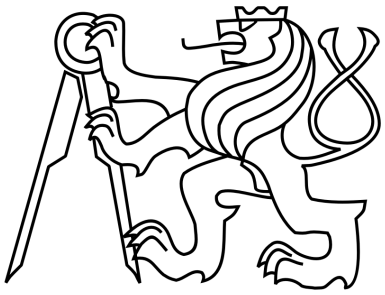
České vysoké učení technické
v Praze

Fakulta elektrotechnická
Katedra kybernetiky



BAKALÁŘSKÁ PRÁCE

Optimální směrování v sítích s omezeními pomocí evolučních algoritmů

Autor: Tomáš Polomský
Vedoucí práce: Ing. Jiří Kubalík, Ph.D.

2009

Abstrakt

V této práci řeším problém optimálního směrování v sítích s omezeními. Úkolem je z množiny linek v dané síti vybrat nejlevnější podmnožinu tak, aby bylo možné přenášet data ze zdrojových do cílových uzlů a přitom splnit různé omezující podmínky. Tyto podmínky zahrnují požadavky na přenosovou kapacitu a zpoždění jednotlivých linek a také omezení na celkové zpoždění. K dalším podmínkám patří, že některé zprávy musí být přenášeny pouze po zabezpečených protokolech. Tento problém patří mezi NP-těžké a větší instance nemohou být vyřešeny optimálně v rozumném čase. Tento problém řeším pomocí dvou algoritmů: evolučního algoritmu a Prototype Optimisation with Evolved Improvement Steps. Popisuji princip jejich funkce, konkrétní implementaci a srovnání výsledků těchto algoritmů oproti jiným způsobům řešení tohoto problému: celočíselným lineárním programováním, Column Generation a Lagrangean Decomposition.

Abstract

In this thesis I solve a optimal routing problem with constraints. The objective is to select the cheapest subset of links in the given network which enables routing messages from source nodes to target nodes with respect to various constraints. These constraints include particular capacity and delay constraints for each message and a global delay constraint. Moreover some messages require transfer using secured protocols. This problem is NP-hard and larger instances can not be solved optimally in practice. The problem I solve using two algorithms: Genetic Algorithm and Prototype Optimisation with Evolved Improvement Steps. I describe the principle of their function, the implementation and compare them with other methods for solving the problem. Integer Linear Programming, Column Generation and Lagrangean Decomposition.

Poděkování

Chci poděkovat svému vedoucímu práce za jeho podnětné připomínky a zlepšující návrhy.

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literatura, projekty, SW, atd.) uvedené v přiloženém seznamu.

V Praze dne 8. 7. 2009

Polomský
podpis

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Tomáš P o l o m s k ý

Studijní program: Softwarové technologie a management

Obor: Inteligentní systémy

Název tématu: Optimální směrování v sítích s omezeními pomocí evolučních algoritmů

Pokyny pro vypracování:

1. Prostudujte stávající klasické a netradiční postupy (založené na evolučních algoritmech) pro optimalizaci směrování transportů v síti s omezenou kapacitou.
2. Naimplementujte algoritmus POEMS pro tento problém.
3. Experimentálně ověřte funkčnost algoritmu na standardních testovacích datech, dosažené výsledky vyhodnoťte.

Seznam odborné literatury: Dodá vedoucí práce.

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

Platnost zadání: do konce zimního semestru 2009/2010

L.S.

prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry

doc. Ing. Boris Šimák, CSc.
děkan

V Praze dne 23. 2. 2009

Obsah

1 Úvod **1**

2 Formulace problému **3**

3 Evoluční algoritmus **5**
3.1 Jedinec 6
3.2 Populace 6
3.3 Fitness 6
3.4 Křížení 7
3.5 Mutace 7
3.6 Generování počáteční populace 7
3.7 Selekce 8
3.8 Nahrazovací strategie 8

4 POEMS **9**
4.1 Akce 10
4.2 Sekvence akcí 10
4.3 Populace a rozdělení do šuplíků 11
4.4 Vnitřní evoluční algoritmus 11
4.4.1 Jedinec 11
4.4.2 Fitness 11
4.4.3 Mutace 12
4.4.4 Generování počáteční populace 12
4.4.5 Nahrazovací strategie 12
4.5 Běh algoritmu 12

5 Implementace **14**
5.1 Evoluční algoritmus 14
5.1.1 Jedinec 14
5.1.2 Populace 15
5.1.3 Fitness 15
5.1.4 Dijkstrův algoritmus 16
5.1.5 Křížení 17
5.1.6 Mutace 17
5.1.7 Generování počáteční populace 17
5.1.8 Selekce 18
5.1.9 Běh algoritmu a nahrazovací strategie 18
5.1.10 Úpravy algoritmu 18
5.2 POEMS 20
5.2.1 Prototyp 20
5.2.2 Akce 21
5.2.3 Sekvence akcí 21
5.2.4 Vlastnosti vnitřního evolučního algoritmu 21
Fitness 21
Křížení 22
Mutace 22
Opravy 23
Generování počáteční populace 23
Selekce 24
5.2.5 Běh algoritmu a nahrazovací strategie 24

6 Experimenty **26**
6.1 Evoluční algoritmus 27
6.2 POEMS 27
6.3 Výsledky experimentů 28

7 Závěr **31**

Kapitola 1

Úvod

V této práci řeším problém návrhu sítě. Je potřeba propojit různá zařízení do stejné komunikační sítě z důvodu efektivní výměny informací. Protože zasílané zprávy jsou kritické z hlediska bezpečnosti, musí být splněna některá kritéria. Nejdůležitějším kritériem je, že všechny zprávy musí být doručeny do stanovené doby. Vybrané zprávy musí být přeneseny rychleji něž ostatní a celkový čas k přenesení všech zpráv musí být ve stanoveném limitu. Dále pak, některé zprávy musí být přeneseny přes zabezpečené síťové linky, což pravděpodobně způsobí vyšší náklady za transport. Cílem optimalizace je nalézt síť s minimální cenou, která bude dostačující na přenesení zpráv při maximální zátěži.

Jednou z metod, která se dá použít na řešení tohoto problému je celočíselné lineární programování. Výhodou je, že pokud najde řešení, je vždy optimální. Tento problém však patří mezi NP-těžké — není znám (a pravděpodobně ani neexistuje) algoritmus, který by tento problém vyřešil v polynomiálním čase. Proto celočíselné lineární programování nemusí najít v rozumném čase řešení (kvůli hledání optimálního výsledku). Z tohoto důvodu se tento problém řeší pomocí heuristických metod, které sice nezaručují optimální výsledek, ale umožňují řešit i větší instance.

Pro řešení této úlohy jsem se rozhodl použít dva algoritmy a to evoluční algoritmus a algoritmus POEMS [5]. Vybral jsem je proto, že jsem chtěl vyzkoušet jestli evoluční algoritmy můžou efektivně řešit tyto obtížné problémy. Výsledky chci porovnat s jinými metodami řešení tohoto problému

jako je celočíselné lineární programování, Column Generation a Lagrangean Decomposition. U těchto termínů neznám vhodný český ekvivalent a proto budu používat jejich anglické názvy.

V druhé kapitole uvedu formální zadání problému. Ve třetí kapitole popíšu obecně evoluční algoritmus. Obsahem čtvrté kapitoly je obecný popis algoritmu POEMS. V páté kapitole popíšu konkrétní implementace evolučního algoritmu a algoritmu POEMS. V šesté kapitole uvedu porovnání výsledků algoritmů a v sedmé kapitole shrnu výsledky této práce.

Kapitola 2

Formulace problému

Je dán neorientovaný graf $G = (V, E)$ s množinou hran E definovanou na množině uzlů $V = S \cup T \cup M$, která se dělí na zdrojové, cílové a spojovací uzly. Množina hran (linek) $E = E^+ \cup E^- \cup E^*$ je rozdělena na množinu hran E^+, které podporují jen zabezpečené protokoly, E^-, které podporují jen nezabezpečené protokoly a množinu hran E^*, které podporují oba druhy protokolů.

Dále je dána množina m transportů T, které modelují veškeré zprávy, které by měly být přenášeny po síti. Transport k je definován pěticí $(s_k, t_k, v_k, \sigma_k \delta_k)$ kde $s_k, t_k \in V$ určují zdrojový a cílový uzel, $v_k \in \mathbb{R}^+$ určuje velikost transportu a $\sigma \in \{0, 1\}$ označuje zda transport musí být veden po zabezpečených protokolech. Časově kritické transporty musí být přeneseny v čase $\delta_k > 0$; jinak uvažujeme $\delta_k = \infty$.

Pro každou hranu je zadána nezáporná cena $c_{ij} \geq 0$, zpoždění $d_{ij} \geq 0$ a kapacita $u_{ij} \geq 0$. K ceně hran se dále přičítá cena za použití protokolů

$$p_{ij}^k = \begin{cases} \min(p_{sec}, p_{insec}), & \text{když } \sigma_k = 0 \wedge (i,j) \in E^* \\ p_{insec}, & \text{když } (i,j) \in E^- \\ p_{sec}, & \text{jinak} \end{cases} \tag{2.1}$$

kde $p_{sec} \geq 0$ označuje cenu zabezpečeného protokolu a $p_{insec} \geq 0$ označuje cenu nezabezpečeného protokolu. Zpoždění při použití daného protokolu jsou

označeny podobně

$$a_{ij}^k = \begin{cases} \min(a_{sec}, a_{insec}), & \text{když } \sigma_k = 0 \wedge (i,j) \in E^* \\ a_{insec}, & \text{když } (i,j) \in E^- \\ a_{sec}, & \text{jinak} \end{cases} \tag{2.2}$$

kde $a_{sec} \geq 0$ označuje zpoždění zabezpečeného protokolu a $a_{insec} \geq 0$ označuje zpoždění nezabezpečeného protokolu.

Cílem řešení tohoto problému je nalézt vhodné směrování pro každý transport k, tedy cestu z s_k do t_k splňující omezení na zpoždění. Jedna hrana může být použita pro více transportů, ale protože jsou všechny transporty posílány najednou, sdílí kapacity hran. Kromě omezení zpoždění pro každý transport (2.3) je zde také podmínka pro celkové zpoždění všech transportů (2.4), které určuje, že celkový součet zpoždění musí být menší než konstanta D.

Cena řešení (2.5) se vypočítá jako součet cen použitých linek a součet cen protokolů využitých jednotlivými transporty. x_{ij} je binární hodnota, určující jestli je hrana $(i,j) \in E$ použita v nějakém směrování, f_{ij}^k určuje, jestli je hrana $(i,j) \in E$ použita pro transport k. Cílem je nalézt nejlevnější řešení splňující všechny omezující podmínky.

$$\forall k : \sum_{(i,j)\in E} \left(d_{ij} + a_{ij}^k\right) \cdot f_{ij}^k \leq \delta^k \tag{2.3}$$

$$\sum_{k=1}^{m} \sum_{(i,j)\in E} \left(d_{ij} + a_{ij}^k\right) \cdot f_{ij}^k \leq D \tag{2.4}$$

$$C = \sum_{(i,j)\in E} \left(c_{ij} \cdot x_{ij} + \sum_{k=1}^{m} f_{ij}^k \cdot p_{ij}^k \right) \tag{2.5}$$

Kapitola 3

Evoluční algoritmus

Prvním z algoritmů, které jsem se rozhodl použít na řešení této úlohy je evoluční algoritmus. Je to stochastický optimalizační algoritmus inspirovaný Mendelovou teorií dědičnosti (transport vlastností z rodičů na potomky) a Darwinovou teorií (náhodné změny struktury jedince a přežití nejsilnějšího). Jeho vlastností je, že v průběhu výpočtu si udržuje množinu řešení (populaci). Jeho výhodou je, že vždy vrátí nějaké řešení, nevýhodou je, že nezaručuje nalezení optimálního řešení. Činnost algoritmu by se dala shrnout následujícím pseudokódem:

Algorithm 3.1 Pseudokód evolučního algoritmu

```
1 Inicializace: Vygeneruj a ohodnoť populaci
2 while (není splněna ukončovací podmínka) {
3         Selekce: vyber z populace jedince-rodiče
4         Křížení: Vytvoř z rodičů potomky
5         Mutace: Náhodně uprav potomky
6         Ohodnocení: Ohodnoť potomky
7         Nahrazení: Nahraď jedince v populaci jejich
              potomky
8 }
```

Na řádku 2 je určena podmínka a dokud platí, tedy dokud není splněna ukončovací podmínka, opakují se řádky 3 až 7. Složené závorky zde uzavírají kód, který se má opakovat.

3.1 Jedinec

Jedinec je konkrétní řešení daného problému. Toto řešení může být zapsáno libovolnou formou (např. binární řetězec, vektor reálných čísel, matice, graf, atd.). Forma zápisu se volí tak, aby byla vhodná vzhledem k operacím jako je křížení a mutace.

3.2 Populace

Populace je množina jedinců (kandidátů na výsledné řešení), kterou si algoritmus udržuje po celou dobu běhu. Z populace jsou selekcí vybíráni jedinci pro křížení a mutaci a dále jsou jedinci v populaci nahrazováni novými. Nahrazovací strategie pak určuje, jakým způsobem bude nakládáno se starými jedinci v průběhu generací. Buď může být nahrazena celá populace novými jedinci nebo pouze její část. Velikost populace je jedním z faktorů ovlivňující efektivnost evolučního algoritmu. Malá populace značně omezuje variabilitu jedinců a tak omezuje možnosti hledání řešení, příliš velká populace zvyšuje náročnost algoritmu na výpočetní čas.

V případě velkých problémů, kde se pro řešení používá více počítačů, se může použít metoda tzv. ostrovů. Každý z počítačů si udržuje vlastní populaci, kterou zlepšuje a nejlepší jedinci se pak vyměňují mezi počítači. Dokonce mohou mít jednotlivé počítače jinak nastaveny parametry algoritmu (např. mít i jiné operátory křížení a mutace).

3.3 Fitness

Označuje kvalitu jedince. Je to jediná informace, kterou dáváme evolučnímu algoritmu o řešeném problému. Hodnota fitness může být vypočítána funkcí, zadaná člověkem, získána výsledkem simulace atd. Cílem algoritmu je potom nalézt jedince s maximální (respektive minimální) fitness. Fitness může být dále snižována (respektive zvyšována) o penalizaci, pokud řešení nesplňuje omezující podmínky, a výše penalizace je úměrná míře nesplnění podmínek. Toto nastavení nutí evoluční algoritmus upřednostňovat řešení, která lépe splňují podmínky.

3.4 Křížení

Křížení vychází z jednoduché myšlenky: „Mám-li dvě dobrá řešení, jejich vhodnou kombinací mohu získat lepší“. Křížení kombinuje jedince a výstupem křížení je jeden nebo dva jedinci. Metoda křížení závisí na způsobu reprezentace řešení. Křížení může být implementováno slepě, to znamená bez jakékoli informace o způsobu uložení dat. Příkladem může být křížení dvou čísel kódovaných jako binární řetězce, kde se náhodně vybírají bity z prvního nebo druhého čísla. Druhým způsobem křížení je implementace vycházející ze znalosti o řešeném problému. Evoluční algoritmy mohou mít také více metod křížení a náhodně mezi nimi vybírat. Podle implementace křížení mohou vzniknout jedinci, kteří vždy splňují omezující podmínky a nebo jedinci, kteří je splňovat nemusí. Řešení tohoto problému se nabízejí dvě:

1. Jedince penalizovat a tím je znevýhodňovat při selekci — nevýhodou této metody je, že evolučnímu algoritmu může trvat výrazně déle, než nalezne řešení splňující omezující podmínky

2. Jedince opravit pomocí opravné funkce — nevýhodou je, že opravná funkce omezuje prohledávaný prostor a pokud je zvolena nevhodně, může dokonce odříznout optimální řešení

3.5 Mutace

Pro zachování různorodosti populace a možnosti vzniků nových znaků v populaci se používá v evolučních algoritmech mutace. Je aplikována náhodně na některé jedince v populaci. Během mutace se změní některé znaky v jedinci.

3.6 Generování počáteční populace

Pro start evolučního algoritmu je nutné nejdřív inicializovat počáteční populaci. Jedince je možné generovat náhodně, na základně apriorní znalosti — tady je však nebezpečí že populaci zinicializujeme špatně, a nebo běhy jiných krátkých algoritmů.

3.7 Selekce

Nalezení jedinců vhodných pro křížení obstarává selekce. Existuje mnoho způsobů selekcí: turnajové křížení, ruletové kolo, atd. Důležité je, aby selekce spravedlivě vybírala jedince a upřednostňovala jedince s lepší fitness. Spravedlivě znamená, že i jedinec s horší fitness může být vybrán.

3.8 Nahrazovací strategie

Společně se selekcí určuje nahrazovací strategie vývoj populace. Rozlišujeme dvě základní nahrazovací strategie:

1. Nahrazena je celá populace — všichni jedinci tak jsou výsledkem křížení jedinců (a případně mutace) jedinců předchozí generace. Nevýhodou je, že tak může dojít ke ztrátě nejlepšího řešení (potomek nejlepšího jedince už není tak dobrý). Tato nevýhoda se dá kompenzovat tak, že nejlepší jedinec je také udržován mimo populaci a i když dojde ke ztrátě v populaci, algoritmus má stále k dispozici nejlepší řešení.

2. Nahrazena je část populace — zbytek populace pochází z původní populace beze změny. Jedinci, kteří přetrvají do další generace beze změny můžou být buď vybráni podle toho, že mají nejlepší fitness a nebo selekcí. Vlastnosti algoritmu, kdy nejlepší jedinec (jedinci) přetrvá do další generace se říká elitismus. Výhodou je, že nejlepší řešení se v populaci neztrácí a účastní se křížení a mutace. Nevýhodou je, že pokud se nachází dosud nejlepší nalezené řešení (nejlepší jedinec) v lokálním optimu, může značnou část populace usměrňovat do tohoto optima a tím znemožnit nalezení lepších výsledků.

Kapitola 4

POEMS

Dalším algoritmem, který jsem se rozhodl použít na řešení je Prototype Optimisation with Evolved Improvement Steps (POEMS)[5]. Jedná se o iterační algoritmus, který opakovaně využívá ve svém cyklu evoluční algoritmus. POEMS si udržuje v průběhu vykonávání jedinou reprezentaci řešení, tzv. prototyp. V jednotlivých cyklech generuje sekvence akcí, které mají zlepšit prototyp. Na šlechtění těchto sekvencí využívá evoluční algoritmus. Po dokončení evolučního algoritmu se nejlepší vyšlechtěná sekvence akcí aplikuje na prototyp a pokud výsledek má lepší fitness než původní prototyp, je původní prototyp nahrazen. Činnost algoritmu shrnuje následující pseudokód:

Algorithm 4.1 Pseudokód algoritmu POEMS

```
1 Inicializace prototypu: Vygeneruj prototyp
2 while (nesplňena ukončovací podmínka) {
3         Inicializace: vygeneruj a ohodnoť sekvence akcí
4         Generování sekvence akcí: Spusť Evoluční
              algoritmus pro prototyp
5         generování kandidáta: aplikuj sekvenci akcí na
              prototyp, výsledek ulož jako kandidáta
6         nahrazení prototypu: pokud je kandidát lepší
              než prototyp, nahraď prototyp kandidátem
7 }
```

4.1 Akce

Obsahuje informaci o způsobu, jakým modifikovat prototyp. Akce dále obsahuje binární hodnotu $\{0, 1\}$ určující, jestli je daná akce aktivní. Neaktivní akce je označena jako NOP a při aplikaci na prototyp jej nezmění. Přitom si však akce stále nese informaci o způsobu modifikace prototypu, takže pokud je během mutací akce nastavena na NOP a zpátky, tuto informaci neztratí.

4.2 Sekvence akcí

Sekvence akcí je seřazený seznam akcí, které se v daném pořadí aplikují na prototyp. Sekvence akcí jsou v každé iteraci POEMS hledány pomocí vnitřního evolučního algoritmu. Všechny sekvence mají stejnou délku (stejný počet akcí). Protože není předem jasný ideální počet aktivních akcí a tedy jak mají být dlouhé sekvence, jejich počet se odhadne a algoritmu se umožňují používat i neaktivní (NOP) akce. Tím je dosaženo, že pokud ideální počet aktivních akcí je nižší než určená délka sekvence, algoritmus tohoto počtu může dosáhnout a protože je sekvence delší, zbytek akcí v sekvenci budou tvořit neaktivní akce. Způsob, jakým se tyto sekvence objeví v populaci popíšu dále. NOP akce tedy umožňují efektivní implementaci variabilní délky sekvencí akcí a dále nesou informaci, která se může projevit v dalších generacích.

4.3 Populace a rozdělení do šuplíků

Populace je množina sekvencí akcí (jedinců). V průběhu vykonávání algoritmu (po ukončení běhu vnitřního gen. algoritmu), může nastat situace kdy jedinci v populaci zhoršují prototyp. Protože jedinci s nejmenším počtem akcí ho většinou zhoršují nejméně, mohlo by dojít k vymizení jedinců s velkým počtem aktivních akcí. Aby se tomu zabránilo je populace rozdělena do tzv. šuplíků. Šuplíků je přesně tolik, jaká je délka sekvence akcí N. Šuplíky jsou očíslovány v rozsahu $n \in \{1 \dots N\}$. Sekvence akcí jsou v šuplících rozděleny tak, že každá sekvence v daném šuplíku má počet aktivních akcí větší nebo roven číslu šuplíku n. To znamená, že v šuplíku $n = 2$ jsou jedinci, kteří mají minimálně 2 aktivní akce. Toto rozložení do šuplíků se udržuje po celý běh vnitřního evolučního algoritmu (od vygenerování množiny sekvencí akcí až do dokončení šlechtění).

4.4 Vnitřní evoluční algoritmus

Vnitřní evoluční algoritmus šlechtí sekvence akcí. Pro tento evoluční algoritmus platí většina vlastností, které sem uvedl v kapitole 3. Protože je však používán pro jiné účely (nešlechtí řešení ale pouze jeho část) jsou na něj kladeny jiné podmínky. Protože je spuštěn mnohokrát během běhu programu, musí rychle konvergovat k řešení. Proto se délka sekvence akcí určuje výrazně kratší než je velikost problému (počet proměnných problému).

4.4.1 Jedinec

Šlechtěnými jedinci pro evoluční algoritmus jsou sekvence akcí.

4.4.2 Fitness

Fitness u sekvence akcí se určuje tak, že sekvence akcí je aplikována na stávající prototyp a výsledku je určena fitness podobně jako v 3.3.

4.4.3 Mutace

Při mutaci sekvencí akcí dochází k mutaci náhodně vybraných akcí v sekvenci. Přitom může dojít ke dvěma typům změn. Změna aktivity akce (aktivní na neaktivní a naopak) a změna parametru akce, tedy způsobu jak modifikovat prototyp. Pokud je akce neaktivní, je vhodné aby se mutací změnila aktivita (změna parametru akce se u neaktivní akce neprojeví — projevila by se, až při jiné mutaci, který by akci aktivovala). Pokud je akce aktivní, může být mutací deaktivována (nastavena na NOP) nebo u akce změněn její parametr.

4.4.4 Generování počáteční populace

Při generování sekvencí akcí se generují jednotlivé akce tak, že se u nich náhodně určí parametr akce a náhodně aktivita těchto akcí.

4.4.5 Nahrazovací strategie

Aby se dosáhlo rychlé konvergence k řešení, je vhodné nahrazovat jenom část populace novými jedinci, a nahrazovat jen ty nejhorší.

4.5 Běh algoritmu

Algoritmus POEMS běží ve dvou vnořených cyklech. Ve vnějším cyklu zlepšuje prototyp a ve vnitřním cyklu hledá nejlepší sekvenci akcí pro zlepšení prototypu.

1. Na začátku algoritmu se vygeneruje prototyp.
2. Provede se vygenerování počáteční populace (sekvencí akcí). Následně se populace zlepšuje:
   (a) V populaci jsou pomocí selekcí vybráni jedinci a jsou zkříženi nebo zmutováni.

   (b) Vzniklí jedinci jsou pak zařazováni do populace tak aby splňovali podmínky šuplíků. Každý nový jedinec tedy nahradí jednoho původního jedince, který leží v jednom ze šuplíků kam nový jedinec může patřit, ale nahradí ho jenom tehdy, pokud nový jedinec je lepší než původní jedinec.

   (c) opakovat od bodu *a* stanovený počet generací evolučního algoritmu

3. V populaci je vybrán nejlepší jedinec, aplikován na prototyp a tím je vytvořen kandidát. Pokud kandidát je lepší než původní prototyp, je prototyp kandidátem nahrazen.

4. Opakovat od bodu *2* stanovený počet iterací

Kapitola 5

Implementace

Pro implementaci jsem volil mezi jazyky C++ a Java. Protože mám větší zkušenosti s jazykem Java, zvolil jsem tento jazyk. Při výběru vývojového prostředí jsem se rozhodoval mezi NetBeans a Eclipse. Obě jsou kvalitní a nakonec jsem se přiklonil k NetBeans.

5.1 Evoluční algoritmus

5.1.1 Jedinec

Jako formu reprezentace řešení jsem zvolil seznam směrování. Každému transportu je přiřazeno jedno směrování, které obsahuje seřazený seznam linek (hran), kterými je transport veden.

Druhou variantu, kterou jsem zvažoval bylo rozšířit tento seznam linek o informaci, který protokol je pro transport použit. Na základě vstupních dat, které jsem měl pro testování k dispozici však toto rozšíření nebylo potřeba. V testovacích sadách byly definovány pouze 2 protokoly. Zabezpečený a nezabezpečený. Díky tomu mohly nastat pouze dvě varianty:

1. linka podporuje pouze jeden typ protokolu — použitý protokol je jednoznačný
2. linka podporuje oba protokoly — použitý protokol je závislý na tom, jestli je u transportu požadováno zabezpečení nebo ne. Pokud není požadováno zabezpečení je vybrán nezabezpečený protokol.

Vybraný způsob reprezentace je na danou úlohu dostačující, protože v testovacích sadách měl zabezpečený protokol vyšší zpoždění i cenu než nezabezpečený — logický předpoklad a proto je možné jednoznačně určit, který protokol má být použit. Tento způsob reprezentace by nevyhovoval, pokud by bylo definováno více protokolů stejné úrovně zabezpečení (nezabezpečený, zabezpečený) a jeden z nich by měl nižší cenu a větší zpoždění než druhý. V takovém případě by nešlo jednoznačně určit, který protokol použít a bylo by nutné tuto informaci přidat do řešení.

5.1.2 Populace

V implementaci algoritmu jsem se rozhodl pro populaci o velikosti 400 jedinců.

5.1.3 Fitness

V algoritmu počítám fitness podle (2.5). Fitness je dále penalizována v případě porušení omezení. Pokud dojde k porušení maximálního součtu zpoždění přes všechny transporty D nebo kde časově kritický transport má větší zpoždění než je povolené, je jedinec ohodnocen maximální možnou cenou — řešení je označeno za nejhorší možné. V případě, že výsledné řešení přetěžuje linky (součet velikostí transportů danou linkou je větší než její kapacita), je k fitness přičtena penalta za každou přetíženou linku. Penalta P je určena jako součet cen všech dostupných linek a ceny všech transportů za předpokladu, že by transport využil všechny linky a použil dražší protokol.

$$P = \sum_{(i,j)\in E} \left(c_{ij} + \sum_{k=1}^{m} p_{sec} \right) \tag{5.1}$$

Takto velikou penaltou je zajištěno, že jakékoli řešení splňující omezující podmínky bude mít menší fitness než řešení porušující omezení a protože je penalta počítána dynamicky, bude dostatečně velká pro každé zadání (počítá se pouze jednou při inicializaci programu).

5.1.4 Dijkstrův algoritmus

Pro naprogramování operátorů evolučního algoritmu (křížení a mutace) a pro generování počáteční populace jsem se rozhodl použít Dijkstrův algoritmus. Vybral jsem ho proto, že jsem potřeboval algoritmus, který mi umožní nalézt cestu v grafu, aby evoluční algoritmus generoval použitelná řešení. Dijkstrův algoritmus umožňuje nalézt nejkratší cestu v grafu za podmínky, že všechny hrany jsou kladně ohodnoceny.

První implementaci jsem udělal pomocí objektových datových typů jako je množina a mapa. Po několika testech na testovacích sadách malého rozsahu jsem zjistil, že samotné generování počáteční populace je časově velmi náročné. Pomocí modulu Profiler z prostředí NetBeans jsem zjistil, že nejvíc času algoritmus tráví na Dijkstrově algoritmu. Proto jsem se rozhodl, vyměnit nevhodnou implementaci využívající objekty za implementaci využívající pole. Nárůst výkonu byl výrazný. Pro další zvýšení rychlosti jsem dále naprogramoval cache výsledků, která se vytváří při požadavku na nalezení směrování a ukládá si nalezená směrování do paměti. Pokud přijde požadavek na směrování, které už bylo v minulosti vypočítáno, výsledek se nemusí počítat a vrátí se rovnou z paměti. Využitím cache jsem ušetřil velké množství výpočetního času už při generování počáteční populace.

U Dijkstrova algoritmu je nutné určit ohodnocení jednotlivých hran a tedy kritérium, podle kterého bude výsledek Dijkstrova algoritmu optimální. Pro první implementaci jsem zvolil dvě kritéria určování délky hran:

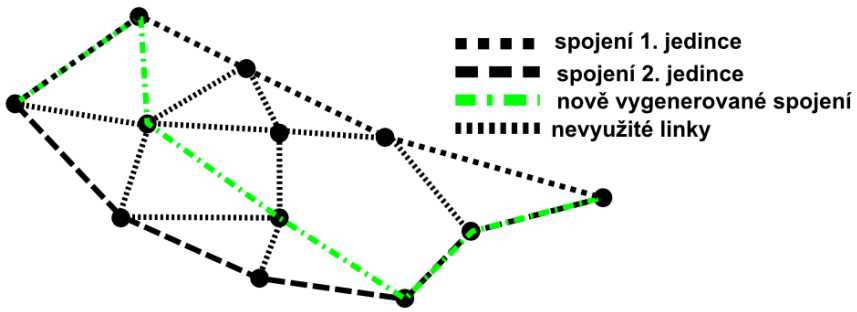
- Podle ceny linky c_{ij}
- Podle zpoždění linky d_{ij}

Cenu linky jsem zvolil proto, že cílem je nalézt nejlevnější řešení, díky tomuto kritériu může algoritmus nalézt nejlevnější cesty. Kritérium nejmenšího zpoždění jsem zvolil proto, že pro některé transporty je určeno maximální povolené zpoždění a tak algoritmus může nalézt pro tyto transporty trasy s malým zpožděním.

5.1.5 Křížení

Křížení probíhá tak, že se z obou jedinců bere postupně směrování pro každý transport a z prvního směrování se použije náhodný počet linek ze začátku (i jen první a minimálně poslední se nepoužije), z druhého směrování se použije náhodný počet linek od konce (i jen poslední a první linka se určitě nepoužije). Tím vznikne začátek a konec nového směrování pro daný transport a prostřední část je vygenerována Dijkstrovým algoritmem s náhodným kritériem určení délky hran.

Na obrázku 5.1 je zobrazen příklad křížení dvou směrování. Ze spojení prvního jedince byla použita jedna linka a ze spojení druhého jedince dvě linky.



Obrázek 5.1: křížení dvou směrování

5.1.6 Mutace

Mutaci jsem naimplementoval tak, že v jedinci, který je vybrán pro mutaci, se náhodně vybere jeden transport, a pro něj se nalezne Dijkstrovým algoritmem směrování pro náhodně vybrané kritérium délky hran.

5.1.7 Generování počáteční populace

Generování jedince probíhalo tak, že pro každý transport se vygenerovalo směrování pomocí Dijkstrova algoritmu s náhodným kritériem délky hran. Tímto způsobem byla vygenerována celá populace.

5.1.8 Selekce

Do programu jsem implementoval turnajovou selekci o velikosti turnaje 3. Z populace se náhodně vyberou 3 jedinci (dokonce se může vybrat i ten samý) a jedinec s nejlepší fitness je vybrán.

5.1.9 Běh algoritmu a nahrazovací strategie

Po vygenerování počáteční populace začne evoluční algoritmus generovat další generace. Nejlepší jedinec v každé generaci je přesunut do další generace beze změny. Tím je zajištěno, že nejlepší řešení bude zachováno. Ostatní jedinci jsou vytvořeni tak, že z původní generace jsou pomocí dvou turnajů vybráni 2 jedinci. Pokud jsou stejní, opakuje se turnaj dokud nebudou vybráni různí jedinci. Tito jedinci jsou navzájem zkříženi a každý z potomků je s pravděpodobností 10 % zmutován. Potom jsou potomci převedeni do další generace. Protože populace má sudý počet členů, vždy jsou vygenerováni 2 potomci a nejlepší jedinec na začátku je převeden do nové generace, stane se na konci generování populace, že zbydou 2 jedinci a pouze jedno místo v populaci. To je vyřešeno tak, že se vezme první vygenerovaný potomek a druhý je nepoužit.

5.1.10 Úpravy algoritmu

Po implementaci výše uvedených metod jsem zjistil, že algoritmus nekonverguje k řešení splňující omezení ani po mnoha generacích. Algoritmus překračoval kapacitu některých linek. Problém jsem zkoušel řešit tak, že jsem pro Dijkstrův algoritmus přidával další kritéria optimality, např. poměr cena kapacita $\frac{c_{ij}}{d_{ij}}$ a kritérium použití nejmenšího počtu linek (každá linka je ohodnocena hodnotou 1). U některých testovacích sad (s vysokou hustotou linek) tato úprava snížila výrazně počet přetížených linek avšak platné řešení stále nebylo nalezeno.

Další úpravu, kterou jsem se rozhodl provést byl způsob generování počáteční populace. Jedinci jsou generováni následovně:

1. pro každý transport je vybráno náhodně kritérium optimality

2. Dijkstrovým algoritmem je nalezeno spojení podle tohoto kritéria
3. Pokud nalezené spojení není přímé (využívá víc než jednu linku) je určitá pravděpodobnost, že bude vyhledáno nové spojení, kde není použit jeden uzel použitý v optimálním spojení. Tato pravděpodobnost se určuje následovně:
   (a) Vygeneruje se celé číslo v intervalu $\langle 0;\ \text{počet uzlů} - 1\rangle$
   (b) 0 znamená použití optimální řešení, 1 znamená hledání řešení bez prvního vnitřního uzlu, 2 hledání bez druhého vnitřního uzlu, atd. — krajní uzly odstraňovány nejsou
   (c) v případě nenalezení cesty se pokračuje bodem a s tím, že předchozí číslo se nepoužije

Dále abych zajistil, že optimálně nalezené směrování budou zastoupeny alespoň v některých jedincích v populaci, upravil jsem generování prvních několika jedinců v populaci. Pro každé kritérium optimality program vygeneruje jednoho jedince tak, že směrování pro všechny transporty jsou generovány Dijkstrovým algoritmem podle vybraného kritéria, při generování těchto jedinců nedochází k hledání jiného směrování bez náhodně vybraného uzlu. Zbytek populace je generován tak, že pro každý transport v jedinci je náhodně vybráno kritérium optimality a vygenerováno směrování. Zde se už může uplatnit generování s vynechaným uzlem.

Tyto úpravy však nevedly k nalezení řešení, které by splňovalo omezující kritéria. Proto jsem se rozhodl upravit křížení tak, aby bylo zkříženy směrování pouze pro 10 % vybraných transportů (první implementace křížila všechny směrování). Tato úprava měla snížit destruktivitu křížení a zajistit větší stabilitu jedinců. Touto úpravou jsem také zrychlil běh programu.

Progresivní penalizace Všechny tyto úpravy však výrazně nezlepšily funkčnost evolučního algoritmu. Jako největší slabinou celého evolučního algoritmu se ukázal Dijkstrův algoritmus. Protože je určen pro hledání optimální cesty v grafu, tak navrhovaná spojení využívala pouze některých linek, což vedlo k soustavnému přetěžování těchto linek. A protože veškeré mutace a křížení probíhaly za pomoci tohoto algoritmu, nebylo možné najít jakékoli

řešení, které by nepřetěžovalo tyto linky. Abych donutil Dijkstrův algoritmus používat i jiné linky než původně nalezené, zavedl jsem progresivní penalizaci linek.

Do každého jedince byla uložena informace o penaltě pro každou linku. Penalta se aplikovala tak, že při určování ceny linek pro Dijkstrův algoritmus se cena linky (určená kritériem optimality) vynásobila penaltou. V počáteční populaci se nastavily všechny penalty na 1, tedy žádná penalizace. Na konci každé generace se pro každého jedince spočítalo vytížení linek. Pokud linka byla přetížena, penalta byla zvýšena o 0,1. V případě že linka byla plně vytížena, penalta nebyla změněna a pokud linka byla vytížena méně než její kapacita, penalta byla snížena o 0,1, s podmínkou že minimální velikost penalty musí být 1.

Při křížení dvou jedinců se první vypočítal aritmetický průměr jejich penalt pro všechny linky, na základě něho se provedlo křížení a tento průměr byl uložen jako penalty do obou jedinců.

Tato úprava měla nevýhodu, že její výsledky nemělo smysl ukládat do cache. Proto vytvořená cache už měla smysl pouze pro generování počáteční populace a zbytek výpočtů se prodloužil. Tuto nevýhodu jsem omezil tak, že do doby než je nalezeno řešení splňující omezení je používána progresivní penalizace. Jakmile je však takové řešení nalezeno, penalizace se přestane používat a požadavky na výpočet jsou znovu řešeny pomocí cache.

5.2 POEMS

Při programování POEMS jsem využil zkušeností s předchozím evolučním algoritmem a několik metod, které zlepšovaly efektivitu evolučního algoritmu jsem implementoval i v POEMS např. progresivní penalizaci.

5.2.1 Prototyp

Prototyp využívá stejnou reprezentaci jako mají jedinci v evolučním algoritmu 5.1.1. V prototypu uplatňuji progresivní penalizaci vyvinutou pro evoluční algoritmus. Ta se uplatňuje, pokud prototyp nesplňuje omezující podmínky a penalty linek v prototypu jsou aktualizovány na konci každé iterace

POEMS. Pokud je linka přetížena, je penalta pro ni zvětšena o 0,1, pokud je linka vytížena plně je penalta zachována a pokud je linka vytížena méně, je penalta snížena o 0,1 (nikdy ne pod 1). Na začátku vygenerovaný prototyp má všechny penalty nastaveny na 1.

5.2.2 Akce

V této implementaci, akce obsahuje číslo transportu (transporty v zadání jsou očíslované), jenž je modifikován, a kritérium určení délky hran pro Dijkstrův algoritmus. Pokud je akce aplikována na prototyp (a je aktivní), najde se pomocí Dijkstrova algoritmu nové směrování pro daný transport (nejkratší trasa z výchozího do cílového uzlu) a tím nahradí původní směrování.

5.2.3 Sekvence akcí

Pro účely algoritmu je nutné stanovit počet akcí N v sekvenci. V této úloze se počet akcí vypočítá jako $N = |T| / 20$.

5.2.4 Vlastnosti vnitřního evolučního algoritmu

V popisu evolučního algoritmu budu používat pojmy jako je sekvence akcí a jedinec. Tyto pojmy jsou v tomto kontextu totožné a je možné je zaměňovat. Oproti předchozímu evolučnímu algoritmu již jedinec neznamená reprezentaci řešení, ale způsob modifikace takového řešení.

Fitness

Fitness každé sekvence akce se určuje tak, že se na prototyp aplikuje sekvence akcí a vypočítá se jeho fitness podle (2.5). K fitness se dále přičte penalta (5.1) podle pravidel uvedených v 5.1.3. Jedinou výjimkou je, pokud sekvence akcí prototyp nezmění (výsledek bude využívat stejná směrování jako původní prototyp), bude nastavena fitness na maximální možnou. Tím je dosaženo, že sekvence akcí neměnící prototyp jsou vyřazovány z populace. Tyto sekvence akcí totiž nezhoršují fitness a proto by mohly ovlivnit populaci a znemožnit nalezení sekvencí zlepšující prototyp a také tímto tlačíme evoluční algoritmus aby generoval sekvence, které prototyp mění.

Detekce změny prototypu probíhá tak, že se aplikuje sekvence akcí a výsledek se porovná s prototypem. Pokud jsou všechna směrování prototypu a výsledku totožná, jsou tyto dvě řešení stejná. Tím je zajištěno, že i pokud sekvence akcí obsahuje inverzní operace (nejenom NOP), bude správně vyhodnocena fitness.

Křížení

Sekvence akcí se kříží tak, že vzniknou 2 nové sekvence akcí a každá z nich bude obsahovat náhodně akce z první nebo druhé sekvence. To je dosaženo tak, že všechny akce jsou přesunuty do jednoho seznamu a z něho je vybráno náhodně N akcí pro prvního jedince a N akcí pro druhého jedince. Tím je také zajištěno změna pořadí jednotlivých akcí v jedincích.

Mutace

Mutace sekvence akcí probíhá následovně:

1. ze sekvence se vybírají jednotlivé akce
2. s pravděpodobností 20 % se vybraná akce zmutuje
3. pokračuje se bodem *1* dokud zbývají ještě nevybrané akce
4. pokud za celý průchod nebyla zmutována ani jedna akce, vybere se jedna náhodně a ta se zmutuje — tím je zajištěno, že mutace sekvence akcí sekvenci změní

Postup mutace jedné akce:

1. pokud je akce označena jako neaktivní (NOP) je nastavena jako aktivní, tím je mutace hotova
2. pokud je mutace aktivní s pravděpodobností 30 % se nastaví jako neaktivní, jinak se pokračuje bodem *3*
3. vygeneruje se nová aktivní akce, pokud je stejná jako původní akce, bod *3* se opakuje

Opravy

Pro efektivní běh algoritmu jsem vytvořil opravnou funkci. Tato funkce se aplikuje na jedince po křížení, mutaci a při generování počáteční populace. Oprava má za cíl:

- Odstranit duplicitní akce. Z vlastností akce vyplývá, že pokud je jedna akce aplikována na jedince opakovaně má stejný efekt jako pokud je aplikována pouze jednou. Při této opravě jsou v jedinci všechny opakující se akce nastaveny na NOP kromě první v pořadí. Tím je také zajištěno správné zařazování do šuplíků. Postup opravy:

  1. seznam akcí se prochází v cyklu od začátku do konce, v každém cyklu je vybrána jedna akce
  2. pokud je vybraná akce NOP pokračuje se bodem *1* s další akcí
  3. pokud se vyskytuje stejná akce dále v seznamu, jsou všechny její výskyty označeny jako NOP
  4. pokračuje se bodem *1*

- Zajistit přítomnost akce upravující přetíženou linku. Pokud prototyp obsahuje přetížené linky a sekvence akcí nemodifikuje žádný transport v prototypu je jasné, že nepřinese žádné zlepšení. Oprava probíhá tak, že náhodně vybraná akce v sekvenci se nahradí akcí, měnící transport využívající přetíženou linku.

Generování počáteční populace

Počáteční populace je generována ve dvou fázích. V první jsou vygenerováni jedinci a v druhé jsou opraveni tak aby splňovali podmínky dané šuplíky. Už ve fázi generování jsou jedinci přiřazeni jednotlivým šuplíkům. Pro každého jedince jsou vygenerovány akce a pro každou akci platí, že s pravděpodobností 75 % je NOP.

Oprava je v tomto případě složitější:

1. Jedinec je první opraven základní opravou — jsou odstraněny duplicitní akce a případně přidána akce upravující transport vedený přes přetíženou linku.

2. Pokud má jedinec dostatečný počet aktivních akcí (podle čísla šuplíku) je oprava hotova.

3. Seznam akcí se prochází od začátku do konce a na každé akci se provádí testy:

   (a) pokud vybraná akce není NOP vybere se další akce ze seznamu a pokračuje se bodem *a*

   (b) pokud je vybraná akce NOP, příznak NOP je zrušen

   (c) pokud takto upravená akce se už vyskytuje v seznamu akcí (na jiné pozici než současné), je vygenerována nová aktivní akce, pokračuje se bodem *c*

   (d) v případě dostatečného počtu aktivních akcí je oprava hotova, jinak se pokračuje bodem a na další akci v pořadí

Selekce

Výběr jedinců probíhá turnajovou selekcí o velikosti 2. Nejdříve se náhodně zvolí šuplík, ze kterého budou jedinci vybíráni. Pak se ze šuplíku náhodně vyberou 2 jedinci a ten s lepší fitness vyhrává.

5.2.5 Běh algoritmu a nahrazovací strategie

Průběh lze shrnout následujícími body:

1. Na začátku algoritmu se vygeneruje prototyp. Pro každý transport se vygeneruje směrování pomocí Dijkstrova algoritmu s náhodným kritériem určování délky linek.

2. Provede se vygenerování počáteční populace sekvencí akcí (tak jak je uvedeno na straně 23 Generování počáteční populace). Následně se populace zlepšuje:

   (a) V populaci jsou pomocí turnajů vybráni 2 jedinci (každý může být z jiného šuplíku). S pravděpodobností 75% jsou zkříženi, jinak jsou zmutováni.

(b) Vzniklí jedinci jsou pak zařazováni do populace. U jedince je spočítán počet aktivních akcí x. Postupně jsou procházeny šuplíky s $n \in \{1..x\}$ a procházeny jsou v neklesajícím pořadí (první je prohledáván šuplík s číslem 1 a nakonec šuplík s číslem x). Šuplíky s vyšší číslem je zbytečné procházet protože jedinec do nich patřit nemůže. Při prohledávání šuplíku se porovnává fitness nového jedince s fitness jedinců v šuplíku. Pokud je fitness nového jedince lepší nebo rovna fitness stávajícího jedince, je původní jedinec nahrazen a průchod je ukončen. Pokud jedinec měl horší fitness než všichni jedinci v příslušných šuplících je zahozen.

(c) opakovat od bodu *a* stanovený počet generací evolučního algoritmu

3. V populaci je vybrán nejlepší jedinec, aplikován na prototyp a tím je vytvořen kandidát. Pokud kandidát má lepší nebo stejnou fitness než původní prototyp, je prototyp kandidátem nahrazen.

4. Opakovat od bodu *2* stanovený počet iterací

Kapitola 6

Experimenty

Po implementaci obou algoritmů jsem tyto algoritmy porovnal mezi sebou a dále s výsledky uvedené v [1]. Protože evoluční algoritmy nezaručují při opakovaném spuštění stejný výsledek, každý z nich jsem spustil 10x na každém z problémů abych dostal přesnější výsledky a omezil možnost výrazně odlišných výsledků. Běžné algoritmy se porovnávají podle časové nebo paměťové složitosti. U evolučních algoritmů toto není vhodné, protože nezaručují optimální výsledek a informace o složitosti tak není dostatečně vypovídající. Evoluční algoritmy se proto porovnávají jiným způsobem. Každý z evolučních algoritmů během svého běhu mnohokrát počítá ohodnocení (fitness) jedinců, které vygeneroval v průběhu výpočtu. Tyto algoritmy se proto porovnávají tak, že se omezí počet výpočtů ohodnocení a porovnají se výsledky algoritmů pokud všechny mají stanovený počet ohodnocení. Protože POEMS využívá evolučního algoritmu ke šlechtění sekvencí, můžeme počítat počet ohodnocení i u POEMS.

Pro porovnání implementovaných algoritmů jsem stanovil počet ohodnocení na 500 000. Tomu jsem uzpůsobil nastavení jednotlivých algoritmů, aby tohoto počtu dosáhly.

Z dostupných testovacích sad jsem vybral 20. Vybral jsem je na problémech, kde jiné porovnávané metody buď nenalezly řešení a nebo měly velký rozestup ceny mezi nejlepším řešením a nejmenší teoretickou cenou. Problémy jsem také vybral s ohledem na jejich velikost, abych mohl porovnat, jak budou algoritmy reagovat na změnu velikosti zadání. Prvních deset problémů

má zadanou síť o 25 uzlech a dalších deset síť o 100 uzlech. Všechny vybrané experimenty měly maximální celkové zpoždění pro všechny transporty $D = 12\,500$.

6.1 Evoluční algoritmus

Použitá konfigurace

- počet jedinců v populaci: 400
- počet generací: 1 250
- pravděpodobnost mutace 10 %
- selekce - turnaj velikosti 3
- nahrazovací strategie: v každé generaci zůstane zachován nejlepší jedinec, ostatní jsou nahrazeni novými jedinci
- kritéria určování délek hran pro Dijkstrův algoritmus:
  - nejmenší cena
  - nejmenší zpoždění
  - nejmenší počet linek

Při testech evolučního algoritmu docházelo v průběhu generací ke ztrátě variability v populaci a během generací klesl počet unikátních jedinců řádově na desítky. Velikost populace byla stále 400 jedinců a tak byla populace naplněna kopiemi jen několika jedinců. Abych zvýšil variabilitu jedinců v populaci, rozhodl jsem se změnit pravděpodobnost mutace z původních 1 % na 10 %.

6.2 POEMS

U evolučního algoritmu jsem konfiguraci stanovil fixně, nezávisle na řešeném problému. U POEMS jsem se rozhodl upravovat konfiguraci v závislosti

na problému. Konfigurace se mění podle toho, kolik transportů je zadáno. Počet šuplíků a tedy i maximální počet akcí v sekvenci jsem stanovil jako 1/20 počtu transportů. Ostatní hodnoty dopočítávám podle velikosti šuplíku abych dosáhl stanovených 500 000 ohodnocení.

Použitá konfigurace

- velikost šuplíku $V\check{S} = 20$
- počet šuplíků $P\check{S} = \frac{|T|}{20}$
- velikost populace $VP = V\check{S} \cdot P\check{S}$
- počet generací $\frac{9}{2} \cdot VP$
- počet iterací $\frac{500\,000}{10 \cdot VP}$
- pravděpodobnost křížení 75%, jinak mutace
- selekce - turnaj velikosti 2
- pravděpodobnost mutace jednotlivé akce 20%
- pravděpodobnost NOP akce při generování sekvence akcí 75%
- kritéria určování délek hran pro Dijkstrův algoritmus:
  - nejmenší cena
  - nejmenší zpoždění
  - nejmenší počet linek

6.3 Výsledky experimentů

Následuje tabulka výsledků 6.1, kterou jsem sestavil z 20 experimentů. První dva sloupce dávají základní informaci o řešeném problému: počet uzlů v síti |V| a počet požadovaných transportů |T|. Sloupec síť specifikuje soubory obsahující konfiguraci problému. V testovací sadě je každý problém uložen

ve dvou souborech. Konfigurace sítě a seznam transportů. Ke každému seznamu transportů patří 3 konfigurace sítě. Například k seznamu transportů 01_F_25_TRANS patří sítě 01_F_25_3N, 01_F_25_5N a 01_F_25_10N. Díky jednotnému pojmenování lze jednoznačně určit název souboru s transporty podle názvu souboru s konfigurací sítě. Proto zmenšení velikosti tabulky uvádím pouze název souboru se sítí. Každý soubor s transporty obsahuje 1 000 transportů. Pokud jich úloha vyžaduje méně, např.: 100, uvažuje se prvních 100 transportů uvedených v souboru.

Další sloupce ukazují nejlepší nalezené hodnoty při řešení daných problémů a mají následující označení:

CLP celočíselné lineární programování

CG Column Generation

LD Lagrangean Decomposition

EA evoluční algoritmus

POEMS Prototype Optimisation with Evolved Improvement Steps

Protože jsem evoluční algoritmus a POEMS spouštěl na každém problému 10 krát, neuvádím ve sloupcích EA a POEMS nejlepší dosažené hodnoty, ale jejich průměr. V každém řádku je pro přehlednost tučně vyznačen výsledek algoritmu, který na daném problému dosáhl nejlepšího výsledku. Výjimkou jsou dva poslední řádky, kde i přesto že POEMS dosáhl menšího průměru, tak evoluční algoritmus měl nejlepší řešení s menší cenou než nejlepší řešení algoritmu POEMS. Proto jsem nemohl rozhodnout, který z algoritmů je lepší a vyznačil jsem oba. V úvodu jsem uvedl, že CLP vždy najde optimální řešení. Na jednom řádku tabulky je uvedena hodnota CLP avšak není nejlepší. Tento výsledek je způsoben tím, protože výpočet CLP trval příliš dlouho a byl předčasně přerušen. Výsledek proto ukazuje nejlepší nalezené řešení v průběhu výpočtu.

|V|	|T|	síť	CLP	CG	LD	EA	POEMS
25	100	01_F_25_10N	nenalezeno	2259	2014	**1614,3**	1697,9
25	100	01_G_25_5N	**367**	379	397	434,1	433,5
25	100	01_F_25_5N	nenalezeno	**1365**	1803	1461,8	1552,6
25	100	00_G_25_3N	**414**	431	434	445,6	443,1
25	100	00_F_25_10N	nenalezeno	2097	1888	**1520,1**	1604,0
25	200	02_F_25_5N	nenalezeno	3302	2971	**2310,2**	2332,3
25	200	00_G_25_3N	**754**	777	792	806,1	793,5
25	200	01_G_25_10N	**552**	571	572	702,5	697,3
25	200	01_F_25_5N	nenalezeno	3780	3082	2558,7	**2521,8**
25	200	02_F_25_10N	nenalezeno	3842	3156	**2455,7**	2545,9
100	100	00_G_100_3N	**720**	773	808	765,5	761,9
100	100	00_G_100_10N	615	**503**	513	529,4	526,4
100	100	02_G_100_10N	nenalezeno	666	**532**	557,7	554,0
100	100	00_F_100_3N	nenalezeno	4941	4218	**3011,5**	3203,2
100	100	01_F_100_3N	nenalezeno	4921	4338	**3106,7**	3410,9
100	200	02_G_100_5N	nenalezeno	1382	1287	1227,4	**1219,4**
100	200	00_F_100_10N	nenalezeno	7294	6192	**4394,5**	4537,6
100	200	02_F_100_3N	nenalezeno	7983	6934	**5103,1**	5194,7
100	200	02_G_100_3N	nenalezeno	1520	1499	**1404,6**	**1400,6**
100	200	01_G_100_3N	nenalezeno	1363	1460	**1369,0**	**1362,5**

Tabulka 6.1: Tabulka výsledků

Z uvedené tabulky nelze jednoznačně určit, který z algoritmů je nejlepší. Nejlepším by mohl být označen evoluční algoritmus, který nalezl nejlepší řešení v polovině experimentů. V případě, že došlo k nalezení optimálního řešení pomocí CLP, byl výsledek nalezený pomocí algoritmů, které jsem implementoval, o 5 až 18 % horší. Z tabulky je však patrné, že evoluční algoritmus a POEMS jsou vhodnější na řešení větších problémů než porovnávané algoritmy.

Kapitola 7

Závěr

Algoritmy, které jsem implementoval, byly úspěšně schopny řešit tento typ problému. Při porovnání s ostatními metodami a mezi sebou nelze jednoznačně určit nejlepší algoritmus. Z výsledků plyne, že na každý problém lépe funguje jiná metoda.

Největší slabinou obou algoritmů se ukázal Dijkstrův algoritmus. Umožňuje sice hledat optimální cesty v grafu, avšak díky tomu značně omezuje prohledávaný prostor řešení a u některých zadání prakticky znemožňuje nalézt řešení splňující omezení. Progresivní penalizace sice umožnila nalézt platné řešení, avšak opravuje pouze nevhodné použití Dijkstrova algoritmu. Řešením by bylo použít jiný algoritmus na hledání cesty v grafu, který by sice nenalézal vždy optimální cestu, ale díky tomu by umožnil prohledat větší prostor řešení.

Obsah přiloženého CD

- elektronická verze bakalářské práce
- soubory testovacích sad použité pro srovnání algoritmů
- zkompilovaný program a ukázkový skript jeho spuštění
- zdrojový kód programu
- výsledky experimentů vygenerovaných programem a srovnávací tabulka algoritmů v pdf

Literatura

[1] Andreas M. Chwatal, Nina Musil, Gunther R. Raidl. Solving a Multi-Constrained Network Design Problem by Lagrangean Decomposition and Column Generation.

[2] D. E. Goldberg: Genetic Algorithms in Search, Optimization, and Machine Learning, Addison- Wesley, 1989.

[3] Z. Michalewicz: Genetic Algorithms + Data Structures = Evolution Programs, Springer, 1998.

[4] Sequential circuit test generation in a genetic algorithm framework GSG Elizabeth, M Rudnick, JH Patel, TM Niermann - IEEE-CAS: Circuits and Systems, ACM Press, New York, NY, 1994

[5] Kubalík, J. and Faigl, J.: Iterative Prototype Optimisation with Evolved Improvement Steps. In Genetic Programming, Proceedings of 9th European Conference, EuroGP 2006. Heidelberg: Springer, p. 154-165. ISBN 3-540-33143-3, 2006.

[6] J. Kubalík: Solving the Sorting Network Problem Using Iterative Optimization with Evolved Hypermutations. Prijato na konferenci GECCO 2009, July 8–12, Canada, 2009.

[7] J. Kubalík: Solving Multiple Sequence Alignment Problem Using Prototype Optimization with Evolved Improvement Steps, Ve sborniku konference International Conference on Adaptive and Natural Computing Algorithms, ICANNGA '09, Kuopio, Finland, strany 183-192, 2009.