UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

Využití optimalizace v logistice

Lukáš Míšek

Bakalářská práce
2020

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2019/2020

ZADÁNÍ BAKALÁŘSKÉ PRÁCE
(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Lukáš Míšek**
Osobní číslo:	**I15016**
Studijní program:	**B2612 Elektrotechnika a informatika**
Studijní obor:	**Řízení procesů**
Téma práce:	**Využití optimalizace v logistice**
Zadávající katedra:	**Katedra řízení procesů**

Zásady pro vypracování

Cílem bakalářské práce bude vyzkoušet možnosti využití optimalizace pro řešení vybraného logistického problému (bin packing, TSP, plánování...).
V teoretické části student popíše vybrané problémy, existující metody pro jejich řešení, v části praktické si vybere konkrétní problém, navrhne a provede experimenty (naprogramuje je ve vybraném programovacím jazyce) a vyhodnotí jejich výsledky.

Rozsah pracovní zprávy: **30-40 stran**
Rozsah grafických prací:
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

HYNEK, Josef. Genetické algoritmy a genetické programování. Praha: Grada, 2008. Průvodce (Grada). ISBN 978-80-247-2695-3.
MITCHELL, Melanie. An introduction to genetic algorithms. Cambridge, Mass.: MIT Press, c1996. ISBN 978-0262133166.
Simulated annealing: introduction, applications and theory. Hauppauge, NY: Nova Science Publishers, 2018. ISBN 978-1536136746.

Vedoucí bakalářské práce: **Ing. Jan Merta**
Katedra řízení procesů

Datum zadání bakalářské práce: **17. prosince 2019**
Termín odevzdání bakalářské práce: **7. května 2020**

Prohlášení

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 9/2012, bude práce zveřejněna v Univerzitní knihovně a prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 15. 03. 2020

Lukáš Míšek

ANOTACE

Cílem bakalářské práce bude vyzkoušet možnosti využití optimalizace pro řešení vybraného logistického problému (bin packing, TSP, plánování...). V teoretické části student popíše vybrané problémy, existující metody pro jejich řešení, v části praktické si vybere konkrétní problém, navrhne a provede experimenty (naprogramuje je ve vybraném programovacím jazyce) a vyhodnotí jejich výsledky.

KLÍČOVÁ SLOVA

Optimalizace, obchodní cestující, umělá inteligence, genetické programování, logistika.

ANNOTATION

Goal of this work is to try use of the optimilization for solving in logistic problém (bin packing, TSP, planing etc). In teoretical part student will describe chosen problem, existing methods, design and realize experiments (programm in chosen programming language) and analyse results.

KEYWORDS

Optimization, travelling seller, artificial inteligence, genetic programming, logistics.

OBSAH

SEZNAM ZKRATEK

array	datový typ pole se statickým rozměrem
bmp	soubor s obrázkem ve formátu bmp
csv	textový soubor s oddělovači
C#	programovací jazyk C#
double	datový typ reálné číslo
char	datový typ znak
GA	genetický algoritmus
int	datový typ celé číslo
jpg	soubor s obrázkem ve formátu jpg
list	datový typ pole s dynamickým rozměrem
pdf	soubor ve formátu pdf
string	datový typ řetězec znaků
TSP	travelling salesman problem
txt	textový soubor
VRP	vehicle routing problém
xls	excel soubor

SEZNAM ILUSTRACÍ

ÚVOD

Logistikou je možné rozumět proces, kdy je přemístěn předmět z jednoho místa do druhého. V současnosti je logistika důležitým aspektem oblastí, jako je zásobování nebo obchod. Na takové procesy je nutné vynaložit určité náklady. Tyto náklady často rostou s objemem přepravovaných předmětů a se vzdáleností, přes kterou musí být předměty přepraveny.

Samotný logistický proces je možné rozdělit na mnoho menších celků, kterými se můžeme samostatně zabývat. Mezi tyto celky lze zařadit oblast marketingu, technologické řešení přepravy nebo samotnou realizaci přepravy. Realizaci přepravy je možné zdokonalit v mnoha ohledech. Jedním z nich je například snížení celkové ujeté vzdálenosti, kterou je nutné překonat přepravními prostředky. Právě snížení celkové ujeté vzdálenosti je příkladem optimalizace, kterým se tato práce zabývá.

Optimalizace se v dnešní době používá ke zdokonalování mnoha existujících procesů. Zdokonalováním procesů se následně mohou snížit náklady tohoto procesu. Optimalizace rovněž může sloužit ke snížení dopadů na životní prostředí nebo umožňuje rozšířit kapacitu procesu se stávajícími prostředky.

1 SEZNÁMENÍ SE S PROBLEMATIKOU

Problematika hledání vhodné kombinace řešení, ať už je to rozložení palet do kamionu, tvorba rozvrhu pro školu, nebo dosazení doktorů na služby, je v dnešní době řešena specializovanými pracovníky. Tito pracovníci používají rozličné metody k nalezení těchto řešení.

Jednou takovou metodou je takzvaná hrubá síla. V tomhle případě jsou systematicky generovány vhodné kombinace, které jsou následně vyhodnocovány. Tento postup přesto, že funguje, tak není optimální z hlediska časové náročnosti. Zejména protože čas pro nalezení vhodné kombinace se exponenciálně zvyšuje s rostoucími vstupy.

Tyto vstupy jsou konkrétní požadavky zadavatelů. V práci tvorba rozvrhů pomocí genetických algoritmů (Horký, 2008) jsou zmíněny dva druhy omezení. Takzvaná tvrdá omezení a měkká omezení. Tvrdá omezení musí být splněna vždy. Měkká omezení nemusí být nutně splněna. Nicméně jejich splnění pozitivně ovlivní ohodnocení kombinace pomocí hodnotící funkce.

Mezi tvrdá omezení patří podmínky, které musí být dodrženy. Například na jedno paletové místo musí být dosazena pouze jedna paleta, kamion nemůže obsahovat více palet, než je jeho kapacita, a skladník nemůže nakládat více než jeden kamion v daný okamžik. Splněním těchto požadavků dosáhneme správného řešení. Toto řešení může být jediné a zároveň jich může existovat více. Měkká omezení jsou taková, která nemusí být splněna. Nicméně jejich splnění může negativně penalizovat ohodnocení výsledné kombinace. Může to být například zvláštní požadavek skladníka, který chce nakládat pouze některé kamiony, palety, které mohou být naloženy pouze do některých kamiónů. Některé kamiony mohou být prioritní a je nutné je naložit dříve a podobně.

Vyhodnocením těchto vstupů získáme kombinaci, která může být správným řešením. Pokud kombinace vyhoví všem tvrdým podmínkám, tak se jedná o správné řešení. Pokud nevyhoví, tak se jedná o nesprávné řešení. Zároveň je kombinace pomocí ohodnocující funkce hodnocena a je jí přiřazena hodnota, která udává, jak vhodná je kombinace. Vyšší fitness skóre představuje vhodnější řešení.

Kombinaci možného řešení získáme náhodným vygenerováním. Po vyhodnocení kombinace je možné hledání dalších kombinací ukončit, nebo pokračovat. V případě pokračování je další generace nových kombinací vytvořena s ohledem na ty nejúspěšnější kombinace. Zároveň jsou kombinace náhodným způsobem nepatrně pozměněny, aby algoritmus neuvízl v lokálním extrému.

1.1 Rozbor zadání

Úkolem práce je vyzkoušet možnosti a využití optimalizace pro řešení vybraného logistického problému. Logistickým problémem je možné rozumět proces, kde je nutné přesunout věci z místa A do místa B, případně libovolného množství dalších míst. Hlavním účelem optimalizace potom může být snížení nákladů přepravy předmětů, zvýšení kapacity přepravy, urychlení plánování přepravy, nebo zjednodušení procesu celé přepravy.

Hlavním předmětem této práce je optimalizovat řešení VRP (vehicle routing problem). To znamená snížit celkové náklady potřebné pro přepravu předmětů. Hlavními parametry budou celková ujetá vzdálenost a počet dopravních prostředků. Vedle těchto dvou parametrů existuje mnoho dalších, které ovlivňují celý proces.

Parametry jako časová okna, kdy může být předmět dodán, nebo typ dopravního prostředku, který je schopen předmět přepravit, nebudou brány v úvahu. Výstupem této práce je algoritmus, nebo postup, který je možné aplikovat v reálné situaci.

1.2 Rešerše literatury

Úkolem práce „Tvorba rozvrhů pomocí genetických algoritmů“ (Horký, 2011) bylo vytvořit program, který umožní uživateli vytvořit rozvrhy pro školu. Tyto rozvrhy jsou tvořeny s ohledem na mnohé požadavky. Například v jedné třídě nemůže být více vyučovaných hodin současně, jeden učitel v danou chvíli může učit pouze jednu třídu, žáci nemohou mít více než jednu hodinu současně a podobně.

Důvodem ke vzniku tohoto problému je komplexnost řešení. Pro požadavky, které jsou omezeny pouze několika podmínkami, je řešení poměrně triviální. Příkladem může být škola se čtyřmi učiteli, čtyřiceti žáky, jednou místností a čtyřmi vyučovacími předměty, kde každý učitel vyučuje právě jeden předmět po dobu jedné hodiny. Zde jsou žáci rozmístěni do jedné třídy a postupně je učí právě jeden učitel. Zde existuje mnoho správných řešení.

Problém nastane ve chvíli, kdy učitelé potřebují vyučovat hodiny v konkrétních třídách a mohou učit pouze v omezeném časovém úseku. Potom nelze předměty rozmístit libovolně jako v předchozím příkladu. Nemůžeme použít třídu, ve které je vyučován konkrétní předmět k výuce jiných předmětů. Řešení je stále poměrně jednoduché. Problém nastane ve chvíli, kdy vzroste počet vyučovaných hodin, učitelů, učeben a vzniknou zvláštní podmínky, za kterých mohou být předměty vyučovány.

1.3 Třídy složitosti NP a P

Každé řešení úlohy je možné rozdělit na několik fází. Zaměřme se na fáze nalezení řešení a ověření řešení. Nalezení řešení i ověření řešení jsou vlastně algoritmy. Tyto algoritmy jsou posloupnosti instrukcí, které řešitel provede, aby dosáhl výsledku. Přitom doba k provedení algoritmu se může lišit. Z toho důvody byly zavedeny třídy složitosti. (Chong, 2019)

Třída složitosti P obsahuje algoritmy, které je možné vyřešit v polynomiálním čase deterministickým počítačem. Kde polynomiální čas je takový čas, jehož výpočet obsahuje výraz s polynomem (Chong, 2019) a deterministický počítač představuje počítač, který v daný okamžik může vykonávat maximálně jednu akci. Příkladem P problému jsou operace sčítání, násobení, případně nalezení největšího čísla v řadě.

Třída složitosti NP je potom algoritmem, který je možné vyřešit v polynomiálním čase nedeterministickým počítačem. Nedeterministickým počítačem rozumíme takový počítač, který může vykonávat více akcí v daný moment (Chong, 2019). Je tedy možné NP problém převést na P problém, pokud máme k dispozici nekonečné množství počítačů, které budou problém počítat paralelně. Příkladem NP problému jsou problémy zvané TSP a VRP.

1.4 TSP (Traveling salesman problem)

Práce „Solving Travelling salesman problem using Harmony search algorithm and other metaheuristics“ (Míča, 2016) pojednává o problému obchodního cestujícího. Tento problém se anglicky nazývá Travelling salesman problem a dále je na něj odkazováno zkratkou TSP. Jedná o úlohu, kde je hledána nejkratší vzdálenost mezi několika body na mapě. Vstupem této úlohy je množina měst a očekávaným výstupem je posloupnost měst, která představuje pořadí, ve kterém budou města navštívena.

Zásadním problémem úloh typu TSP je doba k nalezení řešení. Počet vstupních měst může být v řádu jednotek až v řádu stovek či tisíců. Doba k nalezení řešení vygenerováním všech možných kombinací a jejich vyhodnocení následně dosahuje až faktoriálu počtu vstupů.

Těm nejmodernějším počítačům v dnešní době by trvalo nalezení řešení tímto způsobem velmi dlouho, potenciálně až několik let. Jedná se tedy o problém o NP složitosti. Příklad řešení takového algoritmu je uveden v příručce Google OR-Tools (Traveling Salesman Problem, 2020). Řešení spočívá ve využití matice vzdáleností pro vstup celkem třinácti měst. Výstupem je nejkratší vzdálenost mezi těmito městy. Reálnějším příkladem je nalezení nejkratší vzdálenosti pro vrták, který vrtá díry do desky plošných spojů ze stejné příručky.

1.5 VRP (Vehicle routing problem)

Podobným problémem TSP je Vehicle routing problem a dále je na něj odkazováno zkratkou VRP (Vehicle Routing Problem, 2020). Pokud by neexistovaly omezující podmínky, tak by bylo možné problém zjednodušit na TSP, protože by jeden řidič rozvezl všechny předměty. Optimalizace v tomto případě znamená eliminovat ty nejdelší cesty z tras řidičů.

Příkladem může být logistické depo, které potřebuje rozvést určitý počet předmětů do měst, které jsou nahodile rozmístěny na mapě, a depo by bylo uprostřed. Nejdelší cesty by potom byly ty, které by vedly ze severu na jih přes depo. Vhodnější by bylo poslat jednoho řidiče do severních měst a jednoho do jižních měst. Odpadla by potom dlouhá cesta ze severu na jih.

1.6 Řešení VRP pomocí GA

Hlavním úskalím pro řešení TSP je nalezení trasy s nejkratší vzdálenosti spojující několik měst. Na rozdíl od TSP bude v případě VRP těchto tras více. Vybraným prostředkem pro řešení VRP je v této práci genetický algoritmus (GA).

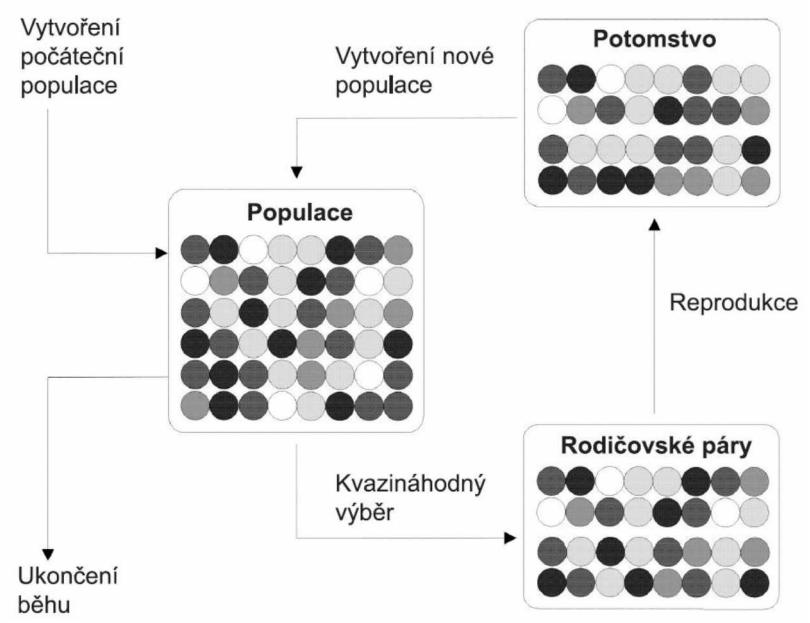
Řešení VRP pomocí GA bude popsáno v následujících kapitolách včetně finálních výsledků. Jako testovací vzorek poslouží souřadnice reálných měst a jejich vzorek zároveň bude simulovat reálný požadavek dopravní společnosti, která se zabývá rozvážkou zboží.

Během tvorby řešení nebudou zohledněny nereálné situace, nebo výjimečné stavy, které běžně nenastávají, nebo k nim nemůže dojít. Například požadavek zákazníka na přepravu objemu stovek až tisíců palet se zbožím, nebo přeprava nepaletových zásilek. Stěžejní pro práci bude vyzkoušet aplikaci GA na VRP.

2 GENETICKÉ ALGORITMY

2.1 Genetický algoritmus

Genetický algoritmus vychází z myšlenky Charlese Darwina (Hynek, 2008), že populace živočichů a rostlin se vyvíjela po mnoho generací. Každá populace je tvořena jedinci, kde každý jedinec má své charakteristické vlastnosti. Tito jedinci soupeří navzájem o prostředky a k rozmnožovacímu procesu se dostanou pouze ti nejsilnější. Tito nejsilnější jedinci se stávají rodiči a vytvoří novou generaci, která ponese jejich vlastnosti. Pro tuto novou generaci potomků se proces opakuje a do každé nové generace se dostávají vlastnosti těch nejsilnějších. Proces tvorby nových generací je zobrazen na Obr. 2.1.



Obr. 2.1 – Tvorba nové populace (Hynek, 2008, s. 14)

2.2 Reprezentace jedince

Každý jedinec v populaci je reprezentován jeho specifickými vlastnostmi (Hynek, 2008). V případě TSP, nebo VRP by vlastnosti jedinců byly sekvence měst, které řidič musí projet. Příklad jedinců je zobrazen v následující části kódu:

Kód v C#

```
{
        "Název": "Rodič_1",
        "Sekvence":
        [ "Pardubice", "Hradec Králové", "Přelouč", "Kolín", "Chrudim" ]
}

{
        "Název": "Rodič_2",
        "Sekvence":
        [ "Hradec Králové", "Přelouč", "Kolín", "Pardubice", "Chrudim" ]
}
```

Tito dva jedinci se následně dostali do rozmnožovacího procesu a stali se rodiči. Nyní v další generaci byl vytvořen další jedinec tedy potomek. Potomek nese vlastnosti jeho rodičů. Zde může být vidět potomek, jehož vlastností je kombinace vlastností jeho rodičů. Potomek dvou rodičů je popsán následujícím kódem:

Kód v C#

```
{
        "Název": "Potomek",
        "Sekvence":
        [ "Pardubice", "Hradec Králové", "Kolín", "Pardubice", "Chrudim" ]
}
```

Výčet vlastností každého jedince se nazývá chromozom (Hynek, 2008). Přitom každá jednotlivá vlastnost je genem. Pro přehlednost jednotlivé geny budou označeny číslicemi, které budou následně použity dále. Zaveďme reprezentaci podle následujícího kódu:

Kód v C#

"Hradec Králové":	1	„Rodič_1":	1
"Přelouč":	2	"Rodič_2":	2
"Chrudim":	3	"Potomek":	3
"Kolín":	4		
"Pardubice":	5		

Původní jedinci budou tedy vypadat následovně:

Kód v C#

```
{
        "Název": "1",
        "Sekvence":
        [ 5, 1, 2, 4, 3 ]
}

{
        "Název": "2",
        "Sekvence":
        [ 1, 2, 4, 5, 3]
}

{
        "Název": "3"
        "Sekvence":
        [ 5, 1, 4, 5, 3 ]
}
```

Všechny tři jedince, kteří aktuálně tvoří populaci, je možné zapsat tímto zkráceným způsobem:

Kód v C#

Jedinec	Chromozom
1	{5, 1, 2, 4, 3}
2	{1, 2, 4, 5, 3}
3	{5, 1, 4, 5, 3}

2.3 Fitness jedince

Další otázkou je, jak vybrat ty nejsilnější jedince. Z tohoto důvodu je nutné jedince ohodnotit. Každému genu v chromozomu jedince bude přiděleno „skóre“, které bude určovat jak silný jedinec ve skutečnosti je (Hynek, 2008).

V následujícím kódu je vidět, že jedinec 3 dosáhl nejvyššího ohodnocení a jedinec 6 dosáhl nejnižšího ohodnocení. Pokud by se rozmnožovacího procesu měli účastnit pouze ti nejsilnější jedinci, tak by se jednalo o jedince 3 a 2, kteří by se podíleli na tvorbě nové generace.

Kód v C#

Jedinec	Chromozom	Fitness
1	{5, 1, 2, 4, 3}	20
2	{1, 2, 4, 5, 3}	25
3	{5, 1, 4, 5, 3}	30
4	{1, 3, 2, 1, 1}	20
5	{5, 4, 3, 2, 1}	15
6	{2, 2, 2, 2, 2}	-5
7	{1, 3, 2, 5, 4}	20

Ohodnocení jedince může být počítáno mnoha způsoby. V příkladu byla použita jako fitness vzdálenost mezi městy a navíc počet duplicit. Z toho důvodu jedinec 6 dopadl jako nejhorší. Otázkou také je, jak velký význam na fitness budou mít jednotlivé geny v chromozomu. Pokud budou mít duplicity příliš vysokou váhu, tak se může stát, že jedinec s duplicitou, který má jinak dobré řešení, bude naprosto vyřazen.

2.4 Selekce

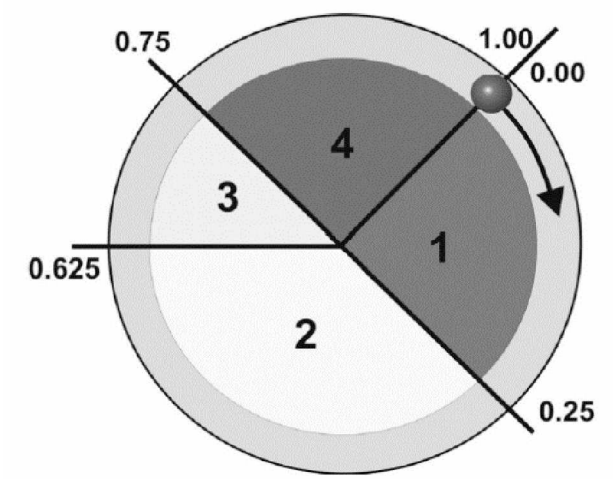
Selekce je způsob, kterým jsou vybírání jedinci do další generace (Hynek, 2008). Většinou se používá fitness funkce, podle které jsou označeni ti nejvhodnější. Vzhledem k tomu, že je algoritmus do určité míry řízen náhodně, tak nelze vyloučit, že s novou generací nebudou vytvořeni slabší jedinci.

Tvorba nové generace ze slabších jedinců není ve skutečnosti negativním jevem. Silný jedinec může obsahovat hodně pozitivních genů, díky kterým má vysoké fitness skóre, ale nemusí obsahovat pozitivní gen slabšího jedince. Pokud by byl slabý jedinec vyřazen z procesu tvorby nové generace, tak by se tato vlastnost ztratila.

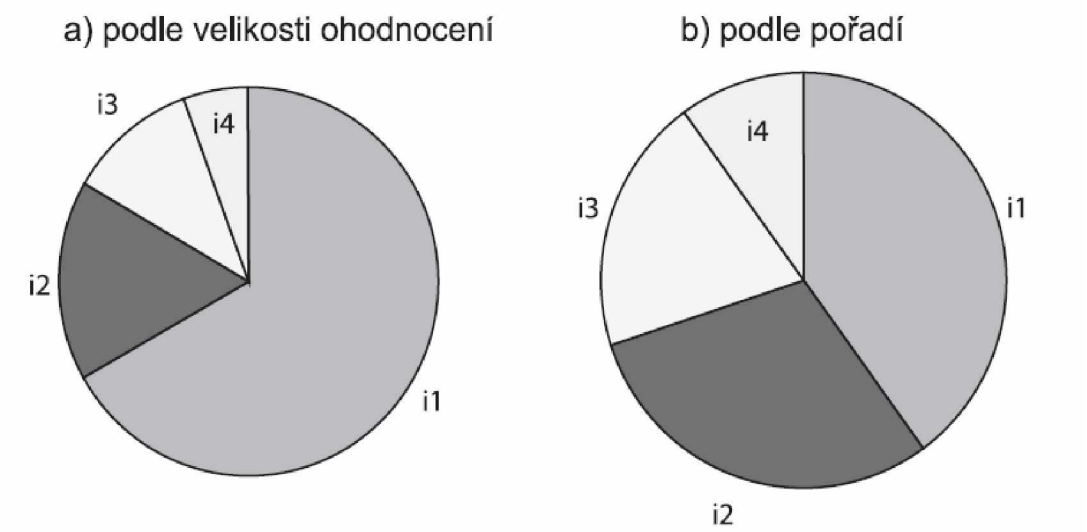
Ve skutečnosti by to znamenalo, že by silný jedinec uvízl v lokálním extrému. Proto existují způsoby, které dávají šanci i těm slabším jedincům. Různé způsoby selekce mají různé výhody a nevýhody.

Existuje například selekce pomocí rulety, jako je na Obr. 2.2. Zde je fitness skóre přímo úměrné ploše na ruletě. V tomto případě je velmi vysoká pravděpodobnost, že jedinec s vysokým fitness skóre (2) se neztratí a jedinec s nízkým fitness skóre (3) má také určitou šanci na úspěch. Nevýhodou je, že příliš silný jedinec (2) může zabrat dominantní postavení a žádný jiný jedinec nebude připuštěn k rozmnožovacímu procesu.

Ruletu je možné pozměnit, aby plocha nebyla přímo úměrná velikosti fitness skóre, ale pořadí. Potom bude ruleta vypadat jako na Obr. 2.3. Zde má slabší jedinec vyšší šanci na podílení se na tvorbě nové generace. Zároveň jsou silnější jedinci stále zvýhodněni díky jejich dobrým genům.



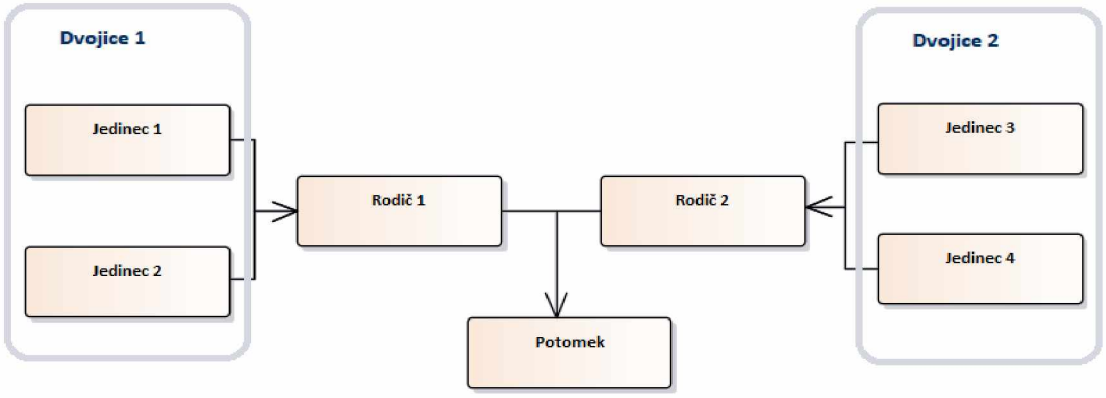
Obr. 2.2 – Selekce ruletou
(Hynek, 2008, s. 23)



Obr. 2.3 – Varianty selekce podle rulety (Hynek, 2008, s. 47)

Dalším způsobem selekce je použití turnaje. Selekce turnajem spočívá v náhodném vybráním 2 dvojic jedinců a z těchto dvojic jsou podle fitness skóre vybráni 2 vítězové. Vítězi se následně budou podílet na tvorbě nové generace. Pokud tedy vybereme dvojici 2 slabších a 2 silnějších jedinců, tak rodiči se stane 1 silný a 1 slabý jedinec. Nevýhodou je, že některý silný jedinec nemusí být vybrán do turnaje právě kvůli náhodě.

Příklad turnaje je zobrazen na Obr. 2.4.



Obr. 2.4 – Selekce pomocí turnaje

2.5 Genetické operátory

Genetické operátory jsou funkcemi, které lze použít k práci s jedinci (Hynek, 2008). Návrh těchto operací je velice důležitý. V případě této práce se jedná o křížení dvou jedinců a o mutace jedince.

Nicméně někdy není možné vhodně navrhnout genetický operátor, protože samotný jedinec je nerozumně, nebo nedostatečně reprezentován. Mějme tuto část kódu:

Kód v C#

Jedinec	Chromozom
1	{5, 1, 2, 4, 3, "Generace_1", "2020-04-16", „Highscore“}
2	{1, 2, 4, 5, 3, "Generace_1", "2020-04-16", „-„}
3	{5, 1, 4, 5, 3, "Generace_2", "2020-04-17", „-„}

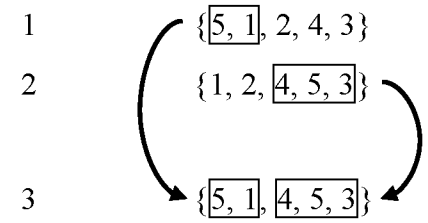
Zde jsou kříženi právě dva jedinci. V genech jsou navíc informace, které neslouží k výpočtu fitness a jsou pouze informativního charakteru, nebo jsou použity během vývoje. Takové informace navíc mohou podstatně zpomalit chod algoritmu a musí na ně být brán ohled během křížení.

2.6 Křížení

Křížení je operace, která zkombinuje 2 jedince a z jejich genů vytvoří nového jedince (Hynek, 2008). Tento nový jedinec bude kombinací jeho 2 rodičů z předešlé generace. Křížení je možné realizovat mnoha způsoby. Toto je hlavní způsob, kterým jsou tvořeni noví jedinci v této práci. Po křížení se může stát potomek silnějším i slabším, než jsou jeho rodiče.

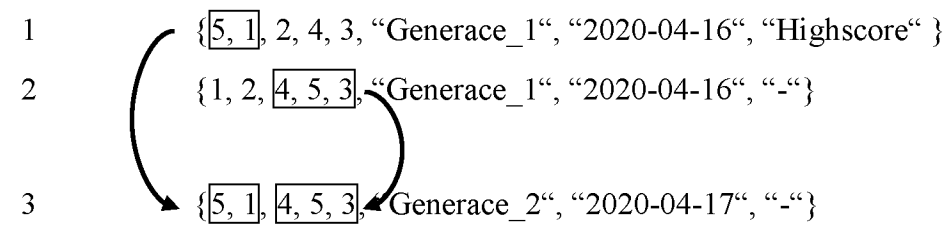
Může být ke křížení použita vždy konkrétní část jedinců. Například první polovina z prvního jedince a druhá polovina z druhého jedince. Tyto poloviny mohou být vybírány náhodně. Je možné každý gen vybírat náhodným způsobem z jedinců. V následujícím kódu je zobrazeno křížení dvou jedinců, kde potomek dědí část vlastností z každého rodiče. Zmíněný kód:

Kód v C#



Křížení je možné omezit pouze na některé geny uvnitř chromozomu. Kdyby dodatečné geny, které skutečně neovlivňují fitness, bylo potřeba držet v chromozomu, tak je možné je z křížení vynechat jako zde:

Kód v C#



2.7 Mutace

Mutace je operace, kdy se jeden nebo více genů náhodně změní (Hynek, 2008). Touto náhodnou změnou může vzniknout gen, který dříve neexistoval. Mutace je způsob, kterým může být algoritmus chráněn od lokálního extrému. Mějme reprezentaci jedinců jako je zde:

Kód v C#

Jedinec	Chromozom
8	{5, 1, 1, 2, 3}
9	{5, 1, 1, 2, 3}
10	{5, 1, 1, 2, 3}
9	{5, 1, 4, 2, 3}

Jedinci mají své sekvence měst a vypadá to, že algoritmus dosáhl extrému a dál již nepokračuje. Problém je ten, že žádný z jedinců neobsahuje hodnotu 4 a křížením už tuto hodnotu není možné získat. Z tohoto extrému je možné uniknout mutací, a to právě tak, že se po přechodu do další generace náhodně změnil jeden z genů v chromozomu.

Další výhodou mutace je, že se může objevit gen, nebo chromozom, který může vést k mnohem silnějším a vhodnějším jedincům. Například v následující situaci vznikl gen s duplicitní hodnotou, ale později se ukázalo, že vzdálenost mezi městy 3 a 1 je mnohem menší, než jiné kombinace. Situace v odstavci je zobrazeno následujícím kódem:

Kód v C#

Jedinec	Chromozom
8	{5, 1, 1, 2, 3}
9	{5, 1, 1, 2, 3}
10	{5, 1, 1, 2, 3}
10	{3, 1, 1, 2, 3}

Zároveň je důležité, aby nebyl výskyt mutace příliš vysoký. Algoritmus by následně mohl být příliš náhodný a nikdy by nemusel dojít do extrému.

2.8 Příklad skládání puzzle

Uvažujme, že máme puzzle, které má celkem 1000 dílků. To znamená, že máme 1000 možných pozic, na který je možné každý dílek umístit. Podle vzorce pro variace (Variace, 2006) s opakováním je to celkem 1000^{1000} možných kombinací a pouze jedna z nich je správná.

První možností je pokusit se řešit takové puzzle „hrubou silou". V tom případě bychom museli všechny tyto možnosti vyzkoušet. Tohle by mohlo trvat potenciálně několik desítek let. Druhou možností je náhodně rozložit dílky po ploše a pokoušet se dílky spojovat. Na začátku by byl náhodně vybrán dílek a k tomu by byl hledán dílek, který zapadne. Tento proces by se opakoval, dokud by nebylo puzzle hotové.

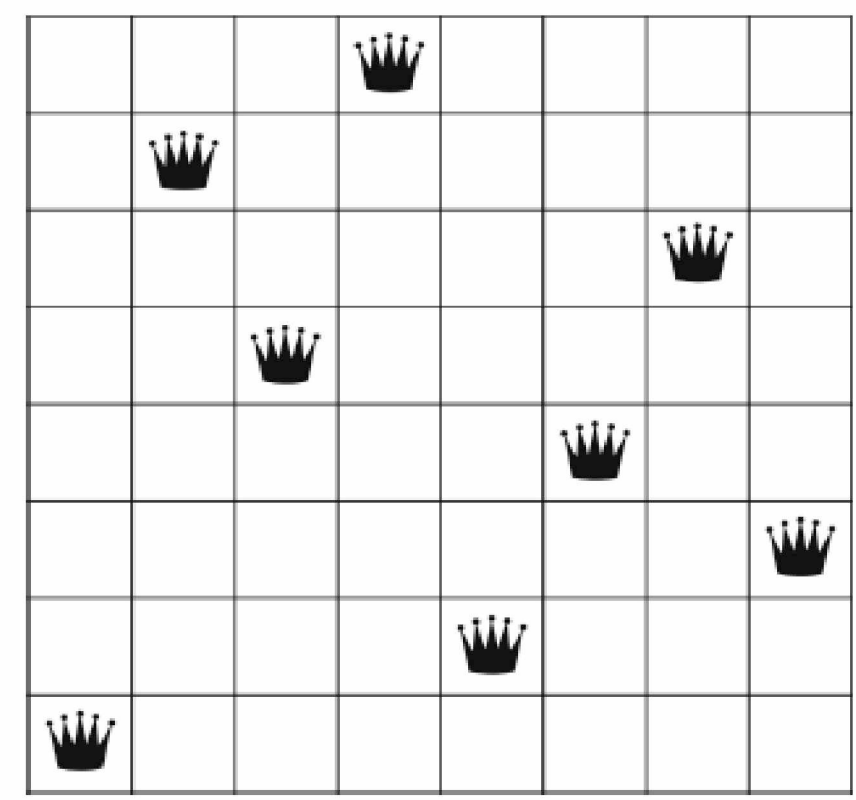
Je možné si představit řešení „hrubou silou", jako poskládání všech 1000 dílků na plochu a vyhodnocení, jestli je puzzle správně poskládáno. Pokud by nebylo správně poskládáno, tak všechno bude přeskládáno znovu a proces by se opakoval, dokud není nalezeno správné řešení. Genetický algoritmus je jako nalezení částečně správného řešení, dílků, které k sobě patří a k těmto dílkům by byly hledány souhlasné dílky. Toto částečné správné řešení je následně použito k nalezení lepšího řešení, dokud by nebylo dosaženo správného řešení.

2.9 Problém N dam

Dalším příkladem je problém N dam (Hynek, 2008). Mějme čtvercovou šachovnici, která má konečný počet polí. Na tuto šachovnici chceme umístit figurky dámy, které nesmí být navzájem ve své dráze. Na Obr. 2.5 je příklad s šachovnicí 8x8, kde musí být rozmístěno 8 dam. Každá dáma se může pohybovat vertikálně, horizontálně, nebo po diagonále. Dámy nesmí být navzájem ve svých dráhách.

V případě 3x3 by bylo podle vzorci o variacích nutné projít 504 kombinací a tato hodnota by velice rychle rostla. Pro případ 10x10 by existovalo 62815650955529470000 kombinací.

Je jisté, že případ by nebylo možné řešit „hrubou silou", protože by nalezení řešení trvalo opět několik let. Jedná se o typický příklad, který je možný řešit genetickým algoritmem. Podrobně popsáno v publikaci (Hynek, 2008).



Obr. 2.5 – Příklad N dam (Hynek, 2008, s. 92)

3 NÁVRH ALGORITMU

Tato kapitola se věnuje tomu, jak byl problém rozebrán a jak bylo navrženo řešení. Řešení je v tomto případě několik a ne všechny se ukázaly jako vhodné. Mezi hlavní parametry, kterými byl algoritmus hodnocen, jsou výsledná posloupnost měst a realizovatelnost algoritmu po programové stránce.

3.1 Úloha, problematika

Výhodou genetického algoritmu je jeho obecnost. Tento algoritmus je možné použít pro širokou škálu úloh. Tyto úlohy mohou být hledání nejkratší vzdálenosti, rozmístění pracovníků ke strojům, naplánování rozvrhu a mnoho dalších podobných úloh.

V tomto typu úloh může existovat mnoho řešení. Například pokud je nutné vytvořit školní rozvrh, tak vždy se potýkáme s omezujícími podmínkami, které ovlivňují, kdy je možné předmět učit. Mezi tyto podmínky může patřit počet učeben, počet vyučovaných předmětů, počet vyučujících, speciální časy, ve které je možné předměty učit, možnost učit více tříd současně, nutnost učit pouze určité dny a mnoho dalšího.

Jednou z možností, jak naplánovat takový rozvrh je zkoušet předměty nahodile poskládat a následně dobré kombinace jenom ladit. Tímto způsobem je možné dosáhnout rychle výsledku. Ladění je ve skutečnosti k částečně správnému řešení zkoušet přidávat chybějící předměty.

Další možností je možné zkoušet poskládat rozvrh hrubou silou a zkusit všechny možnosti, které jsou. Tohle ve skutečnosti funguje, ale se zvyšujícím se počtem omezujících podmínek, vstupů a výstupu se počet kombinací exponenciálně zvyšuje. Tento jev je zvaný kombinatorická exploze a je to zároveň důvod, proč není vhodné tímto způsobem hledat řešení. Velice výkonný počítač by potřeboval několik let pro nalezení řešení tímto způsobem.

Genetický algoritmus umožňuje využít toho, že součástí každého řešení je část vhodného řešení. Je možné vygenerovat skupinu řešení náhodně, ohodnotit je pomocí podmínek a z těch nejlepších poskládat řešení, které jsou jejich kombinacemi. Po několika desítkách kombinací je možné tímto způsobem získat vhodné řešení, které by hrubou silou trvalo několik let.

3.2 Rozbor zadání

Úkolem práce je navrhnout algoritmus, který přebere vstup, reprezentuje požadavek zákazníka a jako výstup poskytne posloupnost měst. Tato posloupnost měst představuje cestu, kterou řidič musí projít.

Cesta má reálnou vzdálenost a vhodným řešením se rozumí takové, které se přibližuje k nejkratší vzdálenosti. V případě nalezení vhodné trasy pro řidiče je vhodným řešením takové řešení, kde se řidič zbytečně nevrací, a města jsou seskupena do skupin.

Vstupy budou omezeny na případy, kdy jeden řidič není schopen všechno zboží na jednu cestu naložit, a proto musí být zboží rozděleno mezi několik řidičů. Díky tomu existuje nekonečně mnoho možností, jak zboží rozdělit.

3.3 Návrh řešení

Řešením by tedy měl být program, který je schopen přečíst seznam předmětů, které je nutné rozvést, tento seznam analyzovat a vytvořit vhodnou posloupnost měst, kterou vrátí jako řešení. Poskytnuté řešení by mělo vyhovět požadavkům zadaným zákazníkem. Dále by mělo poskytnuté řešení umožnit uživateli více než jednu možnost, případně předložené řešení modifikovat.

Tyto požadavky by měly být strukturovány do formátu, který je možné v elektronické formě uložit do souboru. Aplikace na základě tohoto programu stanoví požadavky uživatele. Tyto soubory by měly být navrženy takovým způsobem, že je možné je vytvořit, nebo naplnit daty běžným uživatelem, nebo analytikem.

Uživateli bude umožněno upravit dvě skupiny parametrů a vstupní data. První skupinou jsou parametry, které přímo ovlivňují výsledek.

Do první skupiny patří tyto parametry:

- Cena za kilometr.
- Cena za dopravní prostředek.

Druhá skupina představuje parametry, které ovlivňují genetický algoritmus.

- Počet jedinců v generacích.
- Počet generací.
- Ukončující podmínka.

Vstupy aplikace jsou tvořeny dvěma soubory. Tyto soubory nesou informace o:

- Seznam požadavků, kam mají být předměty dopraveny.
- Seznam souřadnic měst, kde se města nachází na mapě.

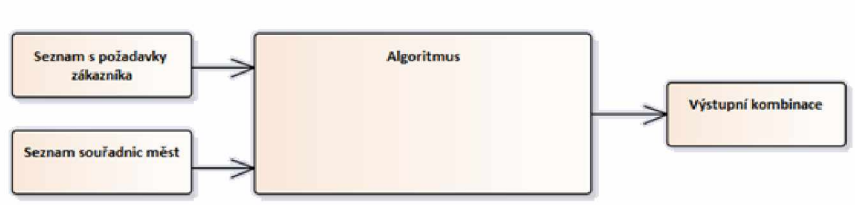
Aplikace na základě požadavků vygeneruje výslednou kombinaci. Uživatel musí mít k dispozici ovládací prvky, které mu umožní vybrat z výsledných řešení. V případě, že žádné řešení neexistuje, tak bude vrácena taková kombinace, která je nejblíže k vhodnému řešení. Uživatel by potom mohl dořešit problém nestandardním řešením.

Výstupem aplikace bude textový soubor. Tento soubor je potom dále možné použít v jiných aplikacích, které slouží k zadání samotného rozvrhu. Případně může být výstup ve formátu jpg, bmp, nebo pdf, které je možné vytisknou. Tato možnost by mohla být požadována uživateli, kteří nepoužívají žádné výpočetní programy.

Aplikace bude realizována v programovacím jazyce C#.

3.4 Blokové schéma

Vstupem aplikace budou textové soubory s požadavky. Prvním vstupním souborem je seznam předmětů, které mají být rozvezeny. Druhým vstupním souborem je seznam souřadnic cílových měst. Výstupním souborem je výstupní kombinace.



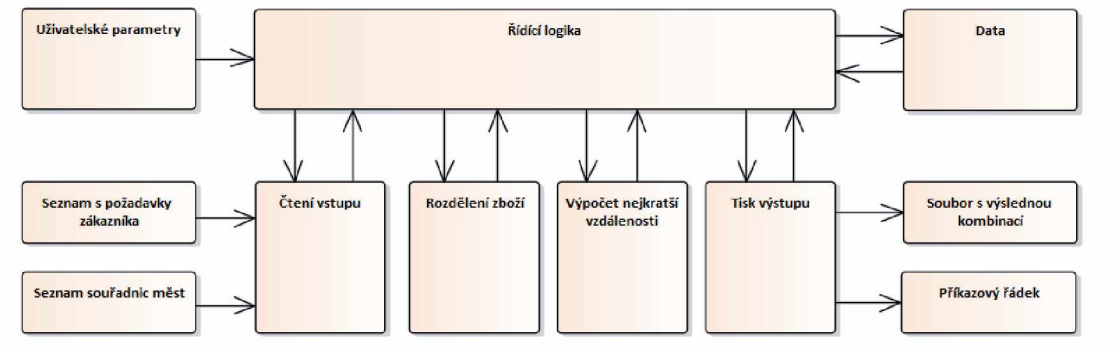
Obr. 3.1 – Blokové schéma algoritmu

Samotný algoritmus může být rozdělen na několik bloků, které vykonávají konkrétní funkce. Bude potřeba číst vstup s požadavky zákazníka, rozdělit zboží na několik celků, vypočítat nejkratší vzdálenost a vytisknout posloupnost jako výstup.

Blok čtení vstupu bude sloužit pouze ke čtení vstupních souborů. Tento blok bude zároveň obsahovat funkce pro práci a dekódování vstupních souborů. Dále blok výstupu bude obsahovat funkce pro zobrazení dat uživateli. Příkazový řádek bude velice důležitý ve chvíli, kdy bude algoritmus vyvíjen a tisk do souboru bude dobrý pro analýzu výsledku.

Blok rozdělení vstupu bude sloužit pro rozdělení zboží na menší celky, které budou následně přiděleny řidičům. Zde budou všechny podmínky, které budou hlídat, že není překročena kapacita řidiče, a že jsou naloženy všechny zakázky.

Výpočet nejkratší vzdálenosti bude řešen pomocí genetického algoritmu. Existují jiné a specializovanější postupy, které je možné použít, ale pro účely této práce bude použit genetický algoritmus. Rychlost tohoto výpočtu bude velice důležitá, protože během chodu algoritmu vznikne mnoho kombinací, které musí být vyhodnoceny.



Obr. 3.2 – Blokové schéma logiky algoritmu

3.5 Komponenty algoritmu

3.5.1 Vnitřní algoritmus

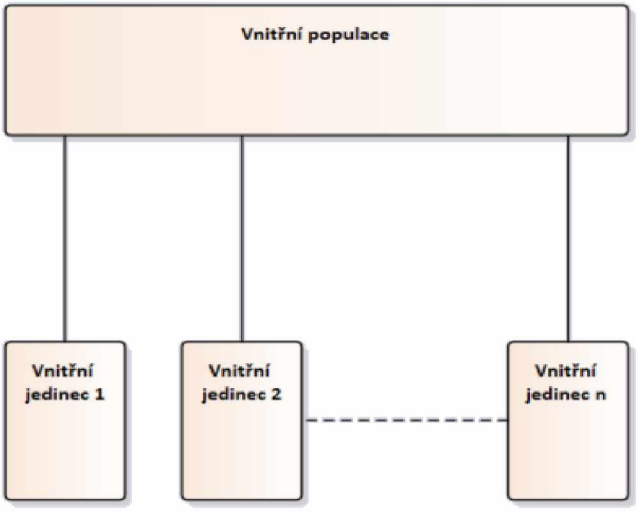
Výstupem vnitřního algoritmu je pouze jeden nejsilnější vnitřní jedinec, který dále tvoří vnějšího jedince. Tento výstupní jedinec představuje posloupnost měst, která je nejkratší nalezenou trasu pro danou množinu měst. Výstupní jedinec je zároveň nejsilnějším jedincem z vnitřní populace. Vnitřní populace je přitom tvořena množinou vnitřních jedinců. Počet těchto jedinců tvořící vnitřní populaci je dán parametrem a všichni jedinci ve vnitřní populaci jsou po několik generací zlepšování, dokud není nalezen nejsilnější.

3.5.2 Vnitřní jedinec

Vnitřní jedinec představuje jednu posloupnost měst. Tuto posloupnost obslouží pouze jeden řidič a pořadí měst v posloupnosti představuje pořadí, ve kterých budou navštívena. Po spojení všech měst uvnitř posloupnosti vnitřního jedince vznikne trasa, která má reálnou vzdálenost. Tato vzdálenost představuje fitness vnitřního jedince.

3.5.3 Vnitřní populace

Vnitřní populace obsahuje konečný počet vnitřních jedinců. Tito vnitřní jedinci jsou podrobeni vnitřnímu genetickému algoritmu a vnitřní jedinec s nejnižší fitness funkcí (nejkratší vzdáleností) je označen za vítěze. Vnitřní populace je zobrazena na Obr. 3.3.



Obr. 3.3 – Vnitřní populace

3.5.4 Vnější algoritmus

Vnější populace obsahuje určitý počet vnějších jedinců a vnější jedinci obsahují konečný počet vnitřních jedinců. Každý vnitřní jedinec je výstupem vnitřního genetického algoritmu. Výstupem vnějšího genetického algoritmu je jeden nejsilnější vnější jedinec.

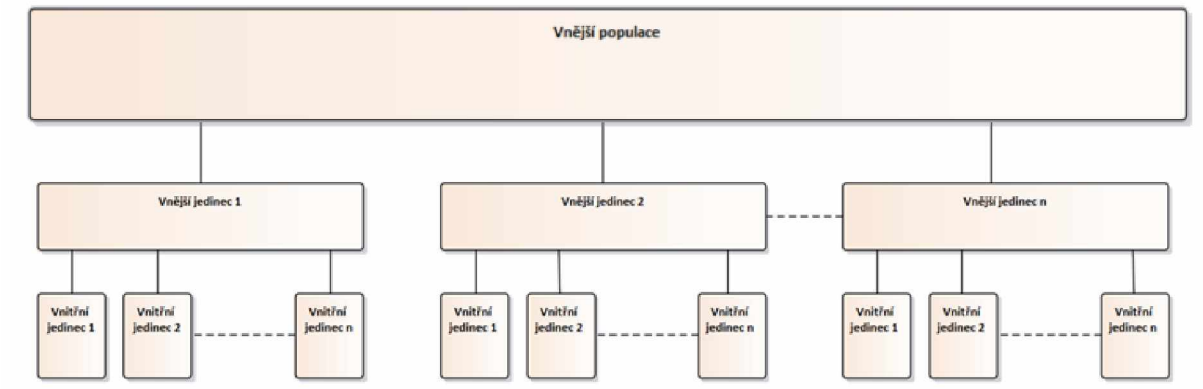
Jinými slovy lze říci, že vnější jedinec je skupina posloupností měst, kde každá posloupnost je jedním vnitřním jedincem. Tito vnější jedinci tvoří vnější populaci a jsou po generacích zlepšování vnějším genetickým algoritmem.

3.5.5 Vnější jedinec

Vnější jedinec je jedna skupina vnitřních jedinců. Vnitřní jedinci přitom jsou posloupnosti měst, kde každý vnitřní jedinec je nejsilnějším jedincem, který prošel vnitřní genetickým algoritmem jako vítěz. Každý vnitřní jedinec má svou fitness a součet fitness všech vnitřních jedinců je fitness vnějšího jedince.

3.5.6 Vnější populace

Vnější populace obsahuje konečný počet vnějších jedinců. Tito vnější jedinci navzájem soupeří a ten nejvhodnější z nich je označen jako vítěz a zároveň výstup algoritmu. Vnější populace je zobrazena na Obr. 3.4.



Obr. 3.4 – Vnější populace

3.6 Pseudokód algoritmu

Mějme tyto parametry:

- Počet generací vnitřního genetického algoritmu je 50.
- Počet generací vnějšího genetického algoritmu je 20.
- Ukončující podmínka není stanovena.
- Celková kapacita předmětů, které mají být rozvezeny je vyšší než 32.

Algoritmus je potom možné popsat následujícím pseudokódem:

```
Načíst vstupy;
Načíst souřadnice;

Roztřídit předměty a vygenerovat populace;

for (1 ... 20) // Vylepšení vnější populace 20x
{

        foreach (vnější jedinec ve vnější populaci)
        {

        Vylepšit vnějšího jedince;

        {

                // Vnější jedinec = skupina vnitřních jedinců
                Vytvoření vnitřní populace z vnějšího jedince;
                foreach (vnitřní jedinec ve vnitřní populaci)
                {
                        for (1... 50)   // Vylepšení vnitřní populace 50x
                        { Vylepšit vnitřního jedince }
                }

        }

Aktualizace nejlepšího vnějšího jedince;

}
Vrácení nejlepšího vnějšího jedince;
```

3.7 Možné výstupy algoritmu

Otázkou, která bude řešena, je, jestli je lepší hledat delší dobu tu nejkratší vzdálenost, nebo jestli je vhodnější hledat skupinu měst, které jsou více pohromadě. Je tedy nutné zjistit, jak dlouho má smysl optimální řešení hledat a jak velký dopad má každý scénář. Veškerá měření budou zaměřena na tyto hlavní body:

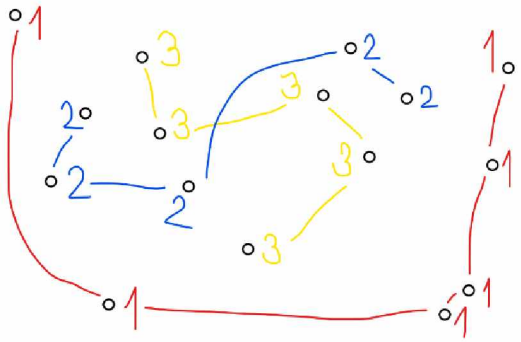
- Vliv počtu generací na genetický algoritmus.
- Vliv ceny za kilometr a řidiče na genetický algoritmus.
- Vliv rozmístění měst na genetický algoritmus.

3.7.1 Města jsou nevhodně rozdělena na úrovni vnějšího jedince

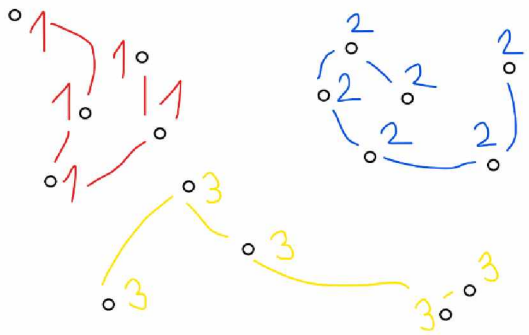
Pokud by nebyla města pohromadě, tak by nejkratší vzdálenosti mohly vypadat takto. V obou případech je nejkratší vzdálenost nalezena, ale v případě na Obr. 3.5 je tato vzdálenost mnohem vyšší než vzdálenost na Obr. 3.6.

Situace na Obr. 3.5 zjevně výrazně prodlouží celkovou trasu a nastává ve chvíli, kdy algoritmus uvízne v lokálním extrému na úrovni vnějšího jedince, ale počet generací vnitřního jedince je dostatečný.

Situace na Obr. 3.6 je optimální výstup algoritmu.



Obr. 3.5 – Neoptimální cesta

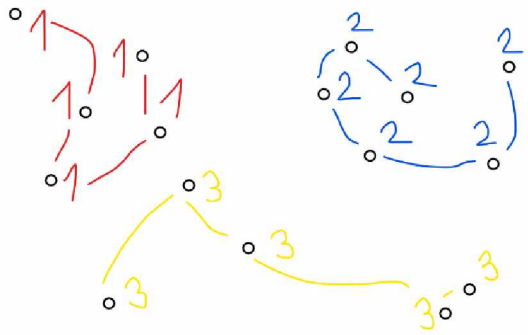


Obr. 3.6 – Optimální cesta

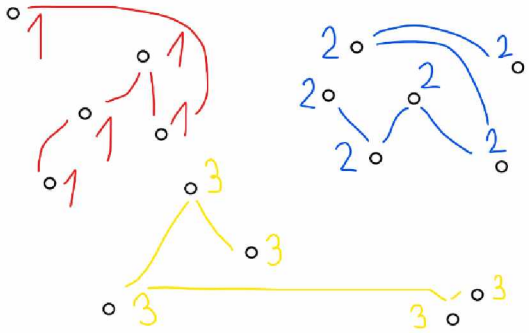
3.7.2 Nebyla dosažena nejkratší vzdálenost

V tomhle případě se může stát, že řidič bude mít vhodně určenou oblast měst, ale algoritmus pro vnitřního jedince neběžel dostatečně dlouho a výsledná cesta není nejkratší. Tohle nemusí nutně vadit, protože se může jednat pouze o minimální nárůst cesty.

Na Obr. 3.7 jsou zvoleny ty nejkratší vzdálenost a na Obr. 3.8 jsou vzdálenosti o něco delší. Oba případy mohou být považovány za optimální řešení.



Obr. 3.7 – Optimální skupina měst s optimální cestou



Obr. 3.8 – Optimální skupina měst s neoptimální cestou

3.8 Odhad optimálního řešení

Logistické společnosti v dnešní době rozdělují zboží, které chtějí rozvézt na menší části. Například Olomoucký kraj je možné rozdělit na Šumperskou, Olomouckou, Bruntálskou, Přerovskou a Ústeckou část. Tyto části jsou blízko u sebe. To by odpovídalo případu, kdy jsou města vhodně rozdělena a jsou blízko u sebe. Nicméně pokud by logistická společnost nerozdělila zboží na části a řidiči by jeli nejkratší trasy, ale do měst, které nejsou poblíž, tak by mohlo dojít k značnému nárůstu konečné vzdálenosti.

V druhém případě by došlo pravděpodobně k mnohem většímu nárůstu konečné vzdálenosti, protože některá města mohou být od sebe desítky až stovky kilometrů. Například by to byl příklad, kdy by řidič jel z Brna do Prahy, pak do Ostravy, pak do Liberce a nakonec do Brna. Zde by došlo ke značnému nárůstu.

V prvním případě, kdyby řidiči obsluhovali pouze Pražský kraj, nebo jen Brněnský kraj, tak nejkratší vzdálenost nehrála tak zásadní roli, protože by necestovali přes celou republiku, ale udělali by si pouze zajížďku v délce několik kilometrů. Odhad je tedy takový, že důležitější vhodně rozdělit města na skupiny, které jsou poblíž a až potom hledat nejkratší vzdálenost.

4 ŘEŠENÍ VLASTNÍ PRÁCE

Tato kapitola se zabývá vlastní realizací práce. Je zde popsáno, jakým způsobem byl samotný algoritmus vyvíjen. Jsou zde zároveň popsány problémy, ke kterým došlo a jejich následné řešení. K realizaci programu byl použit programovací jazyk C# (Microsoft, 1996). Vzhledem k charakteru úlohy nebylo nutné zpracovat pokročilé grafické rozhraní a výstup byl tisknut do příkazové řádky. Vstupem programu jsou txt soubory formátované jako csv soubory s oddělovači.

První vstupní soubor obsahoval souřadnice všech měst. Tento soubor se jmenoval Coordinates.txt a původně byl použit v projektu „souradnice-mest" dostupného zdarma na stránkách webové platformy Github (Zemek, 2018). V souboru je každé město názvem a X a Y souřadnicemi. Druhým vstupním souborem je soubor „ActiveOrders.txt". Tento soubor obsahuje data, která představují požadavky zákazníků. Každý požadavek je označen unikátním identifikátorem ID a obsahuje cílové město a počet požadovaných palet.

4.1 Krok Výpočet nejkratší vzdálenosti

Prvním krokem bylo navrhnout algoritmus pro výpočet nejkratší vzdálenosti. Vstupem tohoto algoritmu je textové pole, které obsahovala města, mezi kterými byla nejkratší vzdálenost počítána. Prvním pokusem bylo počítat vzdálenosti pomocí Dijkstrova algoritmu (Kolář, 2000) a matice vzdáleností (Kunz, 2001).

4.1.1 Matice vzdáleností

Matice vzdáleností byla použita, když byla města ještě v řádu jednotek. Matici bylo jednoduché vytvořit, ale se zvyšujícím se počtem měst se obtížnost značně zvýšila. Tato matice má 2 hlavičky. Jedna hlavička je horizontální, druhá vertikální. Obě hlavičky obsahovaly názvy měst. Buňky, kde se města protínala, byla právě vzdálenost mezi městy. Na diagonále této matice jsou 0 a celá matice je symetrická podle diagonály.

Výhodou je, že matici je jednoduché použít. Nebylo nutné hledat v matici, kde jsou města a bylo možné rovnou číst hledanou vzdálenost. Například pro výpočet vzdálenosti mezi městem 12 a městem 23 stačilo přečíst hodnotu na 12. řádku a 23. sloupci.

Nevýhodou bylo, že pro požadovaný vstup 216 měst byla tato matice příliš velká. Nebylo pro uživatele reálné, aby tuto matici počítal v Excelu, nebo jiných programech. Nicméně není nereálné, aby byla tato matice počítána přímo programem.

4.1.2 Pythagorova věta

Druhou variantou, která byla nakonec použita, je Pythagorova věta (Čučka, 2007). Pythagorova věta je metoda, kterou je možné vypočítat vzdálenost mezi 2 body. K výpočtu stačí znát souřadnice X a Y.

Výhodou tohoto výpočtu je, že není nutné tvořit žádné další matice, nebo uchovávat data o vzdálenostech v extra souborech. Zároveň je možné jednoduše vypočítat vzdálenost mezi 2 body a nejsou prováděny žádné nadbytečné operace.

Nevýhodou je, že je nutné navrhnout a optimalizovat algoritmus, který stahuje souřadnice potřebných měst. Například pokud znám dvě města, mezi kterými počítám vzdálenost, tak musím někde získat souřadnice. Tohle je záležitost optimalizace přístupu do databáze. Bylo vyzkoušeno mnoho variant práce s městy. Nejjednodušší a nejrychlejší se ukázalo pracovat přímo s indexy měst. Tabulka allCitiesTable obsahuje souřadnice všech měst a třída Controller je třída, kde jsou shromažďována taková data. Metoda pro výpočet vzdálenosti mezi městy vypadá následovně:

Kód v C#

```
/// <summary>
/// Vrátí vzdálenost mezi 2 městy. Vstupem jsou indexy měst
/// </summary>
/// <param name="city1"></param>
/// <param name="city2"></param>
/// <returns></returns>
public static double getDistance(int city1, int city2)
{
double x1 = Convert.ToDouble(Controller.allCitiesTable.Rows[city1]["X"].ToString());
double x2 = Convert.ToDouble(Controller.allCitiesTable.Rows[city2]["X"].ToString());
double y1 = Convert.ToDouble(Controller.allCitiesTable.Rows[city1]["Y"].ToString());
double y2 = Convert.ToDouble(Controller.allCitiesTable.Rows[city2]["Y"].ToString());

return Math.Round(Math.Pow(Math.Pow(x2 - x1, 2) + Math.Pow(y2 - y1, 2), 0.5), 4);
}
```

Předchozí varianta byla použita, když v algoritmu byly používány celé názvy měst. Tohle se ukázalo mnohem složitější, protože se na mnoha místech musely názvy překládat a přístup do tabulky allCitiesTable byl mnohem pomalejší.

4.1.3 Reálná vzdálenost oproti vzdušné čáře

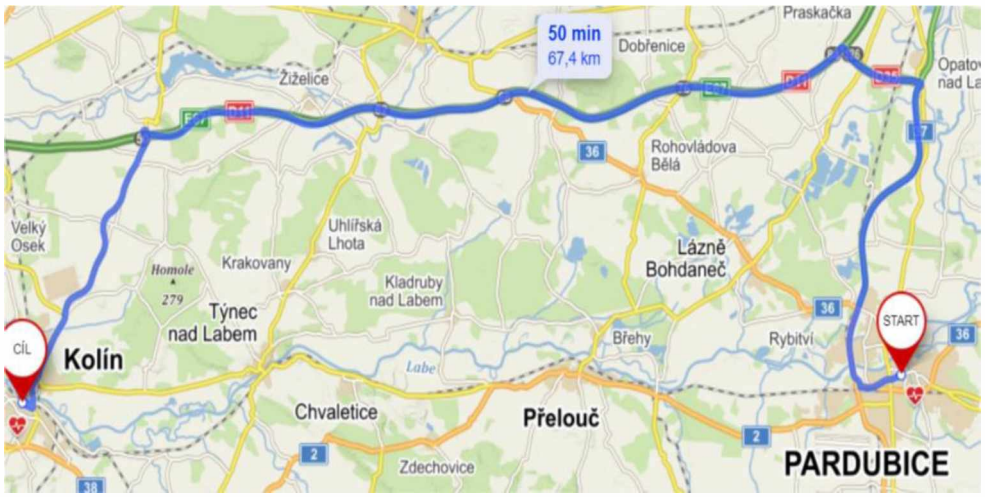
Poslední věcí, kterou je nutné promyslet, je rozdíl mezi skutečnou vzdáleností a vypočtenou vzdáleností. Protože k výpočtu jsou použity X a Y souřadnice mapy, tak dochází ke zkreslení výsledku, protože silnice ve skutečnosti nevedou přímou čarou. Pro potřeby algoritmu je tohle skreslení zanedbatelné, protože rovnoměrně nastává při každém výpočtu a chyba se nenásobí mezi sebou a pro ohodnocení cesty není přesnost důležitá.

Další příčinou zkreslení je samotný zemský povrch. Protože zemský povrch se nachází na vypouklé planetě, tak musíme počítat se vzorci pro neeuklidovskou geometrii. V případě této práce i takové zkreslení je možné pominout, protože výpočty probíhají pouze na poměrně malé ploše.

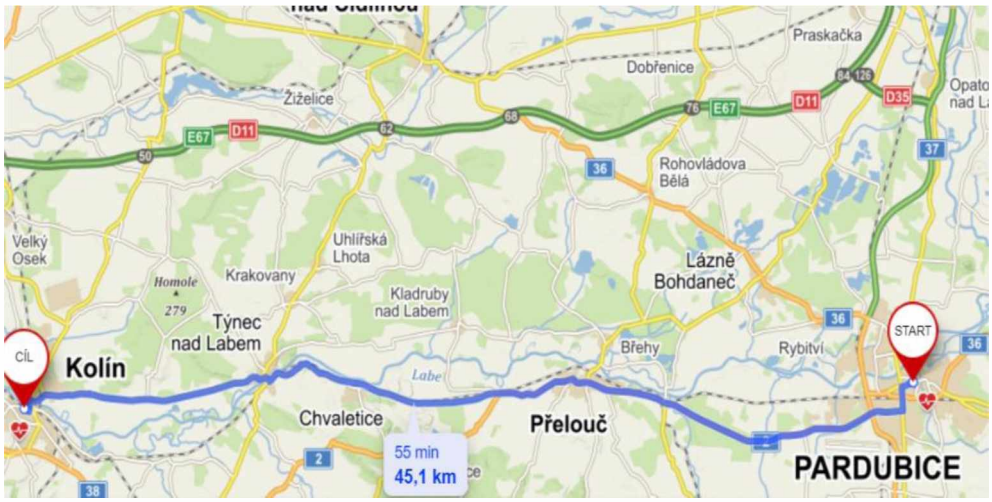
Například cesta z Ostravy do Olomouce je podle výpočtu přibližně 90 km. Nicméně ve skutečnosti je vzdálenost 95,7 km. V jiném případě je vzdálenost z Olomouce do Dubu nad Moravou 13,5 km vzdušnou čarou a 14,1 km skutečná vzdálenost po silnici. Algoritmus by v tomto případě vzdálenost trošičku zkreslil, ale finální výsledek by byl přijatelný.

Problém by byl, kdyby skutečná cesta se nepodobala přímce, ale jednalo by se o křivku jako je půlkružnice, nebo části mnohoúhelníku. Takový případ nastává pouze v případě, že by cesta musela vést přes řeku, nebo jinou překážku, kde by bylo nutné udělat velkou objížďku kvůli jejímu překročení.

Na Obr. 4.1 je ukázka nejrychlejší cesty a na Obr. 4.2 je nejkratší cesta. V drtivé většině případů je nejkratší cesta tou vhodnější, takže z toho důvod nepřesnost kvůli vzdušné čáře není brána v úvahu.



Obr. 4.1 – Nejrychlejší cesta (Mapy.cz, 1996)



Obr. 4.2 – Nejkratší cesta (Mapy.cz, 1996)

4.2 Krok Programování vnitřního jedince

Vnitřní jedinec je v tomto případě posloupnost měst. Tato posloupnost měst představuje pořadí, neboli cestu, kterou řidič bude muset projet. Celková vzdálenost, kterou řidič projede, bude jeho fitness skóre.

4.2.1 Vhodná reprezentace vnitřního jedince

Tato posloupnost měst je reprezentována jako pole datového typu string neboli pole řetězců. Existují další možnosti, jak reprezentovat posloupnost. Příkladem může být pouze jediný řetězec, který by měl uvnitř zakódovanou informaci o městech. Tento způsob byl původně zvoleným řešením, ale ukázalo se, že algoritmus se příliš zpomalí. Zpomalení dosahovalo až pětinásobku původní doby. Třída Algorithm je třída s algoritmem, která obsahuje všechna nastavení, jako je délka algoritmu, počet generací a podobně. Konstanta LENGTH je zde délka algoritmu a zároveň je celkovým počtem měst. Reprezentace vnitřního jedince vypadá v kódu takto:

Kód v C#

```
private string[] sequence = new string[Algorithm.LENGTH];
```

Jedinec sám o sobě je vlastně třídou, která má vlastnost zvanou sekvence. Tato sekvence je datového typu pole string, které reprezentuje posloupnost měst. Dále obsahuje pouze metody, které jsou použity pro výpočty.

Hlavními jsou metody GetFitness, která vypočítá celkovou vzdálenost mezi městy. Metoda GetDuplicity vypočítá počet duplicit v posloupnosti, která nastává po křížení. Další důležitými metodami jsou GetSequence a ShowMe, které jsou použity pro zobrazení posloupnosti.

4.2.2 Reprezentace vnitřní generace

Vnitřní generace je třída, která obsahuje instance třídy vnitřní jedinec. Počet instancí je roven počtu vnitřních jedinců ve vnitřní generaci. Vnitřní generace obsahuje metodu selekce jedinců. Tato metoda náhodně vybere 2 dvojice jedinců a pomocí jejich fitness vybere z nich lepší 2. Tito jedinci jsou následně podrobeni křížení, případně mutaci.

Další metodou je ukončovací podmínka. Tato metoda je vždy zavolána ve chvíli, kdy je populace zlepšena. Metoda zkontroluje, jestli nedošlo k dosažení ukončovací podmínky a pokud došlo, tak vrátí hodnotu True, která je datového typu Boolean. Hlavní část třídy Invidual, která reprezentuje vnitřní generaci vypadá následovně:

Kód v C#

```
private string[] sequence = new string[Algorithm.LENGTH];

/// Konstruktor bez argumentu -> Náhodná tvorba jedince (Používá se na začátku)
public Invidual()

/// Projde celou sekvenci a vypočítá vzdálenost mezi sekvencí měst.
public double getDistance()

/// Operátor mutace. 2 náhodné geny jsou prohozeny
public void mutate()

/// Konstruktor s argumentem -> Tvorba jedince z 2 rodičů
public Invidual(Invidual p1, Invidual p2)

/// Funkce si vypočítá nepoužitá města
/// Následně prochází sekvenci a nahrazuje duplicity těmi nepoužitými
private void fixMe()

/// Ohodnotí jedince a vrátí počet duplicit. Stejný prvek = kladné body
public int getDuplicity()

/// Vrátí sequenci genů jako string (použito pro tisk)
public string getSequence()
```

4.2.3 Zlepšení populace

Zlepšení populace je řešena na úrovni vnitřního jedince. Vnitřní jedinec má ve skutečnosti 2 konstruktory a 1 metodu pro mutaci. První konstruktor nemá žádné vstupy a vytvoří jedince náhodně. Druhý konstruktor má jako vstup 2 jiné jedince, neboli rodiče a z nich je právě vytvořen křížením.

Křížení je prováděno s 80% pravděpodobností a mutace s 5% pravděpodobností. Populace je periodicky zlepšována při přechodu do každé další generace. Může se stát, že se jedinec do další generace dostane bez změny. Zároveň se do další generace vždy dostane ten nejlepší jedinec z předchozí populace.

4.2.4 Fitness jedince

Fitness vnitřního jedince je jednoduchý výpočet, kdy je vypočítán součet všech vzdáleností mezi městy v posloupnosti vnitřního jedince. Tento výpočet je proveden v metodě GetFitness a celková vzdálenost není nikde uložena.

Zároveň je zde jednoduchý výpočet, kdy je proveden dotaz do databáze, která obsahuje všechny souřadnice měst. Odpovědí tohoto dotazu jsou souřadnice města. Obsluhu tohoto dotazu bylo nutné chvíli ladit, protože může značně zpomalit průběh výpočtu celého algoritmu.

Města jsou v posloupnosti reprezentována jejich unikátními identifikátory. Tato reprezentace umožňuje jednodušší dotazy do databází, protože identifikátor je zároveň řádkem v tabulce. Tato reprezentace se velmi dobře podepsala na rychlosti algoritmu.

4.2.5 Zobrazeni jedince

Poslední metodou, která byla potřeba, bylo zobrazení jedince pomocí metody ShowMe. Metoda ShowMe funguje podobně jako metoda GetFitness. Oproti metodě GetFitness ale pouze zobrazuje posloupnost měst do konzole.

Existují další varianty metody ShowMe jako ShowFull, nebo ShowTranslated. Tyto metody zobrazují jiným způsobem posloupnost měst a byly použity během vývoje a ladění.

4.2.6 List<string> – Orders | InvidualOrders | PopulationOrder

První způsob, kterým byl vnější jedinec reprezentován, byl List<string> neboli proměnlivé pole řetězců. Tento způsob měl výhodu toho, že byl velmi intuitivní a graficky názorný. Nevýhodou byly právě zmíněné komplikované křížení a mutace. Zároveň docházelo k mnoha duplicitám a ztrácela se města. Z toho důvodu bylo nutné přidat opravnou metodou FixMe. Tato funkce opravovala vnitřní jedince ve vnějším jedinci.

Tato opravná funkce označila duplicity a nahradila je chybějícími městy. Nevyužitá chybějící města potom přidala na konec poslední posloupnosti, případně jako další cesty.

Zde vznikal problém, že posloupnosti se uměle měnily k jednomu podobnému výsledku a bylo obtížné dělat jakékoli změny v programu.

4.2.7 SuperInvidual

Druhý způsob, který byl použit, byla reprezentace pomocí jednoho velkého pole datového typu string. Tohle pole obsahovalo pouze informace o řidičích, kteří cestu pojedou. Protože každá buňka reprezentovala jedno město a její hodnota řidiče, tak se nikdy nestalo, že by docházelo ke ztrátě informace o městě, nebo o řidiči.

Novým problémem ale bylo to, že mohlo dojít k přetížení řidiče. Tohle přetížení nebyl zásadní problém, protože pomocí fitness funkce bylo možné přidělovat negativní body, které by vnějšího jedince penalizovaly. Tento způsob byl nakonec zvolen a parametry, které slouží k ohodnocení vnějšího jedince, jsou drženy na úrovni Vnější populace.

Poslední detail při tvorbě vnějšího jedince bylo vhodné propojení metod vnějšího jedince a vnitřního jedince. Pokud třeba je požadavek k zobrazení jedince, tak musí být vhodně zobrazeni všichni vnitřní jedinci. Pokud je požadavek k ohodnocení vnějšího jedince, tak jsou ve skutečnosti ohodnoceni všichni vnitřní jedinci.

Kód v C#

```
public void generateRandomOrders(){
List<string> remainingCities = InputOutput.getUniqueColumnsValuesActiveOrders(1);
var rnd = new Random();
int a = rnd.Next(0, remainingCities.Count);
string city = remainingCities[a];
int cityCount = remainingCities.Count-1;
string newOrder = remainingCities[a];
remainingCities.RemoveAt(a);
int newOrderValue =
        Convert.ToInt32(Controller.activeOrdersTable.Rows[a]["Count"].ToString());
for (int i = 0; i < cityCount; i++){
a = rnd.Next(0, remainingCities.Count);
if (newOrderValue +
        Convert.ToInt32(Controller.activeOrdersTable.Rows[a]["Count"].ToString())
        < ORDER_CAPACITY)
{       newOrder = newOrder + "-" + remainingCities[a];
        remainingCities.RemoveAt(a);
        NewOrderValue = newOrderValue +
        Convert.ToInt32(Controller.activeOrdersTable.Rows[a]["Count"].ToString());}
else
{       newOrder = newOrder + ":" + newOrderValue;
        this.ordersList.Add(newOrder);
        newOrder = remainingCities[a];
        remainingCities.RemoveAt(a);
        newOrderValue =+
        Convert.ToInt32(Controller.activeOrdersTable.Rows[a]["Count"].ToString());}
}
if (newOrder.Length > 0){
        newOrder = newOrder + ":" + newOrderValue;
        this.ordersList.Add(newOrder);}}
```

Kód v C#

```
public SuperInvidual(){
routes = new int[InputOutput.getActiveOrdersCount()];
List<string> remainingCities = InputOutput.getUniqueColumnsValuesActiveOrders(1);
var rnd = new Random();
int a = rnd.Next(0, remainingCities.Count);
int driver = 1;
int routeValue = 0;
for (int i = 0; i < remainingCities.Count; i++){
        while (remainingCities[a] == "X") a = rnd.Next(0, remainingCities.Count);
        if (routeValue +
        Convert.ToInt32(Controller.activeOrdersTable.Rows[a]["Count"].ToString())
        < ORDER_CAPACITY)
        {routeValue = routeValue +
         Convert.ToInt32(Controller.activeOrdersTable.Rows[a]["Count"].ToString());
        routes[a] = driver;
        remainingCities[a] = "X";
        }
        else
        { driver = driver + 1;
        routeValue =
        Convert.ToInt32(Controller.activeOrdersTable.Rows[a]["Count"].ToString());
        routes[a] = driver;
        remainingCities[a] = "X";
        }    }
this.driverCount = driver;
startAlgorithm();
}
```

4.3 Pomocné třídy

Vzhledem ke komplexnosti úlohy byly v neposlední řadě vytvořeny pomocné třídy. Tyto třídy slouží pro zpřehlednění celé úlohy a pro jednodušší přidávání dalších funkcionalit do programu. Mezi tyto třídy patří Functions, InputOutput, Cities a Controller. Všechny tyto třídy jsou inicializovány na začátku programu a jsou průběžně volány. Některé slouží skutečně pouze k vývoji a ladění a nejsou nezbytné pro samotný chod programu.

4.3.1 Functions

Jedná se o třídu se statickými metodami, které slouží k zobrazení vstupu. Těmito metodami je možné zobrazit datový typ pole string, pole Int, List string, List int a mnoho dalších datových typů. Během začátku vývoje programu bylo zároveň nutné vytvořit vhodnou reprezentaci měst, která by zjednodušila práci s velkou databází.

Pro reprezentaci vnitřního jedince byla zvolena nejprve číselná hodnota v hexadické soustavě. Třída Functions obsahuje metody pro převod celého názvu města na číselnou hodnotu v hexadické soustavě. Reprezentace vnitřního jedince pomocí číselné hodnoty v hexadické soustavě se později ukázala jako nepraktická, protože převod probíhal velmi často a algoritmus byl značně zpomalen.

Kód v C#

```
/// Převede znak z Dec do Hex soustavy
private static char hexChar(int decValue)

/// Převede číslo z Dec do Hex soustavy
public static string toHex(int decNumber)

/// Zobrazí do konzole všechny prvky v listu
public static void showList(List<string> list)

/// Zobrazí do konzole všechny prvky v listu
public static void showList(List<int> list)

/// Zobrazi do konzole vsechny prvky v poli array
public static void showArray(string[] array)

/// Zobrazi do konzole vsechny prvky v poli array
public static void showArray(int[] array)

/// Zobrazím všechny data z tabulky allCitiesTable do Konzole
public static void showDataAllCitiesTable()
```

4.3.2 InputOutput

Tato třída slouží pro manipulaci se vstupními a výstupními soubory. Vzhledem k tomu, že požadavek zákazníka a databáze se souřadnicemi měst může být rozsáhlá a měnit se, tak byly vytvořeny metody, které tyto požadavky uspokojí.

Nejdůležitějšími metodami jsou ty, které čtou soubory a ukládají je do třídy Controller. Bylo nutné vyvinout převodníky a funkce pro čtení strukturovaných souborů. Zároveň metody tisknou přečtená data do zvláštních souborů. Další funkcí této třídy je tisk dat do souborů. Tato funkcionalita byla hojně využívána během vývoje programu. Vytisknout data do souboru a zobrazovat je v jiných aplikacích jako Excel se ukázalo jako velmi vhodné.

4.3.3 Cities

Tato třída slouží pro výpočty souvisejícími s městy. Původně když byla místo tabulky používána matice nejkratších vzdáleností, tak právě tato třída obsahovala všechny obslužné funkce. Mezi tyto metody patřila tvorba matice nejkratších vzdáleností, přístup do matice, mazání matice, tisknutí matice do souboru, překlad názvů měst a ukládání matice do hlavní třídy Controller. Po tom, co místo matice nejkratší vzdálenosti byla aplikována Pythagorova věta, tak se stala tato třída ve skutečnosti nadbytečná. Zůstaly zde metody, které pouze překládají názvy na identifikátory a obráceně.

Kód v C#

```
/// Vypočte vzdálenost mezi 2 body. x1 a y1 je bod 1. x2 a y2 je bod 2.
/// Výsledek je zaokrouhlen na 4 desetinna mista.
public static double getDistance(int x1, int y1, int x2, int y2)

/// Vrátí vzdálenost mezi 2 městy. Vstupem je název měst
/// Výsledek zaokrouhlen na 4 desetinná místa
public static double getDistance(string city1, string city2)

/// Vrátí vzdálenost mezi 2 městy. Vstupem jsou index měst
public static double getDistance(int city1, int city2)

/// Vrátí index města, podle kterého je možné v allCitiesTable najít jeho X a Y souřadnice
public static int getIndex(string city)

/// Vrátí ID města, když dodám název města
public static string getId(string city)

/// Vrátí index města podle názvu
public static int getIndexId(string city)
```

4.3.4 Controller

Třída Controller obsahuje inicializaci všech pomocných tříd. V této třídě se zároveň nachází tabulky se seznamem souřadnic měst a tabulka se seznamem požadavků zákazníků. Hlavní výhodou shromažďování dat v jedné hlavní třídě je ta, že je možné lehce řídit přistupování k datům.

Zároveň je možné lehce ovládat ostatní třídy a pouštět jednoduše algoritmus. Všechny třídy byly navrženy tak, aby je bylo možné pouštět pro různé druhy vstupů a pro různé parametry. Je možné tyto třídy použít a vyvinout grafické uživatelské rozhraní.

Jednou otázkou, kterou bylo nutné vyřešit, bylo to, jestli bude možné spustit algoritmus s jinými parametry. Tohle se ukázalo jako nepotřebné, protože by byl výsledek zkreslený. Je možné nastavit různé parametry algoritmu na začátku běhu. Nakonec se ukázalo, že nebylo nutné, aby se parametry algoritmu měnily během jeho chodu.

Posledním bodem, který bylo nutné rozhodnout, je forma zobrazení. Je možné výstup zobrazit na mapu, kde by bylo možné graficky zobrazit výsledek aplikace. Nakonec se ukázalo, že stačí výstup zobrazit pouze v příkazové řádce, nebo vytisknout do souboru. Tento soubor může být následně použit pro provedení další analýzy.

Kód v C#

```
// Seznam měst, ze kterých bude vybíráno
public static System.Data.DataTable allCitiesTable = new DataTable("allCities");
public static string pathAllCitiesTable = "..\\..\\..\\..\\" + "Coordinates.txt";
public static string pathDistinctAreasTable = "..\\..\\..\\..\\";

// Aktivní objednávky, kolik palet, který zákazník požaduje
public static System.Data.DataTable activeOrdersTable = new DataTable("activeOrders");
public static string pathActiveOrdersTable = "..\\..\\..\\..\\" + "ActiveOrders.txt";

// Název Kraje, ze kterého budu počítat matici
public static string selectedFile = "Pardubický kraj";
public static string selectedArea = "Pardubice";

public static string pathTest = "..\\..\\..\\..\\" + "Test.txt";
public static string pathOutput = "..\\..\\..\\..\\Output\\";
```

5 METODIKA A VYHODNOCENÍ POKUSŮ

Tato kapitola obsahuje naměřená data. Na základě těchto dat bylo rozhodnuto, jestli je vhodnější hledat optimální skupinu měst, které jsou blízko sebe, nebo jestli je vhodnější hledat přesnější nejkratší vzdálenost. Možných řešení je opravdu mnoho, proto je nutné provést měření mnohokrát a statistický průměr použít jako závěr.

Budou zkoumány 3 hlavní parametry algoritmu:

- Vliv počtu generací vnitřních a vnějších jedinců na jejich fitness.
- Vliv ceny za vzdálenost a auto na fitness jedinců.
- Vliv umístění cílových měst na fitness jedinců.

Z každého měření jsou pořízeny záznamy, které obsahovaly hodnoty jednotlivých parametrů algoritmu, hodnoty jedinců a jejich fitness. Samotný název souboru nese informace o měření. Existují dva druhy záznamů.

- Záznam s nejsilnějším vnějším jedincem.
- Záznam se všemi jedinci během běhu genetického algoritmu.

Na další straně následuje ukázka výstupního souboru, průběhu genetického algoritmu v 15. generaci, s nastavením 20 vnějších generací, 50 vnitřních generací a naměřeného 29. 3. 2020 v 16:23. Soubor se nazývá „Test1_Record_Vnejsi20Vnitrni50_Gen15-2020-03-29_16-23-39.txt“.

Název souboru vždy obsahuje parametry jako počet generací, kde „Vnejsi20“ znamená počet vnějších generací je 20 a „Record“ znamená, že se jedná o celý průběh genetického algoritmu. Na rozdíl od záznamu genetického algoritmu, záznam s nejsilnějším jedincem obsahuje pouze jednoho jedince.

Kód v C#

Parameters Invidual:

POPULATION_SIZE: 10 GENERATION_COUNT: 50

P_CROSSOVER: 80 P_MUTATION: 10

GOAL_DISTANCE: 0 DISTANCE_COST: 10

CAR_COST: 10000

Parameters SuperInvidual:

ORDER_CAPACITY: 32 OVERLOAD_PENALTY: 100

CROSSIN_POINT: 30 CAR_COST: 500

KM_COST: 10

2020-03-29 16:23:39

Super Invidual: 0

Fitness Invidual: 29.7147 Mikulovice-Lány u Dašic-Opatovice nad Labem

Fitness Invidual: 78.1995 Labské Chrčice-Břehy-Jedousov-Borek-Ostřetín

…

Fitness Invidual: 47.9853 Jankovice-Brloh-Lázně Bohdaneč-Chvojenec

Fitness Invidual: 41.1921 Litošice-Bukovina u Přelouče-Neratov-Křičeň-Čeperka

Fitness SuperInvidual: 13400

Inviduals: 14

…

Super Invidual: 9

Fitness Invidual: 29.7147 Mikulovice-Lány u Dašic-Opatovice nad Labem

Fitness Invidual: 78.1995 Ostřetín-Borek-Jedousov-Břehy-Labské Chrčice

…

Fitness Invidual: 47.9853 Chvojenec-Lázně Bohdaneč-Brloh-Jankovice

Fitness Invidual: 41.1921 Litošice-Bukovina u Přelouče-Neratov-Křičeň-Čeperka

Fitness SuperInvidual: 13430

Inviduals: 14

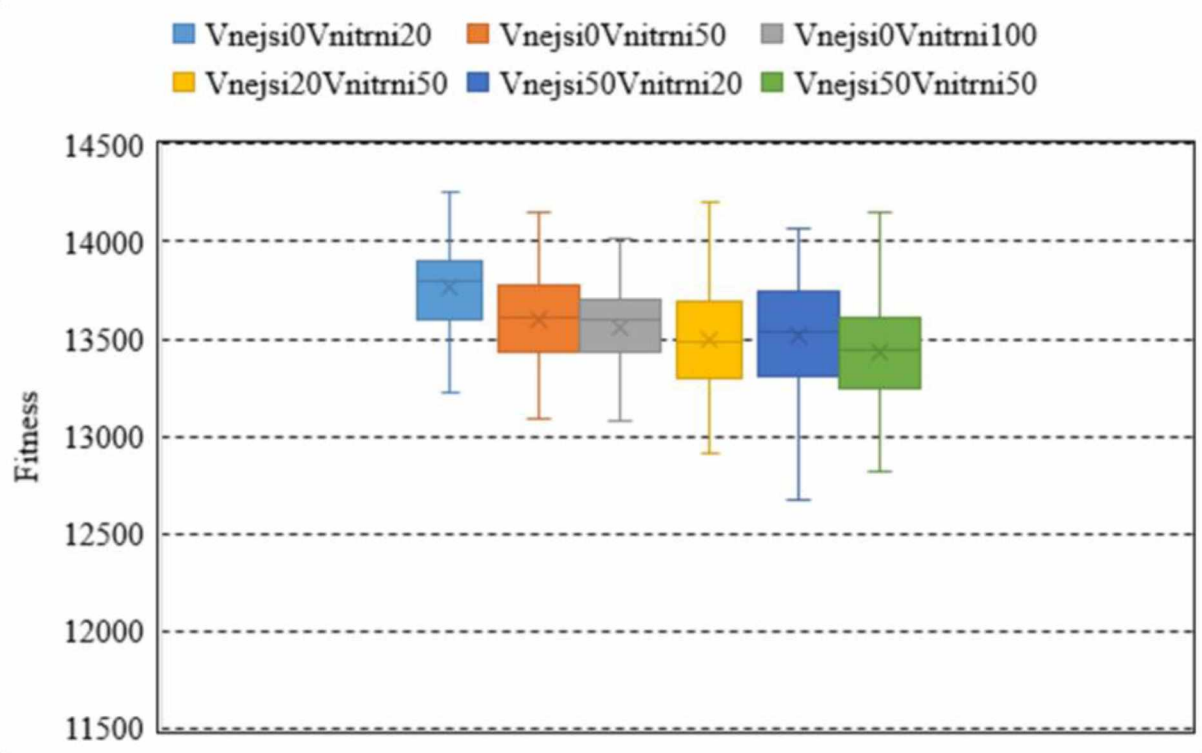
5.1 Zkoumání vlivu počtu generací vnitřního a vnějšího jedince

Pro otestování vlivu generací vnitřního a vnějšího jedince bude provedeno několik variant měření. Otázkou je, jestli je vhodnější hledat kratší vzdálenost, nebo jestli je lepší hledat vhodnější roztřídění měst. Byla provedena měření s těmito parametry:

- Počet generací vnitřního algoritmu nastaven na 20 a vnějšího algoritmu na 0.
- Počet generací vnitřního algoritmu nastaven na 50 a vnějšího algoritmu na 0.
- Počet generací vnitřního algoritmu nastaven na 100 a vnějšího algoritmu na 0.
- Počet generací vnitřního algoritmu nastaven na 20 a vnějšího algoritmu na 50.
- Počet generací vnitřního algoritmu nastaven na 50 a vnějšího algoritmu na 20.
- Počet generací vnitřního algoritmu nastaven na 50 a vnějšího algoritmu na 50.

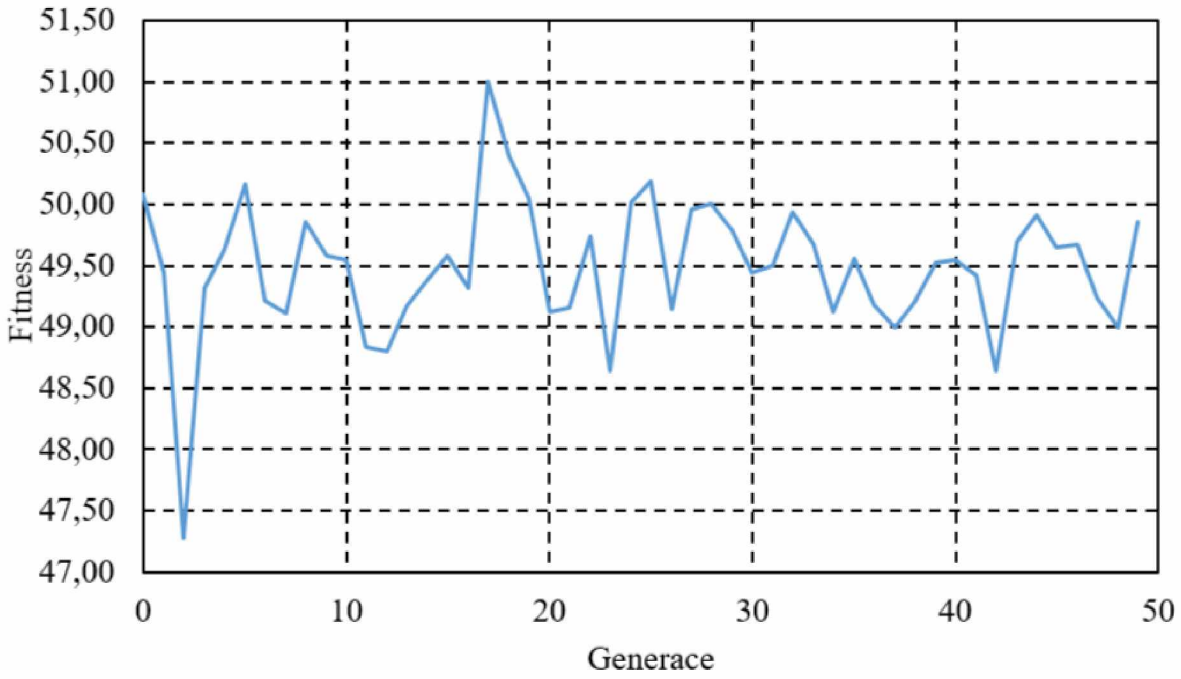
5.1.1 Shrnutí měření

Bylo provedeno celkem 300 měření. Pro každou variantu bylo provedeno měření padesátkrát. Na Obr. 5.1 jsou zobrazeny průměry fitness skóre vítězných jedinců, které byly výstupem měření. Měření jsou označena podle počtu generací. Například Vnejsi0Vnitrni20 je měření, kdy vnější jedinec byl nastaven na 0 generací a vnitřní jedinec na 20 generací.

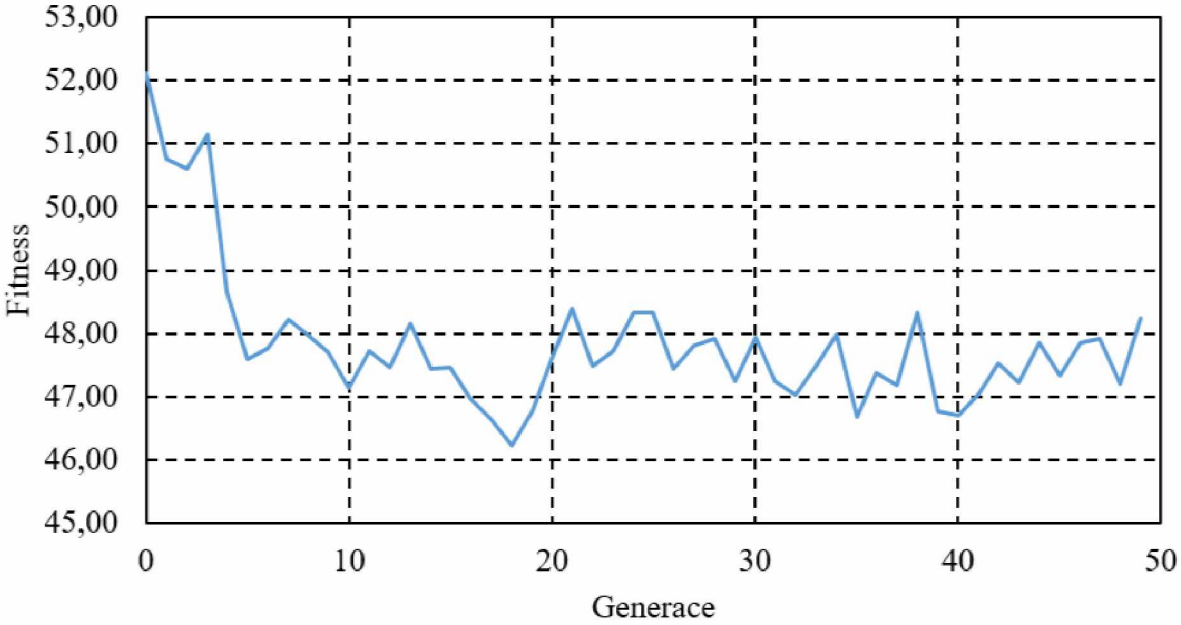


Obr. 5.1 – Parametru fitness vnějších jedinců během měření

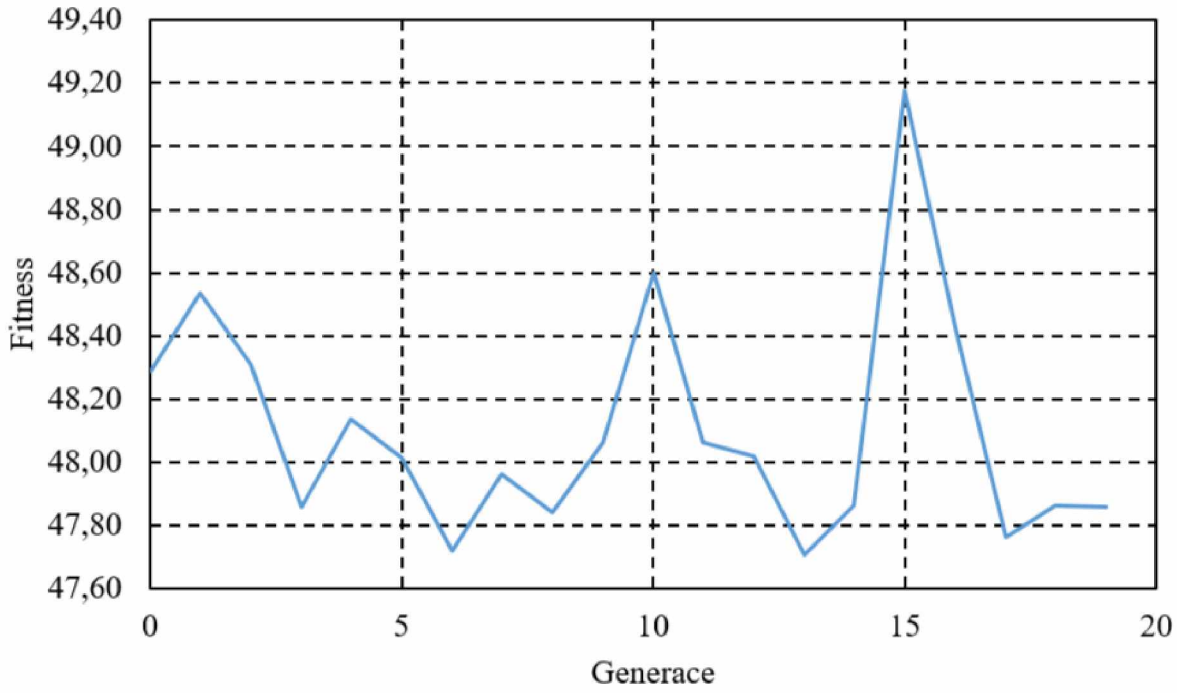
5.1.2 Grafy s naměřenými daty



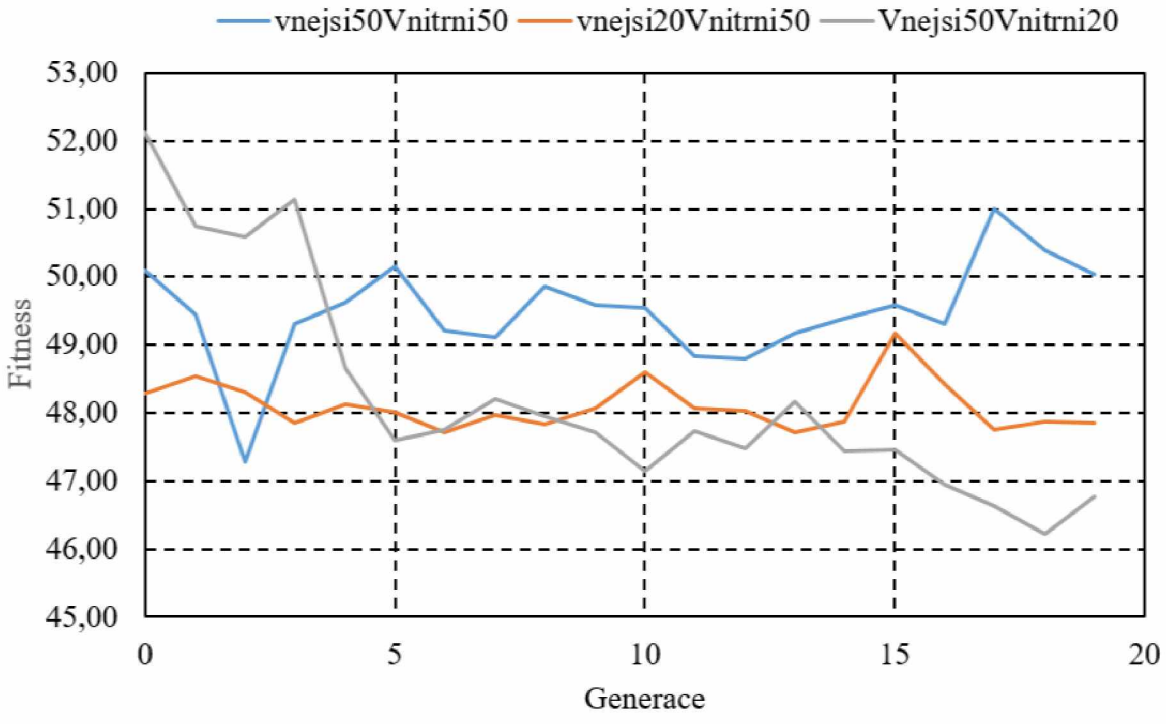
Obr. 5.2 – Parametr fitness nejlepšího jedince během měření Vnejsi50Vnitrni50



Obr. 5.3 – Parametr fitness nejlepšího jedince během měření Vnejsi50Vnitrni20



Obr. 5.5 – Parametr fitness nejlepšího jedince během měření Vnejsi50Vnitrni20



Obr. 5.4 – Srovnání měření fitness pro nejlepší jedince ze všech měření

5.2 Zkoumání vlivu ceny za kilometr a ceny za auto

Pro otestování vlivu ceny za kilometr a ceny za auto jsou provedena měření s těmito parametry:

- Počet generací vnitřního algoritmu nastaven na 20 a vnějšího algoritmu na 0, cena za kilometr je nízká, cena za řidiče je vysoká.
- Počet generací vnitřního algoritmu nastaven na 50 a vnějšího algoritmu na 0, cena za kilometr je nízká, cena za řidiče je vysoká.
- Počet generací vnitřního algoritmu nastaven na 100 a vnějšího algoritmu na 0, cena za kilometr je nízká, cena za řidiče je vysoká.
- Počet generací vnitřního algoritmu nastaven na 20 a vnějšího algoritmu na 50, cena za kilometr je nízká, cena za řidiče je vysoká.
- Počet generací vnitřního algoritmu nastaven na 20 a vnějšího algoritmu na 0, cena za kilometr je vysoká, cena za řidiče je nízká.
- Počet generací vnitřního algoritmu nastaven na 50 a vnějšího algoritmu na 0, cena za kilometr je vysoká, cena za řidiče je nízká.
- Počet generací vnitřního algoritmu nastaven na 100 a vnějšího algoritmu na 0, cena za kilometr je vysoká, cena za řidiče je nízká.
- Počet generací vnitřního algoritmu nastaven na 20 a vnějšího algoritmu na 50, cena za kilometr je vysoká, cena za řidiče je nízká.
- Počet generací vnitřního algoritmu nastaven na 20 a vnějšího algoritmu na 0, cena za kilometr a za řidiče je vyrovnaná.
- Počet generací vnitřního algoritmu nastaven na 50 a vnějšího algoritmu na 0, cena za kilometr a za řidiče je vyrovnaná.
- Počet generací vnitřního algoritmu nastaven na 100 a vnějšího algoritmu na 0, cena za kilometr a za řidiče je vyrovnaná.
- Počet generací vnitřního algoritmu nastaven na 20 a vnějšího algoritmu na 50, cena za kilometr a za řidiče je vyrovnaná.

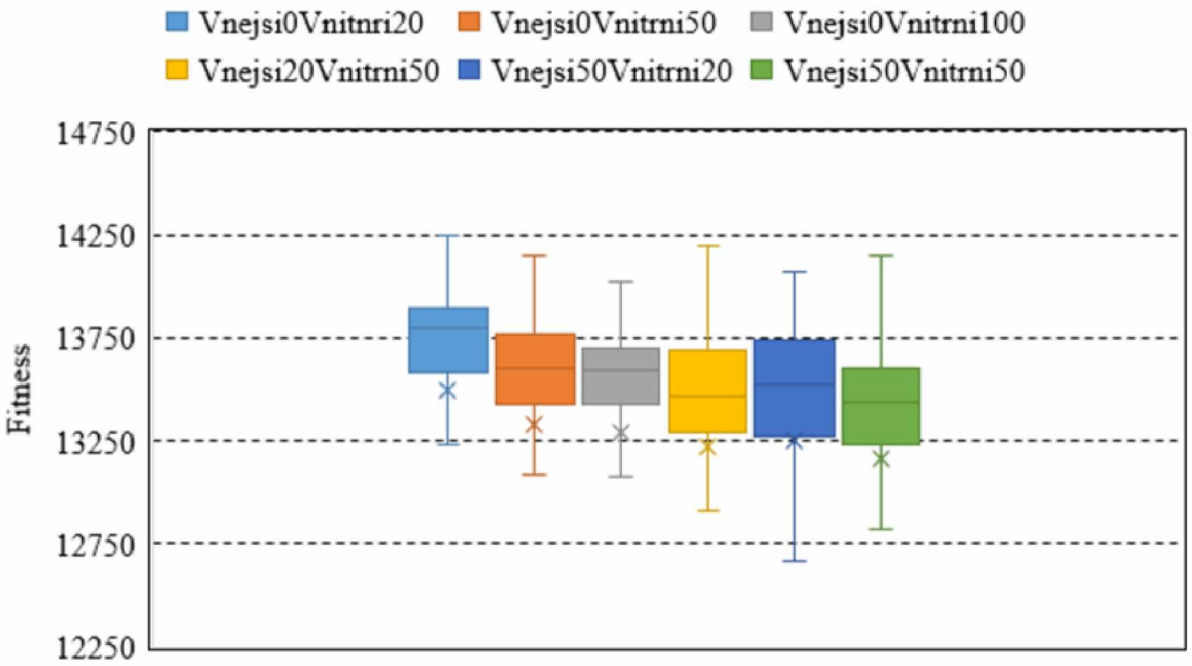
5.2.1 Shrnutí měření

Bylo provedeno celkem 240 měření. Pro každou variantu bylo provedeno měření dvacetkrát. Konkrétní ceny byly nastaveny následovně

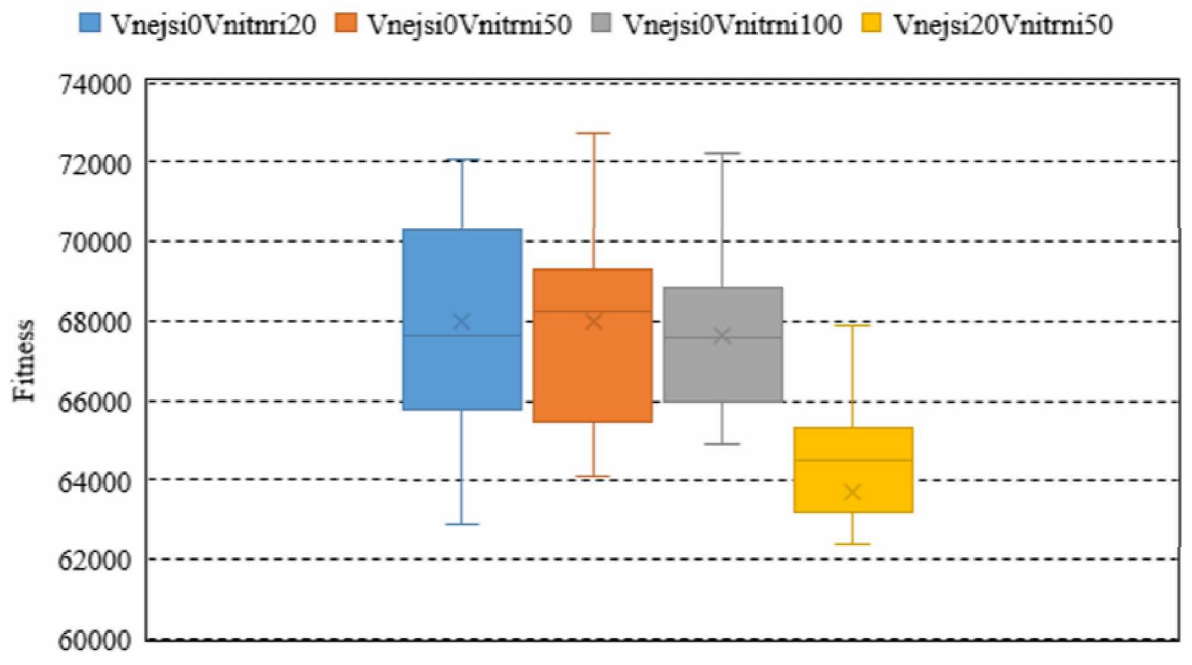
- Vysoká cena za kilometr je 100 za kilometr.
- Nízká cena za kilometr je 1 za kilometr.
- Vyrovnaná cena za kilometr je 10 za kilometr.
- Vysoká cena za auto je 5000 za řidiče.
- Nízká cena za auto je 50 za řidiče.
- Vyrovnaná cena za auto je 500 za řidiče.

Jednotlivé soubory s měření jsou opět pojmenovány podle počtu generací. Tedy Vnejsi50Vnitrni20 je měření, kde vnější jedinec je nastaven na 50 generací a vnitřní jedinec je nastaven na 20 generací.

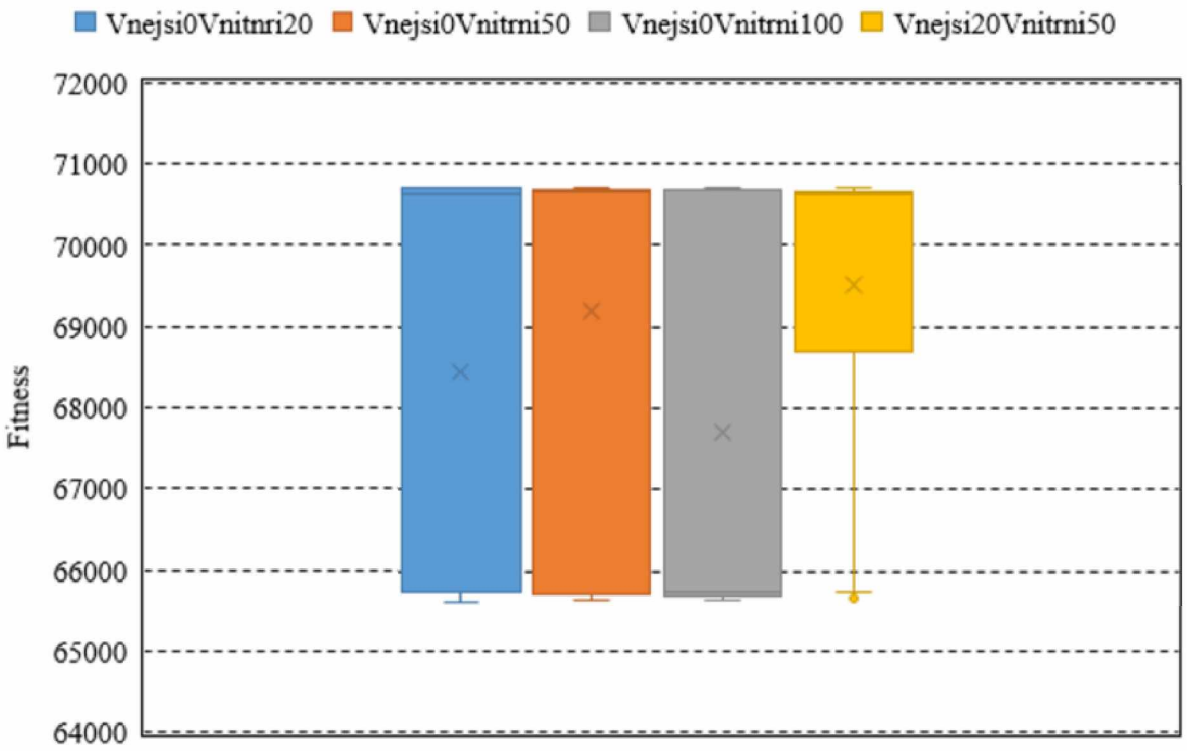
5.2.2 Grafy s naměřenými daty



Obr. 5.6 – Parametr fitness během měření s vyrovnanou cenou



Obr. 5.7 – Parametr fitness pro měření s vysokou cenou za kilometr a nízkou za auto



Obr. 5.8 – Parametru fitness pro měření s nízkou cenou za kilometr a vysokou za auto

5.3 Zkoumání vlivu umístění cílových měst

Pro otestování vlivu umístění cílových měst jsou provedeny následující pokusy:

- Počet generací vnitřního algoritmu nastaven na 20 a vnějšího algoritmu na 0, všechna cílová města se nachází pouze v Pardubickém kraji.
- Počet generací vnitřního algoritmu nastaven na 50 a vnějšího algoritmu na 0, všechna cílová města se nachází pouze v Pardubickém kraji.
- Počet generací vnitřního algoritmu nastaven na 100 a vnějšího algoritmu na 0, všechna cílová města se nachází pouze v Pardubickém kraji.
- Počet generací vnitřního algoritmu nastaven na 20 a vnějšího algoritmu na 0, cílová města se nachází v Pardubickém, Olomouckém kraji a kraj Vysočina.
- Počet generací vnitřního algoritmu nastaven na 50 a vnějšího algoritmu na 0, cílová města se nachází v Pardubickém, Olomouckém kraji a kraj Vysočina.
- Počet generací vnitřního algoritmu nastaven na 100 a vnějšího algoritmu na 0, cílová města se nachází v Pardubickém, Olomouckém kraji a kraj Vysočina.

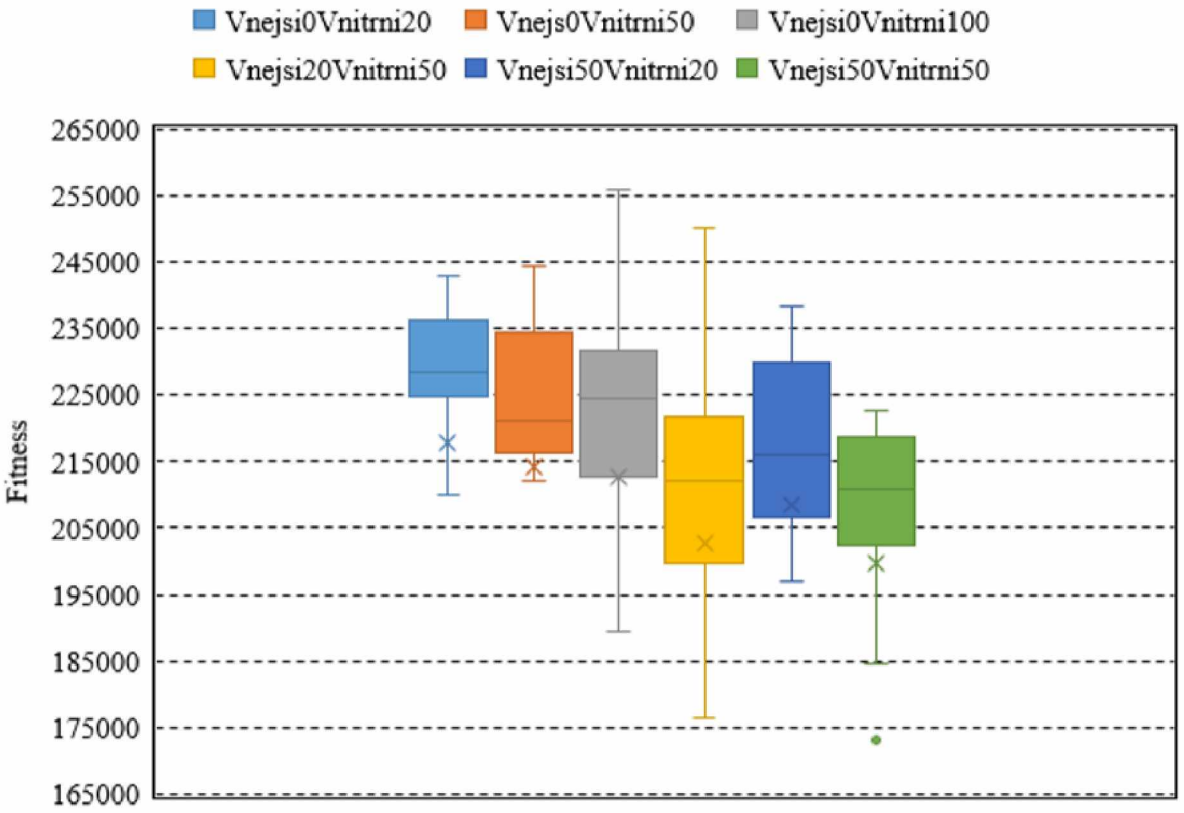
V těchto pokusech bude posuzováno, jak velký vliv má rozmístění měst na algoritmus.

5.3.1 Shrnutí měření

Bylo provedeno celkem 120 měření. Pro každou variantu bylo provedeno měření dvacetkrát. Města byla vybrána celkem ze tří krajů. Tyto kraje jsou Olomoucký kraj, Pardubický kraj a kraj Vysočina. Tyto kraje jsou od sebe vzdáleny vzdušnou čarou přibližně 100 km.

Jednotlivá měření jsou opět nazvána podle počtu generací. Tedy Vnejsi50Vnitrni20 je měření, kde vnější jedinec je nastaven na 50 generací a vnitřní jedinec je nastaven na 20 generací.

5.3.2 Grafy s naměřenými daty



Obr. 5.9 – Parametru fitness během měření s městy daleko od sebe

5.4 Vliv počtu generací na nejkratší vzdálenost

Z Obr. 5.1 je vidět, že algoritmus s pouze vnitřním členem dosáhl viditelného snížení vzdálenosti. Toto snížení se postupně snižovalo, až začalo docházet k saturaci a vzdálenost následně klesala minimálně. Dále je vidět, že použití vnějšího členu umožnilo tuto vzdálenost ještě více snížit.

V kombinaci dvaceti generací vnitřního člena a dvaceti generací vnějšího člena se projevila náhodná mutace, kdy v patnácté generaci několik současných mutací vnějších členů negativně ovlivnilo průměrnou fitness celé vnější generace. Kombinace dvaceti generací vnějšího člena a padesáti generací vnitřního člena umožnila o trošičku kratší vzdálenost, ale čas k dokončení algoritmu se značně zvýšil.

Rozmezí maximálních a minimálních hodnot se jinak drželo přibližně ve stejném rozptylu. Průměrná vzdálenost se postupně snižovala s přibývajícími generacemi.

Na Obr. 5.4 je vidět, jak náhodně vygenerované počáteční cesty ovlivnily cílový výsledek. Pokud byla počáteční cesta nevhodně zvolena, tak došlo ke značnému snížení velmi rychle. Pokud počáteční cesta byla vygenerována vhodně, tak algoritmus moc vzdálenost trasy nesnížil. Zde je vidět důležitost vnějšího algoritmu z důvodu rozmanitosti řešení.

5.5 Vliv ceny na fitness jedince

Podle Obr. 5.6 lze říci, že cena může značně ovlivnit cílovou trasu. V případě, když byla cena za kilometr vysoká, tak dosahovaly lepších výsledků algoritmy, kdy byl počet generací vnitřního jedince vyšší. Vnější jedinec zde fungoval pouze k vyhnutí se lokálnímu extrému.

Na Obr. 5.8 je na druhou stranu vidět, že cena za auto nehrála významnou roli vzhledem k tomu, že předměty musely být naloženy všechny a algoritmus přirozeně vybíral možnosti, kdy počet aut byl nižší. Nicméně pro jiná nastavení je možné, že větší počet aut je výhodnější.

5.6 Vliv rozmístění měst na nejkratší vzdálenost

Z Obr. 5.9 je zřejmé, že počet generací vnitřního jedince očekávaně snížil celkovou vzdálenost. Na druhou stranu zapojení vnějšího jedince měla mnohem větší dopad, protože města byla daleko od sebe a do výběru se mohlo dostat více sekvencí měst. V těchto sekvencích se právě mohly objevit skupiny měst, která byla právě v jednom kraji.

6 ZÁVĚR

Hlavním úkolem práce bylo se pokusit použit optimalizaci v logistice. Pomocí provedených měření se podařilo zjistit, jak dlouho má význam hledat nejkratší cestu, jak může fitness funkce ovlivnit algoritmus a jakou roli má rozdělení měst na menší celky.

Nejdůležitějším poznatkem této práce je, že vzdálenost mezi městy může znatelně ovlivnit optimální trasu. Je důležité tedy, aby byla cílová města rozdělena do menších celků, které jsou poblíž, a pro tyto skupiny byla hledána nejkratší vzdálenost.

Rozdělení měst do menších celků, které jsou poblíž, může být provedeno zvýšením počtu generací pro vnějšího jedince. Vyšší počet generací vnitřního člena přitom sníží samotnou vzdálenost, kterou řidič bude muset projet.

Samotná vzdálenost dosahuje nejznatelnějšího snížení už po dvaceti generacích, další generace se podílejí na stále menším a menším snížení této vzdálenosti. Během těchto pozdějších generací dochází sice k pokroku, ale doba k provedení algoritmu se značně zvyšuje.

V budoucnu by bylo možné na práci pokračovat zejména ve výzkumu vlivu počtu generací. Je možné algoritmus v aktuálním stavu nastavit na různé kombinace počtu generací. Například při počátečních generacích je možné nechat počet generací vnitřního jedince nízko a zvýšit je ke konci algoritmu, nebo po určitých milnících.

Zároveň je možné nastavit počty generací dynamicky. Kupříkladu kdyby fitness skóre dosáhlo zajímavé hodnoty, tak je možné pustit vnitřní algoritmus s vyšším počtem generací a přeskočit nezajímavé vnější jedince.

Další možností je optimalizace samotného algoritmu. Existují části, kde je možné algoritmus urychlit. Příkladem je přístup k datům o souřadnicích. Tato část se zásadně podílí na rychlosti algoritmu a výrazně ovlivňuje počet vstupů, které je možné zpracovat.

Poslední možností je přechod na jiné optimalizační úlohy. Zde to může být výpočet cesty hlavice pro 3D tiskárnu, vrták plošných spojů, nebo aplikace tvorbu rozvrhů. Práce celkově může posloužit jako dobrý základ pro podobné úlohy pouze s několika málo úpravami.

SEZNAM LITERATURY

C#. 1999. [on line]. Washington: Microsoft. [cit. 2020-05-01]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/

ČÁPKA, D. 2020. Hledání nejkratší cesty v grafu. ITnetwork.cz [online]. [cit. 2020-05-06]. Dostupné z: https://www.itnetwork.cz/navrh/algoritmy/algoritmy-grafove/hledani-nejkratsi-cesty-v-grafu

ČUČKA, M. 2007. Pythagorova věta a její důkazy. Brno: Vysoké učení technické v Brně. Diplomová. Vedoucí práce PhDr. Pavlína Račková

HORKÝ, A. 2011. Tvorba rozvrhů pomocí genetických algoritmů. Brno: Vysoké učení technické v Brně. Bakalářská. Vedoucí práce Miloš Minařík Ph.D.

HYNEK, J. 2008. Genetické algoritmy a genetické programování. Praha: Grada. Průvodce. ISBN 978-80-247-2695-3

CHONG, D. 2019. The Aged P versus NP Problem. Towards data science [online]. Toronto: Towards Data Science Inc. [cit. 2020-04-14]. Dostupné z: https://towardsdatascience.com/the-aged-p-versus-np-problem-91c2bd5dce23

KLAUS, V. 2014. Zjišťujeme vzdálenost mezi dvěma GPS souřadnicemi, tentokrát v ASP. NET. Vladimír Klaus BLOG [online]. Praha: Audrey software. [cit. 2020-04-16]. Dostupné z: https://www.vladimirklaus.cz/CZ/clanky-detail/33/zjistujeme-vzdalenost-mezi-dvema-gps-souradnicemi-tentokrat-v-aspnet

KOLÁŘ, J. 2000. Teoretická informatika. 2. vyd. Praha: Česká informatická společnost. [cit. 2020-04-16]. ISBN 80-900-8538-5

KUNZ, M. 2001. Matice vzdáleností. NATURA PLUS [online]. [cit. 2020-05-01]. ISSN 1212-6748. Dostupné z: http://natura.baf.cz/natura/2001/1/20010104.html

MÍČA, O. 2016. Solving Travelling salesman problem using Harmony search algorithm and other metaheuristics [online]. Quaere: Magnanimitas. [cit. 2020-04-12]. ISBN 978-80-87952-15-3. Dostupné z: https://dk.upce.cz//handle/10195/67047

Mapy.cz. 1996. [online]. Praha: Seznam.cz. [cit. 2020-05-01]. Dostupné z: www.mapy.cz

Traveling Salesman Problem. 2020. [online]. Sillicon Valley: Google OR-Tools. [cit. 2020-04-12]. Dostupné z: https://developers.google.com/optimization/routing/tsp

Variace. Matematika.cz. 2006. [online]. Brno: Vydavatelství Nová média. [cit. 2020-03-23]. Dostupné z: https://matematika.cz/variace

Vehicle Routing Problem. 2006. [online]. Sillicon Valley: Google OR-Tools. [cit. 2020-04-12]. Dostupné z: https://developers.google.com/optimization/routing/vrp

ZEMEK, M. 2018. Souradnice-mest. Github [online]. [cit. 2020-05-01]. Dostupné z: https://github.com/33bcdd/souradnice-mest

PŘÍLOHY

A – CD

Příloha A

Příloha k bakalářské práci

VYUŽITÍ OPTIMALIZACE V LOGISTICE

Lukáš Míšek

CD

Obsah