VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2017 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
AUTHOR
Bc. Oleksandr Yarmolskyy

VEDOUCÍ PRÁCE
SUPERVISOR
Ing. Bohumil Novotný

BRNO 2017

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:*** 2016/17

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: 1.2.2017 ***Termín odevzdání:*** 24.5.2017

Vedoucí práce: Ing. Bohumil Novotný
Konzultant:

doc. 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 porovnaní podle rychlosti konvergence na různých topologiích sítí v prostředí Matlab.

KLÍČOVÁ SLOVA

algoritmus, distribuované algoritmy, stochastické algoritmy, konvergence, síť, MATLAB, simulace, 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

algorithm, distributed and stochastic algorithms, convergence, network, MATLAB, simulation, 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, 2016. 73 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 Ing. Bohumilovi Novotnému 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)

EVROPSKÁ UNIE
EVROPSKÝ FOND PRO REGIONÁLNÍ ROZVOJ
INVESTICE DO VAŠÍ BUDOUCNOSTI

OP Výzkum a vývoj
pro inovace

OBSAH

Úvod **12**

1 Distribuované algoritmy **13**
1.1 Problémy distribuovaných algoritmů 14
1.2 Standardní problémy 15

2 Stochastické algoritmy **17**
2.1 Problém globální optimalizace 17
2.2 Aproximační algoritmy 20
2.3 Evoluční algoritmy 21
2.4 Genetické algoritmy 22

3 Matematický nástroj **24**
3.1 Teorie grafů 24
3.2 Lineární algebra 27

4 Výběr algoritmů **30**
4.1 Výběr dvou distribuovaných algoritmů 30
4.1.1 Bellmanův-Fordův algoritmus 30
4.1.2 Dijkstrův algoritmus 34
4.2 Výběr dvou stochastických algoritmů 38
4.2.1 A* algoritmus 39
4.2.2 Push-sum algoritmus 43

5 Praktická část **44**
5.1 Porovnání dvou distribuovaných algoritmů 44
5.1.1 Vytvořené topologie 45
5.1.2 Výsledky porovnání obou algoritmů na klasických topologiích . 49
5.1.3 Výsledky porovnávaní obou algoritmů na nahodilých topologiích 55
5.2 Porovnání dvou stochastický algoritmů 58
5.2.1 Implementace A* 58
5.2.2 Implementace push-sum algoritmu 61
5.2.3 Výsledky porovnání 63
5.3 Porovnání všech algoritmů 64

6 Závěr **67**

Literatura **69**

Seznam symbolů, veličin a zkratek **71**

Seznam příloh **72**

A Obsah přiloženého DVD **73**

SEZNAM OBRÁZKŮ

2.1 Funkce s globálním minimem 18
3.1 Ukázka zobrazení sítě v podobě grafu 24
3.2 Příklad neorientovaného grafu 25
3.3 Speciální příklad neorientovaného grafu, kde každý komunikuje s každým 26
3.4 Příklad orientovaného grafu 26
4.1 Příklad počátečního stavu algoritmu 32
4.2 Příklad zpracování hran při první iteraci algoritmu 33
4.3 Příklad zpracování hran při druhé iteraci algoritmu 33
4.4 Příklad grafu na kterém běží Dijkstrův algoritmus 36
4.5 Sestavení kostry grafu - 1. krok 36
4.6 Sestavení kostry grafu - 2. krok 37
4.7 Sestavení kostry grafu - 3. krok 37
4.8 Sestavení kostry grafu - 4. krok 38
4.9 Sestavení kostry grafu - výsledná kostra grafu 38
4.10 Jednoduchý příklad principu A* 42
5.1 Příklad topologie kruh 45
5.2 Příklad topologie hvězda 46
5.3 Příklad topologie strom 46
5.4 Příklad slabě propojené topologie 47
5.5 Příklad silně propojené topologie 47
5.6 Příklad jednoduché topologie se cenou hran 48
5.7 Porovnání rychlosti dosažení konvergence Bellman-Ford algoritmu na různých topologiích 50
5.8 Porovnání rychlosti dosažení konvergence Dijkstrová algoritmu na různých topologiích 51
5.9 Porovnání obou algoritmů na topologii kruh 52
5.10 Porovnání obou algoritmů na topologii hvězda 53
5.11 Porovnání obou algoritmu na topologii strom 53
5.12 Porovnání obou algoritmů na slabě propojené topologii 54
5.13 Porovnání obou algoritmů na silně propojené topologii 54
5.14 Výsledky spouštění obou algoritmů na nahodilých topologiích 56
5.15 Výsledky spouštění obou algoritmů na nahodilých topologiích s maximálním počtem vrcholů do 160 57
5.16 Ukázká rozdílu mezi grafem a mřížkou 59
5.17 Porovnaní obou algoritmů s maximálním počtem uzlů do 1000 63
5.18 Porovnaní obou algoritmů s maximálním počtem uzlů do 160 64

5.19 Porovnaní všech algoritmů s maximálním počtem uzlů do 1000 65
5.20 Porovnaní všech algoritmů s maximálním počtem uzlů do 160 65

SEZNAM TABULEK

5.1 Bellmanův-Fordův algoritmus na odlišných typech topologii 49
5.2 Dijkstrův algoritmus na odlišných typech topologii 51

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. 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 vybrané Bellman-Fordův a Dijkstrův algoritmus a u stochastických jsou vybrané A* (a star) a Push-Sum.

V praktické části je provedená implementace čtyř algoritmů v prostředí Matlab, kde je dále měřena rychlost jejích konvergence na vytvořených topologiích sítí pomoci matice sousednosti, matice cen a u A* algoritmu pomocí mřížkových struktur, což jsou grafy převedené do mřížové struktury. Základní topologie jsou typu kruh, hvězda, strom, slabě propojená topologie a silně propojená topologie neboli také mesh, 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 [17].

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 (real-time - např. v letectví). 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 [17] 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. Např. 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 tolerovat. 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 (Byzantine failure [1]), 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 [17]:

- 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

Ale ne každý algoritmus se musí potykat se všemi nebo většinou výše uvedených problému. Často se vybere nejlepší možný algoritmus na řešení určitého pro-

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

[2] V problematice sítí procesor může označovat proces, tj. samostatné zařízení např. směrovač, přepínač, apod.

blémů, který má kompromis mezi jeho ostatními parametry (rychlost, čas, chybovost, apod.).

1.2 Standardní problémy

Problémy, které řeší distribuované algoritmy je možné rozdělit na [17]:

- **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 two-phase commit protocol a 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 ν.
  2. **Platnost (Validity):** Pokud všechny procesy nabídnou stejné ν, tak každý korektní proces se rozhodné pro ν.
  3. **Integrita (Integrity):** Každý korektní proces rozhodne pouze o jedné hodnotě a pokud rozhodné o hodnotě ν, tak ν musí byt nabídnuté některým procesem.
  4. **Shoda (Agreement):** Pokud se proces, který nemá chybný stav, rozhodné pro ν, tak každý korektní proces se rozhodne pro ν.

  Typickým algoritmem na řešení shody je Paxos.
- **Distribuované hledaní (Distributed search):** Hledaní nejkratší cesty v topologii nebo systému. Algoritmus na řešení toho problému je např. 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.).
- **Spanning tree generation:** Generování kostry 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 byt 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) [20] [16] [19].

V současnosti se stochastické algoritmy podle [16] a [19] dají rozdělit do několik hlavních odvětví, které budou v dalších kapitolách stručně[1] popsané. Mezi tyto odvětvě patří aproximační algoritmy, genetické a evoluční[2].

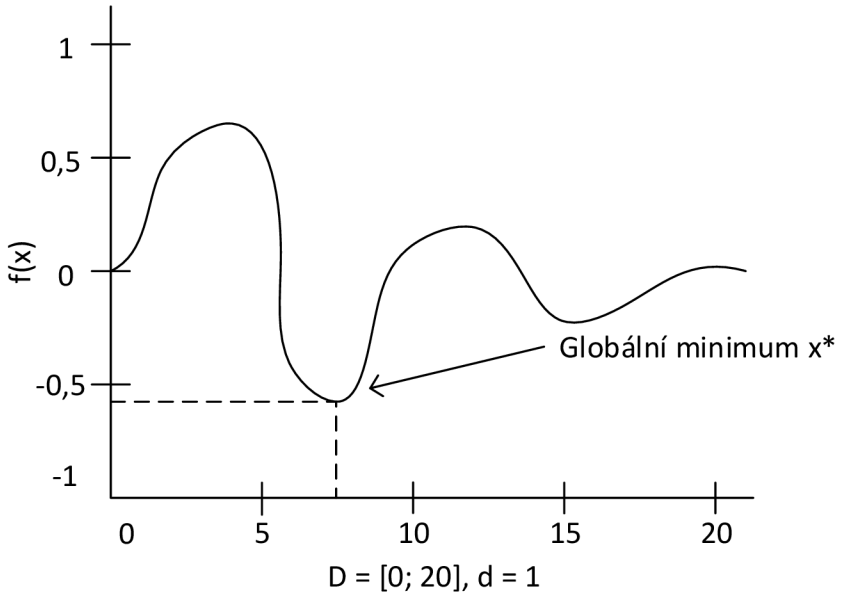
2.1 Problém globální optimalizace

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 více 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 [16] [19].

[1] Detailní matematický popis a výpis nejpoužívanějších algoritmů je nad rámec teto práce - bylo by to na několik velmi tlustých knih.

[2] Existují i další dělení s vlastními algoritmy, které jsou nad rámec tohoto textu.



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 [19].

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 $g(\boldsymbol{x}) = -f(\boldsymbol{x})$ [19].

Analýza problému globální optimalizace ukazuje, že neexistuje deterministický algoritmus řešící obecnou úlohu globální optimalizace (tj. nalezení dostatečně přesné aproximace $\boldsymbol{x}^*$) v polynomiálním čase, tzn. problém globální optimalizace je NP-obtížný (časová složitost roste exponenciálně se zpracováním úkolu) [19].

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) [19].

Pro zjednodušení budeme uvažovat, že prostor, kde hledáme globální minimum je souvislý, jak je definováno podmínkou [19]:

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

Účelovou *f(**x**)* umíme vyhodnotit s požadovanou přesností v každém bodě $\boldsymbol{x} \in D$. Podmínce 2.2 se říká 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 uncostrained continuous problems [19].

Úlohy hledání vázaného extrému (constrained optimization) jsou obvykle formulovány takto [19]:

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

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

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

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

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 náhoda, intuice, analogie 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 quick-sort a insert-sort nikoliv (algoritmy 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 a to přesněji mezi gradientní optimalizační metody. Jedná se o jednoduchý, avšak velmi 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 [20].

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 [20]:

- **Optimalizaci:** Hledání minima/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 (respektive metody) založené na simulaci. Patří sem neuronové sítě (simulují nervový systém) a evoluční algoritmy, které budou dále stručně popsané.

Evoluční algoritmy simulují evoluční proces v přírodě (odtud název) 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 konkretnímu 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 dáne oblasti již používané [20].

2.4 Genetické algoritmy

Genetické algoritmy[3] 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 i algoritmus A*[4]. 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

[3]Detailnější popis genetických algoritmů je na `https://akela.mendelu.cz/~xpopelka/cs/ui/ucici/` nebo jiné k tomu určené literatuře.

[4]Algoritmus A star se používá k vyhledávaní optimální cesty v grafu (kladně ohodnoceném grafu) a pracuje na základě Dijkstrova algoritmu, ale navíc má heuristický prvek.

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 [20].

3 MATEMATICKÝ NÁSTROJ

Při analýze a podrobném popisu distribuovaných algoritmů a stochastických algoritmů použitých v této práci, se nejvíce využívají matematické nástroje z oblasti teorií grafu a lineární algebry. Obě oblasti jsou navzájem úzce propojené a v této kapitole budou oba nástroje stručně popsané [9]:

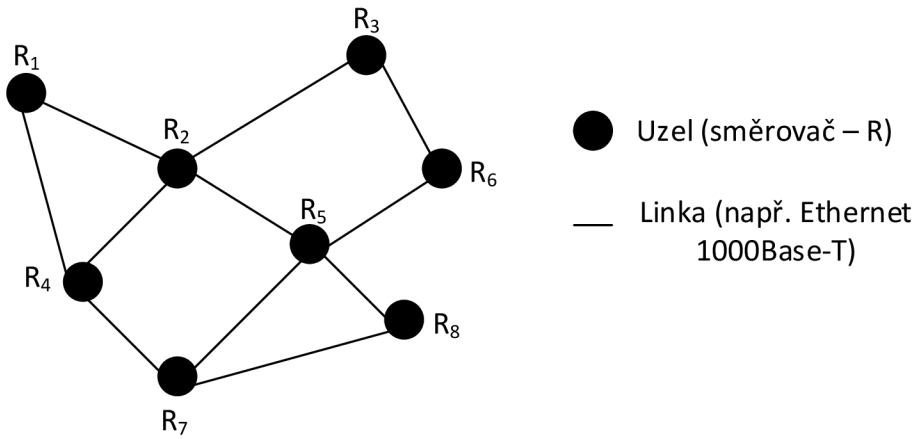
3.1 Teorie grafů

Teorie grafů se zabývá, jak už název napovídá, analýzou grafů. Graf se skládá z množiny vrcholů, které jsou navzájem propojené hranami. Matematická definice grafu je taková [9] [17]:

$$G = (V, E), \tag{3.1}$$

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 3.1. 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.

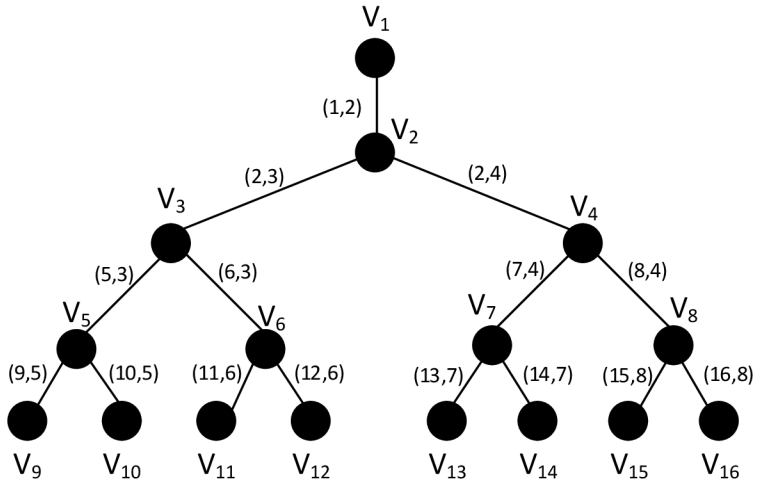


Obr. 3.1: Ukázka zobrazení sítě v podobě grafu

Grafy se dělí na dvě skupiny. První je skupina neorientovaných grafů, kde není počáteční a koncový vrchol u hrany, tzn. všechny vrcholy v grafu jsou rovnocenné.

Na obrázku 3.2 je ukázka neorientovaného grafu. Matematický zápis takového grafu je $\{i,j\} \in \epsilon$ a dále u neorientovaných grafů je používán zápis $(i, j) \in \epsilon$. Jelikož zde není směrová orientace hran, tak platí tvrzení [9]:

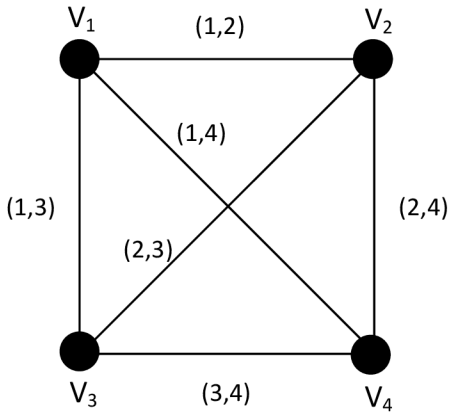
$$(i, j) = (j, i). \tag{3.2}$$



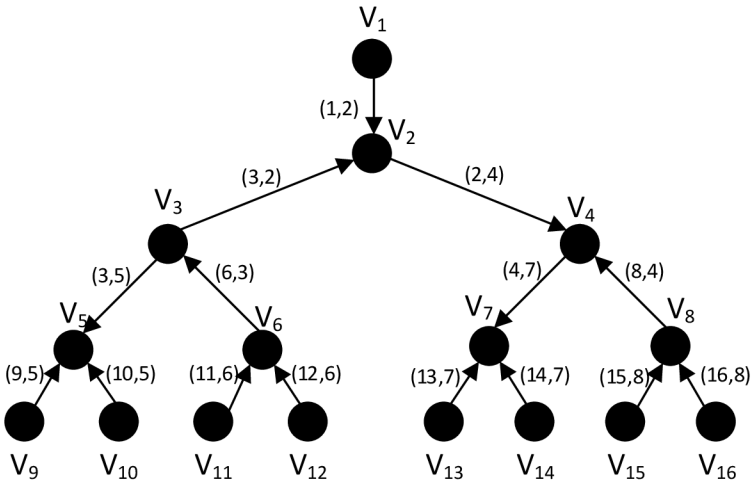
Obr. 3.2: Příklad neorientovaného grafu

Speciálním příkladem neorientovaného grafu je graf, kde každý komunikuje s každým, viz 3.3. Matematický zápis takového grafu je: $V = (v_1,v_2,v_3,v_4)$, $E = \{(v_1,v_2)$, (v_1,v_3), (v_1,v_4), (v_2,v_3), (v_2,v_4), $(v_3,v_4)\}$. Matematický správně by bylo ještě uvést hrany $\{(v_1,v_1)$, (v_2,v_2), (v_3,v_3), $(v_4,v_4\}$. Toto vypadá na nesmysl, ale v praxi se této funkce využívá především jako loopback (simulace komunikace na jiné rozhraní, ale ve skutečnosti komunikují sám se sebou) [9].

Druhou skupinou grafů je orientovaný graf. U orientovaného grafu záleží na pořadí vrcholů, tj. u hrany je počateční a koncový vrchol. Taková hrana se může označit šipkou od počátečního uzlu ke koncovému (odtud název orientovaný) a všechny vrcholy orientovaného grafu jsou nerovnocenné. Na obrázku 3.4 je ukázka orientovaného grafu [9].



Obr. 3.3: Speciální příklad neorientovaného grafu, kde každý komunikuje s každým



Obr. 3.4: Příklad orientovaného grafu

3.2 Lineární algebra

V teorii grafů se používají poznatky z lineární algebry, nejčastěji jako zápis grafu pomocí matice sousednosti, kterou se může zobrazit pouze orientovaný graf. V případě neorientovaného se nejdříve musí provést transformace na orientovaný a pote lze graf zapsat matici sousednosti. Velikost matice je funkcí počtu vrcholů (čím větší síť, tím větší matice). Matematický zápis pro toto tvrzení je: $size(A) = |V| \times |V|$.

Matice se vytváří tak, že pro každou dvojici vrcholů (u,v) si zapamatujeme, jestli z vrcholu u vede hrana do vrcholu v. Z tvrzení popsaného v předešlém odstavci plyne, že rozměr matice musí mít stejnou velikost jako počet vrcholů grafu. Jednotlivé prvky matice se zapisují následujícím postupem: pokud je mezi vrcholy i a j hrana, tak se zapíše na pozici a_{ij} číslo 1 a pokud není, tak se zapíše číslo 0. Pro matematický popis předchozí věty je třeba nadefinovat matici sousednosti pro kterou platí $M = N$ [9] [3]:

$$\begin{pmatrix} a_{11} & \cdots & a_{1M} \\ \vdots & \ddots & \vdots \\ a_{N1} & \cdots & a_{NM} \end{pmatrix} \tag{3.3}$$

Dále když $\{v_i,v_j\} \in \epsilon$, tak prvek matice $a_{ij} = 1$ v opačném případě $a_{ij} = 0$, kde v_i a v_j označují jednotlivé vrcholy. Pro neorientované grafy také platí, že matice sousednosti musí byt symetrická, tj. $(v_i,v_j) = (v_j,v_i)$ a zároveň $a_{ij} = a_{ji}$. Pak matice sousedství vypadá následovně:

$$\begin{pmatrix} a_{11} & \cdots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{N1} & \cdots & a_{NN} \end{pmatrix} \tag{3.4}$$

Úlohou matice sousednosti je tedy popsat jednotlivé hrany grafu pomocí maticového zápisu, což výrazným způsobem zjednodušuje práci s grafy. Matice sousednosti pro obrázek 3.3 z předešle kapitoly by vypadala takto:

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

Matice má ještě jednu zajímavou vlastnost a to takovou, že když matici A umocníme na k, dostaneme výslednou matici jejíž hodnoty reprezentují počet cest délky k z uzlu i do uzlu j a naopak. Pokud budeme u vytvořené matice sousednosti k obrázku 3.3 hledat i počet cest vedoucích přes čtyři hrany, je postup následující:

$$\begin{pmatrix}0&1&1&1\\1&0&1&1\\1&1&0&1\\1&1&1&0\end{pmatrix} * \begin{pmatrix}0&1&1&1\\1&0&1&1\\1&1&0&1\\1&1&1&0\end{pmatrix} * \begin{pmatrix}0&1&1&1\\1&0&1&1\\1&1&0&1\\1&1&1&0\end{pmatrix} * \begin{pmatrix}0&1&1&1\\1&0&1&1\\1&1&0&1\\1&1&1&0\end{pmatrix} = \begin{pmatrix}21&20&20&20\\20&21&20&20\\20&20&21&20\\20&20&20&21\end{pmatrix}$$

Z výsledné matice plyne, že pokud se chceme dostat z uzlu i do uzlu j přes 4 hrany, tak existuje 20 způsobů jak toho docílit. A pokud se chceme i vrátit cestou přes uzly z j do i, existuje 21 možností. Případy v praxi, kde uzly jsou navzájem propojené každý s každým, často nejsou. Proto následující příklad poslouží jako další ukázka [3].

Mějme určitou síť, která je tvořena nesměrovým grafem a kde graf je popsán touto matici sousednosti:

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

Z matice je vidět, že síť je tvořena čtyřmi uzly a na základě jedniček a nul je možné prohlásit kdo s kým sousedi. Takže uzly 1-2, 1-3, 2-4 a 3-4 spolu sousedí a uzly 1-4 a 2-3 ne. Nyní vyřešíme počet cest z uzlu i do uzlu j přes 4 hrany:

$$\begin{pmatrix}0&1&1&0\\1&0&0&1\\1&0&0&1\\0&1&1&0\end{pmatrix} * \begin{pmatrix}0&1&1&0\\1&0&0&1\\1&0&0&1\\0&1&1&0\end{pmatrix} * \begin{pmatrix}0&1&1&0\\1&0&0&1\\1&0&0&1\\0&1&1&0\end{pmatrix} * \begin{pmatrix}0&1&1&0\\1&0&0&1\\1&0&0&1\\0&1&1&0\end{pmatrix} = \begin{pmatrix}8&0&0&8\\0&8&8&0\\0&8&8&0\\8&0&0&8\end{pmatrix}$$

Z výsledné matice je vidět výrazný pokles možnosti. Lze se dostat přes jednu hranu z uzlu i do uzlu j, ale přes čtyři hrany to možné není. Z toho plyne, že čím menší vzájemná dostupnost uzlů, tím je menší počet možností při přechodu více hranami v jednom kroku. Sítě, které byly v příkladech výše, jsou neorientované grafy a můžeme si všimnou vzájemné rovnocennosti uzlů. Tuto skutečnost lze zapsat matematickým vztahem:

$$N(v_i) = \{v_j : (v_i, v_j)\} \in \epsilon \tag{3.5}$$

Pokud bude síť popsána pomocí orientovaného grafu, tak sousední uzly už rovnocenné nebudou a mohou nastat dvě situace. Buď hrana z uzlu vychází nebo do něj vchází. Uzly v orientovaném grafu se dále dělí na dvě skupiny a to na předka a potomka. Rodičovským se označuje počáteční bod hrany a dětským se označuje

koncový bod hrany. Počet sousedů pro předka můžeme matematicky vyjádřit vztahem:

$$\pi(v_i) = \{v_j : (v_i, v_j)\} \in \epsilon \tag{3.6}$$

a počet sousedů pro potomka:

$$\lambda(v_i) = \{v_i : (v_j, v_i)\} \in \epsilon \tag{3.7}$$

Dalším důležitém parametrem v teorii grafů je stupeň vrcholu. Mějme graf G, kde v je jeho vrchol, pak stupeň vrcholu vyjádříme jako:

$$d(v_i) = |N(v_i)|, \tag{3.8}$$

kde d je je stupeň vrcholu, který vyjadřuje počet hran grafu G obsahující vrchol v. Výpočet lze zjednodušit, pokud se graf vyjádří pomocí matice sousednosti. Pak výpočet d bude:

$$d(v_i) = \sum_{j=1}^{n} [A]_{ij} = \sum_{j=1}^{n} [A]_{ji}. \tag{3.9}$$

Vzorec říká, že součet řádků nebo sloupců určuje počet sousedů vrcholu odpovídajícího danému řádku. Vzorec platí pro neorientovaný graf. U orientovaného je výpočet:

$$d_{out}(v_i) = \sum_{j=1}^{n} [A]_{ij}, d_{in}(v_i) = \sum_{j=1}^{n} [A]_{ji}, \tag{3.10}$$

kde $d_{out}(v_i)$ vyjadřuje pro kolik vrcholů grafu je vrchol i dítětem a $d_{in}(v_i)$ vyjadřuje pro kolik vrcholů představuje vrchol i rodiče. Stupeň vrcholu pro daný vrchol dostaneme součtem obou hodnot:

$$d(v_i) = d_{out}(v_i) + d_{in}(v_i). \tag{3.11}$$

Kromě parametrů, které jsou popsané výše, jsou i jiné, např. průměr grafů, který definuje největší možnou vzdálenost mezi dvěma libovolnými vrcholy. Dalším parametrem je hustota grafu, což je poměr mezi skutečným počtem hran v grafu a maximálním možným, který by se mohl v grafu vyskytovat. Hustota může nabývat maximální hodnoty 1, pro případ kde počet hran je rovny maximálnímu počtu hran. Při vypočtu obou parametrů se opět využívá matice sousednosti [3].

4 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. Ze stochastických algoritmů byly vybrané taky dva, ale jejich výběr byl ztížen omezeným počtem algoritmů využívaných v síťové problematice nebo v praxi. Většinou se také jedná pouze o experimentální algoritmy ve fází 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 jedna o nový přístup k řešení problematiky v sítích, který se objevil v posledních letech pomocí samotného rozvoje stochastických algoritmů a výpočetní síly v zařízeních.

4.1 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 zajímavost 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 (rychlost linky, zpoždění, a další parametry specifikované v protokolu) nebo RIP s distance-vector, kde se počítá počet skoku k cílí (hops).

4.1.1 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 také publikoval v roce 1957.

Algoritmus počítá nejkratší cestu v ohodnoceném grafu mezi dvěma libovolnými vrcholy, kde hrana mezi nimi může byt ohodnocena i zápornou hodnotou. V reálných

[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, tzn. každý směrovač v síti ví o všech ostatních a zjednodušené řečeno zná mapu sítě.

sítích 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 uzlu u zlepší hodnota $d[u]$, musí se prozkoumat všechny hrany (u,v) ϵ 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 [4]):

```
bellman-ford(vrcholy, hrany, zdroj)

% Inicializace grafu pro algoritmus
for kazde v ve vrcholy
  if v=zdroj then v.vzdalenost := 0
             else v.vzdalenost := nekonecno
  v.predchudce := null

% Opakovana tzv. relaxace hran
for i od 1 do velikost(vrcholy)-1
  for kazde h v hrany     % h je hrana z u do v
    u := h.pocatek
    v := h.konec
    if u.vzdalenost + h.delka < v.vzdelenost
      v.vzdalenost := u.vzdalenost + h.delka
      v.predchudce := u

% Volitelna cast: kontrola zapornych cyklu
for kazde h v hrany
  u := h.pocatek
  v := h.konec
  if u.vzdalenost + h.delka < v.vzdalenost
```

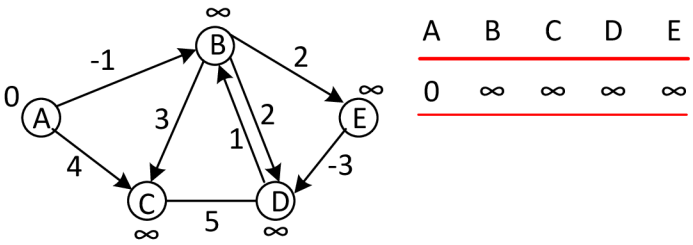
[3]Přesněji do této metody vstupují dva vrcholy a hrana mezi nimi. Pokud je vzdálenost zdrojového vrcholu sečtená s délkou hrany menší než aktuální vzdálenost cílového vrcholu, tak se za předchůdce cílového vrcholu na nejkratší cestě označí zdrojový vrchol (přepočte se vzdálenost cílového vrcholu). V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

```
    error "Graf obsahuje zaporny cyklus."
```

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 byt $[\boldsymbol{V}]$ - **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 [5] [6] [17].

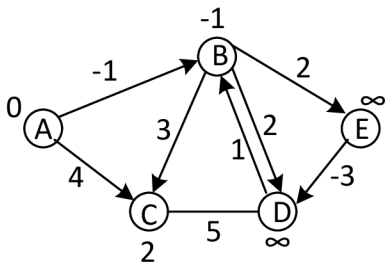
Příklad práce algoritmu

Pro lepší pochopení algoritmu je dobré si ukázat přímo příklad činnosti algoritmu na určitém grafu. Mějme zdrojový vrchol 0 a inicializujme všechny vzdálenosti na nekonečno, kromě samotného vrcholu 0, viz 4.1. Celkový počet vrcholů v grafu je 5, tzn. každá hrana bude počítaná čtyřikrát, což plyne z teorie popsané v předešlém textu.



Obr. 4.1: Příklad počátečního stavu algoritmu

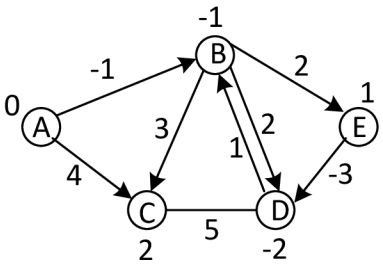
Mějme následujícím pořadí zpracování hran: $(\boldsymbol{B,E})$, $(\boldsymbol{D,B})$, $(\boldsymbol{B,D})$, $(\boldsymbol{A,B})$, $(\boldsymbol{A,C})$, $(\boldsymbol{D,C})$, $(\boldsymbol{B,C})$, $(\boldsymbol{E,D})$. Při první iteraci dostaneme vzdálenosti jako na obrázku 4.2. První řádek zobrazuje počáteční vzdálenosti. Druhý řádek zobrazuje vzdálenosti, když jsou zpracované hrany $(\boldsymbol{B,E})$, $(\boldsymbol{D,B})$, $(\boldsymbol{B,D})$ a $(\boldsymbol{A,B})$. Třetí řádek zobrazuje zpracování hrany $(\boldsymbol{A,C})$ a poslední řádek zobrazuje zpracování hran $(\boldsymbol{D,C})$, $(\boldsymbol{B,C})$ a $(\boldsymbol{E,D})$. Dále při první iteraci dostaneme všechny nejkratší cesty s maximální vzdálenosti o velikosti jedné hrany. Konečné hodnoty druhé iterace jsou na 4.2 v posledním řádku.



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

Obr. 4.2: Příklad zpracování hran při první iteraci algoritmu

Na posledním obrázku 4.3 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 jsou tak minimalizované po druhé iteraci, takže třetí a čtvrtá iterace už nezlepší vzdálenosti [6].



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. 4.3: Příklad zpracování hran při druhé iteraci algoritmu

4.1.2 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 nezáporným ohodnocením hran. Algoritmus udržuje množinu $\boldsymbol{S}$ vrcholů, pro které se už vypočítá délka nejkratší cesty. 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 [8]):

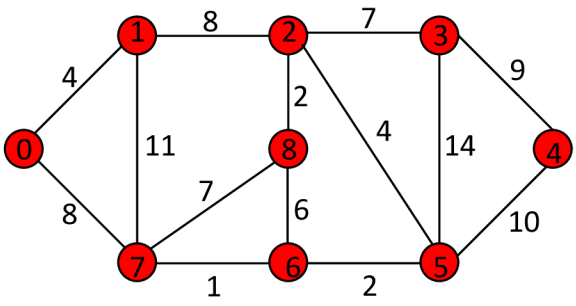
```
Dijkstra(E, V, s):

% Pocatecni inicializace
for kazdy vrchol v ve V:

% Pocatecni neznama vzdalenost ze startu s do vrcholu v
        d[v] := nekonecno

% Potomek na nejkratsi ceste ze startu s k cili
        p[v] := nedefinovano

% Pocatecni vzdalenost ze startu s do s
        d[s] := 0
```

```
% Mnozina nenavstivenych vrcholu
        N := V

% Zacatek behu algoritmu
while N neni prazdny:

% Vytahne nejlepsi vrchol
u := vytahni_min(N)
for kazdeho souseda v u vrcholu u:

% v 1. smycce cyklu u je s d[u] = 0
        alt = d[u] + l(u, v)
                if alt < d[v]
                        d[v] := alt
                        p[v] := u

% Zajisteni cesty ze zdroje k cili
S := prazdna sekvence
u := cil
while definovane p[u]
        vloz u na zacatku S
        u := p[u]
```

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 [7] [17] [3].

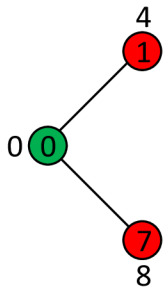
Příklad práce algoritmu

Jako u Bellman-Fordová algoritmu i zde je dobré pro pochopení ukázat činnost algoritmu na příkladu. Mějme následující graf 4.4. Množina $\boldsymbol{N}$ je na začátku prázdna a vzdálenosti z vrcholu 0 k ostatním vrcholům jsou $\{0,\infty,\infty,\infty,\infty,\infty,\infty,\infty\}$. V příkladu pro názornou ukázku činnosti algoritmu je zvolena cesta z vrcholu 0, který je zdrojem (resp. kořenem), k ostatním vrcholům - tzv. sestavení kostry grafu, protože bude vidět celkovou kostru grafu nejmenších cen, kterou si udržuje každý vrchol v grafu a bude možné dohledat nejkratší cestu (resp. nejlevnější) od kořene k libovolnému vrcholů.



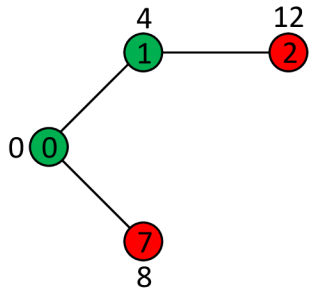
Obr. 4.4: Příklad grafu na kterém běží Dijkstrův algoritmus

V prvním kroku se vezme vrchol s nejmenší vzdálenostní hodnotou (cenou), což je v příkladu vrchol 0 a vloží se do množiny $\boldsymbol{N}$. Takže množina $\boldsymbol{N}$ bude obsahovat $\{0\}$. 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 4.5 je zobrazen subgraf sestavení kostry grafu Dijkstrovým algoritmem, kde zelená barva značí vrcholy sestavení nejkratších cest.



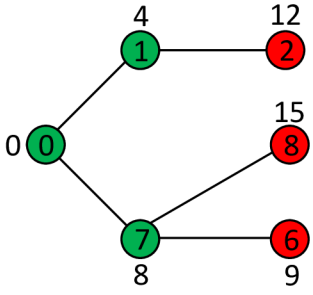
Obr. 4.5: Sestavení kostry grafu - 1. krok

V dalším kroku se vezme vrchol s nejmenší vzdálenostní hodnotou (cenou) a zároveň, který ještě není v kostře grafu. V příkladu to bude vrchol 1. Množina ($\boldsymbol{N}$) nyní obsahuje {0,1} a dále se aktualizuji vzdálenostní hodnoty sousedů vrcholu 1, takže vzdálenostní hodnota vrcholu 2 bude 12, což je zobrazeno v subgrafu 4.6.



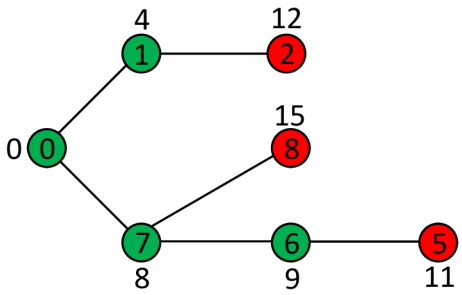
Obr. 4.6: Sestavení kostry grafu - 2. krok

Ostatní kroky jsou podobné. Opět se vezme vrchol s nejmenší cenou a zároveň nepřítomný v kostře grafu. Tentokrát to bude vrchol 7 a množina ($\boldsymbol{N}$) tak obsahuje {0,1,7}. Aktualizují se ceny sousedů vrcholu 7 a cena vrcholu 6 bude 15 a vrcholu 8 bude 9, viz 4.7.



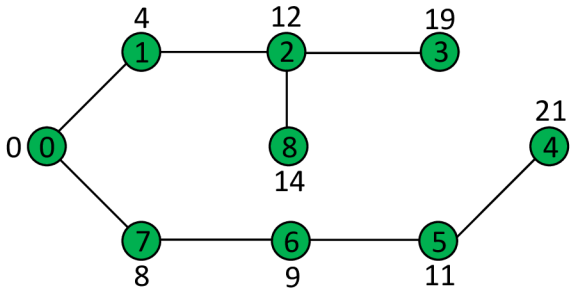
Obr. 4.7: Sestavení kostry grafu - 3. krok

Nyní se vezme vrchol 6 a aktualizuji se ceny sousedů tohoto vrcholu. Množina ($\boldsymbol{N}$) bude obsahovat {0,1,7,6}, viz subgraf 4.8.



Obr. 4.8: 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. nejlevnější). Výsledná kostra grafu je na 4.9 [7].



Obr. 4.9: Sestavení kostry grafu - výsledná kostra grafu

4.2 Výběr dvou stochastických algoritmů

Výběr stochastických algoritmů, jak už bylo naznačeno na začátku kapitoly 4 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, anebo jinému

vědnímu oboru, který nijak nesouvisí se sítěmi. Když už se nějaký vhodný algoritmus našel, tak buď u něj chyběla detailní dokumentace, kde by byl algoritmus vhodně popsán, což by pomohlo pro implementaci v programovacích jazycích anebo byl pouze ve fázi teoretického návrhu, ale v reálné síti nebyl použit a ani nebyl testován na umělých simulovaných sítích.

Příčinou tohoto problému může byt, 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 dvou 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.

4.2.1 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 ceny, která zbývá na cestě k cílí. Konkrétně A* vybírá cestu na základě $f(n)=g(n)+h(n)$, kde n je poslední vrchol cesty, $g(n)$ je cena ze startovního vrcholu k n vrcholu a $h(n)$ je heuristika,

[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`

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 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 [14] [10] [11]. Ukázka pseudokódu (převzato z [2]):

```
funkce A_star(zdroj, cil)
 % Mnozina uz vyhodnocenych uzlu
 closedSet := {}

 % Mnozina objevenych uzlu, ale ktere jeste
 % nebyly vyhodnocene. Na zacatku je znam pouze
 % pocatecni uzel, tj. zdroj.
 openSet := {zdroj}

 % Pro kazdy uzel a uzel jenz muze byt dosahnut
 % nejefektivneji z. Pokud uzel muze byt dosahnut z vice
 % uzlu, priselZ bude obsahovat nejefektivnejsi predesly
 % krok.
 priselZ := prazdna mapa

 % Pro kazdy uzel cenu ze zdrojoveho uzlu k tomuto uzlu.
 gScore := mapa s pocatecni hodnotou nekonecno

 % Cena ze zdroje ke zdroji je nulova.
 gScore[zdroj] := 0

 % Pro kazdy uzel celkova cena cesty ze zdroje k cili
```

```
% u tohoto uzlu. Hodnota je castecne znama heuristikou.
fScore := mapa s pocatecni hodnotou nekonecno

% Pro prvni uzel je hodnota vypocitana heuristikou.
fScore[zdroj] := vypocet_ceny_heuristikou(zdroj, cil)

 while openSet neni prazdny
 aktualni := uzel v openSet majici nejnizsi
             fScore[] hodnotu
  if aktualni = cil
          return rekonstrukce_cesty(priselZ, aktualni)

  openSet.Remove(aktualni)
  closedSet.Add(aktualni)
        for kazdy vrchol v aktualni
   if soused v closedSet
    pokracuj % Ignoruj souseda, ktery uz byl vybran

   % Vzdalenost ze zdroje k aktualnimu uzlu.
   tentative_gScore := gScore[aktualni] +
                       vzdalenost_mezi(aktualni, soused)
   if soused neni v openSet     % Objev novy uzel
   openSet.Add(soused)
   else if tentative_gScore >= gScore[soused]
    pokracuj % Toto neni dobra cesta
   % Toto je nejlepsi cesta do ted. Zapis ji!
   priselZ[soused] := aktualni
   gScore[soused] := tentative_gScore
   fScore[soused] := gScore[soused] +
                     vypocet_ceny_heuristikou(soused, cil)
 return selhani
funkce rekonstrukce_cesty(priselZ, aktualni)
 celkova_cesta := [aktualni]
  while aktualni v priselZ.Keys:
  aktualni := priselZ[aktualni]
  celkova_cesta.append(aktualni)
 return celkova_cesta
```

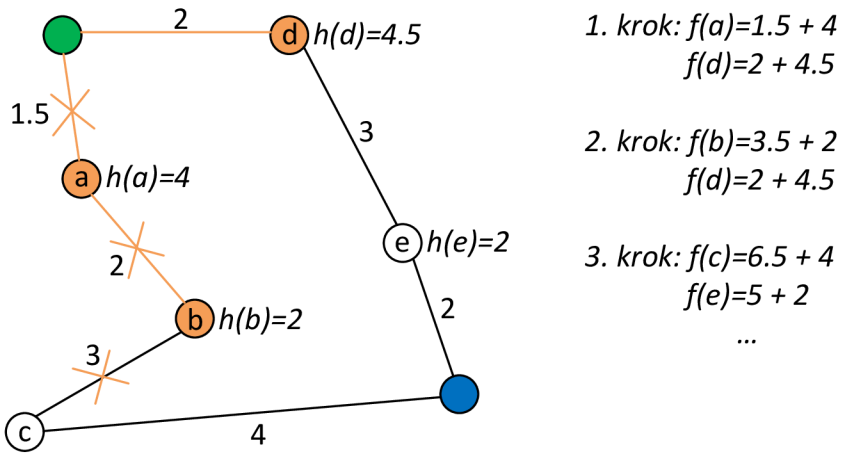
Příklad práce algoritmu

Následující ukázka 4.10 vysvětluje funkčnost algoritmů, kde *h(x)* je heuristika, což je vzdušná vzdálenost k cíli u jednotlivých uzlů *a, b, c, d, e* (Příklad převzat a dále zjednodušen a úpraven z [18]). Zelená barva značí zdroj, modrá cíl a žlutá zpracované uzly.

V prvním kroku se začne ze zdroje a jeho sousedů a do množiny *openSet* se tedy vloží uzel *a* a uzel *d*, které se vyhodnotí. Cena cesty k sousednímu uzlu *a* je 1.5, což je více než k sousednímu uzlu *d*, kde je cena cesty 2. Takže se začne hledat cesta k cíli z uzlu *a*. Dále jak je známo z popisu A* algoritmu v předešle kapitole, tak celková cena se počítá vzorcem *f(n)=g(n)+h(n)*. V příkladu je u uzlu *a* celková cena 5.5 a u zlu *d* je celková cena 6.5 (viz 4.10). Takže platí $f(a) < f(d)$ a tak jako nejvhodnější uzel k cíli se vybere uzel *a*.

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

V třetím kroku se obdobně zpracují uzly *c* a *e*, které se vloží do množiny *openSet*. Výsledek porovnaní celkových cen obou uzlů je $f(c) > f(e)$. Takže jako další nejlepší uzel k cíli bude uzel *e*. Do množiny *closedSet* se vloží oboje již zpracované uzly.



Obr. 4.10: Jednoduchý příklad principu A*

Algoritmus takto pokračuje dokud nesestaví nejkratší cestu k cílovému uzlu. Jako nejlepší se jevila až do třetího kroku cesta přes uzly *a, b*, ale od třetího kroku je nejlepší cesta přes uzly *d, e*. Vyhodnocovala se vždy *f(x)* cena aktuálního uzlu k cíli a ceny sousedních uzlů se následně aktualizovaly. Uzly 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 uzly *a, b*) od třetího kroku (značí oranžové kříže).

4.2.2 Push-sum algoritmus

Druhým vybraným stochastickým algoritmem byl Push-sum algoritmus. Push-sum algoritmus nebo také někdy nazýván push-sum protokol je multiúčelový epidemický se šířící algoritmus, jehož funkcionalita je založena na distribuci hodnot mezi páry agentů, resp. uzlů. 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. Výsledky se mohou lišit navzdory zachování konstantních vstupních dat. Důvodem je nahodilost procesu zpracování výsledků. Push-sum může řešit po své modifikaci různé problémy. Existuje i několik variant[5] algoritmu. V této práci je použita základní varianta algoritmu.

Na začátku v prvním kroku algoritmu je každému uzlu přiřazen počáteční vnitřní stav roven jedné. Počáteční stav uzlu odráží jeho váhu v sítí. V dalším kroku je vybrán náhodně jeden ze sousedu daného uzlu při každé iteraci. Zvolenému uzlu je zaslaná poloviční hodnota a váha odesílajícího uzlu. Odeslané hodnoty se uloží do pamětí odesílatele, což umožní každému uzlu vypočítat poměr těchto hodnot. Tento postup je matematický definován takto [12] [15]:

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 byt implementován do distribuovaného 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 `http://www.cs.cornell.edu/johannes/papers/2003/focs2003-gossip.pdf`.

5 PRAKTICKÁ ČÁST

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 topologiích s postupně rostoucí velikosti (tj. počtem uzlů) 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 jednotlivým typům algoritmů jsou výsledky v podobě grafu, kde jsou zobrazené rychlosti konvergence jednotlivých algoritmů.

5.1 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 4.1, tak Bellmanův-Fordův algoritmus je aplikovan ve směrovacím protokolu RIP a používá distanční vektor, tj. počet skoků k cílí (hops), což jsou další uzly na cestě k cílovému uzlu. 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 topologiích s různou velikosti a nahodilým uspořádáním uzlů. Konkrétně algoritmy se zvolily proto, aby se zjistilo jestli je efektivnější algoritmus, který jako kriterium nejlepší cesty bere počet skoku k cíli nebo algoritmus, který zná celou topologií sítě včetně cen linek a na základě teto znalosti vybírá nejvhodnější cestu k cíli.

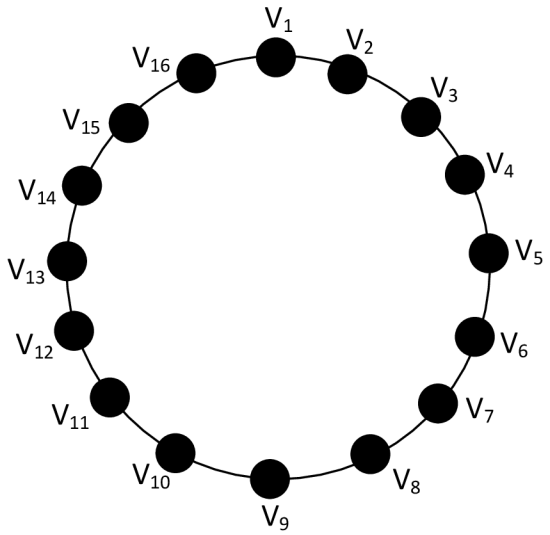
Dále bylo potřeba ujasnit si, jak porovnávat konvergenci u těchto algoritmů. Konvergence se zde dá chápat tak, že je to stav, kdy všechny uzly ví o všech uzlech a je možná komunikace mezi dvěma libovolnými uzly v sítí. V našem případě sítě budou představovat neorientované grafy. Neorientované nejvíce 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. full-duplex přenos).

Takže konvergence je zde stav, kdy je možná komunikace mezi dvěma libovolnými uzly 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 (start), druhá konec (stop) měření času (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 dánem grafu neboli topologii konvergoval.

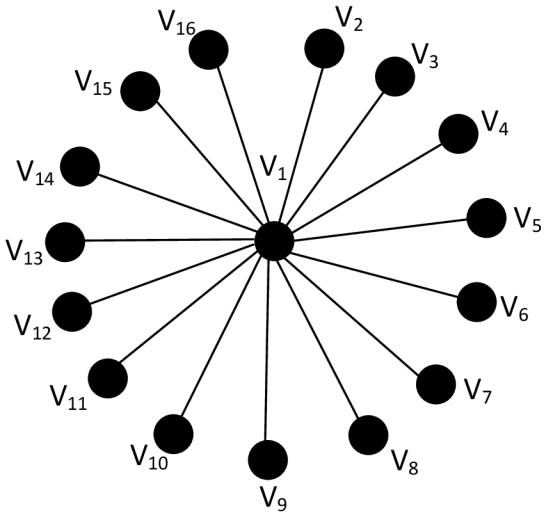
5.1.1 Vytvořené topologie

Prvním krokem realizace testování bylo vytvoření topologií na kterých by se dané algoritmy testovaly. Pomocí matic sousednosti byly sestavené topologie (grafy) o 16 vrcholech. Jedná se o následující topologie: kruh (ring) - obr. 5.1, hvězda (star) - obr. 5.2, strom (tree) - obr. 5.3, slabě propojená topologie (weak connected topology) - obr. 5.4, silně propojena topologie (strong connected topology) neboli také Mesh, kde je každý propojený s každým - obr. 5.5. U silně propojené topologie je zde z důvodu přehlednosti uveden přímo obrázek z prostředí Matlab.

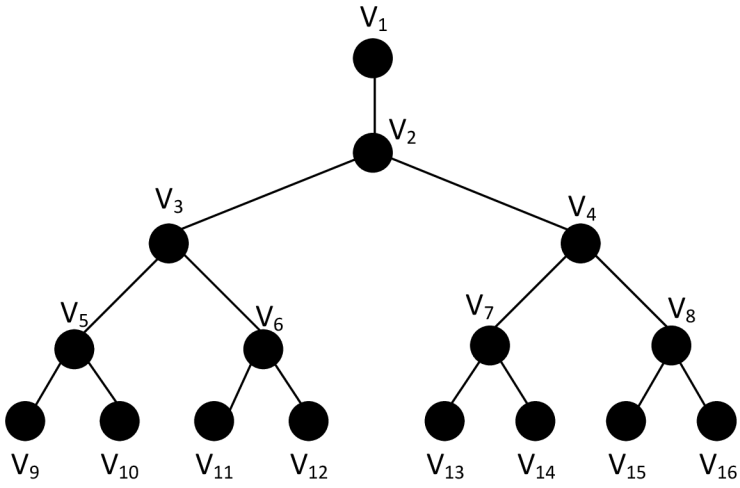


Obr. 5.1: Příklad topologie kruh

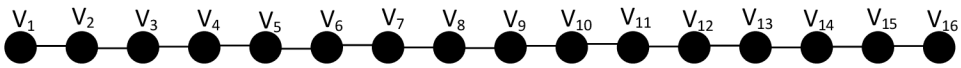
[1]Více informaci o těchto funkcích a jejich pokročile manipulaci na: `https://www.mathworks.com/help/matlab/ref/tic.html`



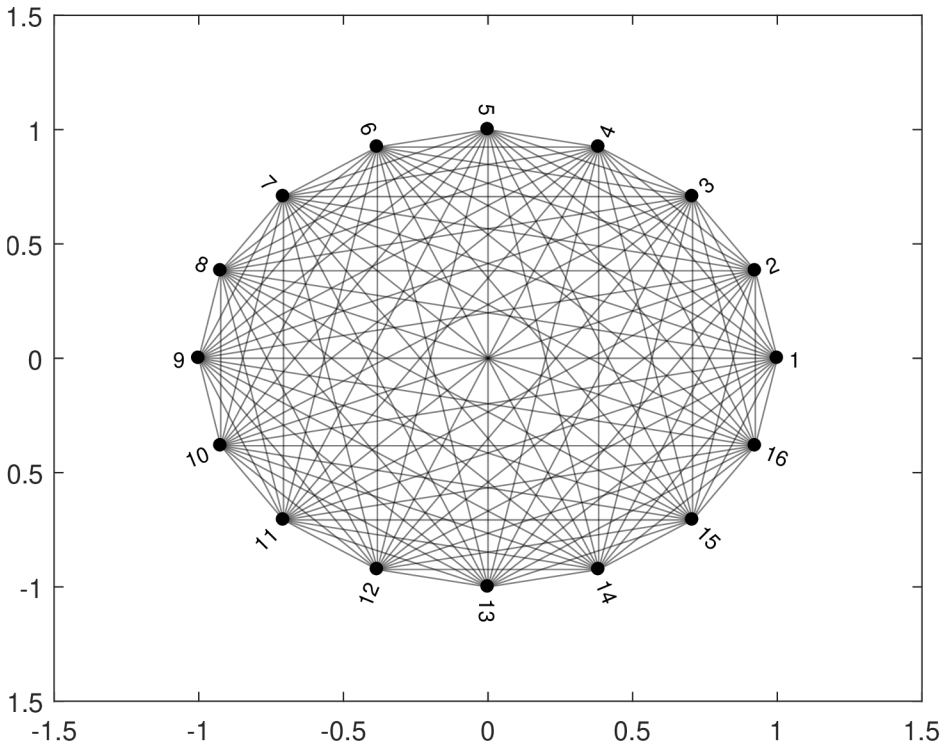
Obr. 5.2: Příklad topologie hvězda



Obr. 5.3: Příklad topologie strom

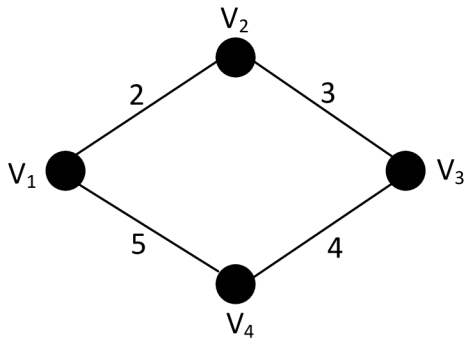


Obr. 5.4: Příklad slabě propojené topologie



Obr. 5.5: Příklad silně propojené topologie

Oba algoritmy popsané v kapitole 4.1 počítají s maticemi cen i když každý jiným způsobem. Takže dalším problémem, který se musel vyřešit, byl převod matice sousednosti na matice cen. Převod se provádí nahrazením prvku s hodnotou 1 prvkem s hodnotou nerovnající se 1, přitom matice musí byt 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, která je na obrázku 5.6 s cenami hran.



Obr. 5.6: Příklad jednoduché topologie se cenou hran

Matice sousednosti a následně matice cen bude:

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

První řešení by mohlo byt takové, že by se ručně přepsaly všechny matice sousednosti na matice cen, ale pro topologie, kde je velký počet vrcholů, je to zdlouhavé 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. 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 dane topologie:
A = ones(16) - diag([1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]);
```

```
% nahrazeni jednicek nahodilym cislem
% v intervalu 0 az 1 pro matici cen hran:
B = rand(size(A));
A(logical(A)) = B(logical(A));

%symetrizace spodniho trojuhelniku podle horniho:
C = triu(A)+triu(A,1)';
```

5.1.2 Výsledky porovnání obou algoritmů na klasických topologiích

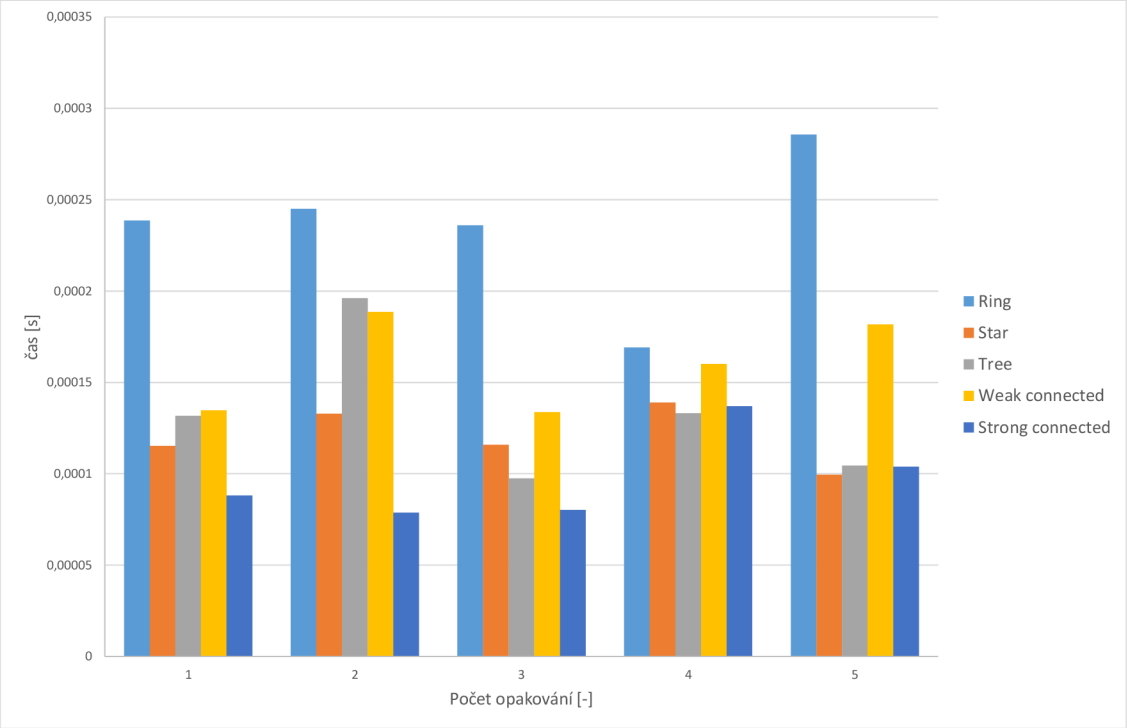
Po vytvoření topologií následovala samotná implementace algoritmů s jejich experimentálním testováním. Implementace algoritmů byla provedena v prostředí Matlab. Na každé topologií popsané v 5.1.1 byl jednotlivý algoritmus spouštěn pětkrát kvůli věrohodnosti výsledků, neboť jeden testovací průchod neposkytuje zcela směrodatná data pro hodnoceni algoritmu. Dále oběma algoritmy se hledala vždy cesta z uzlu 1 do uzlu 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 Excel pro další zpracování.

Jako první byl testován Bellmanův-Fordův algoritmus. Výsledky měření jsou v tabulce 5.1, 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 desetina místa - u původního výsledku v .excel souboru desetinných míst je více a čas je v sekundách.

Počet spouštění [-]	**Ring [ms]**	**Star [ms]**	**Tree [ms]**	**Weak connected [ms]**	**Strenght connected [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. 5.1: Bellmanův-Fordův algoritmus na odlišných typech topologii

Při každém spouštění byla generovaná pro každou topologii náhodná matice cen, ale vždy se hledala cesta od uzlu 1 k uzlu 16. Údaje měření Bellman-Fordová algoritmu z tabulky 5.1 byly zpracované do grafu 5.7. Z grafu je patrně vidět, že nejrychleji algoritmus konverguje u silně propojené topologie (Strong connected), což je vidět u druhého, třetího, čtvrtého a pátého spouštění. Naopak nejhůře konverguje u kruhové topologie (Ring). Výjimkou je poslední spouštění, kde nejpomalejší je stromová topologie (Tree). Důvodem proč algoritmus rychleji konverguje na silně propojené topologií je, že zde existuje více propojení s ostatními uzly, což naopak u kruhu takto není a následná komunikace z uzlu 1 do uzlu 16 musí u kruhu projít přes ostatní uzly na cestě k uzlu 16.



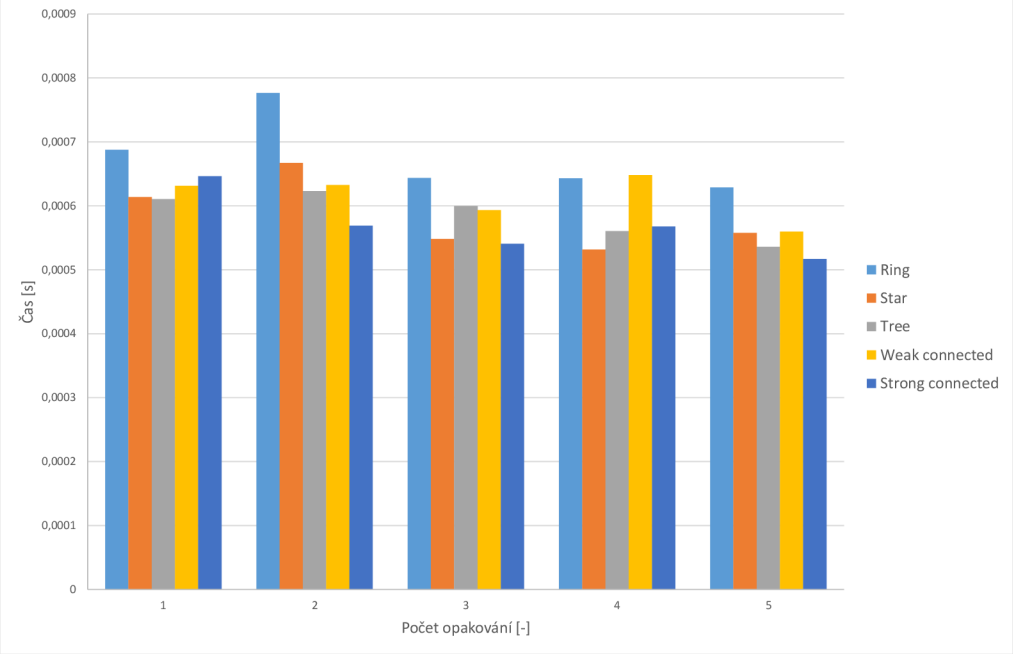
Obr. 5.7: Porovnání rychlosti dosažení konvergence Bellman-Ford algoritmu na různých topologiích

Druhým testovaným algoritmem byl Dijkstrův algoritmus, viz tabulka 5.2. Po zpracování i této tabulky do grafu 5.8, je vidět, že Dijkstrův algoritmus nejrychleji dosáhne konvergence na silně propojené topologií (Strong connected) a to u všech pěti spouštění. Nejhůře je to u kruhu (Ring). Opět vysvětlení je, že u silně propojené topologie existuje více hran mezi uzly a tak se rychleji sestaví kostra grafu. U kruhu existuje miň hran mezi uzly a uzly jsou v stanoveném pořadí s pouze dvěma hranami u každého (vstupní a výstupní), které vedou na stanovené sousedy a tak se musí

projít všechny uzly v kruhu na cestě z uzlu 1 do uzlu 16, což způsobí pomalejší sestavení kostry grafu a tím i pomalejší konvergenci.

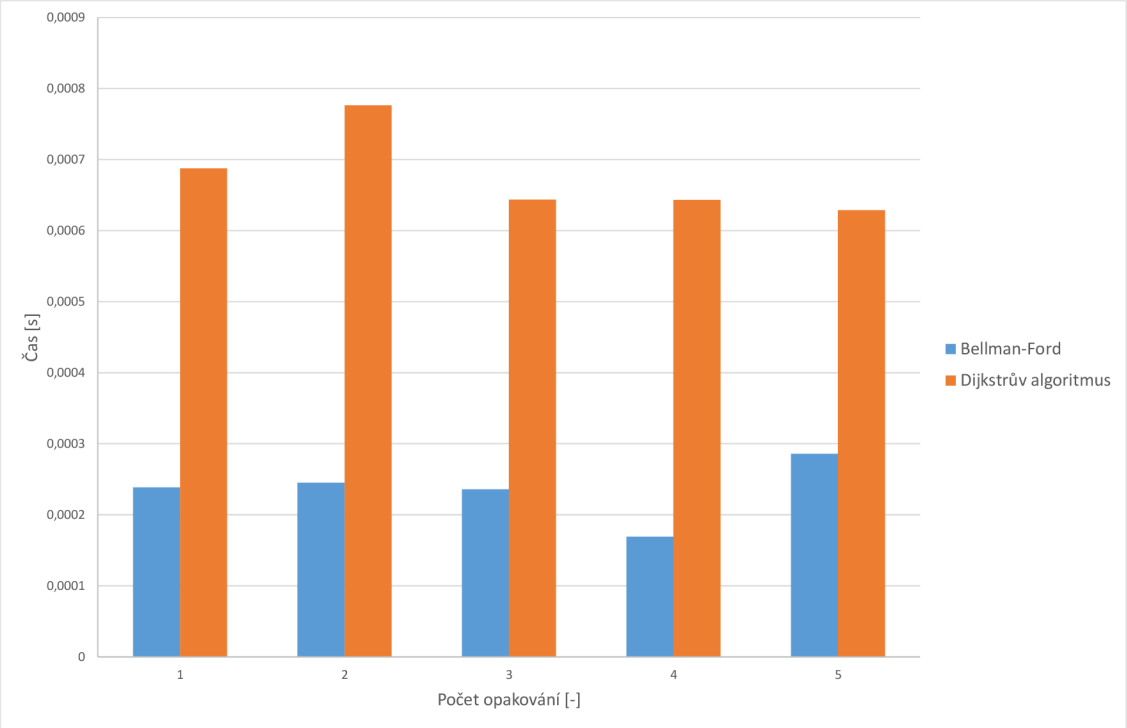
Počet spouštění [-]	**Ring [ms]**	**Star [ms]**	**Tree [ms]**	**Weak connected [ms]**	**Strenght connected [ms]**
1	0,688	0,614	0,610	0,631	0,646
2	0,776	0,667	0,623	0,632	0,569
3	0,644	0,548	0,599	0,593	0,541
4	0,643	0,532	0,561	0,648	0,568
5	0,629	0,558	0,536	0,559	0,517

Tab. 5.2: Dijkstrův algoritmus na odlišných typech topologii



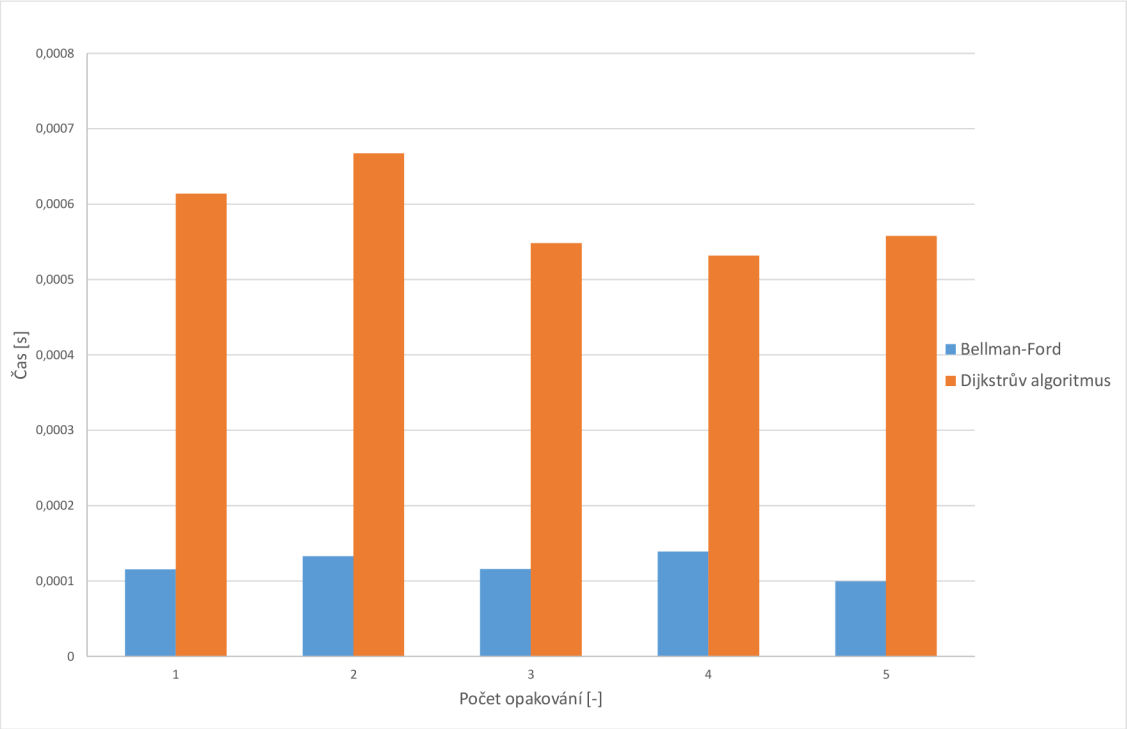
Obr. 5.8: Porovnání rychlosti dosažení konvergence Dijkstrová algoritmu na různých topologiích

Pro lepší porovnání a přehlednost jsou dále oba algoritmy porovnané zvlášť na každé topologii. Na grafu 5.9 je topologie kruh (Ring), graf 5.10 je hvězda (Star), graf 5.11 je strom (Tree), graf 5.12 je slabě propojená topologie (Weak connected) a graf 5.13 je silně propojená topologie (Strong connected).

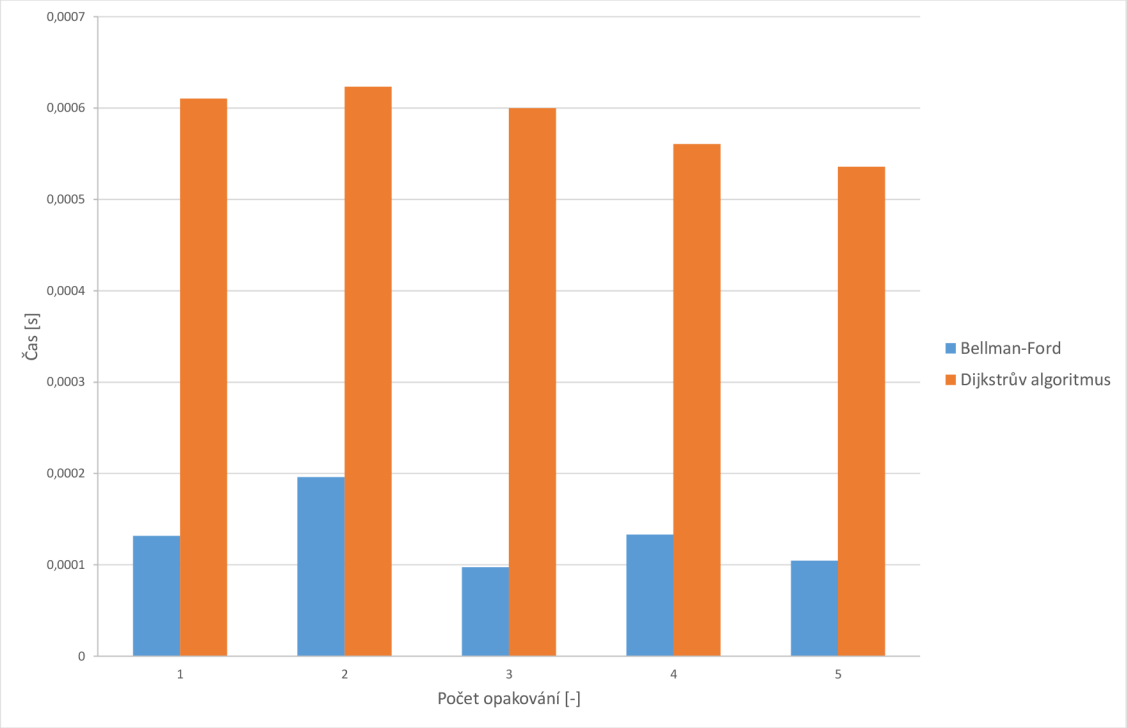


Obr. 5.9: Porovnání obou algoritmů na topologii kruh

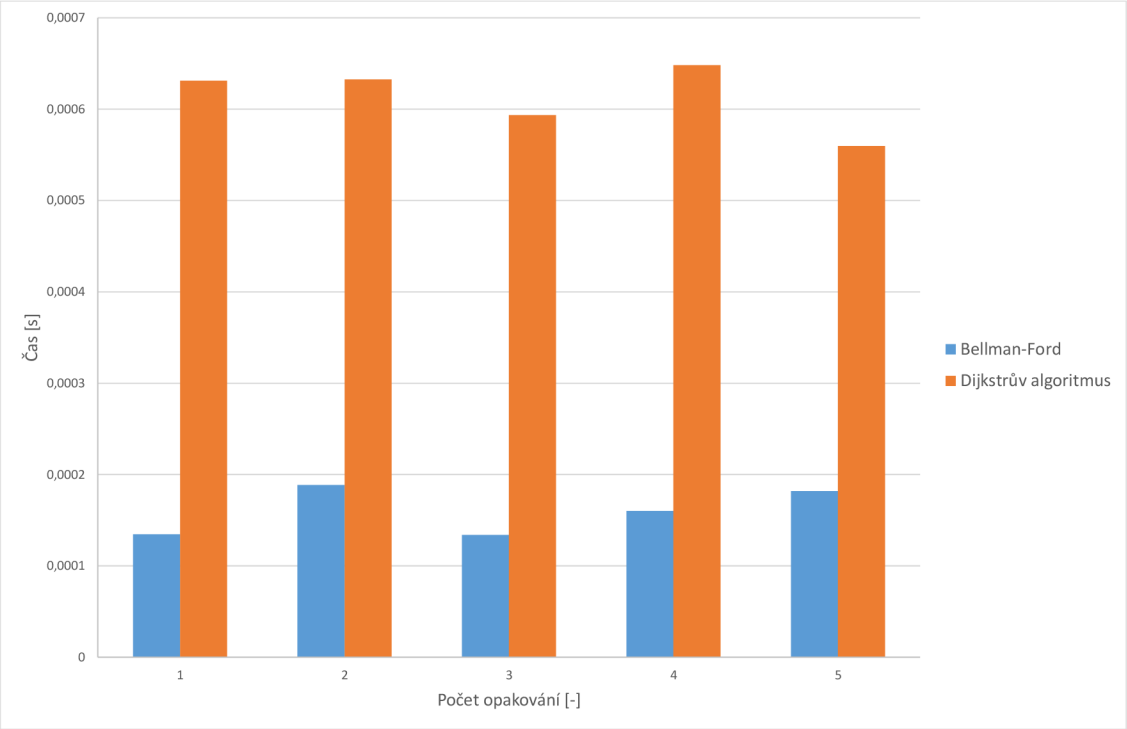
Při porovnání obou algoritmů na každé topologii zvlášť je vidět, že rychlejší algoritmus je Bellmanův-Fordův a to u všech topologií. Dijkstrův je vždy pomalejší. Při tomto porovnání se pořad jednalo o klasické topologie, které se moc v praxi nepoužívají a bylo by lepé algoritmy otestovat na větších a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



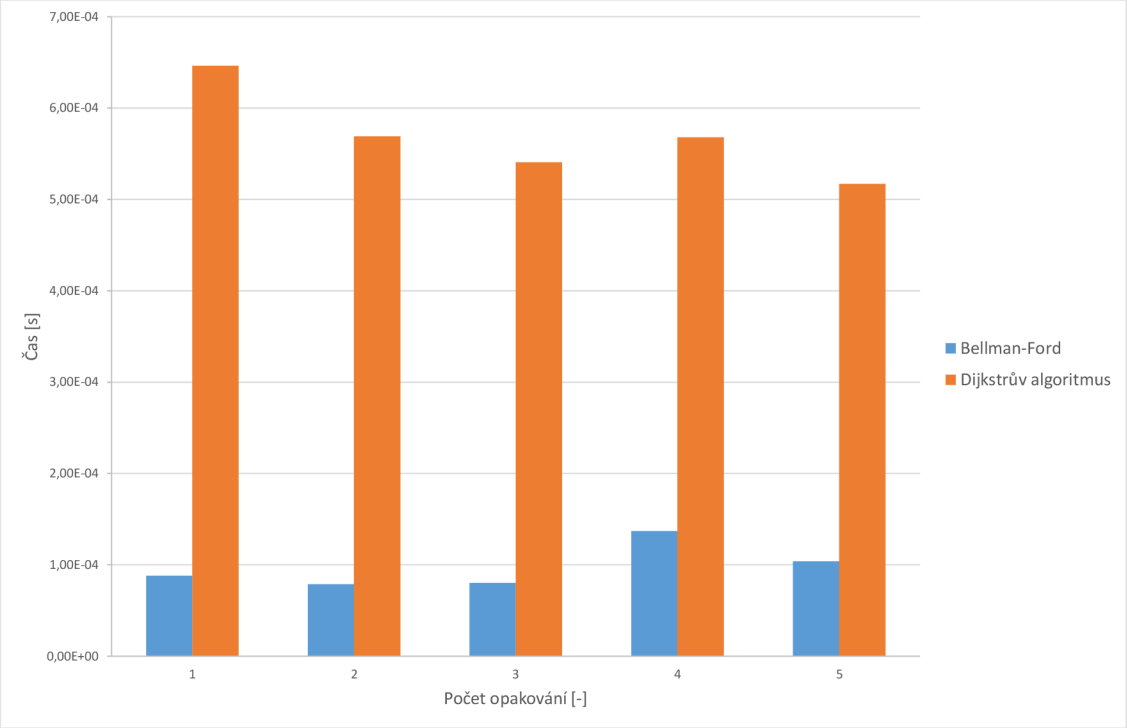
Obr. 5.10: Porovnání obou algoritmů na topologii hvězda



Obr. 5.11: Porovnání obou algoritmu na topologii strom



Obr. 5.12: Porovnání obou algoritmů na slabě propojené topologii



Obr. 5.13: Porovnání obou algoritmů na silně propojené topologii

5.1.3 Výsledky porovnávaní obou algoritmů na nahodilých topologiích

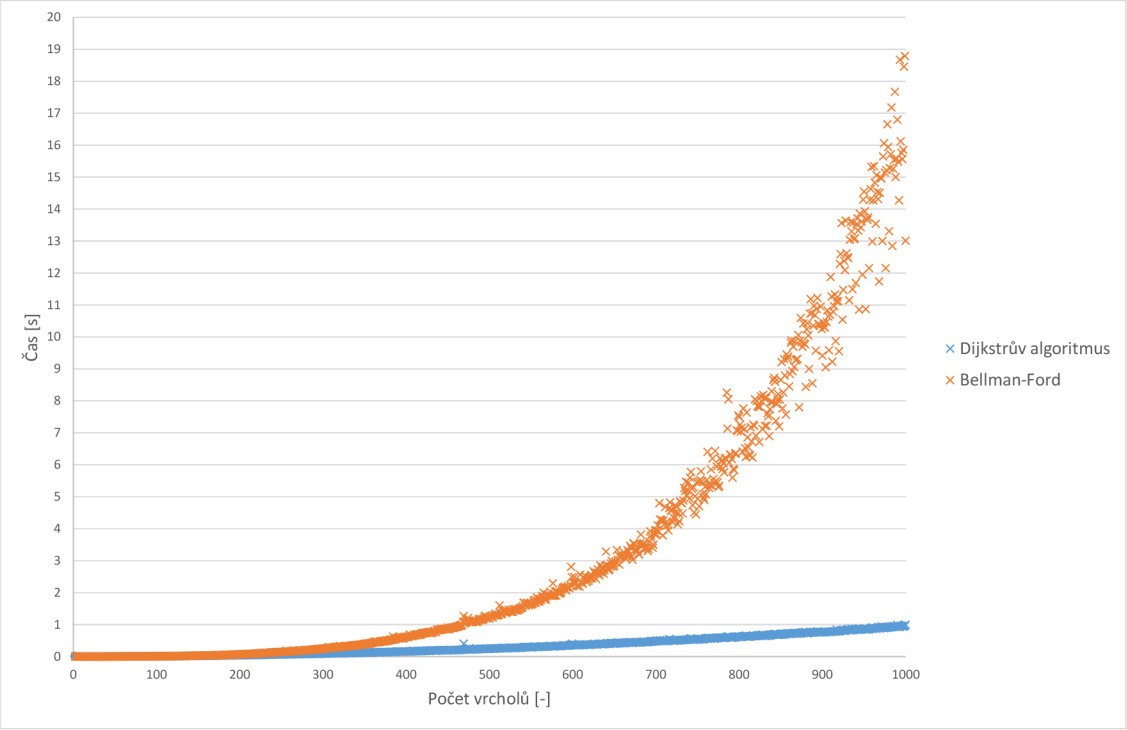
Podkapitola 5.1.2 popisovala výsledky konvergence na klasických topologiích s maximálním počtem vrcholů 16, ale bylo by zajímavé sledovat konvergenci na více reálnějších topologiích s větším počtem vrcholů než 16. Pro tento účel byla sestavená funkce generování náhodné matice cen, která musí byt symetrická podle diagonály, v diagonále musí byt nuly a nakonec matice musí mít nějaké procento prvků s hodnotou nula ve zbytku matice, protože neexistuje úplně propojené sítě ani sítě s velkým počtem funkčních uzlů a linek. Matice představuje nahodile topologie, kde velikost matice představuje počet uzlů (resp. vrcholů) v topologii. Ukázka takové funkce v prostředí Matlab:

```
function maticeCen = maticeCen (velikostMatice)

% generovani cisel v rozsahu 0 az 1:
maticeCen = rand(velikostMatice);
for y = 1:velikostMatice
    for x = y:velikostMatice

% podminka pro nulovou diagonalu:
        if (x==y)
            maticeCen(x,y) = 0;
        else

% navic 30% hodnot (mimo diagonalu) bude 0:
            if maticeCen(x,y)<=0.70
                maticeCen(x,y) = 0;
            end
        end
    end
end
```

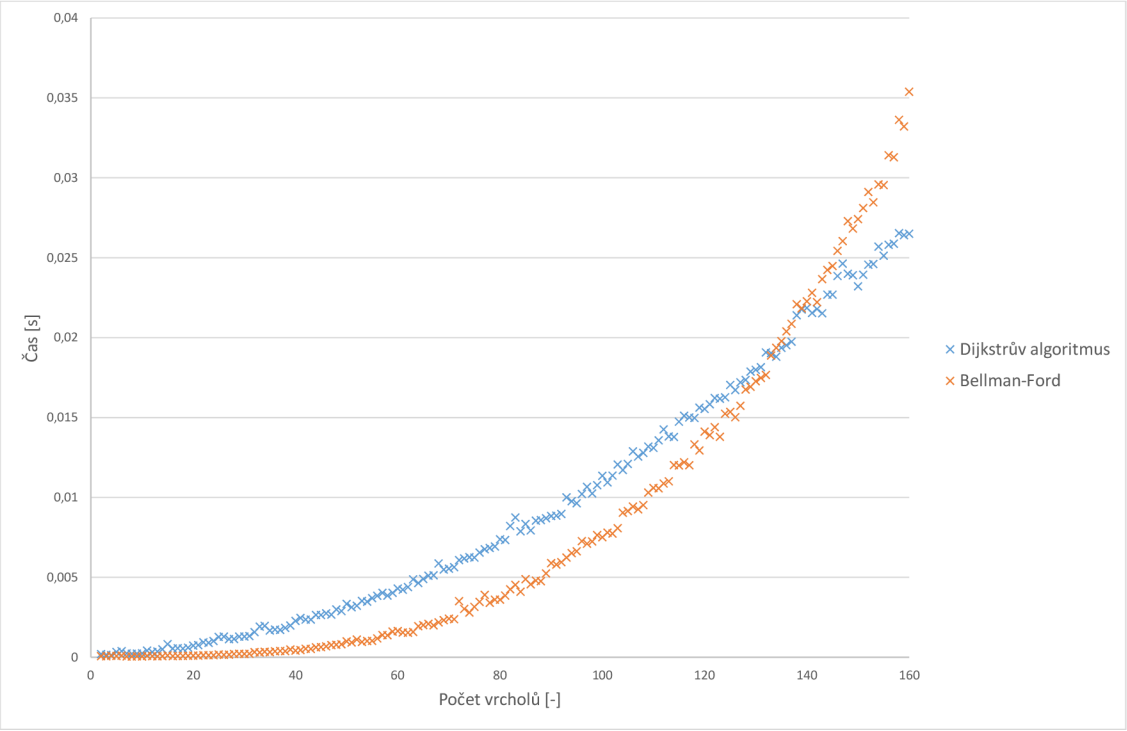
Každý algoritmus byl spouštěn třikrát a velikost matice cen byla generovaná od čtverce 2×2 až po čtverec 1000×1000. Výsledky tohoto testu představuje graf 5.14. V každé iteraci spouštění algoritmu a generovaní velikosti topologie se hledala cesta z uzlu 1 k uzlu n, kde n je velikost matice cen. Z důvodu podobnosti grafů všech tři spouštění je zde uveden jen jeden. Princip vkládaní času do proměnné a exportování do programu Excel je stejný jako v kapitole 5.1.2.

Tabulka naměřených času běhů algoritmů až po stav, kdy všichni ví o všech, pro svou velikost není uvedená, ale při pozornějším prohlédnutí byl pozorován zajímavý efekt: Bellmanův-Fordův algoritmus měl do určité velikosti matice (resp. velikosti grafu neboli také sítě) rychlejší konvergenci. Později začal 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 se stejným počtem spouštění pro oba algoritmy jako v minulém experimentu. Výsledek tohoto experimentu je v grafu 5.15.

Zajímavé je také vidět, že Dijkstrův algoritmus i když počítá s tisícem vrcholů, tak čas se drží okolo jedné sekundy, což je vidět na 5.14. Dále čas oproti Bellman-Fordovu algoritmu (kde čas roste exponenciálně) roste téměř lineárně s počtem vrcholů.



Obr. 5.14: Výsledky spouštění obou algoritmů na nahodilých topologiích



Obr. 5.15: Výsledky spouštění obou algoritmů na nahodilých topologiích s maximálním počtem vrcholů do 160

Ze všech grafů je patrné, že rychlejší algoritmus na klasických topologiích (podkapitola 5.1.2) je Bellmanův-Fordův. Stejný výsledek vyšel i na nahodilých topologiích do asi 135 uzlů (viz 5.15), kde je stálé rychlejší Bellmanův-Fordův. 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 (viz 5.14).

Hranice, kde ještě dominuje Bellmanův-Fordův algoritmus muže byt ovlivněna procesorem, tj. na lepším procesoru může byt hranice místo 135 vrcholů níže a na horším procesoru může byt hranice výše než 135 vrcholů, ale obecně platí, že efektivnější do určité velikosti topologie je Bellmanův-Fordův algoritmus. Tato skutečnost se testovala na dalším zařízení (počítači) s čtyřjádrovým procesorem Intel core i5 s taktem jednotlivého jádra 3,9 GHz, 16 GB operační paměti a operačním systémem Windows 10 Pro 64 bit. Zde byla hranice dominance Bellman-Fordová algoritmu okolo 130 vrcholů, což není velký rozdíl oproti 135.

5.2 Porovnání dvou stochastický algoritmů

V této podkapitole budou porovnané dva stochastické algoritmy, které byly popsané v podkapitole 4.2. 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 probíhá už rovnou na nahodilých topologiích, tj. měření na topologiích typu hvězda, strom, kruh, silně propojené a slabě propojené topologií jako v podkapitole 5.1.2 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 uzlů v sítí. Proto byla zvolená tzv. mřížková struktura.

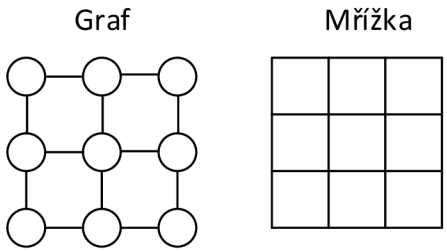
5.2.1 Implementace A*

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 obrázku 5.16. V podstatě se jedná o to, že jednotlivý uzel se převede na čtverec se stranami, kde jednotlivá strana určuje hranu, kterou měl původní uzel. Strana má určitou cenu při pohybu z daného čtverce ven do dalších čtverců. Uzly na obrázku 5.16 mají při převodu na mřížkovou strukturu čtyři strany pohybu - vlevo, dolu, vpravo, nahoru. Každý pohyb z uzlu (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. Zde se počítá jiným způsobem heuristika[2] 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 uzel je uspořádán do určité pozice v mřížce a má tak určitou pozici v kartézských souřadnicích. V této práci se vzdušná vzdálenost od aktuálního uzlu k cílovému uzlu počítá pomocí vzorce euklidovské vzdálenosti. Výpočet vypadá takto [13]:

$$vzdalenost = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{5.1}$$

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



Obr. 5.16: Ukázká rozdílu mezi grafem a mřížkou

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

Algoritmus A* by se mohl uplatnit i ve WSN (Wireless Sencor Network - bezdrátové senzorové sítě) sítích. Místo vzdálenosti by mohly uzly 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 uzlu 1 se souřadnicemi (1,1) k uzlu n se souřadnicemi (n, n), kde n představuje velikost mřížkové mapy. Konkrétně algoritmus běžel na topologiích (resp. mřížkových mapách) s velikosti od 2, tj. mřížková mapa o velikosti 2×2 až po velikost 1000, tj. mřížková mapa 1000×1000 a hledala se cesta z uzlu 1 k poslednímu uzlu v běhu dané topologie (resp. mřížkové mapy). Proto souřadnice koncového uzlu jsou (n, n). Ukázka této části v Matlab kódu:

```
for n = 2:1000
 MAX_X=n;
 MAX_Y=n;
 MAX_VAL=n;

 %pole si udrzuje koordinaty v mape od kazdeho objektu
 MAP=2*(ones(MAX_X,MAX_Y));

 %nastaveni cile a startovni pozice
 j=0;
 x_val = 1;
```

[3]Samotné jádro algoritmu A* pro Matlab prostředí bylo převzato z [1] a následně bylo upravené pro potřeby této práce.

```
 y_val = 1;

 xval=MAX_X;
 yval=MAX_Y;

 xTarget=xval; %X koordinaty cile
 yTarget=yval; %Y koordinaty cle

MAP(xval,yval)=0; %inicializuje mapu s koordinaty cile

 xval=1;
 yval=1;

 xStart=xval; %startovni pozice
 yStart=yval; %startovni pozice
MAP(xval,yval)=1;
```

Zbytek implementace je obdobný jako v kapitole porovnání dvou distribuovaných algoritmů. I zde se uplatnily nahodile ceny hran (resp. ceny pohybu 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 každé iterace a velikosti mřížkové mapy uložily do pole `vysledek`. Takto se měřil čas od začátku hledání cesty ze zdroje k cíli. Výsledky se exportovali z prostředí Matlab do Excel sešitu. Ukázka této části v kódu Matlab prostředí (zkráceno):

```
tic; %zacanek algoritmu a mereni casu

 ... %zde je telo samotnoho algoritmu
 ...
 ...
  cas = toc;
 end % konec algoritmu a mereni casu
 n % velikost topologie
 omega %promenna iterace spousteni
vysledek(n,3*omega-1) = n;
vysledek(n,3*omega) = cas;
```

5.2.2 Implementace push-sum algoritmu

Algoritmus push-sum se zařadil k měření kvůli své vlastnosti komunikační efektivity a vysoké stabilitě při rušení. Takže by se dal vyžít například při rozesílání směrovacích informaci nebo informaci o jednotlivých uzlech a tuto informaci by následně šlo využít pro sestavení např. nejlepší cesty mezi dvěma uzly. Stálé se však jedna o experimentální algoritmus, který ještě nebyl praktický nasazen do žádného standardu ani technologie, proto by bylo zajímavé zjistit jak rychlé zkonverguje, tj. rozešle informaci všem uzlům v dáne topologií.

Implementace algoritmu proběhla podle [12] v prostředí Matlab. Měření času konvergence je obdobné jako u ostatních algoritmů (stejně 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 uzel 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 uzlu, který tuto informaci odeslal. Všechny uzly následně vypočítají odhad průměru, který se počítá poměrem vnitřního stavu a váhy. Tento postup se opakuje dokud systém nedosáhne součinnosti, kterou se rozumí rozdíl mezi maximální a minimální hodnotou v rámci celé sítě (topologie). Tento rozdíl má byt menší 0,00015 dle [12]. 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 uzlům 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 by se mohl uplatnit například při přenosu směrovacích informaci nebo jiných informaci v sítích, proto v této práci byl vybrán. Ukázka kódu implementace algoritmu dle [12] v Matlab prostředí:

```
k = 1;
rozdil = 0.00015;

% zacatek algoritmu a zacatek mereni casu
tic;
 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;
end % konec algoritmu a mereni casu
```

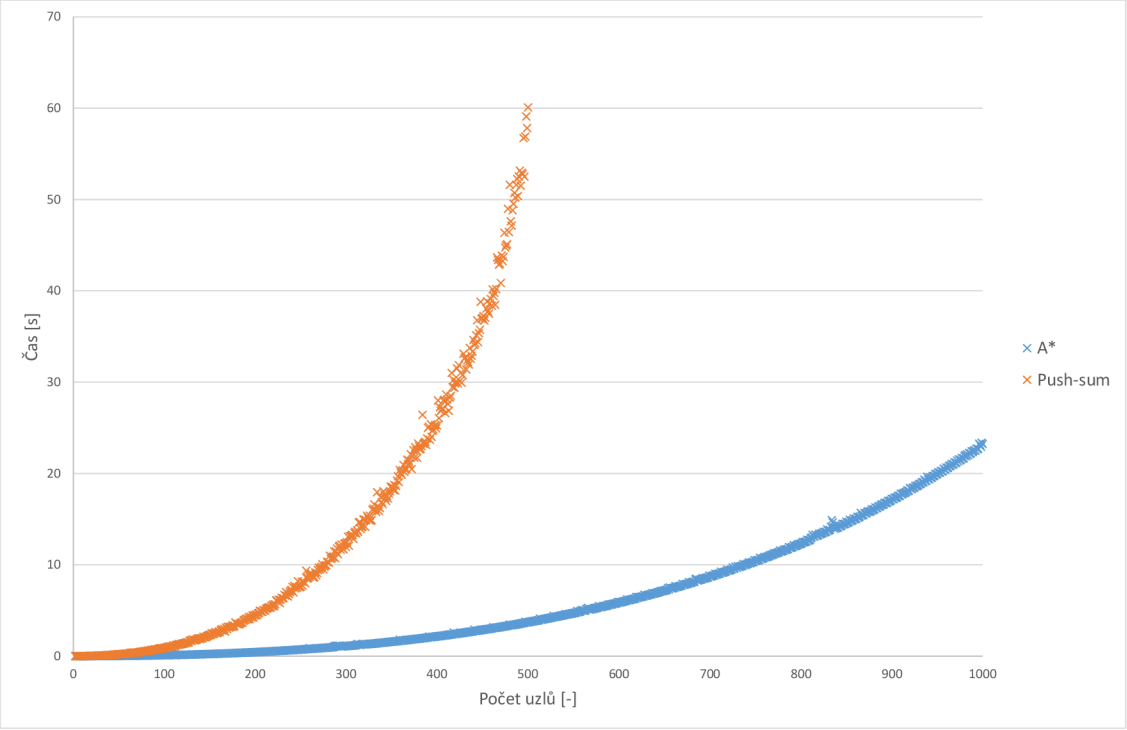
Pro měření času konvergence bylo zapotřebí vytvořit nahodile topologie, které tentokrát byly opět reprezentované maticemi sousednosti. Velikost matic byla zvolena od $n = 2$ až $n = 1000$. Dále matice musí mít alespoň 30% nulových prvků, protože toto odpovídá reálným sítím, jelikož nikdy neexistuje kompletně propojena síť bez výpadku linek nebo uzlů. Dále diagonála musí byt nulová, protože uzel nemůže byt propojený sám se sebou a tak komunikovat sám se sebou. Nakonec matice musí byt symetrická, protože se jedna o neorientované grafy, které odpovídají reálným linkám, kde se komunikuje obousměrně. Ukázka této části kódu v Matlab prostředí:

```
for n = 2:1000
 a = n; b = n;
 A = rand(a,b) < 0.7; % 30% matice bude 0,
                      % coz odpovida realnejsim sitim
```

```
B = triu(A)+triu(A,1)'; % symetrizace spodniho
                        % trojuhelniku podle horniho
B(1:n+1:n*n) = 0; % nastaveni diagonaly na 0
```

5.2.3 Výsledky porovnání

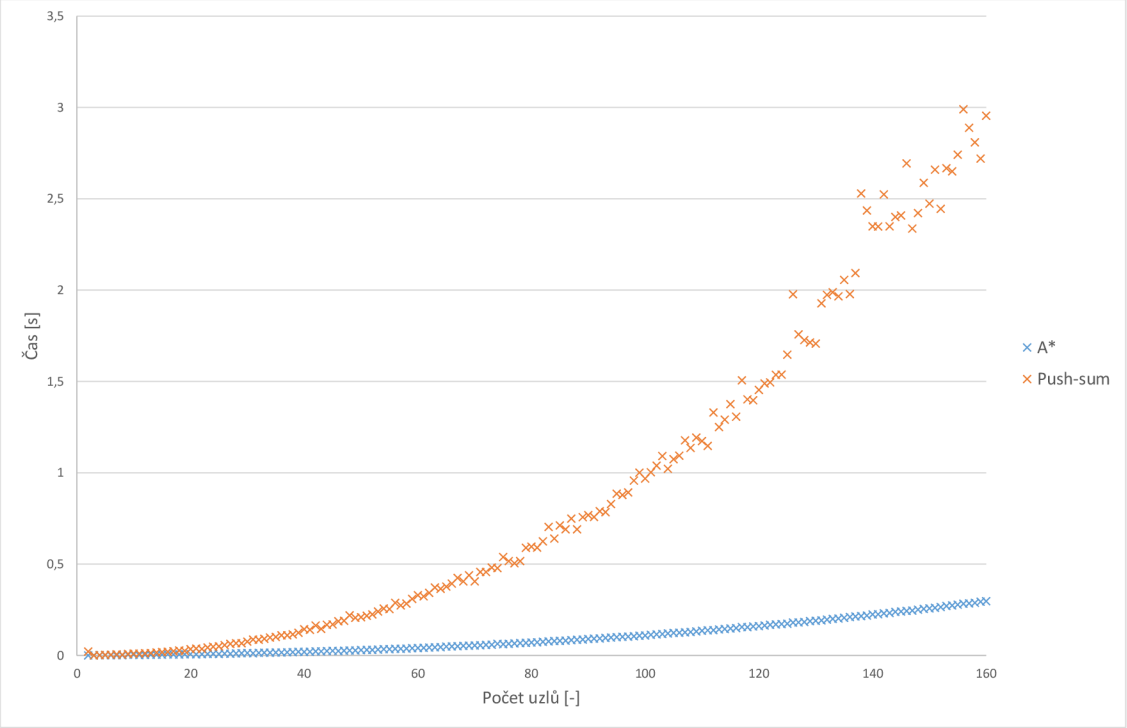
Po osvědčeném měření času konvergence na nahodilých topologiích v kapitole porovnání distribuovaných algoritmů i zde se nejdříve měřilo na velikosti topologií do 1000 uzlů a pak do 160 uzlů. Každá velikost se měřila třikrát. Grafy si jsou podobné a tak je v práci vždy zobrazena pouze jedna ukázka. Tabulky zde nejsou uvedené z důvodů rozsáhlosti naměřených hodnot.



Obr. 5.17: Porovnaní obou algoritmů s maximálním počtem uzlů do 1000

Na obrázku 5.17 je vidět, že jednoznačně rychlejší konvergenci má algoritmus A*. Důvodem může byt, že A* neprochází všechny uzly v topologií oproti tomu push-sum sděluje svoje hodnoty ostatním uzlům dokud tuto informaci neví všechny uzly v topologií a není tak dosaženo součinnosti v síti (více o činnosti algoritmu v subkapitole 5.2.2). Konkretněji A* vždy vybere nejvhodnější uzel k cíli a z tohoto uzlu pokračuje dál. Push-sum rozesílá informace všem uzlům dokud se nedosáhne součinnosti, což vyžaduje všechny uzly sítě.

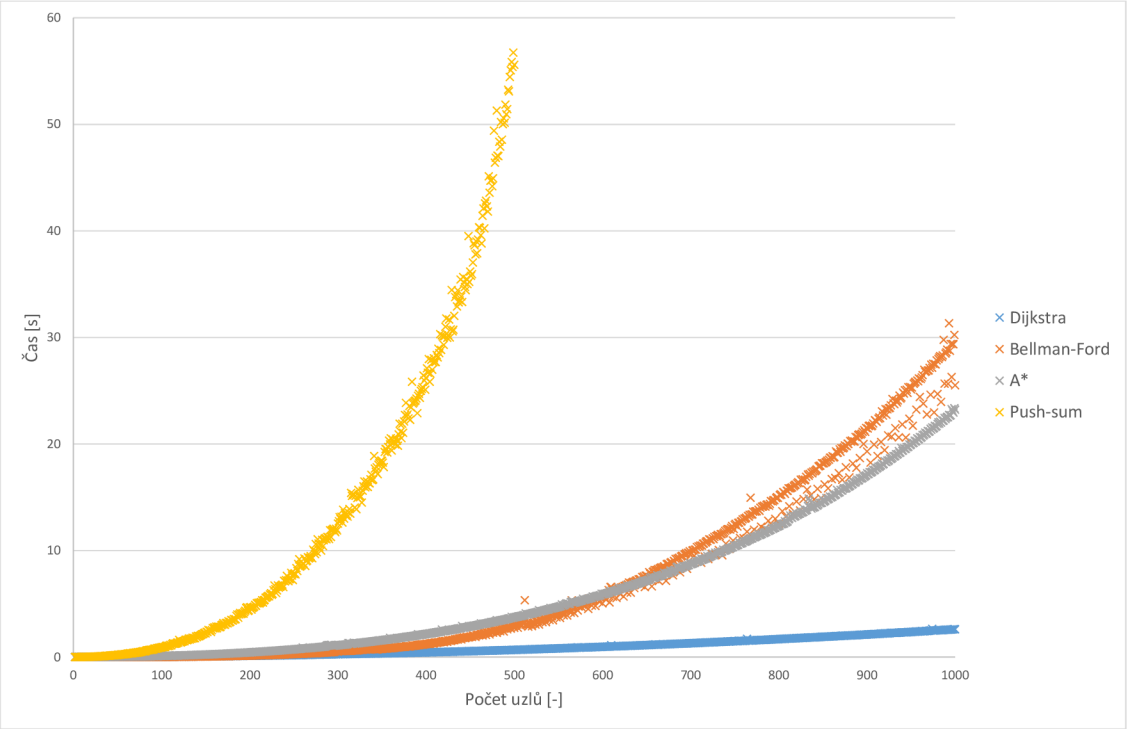
Na obrázku 5.18 je porovnání obou algoritmů do 160 uzlů. I zde výrazně zaostává algoritmus push-sum. Přibližně stejnou rychlost mají do počtu uzlů okolo 20.



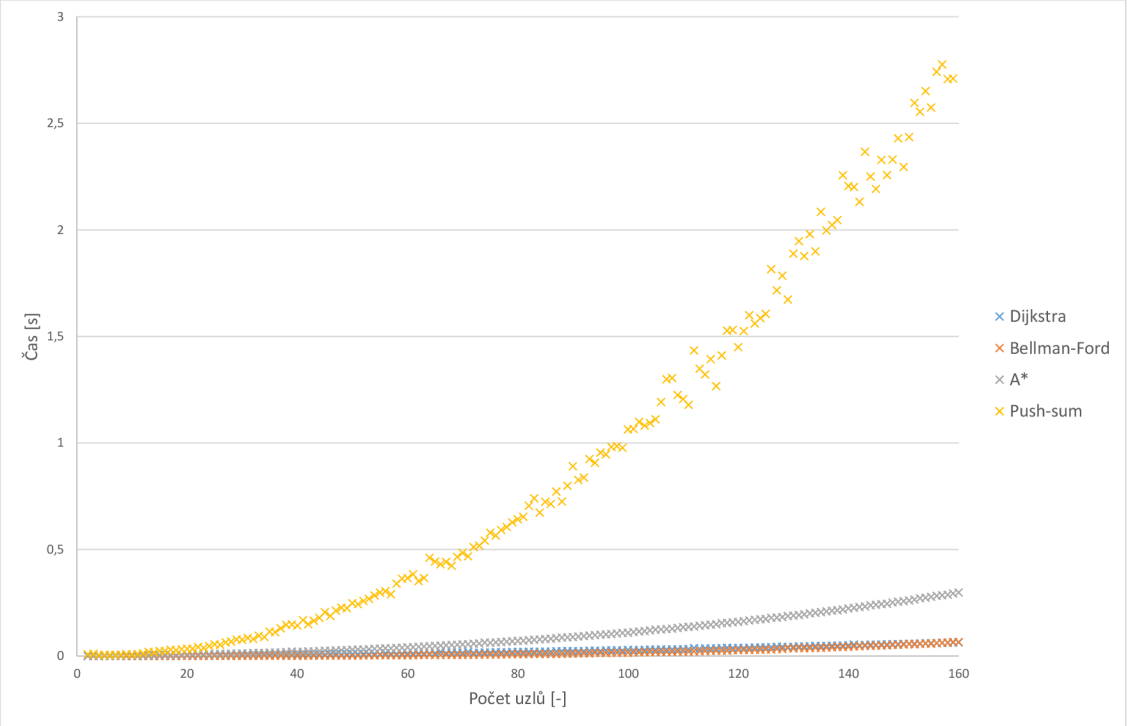
Obr. 5.18: Porovnaní obou algoritmů s maximálním počtem uzlů do 160

5.3 Porovnání všech algoritmů

Poslední částí práce byla věnovaná porovnaní všech algoritmů mezi sebou, což by mělo ukázat, který algoritmus je nejefektivnější. Na obrázku 5.19 je vidět, že nejrychlejší algoritmus je Dijsktrův a Bellmanův-Fordův, ale jak se bylo zjištěno u kapitoly porovnání dvou distribuovaných algoritmů, tak do určité velikosti sítě je rychlejší Bellmanův-Fordův. Dobré výsledky má také algoritmus A*, který ale zaostává za dvěma výše popsanými distribuovanými algoritmy. Horší výsledky u A* můžou byt způsobené výpočtem heuristiky u každého uzlu, kdy se nejdříve zpracují všechny do aktuálního okamžiku známé údaje (ceny, heuristika) a teprv na základě výpočtů z těchto informací se algoritmus rozhodné přes který uzel pokračovat. Nejhůře dopadl algoritmus push-sum, kdy u počtu uzlů 500 už je čas na konvergenci okolo 57 sekund. Důvodem špatných výsledků push-sum algoritmu může byt to, že než zkonverguje, tak musí rozeslat svoje informace všem uzlům v dané topologií a neděla to tak efektivně jako Dijkstrův algoritmus, který si sestavuje pro každý uzel kostru grafu.



Obr. 5.19: Porovnaní všech algoritmů s maximálním počtem uzlů do 1000



Obr. 5.20: Porovnaní všech algoritmů s maximálním počtem uzlů do 160

Pro upřesnění výsledků rychlosti konvergence je tu porovnání všech algoritmů na velikosti topologií s maximálním počtem uzlů do 160, viz 5.20. I zde jsou nejlepší 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 byt ovlivněné vhodně napsaným kódem, který může byt efektivnější než v této práci a dále rychlosti můžou byt ovlivněné pokročilejšími počítači s lepším hardwarem a procesory než na kterých běžely experimenty z této práce. I když v podkapitole 5.1.3 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 různých zařízeních a optimalizace kódů je už nad rámec cílů této práce.

6 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 pochopení principů algoritmů 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 s velikosti topologií v podobě matic sousednosti a matic cen od dva krát dva až tisíc krát tisíc uzlů. Zde se zjistilo, že na všech klasických topologiích a na nahodilých topologiích do velikosti 140 uzlů je rychlejší, tj. rychleji konverguje Bellman-Fordův algoritmus, který se v praxi používá u směrovacího protokolu RIP. Od velikosti větší než 140 uzlů 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 dva krát dva až po tisíc krát tisíc. Výsledky porovnaní v této kapitole ukázaly, že rychleji konverguje algoritmus A* a Push-sum už u topologie o velikosti 500×500 uzlů 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 uzlů 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. Push-sum se hodí podle výsledků pouze na specifické zasílání informaci místo směrovacích informaci pro každý uzel, např. ne pro směrovací protokol.

LITERATURA

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

[2] 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>.

[3] ZILL, D., Wright, W. S., & Cullen, M. R. *Advenced engineering mathematics,* 4th edition. Jones & Bartlett learning, c2011, 1020 s. ISBN: 978-0763779665.

[4] 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>.

[5] 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/>.

[6] 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/>.

[7] 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/>.

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

[9] BAPAT, Ravindra B. *Graphs and matrices.* New Yourk: Springer, 2010, 175 s. ISBN: 978-1-84882-981-7.

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

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

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

[13] 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>.

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

[15] 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.

[16] 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.

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

[18] 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>.

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

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

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

A	matice sousednosti
A*	A star algorithm
IPC	Interprocess Communication Method
OSPF	Open Shortest Path First
RIP	Routing Information Protocol

SEZNAM PŘÍLOH

A Obsah přiloženého DVD **73**

A OBSAH PŘILOŽENÉHO DVD

Přiložené DVD obsahuje elektronickou verzi práce ve formátu „PDF“, soubory s výsledky práce ve formátu uvEXCEL a soubory pro práci v prostředí Matlab, testované ve verzi 2015. 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“)