnoho. Další si uvedeme jen v krátkém přehledu. Implementace pomocí Fibonacciho haldy dosahuje nejlepšího času Dijkstrova algoritmu, jsou-li hrany reálně ohodnocené a jen binární porovnávání je užité v implementaci haldy. Je otázkou, zda jde tento čas zlepšit v případě kdy hrany jsou středně velká celá čísla. Toto je zkoumáno v [47]. Datová struktura radix heap využívá speciální vlastnosti operací haldy v Dijkstrově algoritmu a to postupné operace DEL-MIN vracející vrcholy v nesestupné posloupnosti dle d[*u*]. Nejjednodušší forma jednoúrovňové radix haldy, původně navrhovaná Johnosonem [48], který ji použije k získání *O*(m log log C + n log C log log C). Na grafu s *n* vrcholy, *m* hranami a s ohodnocením hran nezápornými celými čísly ohraničené hodnotou C jednoúrovňová forma radix heap dává čas *O*(m+n log C), dvouúrovňový radix heap dává hranici *O*(m+n log C / log log C) a kombinace radix heap a předtím užívané Fibonacciho haldy dává O(m+n √(log C)). Nejlepší předtím

známé hranice jsou O(m + n log n) použitím Fibonacciho haldy samotné a O(m log log C) použitím datové struktury Boas-Kaas-Zijlstra [49]. Prostor potřebný pro haldu je O(n+C) ale může být redukován použijeme-li hashování klíče.

Další možnou implementací je užití datové struktury – bucket - navržené Dialem [34]. V této implementaci algoritmu, někdy mu říkáme **Dialův algoritmus**, udržujeme pole těchto datových struktur (bucket). Jednotlivý i-tý bucket obsahuje všechny uzly v mající $d(v) = i$. Když se změní vzdálenostní odhad uzlu, je odstraněn z odpovídajícího bucketu a vložen do bucketu korespondujícího s novým d(v) v FIFO pořadí. Uchováváme index L. Na začátku jsou všechny bukety kromě toho s indexem L = 0 prázdné. Uzel, který je vybrán odstraníme z buketu. Je-li bucket prázdný, inkrementujeme index L. Operace odstranění a vložení do buketu mají lineární čas a nejvýše nC bucketů musíme prozkoumat algoritmem. Je-li ohodnocení hran nezáporné je celková složitost $\boldsymbol{O}$**(m + nC)**. Pomocí této datové struktury lze v literatuře najít implementace : double bucket, k-level bucket implementation (v této variantě dokonce může být ohodnocení hran záporné).

Možnou alternativou je udržování hodnoty t - největší odhad vzdálenosti uzlu zpracovávaného doposud a následně vybrat uzel s hodnotou $d(v) \leq t$, jestli takový existuje, jinak uzel s nejmenší vzdáleností, tato strategie je přirozená v bucket a R-heap implementaci.

Jsou-li uzly na zpracování udržované ve frontě - lze ukázat polynomiální časovou složitost pro tuto variantu Dijkstrova algoritmu na sítích s libovolným ohodnocením hran.

6.6 Dantzigův algoritmus

Ve stejné době, kdy Dijkstra uvedl svůj algoritmus, se objevila řada podobných algoritmů (např. [67]). Jedním z nich je i Dantzigova úprava Fordova-Fulkersonova algoritmu (více viz [71]), který je v některé literatuře označen jako Dantzigův algoritmus. V jiných pramenech zase nalezneme pod tímto označením algoritmus pro řešení problému nejkratších cest mezi všemi páry vrcholů (viz další kapitola) řešeného pomocí lineárního programování a Dantzigovy simplexové metody. V této práci bude za Dantzigův algoritmus považován ten, jehož princip uvedl v textu [20] a je navržen pro řešení nejkratších cest z jednoho konkrétního uzlu do všech ostatních.

Nejprve seřadíme všechny hrany v grafu dle jejich rostoucího ohodnocení. V množině S udržujeme vrcholy, jejichž vdálenosti již byly stanoveny algoritmem. Každý vrchol $c \in$ S má svoji kandidátní hranu (c, t). A máme funkci L(i, j), která pro dva vrcholy i, $j \in$ U udává cenu nejkratší cesty z i do j. Algoritmus je následující:

Algoritmus 6.6 Dantzigův algoritmus

DANTZIG(G,s)

```
1     S: = {s}
2     d[s] : = 0
3     inicializuj množinu kandidátů {(s, t)}, kde (s, t) je nejkratší hrana vycházející z s
4     while |S| < n do
5         vybereme platnou kandidátní hranu (c, t) s nejmenší váhou
6         S := S ∪ {t}
7         d[t]:= d[c] + w(c, t)
8         if |S| = n then break
9         přidej k sadě kandidátů nejkratší platnou hranu z uzlu t
10        for každého nepotřebného kandidáta (v, t) do
11            nahraď (v,t) v sadě kandidátů další nejkratší platnou hranou z v
```

Nejdříve je vrcholu *s* přiřazena cena nejkratší cesty rovna nule a množina otevřených vrcholů S, pro které jsou již nejkratší ceny známé obsahuje jediného člena. Pak pod omezením, že vrcholy $c \in$ S jsou blíže k *s* než nečlenské vrcholy *u*, takže L(*s*, *c*) ≤ L(*s*, *u*), množinu S rozšiřujeme dokud nejsou všechny vrcholy otevřené. Výše uvedený algoritmus udržuje kandidátské hrany (*c*, *t*) pro každý uzel $c \in$ S. Také udržujeme informaci o cestách z vrcholů již v S do vrcholů stále vně množinu S k tomu, aby jsme výpočetně snadněji rozšiřovali S. Jestliže kandidátský koncový uzel *t* je vně aktuální množiny S, pak je *t* považovaný za platného uchazeče, jinak je nadbytečný. Dantzigův algoritmus vyžaduje všechny kandidáty platné. Splní-li kandidátský vrchol tyto požadavky, tak je uzel *c* vybrán na skenování. Prohlížením setříděného seznamu hran vedoucích z *c* ve zvyšujícím se cenovém pořadí, dokud nenalezneme užitečnou hranu (*c*, *t*). Pokud platnou hranu (*c*, *t*) nanalezneme, máme zaručeno, že *t* je nejbližší vrchol (s použitím jedné hrany) k *c* a *c* nepatří do S (více v [33]).

Vektor *d* udržuje nejkratší cesty pro vrcholy, které jsou otevřené, tj. $c \in$ S, d[*c*] = L(*s*, *c*). Cena hrany z *c* do *t* je dána w(*c*, *t*) a cena cesty přes vrchol *c* ke kandidátskému uzlu *t* je dána d[*c*] + w(*c*, *t*). Vrchol *t* by mohl být kandidátem dalších již otevřených vrcholů $v \in$ S. Vždy, když to je kandidát s váhou d[*v*] + w(*v*, *t*) asociovanou s jeho kandidaturou. Bude celkově |S| kandidátů s koncovými body rozptýlenými mezi *n*-|S| nových vrcholů neležících v S.

V každém kroku algoritmu je koncový bod s nejmenší váhou kandidáta přidán k množině S. Jeli *c* vrchol takový, že kandidátská váha d[*c*] + w(*c*, *t*) je minimální mezi všemi otevřenými uzly *c*, tak *t* můžeme přidat do množiny S s danou cenou d[*t*] = d[*c*]+ w(*c*, *t*). Pak další kandidát uzlu *t* je přidán do množiny kandidátů. Kandidáti na vrcholy, kteří se staly zbyteční odstraníme (včetně *c*) a postup opakujeme a zastavíme, když |S| = *n*.

Nejkratší cestu mezi uzly *s* a *t* vypočítáme iteračním postupem, který každému vrcholu *v* grafu přiřadí hodnotu d[*v*] znamenající nejkratší vzdálenost tohoto uzlu od počátečního uzlu. Zpětným chodem (od koncového k počátečnímu uzlu) nalezneme příslušnou nejkratší cestu z *s* do *t* jako posloupnost vrcholů, pro které je rozdíl ohodnocení roven vzdálenosti. Vzhledem k tomu, že počítáme minima, stačí, když graf bude souvislý. Pokud ohodnocení hran bude nezáporné, pak graf nemusí být ani orientovaný a ani acyklický.

Uvedený algoritmus nenalezne pouze nejkratší cestu mezi danými dvěma uzly grafu, ale i nejkratší cesty mezi výchozím a všemi ostatními uzly grafu. Hrany, které jsou součástí nejkratších cest, tvoří spolu se všemi uzly podgraf původního grafu - tedy strom nejkratších cest.

Pokud velikost množiny |S| = n, vyžadujeme čas O(n) pro hledání kandidáta s minimálními náklady v poli kandidátů. Poté co jej nalezneme potřebujeme další O(n) k tomu, abychom rozhodli, zda zůstávají další kandidáti užiteční. Další co přispívá k době běhu algoritmu je skenování seznamu hran hledající užitečné hrany. Každá hrana grafu bude zkoumána maximálně jednou a požadované úsilí je tak $O(n^2)$. Celkové úsilí je tak $\boldsymbol{O(n^2)}$. Čas pro třídění $O(n^2 \log n)$ je absorbován v celkové složitosti.

Ač je Dantzigův algoritmus velmi podobný standardnímu Dijkstrově algoritmu, lze si všimnout, že na rozdíl od Dijkstrova algoritmu ve verzi z roku 1959, není generována jen délka nejkratší cesty, ale i nejkratší cesta samotná. Dijsktrův algoritmus musel být upraven tak aby udžoval seznam vrcholů na nejkratší cestě např. uskladněním nejen vzdálenosti nejkratší cesty, ale i předcházejícího vrcholu na nejkratší cestě. V této základní verzi Dijkstrův algoritmus poskytuje rozporuplné výsledky na reálných sítích, neboť takové sítě velmi omezují souvislost (jejich matice sousednosti jsou velmi řídké). Algoritmy pro řešení problému nejkratších cest potřebují buď velmi účinné implementace těchto základních algoritmů nebo by měly být použité heuristiky (více o pár řádků níže) k získání rychlejších řešení (např. pro real-time aplikace).

V literatuře se objevují dohady, že postup známý jako Dijkstrův algoritmus byl objeven v padesátých letech nezávisle řadou analytiků (vybráno z [79]). Jsou silné indície, že v jistých kruzích před publikací Dijkstrova proslulého listu [17] se objevilo nezávisle několik algoritmů. V roce 1959 publikoval Dijkstra krátký referát do Numerische Mathematik. Ke konci téhož roku publikoval Pollack a Wiebenson článek *Solutions of the shortest-route problem - a review* v magazínu

Operations research (OR), kde se porovnává 7 metodik pro řešení problému nejkratších cest. Další zajimavý historický fakt je, že v listopadu 1959 George Dantzig předložil list nazvaný *On the shortest route through a network* pro Management Science [20]. V něm navrhuje algoritmus pro řešení nejkratších cest v případech, kdy vzdálenosti z každého uzlu mohou být snadno tříděné ve zvyšující se posloupnosti a tím pádem můžeme ignorovat dynamicky určitou hranu. Nemá žádný vztah k Dijkstrově listu, ale zřejmě je tam nějaký vztah k Bellmanově práci [15], neboť oba (Dantzig a Bellman) pracovali ve stejné době v RAND. Jediné co mají Dijkstrův a Dantzigův list společné je odkaz na Fordův report *Network flow theory* [16]. Nedávno Dantzig a Thapa [68] poukázali na to, že Dijkstrův algoritmus je očištěný algoritmus nezávisle navržený Dantzigem ve stejném roce. Nicméně se zdá, že oba algoritmy jsou navzájem podstatně rozdílné (viz komentáře v [66]). Dantzigův algoritmus byl první obdařen možností použití strategie bi-direcitonal pro řešení problému nejkratších cest (více později). Nicméně nebyla navrhnuta žádná politika omezení pro obousměrnou verzi jeho algoritmu.

Jak se následně ukázalo Dijkstrův algoritmus má silné kořeny v OR a Computer Science (CS), nejspíše proto se stal nosným algoritmem a Datzigův algoritmus má zastání, jak se mi povedlo nalézt, jen v oblasti operačního výzkumu.

6.7 Bellmanův-Fordův algoritmus

V případě grafů se záporně ohodnocenými hranami není Dijkstrův algoritmus použitelný, přitom w-vzdálenost z daného uzlu *s* zůstává korektně definována pro všechny uzly, k nimž vede z *s* orientovaná cesta neprocházející žádným cyklem se zápornou w-délkou. Pro takové případy zajišťuje Bellmanův-Fordův algoritmus nalezení nejkratších cest a současně dokáže zjistit, zda se v grafu vyskytuje záporný cyklus a tím pádem optimální řešení neexistuje. Základní informace lze nalézt v textech autorů v [15] a [16].

I při formulaci Bellmanova-Fordova algoritmu použijeme techniku relaxace hran popsanou výše, která systematicky snižuje horní meze d[*u*] vzdáleností $d_w(s, u)$ tak dlouho, až dosáhnou svého minima - tedy w-délky nejkratší cesty z *s* do *u*. Jediný rozdíl je ve výběru hran pro relaxaci: zatímco Dijkstrův algoritmus relaxoval vždy jen hrany vycházející z vybraného uzlu, v tomto případě se opakovaně relaxují *systematicky* všechny hrany.

Algoritmus 6.6 Bellmanův-Fordův algoritmus

```
BELLMAN-FORD(G,s,w)
1       INIT_PATHS(G,s)
2       for i := 1 to n - 1 do
3           for každou hranu (u, v) ∈ H
4               do RELAX(u,v,w)
5       for každou hranu (u, v)∈ H
6           do if d[v] > d[u] + w(u, v)
7               then return FALSE
8       return TRUE
```

Opakovaně relaxujeme všechny hrany grafu. V posledním cyklu (řádek 5) provedeme ještě jednou relaxaci přes všechny hrany, ale teď jen zkoumáme, zda nějaká z rozběhnutých relaxací bude úspěšná. Tj. zda-li se takovou relaxací mohlo docílit zmenšení d[*v*]. Pokud toto nastane, pak se

prozradila existence záporného cyklu v grafu a algoritmus vrátí hodnotu FALSE. Pokud graf neobsahuje cyklus záporné w-délky dosažitelný z uzlu *s*, algoritmus skončí s výslednou hodnotou TRUE, pro všechny uzly $u \in U$ platí $d[u] = d_w(s, u)$ a prostřednictvím odkazů $p[u]$ je určen odpovídající strom nejkratších cest z uzlu *s*.

U Bellmanova-Fordova algoritmu nelze postupně vytvářet množinu uzlů, které již mají přesně určenu vzdálenost, neboť i při posledním průchodu hlavním cyklem se mohou změnit hodnoty $d[u]$ všech uzlů.

Asymptotická časová složitost Bellmanova-Fordova algoritmu je ***O*(nm)**, neboť relaxace hrany má v tomto případě konstantní složitost (na rozdíl od Dijkstrova algoritmu, nepotřebuje přesouvat prvky v prioritní frontě, dokonce stačí prvky procházet v lineárním pořadí) a provádí se *nm*-krát. Ověření správnosti tohoto algoritmu lze najít v literatuře např. [6].

Idea algoritmu je následující. Máme graf o *n* uzlech. Nejdelší cesta (procházející všechny uzly) může mít maximálně *n*-1 hran. Proto stačí protočit relaxaci podél všech hran jen (*n*-1)-krát, neboť relaxace na sebe navazující protlačují zmenšené hodnoty podél orientovaných cest a v nejvýše *n*-1 iteraci se dostanou až k posledním uzlům. Pokud by ještě zabrala relaxace v dalším průchodu, tak to znamená, že dospívá k nějakému uzlu znova (ačkoliv se k němu již cesta našla dříve). To neznačí nic jiné než existenci záporně ohodnoceného cyklu.

Výstupem Bellmanova-Fordova algoritmu je booleovská funkce, vracející FALSE v případě detekce záporně ohodnoceného cyklu ve vstupním grafu, jinak vrací TRUE. Zároveň v hodnotách d od jednotlivých uzlů jsou vypočteny vzdálenosti nejkratších cest, které ovšem v případě návratu FALSE nelze využít.

Bellmanův-Fordův algoritmus je obecnější, který funguje i pro záporně ohodnocené hrany. Ale pokud je hran řádově n^2 pak dává složitost $\boldsymbol{O(n^3)}$**.** Což nám určitě nevyhovuje. Zkusíme ho tedy trochu upravit (rozsáhleji o této modifikaci v [6]). Relaxace podél nějaké hrany bude v následujícím kroku úspěšná pokud se některou předchozí relaxací snížila hodnota $d[u]$ (za takové snížení považujeme i počáteční nastavení $d[s] = 0$). Kdybychom si zapamatovali uzly, u kterých došlo v minulém průběhu relaxací ke změně hodnoty $d[u]$, pak bychom jen z těchto uzlů postupovali dál. Dostáváme tak následující modifikaci Bellmanova-Fordova algoritmu.

Algoritmus 6.7 Upravený Bellmanův-Fordův algoritmus

```
BELLMAN_FORD_Q(G,s,w)
1       INIT_PATHS(G,s)
2       INIT_QUEUE(Q); ENQUEUE(Q,s)
3       while not EMPTY(Q)
4           do u := QUEUE_FIRST(Q)
5               for každý uzel v ∈ Adj[u]
6                   do RELAX_Q(u,v,w)

RELAX_Q(u,v,w)
1       if d[v] > d[u] + w(u, v)
2           then p[v] := u
3                d[v] := d[u]
4                ENQUEUE(Q,v)
```

Lze-li $d[v]$ zmenšit, pak to provedeme, upravíme uzlu *v* předchůdce a uložíme jej do fronty. Na rozdíl od původní verze, kde byl počet průchodů hlavním cyklem explicitně stanoven, je nyní ukončení algoritmu vázáno na vyprázdnění fronty Q. V případě existence cyklů se zápornou w-délkou dostupných z uzlu *s* se však fronta nikdy nevyprázdní, a výpočet by tedy neskončil. Tento nedostatek lze odstranit, a tak získáme variantu se stejnými vlastnostmi ohledně záporných cyklů,

jako měl původní algoritmus. Asymptotická výpočetní složitost této varianty je rovněž ***O*(nm)**, neboť v nejhorším případě se do fronty Q budou opakovaně dostávat všechny uzly grafu.

Ačkoli je to nejlepší časová složitost, známá pro algoritmy hledající nejkratší cesty, v praxi je Bellmanův-Fordův algoritmus často pomalejší než jiné metody. Ve velké většině případů bude pro praxi efektivnější využít určité heuristiky rodičovských uzlů. Při nichž vybíráme uzel *v*, jen pokud P(*v*) není ve frontě.

6.8 Informované metody prohledávání

Při třídě problémů řešící problém nejkratších cest na rozsáhlých grafech (např. silniční síť) o jejichž vlastnostech nemáme žádnou pomocnou informaci, nemůžeme výběr uzlů nijak cílevědomě směrovat. Jinak je tomu v případě, že charakter úlohy dovolí definovat nezápornou hodnotící funkci *f*, jejíž nízké hodnoty vyjadřují blízkost k cílovému uzlu. Hodnotící funkce tedy dovoluje expandovat jen nejperspektivnější uzly, a tak postupovat žádoucím směrem. Rychlost postupu ovšem podstatně závisí na kvalitě hodnotící funkce - čím kvalitnější heuristiky pro řešení úlohy se v ní uplatní, tím efektivněji dokáže vést hledání k cíli. Více k následujícímu textu lze nalézt v [1].

Nejjednodušší formu využití hodnotící funkce představuje **gradientní algoritmus** (hillclimbing), který vychází z prohledávání do hloubky. Při expanzi uzlu se následníci seřadí podle hodnotící funkce a do zásobníku otevřených uzlů se uloží tak, že na vrcholu bude nejperspektivnější z nich. Díky tomu postupuje hledání směrem k extremní hodnotě - může se ovšem jednat jen o extrém lokální, který je stále od cíle vzdálen.

Systematičtějším způsobem využívá hodnotící funkce algoritmus **uspořádaného prohledávání**. Pracuje podobně jako gradientní algoritmus, ale otevřené uzly ukládá do prioritní fronty podle jejich ocenění hodnotící funkcí. K expanzi se pak vybírá otevřený uzel s nejmenší hodnotou f - ale to je vlastně nám dobře známý Dijkstrův algoritmus, v němž se namísto (odhadované) vzdálenosti od počátku používá právě hodnotící funkce f! Je-li výpočet hodnoty $f(u)$ nějak závislý na hodnotě f pro předchůdce uzlu, je nutné využít i náležitě upravenou operaci relaxace hrany a upravovat pro dříve generované uzly jejich ohodnocení a zařazovat je znovu do fronty otevřených uzlů.

Až dosud jsme předpokládali, že hodnotící funkce je nějakou ad hoc stanovenou mírou přiblížení k cílovému stavu. Tato funkce ale není zcela spolehlivá a řídit prohledávání pouze podle ní může způsobit, že se expandují uzly velmi vzdálené jak od počátku, tak i od cíle. Výhodné je tedy uvažovat hodnotící funkci ve tvaru $f(u) = g(u) + h(u)$, kde $g(u)$ představuje vzdálenost uzlu u od počátku s a $h(u)$ vzdálenost od u k cíli t. Hodnotící funkce pak vyjadřuje délku cesty řešení procházející uzlem u. Vzdálenost zde (podobně jako u Dijkstrova algoritmu) chápeme v obecném smyslu, ne nutně jako počet hran nejkratší cesty. Operátory úlohy mohou mít totiž přiřazeny různé ceny. Algoritmus uspořádaného prohledávání s takto koncipovanou hodnotící funkcí se nazývá **algoritmus A**.

Algoritmus A tedy prodlouží takovou cestu, která se v daném okamžiku zdá být součástí nejkratší cesty řešení. Je ovšem třeba vyřešit podstatný detail - jak se mají počítat hodnoty $g(u)$ a $h(u)$ definující hodnotící funkci. V praktických úlohách neznáme pravidla přesného výpočtu těchto hodnot, neboť jinak bychom patrně byli schopni úlohu řešit přímo a nikoliv pomocí hledání. Namísto přesných hodnot $g(u)$ a $h(u)$ se tedy musíme spokojit s aproximacemi: $g'(u)$ je odhad vzdálenosti (ceny přechodu) od počátku do uzlu u a $h'(u)$ je odhad vzdálenosti od u k cíli t. Zatímco za hodnoty $g'(u)$ je možné použít průběžně zpřesňovanou hodnotu vzdálenosti počítanou jako v Dijkstrově algoritmu, výpočet hodnot $h´(u)$ je zcela závislý na existenci nějakých kvantifikovatelných

heuristických pravidel, která by dokázala odhadovat vzdálenosti v dosud nevygenerované části stavového prostoru. Funkce $h'(u)$ je tedy numerickým vyjádřením naší heuristické informace, a tak se označuje jako heuristická funkce. Algoritmus A, který se opírá o aproximovanou hodnotící funkci $f'(u) = g'(u) + h'(u)$,budeme nazývat **algoritmem A*.**

Až dosud jsme při hledání cesty řešení nebrali ohled na výslednou délku nalezené cesty. Pokud je součástí zadání požadavek nalézt nejkratší cestu, musíme zjistit, zda a za jakých předpokladů zajistí použití určitého algoritmu splnění této podmínky. Je např. zřejmé, že prohledáváním do šířky (popř. obdobou Dijkstrova algoritmu pro ohodnocené operátory) se nalezne nejkratší cesta, při prohledávání do hloubky to není zaručeno. Algoritmus, který zaručuje nalezení nejkratší cesty řešení (pokud vůbec nějaká cesta existuje), se nazývá **přípustný algoritmus prohledávání**. Všimneme si nyní podmínek přípustnosti algoritmu A*, který pro úplnost vyjádříme ve zjednodušené podobě.

Algoritmus A* je tedy přípustný za poměrně slabých předpokladů o použité heuristické funkci $h'(u)$. Bohužel v praktických úlohách nelze často ověřit ani takto jednoduchou podmínku. Triviální volba $h'(u) = 0$ zaručuje přípustnost, ale v tomto případě dostáváme neinformovaný Dijkstrův algoritmus. Intuitivně je jasné, že větší efekt pro hledání má heuristická funkce, která se co nejvíce zdola přibližuje k ideální funkci $h(u)$.

Jestliže ověření přípustnosti heuristické funkce bylo obtížné, pak je potvrzení konzistence prakticky vyloučeno. Uvedené teoretické výsledky mají tedy jen velmi omezený praktický význam. Konkrétnější odhady počtu uzlů expandovaných při použití algoritmu A* je možno získat za velmi zjednodušujících předpokladů o struktuře stavového prostoru.

Popsanými algoritmy se zdaleka nevyčerpávají všechny vyvinuté varianty heuristického prohledávání. Existuje např. obousměrná verze algoritmu A* nazvaná SOH (symetrické obousměrné hledání) (viz [5]), která je přípustná a kterou lze docílit nižšího počtu expandovaných uzlů než při prohledávání jednostranném. Zajímavá je rovněž varianta iterativního prohlubování IDA* (iterative deepening A*), při níž se na počátku nastaví prahová hodnota hodnotící funkce a uzly s vyšší hodnotou se neuchovávají. Pokud hledání neuspěje, nový práh se nastaví na nejblíže vyšší hodnotu dosaženou v minulém průchodu. Podrobnější přehled základních metod hledání lze nalézt v [25], popis zahrnující i víceprocesorové metody je k dispozici např. v [28].

Uvažujeme problém hledání nejkratší cesty z jednoho bodu do jiného (P2P) ve velkém řídkém grafu, jehož typické aplikace zahrnují plánování cesty pro auta, kola a turistiku nebo hledání spojení ve veřejné dopravě, prostorové databáze a hledání na webu.

Uvažujme problém nalezení nejkratší cesty mezi dvěma uzly v orientovaném grafu (P2P), jehož základní aplikace je poskytování řídících instrukcí jako např. služby mapy a GPS. Většina algoritmů umožňuje hledat jen na malé části grafu a doba výpočtu záleží na počtu navštívených vrcholů. V aplikacích často potřebujeme najít řešení v obrovském prostoru. Heuristické hledání často nalézá řešení jen na malém podprostoru. Při hledání užívá odhady vzdálenosti k cíli k výběru následujícího uzlu na cestě. Pohl v [23] studoval vztahy mezi A* a Dijsktrou vzhledem k P2P problému. A pokud je omezení v A* proveditelné, pak je A* ekvivalentní s Dijkstrou na grafech s nezápornými ohodnoceními hran. Předzpracování probíhá na malém (konst.) množství mezníků. Následuje výpočet a uchování nejkratší cesty mezi všemi vrcholy a každým z těchto mezníků. Dolní hranice jsou vypočitatelné v konstantním čase za použití vzdáleností v kombinaci s trojúhelníkovou nerovností.

V [22] je uveden algoritmus hledání nejkratší cesty používající A* hledání v kombinaci s novou grafovou technikou dolního ohraničení pomocí mezníků a trojúhelníkové nerovnosti počítající optimální nejkratší cesty na orientovaném grafu. Tato technika je efektivní především svým A* hledáním s euklidovskými hranicemi pro problémy silničních sítí a na třídě umělých problémů.

V praxi nejznámější Dijkstrův algoritmus heuristicky zrychluje hodně technik, zatímco optimálnost řešení může stále vyhovovat.Ve většině studií jsou takové techniky individuálně porovnávány. Podle [50] zde uvedeme čtyři známé techniky a jejich možné kombinace. Jejich

zkoumání je v [50] provedeno na několika skutečných automapách, sítích veřejné dopravy a na třech náhodně generovaných grafech. Obvykle taková zlepšení Dijkstrova algoritmu nemohou být prokazatelně asymptoticky rychlejší než originální algoritmus implementovaný pomocí binární haldy. Nicméně empiricky může být ukázané, že vskutku zlepší dobu drasticky pro mnoho reálných dat. Během poslední doby se vyvinulo několik nových technik, které vypočítávají a uchovávají dodatečnou informaci o nejkratších cestách, které využíváme k redukci doby řešení při shortest-path dotazu. Průzkum těchto technik provedli Willham a Wagner v [51].

6.8.1 Goal-directed search

Technika goal-directed search využívá potenciální funkci na množině uzlů. Dané hranové váhy jsou modifikované k tomu, aby nasměrovaly hledání k cílovému uzlu. Potenciál musí plnit podmínku pro každou hranu *h*, takovou, že její délka *d*(*h*) je nezáporná, aby bylo zaručeno optimální řešení. Více o této technice v [52].

6.8.2 Bidirectional search

Nebo-li obousměrné hledání. Současně běží dvě varianty Dijstrova algoritmu - jedna ze zdroje k cíli a jedna z cíle ke zdroji. Obrácená varianta je aplikována na obrácený graf, tj. graf se stejnou množinou uzlů a s obrácenou množinou hran. Udržuje dva vyhledávací prostory a obě části algoritmu se zastaví, když se jejich vyhledávací horizonty setkají (tzn. když jeden uzel je označený jako uzavřený oběma variantami). Tento uzel leží na nejkratší cestě ze zdroje a na nejkratší cestě od cíle. Více o této technice v [53]. Pohl v experimentech [54] ukázal, že hledaný prostor může být redukován. Navíc je i výhodná kombinace této techniky s goal-directed search.

6.8.3 Multilevel approach

Tento přístup využívá hierarchické zdrsnění daného grafu. Tato metoda běží na podgrafu rozšířeného vstupního grafu. Vyžaduje předzpracování, ve kterém je vstupní graf rozložen do l+1 ($l \geq 1$) úrovní a obohacený dodatečnými hranami představující nejkratší cesty mezi uzly. Rozklad závisí na množině uzlů v každé úrovni l. Tyto množiny mohou být stanovené s různými kritériemi - např. nejvyšší stupeň v grafu. K grafu mohou být přidané tři různé druhy hran: upward - vycházející z uzlu, který není vybraný v jedné úrovni k uzlu vybraném v úrovni, downward - vycházející z vybraného k nevybraným uzlům a level edges - přecházející mezi vybranými uzly v jedné úrovni. Váha takové hrany je délka nejkratší cesty mezi koncovými uzly. K nalezení nejkratší cesty mezi dvěma uzly pak postačí Dijstrův algoritmus, aby uvážil relativně malý podgraf multiúrovňového grafu (jistá sada upward a downward hran a sada úrovňových ran procházení v maximální úrovni, které musí být vzaté pro dané zdrojové a cílové uzly). V závislosti na dotazu pak jen malý zlomek ze všech hran použijeme k nalezení nejkratší cesty. Tento princip je výhodný pro automapu a grafy veřejné dopravy.

6.8.4 Shortest path containers

Tyto kontejnery poskytují nezbytnou podmínku pro každou hranu, zda bude uznávaná během hledání.Přesněji hraničící box všech uzlů, které jsou dosažitelné na nejkratší cestě používající tuto hranu jsou uložené. Vyžaduje předzpracování počítající strom všech nejkratších cest. Pro každou hranu $h \in H$ vypočteme množinu S(h) toho uzlu, kde nejkratší cesta začíná hranou h. Použitím daného rozvržení pak skladujeme pro každou hranu bounding box S(h) v asociativním poli C s indexem množiny H. To pak dostačuje k tomu, aby vykonal Dijkstrův algoritmus na podgrafu indukovaného hranami h s cílovým uzlem zahrnutým v C[h]. Tento podgraf může být stanovený za běhu vyloučením všech dalších hran v hledání. Lze si to představit tak, že kontejner je dopravní značka charakterizující region do kterého vedou. Vyhledávací prostor může být ořezaný ignorováním hran, které nepřispívají k nejkratší cestě.

Goal-directed search a shortestpath containers a pár dalších přístupů jsou použitelné jen jestliže známe nebo poskytneme strukturu grafu.Multilevel approach a shortest path containers vyžadují předzpracování, vypočítávající dodatečné hrany a kontejnery v tomto pořadí. Kombinace těchto čtyř technik je velmi přirozená, neboť všechny modifikují vyhledávací prostor Dijkstrova algoritmu nezávisle na sobě. Otázkou je zda vyhledávací prostor kombinace je menší než prostor jednotlivé techniky.

Dalším bodem je dodatečné úsilí potřebné k redukci vyhledávacího prostoru. V goal-directed search musí být váhy hran vypočítány během hledání - což zvyšuje dobu běhu algoritmu danou za navštívené hrany. Většinou se volí kompromis mezi redukcí hledaného prostoru a dodatečného úsilí za hranu ve vyhledávacím prostoru.

Asi nejznámější kombinací technik je algoritmus **ALT** (více v [22]) zlepšující goal-directed search předpočítáním informací o nejkratší cestě z mezníků v kombinaci s obousměrným hledaním.

6.9 Goldbergův–Radzikův algoritmus

Předpokládejme, že uzly u a v jsou otevřené a existuje cesta z u do v v přípustném grafu obsahující záporně ohodnocenou hranu. Pak je lepší skenovat uzel u před uzlem v, od okamžiku kdy víme, že $d(v)$ je větší než opravdová vzálenost od s do v. Na této myšlence je založen algoritmus Goldberga-Radzika (více o něm v reportu [39]).

Pro jednoduchost popisu algoritmu, předpokládáme, že graf G nemá žádné záporně ohodnocené cykly ani cykly s nulovou délkou. Goldbergův-Radzikův algoritmus udržuje množinu otevřených uzlů ve dvou množinách (označme je třeba A a B). Každý otevřený uzel je v právě jedné z nich. Při inicializaci je množina A prázdná a množina B obsahuje vrchol s. Na začátku každého průchodu, algoritmus využívá množiny B (z uzlů, které mají být skenované během průchodu) k výpočtu množiny A. A poté množinu B vyprázdní. A je lineárně uspořádaná množina. Během průchodu jsou prvky odstraněny dle jejich uspořádánání z množiny A a označeny jako uzavřené. Nově otevřené uzly přidáme k množině B. Průchod končí je-li množina A prázdná. Algoritmus končí je-li na konci průchodu množina B prázdná.

Nyní si uvedeme, jak algoritmus počítá množinu A (z množiny B):

1) každý vrchol $u \in B$, který nemá žádnou vycházející záporně ohodnocenou hranu, vymaže z množiny B a označí ho jako uzavřený.

2) množina A se stává množinou uzlů dosažitelných z B v G. Označíme všechny uzly v A jako otevřené.
3) množinu A topologicky seřadíme pro každý pár uzlů $u,v \in$ A, tak že $(u, v) \in$ G, uzel u je předchůdce uzlu v.

Myšlenka v bodu 3) odpovídá principu **parent-scan algoritmu** (vylepšení Bellmanova-Fordova algoritmu, viz níže), ale je rozšířena na všechny hrany v grafu (ne jen na přímého předchůdce). Optimalizace tudíž zavisí na faktu, že uzel bude nejdříve skenovaný, jestliže jiný uzel z množiny A jako předchůdce tohoto uzlu, již skenovaný byl. Abychom dodrželi definované pořadí skenování, bude množina A před každým krokem lineárně seřazena. Díky tomu je v některé literatuře tento algoritmus označován jako **topological-scan** nebo **topological sort algoritmus**. Implementace tohoto algoritmu je nutně o něco komplexnější než originální implementace BF, neboť nás svazuje právě nutnost topologického setřídění.

Algoritmus 6.9 Goldbergův-Radzikův algoritmus

GOLDBERG_RADZIK(G,s)

```
1       A = ∅
2       B = {s}
3       while not EMPTY(B)
4           vypočti A z B
5           B = ∅
6           for všechny vrcholy u ∈ A do
7               for každou hranu (u, v) ∈ H
8                   do RELAX(u,v,w)
9               B = B ∪ {otevřené uzly z RELAX(u,v,w)}
```

Algoritmus dosazuje časové složitosti jako Bellmanův-Fordův algoritmus, tedy běží v O(nm). Ačkoliv se zdá být tímto ekvivalentní k originálnímu Bellmanovu-Fordovu algoritmu, u mnohých problémů je doba běhu algoritmu podstatně rychlejší.

Nyní předpokládejme, že G má záporné nebo nulové cykly. V tomto případě G nemusí být acyklický. Nicméně pokud G obsahuje záporný cyklus, lze algoritmus ukončit. Má-li G cyklus nulové délky, můžeme zkrátit takové cykly a pokračovat ve výpočtu.To snadno uděláme při udržení složitosti algoritmu (viz [38]).

Implementaci Goldberova-Radzikova algoritmu lze zjednodušit použitím DFS, k výpočtu topologického uspořádání v přípustném grafu (viz např. [11]). Místo uzavření cyklů nulové délky, jednoduše ignorujeme zpětné hrany objevené během DFS. Výsledné řazení je v přípustném grafu bez ignorovaných hran. Tato změna neovlivní správnost ani složitost výše uvedeného algoritmu.

Algoritmus lze modifikovat. Pokud při řazení pomocí DFS bude hrana (v,w) zkoumaná první skenovaná na nejkratší cestě tzn. $d(v) + w(v, w) < d(w)$ pak $d(w)$ nastavíme na hodnotu $d(v) + w(v, w)$ a P(w) na v. Tato modifikace běží rovněž v O(nm). Na acyklickém grafu algoritmus skončí po prvním průchodu a běží v O(m+n).

6.10 Incremental Graph Algorithms

V tomto odstavci popíšeme dva algoritmy. První nezávisle vyvinul Pape a Levit [31]. Druhý navrhl Pallottino [32], který představil rámec incremental graph algoritmů tím, že sjednotil tyto dva

algoritmy. Oba algoritmy udržují množinu otevřených uzlů jako dvě podmnožiny S_1 a S_2, první obsahuje otevřené uzly, které byly skenované nejméně jednou a druhá, ve které uzly skenované ještě nebyli. Další uzel na sken je vybraný z S_1. Je-li S_1 prázdná vybereme uzel z S_2. S_1 nazýváme vysoce prioritní množinu a S_2 množinu s nižší prioritou.

Pape-Levit algoritmus udržuje S_1 jako zásobník a S_2 jako frontu. Tento algoritmus je většinou implementovaný s použitím datové struktury dequeue, která je jako fronta dovolující vložení ne jen na konec, ale i na začátek. Více viz [21] a [32]. Tento algoritmus má nejhůře exponenciální časovou složitost $O(n2^n)$ [35].

Pallottinův algoritmus udržuje S_1 a S_2 pomocí front Q_1 a Q_2 a běží nejhůře v $O(n^2m)$.V praxi je více robustnější než Pape-Levit.

6.11 Prahový (Threshold) algoritmus

Glover a spol. navrhl v [36] následující metodu, která kombinuje myšlenky Bellmana-Forda, Dijkstry a incremental graph algoritmu (viz [21, 37, 69]). Tato metoda rozděluje množinu otevřených uzlů na dvě podmnožiny NOW a NEXT, které jsou udržované jako fronta (inspirace z předchozího algoritmu). Na začátku každé iterace je NOW prázdná. Algoritmus také udržuje parametr t – tzv. práh, který je nastaven jako vážený průměr minimálních a průměrných hodnot d[u] v množině NEXT. Během iterace algoritmus přenese uzly v s $d[v] \leq t$ z NEXT do NOW a skenuje uzly v NOW. Uzly, které se stanou otevřené během iterace jsou přidané do množiny NEXT. Algoritmus končí, když je fronta NEXT prázdná na konci iterace. Pokud je ohodnocení hran nezáporné, algoritmus běží v $O(nm)$ dle [12].

6.12 Další algoritmy

Od roku 1959 jsou všechny teoretické výzkumy v problematice hledání nejkratších cest z uzlu s do všech ostatních založeny na Dijkstrově algoritmu navštěvující uzly v pořadí zvyšující se vzdálenosti od s. Dijsktrův algoritmus je založený na řazení vzdáleností - a to neumíme pro orientované grafy udělat v lineárním čase. Nicméně pro neorientované grafy s s celočíselným nezáporným ohodnocením hran máme deterministický algoritmus v lineárním čase a prostoru. Tento algoritmus se vyvaruje úzkému hrdlu řazení vytvořením hierarchické bucket struktury, identifikující páry vrcholů, které mohou být navštívené v nějakém pořadí.

Řada rychlých algoritmů byla vyvinuta pro standardní model RAM (Random access machine) s velikostí slova log n se standardní instrukční sadou. Téměř všechny tyto algoritmy jsou založené na Dijkstrově algoritmu a zlepšují datovou strukturu prioritní fronty. Nalezení algoritmů s lineárním časem jsou stále top vývoje. Zatímco Thorup (viz [45]) vyvinul algoritmus O(n+m) pro neorientované grafy, kde ohodnocení hran je v intervalu 0 až w-1 (w=délka slova), aktuálně nejlepší čas pro orientované řídké grafy na RAM modelu je $O(n+m \log \log n)$.

Ve většině iterací Bellmanova-Fordova algoritmu je operace relaxace nadbytečná. Při inicializaci přidáme všem uzlům proměnnou s(v), dle níž později rozhodneme zda daný uzel budeme skenovat. Prvním takovým vylepšením je parent-scan algoritmus (zmíněn již u Goldbergova-Radzikova algoritmu), založený na myšlence: před skenováním uzlu provedeme nejprve skenování jeho předchůdce. Před každým skenováním uzlu prověříme, zda jeho předchůdce je ve frontě. Jen

tehdy bude uzel skenovaný. Na rozšíření myšlenky je založen algoritmus Goldberga-Radzika (viz příslušný odstavec). Klasický Bellmanův-Fordův algoritmus a všechny jeho odvozené deriváty potřebují O(mn), nicméně některé tyto algoritmy ukazují dobré praktické chování pro hledání nejkratších cest na řídkých grafech – více v [12] a [41]. Lze najít lineární algoritmy pro hledání nejkratších cest, většinou jsou však omezeny na rovinné grafy bez záporně ohodnocených hran. Pro případ povolených záporných hran existuje algoritmus v čase $O(n^{4/3} \log n\, L)$, kde L je absolutní hodnota nejvíce záporné hrany – více v [42].

Většina významných algoritmů může být viděna jako speciální případ modelu nejkratších cest daný Gallem a Pallottinem v [21]. V tomto modelu udržujeme vektor $(d_1, d_2, \ldots, d_n)$ s inicializací danou $d_1 = 0$, $d_i = \infty$ pro všechny $i \neq 1$ a množina uzlů nazvaných seznam kandidátů, s inicializací: $V = \{1\}$. Algoritmy postupují v iteracích a skončí když je V prázdné, typická iterace je následující: odstraň uzel *i* ze seznamu kandidátů V pro každou hranu $(i, j) \in H$ a $j \neq 1$ jestliže $d_j > d_i + a_{ij}$ nastav $d_j = d_i + a_{ij}$.

6.13 Porovnání algoritmů pro řešení problému nejkratší cesty z jednoho uzlu

Mezi základními metodami v této kapitole figurují Dijkstrův a Dantzigův algoritmus, ze stejného období a s podobným principem. Porovnání z hlediska historie jsme si ukázali již v kapitole o Dantzigově algoritmu. Což takhle je porovnat i z hlediska experimentů s jejich implementacemi.

Jako třetí k nim zvolíme heuristické vylepšení základního algoritmu (v tomto případě Bellmanova-Fordova) a to Goldbergův-Radzikův algoritmus.

Experimenty provedeny na počítači s 733 MHz procesorem, operačním systémem Windows XP a s 384 MB operační paměti. Kódy psané v C a kompilované pomocí Mingw kompilátoru ve vývojovém prostředí Dev-C++ ve verzi 4.9.9.2. Výsledky byly ověřeny na školním linux serveru Merlin.

Implementace používají seznamovou reprezentaci sousednosti vstupního grafu (seznam ohodnocených hran). Experimentujeme s několika reprezentacemi grafu se zvyšujícím se počtem *n* vrcholů. V praxi mají problémy často specifickou strukturu, proto neměníme jen počet vrcholů vstupního grafu, ale i počet hran *m*. Generovány jsou tři různé typy grafu s ohledem na počet hran – řídké, husté a úplné (lépe řečeno úplné acyklické grafy, tzn. že jejich matice **W** má pod diagonálou nekonečna). Všechny reprezentace mají minimálně z prvního vrcholu dosažitelné všechny ostatní uzly grafu. Další vlastností všech reprezentací, jak již bylo zmíněno, je acykličnost a v poslední řadě mají všechny hrany kladné ohodnocení. V experimentech tedy nenastane situace, kdy by graf obsahoval cyklus nebo záporně ohodnocenou hranu (neboť většina zvolených algoritmů se ze záporným ohodnocením nevyrovná). Snahou bylo, aby jednotlivé implementace různých algoritmů byly dělané jednotně, aby bylo možné porovnat smysluplně dobu běhu. V následujících tabulkách jsou výsledky experimentů, máme dobu běhu implementace v sekundách (uživatelský čas procesoru bez vstupních a výstupních operací) a počet operací relaxace. Pro zisk referenčních dat provádíme 5 až 10 běhů implementací na stejných vygenerovaných reprezentacích grafu, které pak průměrujeme počtem běhů.

Řídké grafové reprezentace:

	Dijkstrův algoritmus		Dantzigův algoritmus		Goldbergův-Radzikův alg.
n / m	s	poč.relaxací	s	poč.relaxací	poč.relaxací
100 / 193	<0,01	193	<0,01	1765	198
200 / 390	<0,01	390	0,02	6993	390
300 / 586	<0,01	586	0,09	14551	607
400 / 792	<0,01	792	0,22	22830	797
500 / 989	<0,01	989	0,42	38787	963
600 / 1196	<0,01	1196	0,64	50286	1297
700 / 1391	<0,01	1391	0,91	63456	1399
800 / 1591	0,01	1591	1,46	89067	1701
900 / 1790	0,01	1790	1,85	116664	1829
1000 / 1995	0,01	1995	1,99	129742	1981

Tabulka 6.1: Doba běhu jednotlivých implementací algoritmů a počet operací relaxace

Husté grafové reprezentace:

	Dijkstrův algoritmus		Dantzigův algoritmus		Goldberův-Radzikův alg.
n / m	s	poč.relaxací	s	poč.relaxací	poč.relaxací
100 / 3548	0,01	3548	0,04	2731	5584
200 / 14327	0,02	14327	0,42	14959	24010
300 / 31980	0,05	31980	1,49	30427	55441
400 / 56933	0,13	56933	7,97	47784	102338
500 / 88892	0,35	88892	33,57	78640	170969
600 / 127903	0,71	127903	79,31	107610	243640
700 / 174273	1,05	174273	152,78	149013	326113
800 / 227917	1,46	227917	291,91	176829	436422
900 / 287865	1,92	287865	460,93	228604	577234
1000 / 355448	2,73	355448	-	-	666053

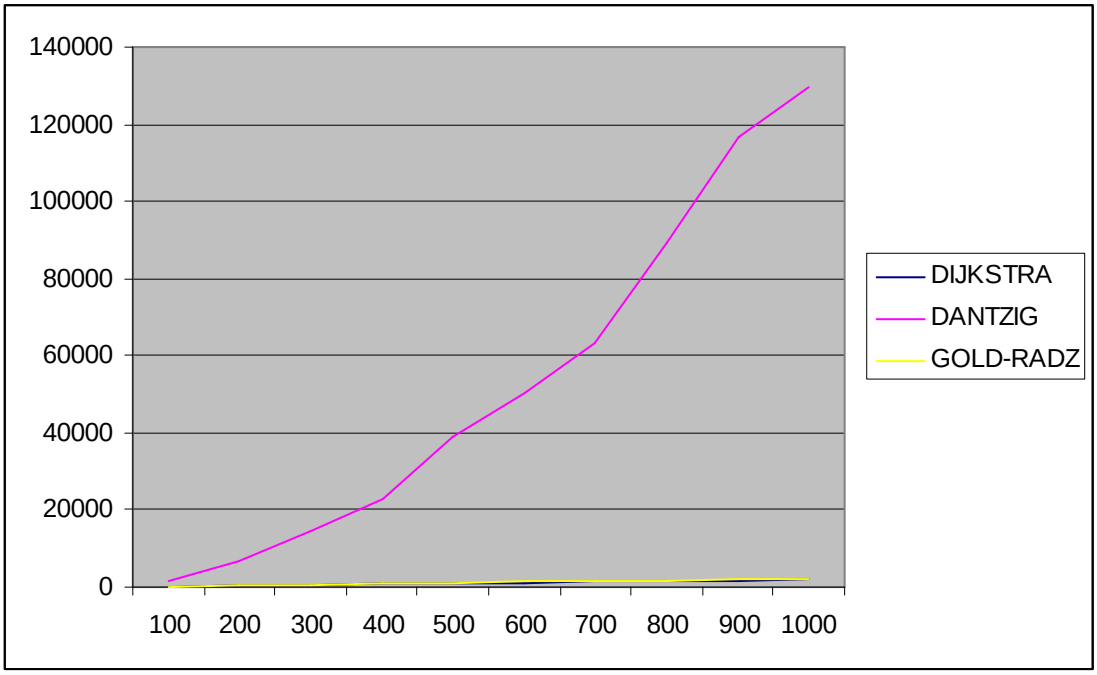
Tabulka 6.2: Doba běhu jednotlivých implementací algoritmů a počet operací relaxace

Úplné acyklické grafové reprezentace:

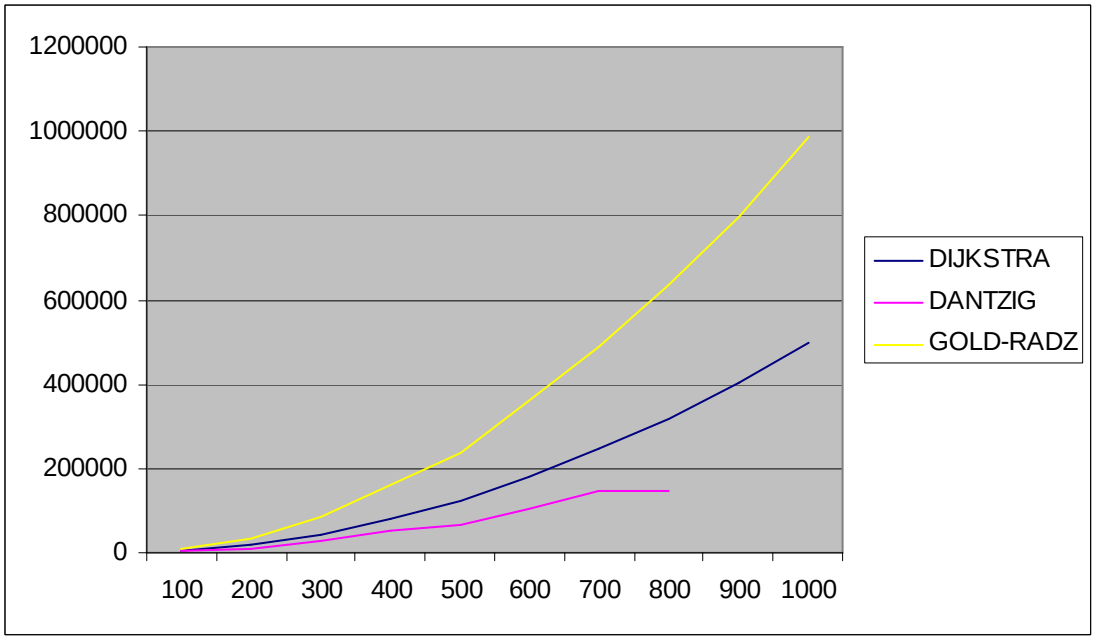
	Dijkstrův algoritmus		Dantzigův algoritmus		Goldbergův-Radzikův alg.
n / m	s	poč.relaxací	s	poč.relaxací	poč.relaxací
100 / 4950	0,01	4950	0,04	3345	8891
200 / 19900	0,02	19900	0,75	9965	34383
300 / 44850	0,08	44850	2,22	29286	84902
400 / 79800	0,22	79800	17,27	51280	159405
500 / 124750	0,6	124750	62,79	68032	237856
600 / 179700	1,02	179700	122,7	102763	359930
700 / 244650	1,42	244650	230,16	149008	488951
800 / 319600	2,07	319600	489,04	146571	636387
900 / 404550	2,49	404550	-	-	797047
1000 / 499500	3,47	499500	-	-	986164

Tabulka 6.3: Doba běhu jednotlivých implementací algoritmů a počet operací relaxace

V grafech 6.1 a 6.2 si všímáme počtu operací relaxace pro jednotlivé implementace v řídkých respektive úplých acyklických reprezentací grafu. V Dijkstrově algoritmu relaxujeme každou hranu vstupního grafu jedenkrát, zatímco u většího počtu hran v grafu Goldbergův-Radzikův algoritmus relaxuje některé hrany vícekrát. U Dantzigova algoritmu naopak redukujeme počet relaxací u větších počtů hran ve vstupním grafu pomocí výběru vhodných kandidátů.



Graf 6.1: Počet operací relaxace v řídkých grafech

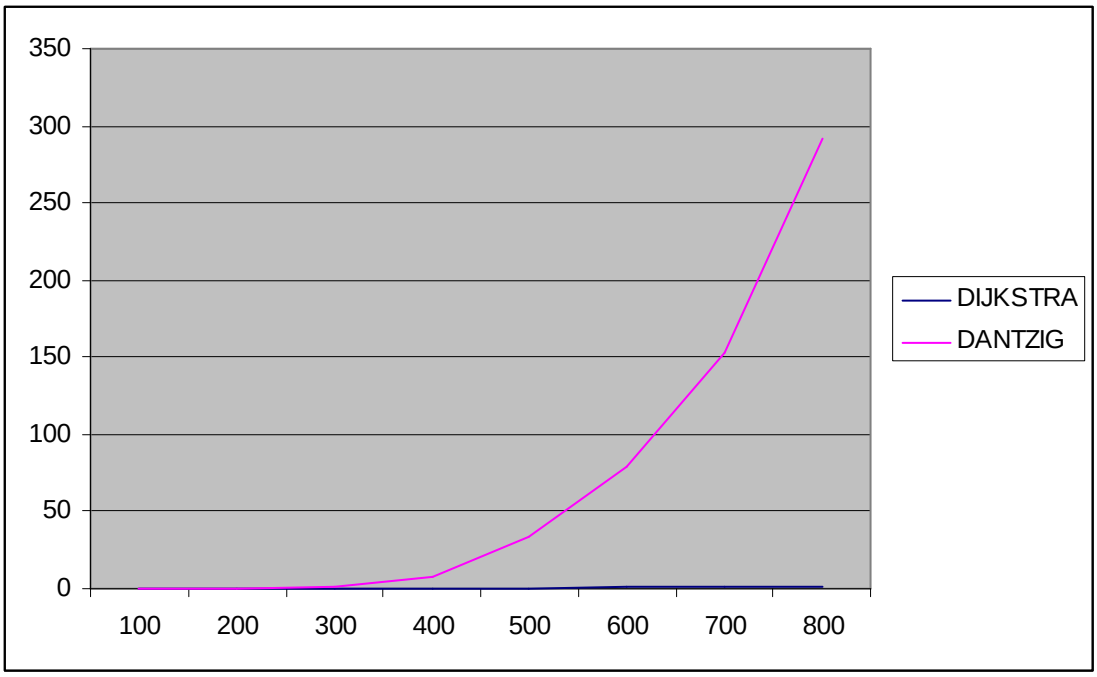


Graf 6.2: Počet operací relaxace v úplných acyklických grafech

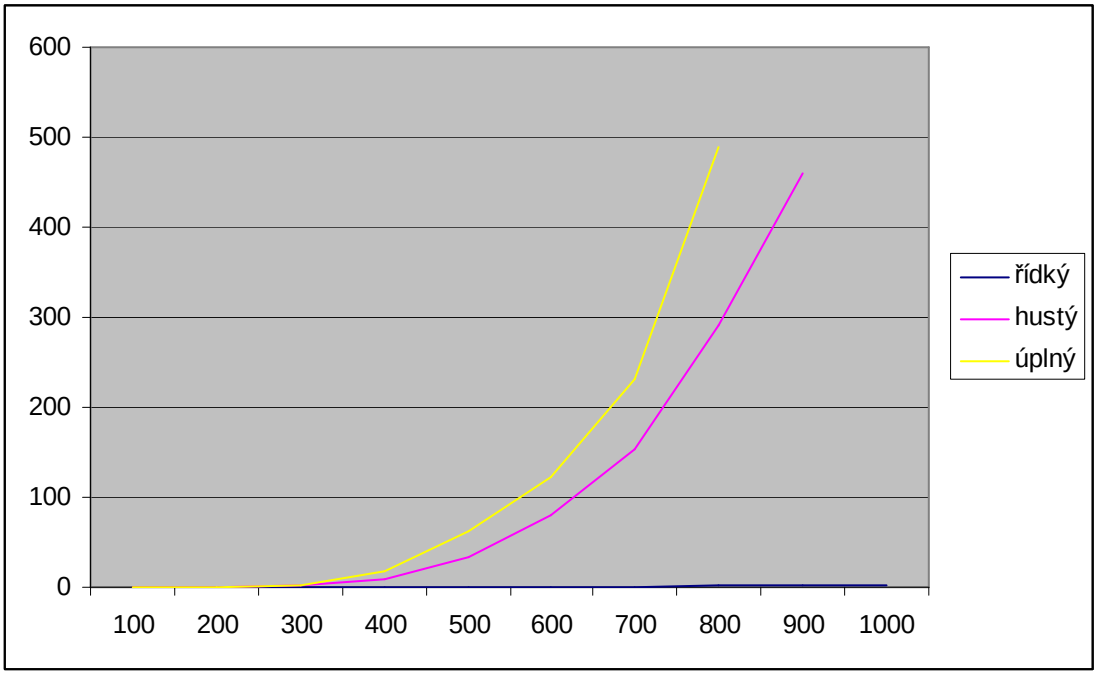
V grafu 6.3 je zobrazena doba běhu v sekundách pro implementaci Dijkstrova a Dantzigova algoritmu pro husté grafové reprezentace. Doba běhu implementace Goldbergova-Radzikova algoritmu je pro stejnou reprezentaci vstupního grafu pod 0,01 sekundy, tudíž ani není zobrazena v

grafu a tabulkách výsledků. To odpovídá teoretických odhadům časové složitosti. Dantzigův algoritmus je výrazně horší oproti implementaci Dijktrova algoritmu, neboť výběr vhodného kandidáta je nesmírně časově náročná operace, která závisí na počtu hran ve vstupním grafu.

V grafu 6.4 ukazujeme závislost doby běhu implementace Dantzigova algoritmu na parametrech reprezentace grafu. Při výběru vhodných kandidátů procházíme vždy všechny hrany, což se negativně projeví na výsledném čase.



Graf 6.3: Porovnání doby běhu implementací v hustých grafech



Graf 6.4: Doba běhu implementace Dantzigova algoritmu v závislosti na typu grafu

Co se týká paměťové náročnosti jsou všechny tři algoritmy na stejné úrovni s $O(n+m)$, to odpovídá paměťové složitosti spojové reprezentace grafu. Dokonce v implementacích Dantzigova a Dijkstrova algoritmu využíváme úplně stejné struktury.

Ač jsou si principem Dantzigův a Dijkstrův algoritmus velmi podobné, v experimentech se ukázal Dantzigův algoritmus jako poražený. Jediné, v čem je v experimentech lepší, je porovnání počtu operací relaxace. Jak už bylo uvedeno, je to způsobeno díky výběru vhodných kandidátských hran. Nicméně nám tato vlastnost snižuje výrazně rychlost, jak si lze všimnout ve výsledcích experimentů. Nejspíš i to je důvod výraznějšího rozšíření Dijkstrova algoritmu v oblastech počítačového vyhodnocování.

Z důvodu porovnání implementací Dijkstrova a Dantzigova algoritmu je zvolena naivní forma Dijkstrova algoritmu, kdy je prioritní fronta implementována pomocí dynamického pole a má teoretickou časovou složitost v nejhorším případě $O(n^2)$. Prvky v této frontě nejsou řazeny a minumum tedy nalezneme až po průchodu celým polem. V implementaci Dantzigova algoritmu nám docela dost času (nezapočítaného do samotného experimentování) zabere počáteční uspořádání všech hran podle stoupající hodnoty jejich ohodnocení. K výběru kandidáta s nejmenší hodnotou procházíme množinu uzlů, do kterých již známe nejkratší cesty. Zásadní z hlediska času je ovšem redukce nepotřebných kandidátských hran, kdy procházíme celým seznamem hran. Experimenty nám jen potvrdily, že pro husté grafy je tato metoda časově nevyhovující. V implementaci Goldbergova-Radzikova algoritmu bylo využito zjednodušení popsané v kapitole u algoritmu, a tím je použití prohledávání do hloubky k výpočtu topologického uspořádání.

Správnost implementace byla ověřena pomocí výstupu jednotlivých implementací na stejnou vstupní reprezentaci grafu. Ve všech testovaných případech byl výstup všech tří implementací stejný tj. vypočteny nejkratší cesty z prvního uzlu do všech ostatních. Výstupní soubory, všechny vstupní reprezentace grafu a zdrojové soubory implementací lze nalézt na přiloženém médiu.

7 Nejkratší cesty mezi všemi páry uzlů

Problém nalezení nejkratších cest mezi všemi dvojicemi uzlů (all-pairs shortest paths problem), neboli určování vzdálenosti mezi libovolnými dvěmi uzly, by se dal jistě chápat jako *n*-krát zopakovaný problém jednodušší tj. hledání nejkratší cesty z jednoho uzlu do všech ostatních (viz předchozí kapitola). Nicméně co nám v tom vadí je časová složitost. V případě grafu s nezáporným ohodnocením hran dostaneme *n*-násobným použitím Dijkstrova algoritmu (za předpokladu realizace prioritní fronty pomocí Fibonacciho haldy) postup s asymptotickou časovou složitostí $\boldsymbol{O(n^2 \log n + nm)}$. Použití binární haldy vede na složitost $O(mn \log n)$ a při sekvenčním uložením fronty v poli získáme algoritmus kubické složitosti $O(n^3)$.

Pokud graf obsahuje i záporně ohodnocené hrany, nelze Dijkstrova algoritmu použít, takže bychom museli opakovat hledání pomocí Bellmanova-Fordova algoritmu. V tomto případě bude časová složitost nalezení nejkratších cest mezi všemi dvojicemi uzlů rovna $O(n^2 m)$, což pro husté grafy dává hodnotu $\boldsymbol{O(n^4)}$ a to není jistě vyhovující. V této kapitole si ukážeme, že lze postupovat efektivněji.

7.1 Nejkratší cesty a násobení matic

Hledáme-li nejkratší cesty (neboli vzdálenosti) z každého uzlu do každého uzlu přímo se nám nabízí maticová struktura, která nám tyto vzdálenosti jako výsledek bude schopna dodat. Výsledkem algoritmu hledání nejkratších cest mezi všemi dvojicemi uzlů je soubor $n \times n$ hodnot (*w*-vzdáleností popř. předchůdců na nejkratších cestách), pro které je přirozenou formou uložení čtvercová matice. Když budeme mít výsledek v podobě matice, je logické aby i vstupní data měla maticový formát. Protože kromě struktury grafu musíme vyjádřit i ohodnocení jednotlivých hran, vytvoříme jedinou matici, která kombinuje matici sousednosti grafu s funkcí ohodnocení *w*. Budeme předpokládat, že graf neobsahuje cykly záporné *w*-délky (z důvodů vysvětlených výše).

Nechť je dán prostý (pokud by graf obsahoval rovnoběžné hrany, lze se jich zbavit vybráním té nejmenší z nich) orientovaný graf G = <H, U> s reálným ohodnocením hran $w : H \rightarrow \mathbf{R}$. **Maticí *w*-délek** grafu G nazýváme čtvercovou matici $\mathbf{W} = [w_{ij}]$ řádu *n*, jejíž prvky jsou definovány vztahy:

$$\begin{aligned} w_{ij} &= 0 \text{ pokud } i = j, \\ w_{ij} &= w(u_i, u_j), \text{ pokud } i \neq j, (u_i, u_j) \in H, \\ w_{ij} &= 1, \text{ pokud } i \neq j, (u_i, u_j) \notin H. \end{aligned} \qquad \textbf{(7.1)}$$

Tato matice **W** nám bude graf charakterizovat. Vychází z matice sousednosti *V* a zahrnuje současně délky hran.

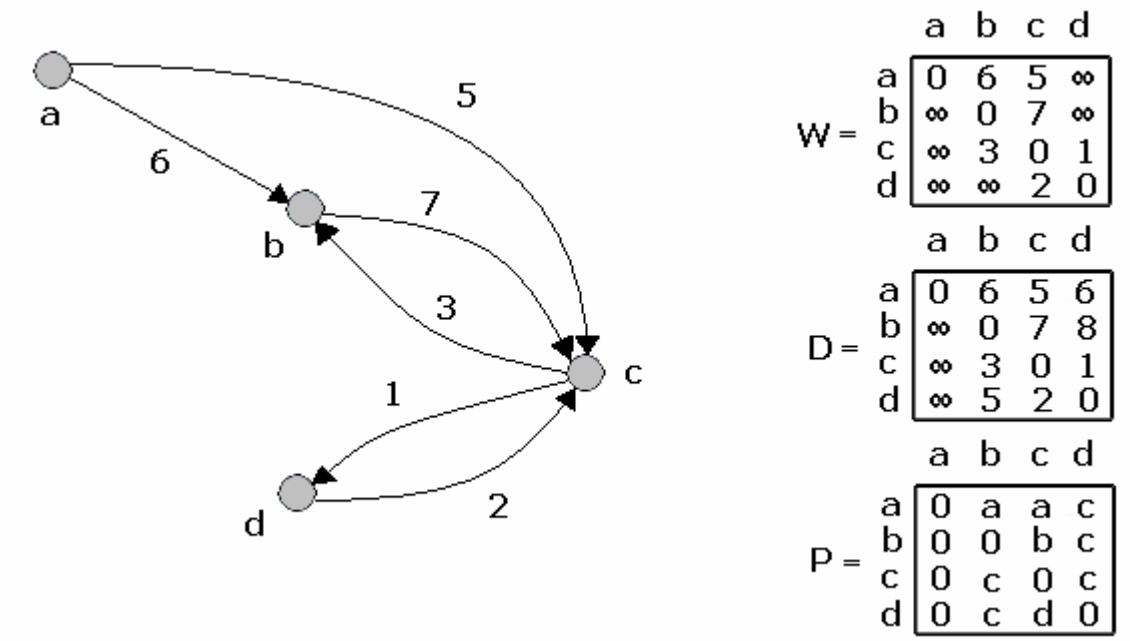
Maticí *w*-vzdáleností grafu G nazýváme čtvercovou matici $\mathbf{D} = [d_{ij}]$ řádu *n*, jejíž prvky jsou definovány vztahem:

$$d_{ij} = d_w(u_i, u_j). \qquad \textbf{(7.2)}$$

V matici **D** nalezneme, jak daleko je z uzlu *i* do *j*, když nás opravdu zajímá ta nejkratší cesta. Chceme-li znát kudy vede, musíme rozšířit pole hodnot předchůdců z Dijkstrova algoritmu, protože zde se již nejedná o předchůdce na jedné cestě, ale musíme znát předchůdce na cestách ze všech uzlů do všech a k tomu nám poslouží matice předchůdců **P**.

Maticí předchůdců grafu G (zkráceně též p-maticí) nazýváme čtvercovou matici **P** = [p_{ij}] řádu *n*, jejíž prvky jsou definovány vztahem:

p_{ij} = 0 (NIL), pokud i = j nebo neexistuje cesta z u_i do u_j **(7.3)**
p_{ij} = u_k, kde u_k je předchůdce uzlu u_j na nějaké minimální cestě z u_i do u_j



Obrázek 7.1: Matice w-délek, w-vzdáleností a předchůdců

Na obr. 7.1 ukazujeme příklad grafu a jemu odpovídajících matic *w*-délek, *w*-vzdáleností a předchůdců. Postupům, pomocí nichž se určí matice *w*-vzdáleností, se budeme věnovat později. Jejich rozšíření vedoucí k získání matice předchůdců lze nalézt v literatuře, ukážeme zde pouze význam a využití této matice. Hodnoty p_{ij}, j = 1, 2, ..., n ležící v *i*-tém řádku matice předchůdců P obsahují stejnou informaci jako (jednorozměrné) pole hodnot p[*u*] po skončení algoritmu hledání cest z uzlu *s* = u_i do všech uzlů grafu. Jim odpovídající podgraf je stromem nejkratších cest z uzlu u_i, takže výčet uzlů tvořících nejkratší cestu z u_i do u_j dostaneme pomocé následujícího algoritmu.

Algoritmus 7.1 Určení cesty pomocí p-matice

CESTA(P,*i*,*j***)**
1 **if** *i* = *j* **then** write(*i*)
2 **else if** p_{ij} = 0
3 **then** write('cesta z ',u_i,'do ',u_j,' neexistuje')
4 **else** CESTA(**P**,*i*,p_{ij})
5 write(*j*)

Nejkratší cesty v grafu o *n* uzlech (a bez záporných cyklů) neobsahují více než (*n*-1) hran. Zkusíme tedy postupně určovat *w*-délky nejkratších cest obsahujících zvyšující se maximální počet hran. Nechť pro dva různé uzly u_i, $u_j \in$ U je P : $u_i \rightarrow u_j$ nějaká nejkratší cesta obsahující nejvýše *m*

hran. Je-li uzel u_k předchůdcem uzlu u_j na této cestě, pak je také dílčí cesta P‘ : $u_i \rightarrow u_k$ nejkratší cestou z u_i do u_k obsahující nejvýše (n-1) hran. Krom toho platí w(P) = w(P') + w_{kj}.

Složitost této procedury je díky třem vnořeným cyklům a konstantní složitosti vnitřní operace určení minima rovna $\Theta(n^3)$. Její struktura je shodná jako u procedury násobení matic. Právě pomocí násobení matic je možné určit počet různých spojení určité délky mezi všemi dvojicemi uzlů. Jelikož matice $\mathbf{D}(n\text{-}1)$ je hledanou maticí nejkratších cest, můžeme její výpočet provést pomocí procedury (více viz [6]) s časovou složitostí $O(n^4)$. Algoritmus je možné implementovat tak, že jeho paměťová složitost bude nižší než $O(n^2 \log n)$, jak by vycházelo při uložení všech počítaných matic. Při výpočtu se vždy potřebuje pouze předchozí a následující matice v uvedené posloupnosti, takže vystačíme jen se dvěma maticemi řádu $n \times n$ (případně se třemi, chceme-li uchovat i matici W). Paměťová složitost bude tedy jen $O(n^2)$.

7.2 Floydův-Warshallův algoritmus

Při výpočtu posloupnosti matic $\mathbf{D}^{(m)}$ použité k získání matice w-vzdáleností postupně zpřesňujeme odhad w-délky nejkratších cest uvažováním stále delších (co do počtu hran) cest. Floydův-Warshallův algoritmus (podrobnější popis v [43]) má rovněž charakter postupného zpřesňování, ale tentokrát budeme rozšiřovat množinu přípustných vnitřních uzlů nejkratších cest. **Vnitřním uzlem** cesty P = $\langle u_1, u_2, ..., u_k \rangle$ je každý její uzel s výjimkou krajních uzlů u_1 a u_k, tedy libovolný z uzlů $\{u_2, u_3, ..., u_{k-1}\}$. Připomeňme, že stále uvažujeme pouze grafy bez záporně ohodnocených cyklů. Algoritmus najde délku nejkratších orientovaných cest mezi každou dvojicí vrcholů, navíc ze všech cest stejné délky vybere tu s nejmenším počtem hran.

Jedná se o iterační algoritmus, který si lze přestavit jako algoritmus počítající jakousi posloupnost matic. Z předchozí matice vypočteme jistým postupem v následující iteraci matici s novými hodnotami. Idea je následující. V každé iteraci zjistíme jaké jsou délky nejkratších cest mezi všemi dvojicemi uzlů, s tou omezující podmínkou, že tyto nejkratší cesty mohou procházet přes vnitřní uzly jen z množiny prvních k uzlů. Postupně počet povolených uzlů zvyšujeme. V matici $\mathbf{D}^0$ nepřipouštíme žádné vnitřní uzly (uvažujeme jen přímé hrany) a to je přesně matice w-vzdáleností **W**. To reprezentuje případ, kdy mám uzel spojený s jiným hranou neboli cestou nemající žádné vnitřní uzly. V další iteraci povolíme do cest zahrnout jeden vnitřní uzel a to jen uzel u_1. V druhé iteraci povolíme vnitřní uzly u_1 a u_2 atd. V k-té iteraci povolím, aby vnitřní uzly cest byly z množiny (u_1, …, u_k).

Nechť $\mathbf{D}^{(k)} = [d^{(k)}_{ij}]$ je matice, jejíž každý prvek $d^{(k)}_{ij}$ vyjadřuje w-délku nejkratší cesty z uzlu u_i do uzlu u_j mající vnitřní uzly pouze z množiny $\{u_1, u_2, ..., u_k\}$. Potom bude zřejmě $d^{(0)}_{ij} = w_{ij}$ a $d^{(n)}_{ij} = d_{ij}$. Od libovolné matice $\mathbf{D}^{(k-1)}$ se dostaneme k matici $\mathbf{D}^{(k)}$ prostřednictvím této úvahy:

- jestliže nejkratší cesta z u_i do u_j mající vnitřní uzly z množiny $\{u_1, u_2, ..., u_k\}$ neprochází uzlem u_k, potom se hodnota prvku $d^{(k)}_{ij}$ rovná hodnotě $d^{(k-1)}_{ij}$.

- jestliže nejkratší cesta $c : u_i \rightarrow u_j$ mající vnitřní uzly z množiny $\{u_1, u_2, ..., u_k\}$ prochází uzlem u_k, potom si ji můžeme představit rozdělenou do dílčích cest $c_1 : u_i \rightarrow u_k$ a $c_2 : u_k \rightarrow u_j$. Cesty c_1 a c_2 jsou pak nutně nejkratšími cestami mezi svými krajními vrcholy a jejich vnitřní uzly jsou vybrány pouze z množiny $\{u_1, u_2, ..., u_{k-1}\}$.

Je tedy možné formulovat následující rekurentní definici hodnot prvků matice $\mathbf{D}^{(k)}$:

$$d_{ij}^{(k)} = w_{ij} \quad \text{pokud k = 0,}$$
$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) \quad \text{pokud k} \geq 1. \qquad \textbf{(7.4)}$$

To je jádro Floydova-Warshallova algoritmu a z této rekurentní definice již můžeme tento algoritmus formulovat:

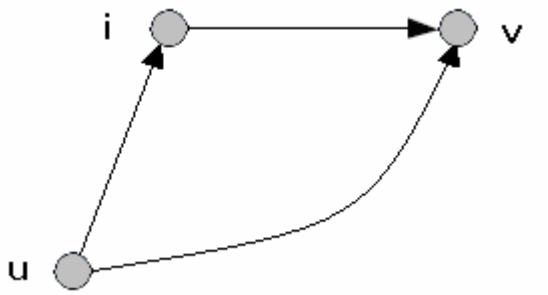
Algoritmus 7.2 Floydův-Warshallův algoritmus

FLOYD-WARSHALL(W)

1 $\mathbf{D}^{(0)} := \mathbf{W}$
2 **for** k := 1 **to** *n* **do**
3 **for** i := 1 **to** *n* **do**
4 **for** j := 1 **to** *n* **do**
5 $d_{ij}^{(k)} := \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
6 **return** $\mathbf{D}^{(n)}$

Všechny dvojice vzdáleností si nejsnáze uložíme do matice $n \times n$. Celý trik Floydova-Warshallova algoritmu spočívá v tom, že vzdálenosti nepočítáme přímo, ale v n iteracích. V i-té iteraci spočítáme matici D^i. Hodnota $D^i[u,v]$ je délka nejkratší cesty z u do v, která smí procházet pouze přes vrcholy $\{1, 2, ..., i\}$. V nulté iteraci začneme s maticí D^0. Hodnota $D^0[u,v]$ je délka hrany u, v. Matice D^0 je tedy matice vzdáleností (upravená matice sousednosti, která místo jedniček obsahuje délky hran a místo nul mimo diagonály má nekonečna). V poslední iteraci skončíme s maticí D^n, která už bude obsahovat hledané vzdálenosti, protože cesty mezi u a v smí procházet přes všechny vrcholy.

Nejkratší cesta mezi u a v, která smí procházet přes vrcholy {1, 2, ..., i}, buď projde přes vrchol i a nebo ne. V prvním případě nejkratší cesta neobsahuje i a její délka je $D^{i-1}[u,v]$. Ve druhém případě cestu rozložíme na dva úseky – před příchodem do i a po jeho opouštění. Ani jeden z úseků neobsahuje i a tak je délka této cesty $D^{i-1}[u,i] + D^{i-1}[i,v]$. Co když ale oba úseky obsahují stejný vrchol w. Složením úseků by pak vznikl tah. V tomto případě lze část tahu mezi oběma výskyty w vypustit (cyklus z w do i a zpět) a dostaneme cestu, která neobsahuje i a byla uvažována v prvním případě. Z toho plyne, že $D^j[w,w]$ je rovno nule pro každé j a w.



Obrázek 7.2: Postup výpočtu dle Floydova-Warshallova algoritmu

K výpočtu $D^i[u,v]$ potřebujeme znát jen hodnoty $D^{i-1}[u,v]$, $D^{i-1}[u,i]$ a $D^{i-1}[i,v]$, ale poslední dvě hodnoty se během i-té iterace nezmění (viz [10]). Proto můžeme nové hodnoty $D^i[u,v]$ zapisovat do stejné matice jako předchozí iteraci – přepsanou hodnotu $D^{i-1}[u,v]$ již potřebovat nebudeme. K výpočtu tak postačí jedna matice, do které budeme zapisovat všechny iterace.

V případě záporných cyklů nemůžeme použít Floydův-Warshallův algoritmus, neboť by jsme mohli dostat nesprávné řešení (řešení by nebyla cesta, ale tah, který navíc nebude optimální).

Určení asymptotické časové složitosti Floydova-Warshallova algoritmu je snadné: provedení minimalizace uvnitř vnořeného cyklu trvá konstantní čas a opakuje se n^3-krát. Algoritmus má tedy časovou složitost $\boldsymbol{O(n^3)}$. Jelikož je jeho struktura velmi podobná algoritmu pro násobení matic, lze jej

urychlit až na složitost $\boldsymbol{O(n^{2,807})}$ pomocí stejného triku jako Strassenův algoritmus zrychluje násobení matic. Nicméně existují rychlejší algoritmy, proto zde toto urychlení nebude dále probíráno (více viz [11] - kapitola 28). Algoritmus se tváří, že počítá sekvenci n+1 matic, nicméně všechny iterace matice **D** se dají počítat na jednom místě (neměníme aktuálně hodnoty na kterých v tu chvíli závisejí ostatní). Asymptotická paměťová složitost je ovšem $O(n^2)$.

Chceme spočítat vzdálenost každé dvojice vrcholů, tak můžeme n-krát použít Dijkstrův algoritmus (na každý vrchol). Lepší je ale využít Floydův-Warshallův algoritmus, počítající všechny vzdálenosti přímo, rychleji a ještě se snadněji implementuje. Dokonce je tak jednoduchý, že pokud nám nezáleží na časové složitosti, ale jen na rychlosti naprogramování, tak je lepší volbou než Dijkstra (podle [10]).

Tímto algoritmem jsme se dozvěděli hodnoty vzdáleností, ale nikoliv kudy vedou nejkratší cesty. Pokud bychom to chtěli vědět, tak bychom si v průběhu algoritmu museli pro každou dvojici u, v pamatovat poslední vrchol, přes který nám nejkratší cesta vede. Tzn. Floydův-Warshallův algoritmus můžeme doplnit výpočtem (se složitostí $\boldsymbol{O(n^3)}$) matice předchůdců **P** poté, co se dokončil výpočet matice w-vzdáleností. Je také možné počítat spolu s posloupností matic $\mathbf{D}^{(k)}$ průběžně i posloupnost matic $\mathbf{P}^{(k)}$, neboť pro ni platí následující rekurentní vztahy. Počáteční nastavení:

$$p_{ij}^{(0)} = 0 \quad \text{(NIL) pokud } i = j \text{ nebo } w_{ij} = \infty, \qquad \textbf{(7.5)}$$
$$p_{ij}^{(0)} = i \quad \text{v ostatních případech (tedy pro } u_i, u_j) \in H.$$

Úprava matice v každé iteraci:

$$p_{ij}^{(k)} = p_{ij}^{(k-1)} \quad \textit{pokud} \quad d_{ij}^{(k-1)} \le d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \;\textbf{,}$$
$$p_{ij}^{(k)} = p_{kj}^{(k-1)} \quad \textit{pokud} \quad d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \qquad \textbf{(7.6)}$$

7.3 Johnsonův algoritmus

Přes nespornou jednoduchost a snadnou implementovatelnost Floydova-Warshallova algoritmu je možné výpočet nejkratších cest realizovat ještě úsporněji, pokud budeme uvažovat řídké grafy. V řídkých grafech neroste počet hran se čtvercem počtu uzlů, ale většinou lineárně s počtem uzlů, takže asymptotická složitost je pak výrazně lepší než $O(n^3)$. Bohužel z této složitosti nelze u Floydova-Warshallova algoritmu uniknout, proto je třeba jiné řešení. V tomto odstavci ukážeme algoritmus navržený Johnsonem (viz [14]), jehož asymptotická složitost pro řídké grafy je $\boldsymbol{O(n^2 \log n + nm)}$. Výsledkem tohoto algoritmu je matice w-vzdáleností nebo signalizace existence záporně ohodnocených cyklů v grafu.

Základní myšlenkou Johnsonova algoritmu je n-násobné použití Dijkstrova algoritmu (viz [70]). O této možnosti jsme se zmínili již na začátku kapitoly s tím, že je omezena jen na grafy s nezáporným ohodnocením hran. Aby jsme tedy mohli Dijkstrův algoritmus v obecném případě použít, je třeba nejprve upravit původní ohodnocení w hran. Nějak ho transformovat, aby výsledné ohodnocení hran bylo nezáporné, ale aby v grafu zůstaly zachovány optimální cesty (tzn. nelze přičíst

ke každému ohodnocení hrany konstantu, neboť by se nám optimální cesty rozhodily. Protože, kolikrát je hrana na dané cestě, tolikrát bychom přičetli uvedenou konstantu a tím bychom více přitížili cestám s větším počtem hran na úkor těch s méně hranami). Kupodivu existuje široká třída **přehodnocení hran** grafu G zachovávající optimálnost cest (viz [1]), které lze provést v čase $O(nm)$. Nové ohodnocení w' musí splňovat následující dvě podmínky:

- pro každou dvojici uzlů $u, v \in U$ je nejkratší cesta pro ohodnocení w shodná s nejkratší cestou pro ohodnocení w'.
- pro každou hranu $(u, v) \in H$ je ohodnocení $w'(u, v)$ nezáporné.

Zavedeme obecný způsob přehodnocení hran, který zaručuje splnění první z výše uvedených podmínek. Pro zjednodušení zápisu vyjadřujeme symbolem $d(u, v)$ vzdálenost uzlů u a v platnou při ohodnocení hran w a symbolem $d'(u, v)$ vzdálenost při ohodnocení w'. Ohodnoceny budou ne jen hrany, ale i vrcholy. Předpokládejme, že mám každý uzel ohodnocen hodnotou $h(u)$, kterou je libovolné reálné číslo.

Nechť je dán prostý orientovaný graf G = <H, U> s ohodnocením hran $w : H \rightarrow \mathbf{R}$ a mějme libovolnou funkci $h : U \rightarrow \mathbf{R}$, která přiřazuje každému uzlu grafu G nějaké reálné číslo. Pro každou hranu $(u, v) \in H$ nyní definujme:

$$w'(u, v) = w(u, v) + h(u) - h(v). \qquad \textbf{(7.7)}$$

Ač je ohodnocení uzlů h jakékoliv, tak pro takto zvolené přehodnocení je splněna podmínka zachování optimality cest (důkaz v [1]).

Nyní se zaměříme na to, jak zvolit funkci h použitou při přehodnocení hran tak, abychom dosáhli splnění druhé z výše uvedených podmínek - nezápornosti ohodnocení hran w'. Johnson formálně původní graf rozšířil přidáním nového uzlu s, který napojil hranami ohodnocenými hodnotou nula na všechny ostatní uzly grafu (přidáme uzel s a n nových hran).

Neboli zadaný graf G = <H, U> s ohodnocením hran $w : H \rightarrow \mathbf{R}$ rozšíříme na graf G' = <H', U'> takto: přidáme nový uzel s, z něhož vedeme orientované hrany do všech uzlů grafu G. Platí tedy $U' = U \cup \{s\}$, $H' = H \cup \{(s, u) : u \in U\}$. Všem nově přidaným hranám určíme ohodnocení w rovné nule.

V takto vytvořeném grafu nemůže uzel s ležet na žádné orientované cestě spojující dvojici uzlů původního grafu G. Uzel s bude tedy pouze počátečním uzlem nejkratších cest vycházejících z něj do všech uzlů původního grafu. Graf G' obsahuje navíc cykly se zápornou w-délkou, právě když takové cykly obsahoval původní graf G (podrobnější popis s obrázky v [6]).

Předpokládejme nyní, že graf G (a tedy ani G') nemá cykly se zápornou w-délkou. Protože jsou z uzlu s dostupné všechny ostatní uzly, jsou v grafu G' korektně definovány vzdálenosti $d(s, u)$ a jsou konečné. Můžeme tedy položit $h(u) = d(s, u)$ pro všechna $u \in U'$. Pro libovolnou hranu $(u, v) \in H'$ z trojúhelníkové nerovnosti získaný vztah: $h(v) \leq h(u) + w(u, v)$, takže definujeme-li ohodnocení w' podle vztahu (7.7), bude platit $w'(u, v) = w(u, v) + h(u) - h(v) \geq 0$.

Touto volbou funkce h (ohodnocení zvoleno jako vzdálenost přidaného uzlu s) je tedy zaručeno splnění druhé z podmínek uvedených pro přehodnocení hran.

Nyní je již možné vyjádřit Johnsonův algoritmus. Jako podprogramy se v něm používají Bellmanův-Fordův algoritmus (viz kapitola 6.7) a Dijkstrův algoritmus (viz kapitola 6.5). S ohledem na tyto algoritmy předpokládáme, že výchozí graf G je zadán spojovou reprezentací pomocí seznamů následníků. Výsledkem algoritmu jsou w-vzdálenosti uložené do matice **D** nebo signalizace výskytu cyklů se zápornou w-délkou. Pro zjednodušení zápisu algoritmu předpokládáme, že každý uzel je identifikován přirozeným číslem z intervalu <1, n>.

Algoritmus 7.3 Johnsonův algoritmus

JOHNSON(G)
1 rozšíření grafu G na graf G‘
2 **if** not BELLMAN-FORD(G‘,*w*,*s*)
3 **then** write „graf obsahuje cyklus záporné délky“
4 **else for** každý uzel $u \in$ U‘
5 **do** $h(u) := d(s, u)$
6 **for** každou hranu $(u, v) \in$ H‘
7 **do** $w'(u,v) := w(u, v) + h(u) - h(v)$
8 **for** každý uzel $u \in$ U
9 **do** DIJKSTRA(G,*w*‘,*u*)
10 **for** každý uzel $v \in$ U
11 **do** $d_{u,v} := d'(u, v) + h(v) - h(u)$
12 **return D**

Neboli přidáme uzel *s*, spočteme vzdálenosti z přidaného uzlu ke všem ostatním uzlům a na základě těchto vzdáleností ohodnotíme uzly. Pomocí tohoto ohodnocení přehodnotíme hrany (na zaručeně nezáporné) a můžeme použít *n*-krát Dijkstrův algoritmus z každého uzlu. Zpětným přehodnocením získáme skutečné vzdálenosti. V grafu mohou být záporně ohodnocené hrany, proto musíme použít Bellmanův-Fordův algoritmus (ale jen jedenkrát) k výpočtu vzdáleností od uzlu *s*. Pokud narazíme na záporné cykly samotný Bellmanův-Fordův algoritmus to detekuje a my počítání můžeme ukončit.

Pro určení asymptotické časové složitosti Johnsonova algoritmu je rozhodující *n*-krát opakované provedení Dijkstrova algoritmu, neboť to spotřebuje čas $\boldsymbol{O(n^2 \log n + nm)}$, kdežto jedno provedení Bellmanova-Fordova algoritmu potřebuje jen $O(nm)$. Tento odhad ovšem předpokládá, že pro implementaci prioritní fronty v Dijkstrově algoritmu se použilo Fibonacciho haldy. Pokud použijeme jen obyčejnou binární haldu, dostáváme výslednou složitost Johnsonova algoritmu $\boldsymbol{O(nm \log n)}$, což je ale stále pro řídké grafy asymptoticky rychlejší než Floydův-Warshallův algoritmus se složitostí $O(n^3)$. Pro paměťovou složitost jsou rozhodující nároky na uložení matice vzdáleností **D**, které činí $O(n^2)$.

7.4 Dynamické programování

Hledání nejkratších cest mezi všemi dvojicemi uzlů je typickým příkladem **optimalizačního problému**, pro jehož řešení je vhodné použít metody **dynamického programování**. Základní myšlenka postupu je podobná jako u algoritmů typu "rozděl a panuj" - tedy rozložit problém na podproblémy, ty rekurzivně vyřešit, a pak kombinovat dílčí řešení do celkového výsledku.

Tento postup dává dobré výsledky, pokud jsou vznikající podproblémy navzájem nezávislé, vede však k extrémně neefektivním algoritmům, jestliže se stejné podúlohy opakují vícekrát. V algoritmech založených na principu dynamického programování se každý podproblém vyřeší pouze jednou, použitelné dílčí výsledky nižších úrovní rozkladu problému se uchovávají, a tak mohou být později opakovaně použity pro určení řešení a vyšší úrovni rozkladu (více o této problematice v [1]).

Jako jednoduchý příklad aplikace této myšlenky může posloužit návrh algoritmu pro výpočet n-tého prvku Fibonacciho posloupnosti definované známým vztahem $F_n = F_{n-1} + F_{n-2}$ pro $n > 1$ s počátečními hodnotami $F_0 = 0$ a $F_1 = 1$. Pokud tuto úlohu řešíme rekurzivním algoritmem kopírujícím tvar definiční rekurentní formule, dostaneme řešení s exponenciální časovou složitostí. Naproti tomu jednoduchý iterační algoritmus, který ve dvou pomocných proměnných uchovává poslední a předposlední spočtený prvek posloupnosti a na jejich základě určí prvek následující, má

pouze lineární časovou složitost. Algoritmy dynamického programování mají většinou lineární či kvadratickou paměťovou složitost, neboť dílčí výsledky uchovávají formou jedno- či dvourozměrných tabulek.

Návrh algoritmů založených na dynamickém programování je obecně možné rozdělit do následujících kroků:

1. určení struktury optimálního řešení
2. vyjádření hodnoty optimálního řešení rekurzivně

Dynamické algoritmy pro problém hledání nejkratších cest mezi všemi vrcholy udržují informaci o nejkratších cestách přidáváním a rušením hran a aktualizací jejich váhy. Aplikace lze nalézt v mnoha oblastech, jako např. dopravní sítě, databázové systémy (kde udržují dálkové vztahy mezi objekty), data flow analýza a kompilátory, formátování dokumentu a především směrování.

Uvedu dva hlavní dynamické algoritmy pro all-pairs shortest paths problém. **Kingův algoritmus** [78] pro orientované grafy s ohodnocením hran kladnými celými čísly menšími než C - $O(n^{2,5} \sqrt{}(C \log n))$, jehož hlavní myšlenkou je určovat dynamicky nejkratší cesty mezi všemi páry vrcholů až do vzdálenosti *d* a přepočítávat delší nejkratší cesty získané z každé aktualizace.

Druhým je algoritmus **Demetresca a Italiana** [40], pro nezáporné reálné váhy hran s $O(n^{2,5} \sqrt{}(S \log^3 n))$, kde S je počet rozdílných hodnot vah. Je založen na lokálních nejkratších cestách (LSP). Cesta *c* je lokální nejkratší cesta, jestliže každá její podcesta je nejkratší cesta (*c* nemusí být bezpodmínečně nejkratší cestou). Historická nejkratší cesta je ta, která byla nejkratší během sekvence aktualizací a žádná její hrana nebyla do té doby aktualizována. Lokálně historická cesta je taková, je-li každá její podcesta historickou nejkratší cestou (LHP). Hlavní myšlenkou je dynamické udržování sady lokálně historických cest, které zahrnují lokální nejkratší cesty a nejkratší cesty jako speciální případy. Snahou je zmenšit počet historických nejkratších cest, třeba transformací aktualizace sekvence za běhu do ekvivalentní delší. Během operace aktualizace se odstraní veškeré udržované cesty, které obsahují aktualizovanou hranu a pak se spustí dynamická modifikace Dijkstrova algoritmu paralélně ze všech uzlů v každém kroku. Nejkratší cesta s minimální váhou je vytažena z prioritní fronty a v kombinaci s existujícími historickými nejkratšími cestami tvoří nové lokální historické cesty. Prostorová složitost algoritmu je $O(mn \log n)$ v nejhorším případě (více ve zmiňovaném textu [40]).

Na tomto algoritmu je možno založit nový statický algoritmus. Počet lokálních cest v grafu v nejhorším případě až $O(mn)$, v reálných grafech je typicky jen jedna lokálně nejkratší cesta mezi nějakým párem uzlů, která je také nejkratší cestou. Takto navrhnuté lokální nejkratší cesty by mohly být využity pro statické algoritmy. Toto omezení ještě hlavně v rámci teoretických poznatků vylepšil Thorup na $O(n^2(\log n + \log^2(m/n)))$

Především z praktického významu v experimentech dosahovali lepších výsledků než teoretické meze a v případě vícekolového řazení dle velikosti dokonce rychlejší než opakované počítání řešení statickým algoritmem. Vedlejším efektem je navržení nového statického algoritmu, který v praxi podstatně redukuje počet kontrolovaných hran a může být na hustých grafech rychlejší než Dijkstrův algoritmus.

7.5 Fredmanův algoritmus

Prvním algoritmem mírně zrychlující $O(n^3)$ – při použití Floydova-Warshallova algoritmu [11] je algoritmus Fredmanův (více o něm v autorově textu [58]) běžící v čase $\boldsymbol{O(n^3 (\log \log n)^{1/3} / (\log n)^{1/3})}$. Je určen pro nezáporně ohodnocené orientované grafy. Fredman

v [46] poprvé realizoval možnost subkubického algoritmu. Uvedl porovnání mezi sumou hranových vah postačující pro řešení all-pairs shortest paths v $O(n^{5/2})$ a tohoto faktu využil k předzpracování a vývoji svého algoritmu, který pak trochu zjednodušil a zrychlil Takaoka (více v [63]).

7.6 Spirův algoritmus

V [77] Spira představil své řešení problému hledání nejkratších cest mezi všemi páry vrcholů na předpokladu, že váhy hran v grafu jsou nezávislé distribuce náhodné proměnné z libovolné distribuce. Hlavním výsledkem jeho práce je navržení algoritmu s průměrnou dobou $\boldsymbol{O(n^2 \log^2 n)}$. Algoritmus je pro orientované grafy s nezáporným reálným ohodnocením hran. Jedná se určitou variantu Dijkstrova algoritmu, kde Spira zlepšil čas *n*-krát spuštěného Dijkstrova algoritmu. Strategie Spiry, Dijkstry i Dantziga je v podstatě stejná. Je specifikovaný konkrétní uzel *s*. Z něho vypočteme nejkratší vzdálenosti do všech ostatních uzlů pátráním podél cesty z konkrétního uzlu *s* stoupající délky. Zlepšení v průměrné době běhu Spirova algoritmu oproti Dijkstrovi přes základní vzor Dantziga je dáno eliminací nadbytečných dlouhých cest v grafu. Vyvarujeme se práci s hranami, které směřují do uzlů, do kterých již známe vzdálenost z uzlu *s*.

Máme nezáporně vážený orientovaný graf a chceme nalézt pro všechny páry vrcholů (*i*, *j*), kde $1 \leq i \neq j \leq n$ nejkratší cestu z *i* do *j*. Vrátíme se nachvilku k Dijkstrovi. Představme si, že nejkratší cesty z uzlu *s* do všech ostatních uzlů nalezneme Dijkstrovým algoritmem. Dalším otevřeným uzlem (v autorově textu označen jako labeled) se stane nejbližší nový uzel (unlabeled) k jednomu z již otevřených uzlů nebo k uzlu *s*. Toto vede na možnost řazení hran před samotným během algoritmu. Spirův algoritmus toho využije a vyžaduje setřízenou seznamovou reprezentaci sousednosti dle rostoucí ceny. Toto řazení lze pro kompletní *n* uzlový graf udělat v čase $O(n^2 \log n)$. Hrany, které mají stejné ohodnocení budou seřazeny dle indexu koncových vrcholů. Vyskytují-li se rovnající se délky cest a hran, může to vést v jistých případech do pomalé implementace. Toto je jediná komplikace tohoto algoritmu. Nicméně naším cílem je nalézt všechny $n(n-1)$ nejkratší cesty v grafu v lepším čase než $O(n^3)$.

Spirův algoritmus je docela podobný *n*-krát použitému Dantzigovu algoritmu. Hlavním rozdílem mezi těmito dvěma algoritmy je Spirovo začlenění slabého pravidla kandidatury a uvolnění silného pravidla kandidatury, které vyžaduje, aby všichni kandidáti byli užiteční. Slabé pravidlo kandidatury si vynutí to, aby veškeré kandidátské hrany (*c*, *t*) byly takové, že $w(c, t) \leq w(c, u)$ pro všechny nové vrcholy *u*. Hlavní motivací tohoto slabého pravidla kandidatury je to, že by jsme mohli redukovat nákladné skenování seznamu sousednosti, což výrazně zpomaluje Dantzigův algoritmus. Všimněte si, že v Dantzigově algoritmu musí být sám vrchol *t* vně množiny S (více v [33]). Pro identifikaci každého následujícího kandidáta s minimálním ohodnocením v $O(\log n)$ by sada kandidátů měla být implementována jako datová struktura binary tree tournament. Oslabené pravidlo kandidatury naznačuje, že kandidát s minimální váhou se již nutně nestane užitečný. Vybereme kandidáta s minimální cenou v kořenu haldy. Pak expandujeme množinu S, ta jako u Dantziga obsahuje vrcholy, do nichž již známe nejkratší cesty. Spira uvedl cenu zvětšeného množství kandidátů, kteří musí být prozkoumaní během kroku algoritmu řezem času stráveného skenováním seznamu sousednosti hledajíce užitečné hrany.

Idea algoritmu je jednoduchá. Nejprve seřadíme hrany do *n* seznamů, každý pro hrany vycházející z každého uzlu. To lze v čase $O(n^2 \log n)$. Vybereme počáteční uzel *s*. Označíme nejbližší uzel k uzlu *s*. Řekněme, že je to uzel *t*. Klasicky jako ve většině předchozích algoritmů aktualizujeme vzdálenost uzlu *t* od počátku *s* ($d[t] = d[s] + w(s, t)$). Nejkratší hranu z uzlu *t* porovnáme s délkou

nejkratší další zbývající hrany uzlu *s*. Můžeme nastálo označit uzel indikovaný tímto porovnáním, jestliže nastane to, že hrana vycházející z uzlu *t* bude kratší.

Algoritmus 7.4 Spirův algoritmus

```
SPIRA(G)
1       for s := 1 to n do
2           S: = {s}
3           d[s] : = 0
4           inicializuj množinu kandidátů {(s, t)}, kde (s, t) je nejkratší hrana vycházející z s
5           while |S| < n do
6               vybereme kandidátní hranu (c, t) s nejmenší váhou
7               if t ∉ S then
8                   S := S ∪ {t}
9                   d[t]:= d[c] + w(c, t)
10                  if |S| = n then break
11                  přidej k sadě kandidátů nejkratší hranu z uzlu t
12              nahraď (c, t) v sadě kandidátů další nejkratší hranou z c
```

Předpokládejme, že máme vypočteny nejkratší cesty k *k*-1 uzlům. Bez ztráty obecnosti řekneme, že jsou to uzly 2, 3, .., *k* a mají nejkratší cesty délky D_2, D_3, .., D_k. Povšimneme si také, že $D_1 = 0$. Předpokládejme, že délka nejkratší zbylé hrany z uzlu *i* je *d*(*i*, *m*) pro $1 \leq i \leq k$. Hledáme minimum (D(1, *i*) + D(*i*, *m*) kde $1 \leq i \leq k$) řekneme, že minimum je D(1, *j*) + d(*j*, *m*). Otevřeme (v autorově textu [77] to symbolizuje, že se uzel dostane do stavu labeled) uzel *m*, jestliže nebyl dosud otevřený. Je-li *m* otevřený, tak vezmeme další hranu v seřazeném seznamu hran z uzlu *j*. D(*j*, *m*), přidáme d(1, *j*) + d(*j*, *m*) a minimalizujeme sadu získanou z minulé sady nahrazením d[*j*] + d(*j*, *m*) novou hodnotou. Jeli *m* nový (unlabeled) označíme jej, d(1, *m*) = d(1, *j*) + d(*j*, *m*). Vypočítáme d(1, *j*) + d(*j*, *m*) a d(1, *m*) + d(*m*, *s*), kde d(*m*, *s*) je minimální hrana v seznamu hran z uzlu *m*. Dále minimalizujeme novou množinu získanou smazáním d(1, *j*) + d[*m*] z předchozí množiny a přidáme tyto dvě nové hodnoty. Identifikujeme úspěšného minimálního kandidáta v nejvýše 2*k*(log *n*) porovnáních pro množinu o *k* prvcích.

Dále provozujeme turnaje nezbytné k zisku všech nejkratších cest z uzlu 1 a potom postupujeme stejně s uzly 2, 3, ..., n. Celkový počet turnajů je malý ve srovnání s předchozím algoritmem s $O(n^3 / \log n)$.

Formálně popíšeme začátek Spirova algoritmu, více lze nalézt v [77]. Nechť G = <H, U> je orientovaný graf, U je množina uzlů a $H = \{d_{ij} : 1 \leq i \leq n, 1 \leq j \neq i \leq n\}$ množina nezáporných hran, kde d_{ij} je vzdálenost uzlů *i* a *j*. V případě, že hrana neexistuje, $d_{ij} = \infty$. Když algoritmus skončí v D_{ij}, bude délka nejkratší cesty z uzlu *i* do uzlu *j* pro n^2 hodnot (*i*, *j*).

Počet kroků potřebných algoritmu k nalezení nejkratší cesty mezi všemi páry uzlů v *n* uzlovém grafu, kde hrany jsou nezávislé náhodné proměnné vybrané z pravděpodobnostní fukce *p* reálných proměnnách *x*, takových $p(x) = 0$ pro $x < 0$, je $\leq O(n^2 \log^2 n)$.

Spira udělal důležitý pravděpodobnostní předpoklad, nazvaný nezávislost koncového bodu tak, že kandidát s minimální cenou ztratí na každém vrcholu s rovnající se pravděpodobností. Celkové očekávané množství výběru kandidátů bude $O(n \log n)$ do té doby, než my vybereme všechny kandidáty alespoň jednou. Každý výběr by nás stál $O(\log n)$ pro odpovídající stromovou manipulaci, takže celkové úsilí k tomu, aby jsme vyřešili problém nalezení nejkratších cest z vrcholu s do všech ostatních je průměrně $O(n \log^2 n)$. A celkové úsilí k nalezení nejkratší cesty mezi všemi páry vrcholu je $O(n^2 \log^2 n)$.

Fredman v novější verzi svého algoritmu rozšiřuje Spirovo řešení (více v autorově textu [46]) řeší problém hledání nejkratších cest mezi všemi páry vrcholů na orientovaném grafu s nezáporně ohodnocenými hranami v čase $O(n^2 \log n \log^i n)$, kde $i \in N$.

Spira ve svém textu uvažuje nad závislostmi ohodnocení hrany v grafu. Z jeho výsledků a empirických výsledků chování Spirova algoritmu dělaného později jinými výzkumníky vyplývá následující. Ačkoli ohodnocení hrany může záviset na uzlu ze kterého hrana vede, je nezávislé na uzlu, do kterého hrana směřuje.

7.7 Chanův algoritmus

Algoritmus s dobou běhu $O(n^3/\log^2 n)$, vylepšující veškeré známé algoritmy pracující s hustými reálně ohodnocenými grafy bez použití rychlého násobení matic. Více o tomto algoritmu lze nalézt v autorově textu [24]. Donedávna mezi algoritmy s nejlepšími výsledky se řadil autorův předchozí algoritmus s $O(n^3 / \log n)$ (více o něm v [55]), založený na jednoduchém geometrickém přístupu nevyžadující explicitní tabulkové vyhledávání nebo slovní triky a **Hanův algoritmus** [60] se složitostí $O((n^3 \log^{5/4} \log n)/ \log^{5/4} n)$, který překvapivě prolomil $O(n^3 / \log n)$, využívající sofistikované slovní triky (word-packing - realizované výběry z tabulky).

Chanův algoritmus míchá elementy z předchozího autorova algoritmu [55] a Hanova algoritmu [60] využívajíce jednodušší geometrický přístup s word-packing triky. Výsledné hodnocení se blíží k limitu čistých kombinatorických algoritmů a to z důvodu, že booleovský maticový problém násobení je speciální případ problému hledání nejkratších cest mezi všemi páry uzlů a nejrychlejší booleovský algoritmus násobení matic známý, který nespoléhá na algebraické techniky (použité např. v Strassenově algoritmu), je stále klasický algoritmus problému čtyř Rusů ze sedmdesátých let s dobou běhu $O(n^3 / \log^2 n)$. Což je nejrychlejší známý čistě kombinatorický algoritmus pro řešení all-pairs shortest paths v neorientovaných grafech.

Otázka zůstává, zda obecně all-pairs shortest paths problém by mohl být řešen v opravdu subkubickém čase $O(n^{3-d})$ pro specifickou konstantu $d>0$, užitím rychlého násobení matic (jako Strassenovo či Coppersmith-Winograd) jako podprogram.

Je známé [56], že libovolný all-pairs shortest paths problém s reálným ohodnocením hran může být redukovaný na problém počítání vzdálenosti dvou libovolných reálných hodnot, čtverec matic $n \times n$ (známý jako problém min-plus násobení matic, dávající dvě matice **A** a **B** a jejich vzdálenost je definován jako matice **C** s c_{ij}:= index k ,který minimalizuje $a_{ik} + b_{kj}$, redukce je dělaná přes rekurzi a nezvyšuje asymptotickou časovou složitost.

V [55] sledujeme, že problém počítání vzdálenosti pravoúhlých matic $n \times d$ a $d \times n$ může být viděno jako geometrický rozsah problému hledání a může být řešený v $O(n^2)$ až do rozměru $d \sim \log n$ použitím známých technik výpočetní geometrie.Vzdálenost dvou matic $n \times n$ může být vyřešen v $O((n^2 n) / d) = O(n^3 / \log n)$.

V nové verzi algoritmu [24] autor dává jiný geometrický pohled na problém počítání vzdálenosti $n \times d$ a $d \times n$ matic a navrhuje nové řešení pro rozměry do $d \sim \log n/\log \log n$. Ačkoliv hodnota d je okrajově horší než předchozí geometrický přístup, pravoúhlá vzdálenost je ve skutečnosti vypočítána v subkvadratickém čase (v předchozím přístupu tento rys není) a vede k nejzazšímu zlepšení pro problém hledání nejkratších cest mezi všemi páry vrcholů.

Jak je možné vypočíst pravoúhlou vzdálenost v $O(n^2)$ čase, když sama má n^2 položek?

Položky této matice jsou malá celá čísla z intervalu 1 až d, a tak lze sbalit vícenásobné položky v jedno slovo w = word size ($w = \Omega(\log n)$). Lze tak uložit seznam n integerů v komprimované formě. Matici C lze uložit jako komprimovaný seznam. Celkový čas je subkvadratický pro malé d.

Přehled algoritmů pro řešení all-pairs shortest paths:

Floyd–Warshall [43]	1962	$O(n^3)$
Fredman [58]	1976	$O(n^3 \log^{1/3} \log n / \log^{1/3} n)$
Takaoka [63]	1992	$O(n^3 \log^{1/2} \log n / \log^{1/2} n)$
Dobosiewicz [57]	1990	$O(n^3 / \log^{1/2} n)$
Han [59]	2004	$O(n^3 \log 5/7 \log n / \log 5/7 n)$
Takaoka [64]	2004	$O(n^3 \log 2 \log n / \log n)$
Zwick [65]	2004	$O(n^3 \log^{1/2} \log n / \log n)$
Chan [55]	2005	$O(n^3 / \log n)$
Han [60]	2006	$O(n^3 \log^{5/4} \log n / \log^{5/4} n)$
Chan [24]	2007	$O(n^3 \log^3 \log n / \log^2 n)$

Tabulka 7.1: Algoritmy hledající nejkratší cesty mezi všemi páry uzlů

7.8 Další algoritmy

Problém hledání nejkratších cest mezi všemi páry vrcholů je nesporně jedním z nejvíce známých problémů v návrhu algoritmů, často studovaný, nicméně složitost problému zůstává stále otevřena. Je až překvapující, že klasické algoritmy pro hledání cesty z jednoho konkrétního uzlu pro reálně ohodnocené grafy (Dijkstrův (kladně ohodnocené hrany) a Bellmanův-Fordův algoritmus) zůstávají nezlepšené a pro řídké grafy řešení problému all-pairs shortest paths s n aplikacemi Dijkstrova algoritmu s použitím scaling metodik dokonce poráží metodiky celočíselného maticové násobení nebo RAM/integer-based techniky.

Saunders a Takaoka uvedli v [30] rámec, definující koncept trigger uzlů, jako zobecnění zpětnovazebních vrcholů. Tyto vrcholy dělí graf do jedinečné kolekce acyklických podgrafů, takových že jsou ovládány těmito trigger vrcholy. Tento rámec poskytuje algoritmům hledajícím nejkratší cesty mezi všemi dvojicemi uzlů časovou složitost $O(mn + nr^2)$, kde r je počet takových trigger vrcholů, nebo-li široký okruh acyklických struktur uvnitř grafu. Je-li r malé, pak je graf považován za téměř acyklický.

Bloniarz [73] poskytl algoritmus s dobou běhu $O(n^2 \log n \log^* n)$. Frieze a Grimmet [61] $O(n^2 \log n)$, ale je vhodný jen pro náhodné grafy.

Rychlejší algoritmy existují pro speciální případy all-pairs shortest paths problému pro instance, kdy je graf neohodnocený nebo rovinný [74]. Pro některé problémy hledání nejkratších cest se zatím dá obtížně najít efektivní algoritmy pro orientované grafy, zatímco pro neorientované byly vyřešené optimálně. Takovým problémem je určování k-nejkratších cest mezi páry uzlů. Daný orientovaný graf G s nezápornými hranami, kladným číslem k, dva uzly s a t - cílem je najít k-nejkratších cest seřazených dle jejich délky. Není-li úloha omezená na acyklické spojení, tak známe pro orientovaný i neorientovaný graf **algoritmus Eppsteinův** [29]. Nejlepší algoritmus pro počítání k-nejkratších cest je **Yenův algoritmus** [26, 27] se složitostí $O(kn(m + n \log n))$ - bylo sice několik pokusů o zlepšení, ale zatím bez větších úspěchů.

Pro speciální případy grafů ohodnocených omezenou množinou celých čísel lze použít algoritmy **Alona-Galila-Margalita** [62] a **Seidela** [76] využívající násobení matic k nalezení vzdáleností mezi všemi páry vrcholů velmi rychle.

Lze využít paralélního zpracování, kdy vlákna integrujeme pomocí mezivýsledků z jednotlivých vláken, k tomu aby redukovala práci dělanou jiným vláknem.

Algoritmus skrytých cest [72] objevuje skryté struktury "nejkratších cest" grafu pomocí ořezávání vzdálených zbytečných hran. Poznamenejme, že algoritmus aktuálně konstruuje každou cestu v obráceném pořadí, přidáváváním hran na konec cesty. Toto ulehčuje dopředný průchod při

konstrukci cesty. Je to jednoduchá modifikace algoritmu konstruování cest ve formě typicky užívané Dijkstrovým algoritmem. Složitost závisí na implementaci haldy - užitím standardní haldy O(mn log n), Fibonacciho haldy O(mm+n^2 log n). Algoritmus skrytých cest vyžaduje grafy s nezáporným ohodnocením hran, nicméně v případě celočíselného ohodnocení hran za pomoci nějakého scaling algoritmu (Gabow-Tarjan [75] a Goldberg[38]) transformuje váhovou funkci do nezáporné váhové funkce, indukující strukturně stejné nejkratší cesty v grafu. Budeme-li řešit nejkratší cesty na takovém grafu, nejdříve použijeme jeden z těchto algoritmů na vytvoření kladně ohodnocených hran a pak aplikujeme algoritmus skrytých cest.

7.9 Porovnání algoritmů pro řešení nejkratší cesty mezi všemi uzly

Mezi základními metodami v této kapitole figuruje Floydův-Warshalův algoritmus. Z novějších a efektivnějších algoritmů si naimplementujeme Spirův algoritmus a jelikož je tento vylepšením *n*-krát provedeného Dantzigova algoritmu, tak pro experimentální porovnání využijeme i implementaci Dantzigova algoritmu pro all-pairs shortest paths problem.

Experimenty provedeny na stejném počítači jako v předchozích experimentech s algoritmy pro hledání nejkratších cest z jednoho konkrétního uzlu do všech ostatních. Výsledky byly rovněž ověřeny na školním linux serveru Merlin.

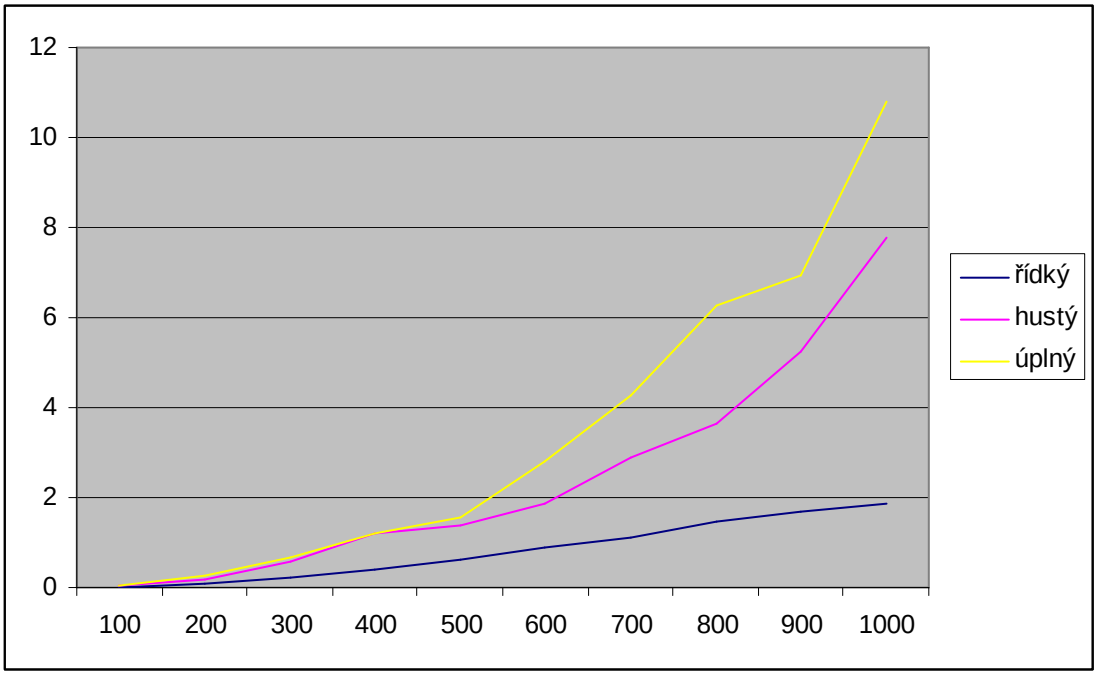
Implementace používají seznamovou reprezentaci sousednosti vstupního grafu (seznam ohodnocených hran), jen pro Floydův-Warshallův algoritmus je využita maticová reprezentace sousednosti. Experimentujeme s několika reprezentacemi grafu se zvyšujícím se počtem *n* vrcholů. V praxi mají problémy často specifickou strukturu, proto neměníme jen počet vrcholů vstupního grafu, ale i rozměr počtu hran. Generovány jsou tři různé typy grafu s ohledem na počet hran (*m*). Všechny reprezentace mají minimálně z vrcholu 1 dosažitelné všechny uzly grafu. Druhou vlastností všech reprezentací je acykličnost. A poslední vlastností je kladné ohodnocení všech hran. V experimentech tedy nenastane situace, kdy by graf obsahoval cyklus nebo záporně ohodnocenou hranu (neboť většina zvolených algoritmů se se záporným ohodnocením nevyrovná). Snahou bylo, aby jednotlivé implementace různých algoritmů byly dělané jednotně, aby bylo možné porovnat smysluplně dobu běhu. V následující tabulce jsou výsledky experimentů, máme dobu běhu implementace v sekundách (uživatelský čas procesoru bez vstupních a výstupních operací) pro jednotlivé vstupní reprezentace grafu. Pro zisk referenčních dat provádíme 5 až 8 běhů implementací na stejných vygenerovaných reprezentacích grafu, které pak průměrujeme počtem běhů.

V grafu 7.1 ukazujeme závislost doby běhu implementace Spirova algoritmu na parametrech reprezentace grafu. Oproti stejnému porovnání u implementace Dantzigova algoritmu v předchozí kapitole je vidět jasné vylepšení. Pokud bychom udělali graf závilosti pro *n*-krát provedený Dantzigův algoritmus dostaneme podobné výsledky jako u grafu 6.4. Při výběru vhodných kandidátů již neprocházíme všechny hrany, což se projeví i na výsledném čase. Z experimentů se nám potvrdil vstupní předpoklad, a to, že Spirův algoritmus vylepšuje dobu běhu *n*-krát provedeného Dantzigova algoritmu.

Výsledky experimentů s implementacemi algoritmů:

	Floydův-Warshallův alg.			n-krát Dantzigův alg.			Spirův alg.		
n	řídký	hustý	úplný	řídký	hustý	úplný	řídký	hustý	úplný
100	0,01	0,01	0,02	0,03	0,1	0,1	0,02	0,03	0,04
200	0,09	0,09	0,09	0,17	0,39	1,03	0,09	0,2	0,25
300	0,31	0,32	0,3	0,47	2,27	3,47	0,23	0,6	0,68
400	0,71	0,79	0,79	0,99	9,05	20,59	0,4	1,21	1,22
500	1,19	1,26	1,24	1,47	39,54	68,35	0,62	1,37	1,56
600	2,07	2,06	2,04	2,23	89,17	132,3	0,88	1,85	2,8
700	2,85	2,89	2,88	2,95	163,62	244,38	1,13	2,91	4,28
800	3,72	4,04	4,2	3,98	310,69	493,69	1,45	3,65	6,28
900	5,49	5,58	5,87	5	508,84	-	1,68	5,25	6,93
1000	7,01	6,92	7,04	6,79	-	-	1,85	7,77	10,82

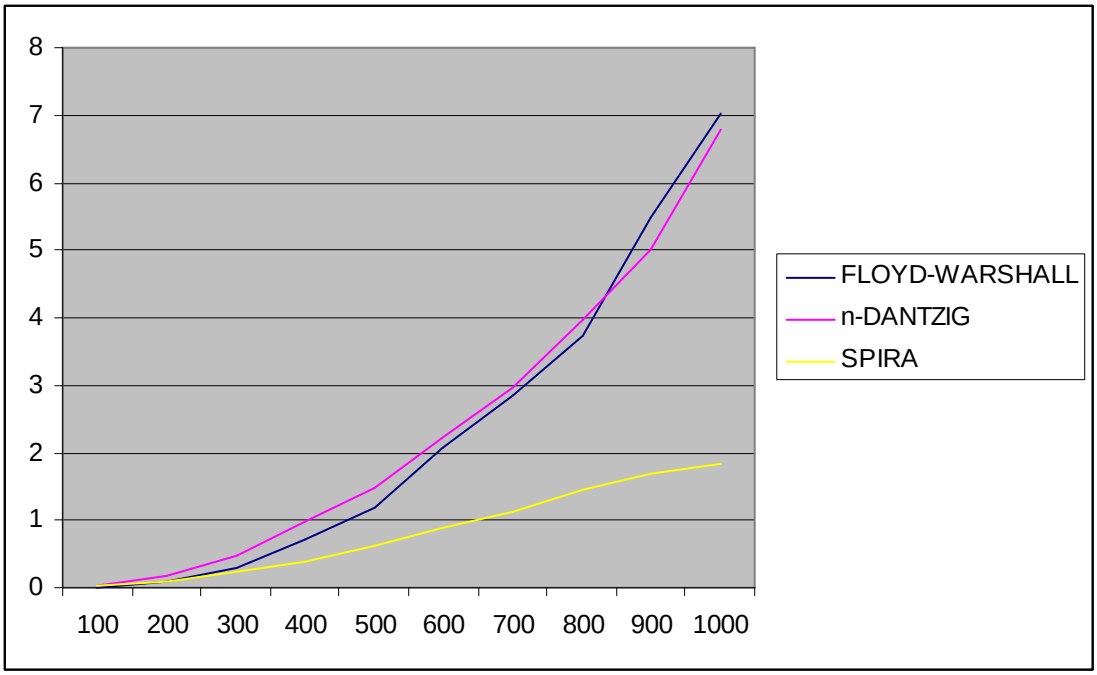
Tabulka 7.1: Doba běhu jednotlivých implementací algoritmů v závislosti na typu grafu



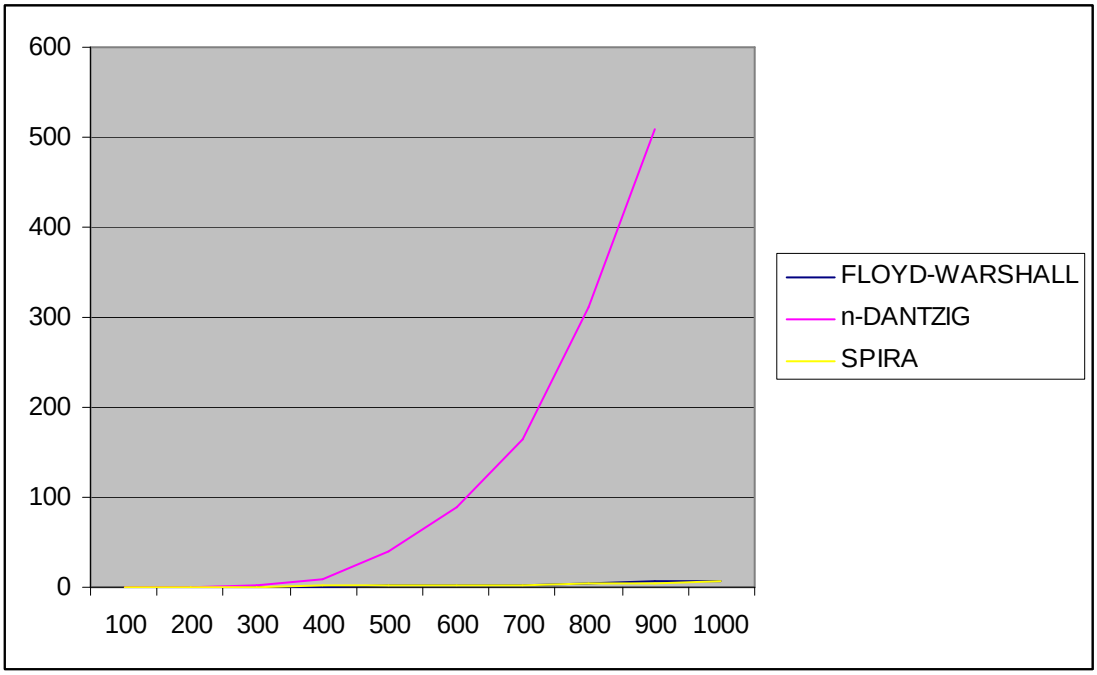
Graf 7.1: Doba běhu implementace Spirova algoritmu v závislosti na typu grafu

V grafech 7.2 a 7.3 si všímáme doby běhu implementací v řídkých respektive hustých reprezentací grafu. Není pro nás překvapení, že u narůstajícího počtu hran ve vstupním grafu roste i doba běhu implementace *n*-krát provedeného Dantzigova algoritmu.

Podíváme se na experimentální výsledky implementace Floydova-Warshallova algoritmu. Pokud bychom udělali graf závislosti doby běhu implementace na počtu hran vstupního grafu asi bychom žádné závislosti nenašli. Je to dáno tím, že pracujeme s maticovou reprezentací, kde jsou uvedeny všechny hrany bez ohledu na to, zda existují nebo ne.



Graf 7.2: Porovnání doby běhu implementací v řídkých grafech



Graf 7.3: Porovnání doby běhu implementací v hustých grafech

Co se týká paměťové náročnosti implementace Floydova-Warshallova algoritmu pracujeme s maticí $n \times n$ tj. $O(n^2)$. U implementace Dantzigova algoritmu sice vystačíme s $O(n+m)$, nicméně před každým z n běhů musíme datové struktury updatovat. Implementace Spirova algoritmu vyžaduje pro každý uzel seřazenou množinu hran z něj vystupujících. Musíme počítat s nejhorším případem, kdy z každého uzlu může vést n-1 hran tzn. náročnost je $O(n^2)$.

V implementaci Floydova-Warshallova algoritmu vystačíme s jednou maticí velikosti $n \times n$ implementovanou pomocí dynamického pole obsahujícího n^2 prvků. Navíc vypočítáváme matici předchůdců pomocí druhého pole. Pro implementaci n-krát běhu Dantzigova algoritmu využijeme vše z implementace Dantzigova algoritmu a s menším updatem ukazatelů na kandidáty zopakujeme pro každý uzel. Implementace Spirova algoritmu je založena na rozdělení hran vstupního grafu do n seznamů, každý pro hrany vycházejícího z jednotlivého uzlu. A využití binární haldy pro rychlý výběr minimálního kandidáta.

Správnost implementací algoritmů rovněž ověřena na základě porovnání jejich výstupních dat. Ty lze nalézt spolu se zdrojovými kódy na přiloženém médiu.

8 Závěr

Cílem práce bylo studium existujících a rešerše nejnovějších a nejefektivnějších metod pro nalezení nejkratších cest v grafu. Ukázalo se, že problém nejkratších cest byl již dlouhou řadu let studován a rozhodně se v dnešní době vývoj nových a efektivnějších metod nezastavil. Každý rok se objeví několik nových řešení, které sice nepatrně nebo jen pro určité druhy problémů zlepšují dosavadní principy, nicméně poznatky a myšlenky z nich mohou pomoci k dalšímu zlepšování. Z tohoto důvodu nebylo možné z hlediska velkého rozsahu nastudovat úplně všechny existující metody. Zaměřil jsem se proto hlavně na ty principy, které měli větší přínos v efektivitě nebo přinesly nějaké nové myšlenky pro následný vývoj nových algoritmů. Velká většina novějších metod, které autoři zveřejňují ve svých pracích, se vyskytuje jen v anglickém jazyce a bylo by škoda se s nimi neseznámit.

Aby tato práce nebyla jen teoretickým přehledem, tak po domluvě s vedoucím práce bylo vybráno šest existujících metod, které byly implementovány v jazyce C. Následně byly experimentálně porovnány mezi sebou dle několika kritérií. Experimenty kromě očekávaných výsledků z teoretických poznatků přinesly i zajímavé skutečnosti. Především odhalily důvod proč se na Dantzigův algoritmus opomenulo a masivněji se začal praktikovat Dijkstrův algoritmus i přes to, že se v principu moc neodlišují a byly představeny v téměř stejný rok. Výsledkem překvapila i implementace Spirova algoritmu. Nepočítal jsem v porovnání s *n*-krát opakovaným Dantzigovým algoritmem takové zlepšení doby běhu.

Co se týče dalšího možného rozšíření práce je několik možných směrů pokračování. Vývoj algoritmů pro hledání nejkratších cest neustále pokračuje a světlo světa spatřují ještě efektivnější metody, než jsou uvedeny v této práci. Hlavně v oblasti problému nejkratších cest mezi všemi páry uzlů je hlavní výzkum veden pro model RAM, kde pro určité problémy dosahuje řešení lineárního času. Další oblastí je dynamické programování. Algoritmy v této oblasti byly v práci uvedeny jen v krátkém výčtu a hlouběji nebyly studovány. Rozšíření je vhodné i v oblasti experimentů. Jistě by bylo přínosné implementovat a porovnat více než tři algoritmy z každé skupiny. Zajímavé by bylo i experimentální porovnání různých implementací prioritní fronty v Dijkstrově algoritmu nebo porovnat různé heuristické vylepšení Bellmanova-Fordova algoritmu.

Literatura

[1] Kolář, J.: Teoretická informatika. Česká informatická společnost, 2000.
[2] Demel, J.: Grafy a jejich aplikace. Academia, Praha, 2002.
[3] Novotný, J.: Základy operačního výzkumu. [studijní opora], FAST VUT Brno, 2006.
[4] Nýdl, V.: Diskrétní matematika I. [skripta], Jihočeská Univerzita v Českých Budějovicích, 1998.
[5] Kolář, J.: A New Possibility in Bi-Directional Search. Kybernetika, 13, 1977, s. 11-22.
[6] Kolář, J., Štěpánková, O., Chytil, M.: Logika, algebra a grafy. SNTL, Praha, 1989, s. 221-266 a 286-298.
[7] Kučera, L.: Kombinatorické algoritmy. SNTL, Praha, 1983.
[8] Nešetřil, J: Teorie grafů. SNTL, Praha, 1979.
[9] Nečas, J.: Grafy a jejich použití. SNTL, Praha, 1978.
[10] Černý, J.: Základní grafové algoritmy. KAM, MFF UK, 2008.
[11] Cormen, T.H., Leiserson, Ch.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT Press & McGraw-Hill, Cambridge, 2001.
[12] Cherkassky, B.V., Goldberg, A.V., Radzik, T.: Shortest paths algorithms: Theory and experimental evaluation. Math Programming, 73, 1996, s. 129-174.
[13] Cherkassky, B.V., Goldberg, A.V., Radzik, T.: Shortest paths algorithms: Theory and experimental evaluation. ACM-SIAM symposium on Discrete algorithms table of contents, 1994, s. 516 – 525.
[14] Johnson, D.B.: Eficient algorithms for shortest paths in sparse network. Journal of the ACM, 24, 1977, s. 1-13.
[15] Bellman, R.E.: On a routing problem. Quart. appl.math., 16:87-90, 1958.
[16] Ford, L.R. Jr.: Network flow theory. Report P-923, The rand corporation, Santa Monica, Cal., 1956.
[17] Dijkstra, E. W.: A note on two problems in connection with graphs. Numer. math., 1959, s. 269-271.
[18] Fredman, M., Tarjan, R.: Fibonacci heaps and their uses in improved network optimisation algorithms. Journal of the ACM, 34, 1987, s. 596-615.
[19] Takaoka, T.: Shortest path algorithms for nearly acyclic directed graphs. Theoretical Computer Science, 203, 1998, s. 143-150.
[20] Dantzig, G.B.: On the shortest path route through a network. Management science, 6, 1960, s. 187-190.
[21] Gallo, G., Pallottino, S.: Shortest paths algorithms. Annals of oper. res., 13, 1988, s. 3-79.
[22] Goldberg, A.V., Harrelson Ch.: Computing the shortest path: A search meets graph theory. 16h annual ACM-SIAM symposium on discrete algorithms, 2005, s. 156 – 165.
[23] Pohl, I.: Bi-direction search. Machine inteligence, 6, Edinburgh University, 1971, s. 124-140.
[24] Chan, T. M.: More algorithms for all-pairs shortest paths in weighted graphs. 39th annual ACM symposium on theory of computing, 2007, s. 590-598.
[25] Mařík, V., Štěpánková, O., Lažanský, J.: Umělá inteligence. Academia, Praha, 1993.
[26] Yen, J. Y.: Finding the k-shortest loopless paths in a network. Management Science, 17, 1971, s. 712–716.
[27] Yen, J. Y.: Another algorithm for finding the k-shortest loopless network paths. 41st meeting of the operations research society of America, 20, 1972.
[28] Nelson, P.C., Toptsis, A.A.: Unidirectional and bidirectional search algorithms. IEEE software, 1992, s. 77-83.
[29] Eppstein, D.: Finding the k shortest paths. SIAM J. Computing, 28, 1998, s. 652-673.
[30] Saunders, S., Takaoka T.:Efficient algorithms for solving shortest paths on nearly acyclic directed graphs. Australasian symposium on theory of computing, 41, 2005, s. 127-131.
[31] Pape, U.: Implementation and efficiency of Moore algorithms for the shortest root problem. Math.Prog., 7, 1974, s. 212-222.

[32] Pallottino, S.: Shortest-path methods: Complexity, interrelations and new propositions. Networks, 14, 1984, s. 257-267.
[33] Bashar, M.: Average case analysis of algorithms for the maximum subarray problem. [master thesis], University of Canterbury, 2007.
[34] Dial, R.B.: Algorithm 360: Shortest path forest with topological ordering. Comm. ACM 12, 1969, s. 632-633.
[35] Kershenbaum, A.: A note on finding shortest paths trees. Networks, 11, 1981.
[36] Glover, F., Klingman, D.: Computational study of an improved shortest path algorithm. Networks, 14, 1984, s. 25-37.
[37] Glover, F., Klingman, D., Philips, N.: A new polynomially bounded shortest paths algorithm. Operation Res., 33, 1985, s. 65-73.
[38] Goldberg, A.V.: Scaling algorithms for the shortest paths problem. 4th ACM-SIAM symposium on discrete algorithms, 1993, s. 222-231.
[39] Goldberg, A.V., Radzik, T.: A heuristic improvement of the Bellman-Ford algorithm. Applied math, 6, 1993, s. 3-6.
[40] Demetrescu, C., Emiliozzi, S., Italiano, G.F.: Experimental analysis of dynamic all pairs shortest path algorithms. 15th annual ACM-SIAM symposium on discrete algorithms table of contents, New Orleans, Louisiana, 2004, s. 369 – 378.
[41] Zhan, F. B., Noon,C.E.: Shortest path algorithms: An evaluation using real road network. Transportation science, 32, 1998, s. 65-73.
[42] Klein, P., Rao, S., Rauch, M., Subramanian, S.: Faster shortest-path algorithms for planar graphs. STOC 1994, CS-94-12.
[43] Flyod, R.W.: Algorithm 97: Shortest path. Comm. ACM, 5, 1962.
[44] Ford, L.R., Fulkerson, D.R.: Flows in net. Princeton University, Princeton, NJ, 1963.
[45] Thorup, M.: Undirected single-source shortest paths with positive integer weights in linear time. Journal of the ACM, 46, 1999, s. 362-394.
[46] Fredman, M.L., Willard, D.E.: Trans-dichotomous algorithms for minimum spanning trees and shortest paths. Journal of computer and system sciences, 48, 1994, s. 533-551.
[47] Ahuja, R.K., Mehlhorn, K., Tarjan, R.E.: Faster algorithms for the shortest path problem. Journal of the ACM, 37, 1990, s. 213-223.
[48] Johnson, D.B.: Efficient special-purpose priority queues. 15th annual allerton conference on communications, control, and computing, 1977, s. 1-7.
[49] Van Emde Boas, P., Kaas, R., Zijlstra, E.: Design and implementation of an efficient priority queue. Math. syst. theory 10, 1977, s. 99-127.
[50] Holzer, M., Schulz, F., Wagner, D., Willhalm, T.: Combining speed-up techniques for shortest-path computations. Journal of experimental algorithmics, 10, 2005.
[51] Willhalm, T., Wagner, D.: Shortest path speedup techniques. In Algorithmic Methods for Railway Optimization, LNCS, Springer-Verlag, New York, 2006.
[52] Hart, P., Nilsson, N.J., Raphael, B.A.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Sys. Sci. Cybernet., 2, 1968.
[53] Ahuja, R., Magnanti, T., Orlin, J.: Network flows theory. Prentice–Hall, Englewood Cliffs, NJ, 1993.
[54] Pohl, I.: Bi-directional and heuristic search in path problems. Techical Report 104, Linear Accelerator Center, Stanford, CA, 1969.
[55] Chan, T.M.: All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. 9th Workshop algorithms data struct., Lecture notes computer science, 3608, Springer-Verlag, 2005, s. 318–324.
[56] Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, MA, 1974.
[57] Dobosiewicz, W.: A more efficient algorithm for the min-plus multiplication. Int. J. Computer Math.,32, 1990, s. 49-60.
[58] Fredman, M.L.: New bounds on the complexity of the shortest path problem. SIAM J. Comput., 5, 1976, s. 49-60.
[59] Han, Y.: Improved algorithm for all pairs shortest paths. Inform. process. lett., 91, 2004, s. 245-250.

[60] Han, Y.: An $O(n^3(\log \log n/ \log n)^{5/4})$ time algorithm for all pairs shortest paths. 14th European sympos. algorithms, Lecture notes computer science, 4168, Springer-Verlag, 2006, s. 411-417.
[61] Frieze, A.M., Grimmet, G.R.: The shortest-path problem for graphs with random arc-lenghts. Dicrete applied mathematics, 10, 1985, s. 57-77.
[62] Alon, N., Galil, Z., Margalit, O.: On the exponent of the all pairs shortest path problem. 32nd annual symposium on foundations of computer science, San Juan, Puerto Rico, 1991, s. 569-575.
[63] Takaoka, T.: A new upper bound on the complexity of the all pairs shortest path problem. Inform. Process. Lett., 43, 1992, s. 195-199.
[64] Takaoka, T.: A faster algorithm for the all-pairs shortest path problem and its application. 10th Int. Conf. Comput. Comb., Lecture notes computer science, 3106, Springer-Verlag, 2004, s. 278-289.
[65] Zwick, U.: A slightly improved sub-cubic algorithm for the all pairs shortest paths problem with real edge lengths. 15th Int. Sympos. Algorithms and Computation, Lecture notes computer science, 3341, Springer-Verlag, 2004, s. 921-932.
[66] Dreyfus, S.: An appraisal of some shortest-path algorithms. Operations research, 17, 1969, s. 395-412.
[67] Whiting, P.D., Hillier, J.A.: A method for finding the shortest route through a road network. 1960.
[68] Dantzig, G.B., Thapa, N.M.: Linear programming 2: Theory and extensions, Springer Verlag, Berlin, 2003.
[69] Glover, F., Glover, R., Klingman, D.: The threshold shortest path algorithm, Networks, 14, 1986.
[70] Johnson, D.B.: A note on Dijkstra's shortest path algorithm. Journal of the Association for Computing Machinery, 20, 1973, s. 385-388.
[71] Ford, L.R., Fulkerson, D.R.: Consructing maximal dynamic flows from static flows. Operations research, 6, 1958, s. 419-433.
[72] Karger, D.R., Koller, D., Phillips, S.J.: Finding the hidden path: time bounds for all-pairs shortest paths. SIAM Journal on computing, 22, 1993, s. 1199-1217.
[73] Bloniarz P.A.: A shortest-path algorithm with expected time $O(n^2 \log n \log^* n)$. Department of computer science, State University of NY, Albany, 1980.
[74] Frederickson G.N.: Planar graph decomposition and all-pairs shortest paths. Journal of the ACM, 38, 1991, s. 162-204.
[75] Gabow H.N., Tarjan R.E.: Faster scaling algorithms for network probléme. SIAM journal on Computing, 1989, s. 1013-1036.
[76] Seidel, R.: On the all-pairs shortest patrh problem. 24th ACM symposium on theory of Computing, 1992, s. 745-749.
[77] Spira, P.M.: A new algorithm for finding all shortest paths in a graph of positive arcs in average time $O(n^2 \log^2 n)$. SIAM Journal on Computing, 2, 1973, s. 28-32.
[78] King, V.: Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. 40th IEEE symposium on foundations of computer science (FOCS'99), 1999, s. 81-99.
[79] Sniedovich, M.: Dijkstra's Algorithm revisited: the OR/MS Connexion. Dokument dostupný na URL http://www.ifors.ms.unimelb.edu.au/tutorial/dijkstra_new/index.html (leden 2009).