VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

OPTIMALIZAČNÍ ALGORITMY V LOGISTICKÝCH KOMBINATORICKÝCH ÚLOHÁCH

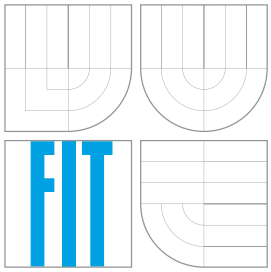
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. DANIEL BOKIŠ

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

OPTIMALIZAČNÍ ALGORITMY V LOGISTICKÝCH KOMBINATORICKÝCH ÚLOHÁCH

ALGORITHMS FOR COMPUTERIZED OPTIMIZATION OF LOGISTIC COMBINATORIAL PROBLEMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. DANIEL BOKIŠ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MARTIN HRUBÝ, Ph.D.

BRNO 2015

Abstrakt

Tato práce se zabývá optimalizačními problémy a především logistickou úlohou *Vehicle Routing Problem* (VRP). V první části je zaveden pojem optimalizace a jsou představeny nejdůležitější optimalizační problémy. Dále jsou v práci uvedeny metody, kterými je možné tyto problémy řešit. Následně jsou vybrané metody aplikovány na problém VRP a jsou uvedena některá jejich vylepšení. Práce také představuje metodu využívání znalostí předchozích řešení, tedy formu učícího algoritmu. V závěru práce jsou experimentálně optimalizovány parametry jednotlivých metod a ověřen přínos představených vylepšení.

Abstract

This thesis deals with optimization problems with main focus on logistic *Vehicle Routing Problem* (VRP). In the first part term optimization is established and most important optimization problems are presented. Next section deals with methods, which are capable of solving those problems. Furthermore it is explored how to apply those methods to specific VRP, along with presenting some enhancement of those algorithms. This thesis also introduces learning method capable of using knowledge of previous solutions. At the end of the paper, experiments are performed to tune the parameters of used algorithms and to discuss benefit of suggested improvements.

Klíčová slova

VRP, logistika, distribuce, optimalizace, kombinatorická optimalizace, heuristika, metaheuristika, učení, učící algoritmus

Keywords

VRP, logistic, distribution, Vehicle Routing Problem, optimization, combinatoric optimization, heuristic, metaheuristic, learning, learning algorithm

Citace

Daniel Bokiš: Optimalizační algoritmy v logistických kombinatorických úlohách, diplomová práce, Brno, FIT VUT v Brně, 2015

Optimalizační algoritmy v logistických kombinatorických úlohách

Prohlášení

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

.....
Daniel Bokiš
27. května 2015

Poděkování

Tímto bych chtěl poděkovat vedoucímu práce Ing. Martinu Hrubému, Ph.D. za jeho odborné rady při konzultacích. Také bych chtěl poděkovat své přítelkyni za jazykovou korekturu práce a podporu.

Obsah

Úvod **3**

1 Kombinatorické optimalizační úlohy **4**
1.1 Logistické úlohy . . . 5
1.1.1 Travelling Salesman Problem . . . 5
1.1.2 Vehicle Routing Problem . . . 6
1.2 Resource-Constrained Project Scheduling Problem . . . 10
1.3 Další kombinatorické úlohy . . . 11

2 Algoritmy pro řešení kombinatorických úloh **13**
2.1 Exaktní algoritmy . . . 13
2.2 Heuristické algoritmy . . . 15
2.3 Metaheuristické algoritmy . . . 16
2.3.1 Genetické algoritmy . . . 16
2.3.2 Tabu prohledávání . . . 17
2.3.3 Simulované žíhání . . . 17
2.3.4 Mravenčí kolonie . . . 18

3 Aplikace optimalizačních algoritmů na VRP **20**
3.1 Obecná tvrzení o řešení . . . 20
3.2 Reprezentace řešení . . . 22
3.3 Tvorba počátečního řešení . . . 23
3.4 Genetický algoritmus . . . 25
3.4.1 Výběr rodičů . . . 25
3.4.2 Křížení . . . 27
3.4.3 Mutace . . . 28
3.4.4 Mutace části populace před křížením . . . 28
3.4.5 Zamezení výskytu klonů . . . 28
3.4.6 Restart stagnující populace . . . 29
3.4.7 Finální genetický algoritmus . . . 30
3.5 Tabu prohledávání . . . 31
3.6 Simulované žíhání . . . 31
3.7 Mravenčí kolonie . . . 32

4 Heuristika využívající charakteristiky předchozích řešení **33**
4.1 Charakteristiky řešení . . . 33
4.2 Runtime hypotéza vylučujících charakteristik . . . 34
4.2.1 Aplikace na VRP . . . 35

4.3 Navržené mutace . . . 36
4.3.1 RTShift . . . 36
4.3.2 RTCluster . . . 37
4.3.3 RTReshuff . . . 38
4.4 Shrnutí . . . 39

5 Implementace **40**
5.1 Parametry a konfigurace programu . . . 40
5.2 Popis tříd . . . 41
5.3 Optimalizace . . . 43

6 Experimenty **45**
6.1 Popis problémů pro srovnávání . . . 46
6.2 Popis experimentování . . . 48
6.3 Výsledky z literatury . . . 50
6.4 Genetické algoritmy . . . 51
6.4.1 Velikost populace . . . 51
6.4.2 Volba selekčního operátoru . . . 52
6.4.3 Počet potomků . . . 53
6.4.4 Pravděpodobnost mutace potomků . . . 53
6.4.5 Počet mutovaných jedinců . . . 54
6.4.6 Parametry částečného nahrazení . . . 55
6.4.7 Zamezení výskytu klonů . . . 55
6.5 Tabu prohledávání . . . 57
6.5.1 Velikost tabu seznamu . . . 57
6.5.2 Počet generovaných sousedních řešení . . . 57
6.6 Simulované žíhání . . . 59
6.6.1 Počáteční teplota . . . 59
6.6.2 Změna teploty . . . 60
6.7 Optimalizace mravenčí kolonií . . . 61
6.8 Metody využívající RT tabulky . . . 62
6.8.1 Výběr nejhorší nebo nejlepší charakteristiky . . . 62
6.8.2 Vliv vylepšení RTShift . . . 63
6.8.3 Parametry mutace RTReshuff . . . 63
6.8.4 Vliv použití metod využívajících RT tabulku . . . 64
6.8.5 Poměr mutací *RTReshuff* a *RTCluster* . . . 66
6.8.6 Počátek zapojení RT metod . . . 67
6.8.7 Vliv použití vyplněné tabulky . . . 68
6.8.8 Dlouhý běh . . . 69

Závěr **72**

Použitá literatura **75**

Seznam použitých zkratek a symbolů **76**

Úvod

V dnešní době je důležité co nejvíce optimalizovat všechny průmyslové činnosti, a to především za účelem snížení nákladů, případně výsledné ceny produktu. Můžeme se bavit o optimalizaci plánování výroby, kde je klíčové rozvržení operací tak, aby byl celkový čas co nejkratší, nebo lze například zmínit optimalizaci logistiky. V oblasti logistiky obvykle plánujeme distribuci nějakého zboží k zákazníkům, přičemž potřebujeme zajistit naplánování takové trasy, která bude nejkratší jak z hlediska ujetých kilometrů, tak z hlediska času, neboť za oba tyto artikly se platí. Se stále zvyšující se cenou pohonných hmot a většími nároky zákazníků na přesnost času dodání, je tak nutnost optimalizace ještě větší, nežli dříve.

Pokud musíme obsloužit více zákazníků, nebo například chceme aktuálně reagovat na situaci na silnici (uzavírky apod.), již zdaleka nepostačí ruční plánování a je nutné řešit tuto optimalizaci pomocí počítače. Dobrým optimalizačním algoritmem pak dokážeme zkrátit celkovou cestu někdy až o desítky procent a ušetřit tak velké množství času a prostředků. Tato úloha však ani pro počítačové zpracování není jednoduchá a především pro velké instance s velkým množstvím zákazníků je rozdíl mezi výsledkem dobrého a špatného algoritmu relativně velký. Tématu optimalizovaného plánování trasy bylo v literatuře věnováno poměrně velké množství odborné literatury, což dokládá například sborník obsahující 500 publikací, který vyšel již v roce 1995 [21]. Od tohoto roku zajisté vzniklo nespočet dalších nových výzkumů, což dokládá smysl stále se tomuto problému věnovat. Matematickým modelem samotných logistických úloh je pak například optimalizační problém *Vehicle Routing Problem* (VRP), kterému se tato práce bude věnovat.

Práce samotná je rozdělena na šest kapitol, kde v první kapitole je zaveden pojem optimalizace a jsou představeny základní optimalizační problémy, především z oblasti logistiky. V druhé kapitole jsou zkoumány různé možnosti řešení těchto problémů, kdy je kladen důraz na představení těchto metaheuristických metod: genetické algoritmy (GA), tabu prohledávání (TABU), simulované žíhání (SA) a optimalizaci mravenčí kolonií (ANT). Tyto čtyři algoritmy jsou v následující kapitole v uvedeném pořadí aplikovány na konkrétní problém VRP a jsou prezentována některá jejich vylepšení. Čtvrtá kapitola představuje učící se systém vycházející z charakteristik předchozích řešení a obsahuje návrh aplikace této metody na zkoumané algoritmy. Samotná implementace algoritmů a celého systému pro řešení VRP je popsána v kapitole 5. V závěrečné kapitole jsou pomocí řady experimentů optimalizovány parametry uvedených algoritmů (GA, TABU, SA a ANT) a je diskutován přínos představených vylepšení a učících metod.

Kapitola 1

Kombinatorické optimalizační úlohy

Optimalizací myslíme v matematice úlohu, při jejímž řešení odpovídáme na otázku „které řešení je nejlepší“, a to pro ty problémy, kde je možné kvalitu jednotlivého řešení ohodnotit jedním číslem, hodnotou *optimalizační funkce*. S těmito problémy se můžeme setkat v mnoha oblastech, ať už matematice, chemii, strojírenství, ekonomii nebo právě v logistice. Optimalizační funkcí se pak snažíme ohodnotit dané řešení pro účely porovnání, což může v praxi odpovídat například počtu operací, času sekvence výrobních operací, nebo délce ujeté trasy. Následně hledáme nejčastěji minimum této funkce, což je typické například pro čas či vzdálenost. V aplikacích, kde chceme hledat její maximum (např. maximalizace plochy), převádíme maximalizační problém na minimalizační prostou negací optimalizační funkce.

Matematicky můžeme obecnou optimalizační úlohu definovat jako [6]:

$$\begin{aligned} &\text{minimalizace } f_0(x) \\ &\quad \text{podléhající } f_i(x) \leq b_i, \quad i = 1, ..., m, \end{aligned} \tag{1.1}$$

kde vektor $x = (x_1, ..., x_n)$ je *optimalizační proměnná* úlohy, funkce $f_0 : \mathbb{R}^n \to \mathbb{R}$ je optimalizační funkce, funkce $f_i : \mathbb{R}^n \to \mathbb{R}, i = 1, ..., m$ jsou *podmínkové funkce* a konstanty $b_1, ..., b_m$ jsou hranice pro dané podmínky. Vektor x^* je nazýván *optimální*, nebo také řešením problému (1.1), jestliže má nejmenší hodnotu optimalizační funkce ze všech vektorů vyhovujících stanoveným podmínkám. Formálně vyjádřeno, pro každé z, kde je splněno $f_1(z) \leq b_1, ..., f_m(z) \leq b_m$, platí, že $f_0(z) \geq f_0(x^*)$.

Jednou z nejzajímavějších podtříd optimalizačních úloh jsou pak *kombinatorické úlohy*. Proměnné v těchto úlohách jsou diskrétní a optimum hledáme v konečné, respektive spočetné množině řešení [27]. Typicky velikost této množiny, tedy prohledávaného prostoru, roste exponenciálně vzhledem k počtu objektů, které reprezentuje (uzly v grafu, výrobní operace atd.). Proto je prohledávání všech těchto možných řešení časově nepřípustné a musíme hledat jiné, efektivnější metody. Řešení, která splňují definované podmínky pak nazýváme jako *platná* a můžeme definovat prostor platných řešení $\mathbb{P}$, ve kterém hledáme optimum.

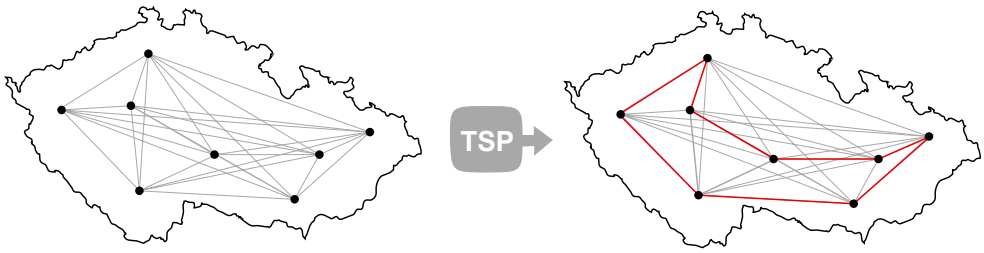
V této kapitole budou definovány základní logistické úlohy, kterými se tato práce dále zabývá. Tyto úlohy typicky rozvrhují, jakým způsobem navštívit zadané místa či zákazníky, a to za splnění určitých kapacitních podmínek. Následně bude zkoumána úloha rozvrhování výrobních operací, což je další kapacitní úloha, která bude zmíněna z důvodů inspirace některými metodami jejího řešení. V závěru kapitoly budou pro úplnost popsány některé další nejznámější kombinatorické úlohy.

1.1 Logistické úlohy

Tato práce je zaměřena na optimalizaci logistických úloh, proto si nejprve definujeme tyto úlohy. V úvodu bude zmíněna klasická logistická úloha obchodního cestujícího, která rozvrhuje pořadí navštívených míst bez dalších kapacitních podmínek. Na tuto úlohu navazuje *Vehicle Routing Problem* (VRP), který představuje kapacitní podmínky, které je nutné v řešení dodržet. Jelikož právě VRP se tato práce dále věnuje, budou u tohoto problému zmíněny další jeho varianty, které jej dále rozvíjejí.

1.1.1 Travelling Salesman Problem

Travelling Salesman Problem (TSP) [17], tedy problém obchodního cestujícího, je jedním z nejznámějších a nejpopulárnějších kombinatorických úloh. Ačkoliv se jedná o problém, kterým se zabývali matematici již v 18. století, má stále své využití (v logistice, vývoji elektrických obvodů atd.) a také slouží jako poměřovací a testovací problém současných badatelů. V jednoduchosti lze problém obchodního cestujícího vysvětlit tak, že obchodní cestující musí při své cestě navštívit každé z přidělených měst právě jedenkrát a nakonec se vrátit do výchozího bodu. Příklad zadání a řešení úlohy je ilustrován obrázkem 1.1. Cílem je pak optimalizovat cestovní náklady, tedy typicky celkovou ujetou vzdálenost, čas cesty, nebo jejich kombinaci, přičemž jsou definovány náklady mezi všemi jednotlivými městy.



Obrázek 1.1: Modelový příklad úlohy obchodního cestujícího. Vlevo vidíme rozmístění měst a vpravo pak možné řešení problému (červeně).

Formálně uvažujme graf $G = (V, E)$, kde $V = \{1, ..., n\}$ je množina uzlů (měst), $E = \{(i, j)|i, j = 1, ..., n\}$ je množina hran (spojnic mezi městy) a $c_{ij} > 0, i, j = 1, ..., n$, nechť je konstanta určující náklady přesunu z města i do města j. Potom hledáme minimum funkce [17]:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \tag{1.2}$$

za splnění podmínek:

$$\begin{aligned}
&\sum_{i=1}^{n} x_{ij} = 1, j = 1, ..., n,\\
&\sum_{j=1}^{n} x_{ij} = 1, i = 1, ..., n,\\
&\sum_{i\in S}\sum_{j\in S} x_{ij} \leq |S| - 1 \text{ pro všechny } S \subset \{1, ..., n\}, S \neq \emptyset,\\
&x_{ij} \in \{0, 1\}, i, j = 1, ..., n.
\end{aligned} \tag{1.3}$$

Odtud je zřejmé, že hledaná matice X je definována tak, že $x_{ij} = 1$, právě když nalezené řešení obsahuje přesun z města i do města j a $x_{ij} = 0$ v ostatních případech. Typicky také uvažujeme takzvaný symetrický problém obchodního cestujícího, kde navíc platí podmínka:

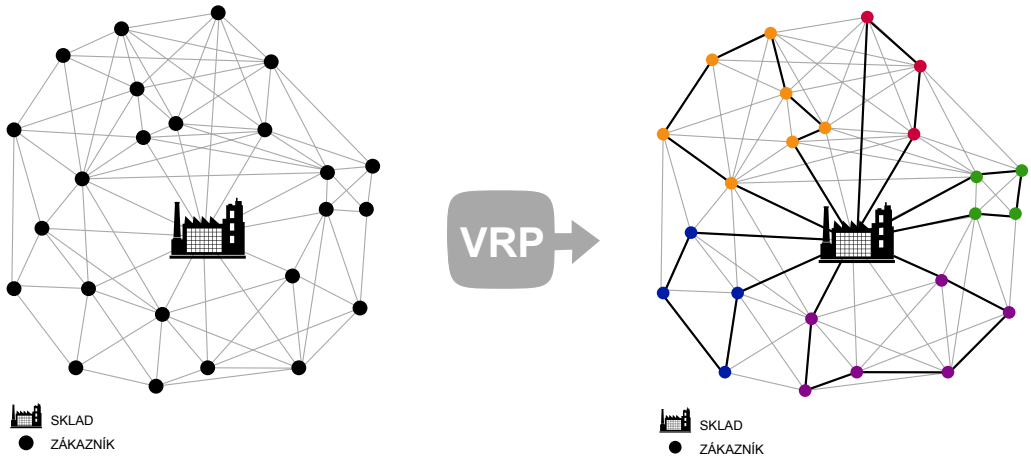
$$c_{ij} = c_{ji}, i, j = 1, ..., n \tag{1.4}$$

Tedy vzdálenost (resp. náklady) mezi městy jsou stejné bez ohledu na směr cesty.

Řešení problému obchodního cestujícího pak může zastupovat jedna permutace množiny měst 1 až n, která určuje, v jakém pořadí má obchodní cestující trasu uskutečnit. Jelikož v TSP nejsou definovány žádné omezující kapacitní podmínky, každá tato permutace je platná a počet platných řešení by mohl být $|\mathbb{P}| = n!$, jelikož se jedná o permutaci bez opakování. Nicméně u TSP v podstatě nezáleží z jakého města vyjíždíme a jakým směrem pojedeme, pokud spojíme všechny místa. Reálně je tak $|\mathbb{P}| = (n-1)!/2$ [30].

1.1.2 Vehicle Routing Problem

Vehicle Routing Problem (VRP), někdy překládaný jako problém okružních jízd, je logistická úloha podobná TSP. V této úloze uzly chápeme jako zákazníky a jeden (v některých variantách i více) uzel bereme jako sklad, tedy výchozí a zároveň koncový bod. Cílem je flotilou vozidel obsloužit všechny zákazníky tak, aby byla minimalizována celková ujetá vzdálenost všech vozidel. Všechna vozidla vyjíždějí ze skladu a na konci své cesty se do něj musí vrátit. Navíc jsou definovány kapacitní podmínky, které určují platnost řešení. Jedná se tak o významný problém v oblasti logistiky, dopravy a distribuce. Ilustraci tohoto problému můžeme vidět na obrázku 1.2.



Obrázek 1.2: Modelový příklad úlohy VRP. Vlevo vidíme rozmístění zákazníků a skladu (z důvodů zjednodušení nebyly zakresleny všechny možné šedé spojnice), vpravo pak možné řešení, kde tučná černá čára označuje cestu vozidla a jednotlivé cesty jsou barevně odlišeny.

Formálně podobně jako u TSP uvažujme úplný graf $G = (V, E)$ a následující notace [41, 25]:

- $V = \{v_0, v_1, ..., v_n\}$ je množina vrcholů, tedy zákazníku, kde uvažujme, že sklad je umístěn ve vrcholu v_0. Potom nechť $V' = V \setminus \{v_0\}$ reprezentuje n zákazníků.
- $E = \{(v_i, v_j) | v_i, v_j \in V; i \neq j\}$ je množina hran, tedy spojnic mezi zákazníky.

- C je matice kladných nákladů c_{ij} mezi zákazníky v_i a v_j. Uvažujme $c_{ij} = c_{ji}$ a $c_{ii} = 0$.
- $d(v_i) : i \in \{1, ..., n\}$ definuje kapacitní požadavek zákazníka i, přičemž sklad má přiřazen fiktivní požadavek $d(v_0) = 0$.
- m vyjadřuje počet vozidel, přičemž jsou všechny identické. Každému vozidlu je pak přiřazena jedna cesta.
- $R_i : i \in \{1, ..., m\} = (v_0, v_{i_1}, ..., v_{i_{k(i)}}, v_0)$ je trasa (cesta) vozidla i. Každá trasa pak začíná a končí ve skladu a počet uzlů trasy R_i je rovno $k(i)$.
- Platí $\forall i(v_i \in V' \Rightarrow \exists j : v_i \in R_j \wedge \nexists k \neq j : v_i \in R_j \wedge v_i \in R_k)$, což značí, že každý zákazník je navštíven právě jednou a právě jedním vozidlem.

V některých případech jsou hodnoty nákladů c_{ij} chápány jako vzdálenost mezi zákazníky, jindy jako doba jízdy a nebo abstraktní cena přepravy [20]. V každém případě je celková cena každé trasy R_i definována jako:

$$Cost(R_i) = \sum_{j=0}^{k(i)} c_{i_j, i_{j+1}} \tag{1.5}$$

Konečně cena celého řešení problému označeného S je určena součtem cen jednotlivých tras jako $Cost(S) = \sum_{i=1}^{m} Cost(R_i)$. Cílem úlohy VRP je tuto celkovou cenu minimalizovat. Dále v textu se setkáme s pojmem hodnotící (*fitness*) funkce $F(S)$. Ve VRP je tímto myšlena celková cena, či délka řešení a můžeme prohlásit, že $F(S) = Cost(S)$.

VRP s omezením délky trasy

Distance-Constrained VRP (DVRP) rozšiřuje základní úlohu o podmínku maximální délky jedné trasy [38]. Někdy se v této variantě mluví o omezení délky trvání jedné trasy, zejména pokud chápeme náklady c_{ij} jako dobu jízdy. Typicky je v této variantě definován servisní čas δ_i, který označuje čas potřebný pro vyložení zboží u zákazníka i [25]. Potom je cena (resp. délka trvání) každé trasy R_i specifikována jako:

$$Cost(R_i) = \sum_{j=0}^{k(i)} c_{i_j, i_{j+1}} + \sum_{j=1}^{k(i)-1} \delta_{i_j} \tag{1.6}$$

Žádné vozidlo či trasa pak nesmí překročit stanovenou maximální cenu (délku, nebo čas) trasy $Cost_{max}$, což je definováno podmínkou:

$$\forall i \in \{1, ..., m\} : Cost(R_i) \leq Cost_{max} \tag{1.7}$$

VRP s kapacitou vozidla

Capacitated VRP (CVRP) je definuje kapacitu vozidel, kterou nesmí během své cesty překročit [25]. K základní úloze VRP navíc definujeme kapacitu vozidla Q a kapacitní podmínku (1.9). Nejprve vyjádřeme celkový kapacitní požadavek D každé trasy R_i jako:

$$D(R_i) = \sum_{j=1}^{k(i)} d(v_{i_j}) \tag{1.8}$$

Dále řešení úlohy musí splňovat podmínku, že požadavky zákazníků na každé trase R_i nesmí překročit kapacitu vozidla Q, tedy:

$$\forall i \in \{1, ..., m\} : D(R_i) \leq Q \tag{1.9}$$

Právě varianta CVRP bývá často kombinována s DVRP jako DCVRP, kdy platí obě výše definované podmínky (1.7) a (1.9) [38]. Přesně touto variantou se bude tato práce dále zabývat, přičemž některé použité instance problému mají definovánu maximální délku trasy (jsou tedy DCVRP) a jiné jsou klasickým CVRP s $Cost_{max} = \infty$. V literatuře jsem se však téměř s pojmem DCVRP nesetkal a někteří autoři tuto variantu označují pouze jako CVRP [22, 24], DVRP [28, 39], nebo jednoduše jako VRP [12]. Poslední zmiňované se zdá jako nejméně matoucí, jelikož se v literatuře téměř nesetkáváme s čistou verzí VRP bez dalších podmínek.

Tato práce se pak zabývá právě problémem DCVRP, označen dále jednoduše jako **VRP**. Tento problém definují notace zavedené v 1.1.2 a zároveň musí platit podmínka dodržení maximální ceny trasy (1.7) a kapacitní podmínka (1.9).

Platným řešením $S \in \mathbb{P}$ této úlohy je pak zařazení zákazníků do jednotlivých tras tak, aby byli všichni zákazníci obslouženi a aby byly u všech tras vozidel splněny kapacitní podmínky (1.7) a (1.9). S konkrétním vyčíslením velikosti $\mathbb{P}$ jsem se v literatuře nesetkal, ale můžeme předpokládat, že je prostor rozmístěním do tras a zavedením dalších podmínek větší, než v případě TSP.

Další varianty VRP

V předchozí sekci byla představena základní úlohy VRP, kterou se budeme v práci dále zabývat. V praxi existuje mnoho dalších variant této úlohy, a pro úplnost budou zmíněny ty nejvýznamnější z nich.

VRP s více sklady

Multiple Depot VRP (MDVRP) je varianta, jak už název napovídá, ve které se vyskytuje více skladů. Pokud jsou zákazníci rozmístění kolem skladů, může se problém řešit jako několik samostatných VRP. Pokud jsou ale promícháni, je nutné řešit MDVRP, které vyžaduje přiřazení zákazníků k jednotlivým skladům. Je totiž nutné splnit podmínku, že každé vozidlo má základnu ve svém skladu a také se tam musí vrátit, nelze tedy začít v jednom skladu a vrátit se do jiného. Cílem je opět obsloužit všechny zákazníky a zároveň minimalizovat celkovou ujetou vzdálenost.

Lehce odlišnou variantou je VRP se satelitními sklady (*VRP with Satellite Facilities*), kde jsou definovány sklady, ve kterých mohou vozidla doplnit zásoby, pokračovat v cestě a na konci dne se vrátit do svého domovského skladu. Tato varianta nalezne uplatnění především v distribuci paliv či maloobchodním prodeji [25].

VRP s časovými okny

VRP with Time Windows (VRPTW) je klasické VRP s přidanou podmínkou, kde ke každému zákazníkovi i je přiřazen časový interval (okno) $[a_i, b_i]$, ve kterém musí být obsloužen. Dále je také skladu přiřazen interval $[a_0, b_0]$, ve kterém je nutné se pohybovat (*scheduling horizon*). Navíc musíme specifikovat dobu trvání cesty mezi jednotlivými zákazníky, jelikož pracujeme s časem, a ne jen se vzdáleností. VRPTW je pak charakterizováno následujícími omezeními [25]:

- Řešení je neplatné, pokud je zákazník obsloužen po horní hranici jeho časového okna.
- Vozidlo, které dorazí k zákazníkovi před spodní hranicí jeho časového okna, je nuceno čekat.
- Každá trasa musí začít a skončit během časového intervalu určeného skladem.
- V případě volných (*soft*) časových oken, je možné obsloužit zákazníka později, ale k optimalizační funkci je v takovém případě přičtena penalizace.

Cílem optimalizace je minimalizovat počet vozidel a součet cestovního a čekacího času potřebného pro obsloužení všech zákazníků v daných časových intervalech. Využití této varianty můžeme nalézt u kurýrních společností či klasickém zásobování s definovanými časy dodávky.

VRP s vyzvednutím a dodávkou

VRP with Pick-Up and Delivering (VRPPD) definuje úlohu, kde každému zákazníkovi i je přiřazeno množství d_i a p_i, reprezentující požadavek na dodání, respektive vyzvednutí dané komodity. Je tedy nutné zajistit, aby se po celou cestu do vozidla zboží vešlo, což typicky činí plánování komplikovanější a je nutné zvýšit počet vozidel, či prodloužit délku trasy. Typicky uvažujeme zjednodušenou situaci, kdy všechno zboží určené na dodání vychází ze skladu a všechno zboží vyzvednuté také končí ve skladu. Není tedy umožněna výměna zboží přímo mezi zákazníky [25]. Řešení je pak platné, pokud během celé cesty není překročena kapacita vozidla Q.

Mírně odlišnou variantou je VRP s nakládáním zezadu (*VRP with Backhauls*), kde jsou použita nákladní auta, která se nakládají zezadu, a tak není žádoucí, aby se nakládala dříve, než budou plně vyloženy. V této variantě je tedy prezentována podmínka, že všechny dodávky musí být uskutečněny před tím, než může být vyzvednuto jakékoliv zboží [25].

VRP s rozdělením dodávky

Poslední zajímavou variantou je tzv. *Split Delivery VRP* (SDVRP), která umožňuje, aby byl zákazník obsloužen více vozidly, pokud to sníží celkovou cenu. Toto je velmi důležité, především pokud se poptávky zákazníků svou velikostí blíží kapacitě vozidel. Typicky je řešení možné dosáhnout rozdělením poptávek na menší, dále nedělitelné části, které jsou pak postupně vozidly odebírány [25].

Formulace úlohy použité v této práci

Na závěr připomeňme definici úlohy probírané dále v této práci jako VRP. Nechť platí notace definované v úvodu 1.1.2, tedy V je množina zákazníků, c_{ij} cena přesunu mezi zákazníkem i a j, $d(v_k)$ označuje kapacitní požadavek klienta k a R_i značí trasu provedenou jedním vozidlem, přičemž počet tras (či vozidel) je m. Dále nechť δ_i označuje délku obsluhy zákazníka i a $Cost_{max}$ definuje maximální cenu trasy. Potom je délka trasy $Cost(R_i)$ definována jako součet všech cen cest mezi zákazníky a délek obsluhy, viz (1.6). Dále definujme kapacitu vozidla Q a součet kapacitních požadavků na trase R_i značený $D(R_i)$, viz (1.8).

Celkově musí u řešení S složeného z jednotlivých tras R_i platit, že každý zákazník je navštíven právě jednou, délka trasy vozidla nesmí překročit stanovený maximální limit a požadavky trasy nesmí překročit kapacitu vozidla, tedy:

$$\forall i \in \{1, ..., m\} : Cost(R_i) \leq Cost_{max}, \tag{1.10}$$

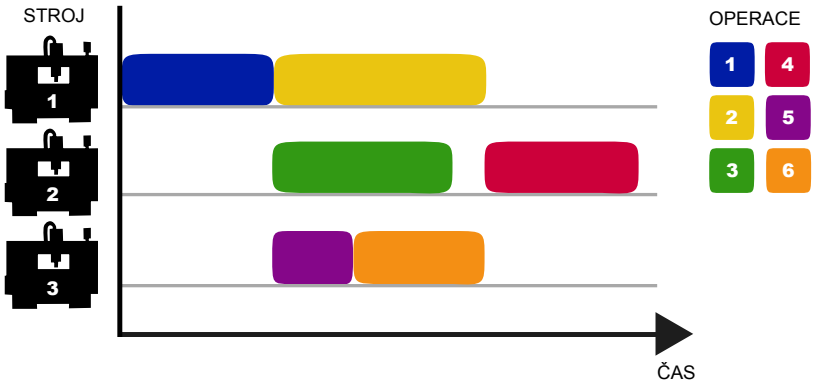
$$\forall i \in \{1, ..., m\} : D(R_i) \leq Q. \tag{1.11}$$

Pak je celková cena řešení S (výsledek hodnotící funkce $F(S)$) dána vztahem $Cost(S) = F(S) = \sum_{i=1}^{m} Cost(R_i)$.

1.2 Resource-Constrained Project Scheduling Problem

Resource-Constrained Project Scheduling Problem (RCPSP), tedy problém rozvrhování výrobních operací, někdy také označován jako *Job Shop Scheduling*. V této úloze přiřazujeme definované operace jednotlivým zdrojům tak, aby celková doba trvání projektu byla co nejkratší. Uplatnění úlohy je tak především v plánování průmyslové výroby.

Některé principy použité při řešení této úlohy je možné převzít i pro VRP, čímž se zabývá kapitola 4. Z tohoto důvodu je zde úloha detailně popsána a jsou uvedeny všechny potřebné notace.



Obrázek 1.3: Modelový příklad úlohy Job Shop Scheduling.

Formálně je úloha s N operacemi $j_1, ..., j_N$ definována jako struktura $G = (J, P, R, c, d, r)$ kde [16]:

- $J = \{j_0, j_1, ..., j_N, j_{N+1}\}$ je množina nepřerušitelných operací, kde j_0 označuje startovací a j_{N+1} pak ukončovací operaci.
- $P \subseteq J \times J$ je relace předcházení operací. $P_i = \{j \in J | (j, i) \in P\}$ nechť je množina operací, které musí být dokončeny před tím, než může začít operace i.
- $R = \{r_1, ..., r_K\}$ je množina K zdrojů.
- $c : R \to \mathbb{N}$ udává kapacitu jednotlivých zdrojů.
- $d : J \to \mathbb{N}$ přiřazuje kladnou dobu trvání jednotlivé operace, kde $d(j_0) = d(j_{N+1}) = 0$.
- $r : J \times R \to \mathbb{N}$ specifikuje požadavek zdroje.

Dále uvažujme, že všechny operace $j \in \{j_1, ..., j_N\}$ jsou následníky operace j_0 a zároveň předcházejí j_{N+1}. Potom řešením instance problému G získáme výsledek ve formě rozvrhu S, který je definován vektorem $s(S) = (s_j)_{j \in J}$, reprezentujícím kladné startovací časy $s_j(S)$ operace $j \in J$. Dále nechť $f(S)$ je vektor času ukončení operace, kde $f_j(S) = s_j(S) + d(j)$ pro všechny $j \in J$. Rozvrh S je pak označen za validní, pokud splňuje podmínky předcházení (1.12) a kapacity (1.13).

$$\forall j \in J, \forall i \in P_j : s_j \geq s_i + d(i) \tag{1.12}$$

$$\forall k \ \in R, \forall t \in \langle 0, f_{N+1} \rangle : \sum_{j \in J : t \in \langle s_j, f_j \rangle} r(j,k) \leq c(k) \tag{1.13}$$

Nechť $M(S) = f_{j_{N+1}}(S) - s_{j_0}(S)$ označuje dobu trvání projektu (*makespan*). Potom je obvykle cílem optimalizace minimalizovat $M(S)$ daného G.

Na obrázku 1.3 můžeme vidět ilustraci problému, kde si můžeme všimnout, že některé operace musejí čekat, než jiné operace na jiném zařízení skončí. Pro jednoduchost je zdroj ilustrován jako průmyslový stroj, který má výrobní kapacitu jednu operaci.

1.3 Další kombinatorické úlohy

Vedle úloh TSP, VRP a RCPSP je možné v literatuře nalézt mnohé další, více či méně zkoumané úlohy. V této části si některé z nich stručně přiblížíme, abychom dostali ucelený obraz, čím se kombinatorická optimalizace může zabývat.

Úlohy s omezeními

Tato sada problémů, v literatuře nazývaná *Constraint satisfaction problem* [33], se vyznačuje tím, že máme definovánu množinu proměnných $X = \{X_i, ..., X_n\}$ a množinu domén $D = \{D_i, ..., D_n\}$, kde pro každou proměnnou definujeme jednu doménu hodnot, kterých může proměnná nabývat. Nakonec definujeme množinu podmínek C, které specifikují povolené kombinace hodnot v daných proměnných. Řešením je pak přiřazení hodnot proměnným tak, aby nebyla porušena žádná ze specifikovaných podmínek.

Jako příklad si můžeme uvést **problém barvení map** [33], kde uvažujeme množinu proměnných jako množinu států $X = \{CZ, SK, DE...\}$ a doménu pro všechny proměnné stejnou $D_i = \{\text{červená}, \text{zelená}, \text{modrá}, \text{žlutá}\}$. Podmínky pak specifikují, že žádné dva sousední státy nemohou mít stejnou barvu, formálně bychom tedy museli vypsat všechny dvojice sousedních států, tedy například $C = \{CZ \neq SK, CZ \neq DE...\}$.

Dalším příkladem může být také velmi známý **problém *n*-dam**, kde musíme rozmístit n dam na hrací pole o velikosti $n * n$ tak, aby se žádně dvě dámy navzájem neohrožovaly, tedy do pozice, kde žádné dvě dámy nesdílí stejný řádek, sloupec, ani diagonálu.

Problém batohu

Tento problém, v literatuře nazývaný *Knapsack problem* je specifikován úložištěm (batohem) dané kapacity C a množinou n objektů, které jsou definovány dvojicí (w_i, v_i), jež určuje váhu, respektive cenu daného objektu. Problém je potom umístit do batohu objekty tak, aby měly dohromady co největší cenu, ale zároveň aby jejich celková váha nepřekročila kapacitu batohu. Formálně je problém popsán rovnicemi (1.14) [34].

$$\begin{aligned} \text{maximalizace} \sum_{i=1}^{n} v_i x_i \\ \text{aby platilo} \sum_{i=1}^{n} w_i x_i \leq C, \end{aligned} \tag{1.14}$$

kde $x_i \in \{0, 1\}$ určuje, zda daný objekt bude vložen do batohu či nikoliv. Kromě výukových aplikací, jako je například počítání snědeného jídla s maximální energií, může mít tento problém reálné využití například také v ekonomii nebo kryptografii [32].

Plánování služeb zdravotních sester

V tomto problému, v literatuře nazývaném *Nurse Scheduling Problem* nebo také *Nurse Rostering Problem*, se zabýváme optimálním plánováním směn zdravotních sester [36]. Tento problém musí splňovat velké množství podmínek a je ovlivňován mnoha parametry.

V zásadě máme definovány zdravotní sestry a směny, kterých může být více typů, například denní a noční. Dále jsou v systému definovány nutné (*hard*) a volné (*soft*) podmínky. Řešení je pak validní jen a pouze, pokud není porušena žádná nutná podmínka a splnění volných podmínek jen dále zvyšují kvalitu daného řešení. Cílem je nalézt řešení, které splňuje všechny nutné a maximum volných podmínek. Nutné a volné podmínky pak velmi záleží na konkrétní aplikaci a například i zákonech a pravidlech dané země a instituce. Pro představu se ale může jednat o následující:

- Nutné podmínky:
  - Požadovaný počet osob na směně musí být naplněn.
  - Žádná sestra nesmí mít více směn během jednoho dne (například i noční směnu jeden den a další den denní).
  - Zaměstnanec nesmí mít naplánovánu službu, pokud má na tento den odsouhlasenou dovolenou.
- Volné podmínky:
  - Všichni zaměstnanci by měli mít vyrovnaný počet víkendových služeb v měsíci.
  - Nemělo by být naplánováno několik služeb v po sobě následujících dnech, obzvláště pokud se jedná o dvanácti hodinové směny.

Shrnutí

V této kapitole byly představeny základní kombinatorické úlohy, přičemž důraz byl kladen na definici úlohy VRP, která bude v práci dále zkoumána. Definice je uvedena v závěru 1.1.2. Dále byla představena úloha RCPSP, a to především pro zavedení notací a úvod do problému, kterým budou dále inspirovány některé metody. V závěru byly krátce přiblíženy ještě některé velmi často řešené problémy, především jako náhled na rozmanitost těchto kombinatorických úloh.

Kapitola 2

Algoritmy pro řešení kombinatorických úloh

V této kapitole bude uvedeno několik možných algoritmů, které mohou řešit kombinatorické optimalizační problémy. Ačkoliv jsou algoritmy obvykle použitelné na jakýkoli problém, potřebné vysvětlení daného algoritmu budeme v některých případech vztahovat ke konkrétní ilustrující úloze. Algoritmy pro řešení optimalizačních úloh mohou být v zásadě rozděleny na tři kategorie: exaktní (někdy také nazývané optimalizační), heuristické (neboli aproximační) a metaheuristické. Protože algoritmů pro řešení těchto úloh existuje opravdu velké množství, uvedeme si zde jen některé zástupce daných kategorií a více se zaměříme na metaheuristické metody, které jsou v práci dále rozvíjeny.

Na úvod je třeba zmínit pojem *suboptimální* řešení, se kterým se setkáváme u metod heuristických a metaheuristických. Jedná se o nejlepší nalezené řešení, u kterého nejsme schopni dokázat, zda je opravdu optimální, jelikož zatím nebyl prozkoumán celý stavový prostor. Navzdory tomu, že se v některých případech opravdu může jednat o optimum, prohlásit řešení za *optimální* jsme schopni pouze u exaktních metod.

2.1 Exaktní algoritmy

Tyto algoritmy jsou typicky matematické metody zkoumající celý stavový prostor za účelem nalezení globálního optima. Jelikož se zvětšujícím se problémem (např. více zákazníků u VRP) roste stavový prostor typicky exponenciálně, může být tento úkol velmi časově náročný, až neproveditelný (v rozumném čase). Nicméně u menších instancí problému je možné tyto metody zdárně použít a budeme mít jistotu, že jsme našli skutečné optimum.

Lineární programování je nejvíce používaná skupina exaktních metod pro řešení optimalizačních problémů. Problém řešitelný lineárním programováním je typicky definován lineární cenovou (optimalizační) funkcí, kterou chceme minimalizovat a množinou omezujících podmínek (*constraint*). Tyto podmínky jsou pak lineárními omezeními proměnných použitých v cenové optimalizační funkci. Lineární programování se podobá lineární algebře, s tím rozdílem, že zde často využíváme nerovností místo rovností [35]. Pro řešení problémů lineárního programování se často používá *simplexová metoda*, která typicky dokáže nalézt řešení v polynomiálním čase. Variantou lineárního programování, kde mohou proměnné nabývat pouze diskrétních hodnot, se pak nazývá **celočíselné programování** (*integer programming*). Takto definované problémy se pak nejčastěji řeší pomocí metody větví a mezí (*branch and bound*), nebo metodou řezných nadrovin (*cutting plane*).

Problém CVRP je v celočíselném programování formulován jako [38]:

$$\text{minimalizace} \sum_{i \in V} \sum_{j \in V} c_{ij} x_{ij} \tag{2.1}$$

za podmínek

$$x_{ij} \in \{0, 1\} \qquad \forall i, j \in V, \tag{2.2}$$

$$\sum_{i \in V} x_{ij} = 1 \qquad \forall j \in V \setminus \{0\}, \tag{2.3}$$

$$\sum_{j \in V} x_{ij} = 1 \qquad \forall i \in V \setminus \{0\}, \tag{2.4}$$

$$\sum_{i \in V} x_{i0} + \sum_{j \in V} x_{0j} = 2K, \tag{2.5}$$

$$\sum_{i \notin S} \sum_{j \in S} x_{ij} \geq r(S) \qquad \forall S \subseteq V \setminus \{0\}, S \neq \emptyset. \tag{2.6}$$

Podmínky (2.3) a (2.4) zaručují, že vždy jen jedna hrana vede k i od každého zákazníka. Podobně podmínka (2.5) určuje kolik hran vede do a od skladu, kde K označuje počet tras v řešení, respektive počet vozidel. Podmínky (2.6), nazývané *capacity-cut constraints*, zaručují dodržení kapacitního omezení vozidla. Určují, že každý řez $(V \setminus S, S)$ definován množinou zákazníků S, je protnut minimálně $r(S)$ hranami, kde $r(S)$ je minimální počet vozidel potřebných k obsloužení všech zákazníků v S [38]. Tato hodnota záleží na použité variantě VRP, konkrétně u CVRP se dá vyjádřit jako [20]:

$$r(S) = \frac{\sum_{i \in S} d_i}{Q} \tag{2.7}$$

Branch and bound

Metoda *větví a mezí* patří mezi základní exaktní metody pro řešení optimalizačních problémů. Vyhledávací strom je v této metodě tvořen dynamicky a ze začátku je tvořen pouze kořenovým uzlem. Dále máme určenu hranici (*bound*), která reprezentuje hodnotu zatím nejlepšího nalezeného řešení. V počátku algoritmu je tato hodnota v mnoha případech získána použitím nějaké jiné heuristiky [10]. V každé iteraci algoritmu je pak pro další prohledávání vybrán uzel, který reprezentuje neprozkoumanou část stavového prostoru platných řešení. Tento uzel je pak rozvětven (*branch*), čímž je tento podprostor dále rozdělen na menší podprostory. Následně jsou tyto vygenerované uzly ohodnoceny a pokud je jejich hodnota lepší, než je aktuální hranice, je toto řešení zapamatováno a hodnota hranice (*bound*) je nahrazena hodnotou takového uzlu. Pokud je hodnota naopak nižší než aktuální hranice, je celý tento uzel zahozen, a to s tím předpokladem, že žádné řešení z jeho podprostoru nebude lepší než ohodnocení jeho samého. V této redukci stavového prostoru pak spočívá optimalizace tohoto algoritmu. Algoritmus je ukončen pokud již nejsou žádné uzly k prohledávání a jako optimální řešení je prohlášeno to, které má v tuto chvíli nejlepší ohodnocení.

Kombinací této metody a metod sečných nadrovin (*cutting plane*) získáme algoritmus *branch and cut*, který se také používá pro řešení optimalizačních úloh. Zde je metoda sečných nadrovin použita pro zúžení prohledávaného prostoru pro metodu větví a mezí.

2.2 Heuristické algoritmy

Heuristické (někdy také *aproximační*) algoritmy využíváme, pokud jsou klasické exaktní metody moc pomalé, nebo nejsou schopny nalézt řešení vůbec. Heuristiky naproti tomu prohledávají jen malou část stavového prostoru, čímž sice mohou minout optimální řešení, ale rapidně je zvýšena rychlost. Tyto metody tak obecně poskytují pouze suboptimální řešení, avšak i to může být přínosné a hlavně jej dostaneme v rozumném čase. Heuristiky mohou být použity samostatně, ale velmi často nacházejí uplatnění například v kombinaci s níže popsanými metaheuristickými algoritmy pro zvýšení jejich efektivnosti. Heuristiky jsou často specificky navržené pro konkrétní problémy, případně třídy problémů, proto si zde uvedeme jen základní postupy použitelné pro grafové úlohy jako je TSP nebo VRP. Ty pak můžeme rozdělit na dva základní typy, a to *konstruktivní* (*tour construction*), které vytvářejí nové řešení postupným přidáváním uzlů, a *vylepšující* (*tour improvement*), které jsou schopny vylepšit stávající platné řešení [19].

Konstruktivní algoritmy

Algoritmus nejbližšího souseda

V této metodě, v originále *nearest-neighbour algorithm*, je platné řešení postupně tvořeno tak, že se v každém kroku rozhodneme pro aktuálně nejvýhodnější variantu pokračování. Například v *TSP* zvolíme náhodně první město a poté zkoumáme všechny zbývající města a vybereme vždy to, které je k tomu aktuálnímu nejblíže. Toto opakujeme pokaždé s posledním vloženým městem a nakonec se vrátíme na začátek. Takové uvažování nemusí být z dlouhodobého hlediska nejvýhodnější, nicméně algoritmus je takto velmi jednoduchý, rychlý a produkuje platná řešení. V případě VRP by byl princip podobný, jen bychom museli brát v potaz i skutečnost, zda další nejbližší zákazník nepřekročí kapacitu vozidla nebo maximální délku trasy.

Algoritmy vkládání

Tato kategorie popisuje velké množství algoritmů, které fungují následujícím způsobem. Nejprve je vytvořena cesta jen ze dvou vrcholů. V dalším kroku jsou pak vždy uvažovány ještě nepoužité vrcholy, které jsou postupně vkládány s ohledem na specifikované kritérium. Toto může být například vrchol s nejkratší vzdáleností od prozatímní cesty, vrchol, který je nejdále, vrchol který svírá největší úhel s danou cestou a podobné [19].

Vylepšující algoritmy

k-opt algoritmus

Tento jednoduchý algoritmus je určen k vylepšování již existujícího, platného řešení, a to v následujících dvou krocích [19]:

1. Mějme vstupní platné řešení.
2. Odstraň k hran z řešení a vyzkoušej spojit k vzniklých cest všemi možnými způsoby. Pokud některou z kombinací vznikne řešení, které je lepší než to vstupní, uvažuj toto řešení jako vstupní a opakuj *Krok 2*. Ukonči pokud již nedošlo k žádnému zlepšení.

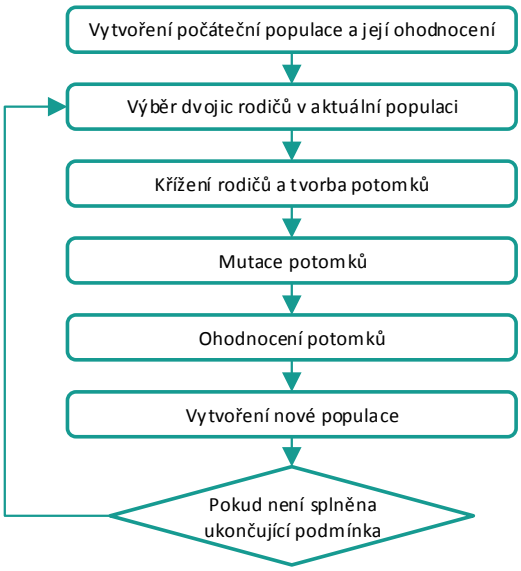
Velmi známou a často používanou variantou tohoto algoritmu je *2-opt* a *3-opt*, díky kterým je možné se zbavit různých křížení v trase, a tím ji vylepšit. Tyto algoritmy jsou však poměrně výpočetně náročné, kde jejich složitost roste s počtem vrcholů n a odstraňovaných hran k. Časová složitost je konkrétně $O(n^k)$.

2.3 Metaheuristické algoritmy

Tyto algoritmy můžeme chápat jako jakési zastřešující algoritmy, které uvnitř svého fungování mohou používat více různých principů a heuristik. Metody jsou často inspirované přírodou a i když jejich fungování není v mnoha případech matematicky dokázáno a závisí ve velké míře na náhodě, v posledních desítkách let se ukazuje, že poskytují slibné výsledky a alternativu ke klasickým, exaktním přístupům. Oproti samostatným heuristikám jsou schopny prohledat větší část stavového prostoru, avšak jsou také více výpočetně náročné.

2.3.1 Genetické algoritmy

Genetické algoritmy (GA) představené Hollandem v článku *Genetic algorithms* [15] se zakládají na darwinovském principu evoluce, hledání optimálního řešení tedy probíhá formou soutěže mezi jedinci v rámci populace. Aby bylo možné posoudit kvalitu jednotlivých jedinců, porovnáváme je podle výsledku hodnotící (*fitness*) funkce. Nová generace individuí pak vzniká pomocí reprodukce a křížení, jedinci v nové populaci tak mají vlastnosti částečně zděděné po svých rodičích a částečně ovlivněné náhodnými mutacemi. Mnohonásobným opakováním tohoto procesu jsme schopni získat populaci řešení, které jsou dostatečně kvalitní.



Obrázek 2.1: Schéma genetického algoritmu.

Obecné schéma algoritmu je zobrazeno na obrázku 2.1. Nejprve je vytvořena nová populace, kde je vygenerované řešení zakódováno v *genech* daného jedince. Následně jsou jedinci ohodnoceni z hlediska kvality jejich řešení, a to hodnotící funkcí. Tato hodnota je dále použita ve fázi výběru rodičů, kde obecně platí, že rodič s kvalitnějším řešením má větší pravděpodobnost být vybrán. Tito zvolení rodiče jsou zkříženi (rekombinací jejich genů),

a tím vzniknou noví potomci. S malou pravděpodobností se pak provede u některých potomků mutace, tedy náhodná změna jejich genů. Typicky nejlepší noví jedinci nahradí část staré populace a celý tento proces se opakuje.

S ohledem na skutečnost, že je průběh algoritmu značně stochastický, dochází k tomu, že každý běh řešení je odlišný a někdy se může stát, že celá populace uvázne v lokálním minimu a výsledek je tak velmi neuspokojivý. Proto se chování genetických algoritmů obvykle popisuje statisticky z několika opakovaných běhů [17].

2.3.2 Tabu prohledávání

Tabu Search je metoda, kde v každé iteraci nalezneme k současnému řešení x několik sousedních pomocí lokálního vyhledávání. Poté je x nahrazeno vždy tím nejlepším sousedním řešením. To znamená, že i když je nové řešení horší než současné, stejně je provedeno nahrazení, čímž se opět dostáváme z lokálního minima. Aby bylo zabráněno cyklení, jsou všechna nedávno prozkoumaná řešení vložena do zakázaného seznamu (tabu), a to na určitou dobu (resp. počet iterací). Abychom nezatěžovali paměť ukládáním dlouhého seznamu celých řešení, typicky jsou ukládány jen některé jejich atributy tak, aby se daly řešení porovnat [38]. Tento základní algoritmus lze dále vylepšovat množstvím dalších funkcí a kombinovat s různými heuristikami, typicky specifickými pro danou úlohu, čímž můžeme získat poměrně silnou optimalizační metodu. Algoritmus je ale vždy v zásadě řízen těmito základními kroky [19]:

1. Mějme počáteční řešení x a nastavme seznam tabu $T := \emptyset$.
2. Nechť $N(x)$ je vygenerovaná množina sousedních řešení x. Pokud $N(x) \setminus T = \emptyset$, pokračuj na *Krok 3*. Jinak vyber nejlepší řešení y z $N(x) \setminus T$ a nastav $x = y$. Aktualizuj seznam tabu T a nejlepší známé řešení.
3. Pokud byly splněny podmínky ukončení (např. počet iterací), vrať nejlepší nalezené řešení. Jinak pokračuj na *Krok 2*.

2.3.3 Simulované žíhání

Simulované žíhání je založeno na modelu představeném Metropolisem v roce 1953, který vychází ze simulace metalurgického postupu žíhání, skládající se ze zahřátí materiálu a jeho postupného, pomalého zchladnutí. V této metodě se snažíme vyhnout uváznutí v lokálním minimu přijímáním i horšího řešení, a to především při vysoké teplotě.

V každé iteraci nalezneme pomocí lokálního vyhledávání sousední řešení y k současnému řešení x. Pokud je nové řešení y lepší než řešení x, je nové řešení přijato, tedy x je nahrazeno y. Naproti tomu, pokud je řešení y horší než x, nové řešení je akceptováno s určitou pravděpodobností, která závisí jak na rozdílu hodnot x a y, tak na aktuální teplotě. Pravděpodobnost přijetí je vyjádřena rovnicí (2.8) [5].

$$p(x, y) = \exp(\frac{F(x) - F(y)}{T}), \tag{2.8}$$

kde F je optimalizační funkce určující výkonnost daného řešení a T aktuální teplota. Celý princip pak můžeme ilustrovat algoritmem 1 [5].

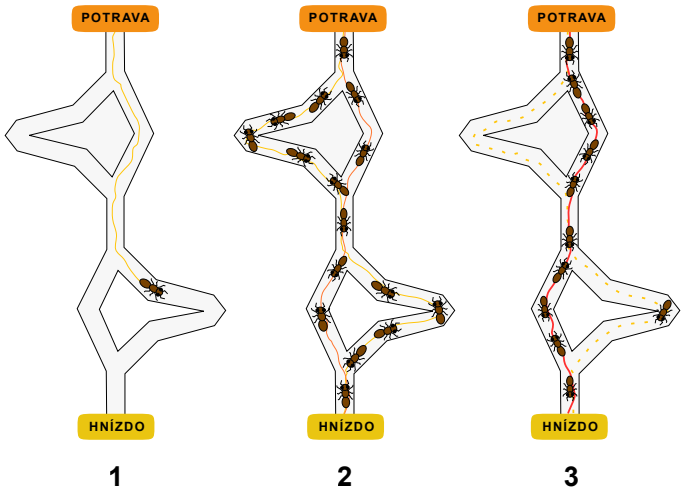
Algoritmus 1 Princip simulovaného žíhání

Vstup: Instance problému VRP
Výstup: Nejlepší nalezené řešení x_{best}

```
Inicializuj řešení x, teplotu T_0 a iteraci iter = 0
while iter ≤ maximum iterací do
    vyber náhodné sousední řešení y
    if F(y) ≤ F(x) then
        x := y
    else if p(x, y) ≤ random(0,1) then
        x := y
    end if
    if F(x) < F(x_best) then
        x_best := x
    end if
    aktualizuj teplotu T_k na T_{k+1}
    iter = iter + 1
end while
```

2.3.4 Mravenčí kolonie

Optimalizace mravenčí kolonií (*Ant Colony Optimization*) je další z úspěšných metaheuristik inspirovaných přírodou. Tato metoda je založena na pozorování reálných mravenčích kolonií hledajících potravu. Při hledání potravy mravenci značí cestu, po které jdou, vylučováním aromatické látky zvané *feromon*. Množství položeného feromonu závisí na délce dané trasy a kvalitě zdroje potravy. Tyto feromony poté poskytují informaci ostatním mravencům, jelikož ti mají tendenci sledovat trasy s největší koncentrací feromonů. Časem se trasy, které jsou kratší a vedou k lepšímu výsledku (zdroji potravy), stávají čím dál více frekventované, a tím pádem se na této trase začnou rychleji akumulovat feromony, což přitahuje více mravenců. Fakt, že mravenci jsou časem schopni nalézt kvalitnější trasy ke své potravě, můžeme využít právě k optimalizaci [5]. Princip této metody je ilustrován na obrázku 2.2, kde vidíme postupné zesilování feromonové stopy na kratší trase a přitahování více a více mravenců na tuto trasu.



Obrázek 2.2: Princip optimalizace mravenčí kolonií.

V samotném řešení pak umělí mravenci zkoumají stavový prostor, kde optimalizační funkce zastupuje kvalitu zdroje potravy a adaptivní paměť reprezentuje feromonové stopy [38]. Umělý mravenec také typicky není naprosto slepý a dokáže říct, který z následujících kroků je ten nejlepší, s ohledem na délky cest atd. Typicky je také provedena diskretizace času, kdy vždy v čase t jsou vysláni všichni mravenci, ti naleznou řešení, aktualizují feromony a poté je tento cyklus opakován v čase $t + 1$.

Algoritmus je především vhodný pro grafové úlohy (TSP, VRP) [5], které nejvíce napodobují realitu. Princip metody můžeme naznačit na problému TSP, pro který byla metoda původně navržena. Zde je každý umělý mravenec jednoduchý agent s následujícím chováním [11]:

- Další město, jenž navštíví volí s pravděpodobností, která je funkcí vzdálenosti mezi těmito městy a množstvím feromonu na jejich spojnici.
- Aby byly uskutečněny pouze validní trasy, jsou již navštívená města zakázána, tedy vkládána do tabu seznamu.
- Po dokončení trasy (navštívení všech měst), položí mravenec odpovídající množství feromonu na každou hranu své trasy.

Pravděpodobnost, že bude mravenec k pokračovat z města i do města j je pak dána rovnicí [11]:

$$p_{ij}^k = \begin{cases} \frac{(\tau_{ij})^\alpha (\eta_{ij})^\beta}{\sum_{h \in allowed_k} (\tau_{ih})^\alpha (\eta_{ih})^\beta} & \text{pokud } j \in allowed_k \\ 0 & \text{v ostatních případech,} \end{cases} \tag{2.9}$$

kde τ_{ij} značí feromonovou stopu na hraně (i, j) a η_{ij} tzv. viditelnost, kterou vyjádříme jako $1/c_{ij}$, kde c_{ij} je vzdálenost mezi městy. Seznam *allowed* je v tomto případě seznam ještě nenavštívených měst, tedy opak tabu seznamu. Pomocí parametrů α a β pak nastavujeme váhu důležitosti feromonů a vzdáleností při výběru.

Aktualizace feromonů celé populace mravenců o velikosti m je pak vyjádřena rovnicí (2.10) [11].

$$\begin{aligned} \tau_{ij}(t+1) &= \rho * \tau_{ij}(t) + \sum_{k=1}^{m} \Delta\tau_{ij}^k \\ \text{kde } \Delta\tau_{ij}^k &= \begin{cases} \frac{1}{L_k} & \text{pokud mravenec použil hranu } (i,j) \text{ ve své trase} \\ 0 & \text{v ostatních případech,} \end{cases} \end{aligned} \tag{2.10}$$

kde ρ je koeficient vypařování feromonů, který je nutný pro vyhnutí se lokálním minimům a L_k je délka trasy mravence k.

Kapitola 3

Aplikace optimalizačních algoritmů na VRP

V předchozí kapitole byla naznačena možná řešení kombinatorických optimalizačních problémů a nyní bude navržen způsob, jak některé tyto metody aplikovat na problém, kterým se tato práce zabývá, tedy problém VRP. Jelikož je tento problém tzv. *NP-hard*, bylo by exaktní řešení tohoto problému časově nepřijatelné, především pokud se bavíme o reálných systémech, kde může být počet zákazníků velmi velký. Obecně nám bude lépe vyhovovat suboptimální řešení získané v rozumném čase kombinací heuristik a metaheuristik.

V této práci budou zkoumány a následně implementovány čtyři metaheuristické algoritmy. Jedná se o genetické algoritmy (GA), tabu prohledávání (TABU), simulované žíhání (SA) a optimalizaci mravenčí kolonií (ANT). V této kapitole budou nejprve uvedeny následující obecné principy použitelné ve všech těchto algoritmech:

- Obecná tvrzení potřebná dále v textu.
- Reprezentace, neboli kódování řešení.
- Tvorba počátečního řešení.

Dále budou v kapitole postupně rozebrány algoritmy GA, TABU, SA a ANT a u každého bude diskutována aplikace na VRP. Především u komplexního genetického algoritmu budou také navržena možná vylepšení a doplňující techniky. Představené metody byly v této práci dále implementovány a v závěru bude zkoumán jejich přínos pomocí řady experimentů.

3.1 Obecná tvrzení o řešení

Pro další použití v práci budou definována následující obecná tvrzení platná pro problém VRP. Na tato tvrzení bude odkazováno dále při vysvětlování představených metod.

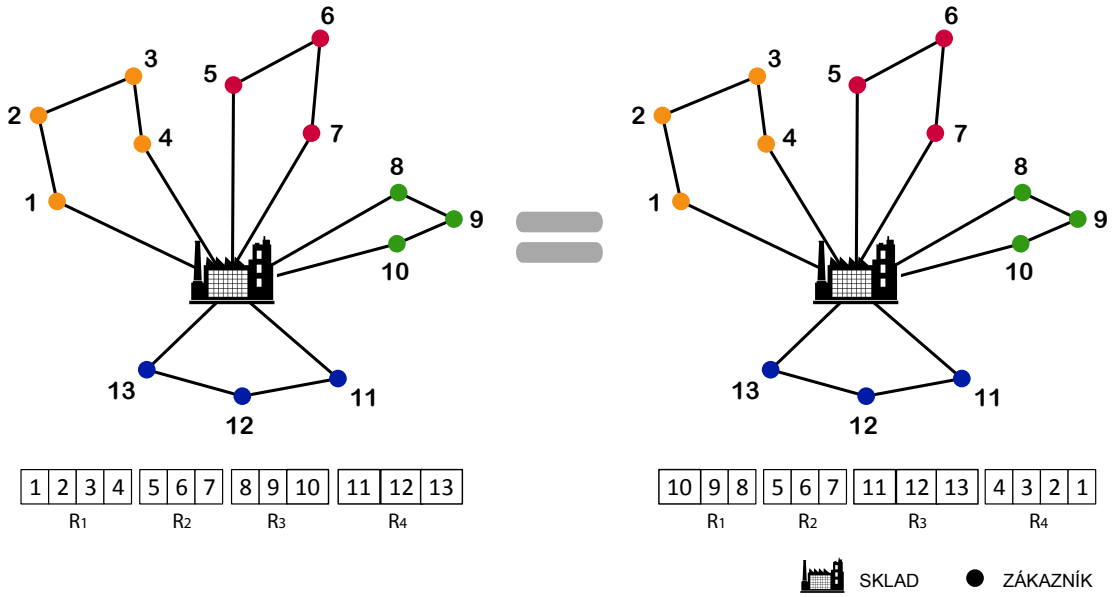
Lemma 1. *Mějme řešení S_1 s trasami R^1 a S_2 s trasami R^2, přičemž platí $S_1 \neq S_2$. Potom $Cost(S_1) = Cost(S_2)$, právě pokud platí:*

$$\forall R_i^1 \ \exists R_j^2 : R_i^1 = R_j^2 \vee R_i^1 = rev(R_j^2) \quad i \in \{1, ..., m_1\}, j \in \{1, ..., m_2\}, \tag{3.1}$$

kde operace $rev(R_i)$ označuje obrácení pořadí zákazníků v trase R_i.

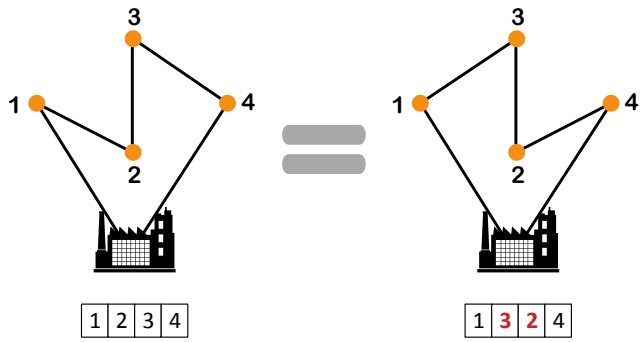
Tato lemma říká, že cena dvou různých řešení je shodná, právě pokud obsahují stejné trasy v jiném pořadí, nebo jsou některé trasy projeté obráceně.

Důkaz. Jestliže je cena hrany symetrická a tedy $c_{ij} = c_{ji}$, platí $Cost(R_i) = Cost(rev(R_i))$, kde $R_i = \{x, y, z\}$ a $rev(R_i) = \{z, y, x\}$, jelikož $c_{0,x} + c_{xy} + c_{yz} + c_{z0} = c_{0z} + c_{zy} + c_{yx} + c_{x0}$. Stejně tak nezáleží na pořadí projetých tras, jelikož celková cena je definována jako $Cost(S) = \sum_{i=1}^{m} Cost(R_i)$ a sčítání je operací komutativní. Tato tvrzení jsou také ilustrována na obrázku 3.1, kde vidíme vizualizaci trasy odpovídající uvedenému kódování. Je vidět, že trasy jsou i přes změny kódování naprosto stejné, a tak musí být stejná i jejich celková cena. □



Obrázek 3.1: Ilustrace platnosti Lemmatu 1. Vlevo cesty řešení S_1 a jejich vizualizace. Vpravo jsou trasy v S_2 jiné, ale vizualizace zůstává stejná.

Lemma 2. *Mějme řešení S. Nechť S' značí řešení vzniklé změnou pořadí zákazníků v libovolné trase R_i řešení S. Potom může dojít ke třem situacím: $Cost(S) \neq Cost(S')$; $Cost(S) = Cost(S')$, pokud je trasa R_i nahrazena za $rev(R_i)$, což dokazuje Lemma 1; a $Cost(S) = Cost(S')$, pokud jsou prohozené vrcholy stejně vzdáleny od sousedních vrcholů na trase (viz obrázek 3.2).*

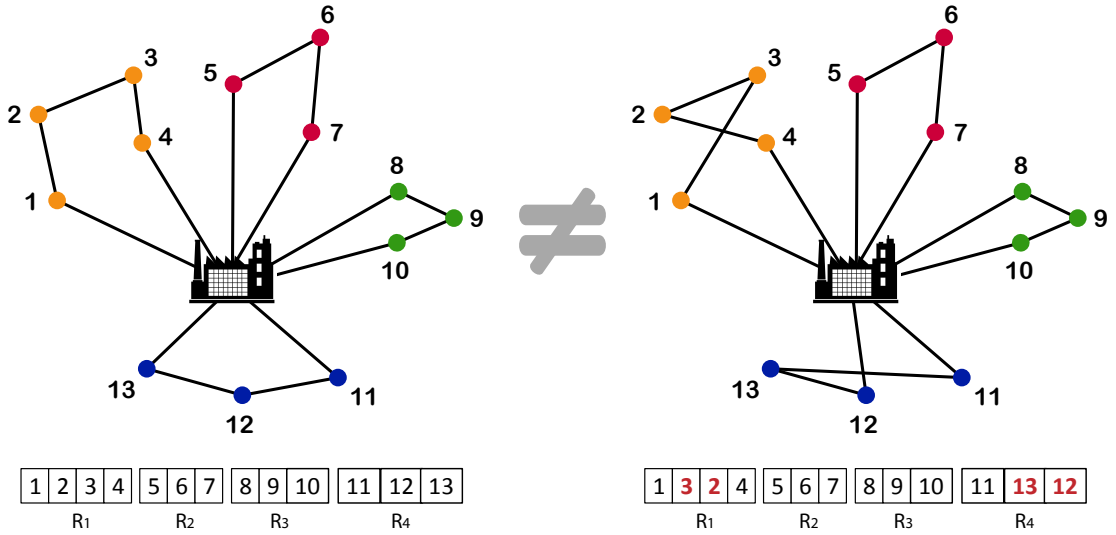


Obrázek 3.2: Ilustrace třetí možnosti zmíněné v Lemmatu 2. Přestože byly na trase prohozeni dva zákazníci, délka trasy zůstává stejná.

Důkaz. Zaměřme se na dokázání první možnosti, tedy $Cost(S) \neq Cost(S')$. Předpokládejme, že každý zákazník $i \in V$ má jiné souřadnice v prostoru. Pomineme-li výjimku uvedenou v třetí možnosti Lemmatu 2, můžeme prohlásit, že platí:

$$\forall i \in V, \forall j \in V \setminus \{i\}, \forall h \in V \setminus \{i, j\} : c_{ij} \neq c_{ih}, \tag{3.2}$$

tedy že vzdálenost z vrcholu i do jiného vrcholu je vždy jedinečná. Potom pokud v trase $R_1 = \{a, b, c, d\}$ vyměníme pozice vrcholů b, c a dostaneme $R_2 = \{a, c, b, d\}$, můžeme prohlásit, že $Cost(R_1) \neq Cost(R_2)$, jelikož $c_{ac} \neq c_{ab}$ a $c_{cd} \neq c_{bd}$. Tato skutečnost je názorně ilustrována na obrázku 3.3. □



Obrázek 3.3: Ilustrace platnosti Lemmatu 2. Vlevo cesty řešení S a jejich vizualizace. Vpravo byly trasy pozměněny prohozením zákazníků, konkrétně těch označených červeně. V odpovídající ilustraci je možné sledovat změnu délky trasy.

Tato Lemma tedy říká, že pokud změníme pořadí zákazníků, změní se také celková cena řešení. Výjimkou jsou uvedené dvě situace, kdy cena může zůstat stejná.

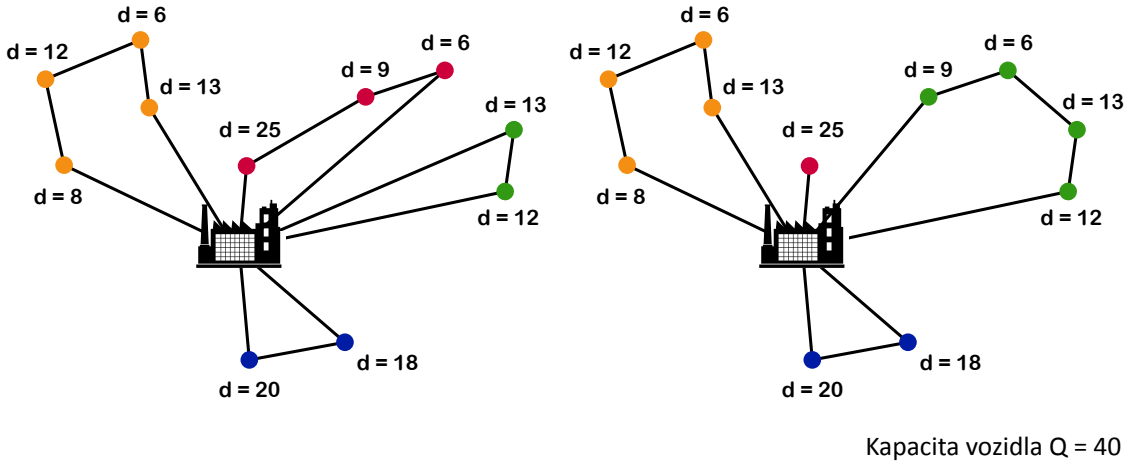
3.2 Reprezentace řešení

Typicky například u GA v základním podání máme řešení reprezentováno (zakódováno) jako binární řetězec, nad kterým jsou následně prováděny křížící a mutační operátory. Toto může být vhodné u numerické optimalizace, ale obvykle není možné tento přístup aplikovat na kombinatorické problémy. U problému TSP se obvykle setkáváme s permutačním kódováním, kde postupná čísla měst určují pořadí jejich průchodu. Nad tímto kódování pak musíme použít specializované operátory, které vždy zachovají permutaci a nevyrábí neplatné řešení.

U problému VRP navíc musíme zakódovat také informaci o několika cestách, tedy o tom kdy se vozidlo vrací do skladu. Jednou z navrhovaných možností je použít klasické permutační zakódování, kde jednoduše zapíšeme postupně navštívené zákazníky. Při dekódování pak postupně přiřazujeme zákazníky do cesty, a to až do chvíle, kdy není vyčerpána kapacita vozidla nebo překročena maximální délka trasy. Po vyčerpání kapacity jednoduše začneme novou trasu. Tímto způsobem vždy efektivně zaplníme kapacitu

vozidla a zajistíme platnost řešení. Výhodou tohoto přístupu může být použití klasických permutačních operátorů. Bylo však prokázáno, že v některých případech není možné tímto způsobem nalézt optimální řešení problému [41].

Například si představme situaci, kde je velmi blízko skladu zákazník, který požaduje zásilku zabírající polovinu kapacity vozidla. V klasickém permutačním kódování by vždy bylo nalezeno řešení, kdy je navštíven tento zákazník a potom ještě někteří další, kteří doplní kapacitu vozidla. Je ale možné, že je výhodnější nejprve poloprázdným vozidlem obsloužit tohoto zákazníka a pak se vydat prázdným vozidlem obsloužit ty vzdálenější. Ilustraci této situace můžeme pozorovat na obrázku 3.4, kde je u jednotlivých zákazníků uveden jejich požadavek a celková kapacita vozidla je $Q = 40$. Je jasně vidět, že se vyplatí zajet k prostřednímu zákazníkovi s požadavkem 25 zvlášť, což je uvedeno v druhé možnosti.



Obrázek 3.4: Modelový příklad ilustrující různé kódování. Vlevo permutační a vpravo kódování se sklady.

Tohoto můžeme dosáhnout jiným způsobem kódování, ve kterém se explicitně vyskytují záznamy o navštívení skladu, tedy na rozdíl od permutačního způsobu přesně popisují jednotlivé trasy řešení. Zde vždy na začátek jednotlivé cesty vložíme speciální uzel (např. 0), který indikuje navrácení do skladu a poté vkládáme postupně zákazníky této trasy. Tento způsob tak umožňuje uskutečnit i cesty, které sice nenaplní kapacitu vozidla, ale je celkově výhodné je provést zvlášť. Nevýhodou tohoto kódování je nutnost použití specializovaných operátorů, jelikož ty permutační by posunem speciálních uzlů skladů mohly vytvářet neplatná řešení.

3.3 Tvorba počátečního řešení

Ať už se budeme bavit o kterékoli metaheuristice, vždy je jejím prvním krokem tvorba počátečního řešení, od kterého se dále pokračuje. Ačkoliv se obvykle v původním návrhu algoritmů uvádí vygenerování počátečního řešení náhodně [17], často se ukazuje, že získání dobrého řešení hned ze startu, může zkvalitnit a hlavně urychlit konvergenci celého algoritmu [29]. Navíc toto řešení generujeme pouze jednou (případně celou populaci u GA), tedy nás nemusí příliš trápit případná časová zátěž. Kombinace s náhodností je však také potřeba, a to především u GA, kde je diverzita populace zásadní. Není tedy žádoucí generovat sice kvalitní, ale podobná řešení. V tomto je problém například při generování počátečního řešení nějakou heuristikou, která produkuje vždy stejný výsledek.

Nejprve si uveďme, jak by bylo možné vytvořit velmi triviální, náhodné řešení problému VRP. Jedním z možných přístupů by bylo začít řešení z uzlu skladu v_0 a vždy náhodně vybrat zatím nenavštíveného zákazníka v_i. Pokud požadavek $d(v_i)$ v součtu s požadavky na této trase nepřekračuje kapacitu vozidla, přidáme jej do aktuální cesty. Pokud by tímto zákazníkem došlo k překročení kapacity, je aktuální cesta uzavřena a zákazník je umístěn do nové trasy. Pokud je limitována délka trasy, musíme podobně kontrolovat i tuto podmínku. Je zřejmé, že tento přístup nebude vůbec optimalizovaný a bude produkovat velmi nekvalitní, nicméně platné řešení.

V této práci jsem navrhl jiný, lépe informovaný způsob, který je inspirován operátorem křížení *Best Cost Route Crossover* [26] prezentovaným dále v sekci 3.4.2. Na základě tohoto operátoru je navržen algoritmus *Best Cost Route Insertion* (BCRI), který dává základ dalším prezentovaným metodám. Funkčnost tohoto algoritmu je popsána pseudokódem 2.

Algoritmus 2 Metoda *Best Cost Route Insertion* (BCRI)

Vstup: Nekompletní řešení S a seznam zákazníků *unvisited*, které chceme do S zařadit
Výstup: Kompletní řešení S'

```
Seřaď náhodně zákazníky v unvisited
for all c in unvisited do
    for all R_i in S do
        // kontrola kapacity trasy
        if D(R_i) + d(c) <= Q then
            // zkoušení každé pozice v trase
            for p := 1 to k(i) do
                Vypočítej přírůstek Δ_p délky trasy při vložení c na pozici p v trase R_i
                // kontrola maximální délky trasy
                if Cost(R_i) + Δ_p > Cost_max then
                    Neber pozici p v potaz
                end if
            end for
        end if
    end for
    Vypočítej přírůstek Δ_0 kdy bude zákazník c umístěn sám do nové trasy
    if Δ_0 < nejmenší Δ_p then
        Přidej do S novou trasu obsahující pouze zákazníka c
    else
        Vlož zákazníka c na pozici p v trase R_i, kde byl přírůstek Δ_p nejmenší
    end if
end for
```

V algoritmu tak pro každého zákazníka vyzkoušíme všechny platné pozice a vložíme jej na tu nejlepší. Aby nebyli zákazníci zařazováni vždy ve stejném pořadí, je na začátku seznam náhodně promíchán, což vnáší do algoritmu potřebnou míru nahodilosti. U každého zákazníka je také vyzkoušena možnost vytvoření nové trasy obsahující pouze jeho, čímž jsme schopni vytvářet i nové trasy za běhu.

Pro vytváření nových řešení dáme na vstup algoritmu prázdné řešení S a seznam všech zákazníků. Algoritmus bude postupně vytvářet jednotlivé trasy až zařadí všechny zákazníky. Výsledné řešení je vždy platné, je přiměřeně kvalitní a použitím náhodného promíchání je s velkou pravděpodobností pokaždé odlišné. Toto jsou přesně vlastnosti, které vyžadujeme.

3.4 Genetický algoritmus

V této sekci budou prezentovány některé významné varianty a detaily z použití genetického algoritmu na problém VRP. V závěru budou také představeny některé možné způsoby zabránění stagnace algoritmu v lokálním minimu. Podkapitola je rozdělena na následující sekce:

- strategie výběru rodičů pro křížení,
- křížení těchto rodičů a tvorba potomků,
- mutace jedinců,
- vylepšení zavedením mutace části populace před křížením,
- zamezení klonů v populaci,
- myšlenka restartu stagnující populace,
- celkový finální algoritmus.

Na začátek je důležité uvést několik notací. Populací GA je Π o velikosti σ jedinců. Dále jako hodnotící (*fitness*) funkci chápeme již definovanou funkci $F(S)$, tedy celkovou cenu řešení S. Zde platí, že čím menší hodnota funkce, tím kvalitnější je řešení.

3.4.1 Výběr rodičů

Jak již bylo naznačeno, výběr rodičů, nazývaný také *selekce*, musí imitovat přirozený výběr, kde má silnější jedinec větší šanci úspěchu. Kombinací silnějších jedinců bychom měli dostat kvalitnější potomstvo, tedy novou populaci. Zároveň však musí být selekční mechanismus zvolen tak, aby byla v populaci zachována dostatečná rozmanitost. Pokud budeme vybírat vždy jen ty nejlepší jedince, za několik generací budou v populaci velmi podobní potomci, což může vést k předčasné konvergenci, tedy uváznutí v lokálním minimu. Šanci výběru tedy musí dostat i slabí jedinci. Naopak ale nesmí být selekční mechanismus příliš volný, jelikož by proces postupoval velmi pomalu a bez výrazného zlepšení mezi generacemi [17]. Představíme si tři nejvýznamnější selekční mechanismy a provedeme srovnání jejich vhodnosti.

Ruletový výběr

Ruletový výběr, nebo také výběr s přímou úměrou k ohodnocení individua, je podle [17] nejčastěji využívaným selekčním mechanismem. Funguje na principu vytvoření ruletového kola, kde je každému jedinci přiřazena výseč o velikosti přímo úměrné kvalitě řešení. Větší pravděpodobnost výběru tak mají silnější jedinci, ale určitou šanci mají i ti slabší. Formálně je pravděpodobnost výběru jedince S_i v populaci o velikosti σ dána vztahem [17]:

$$p_i = \frac{F(S_i)}{\sum_{j=1}^{\sigma} F(S_j)} \tag{3.3}$$

Potom stačí při každém výběru rodiče vygenerovat náhodné číslo $r \in \langle 0, 1 \rangle$ a vybrat i-tého jedince právě když platí vztah [17]:

$$\sum_{j=1}^{i-1} p_j < r \leq \sum_{j=1}^{i} p_j \tag{3.4}$$

Jelikož by nebylo efektivní stále počítat sumy pravděpodobností, typicky zavádíme kumulativní ohodnocení, které je pro každého jedince definováno jako součet předchozích pravděpodobností včetně své vlastní. Poté stačí náhodné číslo postupně porovnávat s tímto kumulativním ohodnocením.

Výběr podle pořadí

Pokud se v populaci vyskytují jedinci s ohodnocením mnohem vyšším, než je průměr populace, jsou v předešlém ruletovém výběru neúměrně zvýhodněni. Jakmile se takoví jedinci rozšíří, totálně začnou dominovat celému výběru a jedinci s malou kvalitou budou vytlačeni, což může opět vést k uvíznutí v lokálním minimu. Aby byl vliv nadprůměrných individuí potlačen, byl Bakerem [3] navržen přístup, kdy velikost výseku nepřidělujeme podle kvality jedince, ale podle jeho pořadí (tzv. *rank selection*). Zde se individua v populaci seřadí vzestupně podle jejich ohodnocení a toto pořadí je využito při samotném výběru. Zatímco mezi ohodnocením jednotlivých jedinců mohou být velké rozdíly, tímto uspořádáním docílíme rovnoměrnému rozdělení, kde pravděpodobnost výběru i-tého (podle pořadí) jedince je dána vztahem [17]:

$$p_i = \frac{i}{\sum_{j=1}^{\sigma} j} = \frac{2i}{\sigma(\sigma + 1)} \tag{3.5}$$

Tento mechanismus nejenže potlačuje vliv nadprůměrných individuí, ale současně automaticky napomáhá udržovat selekční tlak v pozdější fázi výpočtu, kde jsou rozdíly mezi ohodnoceními jedinců velmi malé. Nevýhodou naopak může být situace, kdy v populaci nejsou žádní výrazní jedinci a rozptyl je nepatrný. To vede k velkému rozdílu v prioritě výběru, i když je rozdíl mezi nejlepším a nejhorším jedincem velmi malý [17].

Turnajový výběr

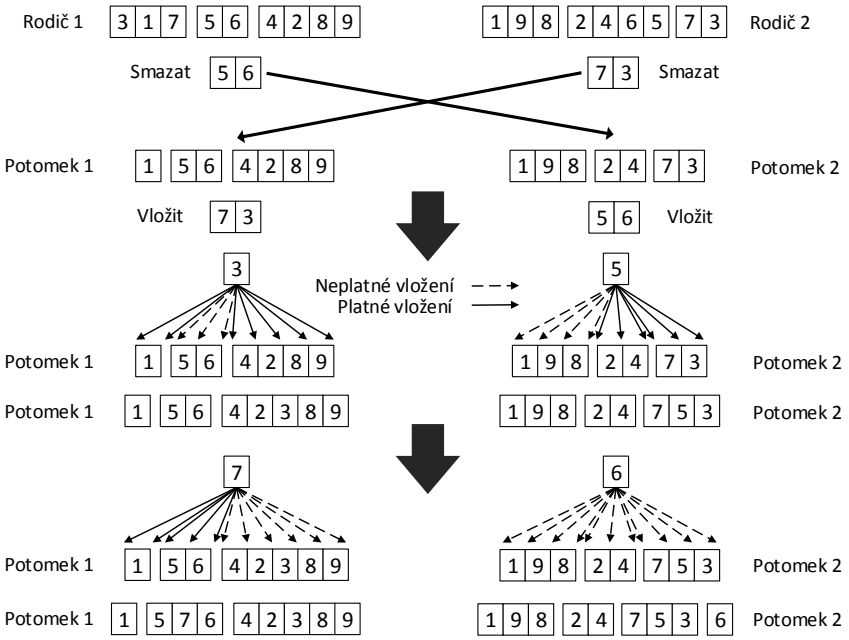
Poslední metoda výběru je nejvíce inspirována procesy v živé přírodě, kde se jedinci často musí fyzicky utkat o účast v reprodukčním procesu. Tato metoda se nazývá turnajovým výběrem (*tournament selection*). Zde se z populace náhodně vybere $k \geq 2$ jedinců, kteří se utkají v turnaji, kde zvítězí vždy ten s nejvyšším ohodnocením. Ve většině aplikací je obvykle použito $k = 2$ (*binární turnaj*) [17]. Goldberg a Deb [13] analyzovali tento způsob výběru a navrhli jeho vylepšenou verzi, která s malou pravděpodobností (např. 5%) dává šanci slabšímu jedinci porazit silnějšího protivníka.

Výběr nejvhodnějšího selekčního mechanismu se může lišit v závislosti na problému, který právě řešíme a také dalších specifikách použitého algoritmu. Vždy je tak vhodné provést řadu experimentů s aktuálním algoritmem pro zjištění optimální metody. Toto bude diskutováno v experimentální části práce.

3.4.2 Křížení

Operátory křížení obecně berou na vstup dvě rodičovské řešení a jejich kombinací vytvoří dva potomky (v některých případech jednoho), kteří dědí vlastnosti těchto rodičů. Jak již bylo naznačeno v 3.2, pro permutační kódování existují permutační operátory (např. *OX*, *PMX*). Ty však nejsou vhodné pro navržené kódování se sklady, jelikož by často docházelo k vytváření neplatných řešení. Permutační operátory totiž jen přeskládají permutační řetězec, nicméně pokud v něm jsou zavedeny i sklady, snadno dojde k vytvoření cest s překročením kapacity vozidla, případně cest prázdných. Proto musíme pro úlohu VRP s tímto kódováním použít jiné, specializované křížení.

V článku *Multi-objective genetic algorithms for vehicle routing problem with time windows* [26] byl navržen křížící operátor původně určený pro VRPTW, ovšem v článku *A meta-genetic algorithm for solving the Capacitated Vehicle Routing Problem* [41] byl tento algoritmus úspěšně aplikován také na problém VRP. Tento operátor nazvaný *Best Cost Route Crossover* (BCRC), funguje na principu vkládání na místo s nejlepší cenou. Právě touto metodou byl inspirován již představený algoritmus BCRI. Obrázek 3.5 ilustruje, jak toto křížení pracuje.



Obrázek 3.5: Princip operátoru *Best Cost Route Crossover*.

Vstupem jsou dvě rodičovská řešení a na výstup jsou generování dva potomci. Nejprve je z každého rodiče náhodně vybrána jedna trasa a zákazníci této cesty jsou následně u druhého rodiče smazány. V dalším kroku se snažíme tyto zákazníky znovu zařadit do příslušného řešení, a to tak, že pro každého z nich vyzkoušíme všechny možné pozice a vybereme tu, která má nejmenší cenu, tedy nejmenší přírůstek k celkové délce trasy. V podstatě tak na řešení aplikujeme algoritmus BCRI, kde na vstup dáme vyjmuté zákazníky.

V článku [26] je popsáno, že pokud není již žádná cesta z hlediska kapacity přijatelná, je zbývajícím zákazníkem vytvořena cesta nová. Již v rámci algoritmu BCRI bylo navrhnuto vylepšení, a to u každého zákazníka zkusit možnost vytvoření nové cesty. Pokud by toto vložení bylo cenově nejvýhodnější (i se vzdáleností zpět do skladu), danou cestu vytvoříme. Tímto můžeme vytvořit řešení, které bude podobné příkladu znázorněném na obrázku 3.4.

3.4.3 Mutace

Úkolem mutace je jednou za čas učinit v jedinci malou, náhodnou změnu. V klasickém pojetí genetických algoritmů, kde používáme binární kódování, se typicky jedná o invertování náhodného bitu. Jelikož je zakódované řešení VRP komplexnější, musí být použit sofistikovanější operátor. Nabízí se náhodná výměna zákazníků, a to jak uvnitř jedné trasy, tak mezi trasami tak, aby byla cesta proveditelná. Tyto náhodné výměny však jen velmi těžko řešení vylepší a je tak potřeba použít inteligentnější, informované metody.

Jednou z často využívaných mutačních technik je již představený vylepšující algoritmus *2-opt*. Konkrétně pro VRP tento algoritmus neaplikujeme na celé řešení, což by nám mohlo narušit podmínku kapacity, ale aplikujeme jej na každou trasu zvlášť. Pomocí tohoto algoritmu je možné z řešení odstranit křížení a tak vylepšit kvalitu jedince, s tím, že si je stále velice podobný. Metoda *2-opt* je však velmi výpočetně náročná, a to hlavně pro větší instance. Proto je vhodnější ji použít právě jako mutační operátor, než ji aplikovat na každé generované řešení.

Někdy je ale naopak žádoucí, aby byla změna více drastická a nový (zmutovaný) jedinec doznal větší změny. Toto především požadujeme pro předcházení předčasné konvergence v lokálním minimu. Pokud totiž budou v populaci jedinci již velmi podobní, opakovaná mutace pomocí operátoru *2-opt* jejich řešení již příliš nezmění a je nutné použít ještě jiný přístup.

V této práci navrhuji využít již specifikovanou metodu BCRI (viz 3.3) a modifikovat ji pro potřeby mutace. Tento algoritmus bude nazván *Best Cost Route Mutation* (BCRM) a funguje následovně:

1. Náhodně z řešení S vyjmi n zákazníků a tyto vlož do seznamu *unvisited*.
2. Spusť algoritmus BCRI s S a tímto seznamem *unvisited*.

V této metodě náhodně vyjmeme několik zákazníků a vložíme je nejlepším možným způsobem zpět. Je zřejmé, že čím bude větší hodnota n, tím větší bude změna řešení. Typicky pak hodnotu n volíme náhodně z předem specifikovaného rozsahu.

3.4.4 Mutace části populace před křížením

Mutace se zpravidla v genetickém algoritmu aplikuje pouze na nově získané potomky, ostatně tak je tomu i v přírodě. V některých algoritmech je naopak možné nalézt mutaci jedinců v populaci mimo křížení [23]. Mutace stávajících jedinců má snahu ještě více posílit diverzitu a vliv mutace na vývoj populace a může tak být přínosné tyto dva přístupy zkombinovat. Obvykle je ve zmíněné mutaci použit elitismus, tedy jistota zachování nejlepších řešení v populaci. V této práci tak budou před křížením mutováni pouze jedinci ze spodních 4/5 seřazené populace, což zabrání ztrátě nejlepších řešení. V experimentální části pak bude diskutován možný přínos zavedení této techniky a vliv počtu takto mutovaných jedinců.

3.4.5 Zamezení výskytu klonů

V populaci genetického algoritmu se často vyhýbáme vzniků klonů (identických řešení), což zaručuje nutnou diverzitu v populaci a snižuje riziko uváznutí v lokálním minimu [28]. Jednou z možností zamezení je pokaždé kontrolovat, zda nově získaný jedinec již v populaci existuje a pokud ne, můžeme jej přidat. Toto však typicky vyžaduje detailní porovnání s každým jedincem v populaci, což může být zvláště u velké populace výpočetně náročné.

Prins tak ve článku *A simple and effective evolutionary algorithm for the vehicle routing problem* [28] navrhuje u VRP použít více striktní podmínku, kde ohodnocení jakýchkoliv dvou řešení v populaci Π musí být od sebe vzdáleno alespoň $\Delta > 0$, tak jako je uvedeno v rovnici (3.6). Populace splňující tuto podmínku je pak nazvána *well spaced*, tedy dobře rozestoupená.

$$\forall S_1, S_2 \in \Pi : S_1 \neq S_2 \Rightarrow |F(S_1) - F(S_2)| \geq \Delta \tag{3.6}$$

Výhodou tohoto přístupu je skutečnost, že rozestoupenost můžeme podle Prinse [28] kontrolovat s konstantní složitostí *O(1)* následovně. Dvě řešení S_1 a S_2 jsou rozestoupené, pokud jejich celočíselné škálované ceny $\lfloor F(S_1)/\Delta \rfloor$ a $\lfloor F(S_2)/\Delta \rfloor$ jsou rozdílné. Můžeme definovat binární vektor U, kde $U(\varphi) = 1$, pokud v populaci Π existuje řešení se škálovanou cenou φ. Nové řešení K pak může být přidáno do Π, pouze pokud $U(\lfloor F(K)/\Delta \rfloor) = 0$, což zjistíme v konstantním čase. Poté co je přidáno, musí být $U(\lfloor F(K)/\Delta \rfloor)$ nastaveno na 1.

Tento způsob tak přináší výkonnou možnost detekovat klony, ale je nutno říci, že musíme velmi vhodně zvolit velikost mezery Δ, tak aby nedocházelo k přílišnému zahazování třeba i kvalitních řešení. Obecně však u problému VRP při změně kódování dochází k poměrně velké změně hodnotící funkce v řádu desetin či jednotek a mělo by tak s malou hodnotou Δ možné tuto techniku použít. Naopak by bylo velmi problematické použití u problémů, kde se funkce mění jen velmi málo a bylo by zahazováno velké množství jedinců.

Kontrolu klonů definujme funkcí *spaced*(A), která vrací 1 (*true*) pokud v populaci A platí podmínka (3.6) a 0 (*false*), pokud je podmínka porušena. V případě použití přímé detekce klonů by pak vracela funkce 1, pokud populace neobsahuje klony, a 0 pokud ano.

3.4.6 Restart stagnující populace

Restart je další technikou, kterou se snažíme vyhnout uváznutí v lokálním minimu zvýšením diverzity jedinců v populaci. V určitých časových intervalech nahrazujeme populaci nově vygenerovanými jedinci, čímž vnášíme nový genetický materiál pro vyváznutí z lokálního optima. Abychom nepřišli o nejlepší nalezená řešení, typicky nahrazujeme jen horší část populace a provádíme tak tzv. *partial restart* [37]. Tato technika je jednoduše implementovatelná do existujícího genetického algoritmu, jelikož pracuje nezávisle na něm.

Cheung a kol. [8] navrhují tato nová řešení zkřížit s jedinci současné populace, což může přinést lepší výsledky, než pouhé vložení mnohdy nekvalitních vygenerovaných řešení. Prins [28] tuto myšlenku rozvíjí u problému VRP do následující podoby, kde ρ je počet řešení, které chceme nahradit, typicky $\rho = \sigma/4$.

1. Vygeneruj novou populaci Ω obsahující ρ jedinců, kde platí, že v $\Omega \cup \Pi$ nejsou žádné klony. Poté Ω seřaď vzestupně podle hodnotící funkce.

2. Pro $k = 1, 2, ...\rho$ pokud platí $F(\Omega_k) < F(\Pi_\sigma)$, tedy pokud nejlepší nové řešení je lepší, než nejhorší v původní populaci, nahraď Π_σ za Ω_k.

3. V opačném případě zkřížíme Ω_k se všemi jedinci v $\Omega \cup \Pi$ a vybereme nejlepšího potomka C. Pokud $F(C) < F(\Pi_\sigma)$, nahradíme Π_σ za C.

V praxi musí být často vygenerováno několik populací Ω, než je dosaženo ρ nahrazení. Jelikož je toto nahrazení především u velkých instancí výpočetně náročné, v [28] Prins navrhuje limitovat nahrazení na 5 pokusů, a to i pokud poté stále není dosaženo ρ nahrazení. V porovnání se slepým nahrazením nově vygenerovanými jedinci tato technika zachovává nejlepší řešení a také nikdy nedojde ke zhoršení toho posledního.

3.4.7 Finální genetický algoritmus

Genetický algoritmus implementující všechny představené techniky a vylepšení nazveme BCGA (*Best Cost Genetic Algorithm*). Tento algoritmus je popsán pseudokódem 3, kde $spaced(A)$ značí funkci definovanou v 3.4.5.

Algoritmus 3 Metoda BCGA

Vstup: Instance problému VRP a konfigurace algoritmu.
Výstup: Nejlepší nalezené řešení Π_1

```
Π := [ ]
for k = 1,..,σ do
    S := vygeneruj nové řešení pomocí BCRI
    Π ← S
end for
seřaď seznam Π vzestupně podle hodnotící funkce jedince
iter := 0
while iter < iterMax do
    // mutace jedinců v populaci
    for i = 1,..,μ do
        náhodně vygeneruj k z ⟨σ/5, σ⟩
        M := mutace jedince Π_k
        if spaced(Π \ Π_k ∪ M) then
            Π_k := M
        end if
    end for
    // křížení
    for j = 1,..,o do
        selekčním mechanismem vyber dva rodiče P_1 a P_2
        křížením P_1 a P_2 získej potomka C
        náhodně vygeneruj k z ⟨σ/2, σ⟩
        if random < p_m then
            M := mutace jedince C
            // Když není mutant klon, nahradíme jím potomka
            if spaced(Π \ Π_k ∪ M) then C := M end if
        end if
        if spaced(Π \ Π_k ∪ C) then
            Π_k := C
            znovu seřaď Π
        end if
    end for
    if (iter mod četnost částečného nahrazení) = 0 then
        algoritmus částečného nahrazení popsaný v 3.4.6
    end if
    iter := iter + 1
end while
```

Konfigurace na vstupu algoritmu určuje hodnoty parametrů BCGA. Jedná se o parametry jako je velikost populace σ, maximální počet iterací $iterMax$, počet generovaných potomků o, pravděpodobnost mutace p_m a počet mutovaných potomků před křížením μ.

Posledním parametrem je pak *četnost částečného nahrazení*, která určuje po kolika iteracích bude vždy spuštěn tento mechanismus. Optimální hodnoty parametrů konfigurace budou diskutovány v části experimentů.

3.5 Tabu prohledávání

U tohoto algoritmu popsaného v 2.3.2 je nutné definovat metodu pro lokální prohledávání, která dokáže nalézt řešení sousední k aktuálnímu. Toto se dá chápat jako mutace aktuálního řešení, jelikož mutace provádí v řešení typicky poměrně malé změny. Z toho důvodu můžeme použít pro lokální vyhledávání stejné metody jako pro mutaci v genetickém algoritmu. Tímto můžeme také snadno pomocí experimentů srovnat výkonnost jednotlivých algoritmů jako takových, jelikož budou používat stejné techniky pro lokální prohledávání.

3.6 Simulované žíhání

U simulovaného žíhání stejně jako v TABU potřebujeme lokální prohledávání pro nalezené sousedního řešení. Zde můžeme opět využít představené mutace a přidat tak SA do porovnání výkonnosti algoritmů.

V algoritmu SA se někdy v literatuře můžeme setkat s mechanismem opakovaného ohřívání (*reheat*) [1], které by mělo zabránit uvíznutí v lokálním extrému. Princip je možné ilustrovat následujícím pseudokódem:

Algoritmus 4 Princip simulovaného žíhání s ohříváním

Vstup: Instance problému VRP
Výstup: Nejlepší nalezené řešení $x_{totalBest}$

```
Inicializuj řešení x a nastav reheat = 0
while reheat < reheatMax do
    T = T_0
    while T > T_min do
        postupuj podle klasického algoritmu simulovaného žíhání
        if F(x) ≤ F(x_totalBest) then
            x_totalBest = x
        end if
    end while
    x = změna ⟨n/2, n⟩ uzlů v řešení x pro větší diverzifikaci
    reheat = reheat + 1
end while
```

V této úpravě algoritmus žíhání běží hned několikrát podle nastaveného parametru $reheatMax$. Navíc je také před každým ohříváním změněno řešení x nějakým poměrně velkým zásahem, jako je například znovu přeskládání většiny zákazníků do jiných pozic, což můžeme provést algoritmem BCRI. Touto úpravou zvládne jednoduchý algoritmus simulovaného žíhání prozkoumat větší část stavového prostoru a nalézt kvalitnější řešení.

Oproti článku *A Simulated Annealing Algorithm for the Vehicle Routing Problem with Time Windows and Synchronization Constraints* [1] byla v této práci navržena úprava, kdy běh vnitřního algoritmu žíhání není omezen počtem iterací, ale je limitován minimální hodnotou teploty T_{min}. Kombinací tohoto parametru s nastavením počáteční teploty a její změny, v podstatě také omezíme běh algoritmu na předem daný počet iterací. Můžeme však tento počet přímo ovlivňovat nastavením teploty a její změny, což dává větší smysl, nežli abstraktní volba limitu iterací.

Nevhodnou volbou limitu iterací totiž dochází v pozdějších fázích algoritmu k jevu, kdy teplota je již v řádu miliontin a méně, a algoritmus tak ztrácí své výhody, jelikož prakticky nikdy nepřijme horší řešení. Nastavení minimální teploty také vychází z praktické předlohy, kdy bychom materiál považovali za zchlazený, pokud jeho teplota dosáhne jisté hranice.

3.7 Mravenčí kolonie

V článku *Applying the Ant System to the Vehicle Routing Problem* [7] je představeno nasazení tohoto algoritmu optimalizace mravenčí kolonií na problém VRP. Jelikož je problém VRP po přiřazení zákazníků do jednotlivých tras v podstatě několik samostatných TSP, článek se tak inspiruje především aplikací představenou v původním Dorigově článku *Ant system: optimization by a colony of cooperating agents* [11]. Pro vyřešení VRP umělí mravenci postupně vytvářejí trasu tak, aby byli navštíveni všichni zákazníci. Vždy, když by vložení dalšího vybraný zákazník vedlo k neplatnému řešení z důvodů překročení kapacity nebo maximální vzdálenosti, je vložen uzel skladu, a tím započata nová trasa.

To, jak budou umělí mravenci vybírat dalšího zákazníka k navštívení, zcela závisí na hodnotě pravděpodobnosti vybrání p_{ij}^k. Možnost pro vylepšení algoritmu při řešení specifického problému VRP je tedy především v modifikaci této pravděpodobnosti. V základní verzi navržené pro TSP je uvažována pouze feromonová stopa a tzv. viditelnost, která sleduje jak moc jsou od sebe zákazníci vzdáleni. Nicméně ve VRP je důležitá nejen vzdálenost zákazníků od sebe, ale také relativní vzdálenost ke skladu. V [7] tak navrhují zavedení měřítka úspory (*savings*), které vyjadřuje vhodnost navštívení zákazníků i a j v jedné trase. Toto můžeme vyjádřit jako: $\mu_{ij} = c_{i0} + c_{0j} - c_{ij}$. Vysoká hodnota μ_{ij} pak znamená, že navštívení zákazníka j po i je dobrá volba.

Dále je v [7] představeno měřítko zohledňující dobré využití kapacity vozidla. Nechť Q_i označuje zatím použitou kapacitu vozidla po navštívení zákazníka i. Potom vysoké hodnoty měřítka $\kappa_{ij} = (Q_i + d(v_j))/Q$ ukazují dobré využití, pokud je navštíven zákazník j bezprostředně po i.

Výpočet pravděpodobnosti vybrání zákazníka j po i pro mravence k může být následně upraven na:

$$p_{ij}^k = \begin{cases} \frac{(\tau_{ij})^\alpha (\eta_{ij})^\beta (\mu_{ij})^\gamma (\kappa_{ij})^\lambda}{\sum_{h \in allowed_k} (\tau_{ih})^\alpha (\eta_{ih})^\beta (\mu_{ij})^\gamma (\kappa_{ij})^\lambda} & \text{pokud } j \in allowed_k \\ 0 & \text{v ostatních případech,} \end{cases} \tag{3.7}$$

kde parametry γ a λ určují vliv představených měřítek.

Takto upravenou pravděpodobností bychom měli algoritmus mravenčích kolonií více optimalizovat pro nasazení na specifický problém VRP a podle [7] opravdu zkvalitňují nalezené řešení. Zároveň se jedná jen o malé zásahy do algoritmu, což je vždy žádoucí. Reálný přínos těchto metrik v našem případě bude v závěrečné kapitole ověřen experimentálně.

Kapitola 4

Heuristika využívající charakteristiky předchozích řešení

Všechny doposud představené principy použité u metaheuristických algoritmů jsou silně stochastické a spoléhají především na náhodu. Toto je také jeden ze základních principů jejich fungování, ale na druhou stranu to činí hledání řešení velmi nepředvídatelné. V opakovaném experimentu tak můžeme jednou dostat výborné řešení a podruhé řešení podprůměrné. Hrubý v práci *A Dynamic Analysis of Resource-Constrained Project Scheduling Problems for an Informed Search via Genetic Algorithm Optimization* [16] představuje heuristiku, která vnáší do tohoto náhodného procesu i značnou míru determinismu, a to sledováním charakteristik již nalezených řešení a poučením se z předchozích chyb. Ačkoliv je v tomto článku heuristika použita na problém RCPSP, tato myšlenka by mohla být použitelná i v případě VRP.

V této kapitole budou nejprve představeny použité charakteristiky řešení a hypotéza vyvozující závěry z charakteristik předchozích řešení. Dále budou navrženy tři mutace postavené na základě těchto poznatků. Tyto mutace bude následně možné použít jak pro genetické algoritmy, tak pro algoritmy využívající lokální prohledávání, tedy tabu prohledávání a simulované žíhání.

4.1 Charakteristiky řešení

Charakteristiky $CH(S)$ vyjadřují vztahy mezi dvěma elementy řešení S. V problému RCPSP se jedná o vztah mezi operacemi rozvrhu, v problému VRP půjde o vztahy mezi pozicí zákazníků v cestě. Ve článku [16] jsou představeny tyto tři charakteristiky $ch(i, j)$ operací $i, j \in J$:

- *PSE* - operace i a j startují paralelně, tedy $s_i(S) = s_j(S)$.
- *FLE* - operace i skočí dříve, nebo stejně jako operace j, formálně $f_i(S) \leq f_j(S)$.
- *STL* - operace i startuje dříve, než operace j, tedy $s_i(S) < s_j(S)$.

Na první pohled je vidět, že se tyto charakteristiky velmi liší od toho, co bychom mohli použít pro problém VRP, musíme tedy přijít s jinými specifiky. Je nutné vytvořit relaci pouze mezi dvěma zákazníky, aby možností nebylo příliš mnoho a s výsledkem se lépe pracovalo.

Bylo by možné se inspirovat charakteristikou STL a sledovat, zda je zákazník i navštíven dříve, než zákazník j. Toto však není příliš vhodné, jelikož dle Lemmy 1 mohou být jednotlivé trasy zpřeházeny a přitom se stále jedná o shodné řešení. Pokud by tedy i a j byly v jiné trase, někdy by mohlo být i před j a jindy zase naopak, přičemž se řešení nemusí vůbec změnit.

Z toho vyplývá, že se spíše než na globální řešení musíme dívat na rozmístění zákazníků v rámci jednotlivých tras, jelikož změny v trase přímo ovlivňují řešení dle Lemmy 2. Mohli bychom sledovat, zda je zákazník i v rámci jedné trasy navštíven dříve než j, pokud jsou oba součástí jedné trasy. Ovšem podle Lemmy 1 je vzdálenost jedné trasy stejná nezávisle na směru projetí. U trasy stejné vzdálenosti tak může být někdy i před j a jindy naopak.

Vhodnou charakteristikou by tak mohla být sousednost dvou zákazníků na trase. V tomto případě nezáleží na směru projetí a charakteristika má také praktickou oporu v tom, že bude jistě vhodnější navštívit za sebou zákazníky, kteří jsou například blíže u sebe, než ty co jsou daleko. Druhou možnou charakteristikou pak může být, že jsou zákazníci i a j spolu ve stejné trase. Toto má velký potenciál, jelikož správné zařazení zákazníků do jednotlivých tras je klíčovým úkolem pro nalezení kvalitního řešení a tato charakteristika toto pomůže sledovat.

Můžeme prohlásit, že druhá charakteristika, dále označovaná $CLUST$, dokáže popsat, jak jsou zákazníci rozděleni do jednotlivých cest. Naproti tomu první charakteristika $NEIGH$ popisuje, jak jsou zákazníci v těchto cestách seřazeni vedle sebe. Obě charakteristiky se tak doplňují a dokáží dobře popsat dané řešení. Formálně můžeme uvedené charakteristiky definovat rovnicemi (4.1) a (4.2).

$$\forall i \in \{1, ..., m\}, R_i = (v_0, v_{i_1}, ..., v_{i_{k(i)}}, v_0) : NEIGH(v_{i_x}, v_{i_y}) \Leftrightarrow y = x + 1, \quad \text{kde } x, y \in \{1, ..., k(i)\} \tag{4.1}$$

$$\forall i, j \in V, k \ \in \{1, ..., m\} : i, j \in R_k \Rightarrow CLUST(i, j). \tag{4.2}$$

4.2 Runtime hypotéza vylučujících charakteristik

V práci [16] Hrubý definuje pojem charakteristiky vylučující existenci řešení o délce trvání (makespanu) m. Z tohoto je dále dokazováno, že pokud jsou vyloučena řešení o makespanu m, pak jsou také vyloučena všechna řešení s makespanem $m' < m$. Tato skutečnost může být využita při optimalizaci tak, že bude řešení pozměňováno, aby neobsahovalo špatné vylučující charakteristiky. Tyto charakteristiky mohou být buď předloženy na začátku algoritmu, nebo mohou být tvořeny za běhu (*runtime*).

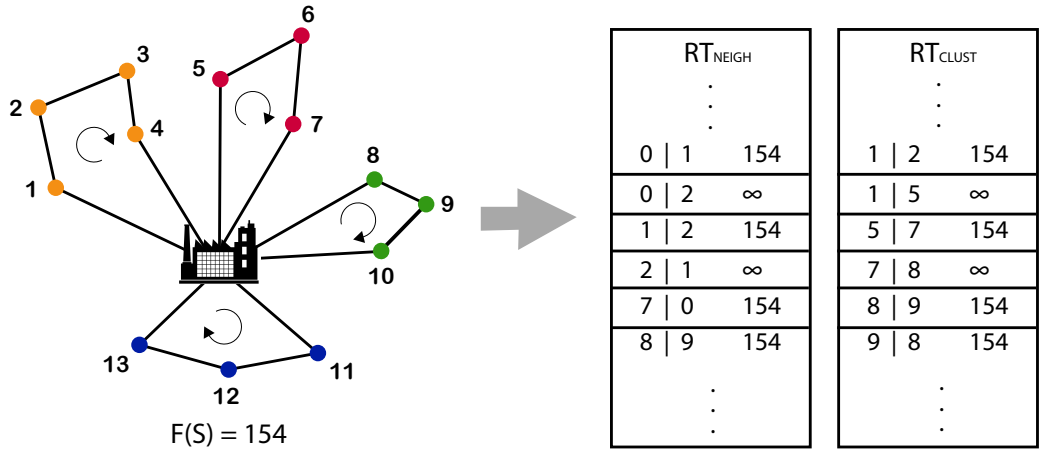
Dále uvažujme vygenerované řešení S_1 o makespanu m_1 a extrahované charakteristiky $CH(S_1)$. Je jisté, že všechny charakteristiky v $CH(S_1)$ nevylučují řešení o makespanu m_1, ale článek předkládá hypotézu, že mohou vylučovat řešení s makespanem $< m_1$. Dále mějme druhé řešení S_2 s makespanem $m_2 < m_1$. Charakteristiky tohoto řešení $CH(S_2)$ obsahují nové znalosti a průnikem s předchozími, tedy $CH(S_1) \cap CH(S_2)$ získáme hypotézu vylučující řešení s makespanem $< m_2$.

Ve článku je následně definována globální statistika nazvaná *RT system*, která definuje matici RT_{ch} o velikosti $|J| \times |J|$ pro každou charakteristiku ch. Na začátku jsou všechny prvky matice nastaveny na hodnotu ∞. Jednotlivé prvky matice $RT_{ch}(i, j)$ pak mají vyjadřovat aktuální hypotézu vylučující charakteristiky $ch(i, j)$.

U každého nalezeného řešení S s makespanem m je pak provedena aktualizace matice tak, že $RT_{ch}(i,j) := m$, pokud $m < RT_{ch}(i,j)$ a charakteristika $ch(i,j) \in CH(S)$. Poté je každá $ch(i,j)$ s hodnotou $m_b = RT_{ch}(i,j), m_b < \infty$ interpretována jako charakteristika vylučující řešení s makespanem $< m_b$.

4.2.1 Aplikace na VRP

U problému VRP nebudeme dokazovat předložená tvrzení a spíše se zaměříme na aplikaci této hypotézy a ověření její činnosti v experimentech. RT tabulky (matice) budou v tomto případě o velikosti $|n+1| \times |n+1|$, kde n označuje počet zákazníků. Hodnota $n+1$ je zvolena proto, abychom mohli v charakteristice $NEIGH$ sledovat sousednost se sklady, tedy informaci o tom, který zákazník byl v cestě první, nebo poslední. V průběhu algoritmu bude analyzováno každé nalezené řešení a extrahovány charakteristiky $NEIGH$ a $CLUST$ vyskytující se v tomto řešení. Pro každou nalezenou charakteristiku budou postupně aktualizovány dvě tabulky RT_{NEIGH} a RT_{CLUST} uvedeným způsobem $RT_{ch}(i,j) := m$, pokud $m < RT_{ch}(i,j)$, přičemž roli makespanu m zde bude zastupovat ohodnocení řešení $F(S)$.



Obrázek 4.1: Ilustrace aktualizace RT tabulek z ukázkové instance VRP.

Na obrázku 4.1 je vizualizována ukázka zápisu charakteristik řešení do RT tabulek. Zde vidíme, že zatímco relace $CLUST$ je symetrická (viz např. vztah 8 a 9), sousednost $NEIGH$ chápeme jako orientovanou. Pokud bychom v trase navštívili nejprve uzel 3 a poté 2, vzniklo by překřížení a délka trasy by tedy byla jiná.

Na obrázku můžeme také vidět význam informací v RT tabulce. Například pokud by se v nějakém řešení objevila charakteristika $CLUST(2,10)$, její hodnota v RT tabulce by byla jistě velmi špatná a tuto charakteristiku bychom chtěli vždy odstranit. Podobně se chceme vyhnout například charakteristice $NEIGH(11,13)$, jelikož v tomto případě jistě dojde k překřížení při cestě do uzlu 12. Vylučování špatných charakteristik má tedy smysl i v případě VRP.

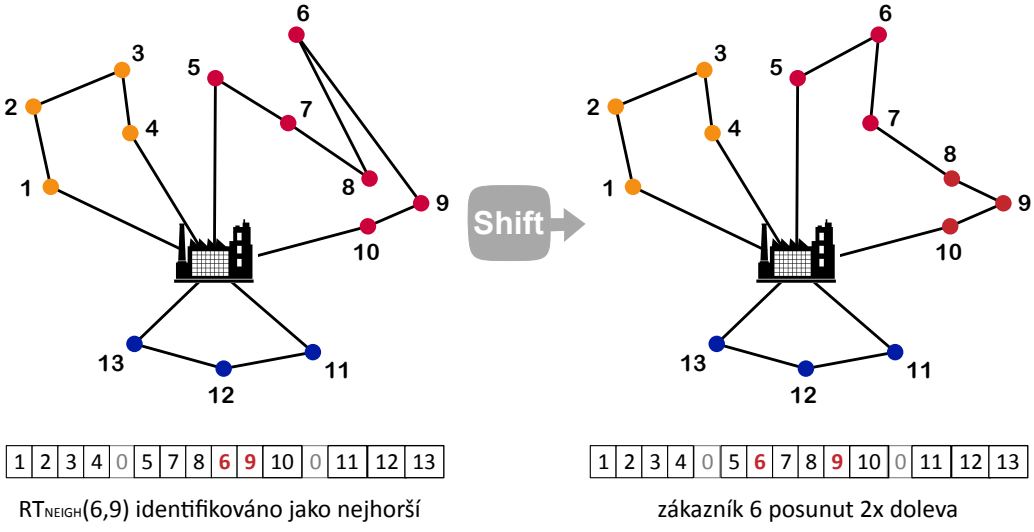
4.3 Navržené mutace

V článku [16] je optimalizace založená na informacích z RT tabulky prováděna pomocí mutace genetického algoritmu. Konkrétně je zde mutace založena na identifikaci nejhorší charakteristiky v daném řešení a jejím následném odstranění. Samotné odstranění charakteristiky $ch(i, j)$ je pak docíleno posouváním operace i nebo j v rozvrhu. Posunutím tak dostaneme jiný rozvrh, který může mít lepší, nebo také horší ohodnocení.

V samotné mutaci jsou analyzovány charakteristiky daného řešení a je vybráno m těch nejhorších, tedy s nejvyššími hodnotami $RT_{ch}(i, j)$. V práci je také diskutována možnost vybrat naopak ty nejlepší charakteristiky a ty poté odstraňovat. Tato možnost bude dále také analyzována. Pro každou operaci vyskytující se v těchto charakteristikách je následně náhodně vybrán směr posunu a operace je tímto směrem posunuta o daný počet pozic. Počet pozic pak může být limitován nějakou hodnotou, nebo určen pouze dodržováním podmínek předcházení.

4.3.1 RTShift

Na základě principu výše popsané mutace použité u problému RCPSP byla navržena mutace *RTShift* využívající informace v tabulce RT_{NEIGH}, tedy informace o sousednosti zákazníků na trase. Nejprve jsou extrahované charakteristiky $NEIGH(i, j)$ daného řešení seřazeny sestupně podle hodnoty $RT_{NEIGH}(i, j)$. Do tohoto seznamu jsou přidány pouze ty charakteristiky, jejichž hodnota $RT_{NEIGH}(i, j) < \infty$. Ze seznamu je pak vybrán požadovaný počet prvních charakteristik (těch nejhorších) a u každé je náhodně určen zákazník i nebo j. Směr posunu je stanoven tak, aby se zákazník vždy vzdaloval od svého souseda. Jelikož u charakteristiky $NEIGH(i, j)$ zákazník i předchází zákazníka j, bude při vybrání i zvolen směr vlevo a u j vpravo. Vzhledem k tomu, že jsou v RT_{NEIGH} uloženy i charakteristiky sousednosti se skladem v_0, pokud $i = v_0$ nebo $j = v_0$, je automaticky pro posun vybrán ten druhý zákazník, tak aby nikdy nebylo pohnuto se skladem jako takovým.



Obrázek 4.2: Ilustrace posunu zákazníka 6 mutací *RTShift*.

U každého vybraného zákazníka je náhodně zvolena velikost posuvu $shifts$ a to z rozsahu $\langle 0, 7 * maxShifts, maxShifts \rangle$. Konstanta $maxShifts$ určuje maximálního posun a byla nastavena na 1/5 počtu zákazníků instance. Zákazník je poté posunut o $shifts$

pozic v předem daném směru. Výjimkou je pokud by zákazník byl již posunut za hranice aktuální trasy, v takovém případě bude zařazen buď na začátek, nebo konec této trasy. Tímto je zaručeno zachování platnosti podmínek kapacity a maximální vzdálenosti trasy. Na druhou stranu je však možné touto mutací měnit pouze pořadí zákazníků v rámci jednotlivých tras a ne je přiřazovat do jiných. Vzhledem k platnosti Lemmatu 2 je však i tímto způsobem možné měnit vzdálenost daného řešení. Ilustraci funkčnosti této mutace je možné vidět na obrázku 4.2, kde posunem jednoho zákazníka dostáváme jasně kvalitnější řešení.

Použití mutace *RTShift* může být spíše vhodné pro optimalizaci pořadí zákazníků v jednotlivých trasách a metoda by měla být kombinována s jinými metodami schopnými měnit zařazení zákazníků do tras. Tuto úlohu pak může v genetickém algoritmu zastupovat křížení, popřípadě další mutace jako například dále navržená *RTCluster*.

Vylepšení metody

Základní verze metody provádí posuny zákazníků neinformovaně, a tak se může často stát, že je řešení místo zlepšení zhoršeno. Bylo by lepší nějakým způsobem určit, jak daleko, respektive na které místo je nejlepší daného zákazníka posunout. Je ale neefektivní přepočítávat délku celého řešení při každém posunutí a pak se zpětně navracet na nejlepší možnou pozici. Jinou možností je využít znalosti, které máme k dispozici v tabulce RT_{NEIGH}. Ty nám neříkají pouze jaké kombinace jsou špatné, ale také které byly nejlepší.

Prozkoumáme tedy všechny pozice *pos* zákazníka i od 1 po $shifts$ v daném směru a vždy vypočítáme hodnotu $RT(pos) = RT_{NEIGH}(h, i) + RT_{NEIGH}(i, j)$, kde h je předcházející zákazník při vložení na pozici *pos* a j ten následující. Pokud jen jedna z těchto hodnot je ∞, budeme $RT(pos)$ uvažovat jako dvojnásobek té druhé. Získané hodnoty $RT(pos)$ seřadíme vzestupně a zákazníka i vložíme na pozici *pos* s nejlepší hodnotou. Tímto zanalyzujeme všechny možné pozice a započítáním obou sousedních charakteristik získáme vždy informaci, jak by mohlo být takové vložení kvalitní.

Vezměme si situaci na obrázku 4.2. Pokud bude proměnná $shifts$ např. rovna čtyřem, zákazník 6 bude v klasické verzi algoritmu slepě posunut na začátek celé trasy, což by ale neprodukovalo velmi dobré řešení. Pomocí vylepšení jsou zanalyzovány všechny pozice mezi vrcholem 5 a 9 a je vybráno to nejlepší. Použitím tohoto vylepšení sice vzniká další výpočetní zátěž, ale to by mělo být vyváženo lepšími výsledky, což bude ověřeno v experimentech.

4.3.2 RTCluster

V další navržené mutaci bude využitá druhá charakteristika, a to pomocí tabulky RT_{CLUST}. V této tabulce jsou uloženy informace o vhodnosti umístění dvou zákazníků do stejné trasy. Cílem mutace bude identifikovat špatné kombinace a tyto zákazníky následně umístit do rozdílných cest.

S touto myšlenou je možné rozvinout mutaci BCRM představenou v 3.4.3. V této mutaci jsou náhodně vybráni zákazníci, kteří jsou z řešení vyjmuti a následně znovu vloženi nejlepším možným způsobem, tedy kontrolou všech přípustných pozic algoritmem *Best Cost Route Insertion*. Často tak dochází k přesunutí zákazníka do jiných tras, což je přesně to, čeho chceme u metody *RTCluster* dosáhnout.

Informace uložené v tabulce RT_{CLUST} následně můžeme využít pro informované rozhodnutí, kteří zákazníci budou z řešení vyjmuti. V samotném algoritmu tak nejprve zkoumáme všechny dvojice zákazníků, kteří jsou spolu umístěni v jedné trase. Tyto charakteristiky následně seřadíme sestupně podle hodnoty $RT_{CLUST}(i, j)$. V dalším kroku vybereme požado-

vaný počet nejhorších charakteristik $CLUST(i, j)$ a oba zákazníky i a j vyjmeme z řešení a vložíme do seznamu *unvisited*, který je vstupem metody BCRI.

Tímto jsou vždy vyjmuti ti zákazníci, jenž spolu vykazují nejhorší výsledky v dané trase a je velká šance, že budou přemístěni jinam. Aby se uvolnilo více míst pro vkládání těchto informovaně vyjmutých zákazníků, jsou touto metodou vybrány pouze 4/5 požadovaných zákazníků a zbytek je vybrán náhodně.

4.3.3 RTReshuff

Další navržená mutace stejně jako *RTShift* využívá charakteristiku $NEIGH$ a taktéž je určena pouze pro přeskládání zákazníků v rámci jednotlivých tras. Tato metoda nepracuje na principu odstraňování nejhorších charakteristik, ale naopak přeskládává trasy podle těch nejlepších. Ideou je rozbití a následné složení některých tras za využití znalostí uložených v tabulce RT_{NEIGH}. Samotný algoritmus pro každou rozbitou trasu R_i funguje následovně:

1. Všechny zákazníky z R_i vlož do seznamu *unvisited* a do proměnné *current* vlož uzel skladu, tedy v_0. Tento uzel také znovu vlož do R_i.

2. Pro všechny zákazníky $j \in unvisited$ spočítej hodnoty $after = RT_{NEIGH}(current, j)$ a $before = RT_{NEIGH}(j, current)$. Menší z těchto hodnot vlož spolu s j do seznamu $rtCustomers$.

3. Seřaď seznam $rtCustomers$ vzestupně a vyber zákazníka *next* náhodně z nejlepších 10 %.

4. Vlož zákazníka *next* do R_i a smaž jej ze seznamu *unvisited*. Pokud *unvisited* není prázdný, nastav proměnnou *current* na *next* a pokračuj na *Krok 2*. Jinak vlož do R_i ukončující uzel skladu.

V algoritmu tedy vycházíme ze skladu, postupně zkoumáme zbývající zákazníky a volíme souseda vždy podle nejlepší hodnoty v RT_{NEIGH}. Abychom pokaždé pouze nerekonstruovali nejlepší nalezené řešení výběrem nejlepšího zákazníka ze seznamu $rtCustomers$, je následující zákazník vždy vybrán náhodně z nejlepších 10 %, což vnáší do algoritmu i jistý prvek náhody. V algoritmu zkoumáme obě hodnoty $before$ a $after$, jelikož v praxi nezáleží v jakém směru bude trasa projeta a v minulých řešeních se tak mohl nejlepší sousední zákazník jevit jak před, tak po tom zkoumaném.

Hlavní myšlenkou tohoto algoritmu je zlepšit kvalitu řešení, které má některé trasy správně seřazené, ale u některých dochází k překřížení cest, nebo podobným problémům zvyšujícím délku cesty. Pokud jsou tyto špatné trasy vybrány pro přeskládání a v minulosti se tato cesta či její část vyskytla v jiném řešení v lepší formě, je velká šance na celkové zlepšení daného řešení. Zároveň má tato metoda potenciál jen málo často řešení zhoršit, jelikož pokud jsou mutovány trasy zatím nejlepšího nalezené řešení, je možné, že přeskládáním nedojde k žádné změně, protože se budou využívat charakteristiky právě tohoto řešení. Na druhou stranu by nemělo příliš často docházet k duplikaci řešení, čemuž má za úkol zabránit zaprvé přeskládání jen určité části tras a zadruhé zavedení náhodného výběru z nejlepších 10 % seřazeného seznamu. Nejlepší hodnoty procent mutovaných tras a procent seznamu zařazené do náhodného výběru budou zkoumány a diskutovány v experimentech.

4.4 Shrnutí

Ukládání charakteristik předchozích řešení do RT tabulek a jejich následné aktualizace představují jistou formu učení algoritmu. Zároveň jsou paměťové požadavky na uložení získaných zkušeností poměrně malé, jelikož ukládáme čísla pouze do dvou tabulek velikosti $|n+1| \times |n+1|$. Dále můžeme říci, že míra zaplnění RT tabulek dává také určitou představu o velikosti již prozkoumaného prostoru řešení [16].

V experimentální části práce budou diskutovány přínosy těchto metod oproti náhodnému přístupu a jejich možné kombinace. Také budou optimalizovány parametry metody *RTReshuff* a ověřen přínos představeného vylepšení mutace *RTShift*.

Kapitola 5

Implementace

V práci byly implementovány představené optimalizační algoritmy GA, TABU, SA a ANT s využitím poznatků prezentovaných v kapitole 3. Algoritmy byly nejprve implementovány s využitím náhodných mutací BCRM a poté byly dodány RT metody využívající znalosti z předešlých charakteristik, které byly představeny v kapitole 4. Navržené RT mutace byly použity u algoritmů GA, TABU a SA a dále v textu budou algoritmy používající RT metody označeny jako RTGA (genetický algoritmus), RTTABU (tabu prohledávání) a RTSA (simulované žíhání).

Program je spouštěn konzolově a je ovládán řadou parametrů, které budou zmíněny dále v této kapitole. Samotná implementace programu byla realizována v objektovém programovacím jazyce *Java*, přičemž je program schopen nalezené cesty vizualizovat v grafu. Pro vizualizaci byla použita knihovna *GraphStream*[1]. Výstup takové vizualizace je prezentován na obrázku 5.1. Nejlepší nalezené řešení pro zadanou instanci VRP je po dokončení běhu vypsáno na standardní výstup, případně uloženo do databáze.

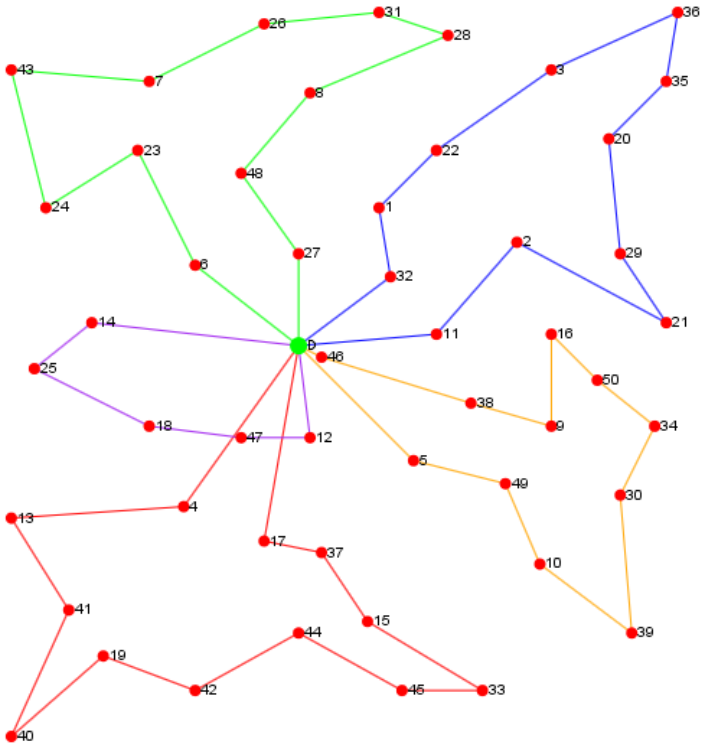
V kódu byly využity principy jazyka *Java* jako jsou abstraktní rodičovské třídy, rozhraní a generika (*generics*), a to tak, aby byla co nejvíce využita univerzálnost kódu a minimalizována jeho duplicita. Dále v této kapitole bude uveden popis použitých tříd. Kód byl také analyzován nástroji pro sledování využití pracovní paměti a procesoru a byl optimalizován tak, aby byly tyto zdroje využívány co nejefektivněji. Některé poznatky z optimalizace budou uvedeny v závěru kapitoly.

5.1 Parametry a konfigurace programu

Jednotlivé algoritmy a metody vyžadují nastavení spousty parametrů, obvykle ve formě čísel. Příkladem budiž velikost populace v GA, délka tabu seznamu či počáteční teplota algoritmu SA. Aplikace samotná pak pracuje s několika dalšími parametry, jako je název souboru instance, volba použité optimalizační metody a další. Těmito parametry je pak zvoleno jaký bude spuštěn algoritmus a jeho varianta a jakou instanci problému VRP bude řešit.

Parametry jsou v aplikaci reprezentovány statickými proměnnými, které jsou centrálně umístěny ve třídě `Config`. Tato třída je také vybavena funkcí, která zpracovává argumenty předané z příkazové řádky a nastavuje odpovídající hodnoty parametrů. Argumenty jsou pak uváděny ve formě `název=hodnota`. Takto lze beze změny kódu nastavit celý běh programu, čehož bylo využíváno při experimentování.

[1] Knihovna *GraphStream* je volně dostupná na *http://graphstream-project.org/*



Obrázek 5.1: Vizualizace grafu řešení úlohy VRP nalezené implementovanou aplikací. Čísla u zákazníků označují jejich pořadové číslo.

Jediným povinným argumentem aplikace je cesta k souboru obsahující deklaraci instance problému VRP. Formát tohoto souboru je převzat z knihovny TSPLIB, což je všeobecně používaná forma deklarace instance v logistických optimalizačních úlohách. Ze souboru jsou načteny úvodní vlastnosti instance jako je *počet uzlů*, *kapacita vozidla*, *maximální délka trasy* a *čas obsluhy*. Následuje deklarace všech zákazníků, u kterých jsou vždy uvedeny jejich souřadnice a kapacitní požadavek. Všechny tyto informace jsou třídou `Parser` extrahovány do třídy `Dataset`, která dále reprezentuje řešenou instanci VRP.

5.2 Popis tříd

V této kapitole budou stručně popsány vytvořené třídy aplikace a naznačeny jejich vazby. Třídy můžeme rozdělit na čtyři typy: řídící třídy, objektové třídy, třídy řešení a třídy metod. V této části budou postupně představeny uvedené skupiny a popsány jednotlivé jejich třídy.

Řídící třídy jsou základní třídy podporující funkcionalitu celé aplikace. Jedná se o následující:

- `DIP` - hlavní třída řídící celou aplikaci.
- `Config` - třída obsahující veškeré parametry algoritmů a aplikace. Součástí je metoda na zpracování argumentů z příkazové řádky a jejich překlad na nastavení parametrů.
- `GUI` - objekt této třídy reprezentuje vizualizaci zmíněnou knihovnou *GraphStream* včetně jejího nastavení. Obsahuje metodu pro vykreslení zadaného kódování řešení.
- `Parser` - metody této třídy provádějí načtení souboru instance do objektu `Dataset` a také zpracování vstupní RT tabulky do objektu `RTTable`.

- `Tools` - tato třída zprostředkovává různé obecné metody jako je generování náhodného čísla ze zadaného rozsahu, nebo zpřístupnění vzdáleností mezi zákazníky a parametrů použité instance VRP.
- `DatabaseHandler` - statické metody v této třídě jsou zodpovědné za uložení výsledků běhu do databáze. To je především nutné pro vyhodnocování experimentů.

Objektové třídy zde nechápeme jako čisté POJO[2], ale jako běžné objekty, které se v aplikaci vytvářejí a používají. Ty jsou následující:

- `Customer` - objekt reprezentující jednoho zákazníka nesoucí všechny jeho informace.
- `Dataset` - tento objekt reprezentuje načtenou instanci úlohy VRP a všechny její parametry jako seznam zákazníků, kapacitu vozidla apod.
- `Coding` - objekty této třídy tvoří kódování řešení a v podstatě reprezentují jednotlivé řešení. Jedná se o rozšíření třídy seznamu `ArrayList<Customer>`. Obsahuje metody pro zjištění délky zakódovaného řešení a další.
- `Cluster` - potomek třídy `Coding`, který reprezentuje jednu trasu řešení. S tímto objektem se pracuje v některých metodách a jeho instance jsou následně sloučeny do kompletního kódování objektu `Coding`.
- `RTTable` - objekt reprezentující RT tabulku, kde nezáleží na použité charakteristice, jelikož struktura obou je stejná. Obsahuje metody vypsání tabulek a přístupy k jednotlivým prvkům.
- `RTSelectable` - instance tohoto objektu jsou používány u RT mutací pro vybrání charakteristik pro jejich následné seřazení a zpracování. Obsahují tak zákazníka, hodnotu RT charakteristiky a některé další údaje.

Třídy řešení jsou jádrem celé aplikace, jelikož v nich probíhá samotný výpočet optimalizace. Jednotlivé tyto třídy pak reprezentují použité algoritmy a jedná se o následující:

- `AbstractSolver` - abstraktní třída řešení poskytující základ ve formě inicializací matice vzdáleností a RT tabulek a metody pro aktualizaci RT hodnot. Specifikuje abstraktní metodu `solve()`, ve které je prováděn samotný výpočet. Následující třídy pak dědí z této třídy.
- `GeneticSolver` - tato třída obsahuje jádro genetického algoritmu a potřebné metody.
- `TabuSolver` - v této třídě je prováděn výpočet algoritmu TABU.
- `SimulatedAnnealingSolver` - jádro algoritmu simulovaného žíhání.
- `AntSolver` - tato třída popisuje chování algoritmu ANT, přičemž využívá objekty mravenců třídy `Ant`.

Třídy metod obsahují hlavní logiku představených mutací, křížení apod. Jedná se o statické třídy obsahující soběstačné metody navracející požadovaný výsledek. Konkrétně se jedná o následující:

[2]POJO, neboli *Plain Old Java Object*, je jednoduchý objekt typicky obsahující pouze proměnné a metody přístupu k nim. Obvykle neobsahuje další výkonnou logiku.

- `Initiator` - třída obsahující statické metody navracející počáteční řešení.
- `Mutation` - jedna z nejrozsáhlejších tříd obsahuje všechny statické metody mutace.
- `Crossover` - v této třídě jsou statické metody provádějící křížení.
- `General` - zde jsou uvedeny obecné metody použité na více místech, konkrétně algoritmus BCRI.
- `Ant` - objekt umělého mravence zkoumajícího prostor řešení metodou `walk()`.

5.3 Optimalizace

Při implementaci je také nutné brát ohled na výkonnost aplikace, a tak byly použity různé optimalizace, z nichž některé budou v následujícím textu zmíněny.

Optimalizace výpočtu délky a kapacity trasy

Všechny algoritmy a metody velmi často pracují s ohodnocením daného řešení S reprezentovaného objektem `Coding`. Bylo by tedy velmi výpočetně náročné pokaždé znovu počítat toto ohodnocení, tedy délku trasy řešení. Proto je délka vždy uložena do pomocné proměnné (*cache*) a je tedy počítána vždy jen jednou. Samozřejmě ale musíme tuto cache zneplatnit, pokud dojde k jakékoliv změně řešení. Je tak třeba přetížit všechny modifikační metody třídy `ArrayList` jako `add()`, `remove()` atd.

Dále u objektu `Cluster` použitého v některých metodách využíváme výpočet kapacity této trasy. Ten může být uložen obdobně jako celková délka. Navíc však můžeme při přidávání a odebírání zákazníků z trasy přímo měnit hodnotu této proměnné, a to přičtením a odečtením požadavku daného zákazníka. I takto malá optimalizace tak může pomoct k celkovému zrychlení, pokud se operace provádí velmi často.

Optimalizace metody *RTClust*

Běh programu byl také analyzován pomocí nástrojů pro sledování využití pracovní paměti a procesoru a zjištění problémových míst. Bylo zjištěno, že u metody *RTClust* je generováno obrovské množství objektů třídy `RTSelectable`, které musí být následně setříděny podle hodnoty, což spotřebuje mnoho výpočetního času. To je z toho důvodu, že v této metodě zkoumáme všechny dvojice zákazníků, kteří jsou spolu v jedné trase. Již u nejmenších instancí se pak jedná přibližně o 300 dvojic. Je jisté, že všechny musíme prozkoumat, ale není nutné všechny ukládat do seznamu pro seřazení. To zvyšuje jak paměťové, tak výpočetní nároky na seřazení.

Toto se dá optimalizovat následovně. Víme, kolik charakteristik chceme vybrat a také víme, že chceme vybrat ty nejhorší. Můžeme tak ukládat informaci o minimální uložené hodnotě RT. Pokud zkoumaná charakteristika má lepší ohodnocení, je jasné, že již nebude vybrána, a tak ji ani nemusíme přidávat. Naopak ji přidáme, pokud je horší, než minimum. Abychom našli alespoň předepsaný počet charakteristik, spouštíme tyto kontroly až po naplnění této kvóty. Tímto je délka seznamu, nutná pro seřazení, zmenšena na pouhé 2 % původního počtu. Toto dokáže mutaci *RTClust* podstatně zrychlit.

Implementace rulety

Zajímavá je také implementace rulety u ruletového výběru. Toto je používáno jak v genetických algoritmech, tak u optimalizace mravenčí kolonií. Klasickým způsobem bychom museli pro každého jedince vypočítat hodnotu pravděpodobnosti a vložit do seznamu. Poté bychom museli seznam projít znovu a vydělit všechny pravděpodobnosti jejich součtem. Ve stejném kroku bychom mohli počítat také kumulativní pravděpodobnost, se kterou se dále lépe pracuje. Následně vygenerujeme náhodné číslo a seznam postupně procházíme, než narazíme na hodnotu menší, než je vygenerované číslo. Odpovídající záznam reprezentuje výsledek. Zejména u mravenčí kolonie, kde tuto operaci provádíme opravdu velmi často u každého dalšího zákazníka, je tento systém velmi neefektivní.

Je tak v rámci optimalizace možné vytvořit pole o potřebné velikosti a do něj postupně vložit jednotlivé pravděpodobnosti, a to kumulativně, tedy stálým přičítáním pravděpodobnosti k součtu. Tyto hodnoty již nemusíme škálovat celkovým součtem, protože poměr mezi hodnotami by zůstal zachován i po vydělení všech stejnou hodnotou. Je tedy nutné jen upravit rozsah generovaného náhodného čísla po poslední kumulativní pravděpodobnost. Poté je náhodné číslo v poli vyhledáno pomocí optimalizované metody `Arrays.binarySearch()`, čímž ušetříme další průchod polem. Takto se dají jednoduše ušetřit téměř dva zbytečné průchody celým seznamem, ten pro dělení součtem a ten pro vyhledání.

Obecné principy optimalizace

V rámci algoritmu se vyskytuje poměrně často seřazení seznamu. Ať už seznamu kódování v populaci, nebo právě u RT mutací. Pro řazení je na všech místech použit optimalizovaný přístup poskytnutý metodou `Collections.sort()`, která má velmi dobrou lineárnělogaritmickou složitost $O(N \cdot log(N))$. Tato metoda vyžaduje implementaci rozhraní `Comparable<>`, kterým určíme, podle jaké hodnoty se má řazení provést.

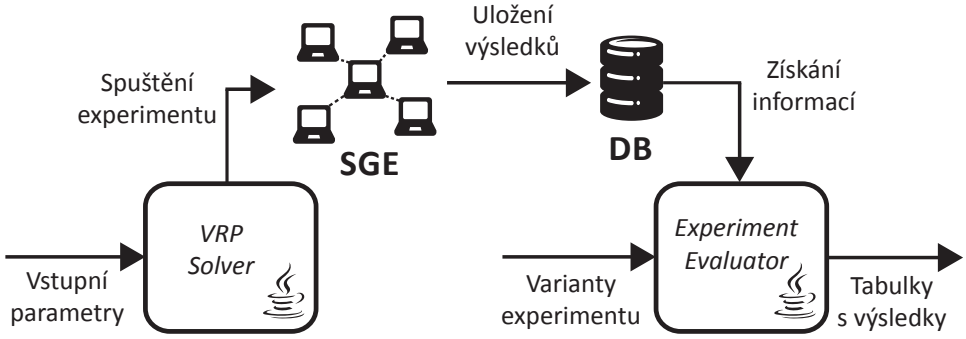
V rámci implementace je na velkém množství míst použit seznam, který je následně procházen. Procházení seznamu je v Javě možné hned několika způsoby: pomocí iterátoru, klasickým cyklem `for` nebo `while` a nebo zkráceným cyklem `for` (*for-each*). Poslední způsob procházení je nejrychlejší a zároveň také nepřehlednější. Tam, kde je to možné použijeme tento způsob.

Kapitola 6

Experimenty

V této části bude nejprve definováno na jakých úlohách a jakým způsobem experimenty probíhaly a byly vyhodnocovány. Dále budou pro srovnání uvedeny vybrané výsledky z literatury používající stejné srovnávací úlohy. V práci byla provedena řada experimentů s implementovanými algoritmy GA, TABU, SA a ANT. Každému algoritmu bude věnována jedna podkapitola, kde bude několika experimenty zjištěno optimální nastavení parametrů těchto metod a ověřen přínos představených vylepšení. V závěrečné části bude nejprve diskutováno nastavení metod založených na RT tabulkách a poté zkoumán celkový přínos těchto mutací pro jednotlivé algoritmy.

Pro experimentování byla vytvořena *experimentální infrastruktura* naznačená obrázkem 6.1. Všechny experimenty jsou hromadně konzolově spouštěny na školním distribuovaném systému *Sun Grid Engine* a po skončení každého běhu jsou do databáze zapsány důležité údaje jako: nalezené řešení a jeho vzdálenost, čas dokončení, délka trvání, použitý algoritmus a jeho varianta a parametry spuštění programu. Pomocí těchto informací je pak databázovými dotazy možné získat všechny potřebné statistiky pro následné porovnávání. Pro tento účel byl vytvořen vyhodnocovací program *Experiment Evaluator* v jazyce *Java*, který dokáže pro zadané varianty algoritmu vypsat srovnávací tabulky tak, jak je možné vidět dále ve všech tabulkách v této kapitole. Navrženou infrastrukturou bylo významně usnadněno provádění experimentů a jejich vyhodnocování.



Obrázek 6.1: Ilustrace použití experimentální infrastruktury.

6.1 Popis problémů pro srovnávání

Všechny experimenty byly prováděny na dvou sadách *euklidovských* srovnávacích problémů, kde je vzdálenost mezi dvěma body daná klasickou *euklidovskou vzdáleností* těchto bodů. Pozice zákazníků je zde určena dvojicí souřadnic v prostoru. Souřadnice samotné pak nemají uvedenou jednotku, proto dále budeme značit vzdálenost námi zavedenou abstraktní jednotkou délky, značenou *JD*.

První sada obsahuje 14 klasických problémů vytvořených Christofidesem a kol. [9]. Tyto problémy obsahují 50 až 199 zákazníků, jedná se tedy spíše o menší instance. Instance 6 - 10, 13 a 14 mají navíc definovánu maximální délku jedné trasy, která nesmí být překročena. Tyto mají zadán také tzv. *service time*, tedy čas obsluhy, který bude stráven u každého zákazníka a se kterým se musí počítat při dodržování maximální délky trasy. Čas obsluhy je zde chápán jako dálka, kterou by vozidlo ujelo za daný čas, a je tak možné jej jednoduše sečíst s délkou trasy. V této sadě pak platí, že problémy 1 - 5, 11 a 12 a 6 - 10, 13 a 14 jsou co se týče struktury shodné, jen se liší použitím podmínek maximální délky trasy. Dále platí, že zákazníci jsou v instancích rozmístěni náhodně a to kromě 11 - 14, kde jsou umístěni do shluků.

Golden a kol. [14] později usoudili, že je potřeba větších instancí a jejich sada obsahuje 20 rozsáhlých problémů, které obsahují 200 až 483 zákazníků. Pouze instance 1 - 8 mají definovánu maximální délku jedné trasy, ovšem bez definice servisního času, v tomto případě se tedy bere v potaz jen délka samotná. V této sadě jsou zákazníci uspořádání do pravidelných geometrických útvarů. Zatímco v problémech 1 - 8 jsou zákazníci uspořádáni do soustředných kruhů kolem skladu, u instancí 9 - 12 jsou uspořádány v pravidelném kosočtverci se skladem umístěným na spodním vrcholu a u 13 - 16 se jedná o pravidelný čtverec se skladem uprostřed. V poslední části sady, tedy v instancích 17 - 20, jsou pak zákazníci uspořádáni do tvaru pravidelné šesticípé hvězdy kolem centrálního skladu.

Instance	Uzlů	Kapacita	Max. délka [JD]	Čas obsluhy [JD]	Nejlepší známé řešení [JD]
1	51	160	∞	0	524,61
2	76	140	∞	0	835,26
3	101	200	∞	0	826,14
4	151	200	∞	0	1028,42
5	200	200	∞	0	1291,29
6	51	160	200	10	555,43
7	76	140	160	10	909,68
8	101	200	230	10	865,94
9	151	200	200	10	1162,55
10	200	200	200	10	1395,85
11	121	200	∞	0	1042,11
12	101	200	∞	0	819,56
13	121	200	720	50	1541,14
14	101	200	1040	90	866,37

Tabulka 6.1: Vlastnosti instancí srovnávací sady Christofides.

V tabulkách 6.1 a 6.2 je možné vidět vlastnosti jednotlivých instancí, kde jsou ve sloupcích postupně *číslo instance*, *počet uzlů*, *kapacita vozidla*, *maximální délka trasy* a *čas obsluhy*. V posledním sloupci nalezneme *nejlepší známé řešení*, což je délka nejlepšího publikovaného výsledku dané instance. V případě první sady se jedná o optimální řešení získané exaktními algoritmy, avšak v druhé sadě se jedná jen o suboptimální řešení získané různými výzkumnými skupinami. Tyto hodnoty byly převzaty z aktualizovaného webu zabývajícího se úlohou *Vehicle Routing Problem* [25].

Instance	Uzlů	Kapacita	Max. délka [JD]	Čas obsluhy [JD]	Nejlepší známé řešení [JD]
1	240	550	650	0	5627,54
2	320	700	900	0	8444,50
3	400	900	1200	0	11036,22
4	480	1000	1600	0	13624,52
5	200	900	1800	0	6460,98
6	280	900	1500	0	8412,80
7	360	900	1300	0	10181,75
8	440	900	1200	0	11643,90
9	255	1000	∞	0	583,39
10	323	1000	∞	0	741,56
11	399	1000	∞	0	918,45
12	483	1000	∞	0	1107,19
13	252	1000	∞	0	859,11
14	320	1000	∞	0	1081,31
15	396	1000	∞	0	1345,23
16	480	1000	∞	0	1622,69
17	240	200	∞	0	707,79
18	300	200	∞	0	997,52
19	360	200	∞	0	1366,86
20	420	200	∞	0	1820,09

Tabulka 6.2: Vlastnosti instancí srovnávací sady Golden.

6.2 Popis experimentování

Jelikož v následující části bude uvedeno poměrně velké množství experimentů, bylo nutné vyvinout jednoduché a stručné srovnávací kritérium. Jelikož se práce zabývá stochastickými algoritmy, je nutné provést průměrování z mnoha běhů algoritmu. Konkrétně byl program vždy spuštěn s uvedeným nastavením na všech instancích obou sad, a to v počtu 50 jednotlivých spuštění na každou instanci. V programu je pro generování náhodných čísel použit objekt `java.util.Random`, který podle dokumentace nastavuje seed (počáteční stav) tak, že je velmi pravděpodobně odlišný při každém volání konstruktoru objektu. Tímto je zaručena unikátnost každého spuštění. V každém běhu pak byla vyhodnocena nejlepší nalezená cesta a pak byly statisticky vypočítány hodnoty průměrné a nejlepší trasy u každé jednotlivé instance. Pro lepší vypovídající schopnost byla vypočítána odchylka od nejlepšího známého řešení podle vzorce:

$$\omega = \frac{c_{FOUND} - c_{BKS}}{c_{BKS}} * 100\ [\%], \tag{6.1}$$

kde c_{FOUND} značí vzdálenost námi nalezeného řešení a c_{BKS} vzdálenost nejlepšího známého řešení.

Pro lepší srovnání jednotlivých experimentů je vhodné vypočítat průměrnou hodnotu této odchylky přes celou sadu problémů. Takto bude možné srovnat dvě varianty algoritmu v dané sadě problémů pouze pomocí dvou čísel: průměrné odchylky průměrného řešení (ω_{avg}) a průměrné odchylky nejlepšího řešení (ω_{best}). Tento způsob vyhodnocení pro následné porovnání je v komunitě *VRP* poměrně běžný a je možné se s ním setkat například v [28], [24] nebo [7]. Všechny odchylky jsou uvedené v procentech a značí tak o kolik procent je dané řešení horší, než optimum, potažmo nejlepší nalezené řešení.

Dále je u experimentování důležité zavést srovnávací kritérium, pomocí kterého je limitován běh různých algoritmů a jejich variant. Tento limit bude omezovat běh algoritmu tak, aby všechny varianty měly stejnou šanci nalézt řešení. V komunitě kolem VRP bohužel není ustálené žádné takové kritérium. Wink a kol. v [41] limitují každý běh vygenerováním 100 000 potomků, zatímco například Berger v [4] zastaví běh, pokud nebyl výsledek v posledních 20 generacích vylepšen alespoň o 1 %. Můžeme se také inspirovat v komunitě kolem RCPSP, kde je běžným měřítkem počet vyhodnocení řešení [16, 40], v našem případě zjištění vzdálenosti jednotlivého řešení. Toto kritérium je vhodné pro porovnávání různých metod (GA, TABU, SA atd.), jelikož všem dává stejný počet prozkoumaných řešení. Zmíněné měřítko také správně reflektuje větší potřeby složitějších metod s návratem, kdy je v jedné chvíli prozkoumáno více řešení, což by mělo být při porovnání experimentů zohledněno. Pokud není uvedeno jinak, u všech experimentů byl nastaven limit 1 000 000 (1M) vyhodnocení.

Jelikož se práce především zabývá implementací RT metod do jednotlivých algoritmů, byly všechny experimenty (až na uvedené výjimky) provedeny s použitím těchto metod. Bylo tím eliminováno riziko, že by byly jednotlivé parametry algoritmů vyladěny pouze pro náhodné mutace a při použití pokročilých RT metod by došlo k degradaci. Tímto mohou být v závěru náhodné mutace lehce znevýhodněny, ale rozdíly v optimálním nastavení parametrů by neměly být tak velké. Konkrétně byly představené RT metody použity v poměru: 30 % *RTReshuff*, 25 % *RTShift*, 25 % *RTCluster* a 20 % náhodné *BCRM*.

Statistický rozptyl ve výsledcích

Jelikož jsou všechny zkoumané algoritmy založeny na náhodných jevech, může být každý běh odlišný, jak již bylo řečeno. Průměrováním padesáti běhů tyto odchylky částečně eliminujeme, ale je důležité uvést, že i výsledky opakované sady běhů se mohou lišit. To, jak moc se budou lišit můžeme nazvat statistickým rozptylem. Tato informace bude užitečná ve zhodnocení experimentů, kde budeme vědět, kdy je rozdíl natolik velký, že je opravdu způsoben lepším chováním algoritmu a ne jen touto odchylkou.

Proto bylo provedeno 10 opakování stejného experimentu za účelem zjištění těchto hodnot. Je nutno podotknout, že jako jeden experiment je považováno padesáti násobné spuštění algoritmu na každou instanci obou sad. Konkrétně byl spuštěn algoritmus RTGA s velikostí populace 30 jedinců. Z deseti získaných hodnot byl spočítán průměr a také odchylka minimální hodnoty od průměru (*záporná odchylka*) a odchylka maximální hodnoty (*kladná odchylka*). Uvedena je také klasická směrodatná odchylka σ. Výsledky jsou uvedeny v tabulce 6.3.

	Christofides		Golden	
	ω_{avg} [%]	ω_{best} [%]	ω_{avg} [%]	ω_{best} [%]
Průměrná hodnota	7,354	4,1	22,174	16,415
Záporná odchylka	-0,074	-0,19	-0,124	-0,415
Kladná odchylka	+0,066	+0,28	0,146	0,295
Směrodatná odchylka σ	0,033	0,166	0,097	0,218

Tabulka 6.3: Vyhodnocení statistického rozptylu pro vypočítanou hodnotu odchylky průměrného a nejlepšího řešení.

Z výsledků absolutních odchylek můžeme vyvodit, že u ω_{avg} může dojít ke kolísání hodnoty cca. $\pm 0,07$ %, resp. $\pm 0,13$ % u druhé sady, zatímco u ω_{best} je to přibližně $\pm 0,23$ %, resp. $\pm 0,35$ %. Pravidlo 2σ, které udává, že valná většina hodnot je od průměru vzdálena maximálně 2σ, toto potvrzuje s hodnotami $2\sigma_{\omega_{avg}} = 0,066$ %, respektive $0,194$ % u první sady a $2\sigma_{\omega_{best}} = 0,332$ %, respektive $0,436$ % u sady druhé. Větší rozptyl u nejlepšího řešení je poměrně pochopitelný, jelikož se jedná vždy jen o jednu hodnotu z celého experimentu, a tak není tato hodnota obecně velmi směrodatná a spíše budeme dále porovnávat hodnoty ω_{avg}. Můžeme si také všimnout přibližně třikrát větší odchylky u druhé sady s rozsáhlejšími instancemi. To může být způsobeno větší náchylností na náhodné jevy při větším počtu zákazníků a také celkově větší odchylkou od optima.

Závěrem můžeme vyslovit tvrzení, že pokud budou výsledky ω_{avg} dvou experimentů od sebe vzdáleny alespoň 4σ, tedy přibližně **0,14 %**, respektive **0,39 %**, je možné tvrdit, že je prokazatelně lepší ten s menší hodnotou. V opačném případě si nemůžeme být jisti, zda je rozdíl daný lepšími vlastnosti algoritmu, nebo pouze statistickým rozptylem.

6.3 Výsledky z literatury

Pro porovnání je třeba uvést výsledky dosažené v literatuře různými metodami, abychom získali přehled, jaké hodnoty můžeme očekávat. Pro klasické sadě *Christofides* bylo nalezeno více publikovaných článků. Proto jsou uvedeny jak základní a již poměrně staré heuristiky a metody, tak relativně nově publikované články s výkonnými metodami. Rozsáhlejší druhé sadě problémů se pak věnuje menší část článků a jedná se především o optimalizované metody pro rozsáhlé instance z poslední doby. Popis použitých zkratek algoritmů spolu s citací daných článků je v následujícím seznamu:

- **CW** - *Clark and Wright savings algorithm.* Výsledky převzaty z [12].
- **MJ** - *Mole and Jameson generalized savings algorithm.* Výsledky převzaty z [12].
- **GM** - *Gillett and Miller Sweep algorithm.* Výsledky převzaty z [12].
- **CMT1** - *Christofides, Mingozzi and Toth two-phase algorithm.* Převzato z [12].
- **OSA** - Algoritmus simulovaného žíhání představený Osmanem. Převzato z [12].
- **OTS** - Algoritmus tabu prohledávání představený Osmanem. Převzato z [12].
- **TABUROUTE** - Heuristika tabu prohledávání představená v [12].
- **ANT** - Aplikace algoritmu mravenčí kolonie na problém VRP uvedená v [7].
- **HG** - Hybridní genetický algoritmus představený v [4].
- **SGA** - Genetický algoritmus představený v článku [28].
- **GENPSO** - Hybridní algoritmus spojující genetický algoritmus s optimalizací hejnem částic [22].
- **AGES** - *Active-guided evolution strategies* představené v článku [24].
- **RTR** - *Record-to-record travel* je algoritmus deterministického žíhání představený Goldenem a kol. spolu s používanou sadou instancí v [14].
- **VNS** - *Variable neighborhood search* je heuristika určená především pro velké instance z [18].
- **GTS** - *Granular tabu search* představený Toth a Vigem v [39].

	CW	MJ	GM	CMT1	ANT	OSA
Průměrná odchylka [%]	11,27	10,91	7,3	4,79	4,43	2,11
Průměrný čas [min]	-	-	-	-	18,1	-
	SGA	TABUROUTE	OTS	HG	GENPSO	AGES
Průměrná odchylka [%]	0,9	0,86	0,79	0,49	0,09	**0,03**
Průměrný čas [min]	0,66	46,8	-	21,25	0,95	2,8

Tabulka 6.4: Výsledky z literatury pro sadu instancí Christofides.

Výsledky všech metod byly převzaty nebo přepočítány do hodnot průměrné odchylky od nejlepšího známého řešení, tedy hodnot dále používané v této práci. U většiny je také uveden průměrný čas v minutách, což je další běžně publikovaná míra, která je vypočítána průměrem času řešení všech instancí v dané sadě. Je nutné podotknout, že v každém článku byl typicky použit počítač s jiným procesorem a tedy jiným výpočetním výkonem, a tak jsou tyto časy pouze orientační. Použité procesory zde z důvodu zjednodušení nejsou uvedeny. Samotné výsledky jsou uvedeny v tabulce 6.4 pro první sadu, respektive 6.5 pro sadu druhou.

	VNS	RTR	GTS	SGA	GENPSO	AGES
Průměrná odchylka [%]	5,54	3,94	2,9	1,93	0,68	**0,02**
Průměrný čas [min]	0,003	37,15	17,5	31	4,2	24,4

Tabulka 6.5: Výsledky z literatury pro sadu instancí Golden.

Z uvedených údajů vidíme, že rozdíly mezi výsledky algoritmů mohou být poměrně velké. Stejně tak jsou velké rozdíly v době trvání, kdy metoda AGES dosahující nejlepších výsledků patří mezi ty spíše pomalejší. Různé práce tak zřejmě mají jiné cíle a zatím co se některé práce zaměřují na rychlost algoritmu, jiné kladou důraz na kvalitu výsledků i za cenu vysoké výpočetní náročnosti.

6.4 Genetické algoritmy

Genetický algoritmus je poměrně komplikovaná a robustní metoda, která vyžaduje správné nastavení mnoha parametrů, tak aby fungovala co nejlépe. Při každém jejím použití je tak zásadní, a zároveň komplikované, správně všechny tyto parametry optimalizovat. Tato optimalizace se neobejde bez experimentů, na kterých je nutné vyzkoušet různá nastavení všech parametrů. Podle výsledků pak můžeme zvolit jejich nejoptimálnější hodnoty pro právě řešený problém. V následujících experimentech tak budou představeny a prozkoumány různé parametry představeného algoritmu RTGA.

6.4.1 Velikost populace

Podle Alandera v článku *On optimal population size of genetic algorithms* [2], je jedním z nejdůležitějších parametrů genetického algoritmu právě velikost populace σ. V přírodě se ukazuje, že velké populace jsou více stabilní a více vzdorují evoluci, nežli ty menší [2]. Často tak v malých izolovaných populacích nacházíme velkou diverzitu, která je při optimalizaci žádoucí. Volba správné velikosti však také velmi závisí na konkrétní aplikaci, a je tak vždy důležité nalézt správnou velikost pro daný problém [31]. Protože je jádro algoritmu RTGA částečně inspirováno Prinsovým článkem *A simple and effective evolutionary algorithm for the vehicle routing problem* [28], budeme vycházet z hodnot zde uvedených. V tomto článku Prins došel k závěru, že nejlépe algoritmus pracuje, pokud je velikost populace v rozmezí od 25 do 50 jedinců. V experimentu EXPGA1 budou tedy primárně vybrány velikosti z tohoto rozsahu. Abychom si udělali představu o průběhu i mimo tento interval, byla pro porovnání zvolena také velikost menší (10 a 15 jedinců), větší (100 jedinců) a výrazně větší (500 jedinců). Výsledky experimentu pro vybraných osm velikostí populace je možné vidět v tabulce 6.6.

Dataset	$\sigma = 10$		$\sigma = 15$		$\sigma = 25$		$\sigma = 30$	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	6,15	2,97	6,17	**2,91**	**6,06**	3,24	6,08	3,18
Golden	21,84	16,05	22,02	16,25	**21,74**	**16,08**	22,18	16,42
	$\sigma = 40$		$\sigma = 50$		$\sigma = 100$		$\sigma = 500$	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	6,13	3,28	6,16	2,83	6,80	3,74	10,22	6,46
Golden	22,15	16,59	22,64	16,58	24,63	18,77	33,39	27,21

Tabulka 6.6: EXPGA1 - Vliv velikosti populace σ na výsledky algoritmu RTGA.

Z výsledků je možné vidět, že obecně dosahují lepších výsledků menší populace. Pokud se podíváme na větší populace $\sigma = 100$ a $\sigma = 500$, vidíme, že se výsledky s velikostí populace zhoršují. Naopak si můžeme všimnout, že u velmi malých populací $\sigma = 10$ a $\sigma = 15$ nedochází k téměř žádnému zhoršení kvality řešení. Obecně tak můžeme potvrdit předpoklad z článku [28], s tím, že v našem případě by bylo možné použít i menší populace. Nejlepší výsledky z tohoto intervalu pak především pro druhou sadu instancí podává populace $\boldsymbol{\sigma = 25}$, v dalším vývoji tak budeme pracovat s touto velikostí.

6.4.2 Volba selekčního operátoru

V sekci 3.4.1 jsou uvedeny tři možnosti výběru rodičů pro křížení: klasický ruletový výběr, výběr podle pořadí (*rank*) a turnajový výběr. Goldberg a Deb se v článku *A comparative analysis of selection schemes used in genetic algorithms* [13] zabývají důkladnou analýzou těchto tří způsobů výběru jedinců, bohužel však na úloze řešení diferenciálních rovnic. Ve svém článku došli k závěru, že klasický ruletový výběr konverguje nejpomaleji. Výběr podle pořadí a binární turnaj jsou pak na tom podle nich výkonnostně velmi podobně, přičemž upřednostňují turnajový výběr vzhledem k jeho jednoduchosti, a tedy i menší výpočetní náročnosti. Jelikož má úloha VRP jiná specifika, provedl jsem vlastní experiment s těmito selekčními operátory, abych zjistil nejlepší variantu pro tuto úlohu.

Dataset	**Ruleta**		**Rank**		**Turnaj**	
	ω_{avg} [%]	ω_{best} [%]	ω_{avg} [%]	ω_{best} [%]	ω_{avg} [%]	ω_{best} [%]
Christofides	**6,00**	**3,07**	6,25	3,56	6,12	3,36
Golden	**21,74**	**16,08**	24,16	18,22	21,90	16,16

Tabulka 6.7: EXPGA2 - Volba nejlepšího selekčního operátoru algoritmu RTGA.

Ze zjištěných dat uvedených z tabulky 6.7 můžeme říci, že metoda ruletového výběru v našem případě dosahuje o něco málo lepších výsledků, než metoda turnaje. Výběr podle pořadí (*rank*) pak vykazuje o něco horší výsledky, především u druhé sady. Jak je uvedeno v 3.4.1, výběr podle pořadí může způsobovat problémy, pokud v populaci nejsou výrazně lepší řešení a rozdíly ohodnocení jedinců v populaci nejsou příliš velké. Výběr podle pořadí pak může neúměrně zvýhodňovat řešení, která nejsou o tolik kvalitnější. Z toho je možné usoudit, že k této situaci dochází i v našem případě, a proto lépe fungují jiné operátory. Pro další experimenty bude použit **ruletový výběr**, jelikož u obou sad dosahuje nejlepších výsledků.

6.4.3 Počet potomků

Tento parametr ovlivňuje, kolik potomků bude každou iteraci algoritmu vytvořeno a zařazeno do populace. Pokud bychom v algoritmu nepoužívali jiné techniky než křížení a mutaci potomků, příliš by na tomto parametru nezáleželo, jelikož by v rámci jednoho běhu byl tak jako tak vytvořen stejný počet potomků. Samozřejmě za odlišný počet generací, ale toto v experimentech není limitováno. V implementovaném algoritmu však část populace také přímo mutujeme, proto kombinací počtu mutovaných jedinců s počtem potomků v podstatě určujeme váhu jednoho a druhého.

Dataset	$o = \mathbf{1/10}$		$o = \mathbf{1/8}$		$o = \mathbf{1/5}$		$o = \mathbf{1/4}$	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	6,73	3,30	6,45	3,31	6,17	3,00	6,26	3,15
Golden	23,89	18,03	23,14	16,82	22,13	15,96	22,02	16,27
	$o = \mathbf{1/3}$		$o = \mathbf{1/2}$		$o = \mathbf{2/3}$		$o = \mathbf{3/4}$	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	**5,99**	2,81	6,02	2,75	6,06	**2,65**	6,18	3,15
Golden	**21,62**	**15,70**	21,87	15,91	21,83	15,85	21,93	16,19

Tabulka 6.8: EXPGA3 - Volba počtu generovaných potomků o v algoritmu RTGA.

Samotný počet potomků je pak vhodné definovat poměrem k velikosti populace σ, označeným o. V experimentu bylo zvoleno několik takovýchto poměru. Je důležité říci, že v navrženém algoritmu noví potomci nahrazují jedince pouze ze spodní poloviny seřazené populace. Je tedy aplikován elitismus a pokud bude $o > 1/2$, bude do populace zařazeno pouze $\sigma/2$ nejlepších z nich. Výsledky experimentu EXPGA3 jsou uvedeny v tabulce 6.8.

Z výsledků je možné vidět, že horších výsledků je dosaženo volbou malého počtu potomků. Naopak nejlepších výsledků je dosaženo při volbě $o = 1/3$, $1/2$, nebo $2/3$. Zde se rozdíly mezi jednotlivými variantami pohybují v rámci statistického rozptylu, nicméně především u druhé sady je nejlepších výsledků dosaženo u $\boldsymbol{o = 1/3}$ s rozdílem 0,21 %. Tento počet tak zvolíme pro další experimenty.

6.4.4 Pravděpodobnost mutace potomků

V GA jsou typicky potomci mutování s určitou pravděpodobností. Velikost této pravděpodobnosti pak určuje, jak moc se účinky mutace projeví v celém běhu algoritmu. Pokud je pravděpodobnost velmi nízká, algoritmus spoléhá především na roli křížení a naopak při velmi velké pravděpodobnosti může dojít k degradaci algoritmu na primitivní náhodné prohledávání. Typicky tak nastavujeme pravděpodobnost mutace p_m do 10 %. V našem případě však neprovádíme jen náhodné, neinformované změny řešení a mutace zastupuje informované lokální prohledávání. Z tohoto důvodu by měla být pravděpodobnost mutace spíše vyšší, aby tyto metody vynikly. Prins v článku [28] využívajícím lokální prohledávání jako mutaci, navrhuje nastavení p_m až na relativně vysokých 20 %. Je také důležité zmínit, že v použitém genetickém algoritmu jsou nezávisle v každé generaci mutováni náhodní jedinci v populaci, a tím pádem je mutace použita na dvou místech, což může snižovat význam mutování potomků. Experimentálně byly prozkoumány různé nastavení od $p_m =$ 5 % po vysokých 50 %. Výsledky EXPGA4 jsou uvedeny v tabulce 6.9.

Dataset	p_m = **5 %**		p_m = **10 %**		p_m = **20 %**		p_m = **30 %**		p_m = **50 %**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	6,08	2,99	6,02	**2,73**	**5,95**	2,86	6,11	3,06	6,08	3,43
Golden	21,84	15,95	**21,79**	**15,87**	21,80	15,96	22,06	16,19	22,79	16,87

Tabulka 6.9: EXPGA4 - Vliv pravděpodobnosti mutace potomků p_m v algoritmu RTGA.

Z výsledků je možné vidět, že lepších výsledků je dosaženo u $p_m \leq 20$ %. Toto je více zřejmé u druhé, rozsáhlejší sady. Rozdíly výsledků u třech nejmenších pravděpodobností jsou velmi malé u obou sad a mohou být také způsobeny statistickým rozptylem. Jelikož navržené RT metody pracují jako mutace, je vhodné podpořit jejich vliv, a proto bude pro další experimenty zvolena vyšší $\boldsymbol{p_m = 20}$ **%**.

6.4.5 Počet mutovaných jedinců

Jak již bylo zmíněno u předchozího experimentů, v každé iteraci je před křížením zmutováno několik náhodně vybraných jedinců v populaci. Toto zvyšuje význam mutace a dále zvyšuje diverzitu populace. Následující experiment EXPGA5 zkoumá význam této modifikace a volby počtu takto mutovaných jedinců. Budou uvedeny výsledky jak bez použití této mutace, tak pro počet mutovaných jedinců μ uvedený jako zlomek velikosti populace σ. Výsledky experimentu jsou uvedeny v tabulce 6.10.

Dataset	μ = **0**		μ = **1/12**		μ = **1/8**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	6,78	3,42	5,87	2,62	5,81	2,42
Golden	21,72	16,04	21,43	15,88	21,12	15,21
	μ = **1/6**		μ = **1/4**		μ = **1/2**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	5,82	2,50	**5,80**	**2,41**	5,96	2,84
Golden	**20,81**	**14,87**	21,13	15,38	21,39	15,65

Tabulka 6.10: EXPGA5 - Volba počtu mutovaných jedinců před křížením v algoritmu RTGA.

Z výsledků můžeme jednoznačně říci, že použití této mutace zlepšuje řešení, a to přibližně o jeden procentní bod odchylky u obou sad problémů. U první sady jsou pak ve výsledcích poměrů mutovaných jedinců jen velmi malé rozdíly, až na poslední možnost $\mu = 1/2$, která je prokazatelně horší. Oproti tomu u druhé sady jasně vidíme sestupující trend z obou stran směrem k volbě $\boldsymbol{\mu = 1/6}$, která vykazuje jednoznačně nejlepší výsledky, a to i pokud přihlédneme ke statistickému rozptylu. Pro další postup tedy zvolíme tuto hodnotu.

6.4.6 Parametry částečného nahrazení

Částečné nahrazení je technika předcházení uváznutí v lokálním minimu, představená v 3.4.6. Tato metoda má dva důležité parametry. První parametr je velikost nahrazení ρ, kterou budeme uvádět ve zlomku velikosti populace σ. Tento parametr tedy určuje, jak velká část původní populace bude nahrazena novými jedinci. Druhým parametrem je pak četnost tohoto nahrazení. Tento limit bude uveden v počtu generací *RI*, po kterém se vždy metoda spustí. Dohromady oba parametry přímo ovlivňují vliv metody na běh experimentu.

Metoda částečného nahrazení je poměrně výpočetně náročná, jelikož v nejhorším případě dochází ke generování až pěti populací a mnoha křížení. Pokud bude nahrazení příliš velké a časté, bude neúměrně zpomalovat celý proces a významně vyčerpávat limit počtu vyhodnocení. Myšlenka tohoto algoritmu je ostatně dodat nové řešení ve chvíli, kdy již algoritmus stagnuje a nedokáže se posunout. Obecně by tedy měla tato technika přispět pro nalezení lepšího řešení, ale nemělo by k nahrazení docházet moc často, aby se nevyčerpávaly systémové zdroje. Prins v článku [28] a Cheung s kolektivem v [8] navrhují nahrazení 1/4 populace, přičemž se přímo nezmiňují o tom, jak často se nahrazení spouští. Pro zjištění optimálních hodnot byl proveden experiment EXPGA6 s různými kombinacemi obou parametrů a také bez použití této techniky. Výsledky jsou uvedeny v tabulce 6.11.

Dataset	**Vypnuto**		ρ **= 1/4** *RI*=**300**		ρ **= 1/2** *RI*=**500**		ρ **= 1/5** *RI*=**10 000**		ρ **= 1/8** *RI*=**10 000**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	5,81	2,73	5,82	**2,50**	6,00	3,13	5,71	2,79	**5,60**	2,55
Golden	20,44	**14,35**	20,93	14,87	21,39	15,88	20,66	15,44	**20,23**	14,88

Tabulka 6.11: EXPGA6 - Experiment zkoumajícího vliv částečného nahrazení populace v algoritmu RTGA.

Poměrně překvapivě bylo zjištěno, že výsledky bez použití této metody jsou samy o sobě relativně dobré a naopak nesprávným použitím metody dochází ke zhoršení. Toto vidíme ve zmiňovaném častém použití, konkrétně $RI = 300$ a 500. Lepších výsledků je dosaženo méně častým nahrazením spíše menší části populace, kde dochází ke zlepšení, i když ne až tak výraznému. Nejlepších výsledků je pak dosaženo poslední konfigurací $\boldsymbol{\rho = 1/8}$ a $\boldsymbol{RI = 10000}$, a to u obou sad problémů.

Z těchto výsledků můžeme usoudit, že buďto v běhu programu nedochází tak často k uvíznutí v lokálním minimu, nebo tomuto jevu tato metoda nedokáže až tak efektivně zabránit. Je také možné, že jsou nově vygenerované řešení i po křížení poměrně nekvalitní, a tak nemusí být do populace vůbec zařazeny, nebo přispějí k celkovému zlepšení jen velmi málo.

6.4.7 Zamezení výskytu klonů

V kapitole 3.4.5 byly diskutovány dva způsoby zamezení výskytu stejných řešení v populaci. Buď jsou klony detekovány přímo porovnáváním kódování řešení, nebo definováním tzv. *spacingu*, tedy mezery, kterou musí dodržovat ohodnocení jedinců v populaci. V následujícím experimentu EXPGA7 budou porovnány oba tyto přístupy spolu se systémem bez detekce klonů. Dále bude zkoumáno několik velikostí mezery Δ, což udává jak přísné toto pravidlo je. Velikost mezery by pak měla být spíše menší. Pokud bude mezera příliš velká, hrozí, že budeme vyřazovat i slibná, lehce odlišná řešení.

Prins v článku [28] uvádí rozumný rozsah jako $0.2 \leq \Delta \leq 5$ a nakonec volí hodnotu $\Delta = 0.5$. V experimentu EXPGA7 zvolíme hodnoty z tohoto rozsahu, ale pro ověření této hypotézy také hodnoty o něco menší a větší.

Dataset	**Bez detekce**		**Přímé porovnání**		**$\Delta = 0.01$**		**$\Delta = 0.1$**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	5,88	3,02	5,31	2,30	**5,18**	**2,03**	5,26	2,34
Golden	19,59	14,24	**19,26**	13,81	19,28	**13,69**	19,46	14,25
	$\Delta = 0.2$		**$\Delta = 0.5$**		**$\Delta = 1$**		**$\Delta = 5$**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	5,36	2,56	5,64	2,91	6,01	2,84	9,55	5,53
Golden	19,71	14,12	20,47	14,65	21,27	15,63	27,16	20,30

Tabulka 6.12: EXPGA7 - Experiment sledující význam použití detekce klonů v populaci algoritmu RTGA

Z výsledků uvedených v tabulce 6.12 vyplývá, že špatná volba mezery Δ přináší horší výsledky, než ponechání klonů v populaci, a to přibližně od hodnoty $\Delta > 1$. Naopak přímé porovnání řešení při detekci klonů dosahuje téměř stejně dobrých výsledků jako použití malé mezery Δ, a to především u druhé rozsáhlejší sady problémů. Nicméně se jako nejlepší volba z hlediska výsledků jeví $\mathbf{\Delta = 0.01}$, která by měla být teoreticky také výpočetně méně náročná, než přímé porovnání. Konkrétně byl průměrný čas u této varianty 0,9, respektive 6,6 minut a u přímého porovnání 0,95, respektive 6,9 minut. Nejedná se tedy o tak velké rozdíly, což je zřejmě způsobeno malou použitou populací, ve které i přímé porovnání není příliš výpočetně náročné. U větší populace by byly rozdíly v čase jistě větší.

Shrnutí

V této části byly diskutovány experimenty s mnoha parametry navrženého genetického algoritmu. Postupným nastavováním jednotlivých parametrů bylo u první sady problémů dosaženo zlepšení z odchylky $\omega_{avg} = 6,80$ a $\omega_{best} = 3,74$ u špatného nastavení EXPGA1 až na odchylku $\omega_{avg} = 5,18$ a $\omega_{best} = 2,03$ u EXPGA7. U druhé sady to pak bylo $\omega_{avg} = 24,63$ a $\omega_{best} = 18,77$, respektive $\omega_{avg} = 19,26$ a $\omega_{best} = 13,69$. Došlo tedy ke zlepšení 1,62 % a 1,71 % odchylky u první sady a 5,37 % a 5,08 % u sady druhé. Pokud si tato procenta představíme jako úsporu při reálné přepravě, není to vůbec zanedbatelné na to, že byly použity stejné techniky, pouze byly optimalizovány jejich parametry. Toto dokládá důležitost provedených experimentů, a zároveň ukazuje skutečnost, že jsou genetické algoritmy na tyto parametry velmi náchylné. Je třeba také zmínit, že v tomto srovnání nebyly brány v potaz experimenty, které šly s nastavením parametrů spíše do extrémů a vykázaly tak významně horší výsledky. Příkladem může být $\sigma = 500$ u EXPGA1, nebo $\Delta = 5$ v EXPGA7.

Konkrétní zvolené hodnoty parametrů byly vždy v závěru každého experimentu označeny tučně. S tímto nastavením tak budeme pracovat v následujících experimentech používající algoritmus RTGA.

6.5 Tabu prohledávání

Na rozdíl od komplikovaného a robustního genetického algoritmu, tabu prohledávání je velmi jednoduché a nevyžaduje nastavování velkého množství parametrů. V podstatě jediné parametry zde použité jsou velikost tabu seznamu a počet generovaných sousedních řešení. Vliv obou těchto parametrů bude v následujících experimentech zkoumán a budou vybrány nejlepší hodnoty pro řešení úlohy VRP.

6.5.1 Velikost tabu seznamu

Hlavní úlohou tabu seznamu je zamezit cyklení v okolí jednoho řešení. Toho se docílí ukládáním nedávno prozkoumaných řešení a vyhýbání se jejich opětovnému rozvíjení. Velikost tohoto seznamu pak ovlivňuje jak moc do historie je algoritmus schopen se dívat, protože čím větší seznam bude, tím starší řešení bude obsahovat. Na druhou stranu však výpočetní náročnost operace kontroly, zda seznam obsahuje generovaného řešení, roste lineárně s velikostí seznamu. Je nutné zmínit, že porovnání řešení je založeno na porovnání délky trasy. Může se totiž stát, že se objeví trasy, které mají jen odlišené pořadí cest, což by se jevilo jako jiné řešení, přičemž jsou v podstatě identické (viz Lemma 1). Situace, kdy by dvě odlišná řešení měla naprosto stejnou délku vyjádřenou v datovém typu `double`, je téměř vyloučena.

Abychom prozkoumali vliv velikosti seznamu na kvalitu řešení, byl provedena experiment, kde byl zastoupen jak krátký seznam o velikosti $TS = 10$, tak velmi dlouhý seznam o velikosti 1000 řešení. Pro zjištění průběhu mezi těmito limity, byly zvoleny také velikosti 50, 100 a 150. Výsledky experimentu EXPTABU1 jsou v tabulce 6.13.

Dataset	$TS = \mathbf{10}$		$TS = \mathbf{50}$		$TS = \mathbf{100}$		$TS = \mathbf{150}$		$TS = \mathbf{1000}$	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	14,63	8,43	8,32	4,69	8,20	4,54	**8,04**	4,49	8,10	**4,11**
Golden	39,85	33,48	39,20	33,37	38,88	33,58	**38,77**	**32,86**	38,92	33,51

Tabulka 6.13: EXPTABU1 - Volba velikosti tabu seznamu TS algoritmu RTTABU.

Z výsledků můžeme vidět, že velmi krátký seznam podává především u první sady problémů výrazně horší výsledky, což jen dokládá správnou činnost tabu seznamu. Výsledky při použití $TS = 50$ jsou pak jen o málo horší, než u větších velikostí, mezi kterými je jen malý rozdíl. Obecně se zdá, že zhruba od $TS = 100$ již prodloužením seznamu zřejmě nedochází k tak zásadnímu zlepšení výkonnosti algoritmu. Jak ale bylo řečeno, větší délky seznamu způsobují větší výpočetní náročnost, proto můžeme jako kompromis mezi rychlostí a kvalitou zvolit $\boldsymbol{TS = 150}$, který podává lepší výsledky než $TS = 100$.

6.5.2 Počet generovaných sousedních řešení

Tento parametr uvádí, kolik nových řešení bude prozkoumáno v okolí současného řešení. Jelikož je z těchto generovaných řešení zvoleno vždy to nejlepší, které postupuje dále, lze očekávat, že s větším počtem roste šance nalezení lepšího řešení a lze očekávat rychlejší postup k optimu. Naopak při malém počtu bude postup spíše obezřetný a pomalý. Počet by však zároveň neměl být moc velký, aby se algoritmus také pohyboval dále a ne jen hledal spoustu řešení v několika bodech.

Tyto domněnky budou ověřeny v experimentu EXPTABU2, kde byl zvolen počet sousedních řešení η z intervalu 5 až 100. Je nutné zmínit, že pro všechny varianty bylo nastaveno $TS = 150$.

Dataset	η=**5**		η=**10**		η=**20**		η=**50**		η=**100**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	8,70	4,94	**8,04**	4,49	8,30	**4,36**	10,28	5,27	12,13	6,40
Golden	45,44	39,21	38,90	33,10	35,06	28,89	32,64	25,35	**30,94**	**23,87**

Tabulka 6.14: EXPTABU2 - Vliv volby počtu generovaných sousedních řešení η v algoritmu RTTABU.

Z výsledků v tabulce 6.14 vidíme u obou sad problémů naprosto rozdílné výsledky. Zatímco sada menších problémů podává nejlepší výsledky při $\eta = 10$ a se zvyšováním η se jasně zhoršuje, o druhé sady rozsáhlých problémů je tomu přesně naopak a se zvyšujícím se η se výsledky dramaticky zlepšují.

Z toho vyplývá, že by tento parametr mohl být závislý na velikosti dané instance, což bude ověřeno v dalším experimentu, kde hodnota η bude určena jako poměr počtu zákazníků n dané instance. Cílem tohoto experimentu je najít takovou hodnotu, aby byly dosaženy nejlepší možné výsledky u obou sad problémů. Výsledky experimentu EXPTABU3 s několika hodnotami poměru jsou uvedeny v tabulce 6.15.

Dataset	$\eta = \frac{n}{3}$		$\eta = \frac{n}{4}$		$\eta = \frac{n}{5}$	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	9,54	5,02	8,91	4,73	**8,83**	**4,53**
Golden	**30,82**	**23,59**	31,53	24,60	31,76	24,89

Tabulka 6.15: EXPTABU3 - Experiment volby počtu generovaných sousedních řešení η algoritmu RTTABU v závislosti na velikosti instance.

Dataset	TS = **150** η = **10**		TS = **200** η = **100**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	**8,04**	**4,49**	11,73	5,97
Golden	38,90	33,10	**30,67**	**23,80**

Tabulka 6.16: EXPTABU4 - Porovnání dvou vyladěných variant nastavení RTTABU.

Z ohledem na výsledky uvedené experimentu EXPTABU3 můžeme prohlásit, že velikost η nebude přímo závislá na počtu zákazníků. Nejlepší výsledky obou instancí totiž opět stojí na opačném konci tohoto experimentu. Optimální hodnota η tak může být závislá například na tom, že zatímco v první sadě jsou zákazníci rozptýleni náhodně, v druhé sadě jsou umístěni systematicky a algoritmus se tak může chovat rozdílně.

Jedinou možností jak dosáhnout nejlepších výsledků u obou sad, tak bude zvolení parametru η odlišně pro každou sadu. Tabulka 6.16 ukazuje výsledky dvou optimalizovaných variant pro jednotlivé sady instancí. U varianty pro sadu Golden byla zvětšena velikost tabu seznamu na 200, jelikož pokud je každou iteraci generováno 100 sousedních řešení, do seznamu o velikosti 150 by se nevešly ani dvě generace těchto řešení.

Shrnutí

Uvedenými experimenty byla zjištěna nutnost nastavení různých parametrů u jednotlivých sad instancí VRP. Zatímco u volby velikosti seznamu jsou dosaženy nejlepší výsledky obou sad použitím stejné hodnoty, u počtu generovaných sousedních řešení byly výsledky velmi odlišné. Proto bylo také u druhé sady dosaženo většího zlepšení odchylky ω_{avg} o 8,53 %, respektive 9,78 % u ω_{best}. U menší sady pak bylo dosaženo od EXPTABU1 zlepšení pouze 0,28 % u ω_{avg} a 0,2 % u ω_{best}. Je však nutno říci, že EXPTABU1 byl proveden s $\eta = 10$, což se později ukázalo jako vhodná volba pro první sadu. Pokud by ze začátku byla zvolena jiná hodnota, zlepšení by bylo zajisté větší.

Z výsledků je vidět, že i u tak jednoduchého algoritmu jako TABU, je nutné experimentálně zjistit optimální hodnoty použitých dvou parametrů. Výsledky RTTABU pak zaostávají za výsledky RTGA přibližně o 3 % u menší a 11 % u větší sady. Z ohledem na jednoduchost algoritmu to není špatný výsledek. Co se týče rychlosti tohoto algoritmu, u první sady byla průměrná délka běhu jedné instance 1,12 minut a u druhé 10,55 minut. To je o cca 25 %, respektive 55 % déle, než trvá RTGA. Genetický algoritmus tak stejný počet řešení prozkoumá za kratší dobu, což může být způsobeno absencí kontroly tabu seznamu, nebo také pokud je operace křížení méně výpočetně náročná než mutace, resp. lokální prohledávání.

6.6 Simulované žíhání

Ačkoliv je algoritmus simulovaného žíhání lehce komplikovanější než tabu vyhledávání, vyžaduje také jen velmi málo parametrů. Zásadní roli v tomto algoritmu hraje aktuální teplota, které se budou týkat zkoumané parametry. Konkrétně můžeme ovlivnit počáteční teplotu algoritmu a její změnu, což bude předmětem následujících experimentů. U všech experimentů bude minimální teplota představená v 3.6 nastavena na hodnotu $T_{min} = 0,01$.

6.6.1 Počáteční teplota

Nastavením počáteční teploty T_0 přímo ovlivňujeme, jak dlouho poběží vnitřní algoritmus žíhání a také jaká řešení budou přijata. Jelikož je běh omezen minimální teplotou (viz 3.6), čím bude počáteční teplota větší, tím později se také dostane na toto minimum. V 3.6 je představena technika zahřívání a tedy, čím bude teplota nižší a běh kratší, tím více se opětovné zahřívání projeví. Teplota má ale přímý vliv také na pravděpodobnost přijetí řešení a jak je uvedeno v 2.3.3, čím vyšší bude teplota, tím větší je pravděpodobnost přijetí horšího řešení. Příliš velká teplota by tak mohla velmi často přijímat horší výsledky a naopak při nastavení nízké teploty se málo projeví výhody tohoto algoritmu. V článku *A Simulated Annealing Algorithm for the Vehicle Routing Problem with Time Windows and Synchronization Constraints* [1], kterým byl algoritmus RTSA inspirován, je experimentálně zvolena $T_0 = 20$. Pro experiment EXPSA1 byly zvoleny teploty jak velmi nízké ($T_0 = 10$), tak velmi vysoké ($T_0 = 10000$) a teploty mezi těmito limity. Výsledky experimentu jsou uvedeny v tabulce 6.17 a budou diskutovány spolu se změnou teploty v závěrečném shrnutí.

Je nutné zmínit, že změna teploty se provádí násobením aktuální teploty koeficientem z intervalu $\langle 0, 1\rangle$ a teplota tedy klesá logaritmicky s počtem iterací. Reálně to znamená, že při $T_0 = 100$ a $T_\Delta = 0,99$ bude vykonáno 917 iterací a při stonásobné teplotě $T_0 = 10000$ 1375 iterací. Při zvýšení počáteční teploty 100x tak bude zvýšen počet iterací pouze o cca 50 %. Je tedy jasné, že zvyšování počáteční teploty po malých intervalech nebude znamenat velký rozdíl v počtu iterací.

Dataset	$T_0 = \mathbf{10}$		$T_0 = \mathbf{20}$		$T_0 = \mathbf{50}$	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	11,60	7,19	10,93	7,06	10,45	6,60
Golden	40,44	**34,89**	**40,31**	35,41	40,42	35,47
	$T_0 = \mathbf{100}$		$T_0 = \mathbf{1000}$		$T_0 = \mathbf{10\ 000}$	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	10,10	**5,83**	**10,05**	6,11	10,11	5,98
Golden	40,47	35,15	42,71	37,71	43,48	38,05

Tabulka 6.17: EXPSA1 - Volby počáteční teploty T_0 algoritmu RTSA.

6.6.2 Změna teploty

Parametrem změny teploty T_Δ se v každé iteraci algoritmu simulovaného žíhání vynásobí aktuální teplota, a tak tato hodnota přímo určuje, jak rychle bude teplota klesat. Proto nastavení tohoto parametru spolu s počáteční teplotou přímo ovlivňuje, jak dlouho vnitřní algoritmus poběží. Pokud usoudíme, že je žádoucí, aby vnitřní algoritmus žíhání běžel déle, spíše než počáteční teplotu, bychom měli zmírnit právě její klesání. Jak již bylo řečeno, nastavení vysokých teplot způsobí velmi velkou pravděpodobnost přijetí horšího řešení (přes 99 %), což nemusí být žádoucí. Změna teploty T_Δ je typicky číslo blízké 1, kterým je v každé iteraci vynásobena aktuální teplota. Čím bude T_Δ vyšší, tím pomaleji bude teplota klesat a bude provedeno více iterací. V článku [1] bylo zvoleno $T_\Delta = 0,99$, proto pro experimenty vybereme hodnoty vyšší a nižší od této. Výsledky experimentů jsou v tabulce 6.18, kde byla zvolena $T_0 = 100$.

Naopak od počáteční teploty, změna T_Δ má velký vliv na počet iterací. Jak již bylo řečeno, při $T_0 = 100$ a $T_\Delta = 0,99$ bude vykonáno 917 iterací. Pokud zvýšíme T_Δ na 0,999, bude počet iterací již 9206, což je desetinásobek. Toto jen potvrzuje tvrzení, že bychom pro delší běh měli zvýšit spíše T_Δ než T_0.

Dataset	$T_\Delta = \mathbf{0,9999}$		$T_\Delta = \mathbf{0,999}$		$T_\Delta = \mathbf{0,995}$		$T_\Delta = \mathbf{0,992}$	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	11,34	6,35	10,95	5,80	10,22	6,09	10,12	**5,66**
Golden	38,48	**29,49**	**37,74**	30,59	38,71	32,85	39,20	34,05
	$T_\Delta = \mathbf{0,99}$		$T_\Delta = \mathbf{0,98}$		$T_\Delta = \mathbf{0,95}$		$T_\Delta = \mathbf{0,9}$	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	**10,10**	6,01	10,14	6,27	10,47	6,76	11,45	7,59
Golden	39,61	33,96	40,46	35,83	42,33	37,69	43,79	38,85

Tabulka 6.18: EXPSA2 - Vliv různé změny teploty T_Δ algoritmu RTSA.

Vyhodnocení

Z výsledků v tabulkách 6.17 a 6.18 můžeme usoudit, že v obou případech vyhovují jednotlivým sadám instancí VRP naprosto jiné hodnoty. Zatímco na první sadě *Christofides* algoritmus vykazuje nejlepší výsledky při volbě $T_0 = 1000$ a $T_\Delta = 0,99$, u druhé sady

Golden jsou to hodnoty $T_0 = 20$ a $T_\Delta = 0,999$. Ukazuje se tak, že v případě první sady je preferován rychlý sestup, a tím pádem i velká činnost opětovného zahřívání a naopak v druhé sadě delší žíhání s občasným zahřátím. Jelikož jsou tyto dvě nastavení opravdu rozdílná, byly podobně jako u TABU zvoleny hodnoty parametrů v závislosti na sadě instancí. Toto ilustruje závěrečný experiment EXPSA3 uvedený v tabulce 6.19, kde vidíme tyto dvě varianty a můžeme pozorovat nejlepší dosažené výsledky u obou sad problémů. Naproti tomu můžeme znovu potvrdit odlišnost, pokud se podíváme na výsledky s použitím varianty optimalizované pro druhou sadu, kde je zřetelné zhoršení v obou případech.

Dataset	$T_0 = \mathbf{1000}$ $T_\Delta = \mathbf{0,99}$		$T_0 = \mathbf{20}$ $T_\Delta = \mathbf{0,999}$	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	**9,99**	**6,42**	11,70	6,87
Golden	41,62	35,94	**36,50**	**29,62**

Tabulka 6.19: EXPSA3 - Porovnání dvou vyladěných variant nastavení RTSA.

Dosažené zlepšení optimalizací obou parametrů je nejlépe patrné v závěrečném experimentu EXPSA3, kde vidíme velmi rozdílné výsledky při různém nastavení. Celkově tento algoritmus trochu zaostává za již prezentovanými algoritmy RTGA a RTTABU a to přibližně o 5 %, resp. 2 % u první sady a 17 %, resp. 6 % u druhé. Výpočetní doba algoritmu je téměř shodná s RTTABU, což dává smysl, jelikož obě tyto metody zkoumají nová řešení lokálním prohledáváním.

6.7 Optimalizace mravenčí kolonií

V práci byla také implementována metoda optimalizace mravenčí kolonií (ANT). Nejprve byl v 2.3.4 prezentován základním algoritmus této metody, navržený pro problém TSP. Dále byly v 3.7 uvedeny vylepšení pro problém VRP, které v článku *Applying the Ant System to the Vehicle Routing Problem* [7] představil Bullnheimer a kol.

Obě tyto varianty byly prozkoumány v experimentu EXPANT1, jehož výsledky jsou uvedeny v tabulce 6.20. Algoritmus je výpočetně velmi náročný, proto byl experiment proveden pouze na první, menší sadě problémů.

Dataset	Základní varianta		Varianta pro VRP	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	**15,65**	**11,94**	35,27	31,40

Tabulka 6.20: EXPANT1

Z výsledků vidíme, že výsledky algoritmu velmi zaostávají za všemi zatím představenými algoritmy. Je také s podivem, že varianta používající vylepšení pro VRP ve výsledku dopadla mnohem hůře, než základní algoritmus. To i přesto, že byly nastaveny stejné váhy u výpočtu pravděpodobnosti vybrání jako v článku [7].

Průměrný čas řešení jedné instance byl pak 64,52 minut, což je přibližně sedmdesátkrát pomalejší, než například algoritmus RTGA. Toto je zřejmě způsobeno tím, že v každém kroku umělého mravence musí být spočítáno mnoho pravděpodobností vybrání dalšího zákazníka. V součtu je tak jedno řešení prozkoumáno za mnohem delší dobu, než například pokud v rámci mutace změníme pořadí několika zákazníků.

Vzhledem ke špatným výsledkům a velmi pomalému běhu nebudeme v práci tuto metodu dále rozebírat. Je také nutné podotknout, že navržené učící se RT mutace nejsou jednoduše použitelné v tomto algoritmu a bude tak lepší se zaměřit na algoritmy GA, TABU a SA, které sdílí podobné principy.

6.8 Metody využívající RT tabulky

V této části budou zkoumány metody využívající charakteristiky předchozích řešení uložených v RT tabulce, které byly představeny v kapitole 4.3. Nejprve budou diskutovány různé varianty a parametry těchto metod a následně bude zhodnocen celkový přínos jejich použití v různých představených algoritmech. Představené RT mutace pak budou voleny s různou pravděpodobností a u nejlepší kombinace bude dále zkoumán nejlepší poměr pro jednotlivé algoritmy. V závěru kapitoly bude diskutován možný přínos použití uložených charakteristik předložených na vstup nového běhu algoritmu. Úvodní varianty RT metod budou zkoumány na algoritmu RTGA, jelikož ten zatím dosahoval nejlepších výsledků a optimalizace mutací by se měla dotknout všech metod podobným způsobem. Od zhodnocení přínosu metod již budou uvedeny vždy výsledky pro každý algoritmus zvlášť, tak aby bylo nastavení optimalizováno pro každý použitý algoritmus co nejlépe.

6.8.1 Výběr nejhorší nebo nejlepší charakteristiky

V článku *A Dynamic Analysis of Resource-Constrained Project Scheduling Problems for an Informed Search via Genetic Algorithm Optimization* [16] Hrubý uvažuje dva způsoby výběru operací pro posun v mutaci založené na znalostech v RT tabulkách. Jeden vybírá ty, které se vyskytují v nejhorších charakteristikách a druhý naopak vybírá ty nejlepší. V článku je konstatováno, že k lepším výsledkům vede výběr a následný posun těch nejhorších, což vcelku odpovídá charakteru navržené mutace.

Rozhodl jsem se provést podobný experiment, kde v navržených mutacích *RTShift* a *RTCluster* budou vybráni zákazníci pro posun buď z nejhorších (ALG_{worst}) nebo nejlepších (ALG_{best}) charakteristik. Experiment byl proveden se všemi třemi algoritmy RTGA, RTTABU a RTSA, přičemž byly použity mutace *RTShift* a *RTCluster* s pravděpodobností 50 %. Výsledky experimentu EXPRT1 jsou uvedeny v tabulkách 6.21 a 6.22.

Dataset	RTGA_{best}		RTGA_{worst}		RTTABU_{best}		RTTABU_{worst}	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	7,19	4,15	**6,83**	**3,81**	18,18	14,05	**10,84**	**7,77**
Golden	21,37	15,90	**20,21**	**14,69**	30,92	22,96	**29,22**	**22,95**

Tabulka 6.21: EXPRT1 - Srovnání výběru nejhorších a nejlepších charakteristik pro RT mutaci v algoritmu RTGA a RTTABU.

Dataset	RTSA_{best}		RTSA_{worst}	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	23,60	19,31	**14,20**	**10,51**
Golden	43,55	36,58	**37,01**	**31,29**

Tabulka 6.22: Pokračování EXPRT1 - Srovnání výběru nejhorších a nejlepších charakteristik pro RT mutaci v algoritmu RTSA.

Z výsledků experimentů můžeme jen potvrdit závěr z [16], že algoritmy vybírající nejhorší charakteristiky produkují lepší výsledky. Můžeme si všimnout menšího rozdílu obou variant v RTGA, což je nejspíš způsobeno vlivem křížení, které stále pracuje shodně. U metod RTTABU a RTSA spoléhajících čistě na mutaci, je pak rozdíl podstatně větší.

6.8.2 Vliv vylepšení RTShift

V kapitole 4.3.1 bylo navrženo vylepšení mutace *RTShift*, které provádí posunutí zákazníků informovaně a vybírá vždy tu nejlepší pozici, kam se posune. Ačkoliv toto vylepšení představuje zvýšení výpočetní zátěže, rozdíl v nalezených řešeních by měl být významný. V následujícím experimentu EXPRT2 bude použita čistě tato mutace v RTGA, a to nejprve bez vylepšení a poté s ním. V tabulce 6.23 jsou uvedeny dosažené výsledky spolu s porovnáním průměrného časů běhu $time_{avg}$.

Dataset	**bez vylepšení**			**s vylepšením**		
	ω_{avg} [%]	ω_{best} [%]	$time_{avg}$ [min]	ω_{avg} [%]	ω_{best} [%]	$time_{avg}$ [min]
Christofides	10,79	5,99	0,528	**7,77**	**4,20**	0,545
Golden	29,95	22,84	3,186	**23,27**	**17,59**	3,284

Tabulka 6.23: EXPRT2 - Srovnání výsledků a času běhu základní a vylepšené mutace *RTShift* v algoritmu RTGA.

Z výsledků můžeme jasně vidět výrazné zlepšení u obou sad problémů při použití vylepšené verze metody *RTShift*. Ukazuje se tedy, že informované posouvání zákazníků předčí to náhodné. Přitom časy obou variant nejsou příliš odlišné, kdy jen o něco déle trvá metoda s vylepšením, což odpovídá předpokladům.

6.8.3 Parametry mutace RTReshuff

U mutace *RTReshuff* představené v kapitole 4.3.3 byly uvedeny dva parametry, které hrají roli ve funkci této metody. Prvním parametrem je určeno kolik tras z řešení bude mutováno a kolik naopak ponecháno. Vyjádřeme tento parametr v procentech tras, které budou mutovány, jako R_{mut}. Samotný počet tras pak bude samozřejmě záležet na konkrétní instanci a počtu tras v řešení. V experimentu EXPRT2 bude zkoumáno mutování jak všech tras ($R_{mut} = 100$ %), tak postupně snižující se počet až na 25 %. U velkého procenta může někdy docházet k přílišné nežádoucí duplikaci nejlepšího řešení, zatímco malé procento může vést jen k malé změně zpomalující celý proces. Výsledky experimentu EXPRT2 jsou uvedeny v tabulce 6.24.

Druhým důležitým parametrem v této mutaci je procento seřazeného seznamu, ze kterého bude vybrán vždy další soused, označme jej jako P_{best}. Tímto parametrem je do mutace vnesen prvek náhody, tak aby nedocházelo u všech řešení s podobným rozložením tras ke shodnému výsledku. Čím menší toto procento bude, tím více hrozí tento efekt. Naopak při velkých hodnotách může často docházet ke zhoršení řešení, protože budou vybíráni i nekvalitní sousedé. V experimentu byly použity hodnoty od 0 %, tedy vybrání vždy nejlepšího souseda, až po $P_{best} = 50$ %. Výsledky EXPRT3 jsou uvedeny v tabulce 6.25. V obou uvedených experimentech byl použit algoritmus RTGA pouze s použitím mutace *RTReshuff*.

Dataset	R_{mut} = **100%**		R_{mut} = **75%**		R_{mut} = **66,6%**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	5,29	2,17	5,11	**1,89**	5,09	2,08
Golden	20,84	15,13	20,54	14,89	20,50	15,11
	R_{mut} = **50%**		R_{mut} = **33,3%**		R_{mut} = **25%**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	5,08	2,11	5,05	2,20	**5,03**	2,15
Golden	20,46	15,14	20,53	**14,67**	**20,33**	14,79

Tabulka 6.24: EXPRT3 - Vliv počtu mutovaných tras R_{mut} mutace *RTReshuff* v algoritmu RTGA.

Dataset	P_{best} = **0%**		P_{best} = **5%**		P_{best} = **10%**		P_{best} = **25%**		P_{best} = **50%**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	**4,95**	**2,15**	5,00	2,40	5,12	2,34	6,97	3,75	7,35	4,02
Golden	**17,77**	**12,65**	18,46	13,01	20,50	14,64	21,95	15,71	21,89	15,92

Tabulka 6.25: EXPRT4 - Volby parametru P_{best} mutace *RTReshuff* v algoritmu RTGA.

Z výsledků EXPRT3 vidíme, že se algoritmus *RTReshuff* chová lépe, pokud je mutována jen část tras. Výsledky především první sady se pak příliš nemění s volbou tohoto poměru, ale o něco lepší výsledky nacházíme s nižší hodnotou R_{mut}. Toto potvrzují také výsledky druhé sady instancí, kde je dosaženo nejlepších výsledků volbou $R_{mut} = 25$ %. Zatímco odchylky nejsou příliš rozdílné, můžeme s jistotou říci, že čím menší procento tras bude mutováno, tím méně výpočetního výkonu bude metoda vyžadovat. Konkrétně při $R_{mut} = 100$ % byl průměrný čas pro vyřešení instance 1,01, respektive 7,78 minut, u $R_{mut} = 25$ % to bylo 0,75, respektive 5,62 minut. Jedná se tedy o cca 25% časovou úsporu. Proto pro další použití volíme $\boldsymbol{R_{mut} = 25}$ **%**, které kombinuje dobré výsledky a rychlejší výpočetní čas.

Experiment EXPRT4, jehož výsledky jsou v tabulce 6.25, dopadl více překvapivě. Z výsledků vidíme, že zavádění náhodnosti výběru výsledky zhoršuje lineárně s velikostí P_{best}. Nejlépe pak dopadlo vybrání vždy nejlepšího souseda. Bylo předpokládáno, že by toto mohlo vést k přílišnému klonování nejlepšího nalezeného řešení, avšak experiment toto vyvrací. Zřejmě je to tím, že je mutována jen menší část tras ($R_{mut} = 25$ %), a je naopak příznivé, když jsou vybrané trasy zmutovány co nejlépe, i když se již mohou objevovat v jiném řešení. V dalším použití algoritmu tak budou vybíráni vždy ti nejlepší sousedé, tedy $\boldsymbol{P_{best} = 0}$ **%**.

6.8.4 Vliv použití metod využívajících RT tabulku

V kapitole 4.3 byly navrženy tři mutace využívající specifika řešení průběžně ukládaná do RT tabulek. Cílem následujících experimentů je prozkoumat, jestli použití těchto metod vylepšují řešení a také nalézt jejich nejlepší kombinaci. Vzhledem k charakteru těchto metod je vhodné je spustit až po uběhnutí nějaké doby, během které se alespoň z části RT tabulka naplní daty. V prvních 50 iteracích se tak bude používat pouze náhodný přístup a až poté se zapojí tyto metody.

Experiment bude proveden nejprve s náhodným přístupem, tedy s mutací *BCRM* popsanou v 3.4.3. Poté budou spuštěny jednotlivé metody samostatně a nakonec budou prozkoumány všechny jejich kombinace. V případě kombinace budou metody spuštěny se stejnou pravděpodobností, tedy 50% v případě kombinace dvou metod, respektive 33% u kombinace všech. Zvolené poměry nemusí být ideální, ale cílem tohoto experimentu je získat celkový náhled na prospěšnost těchto kombinací. Experiment bude proveden pro všechny tři zkoumané algoritmy, tedy RTGA, RTTABU a RTSA. Samotné výsledky jsou uvedeny v tabulkách 6.26, 6.27 a 6.28.

Dataset	**Random**		**RTShift**		**RTCluster**		**RTReshuff**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	7,36	4,17	7,77	4,20	6,51	3,70	5,17	2,19
Golden	20,39	15,11	23,27	17,59	18,83	13,75	17,66	12,22
	Cluster+Shift		**Cluster+Reshuff**		**Shift+Reshuff**		**Vše**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	6,83	3,81	**4,96**	**2,16**	6,07	2,94	5,53	2,41
Golden	20,21	14,69	**16,35**	**11,22**	19,74	14,12	17,74	12,35

Tabulka 6.26: EXPRT5 - Vliv využití RT mutací v algoritmu RTGA.

Dataset	**Random**		**RTShift**		**RTCluster**		**RTReshuff**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	15,67	13,48	30,66	21,26	11,62	8,15	32,50	23,70
Golden	32,12	29,11	51,98	43,14	**25,62**	**19,96**	51,82	43,04
	Cluster+Shift		**Cluster+Reshuff**		**Shift+Reshuff**		**Vše**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	10,84	7,77	**6,59**	**3,35**	24,38	14,81	7,35	3,57
Golden	29,22	22,95	29,19	20,62	49,66	39,43	30,64	21,40

Tabulka 6.27: EXPRT6 - Vliv využití RT mutací v algoritmu RTTABU.

Dataset	**Random**		**RTShift**		**RTCluster**		**RTReshuff**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	16,12	13,53	39,01	32,55	16,71	12,81	38,36	32,86
Golden	36,49	32,25	75,36	63,60	35,55	29,16	71,29	61,90
	Cluster+Shift		**Cluster+Reshuff**		**Shift+Reshuff**		**Vše**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	14,20	10,51	**8,32**	**4,61**	34,92	28,75	9,24	5,51
Golden	37,01	31,29	**29,78**	**23,61**	71,59	61,82	30,97	24,75

Tabulka 6.28: EXPRT7 - Vliv využití RT mutací v algoritmu RTSA.

Ze zjištěných dat můžeme vidět, že u všech algoritmů dosahuje kombinace všech RT metod lepších výsledků než náhodný přístup. Toto je hlavně viditelné u RTTABU a RTSA, kde je rozdíl daleko větší než u RTGA. Je to zřejmě způsobeno existencí křížení v RTGA, přičemž RTTABU a RTSA jsou závislé pouze na mutaci.

U metod RTTABU a RTSA si dále můžeme všimnout velmi špatných výsledků u použití pouze *RTShift*, *RTReshuff*, nebo jejich kombinace. To je způsobeno charakterem metod, kdy tyto metody dokáží měnit pořadí zákazníků pouze v rámci daných tras a ne mezi nimi. U těchto metod tak nebylo možné změnit zařazení do tras od počátečního řešení. Naopak u RTGA úlohu rozmístění do jiných tras může zastupovat křížení, což potvrzují daleko lepší výsledky při použití těchto mutací.

Druhou možností správného použití *RTShift* a *RTReshuff* je kombinace například s metodou *RTCluster*. Toto se ve všech případech ukazuje lepší, než použití metod samotných, přičemž kombinace *RTClust+Reshuff* dopadla ve všech algoritmech úplně nejlépe. Tato kombinace pak předčí i kombinaci všech tří metod, kdy zavedení *RTShift* výsledky zhoršuje ve všech případech.

Výsledky druhé sady se tomu trochu vymykají u RTTABU, které dopadly nejlépe pouze při použití metody *RTCluster*, přičemž výsledky kombinace *RTClust+Reshuff* jsou druhé nejlepší. Je tedy možné, že tomuto algoritmu by svědčil větší podíl mutace *RTCluster*. Naopak u genetického algoritmu je samotnou mutací *RTReshuff* dosaženo lepších výsledků než *RTCluster* a je tak možné, že by bylo lepší použít větší poměr naopak u *RTReshuff*. Kombinaci *RTClust+Reshuff* se tak budeme dále věnovat a v následujícím experimentu bude zkoumán optimální poměr těchto metod pro jednotlivé algoritmy.

6.8.5 Poměr mutací *RTReshuff* a *RTCluster*

Jak již bylo řečeno, ve všech algoritmech je nejlepších výsledků dosaženo použitím mutací *RTClust+Reshuff*, resp. jen *RTCluster*. Jednotlivé algoritmy však zdá se preferují různé poměry zmíněných metod. V následujícím experimentu bude prozkoumáno několik poměrů, přičemž budeme uvádět pravděpodobnost mutace *RTReshuff* značenou jako $P_{Reshuff}$.

Jelikož u RTGA je potenciál lepších výsledků u větších hodnot $P_{Reshuff}$, budou zvoleny hodnoty spíše z rozsahu $P_{Reshuff} > 50$ %. Naopak u algoritmů RTTABU a RTSA pak budou spíše voleny hodnoty $P_{Reshuff} < 50$ %. Nicméně vždy bude uveden i výsledek s hodnotami mimo tyto rozsahy pro ověření uvedených hypotéz. Výsledky pro jednotlivé algoritmy jsou uvedeny v tabulkách 6.29, 6.30 a 6.31.

Dataset	$P_{Reshuff} = \mathbf{10\%}$		$P_{Reshuff} = \mathbf{25\%}$		$P_{Reshuff} = \mathbf{50\%}$		$P_{Reshuff} = \mathbf{60\%}$	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	5,72	2,95	5,23	2,47	4,96	2,16	5,00	2,08
Golden	17,10	12,26	16,48	11,95	16,35	**11,22**	16,46	11,66
	$P_{Reshuff} = \mathbf{70\%}$		$P_{Reshuff} = \mathbf{75\%}$		$P_{Reshuff} = \mathbf{80\%}$		$P_{Reshuff} = \mathbf{90\%}$	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	4,99	1,92	**4,90**	1,88	4,98	**1,80**	**4,90**	2,12
Golden	**16,31**	**11,22**	16,46	11,55	16,54	12,15	17,02	12,61

Tabulka 6.29: EXPRT8 - Poměr mutace *RTReshuff* a *RTCluster* v algoritmu RTGA.

Dataset	$P_{Reshuff}$ = **5%**		$P_{Reshuff}$ = **10%**		$P_{Reshuff}$ = **15%**		$P_{Reshuff}$ = **20%**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	5,92	2,91	**5,79**	2,63	5,86	**2,56**	5,89	2,68
Golden	**23,69**	**17,76**	25,64	19,34	26,42	18,83	27,06	19,75
	$P_{Reshuff}$ = **25%**		$P_{Reshuff}$ = **50%**		$P_{Reshuff}$ = **75%**		$P_{Reshuff}$ = **90%**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	5,99	2,66	6,59	3,35	8,85	4,37	10,94	5,26
Golden	27,69	20,05	29,19	20,62	31,57	23,94	34,71	26,70

Tabulka 6.30: EXPRT9 - Poměr mutace *RTReshuff* a *RTCluster* v algoritmu RTTABU.

Dataset	$P_{Reshuff}$ = **5%**		$P_{Reshuff}$ = **10%**		$P_{Reshuff}$ = **15%**		$P_{Reshuff}$ = **20%**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	8,07	4,86	7,98	4,45	8,00	4,49	**7,92**	4,34
Golden	29,22	23,25	29,18	23,51	**29,08**	**22,15**	29,12	22,68
	$P_{Reshuff}$ = **25%**		$P_{Reshuff}$ = **50%**		$P_{Reshuff}$ = **75%**		$P_{Reshuff}$ = **90%**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	7,97	**4,14**	8,32	4,61	9,01	5,03	10,96	6,55
Golden	29,09	22,85	29,78	23,61	31,25	25,65	34,13	28,82

Tabulka 6.31: EXPRT10 - Poměr mutace *RTReshuff* a *RTCluster* v algoritmu RTSA.

Z uvedených výsledků můžeme potvrdit, že algoritmu RTGA svědčí vyšší zastoupení mutace *RTReshuff*, kdežto RTTABU a RTSA spíše vyžadují větší poměr metody *RTCluster*. Konkrétně u RTGA nebyly moc velké rozdíly od $P_{Reshuff} \geq 50$ %, nicméně bylo zjištěno, že metoda *RTReshuff* je méně výpočetně náročná, než metoda *RTCluster*, proto byla pro RTGA zvolena hodnota $\boldsymbol{P_{Reshuff} = 75}$ **%**.

U algoritmu RTTABU opět vidíme různé výsledky u obou sad instancí, což může být způsobeno také jejich rozdílným nastavením parametrů. Zatímco první sada podává nejlepší výsledky se zastoupením *RTReshuff* kolem 10-15 %, u druhé sady se výsledky jasně zlepšují se snižující se pravděpodobností. U druhé sady tedy zvolíme $\boldsymbol{P_{Reshuff} = 5}$ **%** a u první $\boldsymbol{P_{Reshuff} = 10}$ **%**.

V případě RTSA jsou pak rozdíly u obou sad velmi malé u $P_{Reshuff} \leq 25$ % a mohou být z části způsobeny statistickým rozptylem. Proto také z důvodů výpočetního výkonu bude zvolena spíše vyšší hodnota $\boldsymbol{P_{Reshuff} = 20}$ **%**.

6.8.6 Počátek zapojení RT metod

V této sekci již bylo zmíněno, že RT metody potřebují jistý čas pro dostatečné naplnění RT tabulek daty, ze kterých budou metody vycházet. Pokud se tyto metody mutace začnou rozhodovat na základě prázdné tabulky, jistě neprovedou správné výměny a posuny zákazníků. RT mutace se tak spouštějí až od určité iterace algoritmu, kdy je již RT tabulka alespoň z části naplněna. Do té doby algoritmus používá pouze náhodné přístupy. Právě optimální počet iterací IT_{RT} bude zkoumán následujícím experimentem. Je také nutné zmínit, že dosavadní experimenty byly provedeny s $IT_{RT} = 50$. Výsledky pro jednotlivé algoritmy jsou uvedeny v tabulkách 6.32, 6.33 a 6.34.

Dataset	IT_{RT} = **10**		**100**		**1 000**		**10 000**		**50 000**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	5,19	2,23	4,80	2,21	4,62	**2,08**	**4,61**	2,14	4,96	2,40
Golden	18,25	12,36	15,93	10,97	15,03	10,53	**14,62**	**10,06**	15,61	11,22

Tabulka 6.32: EXPRT11 - Volba iterace IT_{RT}, od které se zapojí RT metody v RTGA.

Dataset	IT_{RT} = **10**		**100**		**1 000**		**10 000**		**50 000**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	5,74	2,79	5,88	3,27	5,76	**3,16**	**5,49**	3,29	5,85	3,12
Golden	23,49	17,47	**23,40**	**16,66**	24,36	17,30	32,15	29,28	32,12	28,98

Tabulka 6.33: EXPRT12 - Volba iterace IT_{RT}, od které se zapojí RT metody v RTTABU.

Dataset	IT_{RT} = **10**		**100**		**1 000**		**10 000**		**50 000**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	7,90	4,44	7,99	4,61	**7,79**	4,67	7,80	**4,65**	7,85	4,69
Golden	29,35	22,84	29,11	22,66	**29,06**	**22,41**	29,73	23,02	30,91	22,95

Tabulka 6.34: EXPRT13 - Volba iterace IT_{RT}, od které se zapojí RT metody v RTSA.

Z výsledků vidíme, že RT metody opravdu potřebují čas na zaplnění RT tabulek. Daleko lepších výsledků dosahují, pokud se počká až 10 000 iterací. Naopak pokud budeme čekat moc dlouho (IT_{RT} = 50 000), metody se již tolik nezapojí do algoritmů, a pak dosahují horších výsledků. Z experimentu také vyplývá, že zatímco algoritmům RTGA a RTTABU vyhovuje $\boldsymbol{IT_{RT} = 10000}$, u algoritmu RTSA je to spíše hodnota $\boldsymbol{IT_{RT} = 1000}$. Také vidíme, že u RTTABU opět dochází k odlišnému chování u obou sad, kdy druhá sada rozsáhlejších instancí dosahuje výrazně lepších výsledků při použití nižších hodnot IT_{RT}. Konkrétně v tomto případě bude použito $\boldsymbol{IT_{RT} = 100}$.

6.8.7 Vliv použití vyplněné tabulky

V předešlém experimentu byla diskutována prodleva, než se spustí RT metody, a to z důvodu nevyplněných RT tabulek ze začátku experimentu. Myšlenkou následujícího experimentu je poskytnou algoritmu vyplněnou tabulku z předešlých běhů, což umožní spustit tyto metody hned od začátku. Hrubý v článku [16] také tuto možnost uvažuje a předkládá, že v takovém případě je počáteční nekvalitní populace velmi vzdálená od poznatků v RT tabulkách a RT mutace pak nemají stejný efekt. Na druhou stranu i z nekvalitních řešení je možné selektivně odstraňovat nejhorší charakteristiky a pomocí informací z RT tabulky bychom se tak měli rychleji dostat na výsledky, které dosáhl běh generující tuto tabulku. Poté by algoritmus mohl mít více času na další vylepšení.

Následující experiment tak bude používat RT tabulky, které byly před samotným během vygenerovány desetinásobným navazujícím spuštěním příslušného algoritmu u každé instance. Následně bylo klasicky spuštěno 50 samostatných běhů algoritmu používající stejnou tabulku. V experimentu je také zkoumáno, jaký má vliv použití RT tabulek vygenerovaných jinými algoritmy. Tabulky byly vygenerovány algoritmy RTGA, RTTABU i RTSA a byly vyzkoušeny všechny kombinace použití. Výsledky ukazují tabulky 6.35, 6.36 a 6.37.

Dataset	**RT z GA**		**RT z TABU**		**RT z SA**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	**3,39**	**1,86**	4,36	2,09	4,85	2,37
Golden	14,03	9,22	13,62	9,31	**13,39**	**9,11**

Tabulka 6.35: EXPRT14 - Použití vygenerovaných RT tabulek na vstup algoritmu RTGA.

Dataset	**RT z GA**		**RT z TABU**		**RT z SA**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	6,14	**4,03**	**4,85**	4,34	5,33	4,12
Golden	25,01	18,43	25,82	18,91	**24,96**	**18,39**

Tabulka 6.36: EXPRT15 - Použití vygenerovaných RT tabulek na vstup algoritmu RTTABU.

Dataset	**RT z GA**		**RT z TABU**		**RT z SA**	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	8,28	5,98	**5,22**	**4,50**	5,86	5,25
Golden	**31,27**	**26,99**	31,56	27,70	32,37	27,65

Tabulka 6.37: EXPRT16 - Použití vygenerovaných RT tabulek na vstup algoritmu RTSA.

Z výsledků EXPRT14-16 můžeme vidět, že použití RT tabulek na vstupu může algoritmu ve skutečnosti pomoci. Toto vidíme především na první sadě, kdy jsou výsledky výrazně lepší, než v posledním experimentu. Naopak u druhé sady došlo ke zhoršení u algoritmu RTTABU i RTSA.

Dále je zajímavé to, že různé algoritmy na různých sadách dosahovaly nejlepší výsledky s tabulkami od odlišných algoritmů. Například algoritmus RTSA dosáhl u první sady nejlepších výsledků s tabulkami od TABU a u druhé sady s tabulkami od GA. Ostatní výsledky jsou uvedeny tučně v tabulkách. Nemůžeme ale říci, že by tabulky některé metody byly vyloženě kvalitnější, a byly tak nejpřínosnější pro všechny metody.

Před samotným během algoritmu by tak bylo možné spustit několik krátkých běhů jiného algoritmu, naplnit tím RT tabulku a poté začít s RT metodami. V takovém případě by možná bylo nutné posunout hranici IT_{RT} z nuly spíše na nějaké menší číslo, jelikož bychom nechtěli ať takové naplnění trvá déle než celý běh algoritmu. Je pak otázka, jak moc by toto zasáhlo do výpočetní složitosti algoritmu, a zdali by složitost vyvážily lepší výsledky.

6.8.8 Dlouhý běh

Hlavním přínosem použití RT tabulek v předchozím experimentu je dle mého názoru v urychlení konvergence k dobrému řešení a tím také více času na další hledání. V závěrečném experimentu je tedy prodloužen běh algoritmu desetkrát a to s limitem vyhodnocení 10M řešení. Metody poběží logicky déle, ale bude zajímavé zjistit, jak moc se tímto zlepší jejich výsledky. V experimentu EXPRT17 jsou srovnány nejlepší dosažené výsledky jednotlivých algoritmů pro limit 1M řešení (varianta 1M) a pro dlouhý běh 10M. Výsledky experimentu jsou uvedeny v tabulkách 6.38 a 6.39.

Dataset	RTGA1M		RTGA10M		RTTABU1M		RTTABU10M	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	4,61	2,08	**3,23**	**1,18**	5,49	3,16	**4,19**	**2,11**
Golden	14,62	10,06	**9,85**	**6,09**	23,40	16,66	**21,40**	**16,30**

Tabulka 6.38: EXPRT17 - Dlouhý běh jednotlivých algoritmů RTGA a RTTABU.

Dataset	RTSA1M		RTSA10M	
	ω_{avg}	ω_{best}	ω_{avg}	ω_{best}
Christofides	7,79	4,65	**5,80**	**3,17**
Golden	29,06	**22,41**	**27,55**	23,08

Tabulka 6.39: Pokračování EXPRT17 - Dlouhý běh algoritmu RTSA.

Výsledky dlouhého běhu (10M) pak podle očekávání předčily výsledky toho kratšího (1M). Jedinou výjimkou je odchylka ω_{best} u algoritmu RTSA, ovšem hodnota nejlepšího řešení více podléhá statistické odchylce a může se tak jednat o náhodu. U první sady problémů došlo u všech algoritmů ke zlepšení a je vidět, že se dále snižují rozdíly mezi těmito algoritmy. Nicméně zatímco u druhé sady je při použití algoritmu RTGA zlepšení poměrně výrazné (4,77 %), u algoritmů RTTABU a RTSA se jedná pouze o 2 %, respektive 1,5 %. A to i přes to, že je odchylka vyšší než dvacet procent a mohli bychom očekávat větší zlepšení než u RTGA, kde se s 14,5 % již více blížíme nejlepším výsledkům a můžeme očekávat dosažení limitu implementovaných metod. Je tak ukázáno, že metody RTTABU a RTSA by zřejmě dalším zvyšováním limitu vyhodnocení již především u druhé sady nedosáhly příliš lepších výsledků a uvedené výsledky můžeme považovat za nejlepší, které jsou schopny podat. Naopak algoritmus RTGA vykazuje potenciál dalšího zlepšení, což může být dáno také robustností metody a použitými technikami pro zvyšování diverzity populace.

Pokud porovnáme výsledky tohoto experimentu s předešlým experimentem načítání tabulek, dojdeme k závěru, že u první sady jsou výsledky jen přibližně o půl procenta lepší v případě dlouhého běhu. Podobného efektu, jako je dlouhý běh, tak můžeme dosáhnout u malých instancí i pomocí načtení RT tabulek při startu metody. Naopak u druhé sady, která sama o sobě v experimentu EXPRT14-16 nedopadla příliš dobře, můžeme prohlásit, že delší běh podává výrazně lepší výsledky.

Shrnutí

V závěru této sekce budou zhodnoceny získané poznatky při aplikaci RT metod a bude ukázáno, která metoda dosahuje nejlepších výsledků. Nejprve bylo ověřeno, že je lepších výsledků u metod *RTCluster* a *RTShift* dosaženo vybírání vždy nejhorších charakteristik a také byl ukázán význam vylepšení metody *RTShift*. Dále byly optimalizovány parametry *RTReshuff*, kde v závěru dochází k mutování čtvrtiny tras a jejich rekonstrukci podle nejlepších výsledků uvedených v RT tabulce.

Experimenty EXPRT5-7 ukázaly, že použitím RT metod dochází ke zlepšení výsledků oproti náhodnému přístupu. Je ale nutné zvolit správnou kombinaci těchto metod. Dále je důležité zvolit správnou dobu, než se začnou tyto metody zapojovat, což dokazuje EXP11-13. Celkově byly optimalizovány kombinace metod a počátek jejich nasazení pro všechny algoritmy a bylo dosaženo následujících výsledků: U RTGA došlo ke zlepšení první sady z $\omega_{avg} = 7,36$ % a $\omega_{best} = 4,17$ % u náhodného přístupu na $\omega_{avg} = 4,61$ % a $\omega_{best} = 2,08$ %. U druhé sady pak došlo ke zlepšení z $\omega_{avg} = 20,39$ % a $\omega_{best} = 15,11$ % na $\omega_{avg} = 14,62$ % a $\omega_{best} = 10,06$ %. V podstatě tak bylo závěrečnými průměrnými výsledky dosaženo těch nejlepších u náhodného přístupu, což je velký posun.

U metody RTTABU pak došlo v první sadě ke zlepšení ω_{avg} i ω_{best} o přibližně 10 % a u druhé sady o cca 9 %. U RTSA pak bylo zlepšení o něco menší s 8,5 % u první a 7,5 % u druhé sady. Většího absolutního zlepšení u těchto dvou algoritmů bylo dosaženo, protože podávaly náhodným přístupem daleko horší výsledky, než robustnější RTGA. Můžeme tak říci, že v algoritmech RTTABU a RTSA má použití RT metod větší užitek, ale obecně lepšího výsledku dosahuje algoritmus RTGA.

V práci byl dále proveden experiment zkoumající načtení předvyplněné tabulky při startu algoritmů a spuštění RT metod již od první iterace. Zde se především u první sady instancí podařilo dosáhnout lepších výsledků, proto i myšlenka předzpracování úlohy pro vytvoření vyplněné RT tabulky by mohla mít smysl. Je však otázka, jestli by tato technika podávala lepší výsledky než jednoduché prodloužení běhu algoritmu. Toto bylo zkoumáno posledním experimentem, kde byly zjištěny ještě lepší výsledky, a to především pro druhou sadu problémů.

Celkově bychom mohli zhodnotit, že u menších problémů první sady nebyly nejlepší výsledky použitých tří algoritmů, dosažené při posledním experimentu EXPRT17, příliš rozdílné, kdy ω_{avg} bylo u RTGA **3,23 %**, u RTTABU 4,19 % a v případě RTSA 5,8 %. Vidíme, že nejhorší výsledky podává RTSA, ale rozdíl mezi RTGA a RTTABU je menší než jedno procento. Přitom algoritmus RTGA je mnohem komplexnější a složitější na implementaci. Toto se zřejmě projevuje u druhé sady rozsáhlejších problémů, kde RTGA jasně dominuje s výsledkem $\omega_{avg} =$ **9,85 %** oproti 21,40 % u RTTABU a 27,55 % u RTSA. To může být způsobeno jednak velikostí instancí, tak také jejich geometrickou pravidelností, se kterou mohou mít metody problém.

Závěr

V této práci byly zkoumány optimalizační problémy s hlavním zaměřením na logistickou úlohu *Vehicle Routing Problem* (VRP), která byla v úvodu definována. Tato úloha je matematickým modelem reálných logistických problémů a její řešení tak může mít i praktické využití. Dále byly uvedeny některé možné způsoby řešení těchto problémů, přičemž byl kladen důraz na metaheuristické metody. Některé metaheuristické metody byly rozpracovány pro možnost aplikace na specifický problém VRP. Konkrétně se jedná o: genetické algoritmy (GA), tabu prohledávání (TABU), simulované žíhání (SA) a optimalizaci mravenčí kolonií (ANT). Především u genetického algoritmu pak byla popsána další možná vylepšení běhu metody.

Za nejvýznamnější výsledky práce považuji následující body: Definování a dokázání obecných tvrzení o problému VRP, ve kterých byly rozpracovány teoretické aspekty úlohy. Následně bylo možné se na tyto Lemmy v textu odkazovat a formálně jimi podložit výklad. Dále bylo definováno kódování úlohy VRP tak, aby bylo jednak použitelné nezávisle na algoritmu, a také aby pomocí tohoto kódování bylo možné nalézt optimální řešení. Principy křížení prezentovaném v článku *Multi-objective genetic algorithms for vehicle routing problem with time windows* [26] jsem aplikoval pomocí metody *Best Cost Route Insertion* také na tvorbu počátečních řešení a mutaci. Tyto principy je možné využít jak pro algoritmus GA, tak pro lokální prohledávání v TABU a SA. Operátory všech těchto algoritmů pak využívají v jádru stejnou metodu, což umožňuje porovnání výkonnosti algoritmů jako takových. V práci byla dále implementována experimentální infrastruktura umožňující metodické provádění a vyhodnocování velkého množství experimentů, čehož bylo v práci také využito.

Významným přínosem je také přizpůsobení učící se metody představené pro optimalizační problém *Resource-Constrained Project Scheduling Problem* (RCPSP) v článku *A Dynamic Analysis of Resource-Constrained Project Scheduling Problems for an Informed Search via Genetic Algorithm Optimization* [16]. Tato metoda využívá definované charakteristiky předchozích řešení a bylo tak nutné formulovat charakteristiky řešení úlohy VRP. V této práci byly na základě definovaných charakteristik navrženy tři mutace použitelné pro optimalizaci VRP. Přínos těchto mutací byl potvrzen v experimentech, kdy u všech třech algoritmů bylo možné jejich využitím dosáhnout lepších výsledků, než náhodným přístupem. Je tedy vidět, že tento princip má uplatnění u optimalizační úlohy VRP a může tak mít další potenciál úspěšného nasazení i pro jiné optimalizační problémy.

V experimentech byly nejprve optimalizovány parametry jednotlivých metod a následně zkoumáno použití RT metod pro takto optimalizované algoritmy. Bylo zjištěno, že nejlepší výsledky podává genetický algoritmus (RTGA), který byl zároveň také nejrychlejší. Tento algoritmus navíc vykazoval největší zlepšení při spuštění dlouhého běhu a je vidět, že má nejlepší potenciál pro stálý vývoj řešení. O něco málo horších výsledků pak bylo dosaženo algoritmem RTTABU, který je však implementačně daleko jednodušší než RTGA a může tak také nalézt své uplatnění.

Literatura

[1] Afifi, S.; Dang, D.-C.; Moukrim, A.: A Simulated Annealing Algorithm for the Vehicle Routing Problem with Time Windows and Synchronization Constraints. In *Learning and Intelligent Optimization*, Springer, 2013, s. 259–265.

[2] Alander, J.: On optimal population size of genetic algorithms. In *CompEuro '92 . 'Computer Systems and Software Engineering',Proceedings.*, May 1992, s. 65–70.

[3] Baker, J. E.: Adaptive selection methods for genetic algorithms. In *Proceedings of an International Conference on Genetic Algorithms and their applications*, Hillsdale, New Jersey, 1985, s. 101–111.

[4] Berger, J.; Barkaoui, M.: A hybrid genetic algorithm for the capacitated vehicle routing problem. In *Genetic and Evolutionary Computation—GECCO 2003*, Springer, 2003, s. 646–656.

[5] Bianchi, L.; Dorigo, M.; Gambardella, L. M.; aj.: A Survey on Metaheuristics for Stochastic Combinatorial Optimization. ročník 8, č. 2, Červen 2009: s. 239–287.

[6] Boyd, S.; Vandenberghe, L.: *Convex Optimization*. Cambridge University Press, 2004, ISBN 9780521833783.

[7] Bullnheimer, B.; Hartl, R. F.; Strauss, C.: Applying the Ant System to the Vehicle Routing Problem. In *2nd International Conference on Metaheuristics - MIC97*, 1997.

[8] Cheung, B.-S.; Langevin, A.; Villeneuve, B.: High performing evolutionary techniques for solving complex location problems in industrial system design. *Journal of Intelligent Manufacturing*, ročník 12, č. 5-6, 2001: s. 455–466.

[9] Christofides, N.; Mingozzi, A.; Toth, P.; aj.: *Combinatorial optimization*, kapitola Vehicle Routing Problem. Wiley, 1979, ISBN 9780471997498.

[10] Clausen, J.: Branch and bound algorithms-principles and examples. *Department of Computer Science, University of Copenhagen*, 1999: s. 1–30.

[11] Dorigo, M.; Maniezzo, V.; Colorni, A.: Ant system: optimization by a colony of cooperating agents. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, ročník 26, č. 1, 1996: s. 29–41.

[12] Gendreau, M.; Hertz, A.; Laporte, G.: A tabu search heuristic for the vehicle routing problem. *Management science*, ročník 40, č. 10, 1994: s. 1276–1290.

[13] Goldberg, D. E.; Deb, K.: A comparative analysis of selection schemes used in genetic algorithms. *Urbana*, ročník 51, 1991: s. 61801–2996.

[14] Golden, B.; Wasil, E.; Kelly, J.; aj.: The Impact of Metaheuristics on Solving the Vehicle Routing Problem: Algorithms, Problem Sets, and Computational Results. In *Fleet Management and Logistics*, editace T. Crainic; G. Laporte, Springer US, 1998, ISBN 978-1-4613-7637-8, s. 33–56.

[15] Holland, J. H.: Genetic algorithms. *Scientific american*, ročník 267, č. 1, 1992: s. 66–72.

[16] Hrubý, M.: A Dynamic Analysis of Resource-Constrained Project Scheduling Problems for an Informed Search via Genetic Algorithm Optimization, 2014, v recenzi.

[17] Hynek, J.: *Genetické algoritmy a genetické programování*. Grada Publishing a.s., 2008, ISBN 9788024726953.

[18] Kytöjoki, J.; Nuortio, T.; Bräysy, O.; aj.: An efficient variable neighborhood search heuristic for very large scale vehicle routing problems. *Computers & Operations Research*, ročník 34, č. 9, 2007: s. 2743–2757.

[19] Laporte, G.: The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, ročník 59, č. 2, 1992: s. 231–247.

[20] Laporte, G.: The vehicle routing problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, ročník 59, č. 3, 1992: s. 345 – 358, ISSN 0377-2217.

[21] Laporte, G.; Osman, I. H.: Routing problems: A bibliography. *Annals of Operations Research*, ročník 61, č. 1, 1995: s. 227–262.

[22] Marinakis, Y.; Marinaki, M.: A hybrid genetic–Particle Swarm Optimization Algorithm for the vehicle routing problem. *Expert Systems with Applications*, ročník 37, č. 2, 2010: s. 1446–1455.

[23] Mendes, J. J. d. M.; Gonçalves, J. F.; Resende, M. G.: A random key based genetic algorithm for the resource constrained project scheduling problem. *Computers & Operations Research*, ročník 36, č. 1, 2009: s. 92–109.

[24] Mester, D.; Bräysy, O.: Active-guided evolution strategies for large-scale capacitated vehicle routing problems. *Computers & Operations Research*, ročník 34, č. 10, 2007: s. 2964–2975.

[25] Networking and Emerging Optimalization Research Group, Malaga, Španělsko: Vehicle Routing Problem [online]. `http://neo.lcc.uma.es/vrp/`, Leden 2013, [cit. 2015-01-03].

[26] Ombuki, B.; Ross, B. J.; Hanshar, F.: Multi-objective genetic algorithms for vehicle routing problem with time windows. *Applied Intelligence*, ročník 24, č. 1, 2006: s. 17–30.

[27] Papadimitriou, C.; Steiglitz, K.: *Combinatorial Optimization: Algorithms and Complexity*. Dover Books on Computer Science Series, Dover Publications, 1998, ISBN 9780486402581.

[28] Prins, C.: A simple and effective evolutionary algorithm for the vehicle routing problem. *Computers & Operations Research*, ročník 31, č. 12, 2004: s. 1985–2002.

[29] Rahnamayan, S.; Tizhoosh, H. R.; Salama, M. M.: A novel population initialization method for accelerating evolutionary algorithms. *Computers & Mathematics with Applications*, ročník 53, č. 10, 2007: s. 1605 – 1614, ISSN 0898-1221.

[30] Rao, W.; Jin, C.: A method for analyzing solution space of traveling salesman problem based on complex network. *International Journal of Innovative Computing Information and Control*, 2013: s. 3685–3700.

[31] Rexhepi, A.; Maxhuni, A.; Dika, A.: Analysis of the impact of parameters values on the Genetic Algorithm for TSP. *International Journal of Computer Science Issues (IJCSI)*, ročník 10, č. 1, 2013.

[32] Ruan, N.: An interesting cryptography study based on knapsack problem. In *Computer Modelling and Simulation (UKSim), 2013 UKSim 15th International Conference on*, IEEE, 2013, s. 330–334.

[33] Russell, S.; Norvig, P.: *Artificial Intelligence: A Modern Approach*. Prentice Hall series in artificial intelligence, Prentice Hall, 2010, ISBN 9780136042594.

[34] Sahni, S.: Approximate algorithms for the 0/1 knapsack problem. *Journal of the ACM (JACM)*, ročník 22, č. 1, 1975: s. 115–124.

[35] Schulze, M. A.: Linear programming for optimization. *Perceptive Scientific Instruments, Inc*, 1998.

[36] Solos, I. P.; Tassopoulos, I. X.; Beligiannis, G. N.: A Generic Two-Phase Stochastic Variable Neighborhood Approach for Effectively Solving the Nurse Rostering Problem. *Algorithms*, ročník 6, č. 2, 2013: s. 278–308.

[37] Tendresse, I.; Gottlieb, J.; Kao, O.: The Effects of Partial Restarts in Evolutionary Search. In *Artificial Evolution*, *Lecture Notes in Computer Science*, ročník 2310, editace P. Collet; C. Fonlupt; J.-K. Hao; E. Lutton; M. Schoenauer, Springer Berlin Heidelberg, 2002, ISBN 978-3-540-43544-0, s. 117–127.

[38] Toth, P.; Vigo, D.: *The Vehicle Routing Problem*. Monographs on Discrete Mathematics and Applications, Society for Industrial and Applied Mathematics, 2002, ISBN 9780898715798.

[39] Toth, P.; Vigo, D.: The granular tabu search and its application to the vehicle-routing problem. *Informs Journal on computing*, ročník 15, č. 4, 2003: s. 333–346.

[40] Weglarz, J.: *Project Scheduling: Recent Models, Algorithms and Applications*. International Series in Operations Research & Management Science, Springer US, 2012, ISBN 9781461555339.

[41] Wink, S.; Back, T.; Emmerich, M.: A meta-genetic algorithm for solving the Capacitated Vehicle Routing Problem. In *Evolutionary Computation (CEC), 2012 IEEE Congress on*, June 2012, s. 1–8.

Seznam použitých zkratek a symbolů

zkratka	celý název	vysvětlení
TSP	Travelling Salesman Problem	Problém obchodního cestujícího.
VRP	Vehicle Routing Problem	Problém logistiky vozidel.
CVRP	Capacitated Vehicle Routing Problem	Problém logistiky vozidel s kapacitním omezením.
RCPSP	Resource-Constrained Project Scheduling Problem	Problém rozvrhování výrobních operací.
GA	Genetic Algorithm	Genetický algoritmu.
TABU	Tabu Search	Tabu vyhledávání.
SA	Simulated Annealing	Simulované žíhání.
ANT	Ant Colony Optimization	Optimalizace mravenčí koloní.
BCRI	Best Cost Route Insertion	Algoritmus nejlepšího vložení zákazníků do trasy.
BCRC	Best Cost Route Crossover	Křížení využívající BCRI.
BCRM	Best Cost Route Mutation	Mutace využívající BCRI.