VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

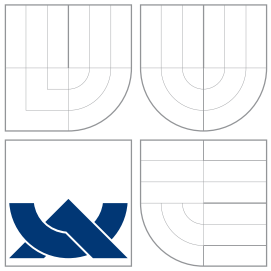
ŘEŠENÍ OPTIMALIZAČNÍCH ÚLOH INSPIROVANÉ ŽIVÝMI ORGANISMY

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

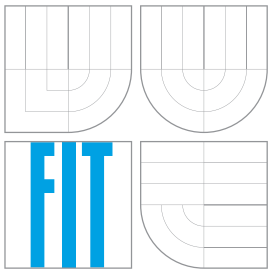
AUTOR PRÁCE
AUTHOR

Bc. MILOŠ POPEK

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

ŘEŠENÍ OPTIMALIZAČNÍCH ÚLOH INSPIROVANÉ ŽIVÝMI ORGANISMY

SOLVING OF OPTIMISATION TASKS INSPIRED BY LIVING ORGANISMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MILOŠ POPEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. DAVID MARTINEK

BRNO 2010

Abstrakt

S řešením optimalizačních problémů se setkáváme v každodenním životě, kdy se snažíme zadané úkony provést nejlepším možným způsobem. Ant Colony Optimization je algoritmus inspirovaný chováním mravenců při hledání potravy. Ant Colony Optimization se úspěšně používá na optimalizační úlohy, na které by nebylo možné klasické optimalizační metody použít. Genetický algoritmus je inspirován přenosem genetické informace při křížení. Stejně jako ACO algoritmus se používá pro řešení optimalizačních úloh. Výsledkem mé diplomové práce je vytvořený simulátor pro řešení zvolených optimalizačních úloh pomocí ACO algoritmu a GA a porovnání dosažených výsledků na implementovaných úlohách.

Klíčová slova

Ant Colony Optimization, genetický algoritmus, optimalizace, problém obchodního cestujícího, problém rozvržení úloh na dílně, problém pokrytí množin, simulace, samoorganizace

Abstract

We meet with solving of optimization problems every day, when we try to do our tasks in the best way. An Ant Colony Optimization is an algorithm inspired by behavior of ants seeking a source of food. The Ant Colony Optimization is successfuly using on optimization tasks, on which is not possible to use a classical optimization methods. A Genetic Algorithm is inspired by transmision of a genetic information during crossover. The Genetic Algorithm is used for solving optimization tasks like the ACO algorithm. The result of my master's thesis is created simulator for solving choosen optimization tasks by the ACO algorithm and the Genetic Algorithm and a comparison of gained results on implemented tasks.

Keywords

Ant Colony Optimization, Genetic Algorithm, optimization, Traveling Salesman Problem, Job Shop Scheduling Problem, Set Covering Problem, simulation, self-organization

Citace

Miloš Popek: Řešení optimalizačních úloh inspirované
živými organismy, diplomová práce, Brno, FIT VUT v Brně, 2010

Řešení optimalizačních úloh inspirované živými organismy

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Davida Martinka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Miloš Popek
26.5.2010

Poděkování

Děkuji Ing. Davidovi Martinkovi za odborné vedení práce, za poskytování rad a materiálů.

Obsah

Kapitola 1

Úvod

S řešením optimalizačních problémů se setkáváme v každodenním životě, kdy se snažíme zadané úkony provést nejlepším možným způsobem. Jestliže chceme řešit optimalizační problém algoritmicky, pak musíme vytvořit odpovídající matematický model k danému problému. Kf připravenému matematickému modelu optimalizačního problému navrhneme jeho algoritmické řešení. Algoritmus řešení optimalizačního problému může být exaktní, vycházející například z lineárního, nebo nelineárního programování, nebo numerických metod[12].

Ne vždy lze exaktní algoritmus na řešení problému použít. Stavový prostor daného problému může být příliš veliký pro řešení exaktním algoritmem, nebo pro daný problém exaktní algoritmus neznáme. Pak přicházejí na řadu algoritmy založené na stochastickém rozhodování, jako jsou například genetické algoritmy(GA), optimalizace pomocí mravenčích kolonií(ACO), simulované žíhání, metoda Monte-Carlo[14]. Nevýhodou při použití těchto algoritmů je, že produkují v zadaném výpočetním čase pseudooptimální výsledky. Tyto pseuooptimální výsledky jsou ale dostatečně kvalitní již po relativně malém počtu iterací.

Aplikací výsledků a poznatků, dosažených při procesu optimalizace, do pracovních postupů lze ušetřit nemalé vstupní zdroje nebo podstatně zkrátit dobu výrobního procesu, a tím snížit celkové výrobní náklady. Proto dochází v oblasti optimalizací k intenzivnímu výzkumu. Optimalizace pomocí mravenčích kolonií(ACO) se ukazuje jako stabilní metoda, která dosahuje velmi kvalitních výsledků v časově proměnlivém prostředí. Použití této metody při optimalizaci kombinatorických optimalizačních úloh je předmětem neustálého výzkumu i hlavním tématem této práce.

V následující kapitole jsou popsány základní cíle této diplomové práce a velmi stručná charakteristika současného stavu řešené problematiky. V další kapitole je formálně popsána optimalizační úloha, definovány jednotlivé fáze optimalizačního procesu a popis jednotlivých typů optimalizačních algoritmů. Kapitola Ant Colony Optimization popisuje chování mravenčích kolonií při získávání potravy a rozebírá pojmy emergence, samoorganizace a rojová inteligence. Dále je zde rozebrán optimalizační algoritmus Ant Colony Optimization(ACO) a jeho použití při řešení optimalizačních úloh. V kapitole Genetický algoritmus je popsán princip fungování genetického algoritmu a je zde také detailně rozebráno použití genetického algoritmu při řešení optimalizačních úloh. V dalších kapitolách jsou popsány jednotlivé optimalizační úlohy a postup, který bude použit při jejich řešení pomocí ACO a GA. V kapitole Implementace je popsána struktura celého programu simulátoru i detailně vysvětlen obsah a funkčnost jednotlivých tříd. Závěrečná kapitola shrnuje výsledky provedených simulací a popisuje nalezené skutečnosti, které vyplývají ze simulací. Dále jsou zde uvedena možná rozšíření a vylepšení této diplomové práce.

Kapitola 2

Formulace cíle a charakteristika současného stavu

Motivací a inspirací pro tuto práci je příroda kolem nás. Můžeme sledovat nepřeberné množství živočichů, rostlin a jejich nejrozličnějších způsobů jak přežít. S trochou nadsázky můžeme planetu Zemi přirovnat k simulátoru, který byl inicializován před 4,6 miliardami let(vznik planety Země). První zajímavé výsledky začal dávat před 3,85 miliardami let(vznik života na Zemi). Jeho hodnotící funkce se nazývá evoluce.

2.1 Formulace cíle této diplomové práce

Hlavním cílem této práce je implementace algoritmu Ant Colony Optimization (dále jen ACO) a zhodnocení výsledků dosažených při řešení navržených optimalizačních úloh. Celá práce se skládá z těchto dílčích cílů:

- Návrh a implementace jednoduchého simulátoru včetně implementace ACO algoritmu. Simulátor bude řešit optimalizační úlohy pomocí ACO algoritmu. Průběh simulace(řešení optimalizační úlohy pomocí ACO algoritmu) bude zobrazen jednoduchým grafickým výstupem. Zadání optimalizační úlohy bude načítáno ze souboru. Simulátor bude umožňovat nastavení parametrů simulace a jednotlivých parametrů ACO algoritmu.

- Návrh a implementace optimalizačních úloh řešených pomocí ACO algoritmu. Vytvoření matematických modelů zvolených optimalizačních úloh a jejich implementace do prostředí simulátoru, tak aby mohly být řešeny pomocí implementovaného ACO algoritmu.

- Zhodnocení dosažených výsledků zvolených optimalizačních úloh při použití ACO algoritmu. Zhodnocení použitelnosti ACO algoritmu při řešení zvolených optimalizačních úloh. Zhodnocení toho, jak jednotlivé parametry ACO algoritmu ovlivňují kvalitu nalezeného řešení optimalizační úlohy.

- Implementace genetického algoritmu (dále jen GA) do simulátoru. Simulátor bude umožňovat řešení optimalizačních úloh kromě ACO algoritmu i pomocí GA.

- Zhodnocení dosažených výsledků zvolených optimalizačních úloh při použití GA a porovnání s výsledky ACO algoritmu. Zhodnocení použitelnosti GA při řešení zvolených

optimalizačních úloh. Porovnání kvality a rychlosti nalezených řešení optimalizačních úloh při použití ACO algoritmu a GA.

2.2 Charakteristika současného stavu

V roce 1992 zveřejnil Marco Dorigo ACO algoritmus inspirovaný chováním mravenců při hledání potravy[5]. Algoritmus se stal objektem dalšího zkoumání a dočkal se několika variant a vylepšení. ACO algoritmus je v současné době úspěšně používán v různých odvětvích:

- Směrování packetů v síti. Díky adaptivnosti a optimálnosti dosahuje ACO algoritmus nízké ztrátovosti packetů[8].
- Optimalizace výrobního procesu. Lepším rozvržením úloh na jednotlivých strojích snižuje výrobní čas, a tím zvyšuje kapacitu výrobní linky [3].
- Výrobu mikročipů. Nalezením optimální cesty pro propojení jednotlivých částí mikročipu se sníží výrobní cena [10].
- Logistika. Nalezení optimální trasy pro rozvoz surovin a zboží [8].
- Plánování posádek na leteckých linkách [9].

Kapitola 3

Optimalizace

Optimalizace je odvětví matematiky, které se snaží o nalezení minimálních či maximálních hodnot zkoumaných funkcí při daných omezujících podmínkách[15]. Formálně pak můžeme optimalizační úlohu na minimalizaci zapsat takto:

$$\begin{array}{ll} \text{minimalizuj} & f_0(x) \\ \text{vzhledem k} & f_i(x) \leq b_i, i = 1, ..., m. \end{array}$$

Kde vektor $x = (x_1, ..., x_n)$ je proměnná optimalizační úlohy. Funkce $f_0 : R^n \rightarrow R$ je účelová funkce úlohy a funkce $f_i : R^n \rightarrow R, i = 1, ..., m$, jsou omezující funkce a konstanty $b_1, ..., b_m$ jsou hranice pro omezující funkce. Vektor x^* se nazývá optimálním řešením, jestliže hodnota účelové funkce je minimální mezi všemi možnými vektory splňujícími omezující podmínky:

$$\begin{array}{ll} \text{pro všechny z} & f_1(z) \leq b_1, ..., f_m(z) \leq b_m \\ \text{dostáváme} & f_0(z) \geq f_0(x^*) \end{array}$$

3.1 Optimalizační proces

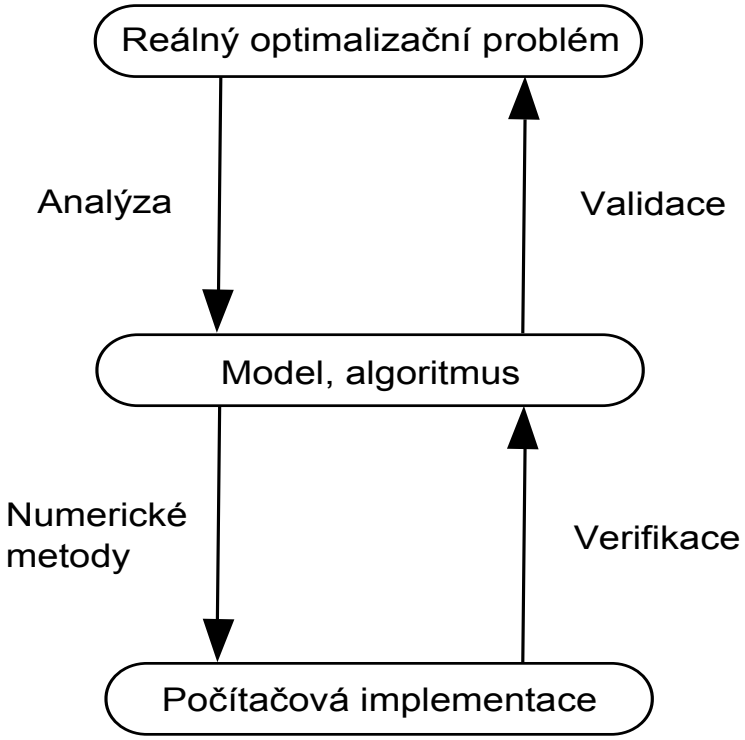
Optimalizační proces se skládá z několika fází (obr. 3.1)[4]. Zpravidla začíná zadáním optimalizačního problému z reálného světa. Zadání problému je obvykle pouze slovní a dosti obecné. Výsledným výstupem této fáze je slovně specifikovaný optimalizační problém s požadovaným cílem.

Analýzou dané problematiky a přesnou specifikací podmínek optimalizovaného problému vytvoříme matematický model, který obsahuje všechny podstatné vlastnosti zkoumaného problému, a navrhneme algoritmus řešení problému v rámci modelu. Výstupem této fáze je navržený matematický model odrážející podstatné rysy z reálného světa včetně algoritmů k řešení tohoto modelu.

Dalším krokem je vytvoření počítačové implementace navrženého modelu a algoritmu. V této fázi se řeší problémy s rychlostí implementovaných algoritmů nebo přesností zaokrouhlování. Výstupem této fáze je vytvořený program ve zvoleném programovacím jazyce, který reprezentuje navržený matematický model a realizuje navržené algoritmy.

Verifikace je proces, při kterém ověřujeme, zda implementované řešení odpovídá navrženému modelu. V této fázi se testují výstupy programu na předem definované testovací vstupy. Výsledkem této fáze je ujištění, že se implementovaný program plně shoduje s navrženým matematickým modelem a navrženými algoritmy.

Proces validace je založen na kontrole, zda výsledky produkované navrženým modelem odpovídají výsledkům z reálného světa. Výsledkem této závěrečné fáze je rozhodnutí, zda a do jaké míry model produkuje reálné výsledky a zda tak může být program nasazen na řešení problémů z reálného světa.



Obrázek 3.1: Optimalizační proces

3.2 Optimalizační algoritmus

Jestliže máme vytvořený matematický model optimalizační úlohy, hledáme algoritmus, který by dokázal daný model vyřešit a nalézt hledané řešení optimalizační úlohy. Optimalizační algoritmy se dělí podle svého přístupu k determinismu:

- Deterministické algoritmy - Založeny na exaktním přístupu k řešenému problému. Vychází z prohledávání stavového prostoru možných řešení. Produkují opakovatelná řešení (stále stejná).
- Stochastické algoritmy - Založeny na pravděpodobnostním přístupu k řešenému problému. Dochází k výběru možného řešení na základě pravděpodobnosti. Produkují neopakovatelná (těžko opakovatelná) řešení.
- Smíšené algoritmy - Založeny na jisté míře pravděpodobnostního výběru. Míra pravděpodobnosti zvolení určitého řešení odpovídá míře vhodnosti daného řešení. Produkují poměrně dobře opakovatelná řešení.

Kromě klasických deterministických optimalizačních přístupů, jako jsou lineární a nelineární programování nebo dynamické programování, existují metody založené na nedeterministické složce.

Mezi metody založené pouze na nedeterministické složce (stochastické metody) patří metoda Monte-Carlo nebo simulované žíhání. Stochastické metody nedávají vždy optimální řešení a za daný časový úsek prohledají oproti deterministickým metodám menší stavový prostor možných řešení. Jejich uplatnění se ale najde u úloh, kde není žádný deterministický algoritmus znám nebo je stavový prostor možných řešení příliš rozsáhlý a řešení pomocí deterministických algoritmů by trvalo příliš dlouho.

Spojením deterministického a nedeterministického přístupu vznikly metody smíšené. Velkou skupinu mezi smíšenými algoritmy představují evoluční algoritmy. Evoluční algoritmy označují skupinu algoritmů, které jsou inspirovány přírodními procesy evoluce:

- Reprodukce - Inspirace pro genetické algoritmy (GA). Nová generece potomků vniká křížením otcovských a mateřských genů.
- Rojová inteligence u mravenců - Základ pro Ant Colony Optimization (ACO). Decentralizovaný samoorganizující systém.
- Přežití nejsilnějších jedinců - Pro výběr rodičů u genetických algoritmů se volí z nejlepších jedinců reprezentujících nejlepší stávající řešení.

Všechny evoluční algoritmy jsou založeny na principu uchovávání nejlepšího řešení, které bylo doposud nalezeno. Na počátku je náhodně zvoleno několik řešení, nejlepší z nich se vybere a je v další iteraci hledání nových řešení mírně upřednostněno. Velikost míry, jak je dané řešení upřednostněno, je vyjádřením toho, jak je dané řešení kvalitní. K tomu, aby nalezená řešení mohla být vyhodnocena a mohlo být z nich vybráno nejkvalitnější řešení, je potřeba mít účelovou funkci, která jednotlivá řešení ohodnotí. Účelová funkce vychází ze specifikace dané optimalizační úlohy a přiděluje nejvyšší ohodnocení řešením, která nejlépe splňují požadavky úlohy. Vlastnost, že evoluční algoritmy opouštějí horší řešení a adaptují se na lepší, způsobuje, že jsou evoluční algoritmy robustní - nezávislé na počátečních podmínkách a dokáží nalézt velmi kvalitní řešení i v čase proměnlivém prostředí. Další kladnou vlastností evolučních algoritmů je, že za poměrně malého počtu ohodnocení prostřednictvím účelové funkce jsou schopny nalézt relativně kvalitní řešení. Evoluční algoritmy se úspěšně používají na náročných optimalizačních úlohách, které se nedají ani jinými metodami řešit.

Kapitola 4

Ant Colony Optimization (ACO)

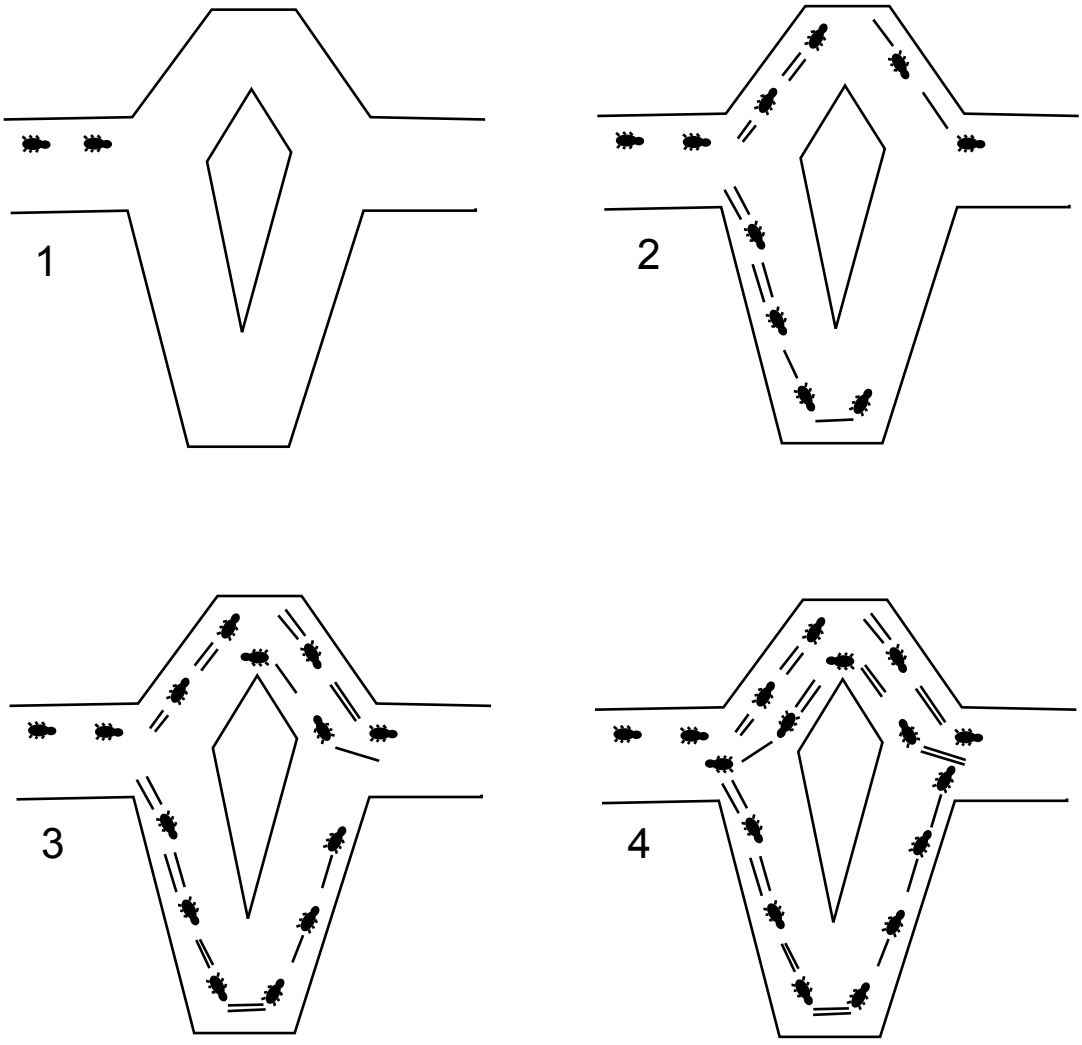
Kolonie mravenců dokáže nalézt nejkratší cestu mezi mraveništěm a zdrojem potravy, aniž by to kterýkoliv mravenec měl jako záměr. Každý mravenec se řídí podle jednoduchých pravidel. Při cestě od zdroje potravy zpět do mraveniště pokládá mravenec chemickou látku zvanou feromon a zanechává tak za sebou feromonovou stopu. Mravenci směřující z mraveniště za potravou následují cesty s větší koncentrací feromonu. Vzniká pozitivní zpětná vazba, která napomůže k objevení nejkratší cesty(na principu emergence) mezi mraveništěm a zdrojem potravy.

4.1 Mravenčí kolonie

Mravenci podobně jako včely žijí v koloniích, a proto patří do skupiny tzv. sociálního hmyzu. Z pohledu optimalizačních metod je na těchto sociálně žijících koloniích nejzajímavější jejich proces shánění potravy. Včely, které se vrátí s medem do úlu, hlásí pomocí speciálního tance ostatním včelám polohu, vzdálenost a druh potravy. Tímto chováním je inspirován algoritmus Bee Colony Algorithm [6], který lze použít na řešení kombinatorických úloh, jako jsou Travelling Salesman Problem (TSP) nebo Job Shop Scheduling Problem (JSP). Z chování při shánění potravy u mravenců vychází algoritmus Ant Colony Optimization (ACO). Mravenec nesoucí potravu do mraveniště za sebou zanechává feromonovou stopu. Ostatní mravenci následují feromonovou stopu při hledání potravy. Mravenci dovedou rozlišovat mezi intenzitou jednotlivých feromonových stop, a to jim umožňuje zvolit si tu nejvíce atraktivní stopu. Na základě tohoto principu dovedou mravenci nalézt nejkratší cestu mezi mraveništěm a zdrojem potravy. Podstata principu nalezení nejkratší cesty je ukázána na obrázku 4.1:

- Na prvním snímku (obr. 4.1) přicházejí k místu, kde se musí rozhodnout mezi dvěma cestami. Na žádné z cest není feromová stopa.
- Na druhém snímku (obr. 4.1) je znázorněno, že se mravenci zhruba v 50% rozhodli pro kratší cestu a v 50% pro delší cestu.
- Na třetím snímku (obr. 4.1) se první mravenci, kteří zvolili kratší cestu a již našli potravu, vrací po kratší cestě zpět k mraveništi.
- Na čtvrtém snímku (obr. 4.1) se vracející mravenci z kratší cesty vrátili k místu dělení obou cest a stále za sebou zanechávají feromonovou stopu. Mravenci, kteří směřují od mraveniště k potravě, přicházejí k místu, kde se obě cesty dělí. Na kratší cestě je

v tuto chvíli již více feromonu, a tak je tato cesta pro ně mnohem atraktivnější a s větší pravděpodobností zvolí kratší cestu. S postupem času převáží množství feromonové stopy na kratší cestě nad množstvím feromonové stopy na delší cestě tak, že většina mravenců bude volit kratší cestu a feromonová stopa na delší cestě postupně vyprchá.



Obrázek 4.1: Pokus s dvojitým mostem

4.2 Emergence

Pojem emergence se objevuje u systému s architekturou návrhu zdola nahoru. Vytvářejí se jednodušší entity s elementárním chováním. Definují se jejich vzájemné interakce, ale i interakce s okolím. Právě tyto interakce, které jsou v lokálním měřítku zanedbatelné, v globálním měřítku evokují určitý stupeň inteligentního chovaní - princip emergence, kdy

z jednoduchého chování jednotlivce a vzájemných interakcí s ostatními jedinci a s okolím vznikají dovednosti a poznatky vyššího řádu.

Příkladem může být mravenec nesoucí potravu do mraveniště, který za sebou zanechává feromonovou stopu. Ostatní mravenci následují feromonovou stopu při hledání potravy. Mravenec, který se vracel do mraveniště nejkratší cestou, se vrátil jako první. Jeho stopa je nejintenzivnější a ostatní ji následují. Tímto způsobem dovedou najít nejkratší cestu od mraveniště k potravě, aniž by to kterýkoliv jedinec měl jako úmysl.

Mezi základní charakteristické znaky emergence pak patří[2]:

- Inovace - V systému se objevují nové skutečnosti.
- Soudržnost a soulad - Celek funguje na principu samoorganizace.
- Globální úroveň - Některé znaky jsou charakteristické pouze pro celek jako takový.
- Dynamičnost - Systém se vyvíjí.
- Pozorovatelnost - Dá se sledovat.

Emergenci lze kategorizovat na:

- Slabou - Efektu emergence lze dosáhnout i pomocí jedince (například Langtonův mravenec).
- Silnou - Efektu emergence lze dosáhnout pouze spoluprací v celku. Celek je více než všechny jeho součásti (odpovídá výše zmíněnému příkladu).

4.3 Samoorganizace

Samoorganizace je proces, při kterém jsou jednotlivé části systému spojovány do komplexnějšího celku bez jakéhokoliv vedení. Nejvíce příkladů systémů založených na samoorganizaci pochází z vědních oborů jako jsou fyzika nebo chemie (struktura a složení látek).

Pojem samoorganizace je velmi úzce spjat s emergencí a má s ní některé rysy podobné. Přesto může existovat systém se samoorganizací, který nevykazuje známky emergence a naopak. Dopravní situace na silnicích je jeden z příkladů samoorganizace. Každý jednotlivec (automobil) dodržuje určitá pravidla a celý systém (dopravní situace) funguje, aniž by potřeboval centrální řízení.

Mezi základní principy patří[7]:

- Pozitivní zpětná vazba - Způsobuje kumulování příčiny, a tím příčinu posiluje.
- Negativní zpětná vazba - Působí proti změně, která ji vyvolala, a tím reguluje stav systému.
- Neustávání fluktuací - Náhodné jevy pomáhají systému neustat v hledání globálního nejlepšího řešení.
- Mnohonásobné vzájemné interakce.

4.4 Rojová inteligence

Rojová inteligence je pojem, se kterým se setkáváme při simulacích kolektivního jednání v decentralizovaném samoorganizačním systému[1]. Takový systém je složen z populace agentů s jednoduchým chováním, kteří interagují se svým okolím, ať už to jsou ostatní agenti nebo prostředí. Přestože se jedná o systém bez centrálního řízení, tak na základě lokálních interakcí vzniká efekt domnělého centrálního řízení a dochází k emergenci globálního chování. K dosažení těchto rysů je třeba, aby systém obsahoval určitý počet jedinců a došlo tak k vytvoření tzv. roje. V souvislosti s projevy inteligentního chování takového systému hovoříme o inteligenci roje.

4.5 Ant Colony Optimization (ACO)

Algoritmus Ant Colony Optimization (dále jen ACO) je založen na pravděpodobnostním průchodu grafem a hledání nejlepší cesty grafem, která reprezentuje řešení optimalizační úlohy.

ACO algoritmus poprvé publikoval Marco Dorigo v roce 1992[5]. Inspirací pro ACO algoritmus bylo hledání cesty mezi mraveništěm a zdrojem potravy u skutečných mravenců. Algoritmus byl navržen pro nalezení optimální cesty grafem. Řešení, pomocí algoritmu ACO, spočívá v nasazení umělých mravenců. Tito mravenci prochází všemi možnými cestami grafu a zanechávají za sebou virtuální feromonovou stopu, podél delších cest méně a podél kratších více. Po prvním kole objevování se další procházející mravenci orientují podle intenzity zanechané stopy a volí cesty s větší intenzitou, tedy ty kratší. Tímto způsobem se vytvoří na nejkratších cestách vrstva feromonu a na méně efektivnějších trasách stopa vyprchá. Po oznámení výsledků ACO algoritmu byl ACO algoritmus použit telekomunikačními společnostmi ve Francii a Velké Británii pro optimalizaci směrování v jejich sítích. Pro potřeby směrování v sítích byl vytvořen algoritmus AntNet, který je založen na algoritmu ACO. Použití umělých mravenců je pro vyhledávání cest v síti velmi vhodné díky těmto jejich vlastnostem[8]:

- Optimálnost - Dovedou nalézt nejlepší cestu.
- Adaptivnost - Nepřestávají hledat nové lepší cesty.
- Robustnost - Při chybě ve spojení systém nespadne, ale hledají se jiná řešení.

Algoritmus AntNet v porovnání s algoritmem OSPF (Open Shortest Path First), který je oficiálním směrovacím protokolem v síti internet[8], dosahuje lepších výsledků(tab. 4.1).

Zatížení (%)	Ztrátovost paketů u OSPF (%)	Ztrátovost paketů u AntNet (%)
110	8,95	2,19
150	33,2	12,86
200	49,9	26,65

Tabulka 4.1: Porovnání AntNet s algoritmem OSPF

ACO algoritmus má několik variant a vylepšení, které mohou příznivě ovlivnit dosažené výsledky:

- Max-Min ant system - Je stanoveno minimální a maximální množství feromonu, které se může nacházet na cestách mezi jednotlivými body grafu.
- Elitist ant system - Kromě pokládání feromonu všemi mravenci, je v každé iteraci navíc pokládán feromon na dosud nejlepší nalezenou cestu.
- Rank-based ant system - Množství feromonu, které pokládá každý mravenec, je přímo úměrné kvalitě cesty, kterou mravenec nalezl.

4.6 ACO algoritmus pro řešení TSP

Vysvětlení podstaty ACO algoritmu lze nejlépe ukázat na úloze obchodního cestujícího (TSP - Traveling Salesman Problem). Tato úloha se svým zadáním nejvíce podobá chování reálných mravenců - body v prostoru je potřeba pospojovat nejkratší cestou:

- Před vlastním algoritmem je potřeba inicializovat cesty mezi jednotlivými městy. Každé město je propojeno se všemi zbylými městy. Délka cesty mezi dvěma městy je dána geometrickou vzdáleností mezi městy.
- Dále je potřeba náhodně rozmístit všechny mravence do měst, odkud budou začínat hledat své cesty přes všechna města.
- Hlavní smyčka ACO algoritmu se opakuje po iteracích. V každé iteraci projde každý mravenec všechna města na mapě. Pokaždé, když nějaký mravenec dokončí cestu všemi městy, je zjištěna délka jeho cesty. Jestliže je délka jeho cesty menší než délka doposud nalezené nejkratší cesty, je jeho cesta uložena jako nejkratší cesta. Ať už je jeho cesta nejkratší nebo není, tak se na všechny úseky cesty, které prošel, přidá takové množství feromonu, které odpovídá kvalitě cesty, kterou mravenec nalezl.
- Mravenci hledají cestu všemi městy na mapě tak, že začínají v libovolném startovacím městě. Pak je pro všechny cesty, které spojují město, ve kterém se mravenec nachází, s ostatními nenavštívenými městy, určena atraktivita cesty, která je nepřímo úměrná délce cesty a přímo úměrná množství feromonové stopy, která se na cestě nachází. Mravenec se poté náhodně rozhodne pro jednu z možných cest, kdy jsou cesty s větší atraktivitou zvolené s větší pravděpodobností.
- Lokální úprava feromonové stopy znamená odebrání určitého množství feromonu na úseku cesty, který mravenec právě prošel. Lokální úprava napomáhá k tomu, aby mravenci nechodili pouze jednou (nejlepší) cestou, ale hledali i jiná řešení. Toto odebrání feromonu je dočasné a po dokončení iterace všemi mravenci je hodnota feromonové stopy vrácena na hodnotu, kterou měla na počátku iterace.
- Po každé iteraci, když všichni mravenci naleznou svou cestu přes všechna města na mapě, je sníženo množství feromonové stopy na všech cestách. Jedná se o proces evaporace, který napodobuje vyprchávání feromonové stopy v přírodě. Evaporace pomáhá k nalezení nejlepší cesty tím, že méně kvalitní řešení s časem mizí a zůstávají jen řešení, která jsou opakovaně nacházena.

Vzorec pro výpočet množství feromonu, které se má přidat na cestu:

$$F = \frac{DeltaMax}{DelkaCesty}$$

kde:

- F je množství feromonu, které bude přidáno na cestu.
- $DeltaMax$ je konstanta.
- $DelkaCesty$ je celková délka nalezené cesty.

Popis ACO algoritmu aplikovaného na TSP pomocí pseudokódu:

```
// INICIALIZACE:
// c ... počet měst
FOR i = 1 TO c BEGIN
    FOR j = 1 TO c BEGIN
        Vytvoř cestu z města i do města j.
        Vypočítej délku cesty.
        Nastav počáteční množství feromonu na cestě.
    END
END

// m ... počet mravenců
FOR k = 1 TO m BEGIN
    Umísti mravence do náhodně zvoleného města.
END

// VLASTNÍ VÝPOČET:
// t_max ... počet kroků simulace
FOR t = 1 TO t_max BEGIN
    FOR k = 1 TO m BEGIN
        Vytvoř seznam nenavštívených měst.
        Nastav délku nalezené cesty grafem na 0.
        Nastav aktuální město cesty na město, ve kterém se mravenec právě
        nachází.
        WHILE Není seznam nenavštívených měst prázdný BEGIN
            Vyber další navštívené město ze seznamu nenavštívených měst podle
            níže definovaného vzorce.
            Přičti k délce nalezené cesty grafem vzdálenost mezi aktuálním
            městem cesty a dalším navštíveným městem.
            Proveď lokální úpravu feromonové stopy mezi aktuálním
            městem cesty a dalším navštíveným městem.
            Odeber další navštívené město ze seznamu nenavštívených měst.
            Nastav další navštívené město jako aktuální město cesty.
        END
        Přidej feromonovou stopu na nově nalezenou cestu.
        IF Je délka nalezené cesty grafem nejmenší nalezená délka cesty BEGIN
            Nastav nově nalezenou cestu jako nejkratší cestu.
            Nastav délku nově nalezené cesty grafem jako nejkratší nalezenou
            délku cesty.
        END
    END
```

```
    Aktualizuj feromon(sniž množství feromonu) na všech cestách - evaporace.
END
```

Vzorec pro pravděpodobností výběr dalšího navštíveného města ze seznamu nenavštívených měst:

$$p_k(i,j) = \frac{\tau_{ij}^{\alpha} * d_{ij}^{\beta}}{\sum_{g \in J_k(i)} \tau_{ig}^{\alpha} * d_{ig}^{\beta}}$$

kde:

- $p_k(i,j)$ je pravděpodobnost, že mravenec k, který je ve městě i, zvolí jako další město j.
- τ_{ij} je množství feromonové stopy na cestě mezi městem i a městem j.
- α je parametr vyjadřující důležitost feromonové stopy.
- d_{ij} je hodnota nepřímo úměrná vzdálenosti mezi městem i a městem j: $d_{ij} = \frac{1}{|ij|}$
- β je parametr vyjadřující důležitost vzdálenosti mezi městy.
- $J_k(i)$ je množina nenavštívených měst.

U ACO algoritmu můžeme nalézt vlastnosti typické pro emergenci a systém fungující na principu samoorganizace:

- Mnohonásobné vzájemné interakce - Mravenci spolu navzájem komunikují nepřímo pomocí feromonové stopy.
- Neustávání fluktuací - Nahodilost je u ACO algoritmu realizována pravděpodobnostní volbou dalšího navštíveného města.
- Dynamičnost - Systém se vyvíjí a mravenci nacházejí lepší řešení na základě předchozích nalezených cest.
- Pozitivní zpětná vazba - Způsobuje kumulování feromonové stopy na kvalitních cestách, a tím tyto kvalitní cesty posiluje.
- Silná emergence - Efekt nalezení nejkratší cesty grafem je docílen vzájemnou interakcí několika mravenců v průběhu dostatečného množství iterací.

4.7 Společný ACO algoritmus pro řešení TSP, JSP a SCP

Obdobný algoritmus bude použit i u úloh rozvržení úloh na dílně(Job Shop Scheduling Problem - JSP) a u úlohy pokrytí(Set Covering Problem - SCP), rozdíl je ve vytvoření a správě seznamu kandidátních uzlů (měst), protože všechny uzly nebudou hned vždy dostupné:

- u JSP města reprezentují operace jednotlivých úloh, které mají být provedeny. U SCP města reprezentují množiny, které mohou být vybrány pro pokrytí všech bodů.
- u JSP jsou v inicializační části do seznamu nenavštívených měst vloženy počáteční operace všech úloh. U SCP jsou do seznamu nenavštívených měst vložena všechna města(množiny).

- u JSP je po každé pravděpodobnostní volbě dalšího navštíveného města A, ze seznamu nenavštívených měst, do seznamu nenavštívených měst přidáno město, reprezentující operaci, která navazuje v rámci úlohy na operaci, kterou reprezentuje zvolené město A. U SCP jsou po každé pravděpodobnostní volbě dalšího navštíveného města A odebrány ze seznamu nenavštívených měst města reprezentující množiny, které obsahují pouze body, které jsou již pokryty doposud nalezeným řešením.

- u JSP délka nalezené cesty nezávisí pouze na délce trvání jednotlivých operací, ale také na vytíženosti strojů, na kterých mají být operace provedeny. U SCP je celková délka cesty dána součtem cen všech množin, které byly zvoleny jako řešení úlohy.

Na základě odlišností mezi TSP, JSP a SCP lze algoritmus zobecnit a konkrétní implementaci inicializace a správy seznamu nenavštívených měst a získání délky nalezené cesty se přesune do tříd optimalizační úloh. Procedury, které se přesunou do tříd jednotlivých optimalizačních úloh:

- `Inicializuj cesty mezi jednotlivými městy.` - Nastavení délky jednotlivých cest a počátečního množství feromonové stopy. Délka cesty u TSP je dána geometrickou vzdáleností a musí se spočítat. U JSP je délka cesty dána dobou trvání operace A, jestliže určujeme délku cesty mezi operacemi AB, nebo dobou trvání operace B, jestliže určujeme délku cesty mezi operacemi BA. U SCP je délka cesty dána cenou množiny B a počtem bodů, které obsahuje množina B, a které současně nejsou pokryty doposud nalezeným řešením S. Délky cest jsou u SCP přepočítávány po každém přechodu do dalšího města a v inicializační části nejsou délky mezi městy počítány vůbec.

- `Vytvoř seznam nenavštívených měst.` - Vytvoří seznam všech měst, která jsou na počátku úlohy dostupné. U TSP jsou to všechna města. U JSP jsou to pouze ta města, která reprezentují počáteční operace všech úloh. U SCP jsou to všechna města.

- `Udělej přechod z aktuálního města do dalšího navštíveného města.` - u TSP se pouze ze seznamu nenavštívených měst odebere další navštívené město. U JSP se odebere ze seznamu nenavštívených měst další navštívené město. Navíc se do seznamu nenavštívených měst přidá město reprezentující operaci, která následuje po operaci reprezentované dalším navštíveným městem, pokud není operace reprezentovaná dalším navštíveným městem poslední operací dané úlohy. U SCP je ze seznamu nenavštívených měst odebráno další navštívené město a kromě toho jsou odebrána i města reprezentující množiny, které obsahují pouze body, které jsou již pokryty doposud nalezeným řešením.

- `Zjisti délku nalezené cesty.` - u TSP je celková délka cesty sumou všech jednotlivých cest mezi městy. Celková délka cesty u JSP nezáleží pouze na jednotlivých délkách cest mezi městy(na době trvání jednotlivých operací), ale i na vytíženosti strojů. U SCP je celková délka cesty dána součtem cen všech množin, které byly zvoleny jako řešení úlohy.

Zde je pak společný ACO algoritmus pro TSP, JSP a SCP:

```
// INICIALIZACE:
// c ... počet měst
Inicializuj cesty mezi jednotlivými městy.
```

```
// VLASTNÍ VÝPOČET:
// t_max ... počet kroků simulace
FOR t = 1 TO t_max BEGIN
    FOR k = 1 TO m BEGIN
        Vytvoř seznam nenavštívených měst.
        // m ... počet mravenců
        FOR k = 1 TO m BEGIN
            Umísti mravence do náhodně zvoleného města ze seznamu
            nenavštívených měst.
        END
        Nastav aktuální město cesty na město, ve kterém se mravenec právě
        nachází.
        WHILE Není seznam nenavštívených měst prázdný BEGIN
            Vyber další navštívené město ze seznamu nenavštívených měst
            podle níže definovaného vzorce.
            Proveď lokální úpravu feromonové stopy mezi aktuálním
            městem cesty a dalším navštíveným městem.
            Udělej přechod z aktuálního města do dalšího navštíveného města.
            Nastav další navštívené město jako aktuální město cesty.
        END
        Zjisti délku nalezené cesty.
        Přidej feromonovou stopu na nově nalezenou cestu.
        IF Je délka nalezené cesty grafem nejmenší nalezená délka cesty BEGIN
            Nastav nově nalezenou cestu jako nejkratší cestu.
            Nastav délku nově nalezené cesty grafem jako nejkratší
            nalezenou délku cesty.
        END
    END
    Aktualizuj feromon(sniž množství feromonu) na všech cestách - evaporace.
END
```

Kapitola 5

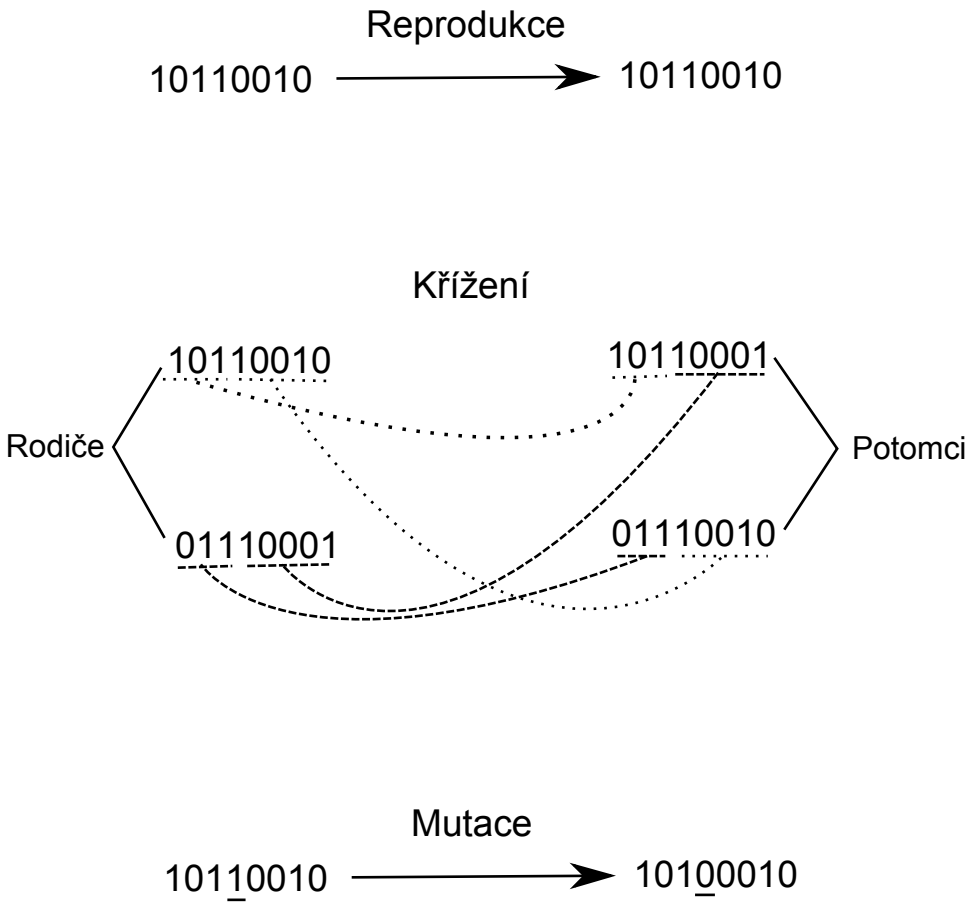
Genetický algoritmus (GA)

Genetický algoritmus získal svůj název podle toho, že se inspiruje procesy, ke kterým dochází v rámci evoluce - křížení, dědičnost, přirozený výběr nebo mutace. Genetický algoritmus patří mezi smíšené optimalizační algoritmy. Pomocí tohoto algoritmu lze řešit úlohy, pro které neexistuje exaktní algoritmus, nebo by výpočet exaktním algoritmem trval neúměrně dlouho. Podstata genetického algoritmu spočívá v cyklickém vytváření nových generací. Každá generace obsahuje daný počet jedinců. Každý jedinec představuje řešení zadané úlohy. Jedinec je složen z předem daného počtu genů. Geny mohou nabývat různých hodnot(obvykle pouze binární - 0 a 1). Kombinace hodnot jednotlivých genů reprezentuje řešení zadané úlohy. Popis podstaty genetického algoritmu [13]:

- v inicializační části je vytvořena první generace jedinců. Každý jedinec reprezentuje náhodně zvolené řešení zadané úlohy.
- Algoritmus se opakuje po generacích(iteracích). V každé generaci je pro všechny jedince spočítána jejich fitness hodnota. Pro každou úlohu se fitness hodnota počítá trochu odlišně. Můžeme to přirovnat k výpočtu délky cesty u ACO. Fitness hodnota udává, jak kvalitní je řešení, které daný jedinec představuje. Poté jsou náhodně zvoleni jedinci, ze kterých bude vytvořena nová generace. Jedinci s větší hodnotou fitness mají větší šanci na to, že budou vybráni pro vytvoření příští generace.
- Nová generace jedinců vznikne z vybraných jedinců operacemi reprodukce, křížení, mutace a případně jejich kombinací:
  - Reprodukce - Jedinec zůstane zachován beze změny a je zkopírován do nové generace (obr. 5.1). O tom, jestli bude daný jedinec zkopírován beze změny, rozhoduje faktor křížení. Faktor křížení je parametr, který nabývá hodnot $\langle 0, 1\rangle$. Pokud má faktor křížení hodnotu $0,8$, tak každý jedinec bude s pravděpodobností $0,8$ křížen s jiným jedincem a s pravděpodobností $0,2$ bude reprodukován.
  - Křížení - z vybraných jedinců(obvykle dvou rodičů), vzniknou noví jedinci(obvykle dva potomci), tím způsobem, že skombinují části rodičů mezi sebou navzájem (obr. 5.1). O tom, jak velká část genů se pro vytvoření nových jedinců vezme od jednotlivých rodičů, určuje parametr bod křížení. Bod křížení nabývá hodnot od 1 do $(n-1)$, kde n je počet genů každého jedince. Pokud bude mít bod křížení hodnotu 3, tak bude nový jedinec tvořen prvními třemi geny jednoho rodiče a čtvrtým až n-tým genem druhého rodiče.

– Mutace - Náhodná změna určité části jedince (obr. 5.1). Pravděpodobnost, že u jedince dojde k mutaci, je dána faktorem mutace. Faktor mutace je parametr, který nabývá hodnot $\langle 0, 1\rangle$. Jestliže je hodnota faktoru křížení $0,05$, pak pravděpodobnost, že u jedince dojde k mutaci, je $0,05$.

- Algoritmus může být ukončen, pokud je nalezeno řešení, které je dostatečně kvalitní. Nebo pokud počet generací dosáhl předem stanovené maximální hodnoty.



Obrázek 5.1: Operace používané pro vytvoření nové generace jedinců

Rovnice pro výpočet pravděpodobnosti zvolení jedince pro vytvoření nové generace na základě jeho fitness hodnoty:

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j}$$

kde:

- p_i je pravděpodobnost zvolení jedince i.
- f_i je fitness hodnota jedince i.
- $\sum_{j=1}^{n} f_j$ je součet fitness hodnot všech jedinců.

Popis genetického algoritmu pomocí pseudo kódu:

```
// INICIALIZACE:
// n ... počet jedinců
Vytvoř počáteční generaci n jedinců.

// VLASTNÍ VÝPOČET:
// t_max ... maximální počet generací(iterací) simulace
FOR t = 1 TO t_max BEGIN
    Spočítej fitness pro všechny jedince současné generace.
    // n ... počet jedinců
    FOR k = 1 TO n BEGIN
        Na základě fitness hodnot proved' pravděpodobnostní výběr jedince.
        Proved' operaci křížení, nebo reprodukce(výběr operace záleží
        na faktoru křížení).
        Proved' operaci mutace(pravděpodobnost mutace je dána faktorem
        mutace).
    END
    Nově vytvořenou generaci jedinců označ za současnou generaci.
END
```

U GA můžeme nalézt vlastnosti typické pro emergenci a systém fungující na principu samoorganizace:

- Neustávání fluktuací - Nahodilost je u GA algoritmu realizována pravděpodobnostním způsobem generováním počáteční generace, pravděpodobnostním způsobem selekce, křížení a mutace.
- Dynamičnost - Systém se vyvíjí a lepší řešení jsou nalézána na základě předchozích nalezených řešeních, a to díky principu selekce dle fitness hodnoty a křížení.
- Mnohonásobné vzájemné interakce - Informace o systému jsou uchovávány v genech samotných jedinců. Při operaci křížení dochází k výměně těchto informací, a tak i k jistému druhu nepřímé komunikace.
- Pozitivní zpětná vazba - Způsobuje přežívání jedinců reprezentujících kvalitní řešení, a tím posiluje kvalitní řešení reprezentovaná těmito jedinci.
- Silná emergence - Efekt nalezení nejkvalitnějšího řešení je docílen vzájemnou interakcí(především křížení) dostatečného počtu jedinců v průběhu dostatečného množství generací.

5.1 GA pro řešení TSP, JSP a SCP

Pro řešení TSP a JSP pomocí GA je potřeba pracovat s generacemi jedinců, kteří splňují následující podmínky:

- Aby všichni jedinci měli n genů, kde n je počet měst v dané optimalizační úloze.
- Geny budou nabývat hodnot od 1 do n, kde n je počet měst v dané optimalizační úloze. Přitom každá hodnota genu musí být unikátní(žádné dva geny jednoho jedince nesmí být shodné).
- Operace křížení a mutace u jedinců nesmí způsobit, aby řešení reprezentované jedincem bylo v rozporu se specifikací optimalizační úlohy. Toto platí hlavně u JSP, aby nedošlo k přehození pořadí operací v rámci jedné úlohy.

Pseudokód pro operaci křížení je navržen tak, aby řešení reprezentované vzniklým jedincem nebylo v rozporu se specifikací optimalizační úlohy:

```
// Operace křížení:
Zvol náhodný bodKrizeni, kde bodKrizeni > 1 a současně bodKrizeni < počet
měst.
Vytvoř 1 až bodKrizeni(včetně) genů nového jedince z 1 až
bodKrizeni(včetně) genů prvního rodiče.
Zbylé geny nového jedince získej z genů druhého rodiče. A to tím způsobem,
že postupně kopíruj hodnoty genů druhého rodiče, které nebyly dosud
v genech nového jedince použity.
```

Druhý nový jedinec vznikne stejným postupem s prohozenou rolí rodičů (tab. 5.1).

Rodiče	Bod křížení	Potomci
45231, 21543	2	45213, 21453
45231, 21543	3	45213, 21543
35421, 14253	2	35142, 14352
35421, 14253	3	35412, 14235

Tabulka 5.1: Křížení jedinců

Řešení SCP pomocí GA se liší od TSP a JSP, protože geny jedinců nereprezentují města, ale body, které mají být pokryty:

- Jedinci mají n genů, kde n je počet bodů v dané optimalizační úloze, které mají být pokryty.
- Geny budou nabývat hodnot od 1 do m, kde m je počet měst(množin) v dané optimalizační úloze. Přitom každá hodnota genu nemusí být unikátní(dva geny jednoho jedince mohou být shodné).
- Operace křížení muže probíhat bez ohlížení na to, že by došlo ke vzniku jedince, který nesplňuje zadání úlohy. U operace mutace je potřeba hlídat pouze podmínku, aby gen zmutoval jen na množinu, která obsahuje bod reprezentovaný genem.

Z toho vyplývá několik procedur, které je potřeba řešit individuálně, kvůli odlišnostem v povaze úloh, a bubou muset být přesunuty do třídy jednotlivých optimalizačních úloh:

- `Vytvoř počáteční generaci n jedinců.` - Vygenerovat množinu náhodně vytvořených jedinců(řešení), tak aby vytvoření jedinci(řešení) náleželi do množiny všech možných řešení dané optimalizační úlohy. U TSP jsou jako řešení úlohy možné všechny kombinace vytvořené ze seznamu všech měst. Počet všech možných řešení TSP je popsán v kapitole Traveling Salesman Problem (TSP). U JSP je počet možných řešení omezen podmínkou, že operace v rámci dané úlohy provedeny v určeném pořadí, a tak je prostor možných řešení i vytvoření náhodného řešení omezeno. U SCP je množina všech řešení omezena podmínkou pokrytí všech zadaných bodů úlohy.

- `Proveď operaci křížení.` - Vytvořit nového jedince z rodičovských jedinců. U TSP a JSP bude algoritmus operace křížení stejný. Důvodem je, aby nedošlo k vytvoření jedince, který by neodpovídal řešení úlohy. U SCP bude algoritmus křížení jednodušší. Nový jedinec vznikne prostým zkopírováním příslušných částí rodičovských jedinců podle bodu křížení.

- `Proveď operaci mutace.` - Bude řešena u každé úlohy individuálně.

- `Získej cestu reprezentovanou jedincem ze současné generace.` - Tato procedura je zavedena kvůli SCP, protože geny jedince reprezentují body a nikoliv města. U TSP a JSP jde v této proceduře o pouhé předání genů, které reprezentují města na cestě.

Pro výpočet fitness hodnoty je použita procedura pro získání délky nalezené cesty:

$$f_i = \frac{DeltaMax}{DelkaCesty}$$

kde:

- f_i je fitness hodnota jedince i.

- $DeltaMax$ je konstanta.

- $DelkaCesty$ je celková délka nalezené cesty. Pro výpočet se používá stejná procedura jako u ACO.

Pro náhodný výběr jedince na základě fitness hodnoty je použit princip rulety. Použití principu rulety na příkladu:

- Je stanovena fitness hodnota pro všechny jedince v generaci(obr. 5.2).

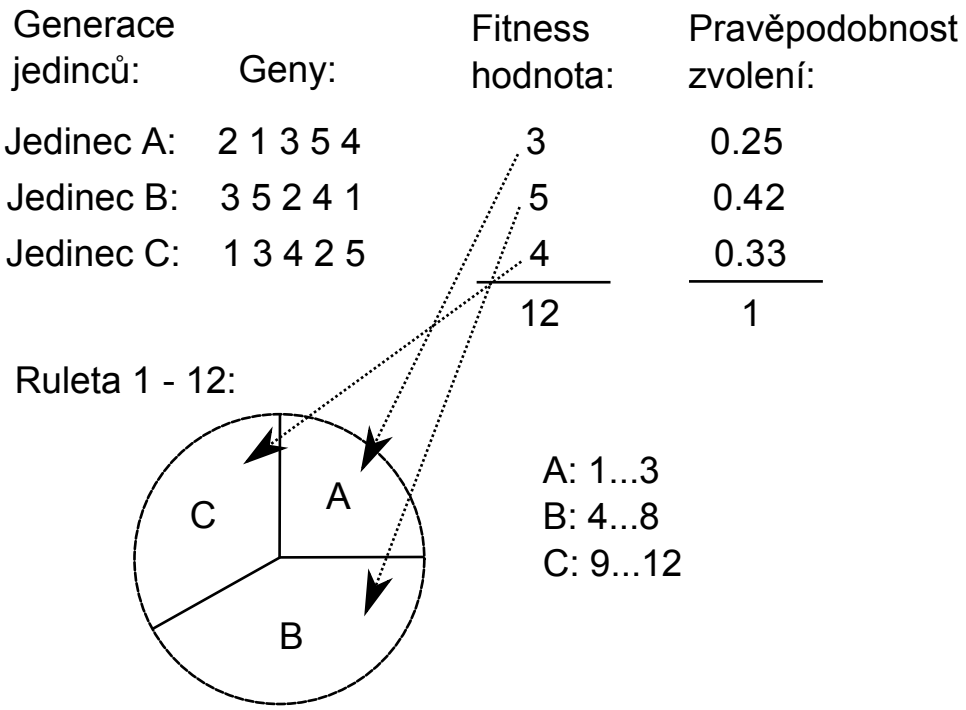
- Každému jedinci je na ruletě přidělen takový prostor, který odpovídá pravděpodobnosti jeho zvolení(obr. 5.2).

- Při náhodném výběru jedince je náhodně vybráno číslo na ruletě. Náhodně zvolený jedinec je ten jedinec, který připadá na vybrané náhodné číslo na ruletě. Pokud bude náhodně vybrané číslo například 5, pak náhodně zvolený jedinec(na příkladu z obrázku 5.2) bude jedinec B.

GA v pseudokódu:

```
// INICIALIZACE:
// n ... počet jedinců
Vytvoř počáteční generaci n jedinců.

// VLASTNÍ VÝPOČET:
// t_max ... maximální počet generací(iterací) simulace
FOR t = 1 TO t_max BEGIN
    // n ... počet jedinců
    FOR k = 1 TO n BEGIN
        Získej cestu reprezentovanou jedincem ze současné generace.
        Spočítej délku cesty.
        IF Je délka cesty grafem nejmenší nalezená délka cesty BEGIN
            Nastav nově nalezenou cestu jako nejkratší cestu.
            Nastav délku nově nalezené cesty grafem jako nejkratší
            nalezenou délku cesty.
        END
        Spočítej fitness hodnotu jedince na základě délky cesty.
    END
    // n ... počet jedinců
    FOR k = 1 TO n BEGIN
        Na základě fitness hodnot proveď pravděpodobnostní výběr jedince.
        Proveď operaci křížení, nebo reprodukce(výběr operace záleží
        na faktoru křížení).
        Proveď operaci mutace(pravděpodobnost mutace je dána faktorem
        mutace).
    END
    Nově vytvořenou generaci jedinců označ za současnou generaci.
END
```



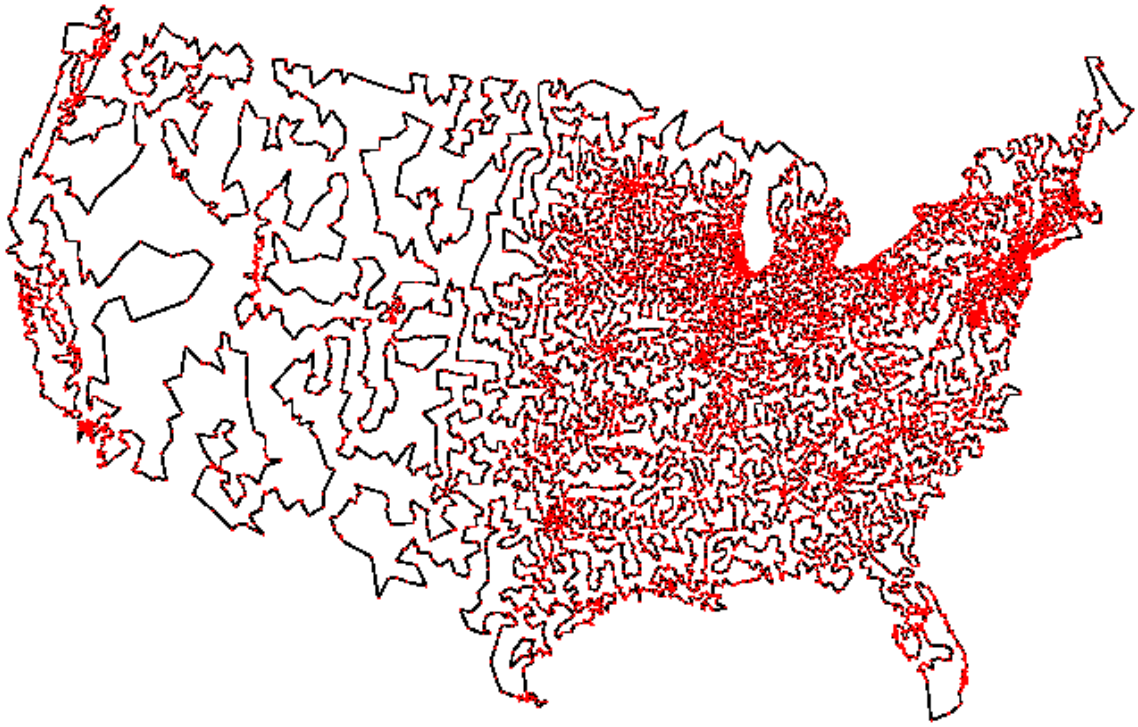
Obrázek 5.2: Princip rulety pro pravděpodobnostní výběr jedince na základě jeho fitness hodnoty

Kapitola 6

Traveling Salesman Problem (TSP)

Traveling Salesman Problem (dále jen TSP), problém obchodního cestujícího, je kombinatorická optimalizační úloha. S TSP se setkáváme v praxi v nejrůznějších oborech jako například logistika a plánování, nebo při výrobě mikročipů. TSP je definován takto:

- Máme množinu n měst a pro každou dvojici měst definovanou vzdálenost. Přitom vzdálenost z města A do města B je stejná jako vzdálenost z města B do města A.
- Cílem je nalézt takovou cestu, která prochází všemi městy právě jednou, začíná a končí ve stejném městě, a která je ze všech možných cest nejkratší (obr. 6.1).

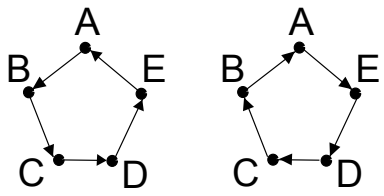


Obrázek 6.1: Traveling Salesman Problem (TSP)

Tato optimalizační úloha je NP-těžká a s rostoucím množstvím měst roste náročnost úlohy exponenciálně, takže nelze použít metody založené na průchodu stavového prostoru. Toto

tvrzení si dokážeme:

Máme množinu n měst. Zvolíme libovolné město jako startovní město cesty. Zbude nám $n-1$ měst, ze kterých sestavujeme cestu. Počet všech možných cest je tak $(n-1)!$. Protože cesta začíná a končí ve stejném městě, lze jedna cesta vytvořit dvěma způsoby(ABCDE nebo AEDCB, obr. 6.2), proto je potřeba počet cest dělit dvěma.



Obrázek 6.2: Dvě možnosti sestavení stejné cesty

Výsledný vzorec pro výpočet možných cest je:

$$p_k(i,j) = \frac{(n-1)!}{2}$$

Tabulka demonstruje nárůst možných cest se vzrůstajícím počtem měst(tab. 6.1).

Počet měst (%)	Počet možných cest
3	1
5	12
7	360
10	181440
15	43589145600
20	60822550204416000
25	310224200866620000000000
30	4420880996869850000000000000000
35	147616399519802000000000000000000000000
40	10198941040598700000000000000000000000000000000
45	1329135787394220000000000000000000000000000000000000000
50	304140932017134000000000000000000000000000000000000000000000000

Tabulka 6.1: Nárůst možných cest se vzrůstajícím počtem měst

I kdyby sestavení jedné cesty a zjištění její délky trvalo jednu nanosekundu(10^{-9} s), tak by výpočet všech možných cest trvalo neúměrně dlouho, jak je možné vidět v následující tabulce:

Tabulka ukazující časovou náročnost se vzrůstajícím počtem měst(kdyby sestavení jedné cesty a zjištění její délky trvalo jednu nanosekundu(10^{-9} s)) (tab. 6.2).

Už při relativně malém počtu měst je metoda založená na průchodu stavového prostoru nepoužitelná, proto jsou k nalezení nejkratší cesty používány stochastické a smíšené algoritmy. Tyto algoritmy nemusí nutně nalézt skutečně nejkratší cestu, ale v rozumném čase dovedou nalézt cestu, která se k nejkratší cestě velmi přibližuje.

Počet měst (%)	Čas výpočtu v rocích
20	2
25	$9,84 * 10^6$
30	$1,4 * 10^{14}$
35	$4,68 * 10^{21}$
40	$3,23 * 10^{29}$

Tabulka 6.2: Časovou náročnost výpočtu

6.1 Řešení TSP pomocí ACO

Základní kostra ACO algoritmu pro řešení TSP byla již popsána v kapitole Ant Colony Optimization (ACO). Zde budou podrobněji popsány procedury, které se vzhledem k odlišnostem jednotlivých optimalizačních úloh, musí být řešeny individuálně. Jedná se o tyto úlohy:

- `Inicializuj cesty mezi jednotlivými městy.` - Nastavení délky jednotlivých cest a počátečního množství feromonové stopy.
- `Vytvoř seznam nenavštívených měst.` - Vytvoří seznam všech měst, která jsou na počátku úlohy dostupné.
- `Udělej přechod z aktuálního města do dalšího navštíveného města.` - Správa seznamu nenavštívených měst, kterou je potřeba provést při sestavování cesty všemi městy.
- `Zjisti délku nalezené cesty.` - Zjištění délky cesty, a tím určení kvality řešení optimalizační úlohy.

6.1.1 Inicializace cest mezi jednotlivými městy

Tato část zahrnuje nastavení délky jednotlivých cest a počátečního množství feromonové stopy. Před vlastní inicializací cest je potřeba inicializovat města. To proběhne načtením parametrů měst ze souboru. Délka cesty u TSP je dána geometrickou vzdáleností a musí se spočítat:

$$d_{AB} = d_{BA} = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2}$$

kde:

- d_{AB} je vzdálenost mezi městy A a B. Vzdálenost mezi městy A a B je stejná jako vzdálenost mezi městy B a A.
- x_A je x-ová souřadnice města A.
- x_B je x-ová souřadnice města B.
- y_A je y-ová souřadnice města A.
- y_B je y-ová souřadnice města B.

Pro cesty je připravena třída Cesta, která zapouzdřuje proměnné délka cesty a množství feromonu. Cesty jsou uloženy do dvourozměrné matice $m * m$, kde m je počet měst. Matice obsahuje ukazatele na instance třídy Cesta. Toho je využito k zachycení faktu, že cesta mezi městy A a B je shodná jako cesta mezi městy B a A. A proto je v matici do obou příslušných polí uložen stejný odkaz na jednu instanci třídy Cesta:

```
// Inicializuj cesty mezi jednotlivými městy:
// m ... počet měst
Cesta[][] cesty = new Cesta[m][m];
FOR i = 1 TO m BEGIN
    FOR j = 1 TO m BEGIN
        IF i != j BEGIN
            Vytvoř instanci cesta třídy Cesta.
            Spočítej délku cesty mezi městy i a j. Nastav spočítanou
            délku jako délku cesty instance cesta.
            Nastav počáteční množství feromonové stopy u instance cesta.
            cesty[i][j] = cesta;
            cesty[j][i] = cesta;
        END
    END
END
```

6.1.2 Vytvoření seznamu nenavštívených měst

Tato část zahrnuje vytvoření seznamu všech měst, která jsou na počátku úlohy dostupné. U TSP jsou to všechna města.

Pseudokód inicializace seznamu:

```
// Vytvoř seznam nenavštívených měst:
// m ... počet měst
FOR i = 1 TO m BEGIN
    Přidej město i do seznamu nenavštívených měst.
END
```

6.1.3 Přechod z aktuálního města do dalšího navštíveného města

Tato část zahrnuje správu seznamu nenavštívených měst při vytváření cesty. U TSP je pouze další navštívené město odebráno ze seznamu nenavštívených měst.

Pseudokód správy seznamu:

```
// Udělej přechod z aktuálního města do dalšího navštíveného města:
Ze seznamu nenavštívených měst odeber další navštívené město.
```

6.1.4 Zjištění délky nalezené cesty

Tato část zahrnuje zjištění celkové délky nalezené cesty. U TSP je celková délka cesty sumou všech jednotlivých cest mezi městy. Záleží na pořadí měst, jak jimi mravenec procházel.

Nalezená cesta je reprezentována polem indexů jednotlivých měst tzn., že například `nalezenaCesta[1] = 3` znamená, že první město, kterým mravenec prošel bylo město s indexem 3. Pseudokód zjištění celkové délky cesty:

```
// Zjisti délku nalezené cesty:
delkaCesty = 0
// m ... počet měst
FOR i = 1 TO m BEGIN
    IF i < m THEN j = i + 1
    ELSE j = 1
    delkaCesty += cesty[nalezenaCesta[i]][nalezenaCesta[j]].DelkaCesty();
END
```

6.2 Řešení TSP pomocí GA

Základní kostra GA pro řešení TSP byla již popsána v kapitole Genetický algoritmus (GA). Zde budou podrobněji popsány procedury, které se vzhledem k odlišnostem jednotlivých optimalizačních úloh, musí řešit individuálně:

- `Vytvoř počáteční generaci n jedinců.` - Vygenerovat množinu náhodně vytvořených jedinců(řešení), tak aby vytvoření jedinci(řešení) náleželi do množiny všech možných řešení dané optimalizační úlohy.
- `Proveď operaci křížení.` - Vytvořit nového jedince z rodičovských jedinců. U TSP a JSP bude algoritmus operace křížení stejný. Algoritmus byl již popsán v kapitole Genetický algoritmus (GA), a proto zde nebude rozebírán.
- `Proveď operaci mutace.` - Mutace náhodného genu u TSP musí nutně způsobit změnu druhého genu, protože geny každého jedince reprezentují permutace všech měst a nesmí dojít k tomu, aby dva geny jednoho jedince měli shodnou hodnotu.
- `Získej cestu reprezentovanou jedincem ze současné generace.` - u TSP a JSP jde v této proceduře o pouhé předání genů, které reprezentují města na cestě, proto zde nebude tato procedura popisována.

6.2.1 Vytvoření počáteční generace n jedinců

Tato část zahrnuje vytvořených množiny jedinců(řešení), tak aby vytvoření jedinci(řešení) náleželi do množiny všech možných řešení. U TSP jsou jako řešení úlohy možné všechny kombinace vytvořené ze seznamu všech měst.

Geny, ze kterých jsou jedinci složeni, jsou reprezentovány polem typu integer. Jsou zde použity procedury, které byly popsány výše: `Vytvoř seznam nenavštívených měst.` a `Udělej přechod z aktuálního města do dalšího navštíveného města.`. Pseudokód pro vytvoření náhodného jedince:

```
// Vytvoř náhodného jedince:
// m ... počet měst
int[] geny = new int[m]
Vytvoř seznam nenavštívených měst.
FOR i = 1 TO m BEGIN
    Vyber náhodný indexMesta - číslo v rozsahu od 1 do m.
    WHILE TRUE BEGIN
        IF náhodně zvolené město je v seznamu nenavštívených měst BEGIN
            geny[i] = indexMesta
```

```
            Udělej přechod z aktuálního města do dalšího navštíveného města,
            kde další navštívené město je město, které má indexMesta.
            BREAK
        END
        ELSE BEGIN
            IF indexMesta < m THEN indexMesta += 1
            ELSE indexMesta = 0
        END
    END
END
```

6.2.2 Proveď operaci mutace

Tato část zahrnuje změnu genů jedinců(řešení), tak aby změnění jedinci(řešení) náleželi do množiny všech možných řešení. U TSP jsou jako řešení úlohy možné všechny kombinace vytvořené ze seznamu všech měst:

```
// Vytvoř náhodnou mutaci genů jedince:
// m ... počet měst
Vyber náhodný indexGenu1 - číslo v rozsahu od 1 do m.
Vyber náhodný indexGenu2 - číslo v rozsahu od 1 do m.
int temp = geny[indexGenu1]
geny[indexGenu1] = geny[indexGenu2]
geny[indexGenu2] = temp
```

Kapitola 7

Job Shop Scheduling Problem (JSP)

Job Shop Scheduling Problem (dále jen JSP), problém rozvržení úloh na dílně, je kombinatorická optimalizační úloha. S touto optimalizační úlohou se v praxi setkáváme při plánování procesu výroby. Výsledky nasazení optimalizace na výrobní proces nevedou jen k efektivnějšímu využití výrobních kapacit, ale pomohou při odhadu množství vyprodukovaných výrobků nebo při plánování zásob zdrojových surovin a v neposlední řadě například při výpadku některého z výrobních prostředků. JSP je definován takto:

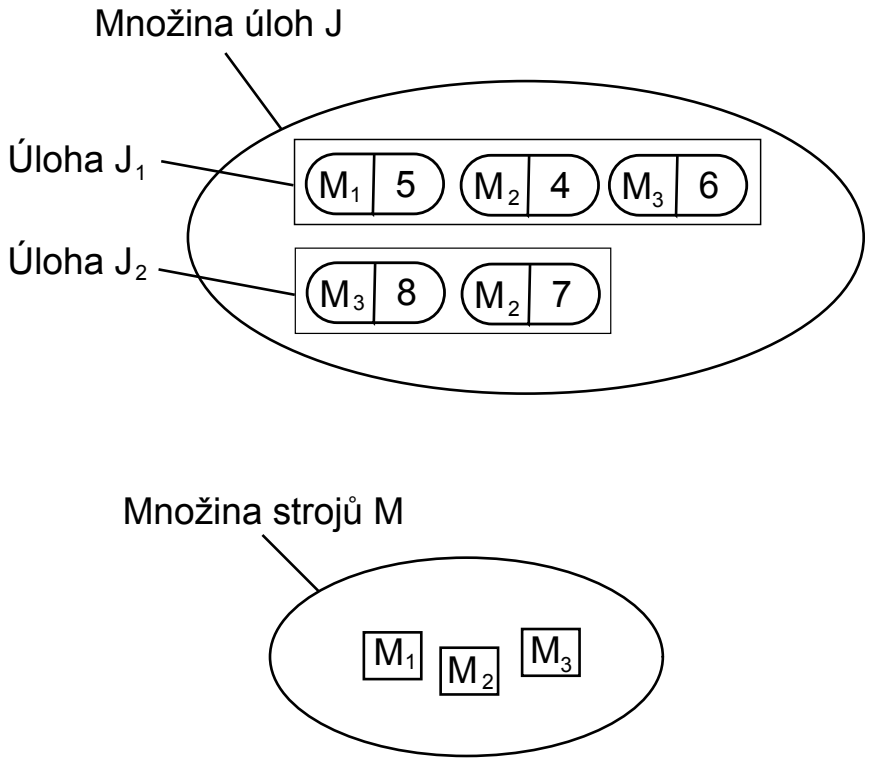
- Máme množinu J obsahující n úloh: $J = \{J_1, J_2, ..., J_n\}$ (obr. 7.1).
- Máme množinu M obsahující m strojů: $M = \{M_1, M_2, ..., M_m\}$ (obr. 7.1).
- Každá úloha J_i se skládá z retězce operací m_i: $m_i = \{O_{i1}, O_{i2}, ..., O_{im}\}$, které musí být provedeny ve stanoveném pořadí (obr. 7.2).
- O_{ij} je j-tá operace, J_i-té úlohy, která má být realizována na stroji M_x za dobu T_{ij} bez přerušení.
- Každý stroj M_x může provádět maximálně jednu operaci současně.
- Každá úloha J_i může být prováděna maximálně na jednom stroji současně.
- Cílem je minimalizovat čas zpracování všech úloh J_i.

Tato optimalizační úloha je také NP-těžká. Nárůst stavového prostoru nelze jednoznačně určit jako u TSP, protože graf vytvořený propojením všech operací jednotlivých úloh nemá všechny hrany obousměrné (obr. 7.3). Nelze tak sestavit cestu, ve které by například operace O_{12} předcházela operaci O_{11}.

7.1 Řešení JSP pomocí ACO

Základní kostra ACO algoritmu pro řešení JSP byla již popsána v kapitole Ant Colony Optimization (ACO). Zde budou podrobněji popsány procedury, které se vzhledem k odlišnostem jednotlivých optimalizačních úloh, musí být řešeny individuálně. Jedná se o tyto úlohy:

- `Inicializuj cesty mezi jednotlivými městy.` - Nastavení délky jednotlivých cest a počátečního množství feromonové stopy.



Obrázek 7.1: Množina úloh a množina strojů

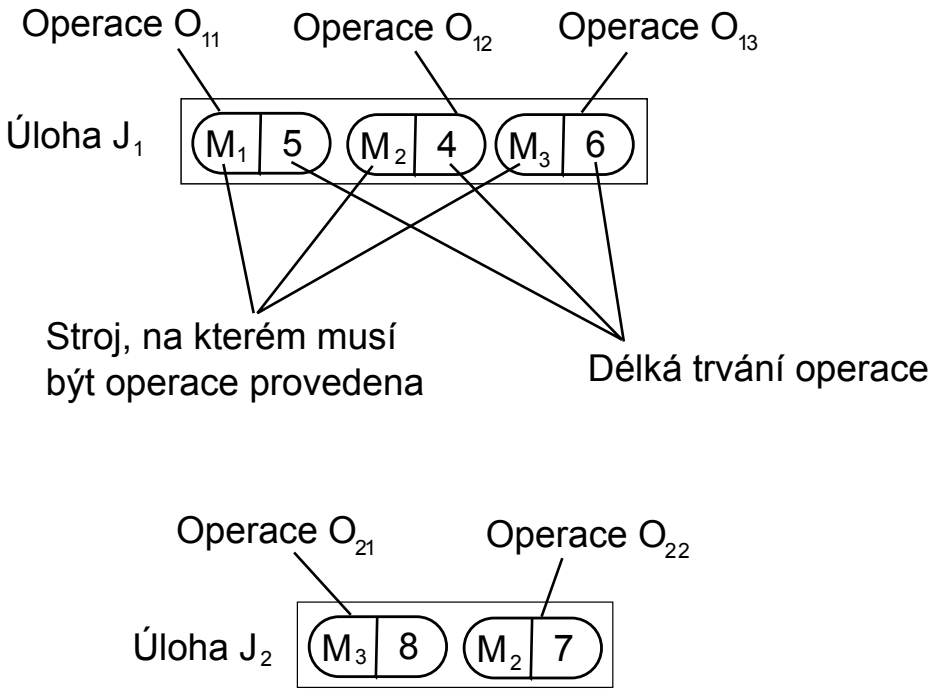
- `Vytvoř seznam nenavštívených měst.` - Vytvoří seznam všech měst, která jsou na počátku úlohy dostupné.
- `Udělej přechod z aktuálního města do dalšího navštíveného města.` - Správa seznamu nenavštívených měst, kterou je potřeba provést při sestavování cesty všemi městy.
- `Zjisti délku nalezené cesty.` - Zjištění délky cesty, a tím určení kvality řešení optimalizační úlohy.

7.1.1 Inicializace cest mezi jednotlivými městy

Tato část zahrnuje nastavení délky jednotlivých cest a počátečního množství feromonové stopy. Před vlastní inicializací cest je potřeba inicializovat města. To proběhne načtením parametrů měst ze souboru. Délka cesty mezi operacemi je dána dobou trvání výchozí operace. Délka cesty mezi operacemi A a B je dána dobou trvání operace A. V opačném směru cesty, mezi operacemi B a A, je dána dobou trvání operace B:

Pro cesty je připravena třída Cesta, která zapouzdřuje proměnné délka cesty a množství feromonu. Cesty jsou uloženy do dvourozměrné matice $m * m$, kde m je počet měst.:

```
// Inicializuj cesty mezi jednotlivými městy:
// m ... počet měst
Cesta[][] cesty = new Cesta[m][m];
```



Obrázek 7.2: Úlohy a operace

```
FOR i = 1 TO m BEGIN
    FOR j = 1 TO m BEGIN
        IF i != j BEGIN
            Vytvoř instanci cesta1 třídy Cesta.
            Přečti dobu trvání operace i a nastav jí jako délku
            cesty u cesta1.
            Nastav počáteční množství feromonové stopy u instance cesta1.
            cesty[i][j] = cesta1;

            Vytvoř instanci cesta2 třídy Cesta.
            Přečti dobu trvání operace j a nastav jí jako délku
            cesty u cesta2.
            Nastav počáteční množství feromonové stopy u instance cesta2.
            cesty[j][i] = cesta2;
        END
    END
END
```

7.1.2 Vytvoření seznamu nenavštívených měst

Tato část zahrnuje vytvoření seznamu všech měst, která jsou na počátku úlohy dostupné. U JSP jsou to všechny počáteční operace všech úloh.

Pseudokód inicializace seznamu:

```
// Vytvoř seznam nenavštívených měst:
// m ... počet operací(měst)
FOR i = 1 TO m BEGIN
    IF je operace i počáteční operací BEGIN
      Přidej operaci i do seznamu nenavštívených měst.
    END
END
```

7.1.3 Přechod z aktuálního města do dalšího navštíveného města

Tato část zahrnuje správu seznamu nenavštívených měst při vytváření cesty. U JSP se odebere ze seznamu nenavštívených měst další navštívené město. Navíc se do seznamu nenavštívených měst přidá město reprezentující operaci, která následuje po operaci reprezentované dalším navštíveným městem, pokud není operace reprezentovaná dalším navštíveným městem poslední operací dané úlohy.

Pseudokód správy seznamu:

```
// Udělej přechod z aktuálního města do dalšího navštíveného města:
Ze seznamu nenavštívených měst odeber další navštívené město.
IF není operace, reprezentovaná dalším navštíveným městem, poslední operací
dané úlohy BEGIN
    Vlož do seznamu nenavštívených měst operaci, která následuje za operací,
    reprezentovanou dalším navštíveným městem.
END
```

7.1.4 Zjištění délky nalezené cesty

Tato část zahrnuje zjištění celkové délky nalezené cesty. Celková délka cesty u JSP nezáleží pouze na jednotlivých délkách cest mezi městy(na době trvání jednotlivých operací), ale i na vytíženosti strojů.

Pro reprezentaci strojů je vytvořena třída Stroj, která umožňuje provedení operací. Jedna instance této třídy představuje celou skupinu strojů, které provádí jeden typ operací. Na příkladu z obr. 7.1 by bylo potřeba tři instance třídy Stroj(M_1, M_2 a M_3). Instance M_1 může zahrnovat například dva stroje(dvě výrobní linky stejného typu), takže pak mohou být současně prováděny dvě operace, které musí být provedeny na stroji typu M_1:

```
// Třída Stroj:
class Stroj {
    int pocetLinek
    int[] linky = new int[pocetLinek]
    // procedura vracející čas zpracování požadované operace
    public int ProvedOperaci(int pocatecniCasOperace, int dobaTrvaniOperace)
    {
        Najdi linku s minimální vytížeností.

        IF vytíženost linky trvá déle než je pocatecniCasOperace BEGIN
            Vytíženost linky prodluž o dobaTrvaniOperace.
        END
        ELSE BEGIN
```

```
            Vytíženost linky nastav na
            pocatecniCasOperace + dobaTrvaniOperace.
        END

        Jako návrátovou hodnotu předej vytíženost linky.
    }
}
```

Nalezená cesta je reprezentována polem indexů jednotlivých měst tzn., že například `nalezenaCesta[1] = 3` znamená, že první město, kterým mravenec prošel bylo město s indexem 3. Pseudokód zjištění celkové délky cesty:

```
// Zjisti délku nalezené cesty:
// v poli typu int bude ukládán čas skončení jednotlivých operací daných úloh
int[] ulohy = new int[0];
// n ... počet úloh
FOR i = 1 TO n BEGIN
    ulohy[i] = 0
END
// m ... počet měst
FOR i = 1 TO m BEGIN
    Zjisti úlohu u, ke které patří operace reprezentovaná městem i.
    Zjisti dobu trvání t operace reprezentované městem i.
    Zjisti na jakém stroji má být provedena operace reprezentovaná městem i.
    Zalolej proceduru ProvedOperaci(ulohy[u], t) pro příslušnou instanci
    třídy stroj a návratovou hodnotu procedury ulož do ulohy[u].
END
Celková délka cesty je maximum z hodnot uložených v poli ulohy.
```

7.2 Řešení JSP pomocí GA

Základní kostra GA pro řešení JSP byla již popsána v kapitole Genetický algoritmus (GA). Zde budou podrobněji popsány procedury, které se vzhledem k odlišnostem jednotlivých optimalizačních úloh, musí řešit individuálně:

- `Vytvoř počáteční generaci n jedinců.` - Vygenerovat množinu náhodně vytvořených jedinců(řešení), tak aby vytvoření jedinci(řešení) náleželi do množiny všech možných řešení dané optimalizační úlohy.
- `Proveď operaci křížení.` - Vytvořit nového jedince z rodičovských jedinců. U TSP a JSP bude algoritmus operace křížení stejný. Algoritmus byl již popsán v kapitole Genetický algoritmus (GA), a proto zde nebude rozebírán.
- `Proveď operaci mutace.` - Mutace náhodného genu u JSP znamená prohození hodnot dvou genů.
- `Získej cestu reprezentovanou jedincem ze současné generace.` - u TSP a JSP jde v této proceduře o pouhé předání genů, které reprezentují města na cestě, proto zde nebude tato procedura popisována.

7.2.1 Vytvoření počáteční generace n jedinců

Tato část zahrnuje vytvořených množiny jedinců(řešení), tak aby vytvoření jedinci(řešení) náleželi do množiny všech možných řešení. U JSP je počet možných řešení omezen podmínkou, že operace v rámci dané úlohy provedeny v určeném pořadí, a tak je prostor možných řešení i vytvoření náhodného řešení omezeno.

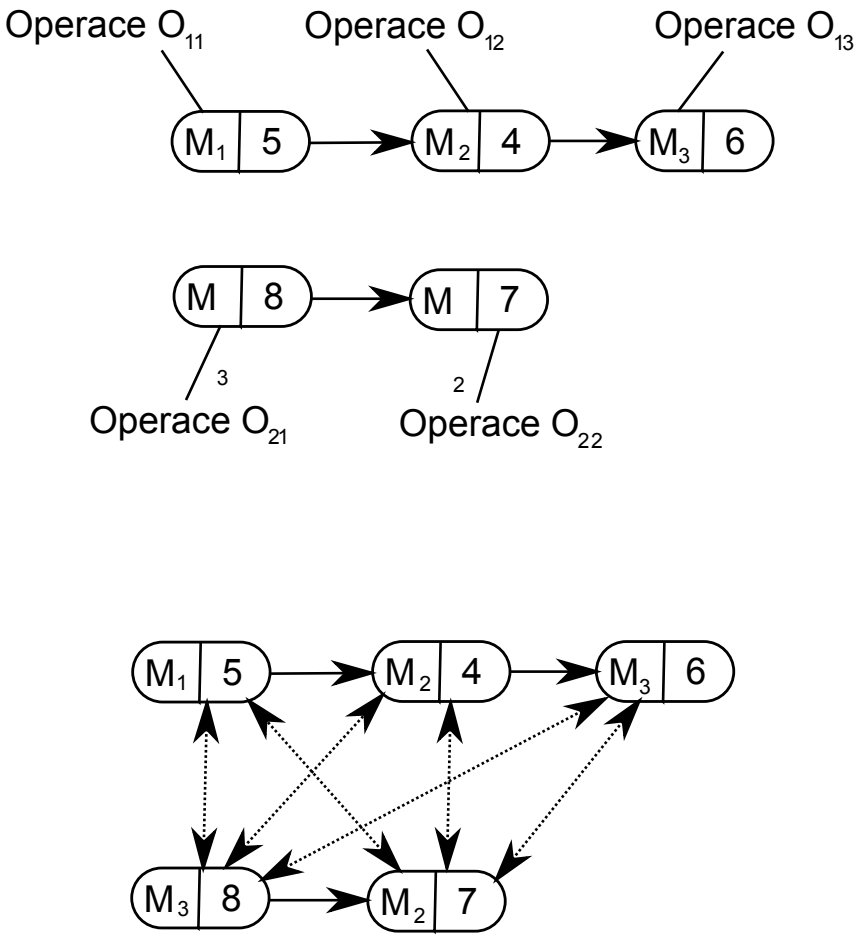
Geny, ze kterých jsou jedinci složeni, jsou reprezentovány polem typu integer. Jsou zde použity procedury, které byly popsány výše: `Vytvoř seznam nenavštívených měst.` a `Udělej přechod z aktuálního města do dalšího navštíveného města.`. Pseudokód pro vytvoření náhodného jedince:

```
// Vytvoř náhodného jedince:
// m ... počet měst
int[] geny = new int[m]
Vytvoř seznam nenavštívených měst.
FOR i = 1 TO m BEGIN
    Vyber náhodný indexSeznamu - číslo v rozsahu od 1 do počtu měst
    v seznamu nenavštívených měst.
    Najdi město M v seznamu nenavštívených měst, které je na pozici
    indexSeznamu.
    Přiřaď do geny[i] index města M.
    Udělej přechod z aktuálního města do dalšího navštíveného města, kde
    další navštívené město je město M.
END
```

7.2.2 Proveď operaci mutace

Tato část zahrnuje změnu genů jedinců(řešení), tak aby změnění jedinci(řešení) náleželi do množiny všech možných řešení. U JSP se musí kontrolovat, aby nedošlo k prohození pořadí operací v rámci jedné úlohy:

```
// Vytvoř náhodnou mutaci genů jedince:
// m ... počet měst
Vyber náhodný indexGenu - číslo v rozsahu od 1 do m - 1.
IF operace s indexem geny[indexGenu] nepatří do stejné úlohy jako operace
s indexem geny[indexGenu + 1] THEN BEGIN
    int temp = geny[indexGenu]
    geny[indexGenu] = geny[indexGenu + 1]
    geny[indexGenu + 1] = temp
END
```



Obrázek 7.3: Sestavení grafu z operací

Kapitola 8

Set Covering Problem (SCP)

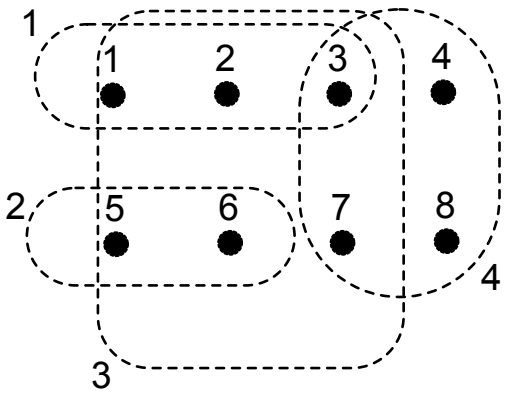
Set Covering Problem (dále jen SCP), problém pokrytí, je kombinatorická optimalizační úloha. Se SCP se setkáváme v praxi v nejrůznějších oborech jako například při plánování leteckých linek, při plánování rozmístění výrobních zařízení, nebo v logistice při rozvržení vytíženosti nákladních aut pro rozvoz [11]. SCP je definován takto:

- Máme množinu M, která obsahuje m bodů: $M = \{1, ..., m\}$
- Máme n množin, které jsou podmnožiny množiny M: $M_j \subseteq M$, kde $j \in \{1, ..., n\}$
- Každá množina má svou kladnou cenu C_j: $C_j \geq 1$, kde $j \in \{1, ..., n\}$
- Cílem je nalézt množinu $J \subset \{1, ..., n\}$, tak aby pokrývala množinu M: $\cup_{j \in J} M_j = M$ a současně, aby byla celková cena nejnižší: $Min \sum_{j \in J} M_j$

  Na obrázku 8.1 je znázorněno zadání úlohy. Je zde osm bodů a čtyři množiny. Pro uchování informace o tom, které body jsou obsaženy ve které množině, se používá matice o rozměrech $m \times n$ (m - počet bodů, n - počet množin), která se nazývá matice pokrytí. Hodnoty v matici nabývají hodnot 1 (daný bod je obsažen v dané množině) nebo 0 (daný bod není obsažen v dané množině) viz 8.1. Pokud by byla cena všech množin stejná (např.: $C_1 = C_2 = C_3 = C_4 = 1$), pak by řešení úlohy bylo $J = \{3, 4\}$, tzn. pro pokrytí by se použili množiny 3 a 4. Celková cena pokrytí by byla $1 + 1 = 2$. Pokud by ale platilo $C_3 > C_1 + C_2$ (např.: $C_1 = 1$, $C_2 = 1$, $C_3 = 3$), pak by řešení úlohy bylo $J = \{1, 2, 4\}$ a celková cena pokrytí by byla $1 + 1 + 1 = 3$.

Bod/Množina	1	2	3	4
1	1	0	1	0
2	1	0	1	0
3	1	0	1	1
4	0	0	0	1
5	0	1	1	0
6	0	1	1	0
7	0	0	1	1
8	0	0	0	1

Tabulka 8.1: Tabulka zachycující matici pokrytí



Obrázek 8.1: Body a množiny

Tato optimalizační úloha je NP-těžká a s rostoucím množstvím bodů a úloh roste náročnost úlohy exponenciálně, takže nelze použít metody založené na průchodu stavového prostoru.

8.1 Řešení SCP pomocí ACO

Základní kostra ACO algoritmu pro řešení SCP byla již popsána v kapitole Ant Colony Optimization (ACO). Zde budou podrobněji popsány procedury, které se vzhledem k odlišnostem jednotlivých optimalizačních úloh, musí být řešeny individuálně. Jedná se o tyto úlohy:

- `Inicializuj cesty mezi jednotlivými městy.` - Nastavení délky jednotlivých cest a počátečního množství feromonové stopy.
- `Vytvoř seznam nenavštívených měst.` - Vytvoří seznam všech měst, která jsou na počátku úlohy dostupné.
- `Udělej přechod z aktuálního města do dalšího navštíveného města.` - Správa seznamu nenavštívených měst, kterou je potřeba provést při sestavování cesty všemi městy.
- `Zjisti délku nalezené cesty.` - Zjištění délky cesty, a tím určení kvality řešení optimalizační úlohy.

8.1.1 Inicializace cest mezi jednotlivými městy

Tato část zahrnuje nastavení délky jednotlivých cest a počátečního množství feromonové stopy. Před vlastní inicializací cest je potřeba inicializovat města. To proběhne načtením parametrů měst ze souboru. U SCP města nereprezentují body, ale množiny. Délky cest mezi jednotlivými městy se mění s každým dalším navštíveným městem, proto nejsou délky cest

zpočátku vůbec počítány. Délky cest se vypočítají až po vybrání náhodného startovacího města:

$$d_{AB} = \frac{cost_B}{covVal_B}$$

$$covVal_B = \sum_{i \in cov(B,S)} minCost(i)$$

kde:

- d_{AB} je vzdálenost mezi městy A a B.
- $cost_B$ je cena množiny reprezentované městem B.
- $cov(B, S)$ je množina bodů, které obsahuje množina reprezentovaná bodem B, a které nejsou pokryté stávajícím řešením S.
- $minCost(i)$ je minimální cena množiny ze všech množin, které obsahují bod i.

Pro cesty je připravena třída Cesta, která zapouzdřuje proměnné délka cesty a množství feromonu. Cesty jsou uloženy do dvourozměrné matice $m * m$, kde m je počet měst. Matice obsahuje ukazatele na instance třídy Cesta. Toho je využito k zachycení faktu, že cesta mezi městy A a B je shodná jako cesta mezi městy B a A. A proto je v matici do obou příslušných polí uložen stejný odkaz na jednu instanci třídy Cesta:

```
// Inicializuj cesty mezi jednotlivými městy:
// m ... počet měst
Cesta[][] cesty = new Cesta[m][m];
FOR i = 1 TO m BEGIN
    FOR j = 1 TO m BEGIN
        IF i != j BEGIN
            Vytvoř instanci cesta třídy Cesta.
            Nastav délku cesty na -1.  // při inicializaci se nenastavuje
            Nastav počáteční množství feromonové stopy u instance cesta.
            cesty[i][j] = cesta;
            cesty[j][i] = cesta;
        END
    END
END
```

8.1.2 Vytvoření seznamu nenavštívených měst

Tato část zahrnuje vytvoření seznamu všech měst, která jsou na počátku úlohy dostupné. U TSP jsou to všechna města.

Pseudokód inicializace seznamu:

```
// Vytvoř seznam nenavštívených měst:
// m ... počet měst
FOR i = 1 TO m BEGIN
    Přidej město i do seznamu nenavštívených měst.
END
```

8.1.3 Přechod z aktuálního města do dalšího navštíveného města

Tato část zahrnuje správu seznamu nenavštívených měst při vytváření cesty. U SCP je nejdříve odebráno další navštívené město ze seznamu nenavštívených měst. Poté jsou ze seznamu postupně odebrány všechna města reprezentující množiny, které obsahují pouze body, které jsou již pokryty doposud nalezeným řešením(cestou). Při každém přechodu jsou také přepočítány vzdálenosti mezi dalším navštíveným městem a všemi zbývajícími městy v seznamu nenavštívených měst. Pseudokód správy seznamu:

```
// Udělej přechod z aktuálního města do dalšího navštíveného města:
Ze seznamu nenavštívených měst odeber další navštívené město.
// m ... počet měst v seznamu nenavštívených měst
FOR i = 1 TO m BEGIN
    Zjisti pro množinu reprezentovanou městem i počet bodů n, které
    množina obsahuje a současně nejsou pokryty množinami, které tvoří
    doposud nalezené řešení.
    IF n = 0 THEN odeber město i ze seznamu nenavštívených měst.
    ELSE Spočítej délku cesty mezi dalším navštíveným městem a městem i.
END
```

8.1.4 Zjištění délky nalezené cesty

Tato část zahrnuje zjištění celkové délky nalezené cesty. U SCP je celková délka cesty sumou cen všech množin, které jsou reprezentovány městy v nalezené cestě. Počet měst v nalezené cestě nemusí vždy odpovídat počtu všech měst, protože na pokrytí všech bodů nemusí být použity všechny množiny.

Nalezená cesta je reprezentována polem indexů jednotlivých měst tzn., že například `nalezenaCesta[1] = 3` znamená, že první město, kterým mravenec prošel bylo město s indexem 3. Pokud je počet měst v nalezené cestě menší než celkový počet měst, je na pozici chybějících měst -1. Pseudokód zjištění celkové délky cesty:

```
// Zjisti délku nalezené cesty:
celkovaDelkaCesty = 0
// m ... počet měst
FOR i = 1 TO m BEGIN
    IF nalezenaCesta[i] >= 0 THEN BEGIN
        celkovaDelkaCesty += cena množiny nalezenaCesta[i]
    END
END
```

8.2 Řešení SCP pomocí GA

Základní kostra GA pro řešení SCP byla již popsána v kapitole Genetický algoritmus (GA). Zde budou podrobněji popsány procedury, které se vzhledem k odlišnostem jednotlivých optimalizačních úloh, musí řešit individuálně:

- `Vytvoř počáteční generaci n jedinců.` - Vygenerovat množinu náhodně vytvořených jedinců(řešení), tak aby vytvoření jedinci(řešení) náleželi do množiny všech možných řešení dané optimalizační úlohy.

- `Proveď operaci křížení.` - Vytvořit nového jedince z rodičovských jedinců. U SCP se jedná o základní algoritmus, který byl již popsán v kapitole Genetický algoritmus (GA), a proto zde nebude rozebírán.
- `Proveď operaci mutace.` - Mutace náhodného genu u SCP znamená vybrat jinou množinu, pokud jiná existuje, která také obsahuje bod reprezentovaný mutovaným genem.
- `Získej cestu reprezentovanou jedincem ze současné generace.` - u SCP je v genech jedinců uchovávána informace o pokrytí jednotlivých bodů množinami. V této proceduře je tato informace převedena na cestu(posloupnost množin), tak aby vytvořená cesta odpovídala pokrytí jednotlivých bodů množinami.

8.2.1 Vytvoření počáteční generace n jedinců

Tato část zahrnuje vytvořených množiny jedinců(řešení), tak aby vytvoření jedinci(řešení) náleželi do množiny všech možných řešení. U SCP geny reprezentují body, které mají být pokryty, a nabývají hodnot od 1 do n, kde n je počet množin (např.: Jedinec 1 2 1 1 2 reprezentuje řešení, ve kterém jsou body 1,3 a 4 pokryty množinou 1 a body 2 a 5 jsou pokryty množinou 2). Proto je nejdříve vygenerována náhodná cesta. Z vygenerované cesty se odvodí odpovídající jedinec.

Geny, ze kterých jsou jedinci složeni, jsou reprezentovány polem typu integer. Jsou zde použity procedury, které byly popsány výše: `Vytvoř seznam nenavštívených měst.` a `Udělej přechod z aktuálního města do dalšího navštíveného města.`. Pseudokód pro vytvoření náhodného jedince:

```
// Vytvoř náhodnou cestu:
// m ... počet měst
int[] cesta = new int[m]
Vytvoř seznam nenavštívených měst.
FOR i = 1 TO m BEGIN
    Vyber náhodný indexSeznamu - číslo v rozsahu od 1 do počtu měst
    v seznamu nenavštívených měst.
    Najdi město M v seznamu nenavštívených měst, které je na pozici
    indexSeznamu.
    Přidej město M do cesta[i].
    Udělej přechod z aktuálního města do dalšího navštíveného města, kde
    další navštívené město je město M.
END
// Vytvoř náhodného jedince: k... počet všech bodů
int[] geny = new int[k]
FOR i = 1 TO m BEGIN
    IF cesta[i] >= 0 THEN BEGIN
        FOR všechny body, které množina reprezentovaná městem cesta[i]
        obsahuje BEGIN
            IF geny[indexBodu] < 0 THEN geny[indexBodu] = cesta[i]
        END
    END
END
```

8.2.2 Proveď operaci mutace

Tato část zahrnuje změnu genů jedinců(řešení), tak aby změnění jedinci(řešení) náleželi do množiny všech možných řešení. U SCP se změní množina, která pokrývá bod reprezentovaný genem, pokud takových množin existuje více. Na příkladu z obrázku 8.1 může třetí gen(reprezentující bod 3) nabývat hodnot 1, 3, nebo 4, protože ho pokrývají množiny 1, 3 a 4. Když bude třetí gen mít hodnotu 1, pak může zmutovat na hodnotu 3, nebo 4:

```
// Vytvoř náhodnou mutaci genů jedince:
// m ... počet měst
Vyber náhodný indexGenu - číslo v rozsahu od 1 do m.
IF bod s indexem indexGenu je pokryt více množinami THEN BEGIN
    Zvol náhodně množinu M, která je jiná než množina geny[indexGenu],
    a která pokrývá bod s indexem indexGenu.
    geny[indexGenu] = M
END
```

8.2.3 Získej cestu reprezentovanou jedincem ze současné generace

Tato část zahrnuje převedení informace o pokrytí jednotlivých bodů množinami do odpovídající cesty(posloupnosti množin):

```
// Získej cestu reprezentovanou jedincem ze současné generace:
// n ... počet měst(množin)
int[] cesta = new int[n]
boolean[] pouziteMnoziny = new boolean[n]
FOR i = 1 TO n BEGIN
    cesta[i] = -1
    pouziteMnoziny[i] = false
END
int indexCesty = 0
// n ... počet bodů
FOR 1 TO n BEGIN
    IF pouziteMnoziny[geny[i]] = false THEN BEGIN
        cesta[indexCesty] = geny[i]
        pouziteMnoziny[geny[i]] = true
        indexCesty++
    END
END
```

Cesta získaná tímto algoritmem nemusí obsahovat všechna města. Cesta může být kratší než je počet všech měst, a proto je zprava doplněna hodnotami -1(např.: $34-1-1$ je cesta reprezentující řešení pro úlohu z obr. 8.1)

Kapitola 9

Implementace a popis simulátoru

Implementačním jazykem je Java(JDK 1.6). Díky zvolení Javy je program přenositelný na různé platformy. Simulátor obsahuje implementované optimalizační algoritmy ACO algoritmus a GA. Optimalizační úlohy TSP, JSP a SCP nejsou implementované jako část simulátoru, ale jsou implementovány jako pluginy a do simulátoru jsou načteny za běhu programu. Toto schéma umožňuje implementaci nových optimalizačních úloh bez nutnosti úpravy kódu vlastního simulátoru.

9.1 Architekura simulátoru

Pro architekturu programu byl použit návrhový vzor model-view-controller(obr. 9.1):

- Model - Třídy sloužící pro reprezentaci problematiky ACO algoritmu a GA. Tvoří část simulátoru, která hledá nejlepší řešení optimalizačních úloh.
  - Simulator - Třída, která se stará o řízení simulací. Zpracovává akce zaslané z části Controller. Umožňuje správu parametrů simulací.
  - Enviroment - Třída, ve které je implementován ACO algoritmus a GA. Třída inicializuje a volá metody tříd, které jsou použity při hledání řešení optimalizačních úloh.
  - City - Třída pro reprezentaci měst, ze kterých jsou sestavovány nalezené cesty. Třída je použita u ACO algoritmu i u GA. Parametry jednotlivých měst jsou načteny ze souborů v inicializační části. Města a jejich parametry představují zadání optimalizačních úloh.
  - Route - Třída pro reprezentaci cesty. Cesta je posloupnost měst. Každá nalezená cesta je řešením dané optimalizační úlohy. Tato třída je použita u ACO algoritmu i u GA.
  - Ant - Třída použitá pro reprezentaci mravence u ACO algoritmu. V této třídě je implementována metoda pro nalezení cesty. Při hledání cesty se mravenec řídí pravděpodobnostním výběrem na základě množství feromonové stopy a délky jednotlivých úseků.
  - Chromozome - Tří pro reprezentaci jedince u GA.
  - Roulette - Třída použita pro modelování selekce jedinců na základě jejich fitness hodnoty u GA.

  - MinPathResult - Třída použitá pro reprezentaci nejlepšího nalezeného řešení. Třída je použita u ACO algoritmu i u GA.
  - AutomationTest - Třída pro automatické spuštění simulací s parametry v nastaveném rozmezí. Tato třída je určena pro testování vhodnosti parametrů u ACO algoritmu i u GA. Výsledky simalucí jsou ukládány do souboru typu csv.

- View - Třída používaná ke grafické prezentaci nalezeného řešení optimalizační úlohy uživateli.
  - Screen - Třída pro grafické zobrazení měst a nalezených cest, u ACO algoritmu jsou zobrazeny i feromonové stopy.
  - Report - Třída určená pro grafické zobrazení nalezeného řešení. Zobrazení je specifické pro každou optimalizační úlohu.

- Controller - Třída, která zpracovává uživatelské akce. Umožňuje ovládání simulátoru.
  - SimulatorController - Hlavní třída pro zpracování uživatelských akcí. Volá metody třídy Simulator.
  - SimulationSettingsACO - Třída umožňující nastavení parametrů simulací pro ACO algoritmus.
  - SimulationSettingsGA - Třída umožňující nastavení parametrů simulací pro GA.
  - AutomationTestSettingsACO - Třída umožňující nastavení parametrů automatických testů pro ACO algoritmus.
  - AutomationTestSettingsGA - Třída umožňující nastavení parametrů automatických testů pro GA.

9.2 Implementace optimalizačních úloh

Optimalizační úlohy jsou implementované jako pluginy a do simulátoru jsou načteny za běhu programu. Pro implementaci pluginů byl použit Java Plug-in Framework (JPF). V simulátoru je vytvořeno rozhraní PluginInterface, které definuje, jaké procedury musí být v pluginu implementovány, aby mohla být optimalizační úloha řešena v simulátoru pomocí ACO algoritmu a GA:

- GetScreenDimension - Načtení rozměrů obrazovky pro vykreslení stavu prostředí(měst, feromonových stop a nalezených cest) ze souboru.
- InitializeCities - Inicializace měst ze souboru.
- GetCityPosition - Vrací souřadnice obrazovky, kde má být vykresleno dané město.
- InitializePaths - Inicializace cest mezi jednotlivými městy.
- InitializeNotVisitedCities - Vytvoření seznamu nenavštívených měst.
- MakeStep - Úprava seznamu nenavštívených měst při přechodu do dalšího navštíveného města.
- GetLengthOfPath - Vrací celkovou délku dané cesty.

- ShowReport - Vykreslení nejlepšího nalezeného řešení na grafický výstup.
- CreateRandomChromozome - Vytvoření náhodného jedince pro GA.
- CreateChromozomeFromPath - Vytvoření jedince pro GA z dané cesty.
- GetPathFromChromozome - Vytvoření cesty z genové reprezentace jedince u GA.
- CrossOver - Operace křížení dvou jedinců u GA.
- Mutation - Operace mutace jedince u GA.
- OptimalizationA - Pro možnou implementaci lokální optimalizace.
- OptimalizationB - Pro možnou implementaci lokální optimalizace.

9.2.1 Implementace a použití pluginu

Pluginy jsou do simulátoru načítány z adresáře plugins. Plugin je vytvořen jako soubor zip(např.: TSP-1.0.0.zip). V zip souboru musí být soubor plugin.xml a adresář aco(obr. 9.2). Adresář aco obsahuje zkompilovanou třídu s optimalizační úlohou(obr. 9.3). Struktura souborů plugin.xml a TSPPlugin.java je popsána níže.

Struktura souboru plugin.xml:

```
<?xml version=„1.0\ ?>
<!DOCTYPE plugin PUBLIC „-//JPF//Java Plug-in Manifest 1.0\
 „http://jpf.sourceforge.net/plugin_1_0.dtd\>
<plugin id=„TSP\ version=„1.0.0\ class=„aco.TSPPlugin\>
<runtime>
<library id=„TSP\ path=„/\ type=„code\ />
</runtime>
</plugin>
```

9.3 Popis simulátoru

Pro inicializaci simulace je potřeba nejdříve zvolit optimalizační úlohu v menu Task, poté optimalizační algoritmus v menu Algorithm a nakonec načíst optimalizační úlohu ze souboru v menu Simulation(obr. 9.4).

Simulaci můžeme krokovat tlačítkem šipka, nebo spustit nastavený počet iterací tlačítkem s dvojitou šipkou. V záložce Simulation je zobrazeno hledání cesty. V záložce Result je vypsána nalezená cesta i s informacemi o délce, čase a počtu iterací. V záložce Result detail je grafická reprezentace nalezeného řešení. V menu Simulation lze nastavit parametry ACO algoritmu(Settings ACO) i GA(Settings GA). Nastavené parametry jsou platné až po nové inicializaci optimalizační úlohy ze souboru. Celý proces spuštění simulace můžeme popsat kroky:

- Zvolit optimalizační úlohu.
- Zvolit optimalizační algoritmus.
- Nastavit parametry příslušného optimalizačního algoritmu.

- Inicializovat optimalizační úlohu ze souboru.
- Spustit nebo krokovat simulaci.
- Pozorování dosažených výsledků.

V menu Test lze nastavit parametry pro automatický test ACO algoritmu(Settings ACO) i GA(Settings GA) a hlavně spustit automatický test(Run). Při spuštění automatického testu jsou postupně voleny všechny možné kombinace parametrů, jejichž rozsah byl nastaven v Settings ACO nebo Settings GA(záleží na tom, který algoritmus se testuje). Výsledek automatického testu je uložen do souboru se stejným názvem, jako má inicializační soubor optimalizační úlohy, ale s příponou csv. Pro spuštění automatického testu platí obdobný postup jako u simulace:

- Zvolit optimalizační úlohu.
- Zvolit optimalizační algoritmus.
- Nastavit rozsahy parametrů příslušného optimalizačního algoritmu a další parametry automatického testu.
- Inicializovat optimalizační úlohu ze souboru.
- Spustit automatický test.
- Pozorování dosažených výsledků uložených v souboru typu csv.

9.4 Implementovaná vylepšení

9.4.1 Inicializace pomocí metody Greedy search

Pro inicializaci ACO algoritmu i GA lze využít metodu založenou na algoritmu Greedy search. Nejprve jsou ze všech možných počátečních měst algoritmem Greedy search nalezena řešení dané optimalizační úlohy. Při inicializaci ACO algoritmu je soubor řešení nalezených pomocí Greedy search použit jako, kdyby to byli mravenci, kteří dokončili cestu všemi městy, a všichni položí na cesty odpovídající množství feromonu. Při inicializaci GA je soubor řešení nalezených pomocí Greedy search použit jako vzor pro vytvoření odpovídající části počáteční populace.

9.4.2 Lokální optimalizace

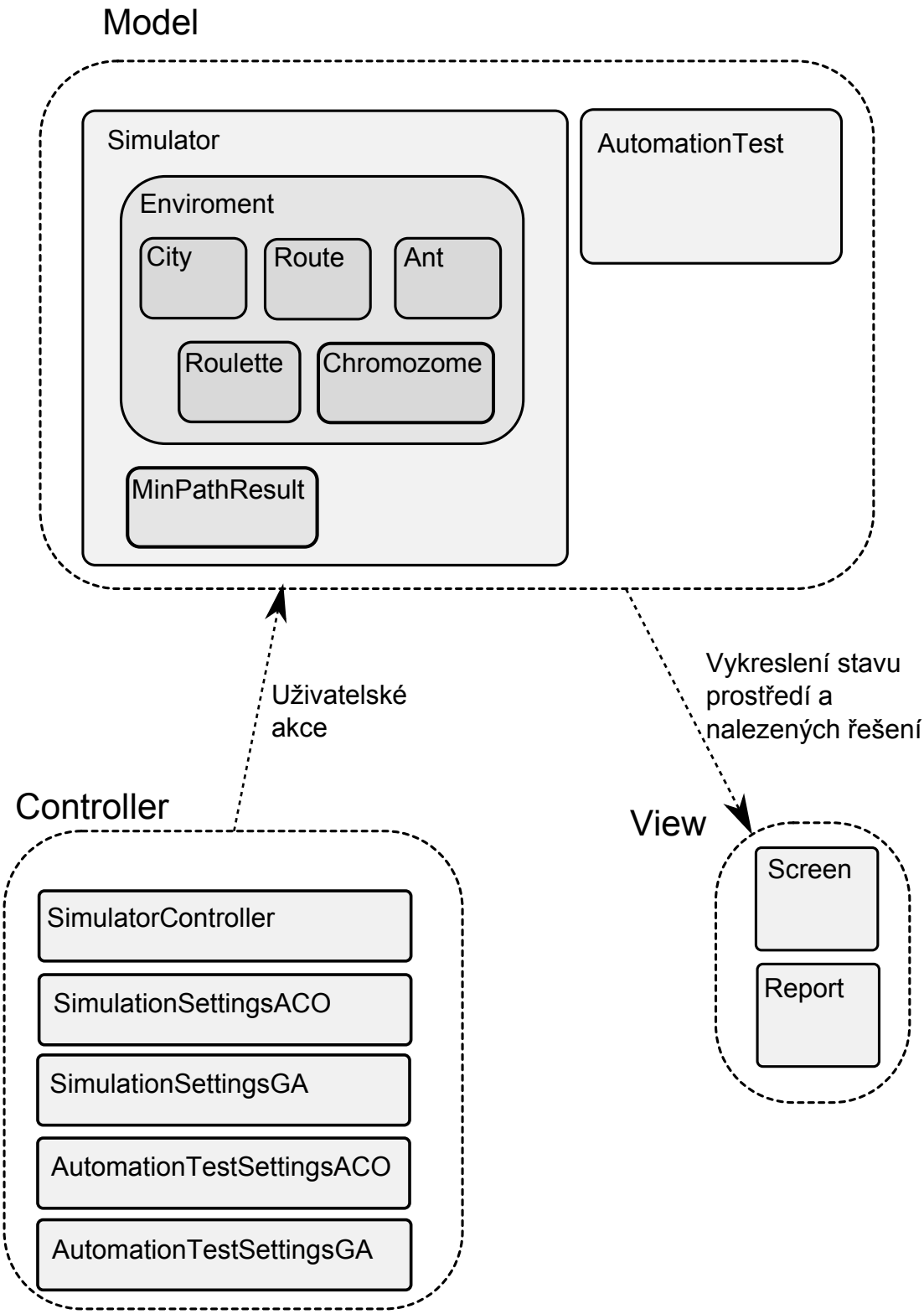
V pluginu jsou rezervovány dvě metody pro případnou lokální optimalizaci. Lokální optimalizace probíhá u ACO algoritmu ve chvíli, kdy mravenec projde všechna nenavštívená města a tím dokončí svou cestu. U GA lokální optimalizace probíhá na nově vytvořeném jedinci, ten může vzniknout operací křížení, nebo reprodukce, případně může zmutovat.

Lokální optimalizace u TSP

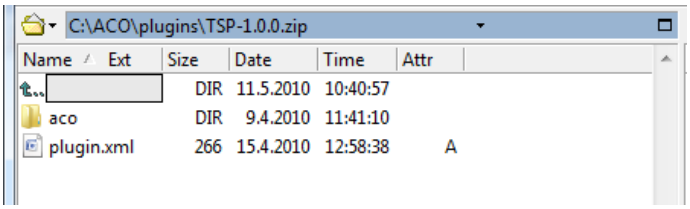
Lokální optimalizace u TSP probíhá na dvou párech po sobě jdoucích měst v nalezené cestě:

- Město A a město B jsou města, která jsou v nalezené cestě za sebou: Za městem A následuje bezprostředně město B.

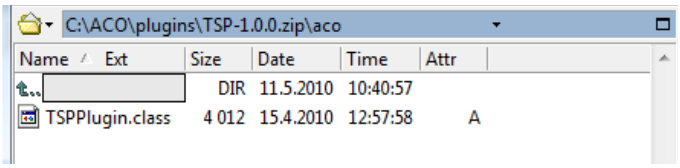
- Město C a město D jsou města, která jsou v nalezené cestě za sebou: Za městem C následuje bezprostředně město D.
- Jestliže platí $|AB| + |CD| > |AC| + |BD|$ prohoď pozici měst B a C v nalezené cestě.



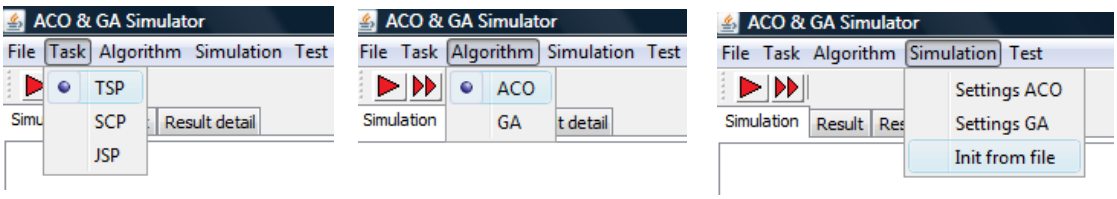
Obrázek 9.1: Architektura simulátoru



Obrázek 9.2: Struktura pluginu



Obrázek 9.3: Struktura pluginu



Obrázek 9.4: Volba úlohy a algoritmu, inicializace úlohy.

Kapitola 10

Výsledky experimentů

10.1 Vliv jednotlivých parametrů na výsledky ACO algoritmu

Vliv jednotlivých parametrů na ACO algoritmus bude testován na úloze obchodního cestujícího(TSP), protože je z implementovaných úloh nejnázornější.

10.1.1 Počet mravenců

Zjištění jaký má vliv počet mravenců na výsledky ACO algoritmu. Testování proběhlo na úlohách s 20, 30, 40 a 60 městy(TSP20.txt, TSP30.txt, TSP40.txt a TSP60.txt). Pro výpočet bylo použito nastavení bez počáteční inicializace pomocí metody Greedy search i bez lokální optimalizace.

Parametry:

```
DELTA_MAX/EVAPORATION_CONSTANT/SOLUTION_BEST_PUT_PHEROMONE/
ROUND_BEST_PUT_PHEROMONE/ALL_PUT_PHEROMONE/ALPHA/BETA/
MIN_PHEROMONE/MAX_PHEROMONE/INITIAL_PHEROMONE/GREADY_SEARCH/
OPT_PERCENT/OPT_TYPE/OPT_COUNT
```

Nastavené hodnoty:

```
21000.0/0.1/true/false/true/1.0/10.0/1.0/1000.0/50.0/false/0/0/0
```

Z naměřených hodnot, které jsou uvedeny v tabulce 10.1, jsou vytvořené grafy(obr. 10.1). Z tabulky a grafů je vidět, že řešení se stabilizuje při použití tří mravenců (při menším počtu měst) až devíti mravenců (při větším počtu měst).

10.1.2 Delta max

Zjištění jaký má vliv parametr Delta max na výsledky ACO algoritmu. Testování proběhlo na úlohách s 20 a 40 městy(TSP20.txt a TSP40.txt). Pro výpočet bylo použito nastavení bez počáteční inicializace pomocí metody Greedy search i bez lokální optimalizace.

Parametry:

```
COUNT_OF_ANTS/EVAPORATION_CONSTANT/SOLUTION_BEST_PUT_PHEROMONE/
ROUND_BEST_PUT_PHEROMONE/ALL_PUT_PHEROMONE/ALPHA/BETA/MIN_PHEROMONE/
MAX_PHEROMONE/INITIAL_PHEROMONE/GREADY_SEARCH/OPT_PERCENT/OPT_TYPE/OPT_COUNT
```

Počet mravenců	TSP20.txt	TSP30.txt	TSP40.txt	TSP60.txt
1	1338,108754	1555,164153	1726,70974	4979,855818
3	1314,171453	1540,397951	1713,15444	4897,093321
5	1308,864406	1540,397951	1695,92602	4856,688272
7	1310,577229	1540,397951	1704,776381	4869,257477
9	1307,985761	1540,397951	1700,315394	4835,847764
11	1307,941294	1540,397951	1699,698026	4849,07865
13	1308,819939	1540,397951	1695,198699	4871,604217
15	1308,864406	1540,397951	1694,74946	4835,18716
17	1307,985761	1540,397951	1692,403599	4841,585855
19	1309,698584	1540,397951	1679,650731	4838,422314

Tabulka 10.1: Tabulka závislosti délky nejlepší nalezené cesty na počtu mravenců

Nastavené hodnoty:

`7/0.1/true/false/true/1.0/10.0/1.0/1000000.0/5.0/false/0/0/0`

Z naměřených hodnot, které jsou uvedeny v tabulce 10.2, nelze jednoznačně určit hodnotu parametru Delta max, která by byla výhodnější než ostatní. Hodnotu parametru Delta max je tak možné libovolně volit s ohledem na minimální a maximální množství feromonové stopy, které může být na cesty položeno.

Delta max	TSP20.txt	TSP40.txt
1000	1307,896826	1690,592087
11000	1308,819939	1685,581107
21000	1307,985761	1686,359841
31000	1305,349826	1687,922499
41000	1307,018181	1686,802886
51000	1306,184004	1687,408742
61000	1307,852359	1685,907911
71000	1307,062649	1696,994238
81000	1306,228471	1682,449538
91000	1304,471181	1689,592754
101000	1306,228471	1685,825937

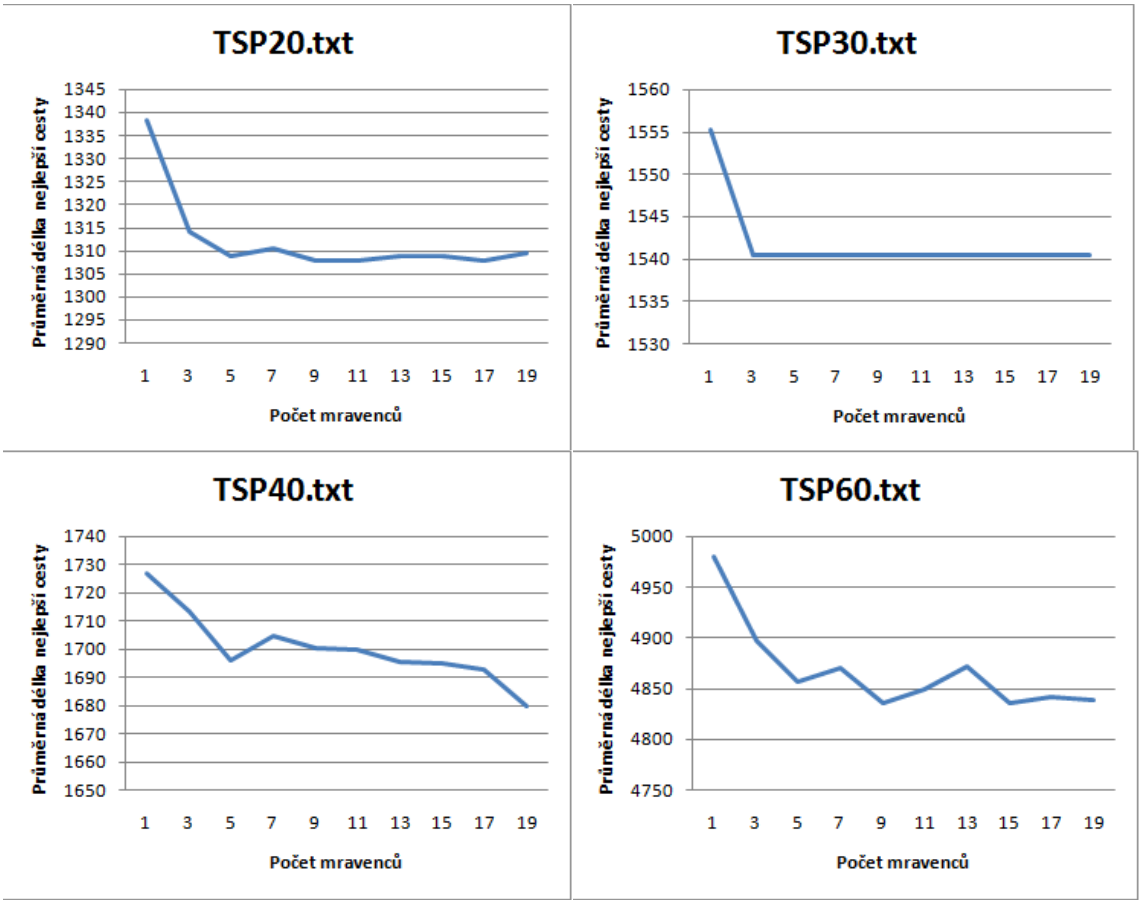
Tabulka 10.2: Tabulka závislosti délky nejlepší nalezené cesty na parametru Delta max

10.1.3 Evaporace

Zjištění jaký má vliv parametr Evaporation constant na výsledky ACO algoritmu. Testování proběhlo na úlohách s 20 a 40 městy(TSP20.txt a TSP40.txt). Pro výpočet bylo použito nastavení bez počáteční inicializace pomocí metody Greedy search i bez lokální optimalizace.

Parametry:

`COUNT_OF_ANTS/DELTA_MAX/SOLUTION_BEST_PUT_PHEROMONE/`



Obrázek 10.1: Grafy závislosti délky nejlepší nalezené cesty na počtu mravenců

```
ROUND_BEST_PUT_PHEROMONE/ALL_PUT_PHEROMONE/ALPHA/BETA/
MIN_PHEROMONE/MAX_PHEROMONE/INITIAL_PHEROMONE/GREADY_SEARCH/
OPT_PERCENT/OPT_TYPE/OPT_COUNT
```

Nastavené hodnoty:

```
7/21000.0/true/false/true/1.0/10.0/1.0/1000000.0/5.0/false/0/0/0
```

Z naměřených hodnot, které jsou uvedeny v tabulce 10.3, vyplývá, že se zvyšující se hodnotou parametru Evaporation constant se zhoršuje nalezené řešení, proto je dobré volit Evaporation constant v okolo hodnoty $0,1$.

10.1.4 Typ ACO algoritmu

Typ ACO algoritmu je dám kombinací tří parametrů rozhodujících o tom, kteří mravenci budou pokládat feromonové stopy na nalezené cesty. Zde bude zjištěno jaký vliv má tato kombinace parametrů na výsledky ACO algoritmu. Testování proběhlo na úlohách s 20, 30, 40 a 60 městy(TSP20.txt, TSP30.txt, TSP40.txt a TSP60.txt). Pro výpočet bylo použito nastavení bez počáteční inicializace pomocí metody Greedy search i bez lokální optimalizace.

Parametry:

Evaporation constant	TSP20.txt	TSP40.txt
0,5	1309,698584	1685,365173
0,5	1309,743051	1685,102641
0,5	1307,896826	1692,174982
0,5	1307,107116	1689,41173
0,5	1308,775471	1699,689071
0,5	1307,107116	1693,027299
0,5	1314,756384	1697,592196
0,5	1308,775471	1695,597653
0,5	1310,621696	1691,491853

Tabulka 10.3: Tabulka závislosti délky nejlepší nalezené cesty na parametru Evaporation constant

`COUNT_OF_ANTS/DELTA_MAX/EVAPORATION_CONSTANT/ALPHA/BETA/MIN_PHEROMONE/MAX_PHEROMONE/INITIAL_PHEROMONE/GREADY_SEARCH/OPT_PERCENT/OPT_TYPE/OPT_COUNT`

Nastavené hodnoty:

`7/21000.0/0.1/1.0/10.0/1.0/1000000.0/5.0/false/0/0/0`

Z naměřených hodnot, které jsou uvedeny v tabulce 10.4, vyplývá, že dvě kombinace nastavených parametrů dosahují lepších výsledků než ostatní kombinace:

- Feromon pokládá mravenec, který našel doposud nejlepší cestu(SB = 1).
- Feromon pokládá mravenec, který našel doposud nejlepší cestu(SB = 1). + Feromon pokládá mravenec, který našel nejlepší cestu v dané iterci(RB = 1).

SB	RB	ALL	TSP20.txt	TSP30.txt	TSP40.txt	TSP60.txt
0	0	1	1308,864406	1540,39795141597	1703,54796	4888,999566
0	1	0	1309,654116	1540,39795141597	1692,899227	4874,328194
0	1	1	1308,864406	1540,39795141597	1701,333051	4885,893553
1	0	0	1304,784505	1540,39795141597	1675,967979	4760,906625
1	0	1	1309,743051	1540,39795141597	1693,02857	4881,053546
1	1	0	1302,580489	1540,39795141597	1676,75954	4790,34447
1	1	1	1306,184004	1540,39795141597	1691,081215	4855,983451

Tabulka 10.4: Tabulka závislosti délky nejlepší nalezené cesty na typu ACO algoritmu

10.1.5 Parametry Alpha a Beta

Zjištění jaký mají vliv parametry Alpha a Beta na výsledky ACO algoritmu. Testování proběhlo na úlohách s 20 a 40 městy(TSP20.txt a TSP40.txt). Pro výpočet bylo použito nastavení bez počáteční inicializace pomocí metody Greedy search i bez lokální optimalizace.

Parametry:

`COUNT_OF_ANTS/DELTA_MAX/EVAPORATION_CONSTANT/SOLUTION_BEST_PUT_PHEROMONE/ROUND_BEST_PUT_PHEROMONE/ALL_PUT_PHEROMONE/MIN_PHEROMONE/MAX_PHEROMONE/INITIAL_PHEROMONE/GREADY_SEARCH/OPT_PERCENT/OPT_TYPE/OPT_COUNT`

Nastavené hodnoty:

`7/500/21000.0/0.1/true/false/true/1.0/1000000.0/5.0/false/0/0/0`

Z naměřených hodnot, které jsou uvedeny v tabulce 10.5, jsou vytvořené grafy(obr. 10.2). Z tabulky a grafů je vidět, že kvalita nalezené cesty velmi záleží na vzájemném nastavení parametrů Alpha a Beta. Na grafech je vidět, že při příliš velkém rozdílu mezi hodnotou parametru Alpha a hodnotou parametru Beta, se kvalita řešení zhoršuje. U obou úloh je optimální volit hodnotu parametru Beta o čtyři(tři) větší než je hodnota parametru Alpha ($Beta = Alpha + 4$).

Alpha	Beta	TSP20.txt	TSP40.txt
1	5	1302,669424	1680,590762
1	8	1304,426714	1686,174096
1	11	1307,941294	1694,538089
1	14	1310,577229	1711,927151
1	17	1315,626237	1729,540801
3	5	1317,731108	1695,945059
3	8	1308,717076	1706,426365
3	11	1317,333013	1711,756111
3	14	1318,568633	1728,189526
3	17	1311,250148	1740,079572
5	5	1329,392261	1711,628517
5	8	1315,393181	1712,306707
5	11	1322,703356	1707,581331
5	14	1324,521368	1714,095507
5	17	1315,304246	1731,48293
7	5	1328,81874	1706,266827
7	8	1321,649297	1713,54557
7	11	1317,047608	1711,846604
7	14	1319,998184	1720,90696
7	17	1325,798082	1739,430158

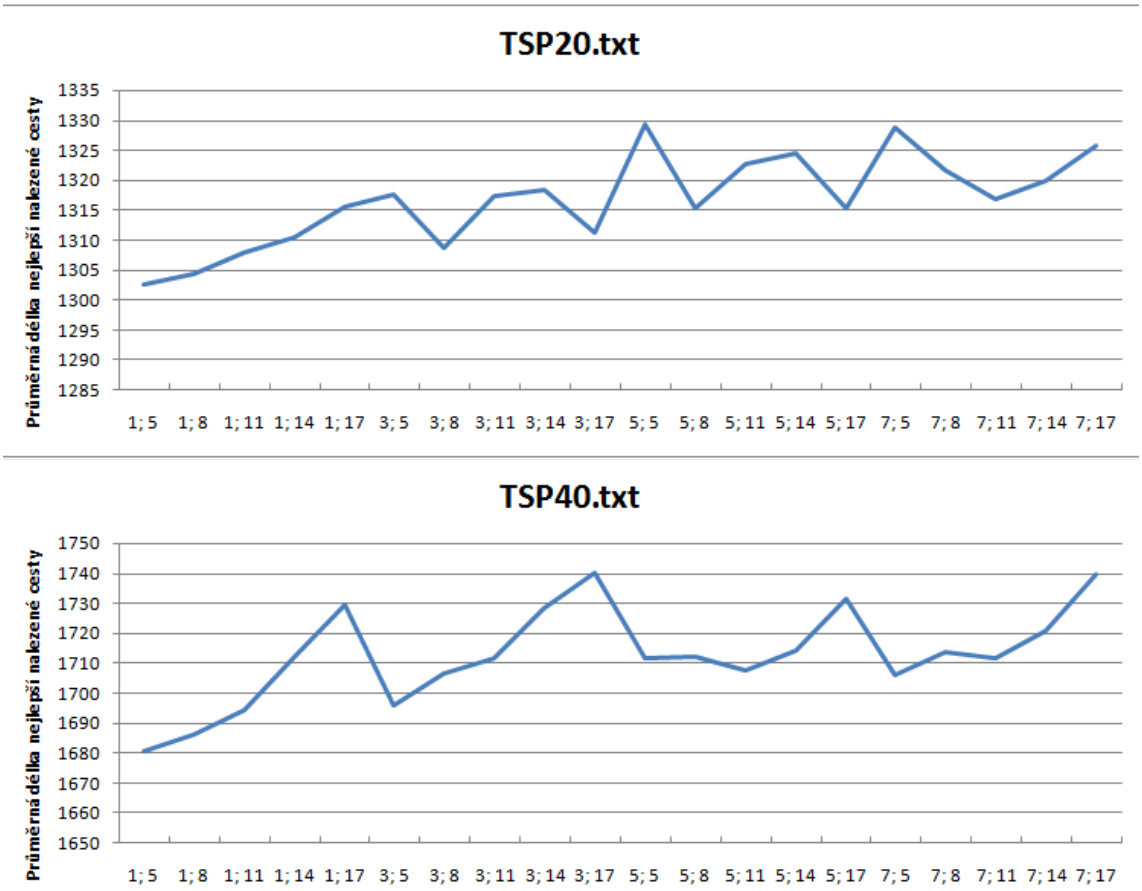
Tabulka 10.5: Tabulka závislosti délky nejlepší nalezené cesty na parametrech Alpha a Beta

10.1.6 Počáteční inicializace pomocí metody Greedy search

Zjištění jaký má vliv počáteční inicializace pomocí metody Greedy search na výsledky ACO algoritmu. Testování proběhlo na úlohách s 20, 30, 40 a 60 městy(TSP20.txt, TSP30.txt, TSP40.txt a TSP60.txt). Pro výpočet bylo použito nastavení bez lokální optimalizace.

Parametry:

`COUNT_OF_ANTS/DELTA_MAX/EVAPORATION_CONSTANT/SOLUTION_BEST_PUT_PHEROMONE/ROUND_BEST_PUT_PHEROMONE/ALL_PUT_PHEROMONE/ALPHA/BETA/MIN_PHEROMONE/MAX_PHEROMONE/INITIAL_PHEROMONE/OPT_PERCENT/OPT_TYPE/OPT_COUNT`



Obrázek 10.2: Grafy závislosti délky nejlepší nalezené cesty na parametrech Alpha a Beta

Nastavené hodnoty:

`7/21000.0/0.1/true/false/true/1.0/4.0/1.0/1000000.0/5.0/0/0/0`

Z naměřených hodnot, které jsou uvedeny v tabulce 10.7, vyplývá, že při použití počáteční inicializace pomocí metody Greedy search dostáváme mírně horší výsledky, než kdybychom jí nepoužili.

Greedy search	TSP20.txt	TSP30.txt	TSP40.txt	TSP60.txt
0	1302,536022	1540,397951	1679,303734	4786,175875
1	1303,548069	1540,397951	1679,578724	4794,372394

Tabulka 10.6: Tabulka závislosti délky nejlepší nalezené cesty na počáteční inicializace pomocí metody Greedy search

10.1.7 Lokální optimalizace

Zjištění jaký má vliv lokální optimalizace na výsledky ACO algoritmu. Testování proběhlo na úlohách s 40, 60 a 70 městy(TSP40.txt, TSP60.txt a TSP70.txt).

Parametry:

```
COUNT_OF_ANTS/MAX_TIME/DELTA_MAX/EVAPORATION_CONSTANT/
SOLUTION_BEST_PUT_PHEROMONE/ROUND_BEST_PUT_PHEROMONE/ALL_PUT_PHEROMONE/
ALPHA/BETA/MIN_PHEROMONE/MAX_PHEROMONE/INITIAL_PHEROMONE/GREADY_SEARCH/
OPT_PERCENT/OPT_TYPE
```

Nastavené hodnoty:

```
7/700/21000.0/0.1/true/false/true/1.0/4.0/1.0/1000000.0/5.0/false/100/0
```

Z naměřených hodnot, které jsou uvedeny v tabulce 10.7, vyplývá, že použití lokální optimalizace u ACO algoritmu nepřináší žádný výrazně lepší výsledek.

Počet lokálních optimalizací	TSP40.txt	TSP60.txt	TSP70.txt
0	1677,518569	4770,658577	5113,671246
2	1679,036626	4769,289418	5140,656156
4	1676,506531	4760,96042	5123,093859
6	1681,096781	4770,6478	5112,960171
8	1678,024588	4761,296121	5151,055246
10	1676,902053	4760,277937	5106,488552
12	1678,646998	4755,956232	5110,241346
14	1677,905676	4755,676995	5125,857349

Tabulka 10.7: Tabulka závislosti délky nejlepší nalezené cesty na počtu lokálních optimalizací

10.2 Vliv jednotlivých parametrů na výsledky GA

Vliv jednotlivých parametrů na ACO algoritmus bude testován na úloze obchodního cestujícího(TSP), protože je z implementovaných úloh nejnázornější.

10.2.1 Počáteční inicializace pomocí metody Greedy search

Zjištění jaký má vliv počáteční inicializace pomocí metody Greedy search na výsledky GA. Testování proběhlo na úlohách s 20 a 30 městy(TSP20.txt a TSP30.txt). Pro výpočet bylo použito nastavení bez lokální optimalizace.

Parametry:

```
COUNT_OF_CHROMOZOMES/MAX_GENERATION/DELTA_MAX_GA/CROSSOVER_FACTOR/
MUTATION_FACTOR/OPT_PERCENT/OPT_TYPE/OPT_COUNT
```

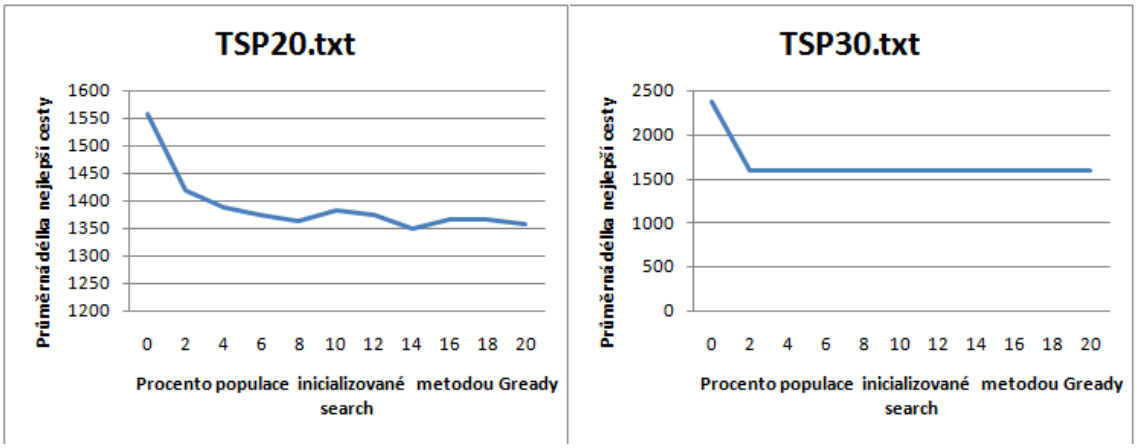
Nastavené hodnoty:

```
2000/1000/210000.0/0.9/0.1/0/0/0
```

Z naměřených hodnot, které jsou uvedeny v tabulce 10.8, vyplývá, že při použití počáteční inicializace pomocí metody Greedy search dostáváme výrazně lepší výsledky, než kdybychom jí nepoužili. Stačí inicializovat i jen několik málo procent celkové populace(do 10%) viz. grafy 10.3.

Populace inicializovaná pomocí Greedy search [%]	TSP20.txt	TSP30.txt
0	1556,135042	2366,098704
2	1417,971622	1591,348508
4	1388,066452	1592,389678
6	1375,213301	1596,149442
8	1365,124347	1593,444044
10	1382,144808	1595,810101
12	1373,578895	1593,713637
14	1350,064573	1590,915345
16	1366,294991	1590,791937
18	1367,91414	1590,915345
20	1357,946805	1589,487068

Tabulka 10.8: Tabulka závislosti délky nejlepší nalezené cesty na počáteční inicializace pomocí metody Greedy search



Obrázek 10.3: Grafy závislosti délky nejlepší nalezené cesty na počáteční inicializace pomocí metody Greedy search

10.2.2 Lokální optimalizace

Zjištění jaký má vliv lokální optimalizace na výsledky GA. Testování proběhlo na úlohách s 20, 30, 40 a 60 městy(TSP20.txt, TSP30.txt, TSP40.txt, a TSP60.txt).

Parametry:

```
COUNT_OF_CHROMOZOMES/MAX_GENERATION/DELTA_MAX_GA/INITIAL_GREADY/
CROSSOVER_FACTOR/MUTATION_FACTOR/OPT_PERCENT/OPT_TYPE
```

Nastavené hodnoty:

```
1000/100/21000.0/10.0/0.9/0.1/100/0
```

Z naměřených hodnot, které jsou uvedeny v tabulce 10.9, vyplývá, že při použití lokální optimalizace dostáváme nejlepší řešení. Počet lokálních optimalizací, které jsou potřeba k dosažení nelepšího řešení, roste s počtem měst.

Počet lokálních optimalizací	TSP20.txt	TSP30.txt	TSP40.txt	TS60.txt
0	1379,018302	1591,178837	1807,390332	5087,958718
2	1302,269217	1540,397951	1675,517208	4799,820178
4	1302,269217	1540,397951	1674,988474	4679,653691
6	1302,269217	1540,397951	1674,988474	4642,841973
8	1302,269217	1540,397951	1674,988474	4642,841973
10	1302,269217	1540,397951	1674,988474	4642,841973
12	1302,269217	1540,397951	1674,988474	4642,841973
14	1302,269217	1540,397951	1674,988474	4642,841973

Tabulka 10.9: Tabulka závislosti délky nejlepší nalezené cesty na lokální optimalizaci

10.2.3 Parametry Crossover factor a Mutation factor

Zjištění jaký mají vliv parametry Crossover factor a Mutation factor na výsledky GA. Testování proběhlo na úlohách s 60 městy(TSP60.txt). Pro výpočet bylo použito nastavení s lokální optimalizací.

Parametry:

```
COUNT_OF_CHROMOZOMES/MAX_GENERATION/DELTA_MAX_GA/INITIAL_GREADY/
OPT_PERCENT/OPT_TYPE/OPT_COUNT
```

Nastavené hodnoty:

```
1000/200/21000.0/10.0/100/0/6
```

Z naměřených hodnot, které jsou uvedeny v tabulce 10.10, vyplývá, že s rostoucí hodnotou parametru Mutation factor rychle klesá efektivita GA (zvyšuje se iterace, ve které je nalezena nejlepší cesta), viz. grafy 10.4. Dále je patrné, že s rostoucí hodnotou parametru Crossover factor také mírně klesá efektivity. To lze vysvětlit tím, že mutace a křížení mění jedince, na kterých proběhla lokální optimalizace a tím do určité míry degradují efekt lokální optimalizace. Snižování hodnoty parametru Crossover factor, ale neznamená nutně zvyšující se efektivitu GA(viz. tabulka 10.11). Jako nejlepší se v tomto případě jeví hodnota parametru Crossover factor $= 0,5$.

10.3 Porovnání ACO algoritmu a GA na implementovaných optimalizačních úlohách

10.3.1 Traveling Salesman Problem (TSP)

Porovnání výsledků dosažených ACO algoritmem a GA na úloze obchodního cestujícího bylo provedeno na úlohách se 20, 40, 60 a 100 městy (TSP20.txt, TSP40.txt, TSP60.txt a TSP100.txt).

Parametry pro ACO algoritmus:

```
COUNT_OF_ANTS/MAX_ITERATION/DELTA_MAX/EVAPORATION_CONSTANT/
SOLUTION_BEST_PUT_PHEROMONE/ROUND_BEST_PUT_PHEROMONE/ALL_PUT_PHEROMONE/
ALPHA/BETA/MIN_PHEROMONE/MAX_PHEROMONE/INITIAL_PHEROMONE/GREADY_SEARCH/
OPT_PERCENT/OPT_TYPE/OPT_COUNT
```

Nastavené hodnoty pro ACO algoritmus:

```
5/1000/21000.0/0.1/true/false/false/1.0/5.0/1.0/1000.0/100.0/0.4/false/0/0/0
```

Parametry pro GA:

```
COUNT_OF_CHROMOZOMES/MAX_GENERATION/DELTA_MAX_GA/INITIAL_GREADY/
CROSSOVER_FACTOR/MUTATION_FACTOR/OPT_PERCENT/OPT_TYPE/OPT_COUNT
```

Nastavené hodnoty pro GA:

```
1000/2000/21000.0/10.0/0.5/0.0/100/0/2
```

Z dat uvedených v tabulkách 10.12, 10.14, **??** a 10.15 je vidět, že GA s počáteční inicializací pomocí metody Greedy search a lokální optimalizací dosahuje podstatně lepší výsledků než ACO algoritmus.

10.3.2 Job Shop Scheduling Problem (JSP)

Porovnání výsledků dosažených ACO algoritmem a GA na JSP bylo provedeno na úlohách JSP110.txt a JSP440.txt. Úloha JSP110.txt obsahuje 12 úloh, které jsou složeny ze 110 operací. Úloha JSP440.txt obsahuje 48 úloh, které jsou složeny ze 440 operací.

Parametry pro ACO algoritmus:

```
COUNT_OF_ANTS/MAX_ITERATION/DELTA_MAX/EVAPORATION_CONSTANT/
SOLUTION_BEST_PUT_PHEROMONE/ROUND_BEST_PUT_PHEROMONE/ALL_PUT_PHEROMONE/
ALPHA/BETA/MIN_PHEROMONE/MAX_PHEROMONE/INITIAL_PHEROMONE/GREADY_SEARCH/
OPT_PERCENT/OPT_TYPE/OPT_COUNT
```

Nastavené hodnoty pro ACO algoritmus:

```
2/1000/300.0/0.1/true/false/false/1.0/1.0/1.0/1000.0/100.0/0.0/false/0/0/0
```

Parametry pro GA:

```
COUNT_OF_CHROMOZOMES/MAX_GENERATION/DELTA_MAX_GA/INITIAL_GREADY/
CROSSOVER_FACTOR/MUTATION_FACTOR/OPT_PERCENT/OPT_TYPE/OPT_COUNT
```

Nastavené hodnoty pro GA:

```
1000/1000/1000.0/10.0/0.9/0.1/0/0/0
```

Z dat uvedených v tabulkách 10.16 a 10.17 je vidět, že GA s počáteční inicializací pomocí metody Greedy search dosahuje lepších výsledků než ACO algoritmus.

10.3.3 Set Covering Problem (SCP)

Porovnání výsledků dosažených ACO algoritmem a GA na SCP bylo provedeno na úloze SCP63.txt, která obsahuje 40 bodů a 63 množin.

Parametry pro ACO algoritmus:

```
COUNT_OF_ANTS/MAX_ITERATION/DELTA_MAX/EVAPORATION_CONSTANT/
SOLUTION_BEST_PUT_PHEROMONE/ROUND_BEST_PUT_PHEROMONE/ALL_PUT_PHEROMONE/
ALPHA/BETA/MIN_PHEROMONE/MAX_PHEROMONE/INITIAL_PHEROMONE/GREADY_SEARCH/
OPT_PERCENT/OPT_TYPE/OPT_COUNT
```

Nastavené hodnoty pro ACO algoritmus:

```
2/1000/300.0/0.1/true/false/false/1.0/1.0/1.0/1000.0/100.0/0.0/false/0/0/0
```

Parametry pro GA:

```
COUNT_OF_CHROMOZOMES/MAX_GENERATION/DELTA_MAX_GA/INITIAL_GREADY/
CROSSOVER_FACTOR/MUTATION_FACTOR/OPT_PERCENT/OPT_TYPE/OPT_COUNT
```

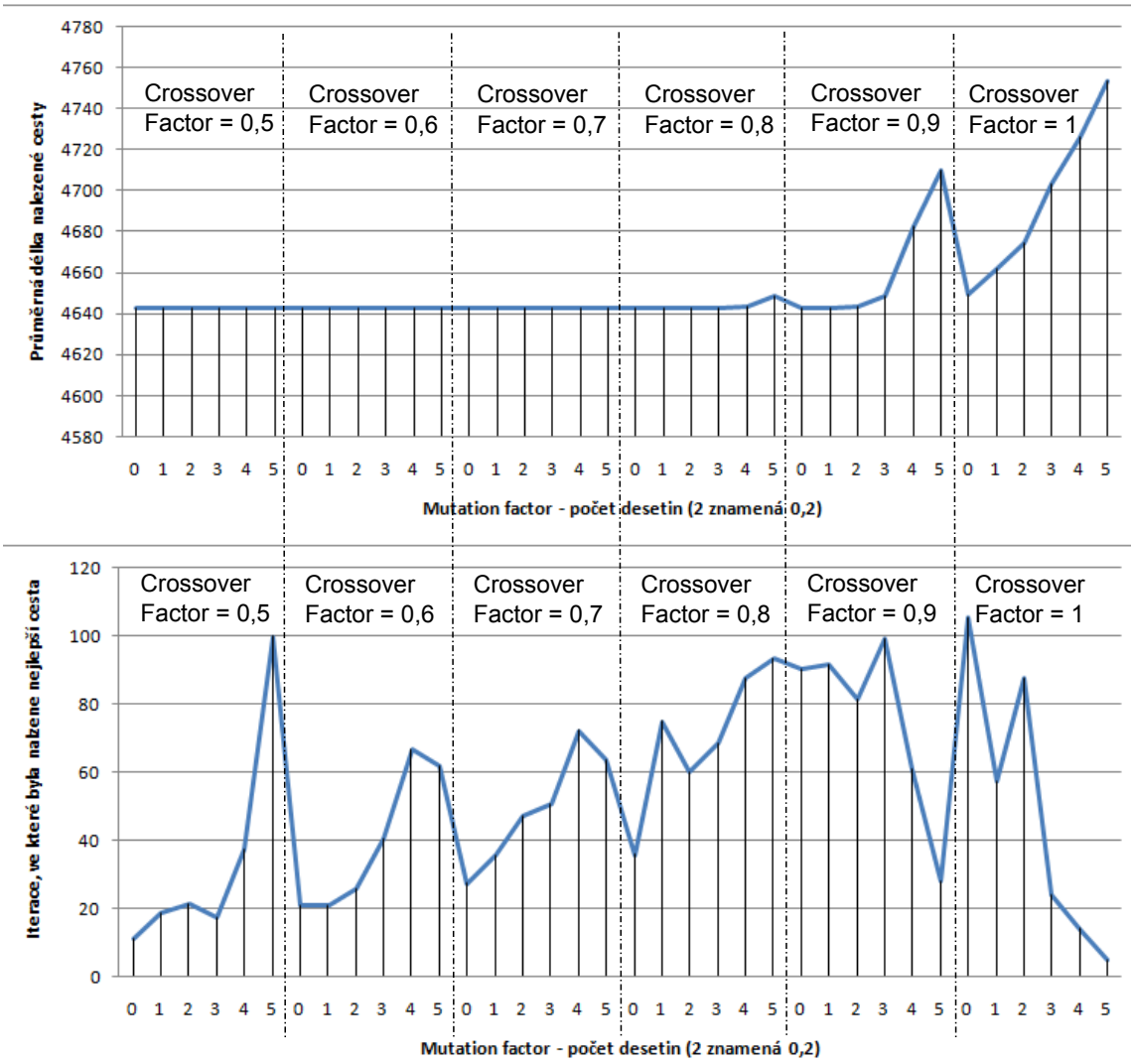
Nastavené hodnoty pro GA:

```
1000/1000/1000.0/10.0/0.9/0.1/0/0/0
```

Z dat uvedených v tabulce 10.18 je vidět, že GA s počáteční inicializací pomocí metody Greedy search dosahuje lepších výsledků než ACO algoritmus.

Crossover	Mutation	Nejlepší cesta(průměr)	Iterace(průměr)
0,5	0	4642,841973	11,3
0,5	0,1	4642,841973	18,6
0,5	0,2	4642,841973	21,3
0,5	0,3	4642,841973	17,5
0,5	0,4	4642,841973	37,5
0,5	0,5	4642,841973	99,9
0,6	0	4642,841973	20,9
0,6	0,1	4642,841973	20,8
0,6	0,2	4642,841973	25,8
0,6	0,3	4642,841973	40,4
0,6	0,4	4642,841973	66,6
0,6	0,5	4642,841973	61,8
0,7	0	4642,841973	27,4
0,7	0,1	4642,841973	35,6
0,7	0,2	4642,841973	47,2
0,7	0,3	4642,841973	50,7
0,7	0,4	4642,841973	72,3
0,7	0,5	4642,841973	63,8
0,8	0	4642,841973	35,8
0,8	0,1	4642,841973	75
0,8	0,2	4642,841973	60
0,8	0,3	4642,841973	68,5
0,8	0,4	4643,524732	87,7
0,8	0,5	4648,770578	93,3
0,9	0	4642,841973	90,2
0,9	0,1	4642,841973	91,7
0,9	0,2	4643,524732	81,4
0,9	0,3	4648,870924	99,4
0,9	0,4	4682,77437	61,1
0,9	0,5	4710,13504	28,1
1	0	4649,568364	105,5
1	0,1	4662,112002	57,5
1	0,2	4674,676974	87,8
1	0,3	4703,058619	23,9
1	0,4	4725,974665	14,1
1	0,5	4753,465017	5

Tabulka 10.10: Tabulka závislosti délky nejlepší nalezené cesty na parametrech Crossover factor a Mutation factor



Obrázek 10.4: Grafy závislosti délky nejlepší nalezené cesty na parametrech Crossover factor a Mutation factor

Crossover	Mutation	Nejlepší cesta(průměr)	Iterace(průměr)
0	0	4712,34950954456	1
0,1	0	4642,84197263482	63,5
0,2	0	4642,84197263482	23
0,3	0	4642,84197263482	38
0,4	0	4642,84197263482	33
0,5	0	4642,84197263482	20,5
0,6	0	4642,84197263482	39
0,7	0	4642,84197263482	22
0,8	0	4642,84197263482	41,5

Tabulka 10.11: Tabulka závislosti délky nejlepší nalezené cesty na parametrech Crossover factor a Mutation factor

Čas [ms]	ACO	GA
20	1422,58442914488	1302,26921746366
50	1314,92674192889	1302,26921746366
100	1302,26921746366	1302,26921746366

Tabulka 10.12: Tabulka s nejlepšími nalezenými cestami pomocí ACO algoritmu a GA na úloze se 20 městy

Čas [ms]	ACO	GA
100	1793,22458184756	1674,98847424775
200	1746,24031108379	1674,98847424775
300	1684,809548	1674,98847424775
400	1674,98847424775	1674,98847424775

Tabulka 10.13: Tabulka s nejlepšími nalezenými cestami pomocí ACO algoritmu a GA na úloze se 40 městy

Čas [ms]	ACO	GA
100	1793,22458184756	1674,98847424775
200	1746,24031108379	1674,98847424775
300	1684,809548	1674,98847424775
400	1674,98847424775	1674,98847424775

Tabulka 10.14: Tabulka s nejlepšími nalezenými cestami pomocí ACO algoritmu a GA na úloze se 40 městy

Čas [ms]	ACO	GA
500	8471,64090533926	6877,01251566727
1000	8334,61588483643	6845,40170942383
1500	8064,00812680947	6838,0726386693

Tabulka 10.15: Tabulka s nejlepšími nalezenými cestami pomocí ACO algoritmu a GA na úloze se 100 městy

Čas [ms]	Délka ACO(průměr)	Nejlepší ACO	Délka GA(průměr)	Nejlepší GA
20	249,3	242	232	232
40	248,3	235	232	232
60	244	232	232	232

Tabulka 10.16: Tabulka s nejlepšími nalezenými cestami pomocí ACO algoritmu a GA na úloze se JSP110.txt

Čas [ms]	Délka ACO(průměr)	Nejlepší ACO	Délka GA(průměr)	Nejlepší GA
100	1054,35	1003	933,2	928
200	1044,05	1001	930,8	207
300	1030,3	997	930,3	928

Tabulka 10.17: Tabulka s nejlepšími nalezenými cestami pomocí ACO algoritmu a GA na úloze se JSP440.txt

Čas [ms]	Délka ACO(průměr)	Délka GA(průměr)
100	88,15	78
200	87,15	78
300	86,6	78

Tabulka 10.18: Tabulka s nejlepšími nalezenými cestami pomocí ACO algoritmu a GA na úloze se JSP440.txt

Kapitola 11

Závěr

Implementoval jsem ACO algoritmus a GA pro řešení optimalizačních úloh. Dále jsem implementoval tři optimalizační úlohy. Zaměřil jsem se na úlohu obchodního cestujícího, na které jsem otestoval vliv jednotlivých parametrů ACO algoritmu a GA na kvalitu nalezeného řešení. Na všech optimalizačních úlohách jsem porovnal výsledky dosažené pomocí ACO algoritmu a GA. Ve všech případech lépe dopadl GA. Horší výsledky ACO algoritmu přisuzuji tomu, že ACO algoritmus musel být zachován v obecné podobě, aby zvládl řešení různých úloh, a proto nemohl být speciálně upraven pro danou optimalizační úlohu. Implementovaný GA dosahuje v kombinaci s počáteční inicializací pomocí metody Greedy search a lokální optimalizací velmi dobrých výsledků. Především u úlohy obchodního cestujícího dosahuje GA, při použití lokální optimalizace, kvalitních řešení za velmi krátký čas.

Vytvořil jsem simulátor pro řešení optimalizačních kombinatorických úloh pomocí ACO algoritmu a GA, do kterého lze prostřednictvím pluginů doprogramovat další optimalizační úlohy. Možným rozšířením je tak možnost implementovat další optimalizační úlohy. Další možnost zlepšení simulátoru spočívá v efektivnější implementaci ACO algoritmu prostřednictvím větší specializace ACO algoritmu na jednotlivé optimalizační úlohy. Další zajímavé rozšíření simulátoru představuje propojení ACO algoritmu a GA do jednoho hybridního algoritmu. Například je možné použít ACO algoritmus jako inicializační část GA.

Literatura

[1] IEEE Swarm Intelligence Symposium 2008. Swarm intelligence symposium [online]. http://www.computelligence.org/sis/2008. [Květen 2010].

[2] YANEER BAR-YAM. A mathematical theory of strong emergence using multiscale variety [online]. http://necsi.org/research/multiscale/MultiscaleEmergence.pdf. [Květen 2010].

[3] Pavel Burian. Uplatnění agentových kolon při ovládání a optimalizaci průmyslových procesů [online]. http://www.cssi.cz/cssi/uplatneni-agentovych-kolon-pri-ovladani-optimalizaci-prumyslovych-procesu-1. [Květen 2010].

[4] John W. Chinneck. *Practical Optimization: A Gentle Introduction*. Carleton University Press, 2000.

[5] Marco Dorigo. *Optimization, Learning and Natural Algorithms*. Politecnico di Milano, 1992.

[6] A.A. Afify D.T. Pham. Manufacturing cell formation using the bees algorithm [online]. http://conference.iproms.org/sites/conference.iproms.org/files/papers2007/122.pdf. [Květen 2010].

[7] Francis Heylighen. The science of self-organization and adaptivity [online]. http://pespmc1.vub.ac.be/papers/EOLSS-Self-Organiz.pdf. [Květen 2010].

[8] Steven Johnson. *Emergence, The Connected Lives of Ants, Brains, Cities, and Software*. Simon & Schuster, 2002. ISBN 0-684-86876-8.

[9] Niklas Kohl. Solving the world's largest crew scheduling problem [online]. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.129.1044&rep=rep1&type=pdf. [Květen 2010].

[10] Oleg Kovářík. Ant colony optimization with castes [online]. http://ai.ms.mff.cuni.cz/~sui/ants.pdf. [Květen 2010].

[11] Christian Blum Marco Dorigo, Mauro Birattari. *Ant colony optimization and swarm intelligence*. Simon & Schuster, 2002. ISBN 0-684-86876-8.

[12] Arnold Neumaier. Global optimization [online]. http://www.mat.univie.ac.at/~neum/glopt.html. [Květen 2010].

[13] Marek Obitko. Genetic algorithms [online]. http://www.obitko.com/tutorials/genetic-algorithms. [Květen 2010].

[14] Petr Peringer. *Modelování a simulace, studijní opora*. 2006.

[15] Lieven Vandenberghe Stephen Boyd. *Convex Optimization*. Cambridge University Press, 2004. ISBN 978-0-521-83378-3.