VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2016 Bc. Jan Němec

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

EFEKTIVITA EVOLUČNÍCH ALGORITMŮ
EFFECTIVENESS OF EVOLUTIONARY ALGORITHMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Jan Němec**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Radek Fujdiak**
SUPERVISOR

BRNO 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Telekomunikační a informační technika**
Ústav telekomunikací

Student: Bc. Jan Němec ***ID:*** 147447
Ročník: 2 ***Akademický rok:*** 2015/16

NÁZEV TÉMATU:

Efektivita evolučních algoritmů

POKYNY PRO VYPRACOVÁNÍ:

V diplomové práci student implementuje vybraný evoluční algoritmus společně s vybranými složitostními problémy (z kategorie NP-úplný). Proběhnou rozsáhlá experimentální měření efektivity vybraného algoritmu a to na základě různých vstupních parametrů, které mohou ovlivňovat efektivitu výpočtů. Student bude mít za úkol analyzovat možnosti vybraného algoritmu a navrhnout nejlepší možné řešení v rámci vstupních parametrů. Práce bude podložena kvalitní odbornou a vědeckou literaturou, vybraný algoritmus a jeho implementace, společně s jeho parametry by měly být vyčerpávajícím způsobem prakticky i matematicky popsány.

DOPORUČENÁ LITERATURA:

[1] VOLNÁ E. "Evoluční algoritmy a neuronové sítě". S. 1-152. Ostrava 2012.

[2] HYNEK J. "Genetické algoritmy a genetické programování". Grada. S. 1-179. 2008.

[3] POHLHEIM H. "Evolutionary Algorithms: Overview, Methods and Operators". GEATbx version 3.8. Dec 2006.

Termín zadání: 1.2.2016 ***Termín odevzdání:*** 25.5.2016

Vedoucí práce: Ing. Radek Fujdiak
Konzultant diplomové práce:

doc. Ing. Jiří Mišurec, CSc., *předseda oborové rady*

ABSTRAKT

Tato diplomová práce se zabývá evolučními algoritmy. Jejím úkolem je vybrat vhodný evoluční algoritmus, který bude rešit vhodný problém. V tomto případě se jedná o genetický algoritmus, který bude řešit problém obchodního cestujícího. Výsledkem této diplomové práce bude implementace algoritmu, určení jeho ideálního nastavení a změření výsledků pro různá vstupní data.

KLÍČOVÁ SLOVA

Evoluční algoritmy, genetické algoritmy, problém obchodního cestujícího, optimalizace, JAVA

ABSTRACT

This master's thesis is focused on evolutionary algorithms. The goal of this thesis is to chooche a proper algorithm which will solve a chosen problem. In this case the chosen algorithm is the genetic algorithm and the chosen problem is the travelling salesman problem. The result of this thesis will be implementation of the algorithm, finding the proper setup and lastly the measurment of the results for various input data.

KEYWORDS

Evolutionary algoritms, genetic algoritms, travelling salesman problem, optimization, JAVA

NĚMEC, Jan *Efektivita evolučních algoritmů*: diplomová práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2016. 64 s. Vedoucí práce byl Ing. Radek Fujdiak

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Efektivita evolučních algoritmů“ jsem vypracoval(a) samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor(ka) uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil(a) autorská práva třetích osob, zejména jsem nezasáhl(a) nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom(a) následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora(-ky)

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu Ing. Radkovi Fujdiakovi, za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

Brno ....

podpis autora(-ky)

Faculty of Electrical Engineering
and Communication
Brno University of Technology
Purkynova 118, CZ-61200 Brno
Czech Republic

http://www.six.feec.vutbr.cz

PODĚKOVÁNÍ

Výzkum popsaný v této diplomové práci byl realizován v laboratořích podpořených z projektu SIX; registrační číslo CZ.1.05/2.1.00/03.0072, operační program Výzkum a vývoj pro inovace.

Brno . . . . . .

. . . . . .

podpis autora(-ky)

EVROPSKÁ UNIE
EVROPSKÝ FOND PRO REGIONÁLNÍ ROZVOJ
INVESTICE DO VAŠÍ BUDOUCNOSTI

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

ÚVOD

I přes to, že žijeme v moderní době, kdy vývoj technologií jde stále více dopředu ve všech směrech, obzvláště pak v informatice, tak i přes to dnes existují komplikované problémy, u nichž buď nevíme, jak dosáhnout ideálního řešení, nebo postup známe, ale řešení takového problému pak zabírá nepřijatelně velkou dobu. Výzkumníci se tedy snaží stále přicházet s novými postupy, jak takové problémy řešit.

Jedním z nápadů se při řešení takových problémů bylo inspirovat přírodou kolem nás, konkrétně všemi živými organismy, které se v přeneseném slova smyslu také zabývají problémem, na který není známý postup řešení, ale za to je znám výsledek – přežití. Jednotlivé organismy kolem nás se snaží neustále za pomocí různých biologických postupů vyvíjet tak, aby měly co nejvyšší šanci na přežití.

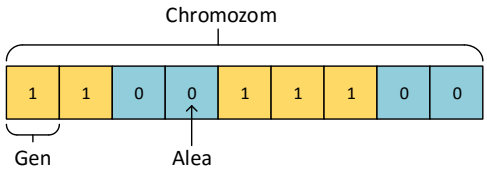
Z těchto přírodních procesů se tedy vyvinul speciální směr v infromatice, který spadá do oboru umělé inteligence, zvaný jako evoluční algoritmy. Evoluční algoritmy tedy napodobují evoluční procesy živých organismů za účelem řešení komplexních problémů. Evoluční algoritmus je však pouze jakýsi vzor, jak problém řešit a existuje mnoho druhů typů konkrétních implementací. Evoluční algoritmy se v dnešní době používají pro řešení problémů z různých oborů, od umění až po chemii [9].

Tato diplomová práce se zabývá výkonností evolučních algoritmů. V teoretické části budou popsány obecně evoluční algoritmy a následně se práce zaměří na konkrétní typ implementace, v tomto případě genetické algoritmy. Na nich budou vysvětleny principy různých funkcí z pohledu implementace genetických algoritmů do programovacího jazyka. Následně se práce bude věnovat problému, který genetický algoritmus bude řešit, popíše jeho základní vlastnosti a problematiku řešení. V praktické části se práce bude zabývat implementací genetického algoritmu do vybraného programovacího jazyka, popisu jednotlivých stěžejních částí algoritmu. Následně se práce zaměří na určení optimálního nastavení algoritmu, které bude dosaženo pomocí testování a následně ověření tohoto nastavení na různých vstupních datech. V poslední části budou tyto výsledky porovnány s ostatními pracemi, které se touto problematikou taktéž zabývaly.

1 EVOLUČNÍ ALGORITMY

Evoluční algoritmy vycházejí z Darwinovy teorie evoluce, která pojednává o tom, že každá nová populace jedinců se vyvíjí z té předchozí, přičemž jedinci, kteří mají lepší genetický materiál mají vyšší šanci, že předají svůj genetický materiál a budou mít více potomků a tím se populace bude vyvíjet tak, že dokáže přežít ve svém prostředí.

Evoluční algoritmy jsou tedy takové algoritmy, které simulují průběh evoluce za účelem vyřešení problému. K tomu používají evoluční operátory, které se nazývají selekce, křížení a mutace. Mezi základní entity v evolučních algoritmech patří chromozom, který se skládá z genů, přičemž hodnota, které gen může nabývat se označuje jako alea, grafické znázornění chromozomu pak lze vidět na Obr. 1.1. Skupina chromozomů se pak označuje jako populace [2].



Obr. 1.1: Chromozom

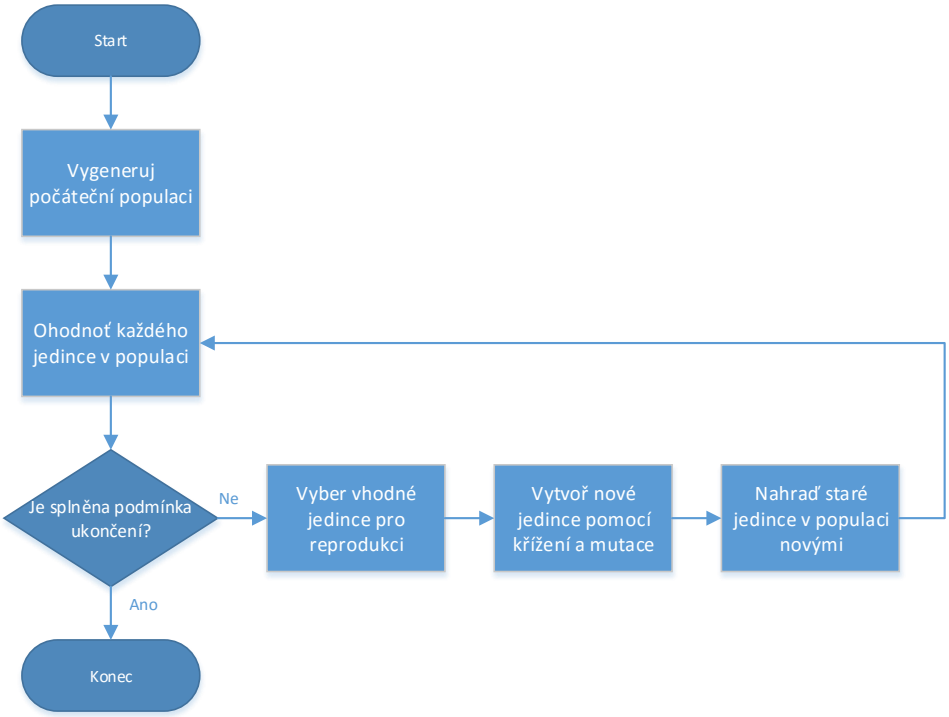
Pod pojmem evoluční algoritmus si nelze představit jeden exaktní algoritmus, ale spíše se jedná o soubor různých algoritmů, které mají stejný základ, ale liší se ve věcech jako je reprezentace chromozomu na úrovni datové struktury a nebo v odlišnostech při implementaci evolučních operátorů. Základní schéma evolučních algoritmů může být viděno na Obr. 1.2.

Mezi dnes nejpoužívanější evoluční algoritmy patří tyto [1]:

- genetické algoritmy,
- genetické programování,
- evoluční strategie,
- evoluční programování.

Evoluční algoritmy mají využití ve spoustě oborů, mezi obvyklé úlohy, při jejichž řešení se používají evoluční algoritmy, je například optimalizace, ale dají se s jejich pomocí řešit i další problémy, například návrh elektrických obvodů, plánování různých úloh nebo i například vytváření nových chemických sloučenin a porovnávání jejich vlastností [9]. Například americká společnost NASA použila evoluční algoritmy k nalezení několika nových designů antén, kdy vědci znali potřebné parametry, jaké by anténa měla mít, ale nevěděli přesně jak jich dosáhnout.

Zjednodušeně by se dalo říct, že evoluční algoritmy se používají v případě, kdy je znám požadovaný výsledek, ale neví se, jak přesně k němu dojít. Vzhledem k tomu, že



Obr. 1.2: Základní schéma evolučních algoritmů

evoluční algoritmy jsou svým způsobem náhodné však nikdy není jisté, že se najde to nejoptimálnější řešení. Správnost řešení, které je výsledkem, je velmi závislé na správném popsání tzv. fitness funkce, která slouží právě k ohodnocení jednotlivých řešení, které vznikají evolucí počáteční populace [2].

1.1 Fitness

Fitness je hodnota, která vyjadřuje kvalitu každého chromozomu, tedy , jestli je chromozom vhodným kandidátem pro řešení problému. V evolučních algoritmech neexistuje žádný přesný postup, jak fitness funkci navrhnout, správnost fitness funkce záleží na podstatě řešeného problému. To, jak je funkce navrhnutá je jedním z nejpodstatnějších kriterií pro získání nejvhodnějšího řešení problému. Pokud je funkce navrhnutá špatně, pak nebudou získané výsledky ani zdaleka optimálním řešením[2].

1.2 Evoluční operátory

Jak již bylo zmíněno, evoluční algoritmy používají několik základních evolučních operátorů, které byly odvozeny z evoluční teorie. Nutno dodat, že tyto operátory jsou

zjednodušenou verzí podobných procesů, které opravdu probíhájí v evoluci živých organizmů.

1.2.1 Selekce a elitismus

Operátor selekce slouží k výběru jedinců z populace, kteří se pak stanou rodiči a předají dál svůj genetický materiál a nebo automaticky postoupí do další generace. Stejně jako u fitness funkce, i správné nastavení selekce rozhoduje o celkovém výsledku algoritmu. Tímto nastavením se myslí, jací jedinci budou vybráni k postupu do další generace. Dalo by se očekávat, že pokud z aktuální populace budou vybíráni jenom nejsilnější jedinci, bude nalezeno optimální řešení rychleji, avšak v tomto případě hrozí, že nakonec populace uvízne v lokálním maximu a nebude tak nalezeno nejoptimálnější řešení. Proto je dobré do selekce zahrnout i jedince se slabším ohodnocením, jelikož mohou přinést do genomu prospěšné změny sloužící k nalezení nejlepšího řešení. Podíl silných a slabších chromozomů v populaci se dá ovlivňovat tzv. selekčním tlakem. Čím vyšší selekční tlak je tím víc se nachází v populaci pouze silnější chromozomy na úkor těch slabších. Jak již bylo řečeno, tímto se nabízí možné riziko uvíznutí v lokálním maximu z důvodu malé různorodosti genetického materiálu celé populace. Pokud by byl ovšem selekční tlak příliš nízký, tak by populace nekonvergovala k optimálnímu řešení dostatečně rychle nebo vůbec. Je tedy důležité mít správně vyvážený tlak, aby populace konvergovala k silnějším jedincům ale ne na úkor likvidace těch slabších.

Elitismus je pak jakousi zvláštní formou selekce, slouží k výběru nejlepších chromozomů z aktuální generece, které automaticky přejdou do další generace [7].

1.2.2 Křížení

Křížení v EA, stejně jako v reálném světě, že rodičové zkříží svůj genetický materiál a tím vytvoří potomka. V reálném světě jsou rodičové 2, ašak u GA může být rodičů více. Zkombinováním svého genetického materiálu tak vznikají potomci, kteří můžou mít vyšší šanci na přežití (v případě EA na řešení problému) ale můžou vzniknout i potomci, jejichž genetický materiál je slabí a jsou odsouzeni k záhubě.

U evolučních algoritmů je několik typů křížení, které se používají v závislosti na vybraném algoritmu nebo kódování chromozomu [7].

1.2.3 Mutace

Mutace narozdíl od křížení nevytváří nový genetický materiál kombinací více chromozomů, ale pouze mění jednotlivé aley v genech stávajicích chromozomů. Pokud bychom používali v běhu EA pouze křížení, může nastat situace, kdy řešení uvázne

v lokálním maximu a není schopné ho překonat, přičemž ze lokálním maximem se mouhou nacházet daleko optimálnější řešení. Použitím mutace můžeme překonávat lokální maxima a hledat tak nejoptimálnější řešení.

Metoda mutace se tedy může používat v kombinaci s křížením, ale nikdy ne samostatně, jelikož pouze náhodné změny v chromozomech k nalezení optimálního řešení nestačí.

2 GENETICKÉ ALGORITMY

Genetické algoritmy vychází z principů evolučních algoritmů. Stejně jako v EA je jejich základní jednotkou chromozom, který se skládá z genů a skupina chromozomů pak tvoří populaci. Také používají operátory selekce, křížení a mutace. Chromozom u genetických algoritmů se zpravidla kóduje jako binární řetězec, ale u určitých problémů se dají chromozomy kódovat například i jako celočíselné hodnoty.

Jejich hlavní výhodou je poměrná jednoduchost implementace. V praxi se GA používají k řešení optimalizačních problémů (jako například TSP), ale dají se použít i na jiné problémy. Narozdíl od gradientních metod, které pracují s jedním řešením, které postupně vylepšují, pracují s GA s větší množinou řešení. Tím se za cenu většího nároku na výpočetní výkon zkracuje doba hledání řešení.

Co se týče řešení, tak jednou z nevýhod GA je to, že né vždy jsou schopny nalézt to nejoptimálnější řešení, ale jsou schopny nalézt pouze přibližné řešení, které se však od optimálního nemusí tolik lišit.

2.1 Operátory genetických algoritmů

V kapitole o EA byly teoreticky popsány evoluční operátory. Tato část se bude také věnovat operáturům avšak už z pohledu GA s podrobnějším popisem procesů.

2.1.1 Selekce

U GA se u selekce nejčastěji používají dva základní modely – turnajová selekce a selekce založená na principu ruletového kola, která se pak dělí ještě na další dve metody.

Ruletová selekce založená na hodnotě fitness

Jak již název napovídá, tento typ selekce si lze představit jako hru na ruletovém kole s tím rozdílem, že jednotlivé dílky nejsou stejně velké. Nejprve dojde k součtu hodnot fitness všech jedinců v populaci a je tak určena velikost ruletového kola.

Následně dojde k přiřazování dílů jednotlivým chromozomů. Chromozomy, které mají vyšší fitness dostanou na ruletovém kole větší dílek, než ty s nižší hodnotou, jednoduše řečeno, cím vyšší má chromozom fitness, tím větší je jeho dílek. Následně se kolo roztočí a je vybranám chromozom, na jehož dílku se pomyslná kulička zastaví. Z toho vyplívá, že chromozomy s větší hodnotou fitness budou vybrány vícekrát.

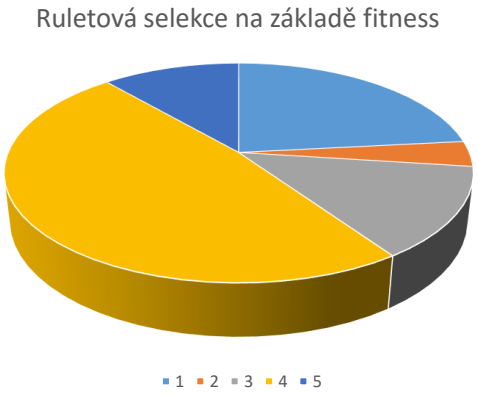
Toto lze vidět ná následujícím příkladu. V tab. 2.1 máme 5 chromozomů 1-5, a každý má svoji hodnotu fitness. Následně se dle vzorce2.1, kde *Pi* značí pravděpodobnost selekce, *fi* fitness chromozomu a *fj* součet všech fitness v populace

$$Pi = \frac{fi}{\sum_{j=1}^{n} fj} \tag{2.1}$$

Tab. 2.1: Ruletová selekce na základě fitness

Chromozom	Fitness	Pi
1	305	0,23318
2	49	0,037462
3	175	0,133792
4	631	0,482416
5	148	0,11315
	Sum = 1308	

určí, jakou pravděpodobnost výběru má každý chromozom. Na Obr. 2.1 pak lze vidět grafickou reprezentaci ruletového kola, kde velikost jednotlicých dílů odpovídá právě šanci jednotlivých chromozomů na selekci.



Obr. 2.1: Ruletová selekce na základě fitness

Ruletou se otáčí tak dlouho, dokud není vybrán požadovaný počet chromozomů. Pravděpodobnost výběru jednotlivého chromozomu se dát určit tak, že hodnotu fitness vybraného chromozomu podělím součtem hodnot fitness všech algoritmů viz. vzorec 2.1.

Nevýhodou této metody je to, že pokud se v populaci nachází chromoz s výrazně vyšší hodnotou fitness než ostatní, pak zabere na ruletovém kole tolik místa, že je prakticky nemožné, aby byly vybrány další chromozomy [3].

Ruletová selekce založená na pořadí

Narozdíl od ruletové selekce na základě fitness funkce se u této metody používá systém ohodnocení. Nejprve dojde k seřazení všech chromozomů v populaci na základě jejich fitness a následně se jejich šance na výběr určují právě podle jejich pořadí. Pokud máme populaci o velikosti N, pak nejhorší chromozom na základě fitness bude první v pořadí, druhý nejhorší chromozom bude druhý a pořadí nejlepšího chromozomu bude N.

Následně se pak chromozomům přiřadí nová hodnota fitness, která slouží pouze pro účely aktuální selekce. V závisloti na tom, zda se používá lineární nebo nelineární pořadí použije příslušný vzorec pro výpočet nové hodnoty fitness. Nakonec se jednotlivé dílky na ruletovém kole rozdělí podle nově přiřazených hodnot fitness.

Jako příklad můžeme použít ruletovou selekci založenou na linármín pořadí. Pro lepší porovnání obou ruletových operátorů použijeme stejné chromozomy jako v tabulce 2.1.

Mějme tedy tabulku chromozomů nyní již však seřazenou podle hodnot fitness. Následně přikročíme k přiřazení nových hodnot fitness založených na lineárním pořadí. K výpočtu nové hodnoty slouží vzorec 2.2

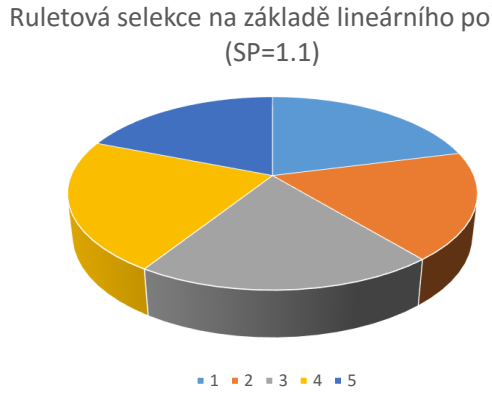
$$NFitness(Rank) = 2 - SP + 2 \times (SP - 1) \times \frac{Rank - 1}{Nchro - 1} \tag{2.2}$$

Kde hodnota *NFitness* značí nově přiřazenou hodnotu fitness, *SP* značí selektivní tlak, který může nabývat rozsahu 1.0–2.0, *Rank* značí aktuální umístění chromozomu a *Nchro* označuje celkový počet chromozomů v populaci.

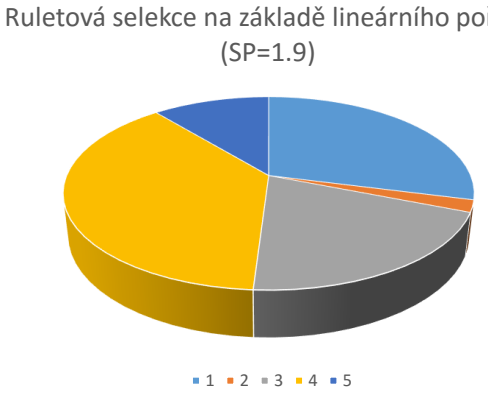
Všechny hodnoty pak můžeme vidět v nové tab. 2.2 Pro lepší porozumění selekčnímu tlaku byly zvoleny dvě hodnoty, kdy nízký selekční tlak dává každému chromozomu téměř stejnou šanci a vysoký selekční tlak naopak prioritizuje chromozomy s vyšší hodnotou fitness [3]. Grafickou reprezetaci, ve které je vidět rozdíl rozložení jednotlivých dílů při použití různých selekčních tlaků, lze vidět na Obr. 2.2 a Obr. 2.3.

Tab. 2.2: Ruletová selekce na základě hodnocení

Chromozom	Fitness	Rank	NFitness(SP=1.1)	Pi	NFitness(SP=1.9)	Pi
1	305	4	1,05	0,21	1,45	0,29
2	49	1	0,9	0,18	0,1	0,02
3	175	3	1	0,2	1	0,2
4	631	5	1,1	0,22	1,9	0,38
5	148	2	0,95	0,19	0,55	0,11



Obr. 2.2: Ruletová selekce na základě pořadí s rozdílným selekčním tlakem



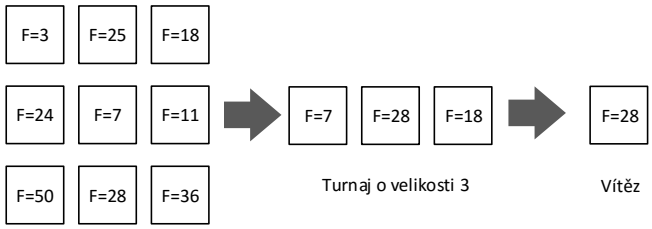
Obr. 2.3: Ruletová selekce na základě pořadí s rozdílným selkečním tlakem

Turnajová selekce

Turnajová selekce, jak již z jejího názvu vypovídá vybírá chromozomu na základě turnajů. Ná základě velikosti turnaje (nejmenší turnaj má velikost 1, největší pak odpovídá počtu chromozomů v celé populaci) se z populace vybere náhodně tolik chromozomů, aby se naplnil turnaj. Na základě své fitness funkce pak mezi sebou jednotlivé chromozomy soutěží, vyhrává ten s nejvyšší hodnotou, který je následně vybrán. Selekční tlak v této metodě je ovliněn velikostí turnaje, při velikosti 1 je selkeční tlak nejnižší, jelikož se v podstatě jedná o náhodný výběr, nejvyššího selekčního tlaku se dosahuje v případě, že je velikost turnaje odpovídá velikosti populace, pak vítězí pouze ten nejsilnější chromozom[8]. Princip turnajové selekce je zobrazen na Obr. 2.4.

2.1.2 Křížení

Za dobu, co jsou vyvíjeny genetické algoritmy vzniklo opravdu spoustu různých metod pro křížení chromozomu. Ne každou metodu však použít na každý chromozom.



Obr. 2.4: Turnajová selekce

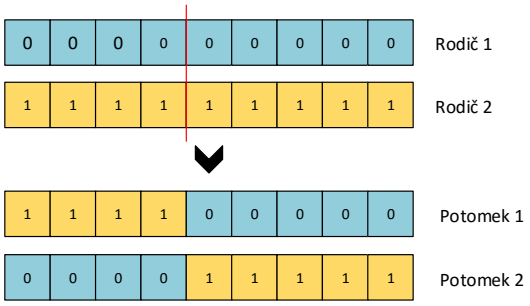
V případě binárních chromozomů existuje několik základních typů křížení.

Jsou to takzvaně n-bodové křížení. Jejich vstupem jsou dva rodiče a výstupem dva noví potomci. Princip jejich funkčnosti spočívá v tom, že u obou rodičů jsou na náhodných místech geny rozděleny a jejich kombinací vznikají noví potomci. Počet těchto bodů pak označuje typ křížení.

Existují tři základní typy n-bodového křížení a to

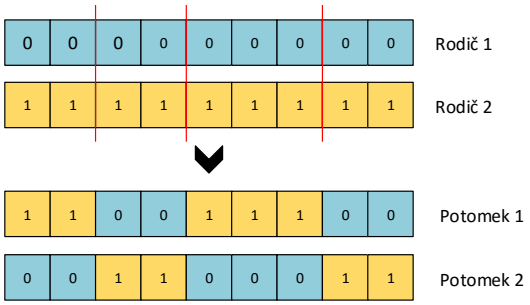
- jednobodvé křížení,
- vícebodové křížení,
- uniformní křížení.

Jak již název napovídá, tak u jednobodového křížení je vybrán pouze jeden bod a potomci jsou vzájemně nakombinováni ze vzniklých půlek chromozomů rodičů, tak jak můžeme vidět na Obr. 2.5

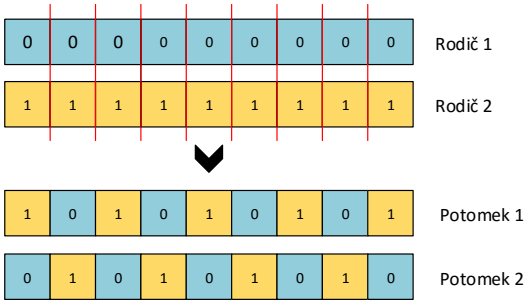


Obr. 2.5: Jednobodové křížení

Druhým typem je takzvané vícebodové křížení, do kterého lze zařadit i uniformní křížení. U vícebodového typu křížení je vygenerováno několik bodů rozdělení a výslední potomci jsou opět složeni z těchto celků. Extrémní variantou vícebodového křížení je křížení uniformní, kde počet dělících bodů odpovídá velikosti chromozomu a chromozom je tak vlastně rozdělen na jednotlivé geny. Potomci pak vznikají z lichých genů jednoho rodiče a sudých genů druhého rodiče [9]. Obě tyto metody lze vidět na obr. 2.6 respektive na Obr. 2.7.



Obr. 2.6: Vícebodové křížení



Obr. 2.7: Uniformní křížení

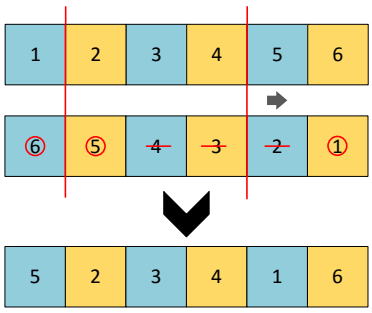
Speciální metody křížení

Podstatou n-bodových typů křížení je to, že jen kopírují svůj obsah do potomků, ale již se nestarají co zkopírují. Tento přístup nám vyhovuje do doby, než potřebujeme zajistit, aby se v chromozomu nevyskytovaly stejné hodnoty. Použití těchto metod lze tedy vyloučit, jelikož nelze zaručit, že každý potomek bude mít unikátní geny.

Byly proto vyvinuty speciální metody křížení, které během vytváření potomka kontrolují, jaké genyy již byly zkopírovány a nebo byly navrhnuty tak, že k možnosti zkopírování stejného genu dojít ani nemůže. Vstupem pro tyto metody jsou opět dva rodiče, vetšinou však generují pouze jednoho potomka. Existuje několik možných metod křížení a jejich různých variací, zde však budou popsány následující

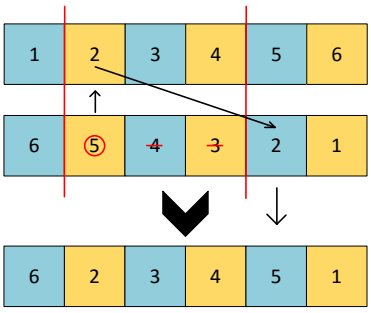
- Order One Crossover,
- PMX Crossover,
- Edge Recombination Crossover.

Prvním a nejjednodušším typem je Order One Crossover. Ve své podstatě je tato metoda podobná metodám n-bodového křížení. První krokem je rozdělení prvního rodiče. Dvěma náhodně vygenerovanými body se v prvním rodiči označí část, která je následně zkopírována do rodiče. Následně se začnou vyčítat geny z druhého rodiče, od bodu, který udává konec rozdělení a jsou ukládány do potomka. Geny, které již byly zkopírovány do potomka se přeskakují[12]. Grafickou reprezentaci této metody křížení lze vidět na Obr. 2.8.



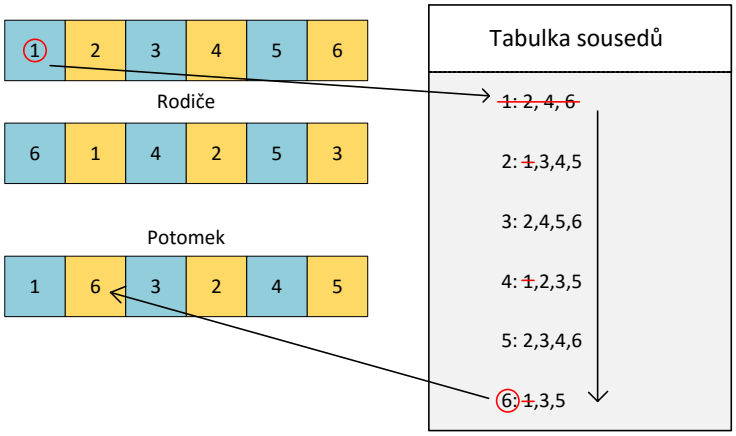
Obr. 2.8: Order One Crossover

Druhou metodou je PMX Crossover [14]. Prvním krokem je zde, stejně jako u Order One Crossover ohraničení části u prvního rodiče, která je rovnou zkopírována do potomka, od tohoto bodu se však již tyto metody rozcházejí. U PMX se následně ve stejné části druhého rodiče hledají geny, které nebyly zkopírovány do potomka. Pokud je takový gen nalezen, je zjištěna hodnota genu na stejné pozici u prvního rodiče a následně se hledá pozice této hodnoty v chromozomu druhého rodiče. Pokud tato pozice leží uvnitř vybrané části, tak se celý proces opakuje dokud není nalezena pozice mimo vybranou část. Na tuto pozici je následovně uložena hodnota, která byla vybrána na začátku (gen, který nebyl zkopírován do potomka z druhého rodiče). Po projití celé vybrané části se jednoduše zkopírují všechny geny z druhého rodiče, které chromozom dosud neobsahuje [11]. Podrobnější vysvětlení lze získat z Obr. 2.9.



Obr. 2.9: PMX Crossover

Poslední metodou je Edge Recombination Crossover. Je to metoda, která kombinuje již vzniklé cesty rodičů do nově vzniklého potomka. Tato metoda vytváří tabulky sousedních genů pro jednotlivé geny. Následně zkopíruje první gen z prvního rodiče do potomka a vyškrtne jeho tabulku sousedů. Následně je odstraněn ze všech ostatních tabulek sousedů, které ho obsahují tento gen. Následně je do potomka zkopírován gen, jehož tabulka sousedů je nejmenší. Tento proces se opakuje tak dlouho, dokud není naplněn potomek. Grafické znázornění jedné iterace této metody lze vidět na Obr. 2.10.

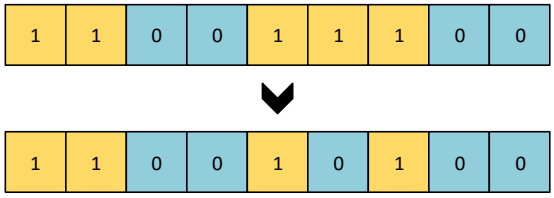


Obr. 2.10: Edge Recombination Crossover

2.1.3 Mutace

Stejně jako kížení, tak i mutace obsahuje několik základních metod a několik speciálních, které se používají ve vybraných případech. Klasické metody mutace většinou mění hodnotu vybraného genu v chromozomu.

Do této kategorie patří například inverzní mutace binárního chromozomu. Náhodný gen takového chromozomu invertuje svoji hodnotu, tedy buď se změní z 0 na 1 nebo obráceně [6]. Tuto mutaci lze vidět na Obr. 2.11.



Obr. 2.11: Mutace binárních hodnot

Druhou metodou z této kategorie je mutace reálného chromozomu (chromozom tvořený z reálných čísel). K hodnotě náhodně vybraného genu se buď přičte nebo odečte malá hodnota a tím dojde k mutaci chromozomu [8]. Mutace je graficky znázorněna na Obr. 2.12.

Obr. 2.12: Mutace reálných hodnot

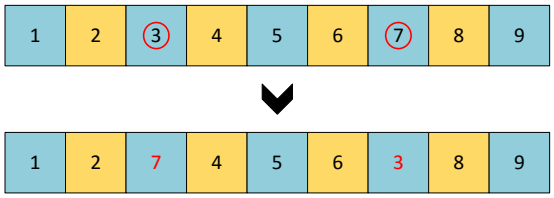
Speciální metody mutace

Běžné metody mutace mění většinou hodnoty genů. Když ale není změna hodnoty genu možností, je nutné použít speciální metody mutací. Takové metody nemění hodnoty genu, ale mění jejich pozice.

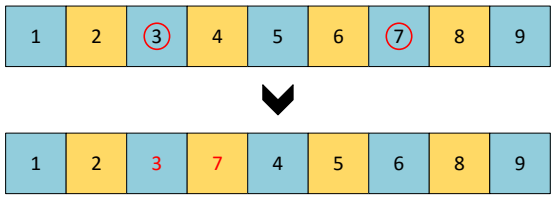
Základními dvěma typy těchto mutací jsou

- Swap Mutation,
- Insert Mutation.

Tyto mutace náhodně vybírají dva geny a následně pracují s jejich polohami [10]. Metoda Swap Mutation pozice těchto genů prohazuje, jak lze vidět na Obr. 2.13, kdežto mutace typu Insert Mutation tvoří z těchto genů sousedy, což je graficky znázorněno na Obr. 2.14.



Obr. 2.13: Swap Mutation

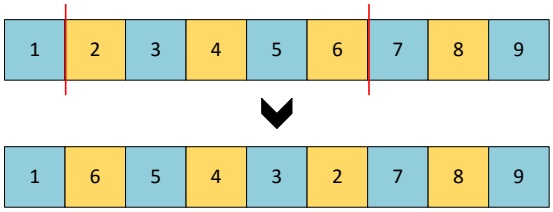


Obr. 2.14: Insert Mutation

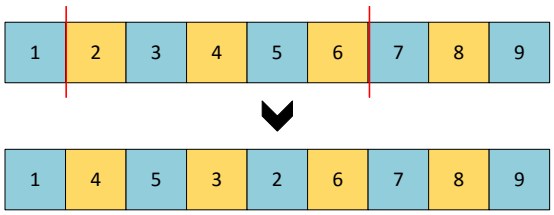
Existují však i typy mutací, které nepracují jen se dvěma geny, ale s celou skupinou genů [10]. Do této skupiny patří

- Inverse Mutation,
- Scramble Mutation.

Tyto typy mutací náhodně zvolí část chromozomu a pak s ní dále pracují. V případě Inverse Mutation jsou geny v této části zrcadlově obráceny, což lze vyvodit z Obr. 2.15 Naopak metoda Scramble Mutation geny v této části náhodně promíchá, to je znázorněno na Obr. 2.16.



Obr. 2.15: Inverse Mutation



Obr. 2.16: Scramble Mutation

3 PROBLÉM OBCHODNÍHO CESTUJÍCÍHO

Problém obchodního cestujícího (TSP z anglického Traveling Salesman Problem) je klasickým problémem optimalizace v oblasti výpočetní techniky.

Pokud je zadáno několik měst a známe vzdálenosti mezi nimi, pak TSP řeší problém, jak navštívit všechna města v pokud co nejkratší trase a vrátit se zpátky do výchozího města. Jde jednoduše o pořadí, v jakém budou města navštívena, můžeme to nazývat například okruh.

Tento jednoduše znějící problém je ve skutečnosti jedním z nejvíce řešených problémů v matematice, ale jeho teoretické uplatnění je v několika dalších oborech, jako je například genetika, výroba, telekomunikace a další.

Tento problém byl poprvé popsán někdy kolem roku 1800 dvěma matematiky, Williamem Hammiltonem a Thomasem Kirkmanem. Obecně byl problém popsán někdy kolem roku 1930 Karlem Mengerem.

V praxi se tedy často volí řešení problému pomocí genetických algoritmů, které sice ze své povahy nemusí vždy najít optimální řešení, ale jsou schopny v relativně rozumném čase najít výsledky, které se blíží optimálnímu řešení [5].

Existují dva typy TSP, symetrický a asymetrický. Symetrický TSP znamená, že mezi dvěmi městy leží cesta o stejné vzdálenosti. Naproti tomu u asymetrického TSP nemusí být města propojena v obou směrech, to znamená, že se dá dostat pouze z jednoho města do druhého a už ne nazpátek a pokud jsou města propojeny v obou směrech, tak nemusí být tyto cesty stejně dlouhé. Asymetrické TSP lze použít pro ucpané cesty, jednosměrky ve městech, atd, tato práce se však zabývá pouze symetrickou verzí TSP [5].

Ačkoliv je TSP ve svém konceptu jednoduchý, je těžké získat přesný výsledek hlavně kvůli tomu, že se vzrůstajícím počtem měst vzrůstá i počet jednotlivých možností podle následujícího vzorce. Jedná se o tzv. NP-těžký problém, není tedy známo, jak pro každý vstup najít řešení v rozumném čase a zda vůbec existuje algorimus, který dokáže získat optimální výsledek taktéž v rozumném čase.

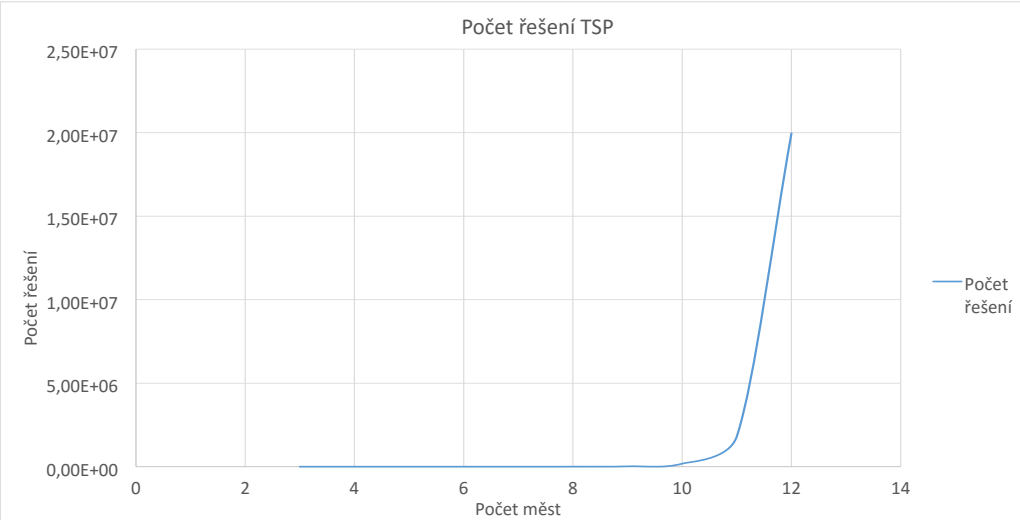
Počet jednotlivých řešení TSP roste s počtem měst, přesněji s jeho faktoriálem. Pokud máme n měst, tak v počátečním městě máme $n-1$ možností jak navštívit druhé město, $n-2$ možností pro třetí město atd. Násobením těchto hodnot dostaneme, že počet řešení se rovná $(n-1)!$ [5]. Jelikož však řešíme symetrický TSP, je jedno v jakém směru trasu projedeme a proto můžeme počet celkových řešení vydělit dvěmi, finální vzorec pro výpočet počtů řešení P pro n měst je tedy

$$P = \frac{(n-1)!}{2} \tag{3.1}$$

V následující tabulce 3.1 a grafu 3.1 lze vidět, jak rapidně počet všech možných kombinací narůstá.

Tab. 3.1: Počet řešení TSP pro různý počet měst

Počet měst	Počet řešení
3	1
4	3
5	12
6	60
7	360
8	2520
9	20160
10	181440
11	1814400
12	19958400



Obr. 3.1: Řešení ideální cesty mezi všemi městy

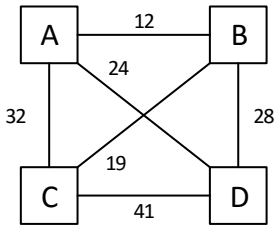
3.1 Praktický příklad

Prakticky se dá vysvětli TSP na následujícím příkladu. Mějme pět měst A,B,C,D mezi nimiž leží cesty o předem známé vzdálenosti, které lze najít v tab. 3.2.

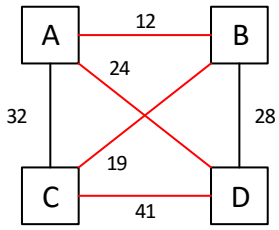
Pro lepší představu jsou tato města a cesty mezi nimi zobrazeny na Obr. 3.2. Pak pokud obchodník začíná ve městě A, musí projít všechna města a vrátit se zpět do města A a projdeme všechny možné kombinace, tak zjistíme, že trasa s nejnižší vzdáleností je A–B–C–D–A (nebo v opačném směru A–D–C–B–A) tak, jak je znázorněno na Obr. 3.3.

Tab. 3.2: Vzádelenost mezi městy

	A	B	C	D
A	0	12	32	24
B	12	0	19	28
C	32	19	0	41
D	24	28	41	0



Obr. 3.2: Jednoduché znázornění měst a cest mezi nimi



Obr. 3.3: Řešení ideální cesty mezi všemi městy

4 IMPLEMENTACE ALGORITMU

Genetický algoritmus byl implementován v jazyce *JAVA* s použitím vývojového prostředí *Eclipse*. Implementaci bylo nejprve nutné rozdělit do několika tříd, každá pak obsahuje dílčí části algoritmu, které dohromady dávají celý funkční algoritmus.

Program umí řešit dva typy problémů, prvním z nich je *Hello World!*, tedy úloha, kdy z počátečního náhodně vygenerovaného řetězce slov genetický algoritmus sestaví požadovaný výsledek. Druhou úlohou je již zmíněný TSP.

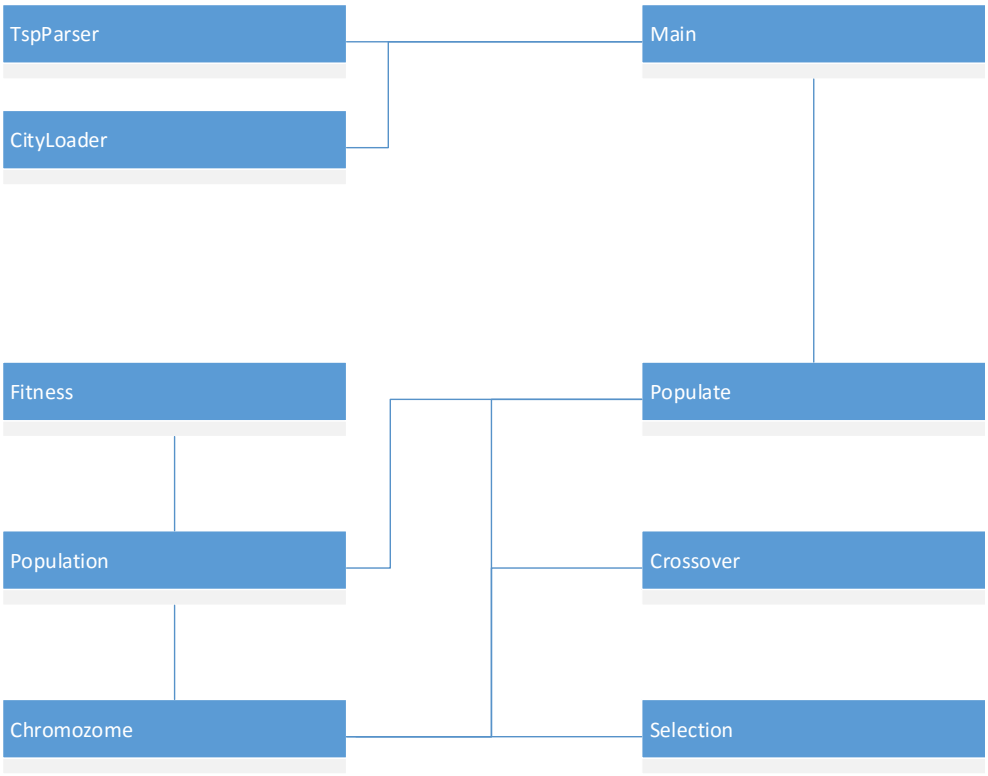
Třída *Main* slouží ke spouštění celého programu a nastavují se v ní jednotlivé parametry pro genetický algoritmus, k načtení vstupních dat se používají třídy *TspParser* pro načtení dat z databáze *TSPLib* nebo *CityLoader* pro načtení uměle vytvořených dat.

Třída *Populate* slouží k chodu genetického algoritmu, v ní se zajišťuje chod genetického algoritmu, od vytvoření počáteční populace, ohodnocení jedinců a následného vytváření dalších populací. V této třídě taky dochází k exportování naměřených dat do výstupního souboru.

Třídy *Chromozome* a *ChromozomeHW* slouží k vytváření chromozomů, ze kterých se následně pomocí tříd *Population* a *PopulationHW* vytvoří jednotlivé populace.

Třída *Fitness* obsahuje metody k ohodnocení jednotlivých chromozomů v populaci, tyto hodnoty následně využívá třída **Selection** a *SelectionHW* k selekci vhodných chromozomů pro vytvoření rodičů. Pomocí metod obsažených ve třídách *Crossover* a *CrossoverHW* se následně tyto chromozomy kříží a vznikají tak noví potomci. Tyto třídy taktéž obsahují metody pro mutaci nově vzniklých potomků.

Na Obr. 4.1 lze vidět zjednodušený diagram, který reprezentuje vztah jednotlivých tříd pro řešení TSP.



Obr. 4.1: Vztahy mezi třídami pro řešení TSP

4.1 Vstupní data

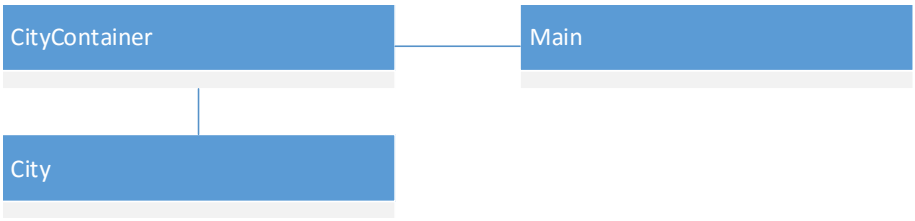
Aby bylo možné použít genetický algoritmus, je nejprve nutno do něj vložit nějaká vstupní data. V případě tohoto algoritmu se jedná o jakési plány, které reprezentují určitý počet měst, jejich souřadnice a vzdálenosti mezi nimi.

Byly použity dva zdroje pro vstupní data. Prvním zdrojem je program *CityGenerator*, který slouží k umělému a náhodnému vytvoření vstupních dat. Druhým zdrojem je pak knihovna *TSPLib*, která obsahuje velké množství dat, které jsou založeny na reálném geografickém podkladu, většinou se jedná o sadu existujících měst v regionech či celých státech.

4.1.1 CityGenerator

CityGenerator je, jak již bylo dříve zmíněno program, který slouží k umělému vytváření TSP dat pro řešení. Tento program se skládá celkem ze tří tříd, jejichž vztah lze vidět na Obr. 4.2.

První část, třída *City* slouží k reprezentaci města, a obsahuje informace o jeho poloze, konrétně se jedná o souřadnice X a Y ve dvourozměrném prostoru a jeho název.

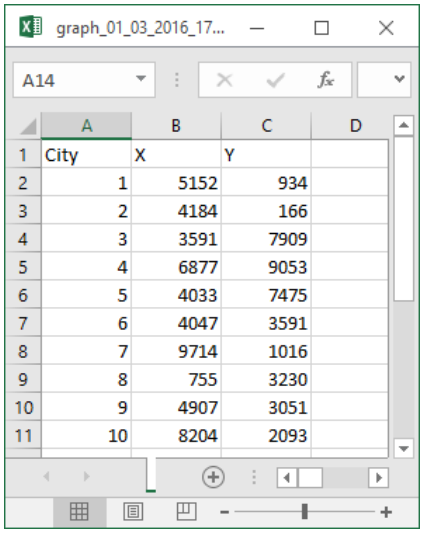


Obr. 4.2: Vztahy mezi třídami pro CityGenerator

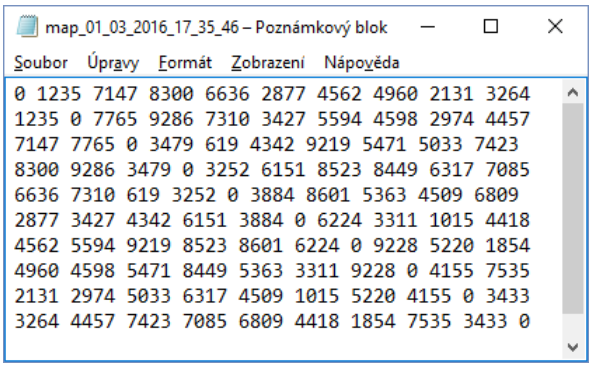
Druhou částí je třída *CityContainer*, která slouží ke generování měst a následně i k vypočítání vzdáleností mezi jednotlivými městy. Generování probíhá tak, že je nejprve určen rozměr plochy, na které se města budou vytvářet a následně jsou náhodně generovány souřadnice X a Y pro jednotlivá města. Po vygenerování zvoleného počtu měst dojde k výpočtu vzdáleností mezi městy. Vzdálenosti se počítají podle vzorce 4.1 pro určení vzdálenosti mezi dvěma body v dvourozměrném prostoru, kde x_1 a y_1 jsou souřadnice prvního města a x_2 a y_2 jsou souřadnice druhého města.

$$|XY| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{4.1}$$

Poslední, třetí části je třída *Main*, ve které probíhá za pomocí třídy *CityContainer* k vygenerování měst a následnému vytvoření výstupních souborů. Celkem se jedná o dva výstupní soubory, první souboru obsahuje souřadnice jednotlivých měst, slouží k následnému grafickému znázornění poloh měst. Ve druhém výstupním souboru je uložena tabulka vzdáleností mezi jednotlivými městy, příklady výstupních souborů lze vidět na Obr. 4.3 respektive na Obr. 4.4.



Obr. 4.3: Výstupní soubor obsahující souřadnice měst



Obr. 4.4: Výstupní soubor obsahující tabulku vzdáleností

4.1.2 TSPLib

TSPLib je projekt univerzity v Heidelbergu [4]. Jedná se o databázi vstupních dat pro TSP, která obsahuje stovky záznamů. Obsahuje materiály pro několik typů TSP, ale jelikož se tato diplomová práce zabývá symetrickým TSP, bude v následujícím textu popsány právě tyto vstupní data.

Data lze stáhnout přímo ze stránek univerzity, jedná se soubory s příponou *.tsp*. Jedná se pouze o vstupní data, jejich použití pak záleží pouze na uživateli. Formát souboru se dělí na dvě části, hlavičku a samotná data.

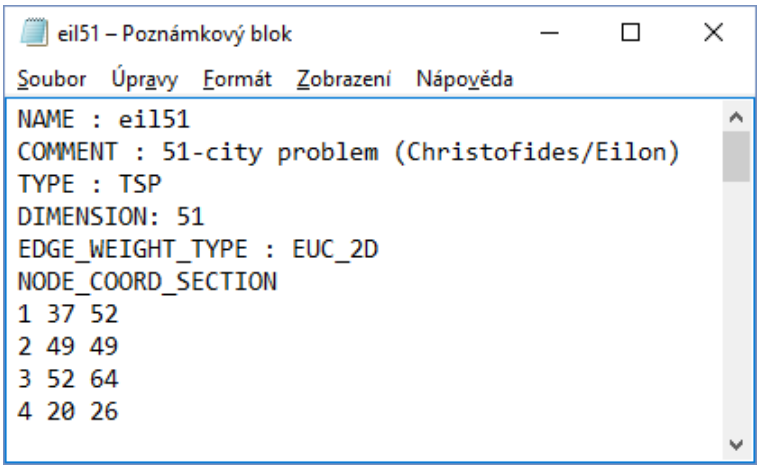
V hlavičce souboru je několik dat upřesňující jejich typ. Tyto data se vyskytují v páru (*Klíčové slovo*):(*Hodnota*). V tabulce 4.1 lze vidět popis důležitých klíčových slov.

Tab. 4.1: Klíčová slova hlavičky souboru z databáze TSPLib

Klíčové slovo	Popis
Name	Návev řešeného problému
Type	Typ řešeného problému
Comment	Komentář k řešenému problému
Dimension	Počet měst
Edge_Weight_Type	Formát dat

Pro lepší pochopení je na Obr. 4.5 zobrazen reálný soubor s označením *eil51.tsp*. U klíčového slova *Name* můžeme vidět hodnotu *eil51*, označující název souboru, u klíčového slova *Comment* pak komentář k danému souboru, většinou jsou v něm uvedeni tvůrci podkladu, u klíčového slova *Type* je hodnota TSP označující, že se jedná o symetrický TSP. U klíčového slova *Dimension* je vidět počet měst, které soubor obsahuje, hodnota *EUC2D* u klíčového slova *EdgeWeightType* nám říká, že jsou

data ve formátu dvourozměrného Eukleidovského prostoru a klíčové slovo *NodeCoordSection* nám označuje, že za nim se již vyskytují jednotlivé souřadnice. Jak lze vidět na obrázku, tak číslo 1 označuje pořadí města, a dvě následující čísla 37 a 52 označují souřadnice X a Y.



Obr. 4.5: Struktura souboru z databáze TSPLib

4.2 Načítání vstupních dat

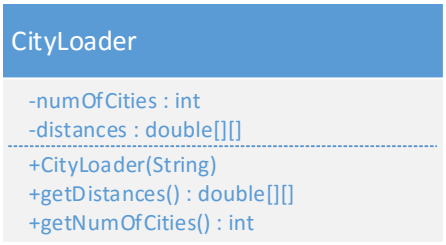
První věcí, kterou je potřeba zvládnout, při řešení TSP, je načíst vstupní data. Nabízí se možnost data tvrdě zakódovat přímo do programu, čímž by odpadla nutnost tvořit speciální třídy pro dynamické načítání vstupních hodnot, ale tato metoda je poněkud těžkopádná, jelikož pak program postrádá potřebnou rozšiřitelnost z důvodu, že je nová data do programu potřeba dopisovat ručně, což při větším počtu měst může zabrat daleko více času, než vytvoření třídy pro dynamické načítání dat ze souboru nezávisle na počtu měst.

Jelikož program používá dva typy vstupních dat, prvním typem jsou uměle vytvořená data pomocí *CityGeneratoru* a druhým typem jsou data z databáze *TSPLib*, bylo potřeba navrhnout i dvě metody pro načítání těchto dat, jelikož oba typy se liší svým zápisem. Proto byly vytvořeny dvě následující třídy, *CityLoader*, která slouží k načítání dat z programu *CityGenerator* a *TspParser*, která, jak už název napovídá, slouží k načítání dat z *TSPLib*. V následujících dvou kapitolách bude popsán obecný chod těchto dvou tříd.

4.2.1 CityLoader

Strukturu třídy *CityLoader* lze vidět na Obr. 4.6. Hlavní částí třídy je konstruktor *CityLoader*, ve kterém se odehrává vše důležité. Vstupním argument je proměnná

typu *String*, ve které je uložena cesta k souboru ze *CityGeneratoru*, který obsahuje tabulku vzdáleností mezi jednotlivými městy.

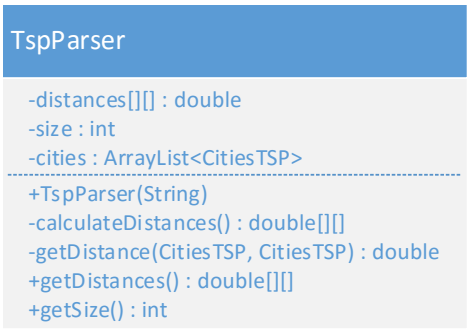


Obr. 4.6: Struktura třídy CityLoader

V konstruktoru třídy nejprve dojde k inicializaci proměnných a následně se začnou načítat data do tabulky ze vstupního souboru. Pomocí dvou funkcí pak lze ze třídy získat jak samotnou tabulku vzdáleností, tak i informaci o celkovém počtu měst.

4.2.2 TspParser

Třída *TspParser*, jejíž strukturu lze vidět na Obr. 4.7 je o trochu složitější, než třída *CityLoader*, jelikož třída podporuje načítání dat pouze z určitého typu souboru, který obsahuje souřadnice jednotlivých měst, ze kterých až následovně počítá jejich vzájemné vzdálenosti.



Obr. 4.7: Struktura třídy TspParser

Načítání dat ze vstupního souboru opět probíhá v konstruktoru třídy, který má vstupní argument typu *String*, který je cestou k samotnému souboru. Načítání začíná opět inicializací proměnných, následně je nutno zjistit, kolik měst vstupní soubor obsahuje. K tomu slouží jeden z parametrů hlavičky souboru, přesněji položka *Dimension*, jejíž hodnota se rovná počtu měst. Všechny ostatní položky v hlavičce jsou přeskočeny a následuje vyčítání souřadnic. Pro každé město se načtou jeho

souřadnice X a Y, a následně je vytvořen objekt ze třídy *TSPCities*, který je obdobou třídy City. Tyto objekty jsou následně uloženy do proměnné typu *ArrayList*. Po načtení všech měst jsou spočítány jednotlivé vzdálenosti mezi nimi a uloženy do tabulky.

4.3 Implementace chromozomu a populace

Chromozom je základní stavební jednotkou genetického algoritmu. Jeho návrh je tedy nedílnou součástí správného návrhu, aby bylo možné získat kvalitní výsledky. V případě TSP se používá chromozom s permutačním kódováním, to znamená, že každá hodnota v chromozomu se vyskytuje pouze jednou. Velikost chromozomu, tedy počet jeho genů se rovná počtu měst, který obchodní musí navštívit. Jelikož cestující začíná a končí ve stejném městě, není nutné toto město do chromozomu kódovat. Samotný chromozom pak reprezentuje cestu, pořadí, ve kterém cestující jednotlivá města navštíví. Je nutné dodržet podmínku, aby se za žádnou cenu nestalo, že by se v chromozomu jedno město objevilo více než jednou, jelikož by pak došlo k znehodnocení výsledků.

K lepšímu pochopení je chromozom pro TSP popsán na následujícím příkladu. Mějme 5 měst, označené A,B,C,D,E. Obchodní cestující začíná ve městě A a musí nalézt trasu tak, aby navštívil každé město pouze jednou. Jelikož jeho trasa začíná ve městě A, není nutné toto město již do chromozomu kódovat a v chromozomu se tedy budou vyskytovat pouze města B,C,D,E.

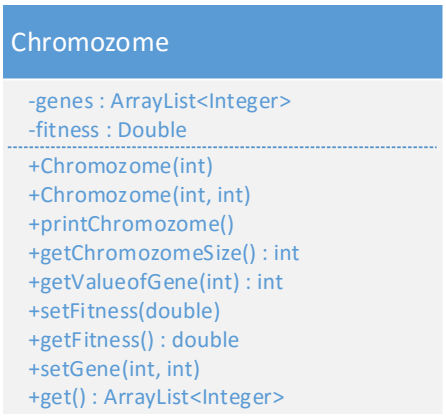
Implementace populace oproti chromozomu již tak složitá není, jelikož jde pouze o to, vygenerovat potřebný počet chromozomů, které bude populace obsahovat.

4.3.1 Chromozom

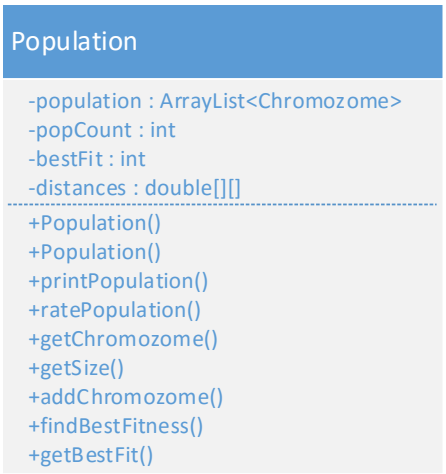
Strukturu třídy, která reprezentuje chromozom lze vidět na Obr. 4.8 Generování chromozomu opět probíhá v konstruktoru třídy, kde dojde k vygenerování proměnných typu *Integer* v rozsahu $\langle 2; n\rangle$ kde n značí počet měst. Aby nebyl každý vygenerovaný chromozom stejný, dojde pak ještě k náhodnému promíchání hodnot.

4.3.2 Populace

Po vytvoření chromozomu je nutné je uspořádat do jedné skupiny, aby s nimi bylo možné lépe manipulovat. K tomu slouží třída Population, jejíž strukturu lze vidět na Obr. 4.9 Tato třída není nějak sloužitá, při vytváření populace v konstruktoru dojde pouze k naplnění *ArrayListu*, do kterého se ukládají chromozomy vytvořené voláním konstruktoru třídy reprezentující chromozom.



Obr. 4.8: Struktura třídy Chromozome



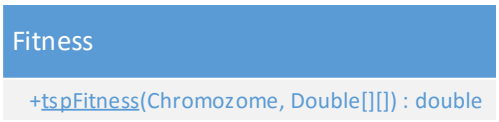
Obr. 4.9: Struktura třídy Population

4.4 Implementace fitness

Pokud postupujeme podle základního schematu evolučních algortimů, pak po vytvoření chromozomů a následně populace je nutné populaci ohodnotit. K tomu slouží fitness funkce, která je pokaždé jiná v závislosti na řešeném problému.

Při řešení TSP se jako vhodné ohodnocení chromozomu nabízí součet vzdálenosti trasy, kterou daný chromozom reprezentuje. Z toho tedy vyplývá, že čím menší hodnota fitness chromozomu je, tím lepší poskytuje řešení.

Při implementování fitness funkce, byla vytvořena třída *Fitness*, která obsahuje statickou funkci *tspFitness*, která slouží právě k ohodnocení jednotlivých chromozomů, jak můžeme vidět na struktuře třídy na Obr. 4.10.



Obr. 4.10: Struktura třídy Fitness

Samotná fitness funkce vypadá následovně[1].

Vstup: Chromzom, Tabulka vzdáleností
Výstup: Hodnota fitness
1 inicializace proměnné fitness;
2 fitness = vzdálenost mezi prvním městem a prvním městem v chromozomu;
3 **for** *i=0, j=1; j< Velikost chromozomu; i++, j++* **do**
4 | fitness += vzdálenosti mezi jednotlivými městy v chromozomu
5 **end**
6 fitness = vzdálenost mezi prvním městem a posledním městem v chromozomu;
7 vrať fitness;

Algoritmus 1: Kód funkce tspFitness

Nejprve je inicializována proměnná fitness, do které se hodnota ukládá. Poté dojde k přičtení vzdálenosti mezi počátečním městem a prvním městem v chromozomu. Následně se do proměnné přičítají ostatní vzdálenosti mezi městy, tak jak chromozom reprezentuje cestu. Vzdálenosti jsou vyčítány z tabulky vzdáleností. Nakonec je přičtena vzdálenost mezi posledním městem v chromozomu a počátečním městem cesty, aby byla zkompletována celá cesta a funkce vrátí konečnou hodnotu fitness.

4.5 Implementace selekce

Po ohodnocení jednotlivých chromozomů již můžeme přistoupit k selekci, tedy k vybrání vhodných chromozomů, kteří se stanou rodiči a vytvoří potomky, kterými bude naplněna další generace.

Proto byla vytvořena třída *Selection* 4.11, která obsahuje statickou funkci *tournamentSelection* [2], která reprezentuje turnajovou selekci.

Obr. 4.11: Struktura třídy Selection

Vstup: Velikost turnaje, Populace
Výstup: Vybraný chromozom
1 inicializace proměnné tournament;
2 inicializace generátoru náhodných hodnot;
3 **for** *i=0; i<velikost turnaje; i++* **do**
4 | naplnenění turnaje náhodně vybranými chromozomy z populace;
5 **end**
6 **for** *j=0; j<velikost turnaje; j++* **do**
7 | najití nejlepšího chromozomu v turnaji
8 **end**
9 vrať nejlepší chromozom;

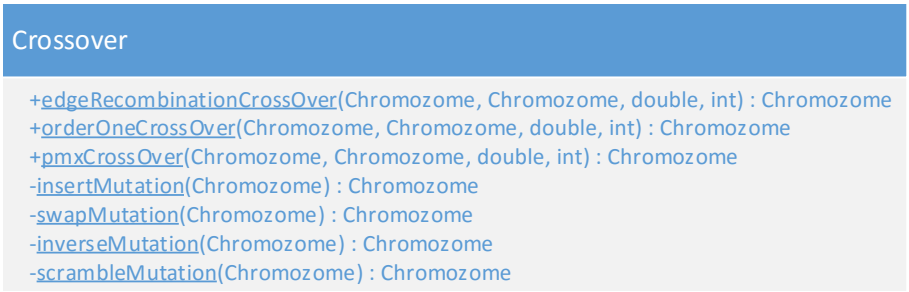
Algoritmus 2: Kód funkce tournamentSelection

Nejprve jsou inicializovány potřebné proměnné, generátor náhodných čísel slouží k náhodnému výběru chromozomů z populace. Vybírání chromozomů probíhá tak dlouho, dokud není naplněn celý turnaj. Po naplnění turnaje proběhne vyhledání chromozomu s nejlepší fitness, který je následně vybrán jako návratová hodnota funkce.

4.6 Implementace křížení a mutace

Při implementaci křížení a mutace je nutno dodržet podmínku, aby se u nově vzniklých potomků nevyskytovaly stejné geny. Proto nelze použít standardní techniky křížení, ale je nutné použít specializované techniky pro křížení chromozomů s permutačním kódováním. To stejné platí i pro mutaci.

Proto byla vytvořena třída *Crossover* 4.12, která v sobě tyto techniky implemen-



Obr. 4.12: Struktura třídy Crossover

tuje a zároveň implementuje i techniky pro následnou mutaci vzniklých potomků.

4.6.1 Křížení

Pro křížení byly vybrány celkově 3 techniky křížení a to:

- Order One Crossover,
- PMX Crossover,
- Edge Recombination Crossover,

které byly teoreticky popsány v kapitole 2.1.2 a v této kapitole bude popsána jejich praktická implementace.

Jako první byla implementována funkce *orderOneCrossOver* [3], jejíž kód je pro implementaci poměrné jednoduchý, jelikož zde nedochází k žádným složitým funkcím.

```
   Vstup: Rodič1, Rodič2, Šance na mutaci, Typ mutace
   Výstup: Potomek
 1 inicializace proměnných;
 2 inicializace generátorů náhodných hodnot;
 3 generování hodnot, které určí vybranou část chromozomu Rodiče1;
 4 for bod1; bod1<bod2+1; bod1++ do
 5 |   kopírování genů z vybrané části Rodič1 do potomka;
 6 end
 7 while Dokud není naplněn potomek do
 8 |   kopíruj geny z Rodič2, které potomek dosud neobsahuje;
 9 end
10 vygeneruj náhodné číslo;
11 if náhodné číslo<pravděpodobnost mutace then
12 |   vyber zvolenou mutaci a zmutuj potomka;
13 end
14 vrať potomka;
```

Algoritmus 3: Kód funkce orderOneCrossOver

Po inicializaci proměnných a generátorů náhodných čísel, dochází k vygenerování dvou náhodných čísel, *bod1* a *bod2* za podmínky, že $bod1 < bod2$, které ohraňičují část chromozomu z *Rodič1*, která je následně celá zkopírována do potomka. Poté dochází ke kopírování genů z *Rodič2*, při kopírování se musí dodržet podmínka, že se nesmí zkopírovat gen, který již potomek obsahuje. Po naplnění potomka geny dochází k vygenerování hodnoty, která když splňuje podmínku, že je menší než vstupní hodnota mutace, tak dojde k následnému zmutování potomka dle vybraného typu mutace. Nakonec je potomek vrácen jako návratová hodnota.

Druhou metodou křížení je funkce *pmxCrossover* [4].

```
   Vstup: Rodič1, Rodič2, Šance na mutaci, Typ mutace
   Výstup: Potomek
 1 inicializace proměnných;
 2 inicializace generátorů náhodných hodnot;
 3 generování hodnot, které určí vybranou část chromozomu Rodiče1;
 4 for bod1; bod1<bod2+1; bod1++ do
 5 |   kopírování genů z vybrané části do potomka;
 6 end
 7 while Projdi všechny geny vybrané části do
 8 |   if Pokud hodnotu z vybrané části neobsahuje Rodič2 ve stejné části then
 9 |   |   ulož hodnotu na dané pozici v Rodič2;
10 |   |   ulož hodnotu hodnotu na dané pozici v Rodič1;
11 |   |   najdi index této hodnoty v Rodič2;
12 |   |   while Dokud není splněna podmínka do
13 |   |   |   if Pokud je index mimo vybranou oblast then
14 |   |   |   |   Vlož uloženou hodnotu z Rodič2 do potomka na index;
15 |   |   |   |   podmínka splněna;
16 |   |   |   |   else
17 |   |   |   |   |   ulož hodnotu z Rodič1 na daném indexu;
18 |   |   |   |   |   ulož index této hodnoty v Rodič2;
19 |   |   |   |   end
20 |   |   |   end
21 |   |   end
22 |   end
23 end
24 for i; i<velikost chromozomu do
25 |   Zkopíruj ostatní geny mimo vybranou oblast z Rodič2, které potomek
   |   ještě neobsahuje
26 end
27 vygeneruj náhodné číslo;
28 if náhodné číslo<pravděpodobnost mutace then
29 |   vyber zvolenou mutaci a zmutuj potomka;
30 end
31 vrať potomka;
```

Algoritmus 4: Kód funkce pmxCrossOver

Po inicializaci proměnných dochází stejně jako v *Order One* k vybrání části chromozomu z *Rodič1*, která je zkopírována do potomka. Zde však podobnost mezi

algoritmy končí. V případě *PMX* se prohledávají geny na stejných pozicích, jako je vybraná část, v *Rodič2*. Pokud hodnota genu v *Rodič2* byla zkopírována do potomka, přechází se další hodnotu. Pokud ne, uloží se hodnota genu na dané pozici v *Rodič2*, hodnota genu na dané pozici v *Rodič1* a následně se zjistí index této hodnoty v *Rodič2*. Jestliže je tento index mimo vybranou oblast, uloží se na dané místo vloží se získaná hodnota z *Rodič2* na uložený index. Pokud ne, tak algoritmus prohledává a porovnává hodnoty ve vybrané části, dokud nenajde index mimo vybranou oblast. Pro projití celé vybrané části, se jednoduše zkopírují geny z *Rodič2*, které ještě nejsou obsaženy v potomkovi. Po naplnění celého potomka novými geny se provede v případě splnění podmínky vybraný typ mutace a potomek je vrácen jako návratová hodnota.

Poslední metodou mutace je funkce *edgeRecombinationCrossOver* [5].

```
   Vstup: Rodič1, Rodič2, Šance na mutaci, Typ mutace
   Výstup: Potomek
 1 inicializace proměnných;
 2 for i=0;i<velikost chromozomu; i++ do
 3 |   Vygeneruj sousedy pro všechny geny obsažené v Rodič1 a Rodič2;
 4 end
 5 ulož první gen z Rodič1 do potomka;
 6 for j=0;j<velikost tabulky všech sousedů; j++ do
 7 |   vymaž hodnotu prvního genu ze všech tabulek sousedů, kteří ji obsahují;
 8 end
 9 for k=1;k<velikost chromozomu; k++ do
10 |   najdi nejmenší tabulku sousedů, která obsahovala přechozí uložený gen;
11 |   vlož do potomka na pozici k gen, ke kterému patřila nejmenší tabulka
   |   sousedů;
12 |   for l=0;l<velikost tabulky všech sousedů; l++ do
13 |   |   vymaž tabulku sousedů pro tento gen z tabulky všech sousedů;
14 |   end
15 end
16 vygeneruj náhodné číslo;
17 if náhodné číslo<pravděpodobnost mutace then
18 |   vyber zvolenou mutaci a zmutuj potomka;
19 end
20 vrať potomka;
```

Algoritmus 5: Kód funkce edgeRecombinationCrossOver

Po inicializaci proměnných se nejprve vytvoří tabulka, která obsahuje tabulku sousedů pro jednotlivé geny z *Rodič1* a *Rodič2*. Pokud má gen stejného jednoho, nebo oba sousedy v obou chromozomech, je každý duplicitní soused použit jenom jednou. Následně dojde ke zkopírování prvního genu z *Rodič1* do potomka a následně je tabulka sousedů tohoto genu vymazána z tabulky všech sousedů. Poté se opakuje cyklus, kde je vymazána hodnota genu, který byl uložen do potomka, z tabulky sousedů, které ho obsahují. Z těchto tabulek je vybrána ta nejmenší a gen, kterému tato tabulka je vložen na další pozici v potomkovi. Toto se opakuje tak dlouho, dokud není naplněn celý potomek.

4.6.2 Mutace

Pro mutaci byly vybrány celkově 4 metody mutací a to:

- Insert Mutation
- Swap Mutation
- Inverse Mutation
- Scramble mutation

které byly teoreticky popsány v kapitole 2.1.3 a v této kapitole bude popsána jejich praktická implementace.

Jako první byla naimplementována metoda *Insert Mutation* do funkce *instert-Mutation* [6].

```
   Vstup: Chromzom
   Výstup: Zmutovaný chromozom
1  inicializace proměnných;
2  inicializace generátorů náhodných čísel;
3  generování náhodných čísel;
4  for i=náhodné číslo2; i> náhodné číslo1; i– do
5  |  prohoď gen na pozici i s genem na pozici i − 1;
6  end
7  vrať zmutovaného potomka;
```

Algoritmus 6: Kód funkce insertMutation

Po inicializaci proměnných se vygenerují dvě náhodná čísla, kdy první z nich označuje pozici genu, ke kterému se přesune gen, který leží na pozici druhého vygenerovaného čísla.

Druhou metodou je funkce *swapMutation* [7], která funguje podobně, jako první metoda.

Vstup: Chromzom
Výstup: Zmutovaný chromozom
1 inicializace proměnných;
2 inicializace generátorů náhodných čísel;
3 generování náhodných čísel;
4 prohoď geny v chromozomu na pozicích náhodné číslo1 a náhodné číslo2;
5 vrať zmutovaný chromozom;

Algoritmus 7: Kód funkce swapMutation

Opět velice jednoduchá funkce, po inicializaci proměnných a generátorů náhodných čísel, jsou vygenerovány dvě náhodné hodnoty, které musí byt menší nebo rovny velikosti chromozomu a geny, které mají pozice na těchto hodnotách jsou vzájemně prohozeny. Následně je vrácen zmutovaný potomek jako návratová hodnota funkce.

Třetí metodou je funkce *inverseMutation* [8]. Narozdíl od předchozích dvou nepracuje pouze se dvěma geny, ale může pracovat i s více geny.

Vstup: Chromzom
Výstup: Zmutovaný chromozom
1 inicializace proměnných;
2 inicializace generátorů náhodných čísel;
3 generování náhodných čísel;
4 zrcadlově prohoď pořadí genů ve vybrané části;
5 vrať zmutovaný chromozom;

Algoritmus 8: Kód funkce inverseMutation

Po inicializaci proměnných, gerátorů náhodných čísel a vygenerování těchto čísel se vybere část chromozomu v rozsahu ⟨*náhodné číslo1;náhodné číslo2*⟩ ve které je následně zrcadlově prohozeno pořadí genů a takto upravený chromozom se následně vrátí jako návratová hodnota.

Posledním typem mutace je metoda *Scramble Mutation* reprezentována funkcí *scrambleMutation* [9]. Je podobná metodě *Inverse Mutation*, s rozdílem, že geny ve vybrané části chromozomu se zrcadlově neotáčí, ale náhodně promíchají.

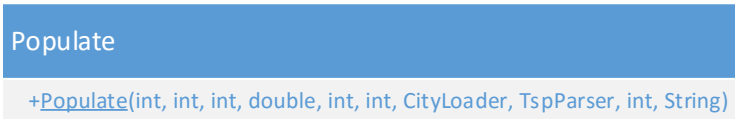
Vstup: Chromzom
Výstup: Zmutovaný chromozom
1 inicializace proměnných;
2 inicializace generátorů náhodných čísel;
3 generování náhodných čísel;
4 náhodně promíchej pořadí genů ve vybrané části;
5 vrať zmutovaný chromozom;

Algoritmus 9: Kód funkce scrambleMutation

Když proběhne počáteční inicializace proměnných, generátorů náhodných čísel a vygenerování náhodných čísel, tak dojde k výběru části chromozomu v rozsahu ⟨*náhodné číslo1;náhodné číslo2*⟩ ve kterém je pak následovně náhodně promícháno pořadí jednotlivých genů. Následně se již zmutovaný chromozom vrací jako návratová hodnota funkce.

4.7 Základní nastavení a export dat

Pro běh samotného algoritmu se používá třída *Populate* jejíž strukturu lze vidět na Obr. 4.13.



Obr. 4.13: Struktura třídy Populate

Obsahuje statickou funkci *populateTSP* [10], která řídí běh genetického algoritmu.

```
Vstup: Typ selekce, Typ křížení, Typ mutace, Pravděpodobnost
       mutace, Počet generací, Počet
       Chromozomů, cityLoader, tspParser, Velikost turnaje, Název
       vstupního souboru
1  inicializace proměnných;
2  export základní hlavičky do formátu .csv;
3  vygeneruj počáteční populaci;
4  ohodnoť počáteční populaci;
5  while Dokud neproběhne zadaný počet populací do
6      while velikost aktuální populace != počet chromozomů do
7          switch typ selekce do
8              vyber Rodič1;
9              vyber Rodič2;
10         endsw
11         switch typ křížení do
12             vytvoř potomka;
13         endsw
14         přidej potomka do aktuální generace;
15     end
16     ohodnoť aktuální populaci;
17     najdi nejlepší chromozom;
18     zapiš to výstupního souboru aktuální generaci a nejlepší hodnotu fitness;
19 end
20 zapiš do výstupního souboru dobu po kterou genetický algoritmus běžel;
21 uzavři výstupní soubor;
```

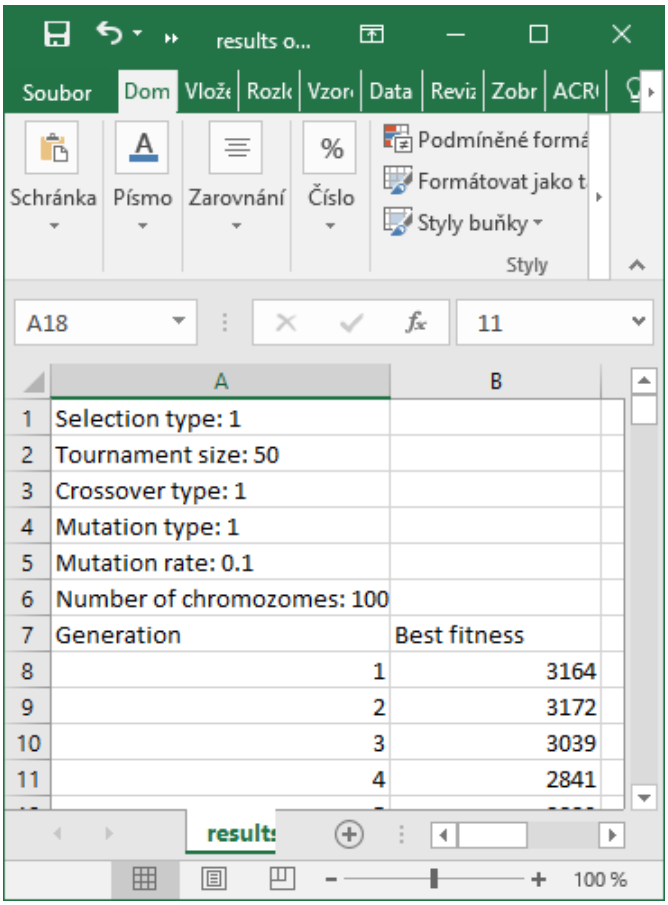
Algoritmus 10: Kód funkce populateTSP

Funkce *populateTSP* začíná inicializací proměnných. Následně dojde k vytvoření výstupního souboru ve formátu *.csv*, do kterého je následně zapsána základní informační hlavička, jenž obsahuje následující informace.

- Typ selekce
- Velikost turnaje
- Typ křížení
- Typ mutace
- Pravděpodobnost mutace
- Počet chromozomů

Poté je vygenerována počáteční populace, která je následně ohodnocena. Pak již začíná samotný běh algoritmu ve kterém se nejprve vytvoří nová populace, která je následně naplnění pomocí selekce a křížení, jehož součástí je i mutace. Po naplnění

nové generace dojde opět k jejímu ohodnocení a na základě nejnižší hodnoty fitness se vybere nejlepší chromozom. Do výstupního souboru se zapíše aktuální generace a hodnota fitness nejlepšího chromozomu. Toto se opakuje tak dlouho, dokud není dosaženo zadaného počtu generací. Nakonec se do výstupního souboru zapíše celkový čas po který algoritmus běžel, uzavře se výstupní soubor a tím funkce končí. Výstupní soubor lze vidět na Obr. 4.14.



Obr. 4.14: Výstupní soubor

Samotný program se pak spouští ve třídě *Main*, ve které se volá již zmíněná funkce *populateTSP*. Nejprve je však nutno nastavit několik proměnných. Jako první se nastavuje cesta ke vstupnímu souboru dat, buď k tabulce vzdáleností, která je výstupem programu *CityGenerator* nebo k souboru z databáze *TSPLib*.

Pak se již nastavují samotné parametry algoritmu, jako první je typ selekce, který může nabývat pouze hodnoty 1, jelikož jiná selekce používána není.

Druhou vstupní proměnnou je typ křížení, který může nabývat hodnot

1. Order One Crossover,
2. PMX Crossover,
3. Edge Recombination Crossover.

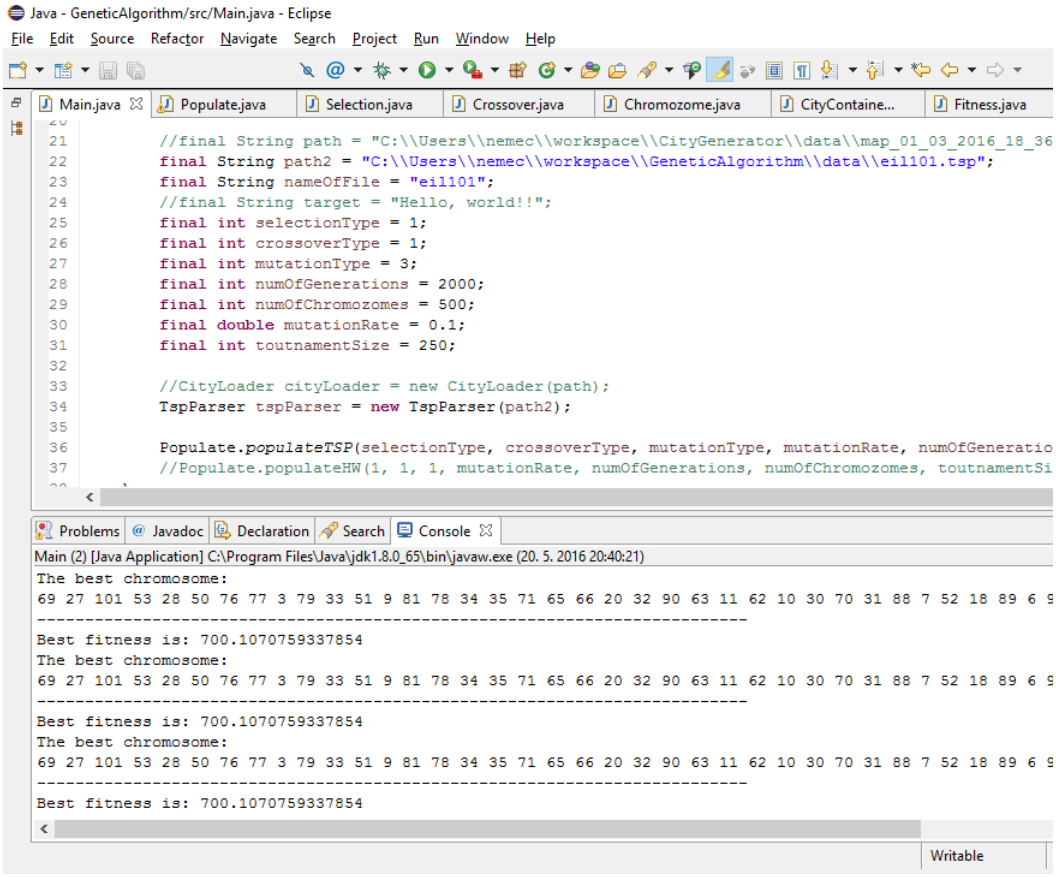
Třetí vstupní proměnnou je typ mutace, který může nabývat hodnot

1. Insert Mutation,
2. Swap Mutation,
3. Inverse Mutation,
4. Scramble Mutation.

Následně se nastavují proměnné

- počet generací,
- počet chromozomů v populaci,
- pravděpodobnost mutace,
- velikost turnaje.

Po nastavení těchto hodnot se již může zavolat funkce *populateTSP* ze třídy *Populate*, do které se tyto hodnoty vloží a tím je nastavení genetického algoritmu. Jeho průběh lze sledovat v konzoli, kam je po každé generaci vypsán nejlepší chromozom a jeho fitness. Na Obr. 4.15 lze prostředí *Eclipse*, ve kterém byl algoritmus vyvíjen a zároveň lze v něm i algoritmus spustit. Lze také vidět praktické nastavení jednotlivých vstupních proměnných a výstup algoritmu do konzole.



Obr. 4.15: Nastavení algoritmu

Na obr. 4.15 si také lze všimnout, že třída *Populate* obsahuje ještě druhou funkci a to *populateHW*. Tato třída slouží k vygenerování zadaného řetězce pomocí genetických algoritmů. Jelikož toto však nebylo náplní této diplomové práce, je spíš třída určená jen pro zajímavost, že se pomocí genetických algoritmů dají řešit opravdu všlijaké problémy.

4.8 Dokumentace

Jelikož nebylo možné v této kapitole popsat detailně všechny naimplementované funkce, byla vytvořena dokumentace, v níž jsou popsány jednotlivé funkce, jejich vstupní argumenty a jejich návratové typy.

Dnes již existuje mnoho automatizovaných postupů, jak dokumentaci generovat a není potřeba ji psát celou ručně, což by se dalo označit za velmi neefektivní.

Jedním z nástrojů, který umí vygenerovat dokumentaci ze zdrojových souborů se jmenuje *Doxygen*. Jedná se o standardní nástroj, který generuje dokumentaci primárně z jazyka *C++*, ale podporuje i další programovací jazyky, jako například *C*, *Objective-C*, *C#*, *JAVA*, *PHP* a další.

Doxygen umí generovat on-line dokumentaci za použití *HTML* a také ve formátu LaTeX. *Doxygen* lze také nakonfigurovat, aby extrahoval strukturu kódu ze zdrojových souborů, které postrádají dokumentaci. Toto je velmi nápomocné při analyzování zdrojových kódu, které obsahují tisíce řádků kódu a je těžké se v nich vyznat. Dále umí automaticky vizualizovat vztahy mezi jednotlivými třídami, jako jsou grafy závislosti, diagramy dědičnosti a spolupráce. V neposlední řadě je možno *Doxygen* použít i pro tvorbu normální dokumentace.

Doxygen je vyvíjet pod platformou *Mac OS X* a *Linux*, ale je velice multiplatformní, takže ho lze spustit po většinou distribucí operačního systému *Unix* a hlavně také *Windows*.

4.8.1 Instalace a nastavení Doxygenu

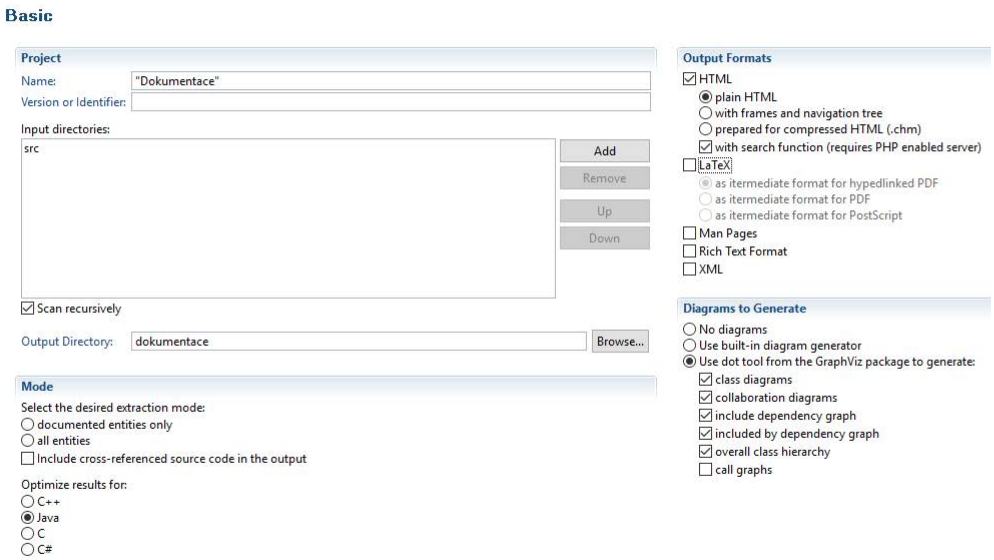
Aby bylo možné používat *Doxygen*, je nutné splnit dvě podmínky. Nainstalovat si samotný *Doxygen* na svůj počítač a zadruhé zprovoznit plug-in ve vývojovém prostředí, který se bude starat o generování samotné dokumentace.

Instalace jedné nebo druhé části je velice uživatelsky přivětivá a i uživatel, který nemá s *Doxygenem* dosud žádnou zkušenost nebude mít problémy jej nainstalovat. Samotný *Doxygen* je nejlepší stáhnout ze stránek autora a v závislosti na používaném operačním systému zvolit vhodnou verzi. Následně je nutno nainstalovat i vhodný plug-in. V závislosti na použitém vývojovém prostředí existuje několik možností. Genetický algoritmus z této diplomové práce byl vyvíjen v prostředí *Eclipse* a proto

byl vybrán plug-in s názvem *Eclox*. Ten se do *Eclipse* instaluje pomocí integrovaného instalátoru.

4.8.2 Nastavení a export

Po instalaci je již možné přistoupit k nastavení samotného *Doxygenu* a exportu dokumentace. Nejprve je nutné provést nastavení *Doxygenu*. Na Obr. 4.16 lze vidět, jak takové nastavení vypadá.



Obr. 4.16: Nastavení Doxygenu

Důležité jsou následující položky

- name,
- input directories,
- output directory,
- optimize results for,
- output formats.

Položka *name* slouží k nastavení jména dokumentace. V *Input directories* se nastavují vstupní složky, které obsahují zdrojové kódy, ze kterých bude výsledná dokumentace tvořena, v položce *Output Directory* se nastavuje výstupní složka. *Optimize results for* slouží k upřesnění, v jakém jazyce byl zdrojový kód napsán a Doxygen podle toho upravuje dokumentaci. V poslední položce *Output Formats* se natavují výstupní formáty, je na výběr ze dvou, *HTML* a LaTeX.

Po nastavení je již možné exportovat samotnou dokumentaci. *Doxygen* proskenuje jednotlivé třídy a sestaví jejich struktury. Pokud chceme popsat jednotlivé funkce, jejich vstupní argumenty a návratové hodnoty, obsahují je-li funkce. Následně

je možno popsat celé třídy. Popis se provádí pomocí specializovaných komentářů. Příklad je zobrazen na Obr. 4.17.

```
/**
 *
 * \brief Trida obsahujici funkce pro vypocet hodnot fitness
 * @author Jan Nemec
 *
 */

public class Fitness {

    /**
     * Funkce pro vypocet fitness hodnoty chromozomu pro TSP.
     * Scita vzdalenosti mezi jednotlivymi mesty, tak
     * jak cestu reprezentuje
     * chromozom.
     * @param chromozome Vstupni chromozom.
     * @param table Tabulka reprezentujici vzdalenosti mezi mesty.
     * @return Metoda vraci ciselnou hodnotu fitness.
     */
    public static double tspFitness(Chromozome chromozome, double[][] table){
```

Obr. 4.17: Ukázka popisu

Nad deklarací třídy můžeme vidět, že je zde uveden krátký popis třídy a jméno autora. Popis funkce lze vidět nad funkci *fitnessTSP*. Opět obsahuje krátký popis funkce a následně jsou deklarovány vstupní parametry v pořadí v jaké do funkce vstupují a nakonec návratová hodnota.

4.8.3 Ukázka dokumentace

Do nastavené složky pro výstup se po vygenerování vytvoří složka *HTML*, složka *Latex* nebo případně oboje, v závislosti na nastaveném formátu výstupu. Pro tuto diplomovou práci byla vygenerována dokumentace v *HTML*, které je k dispozici buď na přiloženém CD v případě tištěné verze nebo jako příloha v případě elektronické verze.

Ve složce *HTML* se nachází soubor *index.html*, kterým se dokumentace spouští, otevře se ve výchozím internetovém prohlížeči. Po spuštění se zobrazí úvodní strana, která obsahuje pouze název dokumentace.

Všechny důležité informace se ukrývají v záložce *Classes*. Ta obsahuje seznam všech tříd a jejich popis. Po rozklinutí třídy se objeví struktura třídy, která obsahuje veškeré proměnné a popis jednotlivých funkcí, jak lze vidět na Obr. 4.18.

Chromozome Class Reference

Trida reprezentujici chromozom pro TSP. More...

Public Member Functions

	Chromozome (int numOfGenes)
	Chromozome (int size, int value)
void	**printChromozome** ()
int	**getChromozomeSize** ()
int	**getValueOfGene** (int position)
void	**setFitness** (double fitness)
double	**getFitness** ()
double	**getSpecialFitness** ()
void	**setSpecialFitness** (double specialFitness)
void	**setGene** (int gene, int pos)
ArrayList< Integer >	**get** ()

Detailed Description

Trida reprezentujici chromozom pro TSP.

Author

Jan Nemec

Constructor & Destructor Documentation

Chromozome.Chromozome (int numOfGenes)

Konstruktor, ktery vytvari kompletni chromozom a naplni jej hodnotami.

Parameters

numOfGenes Pocet genu v chromozomu

Obr. 4.18: Ukázka dokumentace

5 NAMĚŘENÉ VÝSLEDKY

5.1 Určení optimálního nastavení algoritmu

Aby bylo možné určit výkonnost algortimu, bylo nejprve nutné najít jeho optimální nastavení. Jako vstupní testovací soubor byl vybrán *eil101* z databáze *TSPLib*. Následně byly vybrány testovací hodnoty, aby bylo možné určit jejich vliv na výkonnost algoritmu, jejich přehled lze vidět v Tab. 5.1 První kolo měření sloužilo

Tab. 5.1: Tabulka vstupních hodnot

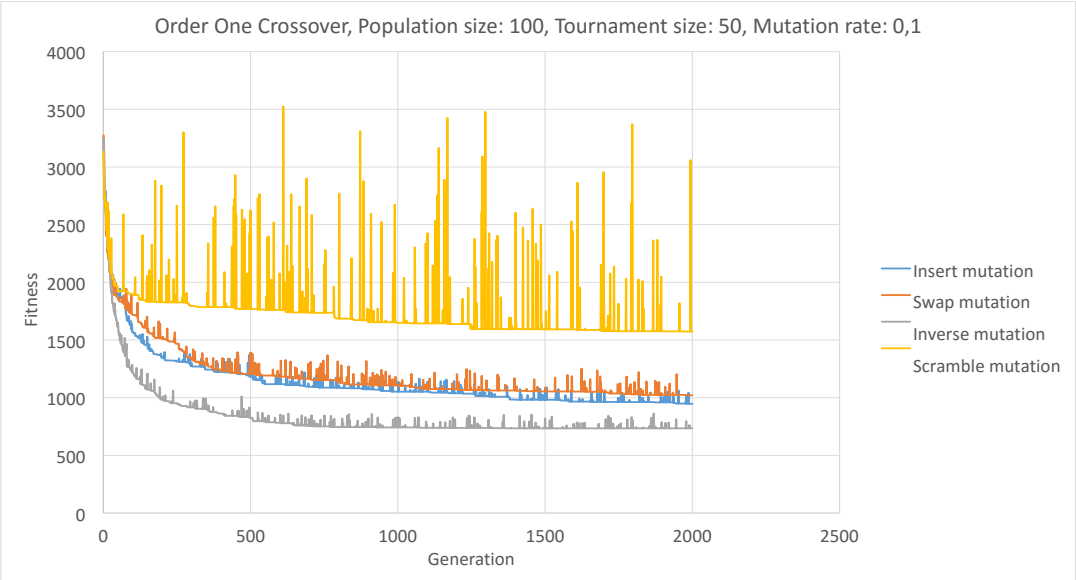
Typ křížení	Order One, PMX, Edge Recombination
Typ mutace	Insert, Swap, Inverse, Scramble
Pravděpodobnost mutace	0,1 0,2 0,3
Velikost populace	100, 200, 300, 500
Velikost turnaje	50, 100, 150, 250

k určení nejlepšího typu mutace. Měření postupně probíhalo pro pravděpodobnost mutace *0,1* a všechny typy křížení, mutace a všechny velikosti populace a turnaje a byly získány následující výsledky. Z grafu 5.1 pro určení výkonnosti jednotlivých mutací s použitím metody křížení Order One je patrné, že mutace typu *Inverse* má nejlepší výkonnost. Nejrychleji aproximu k výsledku a zároveň taky dosahuje znatelně lepších výsledků než ostatní metod. Metody *Swap* i *Insert* mají zhruba stejný průběh, avšak metoda *Scramble* dosahuje nejhorších hodnot a má obrovské výkyvy co se týče průběžných výsledků, lze ji tedy označit za naprosto nedostačující jak z hlediska rychlosti aproximace, tak i z hlediska získaného výsledku. Stejný průběh mají i grafy pro větší hodnotu velikosti populace a turnaje.

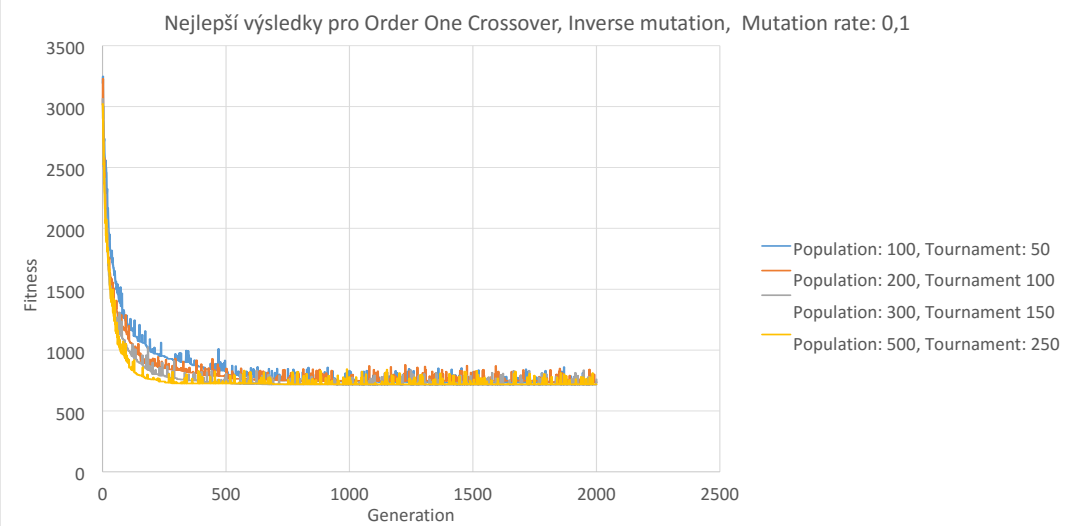
Pokud srovnáme průběhy v grafech pro mutaci *Inverse*, tak zjistíme, že velikost populace a turnaje značně ovlivňuje rychlost aproximace k optimálnímu výsledu, jak lze vidět v grafu 5.2. Lze tedy říci, že velikost populace a turnaje nám ovlivňuje rychlost aproximace k optimálnímu výsledku. Avšak velikost populace ovlivňuje dobu, po kterou algoritmus běží. Pro malý počet vstupních měst je výsledek zanedbatelný, avšak s nárůstem počtu měst se doba trvání rapidně zvyšuje.

Po naměření hodnot i pro další metody křížení lze porovnat jejich vliv na výkonnost za použití metody mutace *Inverse*. Po měření byly jednotlivé hodnoty vyneseny do grafu 5.3.

Z grafu lze vyčíst, že metody *Order One* a *PMX* mají zhruba stejný průběh přičemž metoda *Edge Recombination* za nimi zaostává co se týče rychlosti aproximace k optimálnímu výsledku. Vzhledem k její vyšší časové náročnosti se tedy taky jeví jako neefektivní.



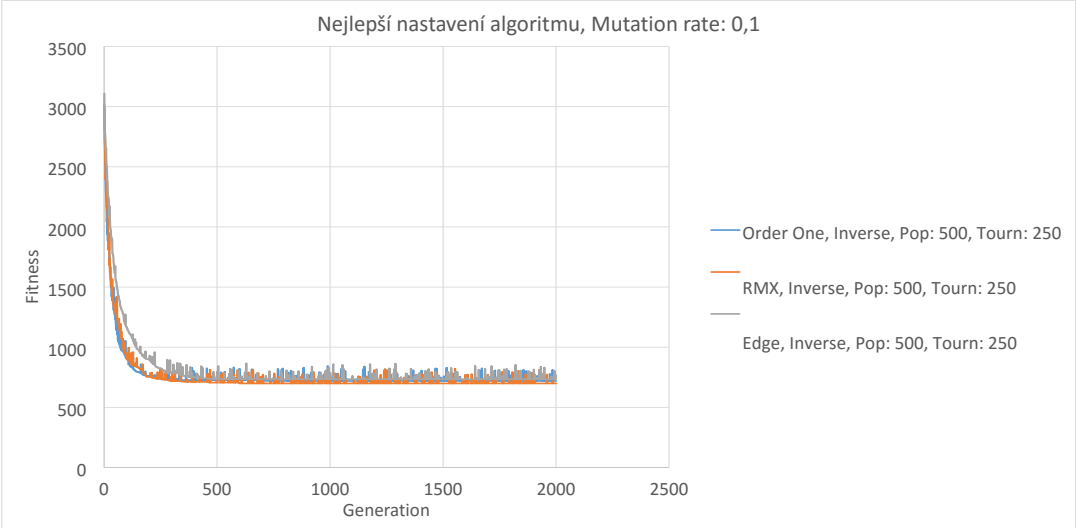
Obr. 5.1: Vliv mutace na výsledek



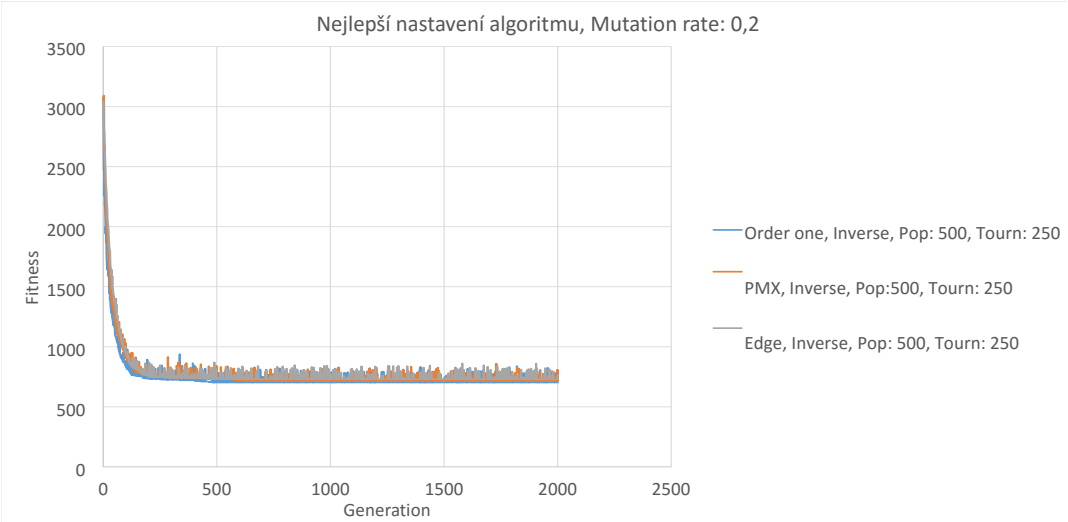
Obr. 5.2: Vliv populace na výsledek

Stejné měření pro vstupní hodnoty byly provedeny i pro hodnoty pravděpodobnosti mutace *0,2* a *0,3*. Následně byly shromážděny nejlepší výsledky pro jednotlivé typy křížení, které lze vidět v grafu 5.4 a grafu 5.5. Lze pozorovat, že nejrychlejší aproximaci k optimálnímu výsledku má metoda *Order One*. Z grafů lze taky vyčíst, že velikost pravědpodobnosti v měřených hodnotách nějak zvlášť neovlivňuje získáné výsledky ani rychlost aproximace.

Následně již zbývalo pouze určit nejlepší hodnotu pravděpodobnosti mutace. Nejlepší výsledky pro každou hodnotu byly vloženy do grafu 5.6, ze kterého lze pozorovat, že pro všechny hodnoty jsou výsledné grafy téměř totožné. Z toho lze



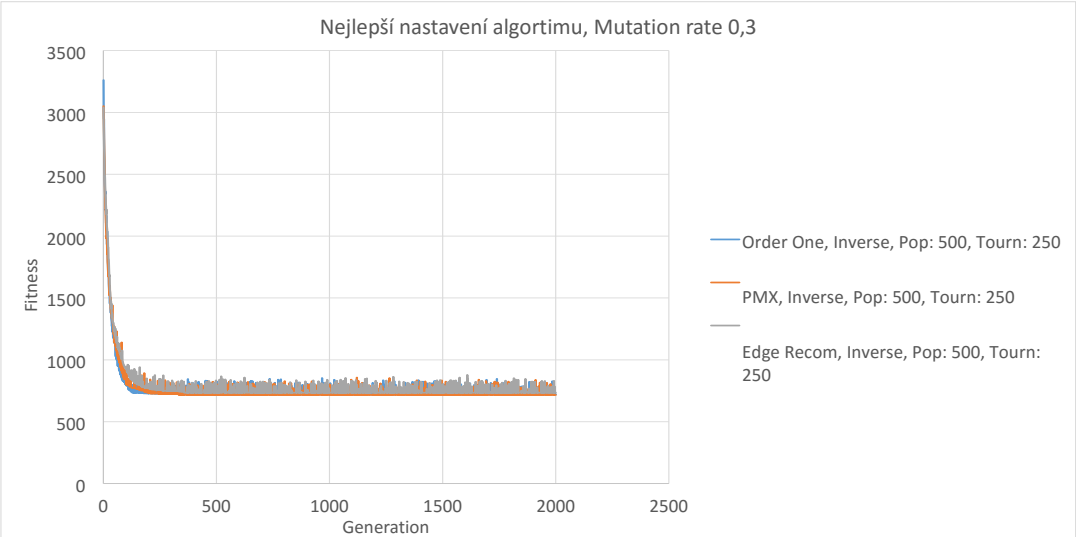
Obr. 5.3: Nejlepší výsledky algoritmu pro zvolenou mutaci



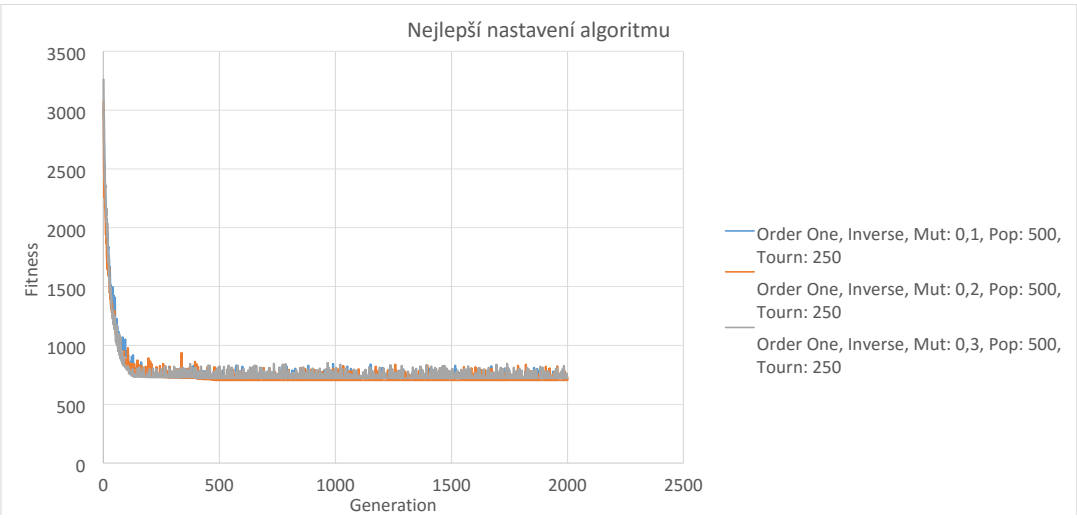
Obr. 5.4: Nejlepší výsledky algoritmu pro zvolenou mutaci

usoudit, že pokud je mutace v intervalu $\langle 0{,}1;0{,}3\rangle$ tak získáme ideální kompromis mezi rychlostí aproximace a optimálním výsledkem.

Ideální nastavení algoritmu ze získaných hodnot lze shrnout tak, že pokud chceme získat přesný výsledek, je nejlepší použít mutaci typu *Inverse*, mezi metodami křížení *Order one* a *PMX* je nepatrný rozdíl, avšak metoda *Order One* vykazuje nepatrně lepší výsledky co se týče rychlosti aproximace a pravděpodobnost mutace ve výše zmíněném intervalu nijak zásadně neovlivňuje získané výsledky. Nejlepší nastavení algoritmu je pro lepší přehlednost znázorněno v tabulce 5.2.



Obr. 5.5: Nejlepší výsledky algoritmu pro zvolenou mutaci



Obr. 5.6: Nejlepší nastavení algoritmu

Tab. 5.2: Souhrn nejlepšího nastavení algoritmu

Nejlepší typ mutace	Inverse
Nejlepší typ křížení	Order One
Velikost populace	500, čím větší, tím lepší
Velikost turnaje	Polovina velikosti populace
Pravděpodobnost mutace	Interval <0,1;0,3 >

5.2 Naměřené výsledky pro různá vstupní data

Po zjištění ideálního nastavení algoritmu, lze přistoupit k měření dalších vstupních dat. Budou použity celkem 4 soubory z databáze *TSPLib* a 4 vygenerované soubory z programu *CityGenerator*. Pro soubory z databáze *TSPLib* budou porovnány naměřené výsledky porovnány s optimálními řešeními, které také nabízí. Výstupem tohoto měření tedy bude tabulka 5.3 obsahující naměřené hodnoty, optimální řešení a případná odchylka od optimálního řešení.

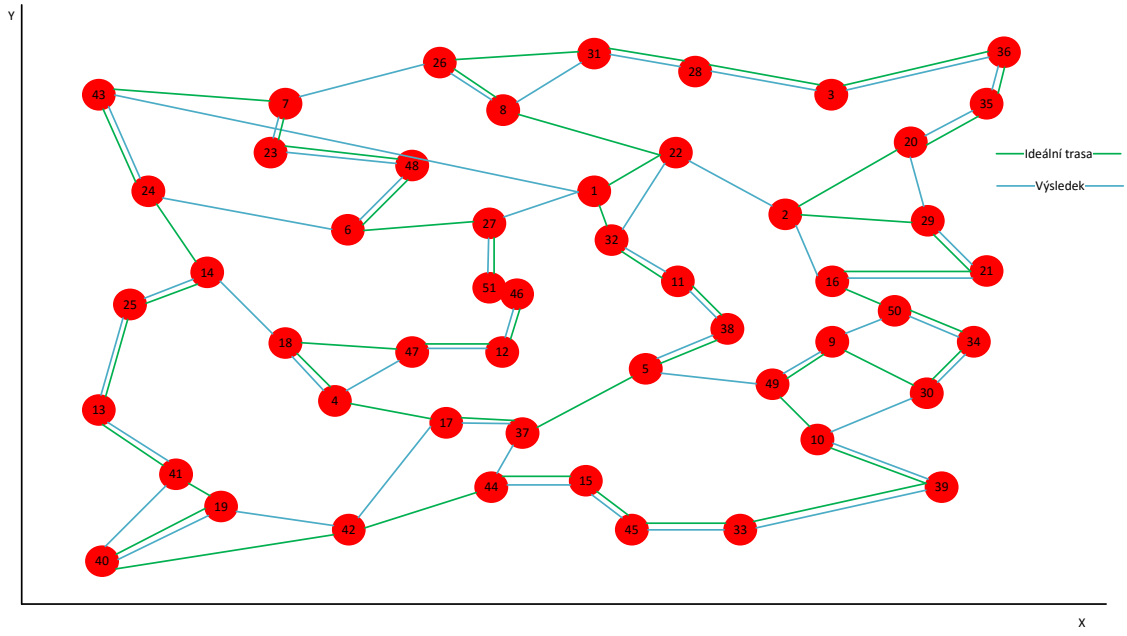
Tab. 5.3: Naměřené výsledky

Název souboru	Výsledné řešení	Optimální řešení	Odchylka
Eil51	452	426	6,1 %
Eil101	708	629	12,6 %
Lin318	47490	42029	13 %
Pr1002	476496	259045	84 %
100 měst	85367	-	-
200 měst	125993	-	-
400 měst	177293	-	-
800 měst	284391	-	-

Lze pozorovat, že pro první tři vstupní soubory je odchylka od optimálního řešení téměř stejná. U čtvrtého souboru vzrostla odchylka na 84 %, což je pravděpodobně způsobeno tím, že algoritmus neprošel dostatečně velkým potřebným počtem iterací, nebo velikost populace nebyla dostatečně vysoká a algoritmus tak aproximoval k výsledku příliš dlouho.

Na Obr. 5.7 lze vidět grafické srovnání dosaženého výsledku a optimálního řešení. Pozice jednotlivých měst byly vyneseny přesně v souladu se souřadnicemi, které obsahoval vstupní soubor *Eil101* a následně byly zakresleny dvě cesty, modře označená označuje cestu, která byla výsledkem genetického algoritmu, zelená cesta pak označuje optimální cestu, jejíž trasu lze získat z databáze *TSPLib*[4].

Lze pozorovat, že v určitých částech se výsledky shodují, což dosvědčuje, že nastavení algoritmu je dobré, avšak v několika částěch se cesty rozcházejí, většinou se však jedná o minimální rozdíly. Největší rozdíl je však na konci obou cest, kde řešení, které vzniklo z genetického algoritmu volí nesmyslně cestu z města 43 do města 1, což nepochybně přidává nejvíc na vzniklé odchylce. Naopak u ideálního řešení lze vidět, jak elegantě cestu řeší a neobjevují se zde nesmyslně dlouhé jednotlivé úseky.



Obr. 5.7: Nastavení algoritmu

5.3 Porovnání výsledků

Bohužel se nepodařilo najít práci, která se takto komplexně věnuje nastavení všech jednotlivých parametrů genetického algoritmu najednou. Je však několik prací, které se věnují jednotlivým sekcím zvlášť, tj. jedna z prací se zabývá optimální metodou mutace, druhá zase nejlepším typem křížení a poslední práce nejlepším typem selekce.

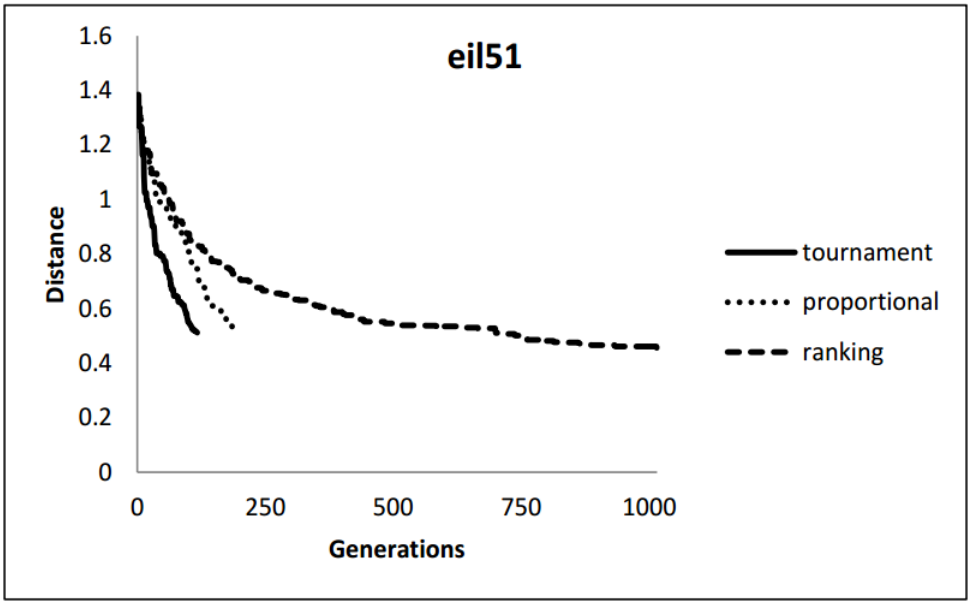
Pokud budeme postupovat tak, jak pracují genetické, potažmo evoluční algoritmy, tak prvním výsledkem, který bude porovnán je selekce. Touto problematikou se zabývala práce [3]. V této práci byly testovány celkově tři metody pro selekci a to

- ruletová selekce využívající hodnotu fitness,
- ruletová selekce využívající pořadí,
- turnajová selekce.

Pro křížení byla použita metoda *Order One* a pro mutaci metoda *Inverse*. Na Obr. 5.8 lze vidět dosažené výsledky. Z hlediska rychlosti aproximace je nejlepší metodou pro selekci právě turnajová selekce, avšak ruletová selekce využívající pořadí dosahuje nejlepšího výsledku, ovšem pak za cenu několikanásobně pomalejší aproximace k optimálnímu výsledku.

Optimální metodou pro křížení se zabývala práce[13]. Bylo testováno celkem pět metod pro křížení a to

- Order One,
- NWOX,



Obr. 5.8: Srovnání metod pro mutaci

- PMX,
- UPMX,
- CX.

Jako metoda pro selekci byla vybrána ruletová selekce využívající hodnoty fitness a jako mutace metoda *Inverse*.

Výsledný obrázek s naměřenými hodnotami nelze přiložit, jelikož dokument ho osahuje ve špatné kvalitě, ale jako nejleoší metoda byla zjištěna *Order One*.

Nejlepší metodou pro mutaci se zabývala práce[10]. Byly testovány metody

- Swap,
- Inverse,
- CIM,
- Thors.

Pro selekci byla vybrána metoda ruletové selekce založená na hodnotě fitness a jako křížení metoda *Order One*.

Z důvodu špatné kvality opět nelze přiložit výsledný graf, ale jako nejlepší metodou pro mutaci byla zjištěna metoda *Inverse*.

V tabulce 5.4 jsou porovnány jednotlivé položky optimálního nastavení získané touto prací a ostatními pracemi.

Tab. 5.4: Srovnání dosažených výsledků

Typ operace	Tato práce	Referenční práce
Selekce	Turnajová	Turnajová/Ruletová, pořadí
Křížení	Order One	Order One
Mutace	Inverse	Inverse

Ze srovnání lze tedy říci, že výsledky optimálního nastavení genetického algoritmu, řešícího TSP, které zjistila tato diplomová práce se shodují s výsledky, kterých bylo dosaženo i v ostatních pracech zabývajících se tímto tématem.

6 ZÁVĚR

Tato diplomová práce se zabývala evolučními algoritmy. V první kapitole byly popsány evoluční algoritmy, jejich princip fungování, operátory, které využívají a následně pak nastíněny problémy, k jejichž řešení se dají využít.

Ve druhé kapitole již byla popsána konkrétní varianta evolučních algoritmů, genetické algoritmy. Detailně bylo popsáno několik genetických operátorů pro selekci, křížení a mutaci doplněné o ilustrace jejich fungování.

Třetí kapitola této práce se zabývá problémem obchodního cestujícího, je zde vysvětlen princip problému, jeho historie a následně typy, na které se dělí.

Čtvrtá kapitola se již zabývá samotnou implementací genetického algoritmu v jazyce JAVA. Jsou postupně popsány vstupní data, jejich generování a načítání do programu a implementace chromozomu a populace. Následně je popsána metodika hodnocení chromozomu funkcí Fitness. Následně jsou popsány třídy pro operátory selekce, křížení a mutace, jejich struktura a pseudokódy znázorňující jak byly naimplementovány. Poté je vysvětlen princip, jakým se nastavuje genetický algoritmus, je popsána implementace jádra genetického algoritmu a jakým způsobem jsou exportovány výsledky. V závěru kapitoly je popis dokumentace, která byla vytvořena pro lepší chápaní jednotlivých funkci obsažených v celém programu.

V poslední kapitole této práce jsou prezentovány získané výsledky, je zde nastíněno, jaký způsobem probíhalo získaní optimálního nastavení, následně je správnost tohoto nastavení ověřena na několika vstupních datech. V poslední části je srovnání optimálního nastavení získané touto prací s ostatními pracemi, které se zabývaly stejným problémem.

Byla potvrzena správnost těchto výsledků, jelikož se tyto nastavení shodovaly, čímž lze pokládat výsledky této diplomové práce za správné.

Co se týče rozšíření této diplomové práce, bylo by možné naimplementovat několik dalších genetických operátorů především pro selekci, aby bylo možné ověřit jejich vliv v kombinaci s ostatními operátory.

LITERATURA

[1] KUNIYIL, Mithun. *Introduction to Genetic Algorithms* [online]. 2014, 2015-5-4, : 81 [cit. 2015-12-11]. Dostupné z: https://www.scribd.com/doc/54585910/2/History-Of-Genetic-Algorithms.

[2] VOLNÁ, Eva. *Evoluční algoritmy a neurovnové sítě* [online]. 2012, : 152 [cit. 2015-12-11]. Dostupné z: http://www1.osu.cz/~volna/Evolucni_algoritmy_a_neuronove_site.pdf

[3] GERAGHTY, John a Noraini Mohd RAZALI. *Genetic Algorithm Performance with Different Selection Strategies in Solving TSP.* Proceedings of the World Congress on Engineering 2011 [online]. London, 2011, 2011-3-6, 2: 6 [cit. 2015-12-11]. ISSN 2078-0966. Dostupné z: http://www.iaeng.org/publication/WCE2011/WCE2011_pp1134-1139.pdf

[4] REINELT, Gerhard. *TSPLIB*. Universität Heidelberg [online]. [cit. 2015-12-11]. Dostupné z: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

[5] *Traveling Salesman Problem* [online]. 2015 [cit. 2015-12-11]. Dostupné z: http://www.math.uwaterloo.ca/tsp/index.html

[6] OBITKO, Marek. *Introduction to Genetic Algorithms: Crossover and Mutation* [online]. 1998 [cit. 2015-12-11]. Dostupné z: http://www.obitko.com/tutorials/genetic-algorithms/crossover-mutation.php

[7] POPELKA, Ondřej. *Umělá inteligence: Učící se algoritmy* [online]. 2015-1-10 [cit. 2015-12-11]. Dostupné z: https://akela.mendelu.cz/~xpopelka/cs/ui/ucici/

[8] POHLHEIM, Hartmut. *Evolutionary Algorithms: Overview, Methods and Operators* [online]. 2006-12, : 101 [cit. 2015-12-11]. Dostupné z: http://www.geatbx.com/download/GEATbx_Intro_Algorithmen_v38.pdf

[9] BURGET, Radim. *Teoratická informatika*. Brno, 2013. Vysoké učení technické v Brně.

[10] ABDOUN, Otman, Chakir TAJANI a Jaafar TAJANI. *Analyzing the Performance of Mutation Operators to Solve the Travelling Salesman Problem* [online]. 2014, : 18 [cit. 2015-12-15]. Dostupné z: http://arxiv.org/ftp/arxiv/papers/1203/1203.3099.pdf

[11] ZAKIR H. Ahmed. *Genetic Algorithm for the Traveling Salesman Problem using Sequential Constructive Crossover Operator.* IJBB 3(6). 2010 [cit. 2016-05-24]

[12] GOLDBERG, David. *Algorithm in Search, Optimization, and Machine Learning.* Addison Wesley, 2009. [cit. 2016-05-24]

[13] ABDOUN Otman, ABOUCHABAKA Jaafar. *A Comparative Study of Adapative Crossover Operators for Genetic Algorithms to Resolve the Travelling Salesman Problem* [online]. 2011 [cit. 2016-05-24] Dostupné z: `https://arxiv.org/ftp/arxiv/papers/1203/1203.3097.pdf`

[14] WHILEY, Darrel, STARKWEATHER, Timothy, FUQUAY D'Ann. *Scheduling problems and traveling salesman: The genetic edge recombination operator* International Conference in Genetic Algorithms, 1989. [cit. 2016-05-24]

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

EA	Evoluční Algoritmy
GA	Genetické Algoritmy
TSP	Problém obchodního cestujícího – Travelling Salesman Problem