VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

AUTOMATICKÉ UMÍSŤOVÁNÍ UZLŮ V ACYKLICKÉM ORIENTOVANÉM GRAFU DO GUI

AUTOMATIC NODE-PLACEMENT IN AN ORIENTED ACYCLIC GRAPH IN A GUI APPLICATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **JAN JUDA**
AUTHOR

VEDOUCÍ PRÁCE **Doc. Dr. Ing. DUŠAN KOLÁŘ**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Juda Jan**
Program: Informační technologie
Název: **Automatické umísťování uzlů v acyklickém orientovaném grafu do GUI**
Automatic Node-Placement in an Oriented Acyclic Graph in a GUI Application
Kategorie: Algoritmy a datové struktury

Zadání:

1. Nastudujte problematiku automatického rozmísťování uzlů v orientovaných acyklických grafech. Seznamte se s existujícími algoritmy. Zaměřte se na algoritmy s preferovaným umístěním zdrojových a cílových uzlů.
2. Navrhněte aplikaci, která bude podporovat automatické rozmísťování uzlů v acyklických grafech a bude podporovat podgrafy, polygonální uzly, umísťování uzlů, volbu mezi různými algoritmy. Vezměte do úvahy, že aplikace musí pracovat jak v režimu s GUI, tak z příkazové řádky.
3. Navrženou aplikaci implementujte - jako vstup a výstup uvažte specifický formát XML.
4. Otestujte aplikaci na netriviálních vstupech čítajících i stovky uzlů.
5. Zhodnoťte přínos své práce, diskutujte možná rozšíření, případné nedostatky.

Literatura:

- Demel: Grafy a jejich aplikace, Academia, 2002.
- Dle doporučení vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání a rozpracovaný bod 3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Kolář Dušan, doc. Dr. Ing.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 21. října 2019

Abstrakt

Cílem této práce je vytvořit aplikaci pro automatické rozmísťování uzlů v acyklických orientovaných grafech. Práce se především zaměřuje na pokročilé možnosti při tvorbě umístění uzlů, z kterých za zmínku stojí výběr polohy vybraných uzlů, rozdělení grafu na podgrafy či podporu polygonálních uzlů.

V řešení jsou popsány vybrané algoritmy, které jsou použity ve výsledné aplikaci, a to konkrétně Fruchterman-Reingoldův silou orientovaný algoritmus, algoritmus Kamada-Kawai a algoritmus založený na Meyerových metodách samo-organizujících se grafů.

Abstract

The goal of this work is to create an application for automatic node placement of acyclic oriented graphs. The work is mainly focusing on advanced possibilities of graph layout, for example selection of location of selected nodes, division of a graph into sub-graphs or support of polygonal nodes.

The solution describes chosen algorithms, which are being used in the resulting application. Specifically, Fruchterman-Reingold force oriented algorithm, algorithm Kamada-Kawai and an algorithm based on Meyer's self-organizing graphs.

Klíčová slova

Graf, grafy, grafové algoritmy, rozmísťovací algoritmy, acyklické grafy, orientované grafy, umístění uzlů, rozmístění uzlů, Fruchterman-Reingold, Kamada-Kawai, ISOM, Meyerovy metody samo-organizujících se grafů, silou řízené algoritmy, složitost, rovinný graf

Keywords

Graph, graph algorithms, layouting algorithms, acyclic graphs, oriented graphs, node placement, graph layout, Fruchterman-Reingold, Kamada-Kawai, ISOM, self-organizing graphs, force directed algorithms, time complexity, planar graph

Citace

JUDA, Jan. *Automatické umísťování uzlů v acyklickém orientovaném grafu do GUI.* Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Doc. Dr. Ing. Dušan Kolář

Automatické umísťování uzlů v acyklickém orientovaném grafu do GUI

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana docenta Dušana Koláře. Další informace a podporu mi poskytl pan Mgr. Tomáš Lestyan. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....

Jan Juda

27. května 2020

Obsah

Kapitola 1

Úvod

Práce se zabývá problematikou rozmísťování uzlů v acyklických orientovaných grafech. Hned na začátek mi přijde důležité vysvětlit základní pojmy použité v názvu. Grafem rozumíme 2 množiny — množinu uzlů a množinu hran, které spojují tyto uzly. Acyklický graf je takový, ve kterém se nevyskytují žádné cykly (kružnice). Orientovaný graf je takový, ve kterém mají všechny hrany určený směr.

Proč se vůbec zabývat rozmísťováním uzlů v grafech? Předpokládejme, že máme graf, který chceme zobrazit, tak ho budeme muset určitým způsobem umístit do roviny. To lze provést mnoha způsoby, například náhodně, ale pokud chceme, aby byl graf srozumitelný, pochopitelný a aby po vizuální stránce vypadal „hezky“, tak budeme muset přijít s nějakou sofistikovanější metodou. Další otázkou je, jak vůbec definovat, co je to hezké čitelné rozmístění uzlů grafu. Nejspíše bude nutné zavést nějaká měřítka a požadavky, například že nechceme, aby se uzly grafu překrývaly. Jiným požadavkem může být to, aby se nám nekřížily hrany grafu, nebo pokud je to nevyhnutelné (což často bude), tak aby se nám křížily co nejméně. A zde narážíme na první kámen úrazu, neboť určení minimálního počtu křížení hrany s hranou v grafu je výpočetně dosti složitý problém. Naštěstí existují algoritmy, které, přestože nemohou zajistit perfektní výsledek, uzly grafu dokáží do roviny umístit tak, že výsledek vypadá velmi dobře.

Tato práce byla řešena ve spolupráci s nadnárodní firmou NXP Semiconductors Czech Republic s.r.o. (dále jen NXP). Firma NXP vyrábí mikrokontroléry a vestavěné systémy, ke kterým poskytuje a vyvíjí konfigurační nástroje. Tyto nástroje v sobě obsahují nejrůznější diagramy (grafy) zobrazující jednotlivé součástky mikrokontrolérů a propojení mezi nimi. Například nástroj „Clock tool“ zobrazuje diagram hodin, na kterém se vlevo nachází zdroje hodinového signálu, vpravo výstupy hodin a uprostřed všechny ostatní součástky upravující hodinový signál. Navíc jsou součástky v takovémto diagramu rozděleny a seskupeny do různých komponent. V současnosti jsou z větší části tyto grafy kresleny lidmi a aplikace, která je výsledkem tohoto díla, by měla přinést výraznou změnu a generovat výše zmíněné diagramy automaticky, což ušetří hodně lidské práce a tudíž i peněz. Aby byl výsledný program skutečně použitelný, bude muset být schopný vyhovět všem těmto specifickým požadavkům na výsledné rozložení grafů, které vyplývají z výše popsané situace. Mezi nejvýznamnější požadavky můžeme zařadit umístění vybraných uzlů na určitý okraj grafu (například všechny výstupy vpravo), rozdělení grafu do předdefinovaných podgrafů (ze součástek jsou složeny větší komponenty), podpora uzlů polygonálních tvarů a z toho vyplývající podpora uzlů různé velikosti. K těmto požadavkům musíme samozřejmě přidat požadavek na to, aby aplikace fungovala pro obecné orientované acyklické grafy a nejen pro ty, co se používají v NXP.

Dílo je rozděleno do šesti kapitol: Úvod, Nastudované poznatky a teorie, Návrh aplikace, Popis řešení a implementace, Zhodnocení práce a Závěr. V kapitole „Nastudované poznatky a teorie“ najdete popis algoritmů použitých pro tvorbu řešení a stejně tak veškerou potřebnou teorii. V návrhu aplikace se autor věnuje tomu, jak byla aplikace navržena, popisu vstupů a výstupů, rozdělením aplikace do jednotlivých modulů a popisu činnosti těchto modulů. Ve čtvrté kapitole je pak podrobně popsáno, jak byla aplikace naimplementována. Kapitola Zhodnocení práce obsahuje zhodnocení výsledků a srovnání jednotlivých použitých algoritmů. V Závěru je shrnut obsah práce a jsou zde také nastíněny možnosti, jakými směry by se práci dalo rozvíjet či v ní pokračovat.

Kapitola 2

Nastudované poznatky a teorie

Práce se pohybuje v oblasti teorie grafů. V této kapitole shrnu pouze nezbytně nutné poznatky pro pochopení práce a při tomto shrnutí budu vycházet především z knihy Grafy a jejich aplikace [1], ve které lze najít podrobnější a více ucelený popis zkoumané teorie.

V úvodní kapitole 1 jsme si již vysvětlili, co znamenají pojmy graf, acyklický graf a orientovaný graf. Dále jsme v té samé kapitole narazili na zmínku o složitosti problému určení minimálního počtu křížení hran s hranou v grafu, čímž jednak narážíme na teorii složitosti, a jednak přicházíme k tématu rovinnosti grafů (anglicky *planarity*). Jelikož bych rád upozornil na to, jak moc jsou některé problémy a úlohy týkající se grafů složité, tak jsem si sem dovolil vložit malou vsuvku o složitosti, a hned po ní následuje výklad teorie grafů.

Složitost

V teorii složitosti se v kontextu algoritmů bavíme o takzvané asymptotické složitosti, jejíž definici jsem se zde rozhodl neuvádět, jelikož se v porovnání se zde probíranou teorií jedná o relativně triviální látku. V případě zájmu si čtenář může danou definici dohledat v ostatní literatuře, například ve výše zmíněné knize. Zajímavější částí teorie složitosti jsou třídy složitosti úloh, pro jejichž vysvětlení bude nutné definovat dva typy úloh. Prvním typem jsou úlohy optimalizační, kde hledáme nejlepší (neboli optimální) řešení daného problému. Výsledkem takové úlohy je typicky číslo. Druhým typem jsou úlohy rozhodovací, jejichž jméno vychází ze skutečnosti, že cílem takové úlohy je rozhodnout, zda něco platí. Výsledkem je tedy rozhodnutí „ano“ či „ne“.

Úloha, k níž existuje algoritmus, který ji dokáže vyřešit deterministicky v polynomiálním čase (to je takový algoritmus, který má polynomiální složitost), je polynomiální úloha a všechny takovéto rozhodovací úlohy tvoří třídu úloh P. Všechny polynomiální úlohy se obecně označují jako lehce řešitelné, což souvisí s tím, že algoritmy s polynomiálním časem je možné na dnešních počítačích provádět dostatečně rychle a typicky takové algoritmy končí v rozumném čase. Dále platí, že každá úloha, kterou lze polynomiálně redukovat (to znamená převést pomocí polynomiálního algoritmu) na úlohu třídy P, je také úloha třídy P. To znamená, že třída úloh P obsahuje i některé optimalizační úlohy.

Třídou NP úloh pak rozumíme všechny rozhodovací úlohy, které jsou nedeterministicky řešitelné v polynomiálním čase. Dále rozlišujeme další dvě třídy úloh, a to NP-úplné a NP-těžké úlohy. Pro úlohy patřící do obou těchto tříd je společná podmínka taková, že na ně lze libovolná úloha z třídy NP polynomiálně redukovat. Rozdíl je v tom, že NP-úplná úloha sama musí patřit do třídy NP úloh (tedy musí to být rozhodovací nedeterministicky polynomiální úloha) a NP-těžká úloha do třídy NP patřit nemusí. Z toho vyplývá, že třída

NP-těžkých úloh obsahuje jak rozhodovací, tak optimalizační úlohy, zatímco třída NP-úplných úloh obsahuje jen a pouze úlohy rozhodovací.

Třída úloh P je podmnožinou třídy úloh NP. Obecně se věří, že se tyto dvě třídy nerovnají, což ale zatím není matematicky dokázáno.

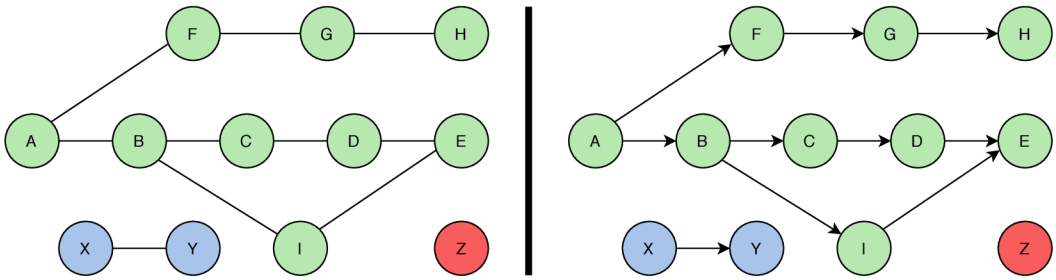
Pokud se vrátíme zpět k úloze určení maximálního počtu křížení hran s hranou v grafu, tak z uvedených definic snadno můžeme vyvodit, do jaké třídy složitosti tato úloha spadá. Jestliže se tedy ptáme na počet, tak se jasně jedná o optimalizační úlohu, a když k této znalosti přidáme informaci o tom, že pro tento problém není známý algoritmus deterministicky řešitelný v polynomiálním čase, tak můžeme říct, že tato úloha patří do třídy NP-těžkých úloh. V teorii grafů lze narazit na mnoho dalších složitých NP-úplných či NP-těžkých úloh.

2.1 Teorie grafů

Hned ze začátku bych rád znovu shrnul základní pojmy teorie grafů, podíval se na základní klasifikaci grafů a postupně přešel k rozmísťování uzlů a k rozmísťovacím algoritmům. Rozhodl jsem se, že nebudu doslovně citovat matematické definice uvedených pojmů, a raději je vysvětlím svými slovy. Tyto přesné definice je možné najít po nahlédnutí do jiné literatury [1].

Graf

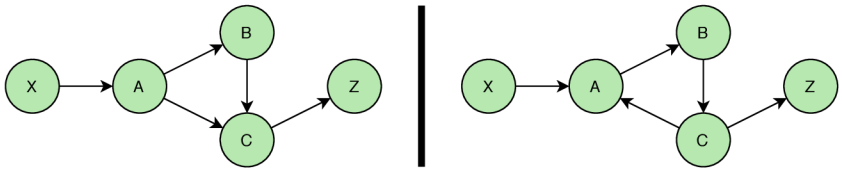
Graf si představme jako množinu uzlů a množinu hran, který spojují tyto uzly. Každá hrana spojuje právě dva uzly, přičemž může spojovat uzel sám se sebou — taková hrana se pak nazývá **smyčka**. V případě, že nás zajímá směr hrany, tak máme hranu orientovanou. U orientované hrany poté určujeme počáteční a koncový uzel. V opačném případě máme hranu neorientovanou, která nemá žádný směr. Graf může obsahovat hrany jak orientované, tak neorientované. Pokud máme graf, který obsahuje čistě orientované hrany, tak takový graf nazýváme **orientovaný graf**. Podobně graf pouze s neorientovanými hranami nazýváme **neorientovaný graf**. Příklad neorientovaného a orientovaného grafu lze vidět na obrázku 2.1. V této práci se budeme soustředit především na orientované grafy.



Obrázek 2.1: **Příklad neorientovaného a orientovaného grafu.** Na obrázku vidíme dva téměř totožné grafy oddělené tlustou svislou čarou. Graf vlevo je neorientovaný a obsahuje pouze neorientované hrany. Graf vpravo je orientovaný, protože obsahuje pouze orientované hrany. Graf vpravo je navíc acyklický.

Acyklický graf

Další z pro nás zajímavých vlastností grafu je, zda obsahuje **kružnice**. Kružnici (neboli cyklus) v grafu můžeme chápat tak, že pokud graf procházíme podél na sebe navazujících hran a dojdeme znovu do uzlu, který jsme již navštívili, tak jsme našli kružnici. Příklad kružnice by mohl být orientovaný graf s uzly `A`, `B` a `C` a hranami `A -> B` (čtěte hrana z uzlu `A` do uzlu `B`), `B -> C` a `C -> A`. Pokud takový graf začneme procházet z uzlu `A`, tak dojdeme do uzlu `B`, poté do uzlu `C` a nakonec se vrátíme do uzlu `A`. Tento graf tedy tvoří cyklus. Kružnice může obsahovat neomezený počet uzlů. Speciální případ kružnice je výše zmíněná smyčka, která má pouze jeden uzel. Graf, který nemá žádnou kružnici ani smyčku, se nazývá **acyklický graf**. Opakem je **cyklický graf**, který alespoň jednu kružnici obsahuje. Příklady obou těchto typů grafu můžete najít na obrázku 2.2.



Obrázek 2.2: **Příklad acyklického a cyklického grafu.** Na obrázku vidíme dva orientované grafy, které se liší pouze orientací hrany `A -> C`. Graf vlevo je acyklický — neobsahuje žádnou kružnici. Graf vpravo je cyklický, protože uzly `A`, `B` a `C` a hrany, které je spojují tvoří kružnici.

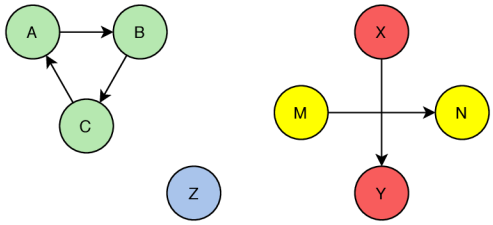
Přítomnost kružnic v grafu je pro tuto práci důležitá z toho důvodu, že pokud chceme graf rozmístit, tak musíme graf projít a každému uzlu určit pozici. Právě způsob, jakým budeme graf procházet, je ovlivněn přítomností cyklů. V acyklickém grafu se nám totiž nemůže stát, že se při použití jednoduššího algoritmu pro procházení grafu zacyklíme. V cyklickém grafu bychom si na přítomnost cyklů museli dávat pozor a zohlednit to při jeho procházení.

Souvislý graf

Souvislý graf je takový graf, který má každé dva uzly propojené cestou (tvořenou z hran a jiných uzlů), přičemž nezáleží na orientaci hran. Souvislost je v tomto díle zmíněna jednak z toho důvodu, že tento pojem budeme potřebovat pro vysvětlení další termínů, a jednak proto, že k nesouvislému grafu můžeme při vykreslování přistupovat tak, že budeme zpracovávat každou jeho souvislou komponentu (maximální souvislý podgraf) zvlášť. Souvislé komponenty můžeme názorně vidět na obrázku 2.3.

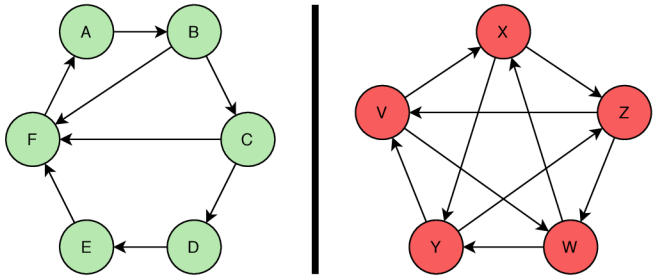
Rovinné grafy

Důležitá vlastnost, která nás při kreslení grafů může zajímat a objevuje se při popisování následujících algoritmů, je rovinnost grafu. **Rovinný graf** (planární graf) je takový graf, který lze do roviny zakreslit bez křížení hran. Samozřejmě je možné i rovinný graf zakreslit tak, aby ke křížení hran došlo, ale hlavní je to, že minimální počet křížení hran je pro rovinný graf vždy 0. Častou úlohou spojenou s rovinnými grafy je, zjistit zda je graf rovinný, a najít jeho rovinné zakreslení. Rovinný graf spolu s jeho rovinným zakreslením se nazývá **topologický rovinný graf**. Pro každý souvislý topologický rovinný graf, který má `n` uzlů, `m` hran a `s` stěn (pro definici stěny grafu nahlédněte do [1]), platí **Eulerova formule** `n + s = m + 2`. Tuto formuli můžeme rozšířit pro nesouvislé topologické rovinné grafy



Obrázek 2.3: **Nesouvislý graf s čtyřmi souvislými komponentami.** Tento graf má čtyři barevné oddělené souvislé komponenty. První `A`, `B` a `C`. Druhá `Z`. Třetí `M` a `N`. Čtvrtá `X` a `Y`. Přestože se hrany `M -> N` a `X -> Y` kříží, tak se jedná o dvě samostatné komponenty, protože jednotlivé dvojice těchto uzlů nejsou nijak vzájemně propojeny.

s `k` souvislými komponentami na `n + s = m + k + 1`. Eulerova formule nám může pomoci k nalezení rovinného zakreslení grafu. Na obrázku 2.4 můžeme vidět rozdíl mezi rovinným a nerovinným grafem. Další příklady rovinných grafů lze vidět na obrázcích 2.1 a 2.2.



Obrázek 2.4: **Příklad rovinného a nerovinného grafu.** Zelený graf vlevo je rovinný a je i rovinně zakreslen. Červený graf vpravo není rovinný graf a neexistuje k němu rovinné zakreslení. Tento konkrétní graf se nazývá K5, protože je to úplný graf (každý uzel je propojen hranou s každým dalším uzlem) a má 5 uzlů.

2.2 Rozmísťovací algoritmy

Pro rozmísťování uzlů bude potřeba využít rozmísťovací algoritmy a v této podkapitole se jim budu věnovat. Typicky se jedná o iterativní algoritmy, které opakovaně prochází graf a upravují pozice uzlů. V rámci této práce jsou použity již existující algoritmy a jejich knihovní implementace. Využívám jejich vlastností k dosažení požadovaného rozmístění. Více o tom, jak jsou jednotlivé algoritmy aplikovány ve výsledném programu, najdete ve 3. kapitole Návrh aplikace.

Dále následuje popis vybraných algoritmů. Popisuji zde výhradně algoritmy použité ve výsledném programu. Všechny tyto tři algoritmy jsou schopné pracovat i s cyklickými grafy. Proč jsem zvolil obecně funkční algoritmy a proč jsem nepoužil žádný algoritmus specifický pro acyklické grafy, je zdůvodněno v části Rozmísťování podgrafů. U každého tohoto algoritmu je uveden příklad vykreslení jednoduchého grafu bez jakýchkoliv pokročilých omezení, aby měl čtenář představu o tom, jak vlastně výsledek takového algoritmu může vypadat. Rozmístění grafu jsou vygenerována výslednou aplikací. Konkrétně bude použit tento vstup:

```
{
    Nodes: [A, B, C, D, E, F, G, H, I, X, Y, Z],
    Edges: [A->B, B->C, C->D, D->E, F->G, A->F, G->H, B->I, I->E, X->Y]
}
```

Tento graf neobsahuje žádné uzly s vyžádanou pozicí ani žádné podgrafy. Pro generování bylo použito tisíc iterací s náhodnou inicializací.

Silou řízené rozmísťovací algoritmy

V tomto a v následujícím odstavci se budu věnovat popisu silou orientovaných (či řízených) algoritmů, přičemž použiji znalosti načerpané z článku Spring Embedders and Force Directed Graph Drawing Algorithms [4]. Silou orientované rozmísťovací algoritmy jsou takové algoritmy, které pro umísťování uzlů přiřazují uzlům a hranám síly, které proti sobě určitým způsobem působí, a v ustáleném stavu těchto sil (či po dostatečném počtu iterací algoritmu) dosáhneme relativně vzhledného rozmístění. Tyto síly jsou často reprezentovány tak, že všechny uzly se od sebe vzájemně odpuzují podobně jako elektricky nabité částice dle Coulombova zákona. Hrany jsou poté popsány jako pružiny, které generují přitažlivou sílu mezi spojenými uzly. Tato přitažlivá síla vrcholů spojených hranou má ale svůj limit, stejně jako když v reálném světě přiblížíte konce pružiny příliš blízko k sobě, tak se začnou odpuzovat. Jednoduše lze tedy říci, že všechny spojené sousední uzly se přitahují a nespojené uzly se odpuzují. To vede ke skutečnosti, že hrany mají tendenci mít všechny stejnou či podobnou délku. Jedná se v podstatě o simulaci fyzikálního systému a pro porozumění těmto algoritmů většinou ani není nutná hluboká znalost teorie grafů.

Z těchto poznatků můžeme o silou řízených algoritmech říci hned několik věcí. Jednak algoritmy pro tvorbu rozmístění uzlů vychází pouze ze struktury grafu, a tudíž není žádná závislost na obsahu, který daný graf reprezentuje. Dále při zvolení vhodných rovnic a vztahů pro popis výše zmíněných sil zjistíme, že algoritmy často vytvářejí pro rovinné grafy rozmístění bez křížících se hran. A přestože zde síly popisujeme pomocí fyzikálních systémů, tak užité rovnice nemusejí nezbytně odpovídat těmto fyzikálním systému a mnoho silou orientovaných algoritmů tyto vztahy různě upravuje, aby došly k co nejlepším výsledkům.

Fruchterman-Reingoldův silou orientovaný algoritmus

Nyní když jsme si vysvětlili princip fungování silou orientovaných algoritmů obecně, tak můžeme přejít ke konkrétnímu popisu Fruchterman-Reingoldova algoritmu. Informace o tomto algoritmu jsem nasbíral z článku Graph Drawing by Force-directed Placement [2] a doporučuji si ho přečíst v případě nejasností či touhy po detailnějších informacích. Tento algoritmus byl navržen, aby fungoval pro obecné neorientované grafy a autoři při jeho tvorbě brali v potaz následující estetická kritéria: Je-li dáno prostorové ohraničení grafu, tak aby grafická reprezentace grafu zcela spadala do prostoru tohoto ohraničení, a aby v něm byly uzly rozloženy rovnoměrně, co nejnižší počet křížících se hran, hrany stejné délky a zobrazení vlastní symetrie grafu. Jejich hlavním cílem bylo získání rychlého a jednoduchého algoritmu.

Základní princip silou řízených algoritmů, že všechny uzly se odpuzují a uzly spojené hranou se přitahují, je v tomto algoritmu zachován. První rozdíl oproti ostatním podobným algoritmům je popis sil, které na uzly působí. Síly jsou popsány následujícími rovnicemi:

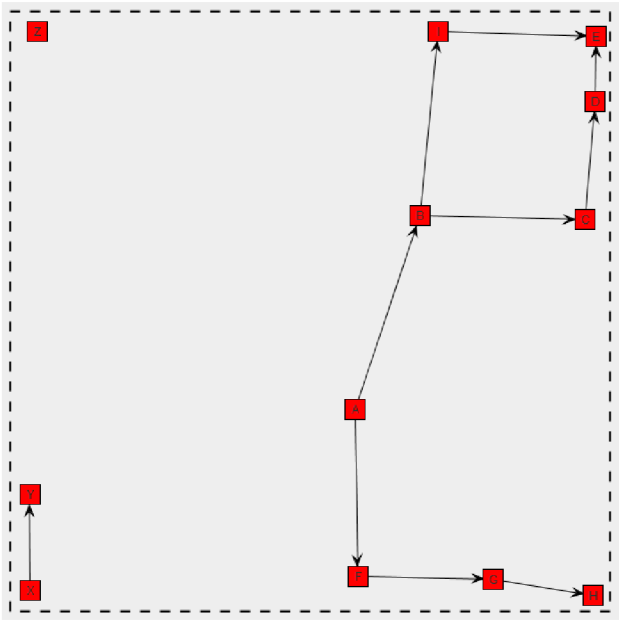
$$f_a(d) = \frac{d^2}{k} \tag{2.1}$$

$$f_r(d) = -\frac{k^2}{d} \tag{2.2}$$

Funkce f_a popisuje přitažlivou sílu (mezi sousedními vrcholy), zatímco funkce f_r popisuje sílu odpudivou (mezi všemi vrcholy). Obě tyto funkce mají parametr d, čímž se myslí vzdálenost mezi dvěma uzly. Symbol k označuje konstantu síly, která se pro každý jednotlivý graf vypočítává rovnicí

$$k = \sqrt{\frac{W * L}{V}} \tag{2.3}$$

kde W a L označují po řadě šířku a výšku ohraničení grafu. V je zde počet uzlů v grafu.



Obrázek 2.5: **Příklad rozmístění uzlů grafu Fruchterman-Reingoldovým algoritmem.** Graf byl vykreslený Fruchterman-Reingoldovým algoritmem za pomoci výsledné aplikace.

Další specifickou vlastností tohoto algoritmu je to, že při iterování používá upravenou metodu simulovaného žíhání. Pro omezení maximální délky posunutí vrcholu se zde používá takzvaná „teplota“, která se na konci každé iterace snižuje, a tudíž s postupem času je možné dělat pouze menší a menší posuny vrcholů. Vstupem algoritmu je graf, velikost ohraničení prostoru grafu a požadovaný počet iterací. Průběh algoritmu je možné popsat následovně: Na začátku se vypočítá konstanta k z rovnice (2.3), a pokud vstupní graf nemá přiřazené počáteční rozmístění uzlů, tak se uzly rozmístí náhodně. Každý vrchol je reprezentován dvěma vektory — pozicí a posunem. Posun všech vrcholů se na počátku každé iterace nastaví na nulu a dále se dá iterace vyjádřit třemi kroky. V prvním kroku se vypočítají odpudivé síly mezi vrcholy a výsledný posun pro každý vrchol se uloží. V druhém se pro všechny hrany spočítají přitažlivé síly a posun vrcholů vzniklý tímto výpočtem se přičte

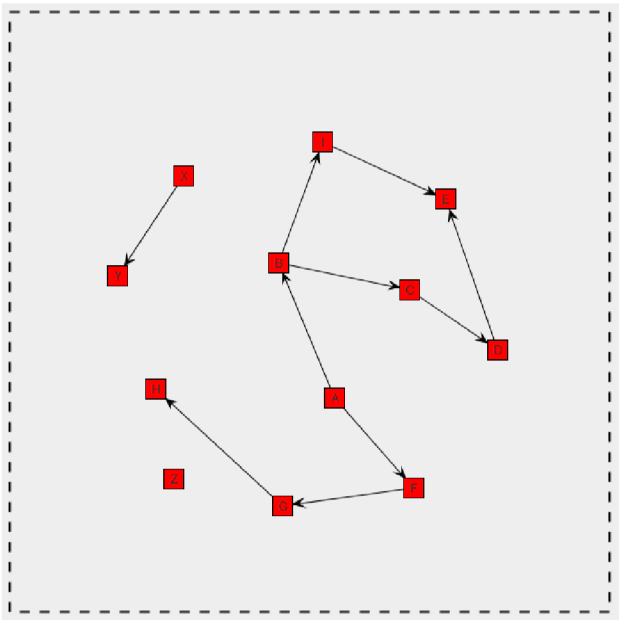
k posunu uloženém v předchozím kroku. V třetím kroku se nejdříve velikost posun každého vrcholu omezí výše popsanou „teplotou“, dále se provede samotné posunutí vrcholů — k vektoru pozice vrcholu se přičte vektor posunu, a nakonec se ještě opraví pozice vrcholů tak, aby se žádný z nich nenacházel mimo ohraničení grafu. Kompletní popis pseudokódu algoritmu lze najít v článku [2].

K dosáhnutí vyšší rychlosti se při výpočtu odpudivé síly nepočítá síla mezi všemi vrcholy, ale pouze mezi vrcholy vzdálenými maximálně $2k$. Toto zrychlení se projeví nejvíce, pokud jsou vrcholy v prostoru rozmístěny pravidelně.

Tento algoritmus je také specifický tím, jak se chová k okrajům grafu. Ty totiž lze reprezentovat různě, například jako pevné těleso generující stejně silnou odpudivou sílu, jakou generují vrcholy, či jako pevnou stěnu. V případě, že se rozhodneme brát okraje grafu jako stěnu, tak je zde otázka, jak se má vrchol zachovat, když do ní „narazí“. Autoři algoritmu se rozhodli, že nejlepší bude brát kraj grafu jako pevnou stěnu, která jednoduše blokuje pohyb za ni, takže když do ní vrchol narazí, tak se o ni zastaví a neodrazí se, ale může po ní klouzat (za předpokladu, že do ní nenarazil pod pravým úhlem). Tuto skutečnost můžeme dobře vidět na obrázku 2.5. Jednotlivé souvislé komponenty jsou od sebe odtlačovány, zatímco od stěn odtlačovány nejsou.

Algoritmus Kamada-Kawai

Algoritmus Kamada-Kawai je další z řady silou orientovaných algoritmů a je též navržen, aby fungoval obecně pro všechny neorientované grafy. Při popisu tohoto algoritmu vycházím především ze článku An Algorithm for Drawing General Undirected Graphs [3].



Obrázek 2.6: **Příklad rozmístění uzlů grafu algoritmem Kamada-Kawai.** Graf byl vykreslený algoritmem Kamada-Kawai za pomoci výsledné aplikace.

Tento algoritmus má podmínku, že vstupní graf musí být souvislý, přičemž pro nesouvislý graf lze rovinu rozdělit do částí a každou nesouvislou část řešit samostatně. Stěžejní myšlenkou tohoto algoritmu je to, že graf modelujeme jako dynamický systém částic (částice zde reprezentují vrcholy), které jsou všechny propojeny pružinami různé délky. Estetické

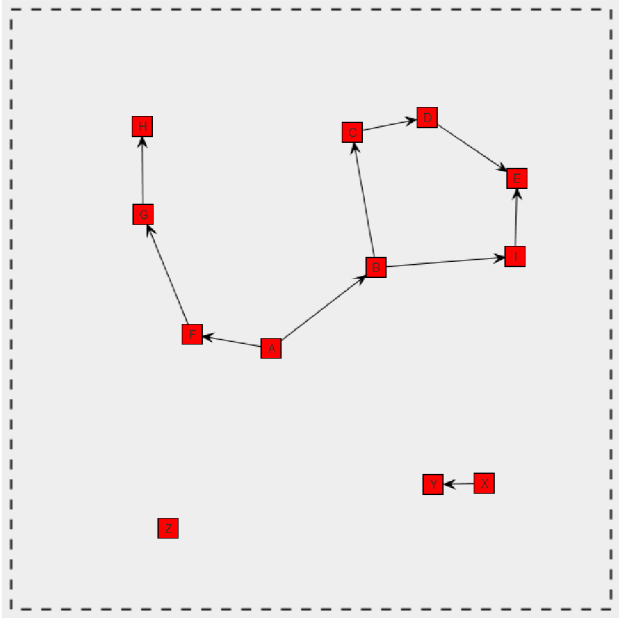
umístění uzlů grafu získáme minimalizováním energie pružin v takovémto systému. Autoři došli k tomu, že nejlepší výsledek lze získat, pokud je počáteční délka pružiny mezi dvěma vrcholy přímo úměrná délce nejkratší cesty mezi těmito vrcholy. Tím se navíc dosáhne toho, že tento algoritmus plně podporuje i kladně ohodnocené grafy (každé hraně je přiřazena kladná váha), protože výpočet nejkratší cesty uvažuje ohodnocení hran. Graf rozmístěný algoritmem Kamada-Kawai je zobrazen na obrázku 2.6.

Pro tento algoritmus je také specifické to, že v každé iteraci pohne pouze s jedním vrcholem, a to s tím, který nejvíce přispěje ke snížení energie systému, což vede k tomu, že v další iteraci stačí pouze přepočítat vliv tohoto uzlu a není nutné pokaždé přepočítat síly pro všechny uzly.

Meyerovy metody samo-organizujících se grafů

Princip Meyerových samo-organizujících se grafů přináší úplně nový pohled na rozmístění uzlů grafu. Tato metoda je založena na algoritmech kompetitivního učení[1]. Algoritmus založený na Meyerových metodách samo-organizujících se grafů bude v tomto díle dále označován zkratkou ISOM[2]. Tento algoritmus a jím využívané metody jsou detailně popsány v díle Self-Organizing Graphs — A Neural Network Perspective of Graph Layout [5], z kterého jsem čerpal poznatky, z nichž zde vycházím.

Základem tohoto algoritmu je myšlenka, že místo toho, aby se neuronová síť učila, jak náš graf vypadá, tak se přímo samotný graf transformuje na strukturu neuronové sítě. Vykreslení grafu tímto algoritmem lze najít na obrázku 2.7.



Obrázek 2.7: **Příklad rozmístění uzlů grafu algoritmem ISOM.** Graf byl vykreslený algoritmem ISOM za pomoci výsledné aplikace.

Zhodnocení implementací následujících algoritmů lze najít v kapitole Zhodnocení práce.

[1] kompetitivní učení (competitive learning) = typ strojového učení bez učitele (unsupervised learning)

[2] ISOM = Implementation of Self-Organizing Maps, česky Implementace samo-organizujích se map, zkratka pochází z dokumentace knihovny JUNG (viz B)

Kapitola 3

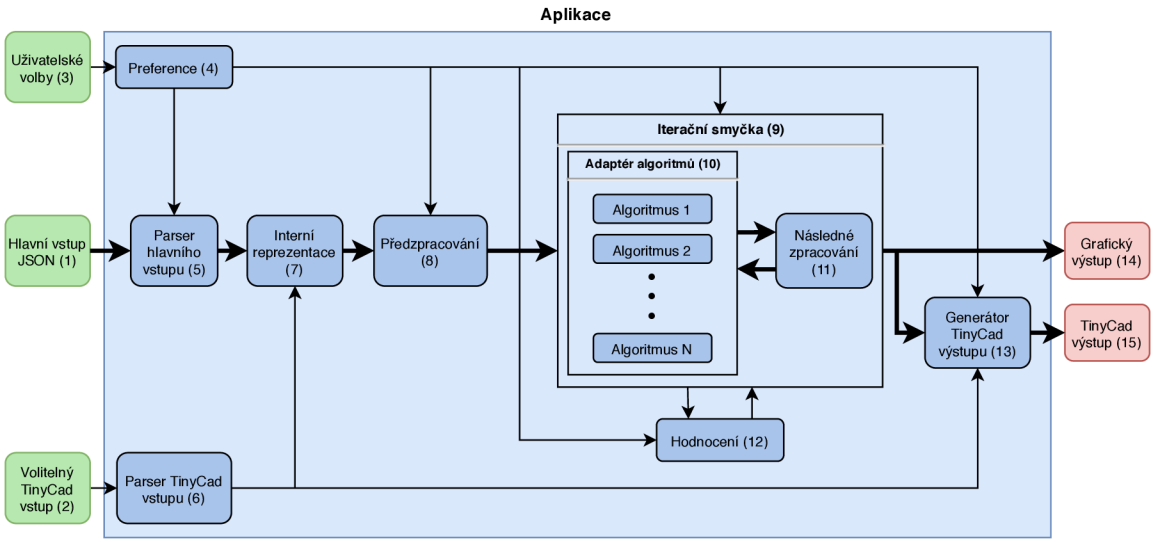
Návrh aplikace

V této kapitole se věnuji návrhu aplikace a její technické specifikaci. Nejdříve zde popisuji návrh celkové struktury programu, a poté v jednotlivých podkapitolách probírám dílčí moduly a jejich funkci.

Program by měl fungovat tak, že jako vstup obdrží textový popis grafu spolu s preferencemi uživatele na vzhled grafu, pomocí aplikace zvoleného algoritmu provede rozmístění grafu do roviny a nakonec výsledek uloží a zobrazí jej uživateli. Podrobnější informace ke vstupům a výstupům algoritmu naleznete v následujících podkapitolách.

3.1 Struktura aplikace

V rámci řešení jsem navrhnul následující strukturu programu, kterou můžete názorně vidět na přiloženém diagramu 3.1.



Obrázek 3.1: **Diagram struktury aplikace.** Diagram znázorňuje strukturu návrhu. Modře je označena samotná aplikace a její moduly. Zeleně jsou vyobrazeny vstupy, červeně pak výstupy. Tlusté šipky zobrazují průchod hlavních dat aplikací, kdežto tenčí šipky zobrazují průchod vedlejších vstupů. Jednotlivé části u sebe mají uvedena čísla v závorce, která jsou použita pro snadnější orientaci při popisování vyobrazených modulů.

Návrh uvažuje rozdělení zpracování hlavního vstupu, vedlejšího vstupu a uživatelských voleb. Data z hlavního vstupu se převedou do interní reprezentace. Poté se zpracuje vedlejší vstup a získané informace se přibalí do interní reprezentace. Ta se předá modulu předzpracování, který provede nad daty přípravné operace, a následně spustí a řídí iterační smyčku. V této smyčce se data opakovaně předají adaptéru vybraného algoritmu. Adaptér obecnou interní reprezentaci upraví do tvaru vyhovujícímu algoritmu, spustí algoritmus a vrátí výsledné rozmístění uzlů grafu v interní reprezentaci. Tu si poté přebere modul následného zpracování, který provede drobnější estetické úpravy pozic uzlů. Iterační smyčka poté pokračuje další iterací. Průběh iterování je navíc ovlivněn vyhodnocovacím modulem, který ohodnocuje kvalitu výsledku na základě preferencí uživatele. Po dokončení iterací získáme výsledky uložené v interní reprezentaci, která je poté graficky zobrazena, a generátor výstupu z ní vygeneruje výstup ve formátu programu TinyCAD[1] a uloží se do souboru specifikovaného uživatelem.

3.2 Vstupy aplikace

Nejdůležitějším vstupem je hlavní vstup (1) ve formátu JSON[2], který nese informaci o uzlech a hranách grafu, podgrafech a vyžádaných pozicích uzlů. Během vývoje si formát hlavního vstupu prošel mnoha změnami. Ze začátku jsem uvažoval vlastní nestandardizovaný formát vstupu. Později, jak se obsah hlavního vstupu začal zvětšovat a komplikovat, jsem dospěl k tomu, že bude jednodušší použít standardizovaný formát a vstup zpracovat s využitím knihovního syntaktického analyzátoru. Zvažoval jsem použití formátu YAML[3], ale nakonec jsem i díky doporučení konzultujícího zvolil formát JSON. Výhodou tohoto formátu je jeho jednoduchá a intuitivní podoba, díky čemuž je hojně rozšířený, používaný a známý. Tento formát se také až na drobnosti hodně podobal mému prvotnímu návrhu.

Povinnými částmi vstupu jsou položka `Nodes`, která obsahuje seznam identifikátorů všech uzlů, a položka `Edges` obsahující seznam všech hran ve formátu `Počáteční uzel -> Koncový uzel`. Volitelně je možné uvést položky `Left, Right, Top a/nebo Bottom` reprezentované navzájem disjunktními množinami uzlů, tudíž není možné mít jeden uzel jak v množině `Left`, tak v množině `Bottom`. Tyto položky slouží ke specifikování vyžádaného umístění uzlů u vybraného okraje grafu, tedy všechny uzly v seznamu položky `Left` se budou ve výsledném rozmístění nacházet na levém okraji grafu. Všechny uzly vystupující v seznamech `Left, Right, Top a Bottom` a v reprezentaci hran musí být uvedeny v seznamu `Nodes`. Žádné jiné položky nejsou ve hlavním vstupu povoleny.

Seznam `Nodes` pak dále může obsahovat i definice podgrafů. Daný podgraf poté povinně obsahuje identifikátor a seznam uzlů v tomto podgrafu, který má stejné vlastnosti jako seznam `Nodes`. Z toho plyne, že aplikace by měla podporovat neomezené zanoření podgrafů.

Identifikátory uzlů musí být unikátní v rámci podgrafu, ale ne v rámci celého grafu. To znamená, že na hlavní úrovni můžeme mít uzel `A` a podgraf `SubGraph`. Tento podgraf poté může obsahovat jiný uzel `A`. Tuto situaci musíme zohlednit při definování hran v seznamu `Edges`. Pokud je identifikátor uzlu unikátní v rámci celého grafu, tak stačí použít tento identifikátor. Pokud ale máme výše popsanou situaci, tak je nutné vnořený uzel identifikovat pomocí tečkové notace `Podgraf.Uzel`. Pokud bychom chtěli modelovat hranu z uzlu `A` na

[1] TinyCAD je program na zobrazování a tvorbu různých diagramů a obvodů. Volně dostupný z www.tinycad.net.

[2] JSON = JavaScript Object Notation (JavaScriptový objektový zápis)

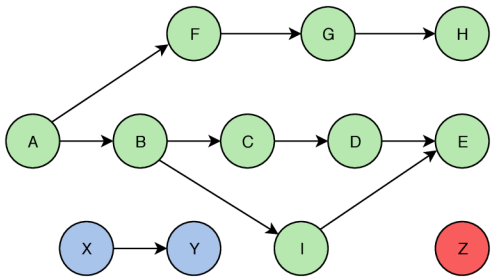
[3] YAML = YAML Ain't markup language (značkovací jazyk YAML)

hlavní úrovni do uzlu `A` v podgrafu `SubGraph`, tak by výsledný identifikátor této hrany byl: `A -> SubGraph.A`.

Příklad souboru se vstupem jednoduchého grafu by mohl vypadat například takto:

```
{
    Nodes: [A, B, C, D, E, F, G, H, I, X, Y, Z],
    Edges: [A->B, B->C, C->D, D->E, F->G, A->F, G->H, B->I, I->E, X->Y],
    Left: [A],
    Right: [E, H, Z],
    Top: [F]
}
```

A jemu odpovídající graf by mohl být rozmístěn například tak, jak je možné vidět na obrázku 3.2.



Obrázek 3.2: **Příklad jednoduchého grafu.** Jednotlivé nezávislé části grafu jsou odděleny barevně. Kresleno ručně.

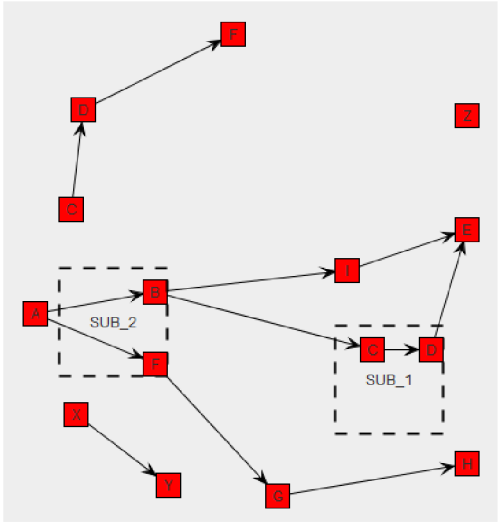
Pro úplnost je uveden i příklad vstupu s dvěma podgrafy. Uzly `C`, `D` a `B`, `F` byly vloženy po dvojicích do těchto nových podgrafů. Poté byly přidány nové uzly `C`, `D` a `F` na hlavní úrovni pro ukázku správného pojmenovávání hran. Upravený vstup s podgrafy:

```
{
    Nodes: [A, {SUB_2: [B, F]}, {SUB_1: [C, D]}, E, G, H, I, X, Y, Z, C, D,
            F],
    Edges: [A->B, SUB_2.B->SUB_1.C, SUB_1.C->SUB_1.D, SUB_1.D->E,
            SUB_2.F->G, A->SUB_2.F, G->H, B->I, I->E, X->Y, C->D, D->F],
    Left: [A],
    Right: [E, H, Z],
    Top: [F]
}
```

Jak můžeme vidět, tak uzel `B` lze adresovat jak pouze identifikátorem (`B` v hraně `A->B`), tak celou cestou i s identifikátorem podgrafu (`SUB_2.B` v hraně `SUB_2.B->SUB_1.C`). Tomuto vstupu odpovídá graf 3.3.

Volitelný vstup (2) ve formátu TinyCAD poskytuje možnost definovat polygonální vzhled uzlů (včetně pozice vstupních a výstupních pinů).

V rámci uživatelských voleb (3) se specifikuje soubor obsahující hlavní vstup a může být specifikován i soubor s volitelným vstupem ve formátu TinyCAD. Dále uživatel musí vybrat, který algoritmus si bude přát použít pro rozmístění uzlů grafu. Mezi další možnosti patří



Obrázek 3.3: **Příklad jednoduchého grafu s podgrafy.** Vychází z grafu 3.2. Uzly `C` a `D` byly spojeny do podgrafu `SUB_1`, uzly `B` a `F` byly spojeny do podgrafu `SUB_2`. Dále byly přidány nové po řadě propojené uzly `C`, `D` a `F`. Důležité je i zmínit to, že přestože se obsah seznamu `Top` nezměnil, tak uzel `F`, který je uveden v tomto seznamu, označuje nově přidaný uzel `F` na hlavní úrovni. Pokud bychom chtěli k hornímu okraji umístit uzel `F` v podgrafu `SUB_2` (čímž bychom k hornímu okraji umístili i celý podgraf `SUB_2`), tak bychom museli do seznamu `Top` uvést `SUB_2.F` — podobně jako při definování hran. Graf byl vykreslen výslednou aplikací.

volba, zda chce uživatel načíst informace o polygonální podobě uzlů ze souboru s volitelným vstupem, a nastavení ovládající předzpracování a závěrečné zpracování. Volby je možné specifikovat pomocí grafického rozhraní nebo pomocí parametrů příkazové řádky. Detailní popis argumentů aplikace a jejich použití lze nalézt v kapitole 4.

Zpracování vstupu

Parser (neboli syntaktický analyzátor) hlavního vstupu (v diagramu struktury 3.1 označen číslem 5) má za úkol zpracovat obsah vstupního souboru a naplnit jím interní reprezentaci. Jelikož formát hlavního vstupu je JSON, tak je zde vhodné začlenit knihovní JSON syntaktický analyzátor.

Modul preference (4) slouží k interpretování argumentů příkazové řádky, uložení uživatelských voleb a zpřístupnění těchto voleb v rámci celého programu tam, kde jsou potřeba, a to zejména při předzpracování a při závěrečném zpracování. Tento modul uchovává i uživatelské volby specifikované v grafickém uživatelském rozhraní.

Pro volitelný vstup bylo potřeba vytvořit samostatný parser formátu TinyCAD (6), a protože tento formát se svojí strukturou opírá o formát XML[4], tak byl použit knihovní XML parser. Tento zpracovaný vstup dále slouží pro určení velikosti jednotlivých uzlů a při generování výstupu jsou z něj převzaty polygonální tvary uzlů.

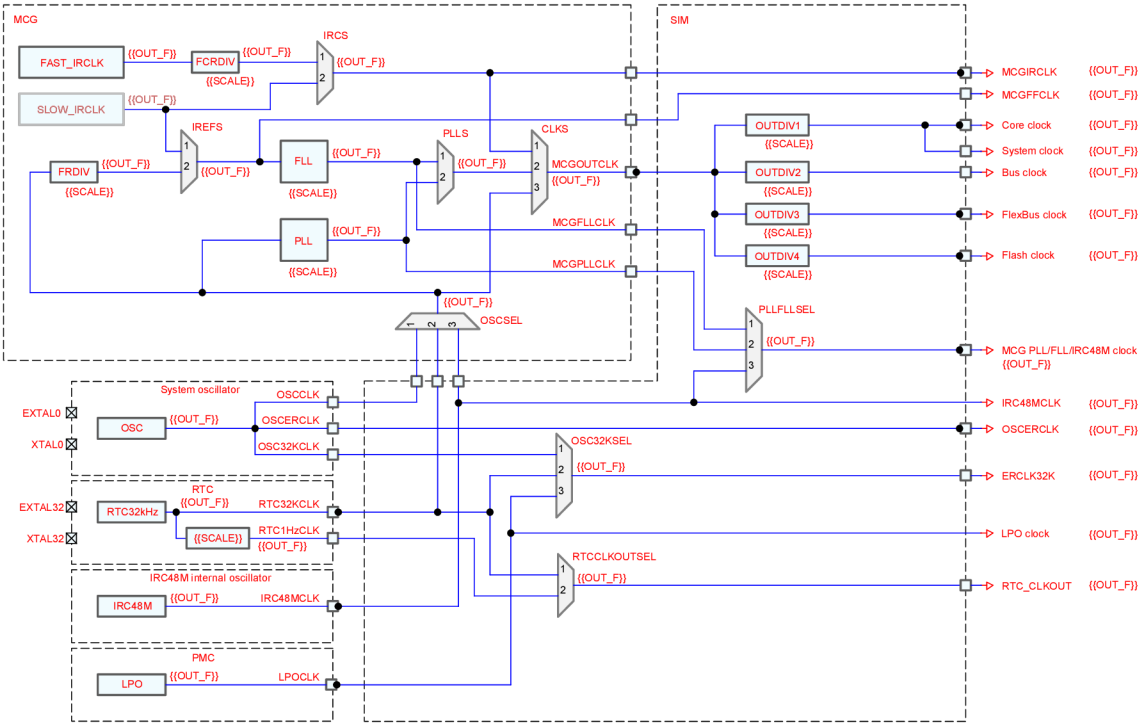
[4]XML = Extensible Markup Language (Rozšiřitelný značkovací jazyk)

3.3 Výstup programu

Než přejdu k popisu samotných částí aplikace, tak mi přijde důležité popsat její výstupy. Aplikace je schopná generovat dva výstupy, a to grafický výstup a výstup do souboru.

Jako výstupní formát grafu pro výstup do souboru jsem zvolit formát, který používá program TinyCAD. Tento formát jsem zvolil především z toho důvodu, že program TinyCAD je často využíván ve firemním prostředí, zejména v NXP, kde se používá přímo ve firmou vyvíjených konfiguračních nástrojích. Obrázek 3.4 je příkladem diagramu zobrazeného v programu TinyCAD z firemního nástroje „Clocks tool“. Další výhodou programu TinyCAD je jeho volná permisivní licence LGPL.

Grafický výstup by měl po dokončení činnosti programu zobrazit výsledek v grafickém okně.



Obrázek 3.4: **Diagram zdrojů hodinového signálu.** Poskytnuto firmou NXP a následně upraveno. Konkrétně se jedná o digram zdrojů hodinového signálu mikrokontroléru Kinetis K64F MK64FN1M0. Na diagramu můžeme vidět uzly různých polygonálních tvarů (obdélníky, lichoběžníky), podgrafy vyznačené čárkovanou čarou (například MCG nebo SIM), vlevo zdroje hodinového signálu (FAST_IRCLK, OSC) a vpravo výstupy hodin (Core clock, System clock). Konkrétní názvy a zkratky prvků uvedené v diagramu nejsou obsahově relevantní a proto nejsou vysvětleny.

Generátor TinyCAD výstupu

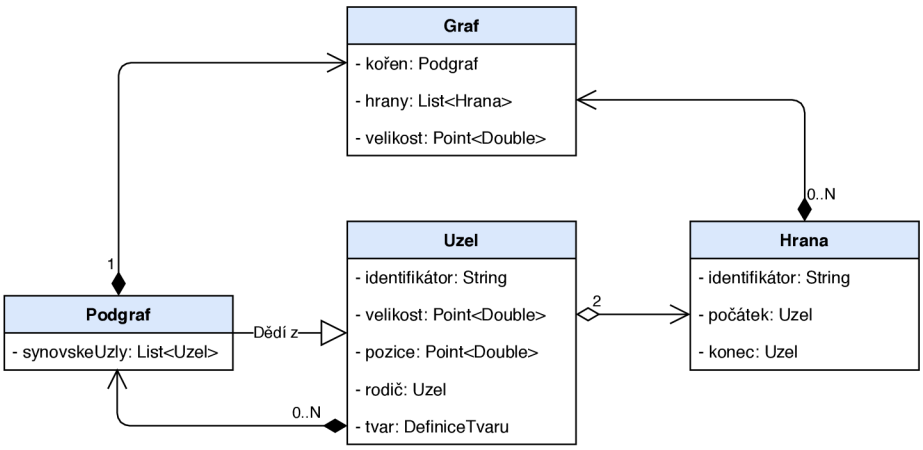
Jakmile se dokončí běh všech iterací, tak se výsledek předá generátoru TinyCAD výstupu (13), který za pomoci obsahu volitelného vstupu a uživatelských požadavků vygeneruje naprosto finální výstup. Uživatel může specifikovat, že si přeje uložit určitý počet nejlepších výsledků. V modulu preference lze potom nalézt cestu ke složce, do které budou výstupní

soubory uloženy. Dále je možné specifikovat například prefix jména souboru, či zda chce uživatel do jména souboru zahrnout i aktuální datum a čas.

Podoba uzlů, které nemají uživatelem definovaný tvar, budiž obdélník s identifikátorem uzlu vepsaným uprostřed. V tomto případě bude nutné ošetřit délku identifikátoru, aby se v důsledku příliš dlouhého názvu uzel zbytečně nerozšířil, či aby text nepřesáhnul okraj tohoto uzlu. Díky podpoře standardizovaného formátu programu TinyCAD je možné si výsledek aplikace snadno prohlédnout jednoduše tak, že výstupní soubor otevřeme v programu TinyCAD.

3.4 Interní reprezentace

Struktura interní reprezentace (7) je jedním z nejdůležitějších částí programu, protože s ní přímo pracují algoritmy použité v předběžném a závěrečném zpracování a adaptéry rozmísťovacích algoritmů z ní generují reprezentaci pro jednotlivé algoritmy. Tím pádem je stěžejní, aby interní reprezentace obsahovala všechny potřebné informace a aby se s ní dalo jednoduše manipulovat.



Obrázek 3.5: **UML diagram tříd interní reprezentace.** Každá třída je v diagramu reprezentována názvem a souborem položek. Šipky označují vztahy mezi třídami. Jasně můžeme vidět dědičnost mezi `Uzlem` a `Podgrafem`. Šipky začínající černým kosočtvercem označují kompozici (příklad: Graf se skládá z hran, a když graf zanikne, tak zaniknou všechny hrany). Šipka s bílým kosočtvercem mezi `Uzlem` a `Hranou` označuje agregaci (Hrana se skládá ze dvou uzlů, ale když hrana zanikne, tak uzly zůstanou existovat — můžou být použity i jinou hranou).

Základem této reprezentace jsou dvě třídy `Uzel` a `Hrana`, které přímo reprezentují uzly a hrany grafu. `Uzel` drží informace o svém identifikátoru, velikosti, pozici, tvaru a rodiči (rodičovském podgrafu). `Hrana` je definována identifikátorem a počátečním a koncovým uzlem. Třída `Podgraf` rozšiřuje třídu `Uzel` o seznam synovských uzlů. Jelikož `Podgraf` je tedy vlastně `Uzel`, tak je možné vytvořit stromovou strukturu uzlů a podgrafů s jedním kořenovým podgrafem. Celý graf je zastřešován třídou `Graf`, která obsahuje kořenový podgraf, seznam všech hran a celkovou velikost grafu. Toto je návrh minimální interní reprezentace. V rámci implementace jsou tyto třídy rozšířeny o další potřebné informace a metody. Díky tomu, že modelujeme podgraf jako uzel, je navíc možné přímo rozmísťovat podgrafy

v prostoru vůči sobě či vůči okolním uzlům. UML[5] diagram interní reprezentace najdete na obrázku 3.5.

3.5 Předzpracování

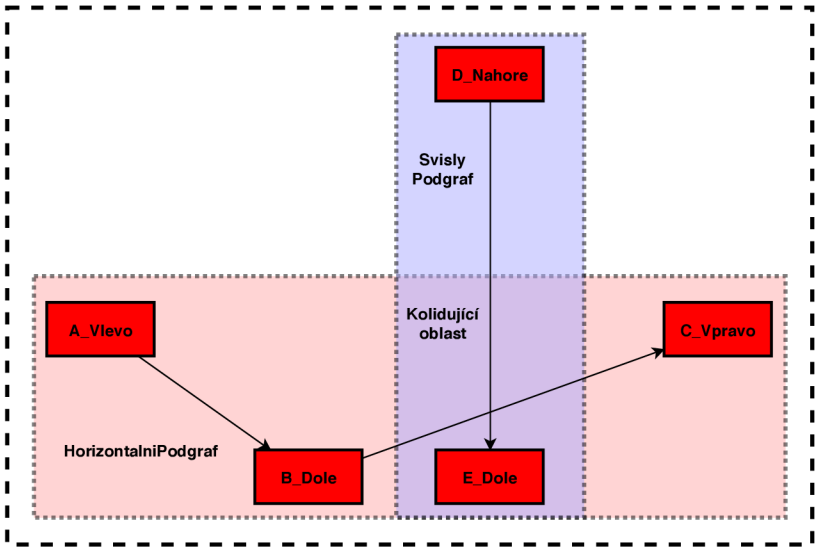
V rámci modulu předzpracování (8) (či také předběžného zpracování) se nad grafem provádí operace předtím, než se dá graf ke zpracování rozmísťovacímu algoritmu. Mezi tyto operace patři především počáteční umístění uzlů do prostoru, ze kterého vychází rozmísťovací algoritmy. Pro vytvoření počátečního rozmístění je nutné vzít v úvahu vyžádanou pozici uzlů a též existenci podgrafů. U podgrafů musíme zajistit to, že všechny uzly patřící do podgrafu se v něm budou nacházet a žádný uzel nepatřící do podgrafu se v něm vyskytovat nebude. Nejprve je potřeba určitým způsobem rozdělit prostor grafu a definovat počáteční velikost podgrafů. Teprve pak je možné postupovat rekurzivně od uzlů a podgrafů na hlavní úrovni a určovat jim jejich počáteční pozici.

Jedním z největších problémů, se kterými se bylo nutné vypořádat, je právě jakým způsobem pracovat s podgrafy, jak definovat jejich velikost, a jak pro ně rozdělit dostupný prostor, aby byly zahrnuty všechny možné varianty, které mohou nastat. Vezměme v potaz například situaci, že máme graf s šesti podgrafy a stem uzlů. Všechny podgrafy jsou podgrafy na hlavní úrovni (jejich rodičovský podgraf je kořen grafu), všechny uzly náleží do jednoho z těchto šesti podgrafů a 30 uzlů má definovanou vyžádanou pozici. V takovémto případě nejspíše budeme chtít, aby podgrafy zabíraly celý prostor grafu (aby bylo co nejméně nevyužitého místa). Dále musíme zvážit to, že v různých podgrafech mohou být uzly s různou vyžádanou pozicí. Jelikož chceme určovat počáteční pozici podgrafu na základě vyžádaných pozic jeho uzlů, tak pokud bude mít podgraf například uzly, které chtějí být vpravo a uzly, které chtějí být dole, tak tento podgraf bude zaujímat pravý dolní roh grafu a žádný jiný podgraf už tuto pozici zaujímat nemůže, protože by nebyly splněny požadavky na vyžádanou pozici uzlů jednoho z těchto podgrafů. Z toho příkladu můžeme jasně vidět, že jednak počet možných kombinací je velmi vysoký, a jednak že ne všechny kombinace uzlů s vyžádanou pozicí patřících do podgrafů je možné korektně vynést do roviny. Pro upřesnění předpokládáme, že tvar podgrafu musí být souvislý (není možné rozdělit podgraf na dvě části), podgrafy se nesmí překrývat a že podporujeme pouze obdélníkový tvar podgrafů. Pokud bychom podporovali i podgrafy polygonálních tvarů, tak bychom rozšířili množinu možných kombinací, ovšem stále bychom našli nevalidní kombinace obsahující konflikty.

Příkladem takové absolutně nevalidní kombinace by mohly být dva podgrafy, přičemž jeden z nich by měl uzly s vyžádanými pozicemi vlevo, vpravo a dole, a druhý z nich by měl uzly, které vyžadují pozici nahoře a dole. Takovéto podgrafy by poté nebylo možné vykreslit bez toho, aby se navzájem nepřekrývaly. V případě, že program detekuje tuto chybu, tak by měl ukončit umísťování uzlů a upozornit uživatele. Zobrazení tohoto nevalidního vstupu najdete na obrázku 3.6. Zde je uvedena jeho textová podoba — příklad chybného vstupu:

[5] UML = Unified Modeling Language (česky Sjednocený modelovací jazyk)

```
{
    Nodes: [{HorizontalniPodgraf: [A_Vlevo, B_Dole, C_Vpravo]},
            {SvislyPodgraf: [D_Nahore, E_Dole]}],
    Edges: [A_Vlevo->B_Dole, B_Dole->C_Vpravo, D_Nahore->E_Dole],
    Left: [A_Vlevo], Right: [C_Vpravo],
    Top: [D_Nahore], Bottom: [B_Dole, E_Dole]
}
```



Obrázek 3.6: **Příklad konfliktu podgrafů s uzly s vyžádanou pozicí.** Na obrázku můžeme vidět svislý podgraf zobrazen světle modře, horizontální podgraf světle červeně a jejich protínající se kolidující oblast světle fialově. Můžeme tak jasně vidět, že daný graf nelze vykreslit bez překrývajících se podgrafů. Kresleno ručně.

Během předzpracování je také možné zkontrolovat a popřípadě opravit uživatelem zadané volby. Poslední z činností modulu předzpracování je spuštění iterační smyčky.

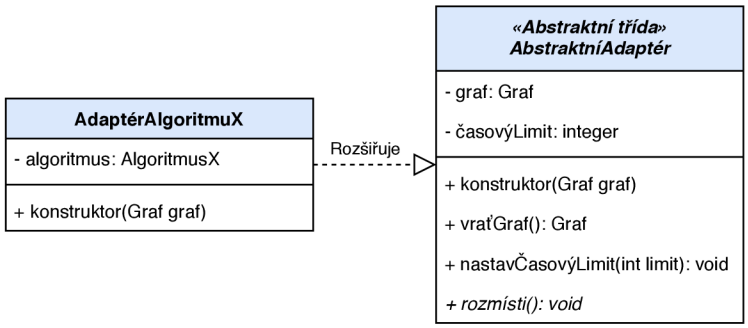
3.6 Iterační smyčka

Iterační smyčka (9) je nejvytíženější část programu, jelikož odvádí nejvíce práce, a dá se tedy říci, že program jejím vykonáváním stráví nejvíce času. Graf se zde opakovaně předává adaptéru algoritmů, aby bylo provedeno rozmístění uzlů grafu, přičemž v rámci jedné iterace může být adaptér zavolán i vícekrát, jelikož je možné (a kvůli podpoře podgrafů i nutné) rozmísťovat uzly grafu po částech. Po každém zavolání adaptéru je poté nutné spustit následné zpracování (11), které zajistí, že se uzly nebudou překrývat a že se budou zcela nacházet v podgrafech, do kterých patří.

Po dokončení každé kompletní iterace je graf ohodnocen a vzhledem k tomuto ohodnocení se ukládá uživatelem zvolený počet nejlepších výsledků, takže na konci všech iterací máme soubor nejlepších získaných výsledků s jejich ohodnoceními. Z tohoto souboru výsledků je pak možné přímo zobrazit grafický výstup (14) či pomocí generátoru TinyCAD výstupu (13) vytvořit výstupní soubor.

3.7 Adaptér algoritmů

Integrování jednotlivých rozmísťovacích algoritmů do aplikace je řešeno pomocí tzv. adaptérů (10). Adaptér je třída, která do sebe zabalí algoritmus a zbytku aplikace k němu zpřístupní jednotné rozhraní. Více se o návrhovém vzoru Adaptér můžete dočíst v knize [6]. Ve výsledku tedy má každý použitý algoritmus svůj vlastní adaptér, v rámci kterého se provádí transformace interní reprezentace grafu do jiné reprezentace, které rozumí daný algoritmus a dokáže s ní pracovat. Poté, co algoritmus zpracuje graf, se v adaptéru také převede výstup algoritmu zpět do interní reprezentace. Velkou výhodou tohoto řešení je to, že pro přidání nového algoritmu do aplikace bude v podstatě stačit pouze naimplementovat adaptér pro daný algoritmus a přidat tento algoritmus mezi uživatelské možnosti. Návrh rozhraní takovéhoto adaptéru najdete na obrázku 3.7.



Obrázek 3.7: **UML diagram adaptéru algoritmu.** Na obrázku můžete vidět 2 třídy. Každá třída je rozdělena na 3 bloky. V prvním bloku je uvedeno jméno třídy, v druhém třídní položky a ve třetím bloku metody třídy. Metody a třídní položky mají následující formát: První je úroveň zapouzdření (symbol „+“ znamená public neboli veřejné, symbol „-“ znamená private neboli soukromé), dále je uveden název a za dvojtečkou je datový typ. Položka `časovýLimit` třídy `AbstraktníAdaptér` je tedy soukromá a její typ je celé číslo (integer). Metody uvedené kurzívou jsou abstraktní. Diagram znázorňuje abstraktní třídu `AbstraktníAdaptér` a třídu `AdaptérAlgoritmuX`, která implementuje abstraktní metody a rozšiřuje třídu `AbstraktníAdaptér`.

Abstraktní třída `AbstraktníAdaptér` má položku `graf`, do které bude konstruktorem uložena reprezentace grafu. Dále tato třída obsahuje položku `časovýLimit` a metodu `nastavČasovýLimit` pro nastavení tohoto limitu. Časový limit by měl být použit v implementaci adaptéru konkrétního algoritmu pro omezení maximální doby běhu rozmísťovacího algoritmu. Třída `AdaptérAlgoritmuX` zde má znázorňovat právě tuto implementaci adaptéru konkrétního algoritmu. Celková myšlenka použití adaptéru algoritmu je taková, že při konstrukci adaptéru si adaptér uloží reprezentaci grafu, vytvoří instanci rozmísťovacího algoritmu a tu si uloží do položky `algoritmus`. Dále při zavolání metody `rozmísti` se převede interní reprezentace grafu do formy, se kterou dokáže daný algoritmus pracovat, rozmísťovací algoritmus se spustí a následně se aktualizuje interní reprezentace v položce `graf` změnami provedenými rozmísťováním. Rozmístěný graf v interní reprezentaci je možné získat metodou `vraťGraf`.

3.8 Následné zpracování

Výstupem z adaptéru algoritmů se ani zdaleka ještě nekončí. Přestože jsou uzly grafu rozmístěny do prostoru, tak je potřeba provést ještě mnoho dokončujících úprav, a to tady v modulu následného zpracování (11). Asi nejdůležitější úprava je korekce rozestupů mezi uzly a celková modifikace pozic uzlů. Tento krok je nezbytný provést z toho důvodu, že rozmísťovací algoritmy typicky neberou v potaz velikost uzlů, a jelikož každý uzel může mít jinou velikost a tvar (v případě, že uživatel specifikuje polygonální podobu uzlů), tak bychom mohli dojít k problému překrývání uzlů či přetékání uzlů z prostoru svých podgrafů. Při řešení těchto problémů však musíme dbát na to, aby naše změny nebyly zbytečně velké, protože se nechceme příliš odchýlit od výsledku rozmísťovacího algoritmu.

Jelikož i podgrafy je možné v rámci rozmísťování přesouvat, tak předtím, než budeme řešit problémy samostatných uzlů, musíme vyřešit pozice podgrafů. V rámci předzpracování (podkapitola 3.5) se rekurzivně rozděluje prostor každého podgrafu všem jeho synovským podgrafům a často se podgrafy roztahují do celé jim dostupné šířky či výšky. Takovéto roztažené podgrafy je poté nutné seřadit a zarovnat do prostoru.

Přesouvání přetékajících uzlů je v podstatě jednoduché. Uzel může logicky přetékat maximálně na dvou stranách (v této části již musí být zajištěno, že žádný uzel není širší nebo vyšší než jemu přiřazený podgraf), a tudíž by k vyřešení problému mělo stačit uzel vždy posunout o minimální nutnou vzdálenost, aby se celý nacházel uvnitř podgrafu. Tímto posunutím ovšem můžeme způsobit částečné překrytí s jiným uzlem, proto je doporučené prvně řešit přetékání a až poté překrývání uzlů.

Řešení problému překrývajících se uzlů je lehce komplikovanější. Ideálně chceme pohnout pouze jedním ze dvou překrývajících se uzlů (uzlů se může překrývat i více naráz, ale vždy testujeme a řešíme právě jen překrývající se dvojice) a to tak, aby tímto přesunem bylo způsobeno co nejméně problémů. V žádném případě není přípustné uzel přesunout tak, že by přetékal nebo se zcela nacházel mimo jeho rodičovský podgraf. Typicky je doporučeno přesouvat menší z uzlů, jelikož přesunutím menšího uzlu spíš způsobíme méně konfliktů než přesunutím většího. Každopádně je vhodné otestovat, zda můžeme jeden z uzlů posunout nějakým vhodným směrem tak, aby nedošlo ke konfliktu, a pokud ano, tak tento přesun provést. V případě, že není možné provést přesun bez následného konfliktu, tak bychom stejně nějaký (například nejmenší) přesun měli provést, a to proto, že operace následného zpracování je možné spouštět iterativně a pokud vyřešíme původní překrytí a způsobíme jinde další, tak je možné že v dalším kroku iterace vyřešíme úspěšně toto nové překrytí bez způsobení dalšího konfliktu.

3.9 Hodnocení

Vyhodnocovací modul (12) vypočítává ohodnocení (skóre) jednotlivých výsledků průběhu iterace. Ohodnocení je definováno tak, že se počítají chyby v rozmístění grafu, tudíž čím nižší ohodnocení, tím lepší máme výsledek. Ohodnocení se počítá jakou součet všech uživatelem vybraných vyhodnocovacích pravidel (score rules) násobených uživatelem definovanou váhou daného pravidla. Vyhodnocovací pravidlo může například počítat počet křížení dvou hran či počet hran probíhajících skrz uzly nepatřící této hraně.

Skóre se s rostoucím počtem iterací zlepšuje, nejde ale o čistě klesající sestup. Z toho důvodu se výsledky iterací sbírají a ukládají v průběhu iterování. Jinak řečeno, v průběhu iterací skóre klesá, ovšem průběžně může lehce vzrůst a poté zase pokračovat v klesání. Výsledek na konci iterací tedy může být horší, než výsledek získaný v průběhu iterování.

Kolísání skóre je dáno hledáním určitého kompromisu mezi ideálním stavem rozmístění dle rozmísťovacího algoritmu a stavem, který je ideální vzhledem k požadavkům uživatele (podgrafy, vyžádané pozice) vynucovaných následným zpracováním. V rámci iterování je tedy hledán jeden z mnoha ustálených stavů rozmísťovacího algoritmu, který zároveň vyhovuje výše zmíněným požadavkům. K nalezení tohoto stavu je nutné přejít z jednoho ustáleného stavu do jiného, a přesně tento přechod způsobí zakolísání skóre.

Kapitola 4

Popis řešení a implementace

Zde je vysvětleno řešení a detailně popsána implementace aplikace. Jako první zde rozebereme uživatelské rozhraní. Dále se v této kapitole setkáme s popisem celé aplikace a jejích modulů v tom pořadí, v jakém se vykonávají při spuštění. Pokud je v rámci vysvětlování implementace zmíněno jméno knihovny či programu, tak je u něj vždy uveden odkaz na přílohu B, kde se o daném softwaru lze dočíst více informací. Aplikace byla implementována v jazyku Java. Konkrétně byla použita Oracle Java verze 8.

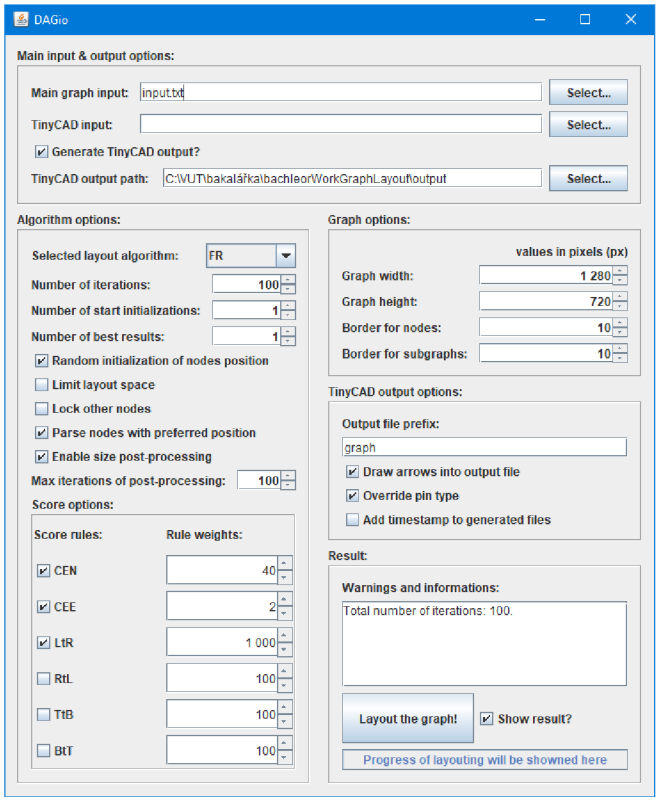
Vytvořená aplikace nese jméno **DAGio**. Tento kreativní název autor vytvořil spojením zkratky DAG (directed acyclic graph) s koncovkou „io“, která má symbolizovat spojení slov Input-Output (vstup-výstup). Celý název se tedy odkazuje na skutečnost, že do aplikace vstupují a vystupují z ní orientované acyklické grafy.

4.1 Uživatelské rozhraní

Aplikace podporuje dva typy uživatelského rozhraní, a to příkazovou řádku a GUI[1]. Obě rozhraní podporují plně všechny dostupné uživatelské volby. Aplikace je navržena tak, že primárním rozhraním je příkazová řádka. Při jejím použití uživatel specifikuje svoje preference zadáním argumentů programu, přičemž argumenty mohou být v libovolném pořadí. GUI lze spustit zadáním argumentu `--gui`. Pro zpracování argumentů příkazové řádky a generování nápovědy byla použita knihovna Picocli (viz podkapitola B). Uživatelské volby jsou poté uloženy do třídy `Preferences`.

Grafické uživatelské rozhraní má dvě části — hlavní okno aplikace a grafické zobrazení výsledků. Hlavní okno GUI funguje tak, že po spuštění načte hodnoty uložené v třídě `Preferences` a použije je jako svoje výchozí hodnoty, které uživatel uvidí. Tím je docíleno toho, že pokud uživatel do příkazové řádky přidá k argumentu `--gui` další argumenty, tak se tyto argumenty zpracují a jimi specifikované hodnoty budou zobrazeny v GUI. Uživatel dále může v tomto okně měnit všechny dostupné volby, spustit rozmísťování uzlů a sledovat průběh rozmísťování (zobrazí se modální dialog s ukazatelem postupu). Po dokončení rozmísťování může uživatel znovu měnit možnosti a znovu spustit rozmísťování uzlů. Aplikace nedovoluje měnit uživatelské volby v průběhu rozmísťování uzlů, ani spouštět více vláken rozmísťování současně. Pokud by uživatel potřeboval rozmísťovat více různých grafů současně (nebo stejný graf s jinými volbami), tak si může spustit další instanci programu nebo použít skript například k opakovanému spuštění aplikace z příkazové řádky.

[1] GUI = Grafické uživatelské rozhraní, anglicky Graphical User Interface

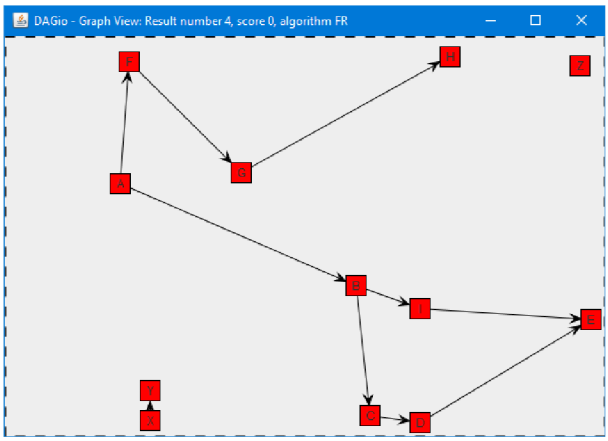


Obrázek 4.1: **Hlavní okno grafického uživatelského rozhraní.** Na obrázku vidíme hlavní okno aplikace se všemi volbami ve výchozím stavu a s vyplněným vstupním souborem a výstupní cestou.

Jak můžeme vidět na obrázku 4.1, tak jednotlivé možnosti jsou graficky seskupeny do skupin týkajících se stejné části aplikace. Nahoře vidíme skupinu volby vstupních a výstupních souborů. Vlevo se nachází skupina možností vztahující se k samotným algoritmům a průběhu rozmísťování. Tato skupina v sobě obsahuje vnořenou skupinu pro volbu hodnoticích pravidel. Po pravé straně jsou postupně seskupeny vlastnosti grafu, a také další možnosti tvorby výstupních souborů. Panel vpravo dole obsahuje informační oblast, ve které se zobrazují varování a chyby v konfiguraci, tlačítko „Layout the graph!“ na spuštění rozmísťování a ukazatel postupu.

Pravý dolní panel má vedle rozmísťovacího tlačítka zaškrtávací políčko „Show result?“. Pokud je toto políčko zaškrtnuté, tak se po dokončení rozmísťování zobrazí samostatné okno s grafickým zobrazením výsledku, viz obrázek 4.2. Grafické zobrazení grafu se tvoří přímo z interní reprezentace (třída `GraphRepresentation`) a dokáže vykreslit celou interní reprezentaci v jakékoliv fázi rozmísťování. To se může hodit například při hledání chyb nebo při dalším rozšiřování programu. Implementace zobrazení grafu používá knihovnu JUNG (viz podkapitola B). Graf se v tomto okně dá posouvat přetažením myši se stisknutým levým tlačítkem. Také je možné ho přibližovat či oddalovat použitím kolečka myši.

Hlavní okno GUI bylo navrženo v grafickém editoru programu Netbeans (viz podkapitola B). Celé GUI (třída `GUI`) je řízeno ovladačem GUI implementovaným ve třídě `GuiController`, který propojuje grafické rozhraní se zbytkem aplikace. Toto propojení je realizováno tvorbou posluchačů (anglicky *listener*). Ti čekají na příchod zpráv o událostech



Obrázek 4.2: **Okno s grafickým zobrazením výsledku rozmísťování.** V hlavičce okna můžeme vidět název aplikace, informaci že se jedná o zobrazení grafu, číslo výsledku (zde máme čtvrtý nejlepší výsledek), hodnocení grafu (zde vidíme skóre 0, tudíž nejlepší možný výsledek se zvolenými hodnoticími pravidly) a algoritmus, který byl pro získání tohoto výsledku použit (zde Fruchterman-Reingoldův algoritmus).

(anglicky *event*) z GUI a propagují příslušné akce do zbytku aplikace. Nejčastěji se jedná o uložení nové uživatelem zadané hodnoty zpět do modulu preferencí a provedení tomu přidružených akcí, jako například zkontrolování správnosti konfigurace a zobrazení chyby.

Všechny možnosti programu s detailním vysvětlením naleznete v příloze Preference — možnosti a argumenty programu.

4.2 Implementace a průchod aplikací

Tato podkapitola popisuje implementaci navrhnuté aplikace. Najdete zde hodně odkazů vztahujících se přímo ke kódu. Budou zde zmiňovány názvy tříd a funkcí. Po přečtení této kapitoly by měl čtenář získat dobrou představu o tom, jak jsou jednotlivé moduly propojeny, jak a v jaké podobě se uchovávají data a hlavně jak vypadá celkový průchod aplikací od zapnutí přes rozmísťování uzlů a generování výstupů až do konce. Nyní následuje popis průchodu aplikací, ze které jsou vedeny odkazy na podkapitoly obsahující detailní popis implementace vybraných modulů.

Po spuštění se program nachází v metodě `main` hlavní třídy `Application` (viz obrázek 4.3), která zaštiťuje a reprezentuje celou aplikaci. Zde se použije modul Preference (třída `Preferences`) ke zpracování argumentů příkazové řádky. Tento modul je navržen jako jedináček[2], což je vhodné z toho důvodu, že potřebujeme uchovávat uživatelské volby a zpřístupnit je v rámci celé aplikace. Dále rozhoduje, zda uživatel požaduje zobrazení grafického uživatelského rozhraní. Pokud ano, tak se vytvoří instance třídy `GuiController` a tato instance otevře hlavní okno GUI. Jakmile uživatel v GUI klikne na tlačítko pro zahájení rozmísťování, tak se v novém vlákně zavolá metoda `applicationRun` třídy `Application`. Tato metoda provádí celý jeden běh rozmísťování od načtení vstupů po generování výsledků. V opačném případě (GUI není vyžadováno) se rovnou spustí běh rozmísťování.

[2]jedináček = návrhový vzor jedináček (anglicky *singleton*) zajišťuje, že bude existovat pouze jedna instance dané třídy (viz [6])

```java
public static void main(String[] args) {
    // Parse preferences
    Preferences preferences = Preferences.getInstance();
    preferences.parseArgs(args);
    // Start GUI if desired
    if (preferences.isGuiRequested()) {
        guiController = new GuiController();
        guiController.openGui();
    } else {
        applicationRun();
    }
}
```

Obrázek 4.3: **Ukázka metody `main` třídy `Application`.**

Na začátku běhu rozmísťování se zkontroluje, zda uživatelem zadaná konfigurace neobsahuje fatální chyby, a začne se zpracovávat hlavní vstupní soubor modulem `InputParser` (viz podkapitola Parser hlavního vstupu). Výsledkem zpracování hlavního vstupu je graf uložený v interní reprezentaci, neboli uložený ve třídě `GraphRepresentation`. Dále, pokud je zadán, tak se zpracuje i soubor s vedlejším TinyCAD vstupem pomocí stejného modulu. Po zpracování vstupů se zjistí, zda je vynuceno úplné deterministické chování, a pokud ano, tak je se pomocí reflexe nahradí generátor náhodných čísel ve standardní matematické knihovně novým generátorem inicializovaným fixní hodnotou. Tento způsob dosažení deterministického chování algoritmů byl zvolen, protože algoritmy přímo používají funkci `Math.random()`. Funkce vynucení deterministického chování není podporována v GUI z toho důvodu, že GUI v knihovním kódu též používá funkci `Math.random()` a není tak možné zajistit kompletní determinismus a reprodukovatelnost výsledků při jeho použití. Následně modul předzpracování (třída `PreProcessing`) zkontroluje, zda je uživatelem zadaná velikost grafu dostatečná (více v oddílu Předzpracování — kontrola velikosti grafu). Tímto jsou všechny přípravné operace hotové a aplikace může přejít k jádru své činnosti zavoláním metody `layout` třídy `Application` (výstřižek kódu viditelný na obrázku 4.4). Této metodě se jako parametr předá graf v interní reprezentaci.

```java
/**
 * Performs the main part of laying out a graph
 * @param graph
 * @return best results of laying out a graph or null if severe error occurred
 */
private static List<GraphRepresentation> layout(GraphRepresentation graph) {
    if (PreProcessing.prepareSubGraphs(graph.getRootGraph())) {
        return new ArrayList<>();
    }
    List<AbstractAdapter> initialLayouts = PreProcessing.initialLayoutAndCreateAdapters(graph);
    return PreProcessing.runAlgorithmIterations(initialLayouts);
}
```

Obrázek 4.4: **Výstřižek z kódu — metoda `layout` třídy `Application`.**

Prvním krokem metody `layout` je příprava podgrafů, která se odehrává v metodě `prepareSubGraphs(SubGraph subgraph)` (`SubGraph` je třída reprezentující podgrafy) patřící modulu předzpracování. Tato příprava spočívá v tom, že rekurzivně, počínaje kořenovým podgrafem, rozdělí podgrafy a jejich prostor na části prostoru a jim náležející uzly či synovské podgrafy. Při tomto rozdělování prostoru je nutné zkontrolovat konflikty (viz

obrázek 3.6 v podkapitole Předzpracování). Na samotné rozdělení se poté používá takzvaný Dělič prostoru implementovaný ve třídě `SpaceDivider`. Druhý krok je tvorba uživatelem zadaného počtu počátečních rozmístění. Pro každé rozmístění je nutné vytvořit kopii grafu a adaptér algoritmu zvoleného algoritmu. V případě zvolení algoritmu BEST se vytvoří pro každé rozmístění další kopie a adaptér každého dostupného rozmísťovacího algoritmu. Více o integrování rozmísťovacích algoritmů do aplikace naleznete v oddílu Adaptéry algoritmů. Poslední krok spočívá ve spuštění iterační smyčky metodou `runAlgorithmIterations` třídy `PreProcessing`, která postupně zpracuje všechna počáteční rozmístění. Jelikož iterační smyčka je dosti komplikovanou částí aplikace, tak si ji a jí používané moduly popíšeme samostatně v další podkapitole 4.3. Nyní předpokládejme, že smyčka doběhla a máme seznam nejlepších rozmístění grafu.

Jakmile jsme získali seznam výsledků, tak nám stačí tyto výsledky grafiky zobrazit a vygenerovat výstupní soubory. Soubory s výslednými rozmístěními grafu generuje generátor výstupu, který se skládá ze své hlavní třídy `OutputGenerator` a z třídy `TinyCadConstants`, která obsahuje řetězce a konstanty vkládané do výstupu. Generátor výstupu postupuje tak, že prvně generuje definice symbolů podgrafů a uzlů, poté generuje symboly podgrafů a uzlů a nakonec vygeneruje hrany. Aplikace aktuálně podporuje pouze hrany v podobě úseček. Zajímavým rozšířením by mohla být podpora pravoúhlých lomených hran.

Parser hlavního vstupu

Modul Parser (neboli syntaktický analyzátor) vstupu slouží zároveň ke zpracovávání hlavního i vedlejšího vstupu. Hlavní vstup ve formátu JSON je analyzován pomocí knihovny GSON (viz B) a při jeho zpracovávání se postupně tvoří interní reprezentace. Vytváří se uzly, podgrafy, hrany, uzlům se přiřazují vyžádané pozice a nakonec se vytvoří kompletní reprezentace.

Pro zpracování vedlejšího vstupu je použit standardní knihovní analyzátor jazyka XML — `XMLEventReader`. Získání polygonálního vzhledu ze souboru ve formátu TinyCAD je lehce složitější, protože každý uzel má svůj grafický symbol a každý symbol má takzvanou definici symbolu. Definice symbolu popisuje obecně vzhled symbolu, včetně polygonálního tvaru. Více symbolů může sdílet jednu definici, přičemž až v samotném symbolu se specifikují konkrétní věci jako identifikátor, jméno, pozice a další. K získání polygonálního tvaru je tedy prvně nutné pro každý uzel najít podle shody identifikátoru odpovídající symbol. Poté je nutné projít soubor znovu a pro každý získaný symbol přečíst odpovídající odkaz na definici symbolu, z ní získat polygonální popis tvaru a ten přiřadit odpovídajícímu uzlu. Uzly (třída `Node`) si uchovávají odkaz na objekt třídy `SymbolDefinition`, ve kterém je uložen načtený vzhled.

Důležité je zmínit také to, že vytvořená aplikace nepodporuje kompletně celý formát TinyCAD souborů, ale jen určitou podmnožinu, a to takovou podmnožinu, aby použitelnost aplikace vyhovovala použití ve firmě NXP. Například není podpořen tvar symbolu typu elipsa. Nepodpoření elipsy bychom mohli odůvodnit tím, že pokud polygonem (mnohoúhelníkem) rozumíme uzavřenou konečnou část roviny ohraničenou úsečkami, tak elipsa nespadá do této definice, tudíž elipsa není polygon (a aplikace má podporovat polygonální uzly).

Předzpracování — kontrola velikosti grafu

Pro účely rozdělování prostoru grafu a možnosti dosáhnout rozumného výsledku byla definována určitá pravidla pro minimální velikost prostoru grafu. První pravidlo se vztahuje

k rozměrům (tím myslíme k výšce a k šířce) grafu a říká, že minimální šířka grafu je rovna součtu šířky (počítáno včetně velikosti uživatelem zadaného dodatečného okraje) nejširšího a nejvyššího uzlu. To platí obdobně i pro výšku. Druhé pravidlo se vztahuje k obsahu plochy grafu a říká, že minimální obsah plochy grafu je roven jeden a půl násobku součtu obsahů obdélníků ohraničujících uzly včetně dodatečného okraje.

Tato pravidla poté aplikujeme i pro vypočítání minimálních rozměrů všech podgrafů, přičemž pokud je synovským uzlem podgrafu další podgraf, tak se i jeho velikost bude vypočítávat rekurzivně. Prvním pravidlem zajistíme, že se každý jeden uzel zcela vejde do prostoru svého rodičovského podgrafu. Druhým pravidlem se snažíme zajistit dostatek prostoru pro variantu, že v grafu máme vysoký počet velmi malých uzlů. Tato dvě pravidla dohromady nejsou příliš omezující a reálně nám sice nezajistí, že se všechny uzly vejdou do svého rodičovského podgrafu bez překrývání, ale zajistí nám, že se program nebude snažit najít rozmístění grafu v příliš malém prostoru.

Pokud kontrola velikosti prostoru celého grafu zjistí, že prostor není dostatečný, tak to oznámí varováním uživateli a prostor grafu zvětší na minimální velikost. Zvětšení v důsledku prvního pravidla se provádí nahrazením velikosti jedené či obou stran grafu. Zvětšení v důsledku druhého pravidla prodlouží obě strany s ohledem na zachování jejich poměru.

Dělič prostoru

Modul Dělič prostoru (třída `SpaceDivider`) je součástí předzpracování a řeší problém určení velikosti podgrafů a jejich umístění vzhledem k uzlům s vyžadovanou pozicí. Problém je řešen sadou heuristických metod.

Uzly mohou požadovat, aby se nacházely u jednoho ze čtyř okrajů grafu (to se vždy stahuje k celému prostoru grafu, nikoliv k jednotlivým podgrafům). Podgrafy mohou obsahovat mnoho uzlů s různými vyžádanými pozicemi. Jelikož nemůžeme rozdělit podgraf na více nesouvislých částí, tak z toho vyplývá, že můžeme každému podgrafu definovat množinu stran grafu, se kterými musí sousedit, v závislosti na svých synovských uzlech (či podgrafech) s vyžádanou pozicí. Jednoduše řečeno pokud máme uzel `A` s vyžádanou pozicí vlevo a podgraf `SUB_A` jenž obsahuje uzel `A`, tak se ve výsledku podgraf `SUB_A` bude muset vyskytovat někde u levého okraje grafu. Pokud navíc budeme mít uzel `B` s vyžádanou pozicí nahoře a podgraf `SUB_B` obsahující uzel `B` a podgraf `SUB_A`, tak podgraf `SUB_B` se dozajista bude muset vyskytovat v levém horním rohu prostoru grafu. Vyžádané strany podgrafů jsou definovány jakožto množiny, protože nezáleží, kolik potomků daného podgrafu se chce u vyžádaného okraje vyskytovat. Stačí vědět, jestli alespoň jeden nebo žádný. Podgrafy tedy mohou vyžadovat sousednost s nula až čtyřmi stranami prostoru grafu. Jak jsme si již uvedli v návrhu modulu Předzpracování, tak je možné, že nastane situace, při které dojde k neřešitelnému konfliktu mezi dvěma či více podgrafy s vyžádanými sousednostmi. Pokud aplikace nalezne tento typ chyby, tak o této skutečnosti uživatele upozorní a rozmísťování ukončí.

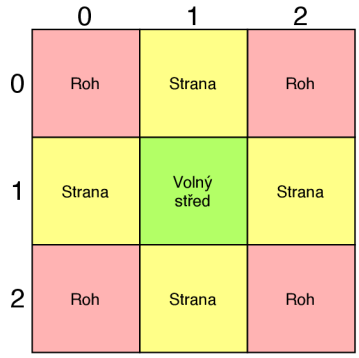
Další věcí na zvážení je to, že rozhodně nechceme v rámci předzpracování prostor rozdělit fixně a určit podgrafům neměnnou pozici. Raději bychom na jejich pozici pouze zavedli nějaké omezení, ale ponechali jim možnost se aspoň částečně v rámci rozmísťování po prostoru grafu pohybovat.

Pro popis řešení si nejprve podgrafy rozdělíme do několika typů v závislosti na jejich vyžádané sousednosti s okraji graf. Prvním typem jsou **volné podgrafy**, čímž rozumíme podgrafy nevyžadující žádnou sousednost a volné je nazýváme proto, že nejsou připoutány k žádné straně grafu. Dále máme podgrafy vyžadující sousednost s jedním okrajem, **jed-**

nostranné podgrafy, ty nejsou až tak zajímavé a můžeme se k nim chovat stejně, jako k uzlům s vyžádanou pozicí. Podgrafy vyžadující sousednost s dvěma okraji se dělí na dvě skupiny, a to na **rohové podgrafy** (vyžadují sousednost se dvěma sousedními stranami) a **řádkové/sloupcové podgrafy** (vyžadují sousednost se dvěma protějšími stranami). Poslední dvě skupiny jsou **třístranné podgrafy** vyžadující sousednosti se třemi stranami a **čtyřstranné podgrafy** vyžadující sousednosti se všemi čtyřmi stranami.

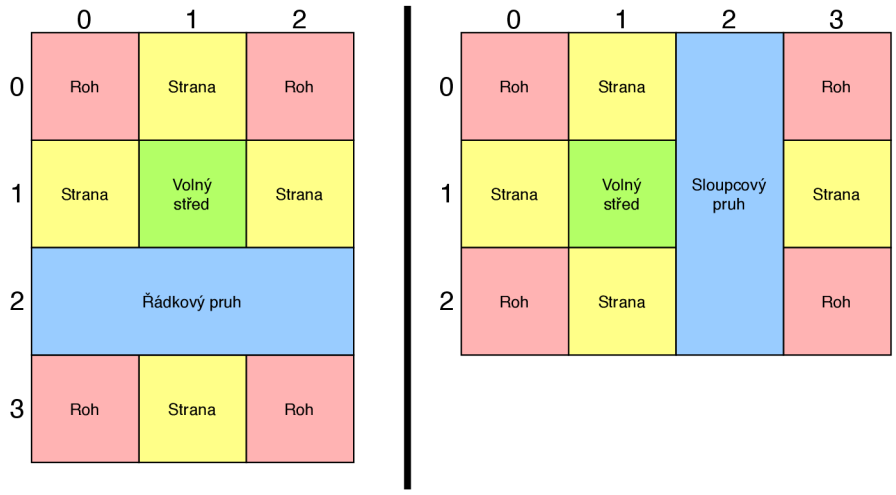
Příprava podgrafu pro rozdělování vypadá tak, že se vezmou všechny synovské uzly a podgrafy (dále jen potomci) zpracovávaného podgrafu, zjistí se jejich vyžadovaná sousednost s okraji grafu a proběhne test, zda se v daném podgrafu nenachází konflikt. Pokud se nenašel konflikt, tak je možné pokračovat. Potomci zpracovávaného podgrafu se rozdělí do skupin podle sousednosti a v této podobě se předají Děliči prostoru na zpracování. Příprava probíhá sestupně hierarchicky od kořenového podgrafu s tím, že kořenovému podgrafu nastavíme velikost rovnou velikosti prostoru grafu, tudíž zajistíme, že má dostupné všechny čtyři okraje grafu. Jelikož sestupujeme po hierarchii podgrafů a vyžádaná sousednost podgrafů je definovaná rekurzivně, tak nikdy nemůže nastat situace, že bychom měli podgraf, jehož potomek požaduje sousednost s okrajem grafu, se kterým zpracovávaný podgraf nesousedí.

Dělič prostoru z počátku vnímá prostor abstraktně (bez konkrétních rozměrů) a rozdělí ho na devět buněk. Toto rozdělení můžete vidět na obrázku 4.5. První fází je umísťování



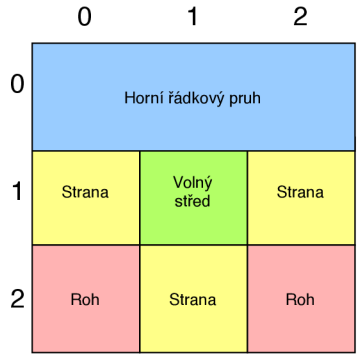
Obrázek 4.5: **Rozdělení prostoru.** Na obrázku vidíme rozdělení prostoru na devět buněk uskupených do tří řádků a tří sloupců. Řádky a sloupce jsou označeny jejich indexy. Jednotlivé buňky jsou rozděleny na tři typy, a to červené rohy, žluté strany a zelený střed.

jednotlivých potomků do buněk v závislosti na jejich vyžádaných pozicích. Tyto buňky mohou obecně obsahovat více potomků. Výjimkou jsou rohové buňky, které mohou obsahovat pouze jeden podgraf, jelikož dva podgrafy ve stejném rohu by znamenaly kolizi. Pokud seznam potomků obsahuje čtyřstranný podgraf, tak tento podgraf je jediným potomkem v tomto seznamu a automaticky musí získat celý prostor svého předka. Všichni volní potomci (volné podgrafy a uzly bez vyžádané pozice) jsou umístěni do středové buňky. Každý jednostranný potomek je vložen do buňky odpovídající jeho straně. Rohové podgrafy obsadí svoje odpovídající rohové buňky. Přítomnost řádkového či sloupcového podgrafu způsobí, že do abstrakce prostoru přidá řádek či sloupec vyplněný jednou buňkou. Upravenou abstrakci prostoru lze vidět na obrázku 4.6. V jednom podgrafu může být více řádkových nebo více sloupcových podgrafů zároveň, není ale možné mít v podgrafu zároveň řádkový a sloupcový podgraf, jelikož by se jednalo o konflikt. Třístranné podgrafy se vkládají podobně, ale místo vytvoření nového sloupce/řádku pouze sloučí existující rohové buňky a buňku strany



Obrázek 4.6: **Upravené rozdělení prostoru.** Rozdělení prostoru upravené po vložení řádkového podgrafu vpravo a po vložení sloupcového podgrafu vlevo.

do jedné v odpovídajícím sloupci či řádku. Kdybychom tedy měli třístranný podgraf vyžadující pravý, horní a levý okraj, tak se celý řádek s indexem nula sloučí do jediné buňky roztahující se před tento řádek (viz obrázek 4.7). Z toho je patrné i to, že podgraf může naráz obsahovat maximálně dva třístranné podgrafy na protilehlých stranách.



Obrázek 4.7: **Druhá varianta upraveného rozdělení prostoru.** Rozdělení prostoru upravené po zpracování třístranného podgrafu vyžadující pravou, horní a levou stranu.

Jakmile je fáze umísťování potomků do buněk hotová, tak je nutné Děliči metodou `applyToRealSpace` předat skutečnou velikost prostoru zpracovávaného podgrafu. Dělič prostoru poté provede redukci místa, která spočívá v odstranění celých prázdných sloupců a řádků a rozšíření neprázdných buněk do prostoru prázdných buněk. Neprázdné buňky si dělí prostor prázdných buněk v závislosti na poměru minimálních velikostí a obsahů svých uzlů. Určování rozměrů buněk probíhá na stejném principu. Prostor je vždy rozdělen tak, aby všechny buňky v řádku měly stejnou výšku a všechny buňky ve sloupci měly stejnou šířku.

Po dokončení aplikace reálných rozměrů na rozdělený abstraktní prostor získáme podčásti prostoru zpracovávaného podgrafu. Všichni potomci tohoto podgrafu mají svoji příslušnou pod-část jeho prostoru, ve které se musí nacházet. Tyto pod-části jsou relativní vůči

svému rodičovskému podgrafu, čímž je myšleno, že se pod-části nemůžou hýbat relativně vůči sobě, ale můžou se hýbat spolu se svým rodičovským podgrafem vzhledem ostatním podgrafům.

Typicky se setkáme s tím, že podgrafy jsou roztažené podél některé strany či rohu prostoru grafu. Pokud máme více podgrafů roztažených podél jedné strany, tak se od této strany sice nemohou vzdálit, ale mohou měnit svoje vzájemné pořadí. Výjimkou jsou volné podgrafy, které zaujímají svoji minimální velikost vzhledem k jejich potomkům a mohou se volně pohybovat v prostoru jim přiřazené pod-části.

Adaptéry algoritmů

Rozmísťovací algoritmy jsou do aplikace integrovány za použití adaptérů vycházejících z abstraktní třídy `AbstractAdapter` a výčtu `Algorithms`, který obsahuje zkratky a krátké popisy dostupných algoritmů. Princip použití adaptérů již byl do značné míry vysvětlen v rámci návrhu aplikace v podkapitole 3.7. Za zmínku navíc stojí to, že každý adaptér konkrétního algoritmu musí implementovat metody `isLockingOtherNodesSupported` a `isLimitingGraphSpaceSupported`, které vrací pravdivostní hodnotu a říkají, zda adaptovaný algoritmus podporuje funkce uzamykání aktuálně nezpracovávaných uzlů a omezení prostoru pro rozmísťování. Pokud se adaptér algoritmu přislíbí k tomu, že jeho algoritmus tyto funkce podporuje, tak to poté musí vzít v úvahu a v metodě `layout` tyto funkce implementovat. Příklad kompletní metody `layout` lze vidět na výstřižku 4.8.

```
@Override
public void layout() {
    // Prepare for part
    if (allowedNodes == null) {
        allowedNodes = new ArrayList<>(graph.getNodes());
    }
    if (part != null) {
        layout.setInitializer(node -> (Point2D) node.getPosition().clone());
        for (Node node : graph.getNodes()) {
            layout.setLocation(node, node.getPosition());
        }
        layout.initialize();
    }
    layout.reset();
    long timeStart = System.currentTimeMillis();
    while (System.currentTimeMillis() - timeStart < timeout && !layout.done()) {
        layout.step();
    }
    // Copy result
    allowedNodes.forEach(n -> n.setPosition(layout.transform(n)));
}
```

Obrázek 4.8: **Příklad implementace metody `layout`.** Na tomto výstřižku je zobrazena implementace metody `layout` adaptéru algoritmu Kamada-Kawai (třída `AdapterKK`).

Pro přidání nového algoritmu do aplikace poté stačí implementovat nového potomka třídy `AbstractAdapter` adaptujícího tento algoritmus. Dále je nutné přidat zkratku a krátký popisek algoritmu do výčtu `Algorithms`. Nakonec se musí přidat nový případ uvažující tento algoritmus do rozhodování ve statické metodě `getAdapterInstance(Algorithms algorithm, GraphRepresentation graph)` třídy `AbstractAdapter`, která pro vybraný al-

goritmus a daný graf vrací novou instanci adaptéru vybraného algoritmu inicializovanou daným grafem. Nikde jinde v celé aplikaci už poté není nutné provádět žádný zásah.

4.3 Iterační smyčka

Iterační smyčka slouží k provedení rozmísťování a získání výsledků. Při spuštění iterační smyčky se vytvoří seznam s kapacitou pro uživatelem zadaný počet uchovávaných nejlepších výsledků. Následně se postupně aplikují adaptéry na počáteční rozmístění grafu vygenerovaná v předzpracování.

Aplikace adaptéru rozmísťovacího algoritmu na počáteční rozmístění grafu znamená, že se na tento graf opakovaně spustí rozmísťovací algoritmus. Celkově se provede uživatelem specifikovaný počet iterací. Jedna kompletní iterace se provede zavoláním metody `adapterOneIteration` třídy `PreProcessing` (implemetace k vidění na obrázku 4.9). Ite-

```
private static boolean adapterOneIteration(SubGraph currentSubgraph, AbstractAdapter adapter) {
    boolean postprocessResult = false;
    for (Entry<BoundingBox, List<Node>> entry : currentSubgraph.getSubgraphParts().entrySet()) {
        adapter.setPartAndLayout(entry.getKey(), entry.getValue());
        boolean localResult = PostProcessing.completePartPostProcess(entry.getKey(), entry.getValue());
        postprocessResult = postprocessResult || localResult;
    }
    for (SubGraph childSubgraph : currentSubgraph.getSubGraphs()) {
        boolean childResult = adapterOneIteration(childSubgraph, adapter);
        postprocessResult = postprocessResult || childResult;
    }
    return postprocessResult;
}
```

Obrázek 4.9: **Implementace metody `adapterOneIteration` třídy `PreProcessing`.**

race spočívá v tom, že se hierarchicky projdou jednotlivé části prostoru grafu (viz Dělič prostoru), a pro každou část se samostatně spustí rozmísťovací algoritmus. Po dokončení běhu rozmísťovacího algoritmu se vždy uloží pouze změny pozic uzlů náležících zpracovávané pod-části prostoru. Nově získané pozice je potřeba upravit pomocí modulu následného zpracování. Jelikož známe, jaké uzly mohly být algoritmem přesunuty, tak stačí následné zpracování aplikovat na tyto uzly a nemusíme po každém spuštění adaptéru algoritmu procházet celý graf.

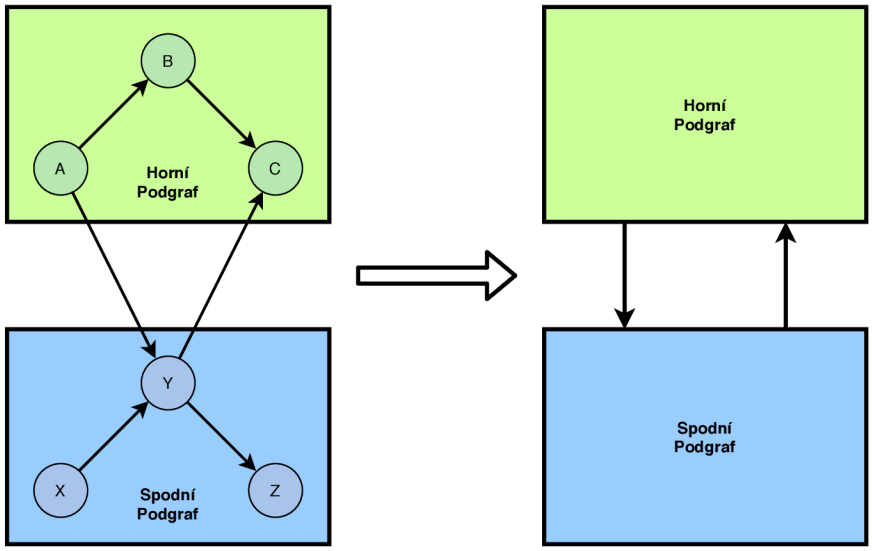
Po dokončení každé kompletní iterace se aktuální stav grafu ohodnotí modulem Hodnocení, a pokud má výsledek lepší skóre, než některý z uložených výsledků, tak se uloží a nejhorší uložený výsledek se zapomene. Pokud ještě nebyl nasbírán daný počet nejlepších výsledků, tak se výsledek vždy uloží.

Na závěr iterační smyčky stačí sesbírané výsledky seřadit podle jejich uloženého skóre a tento seřazený seznam předat dále zbytku aplikace.

Rozmísťování podgrafů

Podgrafy jsou na jednotlivých úrovních grafu rozmísťovány. Toto rozmísťování probíhá tak, že se nejprve pro všechny podgrafy na dané úrovni (ve stejné pod-části prostoru) vytvoří nové hrany, které je budou spojovat s ostatními podgrafy a okolními uzly. Množina nových hran je vytvořena tak, že uzly, které tvoří konce hran a jsou potomky některého ze zpracovávaných podgrafů, jsou v dané hraně nahrazeny svým rodičovským podgrafem. Díky této úpravě hran jsou pak podgrafy propojené se všemi uzly, se kterými jsou propojeni jejich potomci, a je tedy možné k jejich rozmísťování přistupovat stejně, jako k normálním uzlům.

Vedlejším efektem těchto úprav hran je to, že se v určitých případech z původně acyklického grafu stane graf cyklický. Tato situace je názorně vidět na obrázku 4.10. Samotné propojení uzlů je acyklické. Když však abstrahujeme hrany podle výše popsaného postupu, tak z hran `A->Y` a `Y->C` získáme po řadě hrany `Horní Podgraf -> Spodní Podgraf` a `Spodní Podgraf -> Horní Podgraf`, které tvoří kružnici. Tento efekt je hlavním důvodem, proč jsou v aplikaci použity pouze algoritmy, které dokáží pracovat i s cyklickými grafy.



Obrázek 4.10: **Abstrahováním hran podgrafů se graf stává cyklickým.** Vlevo vidíme acyklický graf se dvěma podgrafy, přičemž každý podgraf má tři uzly. Vpravo vidíme abstrakci na úrovni podgrafů, která tvoří cyklický graf.

Hodnocení

Popis funkce modulu hodnocení lze nalézt v kapitole věnující se jeho návrhu 3.9. Zde jsou psána podporovaná hodnoticí pravidla, způsob implementace hodnocení a též jakým způsobem přidat vlastní hodnoticí pravidlo. Seznam dostupných hodnoticích pravidel:

- Křížení hrany a uzlu

  Toto hodnoticí pravidlo počítá počet výskytů křížení hrany a uzlu v grafu.
  Hodnota pro parametr příkazové řádky: `CEN`

- Křížení dvou hran

  Pravidlo počítá počet výskytů křížení dvou hran v grafu.
  Hodnota pro parametr příkazové řádky: `CEE`

- Vyžádaný směr hran zleva doprava

  Pravidlo říká, že hrany mají směřovat zleva doprava. Jsou počítány všechny hrany, jejichž počáteční uzel se nachází více vpravo než jejich koncový uzel.
  Hodnota pro parametr příkazové řádky: `LtR`

- Vyžádaný směr hran zprava doleva

  Opačné pravidlo k pravidlu `LtR`. Říká, že hrany mají směřovat zprava doleva.
  Hodnota pro parametr příkazové řádky: `RtL`

- Vyžádaný směr hran shora dolů

  Pravidlo říká, že hrany mají směřovat shora dolů. Jsou počítány všechny hrany, jejichž počáteční uzel se nachází výše než jejich koncový uzel.
  Hodnota pro parametr příkazové řádky: `TtB`

- Vyžádaný směr hran zdola nahoru

  Opačné pravidlo k pravidlu `TtB`. Říká, že hrany mají směřovat zdola nahoru.
  Hodnota pro parametr příkazové řádky: `BtT`

hodnoticí modul je implementován ve třídě `Score`. Tato třída je výčtem podporovaných hodnoticích pravidel a obsahuje statické metody pro získání výsledku z každého tohoto pravidla. Její metoda `evaluate(GraphRepresentation graph, Map<Score, Integer> scoreRules)` (viz 4.11) dokáže ohodnotit rozmístění grafu v interní reprezentaci za pomoci seznamu hodnoticích pravidel a jejich vah. Hodnocení probíhá tak, že pro každé vybrané hodnoticí pravidlo se spustí funkce, která vrátí počet chyb v grafu dle tohoto pravidla. Výsledky těchto funkcí se vynásobí příslušnými vahami a takto získané hodnoty se sečtou. Tento součet tvoří skóre daného rozmístění grafu.

```java
/**
 * @param graph to evaluate
 * @param scoreRules map of selected score rules with their weights
 * @return evaluates given graph by selected score rules and returns the score
 */
public static long evaluate(GraphRepresentation graph, Map<Score, Integer> scoreRules) {
    long score = 0;
    for (Entry<Score, Integer> entry : scoreRules.entrySet()) {
        Score scoreRule = entry.getKey();
        int weight = entry.getValue().intValue();
        switch(scoreRule) {
            case BtT:
            case LtR:
            case RtL:
            case TtB:
                score += weight * evaluateEdgeDirection(graph, scoreRule);
                break;
            case CEE:
                score += weight * evaluateCrossingsEdges(graph);
                break;
            case CEN:
                score += weight * evaluateCrossingsNode(graph);
                break;
        }
    }
    return score;
}
```

Obrázek 4.11: **Výstřižek z kódu — metoda `evaluate` třídy `Score`.**

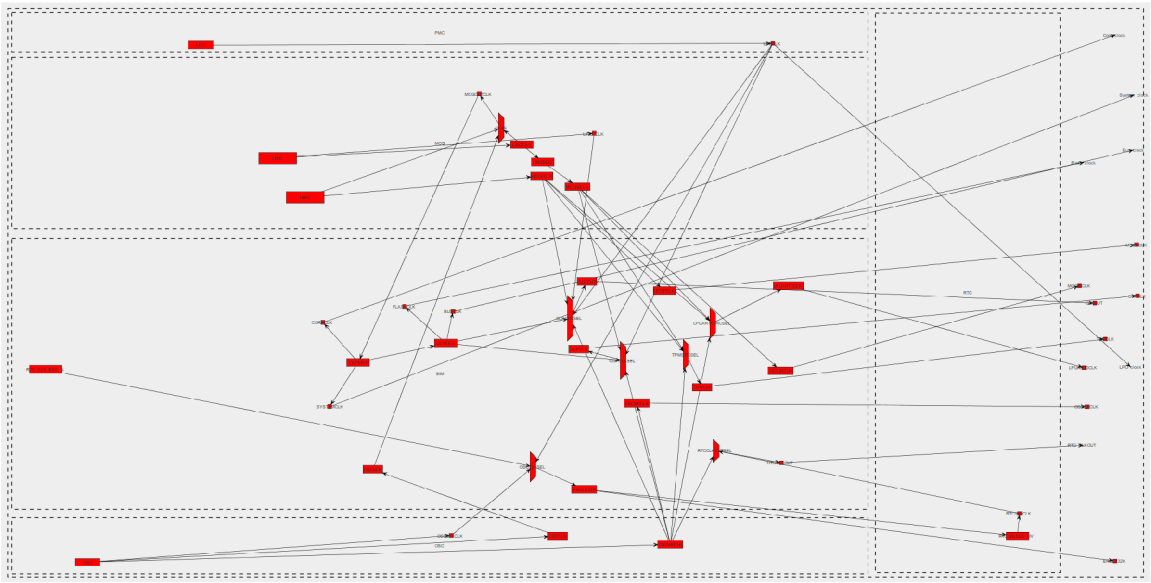
Z popsaného způsobu získávání ohodnocení je možné o získaném hodnocení formulovat několik tvrzení. Čím nižší je hodnocení, tím lepší výsledek máme. Hodnocení není normalizované, takže z ohodnocení rozmístění dvou grafů s různým počtem uzlů a hran není možné

jasně říct, které rozmístění je lepší. Tímto způsobem lze používat pouze skóre různých rozmístění stejného grafu. Dále také samozřejmě není možné porovnávat ohodnocení získané z rozdílných kombinací hodnoticích pravidel a jejich vah.

Výchozí hodnocení je definováno jako `LtR=1000;CEN=40;CEE=2`. Tento zápis znamená, že jsou použita pravidla Vyžádaný směr hran zleva doprava, Křížení hrany a uzlu a Křížení dvou hran. Pravidlo Vyžádaný směr hran zleva doprava zde má velkou váhu tisíc, tudíž toto pravidlo má vysokou přednost před ostatními pravidly. Aplikace tedy upřednostní výsledky, kde co nejvíce hran povede daným směrem i za cenu porušení ostatních pravidel. Pravidlo Křížení hrany a uzlu má váhu čtyřicet, zatímco pravidlo Křížení dvou hran má váhu dva. To znamená, že ve výsledku by se muselo křížit alespoň 21 hran, aby aplikace upřednostnila výsledek, ve kterém se nekříží žádné hrany, ale jedna hrana protíná uzel, jenž dané hraně nenáleží.

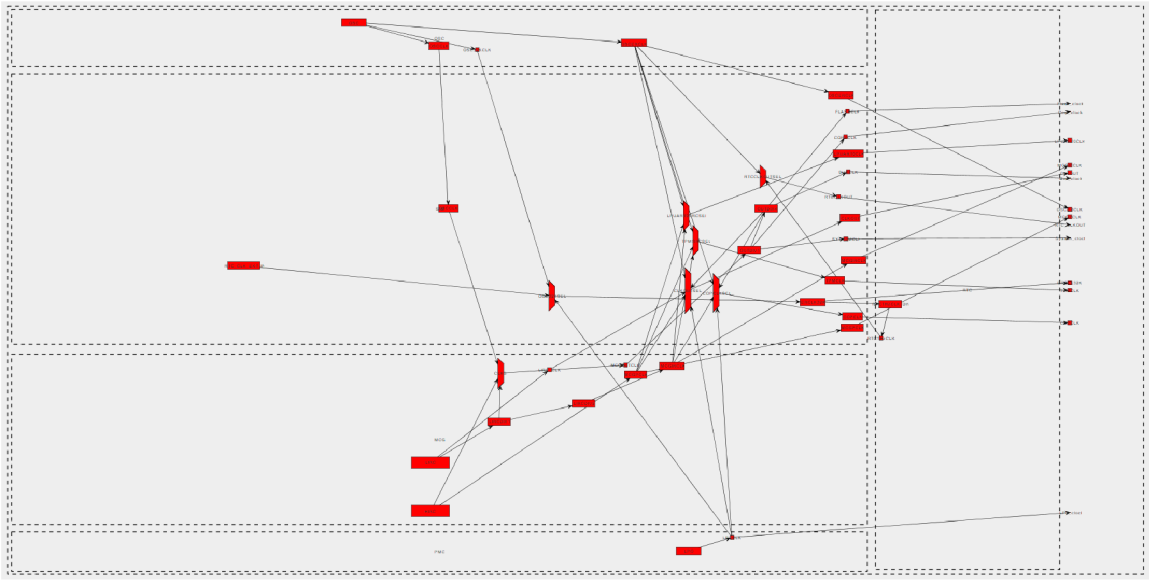
Zcela nové hodnocení je možné vytvořit nejen výběrem jiných hodnoticích pravidel, ale pouze změnou jejich vah. Kdybychom například dali pravidlům Křížení hrany a uzlu a Křížení dvou hran vyšší váhu, než pravidlu Vyžádaný směr hran zleva doprava, tak bychom získali výsledky, ve kterých by se obecně vyskytovalo méně křížení obou typů za cenu více porušení vyžádaného směru hran. Jiným příkladem by mohlo být využití vah na definování dvou vyžádaných směrů hran, přičemž jeden má vyšší prioritu. Mohli bychom tedy požadovat, aby hrany vedly primárně zleva doprava a sekundárně shora dolů.

Na následujících dvou obrázcích si můžeme uvést příklad rozumných rozmístění stejného grafu, přičemž obě rozmístění mají různá ohodnocení. Graf rozmístěný na těchto obrázcích obsahuje 54 uzlů. První rozmístění na obrázku 4.12 bylo vytvořeno s použitím algoritmu ISOM se zapnutou funkcí omezení prostoru rozmísťování. Oproti tomu na ob-



Obrázek 4.12: **Rozmístění grafu s ohodnocením 16930.** Jedná se o diagram hodin mikro-kontroleru MKL03Z. Graf byl rozmístěn výslednou aplikací. Rozmístění má skóre 16930 bodů, ohodnoceno výchozím hodnocením.

rázku 4.13 můžeme vidět druhé rozmístění vygenerované Fruchterman-Reingoldovým algoritmem s uzamknutím aktuálně nezpracovávaných uzlů i s omezením prostoru pro rozmísťování. Druhé rozmístění má více než dvakrát nižší skóre než to první, takže je jednoznačně



Obrázek 4.13: **Rozmístění grafu s ohodnocením 8040.** Jedná se o diagram hodin mikrokontroleru MKL03Z. Graf byl rozmístěn výslednou aplikací. Rozmístění má skóre 8040 bodů, ohodnoceno výchozím hodnocením.

lepší (alespoň dle daných hodnoticích pravidel). Jak lze vidět, tak na druhém rozmístění je pouze sedm hran majících špatný směr.

Přidání vlastního hodnoticího pravidla je velmi jednoduché. Stačí implementovat funkci, která vyhodnocuje toto pravidlo, a upravit výčet `Score` přidáním zkratky a popisu nového pravidla. Poslední úpravou je přidání volání funkce vyhodnocující toto pravidlo na příslušné místo do metody `evaluate` třídy `Score`. Žádné další zásahy do kódu aplikace nejsou potřeba.

Kapitola 5

Zhodnocení práce

Výsledek práce byl vyhodnocen na deseti grafech různých velikostí od 54 do 236 uzlů. Pro každý graf bylo spuštěno těchto sedm konfigurací:

- Fruchterman-Reingoldův algoritmus
- Fruchterman-Reingoldův algoritmus s omezením prostoru pro rozmísťování
- Fruchterman-Reingoldův algoritmus s uzamknutím aktuálně nezpracovávaných uzlů
- Fruchterman-Reingoldův algoritmus s uzamknutím aktuálně nezpracovávaných uzlů i s omezením prostoru pro rozmísťování
- algoritmus ISOM
- algoritmus ISOM s omezením prostoru pro rozmísťování
- algoritmus Kamada-Kawai

Ostatní možnosti programu byly nastaveny stejně pro všechny konfigurace. Počáteční rozmístění bylo vytvořeno deterministicky a bylo vykonáno deset tisíc iterací pro každé jednotlivé spuštění. Celkově bylo tedy provedeno sedmdesát spuštění, což je sedm set tisíc iterací. Tyto výsledky, uvedené v příloze C, byly sesbírávány po dobu dvou týdnů s použitím celkem tří různých zařízení.

Velikost prostoru pro rozmístění grafu byla určena tak, že grafy s méně než sto uzly dostaly prostor 3000x1500 pixelů, a grafy se sto a více uzly dostaly prostor 6000x3000 pixelů. Prostor grafu má dvakrát větší šířku než výšku, jelikož vybrané hodnoticí pravidlo specifikuje, že hrany mají směřovat zleva doprava.

Pro získání těchto výsledků měla být využita funkce vynucení deterministického chování. Při získávání výsledků autor však došel k tomu, že tato funkce až příliš zkresluje výsledné skóre, a tudíž nakonec byly použity pouze výsledky získané s povolenou náhodností v algoritmech, a pouze počáteční rozmístění bylo vytvořeno deterministicky. Bylo empiricky naměřeno, že funkce vynucení deterministického chování zhoršuje výsledné skóre v průměru až o osm procent.

Výsledky (viz příloha C) byly ohodnoceny výchozím hodnoticím pravidlem popsaným v oddílu 4.3. Hodnocení výsledků jednotlivých grafů bylo zprůměrováno a tyto průměry byly porovnávány s celkovým průměrem ostatních konfigurací. Ze získaných výsledků jsme došli k závěru, že konfigurací s nejlepšími výsledky je Fruchterman-Reingoldův algoritmus s funkcemi omezení prostoru pro rozmísťování a uzamknutí aktuálně nezpracovávaných

uzlů. Tato konfigurace vygenerovala o 14,77 procent lepší výsledky než průměr ostatních konfigurací. Příklad jí vygenerovaného grafu lze vidět na obrázku 4.13.

Pokud se podíváme na samotné algoritmy a srovnáme navzájem výsledky jejich nejlepších konfigurací, tak můžeme algoritmy seřadit od nejlepšího následovně. První místo obsadí Fruchterman-Reingoldův algoritmus, jehož nejúspěšnější konfigurace byla popsána výše. Na druhém místě nalezneme algoritmus ISOM v konfiguraci s funkcí omezení prostoru pro rozmísťování. Tato konfigurace vykazovala průměrné ohodnocení pouze o 0,14 procent horší, než vykazoval průměr ostatních konfigurací. Poslední místo drží algoritmus Kamada-Kawai s výsledky o 8,29 procent horšími vůči průměru ostatních konfigurací.

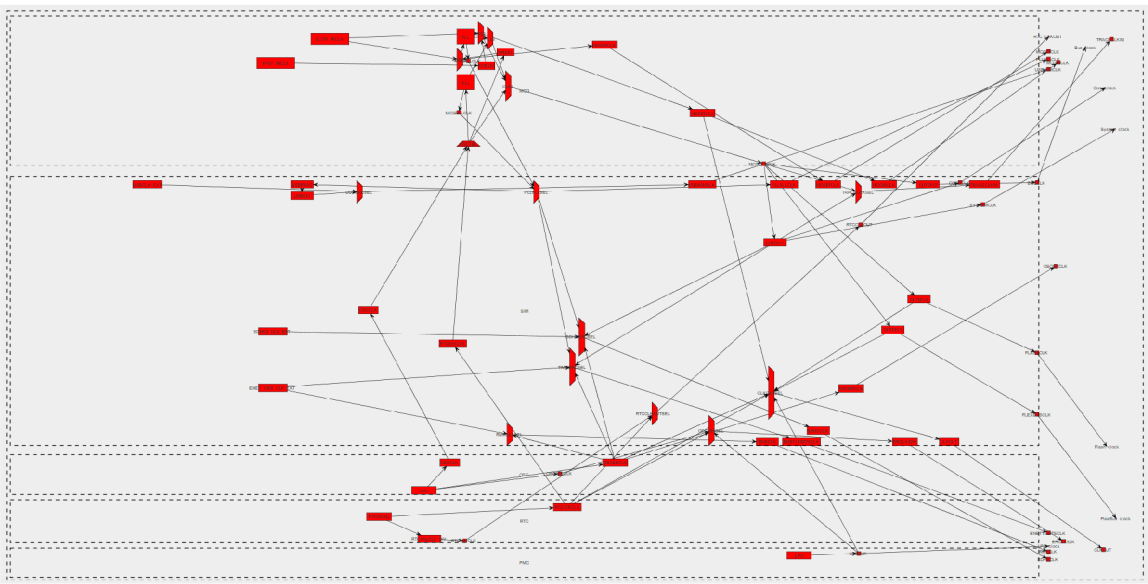
Znamenají tedy tyto závěry, že by měl uživatel vždy použít Fruchterman-Reingoldův algoritmus s jeho nejlepší konfigurací? Další aspekt, který pro hodnocení algoritmů můžeme použít, je rychlost, s jakou algoritmus získá výsledek. Tato rychlost je samozřejmě ovlivněná strojem, na kterém algoritmus běží, a mnoha dalšími faktory. Při získávání výsledků autor došel ke zjištění, že lze rychlost implementací jednotlivých algoritmů relativně seřadit od nejrychlejšího po nejpomalejší. Nejrychlejším algoritmem je bezesporu algoritmus ISOM, který je často až několikrát rychlejší než ostatní dva použité algoritmy. Druhé místo z pohledu rychlosti drží algoritmus Fruchterman-Reingold. Obecně nejpomalejším z těchto tří algoritmů je tedy algoritmus Kamada-Kawai.

Na základě ohodnocení výsledků algoritmů a jejich rychlosti bychom mohli spekulovat o efektivitě daných algoritmů. Jelikož měření rychlosti či doby potřebné pro získání výsledku komplikuje mnoho dalších okolních proměnných, tak by kvantitativní určení efektivity algoritmů bylo příliš složité. Pokud se vrátíme zpět k otázce, jaký algoritmus by měl tedy uživatel použít, tak můžeme definovat jednoduchou pomůcku. Jestliže má uživatel dostatek času a požaduje kvalitní výsledky, tak bude nejlepší použít Fruchterman-Reingoldův algoritmus s funkcemi omezení prostoru pro rozmísťování a uzamknutí aktuálně nezpracovávaných uzlů. Pokud uživatel chce získat výsledky rychle, tak by měl použít algoritmus ISOM v konfiguraci s funkcí omezení prostoru pro rozmísťování. Důležité je také zmínit to, že pokud uživatel bude chtít docílit nejrychleji co nejvyššího počtu iterací, tak by taktéž měl zvolit algoritmus ISOM s omezením prostoru pro rozmísťování.

Nyní zbývá vyhodnotit přínos funkcí omezením prostoru pro rozmísťování (dále jen omezení prostoru) a uzamknutí aktuálně nezpracovávaných uzlů (dále jen uzamknutí uzlů). K zajímavému zjištění dojdeme, při pozorování výsledků funkce omezení prostoru. Použitím samotné této funkce u algoritmu Fruchterman-Reingold bylo docíleno zhoršení výsledků o 4,4 procenta oproti použití stejného algoritmu bez této funkce. Na druhou stranu u algoritmu ISOM tato funkce zlepšila výsledné ohodnocení o 2,31 procent. Zjistili jsme tak, že samotná tato funkce může být jak nápomocná tak škodlivá v závislosti na principu, jakým zvolený algoritmus funguje. U funkce uzamknutí uzlů jsme došli k mnohem pozitivnějším výsledkům, a to, že výsledné skóre Fruchterman-Reingolda algoritmu zlepšila o celých 15 procent. Největšího úspěchu však dosáhla kombinace těchto funkcí, a to zlepšení o 16,9 procent. Můžeme tedy říci, že při použití Fruchterman-Reingoldova algoritmu by měl uživatel vždy povolit obě tyto funkce zároveň.

Ke zhodnocení výsledků samotné práce jako celku lze říci to, že práce splnila zadání, jelikož podporuje všechny nezbytné funkce a je schopna rozmístit i grafy s vysokým počtem uzlů. Příklad slušného rozmístění grafu lze vidět na obrázku 5.1.

V této práci jsou zobrazeny jen grafy s menším počtem uzlů, protože obecně platí, že čím více má graf uzlů, tím více místa pro dosažení rozumného rozmístění je potřeba. Velké grafy by tedy nebylo možné rozumně vložit a jako ukázka funkce mohou dobře posloužit i grafy menších rozměrů.



Obrázek 5.1: **Diagram hodin mikro-kontroleru MK60DN.** Tento graf byl rozmístěn algoritmem Fruchterman-Reingold s povolenou funkcí uzamknutí aktuálně nezpracovávaných uzlů. Graf má 81 uzlů a jeho rozmístění získalo ohodnocení 17514 bodů (ohodnoceno výchozím hodnocením).

Kapitola 6

Závěr

Cílem této práce bylo vytvořit aplikaci pro rozmísťování uzlů v acyklických orientovaných grafech s podporou grafického uživatelského rozhraní, polygonálních uzlů, podgrafů a uzlů s vyžádanou pozicí. Tento cíl, včetně podpory všech požadovaných funkcí, byl úspěšně splněn a výsledná aplikace nesoucí název DAGio byla vytvořena. Program umožňuje uživateli výběr z několika podporovaných algoritmů, má mnoho nastavitelných možností, podporuje ovládání z příkazové řádky i z grafického rozhraní. Aplikace navíc definuje způsob, jakým je možné hodnotit výsledná rozmístění. S dostatkem času a se správným nastavením je aplikace schopná vygenerovat přehledná a srozumitelná rozmístění.

Největším oříškem, na který jsem narazil při navrhování řešení, bylo jakým způsobem přistoupit ke zpracování podgrafů zároveň s ohledem na uzly s vyžádanými pozicemi. Pro tento účel jsem vytvořil komplexní heuristiku rozdělování prostoru. Tato heuristika spolu s moduly předběžného a následného zpracování zajišťuje podporu neomezeného zanoření podgrafů, rozmísťování podgrafů a samozřejmě i správné umístění uzlů s vyžádanými pozicemi.

Díky tvorbě této práce jsem se naučil hodně o teorii grafů. Největším osobním přínosem však pro mě je to, že jsem si vyzkoušel, jaké to je pracovat na půl roku trvajícím akademickém projektu, a kolik práce, úsilí a plánování je s tím spojeno.

Do budoucna v této práci vidím velký potenciál pro její rozšiřování. Jeden z možných směrů zlepšování aplikace by mohlo být přidávání nových rozmísťovacích algoritmů a nových hodnoticích pravidel. Další možností by mohla být kompletní podpora cyklických grafů, jelikož aplikace současně plně podporuje jen grafy acyklické. Skvělé by bylo rozšířit grafické rozhraní ze zobrazování grafů na jejich interaktivní editování. Dále by bylo možné podpořit různé typy hran, například pravoúhlé lomené hrany, hrany ve tvaru křivky a rozpojené hrany (tzv. teleporty). V neposlední řadě by mohla být přidána podpora více typů výstupních a popřípadě i vstupních formátů.

Literatura

[1] Demel, J. *Grafy a jejich aplikace*. Praha: Academia, 2002. ISBN 80-200-0990-6.

[2] Fruchterman, T. M. J. a Reingold, E. M. Graph Drawing by Force-directed Placement. *Software: Practice and Experience*. [online]. 1991, roč. 21, č. 11, [cit. 2020-01-18], s. 1129–1164. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.13.8444. ISSN 00380644.

[3] Kamada, T. a Kawai, S. An Algorithm for Drawing General Undirected Graphs. *Information Processing Letters*. [online]. 1989, roč. 31, č. 1, [cit. 2020-01-18], s. 7–15. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.387.7401. ISSN 00200190.

[4] Kobourov, S. G. Spring Embedders and Force Directed Graph Drawing Algorithms. *CoRR*. [online]. 2012, abs/1201.3011, [cit. 2020-01-18]. Dostupné z: http://arxiv.org/abs/1201.3011.

[5] Meyer, B. Self-Organizing Graphs — A Neural Network Perspective of Graph Layout. In: Whitesides, S. H., ed. *Graph Drawing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, s. 246–262 [cit. 2020-01-18]. ISBN 978-3-540-37623-1.

[6] Pecinovský, R. *Návrhové vzory: 33 vzorových postupů pro objektové programování*. 1. vyd. Brno: Computer Press, 2007. ISBN 978-80-251-1582-4.

Příloha A

Preference — možnosti a argumenty programu

Zde je kompletní podrobný seznam všech dostupných možností programu, včetně detailního popisu co daná funkce dělá. Seznam možností má následující strukturu: tučně jméno možnosti uvedené v GUI, česky jméno možnosti, odpovídající parametr příkazové řádky, popis a vysvětlení možnosti a příklad použití této možnosti v příkazové řádce. Možnosti jsou seřazeny podle toho, jak jsou viditelné v GUI shora dolů zleva doprava. Pokud má nějaký parametr hodnotu, tak mezi parametr a jeho hodnotu lze zapsat mezeru nebo znak = („rovná se“). V příkladech jsou názorně uvedeny obě varianty oddělovače parametru a hodnoty.

- **Main graph input** — Zadání souboru s hlavním vstupem

  Parametr: `-i, --input=FILE`

  Tato možnost dovoluje uživateli vybrat hlavní vstupní soubor. Daný soubor musí existovat a aplikace musí mít oprávnění číst tento soubor. Soubor lze zadat jak relativní, tak absolutní cestou. V GUI lze cestu vyhledat pomocí dialogu, který se zobrazí po kliknutí na tlačítko „Select...“ nacházející se vpravo na stejném řádku.

  Příklad: `-i input.txt` = vybere soubor „input.txt“ v aktuální složce.

- **TinyCAD input** — Zadání souboru s vedlejším vstupem

  Parametr: `-t, --tinycad=TINYCAD`

  Výběr souboru s TinyCAD vstupem pro získání informací o polygonálním tvaru uzlů. Pokud si uživatel nepřeje načíst TinyCAD vstup, tak stačí nechat tuto kolonku v GUI prázdnou. Daný soubor musí existovat a aplikace musí mít oprávnění číst tento soubor. Soubor lze zadat jak relativní, tak absolutní cestou. V GUI lze cestu vyhledat pomocí dialogu, který se zobrazí po kliknutí na tlačítko „Select...“ nacházející se vpravo na stejném řádku.

  Příklad: `-t diagram.dsn` = vybere soubor „diagram.dsn“ v aktuální složce.

- **Generate TinyCAD output?** — Generování TinyCAD výstup do souboru

  Parametr: `--[no-]generate-output`

  Určuje, zda uživatel chce vygenerovat výstup do souboru/souborů. Ve výchozím nastavení je výstup generován.

Příklad: `--no-generate-output` = výstup nebude uložen do souboru.

- **TinyCAD output path** — Zadání cesty ke složce pro výstupní soubory

  Parametr: `-o, --output=DIR`

  Umožňuje uživateli vybrat cestu ke složce, kam budou vygenerovány všechny výstupní soubory. Aplikace musí mít práva zapisovat v dané složce. Cílová složka nemusí existovat, ale potom musí mít aplikace práva vytvořit všechny neexistující předky této složky. V GUI lze cestu vyhledat pomocí dialogu, který se zobrazí po kliknutí na tlačítko „Select...“ nacházející se vpravo na stejném řádku. Pokud složka obsahuje soubory se stejnými jmény jako nově vygenerované soubory, tak budou původní soubory přepsány. Přepsání souborů při opakovaném generování do stejné složky se dá zabránit použitím možnosti „Přidat časovou známku do jmen výstupních souborů“ nebo změnou prefixu výstupních souborů.
  Tato možnost nemá žádný vliv, pokud je generování výstupních souborů vypnuté.

  Příklad: `-o ./vystup` = vygeneruje výstupní soubory do složky „vystup“ v aktuálním adresáři.

- **Selected layout algorithm** — Výběr rozmísťovacího algoritmu

  Parametr: `-a, --algorithm=ALGORITHM`

  Slouží k výběru rozmísťovacího algoritmu. Povolené hodnoty jsou:
  FR = Fruchterman-Reingoldův algoritmus
  KK = algoritmus Kamada-Kawai
  ISOM = algoritmus založený na Meyerových metodách samo-organizujících se grafů
  BEST = Vyzkouší všechny dostupné algoritmy a automaticky vybere nejlepší výsledky. Násobí počet provedených iterací počtem dostupných algoritmů.
  Pokud si uživatel nezvolí algoritmus, tak bude použit Fruchterman-Reingoldův algoritmus.
  Celkový počet provedených iterací se rovná součinu počtu iterací, počtu počátečních rozmístění a počtu vybraných algoritmů, přičemž počet vybraných algoritmů je roven počtu dostupných algoritmů, pokud je zvolen algoritmus BEST, jinak je roven jedné.

  Příklad: `-a KK` = bude použit rozmísťovací algoritmus Kamada-Kawai.

- **Number of iterations** — Počet iterací

  Parametr: `-I, --iterations=ITER_COUNT`

  Určuje, kolik iterací iterační smyčky bude provedeno pro vybraný algoritmus pro jedno počáteční rozmístění. Pokud je vybrán algoritmus BEST, tak se provede zadaný počet iterací pro každý z dostupných rozmísťovacích algoritmů. Obecně platí, čím více iterací proběhne, tím lepší výsledky dostaneme, ale také tím déle bude programu trvat, než dosáhne výsledku. Výchozí hodnota je sto iterací.
  Celkový počet provedených iterací se rovná součinu počtu iterací, počtu počátečních rozmístění a počtu vybraných algoritmů, přičemž počet vybraných algoritmů je roven počtu dostupných algoritmů, pokud je zvolen algoritmus BEST, jinak je roven jedné.

  Příklad: `-I=1000` = bude provedeno tisíc iterací iterační smyčky.

- **Number of start initializations** — Počet počátečních rozmístění

  Parametr: `-S, --start-inits=START_INITS`

Určuje, kolik různých počátečních rozmístění se má v rámci předzpracování vytvořit. Pro každé počáteční rozmístění bude spuštěn zadaný počet iterací vybraného rozmísťovacího algoritmu. Výchozí hodnota je jedno počáteční rozmístění.
Tato možnost nemá žádný vliv, pokud je vypnuta tvorba náhodných počátečních rozmístění.
Celkový počet provedených iterací se rovná součinu počtu iterací, počtu počátečních rozmístění a počtu vybraných algoritmů, přičemž počet vybraných algoritmů je roven počtu dostupných algoritmů, pokud je zvolen algoritmus BEST, jinak je roven jedné.

Příklad: `-S 10` = bude vytvořeno deset počátečních rozmístění.

- **Number of best results** — Počet nejlepších výsledků k uchování

  Parametr: `-N, --number-of-results=RESULTS`

  Udává, kolik nejlepších výsledků se má uchovávat při běhu iterační smyčky. Tento počet uchovaných výsledků poté bude zobrazen a/nebo uložen do souboru. Co to znamená „nejlepší výsledek“ se dočtete v kapitole 3.9. Hodnota nesmí být vyšší než celkový počet iterací. Výchozí hodnota je jeden výsledek.

  Příklad: `-N=10` = bude uchováno deset nejlepších výsledků.

- **Random initialization of nodes position** — Náhodná počáteční inicializace

  Parametr: `--[no-]random`

  Tato možnost určuje, zda bude počáteční rozmístění uzlů v grafu provedeno náhodně nebo deterministicky. Deterministická inicializace na začátku rozmístí uzly do grafu rovnoměrně do řádků a sloupců se stejnými rozestupy. Ve výchozím nastavení probíhá inicializace náhodně. Pro nejlepší výsledek ponechejte zapnutou náhodnou inicializaci. Tvorba náhodných počátečních rozmístění je automaticky vypnuta vždy, když je zapnuté vynucení deterministického chování aplikace.

  Příklad: `--no-random` = počáteční inicializace bude deterministická.

- **Limit layout space** — Omezit prostor pro rozmísťování

  Parametr: `--limit-layout-space`

  V rámci předzpracování je prostor grafu rozdělen na části (toto rozdělení je ovlivněno uzly s vyžádanou pozicí a podgrafy, viz 4.2). Povolení této možnosti způsobí to, že při rozmísťování uzlů v adaptéru vybraného algoritmu bude prostor, v rámci kterého může algoritmus rozdělovat uzly, omezen vždy pouze na aktuálně zpracovávanou pod-část. Ve výchozím nastavení je tato možnost vypnuta. Vliv této možnosti na skóre grafu je prozkoumán v kapitole 5.
  Tato možnost je dostupná pouze pro algoritmy Fruchterman-Reingold a ISOM.

  Příklad: `--limit-layout-space` = prostor pro rozmísťování bude omezen na aktuálně zpracovávanou pod-část.

- **Lock other nodes** — Uzamknout aktuálně nezpracovávané uzly

  Parametr: `--lock-other-nodes`

  V rámci předzpracování je prostor grafu rozdělen na části (toto rozdělení je ovlivněno uzly s vyžádanou pozicí a podgrafy, viz 4.2). Aktivováním této možnosti budou pozice uzlů nepatřící do aktuálně zpracovávané pod-části grafu uzamknuty, takže s nimi rozmísťovací algoritmus nebude pohybovat, což ovlivní průběh rozmísťování. Ve výchozím

nastavení je tato možnost vypnuta. Vliv této možnosti na skóre grafu je prozkoumán v kapitole 5.
Tato možnost je dostupná pouze pro algoritmus Fruchterman-Reingold.

Příklad: `--lock-other-nodes` = uzamkne pozice aktuálně nezpracovávaných uzlů.

- **Parse nodes with preferred position** — Zpracovat uzly s vyžádanou pozicí

  Parametr: `--[no-]nodes-preferred-position`

  Tato možnost umožňuje ovládat, zda budou zpracovávány vyžádané pozice (seznamy `Top`, `Left`, `Right` and `Bottom` ve vstupním souboru, viz 3.2) bez nutnosti upravovat vstupní soubor. Ve výchozím nastavení jsou vyžádaná pozice zpracovávány.

  Příklad: `--no-nodes-preferred-position` = vyžádaná pozice uzlů nebude brána v potaz.

- **Enable size post-processing** — Povolit následné zpracování

  Parametr: `--[no-]sizepostprocess`

  Tato možnost slouží k ovládání následného zpracování uzlů týkající se jejich velikosti (viz 3.8). Konkrétně dokáže povolit nebo zakázat části následného zpracování, které předchází přetékání a překrývání uzlů. Ve výchozím stavu je následné zpracování zapnuto. Následné zpracování se nedoporučuje vypínat, pokud není zapnuta možnost Omezení prostoru pro rozmísťování. Vypínat se též nedoporučuje, pokud chcete dosáhnout nejlepších výsledků. V každém případě, pokud bude následné zpracování vypnuto, tak je možné, že ve výsledku se budou objevovat překrývající se či přetékající uzly.

  Příklad: `--no-sizepostprocess` = zakáže následné zpracování uzlů.

- **Max iterations of post-processing** — Maximální počet iterací následného zpracování

  Parametr: `--max-iterations-postprocess=MAX_ITER`

  Následné zpracování pracuje iterativně, přičemž v každé iteraci vyřeší několik problémů. Tato možnost umožňuje nastavit maximální počet iterací následného zpracování. Výchozí hodnota je 100 iterací. Tuto hodnotu je vhodné zvýšit v případě, že se zobrazí varování říkající, že došlo k dosažení tohoto limitu a zároveň ve výsledku bude možné vidět překrývající se či přetékající uzly.

  Příklad: `--max-iterations-postprocess 200` = maximální počet iterací následného zpracování nastaven na dvě stě.

- **Score options** — Možnosti týkající se výběru hodnotících pravidel

  Parametr: `-s, --score=SCORE_RULE=WEIGHT[;SCORE_RULE=WEIGHT...]`

  Slouží k definování hodnocení. Hodnocení se definuje tak, že uživatel vybere aktivní hodnotící pravidla a přiřadí jim váhu. Hodnotící pravidlo se v GUI vybere zaškrtnutím příslušného políčka. Váha se pak danému pravidlu nastaví zadáním kladné celočíselné hodnoty do kolonky na stejném řádku.
  V příkazové řádce se hodnotící pravidla vyberou zapsáním ve tvaru `PRAVIDLO=VÁHA`. Jednotlivá pravidla se v tomto zápisu oddělují středníkem a celý zápis musí být zapsán bez mezer. Výchozí hodnotou je `LtR=1000;CEN=40;CEE=2`. Vysvětlení této hodnoty a popis všech dostupných hodnotících pravidel lze najít v kapitole 4.3.

Příklad: `-s LtR=50;CEN=50;CEE=10` = definuje hodnocení, které dává stejnou váhu na to, aby hrany mířily zleva doprava a aby hrany nekřížily uzly, a pětinovou váhu na to, aby se hrany navzájem nekřížily.

- **Graph width** — Šířka grafu

  Parametr: `-x, --width=WIDTH`

  Jednoduše udává šířku prostoru pro rozmístění grafu v pixelech. Výchozí hodnota je 1280 pixelů.

  Příklad: `-x 1920` = nastaví šířku grafu na 1920 pixelů.

- **Graph height** — Výška grafu

  Parametr: `-y, --height=HEIGHT`

  Jednoduše udává výšku prostoru pro rozmístění grafu v pixelech. Výchozí hodnota je 720 pixelů.

  Příklad: `-y 1080` = nastaví výšku grafu na 1080 pixelů.

- **Border for nodes** — Velikost okraje uzlů

  Parametr: `--node-border=BORDER`

  Nastaví velikost dodatečného okraje okolo uzlů v pixelech. Výchozí hodnota je 10 pixelů, což znamená, že 10 pixelů na každou stranu od okraje uzlu se nebude nacházet žádný další uzel. Jelikož všechny uzly okolo sebe mají tento okraj, tak to znamená, že minimální rozestup dvou uzlů je dvojnásobek této hodnoty.

  Příklad: `--node-border=5` = nastaví velikost okraje uzlů na 5 pixelů.

- **Border for subgraphs** — Velikost okraje podgrafů

  Parametr: `--subgraph-border=BORDER`

  Nastaví velikost dodatečného okraje okolo podgrafů v pixelech (jedná se o vnější okraj podgrafu). Výchozí hodnota je velikost okraje uzlů.

  Příklad: `--subgraph-border=50` = nastaví velikost okraje podgrafů na 50 pixelů.

- **Output file prefix** — Prefix jmen výstupních souborů

  Parametr: `-p, --output-prefix=OUTPUT_PREFIX`

  Nastaví prefix jmen výstupních souborů. Tento řetězec se objeví ve jméně souboru hned na začátku. Výchozí hodnota je „graph“.
  Tato možnost nemá žádný vliv, pokud je generování výstupních souborů vypnuté.

  Příklad: `-p diagram` = „diagram_pořadí_skóre_algoritmus.dsn“ bude tvar jmen výstupních souborů.

- **Draw arrows into output file** — Generování šipek do výstupních souborů

  Parametr: `--[no-]arrows`

  Pokud je povoleno, tak vzhled všech vstupních pinů (konců hran) nenačtených z vedlejšího vstupu bude zobrazen jako šipka. V opačném případě generované vstupní piny stejně jako výstupní piny nebudou mít grafické znázornění. Ve výchozím stavu je generování šipek povoleno.
  Tato možnost nemá žádný vliv, pokud je generování výstupních souborů vypnuté.

  Příklad: `--no-arrows` = vypne generování šipek do výstupních souborů.

- **Override pin type** — Přepsat tvar pinů ve výstupních souborech

  Parametr: `--override-pin`

  Dovoluje uživateli přepsat podobu načtených vstupních pinů (konců hran) tak, aby vypadaly jako konce šipek. Ve výchozím stavu je tato možnost vypnuta.
  Tato možnost nemá žádný vliv, pokud je generování výstupních souborů vypnuté nebo pokud je generování šipek do výstupních souborů vypnuté.

  Příklad: `--override-pin` = zajistí, že všechny konce hran budou vypadat jako šipky.

- **Add timestamp to generated files** — Přidat časovou známku do jmen výstupních souborů

  Parametr: `--timestamp`

  Přidá časovou známku na konec jmen výstupních souborů, takže výsledný formát jmen bude vypadat následovně „prefix_pořadí_skóre_algoritmus_rok-měsíc-den_hodina-minuta-sekunda.dsn“. Ve výchozím nastavení je tato možnost deaktivována.
  Tato možnost nemá žádný vliv, pokud je generování výstupních souborů vypnuté.

  Příklad: `--timestamp` = přidá časovou známku do jmen výstupních souborů.

- **Show result?** — Zobrazit graficky nejlepší výsledky

  Parametr: `--[no-]show`

  Umožňuje uživateli ovlivnit, zda budou na konci rozmísťování zobrazena okna s nejlepšími výsledky. Ve výchozím stavu je zobrazování výsledků povoleno. Doporučeno vypnout, pokud uživatel ukládá mnoho výsledků, aby nebyl zahlcen mnoha okny.

  Příklad: `--no-show` = vypne zobrazení oken s nejlepšími výsledky.

Dále následuje seznam parametrů dostupných pouze z příkazové řádky:

- **Nápověda**

  Parametr: `-h, --help`

  Zobrazí informace o programu, nápovědu jak pracovat s parametry, seznam všech parametrů se stručným vysvětlením, seznam dostupných algoritmů, seznam dostupných hodnotících pravidel s krátkým popisem a pár příkladů spuštění programu.

  Příklad: `-h` = zobrazí nápovědu.

- **Zobrazení GUI**

  Parametr: `--gui, --GUI`

  Použití toho parametru slouží k zobrazení hlavního okna grafického uživatelského rozhraní.

  Příklad: `--gui` = zobrazí GUI.

- **Vynutit deterministické chování**

  Parametr: `--force-deterministic-behaviour`

  Dovoluje uživateli deaktivovat veškerou náhodnost v aplikaci a získat tak možnost opakovaně reprodukovat stejné výsledky při použití stejného vstupu a stejných nastavení. Při použití této možnosti se zároveň automaticky deaktivuje tvorba náhodného

počátečního rozmístění. Pro dosažení nejlepšího výsledku ponechte povolenou náhodnost v chování aplikace.
Tato funkce je dostupná pouze z příkazové řádky. Důvod lze najít v kapitole 4.2.

Příklad: `--force-deterministic-behaviour` = vynutí deterministické chování rozmísťovacích algoritmů.

Příloha B

Použité nástroje a knihovny

Ta podkapitola obsahuje seznam nástrojů, programů a knihoven použitých k tvorbě výsledné aplikace a celé této práce. U každé položky je uveden název, k čemu byla použita a webová stránka. U knihoven navíc bude napsána i licence.
Seznam použitých knihoven:

- Knihovna JUNG (Java Universal Network/Graph Framework)

  Tato knihovna byla použita zejména díky implementacím rozmísťovacích algoritmů Fruchterman-Reingold, Kamada-Kawai a ISOM. Navíc byla použita i pro grafické zobrazení výsledku rozmísťování.

  Licence: BSD

  Web: jung.sourceforge.net

- Knihovna Picocli

  Knihovna Picocli v této práci slouží pro zpracovávání argumentů příkazové řádky a tvorbu nápovědy.

  Licence: Apache 2.0

  Web: picocli.info

- Knihovna GSON

  Z knihovny GSON byl využit syntaktický analyzátor formátu JSON pro zpracování hlavního vstupu.

  Licence: Apache 2.0

  Web: github.com/google/gson

- Knihovna jUnit

  Knihovna jUnit byla použita pro tvorbu jednotkových testů.

  Licence: Eclipse Public License 1.0

  Web: junit.org

Seznam použitých nástrojů a programů:

- Java™ Platform, Standard Edition Development Kit (JDK™) version 8

  Celá aplikace je implementována v jazyku Java verze 8. K implementaci byl použit Java vývojářský balíček (JDK = Java Development Kit). Pro spuštění programu je vyžadováno běhové prostředí jazyku Java (JRE = Java Runtime Environment) verze 8.

  Web: www.java.com

- Eclipse IDE for Java Developers

  Tento program byl použit jako hlavní vývojové prostředí, ve kterém byla vyvíjena téměř celé aplikace.

  Web: www.eclipse.org

- NetBeans IDE

  Vývojové prostředí NetBeans bylo použito pro tvorbu designu grafického uživatelského rozhraní.

  Web: www.netbeans.org

- GitKraken

  Program GitKraken byl použit pro správu Git repozitáře této práce.

  Web: www.gitkraken.com

- GitLab

  Webový nástroj GitLab zajišťoval Git repozitář pro tuto práci.

  Web: gitlab.com

- Overleaf

  Webový LaTex edit Overleaf sloužil k sepsání této práce.

  Web: www.overleaf.com

Příloha C

Naměřené hodnoty

V následující tabulce C.1 jsou uvedeny naměřené hodnoty použité pro zhodnocení jednotlivých rozmísťovacích algoritmů a jejich konfigurací.

Tabulka C.1: Skóre vygenerovaných rozmístění

	FR	FR LLS	FR LON	FR LLS LON	ISOM	ISOM LLS	KK
MKL03Z	13442	16060	8086	8040	16154	16698	15684
MKL43Z	17252	21632	10268	11554	18878	19260	17636
MK24	26052	20998	18220	18398	26916	27176	26432
LPC844	24242	28018	23150	21960	24816	26172	31370
MK60DN	27708	28906	17514	20294	25532	25870	28224
LPC54114	35622	32962	28306	32976	27622	28610	29666
MK64	25572	28062	20256	22876	27594	27262	29122
LPC55S69	85648	87410	89434	84662	73080	84090	78358
RT1021	102854	104863	85474	77950	110268	82884	97682
RT1052	92810	102340	82726	76194	90436	93082	106638

Sloupce reprezentují jednotlivé měřené konfigurace. Řádky obsahují ohodnocení jednotlivých rozmístění grafů hodin uvedených reálných mikro-kontrolérů. Vysvětlení zkratek konfigurací:

- FR = Fruchterman-Reingoldův algoritmus
- FR LLS = Fruchterman-Reingoldův algoritmus s omezením prostoru pro rozmísťování
- FR LON = Fruchterman-Reingoldův algoritmus s uzamknutím aktuálně nezpracovávaných uzlů
- FR LLS LON = Fruchterman-Reingoldův algoritmus s uzamknutím aktuálně nezpracovávaných uzlů i s omezením prostoru pro rozmísťování
- ISOM = algoritmus ISOM
- ISOM LLS = algoritmus ISOM s omezením prostoru pro rozmísťování
- KK = algoritmus Kamada-Kawai

Ostatní použitá nastavení těchto konfigurací jsou popsány v kapitole Zhodnocení práce.

Následující tabulka C.2 uvádí počty uzlů, podgrafů a hran grafů hodinového signálu výše uvedených mikro-kontrolérů.

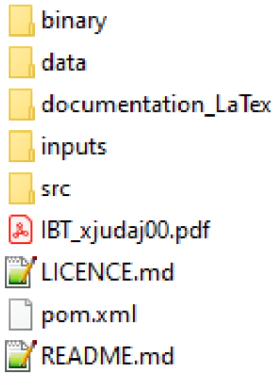
Tabulka C.2: Parametry grafů hodinového signálu uvedených mikro-kontrolérů

Mikro-kontroléry	Uzly	Hrany	Podgrafy
MKL03Z	54	65	5
MKL43Z	64	79	5
MK24	78	93	6
LPC844	79	119	1
MK60DN	81	97	5
LPC54114	82	139	3
MK64	85	103	6
LPC55S69	150	262	3
RT1021	202	278	4
RT1052	236	353	4

Příloha D

Obsah paměťového média

Struktura paměťového média je viditelná na obrázku D.1.



Obrázek D.1: **Struktura paměťového média.**

Popis jednotlivých složek a souborů:

- binary — Složka binary obsahuje sestavenou spustitelnou aplikaci spolu se spouštěcími skripty pro operační systémy Windows a Linux. V této složce se dále nachází podsložka DAGio_lib, která obsahuje knihovny aplikace DAGio.
- data — Složka data obsahuje tabulku v programu Microsoft Excel, ve které jsou uvedeny stejné výsledky jako v tabulce C.1 spolu se statistickými výpočty, jejichž hodnoty jsou použity v kapitole 5.
- documentation_LaTex — Složka obsahuje zdrojové soubory tohoto textu včetně všech použitých obrázků. PDF verzi textu lze z tohoto obsahu vygenerovat.
- inputs — Složka inputs obsahuje vstupní soubory použité pro vygenerování dat v tabulce C.1. Navíc obsahuje i další vstupy vhodné pro vyzkoušení funkčnosti aplikace.
- src — Složka src obsahuje zdrojové kódy aplikace DAGio.
- IBT_xjudaj00.pdf — Tento soubor je PDF verze textu práce.
- LICENCE.md — Soubor LICENCE.md obsahuje licenci pro použití aplikace.

- pom.xml — Soubor pom.xml je konfigurační soubor pro nástroj Maven, který slouží k sestavení aplikace.
- README.md — Soubor README.md obsahuje informace o aplikaci, návod k jejímu sestavení a spuštění a příklady použití aplikace.

Příloha E

Návod na sestavení a spuštění aplikace

Tato příloha obsahuje návod na sestavení a spuštění aplikace spolu s příklady použití. Program byl implementován a testován na systému Windows 10, ale jelikož je jazyk Java multiplatformní, tak by měl program fungovat i na ostatních operačních systémech.

E.1 Spuštění aplikace

Požadavky pro spuštění:
– Nainstalované prostředí Oracle Java Runtime Environment verze 8

Spouštěcí skripty se liší v závislosti na vašem operačním systému. Pro spuštění na systému Microsoft Windows použijte skripty s koncovkou `.bat`. Pro spuštění na systému Linux použijte skripty s koncovkou `.sh`.

Pro zapnutí aplikace DAGio s grafickým uživatelským rozhraním spusťte ze složky binary skript `Launch_DAGio_with_GUI.bat` (nebo `Launch_DAGio_with_GUI.sh`). Toto spuštění vytvoří soubor log.txt, do kterého se bude ukládat standardní chybový výstup. Alternativou k použití skriptu je zavolání příkazu `java -jar DAGio.jar --gui 2>log.txt` ze složky binary. Část `2>log.txt` slouží k přesměrování standardního chybového vstupu do souboru a není povinná.

Pro spuštění aplikace z příkazové řády slouží skript `DAGio.bat` (nebo `DAGio.sh`). Argumenty programu předáte jednoduše jako `DAGio.bat <ARGUMENTY>`, přičemž `<ARGUMENTY>` nahradíte vámi zvolenými argumenty. Příklad `DAGio.bat --help` zobrazí nápovědu programu. Alternativně lze použít příkaz `java -jar DAGio.jar <ARGUMENTY>` zavolaný ze složky binary.

Následují příklady použití aplikace:

- `java -jar DAGio.jar --help` — zobrazí nápovědu programu
- `java -jar DAGio.jar --gui` — zapne aplikaci s aktivním GUI
- `java -jar DAGio.jar -i ../inputs/input.txt -o ../output --no-show -x 600 -y 600 -a ISOM -I 1000` — Spustí tisíc iterací rozmísťování grafu ze souboru input.txt. Prostor grafu bude velký 600 krát 600 pixelů. Bude použit algoritmus ISOM. Bude vygenerován jeden výsledek, který nebude zobrazen a bude uložen do složky output.

Všechny tyto příklady je zamýšleno spouštět ze složky binary. Další příklady lze najít v souboru README.md nebo v rámci nápovědy aplikace po použití argumentu `--help`.

E.2 Sestavení aplikace

Požadavky pro sestavení:
– Nainstalovaný vývojářský balíček Oracle Java Development Kit verze 8
– Nainstalovaný program Maven (verze 3 nebo vyšší)

Pro úspěšné sestavení aplikace postupujte podle následujících kroků:

1. Nejdříve si nainstalujte potřebný software uvedený v požadavcích pro sestavení.
2. Nastavte si proměnnou prostředí se jménem `JAVA_HOME` na umístění vaší instalace vývojářského balíčku Java Development Kit.
3. Přejděte do kořenové složky této práce.
4. V této složce spusťte příkaz `mvn package`. Tento příkaz by měl stáhnout všechny potřebné závislosti programu. Pro tento krok je nutné připojení k internetu. Pokud vše proběhne v pořádku, tak bude vytvořena nová složka target obsahující soubor `DAGio-1.0.jar`.
5. Pro spuštění vygenerovaného souboru `DAGio-1.0.jar` je nutné jej přesunout do složky binary. Soubor `DAGio-1.0.jar` se musí nacházet ve stejném adresáři, jako složka DAGio_lib, která obsahuje programem používané knihovny.