České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačů

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Michal Macháček**

Studijní program: Elektrotechnika a informatika (magisterský), strukturovaný
Obor: Výpočetní technika

Název tématu: **Evoluční metaheuristiky pro hledání bisekce grafu**

Pokyny pro vypracování:

1. Prostudujte problematiku optimálního dělení grafů (GPP – Graph Partitioning Problem) a jeho speciální případ hledání bisekce grafu (GBP – Graph Bisection Problem).
2. Prostudujte existující (meta)heuristické metody pro řešení GBP.
3. Navrhněte a naimplementujte metaheuristiku založenou na evolučním algoritmu pro řešení GBP.
4. Experimentálně ověřte funkčnost navrženého algoritmu na vybraných testovacích instancích.
5. Dosažené výsledky vyhodnoťte a srovnejte s jinými existujícími (meta)heuristikami.

Seznam odborné literatury:

1. Benlic, U. and Hao, J.K.: Hybrid metaheuristics for the graph partitioning problem, Hybrid Metaheuristics Studies in Computational Intelligence, Vol. 434, 2013, pp. 157-185 http://www.info.univ-angers.fr/pub/hao/papers/HMGPP2012.pdf

2. Luke, S.: Essentials of Metaheuristics. Lulu, 2013, 2. vydání, http://cs.gmu.edu/~sean/book/metaheuristics/

Vedoucí: Ing. Jiří Kubalík, Ph.D.

Platnost zadání: do konce letního semestru 2014/2015

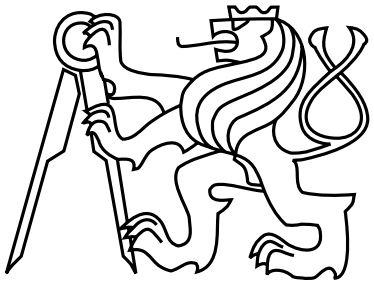
L.S.

doc. Ing. Filip Železný, Ph.D.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 25. 2. 2014

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

Evoluční metaheuristiky pro hledání bisekce grafu

Bc. Michal Macháček

Vedoucí práce: Ing. Jiří Kubalík, Ph.D.

Studijní program: Elektrotechnika a informatika

Obor: Výpočetní technika

3. ledna 2015

Poděkování

Děkuji vedoucímu práce Ing. Jiřímu Kubalíkovi, Ph.D. za poutavé uvedení do problematiky evolučních algoritmů a především za množství užitečných rad a konzultací při tvorbě této práce. Děkuji také své rodině a přátelům za trpělivost a podporu po dobu mého studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Příbrami dne 3. 1. 2015 .....

Abstract

This thesis first defines the graph bisection problem. It summarizes existing heuristic and evolutionary solution methods. The most important parts of the evolutional algorithm are described. It also describes the Kernighan–Lin heuristics for the graph bisection optimization.

New hybrid evolutionary-based algorithm for solving the graph bisection problem is proposed and described in the thesis. It combines the evolutionary algorithm with the Kernighan–Lin heuristics as a local optimizer. The implementation includes a user interface that provides an environment for experimenting with the proposed algorithms and makes it possible to visually inspect and manipulate with the solutions produced by the algorithm.

The proposed algorithm was experimentally evaluated on a set of the most frequently used benchmark graphs found in relevant literature. Results achieved with the proposed algorithm were compared to the pure Kernighan–Lin heuristic ones showing that the proposed algorithm outperforms the Kernighan–Lin ones on all of them but one.

Abstrakt

V úvodu práce definuje problém bisekce grafu. Shrnuje existující heuristické a evoluční metody jeho řešení. U evolučních metod popisuje hlavní součásti evolučního algoritmu. Popisuje Kernighan–Lin heuristiku pro optimalizaci bisekce grafu.

V práci je navržen a podrobně popsán vlastní hybridní evoluční algoritmus kombinující evoluční prohledávání s lokální optimalizaci pomocí Kernighan–Lin heuristiky. Součástí implementace je i grafické uživatelské rozhraní umožňující nastavení a spouštění experimentů. Výsledky výpočtu jsou vizuálně znázorněny s možností manuálních úprav.

Výsledky dosažené navrženým algoritmem jsou experimentálně ověřeny na často používaných testovacích grafech uváděných v relevantních publikacích. Jsou také porovnány s výsledky dosaženými pomocí čisté Kernighan–Lin heuristiky. Na všech testovaných grafech kromě jediného navržený algoritmus dokázal překonat Kernighan–Lin heuristiku.

Obsah

Kapitola 1

Úvod

Problém hledání minimální bisekce grafu (PMBG) spočívá v nalezení rozdělení grafu na dva podgrafy tak, aby součet vah hran vedoucích mezi jednotlivými podgrafy byl minimální. Nutnou podmínkou je, aby se velikosti podgrafů nelišily více než o 1. Praktické využití je například při rozdělení elektrického obvodu na dvě desky tak, aby bylo třeba co nejméně propojek. Dále se může uplatnit při řízení toku v sítích *(load balancing)*, detekci komunit na sociálních sítích, plánování úloh apod.

PMBG je NP těžký problém [9]. Hlavním omezením je proto velká časová náročnost výpočtu rostoucí s velikostí vstupního grafu. Deterministické algoritmy, které řeší PMBG optimálně, jsou použitelné pouze pro malé instance problému. Větší instance se dají řešit pomocí heuristických metod, založených např. na hladovém prohledávání prostoru řešení. Není však zaručeno, že nalezené řešení bude optimální. Příkladem takové heuristiky je Kernighan–Lin algoritmus (K–L) popsaný v kapitole 3.6.

Vzhledem k velikosti stavového prostoru je výhodné při řešení použít evoluční heuristiky nebo metaheuristiky. Jedním z možných přístupů je šlechtění jedinců reprezentujících konkrétní rozdělení vstupního grafu. V odborných publikacích je popsána řada technik používaných při evolučním řešení PMBG.

Evoluční algoritmus typicky realizuje hrubé prohledávání stavového prostoru. Na takto evolučně nalezená řešení je možné uplatnit ještě lokální optimalizaci. Takový postup se nazývá hybridním evolučním algoritmem.

Cílem této práce je navrhnout a implementovat hybridní evoluční algoritmus pro řešení PMBG, který bude rozšířením Kernighan–Lin algoritmu. Jeho evoluční část bude provádět globální vzorkování prohledávaného prostoru. Základním principem bude kombinace evolučního a hladového vytváření sekvence prohazování vrcholů mezi jednotlivými podgrafy. Jako lokální optimalizátor evolučně nalezených řešení bude použita varianta klasického K–L algoritmu.

V práci bude experimentálně ověřen předpoklad, že toto rozšíření bude na testovaných grafech dosahovat lepších výsledků než klasický K–L algoritmus. Bude rovněž provedeno srovnání výsledků vlastní implementace K–L s vybranou jinou implementací. Součástí vyhodnocení výsledků bude i srovnání s nejlepšími dosaženými výsledky jiných algoritmů prezentovanými v relevantních publikacích.

Text je členěn do následujících kapitol. V kapitole 2 jsou definovány všechny potřebné základní pojmy. V kapitole 3 jsou popsány neevoluční metody řešení bisekce grafu. Kapitola 4 upřesňuje popis jednotlivých druhů evolučních algoritmů. Hlavní postupy a součásti evolučních algoritmů pro řešení PMBG jsou popsány v kapitole 5. Navržený hybridní evoluční algoritmus je popsán v kapitole 6 a popis klíčových části jeho implementace je v kapitole 7. Nejčastěji používané testovací grafy jsou uvedeny v kapitole 8. Výběr konkrétních grafů pro experimenty, jejich nastavení a výsledky obsahuje kapitola 9. Práce končí zhodnocením výsledků v kapitole 10 a závěrem v kapitole 11.

Kapitola 2

Definice problému

Problém rozdělení grafu *(graph partitioning problem)* spočívá obecně v rozdělení grafu na k podgrafů tak, aby se počet vrcholů v žádné dvojici podgrafů nelišil více než o 1 a aby byl minimalizován součet vah hran vedoucích mezi jednotlivými podgrafy. Bisekce grafu je speciální případ pro $k = 2$. V této práci se budeme zabývat výhradně hledáním minimální bisekce grafu.

Mějme neorientovaný graf $G = (V, E)$, kde V je množina n jeho vrcholů a E množina e jeho hran. Úkolem grafové bisekce je rozdělit graf G do dvou podgrafů $A(V_A, E_A)$ a $B(V_B, E_B)$ o počtech vrcholů n_A,n_B. V případě sudého n se musí $n_A = n_B$, v případě lichého n bude jeden z podgrafů obsahovat jeden vrchol navíc, tedy

$$|n_A - n_B| \leq 1$$

Každý z vrcholů z V musí být přiřazen právě do jednoho z těchto podgrafů, tedy musí platit

$$V_A \bigcup V_B = V$$

$$V_A \bigcap V_B = \emptyset$$

a z toho vyplývající vztah $n_A + n_B = n$.

Hrana, která spojuje vrcholy ze stejného podgrafu, se nazývá vnitřní nebo interní *(inter–cluster)*. Hrana, která spojuje vrcholy z různých podgrafů, se nazývá vnější, externí nebo hraniční *(intra–cluster, cut edge)*. Rozdělení grafu se provede odstraněním všech hraničních hran. Není přitom požadováno, aby výsledné podgrafy $A(V_A, E_A)$ a $B(V_B, E_B)$ byly po odstranění těchto hran souvislé.

Kvalita rozdělení se hodnotí součtem vah hran, které je potřeba odstranit. Pro kvalitu rozdělení se používá označení velikost řezu *(cutsize)*. V případě, že hrany nejsou ohodnoceny, považujeme jejich váhy za jednotkové a *cutsize* je dáno pouze jejich počtem. Kvalita rozdělení grafu je tím lepší, čím menší je *cutsize*. Snažíme se tedy *cutsize* minimalizovat.

Kapitola 3

Neevoluční metody řešení a heuristiky

Heuristika je obecný postup řešení problému. Nezaručuje nalezení optimálního řešení, je však snadno použitelná a dává rychle výsledky pro obecná vstupní data (bez záruky jejich optimality). Vychází ze zkušeností a obecných postupů v dané problematice.

Pokud je známo geometrické rozložení vrcholů grafu, je možné použít metodu proložení přímkou(3.1) nebo random spheres(3.2). Ostatní popisované metody, tedy Lokální optimalizace(3.3), BFS bisekce(3.4), Coarsening(3.5) a Kernighan–Lin(3.6) nepotřebují geometrickou informaci znát. Pracují s obecnými grafy. V této práci se budeme věnovat obecným grafům bez geometrické informace. V případě vstupních dat se souřadnicemi vrcholů budeme tyto souřadnice ignorovat.

3.1 Proložení přímkou

Metoda je založena na protnutí grafu přímkou tak, aby dělila jeho vrcholy na dvě poloviny s počtem vrcholů lišícím se maximálně o 1. Prvky podgrafů budou odpovídat prvkům v polorovinách vymezených dělicí přímkou. Parametry dělicí přímky lze optimalizovat např. metodou nejmenších čtverců tak, aby výsledná velikost řezu byla co nejnižší. Podrobnější popis v [4].

3.2 Random spheres

Tato metoda konstruuje rozdělení grafu na základě kružnic se středy ve vrcholech a takovými poloměry, aby žádný vrchol nebyl uvnitř více než k takto vymezených kruhů. Autoři ukazují, že existuje dělicí kružnice protínající definovaný omezený počet těchto kružnic a rozděluje vrcholy na části uvnitř sebe a vně v definovaném poměru. Rovněž navrhují algoritmus pro její nalezení. Metoda pracuje obecně v d–rozměrném prostoru. Podrobnější popis v [13].

3.3 Lokální optimalizace

Obecný postup při hledání řešení optimalizující nějaké kritérium. Postupně provádíme menší lokální změny na kandidátském řešení. Tím procházíme jeho okolí v prostoru řešení, posouváme se postupně směrem k jeho nejbližším sousedům. Tento postup iterativně opakujeme

tak dlouho, dokud se nepřestane zlepšovat sledované kritérium nebo dokud není dosaženo jeho požadované hodnoty.

Nevýhodou této metody je velká pravděpodobnost uvíznutí v lokálním extrému, což vyplývá ze samotného principu prohledávání nejbližšího okolí. Použití lokální optimalizace při evolučním hledání bisekce grafu popisuje např. [17].

3.4 Bisekce grafu průchodem do šířky

Tato metoda je založená na průchodu grafem do šířky.

Algoritmus 1 BFS bisekce

Input: Graf G

Output: Bisekce grafu G

1: Zvol počáteční vrchol v_0
2: Projdi graf do šířky s označováním nejkratší vzdálenosti od v_0
3: Nalezni v_{MAX} jako nejvzdálenější vrchol od v_0
4: **return** Rozdělení vrcholů na poloviny bližší k v_0, resp. v_{MAX}

Problematická je už volba počátečního vrcholu. Na ní přitom závisí výsledná kvalita rozdělení. Tento algoritmus může být použitelný pro vytvoření iniciálního rozdělení, které bude vstupem dalšího algoritmu.

3.5 Coarsening

Název by se dal přeložit jako „zhrubnutí“. Shluknutím několika vhodných vrcholů do jednoho zástupného vrcholu se postupně zjednodušuje struktura grafu. Když už je dost jednoduchá, aplikuje se nějaký algoritmus grafové bisekce nad grafem z těchto zástupných vrcholů. Poté se zástupné vrcholy expandují do své původní podoby.

3.6 Kernighan–Lin algoritmus

Kernighan–Lin algoritmus (K–L) nazývaný také K–L heuristika iterativně vylepšuje zadané vstupní rozdělení grafu. Jeho typické použití je lokální optimalizace výstupu nějakého jiného algoritmu. Zlepšení aktuálního řešení se hledá hladovou konstrukcí sekvence prohození dvou vybraných vrcholů mezi oběma podgrafy.

Uvažujme graf G rozdělený na podgrafy A a B. Pro každý vrchol se spočítá součet vah jeho vnitřních hran I_v a vnějších hran E_v.

$$D_v = E_v - I_v$$

je rozdíl mezi vnější a vnitřní hodnotou vah u tohoto vrcholu. Zisk *(gain)* g, kterého dosáhneme prohozením vrcholů a, b je

$$g_{a,b} = D_a + D_b - 2w_{a,b}$$

kde $w_{a,b}$ je váha hrany mezi oběma vrcholy. Pokud taková neexistuje, $w_{a,b} = 0$. Vybereme vrchol z A a z B tak, aby tento výběr maximalizoval zisk při jejich prohození. Vybrané vrcholy vyjmeme z původních množin a přepočítáme ostatní hodnoty D z takto zmenšených A a B. To můžeme pro vrchol x, kde $x \in A - \{a\}$ jednoduše udělat tak, že zohledníme váhy hran $w_{x,a}$ a $w_{x,b}$ podle vztahu

$$D'_x = D_x + 2w_{x,a} - 2w_{x,b}$$

a analogicky pro druhý podgraf. Jednou prohozené vrcholy se již v dalších iteracích neuvažují, protože prohození již prohozeného vrcholu není žádnou změnou. Tato prohazování iterativně opakujeme, dokud původní množiny nevyprázdníme. Nakonec spočítáme, při provedení kolika $k \geq 0$ prvních prohození z výsledné sekvence prohození bude maximalizován výsledný celkový zisk a tolik těchto prohození provedeme.

Algoritmus 2 Kernighan–Lin

Input: Graf $G(V, E)$ a jeho iniciální rozdělení na podgrafy A,B
Output: Optimalizované rozdělení grafu G

1: **repeat**
2: Pro všechny vrcholy z A a B spočítej jejich D
3: **repeat**
4: Vyber vrcholy $a \in A$ a $b \in B$ s maximální ziskem $g[i]$ při prohození
5: Přemísti a z A do X a b z B do Y
6: Přepočítej hodnoty D ze zmenšených A a B
7: **until** Nejsou vyprázdněny A a B
8: Spočítej k tak, aby maximalizovalo zisk $gain = \sum_{i=0}^{k} g[i]$
9: **if** $gain > 0$ **then** Proveď prvních k prohození vrcholů z X a Y v A a B
10: **until** Je dosahováno zisku
11: **return** Optimalizované rozdělení do podgrafů A a B

Algoritmus je podrobně popsán v článku autorů [8] nebo v [16]. Existuje jeho optimalizovaná verze Fiduccia-Mattheyses algoritmus, která pomocí vylepšených datových struktur zlepšuje jeho časovou složitost.

3.7 Další heuristiky

Podle [8] se tyto heuristiky neosvědčily, mohou však posloužit jako zdroj fragmentů pro vývoj nových heuristik.

- **Náhodná řešení**. Rychlé, ale nedává dobré výsledky. Nízká pravděpodobnost úspěchu vzhledem k velikosti prohledávaného prostoru.
- **Ford - Fulkerson**. (Max flow min cut) algoritmus, který hledá maximální tok při minimálním řezu. Problém ale je, že není nijak specifikovatelná velikost výsledných podgrafů.
- **Clustering**. Klasické shlukování, i zde jsou problémy s velikostí výsledných podgrafů.

Kapitola 4

Evoluční metody řešení

Tato kapitola se soustřeďuje na popis evolučních metod řešení PMGB a jejich klíčových součásti. K řešení PMGB je možno použít řadu přístupů z oblasti evolučních algoritmů nebo genetického programování. Dají se kategorizovat do následujících základních tříd – klasické evoluční algoritmy, hybridní algoritmy a hyperheuristiky.

4.1 Klasické evoluční algoritmy

Jsou to klasické evoluční algoritmy, které šlechtí populaci kandidátských řešení. Prvotním vstupem je graf určený k rozdělení. Jedinec v tomto případě reprezentuje jedno konkrétní rozdělení vstupního grafu. Počáteční rozdělení může být buď náhodné, nebo předpočítané některou z heuristik. Fitness jedince je definována velikostí *cutsize* jím reprezentovaného rozdělení. Evoluční algoritmus se snaží *cutsize* minimalizovat.

4.2 Hybridní algoritmy

V tomto případě je evoluční algoritmus kombinován s lokální optimalizací šlechtěných kandidátů. Ukazuje se, že samotný evoluční algoritmus je vhodný pouze na hrubé prohledávání stavového prostoru. Naopak lokální optimalizátor často uvázne v lokálním extrému, protože primárně prohledává nejbližší okolí zlepšovaného kandidáta. V kombinaci EA s lokální optimalizací je tak možné zvýšit jeho výkonnost.

V [9] a jím odkazovaných článcích bylo zjištěno, že lokální optima tvoří konvexní obal okolo globálního optima. Je tedy výhodné použít operátor křížení jako konvexní vyhledávání. Dále autoři zjistili, že operátory křížení mají v tomto případě obecnou tendenci soustředit se na centrální oblast prohledávaného problému.

4.3 Hyperheuristiky

Základní princip je ten, že hyperheuristiky prohledávají prostor heuristik, ne prostor řešení daného konkrétního problému. Pomocí metod genetického programování se z fragmentů (elementárních operací) stávajících heuristik, případně z nově navržených fragmentů, šlechtí

nová heuristika řešící daný problém. Nešlechtí se tedy řešení zadané instance problému, ale heuristika k jeho řešení.

Podrobnější popis hyperheuristik je na [1]. Pravidelně aktualizovaný přehled publikací o hyperheuristikách je na [14]. Hyperheuristikám pro řešení grafových problémům se věnuje např. [11].

Kapitola 5

Evoluční algoritmy

Následující popis je platný zejména pro evoluční algoritmy, které šlechtí jedince odpovídající nějakému rozdělení vstupního grafu. Chromozom reprezentuje jedno konkrétní rozdělení vstupního grafu. Tyto postupy a zkušenosti bude možné využít i při vývoji navrženého (hybridního) evolučního algoritmu. Souhrnný přehled o evolučních přístupech je v [9].

5.1 Reprezentace chromozomu

Genotypem nazýváme jedno konkrétní zakódování řešení. Fenotypem rozumíme jedno konkrétní řešení. **Redundance** nastává v případě, že je možné jedno řešení vyjádřit více různými, ale ekvivalentními zápisy. To rozšiřuje velikost prohledávaného prostoru. Operátory zajišťující diverzitu populace nemusí dokázat rozlišit stejné nebo jen mírně odlišné jedince, kteří mají zcela jiný zápis. Redundance se může projevit různými poměry počtu genotypů na odpovídající fenotypy.

1:1 ... ideální varianta bez redundance

1:N ... nepotřebuje normalizaci

N:1 ... je vhodná normalizace

Redundance však může být v některých speciálních případech užitečná. Podle článku odkazovaného v [12], čím více má dané rozdělení vnitřních hran, tím má i více redundantních zápisů. Proto se takoví kandidáti lépe prosazují. Autoři dosáhli lepších výsledků u *edge* kódování (5.1.2) než u vertex–to–cluster kódování (5.1.1).

Slepota *(blindness)* nastává tehdy, když kódování neumožňuje reprezentovat některé možné instance z prostoru řešení. Může se teoreticky stát i to, že žádné správné řešení nebude moci být daným kódováním reprezentováno, tedy ani nalezeno. Příklad, při kterém k tomuto problému může dojít, je popsán v kapitole 5.1.2.

Podrobný popis typů kódování, redundance a slepoty je uveden v [12].

5.1.1 Vertex–to–cluster kódování

Každý vrchol má přiřazeno číslo podle toho, do kterého podgrafu (clusteru) patří. Rozdělení grafu $G(V, E)$ je tedy reprezentováno polem čísel délky $n = |V|$, jehož prvky mohou nabývat tolika hodnot, na kolik podgrafů je graf rozdělen. Při bisekci je počet podgrafů $k = 2$, takže prvky pole budou mít hodnotu z $(0, 1)$ při indexování podgrafů od 0.

Nevýhodou tohoto kódování je redundance. Existuje $k!$ možností, jak namapovat k čísel na k clusterů. Tedy existuje $k!$ možností, jak zakódovat totéž rozdělení. To zbytečně rozšiřuje prohledávaný prostor. Stejné bisekce ve dvou redundantních zápisech jsou např. `110000` a `001111`. Výhodou je naopak to, že kódování netrpí slepotou, je možné zakódovat do clusterů i izolované vrcholy. Další možnou výhodou je to, že velikost reprezentace rozdělení není závislá na počtu hran.

5.1.2 Edge kódování

V případě tohoto kódování je rozdělení reprezentováno informací o tom, které hrany jsou zachovány a které odstraněny. Typické hodnoty jsou 1...odstraněná hrana, 0...ponechaná hrana. Neboli, pro každou hranu určujeme je–li inter–cluster nebo intra–cluster. Pro reprezentaci bisekce grafu $G(V, E)$ je třeba pole čísel délky $e = |E|$, kde hodnoty jsou z $(0, 1)$.

Prvním nedostatkem tohoto kódování je fakt, že jeho délka může enormně narůstat v grafech s velkým počtem hran. Mnohem vážnějším nedostatkem je ale slepota, protože v případě izolovaných vrcholů nelze zakódovat, do kterého clusteru patří. K izolovaným vrcholům totiž nevede žádná hrana, o které by se dalo zaznamenat že je zachována nebo přerušena. Protože v definici problému izolované vrcholy připouštíme, není toto kódování příliš užitečné pro další experimenty.

5.1.3 BFS uspořádání genomu

Při této metodě jsou při vertex–to–cluster reprezentaci uloženy vrcholy v poli v pořadí podle BFS průchodu grafem. Bez ní je uložení provedeno v pořadí vrcholů podle jejich očíslování, které může být zvoleno v obecném případě libovolně. Struktura grafu se nijak nemění, pouze se symbolicky přečíslují vrcholy.

Výhody tohoto způsobu uložení se projeví při křížení, které potom lépe zachovává shluky sobě blízkých vrcholů.

5.1.4 Fractional code

V [12] je popsána reprezentace rozdělení pomocí $v + 1$ racionálních čísel. V nich je prvních v vyhrazeno vrcholům a zbývající číslo pro počet clusterů. Tato metoda netrpí slepotou. Trpí ovšem redundancí v označování clusterů a redundancí v reprezentacích desetinného čísla zlomkem.

5.1.5 Multi–Attractor Gene Reordering

Autoři [7] navrhli přečíslování uzlů pro lepší zakódování tak, aby jednotlivé sekce grafu byly v kódování pospolu. Začíná od náhodně vybraného vrcholu *(attractor)*. Může být vylepšeno použitím dvou nebo více atraktorů. Ne každá volba atraktorů je dobrá. Experimenty prováděné na grafech z [19] ukázaly zlepšení výkonu.

5.2 Fitness

Z požadavku na co nejmenší *cutsize* bisekce vyplývá přirozená definice fitness jedince i jako velikost řezu jím reprezentovaného rozdělení:

$$Fitness(i) = cutsize$$

Evoluční algoritmus tedy musí fungovat tak, aby minimalizoval fitness. Jiná možnost je transformovat fitness tak, aby hodnota rostla s klesajícím *cutsize*

$$Fitness(i) = \frac{1}{1 + cutsize}$$

Normalizovat fitness jedince i lze podle [18] předpisem

$$Fitness(i) = \frac{(cutsize_w - cutsize_i) + (cutsize_w - cutsize_b)}{3}$$

kde w je nejhorší jedinec v populaci a b nejlepší jedinec v populaci. Tímto způsobem však dosáhneme pouze normalizace, dva rozdílní kandidáti se stejnou fitness se tím nijak neodliší. Přitom je žádoucí, aby metoda výpočtu fitness dokázala rozlišit výhodnějšího kandidáta ze dvou se stejným *cutsize*.

5.3 Křížení

Při křížení se nejvíce využívají klasické operátory

- Jednobodové *(one–point)*
- Vícebodové *(multi–point)*
- Rovnoměrné *(uniform)*

Ukazuje se, že je výhodné zachovávat beze změny tu část genotypu, která je oběma kandidátům společná, viz [18] a [10].

5.4 Mutace

Používají se klasické operátory

- Prohození hodnot na dvou náhodných pozicích v chromozomu
- Náhodná změna na n náhodných pozicích
- Inverze na n náhodných pozicích (při dvouhodnotovém kódování)

5.5 Vyvážení velikostí podgrafů

Hlavní příčinou obtížnosti problému bisekce je požadavek na stejně velké podgrafy. V případě, že by toto nebylo požadováno, jsou k dispozici rychlejší metody řešení. Pokud se však u striktně definované bisekce v průběhu evolučního algoritmu vyskytne kandidát nevyhovující tomuto požadavku, je nutno provést některé z následujících opatření.

- Nevyvážená řešení penalizovat
- Opravné postupy – z velkých podgrafů přesunout některé vrcholy do menších
- Prevence – navrhnout operátor křížení tak, aby neprodukoval nevyvážená řešení

5.6 PROBE

Heuristika PROBE *(Population–Reinforced Optimization–Based Exploration)* využívá principu, při kterém se při vícebodovém křížení zachovává část informace společná oběma rodičům. Na potomkovi vytvořeném pomocí tohoto pravidla následně zkonstruuje zbytek genomu pomocí prohledávání (např. DiferentialGreedy) a výsledek doladí pomocí lokálního optimalizátoru (např. K–L). Podrobný popis je v [2].

Kapitola 6

Navržený algoritmus

V rámci této diplomové práce byl navržen a implementován hybridní evoluční algoritmus pro hledání bisekce grafu. Tato kapitola podrobně popisuje princip všech součástí algoritmu. Nejdůležitější aspekty implementace jsou popsány v kapitole 7.

Základem řešení je evoluční algoritmus. Jeho vstupem je graf a volitelně jeho iniciální rozdělení do dvou clusterů. Toto rozdělení je algoritmem vylepšováno pomocí evolučně vyšlechtěné sekvence vrcholů z jednoho clusteru, ke kterým se hladově hledají vhodné vrcholy k prohození z druhého clusteru. Hladové hledání je založeno na principu použitém v K–L heuristice. Ta však hledá hladově vrcholy z obou clusterů, zatímco navržený algoritmus vybírá vrcholy z prvního clusteru evolučně.

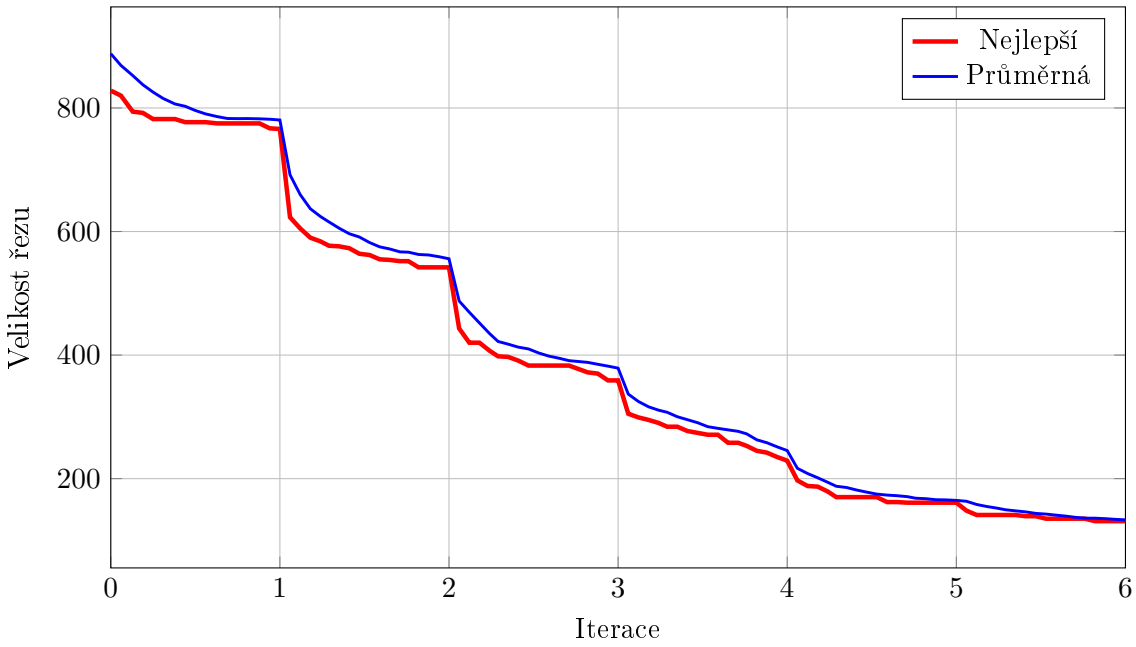
Iterativní evoluční algoritmus je rozšíření evolučního algoritmu o možnost jeho iterovaného spouštění. Nejlepší evolučně nalezené rozdělení po definovaném počtu generací se použije jako výchozí pro nový běh evolučního algoritmu. Pokud rozdělení dosažené na konci aktuální iterace bude horší než rozdělení na jejím začátku, použije se do další iterace to rozdělení, které bylo vstupem aktuální iterace. Tento postup se iterativně opakuje. Zajistí se při něm, že se na začátku každé iterace vygeneruje nová náhodná iniciální populace sekvencí akcí, tj. sekvencí vrcholů z jednoho podgrafu. To snižuje riziko uváznutí algoritmu v lokálním extrému. Ke snížení tohoto rizika přispívá i to, že další iterace bude (v případě zlepšení rozdělení v předchozí iteraci) optimalizovat jiné iniciální rozdělení než optimalizovala iterace předchozí.

Navržený algoritmus také obsahuje možnost použití klasické K–L heuristiky jako lokálního optimalizátoru. Je možné optimalizovat všechny jedince z generace nebo pouze nejlepšího z nich. Pokud tím dojde ke zlepšení *cutsize*, použije se pro další běh algoritmu takto optimalizované řešení namísto horšího evolučně nalezeného.

Hlavní cyklus navrženého algoritmu je popsán v kapitole 6.7. Všechny jeho části jsou podrobně popsány v této kapitole. Typický příklad průběhu iterativního algoritmu je znázorněn na obr. 6.1. Z obrázku je patrný vliv evoluce v každé iteraci. Na začátku iterace jsou zlepšení výraznější než k jejímu konci. Dále je vidět, že počátek nové iterace je vždy na úrovni nejlepšího výsledku té předchozí. Dalším pozorováním je skutečnost, že pozdější iterace již nedokáží nacházet tak velká zlepšení jako počáteční iterace.

Počáteční rozdělení na vstupu algoritmu je možné získat jedním z následujících způsobů:

- Zadat vlastní rozdělení



Obrázek 6.1: Průběh 6 iterací u grafu `u500.10`

- BFS rozdělení s počátkem ve vrcholu s nejvyšším stupněm (viz 3.4)
- Náhodné rozdělení
- Střídavé rozdělení vrcholů do clusterů

Volba iniciálního rozdělení má zásadní vliv na kvalitu výsledku algoritmu. Nelze však obecně stanovit, které z nich je pro konkrétní instanci problému optimální. Spolehlivým vodítkem není ani nižší hodnota *cutsize* vybraného iniciálního rozdělení. I z horšího rozdělení může algoritmus dojít k lepšímu výsledku.

V průběhu vývoje algoritmu se ukázalo, že nejlepších výsledků dosahuje při náhodném iniciálním rozdělení. Při použití BFS rozdělení často nastávalo uváznutí v lokálním extrému, protože takto deterministicky určené rozdělení směřovalo výpočet příliš úzkým směrem.

Běh algoritmu se ukončí při nalezení dostatečně dobrého řešení, jehož hodnota se nastavuje v počáteční konfiguraci. Pokud takového řešení není dosaženo, ukončí se běh po provedení definovaného počtu iterací.

6.1 Reprezentace

Jedinec je reprezentován permutací čísel, které odpovídají číslům vrcholů z jednoho podgrafu. Délka chromozomu je tedy $\lceil \frac{V}{2} \rceil$. Musí být zajištěno, že každý vrchol z daného podgrafu je použit právě jednou. Tyto podmínky musí být splněny při náhodné inicializaci nového jedince i pro jedince vzniklé křížením a mutací.

6.2 Fitness

Základ hodnoty fitness jedince je *cutsize* jím daného rozdělení, tedy počet hraničních hran daného rozdělení. Úkolem evolučního algoritmu je fitness minimalizovat. Fitness jedince se určí na základě sekvence vrcholů z jednoho podgrafu uložených v chromozomu. Ke každému z nich se hladově najde vrchol z druhého podgrafu s největším ziskem při prohození. Tato sekvence akcí se aplikuje na pracovní rozdělení a sleduje se při tom, jaký počet K provedených prohození dává nejlepší výsledek. Není nutné při každém prohození počítat *cutsize* kompletně znovu, stačí aktualizovat fitness pracovního rozdělení na základě zisku dosaženého provedením jednotlivých přesunů uzlů. Celý postup je analogický ke hledání nejvýhodnějšího páru k prohození tak, jak je zaveden v Kernighan–Lin heuristice (3.6). Výsledkem je celé číslo, které je možné dále zjemnit postupem popsaným v kapitole 6.2.2 a jemu odpovídající rozdělení, které lze ještě lokálně optimalizovat.

Při výpočtu fitness se prohazované vrcholy a z prvního clusteru se postupně berou z čísel obsažených v chromozomu. Ke každému z nich se najde ve druhém clusteru takový vrchol b, který s ním při prohození nejvíce vylepší rozdělení. K výpočtu zisku při prohození vrcholů a a b použijeme vzorec

$$g_{a,b} = D_a + D_b - 2w_{a,b}$$

kde $D_v = E_v - I_v$ je rozdíl mezi vnější a vnitřní hodnotou vah vrcholu a $w_{a,b}$ je váha hrany mezi oběma vrcholy. Pokud bude nalezeno více vrcholů s nejlepším ziskem, vybereme jeden z nich náhodně. Jednou vybraný vrchol už v dalším prohledávání neuvažujeme. Před hledáním dalších dvojic je nutno aktualizovat D hodnoty vrcholů sousedících s a a b.

Je třeba zajistit, aby každý vrchol byl prohozen pouze jednou. Dvojí prohození clusteru u daného vrcholu znamená, že vrchol zůstal ve svém původním clusteru. Obecně, pří vyšším počtu p prohození u jednoho vrcholu se při sudé hodnotě p vrchol proti původnímu stavu nepřesune.

Po průchodu celého chromozomu vybereme takový počet K prohození, při kterém je výsledná hodnota *cutsize* nejnižší. Pokud je tato hodnota stejná jako u počátečního rozdělení, budeme preferovat nově nalezeně řešení. Poté na počáteční rozdělení aplikujeme K prvních prohození vrcholů od začátku chromozomu s jejich nalezenými protějšky. Zbylá část chromozomu již nebude pro výpočet v této generaci použita. Chromozom však zachováváme celý, protože část která není nyní použita, se může ještě uplatnit v křížení.

Může dojít k situaci, kdy se pro algoritmus jako optimální řešení jeví buď neprovádět žádné prohození nebo naopak provést prohození všech vrcholů. Obojí znamená stagnaci, protože prohození všech vrcholů je jen redundantní zápis téhož původního rozdělení (v případě bisekce existují právě dva redundantní zápisy téhož rozdělení, které se liší pouze číslem použitým pro daný cluster). Toto chování se často projevuje v situaci, kdy evoluční algoritmus uvázne v lokálním extrému. Experimenty ukázaly, že se uváznutí dá čelit pomocí vynucení M počátečních prohození a povolením pouze maximálně prvních N prohození nebo zjemněním výpočtu fitness metodou popsanou v kapitole 6.2.2. Konkrétní kombinace těchto opatření a volba velikosti příslušných konstant je experimentální záležitostí. Empirické orientační hodnoty získané z řady pokusů na náhodném geometrickém grafu jsou v řádech jednotek pro M a řádově $\frac{2}{3}$ z délky chromozomu pro N.

6.2.1 Vynucené a zakázané prohození

Vynucení prvních M prohození je rozšíření metody výpočtu fitness tak, aby se vždy provedlo minimálně M prohození vrcholů mezi dvěma podgrafy. Postup hledání vrcholů z druhého podgrafu a prohazování se nemění. Tato prohození se musí uskutečnit bez ohledu na to, že díky nim dojde ke zhoršení *cutsize*. Cílem této techniky je vynutit vždy alespoň nějakou změnu, aby nedocházelo ke stagnaci tím, že algoritmus vyhodnotí jako nejvýhodnější prázdnou sekvenci prohození, tedy nedělat žádné prohození.

Analogicky je možné definovat, kolik N prohození bude maximálně možné provádět. Délka chromozomu zůstává nezměněna, ale při výpočtu fitness se zkouší prohazování pouze do této délky. Zachování plné délky chromozomu je nutné proto, aby v něm byly obsaženy všechny možné vrcholy. Tím bude zajištěno, že všechny vrcholy budou mít možnost pomocí křížení a mutace být vybrány pro výměnu. Tato technika zabraňuje nežádoucímu jevu, při kterém algoritmus vyhodnotí jako nejvýhodnější provést všechna prohození (což znamená pouze redundantní přepsání téhož rozdělení).

Kombinace obou těchto technik může rovněž mít pozitivní vliv na diverzitu populace. Při experimentech s nulovými hodnotami M a maximálně velkým N docházelo v populaci k tomu, že se celá skládala ze stejných jedinců. To se dařilo úspěšně eliminovat vhodnou kombinací hodnot M a N.

6.2.2 Zjemnění fitness

Hodnota fitness počítaná na základě počtu odstraněných hran je omezena pouze na celá čísla. Nedá se podle ní určit, nakolik jsou si dva jedinci se stejnou hodnotou podobní. Přitom je žádoucí, aby populace nestagnovala na jednom řešení, které se již nevyplatí pomocí prohazování zlepšovat. Zjemnění fitness se zavádí proto, aby byla dosažena větší preference těch jedinců, kteří jsou vzdálenější lokálnímu extrému. To znamená, že ze dvou jedinců se sekvencí generující různá řešení o stejné kvalitě preferujeme jedince s delší sekvencí. Délka sekvence prohazovaných vrcholů tedy slouží jako základ vedlejšího kritéria pro výpočet fitness.

Úprava hodnoty se provádí přičtením hodnoty nepřímo úměrné K tak, aby delší sekvence znamenala lepší (nižší) fitness. Přičítat lze pouze hodnotu v rozsahu $[0, 1)$, aby nebyl změněn celočíselný základ hodnoty. Tím by došlo k chybné interpretaci výsledné hodnoty jako hodnoty s jiným počtem odstraněných hran. Výpočet probíhá podle vzorce

$$fitness' = fitness + \frac{1}{3 + K}$$

kde hodnotu konstanty je třeba volit tak, aby vzhledem ke způsobu indexování hodnoty K nedošlo k dělení nulou nebo překročení přípustného intervalu velikosti přičítané hodnoty. Provádění zjemnění fitness je možné v konfiguraci globálně pro všechny jedince povolit nebo zakázat.

K	**fitness'**
0	10,333
2	10,200
15	10,056

Tabulka 6.1: Příklad zjemněných hodnot původní fitness 10 s konstantou 3

Z tabulky je patrné, že nejlepší hodnotu fitness bude mít jedinec s nejdelší sekvencí prohození vrcholů.

6.2.3 Lokální optimalizace K–L heuristikou

V navrženém algoritmu je jako lokální optimalizátor použita K–L heuristika. Spouštění algoritmu je možné následujícími způsoby:

- Bez lokální optimalizace
- K–L optimalizace nejlepšího jedince z generace
- K–L optimalizace všech jedinců z generace

Proces lokální optimalizace je součástí metody výpočtu fitness. Pokud při něm dojde ke zlepšení *cutsize*, nastaví se danému jedinci jako fitness tato vylepšená hodnota. Zároveň se uloží jí odpovídající rozdělení pro případný pozdější výpis.

Ukázalo se, že lokální optimalizace umožňuje algoritmu dosahovat lepší výsledky. Nevýhodou je však vyšší náročnost na výpočetní výkon (viz 9.4). Je proto umožněno spouštění K–L s omezujícími parametry tak, aby nebyla doba výpočtu neúměrně dlouhá.

Parametr `maxSwapSequenceLength` omezuje maximální možný počet prohození, které smí K–L provést ve svém hlavním cyklu (řádky 1–10 v algoritmu 2). Kandidátské vrcholy na prohození jsou v implementaci seřazeny sestupně podle svého čísla D (rozdíl mezi hranami do druhého podgrafu a hranami do vlastního podgrafu), což zabraňuje vynechání nejvýhodnějších vrcholů. Sestavování sekvence navíc automaticky skončí v případě, že už je díky setřídění podle D jisté, že nebude nalezen vrchol s lepším ziskem než měl vrchol na začátku seznamu kandidátů.

Parametr `maxSearchBestGainPartnerDepth` omezuje hloubku prohledávání při hledání nejvýhodnějšího vrcholu k prohození. Může se totiž stát, že v seznamu kandidátských vrcholů k prohození je příliš mnoho těch, které mají stejné číslo D. To může být způsobeno buď rozmístěním hran ve vstupním grafu, nebo v situaci, kdy výrazně lepší vrcholy již byly spárovány a zbývá velké množství vrcholů se stejným D. Hodnota parametru omezuje maximální počet testovaných vrcholů při hledání protějšku k prohození pro jeden vrchol.

Volba vhodných hodnot parametrů je opět záležitostí experimentů. Je potřeba zvolit kompromisní hodnoty, které naleznou dostatečné zlepšení v rozumně dlouhém čase. Vliv omezení délky sekvence je pro vybrané grafy znázorněn v .

6.2.4 Pseudokód

Následující pseudokód podrobně popisuje výslednou metodu výpočtu fitness za použití výše popsaných technik. Při výpočtu je k dispozici chromozom daného jedince, iniciální rozdělení iterace i vstupní graf. konfigurační parametry `forceNFirstSwaps` a `allowNFirstSwaps` odpovídají M a N z kapitoly 6.2.1 o počtu vynucených a povolených prohození.

Algoritmus 3 updateFitness

```
1:  K ← −1                                          ▷ V K bude uložen optimální počet prohození
2:  bisection ← iniciální rozdělení iterace
3:  for i = 0 to forceNFirstSwaps do                ▷ První vynucená prohození
4:      Najdi nejvýhodnější vrchol V_gainBest k prohozeni s chromozom[i]
5:      Aplikuj prohození na bisection
6:      V_gainBest již v této generaci neuvažuj
7:      K ← i
8:  for i = forceNFirstSwaps to allowNFirstSwaps do   ▷ Další prohození do max. počtu
9:      Najdi nejvýhodnější vrchol V_gainBest k prohozeni s chromozom[i]
10:     if Prohození zlepší fitness then
11:         Aplikuj prohození na bisection
12:         V_gainBest již v této generaci neuvažuj
13:         K ← i
14: if Optimalizace pomocí K–L then                 ▷ Dle konfigurace algoritmu
15:     if K–L dokáže zlepšit fitness then
16:         bisection ← rozdělení zlepšené pomocí K–L
17: if Zjemňování fitness then                      ▷ Dle konfigurace algoritmu
18:     Fitness += 1/(C+K)                          ▷ Zjemnění fitness viz 6.2.2
```

6.3 Křížení

Pro operátor křížení je důležité, aby pomáhal správným způsobem prohledávat stavový prostor problému. Je také žádoucí, aby produkoval validní jedince (viz popis v 5.3). V navrženém algoritmu jsou použity dva podobné operátory křížení vycházející z těchto předpokladů a z vlastností chromozomu jedince.

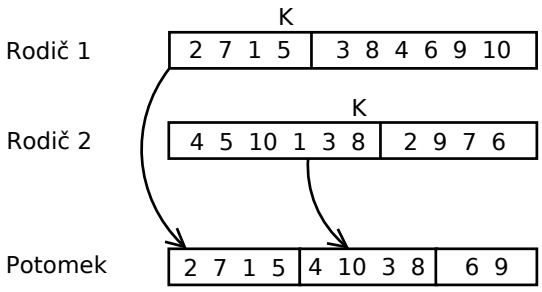
Chromozom jedince reprezentuje pole čísel obsahující sekvenci vrcholů z jednoho clusteru. Při výpočtu fitness je nalezeno číslo K určující optimální počet prohození vrcholu od začátku chromozomu. Zbylá část chromozomu nebyla v daném jedinci využita. Operátory křížení jsou proto navrženy tak, aby do potomka přednostně přenášely z rodičů ty části jejich chromozomů, které leží před pozicí K.

Operátor křížení typu *copy* tvoří ze dvou rodičů jednoho potomka. Zkopíruje do něho nejprve počátek chromozomu prvního rodiče až do pozice K. Poté připojí počátek chromozomu druhého rodiče opět až do pozice K. Zbylé místo zaplní čísly dosud nepoužitých vrcholů v pořadí podle chromozomu rodičů (viz obr. 6.2).

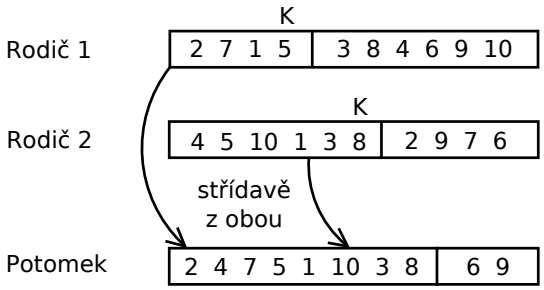
Operátor křížení typu *merge* tvoří ze dvou rodičů jednoho potomka. Do něho postupně střídavě umísťuje vrcholy z chromozomu prvního a druhého rodiče. Takto pokračuje, dokud u obou rodičů nedojde na pozici K. V případě, že jeden z rodičů P_1 má číslo K vyšší ($K_1 > K_2$), provede se K_2 přesunů střídavě a poté se přenesou zbylé hodnoty až do K_1 z rodiče P_1. Zbylé místo zaplní čísly dosud nepoužitých vrcholů v pořadí podle chromozomu rodičů (viz obr. 6.3).

Při všech přenosech oba operátory kontrolují, nebylo–li již právě přenášené číslo vrcholu v potomkovi použito. Pokud ano, přenesení se neprovede a pokračuje se dalším číslem podle

právě probíhající operace. To zajistí tvorbu validních potomků, tj. jedinců, jejichž chromozom obsahuje každý vrchol z prvního clusteru právě jednou.



Obrázek 6.2: Křížení typu *copy*.

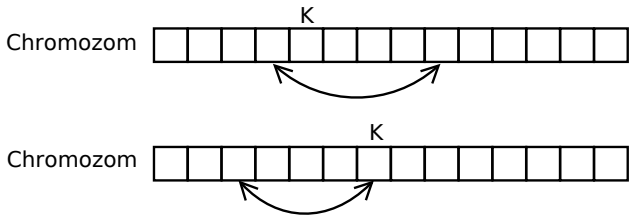


Obrázek 6.3: Křížení typu *merge*.

6.4 Mutace

Mutace je realizována jako prohození dvou prvků v chromozomu na náhodně vybraných pozicích. Ze stejných důvodů jako u křížení zde platí pravidlo, že alespoň jedna tato pozice musí být nejvýše rovna K. To zajistí, že bude mutací změněna ta část chromozomu, která byla použita při prohazování vrcholů (viz obr. 6.4).

V případě volby obou pozic v části chromozomu za K by klesala možnost projevení se mutace ve výpočtu. Pokud by byly zvoleny až v oblasti za maximálním počtem povolených prohození (viz 6.2.1), nemohla by mutace mít v aktuální generaci žádný efekt (v některé příští generaci by se ovšem provedená změna mohla pomocí křížení přenést do aktivní oblasti chromozomu).



Obrázek 6.4: Příklady možných mutací.

6.5 Elitismus

Elitismus je mechanizmus, který přenáší nejlepšího jedince z aktuální generace do generace následující. To umožňuje zachování nejlepšího dosaženého řešení a může zefektivnit běh celého algoritmu.

Při přenosu se nejlepšímu jedinci zachovává hodnota fitness vypočtená ve staré generaci a jí odpovídající rozdělení vrcholů. V nové generaci se tedy jeho fitness již nepřepočítává. Pokud v nové generaci nebude nalezen ještě lepší jedinec, bude opět přenesen o generaci dále. Důsledkem toho je fakt, že při zapnutém elitismu je posloupnost hodnot fitness nejlepších jedinců generace neklesající.

6.6 Selekce

Při výběru rodičů pro křížení je použita metoda deterministického turnaje s nastavitelným počtem n kol. Pro uplatnění turnaje je třeba volit $n > 1$, jednokolový turnaj by odpovídal náhodnému výběru. Volba probíhá v n kolech, ve kterých se fitness náhodně zvoleného jedince porovnává s nejlepší dosud nalezenou hodnotou. Vybrán je jedinec s celkově nejlepší fitness. Determinismus spočívá v pravděpodobnosti $p = 1$, se kterou je v kole vybrán lepší z jedinců.

6.7 Hlavní cyklus algoritmu

Hlavním cyklem algoritmu je opakované iterativní spouštění definovaného počtu generací. Počet generací se nastavuje v parametru `generationsMax`. Maximální počet iterací je nastaven v parametru `iterationsMax`. Jako iniciální rozdělení nové iterace se bere nejlepší řešení dosažené v předchozí iteraci. Běh algoritmu se ukončí po provedení definovaného počtu iterací nebo při nalezení dostatečně dobrého řešení, jehož hodnota se nastavuje v počáteční konfiguraci.

Algoritmus 4 Hlavní cyklus

Input: Graf $G(V, E)$, Iniciální rozdělení grafu G
Output: Optimalizované rozdělení grafu G

```
1: bisection ← iniciální rozdělení
2: for iteration = 1 to iterationsMax do
3:     bisection ← doIteration(bisection);
4: return bisection;
```

6.7.1 Iterace

Iterace spočívá v evolučním zlepšování vstupního rozdělení. Na jejím začátku se vygeneruje zcela nová náhodná populace. To pomáhá vyvést proces prohledávání z lokálního optima dosaženého v předchozí iteraci. K tomu navíc přispívá i skutečnost, že se optimalizuje vždy nové rozdělení (pokud předchozí iterace nedospěla ke zhoršení). Pokud bylo nalezeno řešení se stejným *cutsize* ale jiným rozdělením, jako výsledné bude použito toto nově nalezené.

Algoritmus 5 doIteration

Input: Vstupní rozdělení
Output: Optimalizované rozdělení

```
1: procedure doIteration(bisection)
2:     Vytvoř novou náhodnou populaci
3:     Aktualizuj všem jedincům v populaci fitness          ▷ Viz algoritmus 3
4:     for generation ≤ generationsMax do
5:         doGeneration();
6:     if nejlepší nalezené řešení > vstupní rozdělení then   ▷ Porovnání podle cutsize
7:         return vstupní rozdělení                ▷ Výsledek nesmí být horší než vstup
8:     return nejlepší nalezené řešení;
```

6.7.2 Generace

Generace je standardní procedurou evolučního algoritmu. Dochází při ní k výběru, křížení a mutaci kandidátů. Po výpočtu fitness se podle nastavení algoritmu ještě provádí lokální optimalizace pomocí K–L heuristiky. Všechny operace v průběhu generace jsou realizovány tak, jak je popsáno v této kapitole podle nastavení parametrů algoritmu.

Algoritmus 6 doGeneration

Input: Vstupní rozdělení, populace jedinců

```
1:  procedure doGeneration(bisection, population)
2:      Založ novou prázdnou populaci newPopulation
3:      if Elitismus then                                   ▷ Dle konfigurace algoritmu
4:          newPopulation ← Nejlepší jedinec z population
5:      repeat
6:          Turnajově vyber rodiče R1,R2
7:          if random ≤ P_repl then                         ▷ Náhodně s pravděpodobností P_repl%
8:              newPopulation ← Potomek D1 vzniklý copy křížením R1,R2
9:              newPopulation ← Potomek D2 vzniklý merge křížením R1,R2
10:         else
11:             newPopulation ← R1
12:             newPopulation ← R2
13:         if random ≤ P_mutation then                     ▷ Náhodně s pravděpodobností P_mutation%
14:             Mutace D1
15:             Mutace D2
16:     until nová generace není naplněna
17:     population ← newPopulation
18:     Aktualizuj všem jedincům v population fitness       ▷ Viz algoritmus 3
```

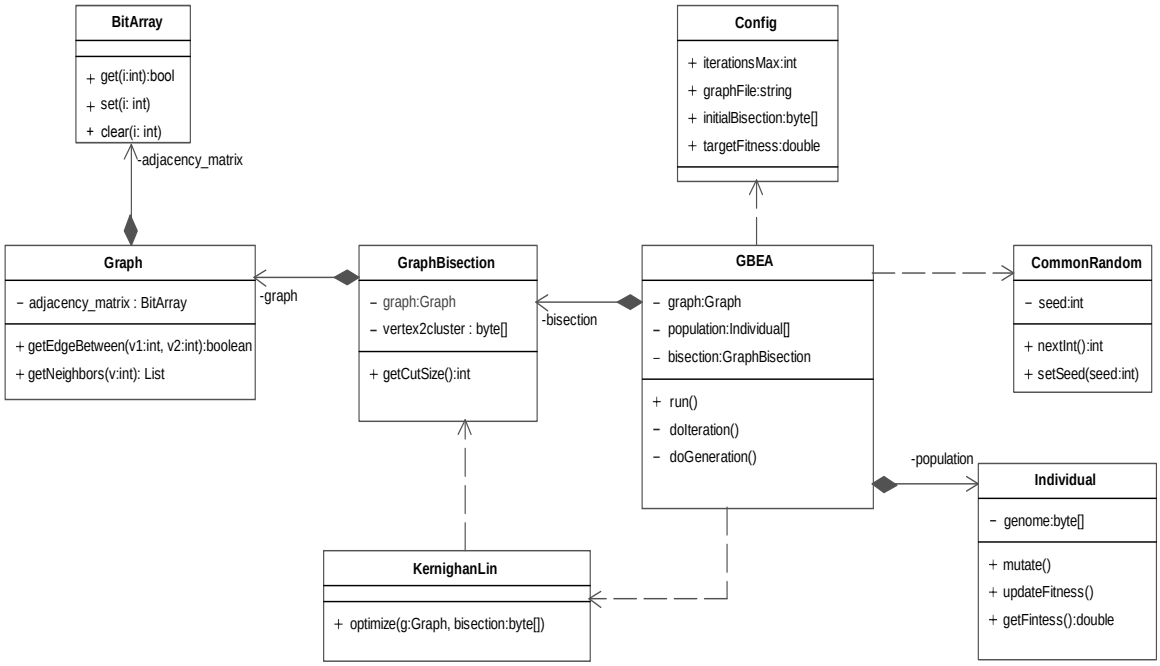
Kapitola 7

Implementace

Následující kapitola popisuje implementaci navrženého algoritmu. Zabývá se především klíčovými součástmi algoritmu v návaznosti na jeho obecný popis v předchozí kapitole. Jsou zde popsány nejdůležitější optimalizační techniky použité pro zvýšení výkonu výsledné aplikace. Jako platforma zvolena Java 7 ve vývojovém prostředí NetBeans.

7.1 Hlavní třídy implementace

Diagram hlavních tříd implementace je na obr. 7.1. Kvůli přehlednosti jsou zakresleny pouze nejpodstatnější třídy a jejich součásti.



Obrázek 7.1: Diagram hlavních tříd navrženého algoritmu.

7.1.1 Třída Graph

Reprezentace grafu v paměti. Obsahuje informace o počtu a vzájemném propojení vrcholů a hran. Nejdůležitější funkce jsou testování, zda je mezi zadanými vrcholy hrana a seznam sousedů daného vrcholu.

7.1.2 Třída GraphBisection

Na základě grafu a rozdělení jeho vrcholů do podgrafů vytvoří datové struktury umožňující získat velikost řezu, seznamy vrcholů obou podgrafů a údaje o počtech vnitřních a vnějších hran všech vrcholů. V implementaci je použita pouze interně,

7.1.3 Třída GBEA

Implementuje navržený hybridní evoluční algoritmus pro řešení PMGB. Obsahuje populaci jedinců, ve které provádí selekci, křížení a mutaci. Provádí hlavní iterační a evoluční cykly. Vypisuje a ukládá průběžné i celkové informace o běhu výpočtu. Veškeré nastavení parametrů a vstupních dat při spouštění získává z instance třídy `Config` (7.1.7).

7.1.4 Třída Individual

Obsahuje chromozom jedince a údaj o jeho fitness. Implementuje důležitou metodu `updateFitness` sloužící k výpočtu fitness. Provádí mutaci svého chromozomu.

7.1.5 Třída Crossover

Obsahuje implementace podporovaných metod křížení jedinců.

7.1.6 Třída KernighanLin

Implementuje K–L optimalizaci zadaného rozdělení grafu. Výsledkem jejího běhu je optimalizované rozdělení a hodnota zisku dosaženého optimalizací.

7.1.7 Třída Config

Obsahuje veškeré konfigurační parametry jednoduše strukturované do sekcí. Načítá a ukládá konfigurační soubory ve formátu XML. Neobsahuje žádné konkrétní hodnoty parametrů, všechny musí být specifikovány v konfiguračním souboru.

7.2 Pomocné třídy a funkce

7.2.1 Reprodukovatelnost experimentu

Pseudonáhodná čísla jsou v průběhu celého výpočtu generována výhradně pomocí třídy `CommonRandom`. Třída umožňuje inicializaci náhodného generátoru zadaným číslem *seed*. Pro

každou jeho hodnotu vrací stejnou sekvenci čísel. To umožňuje při stejné počáteční konfiguraci algoritmu spuštěného se stejnou hodnotou *seed* přesně reprodukovat předchozí výsledky. Pokud se v takto nastaveném novém běhu zadá vyšší počet iterací, až do původního počtu iterací budou výsledky totožné.

7.2.2 Prevence chyb

Při vývoji byl kladen důraz na neustálou automatizovanou kontrolu hodnot a konzistence dat v průběhu výpočtu. Jednorázové a výpočetně nenáročné kontroly jsou trvale zapnuty. Jsou to například kontroly rozsahu vstupních parametrů, počtů vrcholů a hran při načítání grafu, kontroly vyváženosti pracovních rozdělení grafu.

Provádění výpočetně náročných kontrol (jako např. kontrolní přepočet velikosti řezu) je nastavitelné interním konfiguračním parametrem (7.3). Jejich zapnutí není z hlediska výkonu doporučeno, jsou určeny především pro ladění kódu.

7.2.3 Pomocné utility

Implementace obsahuje řadu menších samostatně použitelných utilit. Spouštějí se z příkazové řádky a seznam požadovaných argumentů vypisují při spuštění bez parametrů.

`getcutsize.GetCutSize` ze zadaného grafu a rozdělení spočítá velikost řezu. Minimální možná implementace nezávislá na kódu hlavní aplikace.

`graph.graphinfo` spočítá základní statistiky zadaného grafu a rozložení počtu vrcholů podle jejich stupňů.

`graphconvert.vertex2edges` zkonvertuje soubor s grafem uloženým ve formátu *vertex* do formátu *edge* (viz 7.7).

`kernighanlin.kl` provede K–L optimalizaci zadaného grafu a rozdělení.

`kernighanlin.kltest` provede K–L optimalizaci zadaného grafu a rozdělení postupně pro všechny možné hodnoty maximální délky sekvence prohození v K–L.

7.3 Konfigurace

Konfigurace pro spuštění algoritmu je celá uložena v souboru formátu XML. Účelem tohoto způsobu uložení je uchování a pozdější snadné znovupoužití nastavení experimentu. Konfigurační soubor je přijímán při spouštění z příkazové řádky i uživatelského rozhraní (7.5). Editaci hodnot je možno provádět pohodlně po načtení do uživatelského rozhraní (7.4) nebo ručně přímo v souboru.

Konfigurace umožňuje nastavení následujících parametrů. Obsahuje i části zde neuvedené, určené pro interní ladění a vývoj. Jejich hodnoty je doporučeno neměnit.

`sourceGraphFile`	Soubor se vstupním grafem
`targetFitness`	Cílová hodnota fitness

`initialBisectionType`	Výběr z RANDOM, ALTERNATING, BFS, CUSTOM
`customBisection`	Soubor s rozdělením (při CUSTOM inic. rozdělení)
`iterationsMax`	Počet iterací
`experimentName`	Název experimentu
`runCount`	Počet běhů s různými hodnotami *seed*
`seed`	*Seed* náhodného generátoru (7.2.1)

`evolutionalAlgorithmInternal`	Sekce nastavení evolučního algoritmu
`populationSize`	Velikost populace
`generationsMax`	Počet generací
`Prepl`	Pravděpodobnost křížení v %
`Pmut`	Pravděpodobnost mutace v %
`elitism`	Elitismus
`tournamentSize`	Počet kol turnajového výběru
`forceNFirstSwaps`	Počet vynucených prohození (6.2.1)
`allowNFirstSwaps`	Omezení počtu prohození (6.2.1)
`softenFitness`	Zjemnění fitness (6.2.2)

`KlConfig`	Sekce nastavení K–L algoritmu
`maxSwapSequenceLength`	Omezení počtu prohození v K–L (6.2.3)
`maxSearchBestGainPartnerDepth`	Omezení hloubky hledání v K–L (6.2.3)
`optimizeWholeGeneration`	Lokální optimalizace celé generace
`optimizeBestFromGeneration`	Lokální opt. nejlepšího jedince z generace

Při volbě vlastního iniciálního rozdělení *CUSTOM* je nutné zadat soubor s tímto rozdělením. Musí obsahovat jedinou řádku s posloupností číslic 0 nebo 1 podle toho, v jakém podgrafu leží vrchol na dané pozici. Má tedy přesně takový počet číslic, kolik má vstupní graf vrcholů.

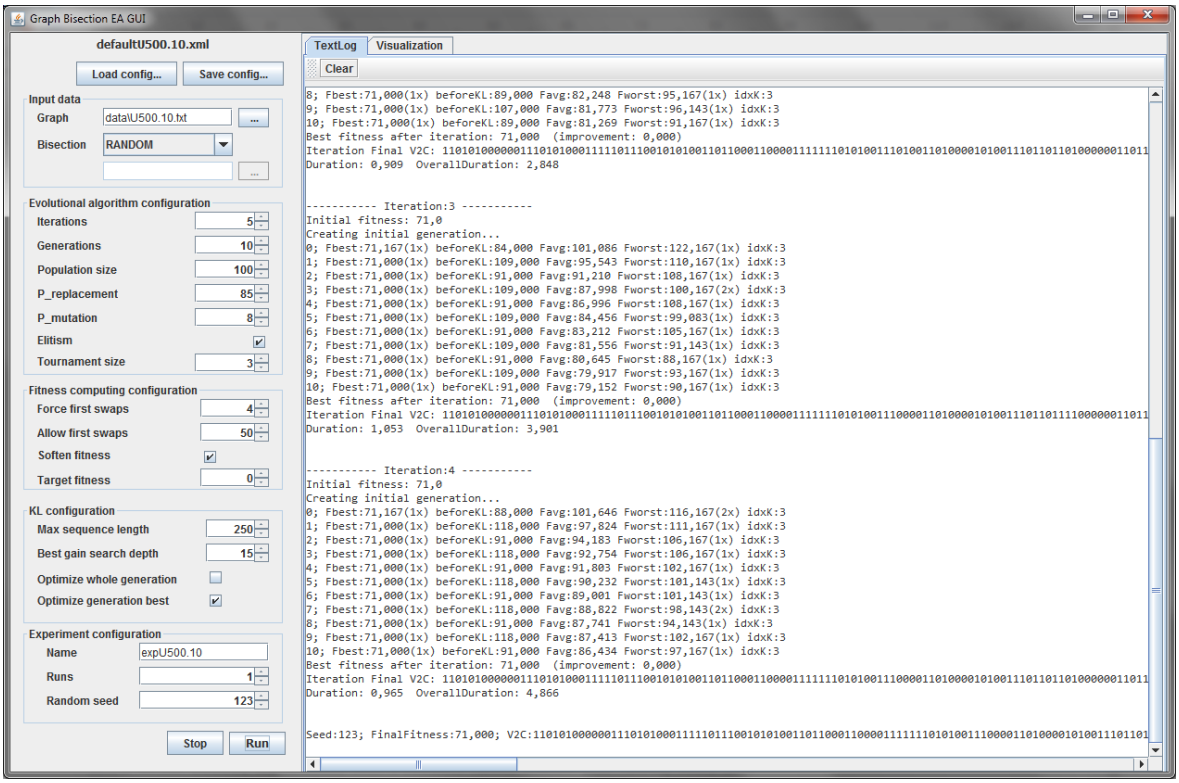
Počet běhů s různými hodnotami *seed* znamená, že bude uskutečněn daný počet běhů s tím, že každý z nich bude mít vždy o jednotku vyšší *seed* než ten předchozí. To umožní zautomatizovat sérii pokusů nutnou pro statistické vyhodnocení výsledků.

Název experimentu slouží pro pojmenování souborů se záznamy o průběhu, jak bude popsáno v kapitole 7.6.

7.4 Uživatelské rozhraní

K navrženému algoritmu bylo naimplementováno i grafické uživatelské rozhraní (obr. 7.2). Jeho hlavním účelem je zjednodušení konfigurace algoritmu a možnost okamžitého spuštění s nastavenými parametry. Po startu načítá konfigurační soubor s názvem `default.xml`. Aktuálně načtená konfigurace je zobrazena v levém horním rohu. Pomocí tlačítek *load config* a *save config* lze načíst jiný konfigurační soubor nebo uložit aktuální nastavení. Přepsání aktuálního souboru konfigurace se provádí pouze na pokyn uživatele.

V levé části okna je možné nastavit všechny konfigurační parametry. Zobrazené názvy ovládacích prvků jsou jednoduše přiřaditelné k parametrům definovaným v kapitole 7.3.

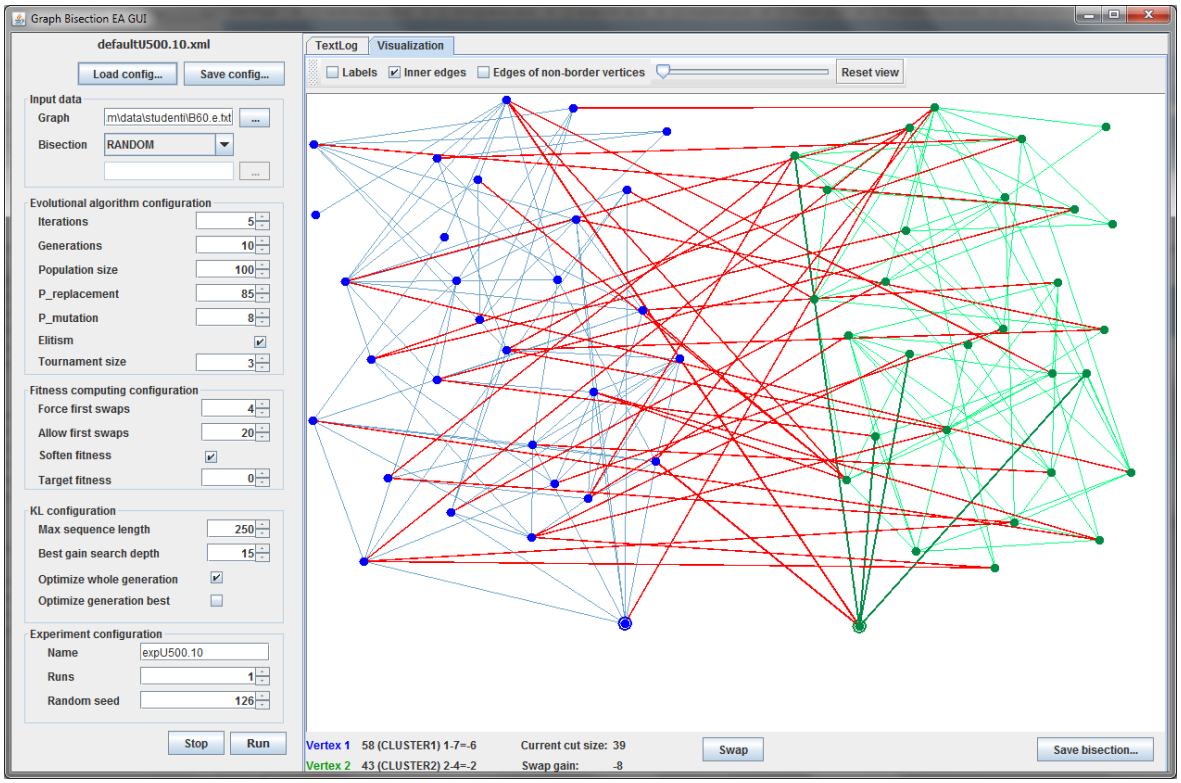


Obrázek 7.2: Uživatelské rozhraní s výpisem průběhu výpočtu.

Spouštění se provádí pomocí tlačítka *run*. Předčasné ukončení lze provést pomocí tlačítka *stop*. Při jeho použití se program ještě pokusí dopočítat aktuálně prováděnou část výpočtu, aby bylo možné zobrazit konzistentní výsledky u poslední iterace. Pokud by tento výpočet byl příliš dlouhý, lze opakovaným stisknutím *stop* vynutit okamžité ukončení. V takovém případě ale nelze zobrazit konzistentní výsledky u poslední iterace.

Aplikace po dokončení výpočtu zobrazuje vizualizaci výsledného rozdělení (v panelu *visualization*, viz obr. 7.3). Modře resp. zeleně jsou vyznačeny vrcholy obou podgrafů. Světlemodře resp. světlezeleně jsou zobrazeny vnitřní hrany, červeně potom hrany mezi vedoucí podgrafy (jejich počet je roven *cutsize*). Je možné vypnout zobrazování vnitřních hran nebo zachovat jen ty hrany, které vedou k vrcholům, jejichž alespoň jedna hrana je hraniční.

Účelem vizualizace je možnost zběžné kontroly rozdělení a případně nalezení dvojice vrcholů k možnému prohození, které mají oba velké číslo D (tedy mnoho červených hran a málo vnitřních hran své barvy). Tyto vrcholy je možné označit a provést ruční prohození (tlačítko *swap*). Takto lze doladit rozdělení a uložit jej tlačítkem *save bisection* do souboru pro pozdější použití jako vstupního rozdělení nového výpočtu.



Obrázek 7.3: Uživatelské rozhraní s vizualizací výsledného rozdělení.

7.5 Průběh výpočtu

Spouštění z uživatelského rozhraní popsané v minulé kapitole se hodí spíše na experimenty s nastavením a kratší jednorázové výpočty. Pokud je třeba provést řadu experimentů s různými vstupy a nastaveními za sebou, lze využít možnost spustit program v dávce jako konzolovou aplikaci. Ta je implementována ve třídě `GBEA.ea` a jejím jediným a povinným argumentem je cesta ke konfiguračnímu XML souboru.

V průběhu výpočtu se zobrazují základní statistiky jednotlivých generací. Po dokončení iterace se zobrazí její výsledek, vypíše se dosažené rozdělení a údaj o době výpočtu. Na konci výpočtu se ještě vypíše souhrn nejlepších výsledků pro všechny běhy s různými *seedy*. Údaje vypisované na konci každé generace mají následující význam (pro názornost použit příklad s konkrétními hodnotami):

```
2; Fbest:29,019(3x) beforeKL:892,000 Favg:75,900 Fworst:164,022(1x) idxK:49
```

Na začátku řádku je číslo generace indexované od nuly. `Fbest` je hodnota fitness nejlepšího jedince v generaci (v závorce počet výskytů takových jedinců v populaci). `BeforeKL` je fitness nejlepšího jedince před provedením lokální optimalizace. `Favg` je průměrná hodnota fitness populace. `Fworst` je hodnota fitness nejhoršího jedince v populaci (v závorce počet výskytů takových jedinců v populaci). `IdxK` je od nuly indexovaný optimální počet prohození použitý v nejlepším jedinci (před lokální optimalizací).

7.6 Ukládání výsledků a konfigurace

Záznamy s průběhem výpočtu a výsledky, které se zobrazují v konzoli nebo v uživatelském rozhraní, se automaticky ukládají do souborů. Pro každý běh s jinou hodnotou *seed* je vytvořen zvláštní soubor. Název souboru je složen z data spuštění, názvu experimentu a hodnoty *seed* ve formátu `datum_názevExperimentu_seed.txt`. Na konci výpočtu je také uložen seznam celkových nejlepších výsledků za jednotlivé běhy s názvem ve formátu `datum_názevExperimentu_Results.txt`. Při spuštění z uživatelského rozhraní se ještě automaticky uloží nastavená konfigurace do souboru `datum_názevExperimentu_config.xml`. Všechny soubory se ukládají do adresáře, ze kterého byl program spuštěn. Záznamy experimentů stejného data, názvu a *seedu* se automaticky přepisují.

7.7 Reprezentace grafu

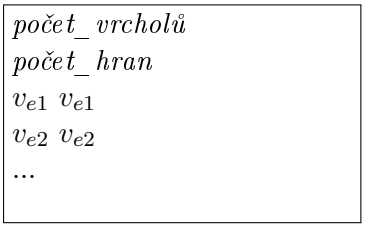
Reprezentace grafu je plně podřízena rychlosti běhu výsledné aplikace i za cenu jisté redundance v uložení grafu. Graf je primárně uložen v úplné matici sousednosti. To umožňuje zjistit v konstantním čase existenci hrany mezi dvěma vrcholy, což je velmi důležité při hledání dvojic vrcholů k prohození. Při provedení prohození je třeba aktualizovat D hodnoty i sousedním vrcholům, proto je nezbytná rychlá implementace získání seznamu sousedů k zadanému vrcholu. Při inicializaci grafu se z tohoto důvodu vytvoří seznamy sousedů pro všechny vrcholy a v případě dotazu se tak nemusejí pokaždé konstruovat z matice sousednosti. Z důvodu kontroly duplicitních hran a možnosti rychlé iterace přes všechny hrany jsou všechny hrany uloženy v kolekci.

Z důvodu úspory paměti při počítaní velkých instancí grafů je pro matici sousednosti použito bitové pole realizované vymaskováním požadovaného bitu na příslušném indexu v poli čísel. Z důvodu nezanedbatelné režie (např. častá kontrola velikosti pole) nebyla použita standardní Java třída `BitSet`, ale vlastní minimalistická implementace.

Vstupní graf je načítán z textového souboru. Podporovány jsou dva nejpoužívanější formáty uložení. Uložení jako seznam sousedů všech vrcholů (budeme jej nazývat *Vertex format*) má tvar

počet_ vrcholů počet_ hran
n_1 n_2 n_3 ...
n_1 n_2 n_3 ...
...

kde jsou počínaje 2. řádkem zapsány sousední vrcholy pro daný vrcholů (n_i je i–tý soused daného vrcholu). Řádků je přesně tolik, kolik je vrcholů a jsou řazeny vzestupně podle čísla vrcholu. Tak tedy lze určit číslo vrcholu podle čísla řádku. V případě izolovaného vrcholu se zapíše prázdný řádek. Uložení seznamu všech hran *(Edge format)* má schéma



kde jsou počínaje 3. řádkem zapsány hrany. Dvojice v_{ei} v_{ei} jsou koncové vrcholy těchto hran. Prázdné vrcholy nemusí být speciálně vypsány. Podle počtu vrcholu načteného z prvního řádku se vytvoří správný počet vrcholů a ty, které se nepropojí zapsanými hranami, zůstanou izolované.

Při použití obou formátů jako vstupu pro různé programy je třeba se ujistit, jak má přesně daná implementace definovaný vstupní formát a nakolik mu odpovídá vstupní soubor. Mohou existovat varianty které mají jiný formát záhlaví, jiné číslování vrcholů apod.

V implementaci navrženého algoritmu je vzhledem k formátu dostupných testovacích dat preferován výše popsaný *Vertex format* s indexováním vrcholů od 1 a očekávanou příponou `.txt`. Je možné použít i soubor *Edge format* s indexováním vrcholů od 1 a bez řádku s počtem hran v záhlaví, který musí mít příponu `.e.txt`. Soubory nesmí obsahovat žádné nadbytečné znaky ani nadbytečné prázdné řádky.

7.8 Rychlý výpočet velikosti řezu

V implementaci je kladen důraz na to, aby se velikost *cutsize* daného rozdělení nepočítala zbytečně často celá znovu pomocí procházení všech hran v grafu. Ve většině případů to není nutné, protože výslednou hodnotu optimalizovaného rozdělení lze získat odečtením zisků provedených prohození od hodnoty rozdělení původního.

Výhodou počítání *cutsize* pouze z grafu a rozdělení je velmi krátká a jednoduchá implementace a tím i spolehlivost výsledku. To je užitečné při ladění jako kontrola správnosti rychlého výpočtu.

7.9 Efektivita implementace K–L algoritmus

Efektivita implementace K–L algoritmu závisí na efektivitě implementace grafu (popis v 7.7). Při provádění operací K–L algoritmu je klíčová volba datové struktury pro uchovávání dosud nespárovaných kandidátských vrcholů setříděných podle D hodnoty (viz 6.2.3). Navíc je nutné, aby zvolená struktura umožnila rychlé zjištění, zda obsahuje vrchol daného čísla a rychlý přístup k prvkům přes jejich index podle setřídění. Z těchto důvodů byla pro uložení použita kolekce `ArrayList` doplněna o hashovací tabulku `HashMap`. Samostatné standardní Java kolekce testované při vývoji nesplňovaly všechna požadovaná kritéria.

Kapitola 8

Testovací data

Následující kapitola popisuje typy grafů nejčastěji používané jako testovací data pro programy řešící PMGB. Jsou zde uvedeny odkazy na stránky, kde jsou tyto grafy k dispozici. Rovněž jsou zde odkazy na výsledky různých autorů při experimentech nad těmito grafy

8.1 Náhodné grafy

Náhodné grafy *(random graphs)* používají ve svém názvu zápis formátu $G(n,p)$ resp. $G(n,m)$. Oba tyto typy se liší způsobem konstrukce grafu.

Graf $G(n,p)$ je neorientovaný, obsahuje n vrcholů a p značí pravděpodobnost přítomnosti hrany v grafu. Vybíráme z množiny všech možných hran v grafu, pravděpodobnost výběru každé hrany je nezávislá na ostatních.

Graf $G(n,m)$ je neorientovaný, obsahuje n vrcholů a právě m hran. Tyto hrany jsou vybírány náhodně a rovnoměrně z množiny všech možných hran v grafu.

Počet všech možných hran úplného obecného neorientovaného grafu $G(V,E)$ je

$$|E| = \binom{|V|}{2} = \frac{n(n-1)}{2}$$

kde $n = |V|$ je počet vrcholů grafu.

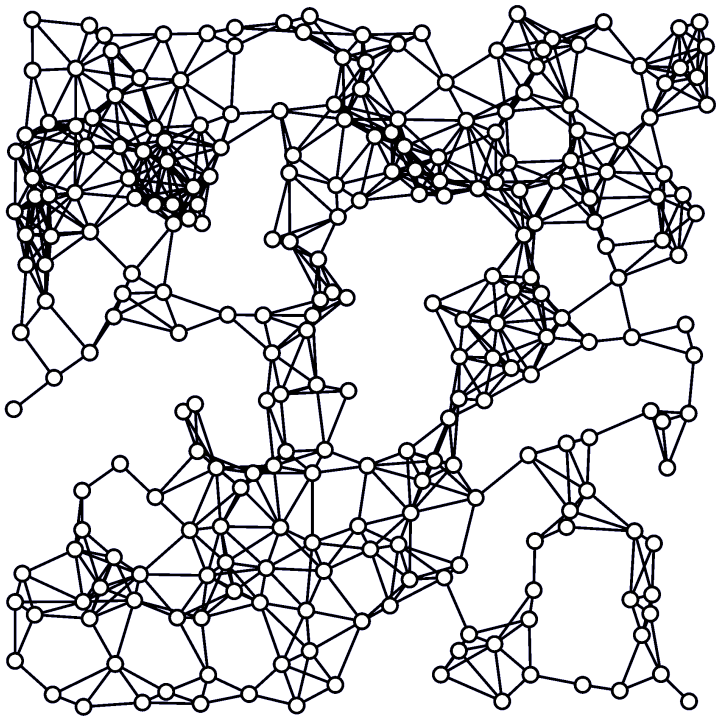
8.2 Náhodné geometrické grafy

Náhodné geometrické grafy *(random geometric graphs)* používají ve svém názvu značení $U(n,r)$. Vyznačují se podobností s grafy reálných VLSI obvodů nebo počítačových sítí, protože mají tendenci tvořit lokální shluky (viz obr. 8.1). Tvoří se následujícím postupem:

1. Náhodné geometrické rozmístění n vrcholů do ohraničené oblasti, např. $[0,1)^2$.

2. Propojení vrcholů u, v hranou se provede tehdy a jenom tehdy, pokud jejich vzdálenost d není větší než definovaná prahová hodnota r, tedy $d(u,v) \leq r$.

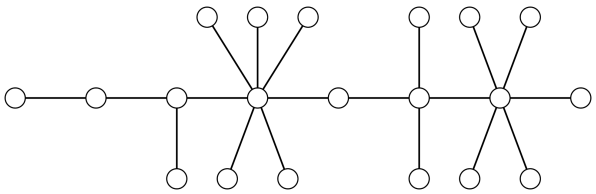
Počet hran nebo komponent výsledného grafu není pevně daný, dá se pouze statisticky odhadnout na základě parametrů grafu.



Obrázek 8.1: Náhodný geometrický graf *(zdroj: en.wikipedia.org)*.

8.3 Housenkový graf

Housenkový graf (z angl. *caterpillar graph*) je strom, který obsahuje jednu centrální linii a jehož všechny ostatní vrcholy jsou s ní přímo propojeny (viz obr. 8.2).



Obrázek 8.2: Caterpillar graf *(zdroj: en.wikipedia.org)*.

8.4 Reálné grafy

Reálné grafy mají strukturu odvozenou z parametrů nějakého reálného objektu nebo výsledků reálných experimentů. Například grafy `add20` resp. `add32`, použité pro experimenty

(viz kap. 9.1), reprezentují strukturu elektronického obvodu, konkrétně 20–bitové resp. 32–bitové sčítačky.

8.5 Často používané testovací grafy

Pro posouzení efektivnosti vlastního algoritmu je ideální použít takové zdrojové grafy, pro které již existují dokumentované experimenty a výsledky s existujícími algoritmy. Zároveň musí být dostupné ke stažení v přesně stejné podobě.

Jako zdroje dat pro tuto práci byly použity následující stránky obsahující kolekce nejčastěji používaných testovacích grafů:

- Walshaw's Graph Partitioning Archive [3]
- Seoul National University, Optimization Lab Dept. Benchmark data [19]
- Paderborn University, AG Monien Research, Graph partitioning [15]

Nejkomplexnější informace ohledně reálných grafů poskytuje [3]. Obsahuje odkazy na soubory s grafy, u některých grafů i odkazy na podrobnější informace a vizualizace. Dále obsahuje tabulky nejlepších dosažených hodnot rozdělení grafů. Stránka [15] obsahuje kromě reálných grafů také kolekce náhodných a náhodných geometrických grafů. Stránka [19] obsahuje nebo odkazuje náhodné grafy s definovanými strukturami.

V použité literatuře jsou konkrétní výsledky uváděny např. v [7],[18], [2], [5].

Kapitola 9

Experimenty

Kapitola popisuje grafy vybrané k experimentům a prezentuje výsledky dosažené na těchto datech implementací navrženého algoritmu. Obsahuje také výsledky vlastní implementace K–L a srovnání s jinou implementací K–L.

Experimenty byly prováděny na počítači s čtyřjádrovým procesorem Intel Core i5–2400 3.1GHz, 4GB RAM, OS Windows 7. Rozvržení úloh bylo takové, aby žádné jádro v jeden okamžik nepočítalo více než jeden graf.

9.1 Testovací grafy

V tabulce 9.1 jsou uvedeny grafy, které byly zvoleny jako testovací sada pro ověření funkčnosti navrženého algoritmu. Ve výběru jsou zastoupené reálné grafy (podrobnosti o těchto konkrétních grafech jsou v [3] a [20]), náhodné a náhodné geometrické grafy a graf s jednoduchou mřížkovou strukturou. Všechny z nich mají zdokumentované výsledky s jinými algoritmy (viz 8.5).

Grafy jsou v jednotlivých kategoriích seřazeny vzestupně podle počtu vrcholů. Nejvyšší počet vrcholů má graf `cti` (16840), nejvyšší počet hran má graf `bcsstk33` (291583).

9.2 Základní experimenty

Pro každý graf bylo spuštěno 10 běhů s různou hodnotou *seed*. Každý běh se skládal z 5 iterací po 15 generacích. Přesná konfigurace je znázorněna v tab. 9.2. Počáteční náhodná řešení pro jednotlivé běhy měla parametry dle tab. B.1. Kompletní výpis hodnot je uveden v příloze B.1.

Při experimentech byly získány hodnoty *cutsize* uvedené v souhrnné tab. 9.4. Sloupec „nejlepší známé“ obsahuje hodnotu *cutsize* nejlepšího známého řešení publikovaného ve zdrojích shrnutých v kap. 8.5. Kompletní výpis hodnot pro všechny běhy je uveden v příloze C.1.

Uváděný čas iterace je průměrný čas výpočtu jedné iterace při hodnotě *seed* 123. Podrobný rozpis v časů všech iterací je v tab. 9.5. V něm je u všech grafů patrné, že dřívější

Graf	**|V|**	**|E|**	**Kategorie**
add20	2395	7462	Reálné grafy
data	2851	15093	
3elt	4720	13722	
uk	4824	6837	
add32	4960	9462	
bcsstk33	8738	291583	
whitaker3	9800	28989	
crack	10240	30380	
wing_nodal	10937	75488	
cti	16840	48232	
u500.05	500	1282	Náhodné geom. grafy
u500.10	500	2355	
u1000.10	1000	4696	
u1000.40	1000	18015	
g500.02	500	2355	Náhodné grafy
g500.04	500	5120	
g1000.01	1000	5064	
g1000.02	1000	10107	
grid64x64	4096	8064	Mřížka 64×64

Tabulka 9.1: Zvolené testovací grafy

Parametr	**Hodnota**
initialBisectionType	RANDOM
iterationsMax	5
runCount	10
seed	123
populationSize	50
generationsMax	15
Prepl	85
Pmut	8
elitism	true
tournamentSize	3
forceNFirstSwaps	3
allowNFirstSwaps	$\frac{\|V\|}{5}$
softenFitness	true
maxSwapSequenceLength	$\frac{\|V\|}{10}$
maxSearchBestGainPartnerDepth	15
optimizeWholeGeneration	true

Tabulka 9.2: Základní nastavení experimentů

Graf	Min.	Průměr	Medián
add20	3635	3744,2	3743,5
data	7444	7543,7	7527,5
3elt	6777	6852,9	6838
uk	3385	3425,2	3434,5
add32	4677	4736,4	4735
bcsstk33	145305	145905,5	145967,5
whitaker3	14296	14468,1	14449
crack	15074	15203,5	15187
wing_nodal	37433	37681,1	37715
cti	23990	24154	24155
u500.05	604	647,3	648
u500.10	1156	1185,7	1175,5
u1000.10	2282	2344,8	2351,5
u1000.40	8973	9046,9	9045,5
g500.02	1155	1181,4	1180
g500.04	2550	2589,6	2585,5
g1000.01	2482	2535,3	2533,5
g1000.02	4994	5083,7	5082
grid64x64	3974	4025	4010

Tabulka 9.3: Iniciální rozdělení

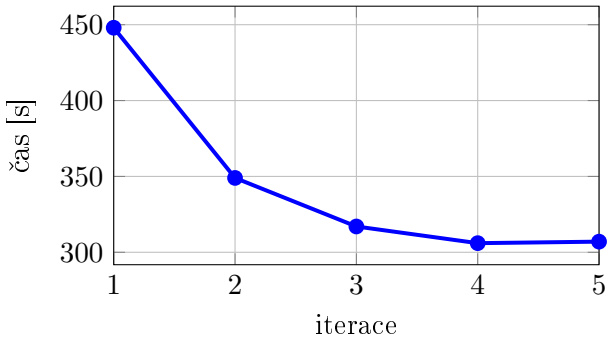
Graf	Nejlepší známé	Navržený EA			Nejlepší nalezeno v		Čas iterace	Nejepší EA vs. nejlepší známé	
		Nejlepší	Průměr	Medián	Iter.	Gener.	[s]	%	Rozdíl
add20	596	639	797,7	780	5	1	86,78	107,21	+43
data	189	189	189,9	190	1	3	124,27	100,00	0
3elt	90	90	90	90	1	2	358,92	100,00	0
uk	19	482	512,7	509	5	1	345,71	2536,84	+463
add32	11	261	314,4	324	3	14	363,11	2372,73	+250
bcsstk33	10171	10171	10171	10171	1	4	1139,73	100,00	0
whitaker3	127	127	127	127	1	3	1533,27	100,00	0
crack	184	184	184	184	1	11	1661,77	100,00	0
wing_nodal	1707	1708	1711,8	1712	2	1	1746,08	100,06	+1
cti	334	334	334	334	1	2	4246,80	100,00	0
u500.05	2	7	12,1	11	3	1	4,28	350,00	+5
u500.10	26	26	38	38,5	2	1	4,43	100,00	0
u1000.10	39	52	68,8	65,5	1	10	16,02	133,33	+13
u1000.40	737	737	737	737	1	1	18,72	100,00	0
g500.02	626	626	629,2	629	2	1	5,51	100,00	0
g500.04	1744	1744	1746,6	1745,5	2	1	6,85	100,00	0
g1000.01	1362	1376	1384,6	1385	5	1	17,86	101,03	+14
g1000.02	3382	3384	3396,4	3394,5	5	1	22,39	100,06	+2
grid64x64	64	64	64	64	1	1	283,57	100,00	0

Tabulka 9.4: Výsledky základních experimentů

Graf	Iterace					Souhrn			
	1	2	3	4	5	Min.	Max.	Průměr	Medián
add20	121,1	83,2	76,8	77,7	75,1	75,1	121,1	86,8	77,7
data	195,3	106,9	106,1	106,4	106,6	106,1	195,3	124,3	106,6
3elt	630,5	282,4	282,5	307,9	291,3	282,4	630,5	358,9	291,3
uk	448,3	349,4	317,1	306,7	307,1	306,7	448,3	345,7	317,1
add32	459,5	335,5	319,8	327,4	373,4	319,8	459,5	363,1	335,5
bcsstk33	1638,5	1033,7	1009,0	1009,0	1008,4	1008,4	1638,5	1139,7	1009,0
whitaker3	2877,1	1196,1	1198,5	1197,8	1196,8	1196,1	2877,1	1533,3	1197,8
crack	3214,5	1289,6	1292,4	1258,0	1254,4	1254,4	3214,5	1661,8	1289,6
wing_nodal	2758,9	1518,5	1477,2	1478,3	1497,5	1477,2	2758,9	1746,1	1497,5
cti	7505,0	3433,0	3427,0	3436,0	3433,0	3427,0	7505,0	4246,8	3433,0
u500.05	5,7	3,8	4,0	3,8	4,2	3,8	5,7	4,3	4,0
u500.10	6,1	4,4	3,9	3,9	3,9	3,9	6,1	4,4	3,9
u1000.10	22,3	15,1	15,0	13,9	13,9	13,9	22,3	16,0	15,0
u1000.40	25,6	17,2	16,7	17,0	17,0	16,7	25,6	18,7	17,0
g500.02	8,4	5,0	4,5	4,8	4,9	4,5	8,4	5,5	4,9
g500.04	11,0	6,4	5,8	5,6	5,5	5,5	11,0	6,8	5,8
g1000.01	27,4	15,5	15,0	16,0	15,4	15,0	27,4	17,9	15,5
g1000.02	37,0	18,7	19,4	18,6	18,3	18,3	37,0	22,4	18,7
grid64x64	559,6	217,5	215,4	211,7	213,7	211,7	559,6	283,6	215,4

Tabulka 9.5: Časy výpočtů iterací [s]

iterace trvají delší dobu, viz například obr. 9.1 pro graf `uk`. Je to způsobeno tím, že v počátečních iteracích algoritmus nachází více možností k dosažení většího zisku než v pozdějších. Je tedy prováděno více přesunů, což vyžaduje delší čas výpočtu.



Obrázek 9.1: Časy výpočtu iterací grafu `uk`

Ze srovnání výsledků navrženého algoritmu s nejlepší známou hodnotou je patrné, že v 11 případech z 19 byla tato nejlepší známá hodnota navrženým algoritmem dosažena. U 3 dalších grafů byla nalezena hodnota méně než o 2% horší než nejlepší známá. U 5 grafů byla nalezena horší hodnota, z toho u 2 výrazně horší.

9.3 Rozšířené experimenty

Pro grafy, u kterých navržený algoritmus v základní konfiguraci nenalezl dostatečně kvalitní řešení, byla provedena další série experimentů s rozšířeným nastavením algoritmu. Oproti základnímu nastavení (tab. 9.2) byly změněny parametry uvedené v tab. 9.6. Byl zdvojnásoben počet jedinců v populaci a počet prováděných iterací. Dále byla nastavena větší povolená délka sekvence prohození u výpočtu fitness i u K–L optimalizace.

Parametr	**Hodnota**
iterationsMax	10
populationSize	100
generationsMax	15
forceNFirstSwaps	3
allowNFirstSwaps	$\frac{4}{5}\|V\|$
maxSwapSequenceLength	$\frac{\|V\|}{5}$

Tabulka 9.6: Rozšířené nastavení experimentů

Souhrn výsledků je uveden v tab. 9.7. Rozšířený běh nalezl u 3 grafů řešení o méně než 3% horší než nejlepší známé. U grafu `u500.05` je velký relativní rozdíl (300%) oproti nejlepší známé hodnotě, ale absolutní rozdíl jsou pouze 2 hrany. Výsledky grafů `uk` a `add32` jsou pořád výrazně horší. Kompletní výsledky jsou v příloze v tab. D.1.

Graf	**Nejlepší známé**	**Rozšířený EA**			**Nejepší vs. nejlepší známé**	
		Nejlepší	Průměr	Medián	%	Rozdíl
add20	596	609	683,9	690,5	102,18	+13
uk	19	430	456,6	458	2263,16	+411
add32	11	121	137,1	136	1100,00	+110
u500.05	2	6	8,8	9,5	300,00	+4
u1000.10	39	40	52,6	52	102,56	+1
g1000.01	1362	1373	1379,3	1380,5	100,81	+11
g1000.02	3382	3387	3392,5	3392	100,15	+5

Tabulka 9.7: Výsledky rozšířených experimentů

Na grafy `u500.05` a `add32` bylo ještě aplikováno experimentálně nalezené nastavení, které zlepšilo nejlepší hodnoty nalezené navrženým algoritmem. Výsledky a hlavní změny v nastavení oproti tab. 9.6 jsou uvedeny v tab. 9.8. U grafu `uk` se dalšími změnami parametrů již nepodařilo hodnoty zlepšit, možné zdůvodnění je uvedeno ve zhodnocení výsledků v kap. 10.

Graf	Nejlepší	Hlavní změna nastavení
add32	72	allowNFirstSwaps 1000 maxSwapSequenceLength 2480
u500.05	3	allowNFirstSwaps 248

Tabulka 9.8: Výsledky se speciálním nastavením

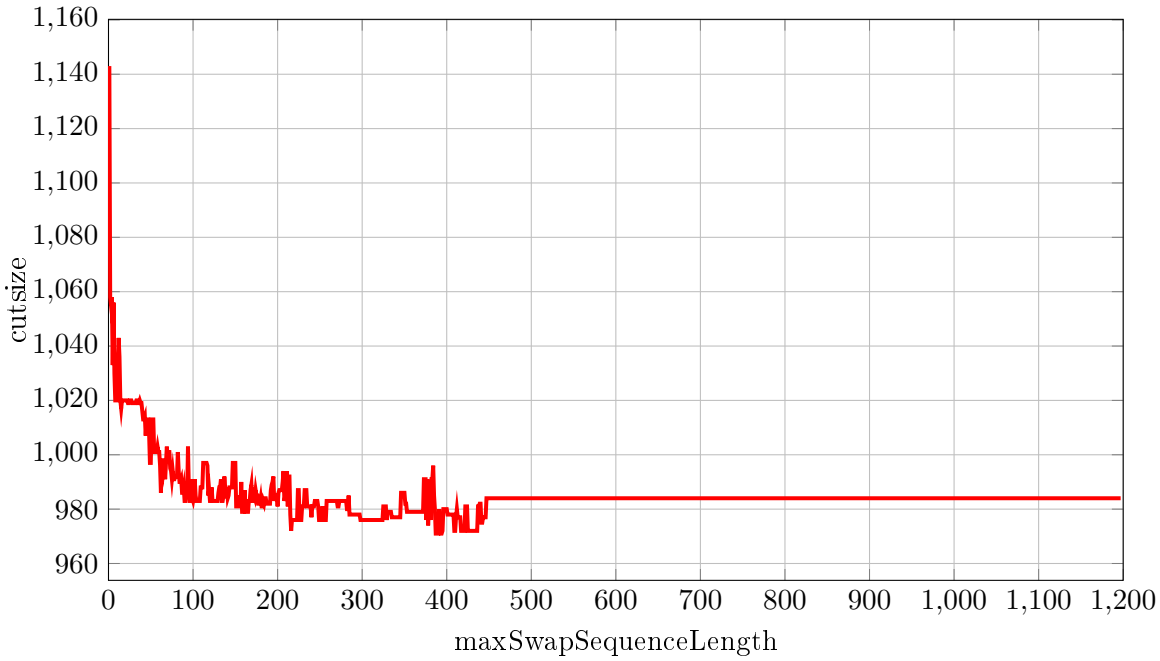
9.4 Vlastní implementace K–L

Jedním z cílů práce je srovnání výsledků navrženého algoritmu s čistou K–L optimalizací. Iniciální rozdělení vygenerovaná při experimentech s 10 hodnotami *seed* pro každý graf byla proto optimalizována K–L algoritmem. Jednalo se o tutéž implementaci, která byla použita k lokální optimalizaci v navrženém algoritmu. Dosažené výsledky jsou shrnuty v 9.9 , kompletní výsledky jsou v příloze E.1. Délka sekvence prohození byla maximální možná pro daný graf, hloubka prohledávání byla omezena parametrem `maxSearchBestGainPartnerDepth` s hodnotou 15.

Graf	Nejlepší	Průměr	Medián
add20	656	846,5	866
data	202	242	238
3elt	90	130,4	135
uk	36	61,6	61,5
add32	284	354,1	332
bcsstk33	10172	11454,8	11909
whitaker3	128	135,4	132,5
crack	186	253,1	213
wing_nodal	1804	2116,4	1976
cti	366	521,3	476
u500.05	27	35,7	31,5
u500.10	70	103,3	106,5
u1000.10	97	158,3	156,5
u1000.40	737	813,7	777
g500.02	640	658	662,5
g500.04	1767	1788,1	1792,5
g1000.01	1410	1441,2	1440,5
g1000.02	3430	3477	3475
grid64x64	64	72	66,5

Tabulka 9.9: Výsledky dosažené vlastní implementací K–L

V rámci experimentů s vlastní implementací K–L byly ještě pomocí utility `kltest` (viz 7.2.3) provedeny K–L optimalizace náhodného rozdělení postupně pro všechny možné hodnoty maximální délky sekvence prohození. Na obr. 9.2 je znázorněna výsledná velikost řezu v závislosti na hodnotě parametru `maxSwapSequenceLength`. Patrný je globální trend klesání



Obrázek 9.2: Velikost řezu grafu `add20` v závislosti na max. délce sekvence v K–L

cutsize, ovšem nelze obecně tvrdit, že větší hodnota `maxSwapSequenceLength` zajistí lepší výsledek. Jev je důsledkem toho, že při větším počtu kandidátských vrcholů k prohození se stejným ziskem nelze určit vrchol, který ve výsledku povede k lepšímu celkovému řešení. Z obr. 9.2 je dále vidět, že pro tento graf a rozdělení je K–L schopno dosahovat zlepšení jen do délky sekvence cca 450.

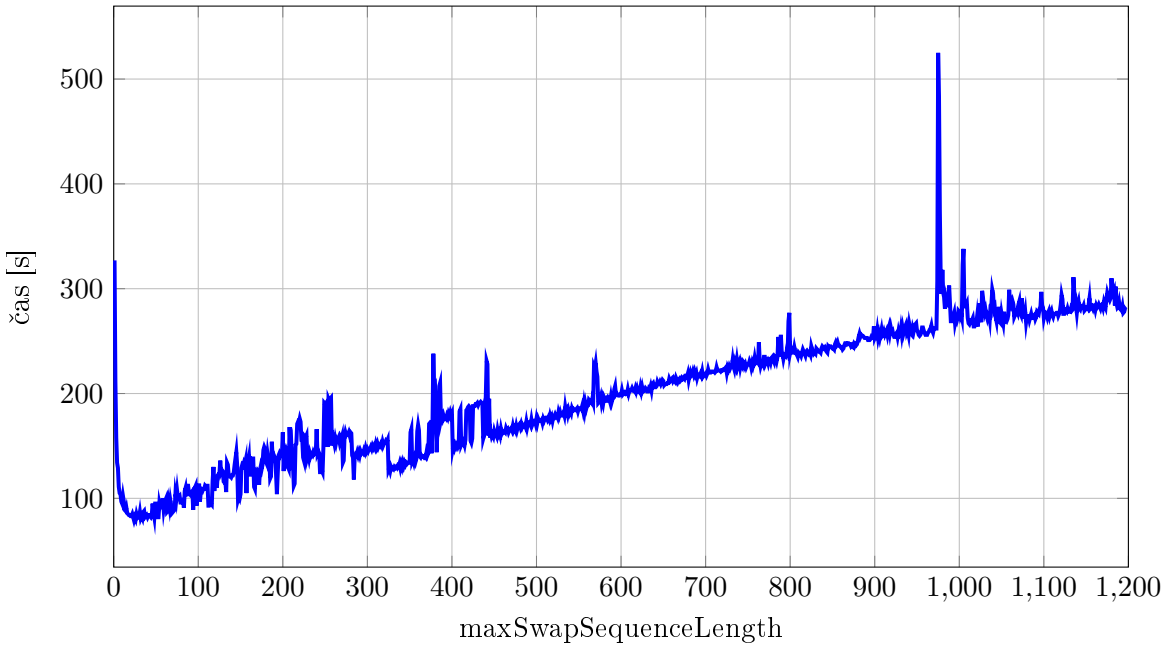
Na obr. 9.3 je časová závislost délky výpočtu na hodnotě `maxSwapSequenceLength`. Je vidět přibližně lineární trend závislosti. Občasné špičky mohou být způsobeny tím, že byl vybrán jeden nebo více vrcholů k prohození s velkým počtem sousedních vrcholů. Pro ty je třeba při prohazování aktualizovat D hodnoty, což způsobí delší čas výpočtu.

U dalšího testovaného grafu `uk` je na grafu závislosti velikosti řezu na parametru `maxSwapSequenceLength` na obr. obr. 9.4 vidět podobný trend jako u grafu `add20`. V tomto případě ale ještě došlo ke zlepšení při nejvyšších cca 100 hodnotách parametru `maxSwapSequenceLength`.

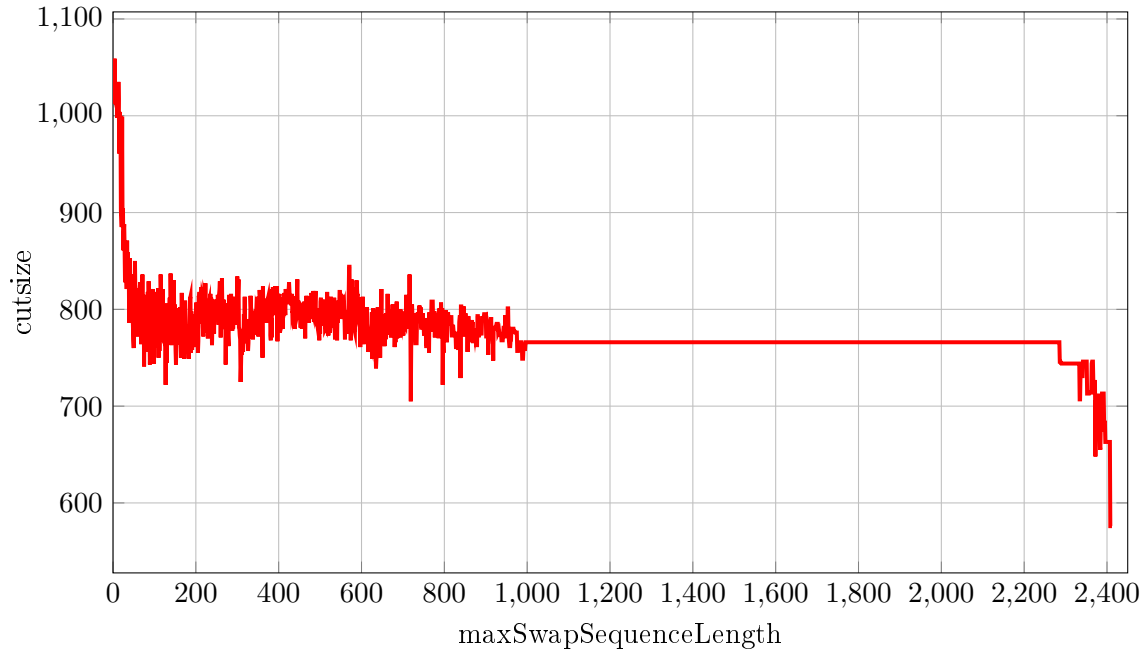
9.5 Srovnání s jinou implementací

Pro ověření výsledků produkovaných vlastní implementací K–L byly provedeny experimenty se stejnými grafy a iniciálními rozděleními pomocí jiné implementace K–L.

Jako referenční byla vybrána implementace dostupná na portále Google Project hosting [6] (FerggoK–L). Jedná se o aplikaci napsanou v jazyce C++. Vstupní graf přijímá ve formátu *edge* (viz 7.7). Výpočet neobsahuje náhodnou složku, pro stejný vstup má vždy stejný výsledek. Podle analýzy kódu pracuje s plnou délkou sekvence prohození a plnou hloubkou



Obrázek 9.3: Doba výpočtu grafu `add20` v závislosti na max. délce sekvence v K–L



Obrázek 9.4: Velikost řezu grafu `uk` v závislosti na max. délce sekvence v K–L

Graf	Nejlepší	Průměr	Medián
add20	635	791,6	794
data	189	216,6	212,5
3elt	135	166,9	163,5
uk	393	536,4	538
add32	353	428,9	437
bcsstk33	10172	11296,7	11796,5
whitaker3	130	182,8	164,5
crack	185	280,2	235
wing_nodal	1745	2039,8	1965,5
cti	407	626,7	588
u500.05	20	33,6	33
u500.10	31	89,7	87
u1000.10	113	196,1	190,5
u1000.40	737	848,2	838
g500.02	649	656,6	657
g500.04	1777	1791,6	1787
g1000.01	1428	1443,7	1441
g1000.02	3449	3487,4	3485,5
grid64x64	64	68,2	64

Tabulka 9.10: Výsledky dosažené implementací FerggoK–L

prohledávání (nelze ověřit z důvodu neexistující podrobné dokumentace). Jediná změna v originální implementaci spočívala v odstranění náhodného generování iniciálního rozdělení, které bylo nahrazeno načítáním rozdělení ze souboru. Vstupní grafy byly zkonvertovány do požadovaného formátu utilitou `vertex2edge` (viz 7.2.3) s tím, že byla ověřena shoda velikosti řezu pro počítaná rozdělení u obou formátů grafů.

Hodnoty získané implementací FerggoK–L jsou shrnuty v tab. 9.10. Kompletní výpis dosažených hodnot je v příloze v tab. F.1.

Kapitola 10

Zhodnocení výsledků

Nejdůležitější hodnoty získané při experimentech jsou uvedeny v souhrnné tabulce 10.1. První dvě trojice sloupců uvádějí souhrn výsledků dosažených navrženým algoritmem při základním a rozšířeném nastavení. Sloupec „nejlepší EA“ obsahuje nejlepší výsledek dosažený pomocí navrženého algoritmu ze všech nastavení včetně speciálních. Poslední dvě trojice obsahují souhrn výsledků dosažených se stejnými počátečními rozděleními pomocí vlastní a FerggoK–L implementace Kernighan–Lin algoritmu.

Graf	**Základní EA**			**Rozšířený EA**			**Nej. EA**	**Vlastní K–L**			**FerggoK–L**		
	Nej.	Prům.	Med.	Nej.	Prům.	Med.		Nej.	Prům.	Med.	Nej.	Prům.	Med.
add20	639	797,7	780	609	683,9	690,5	609	656	846,5	866	635	791,6	794
data	189	189,9	190				189	202	242	238	189	216,6	212,5
3elt	90	90	90				90	90	130,4	135	135	166,9	163,5
uk	482	512,7	509	430	456,6	458	430	36	61,6	61,5	393	536,4	538
add32	261	314,4	324	121	137,1	136	72	284	354,1	332	353	428,9	437
bcsstk33	10171	10171	10171				10171	10172	11454,8	11909	10172	11296,7	11796,5
whitaker3	127	127	127				127	128	135,4	132,5	130	182,8	164,5
crack	184	184	184				184	186	253,1	213	185	280,2	235
wing_nodal	1708	1711,8	1712				1708	1804	2116,4	1976	1745	2039,8	1965,5
cti	334	334	334				334	366	521,3	476	407	626,7	588
u500.05	7	12,1	11	6	8,8	9,5	3	27	35,7	31,5	20	33,6	33
u500.10	26	38	38,5				26	70	103,3	106,5	31	89,7	87
u1000.10	52	68,8	65,5	40	52,6	52	40	97	158,3	156,5	113	196,1	190,5
u1000.40	737	737	737				737	737	813,7	777	737	848,2	838
g500.02	626	629,2	629				626	640	658	662,5	649	656,6	657
g500.04	1744	1746,6	1745,5				1744	1767	1788,1	1792,5	1777	1791,6	1787
g1000.01	1376	1384,6	1385	1373	1379,3	1380,5	1373	1410	1441,2	1440,5	1428	1443,7	1441
g1000.02	3384	3396,4	3394,5	3387	3392,5	3392	3384	3430	3477	3475	3449	3487,4	3485,5
grid64x64	64	64	64				64	64	72	66,5	64	68,2	64

Tabulka 10.1: Celkové zhodnocení výsledků experimentů

Předpoklad při porovnání hodnot dosažených oběma K–L implementacemi byl ten, že dosažené hodnoty budou řádově stejné, nikoliv však nutně shodné. Výsledky tento předpoklad potvrzují, výjimku tvoří pouze grafy `uk` a `u500.10`. Ze srovnání nejlepších a průměrných hodnot a mediánu vyplývá, že vlastní implementace K–L dává v nadpoloviční většině případů lepší výsledky než FerggoK–L. Tento fakt je pozitivní z hlediska budoucího porovnání, nakolik navržený evoluční algoritmus zlepšuje klasický K–L, protože srovnání bude provedeno s implementací dávající lepší výsledky.

Rozdílné výsledky jsou způsobeny implementačními rozdíly ve způsobu uložení a procházení seznamů vrcholů k prohození. Pokud pro daný vrchol existuje jediný kandidát s nejlepším ziskem, musí být v obou implementacích vybrán právě tento. U více vrcholů se stejným ziskem může být díky implementačním rozdílům vybrán v obou případech jiný kandidát. Rozdíl v této volbě se může projevit v celkovém výsledku. Pořadí výběru vrcholů s nejlepším ziskem bylo u několika grafů a rozdělení ověřeno podle rozšířených výpisů průběhu výpočtu obou implementací K–L.

Hlavním cílem práce bylo experimentální ověření, zda navržený algoritmus překonává čistou K–L heuristiku. Z tab. 10.1 je patrné, že již při základním nastavení má navržený algoritmus lepší hodnoty průměru a mediánu ve všech případech kromě grafu `uk`. Nejlepší nalezená hodnota byla ve třech případech shodná (`3elt`, `u1000.40` a `grid64x64`) a pouze u grafu `uk` byl výsledek navrženého algoritmu horší.

Možné vysvětlení špatného výsledku u grafu `uk` spočívá ve struktuře grafu. Graf obsahuje vrcholy pouze stupňů 1–3, jak je patrné z tab. 10.2. Výrazně nejvíce je vrcholů stupně 3, u nich je přitom největší teoretická šance na velké D hodnoty. Při výběru vrcholů k prohození je proto pravděpodobně k dispozici velký počet rovnocenných kandidátů bez možnosti, jak z nich odlišit toho nejlepšího pro celkový výsledek. Čistá vlastní implementace K–L pravděpodobně nalezla výrazně lepší řešení díky tomu, že narozdíl od použití v navrženém algoritmu pracovala na plnou délku sekvence prohození a navíc byl v některém kroku vybrán z řady vrcholů ze stejným ziskem při prohození ten výhodnější. Faktorem výběru přitom byl pouze způsob implementace.

Stupeň	**Počet vrcholů**
1	65
2	668
3	4091

Tabulka 10.2: Počty vrcholů daného stupně u grafu `uk`

Velkou vypovídací hodnotu při posuzování kvality výsledků navrženého algoritmu má srovnání s nejlepšími dosud publikovanými výsledky. V tab. 10.3 je uvedeno srovnání nejlepší hodnoty dosažené navrženým algoritmem a nejlepšími známými hodnotami. Tato tabulka obsahuje nejlepší výsledek ze všech konfigurací běhu navrženého algoritmu včetně speciálních. Stejné srovnání pouze pro základní nebo rozšířené experimenty bylo uvedeno již v tab. 9.4 resp. 9.7. Z tabulky vyplývá, že v 11 případech z 19 bylo dosaženo stejně kvalitní řešení jako nejlepší dosud známé. U 3 grafů bylo nalezeno řešení pouze o jednu hranu horší než nejlepší známé, u 1 grafu řešení horší o 2 hrany. Pouze u 2 grafů bylo nalezeno řešení s výrazně horším procentuálním i absolutním rozdílem.

Graf	**Nejlepší známé**	**Nejlepší navr. EA**	%	**Rozdíl**
add20	596	609	102,18	13
data	189	189	100,00	0
3elt	90	90	100,00	0
uk	19	430	2263,16	411
add32	11	72	654,55	61
bcsstk33	10171	10171	100,00	0
whitaker3	127	127	100,00	0
crack	184	184	100,00	0
wing_nodal	1707	1708	100,06	1
cti	334	334	100,00	0
u500.05	2	3	150,00	1
u500.10	26	26	100,00	0
u1000.10	39	40	102,56	1
u1000.40	737	737	100,00	0
g500.02	626	626	100,00	0
g500.04	1744	1744	100,00	0
g1000.01	1362	1373	100,81	11
g1000.02	3382	3384	100,06	2
grid64x64	64	64	100,00	0

Tabulka 10.3: Porovnání nejlepších známých výsledků s výsledky navrženého algoritmu

Kapitola 11

Závěr

Jedním z hlavních cílů této práce byl návrh hybridního evolučního algoritmu pro řešení PMBG kombinujícího evoluční prohledávání a lokální optimalizaci pomocí Kernighan–Lin heuristiky. Byla prostudována relevantní literatura a popsány stávající techniky používané při evolučním řešení bisekce grafu. Na základě získaných informací a množství experimentů během vývoje byl navržen vlastní algoritmus. Součástí jeho programové realizace je i vlastní efektivní varianta K–L algoritmu.

Pro experimentální část byly vybrány často používané grafy s dobře popsanými výsledky. Po předběžných experimentech byly nastaveny hodnoty parametrů pro základní běh navrženého algoritmu. Již v této fázi bylo dosaženo u 11 grafů stejně kvalitní řešení jako nejlepší známé, ostatní se podařilo v rozšířených experimentech ještě více přiblížit.

Pro porovnání s čistou implementací K–L byl nejprve ověřen předpoklad, že vlastní a vybraná jiná implementace K–L produkují na stejných grafech a počátečních rozděleních srovnatelně kvalitní výsledky. Klíčovým potom bylo srovnání výsledků vlastní K–L a navrženého algoritmu na stejných grafech a počátečních rozděleních. Ukázalo se, že ve všech případech kromě jediného skutečně navržený algoritmus dokázal K–L heuristiku předčít. Tím byl splněn další z hlavních cílů práce.

Srovnání výsledků navrženého algoritmu s nejlepšími známými vyznělo pozitivně, protože bylo v 11 případech dosaženo stejné hodnoty a ve 4 případech hodnoty jen nepatrně horší. Pouze ve 2 případech se dal výsledek navrženého algoritmu označit jako výrazně horší.

Výsledkem práce tedy je navržený a implementovaný hybridní evoluční algoritmus, který výborně obstál ve srovnání s klasickou K–L heuristikou. Jsem spokojený i se srovnáním s nejlepšími dosud známými hodnotami. Rezervy a možnosti vylepšení spatřuji v hlubší analýze grafů s horšími výsledky a aplikaci takto získaných poznatků do návrhu.

Z osobního pohledu oceňuji to, že jsem při tvorbě této práce získal řadu zkušeností ve velmi zajímavém oboru, jakým pro mě evoluční algoritmy jsou.

Literatura

[1] BURKE, E. K. et al. Hyper-heuristics. *J Oper Res Soc*. Dec 2013, 64, 12, s. 1695–1724. ISSN 0160-5682. Dostupné z: <http://dx.doi.org/10.1057/jors.2013.71>.

[2] CHARDAIRE, P. – BARAKE, M. – MCKEOWN, G. P. A PROBE-Based Heuristic for Graph Partitioning. *IEEE Trans. Comput.* December 2007, 56, 12, s. 1707–1720. ISSN 0018-9340. doi: 10.1109/TC.2007.70760. Dostupné z: <http://dx.doi.org/10.1109/TC.2007.70760>.

[3] CHRIS WALSHAW, U. o. G. *The Graph Partitioning Archive* [online]. 2014. Dostupné z: <http://staffweb.cms.gre.ac.uk/~wc06/partition/>.

[4] DEMMEL, J. *Lecture notes for Applications of Parallel Computers : Graph Partitioning* [online]. 2006. Dostupné z: <http://www.cs.berkeley.edu/~demmel/cs267_Spr06/Lectures/Lecture12/lecture_12_graphpartitioning_jd2006.ppt>.

[5] DUN, N. *An Implementation of State-of-the-Art Graph Bisection Algorithms* [online]. 2006. Dostupné z: <http://web.yl.is.s.u-tokyo.ac.jp/~dunnan/pub/grch06.pdf>.

[6] FERGGO. *Google project hosting : Sample implementation of the Kernighan-Lin algorithm for VLSI Design Automation* [online]. 2009. Dostupné z: <https://code.google.com/p/vlsi-design-automation/source/browse/trunk/Kernighan-Lin/main.cpp>.

[7] HWANG, I. – KIM, Y.-H. – MOON, B.-R. Multi-attractor Gene Reordering for Graph Bisection. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, s. 1209–1216, New York, NY, USA, 2006. ACM. doi: 10.1145/1143997.1144188. Dostupné z: <http://doi.acm.org/10.1145/1143997.1144188>. ISBN 1-59593-186-4.

[8] KERNIGHAN, B. – LIN, S. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell Systems Technical Journal*. 1970, 49, 2.

[9] KIM, J. et al. Genetic Approaches for Graph Partitioning: A Survey. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, s. 473–480, New York, NY, USA, 2011. ACM. doi: 10.1145/2001576.2001642. Dostupné z: <http://doi.acm.org/10.1145/2001576.2001642>. ISBN 978-1-4503-0557-0.

[10] KOCHETOV, Y. – PLYASUNOV, A. Genetic local search the graph partitioning problem under cardinality constraints. *Computational Mathematics and Mathematical Physics*. 2012, 52, 1, s. 157–167. ISSN 0965-5425. doi: 10.1134/S096554251201006X. Dostupné z: <http://dx.doi.org/10.1134/S096554251201006X>.

[11] KOOHESTANI, B. *Genetic Hyper-Heuristics for Graph Layout Problems*. PhD thesis, Computer Science and Electronic Engineering, University of Essex, UK, 15 January 2013. Dostupné z: <http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/Behrooz_PHDThesis_FinalVersion.pdf>.

[12] MENOUAR, B. Genetic algorithm encoding representations for graph partitioning problems. In *Machine and Web Intelligence (ICMWI), 2010 International Conference on*, s. 288–291. IEEE, 2010.

[13] MILLER, G. L. et al. Separators for sphere-packings and nearest neighbor graphs. *J. ACM*. 1997, 44, s. 1–29. Dostupné z: <https://www.cs.cmu.edu/~glmiller/Publications/Papers/MiTeThVa97a.pdf>.

[14] MISIR, M. *Hyperheuristics bibliography* [online]. 2014. Dostupné z: <http://allserv.kahosl.be/~mustafa.misir/hh.html>.

[15] PADERBORN UNIVERSITY, D. o. C. S. *AG Monien Research : Graph partitioning* [online]. 2014. Dostupné z: <http://www2.cs.uni-paderborn.de/fachbereich/AG/monien/RESEARCH/PART/>.

[16] POKLUDA, M. Přehled a implementace vybraných algoritmů používaných pro dělení grafu. Master's thesis, VŠB – Technická univerzita Ostrava, Česká republika, 2013.

[17] SANDERS, P. – SCHULZ, C. High Quality Graph Partitioning. Dostupné z: <http://www.cc.gatech.edu/dimacs10/papers/%5B01%5D-high_quality_graph_partitioning_final.pdf>.

[18] INFORMATICS, B. I. U. *Genetic Algorithm and Graph Partitioning* [online]. 2014. Dostupné z: <http://bio.informatics.indiana.edu/jhl2/I690-genomics-Sp06/save/Seung-hee-Bae-GBA.ppt>.

[19] SEOUL NATIONAL UNIVERSITY, O. L. D. o. C. S. *Benchmark data* [online]. 2014. Dostupné z: <http://soar.snu.ac.kr/benchmark/>.

[20] SOPER, A. J. – WALSHAW, C. – CROSS, M. A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph-Partitioning. *J. of Global Optimization*. June 2004, 29, 2, s. 225–241. ISSN 0925-5001. doi: 10.1023/B:JOGO.0000042115.44455.f3. Dostupné z: <http://dx.doi.org/10.1023/B:JOGO.0000042115.44455.f3>.

-

Příloha A

Seznam použitých zkratek

BFS Breadth-first search, prohledávání do šířky

cutsize Váha nebo počet hran odstraněných při bisekci

D Rozdíl počtu externích a interních hran vrcholu ($D = E - I$)

EA Evoluční algoritmus

FerggoKL Vybraná referenční implementace K–L algoritmu

GP Graph partitioning problém

K Optimální počet prohození s maximálním ziskem v evolučním algoritmu

K-L Kernighan–Lin algoritmus, heuristika

PMBG Problém hledání minimální bisekce grafu

⋮

Příloha B

Iniciální rozdělení

Tabulka se všemi hodnotami *cutsize* náhodných iniciálních rozdělení použitých v experimentech

Graf	**Seed**										**Souhrn**		
	123	124	125	126	127	128	129	130	131	132	nejlepší	průměr	medián
add20	3716	3739	3748	3725	3778	3803	3635	3761	3726	3811	3635	3744,2	3743,5
data	7444	7597	7516	7602	7472	7506	7520	7559	7686	7535	7444	7543,7	7527,5
3elt	6847	6777	7014	6818	6801	6829	6868	6945	6783	6847	6777	6852,9	6838
uk	3385	3450	3387	3441	3402	3463	3452	3401	3428	3443	3385	3425,2	3434,5
add32	4677	4786	4699	4764	4728	4727	4742	4785	4707	4749	4677	4736,4	4735
bcsstk33	146028	146075	146038	145881	146228	145907	145715	145793	146085	145305	145305	145905,5	145967,5
whitaker3	14420	14420	14365	14434	14531	14464	14613	14626	14512	14296	14296	14468,1	14449
crack	15274	15218	15235	15164	15179	15419	15153	15074	15124	15195	15074	15203,5	15187
wing_nodal	37694	37537	37787	37433	37744	37736	37566	37607	37768	37939	37433	37681,1	37715
cti	24324	23990	24120	23997	24190	24195	24043	23996	24370	24315	23990	24154	24155
u500.05	649	624	633	665	677	647	627	657	604	690	604	647,3	648
u500.10	1156	1159	1220	1203	1212	1170	1180	1170	1216	1171	1156	1185,7	1175,5
u1000.10	2282	2384	2339	2294	2351	2391	2352	2329	2358	2368	2282	2344,8	2351,5
u1000.40	9079	9044	9047	9077	9086	9094	8973	9007	9038	9024	8973	9046,9	9045,5
g500.02	1161	1192	1155	1190	1175	1162	1178	1199	1182	1220	1155	1181,4	1180
g500.04	2640	2601	2590	2561	2602	2574	2565	2550	2581	2632	2550	2589,6	2585,5
g1000.01	2531	2539	2590	2573	2536	2581	2482	2483	2529	2509	2482	2535,3	2533,5
g1000.02	5096	5080	5045	5155	5072	5151	5084	5065	4994	5095	4994	5083,7	5082
grid64x64	3974	3998	4125	4016	4002	3989	4045	4030	4067	4004	3974	4025	4010

Tabulka B.1: Iniciální rozdělení

Příloha C

Výsledky experimentů

Tabulka se všemi hodnotami *cutsize* získaných při experimentech

Graf	Seed										Souhrn		
	123	124	125	126	127	128	129	130	131	132	nejlepší	průměr	medián
add2	895	873	877	785	772	743	859	775	639	759	639	797,7	780
data	190	190	189	190	190	190	190	190	190	190	189	189,9	190
3elt	90	90	90	90	90	90	90	90	90	90	90	90	90
uk	500	534	508	506	482	516	547	493	510	531	482	512,7	509
add32	313	294	329	336	325	329	261	280	323	354	261	314,4	324
bcsstk33	10171	10171	10171	10171	10171	10171	10171	10171	10171	10171	10171	10171	10171
whitaker3	127	127	127	127	127	127	127	127	127	127	127	127	127
crack	184	184	184	184	184	184	184	184	184	184	184	184	184
wing_nodal	1711	1714	1710	1709	1714	1715	1712	1713	1708	1712	1708	1711,8	1712
cti	334	334	334	334	334	334	334	334	334	334	334	334	334
u500.05	20	18	13	13	8	10	10	11	11	7	7	12,1	11
u500.10	26	26	26	29	47	58	45	46	41	36	26	38	38,5
u1000.10	83	62	52	92	68	74	58	63	76	60	52	68,8	65,5
u1000.40	737	737	737	737	737	737	737	737	737	737	737	737	737
g500.02	632	630	632	635	630	626	628	626	627	626	626	629,2	629
g500.04	1744	1744	1747	1744	1747	1744	1750	1744	1752	1750	1744	1746,6	1745,5
g1000.01	1388	1377	1383	1393	1380	1387	1376	1382	1391	1389	1376	1384,6	1385
g1000.02	3412	3402	3404	3386	3393	3385	3396	3389	3384	3413	3384	3396,4	3394,5
grid64x64	64	64	64	64	64	64	64	64	64	64	64	64	64

Tabulka C.1: Výsledky experimentů

Příloha D

Výsledky rozšířených experimentů

Tabulka se všemi hodnotami *cutsize* získaných při experimentech s rozšířeným nastavením výpočtu

Graf	**Seed**										**Souhrn**		
	123	124	125	126	127	128	129	130	131	132	nejlepší	průměr	medián
add20	691	690	742	746	695	619	800	623	609	624	609	683,9	690,5
uk	466	464	459	443	448	476	477	446	457	430	430	456,6	458
add32	124	153	133	146	152	139	121	144	127	132	121	137,1	136
u500.05	6	8	9	10	10	6	11	10	11	7	6	8,8	9,5
u1000.10	57	54	50	64	46	54	49	40	62	50	40	52,6	52
g1000.01	1377	1373	1380	1373	1382	1379	1382	1384	1382	1381	1373	1379,3	1380,5
g1000.02	3387	3392	3393	3389	3391	3392	3390	3394	3395	3402	3387	3392,5	3392

Tabulka D.1: Výsledky rozšířených experimentů

Příloha E

Výsledky vlastní implementace K–L

Tabulka se všemi hodnotami *cutsize* získanými při výpočtech vlastní implementací K–L algoritmu.

Graf	Seed iniciálního rozdělení										Souhrn		
	123	124	125	126	127	128	129	130	131	132	nejlepší	průměr	medián
add20	904	892	1003	888	814	940	844	713	811	656	656	846,5	866
data	202	236	211	251	233	275	240	218	302	252	202	242	238
3elt	134	90	146	90	174	136	117	175	152	90	90	130,4	135
uk	39	36	87	57	85	51	71	68	66	56	36	61,6	61,5
add32	334	329	437	358	456	297	312	330	404	284	284	354,1	332
bcsstk33	12070	12179	11996	11909	10173	11764	11909	10228	12148	10172	10172	11454,8	11909
whitaker3	133	130	136	166	128	129	133	135	132	132	128	135,4	132,5
crack	306	460	203	207	186	304	219	207	239	200	186	253,1	213
wing_nodal	1837	3019	1804	1897	1904	1816	2157	2048	2554	2128	1804	2116,4	1976
cti	631	778	787	366	366	546	366	600	367	406	366	521,3	476
u500.05	42	32	41	31	29	29	31	56	27	39	27	35,7	31,5
u500.10	70	125	138	122	99	114	83	74	94	114	70	103,3	106,5
u1000.10	189	118	132	130	227	197	124	181	188	97	97	158,3	156,5
u1000.40	812	972	737	737	971	737	880	737	742	812	737	813,7	777
g500.02	650	664	640	666	651	657	663	662	664	663	640	658	662,5
g500.04	1798	1777	1791	1811	1794	1767	1797	1768	1805	1773	1767	1788,1	1792,5
g1000.01	1443	1410	1460	1465	1435	1449	1429	1452	1431	1438	1410	1441,2	1440,5
g1000.02	3482	3456	3515	3514	3490	3465	3461	3489	3430	3468	3430	3477	3475
grid64x64	81	64	69	74	82	64	64	64	64	94	64	72	66,5

Tabulka E.1: Výsledky vlastní implementace K–L

Příloha F

Výsledky jiné implementace K–L

Tabulka se všemi hodnotami *cutsize* získanými při výpočtech implementací FerggoK–L.

Graf	Seed iniciálního rozdělení										Souhrn		
	123	124	125	126	127	128	129	130	131	132	nejlepší	průměr	medián
add20	800	635	793	895	795	792	852	702	908	744	635	791,6	794
data	261	210	212	189	235	213	228	194	196	228	189	216,6	212,5
3elt	223	138	190	153	190	135	171	178	135	156	135	166,9	163,5
uk	536	540	564	499	477	534	563	714	393	544	393	536,4	538
add32	442	353	519	441	457	474	365	430	375	433	353	428,9	437
bcsstk33	12091	11985	12054	11840	10172	11753	10226	10226	12009	10611	10172	11296,7	11796,5
whitaker3	172	243	135	257	157	130	130	146	261	197	130	182,8	164,5
crack	219	419	364	185	415	198	355	251	210	186	185	280,2	235
wing_nodal	2387	2417	1745	1911	1975	2347	2075	1956	1780	1805	1745	2039,8	1965,5
cti	498	764	407	596	655	726	1176	417	448	580	407	626,7	588
u500.05	42	34	37	50	32	29	37	20	27	28	20	33,6	33
u500.10	93	88	75	81	31	86	45	158	143	97	31	89,7	87
u1000.10	178	153	224	282	210	198	113	183	166	254	113	196,1	190,5
u1000.40	812	972	878	862	971	737	887	737	812	814	737	848,2	838
g500.02	651	660	650	662	664	654	661	649	661	654	649	656,6	657
g500.04	1777	1792	1801	1817	1787	1787	1786	1787	1787	1795	1777	1791,6	1787
g1000.01	1452	1433	1428	1454	1429	1442	1431	1469	1459	1440	1428	1443,7	1441
g1000.02	3500	3449	3479	3538	3479	3469	3492	3514	3451	3503	3449	3487,4	3485,5
grid64x64	64	64	64	64	64	64	64	106	64	64	64	68,2	64

Tabulka F.1: Výsledky implementace FerggoK–L

Příloha G

Obsah přiloženého CD