VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2018 Bc. Oleksandr Yarmolskyy

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

VYUŽITÍ DISTRIBUOVANÝCH A STOCHASTICKÝCH ALGORITMŮ V SÍTI
APPLICATION OF DISTRIBUTED AND STOCHASTIC ALGORITHMS IN NETWORK.

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Oleksandr Yarmolskyy**
AUTHOR

VEDOUCÍ PRÁCE — **doc. Ing. Vladislav Škorpil, CSc.**
SUPERVISOR

BRNO 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Telekomunikační a informační technika**
Ústav telekomunikací

Student: Bc. Oleksandr Yarmolskyy ***ID:*** 155262
Ročník: 2 ***Akademický rok:*** 2017/18

NÁZEV TÉMATU:

Využití distribuovaných a stochastických algoritmů v síti

POKYNY PRO VYPRACOVÁNÍ:

V rámci diplomové práce prozkoumejte principy a metody konvergence v přístupových a transportních sítích na základě stochastických a distribuovaných algoritmů. Navrhněte metody testování konvergence a vytvořte model sítě reprezentující různorodé topologie sítě. Na nich otestujte navržené metody. Dosažené výsledky řádně diskutujte.

DOPORUČENÁ LITERATURA:

[1] NANCY A. LYNCH. Distributed algorithms. San Francisco, Calif: Morgan Kaufmann Publishers, 1996. ISBN 9780080504704.

[2] KUSHNER, Harold J. a George YIN. Stochastic approximation and recursive algorithms and applications. New York: Springer, c2003. ISBN 03-870-0894-2.

Termín zadání: 18.6.2018 ***Termín odevzdání:*** 15.8.2018

Vedoucí práce: doc. Ing. Vladislav Škorpil, CSc.
Konzultant:

prof. Ing. Jiří Mišurec, CSc.
předseda oborové rady

ABSTRAKT

Tato diplomová práce se zabývá problematikou distribuovaných a stochastických algoritmů včetně testování jejich konvergence v sítích. V teoretické části jsou výše uvedené algoritmy stručně popsané, včetně jejich dělení, problémů, výhod a nevýhod. Dále jsou vybrané dva distribuované algoritmy a dva stochastické algoritmy a následně jsou stručně popsané. V praktické části je provedeno jejich porovnání podle rychlosti konvergence na odlišných topologiích sítí v prostředí MATLAB.

KLÍČOVÁ SLOVA

distribuované algoritmy, stochastické algoritmy, konvergence, MATLAB, Bellmanův-Fordův algoritmus, Dijkstrův algoritmus, A* algoritmus, Push-sum algoritmus

ABSTRACT

This thesis deals with the distributed and stochastic algorithms including testing their convergence in networks. The theoretical part briefly describes above mentioned algorithms, including their division, problems, advantages and disadvantages. Furthermore, two distributed algorithms and two stochastic algorithms are chosen. The practical part is done by comparing the speed of convergence on various network topologies in MATLAB.

KEYWORDS

distributed algorithms, stochastic algorithms, convergence, MATLAB, Bellman-Ford algorithm, Dijkstra's algorithm, A* algorithm, Push-sum algorithm

OLEKSANDR, Yarmolskyy *Využití distribuovaných a stochastických algoritmů v síti*: diplomová práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2018. 78 s. Vedoucí práce byl Ing. Bohumil Novotný.

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Využití distribuovaných a stochastických algoritmů v síti“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

(podpis autora)

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu doc. Ing. Vladislavovi Škorpilovi, CSc. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

Brno ....

(podpis autora)

Faculty of Electrical Engineering
and Communication
Brno University of Technology
Purkynova 118, CZ-61200 Brno
Czech Republic

http://www.six.feec.vutbr.cz

PODĚKOVÁNÍ

Výzkum popsaný v této diplomové práci byl realizován v laboratořích podpořených z projektu SIX; registrační číslo CZ.1.05/2.1.00/03.0072, operační program Výzkum a vývoj pro inovace.

Brno . . . . . .

. . . . . .
(podpis autora)

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

ÚVOD

Diplomová práce se zabývá problematikou distribuovaných a stochastických algoritmů a jejich konvergencí v sítích. V teoretické části jsou nejdříve stručně popsané distribuované a stochastické algoritmy včetně jejich souvislosti s přístupovými a transportními sítěmi. Dále je představen matematický nástroj, který se používá u distribuovaných a stochastických algoritmů u této práce. Následně jsou vybrané a popsané dva distribuované algoritmy a dva stochastické algoritmy, které mají určité vhodné vlastnosti pro sítě nebo v případě distribuovaných algoritmů se jíž používají v reálných sítích v praxi. Konkrétně z distribuovaných algoritmů jsou vybrány Bellman-Fordův a Dijkstrův algoritmus a u stochastických jsou vybrány A* (a star) a Push-Sum.

V praktické části je provedena implementace zvolených čtyř algoritmů v prostředí MATLAB, kde je dále měřena rychlost jejích konvergence na vytvořených topologiích sítí pomoci matice sousednosti, matice vah a u A* algoritmu pomocí mřížkových struktur, což jsou grafy převedené do mřížové struktury. Základní topologie jsou kruh, hvězda, strom, slabě propojená topologie a silně propojená topologie, které byly použité u měření a srovnání u distribuovaných algoritmů. Dále klasické topologie byly nahrazené nahodilými topologiemi s různým počtem vrcholů. U A* algoritmu jsou topologie v podobě mřížkových map o různých velikostech. Na konci praktické části jsou všechny čtyři algoritmy vzájemně porovnávané a tyto výsledky jsou prezentované v podobě grafů a tabulek.

1 DISTRIBUOVANÉ ALGORITMY

Pojem distribuované algoritmy zahrnuje širokou škálu současných algoritmů, které se používají pro různé aplikace. Původně byl pojem používán v souvislostí s algoritmy, které běží na více procesorech, kde „distribuovaný“ znamená rozdělení (distribuci) přes velkou geografickou oblast. S přibývajícími technologiemi a aplikacemi, se použitelnost pojmu rozšířila tak, že nyní zahrnuje i algoritmy, které běží v lokálních sítích a algoritmy, které běží na více procesorech, ale sdílejí společnou paměť. Stalo se tak, protože tyto algoritmy mají hodně společného [11].

Distribuované algoritmy se objevují v mnoha aplikacích a odvětvích vědy a techniky, např. v telekomunikací, distribuovaných informačních systémech, vědeckých výpočtech a kontrole procesů v reálném čase (anglický real-time - např. v kosmonautice). Důležitou součásti sestavení systémů pro výše uvedené odvětví, je návrh, implementace a analýza distribuovaných algoritmů. V současnosti existuje spousta druhů distribuovaných algoritmů, které řeší určité problémy a podle [11] se rozlišují vlastnostmi:

1. **Metodou meziprocesorové komunikace (The Interprocess Communication Method - IPC):** Distribuované algoritmy běží na více procesorech, které potřebují nějakým způsobem komunikovat. Mezi běžné metody, jak tuto komunikaci zprostředkovat patří přistupování ke společné (sdílené) paměti, zasílání bod-bod (point-to-point) nebo všesměrových (broadcast) zpráv přes malé vzdálenosti, což je např. LAN síť, nebo přes velké vzdálenosti a vykonávat tak vzdálené volaní procedur.

2. **Načasováním (The Timing Model):** Může být vzato v potaz několik odlišných předpokladů ohledně načasování událostí v systému reflektujících různé druhy časové informace, která může být použita algoritmy. V prvním extrému, procesory můžou byt úplně synchronní, provádějící komunikaci a výpočty v perfektní krokové (lock-step) synchronizaci. V druhém extrému procesory můžou být úplně asynchronní a výpočty zde jsou prováděné libovolnou rychlostí v libovolném pořadí. Mezi těmito dvěma extrémy je nepřeberné množství algoritmů, které mají něco z obou způsobů a tak spadají do kategorie částečně synchronních (partially synchronous), kdy mají částečnou informaci o načasování události. Procesory můžou mít hranice na své relativní rychlosti nebo přístup k přibližně synchronizovaným hodinám.

3. **Modelem poruch (The Failure Model):** Hardware na kterém algoritmus běží může být kompletně spolehlivý nebo musí určitou míru chybovosti to-

lerovat. Do chybového chování spadají chyby procesoru, např. když přestane pracovat a to s varováním nebo bez, anebo může nastat mnohem závažnější chyba, které se říká Byzantské selhání[1] (Byzantine Failure), kde chybový procesor má nahodilé chování. Další typem chybovostí jsou problémy komunikačních mechanismů, kam spadají ztráty zpráv nebo jejich duplikace.

4. **Podle řešených problémů (The Problem Addressed):** Algoritmy se liší i podle toho, jaký problém řeší. Typické situace, které algoritmy řeší jsou výše popsané, ale další typy závisí i na tom, jakou konkrétní úlohu konkrétní algoritmus řeší, tj. na obraz algoritmu má klíčový vliv problém, který daný algoritmus řeší. Mezi další úlohy k řešení patří alokace zdrojů, komunikace, shoda mezi distribuovanými procesory, kontrola souběžné databáze, detekce zamknutí (deadlock) procesoru, globální snímky (global snapshots), synchronizace a implementace různých objektů.

1.1 Problémy distribuovaných algoritmů

Velký počet současných distribuovaných algoritmů se musí potýkat s vysokým stupněm nejistoty a čím dál větší mírou nezávislé činnosti. Některé druhy problémů vyplývající z nejistoty a nezávislosti [11]:

- neznámý počet procesorů[2]
- neznámá topologie sítě
- nezávislá vstupní data v různých místech
- několik programů spouštěných najednou, startujících v jiných časech a pracujících rozdílnou rychlostí
- nedeterminismus procesorů
- nejistý čas doručení zpráv
- neznámé pořadí zpráv
- chybovost procesorů a komunikace

[1] Název pochází z problému společného útoku v Byzantské říši, kde všichni generálové chtějí zároveň a společně zaútočit ze všech stran, ale jeden z nich může zradit. Analogie v procesorech je taková, že jeden procesor může být poruchový a je problematické docílit shody neboli konvergence.

[2] V problematice sítí procesor také označuje samostatné zařízení např. směrovač, přepínač apod.

Ne každý algoritmus se musí potýkat se všemi nebo většinou výše uvedených problému. Často se vybere nejlepší možný algoritmus na řešení určitého problémů, který má kompromis mezi jeho ostatními parametry (nejčastěji to jsou rychlost, čas, chybovost, ...).

1.2 Standardní problémy

Problémy, které řeší distribuované algoritmy je možné rozdělit na [11]:

- **Atomická činnost (Atomic Commit):** Je to činnost, kde několik změn je provedeno jedinou operací. Když je atomická činnost úspěšná, znamená to, že všechny změny v systému byly provedené. Pokud je před dokončením atomické činnosti chyba, činnost je přerušena a není aplikovaná žádná změna. Algoritmy na řešení tohoto problému jsou dvoufázový potvrzovací protokol (Two-Phase Commit Protocol) a třífázový potvrzovací protokol (Three-Phase Commit Protocol).
- **Shoda (Consencus):** Algoritmy řešící shodu se snaží docílit souhlasu mezi více procesy na určité činnosti. Konkrétně musí splnit tyto čtyři podmínky:
  1. **Ukončení (Termination):** Každý proces bez chyby rozhodne o nějaké hodnotě α.
  2. **Platnost (Validity):** Pokud všechny procesy nabídnou stejné α, tak každý korektní proces se rozhodné pro toto α.
  3. **Integrita (Integrity):** Každý korektní proces rozhodne pouze o jedné hodnotě a pokud rozhodné o hodnotě α, tak α musí být nabídnuté některým procesem.
  4. **Shoda (Agreement):** Pokud se proces, který nemá chybný stav, rozhodné pro α, tak každý korektní proces se také rozhodne pro α.

  Typickým zástupcem algoritmu na řešení shody, který se nejčastěji uvádí v literatuře, je Paxos.
- **Distribuované hledaní (Distributed Search):** Hledaní nejkratší cesty v topologii nebo systému. Algoritmus na řešení toho problému je např. minimální kostra grafu (Minimum Spanning Tree - MST).
- **Zvolení vůdce (Leader Election):** Je zvolení jediného procesu mezi více procesy při nějaké činnosti jako hlavního koordinátora dané činnosti.

- **Vzájemná výlučnost (Mutual Exclusion):** Je problém přístupu více procesu najednou ke stejnému zdroji.
- **Neblokující datové struktury (Non-Blocking Data Structures):** Řeší problém chybovosti více vláken, kde chyba jednoho vlákna neovlivní vlákno druhé.
- **Spolehlivý přenos (Reliable Broadcast):** Spolehlivý přenos je důležitý základ komunikace v distribuovaných systémech a je definován následujícími vlastnostmi:
    1. **Platnost (Validity):** Pokud korektní proces pošle zprávu, tak nějaký další korektní proces tuto zprávu doručí.
    2. **Shoda (Agreement):** Pokud korektní proces doručí zprávu, tak všechny další korektní procesy tuto zprávu doručí.
    3. **Integrita (Integrity):** Každý korektní proces doručí stejnou zprávu pouze jednou a za podmínky, že tato zpráva byla zaslaná dalším procesem.
- **Replikace (Replication):** Duplikace informačního zdroje pro procesy, např. na více místech v systému.
- **Přidělení zdrojů (Resource Allocation):** Snaha o spravedlivé přidělení nebo přístup ke zdrojům potřebných pro určitou činnost (např. paměť, výpočetní čas u procesoru apod.).
- **Generování kostry grafu (Spanning Tree Generation):** Je to generování kostry grafu u různých topologii grafů.
- **Narušení symetrie (Symmetry Breaking):** Narušení symetrie z důvodů, aby se algoritmus nezacyklil, ale došel k výsledku.

2 STOCHASTICKÉ ALGORITMY

Stochastické algoritmy, oproti distribuovaným algoritmům, při iteracích pracují s náhodnými operacemi, což více odpovídá reálnému světu. To znamená, že úkoly nad kterými pracuji, mají řešení neznámé, nejednoznačné nebo se po nich řešení ani nevyžaduje.

Stochastické algoritmy mají v současně době značné využití v umělé inteligenci, kde se používají k řešení problémů, které dokáže řešit člověk, ale ne stroj, tj. kde nejde algoritmizovat daný problém neboli převést na deterministický algoritmus (posloupnost kroku pro stroj, která končí v „rozumném“ čase). Přesněji to jsou problémy, kde algoritmus, který by vedl k řešení, musí být nedeterministický (stochastický). Deterministickému algoritmu čas potřebný na řešení roste do nekonečna nebo je exponenciální, faktoriální apod.

Z výše uvedeného plyne, že když je algoritmus stochastický, tak při stejném vstupu můžeme docílit různých výstupů (dnešní počítače jsou deterministické - např. co se napíše klávesnici jako vstup, to se zobrazí na výstupu na obrazovce) [22] [10] [21].

V současnosti se stochastické algoritmy podle [10] a [21] dají rozdělit do několik hlavních odvětví, které budou v dalších kapitolách stručně[1] popsané. Mezi tyto odvětví patří aproximační algoritmy, genetické a evoluční[2].

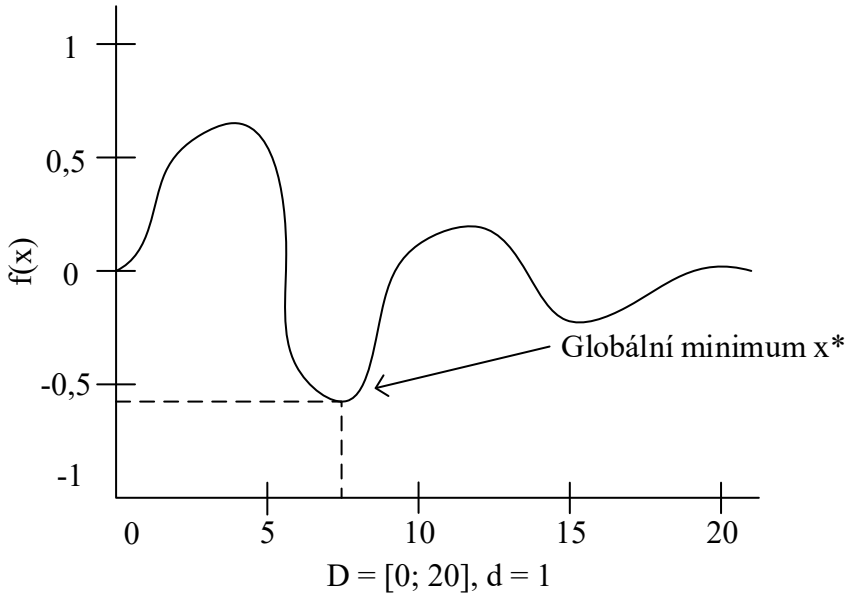
2.1 Problém globální optimalizace

Pro lepší pochopení principu stochastických algoritmů, je nutné nejdříve představit matematický problém globální optimalizace, který ukáže základní myšlenku všech stochastických algoritmů. Problém globální optimalizace je nalezení souřadnic bodu v definičním oboru funkce, který má extrémní hodnotu, tj. minimální nebo maximální. Jak je vidět na obrázku 2.1 funkce v definičním oboru [0, 20] má bodů s minimem více, ale jen jeden bod je globální minimum, tj. nejmenší v celé funkci.

Nalezení obecného řešení globálního minima (pro člověka snadno pochopitelného problému) není triviální záležitost z hlediska algoritmů. Obzvlášť je to těžký úkol, když funkce má lokálních minim v sobě ve větším počtu nebo argument funkce není jedno reálné číslo, ale vektor reálných čísel. Funkce ani v některých případech nemusí byt diferencovatelná (spojitá), ale i zde je třeba zjistit globální minimum nebo se mu aspoň přiblížit s uspokojivou přesnosti [10] [21].

[1] Detailní matematický popis a výpis nejpoužívanějších algoritmů je nad rámec teto práce.

[2] Existují i další dělení s vlastními algoritmy. Obecně genetické a evoluční algoritmy značně zrychlily vývoj ve spoustě vědních oborů, např. medicíně, chemií apod. Důvodem je, že pomocí těchto algoritmů lze simulovat např. nové materiály, sloučeniny atd.



Obr. 2.1: Funkce s globálním minimem

Obecná formulace globálního minima může být:

$$f : D \to \mathbb{R}, D \subseteq \mathbb{R}^d. \tag{2.1}$$

Máme najít bod $\boldsymbol{x}^* \in D$, pro který platí, že $f(\boldsymbol{x}^*) \leq f(\boldsymbol{x})$, pro $\forall \boldsymbol{x}$, $\boldsymbol{x} \in D$. Nalezení bodu $\boldsymbol{x}^* \in D$ je řešením problému globální optimalizace. Bodu $\boldsymbol{x}^*$ říkáme bod globálního minima, definičnímu oboru D se říká doména nebo prohledávaný prostor, přirozené číslo d je dimenze úlohy [21].

Formulace problému globální optimalizace jako nalezení globálního minima není na úkor obecnosti, neboť chceme-li nalézt globální maximum, pak jej nalezneme jako globální minimum funkce $g(\boldsymbol{x}) = -f(\boldsymbol{x})$ [21].

Analýza problému globální optimalizace ukazuje, že neexistuje deterministický algoritmus řešící obecnou úlohu globální optimalizace (tj. nalezení dostatečně přesné aproximace $\boldsymbol{x}^*$) v polynomiálním čase[3]. Tomuto ohraničení se říká, že problém globální optimalizace je NP-obtížný (časová složitost roste exponenciálně se zpracováním úkolu) [21].

[3]Ještě přesněji, polynomiální čas algoritmu je takový algoritmus, jehož nejhorší časovou složitost lze shora ohraničit polynomiální funkcí $W(n) \in O(p(n))$.

Úlohu globální optimalizace je nutné řešit v mnoha vědních oborech a profesích s velkým ekonomickým dopadem, proto je nutné hledat vhodné algoritmy na řešení konkrétních problémů.

Úloha 2.1 se označuje jako hledání volného extrému funkce (Unconstraint Optimalization). Je možné přijatelnost řešení ještě omezit podmínkou, např. nějakými rovnicemi nebo nerovnostmi. Pak jde o problém hledání vázaného extrému (Constrained Optimatization) [21].

Pro zjednodušení budeme uvažovat, že prostor, kde hledáme globální minimum je souvislý, jak je definováno podmínkou [21]:

$$\begin{aligned} D = \langle a_1; b_1\rangle \times \langle a_2, b_2\rangle \times \cdots \times \langle a_d, b_d\rangle = \textstyle\prod_{i=1}^{d} \langle a_i, b_i\rangle, \\ a_i < b_i, i = 1, 2, \ldots, d, \end{aligned} \tag{2.2}$$

Účelovou *f(**x**)* umíme vyhodnotit s požadovanou přesností v každém bodě $\boldsymbol{x} \in D$. Podmínce 2.2 se říká hraniční omezení (Boundary Constrains nebo Box Constrains), protože oblast D je vymezena jako d-rozměrný kvádr. Pro úlohy řešené numericky na počítači nepředstavuje podmínka 2.2 žádné podstatné omezení, neboť hodnoty a_i, b_i jsou omezené datovými typy užitými pro $\boldsymbol{x}$ a *f(**x**)*, tj. většinou reprezentací čísel v pohyblivé řádové čárce. Proto se takové úlohy označují jako neomezené (volné) nepřetržité problémy (Uncostrained Continuous Problems) [21].

Úlohy hledání vázaného extrému (Constrained Optimization) jsou obvykle formulovány takto [21]:

$$\begin{aligned} &\text{Najdi minimum funkce } f(\boldsymbol{x}), \boldsymbol{x} = (x_1, x_2, \ldots, x_d) \text{ a } \boldsymbol{x} \in D \\ &\text{za podmínky: } g_i(\boldsymbol{x}) \leq 0, i = 1, \ldots, p \\ &\qquad\qquad\quad h_j(\boldsymbol{x}) = 0, j = p+1, \ldots, m. \end{aligned} \tag{2.3}$$

Řešení je považováno za přijatelné (Feasible), když $g_i(\boldsymbol{x}) \leq 0$ pro $i = 1,\ldots,p$ a $|h_j(\boldsymbol{x})| - \varepsilon \leq 0$ pro $j = p+1,\ldots,m$. Pro libovolný bod $\boldsymbol{x} \in D$ a zadané kladné číslo ε můžeme definovat průměrné porušení podmínek (Mean Violation) $\bar{v}$ jako

$$\bar{v} = \frac{\Sigma_{i=1}^{p} G_i(\boldsymbol{x}) + \Sigma_{j=p+1}^{m} H_j(\boldsymbol{x})}{m},$$

kde

$$G_i(\boldsymbol{x}) = \begin{cases} g_i(\boldsymbol{x}) & \text{když} \quad g_i(\boldsymbol{x}) > 0 \\ 0 & \text{když} \quad g_i(\boldsymbol{x}) \leq 0 \end{cases}$$

$$H_j(\boldsymbol{x}) = \begin{cases} |h_j(\boldsymbol{x})| & \text{když} \quad |h_j(\boldsymbol{x})| - \varepsilon > 0 \\ 0 & \text{když} \quad |h_j(\boldsymbol{x})| - \varepsilon \leq 0. \end{cases}$$

Existují optimalizační úlohy, kde prohledávaný prostor D není spojitý, ale diskrétní (tzv. diskrétní problémy), např. hodnoty jednotlivých prvků vektoru $\boldsymbol{x}$ jsou celočíselné. K diskrétním problémům patři hledání optimální cesty v grafu.

Z výše uvedeného popisu plyne nemožnost nalezení algoritmu, který by vyřešil problém globální optimalizace v polynomiálním čase, a proto se začalo využívat algoritmů stochastických, které negarantují řešení v konečném počtu kroků, ale naleznou řešení blížící se požadovanému a takové řešení se dá prakticky využít. Zjednodušeně řečeno všechny stochastické algoritmy pracují na tomto principu.

V případě stochastických algoritmů pro globální optimalizaci lze říci, že heuristicky prohledávají prostor D. Heuristika je postup, ve kterém se využívá určité informace pro např. zúžení prohledávaného prostoru nebo odhadem, kde by dané řešení mohlo být. Odhadem se rozumí např. letecká vzdálenost, nejméně zatížená vzdálenost, neobsazená polička apod. Heuristika má analogii v reálném světě v podobě intuice, analogie předešlých situací (podobnost) a zkušenosti. V praktickém životě heuristiky jsou běžně užívané (např. hledaní něčeho, výběr partnera, lov). Navíc většina stochastických algoritmů má v sobě zjevný nebo skrytý proces učení, který je často odvozený z přírodních nebo sociálních procesů. Velká část pracuje s více kandidáty na řešení (s více body v prohledávaném prostoru). Těmto kandidátům se říká populace a v daném prostoru se pohybuji tam, kde dále mezi sebou nachází lepší kandidáty.

2.2 Aproximační algoritmy

Aproximační algoritmy jsou algoritmy, které vyřeší problém s určitou chybou. Příklad využití aproximačních algoritmů je například výpočet odmocnin (např. druhé). Pro výpočet odmocnin je už potřeba použit aproximační algoritmus, u kterého nedochází k nalezení přesného výsledku.

Když je algoritmus iterativní, tak to znamená, že je nutné jej několikrát aplikovat než poskytne uspokojivý výsledek. Algoritmus po každém kroku (iteraci) poskytne částečný výsledek. Například k iterativním algoritmům patří Bubble-Sort, ale algoritmy Quick-Sort a Insert-Sort už nikoliv (všechny tři jsou na řazení čísel). Pokud by se zastavil chod v průběhu řazení u algoritmu Bubble-Sort, získal by se kompletní seznam, ale nebude úplně seřazený. U zbylých dvou by to byl neúplný seznam.

Mezi aproximační algoritmy patři nejznámější Newtonův algoritmus, který ještě přesněji patři mezi gradientní optimalizační metody. Jedná se o jednoduchý, avšak velmi efektivní algoritmus. Používá se na výpočet odmocnin a různých funkcí. Před zahájením výpočtu se musí nastavit tzv. počáteční podmínka, což je první nastavení celého postupu od kterého se budou odvíjet a zpřesňovat další výsledky. U složitějších algoritmů je určování počátečních podmínek složité, někdy složitější než samotný výpočet. U gradientních optimalizačních metod existují funkce, kde špatně zvolená počáteční podmínka zhorší nebo znemožní celý výpočet, tj. metody nepřipouští, aby se řešení v průběhu optimalizace zhoršilo. Oproti tomu existuji stochastické optimalizační metody (stejný princip jako gradientní), které umožňují zhoršení řešení za předpokladu, že v průběhu dalších iterací dojde ke zlepšení řešení, tj. v každém kroku je algoritmus zatížen určitou neurčitostí, a tak může dojit ke zhoršení nebo ke zlepšení výsledku. Tato vlastnost umožňuje stochastickým optimalizačním metodám překonat lokální minimum nebo maximum [22].

2.3 Evoluční algoritmy

Existují problémy, které nelze dobře popsat ani pomocí matematických funkcí a na řešení nestačí jak exaktní, tak aproximační algoritmy. V takových případech je třeba použit nedeterministické algoritmy s předem neznámým počtem kroků nebo neznámou chybou, tj. nevíme, kdy problém vyřeší a s jakou chybovostí. Je vhodný na řešení problému, kde existuje více správných možností a nejsme tak schopni určit co je jediná správná možnost výsledku. Například na řešení problému nejlepší cesty, kde se může vybrat cesta podle ceny, pohodlnosti nebo délky. Při řešení takových problémů existuje celá řada metod, které se můžou při řešení uplatnit. Metodou se zde rozumí propojení mnoha dílčích algoritmů, které dohromady tvoří určitý postup neboli metodu. Nejvíce se tento typ algoritmů uplatňuje v umělé inteligenci a principiálně se soustřeďují na tři hlavní úkoly [22]:

- **Optimalizaci:** Hledání minima nebo maxima funkce.
- **Predikci:** Předpověď neznámého průběhu funkce.
- **Klasifikaci:** Rozdělení prvků do skupin na základě neznámých kritérií.

Algoritmy fungující na výše popsaném principu buď řeší velice konkrétní problém (tzn. algoritmus řeší pouze jediný typ problémů) nebo pokud si jsou problémy podobné, tak jediný algoritmus dokáže řešit několik takových problémů (tzn. být obecný). Početnou skupinou metod jsou učící se algoritmy (resp. metody) založené na simulaci. Patří sem neuronové sítě (simulují nervový systém) a evoluční algoritmy,

které budou dále zjednodušeně popsané.

Evoluční algoritmy kopírují (resp. simuluji) evoluční procesy z přírody a jsou založené na principu generování a testování (pokus-omyl). Hodí se na řešení NP problémů (řešení je možné v polynomiálním čase ověřit). Většina evolučních algoritmů patří do skupiny stochastických algoritmů, které také zahrnují algoritmy jako metoda náhodného prohledávání, simulované žíhání, Monte Carlo a dalších. Hlavní aplikace stochastických algoritmů je řešení složitých optimalizačních úloh, úloh s velkým počtem proměnných a úloh s omezujícími podmínkami. Jiné typy lze na optimalizační úlohu převést (např. predikční a klasifikační). Většina evolučních algoritmů také patři do skupiny učících se algoritmů.

Při nasazení na praktické úlohy je nutné překonat množství problémů. Celou úlohu je třeba vhodným způsobem formalizovat, tj. stanovit omezující podmínky, podmínky konvergence, kvantifikovat proměnné atd. Dále stochastické algoritmy (což jsou i evoluční) nezaručují optimální řešení. Pokud v určitém počtů kroků i zkonverguje (dojde k řešení - výsledku), tak není zaručena dostatečná vzdálenost od skutečného optimálního řešení. Řešením je vícenásobné spouštění výpočtu, nebo použitím hybridních metod, které evoluční algoritmy použijí pouze k nalezení počátečních podmínek a dále z těchto počátečních podmínek je spuštěn deterministicky optimalizační algoritmus. Hybridní metody mají výhodu v malé citlivosti na počáteční podmínky, vynikající schopnost prohledávaní (oboje díky evolučním algoritmům), vyšší přesnost, rychlost a zaručenou konvergenci (poslední tři díky deterministickým optimalizačním algoritmům).

Překážkou použití evolučních algoritmů je nutnost aplikovat algoritmus znovu pro každou novou úlohu, protože je nutné přizpůsobit daný algoritmus konkrétnímu problému. Důvody přizpůsobení jsou dva. První důvodem jsou samotné operace, které se nepodařilo zobecnit. Druhým je zefektivněni algoritmu, tj. snížení časové náročnosti, kde se vloží do výpočtu znalost problému. V praxi se častěji používají aproximační algoritmy, které zaručují jistou maximální chybu a jsou v dané oblasti již používané [22].

2.4 Genetické algoritmy

Genetické algoritmy[4] jsou evoluční stochastické optimalizační metody, které pracují na principu metody přímého prohledávání a na principu generování a testování. Procházejí prostor možných řešení a tato řešení vyhodnocují. Do této kategorie patří

[4]Detailnější popis genetických algoritmů je např. na `https://akela.mendelu.cz/~xpopelka/cs/ui/ucici/`.

i A* algoritmus[5]. Cílem je najít stav, který splňuje omezující podmínky optimalizace a současně poskytuje minimální hodnotu optimalizované funkce. Algoritmus pracuje numericky (bez znalosti přesného tvaru optimalizované funkce) a vyžaduje pouze znalost výsledných hodnot v daném bodě stavového prostoru. Tyto stavy jsou dále předány kriteriální funkci, která stavy porovnává a určí lepší.

Základem genetických algoritmů je simulace evolučních procesů a zákonů dědičnosti. V živé přírodě je spousta otázek, které nejdou přesně nasimulovat, a proto se v živé přírodě pouze inspirují a konkrétní postupy se vytváří podle úlohy. Používá se zde terminologie z biologie a informatiky. Genetické algoritmy se od ostatních optimalizačních metod liší tím, že mají paralelní přístup k řešení úlohy. Mají množinu možných řešení, které se říká populace a mají více prohledávaných bodů, kterým se říká jedinci, kde každý se snaží najít optimum funkce [22].

[5]Algoritmus A* (A star) se používá k vyhledávaní optimální cesty v kladně ohodnoceném grafu (nebo jiném typu vstupních dat, např. grafové mřížky nebo souřadnice) a pracuje na základě Dijkstrova algoritmu, ale navíc má heuristický výběr cesty k vrcholu (resp. heuristický výpočet váhy dané cesty k určitému vrcholu, viz dále v kapitole o A* algoritmu).

3 PŘÍSTUPOVÉ A TRANSPORTNÍ SÍTĚ

Práce se zabývá distribuovanými a stochastickými algoritmy, které se používají v síťové problematice a možnostmi konvergence těchto algoritmů v přístupových a transportních telekomunikačních sítích. Proto je vhodné tyto sítě v této kapitole popsat.

Sítě obecně prodělaly za posledních par desítek let určitý vývoj, kdy kromě obecně známé digitalizace se objevilo i mnoho dalších věci, jako např. navýšení přenosových rychlosti, zlepšení služeb z pohledu uživatelů, kdy uživatelé můžou mít například svou vlastní domácí nebo mini síť, nové služby obecně, nové technologie (Ethernet přes optiku, průmyslový Ethernet), nová přenosová media (resp. techniky, např. pokročilejší modulace), novější a rychlejší síťové prvky (vývoj v oblastí CPU a HW) atd. Posledních par let vývojovým trendem v oblasti sítí byla konvergence ke které se směruje i nyní. Zde se konvergenci rozumí sbližování, a to všech sítí a jejich technologií, signálů přenášených po sítích, protokolů, služeb a dalších věci, co souvisí se sítěmi, do jedné hlavní sítě. Po této jediné sítí bude možné provozovat několik služeb jako např. telefonování, audio a video přenosy, datové přenosy a další. Za zmínku ještě stoji vývoj SDN (softwarově definovaných sítí, anglický Software Defined Networking), ale jejich směr, uplatnitelnost, schopnosti a pozice mezi běžně zaběhnutými postupy ještě není známa. Toto všechno ukáže až čas a praxe.

V této kapitole jsou popsané přístupová síť, která je z pohledu obyčejných uživatelů nejzajímavější, neboť uživatele chtějí platit nebo platí za určité služby, např. streamování hudby a videí, což znamená, že potřebují nepřerušovaný přístup k těmto službám a transportní síť, která je důležitá z pohledu funkčnosti přenosů velkých objemů dat na velké geografické vzdálenosti a která je důležitá z pohledu všech ostatních, např. poskytovatelů internetového připojení. Oboje typy jsou svojí problematikou značně rozsáhlé, a proto v kapitole bude jen stručný přehled.

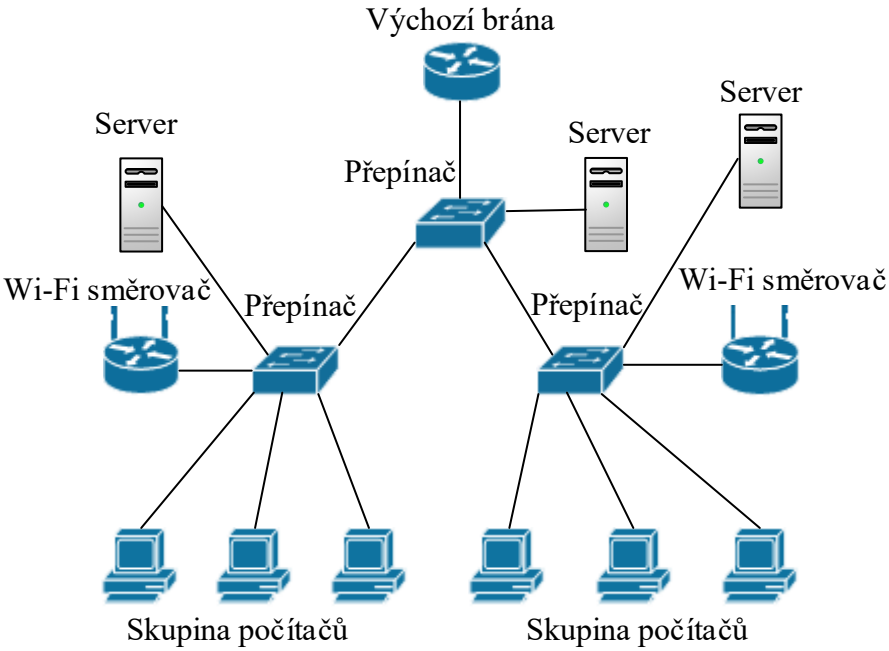
Pokud je síť jako celek dělena na hierarchický model, tak kromě přístupové a transportní vrstvy existuje ještě distribuční vrstva, která se stará o přenos dat mezi přístupovou a transportní vrstvou. Distribuční vrstva není naplní této práce. Transportní vrstvě se ještě podle hierarchického modelu říká páteřní vrstva. Dále pokud v práci budou zkoušený algoritmy, tak to bude vždy v souvislostí s přístupovou a transportní vrstvou.

3.1 Přístupová síť

Přístupová síť umožňuje koncovému uživateli přístup k sítí. Přesněji umožňuje připojení koncového zařízení k prvnímu uzlu sítě, který data (resp. zprávu, komunikaci)

pošle dál v síti, tj. pošle dalším uzlům. Funguje to i obráceně, což je ve směru ze sítě k uživateli, kdy data (resp. zpráva) dojde k nejbližšímu uzlu k uživateli. Uživateli se říká pasivní účastník. Zjednodušeně se dá říct, že přístupová síť je od uživatele (resp. účastníka) až ke vstupu prvního veřejného síťového prvku (např. ústředna u telefonní sítě).

Odborněji se přístupovou sítí nazývá soubor všech technických prostředků, které umožňují přístup zákazníkům ke službám poskytovaným provozovatelem sítě. Z hlediska přenosové techniky přístupová síť propojuje koncové body sítě s účastnickým rozhraními se síťovými uzly transportní sítě, která zajišťuje jejich propojení na centrální prvky sítí telekomunikačních služeb [17].



Obr. 3.1: Ukázka přístupové sítě v podobě LAN

Do přístupové sítě patří i pojem počítačové sítě LAN (Local Area Network) - lokální (někdy také místní) síť, která je nejčastěji v podobě technologie Ethernet. Příklad takové sítě je na obrázku 3.1. V tomto případě existuje nějaká lokální počítačová síť, kterou vlastní např. menší podnik, ale vlastníkem může být i obyčejný uživatel (resp. zákazník, jednotlivec), např. běžný uživatel, který má doma svou síť. V takové síti může být několik počítačů propojených navzájem nebo přes např. přepínač. Dále od přepínače může vést linka na směrovač (tj. v tomto případě výchozí bránu), který směruje na první veřejný uzel, bezdrátový přístupový bod, Wi-Fi

(údajně Wireless Fidelity - bezdrátová věrnost) směrovač, nebo jiný aktivní síťový prvek (může být i pasivní, záleží na architektuře a velikosti LAN), který se ještě nachází v LAN. Za výchozí branou už je většinou konec přístupové sítě a se vstupem na síťový prvek poskytovatele sítě už začíná transportní síť.

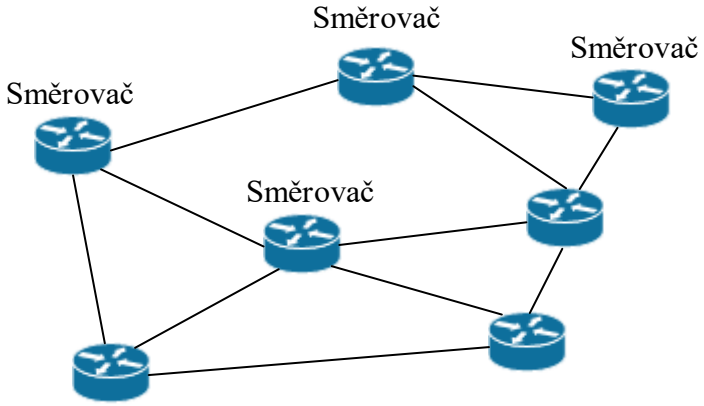
Uvedený příklad na 3.1 je značně zjednodušený. Ve skutečnosti síťové prvky musejí být redundantní (aspoň co se podniků tyče), což na příkladu se jedna zájmena o výchozí bránu, servery a přepínače. Dále architektura může být složitější než je na obrázku, např. může být propojeno více serverů mezi sebou apod. Také záleží na protokolech, službách a komunikaci uvnitř LAN - např. servery můžou v rámci určitého protokolu komunikovat pouze mezi sebou nebo naopak to stejné můžou provádět počítačové stanice. Příklad na obrázku zároveň slouží i jako ukázka použíti distribuovaného systému, kde se můžou aplikovat distribuované (resp. stochastické) algoritmy.

3.2 Transportní síť

Transportní síť umožňuje přenos a směrování mezi prvním a posledním uzlem sítě, tj. zařizuje přenos a směrování u veřejné sítě, kterou vlastní poskytovatel připojení nebo jiný subjekt. Nachází se zde nepřeberné množství přenosových technologií jako jsou přenosy po optických spojích SDH (Synchronní digitální hierarchie, anglický Synchronous Digital Hierarchy), vysokorychlostní Ethernet, který může být realizován metalickým nebo optickým kabelem nebo ATM (Asynchronní transportní mód, anglický Asynchronous Transfer Mode), který je nyní na ústupu, ale stále se ještě používá [17].

Do transportní sítě spadají i pojmy počítačových sítí MAN (Metropolitan Area Network) - velkoměstské sítě a WAN (Wide Area Network) - rozsáhlé sítě. Oboje typy sítí lze realizovat různými přenosovými technologiemi a tyto technologie jsou popsané výše v úvodu podkapitoly. Na obrázku 3.2 je ukázka transportní sítě v podobě WAN, kde uzly jsou zobrazený aktivními síťovými prvky - směrovači. Směrování může probíhat na základě IP protokolů a to verze 4 nebo 6 (detaily nad rámec účelu této práce) nebo jiného protokolu. Příklad zároveň slouží jako ukázka distribuovaného systému, kde se můžou aplikovat distribuované nebo stochastické algoritmy. Pro přehlednost nadpis „směrovač“ není uveden nad každým uzlem (resp. směrovačem).

Jako v předešlé podkapitole o přístupové síti i na obrázku 3.2 došlo ke značnému zjednodušení. Kromě toho, že každý uzel může obsahovat svou vlastní přístupovou síť pod sebou, tak ve skutečností ilustrační síť z obrázku (resp. oblast) může byt značně rozsáhlá, co se uzlů tyče a uzly samotné nemusejí byt pouze v podobě



Obr. 3.2: Ukázka transportní sítě v podobě WAN

směrovačů, ale může jít o telefonní ústředny (resp. PBX ústředny), DNS (Domain Name System) servery nebo multifunkční zařízení, které v sobě má několik funkci nebo služeb zároveň. Uzlů může komunikovat vzájemně několik současně a samotná komunikace je složitější, kdy zároveň může pracovat několik protokolů z několika vrstev referenčního modelů ISO/OSI (resp. TCP/IP). Všechny uzly ale vykazují určité vlastnosti distribuovaného systému, jako je např. autonomnost uzlu apod. Uzly (resp. směrovače) vykonávají určitý proces paralelně mezi sebou a společně komunikují přes síť.

4 MATEMATICKÉ NÁSTROJE

Při práci s distribuovanými a stochastickými algoritmy a algoritmy obecně, které se používají v sítích nebo struktuře podobné sítím, se nejvíce používají matematické nástroje z oblasti teorie grafů a lineární algebry. Grafy z důvodů podobností sítí k této struktuře a lineární algebra z důvodů matematické reprezentace grafů, což jsou nejčastěji maticové výpočty a práce s nimi. V problematice sítí a algoritmů jsou obě oblasti vzájemně propojené a v této kapitole budou oba nástroje popsané:

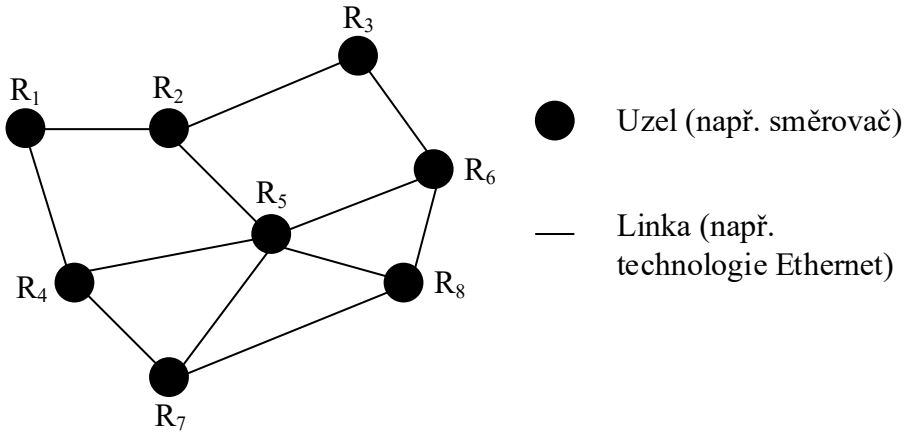
4.1 Teorie grafů

Matematická oblast teorie grafů se zabývá analýzou grafů. Graf se skládá z množiny vrcholů, které jsou navzájem propojené hranami. Matematická definice grafu je taková [2] [11] [20]:

$$G = (V, E), \tag{4.1}$$

kde V je množina vrcholů grafu G a E je množina hran.

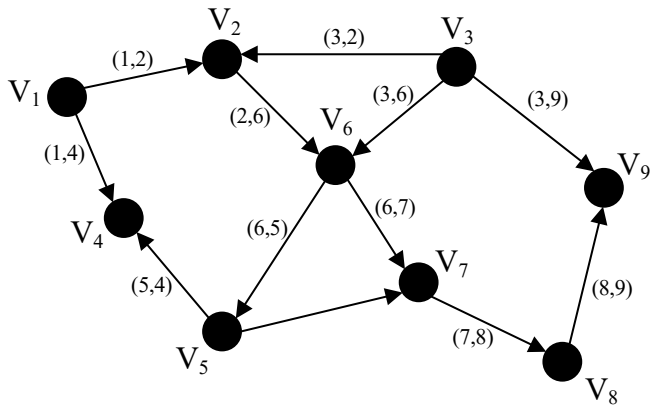
V souvislosti se síťovou problematikou si můžeme představit, že vrcholy jsou jednotlivé uzly (přepínače, směrovače apod.) a hrany linky, kterými jsou jednotlivé uzly propojené, pro představu viz 4.1. Aby se uzly od sebe rozlišily, tak se indexují, jako v následující ukázce matematického zápisu, kde množina vrcholů V má v sobě n vrcholů: $V = (v_1, v_2, \ldots, v_n)$, n je přirozené číslo.



Obr. 4.1: Ukázka zobrazení sítě v podobě grafu

Grafy se dělí na dva typy, kterými jsou orientované a neorientované a jejich několik podtypů, které se dále v textu popíšou.

Prvním typem jsou orientované grafy. U orientovaného grafu záleží na pořadí vrcholů, tj. u hrany je počáteční a koncový vrchol. Dá se říct, že hrana má orientaci. Taková hrana se může označit šipkou od počátečního uzlu ke koncovému a všechny vrcholy orientovaného grafu jsou nerovnocenné, tj. neplatí symetrie 4.2. Na obrázku 4.2 je ukázka orientovaného grafu [2] [20].

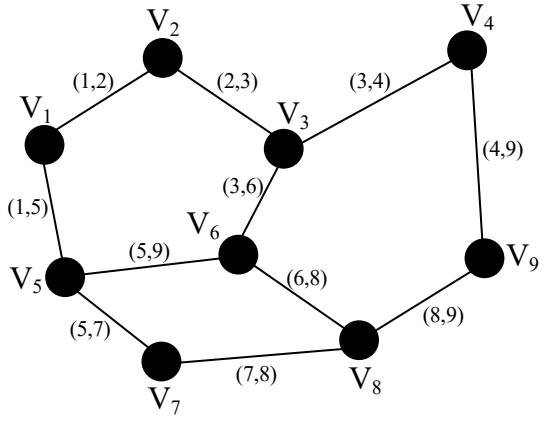


Obr. 4.2: Příklad orientovaného grafu

Druhým typem grafů jsou neorientované grafy, kde není počáteční a koncový vrchol u hrany, tzn. všechny vrcholy v grafu jsou rovnocenné. Dá se říct, že hrana nemá orientaci. Na obrázku 4.3 je ukázka neorientovaného grafu. Matematický zápis takového grafu je $\{i,j\} \in \epsilon$. Jelikož zde není směrová orientace hran, tak platí tvrzení o symetrii [2] [20]:

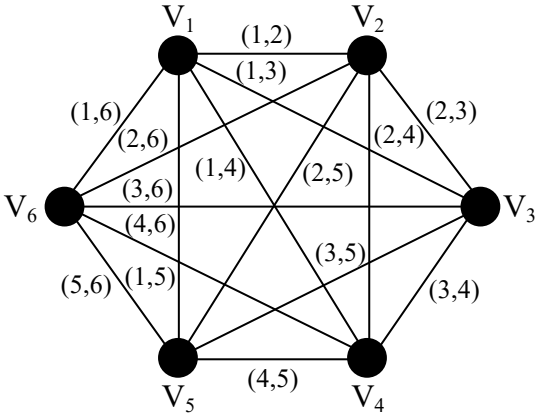
$$(i, j) = (j, i). \tag{4.2}$$

Speciálním podtypem neorientovaného grafu je graf, kde každý vrchol je propojený se všemi ostatními vrcholy v daném grafu, viz 4.4. Odborně se takový graf nazývá úplným grafem. Matematický zápis takového grafu je pro množinu vrcholů: $V = (v_1,v_2,v_3,v_4,v_5,v_6)$. Pro množinu hran daného grafu je zápis delší: $E = \{(v_1,v_2),(v_1,v_3),(v_1,v_4),(v_1,v_5),(v_1,v_6),(v_2,v_3),(v_2,v_4),(v_2,v_5),(v_2,v_6),(v_3,v_4),(v_3,v_5),(v_3,v_6),(v_4,v_5),(v_4,v_6),(v_5,v_6)\}$. Některé hrany nejsou uvedené, protože jsou duplicitní, což znamená, že již v množině jsou, ale v jiném tvaru, tj. zápis (v_1,v_3) je stejný jako (v_3,v_1). Toto navíc plyne z výše uvedeného textu popisujícího symetrií



Obr. 4.3: Příklad neorientovaného grafu

u neorientovaného grafu, viz 4.2. Matematický správně je třeba ještě uvést hrany $\{(v_1,v_1),(v_2,v_2),(v_3,v_3),(v_4,v_4),(v_5,v_5),(v_6,v_6)\}$. Na první pohled se může zdát, že jde o čistě teoretický a nesmyslný zápis z matematiky, ale v reálném světě tento zápis potvrzuje funkci smyčky (loopback[1]) u rozhraní [2] [20].

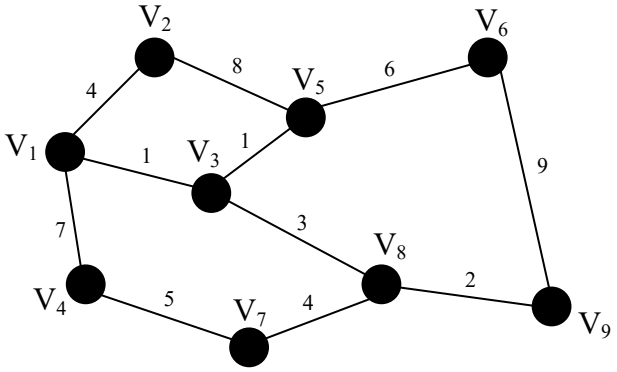


Obr. 4.4: Příklad úplného grafu

Kromě výše popsaných typů grafů je ještě potřeba zmínit se o tzv. vážených grafech. Vážený graf je takový orientovaný nebo neorientovaný graf, kde každá hrana

[1]Simulace komunikace na jiné rozhraní, ale ve skutečnosti dané rozhraní komunikuje samo se sebou.

má přidělenou váhu (weight), která se značí písmenem w. Váha je obvykle dána váhovou funkci w: $\mathbb{E} \to \mathbb{R}$. Pro většinu algoritmů by měla mít kladnou hodnotu. Na obrázku 4.5 je ukázka takového grafu. Jedna se o nejrozšířenější variantu grafu v praxi, která se používá např. u směrovacích protokolů telekomunikačních sítí, protože na základě váhy linky (resp. cesty) se protokoly rozhodují, která cesta je lepší (resp. cíl je blíže, cesta k němu je rychlejší apod.) [2].



Obr. 4.5: Příklad neorientovaného váženého grafu

Pro upřesnění je vhodné ještě uvést par doplňujících pojmů z oblasti teorie grafů [2]:

- **Stupeň vrcholu** je počet hran vycházející z vrcholu. V orientovaném grafech se ještě rozlišuje na vstupní a výstupní stupeň.
- **Stupeň grafu** odpovídá nejvyšší hodnotě stupně vrcholu v grafu G.
- **Průchod** je sekvence vrcholů z určitého startovního vrcholů k určitému cílovému vrcholu.
- **Jednoduchá cesta** je průchod bez opakování vrcholů.
- **Cyklus** je průchod bez opakování vrcholů $(V_1 \dots V_n)$, kde prvním a posledním vrcholem je stejný vrchol, tj. $V_1 = V_n$.
- **Cyklický** graf je tehdy pokud obsahuje alespoň jeden cyklus. Jinak se nazývá **acyklickým**.

4.2 Lineární algebra

Reprezentace grafů pomocí obrázků jako v minulé podkapitole je pro člověka přehledná a názorná, ale pro výpočty na počítači a dále pro podrobnou matematickou analýzu je nutné graf zadat pomocí čísel. Je potřeba matematická reprezentace grafů,

která se nejčastěji realizuje pomocí matice sousednosti, ale existují i další způsoby a metody. Pro účely této práce zcela postačuje reprezentace grafu pomocí matice sousednosti se kterou se dá pracovat dále např. v prostředí MATLAB[2]. Převod grafu na matici sousednosti bude popsán v dalším textu.

Pro lepší pochopení je dobré převod grafu na matici sousednosti ukázat na příkladu, ale nejprve je třeba definovat, co to vlastně matice sousednosti je. Matematická definice matice sousednosti je následující [20]:

Nechť G,E je graf s n vrcholy. Označme vrcholy $v_i,\ldots,v_n$ (může byt libovolné pořadí). Matice sousednosti grafu G je čtvercová matice:

$$A_G = (a_{ij})_{ij=1}^{n} \tag{4.3}$$

která je definovaná předpisem:

$$a_{ij} = \begin{cases} 1 & \text{pro} \quad \{v_i, v_j\} \epsilon E \\ 0 & \text{jinak} \end{cases}$$

Ještě platí, že pro orientovaný graf je výsledná matice nesymetrická podle diagonály. Hrana má orientaci, a tak cesta z vrcholu do vrcholu je možná pouze v jednom směru (nejde zpět přes stejnou hranu), což způsobí, že hodnota s jedničkou bude pouze v jedné části symetrie podle diagonály.

Matice neorientovaného grafu je naopak vždy symetrická, protože cesta přes hranu je možná z vrcholu do vrcholu v obou směrech (tam a zpět přes stejnou hranu). Hodnota s jedničkou tak bude u obou částech symetrie podle diagonály. Pokud je neorientovaný graf úplný, tak matice obsahuje samé jedničky kromě hlavní diagonály.

V telekomunikační praxi, praktickém využíti algoritmů a v této práci se nejčastěji využívá neorientovaných grafů. Proto bude popsán způsob vytváření matice pro neorientovaný graf. Samotný způsob vyvaření matice sousednosti je následující [20]:

1. Nejprve se všechny vrcholy grafu očísluji.
2. Za rozměr matice se zvolí počet vrcholů, tj. pokud má graf deset vrcholů, tak rozměr matice bude 10×10.
3. Pokud vede mezi dvěma vrcholy hrana, zapíšeme do matice na pozici [*číslo prvního vrcholu, číslo druhého vrcholu* - a_{12}] a na pozici [*číslo druhé vrcholu, číslo prvního vrcholu* - a_{21}] jedničku. Jinak se zapíše nula.

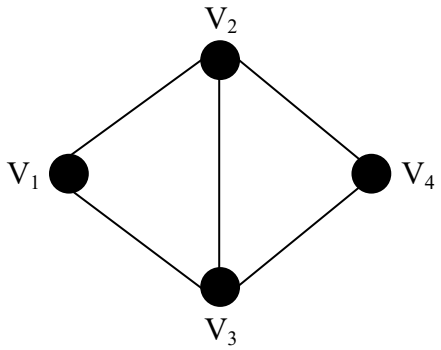
[2]Programové prostředí MATLAB se na maticové výpočty specializuje - odtud i název z anglického MATrix LABoratory (maticová laboratoř).

Nyní ke konkrétnímu příkladu. Na obrázku 4.6 je neorientovaný graf se čtyřmi vrcholy. Nejdříve je ukázaná matice pozic jednotlivých prvků pro vysvětlení zápisu hodnot do těchto pozic z ukázkového grafu. Dále je výsledná matice sousednosti, která podle výše popsaného postupu bude pro tento graf následující:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

Matice má rozměr 4×4, protože v grafu jsou čtyři vrcholy. Matice je symetrická podle diagonály, protože se jedná o neorientovaný graf a v diagonále jsou nulové hodnoty, protože např. pozice a_{11} značí hranu z vrcholu V_1 do vrcholu V_1, což na obrázku je vidět, že taková hrana není, a tak do matice na pozici a_{11} se zapíše nula. Stejné platí pro pozice a_{22}, a_{33}, a_{44}. Pro zjednodušení se dá představit, že nalevo od matice je pozice „odkud“ hrany vedou, tedy startovní vrcholy a nad matici nahoře je pozice „kam“ hrany vedou, tedy cílové vrcholy.

Samotný zápis hodnot tedy probíhá následovně: Z vrcholu V_1 do vrcholu V_2 vede hrana, takže pozice a_{12} bude s hodnotou jedna. Zároveň platí, že z vrcholu V_2 do vrcholu V_1 vede taky hrana, tj. na pozici a_{21} bude jednička též, což značí symetrií, tj. platí $(a_{12}) = (a_{21})$, viz 4.2. Hrana z vrcholu V_1 k vrcholu V_4 neexistují, tj. na pozici a_{14} a zároveň na a_{41} (cesta zpět z vrcholu V_4 na vrchol V_1) bude nula. Obdobně se zapisují zbylé pozice až matice je plná.



Obr. 4.6: Příklad neorientovaného grafu pro ukázku převodu na matici sousednosti

Obor lineární algebry je obsáhly, kromě reprezentace grafů pomocí matice sousednosti existuji ještě další metody, jak číselně zobrazit graf, např. pomocí Laplaca-

ové matice. Dále existují různé typy matic a práce s nimi, např. schodovitá matice, matice transponovaná, nulová matice, jednotková matice apod. Všechno je to nad rámec této práce - zde je ještě vhodné se zmínit o jednotkové matici:

Jednotková matice je čtvercová matice, která má v diagonále samé jedničky a ve zbytku pozic nuly. Matice má stejné vlastnosti jako jednička u normální matematiky, tj. součin libovolné matice s jednotkovou matici se rovná stejné matici na kterou se násobila jednotková matice. Matematické vyjádření tohoto tvrzení je takové [9]:

$$I.A = A, \quad \text{nebo ekvivalentní} \quad A.I = A, \tag{4.4}$$

kde I je jednotková matice a A je jakákoliv matice. Příklad jednotkové matice o rozměru 4×4:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

5 VÝBĚR ALGORITMŮ

Než bylo možné pustit se do praktické části, tak bylo třeba vybrat vhodné algoritmy se kterými se dál pracovalo. Z velkého počtu distribuovaných algoritmů nakonec byly vybrané dva a stejné množství bylo vybrané ze stochastických algoritmů, ale jejich výběr byl ztížen omezeným počtem algoritmů využívaných v síťové problematice nebo v praxi. Většinou se také jedná pouze o experimentální algoritmy ve fázi zkoumání nebo testování, tzn. že ještě nejsou nasazené v protokolech ani zařízeních a navíc jim chybí detailně propracovaná literatura (dokumentace) jako v případě distribuovaných algoritmů. Důvodem nedostatku literatury z velké pravděpodobnosti bude, že se jedná o nový přístup k řešení problematiky v sítích, který se objevil v posledních letech pomocí samotného rozvoje stochastických algoritmů a výpočetní síly v zařízeních.

5.1 Výběr dvou distribuovaných algoritmů

Po zhodnocení aktuálně dostupných distribuovaných algoritmů nakonec byly vybrané dva. První byl Bellmanův-Fordův a druhým byl Dijkstrův. Důvody proč zrovna tyto dva algoritmy jsou dva. Prvním je jejich běžné využíti v sítích. Konkrétně Bellmanův-Fordův je implementovaný ve směrovacím protokolu RIP[1] a Dijkstrův v protokolu OSPF[2]. Druhým důvodem byla zvědavost zjistit, který z těchto dvou algoritmů je rychlejší a afektivnější při vzájemném porovnaní na různých topologiích, jelikož jsou běžně využívané v praxi a co se tyče sítí, jestli je rychlejší a efektivnější OSPF typu *link-state*, kde se počítá tzv. cena linek (přesněji váha, např. rychlost linky, zpoždění, a další parametry specifikované v určitém protokolu) nebo RIP s distančním vektorem (*distance-vector*), kde se počítá počet skoku k cílí (hops).

5.1.1 Bellmanův-Fordův algoritmus

Algoritmus byl navrhnut Alfonsem Shimbelem v roce 1955, ale je pojmenovaný po Richardu Bellmanu a Lesteru Fordu, jenž oba algoritmus publikovali v roce 1958 a 1956. Někdy je nazýván Bellmanův-Fordův-Moorův algoritmus podle Edwarda F. Moorea, který stejný algoritmus paralelně a nezávislé také publikoval v roce 1957.

Algoritmus počítá nejkratší cestu ve váženém grafu mezi dvěma libovolnými vrcholy, kde hrana mezi nimi může být vážená i zápornou hodnotou. V reálných sítích

[1]RIP - Routing Information Protocol je směrovací protokol typu *distance-vector* (využívající vektor vzdálenosti).

[2]OSPF - Open Shortest Path First je směrovací protokol typu *link-state*, kde každý směrovač v síti ví o všech ostatních a zjednodušené řečeno zná mapu sítě.

takový případ nastat nemůže, protože všechny aktuálně používané směrovací protokoly mají výpočet metriky v kladné hodnotě. Algoritmus má velké využití i v jiných oblastech než sítě. Algoritmus využívá metodu relaxace hran[3], která zajišťuje zjištění nejkratší vzdálenosti od hodnoty vrcholu *s*. Pokud je zjištěno, že hodnota v novém vrcholu je vyšší než hodnota aktuálního vrcholu plus ohodnocení hrany z nynějšího vrcholu do vrcholu nového, pak tuto hodnotu snížíme. Vrcholy se prochází několikrát a postupně se tak upravuje hodnota vzdálenosti nejkratších cest. Matematický řečeno algoritmus se snaží postupně zlepšovat hodnotu $d[u]$. Jakmile se u vrcholu u zlepší hodnota $d[u]$, musí se prozkoumat všechny hrany $(u,v)\ \epsilon\ H$ a pokud to bude možné, tak zlepšit hodnotu $d[u]$ (operace relaxace). Algoritmus vrátí hodnotu *true* právě tehdy, když graf neobsahuje cyklus záporné délky dosažitelný z počátečního vrcholu *s*. Obecná implementace v pseudokódu (převzato z [1]):

Pseudokód 1 Bellman-Fordův algoritmus

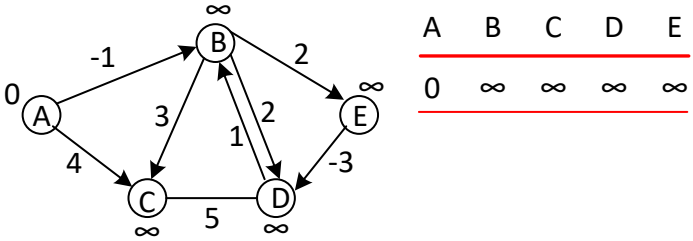
```
function BELLMAN-FORD(vrcholy, hrany, zdroj)
    Inicializace;
    for každé v ve vrcholy do
        if v = zdroj then v.vzdálenost :=0 then
        else if v.vzdálenost := nekonečno then
            v.předchůdce := null
    Opakovaná relaxace hran;
    for i = 1 až velikost(vrcholy) - 1 do
        for každé h v hrany do              ▷ h je hrana z vrcholu u do vrcholu v
            u := h.počátek
            v := h.konec
            if u.vzdálenost + h.délka < v.vzdálenost then
                u.vzdálenost := u.vzdálenost + h.délka
                u.předchůdce := u
    Volitelná část: kontrola záporných cyklů;
    for každé h v hrany do
        u := h.počátek
        v := h.konec
        if u.vzdálenost + h.délka < v.vzdálenost then
            error "Graf obsahuje záporný cyklus"
```

[3]Přesněji do této metody vstupují dva vrcholy a hrana mezi nimi. Pokud je vzdálenost zdrojového vrcholu sečtená s délkou hrany menší než aktuální vzdálenost cílového vrcholu, tak se za předchůdce cílového vrcholu na nejkratší cestě označí zdrojový vrchol (přepočte se vzdálenost cílového vrcholu). V případě nesplnění nerovnosti se neprovádí žádné změny.

V prvním cyklu algoritmus nastaví všem vrcholům kromě zdroje nekonečno a samotnému zdroji nulu. Druhy cyklus upravuje hodnoty vzdálenosti mezi zdrojem a ostatními vrcholy. Třetí cyklus kontroluje, jestli už některá určena hodnota nemůže byt ještě zkrácena. Nejdelší možná cesta může být $[\boldsymbol{V}]$ - **1** a složitost algoritmu je $\boldsymbol{O}(|\boldsymbol{V}|\times|\boldsymbol{E}|)$, kde $[\boldsymbol{V}]$ je počet vrcholů a $[\boldsymbol{E}]$ je počet hran. Jak již bylo uvedeno v předchozích odstavcích, tento algoritmus se používá u směrovacího protokolu RIP [23] [3] [11].

Příklad fungování algoritmu

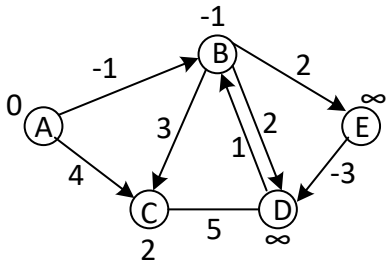
Pro lepší pochopení algoritmu je dobré si ukázat přímo příklad činnosti algoritmu na určitém grafu. Graf je neorientovaný, ale pro přehlednost je šipkami zobrazen směr zpracování jednotlivých hran ze zdrojového vrcholu. Mějme zdrojový vrchol ***A*** s vzdálenosti 0 a na počátku inicializujme všechny vzdálenosti u ostatních vrcholů na nekonečno, viz 5.1. Celkový počet vrcholů v grafu je 5, tzn. každá hrana bude počítaná čtyřikrát, což plyne z teorie popsané v předešlém textu.



Obr. 5.1: Příklad počátečního stavu algoritmu

Mějme následujícím pořadí zpracování hran: (***B,E***), (***D,B***), (***B,D***), (***A,B***), (***A,C***), (***D,C***), (***B,C***), (***E,D***). Při první iteraci dostaneme vzdálenosti jako na obrázku 5.2. První řádek zobrazuje počáteční vzdálenosti. Druhý řádek zobrazuje vzdálenosti, když jsou zpracované hrany (***B,E***), (***D,B***), (***B,D***) a (***A,B***). Třetí řádek zobrazuje zpracování hrany (***A,C***) a poslední řádek zobrazuje zpracování hran (***D,C***), (***B,C***) a (***E,D***). Dále při první iteraci dostaneme všechny nejkratší cesty s maximální vzdálenosti o velikosti jedné hrany. Konečné hodnoty druhé iterace jsou na 5.2 v posledním řádku.

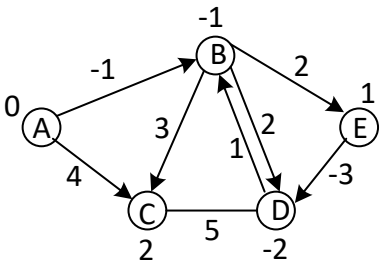
Na posledním obrázku 5.3 je zobrazená druhá iterace algoritmů (iterace je vždy po tenčí červené linii). Druhá iterace zaručuje cesty s maximální vzdálenosti o velikosti dvou hran. Algoritmus zpracuje všechny hrany ještě dvakrát a vzdálenosti



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞

Obr. 5.2: Příklad zpracování hran při první iteraci algoritmu

jsou tak minimalizované po druhé iteraci, takže třetí a čtvrtá iterace už nezlepší vzdálenosti [3].



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1

Obr. 5.3: Příklad zpracování hran při druhé iteraci algoritmu

Ukázkový příklad má schválně i zápornou hodnotu váhy u hrany ***AB***, aby bylo názorně ukázáno, že algoritmus dokáže počítat i s takovýmto případem. V praxi se záporné hodnoty u hran mezi vrcholy nepoužívají. Algoritmus postupně zlepšoval hodnotu vzdálenosti $d[u]$ k jednotlivým vrcholům v grafu. Odtud pochází i název distančně vektorový (*distance-vector*) protokol, kdy jednotlivé vrcholy v grafu (resp. topologií, síti) si udržují hodnoty vzdálenosti k ostatním vrcholům.

5.1.2 Dijkstrův algoritmus

Navržen byl v roce 1956 Edsgarem W. Dijkstrou a publikován v roce 1959. Nejdříve byl uplatněn v armádě a až později se použil v civilním sektoru. Existuje spousta upravených variant algoritmu. Původní varianta od samotného Edsgara Dijkstry počítala nejkratší cestu mezi dvěma vrcholy.

Algoritmus si pro každý vrchol pamatuje délku nejkratší cesty, přes kterou se dá k danému vrcholu dostat. Tato hodnota má označení $d[v]$ a na začátku výpočtu mají všechny vrcholy tuto hodnotu nastavenou na nekonečno $d[v] = \infty$, což znamená, že cestu k danému vrcholu ještě neznáme. Počáteční vrchol má $d[s] = 0$. Dále se udržují dvě množiny $\boldsymbol{Z}$ a $\boldsymbol{N}$, kde $\boldsymbol{Z}$ obsahuje navštívené vrcholy a $\boldsymbol{N}$ nenavštívené. Algoritmus pracuje dokud množina $\boldsymbol{N}$ není prázdná. V každém průchodu se přidá jeden vrchol z množiny $\boldsymbol{N}$ do množiny $\boldsymbol{Z}$, který má nejmenší hodnotu $d[v]$ ze všech vrcholů v množině $\boldsymbol{N}$.

U každého vrcholu u do kterého vede hrana, se provede tato operace: pokud $d[v_{min}] + l(v_{min}) < d[u]$, pak do $d[u]$ přiřaď hodnotu $d[v_{min}] + l(v_{min},u)$, jinak nedělej nic. Kde v_{min} je vrchol z množiny $\boldsymbol{N}$ s nejmenší hodnotou $d[u]$ a $l(v_{min},u)$ je délka hrany z v_{min} do libovolného vrcholu u. Nematematický řečeno Dijkstrův algoritmus řeší problém nejkratších cest z kořene s do ostatních vrcholů grafu pro grafy s nezáporně váženými hranami. Algoritmus udržuje množinu $\boldsymbol{S}$ vrcholů, pro které se už vypočítá délka nejkratší cesty grafu. Algoritmus opakovaně vybírá vrchol $u \epsilon \boldsymbol{V}$ - $\boldsymbol{S}$ s nejkratší cestou a relaxuje hrany vycházející z vrcholu u. Po skončení algoritmu je u každého vrcholu uložena nejkratší cesta od zdroje v $d[u]$. Ukázka pseudokódu je následující (převzato z [7]):

Pseudokód 2 Dijsktrův algoritmus

```
function DIJKSTRA(E, V, s)
    Inicializace;
    for každý vrchol v ve V do
        d[u] := nekonečno
        p[v] := nedefinovano
        d[s] := 0
        N := V
```

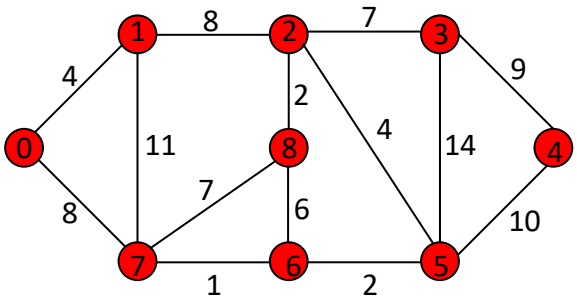
Složitost algoritmu je u základní implementace s prioritní frontou $\boldsymbol{O}(|\boldsymbol{V}|^2+|\boldsymbol{E}|)$ a dá se dále zkrátit pomoci binární haldy na $\boldsymbol{O}((|\boldsymbol{E}|+|\boldsymbol{V}|)\mathbf{log}|\boldsymbol{V}|)$ a pomocí Fibonacciho haldy na $\boldsymbol{O}(|\boldsymbol{E}|+|\boldsymbol{V}|\mathbf{log}|\boldsymbol{V}|)$. Algoritmus je použit u směrovacího protokolu OSPF [4] [11].

Pseudokód 3 Dijsktrův algoritmus (pokračování)

```
Běh jádra algoritmu;
while N není prázdný do
    u := vytáhni_min(N)
    for každý soused v u vrcholu u do
        alt = d[u] + l(u, v)
        if alt < d[v] then
            d[v] := alt
            p[v] := u
    S := prázdna sekvence a u := cíl
    while definované p[u] do
        vlož na začátek S
        u := p[u]
```

Příklad fungování algoritmu

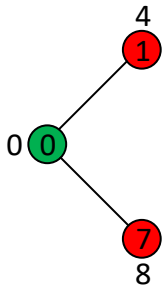
Jako u Bellman-Fordová algoritmu i zde je dobré pro pochopení ukázat činnost algoritmu na příkladu. Mějme následující neorientovaný graf, který je na obr. 5.4. Množina ***N*** je na začátku prázdna a vzdálenosti z vrcholu 0 k ostatním vrcholům jsou {0,∞,∞,∞,∞,∞,∞,∞}. V příkladu pro názornou ukázku činnosti algoritmu je zvolena cesta z vrcholu 0, který je zdrojem (resp. kořenem), k ostatním vrcholům. Jde o tzv. sestavení kostry grafu, kde bude vidět celkovou kostru grafu nejmenších vah, kterou si udržuje každý vrchol v grafu, a tak bude možné dohledat nejkratší cestu (resp. váhově nejlepší neboli s nejmenší celkovou váhou) od kořene k libovolnému vrcholů.



Obr. 5.4: Příklad neorientovaného grafu na kterém běží Dijkstrův algoritmus

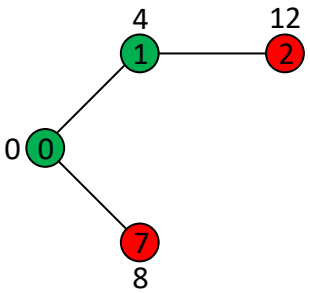
V prvním kroku se vezme vrchol s nejmenší vzdálenostní hodnotou, která je

váhou, nebo také cenou přiřazenou určitému vrcholu podle hrany k němu, což je v příkladu vrchol 0 a vloží se do množiny $\boldsymbol{N}$. Takže množina $\boldsymbol{N}$ bude obsahovat vrchol {0}. Po vložení vrcholu 0 do množiny $\boldsymbol{N}$ se aktualizují vzdálenostní hodnoty sousedních vrcholů. Sousedé vrcholu 0 jsou 1 a 7 a aktualizované vzdálenostní hodnoty budou 4 a 8. Na obrázku 5.5 je zobrazen subgraf sestavení kostry grafu Dijkstrovým algoritmem, kde zelená barva značí vrcholy sestavení nejkratších cest. Červená barva značí aktuálně zpracované vrcholy v číslovaném kroku.



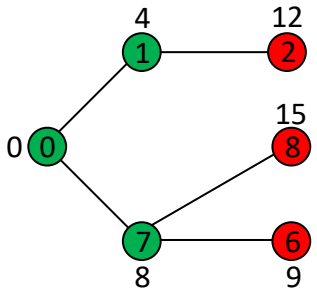
Obr. 5.5: Sestavení kostry grafu - 1. krok

V dalším kroku se vezme vrchol s nejmenší vzdálenostní hodnotou (váhou všech hran k danému vrcholu) a zároveň, který ještě není v kostře grafu. V příkladu to bude vrchol 1. Množina ($\boldsymbol{N}$) nyní obsahuje vrcholy {0,1} a dále se aktualizuji vzdálenostní hodnoty sousedů vrcholu 1, takže vzdálenostní hodnota vrcholu 2 bude 12 (součet vah hran k vrcholu 2 je 4+8), protože vrchol 2 je sousedem vrcholu 1, což je zobrazeno na obr. 5.6.



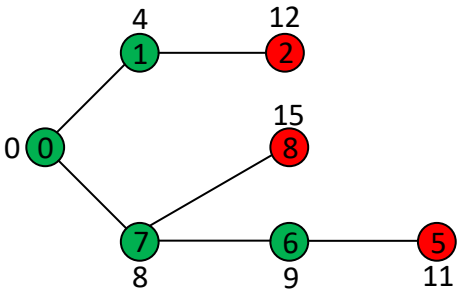
Obr. 5.6: Sestavení kostry grafu - 2. krok

Ostatní kroky jsou podobné. Takže v druhém kroku se opět vezme vrchol s nejmenší vzdálenostní hodnotou, tj. nejmenší váhou k danému vrcholu a zároveň, který ještě není přítomný v kostře grafu. Tentokrát to bude vrchol 7 a množina ($\boldsymbol{N}$) tak bude obsahovat vrcholy {0,1,7}. Aktualizují se vzdálenostní hodnoty (resp. váhy všech hran k určitým vrcholům) u sousedů vrcholu 7 a tak váha vrcholu 8 bude 15 (součet hran k vrcholu je 8+7) a vrcholu 6 bude 9 (součet hran k vrcholu je 8+1), viz 5.7.



Obr. 5.7: Sestavení kostry grafu - 3. krok

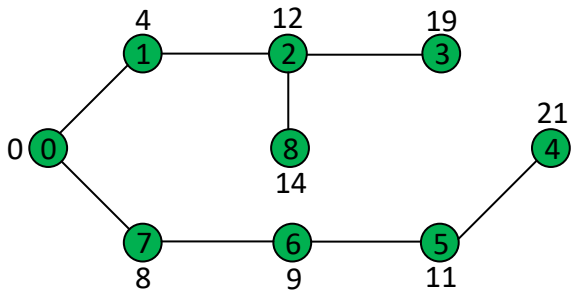
Nyní se vezme vrchol 6 a aktualizuji se vzdálenostní hodnoty sousedů tohoto vrcholu, což bude pouze vrchol 5 a bude mít vzdálenostní hodnotu (resp. váhu všech hran k němu) 11, kde součet vah hran na cestě k vrcholu je 8+1+2. Množina ($\boldsymbol{N}$) tak bude obsahovat vrcholy {0,1,7,6}, viz obr. 5.8.



Obr. 5.8: Sestavení kostry grafu - 4. krok

Postup se opakuje, dokud se neprojdou všechny vrcholy, tj. množina ($\boldsymbol{N}$) bude obsahovat všechny vrcholy grafu na kterém algoritmus běží a je tak možná cesta

mezi libovolnými dvěma vrcholy, která bude navíc nejkratší (resp. váhově nejlepší). Výsledná kostra grafu je na obr. 5.9 [4].



Obr. 5.9: Sestavení kostry grafu - výsledná kostra grafu

Jak je vidět na příkladu, tak každý vrchol si udržuje tzv. kostru grafu, kde každý vrchol pomocí této kostry má přehled o všech ostatních vrcholech v sítí. Odtud vzešel i název tzv. *link-state* (sledujících stav linky) protokolů. Mezi tyto protokoly patří už zmíněny OSPF, ale také IS-IS (Intermediate-System-to-Intermediate-System) a málo známý NLSP (NetWare Link Services Protocol). Při změně stavu linky (např. výpadku) se vypočítá (resp. sestaví) nová kostra grafu. Na zjištění stavu linky slouží v intervalech rozesílané ostatním vrcholům, *hello* zprávy.

5.2 Výběr dvou stochastických algoritmů

Výběr stochastických algoritmů, jak už bylo naznačeno na začátku kapitoly 5 byl složitější. Musely se vybrat algoritmy, které se používají nebo by se daly používat v síťové problematice. Většina stochastických algoritmů je zaměřená na řešení matematických problému, např. nalezení globálního minima nebo maxima u matematické funkce, anebo jinému vědnímu oboru, který nijak nesouvisí se sítěmi, anebo souvisí pouze značně okrajově. Když už se nějaký vhodný algoritmus našel, tak buď u něj chyběla detailní dokumentace, kde by byl algoritmus vhodně popsán, což by pomohlo pro snadnější implementaci v programovacích jazycích a následné hledání chyb, anebo byl pouze ve fázi teoretického návrhu (resp. matematického tvrzení), ale pro využití v reálných sítích je tak spousta problémů nedořešena.

Příčinou výše popsaného problému může být, že tyto algoritmy jsou novátorským přístupem oproti do nynějška používaným distribuovaným algoritmům, který

se rozmohl až s rozvojem samotných sítí, síťových zařízeních a s rozvojem samotných procesorů v síťových zařízeních. Může taky souviset s rozvojem ostatních věd v posledních desetiletích.

Nakonec po zkoumání všech aktuálně dostupných a pro síťovou problematiku vhodných stochastických algoritmů, byly zvolené dva. Prvním zvoleným algoritmem byl A* (A star), který hledá nejkratší cestu mezi dvěma vrcholy grafu, ale navíc využívá heuristiku, která je v tomto případě výpočet nejkratší vzdálenosti vzdušnou čárou ke koncovému vrcholu od aktuálně zpracovaného vrcholu. Druhým algoritmem je tzv. epidemický se šířící Push-sum algoritmus, kde epidemií se zde rozumí analogie k epidemickému šíření nemoci. Zde místo nemoci se šíří informace. V dalších podkapitolách budou oba algoritmy stručně popsané.

5.2.1 A* algoritmus

A*[4] (A star) je algoritmus určený pro hledání nejkratší cesty v grafu nebo v robotice na hledání cesty k cílí. Algoritmus je velice oblíbený a má uplatnění ve spoustě oblastech, kde se řeší hledání cest, např. počítačové hry, navigace, výše uvedena robotika apod. Algoritmus byl prvně popsán v roce 1968 Peterem Hartem, Nilsem Nillssonem a Bertramem Raphaelem jako rozšíření Dijkstrova algoritmu. Oproti Dijkstrovu algoritmu používá ke hledání cesty heuristiku.

A* je informovaný algoritmus, což znamená, že vybírá nejlepší nebo nejvhodnější (na základě času časově nejrychlejší, vzdušnou čárou nejkratší apod.) cestu z mnoha, které vedou k cílí. Podobně jako Dijkstrův algoritmus, A* začíná ze zdrojového vrcholu (z kořene) v grafu a sestavuje kostru grafu, která se zvětšuje o další vrcholy v každém kroku a pokud kostra grafu skončí v koncovém vrcholu, tak se nalezla nejlepší cesta ze zdroje k cíli.

V každé iteraci A* potřebuje určit jakou z částečných cest v kostře grafu rozšířit anebo jestli přidat další cestu. Děla se to na základě celkové váhy, která zbývá na cestě k cílí. Konkrétně A* vybírá cestu na základě $f(n)=g(n)+h(n)$, kde $f(n)$ je celková váha, n je poslední vrchol cesty, $g(n)$ je váha ze startovního vrcholu k n vrcholu a $h(n)$ je heuristika, která počítá nejlevnější cestu z n vrcholu (aktuálně zpracovaného) k cílí. Heuristika je pro každý problém specifická.

Typickou implementaci u A* je použíti prioritní fronty (Priority Queue) pro opakovaný výběr vrcholů s nejmenší cenou spočítanou heuristikou. Fronta je známá jako *openSet*. V každém kroku algoritmu, vrchol s nejnižší $f(x)$ hodnotou je vymazán z fronty a hodnoty f a g sousedních vrcholů jsou náležitě aktualizované a následně jsou tyto sousedé přidané do fronty. Algoritmus pokračuje dokud cílový vrchol nemá nižší

[4]Kromě základní verze algoritmu existují i různé modifikace, což je nad rámec této práce. Více na `http://theory.stanford.edu/~amitp/GameProgramming/Variations.html`

f hodnotu než ostatní vrcholy ve frontě nebo pokud fronta není prázdna. Hodnota f u cíle je tedy délka nejkratší cesty a hodnota h je u zdroje nula. Druhou typický používanou množinou u implementace algoritmu je *closedSet* do které se vkládají již navštívené vrcholy (podobně jako u Dijkstrova algoritmu množina $\boldsymbol{Z}$).

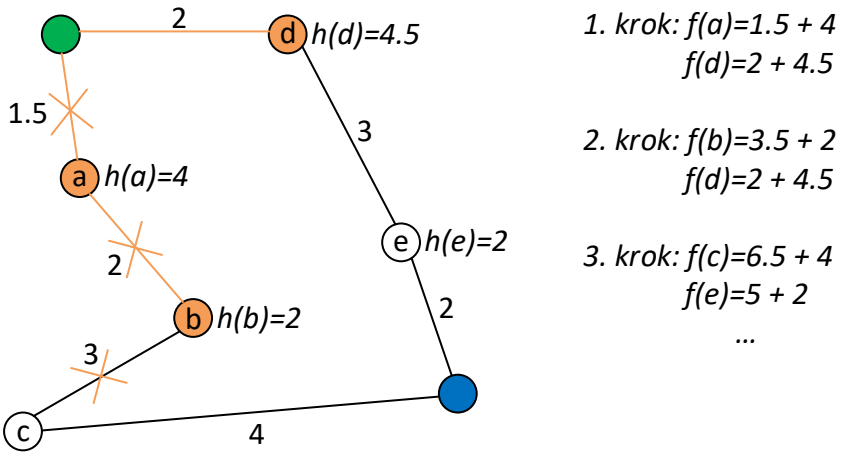
Výše popsaný postup najde pouze délku nejkratší cesty. Pro nalezení posloupnosti vrcholů je třeba cestu projít pozpátku, což umožní každému vrcholu držet přehled o předešlém vrcholu. Takže koncový vrchol ukáže na předešlého a ten zase na dalšího předešlého, dokud nějaký vrchol nebude startovní [19] [14] [15]. Ukázka pseudokódu (převzato z [6]). Z důvodů přehlednosti je v pseudokódu vypuštěna část funkce na výpočet heuristiky a zpětné sestavení (resp. rekonstrukci) cesty:

Pseudokód 4 A* algoritmus

```
function A_STAR(zdroj, cíl)
    Inicializace;
    closedSet := {}
    openSet := {zdroj}
    přišel_z := prázdna mřížka
    gScore := mřížka s počáteční hodnotou nekonečno
    gScore := 0
    fScore := mřížka s počáteční hodnotou nekonečno
    fScore[zdroj] := výpočet_ceny_heuristikou(zdroj, cíl)
    Běh jádra algoritmu;
    while openSet není prázdný do
        aktuální := vrchol v openSet mající nejnižší fScore[]
        if aktuální := cíl then
            return rekonstrukce_cesty(přišel_z, aktuální)
        openSet.Remove(aktuální)
        closedSet.Add(aktuální)
        for každý vrchol v aktuální do
            if soused v closedSet then
                pokračuj
    Funkce na výpočet ceny heuristikou;
    function VÝPOČET_CENY_HEURISTIKOU(zdroj, cíl)
    Funkce na rekonstrukci cesty;
    function REKONSTRUKCE_CESTY(přišel_z, cíl)
```

Příklad fungování algoritmu

Následující ukázka na obr. 5.10 vysvětluje funkčnost algoritmů, kde *h(x)* je heuristika, což je vzdušná vzdálenost k cíli u jednotlivých vrcholů *a, b, c, d, e* (příklad je z [16]). Zelená barva značí zdroj, modrá cíl a oranžová zpracované vrcholy.



Obr. 5.10: Jednoduchý příklad principu A*

V prvním kroku se začne ze zdroje a jeho sousedů. Do množiny *openSet* se tedy vloží vrcholy *a* a *d*, které se dále ohodnotí. Cena cesty k sousednímu vrcholu *a* je 1.5, což je více než k druhému sousednímu vrcholu *d*, kde je cena cesty 2. Takže se začne hledat cesta k cíli z vrcholu *a*. Dále jak je známo z teoretického popisu A* algoritmu, tak celková váha se počítá vzorcem $f(n)=g(n)+h(n)$. V příkladu je u vrcholu *a* celková váha 5.5 a u vrcholu *d* je celková váha 6.5 (viz 5.10). Takže platí $f(a) < f(d)$ a tak jako nejvhodnější vrchol k cíli bude vybrán vrchol *a*.

V druhém kroku se zpracují nenavštívené sousedé vrcholů *a* i zdroje, tj. opět i soused zdroje, což je vrchol *d*. Do množiny *openSet* se vloží soused zdroje vrchol *d* a soused vrcholu *a*, což je vrchol *b*. Celková cena vrcholu *b* k cíli je 5.5 a k vrcholu *d* je 6.5, tj. $f(b) < f(d)$ a jako nejlepší vchol k cíli se vybere vrchol *b*. Do množiny *closedSet* se vloží další dva vrcholy *b* a *d*.

V třetím kroku se obdobně zpracují vrcholy *c* a *e*, které se vloží do množiny *openSet*. Výsledek porovnaní celkových cen obou vrcholů je $f(c) > f(e)$. Takže jako další nejlepší vrchol k cíli bude vrchol *e*. Do množiny *closedSet* se vloží oboje již zpracované vrcholy. Nakonec takto byl zvolen úplný jiný směr cesty k cíli.

Algoritmus takto pokračuje, dokud nesestaví nejkratší cestu k cílovému vrcholu. Jako nejlepší se jevila až do třetího kroku cesta přes vrcholy *a, b*, ale od třetího

kroku je nejlepší cesta přes vrcholy *d, e*. Vyhodnocovala se vždy *f(x)* celková váha aktuálního vrcholu k cíli a ceny sousedních vrcholů se následně aktualizovaly. Vrcholy buď rozšířily kostru grafu (sestavení kostry grafu jako u Dijkstrova algoritmu) nebo se vymazaly, což u příkladu je vidět vymazáním celé levé cesty (tj. přes vrcholy *a, b*) od třetího kroku (značí oranžové kříže).

5.2.2 Push-sum protokol

Druhým vybraným stochastickým algoritmem byl Push-sum algoritmus. Push-sum algoritmus nebo také někdy nazývány Push-sum protokol je multiúčelový epidemický se šířící algoritmus, jehož funkcionalita je založena na distribuci hodnot mezi páry agentů, resp. vrcholů. Je určen pro nahodilou komunikaci v rozsáhlých sítích, kde garantuje rychlou konvergenci a přesnost. Mezi jeho přednosti patří robustnost, škálovatelnost, výpočetní a komunikační efektivita a vysoká stabilita při rušení. Mezi negativa patří, že výsledky se mohou lišit navzdory zachování konstantních vstupních dat. Důvodem je nahodilost procesu zpracování výsledků. Existuje i několik variant[5] algoritmu. V této práci je použita základní varianta algoritmu.

Na začátku v prvním kroku algoritmu je každému vrcholu přiřazen počáteční vnitřní stav roven jedné. Počáteční stav vrcholu odráží jeho váhu v sítí. V dalším kroku je vybrán náhodně jeden ze sousedů daného vrcholu při každé iteraci. Zvolenému vrcholu je zaslaná poloviční hodnota a váha odesílajícího vrcholu. Odeslané hodnoty se uloží do pamětí odesílatele, což umožní každému vrcholu vypočítat poměr těchto hodnot. Tento postup je matematický definován takto [8] [13]:

1. Nechť $\{(\widehat{S_r}, \widehat{W_r})\}$ jsou páry poslané i v t - 1,
2. nechť $s_{t,i} := \sum_r \widehat{S_r}$, $w_{t,i} := \sum_r \widehat{w_r}$,
3. rovnoměrně náhodný výběr agenta (uzlu) $f_t(i)$,
4. odeslání páru $(\frac{1}{2}s_{t,i}, \frac{1}{2}w_{t,i})$ agentu $f_t(i)$ a i,
5. $\frac{s_{t,i}}{w_{t,i}}$ je odhad průměru v t.

Push-sum protokol může být implementován do distribuovaného nebo stochastického (nahodilého) systému, aby vypočítával průměr hodnot všech entit zúčastněných v systému. Při implementaci nelze počítat s dynamickými změnami v síti během výpočtu.

[5] Více o různých variantách algoritmu na konferenčním listu `http://www.cs.cornell.edu/johannes/papers/2003/focs2003-gossip.pdf`.

5.2.3 Stochastický a heuristický výpočet

V podkapitole o algoritmu A* (podkapitola 5.2.1) byl zmíněn pojem heuristika. Se souvislosti s touto prací je dobré vysvětlit a popsat výhody a nevýhody heuristického výpočtu a stochastického výpočtu. V této podkapitole bude vysvětlen rozdíl těchto výpočtů včetně jejich kladu a záporu. Kapitola o stochastických algoritmech vysvětlila původ a princip těchto algoritmů (kapitola 2), ale v této podkapitole je třeba si znovu připomenout a vysvětlit, co vlastně je stochastický proces a heuristika a navzájem je porovnat. Jejich popis a porovnání bude v následujícím textu.

Stochastický proces a stochastický výpočet

Když je proces stochastický, tak to znamená, že je v něm určitá náhoda (resp. proces je náhodný). Je v něm neurčitost a nedá se určit přesný průběh kroku, které vedou k cíli. Říká se, že proces není deterministický, ale je právě nedeterministický. Zpravidla když se jedná o nedeterministický proces, tak v průběhu jeho vykonávaní může nastat nekonečně mnoho směru, kterými se v tomto případě stochastický proces může vydat. Naopak když je proces deterministický, tak stačí vědět počáteční stav nebo aktuální stav určitého vykonávaného procesu a z této informace lze odhadnout další krok nebo dokonce i samotný výsledek celého procesu. V stochastickém procesu další krok nebo výsledek závisí na aktuálním kroku. Výhody a nevýhody stochastických algoritmů [10]:

Výhody

- v průběhu vykonání výpočtu můžou nastat nahodile změny se kterými si algoritmus poradí
- může zaručit nalezení optimálního řešení
- jednoduchá a rychlá implementace
- jednotlivá iterace může mít konstantní čas

Nevýhody

- řešení nemusí být nalezené v optimálním čase, ale bude dostatečně přesné
- řešení může být nalezené v optimálním čase, ale nebude dostatečně přesné
- pouze některé algoritmy naleznou dostatečné přesné řešení, ale v neoptimálním čase (např. *Las Vegas* algoritmus)
- pouze některé algoritmy naleznou řešení v optimálním čase, ale toto řešení nebude dostatečně přesné (např. *Monte Carlo* algoritmus)
- rozhodují se pro další krok na základě náhodného výběru dat z množiny

Heuristika a heuristický výpočet

Heuristický výpočet znamená, že k nalezení určitého řešení problému využívá znalosti neboli informovanost. V reálném světě ekvivalentem heuristiky je využívání intuice, zkušenosti nebo logiky založené na aktuálně všeobecně nabytých znalosti a zkušenosti jako např. u lidského mozku (tzv. lidově řečeno zdravý rozum). Heuristický výpočet vybírá na základě nějaké znalosti z množiny dat nejlepší možný prvek pro další krok v řešení problému (výpočtu). Heuristický výpočet nezaručuje vždy přesné řešení ani nalezení řešení v krátkém čase. Není použitelný na všechny možné vstupní data a existují taková vstupní data, která znemožní výpočet nebo prudce zvýší čas potřebný na nalezení řešení. V praktických úlohách použití takových vstupních dat nehrozí, proto se heuristika v praxi často používá. Pokud heuristika nepovede k dobrým výsledkům, tak lze použit tzv. *metaheuristiku* do které spadají metody s *restartem* nebo „*randomizaci*“. Výhody a nevýhody heuristických algoritmů [10]:

Výhody

- výběr dat z množiny na základě nějaké „znalosti“
- nalezení řešení v relativně krátké době

Nevýhody

- nikdy negarantují, že najdou zcela přesné řešení i když v krátkém čase
- čas potřebný k nalezení řešení závisí na problému a vstupních počátečních datech
- jednotlivá iterace nemusí mít konstantní čas

Z porovnání je vidět, že podle teorie by měly byt rychlejší heuristické algoritmy, které ale nezaručují zcela optimální řešení. Heuristické algoritmy by měly rychleji konvergovat, protože z množiny možností vybírají na základě nějaké znalosti nejefektivnější data (resp. řešení nejbližší data) pro další krok. Naopak stochastické algoritmy sice konverguji pomaleji, ale vždy zaručuji optimální řešení (přesnost vs čas závisí na typu algoritmu). Dále stochastické algoritmy by neměly byt moc náchylné na nahodile změny při výpočtech. Výběr stochastického nebo heuristického přístupu výpočtu v praxi závisí na požadavku a funkci, co se nejvíce hodí pro např. danou aplikaci nebo protokol. Jak je vidět, každý přístup má své výhody a nevýhody a nelze jednoznačně říct, co je pro danou aplikaci nebo protokol lepší. Obecně je rychlejší výpočet pomocí heuristiky, který pro praxi poskytuje dostatečně přesný a rychlý výsledek.

6 POROVNÁNÍ ALGORITMŮ

V této kapitole je samotné porovnání čtyř vybraných algoritmů. Nejdříve se porovnaly mezi sebou dva distribuované algoritmy. Dále dva stochastické algoritmy a nakonec bylo provedené vzájemné porovnání všech čtyř algoritmů mezi sebou. Porovnání se provádělo na nahodilých neorientovaných grafech (resp. topologiích) s postupně rostoucí velikosti, tj. počtem vrcholů a měřil se čas až do doby, kdy daný algoritmus zkonverguje, tj. nálezné řešení. U porovnání distribuovaných algoritmů je kromě testování na nahodilých topologiích, také testování na klasických topologiích typu hvězda, strom, kruh, slabě propojena topologie a silně propojena topologie. Později se od tohoto postupu odstoupilo a přešlo se pouze na nahodilé topologie.

Na konci každé podkapitoly jednotlivých typů (distribuovaných a stochastických algoritmů) algoritmů jsou výsledky v podobě grafu, kde jsou zobrazené rychlosti konvergence jednotlivých algoritmů.

6.1 Porovnání dvou distribuovaných algoritmů

Na úplném začátku bylo zvolené porovnání dvou distribuovaných algoritmů mezi sebou. Jak je známo z teoretického popisu distribuovaných algoritmů v kapitole 5.1, tak Bellmanův-Fordův algoritmus je aplikován ve směrovacím protokolu RIP a používá distanční vektor, tj. počet skoků k cílí (hops), což jsou další vrcholy na cestě k cílovému vrcholu. Dijkstrův algoritmus je aplikován u směrovacího protokolu OSPF a oproti Bellmanu-Fordovi zná celou topologií sítě. OSPF je tzv. *link-state* protokol, což znamená, že každý uzel ví o celkové struktuře sítě (zná kostru grafu). Oba algoritmy se běžně používají v síťové praxi, a proto bylo zajímavé porovnat je mezi sebou, a tak zjistit, který je efektivnější, co se tyče konvergence a rychlosti v různých grafech (resp. topologiích) s různou velikosti a nahodilým uspořádáním vrcholů. Konkrétně algoritmy se zvolily proto, aby se zjistilo, jestli je efektivnější algoritmus, který jako kritérium nejlepší cesty bere počet skoku k cíli nebo algoritmus, který zná celou topologií sítě včetně vah linek a na základě teto znalosti vybírá nejvhodnější cestu k cíli.

Dále bylo potřeba ujasnit si, jak porovnávat konvergenci u těchto algoritmů. Konvergence se zde dá chápat tak, že je to stav, kdy všechny vrcholy ví o všech vrcholech a je možná komunikace mezi dvěma libovolnými vrcholy v sítí. V našem případě sítě budou představovat neorientované grafy. Neorientované nejvíce odpovídají realitě, protože v praxi datový tok v drtivé většině případu prochází po lince v obou směrech, tj. od odesilatele k příjemci a naopak (tzv. plně duplexní přenos - full-duplex).

Takže konvergence je zde stav, kdy je možná komunikace mezi dvěma libovolnými vrcholy v grafu. Dalším problémem bylo určit podle kterého parametru se algoritmy mají porovnávat. Nakonec po uvažování a možné realizaci dané možnosti v prostředí MATLAB, byla zvolena rychlost zpracovaní algoritmu, než dojde do výše popsaného stavu konvergence. K tomuto účelu posloužily funkce přímo dostupné v prostředí MATLAB, které se jmenují *tic* a *toc*[1]. První funkce je začátek (*tic*), druhá konec (*toc*) měření času, tj. je to obdoba stopek. Před měřením času bylo potřeba vybrané algoritmy implementovat v prostředí MATLAB a dále po implementaci se měřil čas běhu samotných algoritmů na různých topologiích. Jinak řečeno se měřila doba až po kterou algoritmus na daném grafu neboli topologii zkonvergoval.

6.1.1 Vytvořené topologie

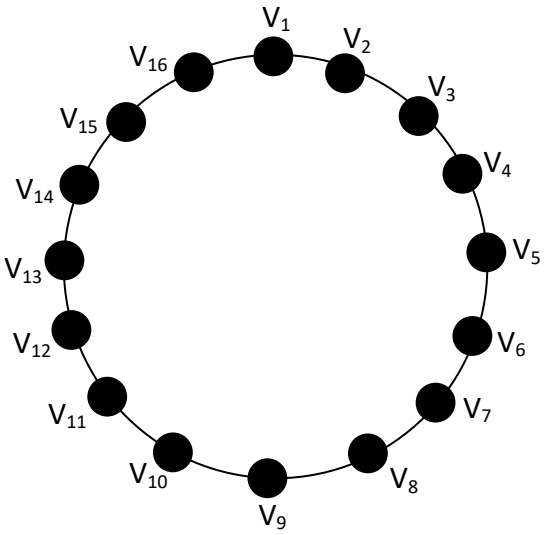
Prvním krokem realizace testování bylo vytvoření topologií (resp. grafů) na kterých by se dané algoritmy testovaly. Pomocí matic sousednosti byly sestavené topologie (resp. grafy) o 16 vrcholech. Jedná se o následující topologie: kruh (Ring) - obr. 6.1, hvězda (Star) - obr. 6.2, strom (Tree) - obr. 6.3, slabě propojená topologie (Weak Connected Topology) - obr. 6.4, silně propojena topologie (Strong Connected Topology) neboli také někdy nazýváno smíšenou (Mesh) je každý propojený s každým (úplný graf) - obr. 6.5. U silně propojené topologie je z důvodu přehlednosti uveden přímo obrázek z prostředí MATLAB.

Oba algoritmy popsané v kapitole 5.1 počítají s maticemi vah i když každý jiným způsobem. Takže dalším problémem, který se musel vyřešit, byl převod matice sousednosti na maticí vah. Převod se provádí nahrazením prvků s hodnotou 1 prvkem s hodnotou nerovnající se 1, přitom matice musí být symetrická podle diagonály a v horním a spodním trojúhelníku matice se nesmí opakovat stejné číslo, neboť by mohlo dojit k zacyklení algoritmu. Pro lepší pochopení a vysvětlení je zde ukázkový příklad: Uvažujme jednoduchou topologii v podobě neorientovaného váženého grafu, která je na obrázku 6.6 s určitými vahami hran. Matice sousednosti a následně matice vah tohoto neorientovaného váženého grafu bude:

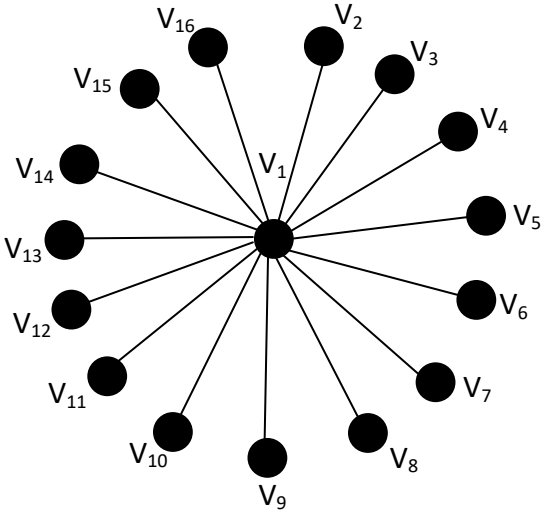
$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \rightarrow \text{Matice vah} = \begin{pmatrix} 0 & 2 & 0 & 5 \\ 2 & 0 & 3 & 0 \\ 0 & 3 & 0 & 4 \\ 5 & 0 & 4 & 0 \end{pmatrix}$$

První řešení by mohlo být takové, že by se ručně přepsaly všechny matice sousednosti na matice cen, ale pro topologie, kde je velký počet vrcholů, je to zdlou-

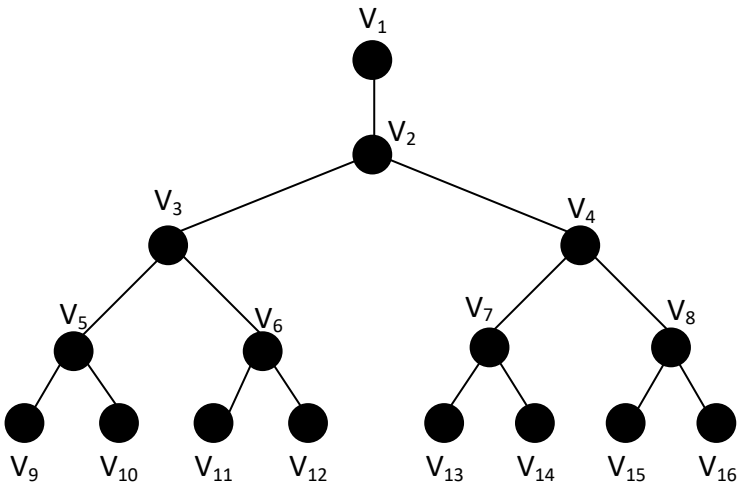
[1]Více informaci o těchto funkcích na: `https://www.mathworks.com/help/matlab/ref/tic.html`



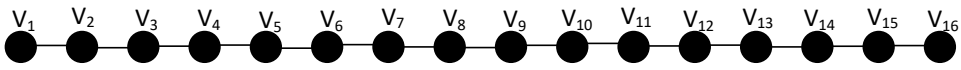
Obr. 6.1: Příklad topologie kruh



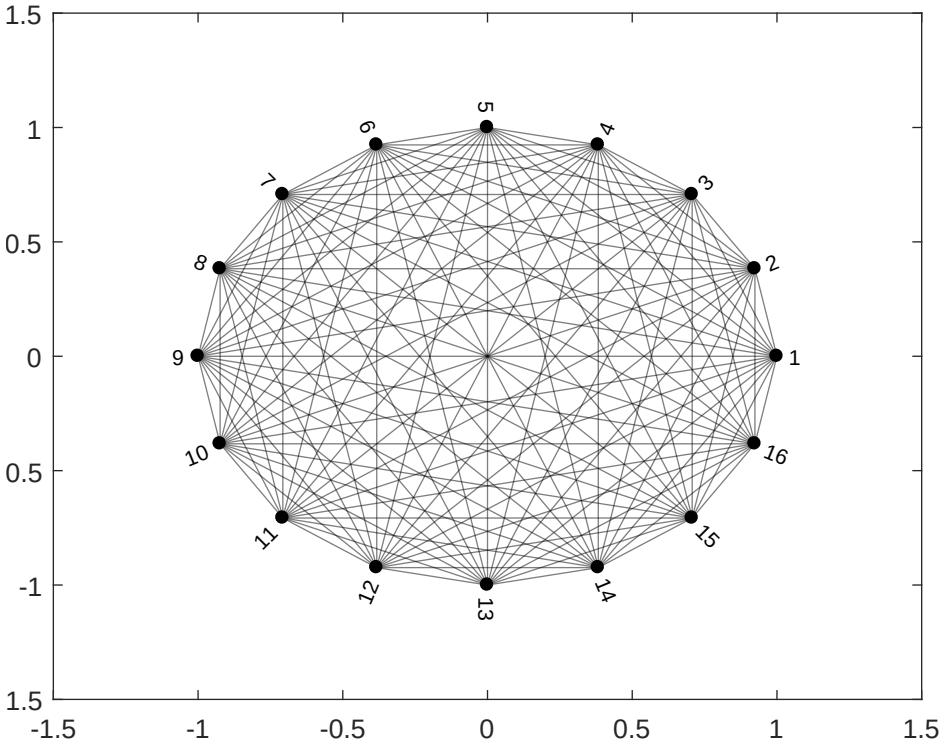
Obr. 6.2: Příklad topologie hvězda



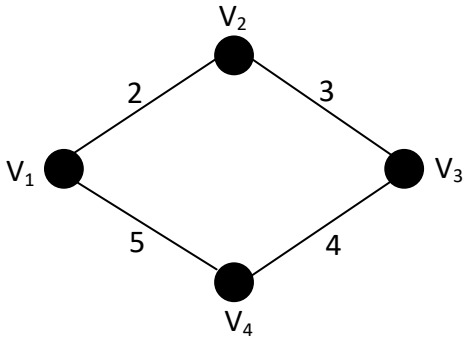
Obr. 6.3: Příklad topologie strom



Obr. 6.4: Příklad slabě propojené topologie



Obr. 6.5: Příklad silně propojené topologie



Obr. 6.6: Příklad jednoduché topologie s váhou hran

havé, tudíž neefektivní a dá se snadno udělat chyba. Efektivnější je pomoci funkci v MATLAB prostředí nahradit všechny jedničky na náhodné číslo v intervalu od nuly až po jedničku (může být i jiný). Dále aby matice byla symetrická podle diagonály, musí se zkopírovat horní trojúhelník matice do dolního nebo naopak. Ukázka kódu z MATLAB prostředí pro silně propojenou topologii je následující:

```
% Matice sousednosti silne propojene topologie:
A = ones(16) - diag([1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]);

% Nahodila cisla intervalu 0-1 matice hran:
B = rand(size(A));
A(logical(A)) = B(logical(A));

% Symetrizace dolniho s hornim trojuhlenikem matice:
C = triu(A) + triu(A,1)';
```

6.1.2 Výsledky porovnání obou algoritmů na klasických topologiích

Po vytvoření topologií následovala samotná implementace algoritmů s jejich experimentálním testováním. Implementace algoritmů byla provedena v programovém prostředí MATLAB o kterém už bylo párkrát v prací zmíněno. Na každé topologií popsané v 6.1.1 byl jednotlivý algoritmus spouštěn pětkrát kvůli věrohodnosti výsledků, neboť jeden testovací průchod nemůže poskytnout zcela směrodatná data pro hodnoceni algoritmu. Dále oběma algoritmy se hledala vždy cesta z vrcholu 1 do vrcholu 16. Čas z funkci *tic* a *toc* byl vložen do proměnné a následně uložen do datového pole, ze kterého se naměřená data exportovala do programu Microsoft Office Excel pro další zpracování.

Jako první byl testován Bellmanův-Fordův algoritmus. Výsledky měření jsou v tabulce 6.1, kde je uveden počet měření a doba (v milisekundách) dosažení konvergence u jednotlivých topologií. Z důvodu přehlednosti, jsou hodnoty do této tabulky zaokrouhlené na tři desetinná místa - u původního výsledku v excel souboru desetinných míst je více a čas je v sekundách.

Při každém spouštění byla generovaná pro každou topologii náhodná matice vah, ale vždy se hledala cesta od vrcholu 1 k vrcholu 16. Údaje měření Bellman-Fordová algoritmu z tabulky 6.1 byly zpracované do obrázku 6.7. Na obrázku je vidět, že nejrychleji algoritmus konverguje u silně propojené topologie, což je vidět u druhého, třetího, čtvrtého a pátého spouštění. Naopak nejhůře konverguje u kruhové topologie. Výjimkou je poslední spouštění, kde nejpomalejší je stromová topologie.

Počet spouštění [-]	**Kruh [ms]**	**Hvězda [ms]**	**Strom [ms]**	**Slabě propojená [ms]**	**Silně propojena [ms]**
1	0,239	0,115	0,132	0,135	0,088
2	0,245	0,133	0,196	0,189	0,079
3	0,236	0,116	0,098	0,134	0,080
4	0,169	0,139	0,133	0,160	0,137
5	0,286	0,099	0,105	0,182	0,104

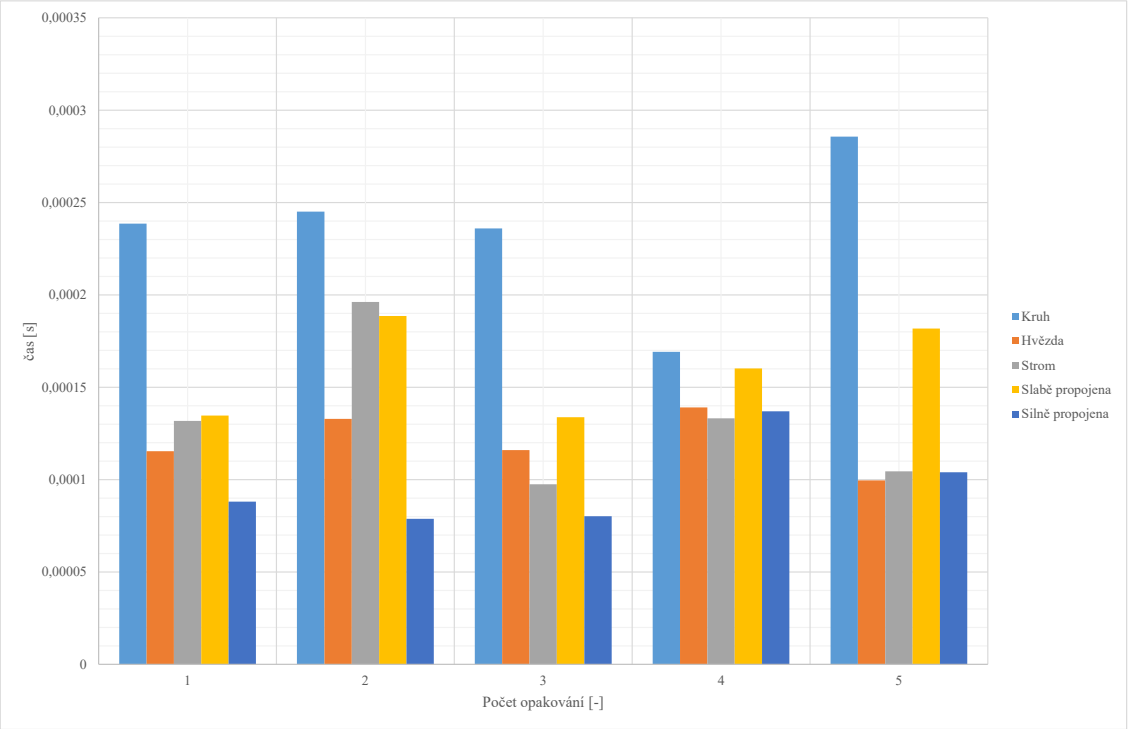
Tab. 6.1: Bellmanův-Fordův algoritmus na odlišných typech topologií

Důvodem proč algoritmus rychleji konverguje na silně propojené topologií je, že zde existuje více propojení s ostatními vrcholy, což naopak u kruhu takto není a následná komunikace z vrcholu 1 do vrcholu 16 musí u kruhu projít přes ostatní vrcholy na cestě k vrcholu 16.

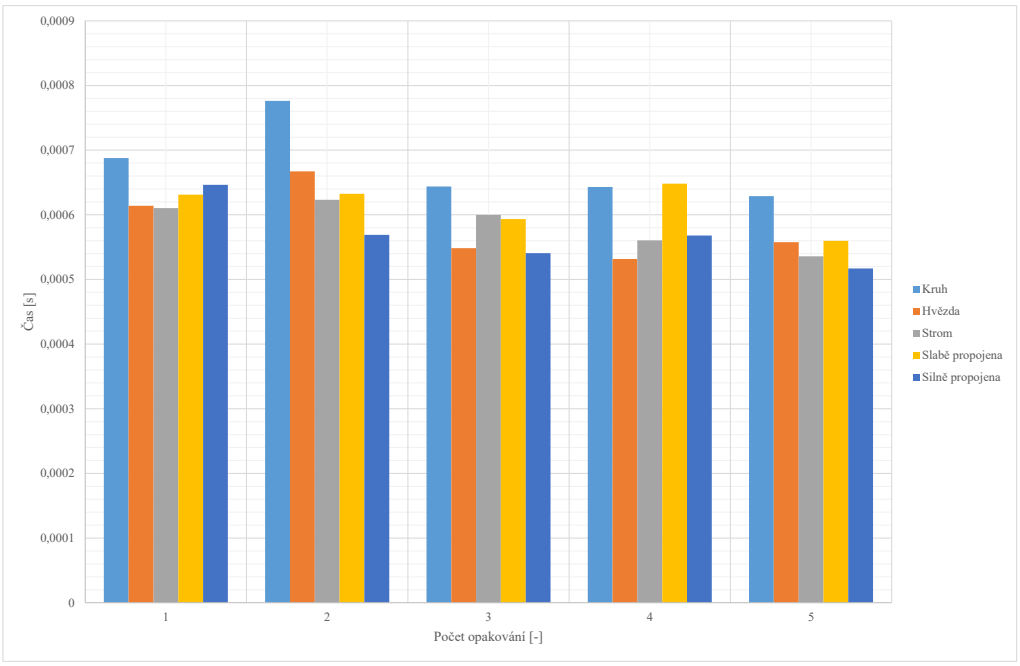
Druhým testovaným algoritmem byl Dijkstrův algoritmus, viz tabulka 6.2. Po zpracování i této tabulky do obrázku 6.8, je vidět, že Dijkstrův algoritmus nejrychleji dosáhne konvergence na silně propojené topologií, a to u všech pěti spouštění. Nejhůře je to u kruhu. Opět vysvětlení je, že u silně propojené topologie existuje více hran mezi vrcholy, a tak se rychleji sestaví kostra grafu. U kruhu existuje miň hran mezi vrcholy a vrcholy jsou v stanoveném pořadí s pouze dvěma hranami u každého (vstupní a výstupní), které vedou na stanovené sousedy, a tak se musí projít všechny vrcholy v kruhu na cestě z vrcholu 1 do vrcholu 16, což způsobí pomalejší sestavení kostry grafu a tím i pomalejší konvergenci.

Počet spouštění [-]	**Kruh [ms]**	**Hvězda [ms]**	**Strom [ms]**	**Slabě propojena [ms]**	**Silně propojena [ms]**
1	0,688	0,614	0,610	0,631	0,646
2	0,776	0,667	0,623	0,632	0,569
3	0,644	0,548	0,599	0,593	0,541
4	0,643	0,532	0,561	0,648	0,568
5	0,629	0,558	0,536	0,559	0,517

Tab. 6.2: Dijkstrův algoritmus na odlišných typech topologií

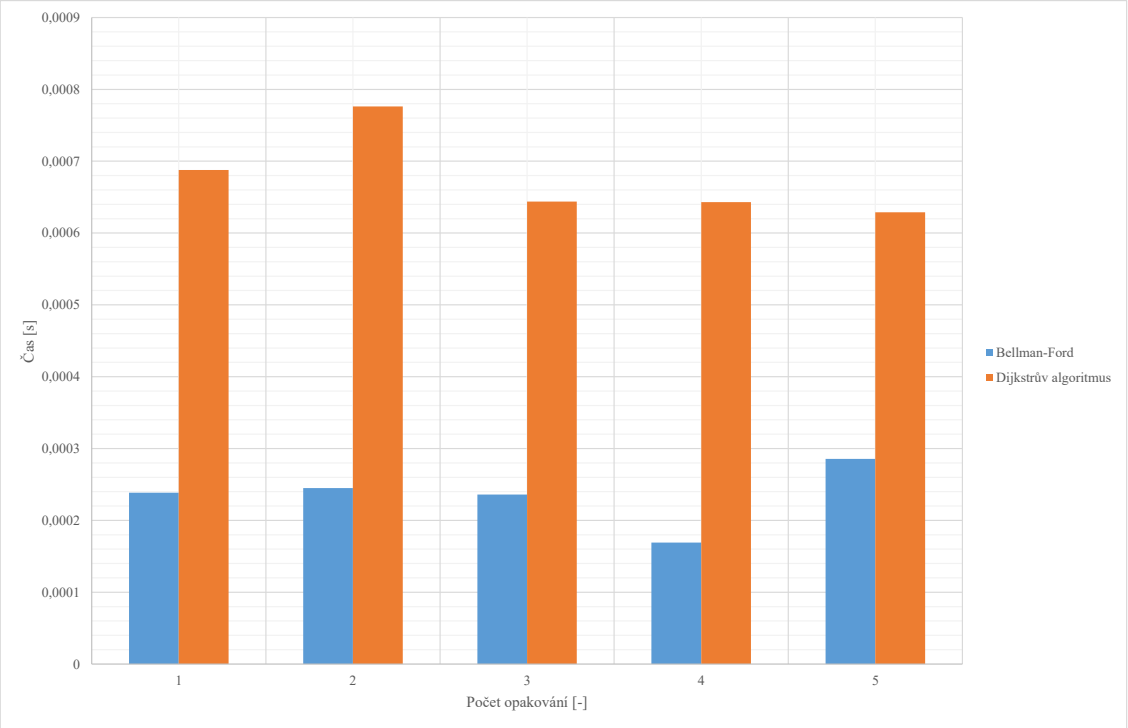


Obr. 6.7: Konvergence Bellman-Ford algoritmu na odlišných topologiích



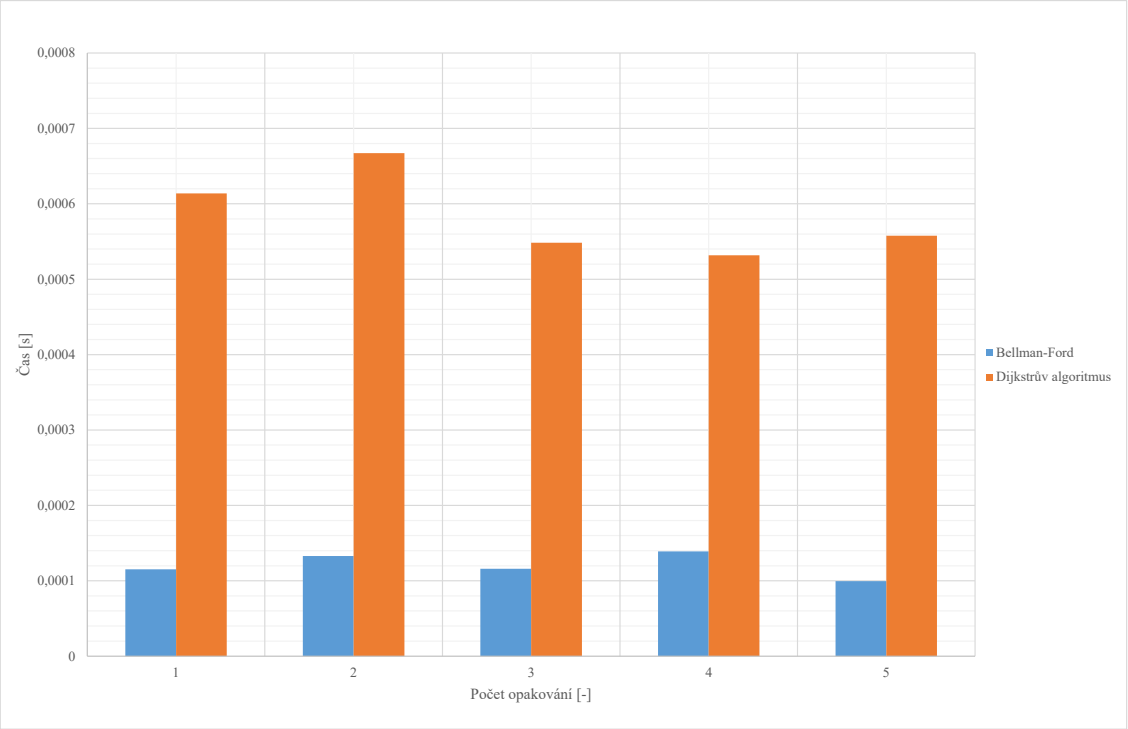
Obr. 6.8: Konvergence Dijkstrová algoritmu na odlišných topologiích

Pro lepší porovnání a přehlednost jsou dále oba algoritmy porovnané zvlášť na každé topologii. Na obr. 6.9 je topologie kruh, na obr. 6.10 je topologie hvězda, na obr. 6.11 je topologie strom, na obr. 6.12 je slabě propojená topologie a na obr. 6.13 je silně propojená topologie.

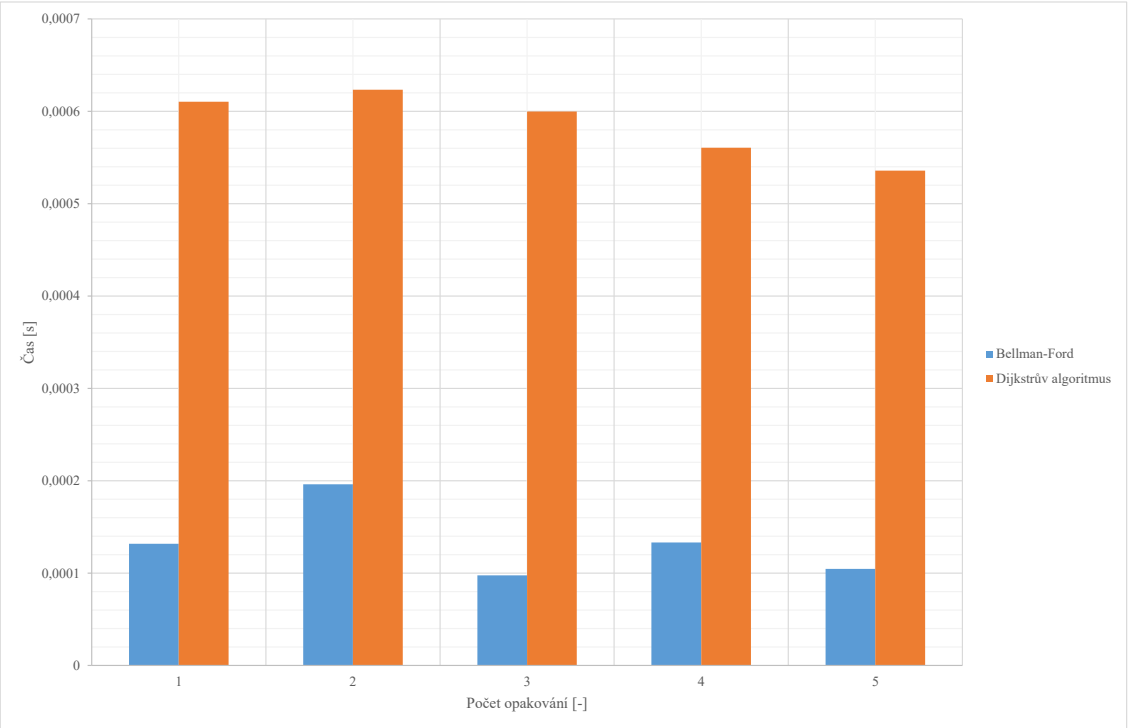


Obr. 6.9: Porovnání obou distribuovaných algoritmů na topologii kruh

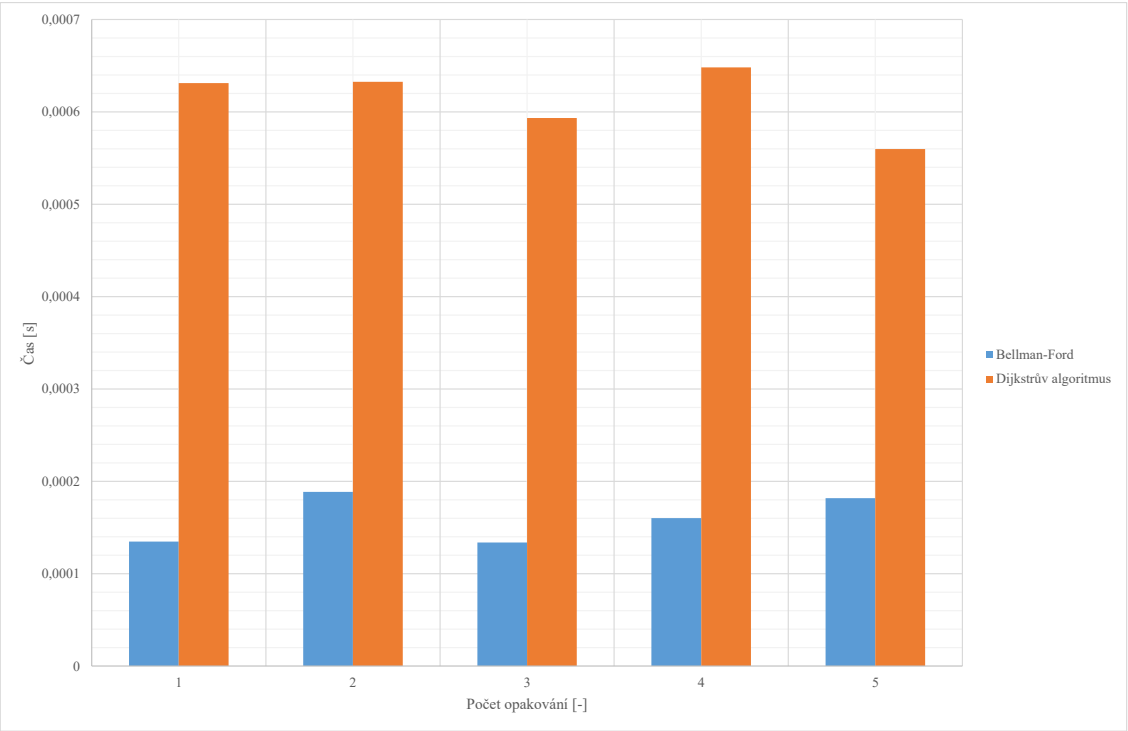
Při porovnání obou algoritmů na každé topologii zvlášť je vidět, že rychlejším algoritmem je Bellmanův-Fordův a to u všech topologií. Dijkstrův je vždy pomalejší. Při tomto porovnání se pořad jednalo o klasické topologie, které mají předem definovanou strukturu, ale lepé by bylo algoritmy otestovat na větších a nahodilejších topologiích. Tato myšlenka vedla k vytvoření topologii v další kapitole.



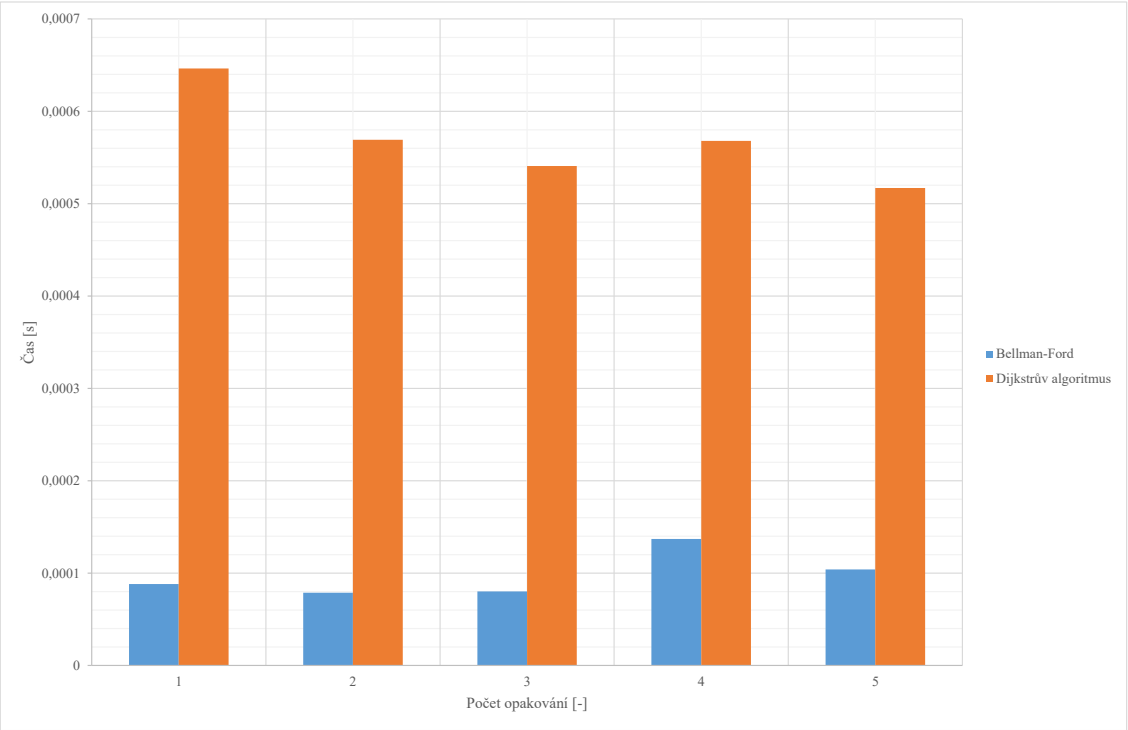
Obr. 6.10: Porovnání obou distribuovaných algoritmů na topologii hvězda



Obr. 6.11: Porovnání obou distribuovaných algoritmu na topologii strom



Obr. 6.12: Porovnání obou distribuovaných algoritmů na slabě propojené topologii



Obr. 6.13: Porovnání obou distribuovaných algoritmů na silně propojené topologii

6.1.3 Výsledky porovnávaní obou algoritmů na nahodilých topologiích

Podkapitola 6.1.2 popisovala výsledky konvergence na klasických topologiích s počtem vrcholů 16, ale později se objevila zvědavost sledovat konvergenci na více reálnějších topologiích s větším počtem vrcholů než 16 a s nahodilou topologií (nestrukturovanou). Pro tento účel byla sestavená funkce generování náhodné matice vah (vážený neorientovaný graf), která musí být symetrická podle diagonály, v diagonále musí být nuly, a nakonec matice musí mít nějaké procento prvků s hodnotou nula ve zbytku matice. Z teorie grafů a se souvislosti se síťovou problematikou neexistují úplně propojené sítě ani sítě s velkým počtem funkčních vrcholů (uzlů) a hran (linek). Vznikl by z toho tzv. úplný graf, kde každý vrchol je propojený s ostatními vrcholy. V reálných sítích toto není možné zajistit, a proto bylo zvoleno, že může byt propojeno maximálně pouze 30% vrcholů, tj. hodnot v matici s jedničkou mimo diagonálu. Matice představuje nahodile topologie, kde její velikost definuje počet vrcholů v topologii. Ukázka sestavení takové funkce v prostředí MATLAB:

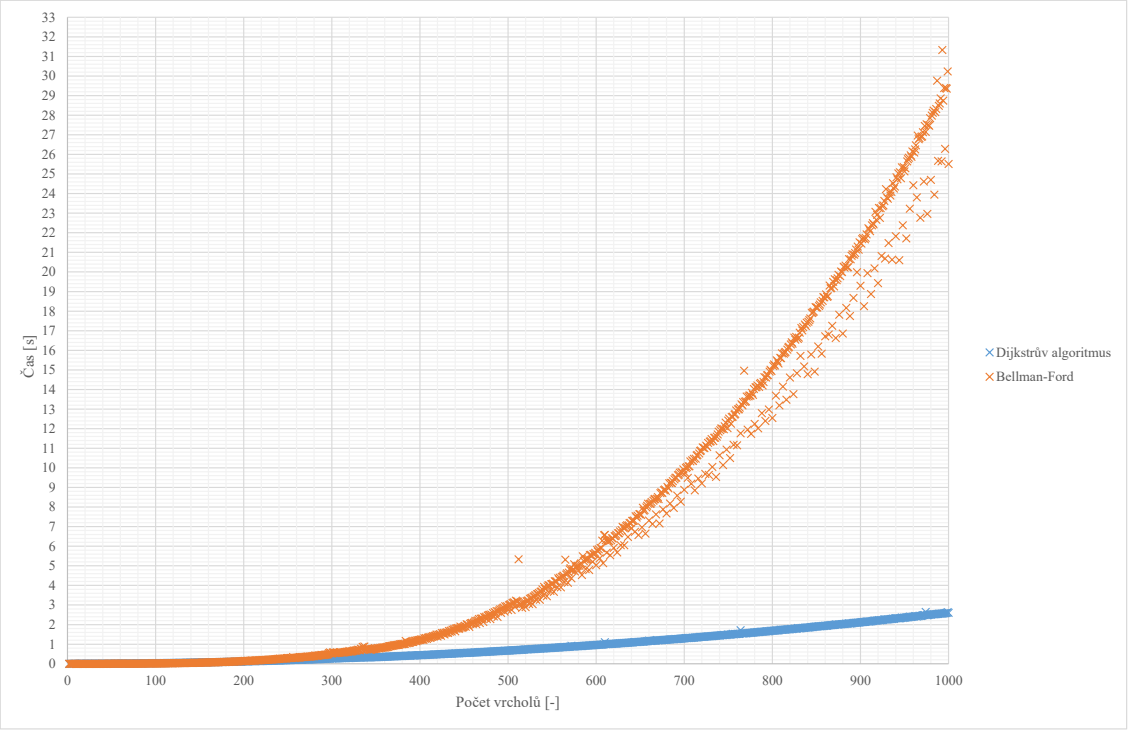
```
1  function maticeVah = maticeVah(velikostMatice)
2
3  % Generovani cisel v rozsahu 0-1:
4  maticeVah = rand(velikostMatice);
5  for y = 1:velikostMatice
6      for x = y:velikostMatice
7
8  % Nulova diagonala:
9          if (x==y)
10             maticeCen(x,y) = 0;
11         else
12
13 % 70% hodnot bude 0:
14             if maticeCen(x,y)≤0.70
15                maticeCen(x,y) = 0;
16             end
17         end
18     end
19 end
```

Každý algoritmus z důvodů dostatku dat byl spouštěn třikrát a velikost matice vah byla generovaná od velikosti 2×2 až po velikost 1000×1000. Výsledky tohoto testu představuje obr. 6.14. V každé iteraci spouštění algoritmu a generovaní velikosti topologie se hledala cesta z vrcholu 1 k vrcholu n, kde n je velikost matice cen.

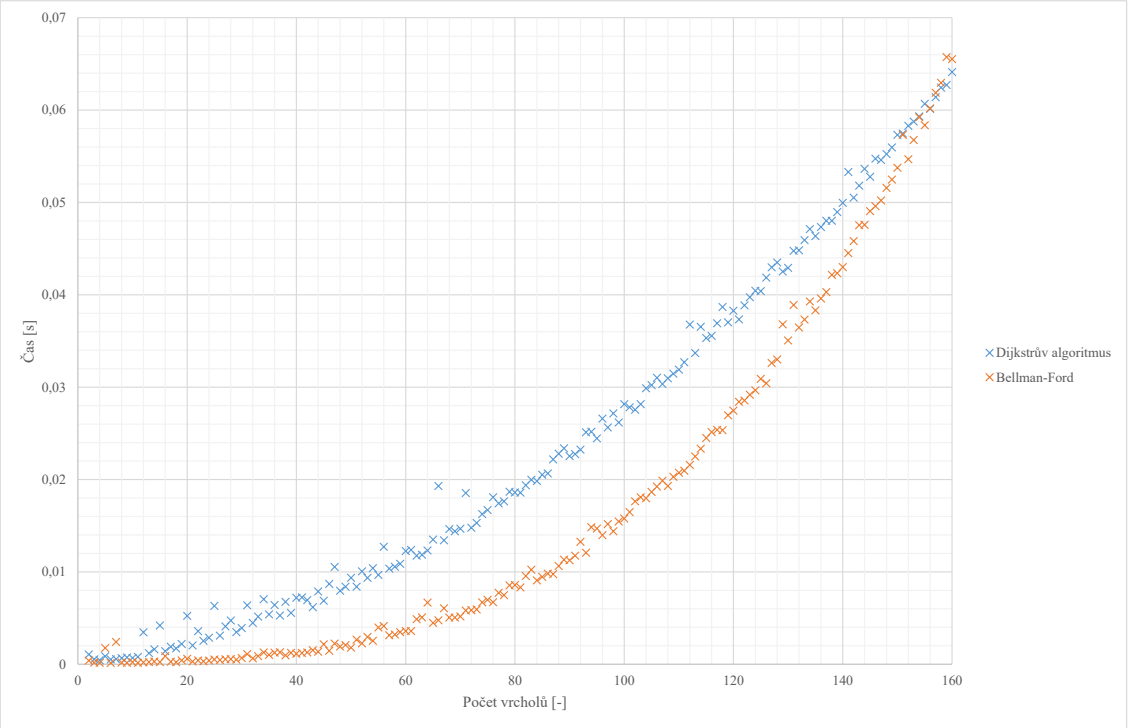
Z důvodu podobnosti obrázků všech tři spouštění je zde uveden jen jeden. Princip vkládaní času do proměnné a exportování do programu Microsoft Office Excel je stejný jako v kapitole 6.1.2.

Tabulka naměřených časů běhů algoritmů až po stav, kdy všechny vrcholy ví o všech ostatních vrcholech, pro svou velikost není uvedená, ale při pozornějším prohlédnutí byl pozorován zajímavý efekt: Bellmanův-Fordův algoritmus měl do určité velikosti topologie (resp. matice nebo grafu) rychlejší konvergenci. Později začal po určitém počtu vrcholů v rychlosti zaostávat až čas potřebný pro konvergenci začal růst exponenciálně s počtem vrcholů. Pro tento účel byl proveden ještě jeden experimentální test s matici o velikosti 160×160 se stejným počtem spouštění pro oba distribuované algoritmy jako v minulém experimentu. Výsledek tohoto experimentu je na obr. 6.15.

Zajímavé je také vidět, že Dijkstrův algoritmus i když počítá s tisícem vrcholů, tak čas konvergence se drží okolo jedné sekundy, což je vidět na obr. 6.14. Dále čas konvergence oproti Bellman-Fordovu algoritmu (kde čas od určitého počtu vrcholů roste exponenciálně) roste téměř lineárně s počtem vrcholů.



Obr. 6.14: Výsledky spouštění obou algoritmů na nahodilých topologiích s počtem vrcholů do 1000



Obr. 6.15: Výsledky spouštění obou distribuovaných algoritmů na nahodilých topologiích s počtem vrcholů do 160

Ze všech obrázků je patrné, že rychlejší algoritmus na klasických topologiích (podkapitola 6.1.2) je Bellmanův-Fordův. Stejný výsledek, že je rychlejší Bellmanův-Fordův algoritmus vyšel i na nahodilých topologiích. Je rychlejší do velikosti topologie asi 155 vrcholů (obr. 6.15). Po tomto počtu nastal zvrat a dále už čas potřebný na konvergenci roste exponenciálně s počtem vrcholů, a tak při větším počtů vrcholů je efektivnější Dijkstrův algoritmus, který i při počtu 1000 vrcholů drží čas potřebný na konvergenci okolo 1 sekundy (obr. 6.14).

Hranice, kde ještě dominuje Bellmanův-Fordův algoritmus muže byt ovlivněna procesorem, tj. na lepším procesoru může být hranice místo 155 vrcholů níže a na horším procesoru může být hranice výše než 155 vrcholů, ale obecně platí, že efektivnější do určité velikosti topologie je Bellmanův-Fordův algoritmus. Tato skutečnost se testovala na dalším počítači, který měl čtyřjádrový procesor Intel Core i5 s taktem jednotlivého jádra 3,5 GHz. Zde byla hranice dominance Bellman-Fordová algoritmu okolo 150 vrcholů, což není velký rozdíl oproti 155. Původním testovacím počítačem, byl počítač s dvoujádrovým procesorem Intel Pentium B960, který měl takt jednotlivého jádra 2,20 GHz.

6.2 Porovnání dvou stochastických algoritmů

V této podkapitole budou porovnané dva stochastické algoritmy, které byly popsané v podkapitole 5.2. Jedna se o A* a Push-sum algoritmy. Testování probíhalo obdobně jako v předchozí podkapitole (porovnání dvou distribuovaných algoritmů), tj. měření času běhu algoritmu až zkonverguje pomoci funkcí *tic* a *toc*. Zde měření probíhalo už rovnou na nahodilých topologiích, tj. měření na topologiích typu hvězda, strom, kruh, silně propojené a slabě propojené topologií jako v podkapitole 6.1.2 už v tomto testování není.

Důvody, proč byl zvolen tento přístup jsou dva. Prvním je, že v praxi u reálných sítí, topologie typu hvězda, strom apod. se málo používají. Druhým důvodem je, že aplikace algoritmu A* na obyčejné topologie, kde se používají matice sousednosti neexistuje, neboť z matice sousednosti (a následně matice cen) nejde získat heuristiku. Pro heuristiku je potřeba mít souřadnice vrcholů v sítí. Proto byla zvolená tzv. mřížková struktura (grid).

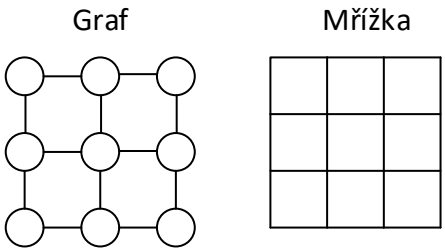
6.2.1 Implementace A*

Jak již bylo uvedeno A* používá mřížkovou strukturu. Na mřížkovou strukturu se dá dívat jako na speciální případ grafu. Rozdíl mezi grafem a mřížkovou strukturou je patrný na obr. 6.16. V podstatě se jedná o to, že jednotlivý vrchol se převede na čtverec se stranami, kde jednotlivá strana určuje hranu, kterou měl původní vrchol. Strana má určitou cenu při pohybu z daného čtverce ven do dalších čtverců. Vrcholy na obrázku 6.16 mají při převodu na mřížkovou strukturu čtyři strany pohybu - vlevo, dolu, vpravo, nahoru. Každý pohyb z vrcholu (který je nyní převeden na čtverec) má určitou cenu stejně jako hrana u klasického grafu. Kde nebyla hrana, tak není umožněn pohyb v tomto směru ze čtverce. Kromě čtyř základních pohybů existují i čtyři diagonální pohyby, což umožní dohromady osm možností pohybu, ale pro účely této práce stačí základní čtyři pohyby. V tomto případě heuristika[2] se počítá jiným způsobem než u možnosti se čtyřmi pohyby.

Mřížková struktura má výhodu, že se z ní dá snadno spočítat heuristika v podobě přímé vzdušné vzdálenosti k cílí. Každý vrchol je uspořádán do určité pozice v mřížce a má tak určitou pozici v kartézských souřadnicích. V této práci se vzdušná vzdálenost od aktuálního vrcholů k cílovému vrcholů počítá pomocí vzorce euklidovské vzdálenosti. Výpočet vypadá takto [18]:

$$\text{vzdálenost} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{6.1}$$

[2]Více o výpočtu heuristiky na `http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html`



Obr. 6.16: Ukázka rozdílu mezi grafem a mřížkou

kde x_1, y_1 jsou kartézské souřadnice aktuálně zpracovaného vrcholu a x_2, y_2 jsou kartézské souřadnice cílového vrcholu.

Algoritmus A* by se mohl uplatnit i ve WSN (Wireless Sencor Network - bezdrátové senzorové sítě) sítích. Místo vzdálenosti by mohly vrcholy v senzorové sítí mít například nějakou funkci nebo identifikačního číslo na základě kterých by se vybírala nejlepší cesta pro komunikaci.

Dále byl algoritmus při implementaci úpraven[3] tak, aby hledal cestu od vrcholu 1 se souřadnicemi (1,1) k vrcholu n se souřadnicemi (n, n), kde n představuje velikost mřížkové mapy. Konkrétně algoritmus pracoval na topologiích (resp. mřížkových grafech) s velikosti od 2, tj. topologie o velikosti 2×2 až po velikost 1000, tj. topologie 1000×1000 a hledala se cesta z vrcholu 1 k poslednímu vrcholu v běhu dané velikosti topologie (resp. mřížkového grafu). Proto souřadnice koncového vrcholu jsou (n, n). Ukázka této části v MATLAB kódu (zkráceno):

```
1  for n = 2:1000
2      MAX_X=n;
3      MAX_Y=n;
4      % Nastaveni cile a startovni pozice
5      xval=1;
6      yval=1;
7      xStart=xval; % Startovni pozice x
8      yStart=yval; % Startovni pozice y
9      xTarget=MAX_X; % Pozice cile x
10     yTarget=MAX_Y; % Pozice cile y
11     ... % Dalsi kod ...
12 end
```

[3]Samotné jádro algoritmu A* pro MATLAB prostředí bylo převzato z [12] a následně bylo upravené pro potřeby této práce.

Zbytek implementace je obdobný jako v kapitole porovnání dvou distribuovaných algoritmů. I zde se uplatnily nahodile váhy hran (resp. váhy pohybů ze čtverce), ale navíc nejlepší možná cesta se vybírala na základě heuristiky, která je popsaná výše.

Konvergence se měřila tak, že na začátek kódu samotného algoritmu se vložila funkce *tic* a na konec algoritmu funkce *toc*, kde funkce *toc* byla dále uložena do proměnné *cas*. Následně se časy každého zkonvergovaného řešení, tj. nalezení cesty od zdrojového vrcholu k cílovému vrcholu při určitých velikostech topologie uložily do pole *vysledek*. Výsledky se exportovali z prostředí MATLAB do Microsoft Office Excel programu. Ukázka této části v kódu MATLAB prostředí (zkráceno):

```
for n = 2:1000
    tic; % Zacanek mereni casu

    ... % Zde je telo samotnoho algoritmu + dalsi kod

    cas = toc; % Vlozeni casu do promenne
end
n % Velikost topologie
omega % Promenna iterace spousteni
vysledek(n,3*omega-1) = n; % Vypis vysledku
vysledek(n,3*omega) = cas; % Vypis vysledku
```

6.2.2 Implementace Push-sum algoritmu

Algoritmus Push-Sum se zařadil k měření kvůli své vlastnosti komunikační efektivity a vysoké stabilitě při rušení. Stálé se však jedná o experimentální algoritmus, který ještě nebyl praktický nasazen do žádného standardu ani technologie, proto bylo zajímavé zjistit jak rychlé zkonverguje na nahodilých topologiích.

Implementace algoritmu proběhla podle [5] a [8] v prostředí MATLAB. Měření času konvergence je stejné jako u ostatních algoritmů (jako u Bellman-Forda, Dijkstry a A*), tj. na začátek samotného algoritmu byl vložen začátek měření času a na konci algoritmu byl vložen konec měření času běhu algoritmu. Funkce algoritmu je dále popsaná.

V každé iteraci si každý vrchol vybere náhodně jednoho ze svých sousedů. Dále tomuto sousedu odešle poloviční hodnotu svého vnitřního stavu a poloviční hodnotu váhy. Odeslaná informace je uložena v pamětí vrcholu, který tuto informaci odeslal. Všechny vrcholy následně vypočítají odhad průměru, který se počítá poměrem vnitřního stavu a váhy [8]. Tento postup se opakuje, dokud systém nedosáhne součinnosti, kterou se rozumí rozdíl mezi maximální a minimální hodnotou v rámci celé sítě (resp. topologie nebo grafu). Podle konzultace se studentem Tomášem Eclérem,

který už má zkušenosti s algoritmem Push-sum, byla zvolená optimální hodnota pro tento rozdíl, která má byt menší 0,00015 [5]. Správnost se ověřuje součtem všech vah a hodnot vnitřních stavu během celého procesu. Takže konvergenci se zde rozumí stav, kdy váhy a vnitřní stavy se rozešlou všem vrcholům v topologií a topologie tak dosáhne součinnosti, tj. rozdíl mezí maximální a minimální hodnotou bude menší 0,00015. Z výše popsaného plyne, že Push-sum se používá pro odhad agregačních funkcí. Ukázka kódu implementace algoritmu v MATLAB prostředí (převzato z [5]):

```
k = 1;
rozdil = 0.00015;
for ... % Cast kodu pro urceni velikosti matic + dalsi kod
% Zacatek algoritmu a zacatek mereni casu:
    tic; % Zacatek mereni casu
    while abs(max(vyber2) - min(vyber2)) > rozdil
        zprava = zeros(size(B,1),size(B,1));
        vaha = zeros(size(B,1),size(B,1));
            for i = 1:1:size(B,1)
                if sum(B(i,:))≠0
                rovnomerna_nahoda = datasample(find(B(i,:)==1),1);
                    zprava(rovnomerna_nahoda,i) = x(i,k)/2;
                    vaha(rovnomerna_nahoda,i) = w(i,k)/2;
                end
            end
            for i=1:1:size(B,1)
                zprava(i,i) = x(i,k)/2;
                vaha(i,i) = w(i,k)/2;
                x(i,k+1) = sum(zprava(i,:));
                w(i,k+1) = sum(vaha(i,:));
            end
        vyber = [vyber zeros(1,size(B,1))'];
        for i = 1:1:size(B,1)
            for j = 1:1:size(B,1)
                vyber(i,k+1) = sum(zprava(i,:))/sum(vaha(i,:));
            end
        end
        k = k + 1;
    vyber2 = vyber(:,end);
    cas = toc; % Konec mereni casu
end % Konec algoritmu a mereni casu
```

Pro měření času konvergence bylo zapotřebí vytvořit nahodile topologie, které tentokrát byly opět reprezentované maticemi sousednosti. Velikost matic byla počítaná od $n = 2$ až po $n = 1000$. Dále bylo zvoleno jako v předešlých kapitolách (více podkapitola 6.1.3), že matice bude mít alespoň 70% nulových prvků. Pokud

by byly obsazené všechny pozice matice (prvky), tak by vznikl úplný graf. Jelikož nikdy neexistuje kompletně propojena síť bez výpadků linek nebo uzlů (resp. hran a vrcholů), tak se souvislosti sítí a teorie grafu je realistické zvolit, že pouze 30% prvků matic (resp. vrcholů grafu, uzlů sítě) je propojených mezi sebou. Dále diagonála musí být nulová, protože vrchol nemůže byt propojený hranou sám se sebou (o tom více subkapitola 4.2). Nakonec matice musí být symetrická, protože se jedna o neorientované grafy, kde se komunikuje obousměrně (analogie reálných sítí s plně duplexním přenosem). Ukázka této části kódu v MATLAB prostředí (zkráceno):

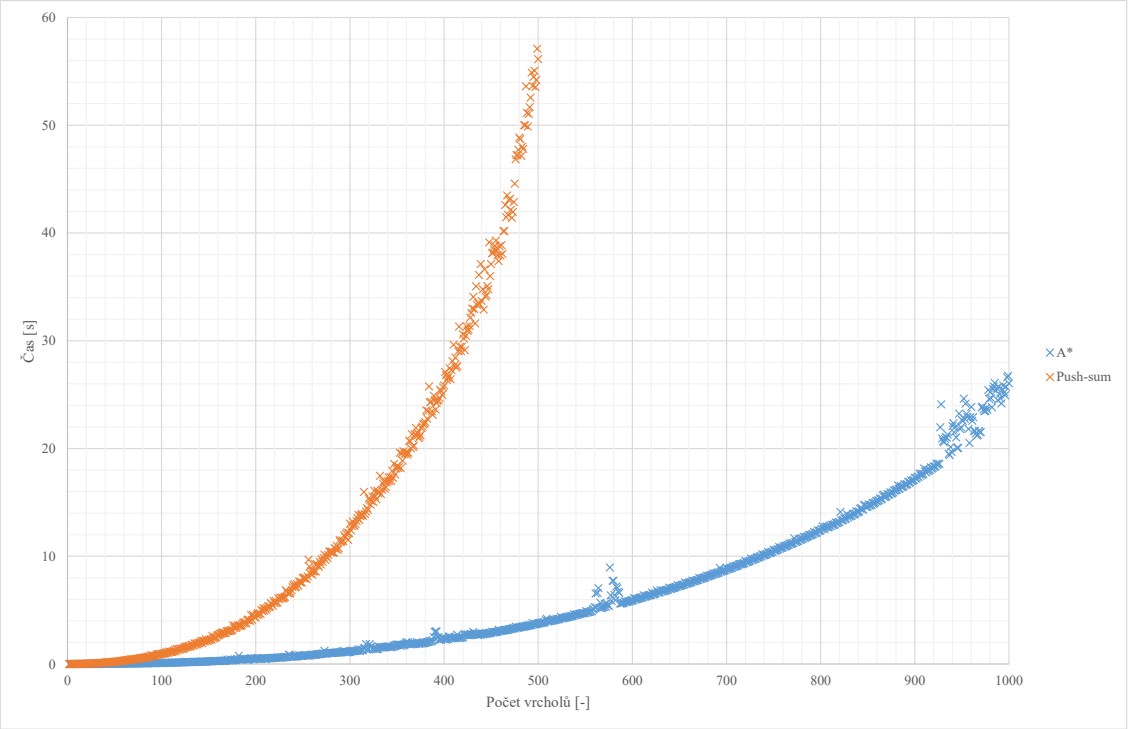
```
1 for n = 2:1000
2 a = n; b = n;
3 A = rand(a,b) ≤ 0.7; % 70% hodnot matice bude 0
4 % Symetrizace spodniho trojuhelniku podle horniho:
5 B = triu(A)+triu(A,1)';
6 B(1:n+1:n*n) = 0; % Nastaveni diagonaly na 0
7 ... % Dalsi kod ...
```

6.2.3 Výsledky porovnání

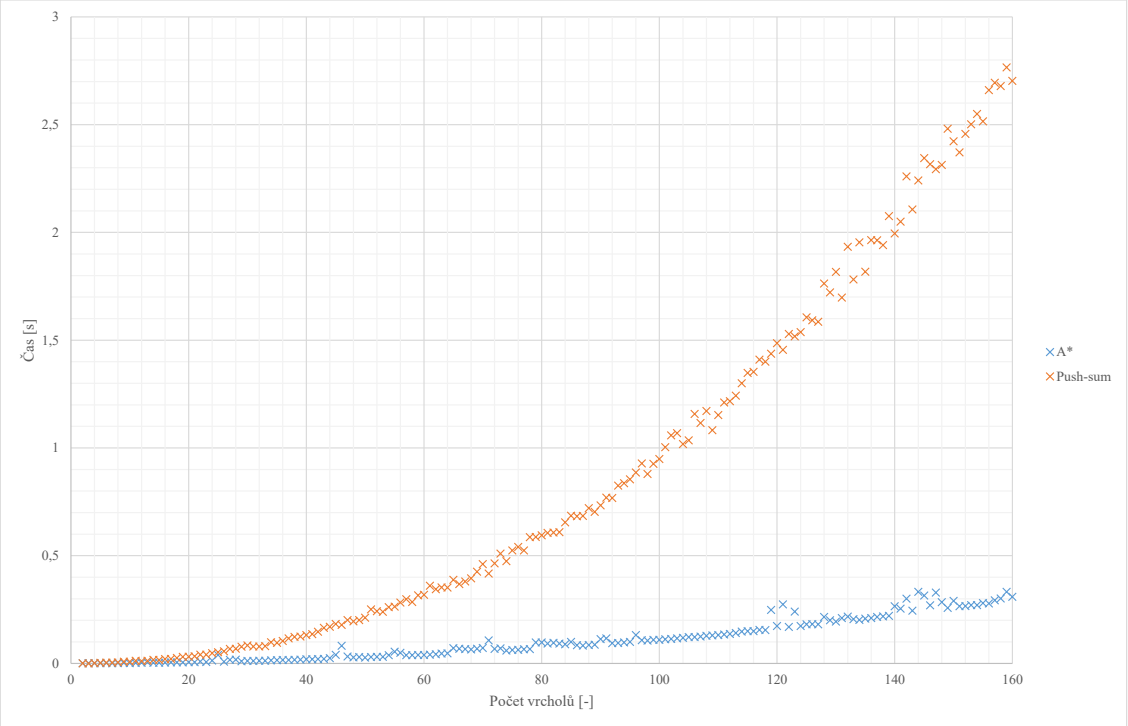
Po osvědčeném měření času konvergence na nahodilých topologiích v podkapitole porovnání distribuovaných algoritmů i zde se nejdříve měřilo na velikosti topologií do 1000 vrcholů a pak do 160 vrcholů. Každá velikost topologie se měřila třikrát. Obrázky ze všech tři měření si jsou podobné, a tak je v práci zobrazen pouze jeden obrázek z důvodů přehlednosti a úspory místa. Tabulky zde nejsou uvedené kvůli velkému počtu naměřených hodnot.

Na obrázku 6.17 je vidět, že jednoznačně rychlejší konvergenci má algoritmus A*. Důvodem může být, že A* neprochází všechny vrcholy v topologií oproti tomu push-sum sděluje svoje hodnoty ostatním vrcholům, dokud tuto informaci neví všechny vrcholy v topologií a není tak dosaženo součinnosti v síti (více o činnosti algoritmu v subkapitole 6.2.2). Konkrétněji A* vždy vybere nejvhodnější vrchol k cíli a z tohoto vrcholu pokračuje dál, zatímco Push-sum rozesílá informace všem vrcholům dokud se nedosáhne součinnosti, což vyžaduje všechny vrcholy topologie.

Na obrázku 6.18 je porovnání obou algoritmů do 160 vrcholů. I zde výrazně zaostává algoritmus Push-sum. Přibližně stejnou rychlost mají do počtu vrcholů okolo 20.



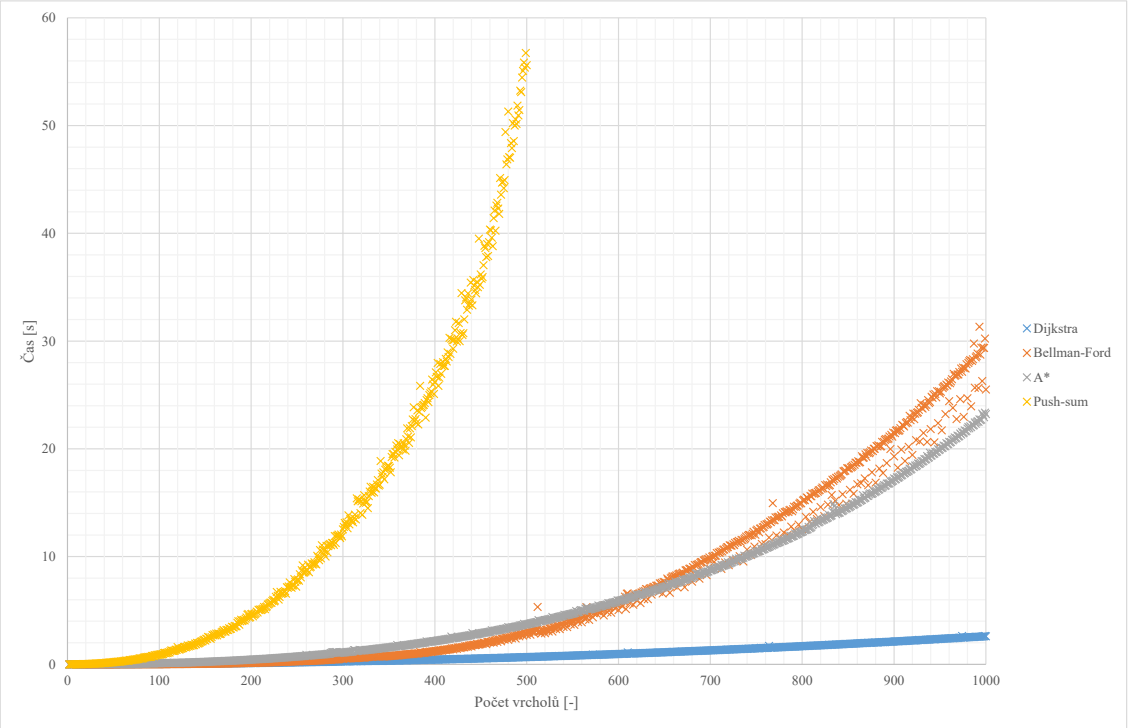
Obr. 6.17: Porovnaní obou algoritmů s maximálním počtem vrcholů do 1000



Obr. 6.18: Porovnaní obou algoritmů s maximálním počtem vrcholů do 160

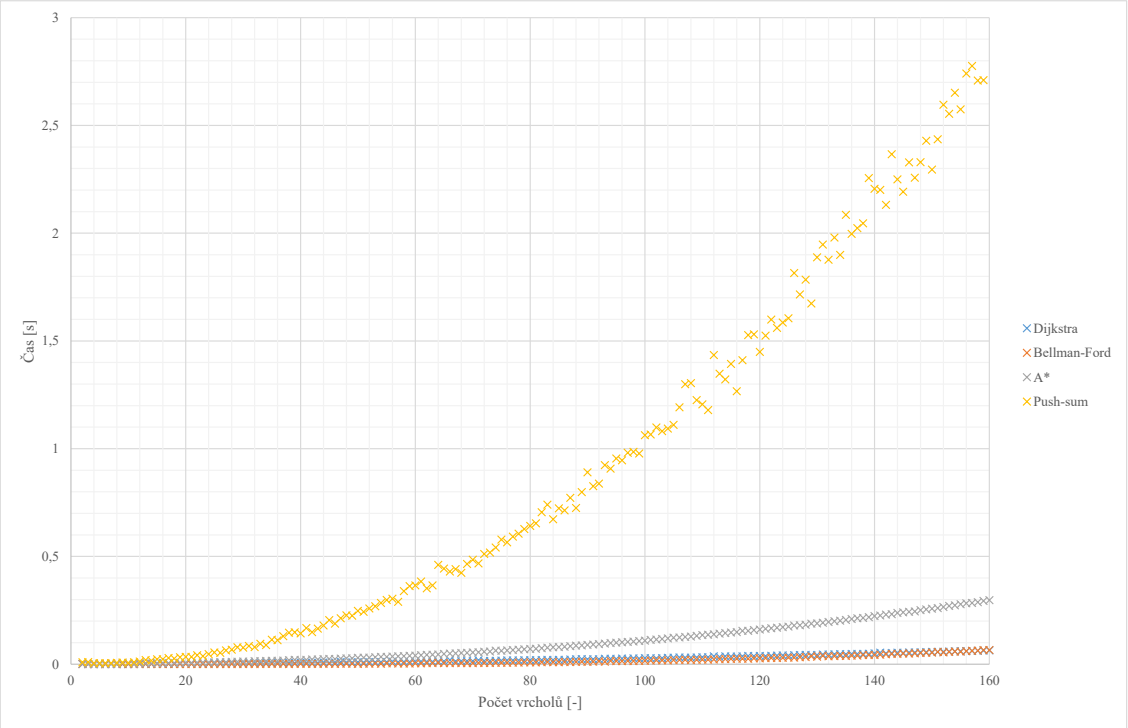
6.3 Porovnání všech algoritmů

Poslední částí práce byla věnovaná porovnaní všech algoritmů mezi sebou, což by mělo ukázat, který algoritmus je nejefektivnější. Na obrázku 6.19 je vidět, že nejrychlejší algoritmus je Dijsktrův a Bellmanův-Fordův, ale u subkapitoly porovnání dvou distribuovaných algoritmů (přesněji 6.1.3) bylo zjištěno, že do určité velikosti topologie je rychlejší Bellmanův-Fordův. Dobré výsledky má také algoritmus A*, který ale zaostává za dvěma výše popsanými distribuovanými algoritmy. Horší výsledky u A* můžou být způsobené výpočtem heuristiky u každého vrcholu, kdy se nejdříve zpracují všechny do aktuálního okamžiku známé údaje (váha a heuristika) a teprve na základě výpočtů z těchto informací se algoritmus rozhodné přes který vrchol pokračovat. Nejhůře dopadl algoritmus Push-sum, kdy u počtu vrcholů 500 už je čas na konvergenci okolo 57 sekund. Důvodem špatných výsledků Push-sum algoritmu může být to, že než zkonverguje, tak musí rozeslat svoje informace všem vrcholům v dané topologií a neděla to tak efektivně jako Dijkstrův algoritmus, který si sestavuje pro každý vrchol kostru grafu.



Obr. 6.19: Porovnaní všech algoritmů s maximálním počtem vrcholů do 1000

Pro upřesnění výsledků rychlosti konvergence je tu porovnání všech algoritmů na velikosti topologií s maximálním počtem vrcholů do 160, viz 6.20. I zde jsou nejlepší



Obr. 6.20: Porovnaní všech algoritmů s maximálním počtem vrcholů do 160

výsledky u Dijktrova a Bellmanova-Fordava algoritmu a za nimi je A*. Nejhorší výsledky jsou u Push-sum algoritmu.

Na závěr je nutno sdělit, že takto porovnávané rychlosti můžou být ovlivněné vhodně napsaným kódem, který může být efektivnější než v této práci a dále rychlosti mohou ještě být ovlivněné pokročilejšími počítači s lepším hardwarem a hlavně procesory než na kterých běžely experimenty z této práce. I když v podkapitole 6.1.3 tato problematika byla nastíněna a testování na jiném počítači ukázalo, že rozdíly experimentu na dvou počítačích nemusí byt tak obrovské, přesto by tato problematika stala za prozkoumání v nějaké budoucí práci. Testování algoritmů na odlišných zařízeních a optimalizace kódů je už nad rámec cílů této práce.

7 ZÁVĚR

Diplomová práce se zabývá problematikou distribuovaných a stochastických algoritmů a jejich využitím v sítích. Přesněji porovnáním rychlosti konvergence různých algoritmů používaných v sítích na různých topologiích, které reprezentují různé sítě. Práce se skládá ze dvou hlavních části. Teoretické části, kde je stručně vysvětlen potřebný teoretický základ pro práci s distribuovanými a stochastickými algoritmy a praktické části, kde jsou výsledky porovnání algoritmů.

V teoretické části jsou nejdříve v prvních dvou kapitolách stručně vysvětlené distribuované a stochastické algoritmy včetně jejich dělení a problémů které řeší. V další kapitole je představen matematický nástroj se kterým se běžně pracuje u distribuovaných a stochastických algoritmů a usnadňuje tak práci s algoritmy v pozdějších kapitolách. Poslední kapitolou je kapitola věnovaná výběru algoritmů. V této kapitole jsou popsané čtyři vybrané algoritmy, které se zkoumaly v této diplomové práci. Čtyři algoritmy se dále dělí na dva distribuované a dva stochastické a těmto dvojicím jsou věnované podkapitoly, kde jsou následně jednotlivé algoritmy popsané a vysvětlené včetně jejich pseudokódů. Zvolené distribuované algoritmy jsou Dijkstrův algoritmus a Bellman-Fordův algoritmus a zvolené stochastické algoritmy jsou A* algoritmus a Push-sum algoritmus.

Praktická část se věnuje popisu implementace algoritmů v prostředí MATLAB, popisu návrhu topologií na kterých se algoritmy následně testovaly a zobrazením výsledků těchto testů. V první kapitole praktické části byly porovnané dva distribuované algoritmy na dvou typech topologií. Nejdříve na klasických, což jsou topologie typu hvězda, kruh, strom, silně propojena topologie a slabě propojená topologie a později na nahodilých topologiích v podobě matic sousednosti a matic vah od velikosti 2×2 až po 1000×1000, tj. od počtu vrcholů 2 až po 1000. Zde se zjistilo, že na všech klasických topologiích a na nahodilých topologiích do velikosti 155 vrcholů je rychlejší, tj. rychleji konverguje Bellman-Fordův algoritmus, který se v praxi používá u směrovacího protokolu RIP. Od velikosti větší než 155 vrcholů u nahodilých topologií je už výrazně rychlejší Dijkstrův algoritmus, který se používá u protokolu OSPF.

V druhé kapitole praktické části jsou porovnávané dva stochastické algoritmy. Zde už nejsou algoritmy testované na klasických topologiích, ale na nahodilých se stejným počtem uzlů jako v kapitole porovnaní dvou distribuovaných algoritmů, tj. od velikosti 2×2 až po 1000×1000. Výsledky porovnaní v této kapitole ukázaly, že rychleji konverguje algoritmus A* a Push-sum už u topologie o velikosti 500 (matice sousednosti a vah má velikost 500×500) vrcholů má čas potřebný na konvergenci až skoro 60 sekund.

V poslední kapitole praktické části se všechny výsledky proložily do jednoho grafu

a z toho se zjistilo, že nejrychleji konvergují distribuované algoritmy, které se běžně používají v praxi, tj. Bellmanův-Fordův a Dijkstrův. Hned za nimi je stochastický algoritmus A*. Nejhorší výsledky má Push-sum algoritmus, kterému čas potřebný na konvergenci porostl exponenciálně s počtem vrcholů v topologií až na hodnotu kolem 60 sekund. Jednoznačně nejrychlejším algoritmem pro velké sítě je Dijkstrův a pro male sítě Bellmanův-Fordův. Algoritmus A* po mírně úpravě pro sítě by se dál použit také.

LITERATURA

[1] www.algoritmy.net. *Bellmanův-Fordův algoritmus v grafu - princip,* [online], [cit. 1. 12. 2016]. Dostupné z URL: <http://www.programming-algorithms.net/article/47389/Bellman-Ford-algorithm>.

[2] BURGET, Radim. *Teoretická informatika,* Brno: VUT, 2013. Fakulta Elektrotechniky a komunikačních technologií.

[3] Dynamic Programming (Bellman-Ford Algorithm). *Bellman-Ford Algorithm,* [online], [cit. 1. 12. 2016]. Dostupné z URL: <http://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/>.

[4] Dynamic Programming (Dikstra's shortest patch). *Dijkstra's algorithm,* [online], [cit. 1. 12. 2016]. Dostupné z URL: <http://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/>.

[5] ECLER, T. *Přítomnost mrtvého uzlu v distribuovaném systému*, [online]. [cit. 1. 7. 2018]. 2016. Dostupné z URL: <https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=129532>.

[6] www.growingwiththeweb.com. *A* pathfinding algorithm - basics and other stuff,* [online], [cit. 1. 5. 2017]. Dostupné z URL: <http://www.growingwiththeweb.com/2012/06/a-pathfinding-algorithm.html>.

[7] gitta.info. *Dijkstra,* [online], [cit. 1. 12. 2016]. Dostupné z URL: <http://www.gitta.info/Accessibiliti/en/html/Dijkstra_learningObject1.html>.

[8] KEMPE, David - DOBRA, Alin - GEHRKE, Johannes. *Gossip-Based Computation of Aggregation Information.* Department of Computer Science, Cornell University, Ithaca, 10 s. NY 14853, USA.

[9] KRUPKOVÁ, V., Fuchs, P., *Matematika 1.* Fakulta elektrotechniky a komunikačních technologií, VUT v Brně.

[10] KUSHNER, Harold J. a George YIN *Stochastic approximation and recursive algorithms and applications,* 2nd edition. New Yourk: Springer, c2003, 495 s. ISBN: 978-1-84882-980-0.

[11] LYNCH, Nancy A. *Distributed Algorithms,* 2nd edition. San Francisco, California: Morgan Kaufman, 1997, 904 s. ISBN: 978-1-55860-348-6.

[12] mathworks.com *A* (A Star) search for path planning tutorial* [cit. 1. 5. 2017]. Dostupné z URL: <http://1url.cz/6tj3o>.

[13] NOVOTNÝ, Bohumil - KENYERES, Martin - PEYKOV, Damyan. *Komparace statistické kredibility reprezentanta průměrné rychlosti konvergence protokolu push-sum.* Fakulta elektrotechniky a komunikačních technologií, VUT v Brně. In: elektrorevue ISSN 1213 - 1539, 2016, svazek 18, 5 s.

[14] redblobgames.com. *Grids and Graphs,* [online], [cit. 1. 5. 2017]. Dostupné z URL: <http://www.redblobgames.com/pathfinding/grids/graphs.html>.

[15] redblobgames.com. *Grids and Graphs - preparetion,* [online], [cit. 1. 5. 2017]. Dostupné z URL: <http://www.redblobgames.com/pathfinding/a-star/implementation.html>.

[16] SlideShare.net. *A* algorithm - AO* search - example - lecture 21,* [online], [cit. 1. 5. 2017]. Dostupné z URL: <https://www.slideshare.net/hemak15/lecture-21-problem-reduction-search-ao-star-search>.

[17] ŠKORPIL, Vladislav. *Přístupové a transportní sítě,* Brno: VUT, 2012. Fakulta Elektrotechniky a komunikačních

[18] theory.stanford.edu *Heuristics - From Amit's Thoughts on Pathfinding,* [online], [cit. 1. 5. 2017]. Dostupné z URL: <http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>.

[19] theory.stanford.edu. *Introduction to A* and others,* [online], [cit. 1. 5. 2017]. Dostupné z URL: <http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html>.

[20] *Teorie grafů - Matice sousednosti,* [online], [cit. 1. 7. 2018]. Dostupné z URL: <http://teorie-grafu.cz/zakladni-pojmy/reprezentace-grafu.php>.

[21] TVRDÍK, Josef. *Stochastické algoritmy pro globální optimalizaci,* první vydání. Ostrava: Ostravská univerzita, 2010, 79 s.

[22] Umělá inteligence I. *Stochastické algoritmy,* [online], [cit. 11. 11. 2016]. Dostupné z URL: <https://akela.mendelu.cz/~xpopelka/cs/ui/ucici/>.

[23] Základní grafové algoritmy - jednoduše a srozumitelně. *Bellmanův-Fordův algoritmus,* [online], [cit. 1. 12. 2016]. Dostupné z URL: <http://algoritmy.eu/zga/nejkratsi-cesta/bellman-forduv-algoritmus/>.

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

A	Matice sousednosti
A*	A star algorithm
ATM	Asynchronous Transfer Mode
CPU	Central Processing Unit
DNS	Domain Name System
HW	Hardware
I	Jednotková matice
IPC	Interprocess Communication Method
IS-IS	Intermediate System to Intermediate System
ISO/OSI	International Organization for Standardization/Open Systems Interconnection Model
LAN	Local Area Network
MAN	Metropolitan Area Network
NLSP	NetWare Link Services Protocol
OSPF	Open Shortest Path First
RIP	Routing Information Protocol
SDH	Synchronous Digital Hierarchy
TCP/IP	Transmission Control Protocol/Internet Protocol
WAN	Wide Area Network

SEZNAM PŘÍLOH

A OBSAH PŘILOŽENÉHO DVD

Přiložené DVD obsahuje elektronickou verzi práce ve formátu „PDF“, soubory s výsledky práce ve formátu „EXCEL“ a soubory pro práci v prostředí MATLAB. Soubory pro práci v prostředí MATLAB (tzv. „M-Files“) byly vytvořené ve verzi MATLAB 2015a. Kompletní obsah přiloženého média je uveden níže:

- Soubor DP_práce (Diplomová práce ve formátu „.PDF“)
- Složka souborů „A_star“ (Zdrojové kódy „.M-FILE“ algoritmu A*)
- Složka souborů „Bellman_Ford“ (Zdrojové kódy „.M-FILE“ Bellman-Fordová algoritmu)
- Složka souborů „Bellman_Ford_vs_Dijkstra“ (Zdrojové kódy „.M-FILE“ porovnání algoritmů Bellman-Forda a Dijkstry)
- Složka souborů „Dijkstra“ (Zdrojové kódy „.M-FILE“ Dijkstrová algoritmu)
- Složka souborů „Push_sum“ (Zdrojové kódy „.M-FILE“ algoritmu Pus-sum)
- Složka souborů „Topologie“ (Zdrojové kódy „.M-FILE“ topologií hvězdy, silně propojené topologie, kruhu, stromu a slabě propojené topologie)
- Složka souborů „Vysledky“ (Výsledky práce jednotlivých porovnání ve formátu „.EXCEL“)