VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

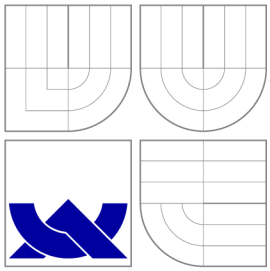
GRAFICKÉ ZNÁZORNĚNÍ SMĚROVACÍCH ALGORITMŮ

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

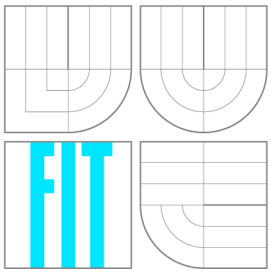
AUTOR PRÁCE
AUTHOR

MARTIN POKORNÝ

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

GRAFICKÉ ZNÁZORNĚNÍ SMĚROVACÍCH ALGORITMŮ

GRAPHICAL VIZUALIZATION OF ROUTING ALGORITHMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MARTIN POKORNÝ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JIŘÍ JAROŠ

BRNO 2009

Abstrakt

Cílem této bakalářské práce je znázornit síť, např. počítačů či procesorů, a komunikaci v této síti. Síť může být znázorněna jako různé topologie, které je možné ručně upravit. Rozestavění a propojení uzlů je uloženo v dodaném souboru. Dalším úkolem programu je na zobrazené topologii znázornit skupinovou komunikaci mezi uzly. Druh skupinové komunikace je závislý na zvoleném směrovacím algoritmu, který je opět uložen v dodaném souboru. Výstup programu lze uložit ve formě rastrového obrázku či dokumentu XML.

Abstract

The aim of this bachelor's thesis is illustrate network topology, e.g . computers or processor, and communication in this network. The network can be displayed like a various topology, which is possible to manually modify. Position and interconnection of vertex is stored in delivered file. Second part of program is on displayed topology demonstrate collective communication between vertex. The kind of collective communication depend on selected routing algorithm, which is again stored in delivered file. Program output can be saved in the form of raster picture or XML document.

Klíčová slova

Teorie grafů, kreslení grafu, pružinový algoritmus, skupinová komunikace, směrovací algoritmus.

Keywords

Graph theory, graph drawing, force-directed placement, collective communication, routing algorithm.

Citace

Martin Pokorný: Grafické znázornění směrovacích algoritmů, bakalářská práce, Brno, FIT VUT v Brně, 2009

Grafické znázornění směrovacích algoritmů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jiřího Jaroše.

.....
Martin Pokorný
20. května 2009

Poděkování

Chtěl bych poděkovat vedoucímu mé práce a všem svým příbuzným a blízkým, za jejich pomoc, neustálou podporu a víru v mé úspěšné dokončení této práce.

Obsah

Kapitola 1

Úvod

Cílem bakalářské práce je vhodným způsobem znázornit digram grafu, který znázorňuje topologii počítačové sítě, tedy zobrazit uzly v této síti a vztahy mezi nimi. Seznam uzlů a jejich vztahů je uložen v souborech, které byli dodány vedoucím této práce. Znázornění digramu sítě není tak jednoduché, jak by se mohlo na první pohled zdát, proto je jedním z dílčích úkolů programu upravit diagram do přijatelného tvaru. Může se stát, že digram vytvořený programem nebude vyhovovat našim kritériím, proto je zde zahrnuta možnost ručního upravení diagramu. Druhou neméně důležitou funkcí programu je, na zvolené topologii znázornit skupinovou komunikaci mezi uzly, která se liší v závislosti na zvoleném směrovacím algoritmu. Samotný výpočet skupinové komunikace není součástí programu. Pravidla a pořadí komunikace uzlů v síti jsou také dodány v souborech, které jsou výstupem jiného programu.

Protože se v našem programu znázorňujeme síť jako diagram grafu, tak jsou v první kapitole uvedeny některé důležité vztahy z oblasti teorie grafů. Je zde uvedeno, co je to vlastně graf, jak se zobrazuje či možné principy uchování jeho reprezentace v paměti počítačů.

V druhé kapitole jsou popsány některé z nejběžnějších topologií. Dále je zde popsán problém paralelních algoritmů. Tyto algoritmy využívají skupinovou komunikaci pro komunikaci mezi procesy. Skupinová komunikace využívá více přístupů, které jsou zde taky stručně rozebrány.

Třetí kapitola se zaměřuje na problém přehledného rozvržení grafu. Aby byl graf pro lidský mozek srozumitelný, musí splňovat určitá kritéria, které jsou zde uvedeny. Jsou zde popsány algoritmy, které se pro rozvržení grafu používají v programu.

Ve čtvrté kapitole je popsán postup analýzy dodaných vstupních souborů a návrh programu. Na ukázkách vstupních souborů je vysvětlena jejich struktura a význam. Návrh obsahuje popis vizuální a funkcionální stránky programu.

Popis implementace programu je popsán v páté kapitole. Postupně jsou zde rozebrány třídy a na příkladech vysvětleny použité metody programování.

Poslední kapitola obsahuje závěr, dosažené výsledky, přínos bakalářské práce a případné možnosti vylepšení programu.

Kapitola 2

Definice a důležité vztahy

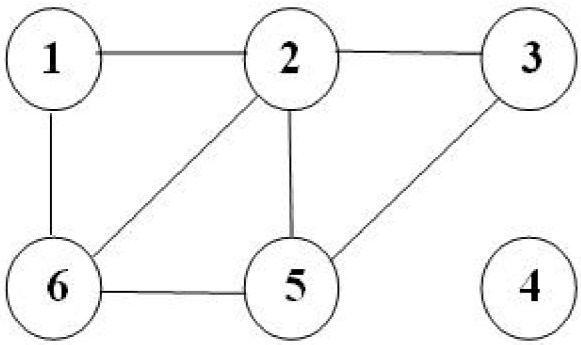
Dříve, než se budeme zabývat samotným zobrazením směrových algoritmů, je potřeba přehledným způsobem znázornit strukturu sítě. Jelikož o síti budeme dále mluvit jako o grafu, je potřebné se seznámit se základními pojmy z teorie grafů a vysvětlit vztahy a metody, které budou využity v následujících kapitolách.

2.1 Graf

Pod pojmem graf si asi většina z nás představí sloupcový či koláčový graf využívaný ve statistice , ale tyto typy grafů zde popisovat nebudeme. Budeme používat graf jako vhodný prostředek pro popis situací, v našem případě pro znázornění nějaké sítě, kterou lze popsat pomocí konečného množství bodů (dále je budeme nazývat vrcholy V) a vztahů mezi nimi (dále značeny jako hrany H).

Definice 2.1.1 **Grafem** nazveme uspořádanou dvojici $G = (V, H)$, kde V je neprázdná konečná množina vrcholů grafu a H je množina neuspořádaných dvojic typu $\{u, v\}$, kde $u \neq v$, t.j.

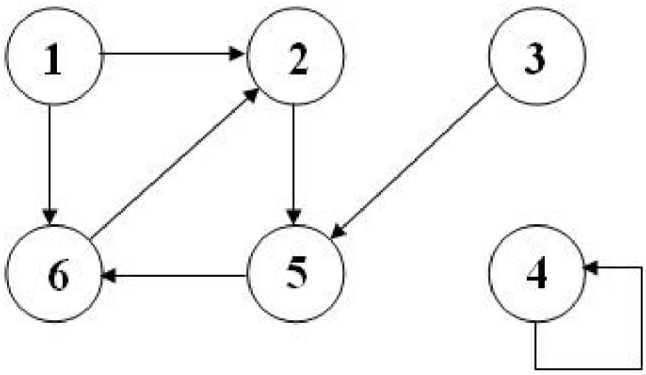
$$H \subseteq \{\{u, v\} | u \neq v, u, v \in V\} = V \circ V \tag{2.1}$$



Obrázek 2.1: Příklad diagramu grafu

Definice 2.1.2 **Digrafem** nazveme uspořádanou dvojici $G = (V, H)$, kde V je neprázdná konečná množina vrcholů grafu a H je množina uspořádaných dvojic typu (u, v), kde $u \neq v$, t.j.

$$H \subseteq \{\{u, v\} | u \neq v, u, v \in V\} = V \times V \tag{2.2}$$



Obrázek 2.2: Příklad diagramu digrafu

Graf a digraf jsou nejjednodušší grafové struktury, ve kterých nejsou dovolené hrany typu $\{u, v\}$ resp. (u, v). V grafu resp. digrafu můžeme pro každou dvojici u,v ve V existovat nejvýše jenda hrana typu $\{u, v\}$ resp. (u, v). Poznamenejme, že v grafu je hrana $\{u, v\}$ totožná s hranou $\{v, u\}$, ale v digrafu (u, v) a (v, u) jsou různé hrany.

2.2 Zobrazení grafu

V teorii grafů je nutné rozlišovat rozdíl mezi grafem a diagramem grafu. Graf je dvojice množin vrcholů a hran, diagram grafu je "obrázek", který jistým způsobem koresponduje s příslušným grafem. K diagramu grafu existuje vždy jen jeden graf, ale k jednomu grafu je možné nakreslit libovolné množství různých diagramů, u kterých na první pohled nemusí být zřejmé, že jsou diagramem stejného grafu.

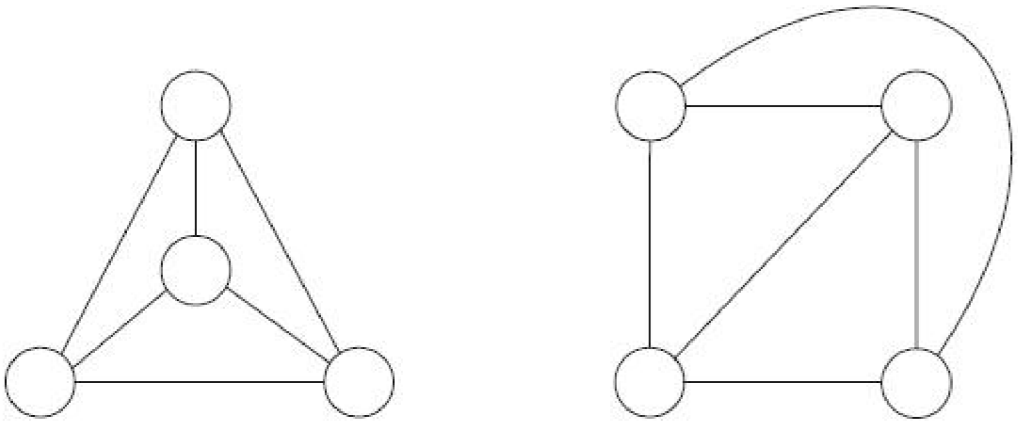
Definice 2.2.1 Graf často reprezentujeme graficky a příslušný obrázek nazýváme diagram grafu. **Diagram grafu** $G = (V, H)$ v nějakém prostoru $\mathcal{P}$ je množina bodů B a množina souvislých čar v prostoru $\mathcal{P}$ takových, že

- Každému vrcholu $v \in V$ zodpovídá právě jeden bod $x_v \in B$, přičemž pro $u, v \in V, u \neq v$ je $x_u \neq x_v$.
- Každé hraně $h \in H$ zodpovídá právě jedna čára $s_h \in S$, přičemž pro $h, k \in H, h \neq k$ je $s_h \neq s_k$.
- Nechť $h = \{u, v\} \in H$, potom čára s_h má koncové body x_u, x_v. Kromě koncových bodu žádná čára neobsahuje bod typu $x_w \in B$.
- Navíc se často žádá, aby byl diagram nakreslený tak, že žádná čára sama sebe neprotíná a dvě čáry mají nejvýše jeden průsečík.

Velmi často se za prostor P bere rovina. Zkoumají se však i diagramy grafů např. v trojrozměrném prostoru, na kulové ploše či anuloidě. Podobně jako diagram grafu lze definovat diagram digrafu, pokud namísto čar použijeme orientované čáry resp. šipky.

Jelikož se budeme v dalších kapitolách zabývat vykreslením grafu v rovině, zavedeme zde ještě pojem rovinný graf resp. digraf, někdy se také nazývá planární graf (digraf).

Definice 2.2.2 Diagram grafu resp. digrafu v rovině nazveme **rovinný**, pokud se jeho hrany neprotínají nikde jinde kromě vrcholů. Graf (digraf) $G = (V, H)$ nazveme rovinný, pokud k němu existuje rovinný diagram.



Obrázek 2.3: Příklad rovinného diagramu grafu

2.3 Reprezentace grafu

Pro reprezentaci grafů a digrafů je možné využít vice způsobů. Některé jsou vhodné pro ilustrování některých pojmů a postupů, jiné jsou vhodné pro ukládání grafových struktur v paměti počítačů. Různé způsoby reprezentace jsou různě náročné na pamět, ale i na přístupovou dobu k prvku struktury. Volba způsobu uložení grafu nebo digrafu většinou závisí na způsobu práce algoritmu, pro který budeme tyto data uchovávat. Nyní se podívejme na některé nejčastější způsoby reprezentace grafů a digrafů.

2.3.1 Diagram grafu

Diagramy grafů jsou velmi vhodné pro ilustraci vlastností grafů a digrafů, avšak s rostoucím počtem hran a vrcholů přestávají být přehledné. Pro ukládání v paměti počítače je tento způsob značně nevhodný, zvláště pokud budeme chtít tyto data ještě v budoucnu zpracovávat nějakým algoritmem.

2.3.2 Seznam vrcholů a hran resp. orientovaných hran

Seznam vrcholů a hran resp. orientovaných hran je již vhodnější způsob pro použití v informatice. Zde množinu vrcholů V reprezentujeme jako jednorozměrné pole V s $n = |V|$ prvkami, kde $V[i]$ i-tý vrchol. Množinu hran uložíme do dvojrozměrného pole H typu $(m \times 2)$,

kde $m = |H|$ je počet hran, $H[j,1]$ je počáteční a $H[j,2]$ koncový vrchol j-té hrany, čímž je dána i orientace hrany v případě digrafu. V případě grafu nezáleží na pořadí vrcholů $H[j,1]$, $H[j,2]$.

2.3.3 Matice sousedností

Matice sousedností je také vhodná pro reprezentaci grafu resp. digrafu v paměti počítačů. Matice sousedností $M = (m_{ij})$ je čtvercová matice typu $n \times n$, kde $n = |V|$ je počet vrcholů grafu resp. digrafu G, jejíž prvky jsou definovány následovně:

$$m_{ij} = \begin{cases} 1, & kdyz\{i,j\} \in H \\ 0, & jinak \end{cases}$$

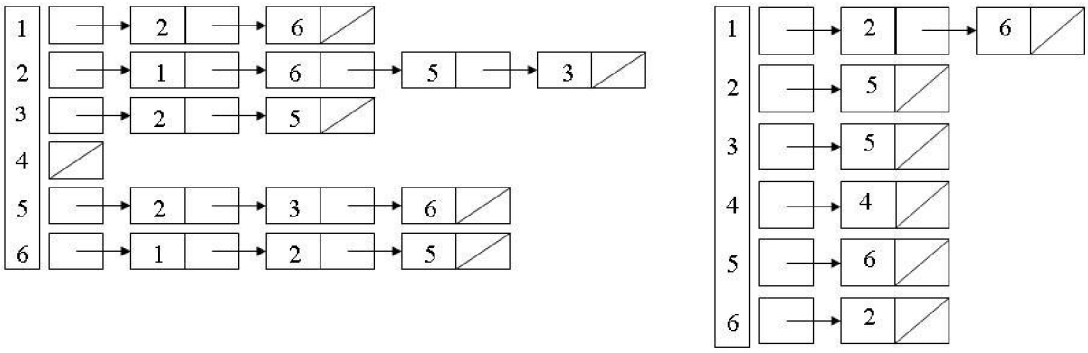
	1	2	3	4	5	6
1	0	1	0	0	0	1
2	1	0	1	0	1	1
3	0	1	0	0	1	0
4	0	0	0	0	0	0
5	0	1	1	0	0	1
6	1	1	0	0	1	0

	1	2	3	4	5	6
1	0	1	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	0
4	0	0	0	1	0	0
5	0	0	0	0	0	1
6	0	1	0	0	0	0

Obrázek 2.4: Příklad zápisu matice sousednosti pro graf 2.1 a digraf 2.2

2.3.4 Seznam vrcholů okolí každého vrcholu grafu

Graf je také možné reprezentovat tak, že ke každému vrcholu v zadáme množinu $V(v)$ seznam jeho nejbližších sousedů. Podobně digraf je možné reprezentovat takm že každému vrcholu v zadáme množinu $V(v)$ t.j. množinu konců hran vycházejících z vrcholu v.



Obrázek 2.5: Příklad zápisu seznamem vrcholů pro graf 2.1 a digraf 2.2

Pro reprezentaci grafů v informatice se využívá všech výše zmíněných přístupů. Nejsou zde uvedeny všechny možné přístupy, jen ty častěji využívané. Pro zajímavost uveďme ještě reprezantaci pomocí *incidenční matice vrcholů a hran*. Výběr vhodného přístupu se liší podle vlastností grafu. Například víme-li, že graf bude mít hodně vrcholů a relativně málo hran, může být matice zbytečné plýtvání pamětí a vyplatí se použít například seznam sousedů.

2.4 Sled, tah, cesta

Pomocí grafu či digrafu se často modelují spojovací sítě, v našem případě půjde o počítačovou síť. Proto je potřebné popsat, jakým způsobem prochází v síti datový paket. Způsob průchodu paketu po síti je možný jen tak, že z vrcholu sítě se paket dostane na některou s ním incidentí hranu a dojde do druhého vrcholu hrany, odtud do další incidentní hrany atd. Pakety v takovéto síti nemají dovelené přeskakovat z vrcholu nebo hrany na neincidentní vrchol či hranu. Způsob průchodu sítí je dále rozlišován podle toho, zda je povoleno či zakázáno jít po jednom úseku vícekrát a dále podle toho, zda je možné navštívit jeden vrchol vícekrát.

Definice 2.4.1 Nechť $G = (V, H)$ je graf. **Sled** v grafu G je libovolná alternujíci(střídavá) posloupnost vrcholů a hran tvaru

$$v_1, \{v_1, v_2\}, v_2, \{v_2, v_3\}, v_3, ..., \{v_{n-1}, v_n\}, v_n. \tag{2.3}$$

Definice 2.4.2 Nechť $G = (V, H)$ je graf. **Tah** v grafu G je takový sled v grafu G, ve kterém se žádná hrana neopakuje.

Definice 2.4.3 Nechť $G = (V, H)$ je graf. **Cesta** v grafu G je takový sled v grafu G, ve kterém se žádný vrchol neopakuje.

Definice 2.4.4 Nechť $G = (V, H)$ je digraf. **Orientovaný sled** v digrafu G je libovolná alternujíci(střídavá) posloupnost vrcholů a hran tvaru

$$v_1, (v_1, v_2), v_2, (v_2, v_3), v_3, ..., (v_{n-1}, v_n), v_n. \tag{2.4}$$

Definice 2.4.5 Nechť $G = (V, H)$ je digraf. **Orientovaný tah** v digrafu G je takový orientovaný sled v digrafu G, ve kterém se žádná hrana neopakuje.

Definice 2.4.6 Nechť $G = (V, H)$ je digraf. **Orientovaná cesta** v digrafu G je takový orientovaný sled v digrafu G, ve kterém se žádný vrchol neopakuje.

Definice 2.4.7 Sled (tah) $m(u, v) = v_1, h_1, v_2, h_2, ..., v_{n-1}, h_{h-1}, v_n$ nazveme **uzavřený**, pokud $v_1 = v_n$. Jinak sled (tah) $m(u, v)$ nazveme **otevřený**.

Definice 2.4.8 **Cyklus (orientovaný cyklus)** je uzavřený tah (orientovaný tah), ve kterém se kromě prvního a posledního vrcholu nevyskytuje žádný vrchol víc než jednou.

Ještě bych zde zmínil některá další možná rozdělení grafů z různých hledisek:

1. Podle orientace hran
    - Orientované – digraf 2.1.2

- Neorientované – graf 2.1.1

2. Podle existence ohodnocení hran
   - ohodnocené (každá hrana je ohodnocena reálným číslem)
   - neohodnocené (hrany nejsou ohodnoceny, jsou rovnocenné)
3. Podle souvislosti
   - souvislé (existuje-li cesta 2.4.3 mezi každou dvojicí vrcholů)
   - nesouvislé
4. Podle existence kružnice (cyklu 2.4.8) v grafu
   - cyklické
   - acyklické (např. stromy)

Rozdělení by mohlo být jistě obsáhlejší, ale zde jsou vybrány jen některé pro ilustraci. Více o teorii grafu se dozvíte ze zdrojů [4], [8] a [12] ze kterých jsem čerpal.

Kapitola 3

Směrovací algoritmy

Hlavním úkolem programu by mělo být zobrazení směrovacích algoritmů na zvolené topologii propojovací sítě, řekneme si tedy něco o propojovacích sítích, komunikačních vzorech a paralelních algoritmech. Věškré informace uvedené v této kapitole jsem nastudoval z knih [5], [2] a opory kurzu Architektura a programování paralelních systémů [14].

3.1 Propojovací sítě

Propojovacích sítě jsou jedna ze základních komponent architektury paralelních počítačů. Propojovacích sítě rozlišujeme do dvou tříd na *přímé* a *nepřímé*. V přímých sítích je každý uzel vysílač i přijímač zároveň, v nepřímých je uzel buď vysílač, nebo přijímač. Na tyto sítě jsou kladeny různé požadavky:

- Malý a konstantní stupeň uzlu (výrobně a finančně méně náročné)
- Malý průměr a malá průměrná vzdálenost (rychlejší komunikace)
- Symetrie (jednodušší návrh algoritmů)
- Hierarchická rekurzivita (graf obsahuje instance sebe sama)
- Vysoká souvislost (odolnost vůči poruchám)
- Vysoká bisekční šířka (zvyšuje se přenosová kapacita mezi rozdělenými částmi)
- Podpora pro směrování a kolektivní operace (topologie by měla umožňovat jednoduché směrovací algoritmy)

Dále u nich můžeme najít tyto základní typy topologií:

- Striktně ortogonální topologie (hyperkrychle, mřížka, torus)
- Hyperkubické topologie (kružnice spojená krychlí, butterfly)
- Stromové topologie (binární strom, stromová mřížka)
- Posuvné topologie (De Bruijnova a Kautzova sít)

3.2 Propojovací sítě

Každý paralelní výpočet se skládá z části kdy se počítá a kdy se komunikuje (zasílání zpráv nebo synchronizace). Komunikace je důležitá část, která se nezanedbatelnou měrou podílí na efektivitě paralelního zpravování. Základní komunikační vzory, vyskytující se ve většině paralelních algoritmů, jsou

- OAB (např. klasické rozhlášení parametrů výpočtu před jeho zahájením)
- AAB (např. Bariéra, výměna mezivýsledků mezi všemi uzly, atd.)
- OAS (např. Rozptýlení balíčků dat pro zpracování - Rozděl a panuj, atd.)
- AAS (např. transpozice matice - každý uzel má jeden řádek a potřebuje jeden sloupec, posílá ostatním odpovídající data a ty rovněž přijímá)

Pro efektivní naplánovaní těchto komunikačních vzorů na danou topologii vzniklo mnoho směrovacích algoritmů, jejichž cílem je odstranit vzájemné blokování, uváznutí, vyhladovění, atd.). Mezi klasické techniky patří právě XY-routing (deterministické), Valiantovo adaptivní směrování atd. Tyto směrovací algoritmy ovšem neberou v potaz časové hledisko (časovou režii) pro vykonání daného komunikačního vzoru (OAS). Proto vznikají nové směrovací algoritmy které se snaží tuto režii minimalizovat (použitím AI, Evolučních algoritmů, simulovaného žíhání, atd.) Výsledky dosažené pomocí těchto technik překonávají konvečně navržené směrovací algoritmy (dosahují na konkrétní topologii lepších časů, při vyloučení uváznutí, atd...) Jejich struktura je však na rozdíl od klasických směrovacích algoritmů nepravidelná, novátorská, atd.

Je proto vhodné tyto algoritmy vizualizovat do srozumitelné podoby např. pro návrháře propojovacích sítí, kteří mohou tyto algoritmy analyzovat, a na základě zjištěných vlastností navrhovat "lepší" propojovací sítě.

3.3 Komunikační vzory

Směrovací algoritmy pro komunikaci v síti rozdělujeme do více skupiny, podle toho kolik uzlů se podílí na komunikaci v síti. Každý uzel v sítí může plnit úlohu vysílače (Transmiter) V, přijímače (Reciever) P či obě úlohy najednou (Transmiter and Reciever). Jak již z názvu vyplývá, vysílač odesílá nějaká data do sítě a přijímač je bude získávat. Počet vysílačů i přijímačů v síti muže být velký, jako celkový počet uzlů U v síti. Podle těchto počtů rozlišujeme tyto skupiny komunikačních vzorů:

1. $V \cap P = \emptyset$, nepřekrývající se sada uzlů
   - One-to-All, $|V| = 1$, $|P| = U - 1$, např. One-to-All Broadcast (OAB) nebo One-to-All Scatter (OAS).
   - One-to-Many, $|V| = 1$, $|P| < U - 1$, např. Multicast (MC).
   - All-to-One, $|V| = U - 1$, $|P| = 1$, např. All-to-One Gather (AOG) nebo All-to-One Reduce (AOR).
   - Many-to-Many, $|V| = M$, $|P| = N$; $M, N < P$, např. nepřekrývající se sada uzlů: Many-to-Many Broadcast (MNB) nebo Many-to-Many scatter (MNS).
2. $|V \cap P| \geq 1|$, Many-to-Many, komunikace s překrývajícím se počtem uzlů

3. $|V \cap P| \geq U|$, All-to-All komunikace jako permutace, All-to-All Scatter (AAS), Broadcast (AAB), Reduce (AAR), a jiné [5], [2].

3.3.1 Komunikace One-to-All

Ve směrové komunikaci One-to-All je vždy jeden uzel označen jako odesílatel (někdy nazýván také jako kořen či iniciátor) a všechny ostatní uzly v síti jsou přijímače. My však budeme uvažovat, že odesílatel je členem skupiny uzlů v síti a sám je také přijímač. V tomto druhu komunikace, jsou dva rozdílné druhy komunikace:

- One-to-All Broadcast: Stejná zpráva je rozesílána od odesílatele všem přijímačům.
- One-to-All Scatter: Každému přijímači je odeslána jiná zpráva. Tento druh komunikace se také někdy nazývá jako personalized (osobní) broadcast.

3.3.2 Komunikace All-to-One

Ve směrové komunikaci All-to-One jsou všechny uzly v síti označeny jako odesílatelé a vždy jen jeden uzel je označen jako přijímač. Opět jsou v tomto druhu komunikace dva rozdílné druhy komunikace:

- All-to-One Reduce: Různé zprávy od různých odesílatelů jsou kombinovány a společně tvoří jedinou zprávu pro přijímač. Tento druh komunikace se také nazývá jako personalized combining (osobní kombinování) či global combining (globální kombinování)[5], [2].
- All-to-One Gather: Různé zprávy od různých odesílatelů jsou zřetězeny dohromady pro přijímač. Pořadí zřetězení je obvykle závislé na ID odesílatele.

3.3.3 Komunikace All-to-All

Ve skupinové komunikaci All-to-All všechny uzly v síti vykonávají svoji vlastní komunikaci typu One-to-All či All-to-One. To znamená, že každý uzel příjme n zpráv od n různých uzlů. Opět jsou v tomto druhu komunikace dva rozdílné druhy komunikace:

- All-to-All Broadcast. Všechny uzly provedou odeslání svého broadcastu. Tento druh komunikace se také někdy nazývá jako gossiping nebo total exchange (celková výměna).
- All-to-All Scatter. Všechny uzly provedou odeslání svého scatteru. Tento druh komunikace se také někdy nazývá jako personalized (osobní) All-to-All broadcast, index, nebo compete exchange (konkurentní výměna).

Kapitola 4

Přehledné rozvržení grafu

Zobrazení nějakého problému či řešení jako graf je dnes velmi časté. Při rozvržení grafu, je však důležité, aby byl graf pro pozorovatele jasný a dobře čitelný. Proto je důležité, aby vyhovoval následujícím podmínkám, které jsou pro lepší pochopení grafu důležité:

- Uzly a hrany by se neměly překrývat.
- Zajistit co nejméně křížení hran.
- Zajistit co nejméně zalomení hran.
- Hrany by měli mít přiměřenou délku.
- Souměrné zobrazení celého grafu.

Pro rozvržení grafů rozlišujeme řadu rozdílných topologií. Z těch známějších třeba hvězda, kruh, mřížka či strom. Algoritmy pro zobrazení těchto topologií se budeme zabývat dále v této kapitole.

4.1 Mapování uzlů na kružnici

Jedná se o velice jednoduchou metodu vykreslení grafu. Je zde třeba vhodně zvolit vzdálenost d mezi dvěma sousedními uzly a následně z této vzdálenosti spočítat poloměr kružnice r, na kterou budeme uzly mapovat. Pro složitější grafy je však tato metoda značně nevhodná, protože výsledný graf je potom velice nepřehledný. Jelikož tento algoritmus má složitost jen O(N), využívá se často pro počáteční inicializaci uzlů pro složitější algoritmy.

$\varphi = 2 * \pi /$`PocetUzlu;`
$r = d/(c * sin(\pi /$`PocetUzlu`$))$`;`
`for( int i = 0; i <` $<$ `PocetUzlu; i++ ) {`
    $p_i(x) = r * cos(-1 * \varphi + \pi/2)$`;`
    $p_i(y) = r * sin(-1 * \varphi + \pi/2)$`;`
}

Tabulka 4.1: Algoritmus mapování na kružnici

Ve výše uvedeném algoritmu 4.1 se uzly postupně mapují na kružnici v pořadí, v jakém byly vloženy do systému. Pokud změníme pořadí, můžeme někdy dosáhnout lepší přehlednosti a někdy i dokonce rovinného 2.2.2 zobrazení.

4.2 Mapování uzlů do mřížky

Zde se jedná o velice jednoduchou metodu zobrazení grafu. U tohoto algoritmu je nutné zadat počet řádků a sloupců mřížky. Podle těchto údajů se vhodně zvolí horizontální a vertikální vzdálenost mezi uzly.

```
posunX = sirkaPlochy/(sloupcu+2)
posunY = vyskaPlochy/(radku+2)
x=posunX
y=posunY
for( int i = 0; i < sloupcu; i++ ) {
   for( int j = 0; j < radku; i++ ) {
      k = i * radku + j  p_k(x) = x;
      p_k(y) = y;
      x+=posunX;
   }
   x=posunX;
   y+=posunY;
}
```

Tabulka 4.2: Algoritmus mapování do mřížky

Tak jako u mapování na kružnici, i v tomto výše uvedeném algoritmu 4.2 se uzly mapují podle pořadí, v jakém byly zadány do systému. Zde lze opět dosáhnout lepšího zobrazení, při vhodném poskládání uzlů, někdy i rovinného 2.2.2 zobrazení.

4.3 Pružinový algoritmus

Tento algoritmus je známý pod anglickými názvy *spring embedder* či *force-directed placement*. Pružinový algoritmus, který v roce 1984 navrhl Peter Eades [6], je nyní jedním z nejpopulárnějších algoritmů pro vykreslení neorientovaných grafů s přímými hranami. Je široce podporovaný v informačních vizualizačních systémech pro jeho jednoduchost a intuitivnost. Eades algoritmus uvažuje dvě estetická kritéria:

- jednotná délka hrany
- souměrnost, pokud je možná

V tomto algoritmu jsou vrcholy grafu označené jako sada bodů a každý pár bodů je spojen pružinou. Pružiny jsou spojeny se dvěma druhy sil: *přitažlivé síly* (*attraction force*) a *odpudivé síly* (*repulsive forces*), závislých na vzdálenosti a vlastnostech spojujícího prostoru.

Zobrazení grafu se blíží k optimálnímu zobrazení v závislosti na snižování energie pružinového algoritmu. Přitažlivá síla (f_a) je aplikována na uzly spojené pružinou, zatímco

odpudivá síla (f_r) je aplikována na nespojené uzly. Tyto síly jsou definované:

$$f_a(d) = k_a \log d \tag{4.1}$$

$$f_r(d) = \frac{k_r}{d^2} \tag{4.2}$$

Kde k_a a k_r jsou konstanty a d je aktuální vzdálenost mezi uzly. Pro spojené uzl, je d délka pružiny. Počáteční rozvržení grafu je konfigurované náhodně. Během každé iterace jsou vypočítány síly pro každý uzel a uzly jsou podle toho následně přesunuty, aby mohla být snížená energie v systému.

```
{ a drawing frame: W × L }
G := (V, E);
k := √(W * L / | V |);
function fa(x) := begin return x²/k end;
function fr(x) := begin return −k²/x end;

for i := 1 to iterations do begin
  {calculate repulsive forces}
  for v in V do begin
    {each vertex has two vectors: .pos and .disp}
    v.disp := 0;
    for u in V do
      if (u≠v) then begin
        Δ := v.pos − u.pos;
        v.disp := v.disp + (Δ/|Δ|)*fr(|Δ|);
      end
    end

  {calculate attractive forces}
  for e in E do begin
    Δ := e.v.pos − e.u.pos;
    e.v.disp := e.v.disp − (Δ/|Δ|)*fa(|Δ|);
    e.u.disp := e.u.disp + (Δ/|Δ|)*fa(|Δ|);
  end

  {limit the maximum displacement to the temparature t}
  {and prevent from being displaced outside frame}
  for v in V do begin
    v.pos := v.pos + (v.disp/|v.disp|)*min(v.disp, t);
    v.pos.x := min(W/2, max(−W/2, v.pos.x));
    v.pos.y := min(L/2, max(−L/2, v.pos.y));
  end
  {reduce the temperature as the layout approaches a better configuration}
  t := cool(t);
end
```

Obrázek 4.1: Pseudokód algoritmu Fruchterman a Reingold [7]

4.3.1 Fruchterman a Reingold

Významné rozšíření a přepracování pružinového algoritmu udělaly v roce 1991 Fruchterman a Reingold [7]. Jejich algoritmus následuje obecně uznávaná estetická kritéria pro vykreslení grafu, včetně rovnoměrného rozložení vrcholů, minimalizovaného křížení hran či jednotné délky hran. Jako v původním algoritmu jsou přitažlivé síly vypočítány jen pro sousední uzly a odpudivé síly jsou vypočteny pro všechny dvojice uzlů. Podle Fruchtermana a Reingolda, uzly ve vzdálenosti d jsou přitahovány vzájemnou přitažlivou silou f_a :

$$f_a(d) = \frac{d^2}{k} \tag{4.3}$$

a odpuzovány od sebe odpuzující silou f_r :

$$f_r(d) = -\frac{k^2}{d} \tag{4.4}$$

kde k je optimální vzdálenost mezi uzly v grafu, vypočítané z počtu uzlů a velikosti kreslicí plochy. Tyto výpočty jsou prováděny iterativně, dokud není dosáhnuto rovnovážného stavu v modelu. Uvnitř každé iterace jsou vypočítány síly pro každý uzel a na konec jsou všechny uzly přesunuty na nové pozice současně. Proces výpočtu je také řízený teplotním parametrem, podobným způsobem jako simulovaného žíhání 4.4.

4.4 Simulované žíhání

V roce 1996 Davidson a Harel [3] popsaly, jak aplikovat simulované žíhání na vykreslení grafu. Jejich algoritmus založený na algoritmu spring-embedder 4.3, pro zobrazení neorientovaných grafů s přímými hranami, zvláště zdůrazňuje estetickou kvalitu grafu. Například uzly a okraje by měly být umístěný tak, že graf je zřetelně zobrazen.

Významným problémem simulovaného žíhání je však jeho efektivita. Obecně jsou tyto algoritmy relativně pomalé. Podstatnějším problémem je, že algoritmus simulovaného žíhání se může zhroutit, jestliže velikost grafu, který má být vykreslen, je velmi velký. Davidson a Harel si všimly, že jejich algoritmus dokáže vykreslit jen grafy s maximálně 30 uzly a 50 hranami, ale kvalita výstupu se rychle zhoršuje s velikostí grafu. Ve skutečnosti, jejich algoritmus je tak časově náročný, že je vhodný jen k upravení hrubého řešení, nalezeného jiným algoritmem.

Hlavní síla simulovaného žíhání je jeho schopnost, vypořádat se s optimalizačními problémy v diskrétním konfiguračním prostoru, který je příliš velký pro důkladné hledání. Cílem je minimalizovat či maximalizovat hodnotu heuristické funkce. Simulované žíhání typicky začíná s náhodně vybraným počátečním uspořádáním a opakovaným pátráním hledá uspořádání, která mohou snížit hodnotu heuristické funkce.

Klíčovou funkcí simulovaného žíhání je zajistit, že se hledání nezastaví v lokálním minimu, ale až v globálním minimu. Tato funkce je založená na analogii k fyzikálnímu žíhání, ve kterém jsou kapaliny pomalu ochlazovány až do krystalického či pevného stavu.

Pokud je kapalina ochlazována pomalu, získá formu krystalu, který představuje systém s nejmenším množstvím energie. Na rozdíl od toho, pokud je ochlazena rychle, energie systému je vyšší než v krystalickém stavu. Když je kapalina ochlazována pomalu, atomy mají dostatek času na to dosáhnout tepelné rovnováhy v každé teplotě.

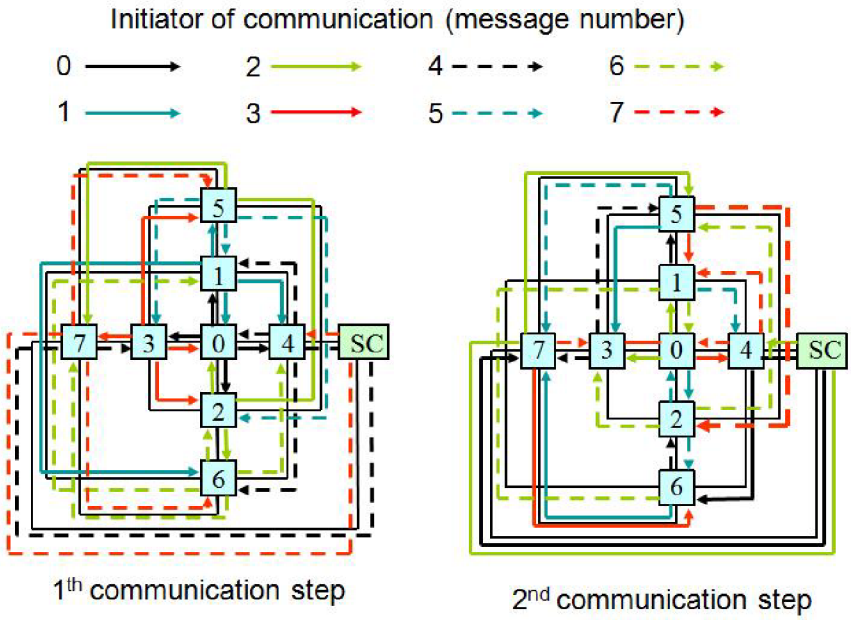
Klíčový prvek v simulovaném žíhání je heuristická funkce, proto musí být vysoce optimalizovaná. Musí také obsahovat kritéria k tomu, aby bylo uspokojivé konečné rozvržení

grafu, např. délka hran či křížení hran. Na hledání této funkce musí být kladen velký důraz, jelikož je velikým zatížením při výpočtu algoritmu. Vzhledem k tomu, že nalezení vhodné heuristické funkce je velice složité a nároký na výpočetní systém jsou příliž vysoké, nebudeme zde již více rozebírat tento algoritmus.

Kapitola 5

Analýza a návrh programu

Cílem práce je vytvořit programový prostředek, jenž graficky znázorní danou topologii. Na této topologii potom bude znázorněn průběh směrovacího algoritmu pro vybraný komunikační vzor. Popis topologie i směrovacího algoritmu bude vstupem algoritmu a výstupem bude něco obdobného jako na obrázku 5.1.



Obrázek 5.1: Návrh možného výstupu programu

Nároky na program byly po konzultaci se zadavatelem rozšířeny o bližší specifikaci podoby a funkcionality výsledného programu či produktu.

Program by měl umět načíst rozložení topologie. Toto rozložení je jedním ze vstupů dodaných zadavatelem. Podoba tohoto souboru je více rozebrána 5.1. Po načtení by tedy měl následně program vykreslit diagram topologie a pomocí algoritmů popsaných v kapitole 4 , upravit rozložení jednotlivých bodů do nějaké vhodné estetické podoby. Dalším požadavkem bylo, aby bylo možné upravit si rozložení grafu ručně a docílit tím ještě lepšího diagramu topologie.

Jak už bylo zmíněno, vstupy programu jsou dva. Druhý soubor obsahuje popis směro-

vého algoritmu. Jeho bližší podoba je rozebrána zde 5.2. Jak je vidět na obrázku 5.1, směrové algoritmy jsou rozděleny na více kroků, proto byl požadavek na možnost zobrazení směrových algoritmů postupně krok po kroku s možností zobrazovat, již dříve provedené kroky. Toto zobrazení je však pro větší topologie a směrové algoritmy s více kroky značně nepřehledné, proto byla uvažována ještě možnost zobrazení jen aktuálního kroku.

Dalším požadavkem bylo, aby bylo možné vykreslené topologie i se směrovými algoritmy ukládat, buď v podobě obrázku, nebo pro pozdější opětovné načtení již upraveného grafu do programu v podobě XML dokumentů. Pro ukládání do obrázku byl ještě kladen důraz na možnost úpravy rozlišení výsledného obrázku.

5.1 Vstupní soubor – Topologie

Popis topologie je uložen v textovém souboru. Jeho vnitřní struktura není nijak složitá. Soubor může obsahovat řádkové poznámky, které začínají znakem #. Dále na prvním neokomentovaném řádku se nachází údaj (číslo) o celkovém počtu uzlů v grafu a hned bezprostředně za ním údaj o maximálním počtu hran, které jsou s uzlem spojeny. Pak již vždy na jednom řádku následuje číslo uzlu odkud povedeme hranu. Po tomto údaji, pořád na stejném řádku, může být více údajů kam naši hranu povedeme. Ještě je zde nepovinný parametr mezi zdrojovým a cílovými uzly, a to zadání typu uzlu. Typy i s vysvětlením jsou uvedeny v tabulce 5.1.

Označení	**Typ (význam)**
T	Transmiter (vysílač)
R	Reciever (přijímač)
B	Transmiter and Reciever (vysílač a přijímač)
N	None

Tabulka 5.1: Význam označení typů uzlů

```
# počet uzlů a max počet hran jednoho uzlu
9  4

# uzel a jeho sousedé
0  1  3
1  0  2  4
2  1  5
3  0  4  6
4  1  3  5  7
5  2  4  8
6  3  7
7  4  6  8
8  5  7
```

Tabulka 5.2: Ukázka vstupního souboru topologie - mřížka 3×3

5.2 Vstupní soubor – Směrovací algoritmus

Popis směrovacího algoritmu je také uložen v textovém souboru, nyní však s příponou `.err`. Jeho vnitřní struktura je již na rozdíl od souboru s topologií značně složitější. Může začínat hlavičkou s nějakými zajímavými informacemi, např. počet kroků, počet vysílačů, počet přijímačů, atd. Jelikož samotné cesty paketů po síti jsou podrobně popsány níže, je pro náš program hlavička nezajímavá a navíc není ve všech souborech se směrovými algoritmy obsažena. Pro nás je důležité až samotné tělo pod hlavičkou. Pro jednodušší vysvětlení se nejprve podíváme na malou část z těla souboru 5.3.

———0. step of comunication ———
————————————-
– OAB source = 0 –
Message from 0 -> 0:
0,
Message from 0 -> 1:
0, 1,
Message from 0 -> 8:
0, 4, 8,
————————————-
– OAB source = 1 –
Message from 1 -> 0:
1, 0,
Message from 1 -> 1:
1,
Message from 1 -> 2:
1, 2,
————————————-
...

Tabulka 5.3: Ukázka vstupního souboru směrovacího algoritmu

Jak je patrné z ukázky 5.3, je nejprve určeno o kolikátý krok směrového algoritmu se jedná, v našem případě o nultý (první). V textu je to zapsáno takto *0. step of comunication*. Další pro nás zajímavý řádek je *OAB source = 0*, který nám říká, kdo je zdrojem přenášeného paketu. Jistě jste si povšimli, že je zde zkratka OAB, což znamená One-to-All Broadcast. Teď by mohl někdo namítnou, jak je to při posílání scatteru. Odpověď je jednoduchá. Tento řádek není povinný, takže u posílání scatteru se nevyskytuje. Posledním druhem a pro nás asi nejdůležitějším jsou řádky s textem: Kde *Message from 0 -> 8* nám

Message from 0 -> 8:
0, 4, 8,

říká odkud a kam se posílá paket a druhý řádek s čísly nám říká, přes které uzly paket poputuje k cíly. Tato výše popsaná konstrukce se pak již jen neustále opakuje až do konce souboru.

5.3 Návrh programu

Správné navržení programu ještě před samotným započetím realizace ve vybraném implementačním jazyce je velmi důležité, proto by na tuto fázi vývoje softwaru neměla být zanedbána. Cílem je ujasnit si, jak by měl program vypadat a jakou by měl mít funkcionalitu. Program byl již od začátku koncipován jako program s grafickým uživatelským prostředím. Rozvržení ovládacích prvků programu bylo navrženo, jako u většiny dnešních kreslících či grafických programů.

5.3.1 Hlavní menu

Konkrétně tedy s vysouvacím menu na horním okraji programu, s obvyklými funkčními tlačítky jako např. vytvoření nového souboru, v našem případě grafu, uzavření programu, nastavení programu, uložení vytvořeného grafu, atd. a pak tlačítka specifická pro náš program např. otevření topologie, uspořádání teologie či otevření směrového algoritmu.

5.3.2 Nástrojová lišta

Dalším prvkem byla nástrojová lišta, která je taktéž velice běžná u dnešních programů. Její funkce je v podstatě podobná vysouvacímu menu popsanému výše. Hlavní rozdíl je však, že nástrojová lišta neobsahuje všechny tlačítka, ale jen ty nejčastěji používané pro urychlení práce s programem. Jak u nástrojové lišty, tak u menu bylo již v návrhu zahrnut požadavek pro ovládání programu pomocí klávesových zkratek.

Obrázek 5.2: Návrh možného rozložení ovládacích komponent

5.3.3 Kreslící plocha

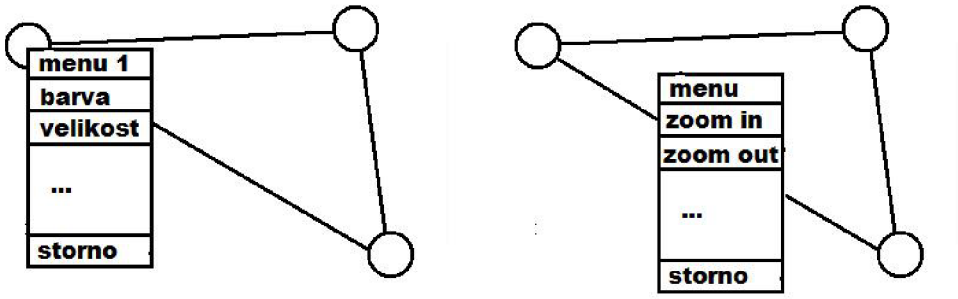
Je zřejmé, že bez menu by program asi nebyl moc užitečný, avšak hlavním úkolem je zobrazovat topologie a směrování, proto nejdůležitější komponentou programu bude plocha pro vykreslení topologie a směrových algoritmů. Zde budou zobrazeny jednotlivé uzly, se kterými bude možné manipulovat, případně ještě nastavit např. velikost nebo barvu. Dále se zde budou vykreslovat hrany mezi uzly a cesty paketů po síti, určené vybraným směrovým algoritmem. U hran i cest by mělo být také možné upravovat velikost čí barvu.

Kromě již zmíněného manipulování s uzly, bude důležité, aby bylo možné na kreslící ploše přiblížit či oddálit zobrazený graf, jelikož u větších grafů by mohlo být zobrazení značně nepřehledné. Toto přibližování bude ovládáno z nástrojové lišty, menu, klávesových

zkratek nebo pomocí plovoucího menu, které by mělo být dalším prvkem zakomponovaným do vykreslovací plochy.

5.3.4 Plovoucí menu

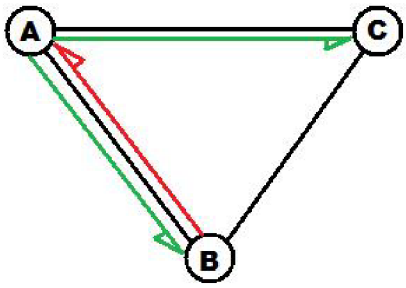
Plovoucí nabídky by měli být ve vykreslovací ploše dvě. Výběr, která bude zobrazena, bude záviset na tom, zda byla nabídka vyvolána při stisku tlačítka nad uzlem či nikoliv. Nabídka nad uzlem bude obsahovat možnosti nastavení daného uzlu, např. barva. Nabídka mimo uzel by měla obsahovat možnosti přiblížení či oddálení. Na obrázku 5.3 je vidět, jak by měly nabídky přibližně vypadat.



Obrázek 5.3: Návrh plovoucích nabídek v kreslící ploše

5.3.5 Zobrazení cesty paketu

Nyní si ještě přibližně popíšeme, jak bude vypadat samotné zobrazení cesty paketu v síti. Po konzultaci se zadavatelem a prostudováním vstupních souboru, bylo dosaženo závěru, že v jeden okamžik může po jedné hraně putovat pouze jeden paket v obou směrech. V praxi tedy jeden z bodu A do bodu B a druhý opačně z bodu A do bodu B. Toto zjištění tedy vedlo k určení pravidla, že zobrazení cesty paketu bude znázorněno vždy na pravé straně od hrany. Lépe je to vidět na obrázku 5.4.



Obrázek 5.4: Návrh zobrazení cesty paketu

Popišme si tedy příklad na obrázku 5.4. Posíláme paket z bodu A do bodu B. Jak je vidět na obrázku, paket zde znázorněný zelenou šipkou, je znázorněna vpravo od hrany. Obdobný princip je vidět i na cestě paketu z bodu A do B a B do A.

5.4 Výběr implementačního jazyka

V požadavcích zadavatele nebyl konkrétně určený implementační jazyk, čili bylo tedy možné si zvolit z více možných variant. Vzhledem k tomu, že jsem se během studia setkal hlavně s jazykem C resp. C++ a Java, rozhodoval jsem se mezi těmito jazyky. Jelikož měla být návrhová topologie objektově orientovaná, zvolil jsem Java. Tento jazyk je čistě objektový a měl jsem s ním již nějaké základní zkušenosti. S jazykem C jsem měl sice větší zkušenosti, ale jen v oblasti strukturovaného programování. Pro učení jazyka jsem používal hlavně knihy [10], [9], [11] a internetový tutoriál [13].

Kapitola 6

Implementace

V této kapitole si řekneme něco o samotné implementaci programu v jazyku Java. Program je rozdělen do několika souborů. Většinou je v každém soboru jen jedna třída, kromě několika souborů, ve kterých jsou obsaženy třídy pro obsluhu tlačítek či posluchačů událostí. Jelikož k učení jazyka Java byly původně použity knihy [10] a [9], byla původní aplikace vytvořená kompletně ve starším grafickém prostředí **AWT** `(java.awt)`, ale postupným vývojem byl skoro celý projekt přepracován do novějšího prostředí **JFC Swing** `(javax.swing)`. Popis implementace je rozdělen do několika částí. Postupně rozebereme jednotlivé třídy, které jsou součástí programu.

6.1 Spuštění programu

Pro spuštění aplikace je nejdříve vytvořena instance třídy `Hlavni.java`. Tato třída nemá skoro žádnou úlohu, jen vytvoří instanci třídy `Okno.java`, která vytvoří hlavní okno našeho programu.

6.2 Hlavní okno programu

Třída `Okno.java` rozšiřuje vlastnosti třídy `JFrame` a jejím úkolem je tedy vytvořit hlavní okno programu. Hned v konstruktoru této třídy je zavolána metoda `void initOkno()`.

6.2.1 Metoda `void initOkno()`

V této třídě jsou nastaveny základní vlastnosti okna, např. rozměry okna, nastavení rozmístění komponent (*layout manager*). Zde se o rozmístění stará `BorderLayout`, který může umístit sice jen 5 komponent. Ty mohou být umístěny vlevo, vpravo, nahoře, dole nebo uprostřed. Pro naši aplikaci je však tento druh rozmístění plně dostačující. Dále se tato třída stará o vytvoření a rozmístění komponent v okně.

První komponentou, která je vytvořena, je *hlavní menu* programu. Tato komponenta je instancí tříd `JMenuBar`, `JMenu` a `JMenuItem`. Třída `JMenuBar` nám vytvoří pouze lištu, do této lišty si pak musíme přidat jednotlivé nabídky `JMenu`, které jsme si již dříve naplnily konkrétními tlačítky `JMenuItem`. Tlačítkům `JMenuItem` však ještě musíme přidat funkcionalitu. Pro toto je zde vytvořena speciální třída `ObsluhaAkci`(odkaz). U vytvořeného menu jsou následně ještě nastaveny klávesové zkratky pro možnosti rychlejšího ovládání. Nakonec je celá komponenta hlavního menu přidána do okna programu.

Druhou komponentou je *nástrojová lišta*, která je instancí tříd `JToolBar` a `JButton`. Pomocí `JToolBar` si vytvoříme lištu, do které pomoci metody

```
JButton pridejTlacitkoListy(Action akce)
```

postupně přidáme tlačítka pro ovládání programu. Tato funkce má jeden parametr, a sice `Action` akce, který nám říká, co bude tlačítko po stisknutí dělat. Jako parametr akce vkládáme instance třídy `ObsluhaAkci` 6.3, která se využívá pro obsluhu většiny událostí v programu. Nakonec opět přidáme tlačítkům klávesové zkratky a vložíme nástrojovou lištu do hlavního okna. Její umístění v `BorderLayout` je nahoře přímo pod hlavním menu.

Dále se zde vytvoří instance třídy `PravyPanel.java` 6.4, která je v programu umístěna vpravo. Obsahuje dvě *záložky*, první pro výpis všech uzlů a druhou pro možnosti zobrazení cest paketů směrovacího algoritmu.

V neposlední řadě je doprostřed okna vložen nový `JPanel`. Tento `JPanel` má nastavené rozmístění komponent na `BorderLayout`. Dolů a vpravo tohoto panelu jsou vloženy posuvníky `Scrollbar`, které se budou využívat pro posunutí obrázku při zvětšení. Uprostřed tohoto panelu bude vložena komponenta `Platno.java` (odkaz), která se stará o samotná vykreslování diagramu grafu.

6.3 Obsluha akcí

Obsluhu akcí tlačítek v programu obstarává třída `ObsluhaAkci`, která je implementována v souboru `Okno.java`. Tato třída rozšiřuje třídu `AbstractAction`. Instanci třídy můžeme vytvořit dvěma způsoby, jelikož obsahuje dva konstruktory.

```
ObsluhaAkci(String jmeno, int akce)
```

První konstruktor vytvoří instanci, která podle parametru int akce vyvolá požadovanou akci, parametr `String jmeno` nám říká, jakou ikonu bude mít tlačítko na nástrojové liště. Pokud vytváříme obsluhu pro akci, kde není potřeba ikony, parametr `String jmeno` nabývá hodnoty `null`.

```
ObsluhaAkci(String jmeno, KeyStroke kuhoz, int akce)
```

Tento konstruktor, funguje skoro stejně, jen s tím rozdílem, že parametr `KeyStroke kuhoz` nám sděluje, pomocí které klávesové zkratky budeme chtít, aby se požadovaná akce provedla.

Samotná funkcionalita této třídy je obsažena v jediné metodě

```
void actionPerformed(ActionEvent e)
```

Kde parametr `ActionEvent e` je pro nás nepodstatný. Pro nás je podstatná hodnota `int akce`, kterou jsme získali již při vytvoření instance. Uvnitř této metody je obsažen jen jeden `switch`, který pomocí hodnoty `int akce` zavolá požadovanou metodu pro provedení příslušné akce. Jak to přibližně vypadá je vidět na ukázce 6.1.

Ukázka je zjednodušena jen pro pochopení principu této třídy. Je zde však dobře vidět, že podle parametru int akce, který se zadává již při vytvoření instance třídy, se provádějí požadované akce.

```
class ObsluhaAkci extends AbstractAction \{
  int akce;\\
  public ObsluhaAkci( int akce) \{
    this.akce=akce
  \}

  public void actionPerformed(ActionEvent e) \{
    switch(akce) \{
      case AKCE1:
          //obsluha akce1
        break;
      ...
      default:
          //případná akce
        break;
    \}
  \}
\}
```

Tabulka 6.1: Ukázka obsluhy akcí v programu

6.4 Ovládací panel

Ovládací panel, v programu umístěný vpravo, slouží pro zobrazení informací o uzlech a cestách paketů. Panel je implementován v souboru `PravyPanel.java` a rozšiřuje třídu `JPanel`. Byl zde umístěn, aby bylo možné, pomocí jednoduchého ovládání, upravovat zobrazení diagramu grafu. Jelikož chceme upravovat jak uzly, tak cesty, je na tomto panelu umístěn `JTabbedPane`, který nám přidává záložky. Mezi těmito záložkami pak bude možné jednoduše přepínat. Záložky pro uzly i cesty jsou implantovány jako `JPanel`.

6.4.1 Záložka uzlů

Konkrétně záložka uzlů obsahuje jen tabulku, která zobrazuje výčet všech uzlů grafu, u kterých máme možnost změnit si jejich barvu. Celá tabulka je tvořena ze dvou komponent, a sice samotné tabulky `JTable` a modelu tabulky `AbstractTableModel`, kde jsou uloženy všechny informace.

Pro vytvoření modelu tabulky uzlů vytvoříme instanci třídy `TabulkaUzlu`, která je rozšířením původního `AbstractTableModel`. Hlavním rozšířením je, že po kliknutí na barvu se nám zobrazí `JColorChooser` pro zadání nové barvy. Tato vlastnost buňky v tabulce je zajištěna pomocí tříd `EditorBarvy` a `RenderBarvy`. Instance těchto tříd jsou přidány jako vlastnosti tabulky pro položky v tabulce, které budou typu `Color`. Níže je znázorněno, jak je to v programu ošetřeno.

```
tableUzly.setDefaultRenderer(Color.class, new RenderBarvy(true));
tableUzly.setDefaultEditor(Color.class, new EditorBarvy());
```

Třída `EditorBarvy` je zavolána po kliknutí do položky v tabulce a zobrazí dialog pro výběr nove barvy. Po zvolení barvy vygeneruje třída položku vybarvenou touto barvou.

6.4.2 Záložka cesty paketů

Záložka pro nastavení cest paketu obsahuje trochu jednodušší tabulku. Opět se skládá z tabulky a jejího modelu, avšak zde je pro model instance třídy `TabulkaSmer`. Tato tabulka obsahuje jen výčet uzlů a check boxy. Ty zde slouží pro možnost zobrazení jen některých *cest*, konkrétně ty které mají u svých zdrojových uzlů paketů označené zobrazení. Dále tato záložka obsahuje `JSlider` pro možnost zobrazení jednotlivých kroků komunikace (cest paketů). Úplně dole jsou pak na záložce dvě tlačítka pro označení nebo odznačení všech uzlů.

Pro vytvoření panelů jsou v této třídě vytvořeny metody, které všechny výše popsané komponenty přidají do záložek, konkrétně tyto dvě metody :

```
protected JComponent vypisUzlu()
protected JComponent vypisKroku()
```

6.5 Kreslící plocha

Plocha, do které budeme vykreslovat diagram grafu a znázorňovat cesty paketů, je implementován ve třídě `Platno.java`, která rozšiřuje třídu `Jpanel` a implementuje posluchače událostí `FocusListener, ActionListener, KeyListener` a `MouseListener`, pomocí kterých budeme odchytávat např. *pohyby a akce myši.*

6.5.1 Vykreslení diagramu grafu

Tato třída má dva hlavní úkoly. První je pomoci metody `void paint(Graphics g)` vykreslit diagram grafu, jehož reprezentace je uložena v instanci třídy `Graf()`(odkaz). O samotné vykreslení jednotlivých uzlů, hran a cest, se tedy stará třída `Graf()`. Úkolem metody `paint()` je, předání grafického kontextu `Graphics g` kreslící plochy třídě `Graf()`, která se postará o samotné vykreslení jednotlivých uzlů, hran a cest, tedy celého diagramu grafu. Na ploše je možné s uzly pohybovat, ale při přímém vykreslování docházelo k nežádoucímu problikávání obrazu. Tento nežádoucí jev byl odstraněn až pomocí dvojitého vykreslování. Princip je vykreslení diagramu do pomocného grafického kontextu, který následně už vykreslíme přímo na obrazovku.

```
public void paint(Graphics g) \{
      Dimension imageSize = getSize();
      Image img = createImage(imageSize.width,imageSize.height);
      Graphics2D bg = (Graphics2D)img.getGraphics();
      graf.paint(bg);
      g.drawImage(img,0,0,null);
\}
```

Tabulka 6.2: Vykreslení digramu grafu s pomocným obrázkem

Nyní si vysvětlíme princip ukázky 6.2. Pomocí `getSize()` zjistíme rozměry vykreslovací plochy a následně si vytvoříme pomocný obrázek, do kterého budeme vykreslovat. Metodou `img.getGraphics()` získáme grafický kontext obrázku, který si přetypujeme na `Graphics2D`. Upravený kontext předáme instanci třídy `Graf()` pro vykreslení diagramu. Nakonec celý obrázek vykreslíme na kreslící plochu.

6.5.2 Pohyb uzlů

Kromě vykreslení diagramu grafu, umožňuje nám tato třída ještě pohybovat s uzly na kreslící ploše. To je obsluhováno pomocí metod:

```
public void mousePressed(MouseEvent e)
public void mouseDragged(MouseEvent e)
public void mouseReleased(MouseEvent e)
public void mouseExited(MouseEvent e)
```

Dále se zde ještě využívají tři pomocné metody:

```
public final boolean pressedOnUzel(Uzel u,MouseEvent e)
public final boolean mouseDraggedUzel(int x,int y,MouseEvent e)
public final Uzel isUzel(int x,int y, boolean pomoc)
```

Na příkladu vysvětlíme, jak to přibližně funguje. Předpokládejme, že jsme stlačili tlačítko myši v momentě, když se kurzor nacházel nad některým z uzlů. Stlačení tlačítka vyvolá obsluhu akce `mousePressed()`, která zavolá metodu `isUzel()`. Tato metoda nám v našem případě, tedy že jsme klikli na uzel, vrátí ukazatel na tento uzel, v opačném případě by nám vrátila hodnotu `null`. Nyní již víme, že se kursor nachází nad uzlem, proto zavoláme metodu `pressedOnUzel()`, která nám tento uzel označí. Pokud by jsme nyní začali pohybovat s uzlem, byla by zavolána metoda obsluhy akce `mouseDragged()`, která volá naši pomocnou metodu `mouseDraggedUzel()`. Jelikož již víme, se kterým uzlem budeme pohybovat, tak tato metoda pouze postupně posunuje uzel a volá metodu `repaint()` pro překreslení diagramu grafu. Nakonec už jen, pokud myš opustí vykreslovací plochu či je uvolněno tlačítko, zrušíme označení uzlu. Pro tyto případy se volají metody `mouseExited()` nebo `mouseReleased()`.

6.6 Reprezentace grafu

Jak bylo dříve uvedeno 2.3, pro reprezentaci grafu v informatice lze využít více způsobů. V našem případě víme vždy počet uzlů i hran grafu, proto padlo rozhodnutí na ukládání grafu jako množinu, přesněji seznam uzlů, hran a cest. Pro účely práce s jednotlivými uzly, hranami a cestami, jsou zde vytvořeny třídy, které vždy obsahují informace o jednom konkrétním objektu, resp. uzlu, hraně nebo cestě. Celá reprezentace je pak uložená v instanci třídy `Graf()`.

6.6.1 Graf

Třída `Graf()` je hlavní třída, která shromažďuje všechny informace o grafu. Tyto informace jsou uloženy v *seznamu*, pro uzly, pomocné uzly i hrany vždy jeden seznam. Pro práci s jednotlivými seznamy objektů jsou zde vytvořeny pomocné metody, které nám usnadňují práci. Pomocí těchto metod můžeme přidávat objekty do seznamů, vyhledávat v seznamech, mazat seznamy či získat počet objektů v seznamech. Také se zde nachází ještě metoda `paint(Graphics2D g)`, pomocí které do grafického kontextu `Graphics2D g`, postupně vykreslujeme všechny objekty uložené v seznamech, konkrétně se jedná o uzly, pomocné uzly a hrany.

6.6.2 Uzel

Třída pro uchovávání informací jednotlivých informací o uzlech resp. pomocných uzlech se jmenuje `Uzel.java`. Pro rozlišení, zda se jedná o normální či pomocný uzel jsou zde dva konstruktory:

```
public Uzel(int x ,int y, String popis, Color barva)
public Uzel(int x ,int y)
```

První konstruktor vytvoří normální uzel, jehož parametry jsou umístění, popisek a barva uzlu. Konstruktor pro pomocné body je jednodušší, obsahuje pouze umístění v ploše. Třída ještě obsahuje metody pro změny vlastností uzlu, např. změna pozice uzlu, změna barvy, změna popisku, pomocné metody pro zjištění, zda je kurzor myši nad daným uzlem resp. pomocným uzlem. Protože body jsou zadány na přesné místo, ale vykresleny jsou jako kruh resp. čtverec, je potřeba pro pozici myši vypočítat, zda je v mezích či nikoliv. Poslední metodou, kterou je důležité zmínit je metoda `paint(Graphics2D g)`, která již jen do grafického kontextu `Graphics2D g` vykreslí samotný bod. Pokud se jedná o normální uzel, vykreslí se kruh s určenou barvou, pro pomocný bod se vykreslí čtverec.

6.6.3 Hrana

K ukládání informací o hraně slouží metoda třídy `Hrana.java`. Hrana si uchovává mnohem více informací než uzel. To je dáno tím, že v této třídě se nevykreslují jen samotné hrany, ale i cesty paketů. Hrany byli původně jen kolmé z bodu A do bodu B, avšak vzhledem k možnému křížení hran je možné zabránit přidáním pomocných bodů, byly později přidány čtyři pomocné body, pomocí kterých je možné tvořit nejen kolmé hrany.

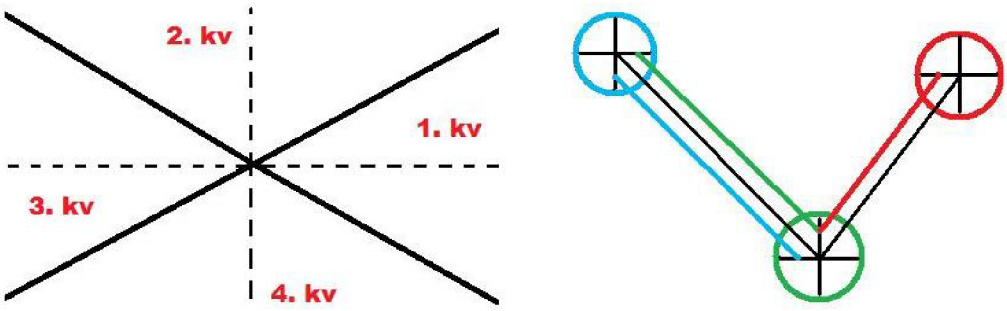
Každá hrana si tedy uchovává informace o svých krajních uzlech, pomocných uzlech, barvu a *seznam cest paketů*, které po hraně budou putovat. Pro ukládání cest paketů je zde třída `Cesta.java`, ve které jsou informace o zdrojovém uzlu paket, kroku kdy má být cesta zobrazena a směru, kterým paket putuje, tedy zda z bodu A do bodu B či naopak.

Metody obsažené v této třídě jsou skoro všechny pro vykreslování hran a cest paketů. Na vykreslování hran a cest mají velký vliv globální proměnné typu *boolean* `historieCesty, vykreslyPomocne, zmrazPomocne, zmrazPohyb` a ještě globální proměnná typu *integer* `aktualniKrok`. K čemu každá slouží bude uvedeno, až když je budeme potřebovat. Hlavní metodou, která se volá pro vykreslení je `paint(Graphics2D g)`. Zde se podle hodnoty globální proměnné `vykreslyPomocne` buď vykreslí kolmá čára mezi body, nebo se vykreslí čára se čtyřmi pomocnými body, se kterými bude možné manipulovat v závislosti na hodnotě proměnné zmrazPohyb. Toto má za úkol vykreslit jen hranu mezi jednotlivými body.

Vykreslení jednotlivých cest paketů závisí na globálních proměnných `historieCesty` a `aktualniKrok`. První zmíněná proměnná nám říká, zda budeme vykreslovat pouze cesty paketů v aktuálně zvoleném kroku nebo budeme chtít vykreslit i cesty v předešlých krocích. Podle druhé proměnné zjistím, až po který krok budeme cesty vykreslovat a dále podle toho malujeme šipku jen u aktuálních kroků.

U vykreslování cest jsme museli řešit problém se správným vykreslením, při změně pozice uzlu resp. pomocného uzlu. Toho bylo docíleno tím, že pozice uzlů vůči sobě byly rozděleny do čtyř kvadrantů 6.1. Při zobrazení cest jen u hran, bez pomocných uzlů, se řeší pozice jen těchto dvou uzlů, avšak při zobrazení s pomocnými uzly, musíme řešit pozice tří uzlů, abychom dosáhly napojení cest v místě pomocného uzlu.

Jak je vidět na obrázku, musíme tedy vždy zjistit vzájemnou polohu bodů. Pro tyto účely je zde metoda `kvadrant()`, která nám vrací číslo kvadrantu, ve kterém se nachází



Obrázek 6.1: Rozložení kvadrantů a ukázky mapování cesty k uzlu

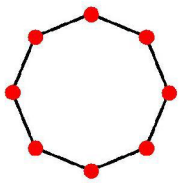
bod B vůči bodu A. Pro vykreslování cest jsou ve třídě dvě metody `malujCestu()` a `malujRovnouCestu()`. Pro zakončení cesty šipkou je zde metoda `malujSipku()`.

6.7 Vyrovnání diagramu grafu

Vyrovnání diagramu grafu je implementována ve třídě `Vyrovnani.java`. Pro samotné vyrovnání se zde využívá algoritmů popsaných v kapitole 4.

6.7.1 Kruh

Pro vyrovnání grafu do kruhu je využit až na malé výjimky algoritmus 4.1. Celý algoritmus je implementován ve statické metodě `kruznice()`. Na obrázku je ukázka vykreslení do kružnice.



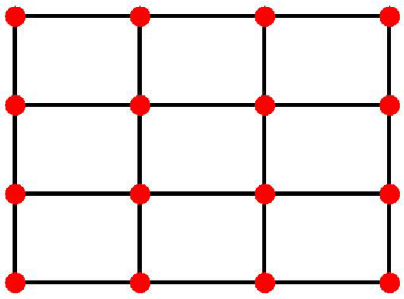
Obrázek 6.2: Ukázka vykreslení grafu do kruhu pomocí programu

6.7.2 Mřížka

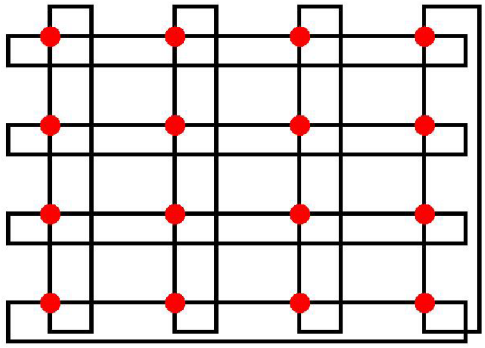
Vyrovnání grafu do mřížky využívá až na nepatrné změny algoritmus 4.2. Celý algoritmus je implementován ve statické metodě `mesh()`, která používá ještě pomocnou metodu `ziskejRozmery()`. Na obrázku je ukázka vykreslení do mřížky.

6.7.3 Torus

Další možné vyrovnání grafu je torus. Toto vykreslení používá metodu `mesh()` pro vykreslení do mřížky a pak svoji vlastní metodu `tori()`, která upraví rozmístění pomocných bodu. Na obrázku je ukázka vykreslení topologie torus.



Obrázek 6.3: Ukázka vykreslení grafu do mřížky pomocí programu



Obrázek 6.4: Ukázka vykreslení grafu do torusu pomocí programu

6.7.4 Pružinový algoritmus

Posledním implementovaným vyrovnáním grafu je pomocí pružinového algoritmu 4.3. Jeho implementace je ve třídě `Springer.java`, avšak i přes velké úsilí se nepodařilo uvést tento algoritmus do provozu schopného vyrovnat i jednoduchý graf do přijatelné podoby.

6.8 Otevírání, ukládání a nastavení

V této části si rozebereme metody pro otevírání vstupních souborů, ukládání reprezentace grafů do souborů a nastavení programu. Tyto metody jsou implementovány v souboru `Okno.java` či `XMLSoubor.java`.

6.8.1 Otevření vstupního souboru – Topologie

Otevření vstupního souboru s topologii je implementováno v metodě `zpracujOtevri()`, která vytvoří `FileDialog`, ve kterém si vybereme požadovaný soubor s topologií. Po úspěšném otevření zavolá pomocnou metodu `zpracujRaddek()`, která postupně čte jednotlivé řádky souboru a přidává uzly či hrany do instance třídy `Graf()`.

6.8.2 Otevření vstupního souboru – Směrovací algoritmus

Pro otevření vstupního souboru se směrovacím algoritmem používáme metodu `smerOtevri()`, která vytvoří `FileDialog` pro výběr vstupního souboru. Po úspěšném otevření souboru předáme soubor metodě `zpracujSmerovani()`, která postupně čte celý soubor a pomocí další metody `nactiCestu()` přidává k jednotlivým hranám, uloženým v instanci třídy `Graf()`, cesty paketu procházející po nich.

6.8.3 Uložení diagramu grafu

Ukládání diagramu grafu se provádí pomocí metody `ulozObrazek()`. V této metodě si vytvoříme instanci třídy `ObrazekOkno()`, která zde slouží pro zadání rozměru obrázku, získání názvu a umístění. Dle získaných rozměrů vytvoříme obrázek, ze kterého pomocí metody `createGraphics()` získáme grafický kontext.

```
BufferedImage img = new BufferedImage(sirka, vyska, typ);
Graphics2D bg = img.createGraphics();
```

Dalším krokem je upravení pozic uzlů a hran, jelikož ukládaný obrázek může nabývat jiných rozměrů než původní, tedy je potřeba diagram grafu upravit. Samotné ukládání je provedeno tímto způsobem:

```
ImageIO.write(img, "jpg", tmpFile);
```

Kde `img` je náš vykreslený obrázek, řetězec `"jpg"` nám udává, jaký typ souboru budeme ukládat a `tmpFile` je soubor, do kterého bude obrázek uložen.

6.8.4 Nastavení programu

Nastavení vlastností programu je implementován v souboru `NastaveniOkno.java`. Tato třída navíc implementuje `ChangeListener` pro možnost reakce na změny nastavení globálních proměnných, které se využívají ke změnám v zobrazení diagramu grafu. Je to *nastavení rozměrů* vykreslovací plochy `Dimension vykreslPloch`, nastavení rozměru a základních barev uzlů resp. hran, zobrazení jen aktuálního kroku cesty, atd. Toto nové nastavení je možné následně buď potvrdit, nebo stornovat.

6.9 Práce s XML dokumenty

K ukládání a otevírání reprezentace grafu ve formátu XML se využívá volně dostupné knihovny **Dom4J** [1]. Implementace metod využívající tuto knihovnu je v souboru `XMLSoubor.java`. Jelikož je tato třída využívána jak pro čtení, tak pro zápis do souboru, jsou zde pro rozlišení typu operace se souborem dva konstruktory.

6.9.1 Ukládání reprezentace do XML dokumentu

Operace ukládání do souboru XML se používá metoda `ulozXml()`, která je implementovaná v souboru `Okno.java`. Zde se vytvoří dialogové okno, kde si vybereme umístění a název,jak se bude soubor jmenovat. Pak se vytvoří instance třídy `XMLSoubor()`, pomocí které budeme, pomocí metod v ní obsažených, postupně ukládat jednotlivé uzly a hrany.

```
void pridejBod(cisloUzlu, poziceX, poziceY, popis, barva, jePomocnyBod)
void pridejHranu(Hrana)
```

V metodě `pridejBod()` jsou parametry celkem jasné. První je pořadové číslo uzlu, pak následuje umístění v diagramu, případný popisek a barva. Poslední parametr je typu boolean a říká nám zda se jedná o pomocný či normální bod. Tyto body jsou v grafu reprezentovány jinak, proto je třeba je rozlišit. Metoda `pridejHranu()` má jako parametr ukazatel na hranu, odkud si všechny požadované informace získá a uloží do souboru.

6.9.2 Načítání reprezentace z XML dokumentu

Otevírání již vytvořených souborů s topologií ve formátu XML je implementováno v metodě `otevriXml()`. Princip je obdobný jako při ukládání, čili nejprve je zobrazen dialog pro výběr souboru. Vybraný soubor se pak vloží jako parametr pro vytvoření instance třídy `XMLSoubor()`, která obsahuje metodu:

```
public void nactiRozestaveni(Graf)
```

V metodě se postupně projde celý soubor a do instance třídy `Graf()`, jenž je parametrem se přidávají načtené informace o uzlech resp. pomocných uzlech a hranách.

6.9.3 Struktura XML dokumentu

Zde se zmíníme ještě o struktuře výstupního souboru XML. Pro lepší pochopení vždy vložím ukázku, kterou si následně popíšeme.

```
<?xml version="1.0" encoding="UTF-8" ?>

<rozestaveni>
<bod index="0" x="83" y="41" popis="uzel 0" red="39" green="51" blue="28"/>
<bod index="1" x="27" y="83" popis="uzel 1" red="11" green="61" blue="20"/>
<pomocny_bod index="0" x="14" y="83" popis="pomBod" red="0" green="0" blue="0"/>
<pomocny_bod index="1" x="17" y="83" popis="pomBod" red="0" green="0" blue="0"/>
<hrana bod1="0" bod2="1" red="0" green="0" blue="0"/>
</rozestaveni>
```

Tabulka 6.3: Ukázka struktury XML dokumentu

V naší ukázce máme uloženy jen dva uzly spojené hranou a dva pomocné uzly. Element začínající `<bod ... />` uchovává informace o uzlu, konkrétně pořadové číslo, umístění, popis a barvu rozloženou do tří složek RGB modelu. Element `<pomocny_bod ... />` zahrnuje informace o pomocném uzlu. Tyto informace mají stejnou strukturu jako u normálního uzlu. Poslední element, v souboru je `<hrana ... />`. O hraně si uchováváme jen které dva uzly spojuje a její barvu.

Kapitola 7

Závěr

Důkladně jsem prostudoval uvedenou literaturu týkající se propojovacích sítí, směrovacích algoritmů, které se zde využívají a algoritmů pro vhodné zobrazení grafů. Následnou implementací v jazyku Java vznikl nástroj, který je schopen vykreslit diagram topologie sítě a znázornit komunikaci v síti. Program je tedy vhodný pro ilustraci a pochopení principů komunikace v paralelních sítích.

Při implementaci programu jsem narazil na několik problémů, které až na jednu výjimku se mně podařilo více či méně úspěšně vyřešit. Jediný problém, který jsem i přes velké úsilí nevyřešil je implementace pružinového algoritmu využívaného pro přehledné rozvržení grafu. I když jsem literaturu o tomto algoritmu důkladně studoval a pochopil princip, nepovedlo se mi algoritmus uvést do provozuschopného stavu. V zadání bakalářské práce nebylo uvedeno, že tento algoritmus musí být implementován, přesto to ve mně zanechalo negativní pocity. Program umí znázornit směrovací algoritmy, čímž byl splněn hlavní požadavek na práci.

Žádný program není bez chyb, tedy ani ten můj. Chyby se najdou vždy, některé dříve a jiné později. Program byl vyzkoušen na počítačích s operačním systémem Windows XP, Windows Vista a Ubuntu. Na žádné z těchto platforem, nebyl sledován nějaký významný problém. Zdrojový kód programu naleznete na přiloženém CD, nachází se tam také zkompilovaná funkční verze a také vstupní soubory s topologiemi a směrovacími algoritmy.

Jak jsem již uvedl, kromě nepovedené implementace pružinového algoritmu, jsem s programem vcelku spokojen. Jako možné rozšíření při pokračování bych uvedl právě implementaci tohoto algoritmu. Vzhledem k tomu, že právě tento algoritmus se měl starat o úpravu rozvržení většiny grafů, není zde implementováno tolik dalších možností pro automatické rozvržení grafu. Dalším možným rozšířením je zvětšení možností nastavení zobrazení topologie, např. vybarvení uzlů pomocí textury, případně upravit algoritmus, který znázorňuje komunikaci v síti.

Hlavním přínosem bakalářské práce pro mě je seznámení se s oblastí informatiky, kterou jsem dosud vůbec neznal, konkrétně principy komunikace mezi procesy u paralelního programování. Aktuálně znám sice jen malou teoretickou část tohoto problému, avšak v budoucnu bych rád získal ještě další informace z této oblasti informatiky, neboť v dnešní době je paralelní programování více využívané, než tomu bylo v minulosti, zvláště také z důvodu více jádrových procesorů. Dalším velkým přínosem je praktická zkušenost s grafickým prostředím jazyku Java, neboť do této doby jsem se setkal jen s vytvářením programů bez grafického prostředí.

Literatura

[1] *dom4j 1.6.1 API - knihovna pro práci s XML* [online]. 2005-05-16 [cit. 2009-15-05], dostupné na URL http://www.dom4j.org/.

[2] Dally, W.; Towles, B.: *Principles and practices of interconnection networks.* Morgan Kaufmann Publishers, 2004, iSBN 0-12200-751-4.

[3] Davidson, R.; Harel, D.: *Drawing graphs nicely using simulated annealing* [online]. 1996 [cit. 2009-15-05], dostupné na URL http://www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/DrawingGraphsNicely.pdf.

[4] Demel, J.: *Grafy a jejich aplikace.* Academia, 2002, iSBN 80-200-0990-6.

[5] Duato, J.; Yalamanchili, S.; Lionel, N.: *Interconnection networks: an engineering approach.* Morgan Kaufmann Publishers, 2003, iSBN 1-55860-852-4.

[6] Eades, P.: *A Heuristic for Graph Drawing* [online]. 1984 [cit. 2009-15-05], dostupné na URL http://www.it.usyd.edu.au/~peter/old_spring_paper.pdf.

[7] Fruchterman, T. M. J.; Reingold, E. M.: *Graph Drawing by Force-directed Placement* [online]. 1991-11-01 [cit. 2009-15-05], dostupné na URL http://www.cs.ubc.ca/local/reading/proceedings/spe91-95/spe/vol21/issue11/spe060tf.pdf.

[8] Gross, J. L.; Yellen, J.: *Graph Theory and Its Applications, Second Edition.* Chapman & Hall/CRC, 2006, iSBN 1-58488-505-X.

[9] Herout, P.: *Java - grafické uživatelské prostředí a čeština.* KOPP, 2001, iSBN 80-7232-150-1.

[10] Herout, P.: *Učebnice jazyka Java.* KOPP, 2004, iSBN 80-7232-115-3.

[11] Horton, I.: *Java 5.* Wrox Press, 2002, iSBN 1-861005-69-5.

[12] Palúch, S.: *Skripta z teorie grafov* [online]. 2001-05-07 [cit. 2009-15-05], dostupné na URL http://frcatel.fri.utc.sk/users/paluch/index.php.

[13] Sun Microsystems, I.: *The Java Tutorials* [online]. 2008-03-14 [cit. 2009-15-05], dostupné na URL http://java.sun.com/docs/books/tutorial/.

[14] Václav., D.: *Parallel systems architecture and programming.* FIT VUT v Brně, 2008, opora kurzu ARC v anglickém jazyce.

Dodatek A

Obsah CD

- Textová část bakalářská práce ve formátu Adobe Acrobat PDF.
- Zdrojové kódy textové části pro program LaTeX.
- Zdrojové kódy programu ve formě NetBeans project.
- Zdrojové kódy programu připravené pro překlad pomocí programu Ant.
- Přeložený program k možnému spuštění.
- Vstupní soubory topologií a směrovacích algoritmů.

Dodatek B

Manual

Pro spuštění programu je potřeba mít na počítači nainstalované prostředí Java SE Runtime Environment verze 6. Program byl koncipován k intuitivnímu ovládání, které je běžné v dnešních programech. Požadovaného výsledku lze dosáhnout ve čtyřech krocích:

1. Nejdříve je zapotřebí načíst topologii, kterou máme uloženou v souboru. Toho dosáhneme stisknutím tlačítka *Otevři* v menu *Topologie*.

2. Po úspěšném načtení se na vykreslovací ploše zobrazí graf topologie, zatím však jen v náhodně zvolené pozici. Nyní můžeme pomocí některého algoritmu z nabídky Topologie upravit rozložení grafu, případně si graf upravit ručně.

3. Jakmile jsme dokončili úpravu rozmístění uzlů, může z druhého souboru, který obsahuje popis směrovacího algoritmu, vykreslit komunikaci mezi jednotlivými uzly. Pro otevření směrovacího algoritmu musíme stisknout tlačítko *Načti* v menu směrování.

4. Následně můžeme uložit dosažené výsledky buď jako obrázek, nebo ve formě XML dokumentu uložit pouze upravené rozložení topologie bez směrové komunikace. Pro ukládání výsledků jsou v nabídce Soubor tlačítka *Ulož obrázek* a *Ulož Xml*. Výsledek může vypadat např. takto B.6.

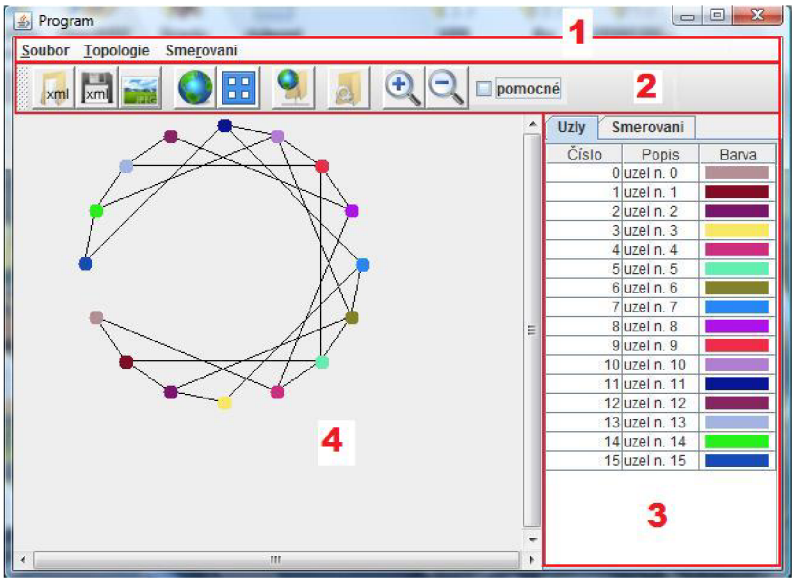
V programu jsou ještě další tlačítka, které dosud nebyly uvedeny. Nyní si je všechny postupně popíšeme. Pro lepší popis je hlavní okno programu rozdělené do čtyř částí B.1.

B.1 Hlavní menu

Jak je vidět na obrázku B.2 má hlavní menu tři nabídky. V nabídce soubor se nachází pět tlačítek, v nabídce Topologie šest a v nabídce směrování pouze jedna.

B.1.1 Nabídka Soubor

- Nový – vymaže rozpracovaný graf z vykreslovací plochy
- Otevři Xml – otevře dokument XML s již dříve uloženou topologii
- Ulož Xml – uložíme aktuálně zobrazenou topologii do dokumentu XML
- Ulož Obrázek – umožní nám aktuálně vykreslený graf uložit jako obrázek formátu jpeg
- Konec – ukončí program



Obrázek B.1: Okno programu

Obrázek B.2: Hlavní menu programu

B.1.2 Nabídka Topologie

- Otevři – otevře soubor s topologií a náhodně vykreslí graf
- Mesh – změní rozmístění grafu do topologie typu mřížka
- Torus – změní rozmístění grafu do topologie typu torus resp. toroid
- Ring – změní rozmístění grafu do topologie typu kruh
- Spring – využije pružinového algoritmu pro upravení rozmístění grafu (algoritmus nepracuje správně)
- Nastavení – otevře nové okno, kde je možné upravit možnosti zobrazení grafu

B.1.3 Směrování

Nabídka Směrování obsahuje pouze tlačítko *načti*, které se využívá pro načtení směrovacího algoritmu ze souboru. Na obrázku jste si jistě všimly, že všechny tlačítka mají klávesové zkratky pro urychlení práce s programem.

B.2 Nástrojová lišta

Nástrojová lišta obsahuje tlačítka pro rychlé použití některých funkcí. Většina tlačítek, které se nacházejí na nástrojové liště, je umístěna i v hlavní nabídce programu.

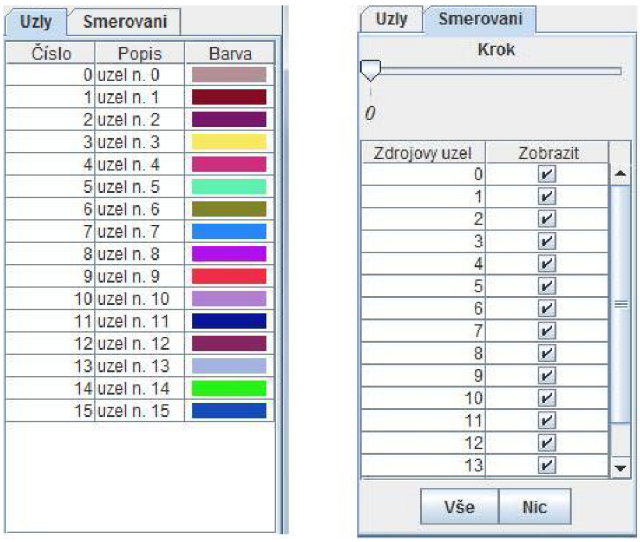
Obrázek B.3: Nástrojová lišta programu

Tlačítka na liště si nyní popíšeme (číslování prvku je podle obrázku B.3):

1. Otevře dokument XML s již dříve uloženou topologii.
2. Uložíme aktuálně zobrazenou topologii do dokumentu XML.
3. Uloží aktuálně vykreslený graf jako obrázek formátu jpeg.
4. Otevře soubor s topologií a náhodně vykreslí graf.
5. Změní rozmístění grafu do topologie typu mřížka.
6. Otevře soubor obsahující informace o směrovacím algoritmu.
7. Otevře nové okno, kde je možné upravit možnosti zobrazení grafu.
8. Tlačítko přiblíží aktuálně vykreslený graf, toto tlačítko má klávesovou zkratku Ctrl+PageUp.
9. Tlačítko oddálí aktuálně vykreslený graf, toto tlačítko má klávesovou zkratku Ctrl+PageDown
10. Zatrhnutí této možnosti nám umožní zobrazit pomocné body, pomocí kterých lze upravovat hrany mezi uzly.
11. Výběrem možnosti zmraz, se při pohybu s hlavním uzlem budou pohybovat i pomocné uzly, při výběru uvolni, pohyb hlavního uzlu neovlivňuje pomocné uzly.

B.3 Záložky s nastavením

Pro přehledné ovládání a možnosti rychlé úpravy vlastností grafu se zde nachází dvě záložky. První záložka nám umožňuje upravit barvu jednotlivých uzlů, druhá upravuje možnosti znázornění směrovacího algoritmu. Jak je vidět na obrázku B.4, ve druhé záložce si můžeme vybrat jen ty uzly, u kterých chceme vidět jejich komunikaci. Dále je zde na výběr možnost kolik kroku chceme zobrazit. Zobrazení komunikace může probíhat dvěma způsoby. Buď chceme zobrazit pouze aktuální krok komunikace, nebo chceme zobrazit aktuální krok a kroky předešlé. Toto lze nastavit v nastavení programu B.5.



Obrázek B.4: Záložky programu

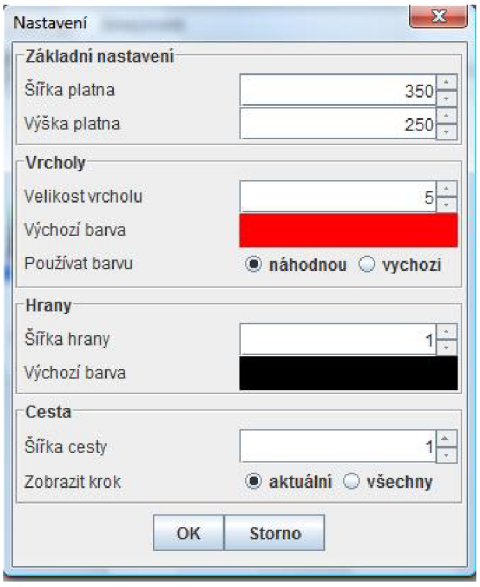
B.4 Vykreslovací plocha

Na vykreslovací ploše je vždy zobrazen diagram grafu. Pomocí myši je možné tento graf ručně upravovat. V kreslící ploše se vyskytují dva druhy uzlů. Hlavní uzly jsou znázorněny jako kolečka vykresleny různými barvami, pomocné uzly jsou znázorněný jako černé čtverce. Jak to vypadá je vidět na obrázku B.6.

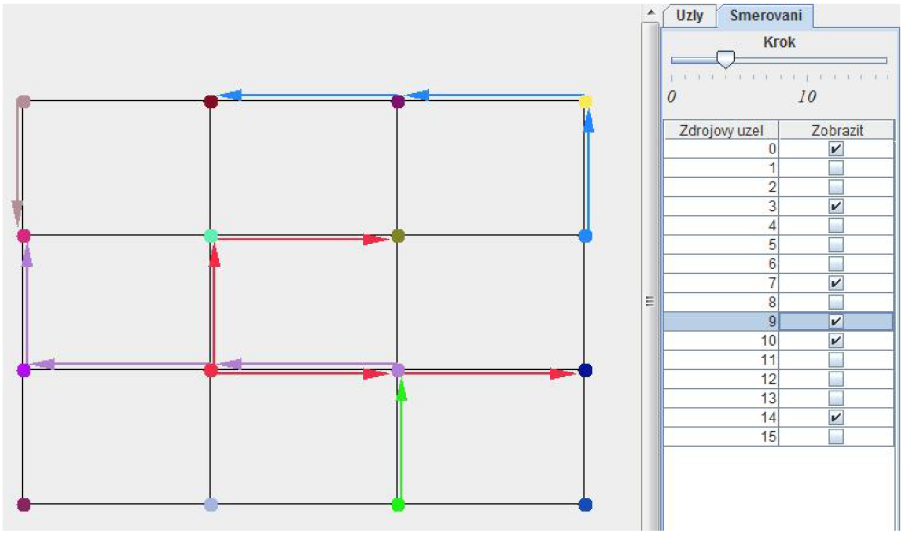
B.5 Nastavení

Posledním důležitým prvkem programu je okno s nastavením zobrazeníB.5.

Zde je možné nastavit rozměry vykreslovací plochy, vlastnosti vrcholu, hran a cest, které znázorňují komunikaci mezi uzly. U uzlů je konkrétně možné změnit jejich velikost, výchozí barvu, pokud bychom při vytváření topologie chtěli všechny uzly stejné a možnost výběru zda budou uzly mít náhodnou barvu či výchozí. V nastavení hran můžeme změnit jejich šířku barvu. Poslední možností je změna šířky cesty a výběr, jestli budeme chtít zobrazit vždy jen aktuální krok směrovacího algoritmu nebo i kroky předešlé.



Obrázek B.5: Okno nastavení



Obrázek B.6: Příklad možného výstupu grafu