VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

OPTIMALIZACE ÚLOH TYPU TSP TECHNIKAMI KOLEKTIVNÍ VÝPOČETNÍ INTELIGENCE

OPTIMISATION OF TSP-BASED TASKS USING COLLECTIVE COMPUTATIONAL INTELLIGENCE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. JAROMÍR FRANĚK**
AUTHOR

VEDOUCÍ PRÁCE **doc. Ing. MICHAL BIDLO, Ph.D.**
SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

153685

Ústav:	Ústav počítačových systémů (UPSY)
Student:	**Franěk Jaromír, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Bioinformatika a biocomputing
Název:	**Optimalizace úloh typu TSP technikami kolektivní výpočetní inteligence**
Kategorie:	Umělá inteligence
Akademický rok:	2023/24

Zadání:

1. Seznamte se s problematikou úloh typu Travelling Salesman Problem (TSP) a možnostmi jejich řešení pomocí přírodou inspirovaných algoritmů. Zaměřte se na algoritmy z kategorie kolektivní výpočetní inteligence.
2. Prostudujte existující možnosti reprezentace a optimalizace různých instancí TSP pomocí algoritmů z předchozího bodu a zpracujte přehledovou studii na toto téma.
3. Zvolte alespoň dva různé koncepty algoritmů z předchozího bodu a tyto implementujte ve vhodném prostředí. Uvažujte různá nastavení algoritmů, případně reprezentací řešení TSP.
4. Proveďte sadu experimentů za účelem co nejkvalitnějšího vyladění parametrů použitých technik a zpracujte srovnávací studii optimalizace TSP s ohledem na vybraná sledovaná kritéria (např. délka výsledných tras TSP, výpočetní náročnost apod.). Snažte se o dosažení kvalitních výsledků pro co největší velikosti instancí TSP.
5. Zhodnoťte dosažené výsledky a diskutujte přínosy a možnosti rozšíření vašeho řešení.

Literatura:

- Dle pokynů vedoucího projektu.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání, demonstrace rozpracovaného řešení z bodu 4.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Bidlo Michal, doc. Ing., Ph.D.**
Vedoucí ústavu:	Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání:	1.11.2023
Termín pro odevzdání:	17.5.2024
Datum schválení:	30.10.2023

Abstrakt

Cílem této práce je porovnání algoritmů kolektivní výpočetní inteligence pro optimalizaci rozsáhlých instancí problémů obchodního cestujícího obsahující až několik tisíc měst. Tato práce bude zaměřená na optimalizace, které jsou založené na chování hejna a optimalizace, které jsou založené na chování včelího roje. V sadě experimentů jsem porovnal jednotlivé algoritmy z těchto dvou přístupů s různými parametry. Podle těchto výsledků jsem navrhl vlastní vylepšení algoritmů pro řešení daného problému. Dále jsem navrhl vylepšení algoritmů o metody podpory diversity, nebo lokálního prohledávání pro diskrétní verze těchto algoritmů. Na finální sadě experimentu jsem porovnal výsledky navržených algoritmů.

Abstract

The aim of this work is to compare collective computational intelligence algorithms for the optimization of large-scale instances of traveling salesman problem consisting of in several of thousands of cities. This work will be focused on optimization based on swarm behavior and optimization based on bee swarm behavior. I have compared individual algorithms from these two approaches with different parameters within a set of experiments. Based on these results, I proposed an improvement to the algorithms for solving the given problem. Then I proposed an improvement to the algorithms using diversity support methods, or local search for discrete versions of these algorithms. Finally, I compared results of proposed algorithms within a final set of experiments.

Klíčová slova

heuristické algoritmy, kolektivní inteligence, optimalizace chováním hejna, optimalizace umělým včelím rojem, problém obchodního cestujícího, optimalizační algoritmy, k-opt

Keywords

heuristic algorithms, swarm intelligence, particle swarm optimalization, artificial bee colony optimalization, traveling salesman problem, optimization algorithms, k-opt

Citace

FRANĚK, Jaromír. *Optimalizace úloh typu TSP technikami kolektivní výpočetní inteligence.* Brno, 2024. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. Michal Bidlo, Ph.D.

Optimalizace úloh typu TSP technikami kolektivní výpočetní inteligence

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Michala Bidla Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Jaromír Franěk
9. května 2024

Poděkování

Rád bych poděkoval Ing. Michalu Bidlovi Ph.D. za podnětné návrhy a cenné rady k této práci, bez kterých by nemohla být v této podobě vypracována.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Problém obchodního cestujícího se zabývá hledáním nejkratší cesty mezi jednotlivými městy tak, aby bylo každé město navštíveno právě jednou a cesta ukončena v počátečním městě. Jedná se o poměrně známý problém, jehož řešení se dá aplikovat v různých oblastech. Mezi nejčastější oblasti jeho využití patří vrtání desek plošných spojů, generální opravy motorů s plynovou turbínou, rentgenová krystalografie a směrování vozidel. [7] Stejně jako existuje mnoho aplikací, také existuje mnoho přístupů k řešení tohoto problému. Přesné algoritmy pro menší počty měst, v opačných případech heuristické nebo aproximační algoritmy, které sice nedosáhnou nejlepšího řešení, ale slouží pro dosáhnutí dostatečně kvalitního řešení v rozumném časovém rámci. Přístup kolektivní výpočetní inteligence spadá do heuristických algoritmů a na ten bude tato práce zaměřena. Tyto přístupy zde budou popsány s ilustracemi jejich rozdílů.

Cílem této práce je porovnání různě vybraných algoritmů kolektivní výpočetní inteligence pro optimalizaci rozsáhlých instancí problému obchodního cestujícího o velikostech v tisících měst. Je zde kladen důraz na optimalizace založené na chování hejna a optimalizace založené na chování včelího roje. Na sadě experimentů jsem porovnal jednotlivé algoritmy z těchto dvou přístupů s různými parametry. Podle předběžných výsledků jsem navrhnul vlastní vylepšení algoritmu, které zjednodušuje operaci výměny. Výsledky jsou také porovnány s výsledky algoritmů z předchozí sady experimentů. Dále jsem navrhl vylepšení algoritmů o metody podpory diverzity a lokálního prohledávání pro diskrétní verze těchto algoritmů. Podpora diverzity je zde důležitá, protože pro diskrétní optimalizace založené na chování hejna neexistuje mnoho studií, které se jí zabývají. Nakonec jsem vyhodnotil nejlepší parametry těchto algoritmů a výsledné délky cest porovnal s nejlepšími známými délkami cest pro tyto instance problému.

Kapitola 2

Problém obchodního cestujícího

Problém obchodního cestujícího, také zkráceně označován jako TSP (Travelling Salesman Problem), je problém, pro jehož řešení musíme najít nejkratší trasy v sadě měst. Tedy cestu, kde každé město musíme navštívit právě jednou a na konci se vrátit do počátečního města. V této kapitole je popsána definice problému, reprezentace, aplikace a problémy spojené s jeho řešením.

2.1 Definice TSP

Problém je NP-úplný, což znamená, že pro problém obchodního cestujícího časová složitost řešení roste exponenciálně v závislosti na velikosti instance. Neexistuje žádný přesný algoritmus, který by dokázal v rozumném čase nalézt řešení pro velikou instanci problému. Součástí je konečná množina měst $M = 1, 2, ...n$ a vzdáleností mezi městy. Cílem je nalézt takovou permutaci měst, pro kterou je součet vzdáleností od počátečního města do koncového minimální. Lze si jej představit jako graf $G = (V, A)$. Jedná se o uspořádanou dvojici, kde V je neprázdnou množinou vrcholů a A je množina hran. Hrana je dvoubodová podmnožina množiny V. [19]

Obrázek 2.1: Příklad problému s cílem získat nejkratší cesty propojující všech 49 sousedících států USA. Převzato z https://medium.com/stanford-cs224w

Cílem objektivní funkce je najít nejkratší Hamiltonové trasy. Tedy nejmenší délky cesty procházející každým vrcholem pouze jednou, s výjimkou počátečního, ve kterém začínáme i končíme cestu. Zde uvedená objektivní funkce a omezení jsou převzaté z [27] a formulované Dantzigem, Fulkersonem a Johnsonem.

$$Z = \min \sum_{i=1}^{n} \sum_{i=1, j \neq i}^{n} d_{ij} * x_{ij} \tag{2.1}$$

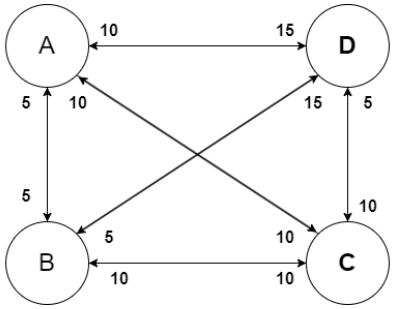
Kde platí omezení:

- každé město může být navštíveno pouze jednou,
  $\sum_{i=1, j \neq i}^{n} x_{ij} = 1, \ \ \forall i = 1, 2, ...n$
- každé město může být opuštěno pouze jednou,
  $\sum_{i=1, j \neq i}^{n} x_{ij} = 1, \ \ \forall j = 1, 2, ...n$
- cesta městy je kompletní a neexistují podcesty.
  $\sum_{i \in 1} \sum_{j \in 1} x_{ij} \leq |S| - 1; \ \ \forall S \subset \{2, ..., n\}, \ \ |S| \geq 2$

2.2 Varianty TSP

V této sekci převzaté z článku [13], jsou popsány možné varianty a modifikace TSP.

První možnou modifikací je asymetrický TSP. V takovém případě pracujeme s grafem, kde se může vyskytovat jedna nebo více cest, kde délka cesty z i do j je jiná než z j do i. Je nutné vzít na vědomí, že pak existuje víc řešení, protože směr cesty ovlivní její délku. Pokud se zároveň jedná o plně propojený graf, pak se existence řešení dá garantovat.



Obrázek 2.2: Příklad asymetrického TSP se třemi asymetrickými cestami mezi vrcholy (A, D), (B, D) a (C, D).

Pokud není řečeno jinak, pak se předpokládá, že se jedná o plně propojený symetrický TSP. Existují ale také varianty a modifikace, které neodpovídají tomuto předpokladu. Další z možných modifikací je neúplný graf, kde nejsou všechna města plně propojena. Jedním ze způsobů řešení této varianty je nastavit chybějícím hranám nejvyšší možnou vzdálenost. Po dokončení běhu algoritmu pro standartní TSP bude zkontrolováno, zda řešení obsahuje některou z těchto hran. Pokud ne, pak bude prohlášeno za optimální. Není garantováno, že takový graf má řešení, pokud ano, pak nelze garantovat jeho nalezení. Druhou možností je zrušit omezení pro navštívení a opuštění města pouze jednou. Dovolením návštěvy měst více jak jednou a průchodu hranami více jak jednou, se dá garantovat, že řešení existuje.

Problém je možné také modifikovat přítomností více obchodních cestujících. Jedná se o rozšíření problému povolením více jak jednoho prodejce. Sada měst bude obsahovat jedno depo, kde se prodejci nacházejí a nákladovou metriku. Cílem je určit soubor tras pro m prodejců tak, aby se minimalizovaly celkové náklady m tras. Metrika nákladů může představovat náklady, vzdálenost i čas.

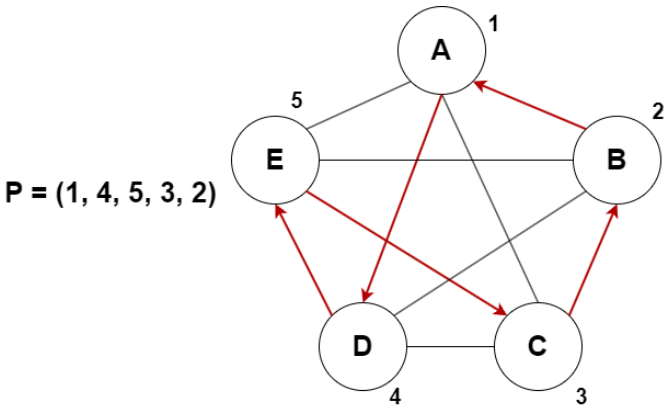
2.3 Reprezentace výsledných řešení TSP

Existuje několik možných reprezentací výsledné cesty. Vynecháme například grafické, které jsou názorné, ale není je možné využít v kódu algoritmu. Níže uvedené reprezentace cest jsou derivovány z reprezentace v článku [1] a článku zabývající se aplikací genetického algoritmu pro řešení TSP [24].

2.3.1 Reprezentace výsledné cesty permutací měst

Reprezentace permutací měst představuje způsob zápisu možné cesty TSP, založený na vyjádření pořadí, v jakém jsou města seřazena. Lze definovat jako P je permutace o délce n. Jedná se o nejpoužívanější a nejjednodušší přístup pro reprezentaci měst, popisující v jakém pořadí jsou města navštívena.

Například pro množinu 5 měst (A, B, C, D, E) označenými písmeny (1, 2, 3, 4, 5). Jedna z možných cest je znázorněná obrázkem 2.3, který znázorňuje permutaci $P = (1, 4, 5, 3, 2)$. Ta představuje cestu, která začíná v městě A, prochází D, následně E, poté C a nakonec B, potom se vrátí na počátek do A. Pořadí měst je určeno pozicí jejich indexů v permutaci.

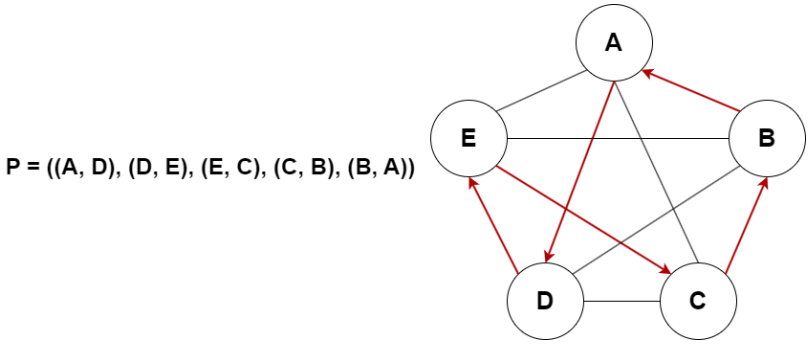


Obrázek 2.3: Příklad reprezentace permutací se znázorněním písmen označujícím město v permutaci.

2.3.2 Reprezentace výsledné cesty permutací hran

Místo posloupnosti měst se jedná o posloupnost hran. Výsledná permutace tedy bude obsahovat dvoubodové podmnožiny měst. Reprezentace sousedních hran poskytuje způsob, jak popsat cestu v grafu zadáním pořadí, ve kterém jsou hrany procházeny.

Příkladem je znovu 5 měst (A, B, C, D, E). Cesta z města A do B by byla označena (A, B). Znázornění permutace představující cestu, jako na příkladu výše, by vypadalo takto $P = ((A, D), (D, E), (E, C), (C, B), (B, A))$.



Obrázek 2.4: Příklad reprezentace permutací hran.

2.3.3 Reprezentace výsledné cesty binární maticí

Pro výše uvedené reprezentace existují také verze s binární reprezentací, které jsou ilustrovány binární maticí o rozměrech $n \times n$, kde n je počet měst v daném problému TSP. V případě reprezentace měst se udává pořadí tak, že pokud se $M_{ij} = 1$, tak město i je v daném řešení na pozici j. V případě reprezentace hran, pokud $M_{ij} = 1$, pak jsou města i a j propojena. V reprezentaci hran není znázorněné počáteční město.

	1	2	3	4	5
A	1	0	0	0	0
B	0	0	0	0	1
C	0	0	0	1	0
D	0	1	0	0	0
E	0	0	1	0	0

Obrázek 2.5: Příklad binární reprezentace měst zobrazující cestu $P = (1, 4, 5, 3, 2)$. Řádky představují města a sloupce jejich pozice.

Jako výslednou reprezentaci jsem zvolil reprezentaci cesty permutací měst, nicméně implementované algoritmy uvedené v kapitole 4 mohou mezi výpočty převádět cestu do jiné reprezentace pro využití operací, které by na tuto reprezentaci nešlo aplikovat. Tyto převody budou u každého algoritmu popsány.

2.4 Algoritmy pro optimalizaci TSP

Existují algoritmy s různými přístupy pro řešení TSP, včetně přesných, heuristických a aproximačních algoritmů. Přesné algoritmy najdou optimální řešení, ale jsou proveditelné pouze pro menší velikosti instancí TSP. Heuristické algoritmy dokážou zpracovat větší instance problému, ale nemohou zaručit nalezení optimálního řešení. Aproximační algoritmy vyvažují rychlost a optimálnost pro zisk řešení v rámci určitého procenta optimálního řešení. Tato sekce je převzata z článku. [21]

2.4.1 Přesné algoritmy

Nejjednodušším přístupem k vyřešení TSP je použití hrubé síly, která zahrnuje zkoušení všech možných permutací cest a výběr nejkratší. Doba běhu této metody je však $\mathcal{O}(n!)$, tj. úměrná faktoriálu počtu měst, což znamená, že se rychle stává nepraktickou i pro malé instance TSP.

Dynamické programování je další technika, kterou lze použít k řešení TSP. Jeden z prvních dynamicky programovaných algoritmů pro TSP je Held-Karpův algoritmus, který má časovou složitost $\mathcal{O}(n^2 2^n)$. Zlepšení časové složitosti je však poměrně náročné.

Ambainis navrhl přesný algoritmus pro TSP s časovou složitostí $\mathcal{O}(1,728^n)$. Tento algoritmus je v současnosti nejlepším algoritmem pro přesné řešení TSP, ale časová složitost je stále příliš vysoká pro řešení velkých instancí TSP.

2.4.2 Heuristické algoritmy

Heuristika je algoritmus „šitý na míru“, danému problému vzhledem ke znalostem, který o něm máme. Heuristiky obecně nezaručují nalezení optimálního řešení. Mezi heuristické algoritmy pro TSP patří například algoritmus dvou porovnání a spárování (Match Twice and Stitch, MTS) a Lin-Kernighanova heuristika.

Heuristika MTS se skládá ze dvou fází. V první fázi, známé jako konstrukce cyklu, se provádějí dvě sekvenční párování pro vytvoření cyklů. První párování vrátí sadu hran s nejmenší délkou, kde každý bod spadá přesně do jedné odpovídající hrany. Dále jsou všechny tyto hrany odstraněny a je provedeno druhé spárování. Druhé párování opakuje proces párování s výhradou, že žádná z hran vybraných v prvním párování nemůže být znovu použita. Výsledky prvního a druhého párování společně tvoří sadu cyklů. Zkonstruované cykly jsou ve druhé fázi spojeny dohromady, aby vytvořily cestu TSP. [14]

Lin-Kernighan je lokální prohledávací algoritmus a jedna z nejúčinnějších heuristik pro řešení TSP. Jako vstup bere dosud nejlepší nalezenou cestu a snaží se ji vylepšit prozkoumáním sousedních cest. Tento proces se opakuje, dokud není dosaženo lokálního minima. Algoritmus používá podobný koncept jako algoritmy pro lokální prohledávání k-opt. Lin-Kernighan vytváří nové trasy přeskupením segmentů staré cesty, někdy i obrácením směru procházení. Je adaptivní a nemá pevný počet hran k nahrazení v kroku, ale má tendenci nahrazovat malý počet hran najednou. [17]

2.4.3 Aproximační algoritmy

Cílem aproximačního algoritmu je získat řešení, které nemusí být nutně optimální, ale je mu velmi blízké. Mezi aproximační algoritmy pro TSP patří například algoritmus pro nalezení nejbližších sousedů (NN) a Christofides-Serdyukův algoritmus.

Algoritmus NN je přístup, kde si obchodník jako další destinaci vybere nejbližší nenavštívené město. Tato metoda rychle vytváří přiměřeně krátkou trasu. Pro n měst náhodně umístěných v rovině algoritmus generuje cestu, která je o 25% delší než nejkratší možná cesta. Některé městské distribuce jsou však navrženy tak, že na nich algoritmus NN nefunguje optimálně, což platí pro symetrické i asymetrické TSP. [10]

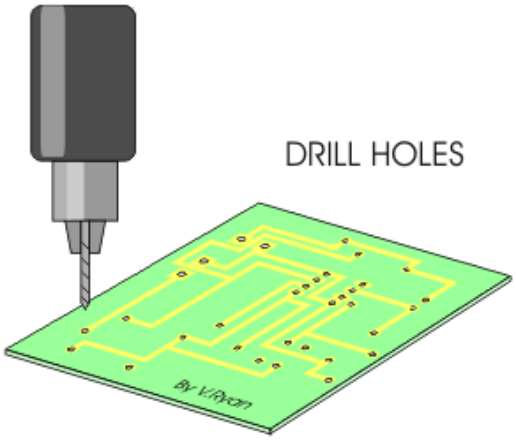
Algoritmus Christofidesů je dalším aproximačním algoritmem pro TSP, který využívá jak minimální kostru, tak řešení problému shody s minimální váhou. Kombinací těchto dvou řešení je algoritmus schopen vytvořit cestu TSP, která je maximálně $1,5x$ delší než optimální trasa. [29]

2.4.4 Metaheuristické algoritmy

Na tento přístup k řešení TSP je zaměřena tato práce. Vzhledem k nedeterministické povaze TSP je hledání optimálních řešení s rostoucí velikostí instancí stále složitější. Prozkoumání celého prostoru platných řešení je výpočetně nákladné a v závislosti na velikosti daného problému může být dokonce nemožné. V posledních letech byly navrženy různé metaheuristické přístupy, jako je optimalizace mravenčích kolonií (ACO), genetické algoritmy (GA), optimalizace hejnem částic (PSO), simulované žíhání (SA), vyhledávání tabu (TS) a biologicky inspirovaná optimalizace. Tyto přírodou inspirované metaheuristické algoritmy se skládají ze dvou klíčových prvků: vykořisťování a průzkum. Průzkum často využívá randomizaci jako svou hnací sílu, zatímco vykořisťování využívá místní znalosti, jako jsou gradienty a historie hledání, aby se soustředilo na místní region. Dobrá rovnováha těchto složek vede k dobré účinnosti algoritmu za vhodných podmínek. [28]

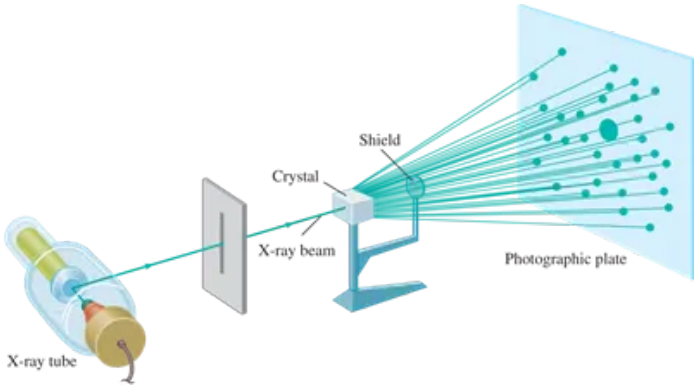
2.5 Aplikace TSP

Přímá aplikace TSP je uplatněná v problému vrtání desek plošných spojů. Pro spojení vodiče na různých vrstvách a pro umístění pinů integrovaných obvodů je třeba provrtat otvory skrz desky. Pokud mají různé velikosti, pak se po vyvrtání dvou otvorů různých průměrů musí hlava stroje přesunout do skříně a změnit vrtací nástroj. To je dost časově náročné. Je tedy nutné zvolit průměr, vyvrtat všechny otvory stejného průměru, vyměnit vrták, zopakovat předchozí postup, dokud nejsou všechny otvory vyvrtány. Na tento problém lze tedy pohlížet jako na sadu TSP problémů, seřazených za sebou. Vzdálenost mezi dvěma otvory je dána dobou, kterou trvá přesunutí vrtací hlavy z jedné pozice do druhé. Cílem je minimalizovat dobu cestování hlavy stroje.



Obrázek 2.6: Vrtání plošného spoje, kde se hlavice posouvá po spoji a na označených místech provrtává otvory. Převzato z https://technologystudent.com/pcb/pcb4a.htm

Další aplikací je analýza struktury krystalů v rentgenové krystalografii. Zde detektor měří intenzitu rentgenových odrazů krystalu z různých pozic. Samotné měření může být dokončeno poměrně rychle, je zde ale značná režie v době polohování, kde je třeba realizovat stovky tisíc pozic. Výsledek experimentu zde nezávisí na pořadí pozic, ale celkový čas závisí na pořadí těchto pozic. Problém tedy spočívá v nalezení pořadí, které minimalizuje celkový čas. [23]



Obrázek 2.7: Měření intenzity rentgenových odrazů krystalu v rentgenové krystalografii. Obrázek je převzat ze stránky https://macromoltek.medium.com/what-is-x-ray-crystallography-1e186bc3d180

TSP má mnoho dalších aplikací v různých oblastech, například:

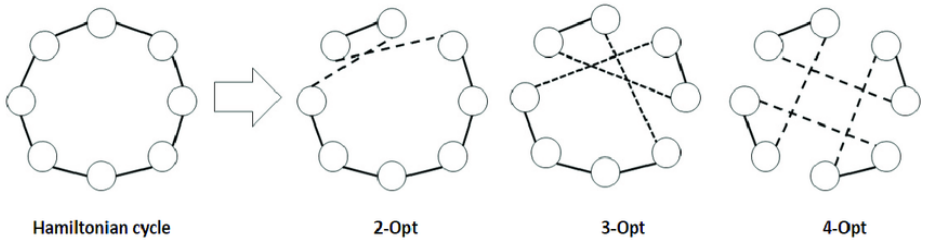
- logistika a řízení dodavatelského řetězce - používá se k optimalizaci tras dodávek a minimalizaci cestovních nákladů na logistiku,
- sekvenování DNA - TSP slouží k určení pořadí, ve kterém se má fragmentovat molekula DNA, aby se získala její úplná sekvence,
- návrh sítě - TSP lze použít k návrhu optimálních sítí pro telekomunikační, dopravní a další systémy,
- výroba a plánování výroby - slouží k plánování a k optimalizaci pořadí, ve kterém jsou různé úkoly prováděny, a snížení výrobních nákladů,
- zpracování obrazu a videa - používá se k optimalizaci pořadí, ve kterém jsou zpracovávány různé části obrazu nebo videa.

2.6 Optimalizace TSP pomocí k-opt

Jedná se o algoritmus aplikující lokální prohledávání, které upravuje spojení mezi městy pro nalezení lepšího řešení. Hodnota k určuje počet hran, které se mají v každém pohybu upravit. Běžně se používá v heuristických a metaheuristických algoritmech pro TSP, jejichž cílem je najít lepší cesty iterativně. Přístup k-opt je běžnou strategií pro zlepšení kvality řešení v TSP, zejména v algoritmech lokálního prohledávání. Efektivita metody se může lišit v závislosti na charakteristikách instance problému.

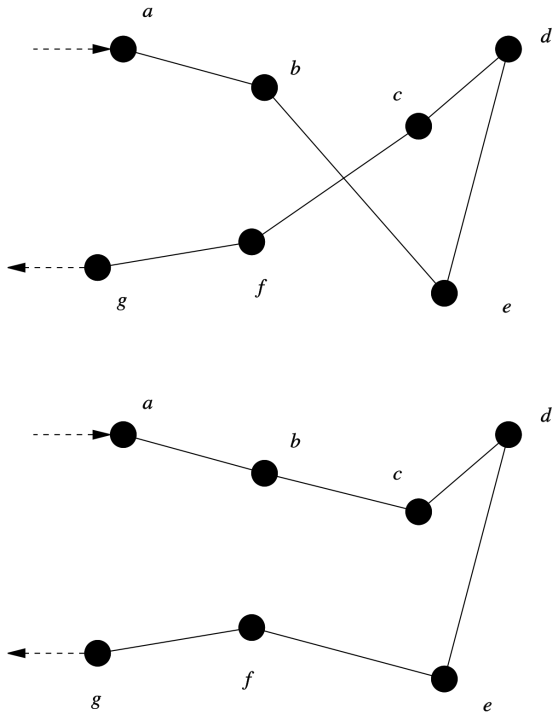
Metody 2-opt a 3-opt jsou výhodné v tom, že provedení a implementace jsou relativně snadné a metody lze kdykoli ukončit. Například 2-optový algoritmus je jednoduchý iterativní přístup, kde v každé iteraci strategicky nahradí dvě hrany za dříve nepropojené. Cílem je eliminovat křížení na cestě pro snížení celkové vzdálenosti. 3-op metoda je podobná 2-opt, ale rozpojí 3 hrany místo 2, které vymění za nejlepší způsob propojení těchto hran. Počet 3 hraných možností propojení je větší než u 2-op. Existují také 4-op, 5-opt, ale počty kombinací hran se exponenciálně zvyšují s rostoucím počtem k, proto se implementace s k větším jak 4 využívají jen v okrajových případech. [22]

Příklad rozpojení a nahrazení hran je znázorněn na obrázku 2.8. Není znázorněn počet kombinací, které tyto algoritmy vyzkouší.



Obrázek 2.8: Ukázka aplikace na Hamiltonovským kruhu. Čárkovanou čarou je znázorněno nové spojení. Převzato z https://www.researchgate.net/figure/K-Opt-Algorithm-Family-Here-the-dashed-lines-represent-the-new-edges-of-the-cycle_fig1_365453089

Na obrázku 2.9 je zobrazeno jak 2-opt upraví existující cestu (a, b, e, d, c, f, g) na cestu (a, b, c, d, e, f, g). Eliminací hran mezi $b => e$ a $c => f$ a následným vytvořením hran $b => c$ a $e => f$. Tím je celková délka cesty zkrácena a jedná se tedy o lepší řešení.



Obrázek 2.9: Ukázka aplikace 2-opt eliminací křížení. Obrázek byl převzat z https://en.wikipedia.org/wiki/2-opt

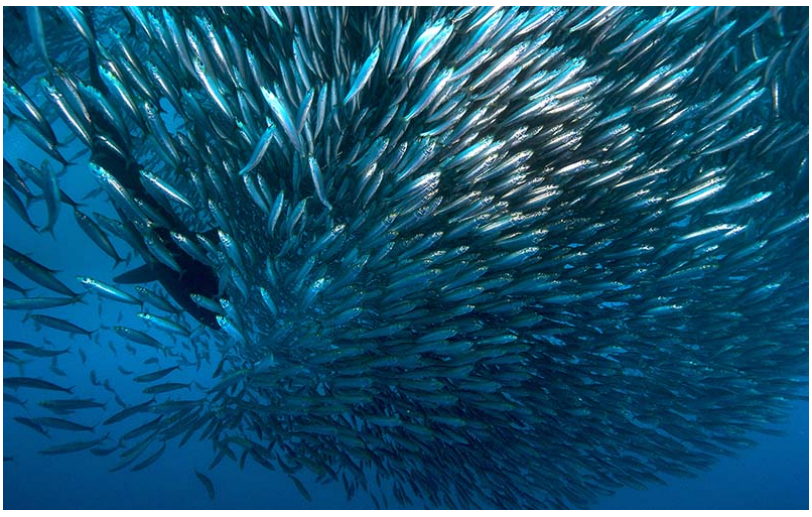
Kapitola 3

Kolektivní výpočetní inteligence

Kolektivní výpočetní inteligence zahrnuje třídu metaheuristických algoritmů inspirovaných chováním vybraných přírodních systémů. V této kapitole je vysvětlena kolektivní výpočetní inteligence. Dále jsou popsány principy algoritmů inspirovaných chováním ptačích hejn a včelích rojů. Přesné popisy diskrétních verzí těchto algoritmů, které jsem implementoval, budou popsány až v následující kapitole 4.

3.1 Princip kolektivní výpočetní inteligence

Jedná se o kolektivní chování decentralizovaných samoorganizovaných systémů. V kontextu této práce se bude jednat o umělé systémy napodobující systémy reálné. Tyto systémy se skládají z populace jednoduchých částic interagujících lokálně mezi sebou navzájem a se svým prostředím. Inspirace přichází z přírody. Částice se řídí velmi jednoduchými pravidly a neexistuje pro ně žádná centralizovaná řídicí struktura. Místní, a to do určité míry náhodné interakce mezi částicemi vedou ke vzniku „inteligentního“ globálního chování. Příklady inteligence hejna v přírodních systémech zahrnují kolonie mravenců, včelstva, hejna ptáků, lov jestřábů, pasení zvířat, růst bakterií, chov ryb a mikrobiální inteligenci. [11]



Obrázek 3.1: Ukázka hejna ryb, které využívají kolektivní chování pro zmatení mořských predátorů. Obrázek byl převzat z https://www.ogsociety.org/journal/featured-articles/351-super-swarms.html

Techniky založené na principu kolektivní výpočetní inteligence lze použít v široké řadě aplikací. Například NASA zkoumá její využití pro mapování planet. Práce M. Anthony Lewise a George A. Bekeye z roku 1992 pojednává o možnosti použití inteligence hejna k ovládání nanobotů v těle za účelem zabíjení nádorů. [20] Hlavní oblasti pro její využití jsou:

- robotika - navrhování skupin robotů, které mohou spolupracovat,
- optimalizace provozu - simulace dopravního proudu v městských oblastech,
- finanční modelování - optimalizace portfolia, řízení rizik a předpovídání tržních trendů,
- řízení dodavatelského řetězce - zlepšením řízení zásob, distribučních tras a prognózování poptávky,
- teorie her - modelování strategických interakcí mezi autonomními agenty,
- lékařská analýza - analýza lékařských snímků pomocí segmentace a klasifikace.

Existuje veliké množství algoritmů, které jsou inspirovány existujícími roji v přírodě a jejich chováním. Mezi nejznámější algoritmy spadající pod tuto oblast patří mravenčí algoritmy, včelí algoritmy, algoritmy simulující chování hejna.

3.2 Optimalizace založená na chování hejna

Optimalizace založená na chování hejna, také označována jako PSO, je výkonný metaheuristický optimalizační algoritmus inspirovaný chováním hejna ptáků v přírodě. Jedná se o simulaci zjednodušeného sociálního systému. Jednotlivý jedinci mají omezený rozsah pozorovatelné oblasti, ale jsou schopni získat další informace z interakce s ostatními jedinci v hejnu. Díky tomu se pak můžou hromadně pohybovat a společně se přesunovat a hledat potravu.

Algoritmus PSO byl poprvé publikován v roce 1995 Jamsem Kennedy a Russellem Eberhartem [16]. V této sekci je demonstrován PSO algoritmus v jeho základní verzi spolu se způsobem jeho diskretizace pro řešení TSP.

3.2.1 Formulace PSO

Jednotlivé částice tvoří hejno, které nazýváme populací. Každá částice prohledává prostor a snaží se dosáhnout nejlepšího řešení. Prohledávaný prostor je tedy sadou řešení, ve kterém částice představuje specifické řešení pomocí své pozice skládající se z N souřadnic v N-rozměrném prostoru.

Na začátku algoritmu se inicializuje populace. Dále je v iteracích prohledáván prostor, kde každá částice prohledává prostor na základě dostupných informací se zavedením lehké náhodnosti. Pro výpočet rychlosti si částice uchovává své (lokální) nejlepší dosažené řešení, které je označováno jako pBest (particle/local best). Dále také zná nejlepší dosud dosažené řešení v hejnu označované jako gBest (global best). Podle těchto parametrů upraví rychlost letu. Následně se pomocí získané rychlosti vypočítá nová pozice, na kterou se částice přesune. Kvalita této pozice je následně ohodnocena pomocí fitness funkce, pokud je lepší než lokální nebo globální řešení, tak ho nahradí.

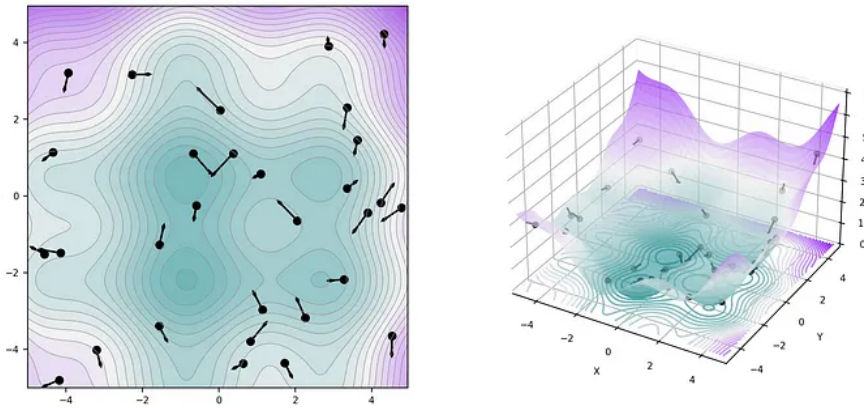
Hejno částic pokračuje v pohybu směrem k slibné oblasti, dokud není přesáhnut počet iterací, nebo už nedochází k podstatnému zlepšení. [8]

Kroky algoritmu

- 1. Inicializace populace představuje nastavení výchozí pozice, rychlosti a nastavení této pozice jako nejlepší lokální řešení. Také se nastaví sociální koeficienty a nejlepší globálního řešení.
- 2. Pro každou částici se vypočítá nová rychlost a následně pozice.
- 3. Pozice je ohodnocena pomocí fitness funkce, pokud je lepší než pBest/gBest, tak se touto pozicí nahradí.
- 4. Kontrola ukončující podmínky, kterou je většinou dosažení maximálního počtu iterací nebo minimální zlepšení nejlepšího nalezeného řešení. Pokud není splněna, pokračuje se znovu krokem 2.

Inicializace

Inicializace je zásadní krok, protože ovlivňuje chování algoritmu při průzkumu a konvergenci. Na začátku je vytvořena populace částic, počet ovlivní rychlost konvergence a velikost prostoru, který bude možné prohledat. Je ale nutné vzít v potaz, že to také zvýší dobu průběhu algoritmu. Každé částici je přiřazena náhodně vygenerovaná pozice X_i, pro $i = 1, 2, ..., N$, kde N představuje počet částic. Krom náhodného generování existují taky metody generace pro lepší pokrytí prohledávaného prostoru. Pozice X_i reprezentuje vektor $X_i = (x_0, x_1, ..., x_m)$, kde m je počet dimenzí prohledávaného prostoru. Rychlost částice se zpravidla generuje náhodně, ale také je možné ji inicializovat jako nulový vektor. Podobně jako pozice reprezentuje vektor $V_i = (v_0, v_1, ..., v_m)$, kde m je opět počet dimenzí prohledávaného prostoru. Poté se nastaví vektor X_i jako nejlepší nalezené lokální řešení pBest. Nakonec se nastaví gBest ohodnocením částic a zvolením nejlepšího.



Obrázek 3.2: Ukázka inicializace PSO v prostoru, kde tečky představují částice rozmístěné v prostoru. Šipky představují směr a velikost rychlosti. Převzato z https://towardsdatascience.com

Průběh algoritmu

Po dokončení inicializace algoritmu se provádí iterace, při kterých pro každou částici vypočte pozici, tu ohodnotí a následně aktualizuje dosud nejlépe nalezené řešení. Na začátku iterace je pro každou částici rychlost aktualizována formulí 3.1:

$$V_i^{t+1} = w * V_i^t + c_1 * r_1 * (P_i^t - X_i^t) + c_2 * r_2 * (G^t - X_i^t) \tag{3.1}$$

Kde w představuje velikost rychlosti předchozí iterace. Pokud je rovna jedné, pak se přičte celá rychlost částice z předchozí iterace. Pokud je naopak nulová, pak nemá žádný vliv na výslednou hodnotu. Sociální koeficienty jsou označovány jako c_1 a c_2. V základním algoritmu jsou nastavené před spuštěním a nemění svou hodnotu během průběhu. Náhodně vygenerovaná čísla mezi 0 a 1 jsou označena r_1 a r_2. Pozice částice i v čase t se označuje X_i^t. P_i^t je nejlepší nalezená pozice částice i v čase t. G^t je nejlepší nalezená pozice hejna v čase t.

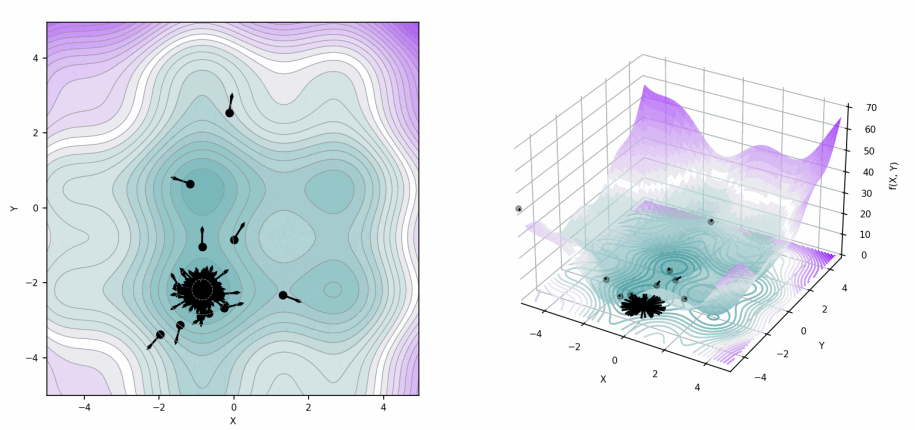
Následně aktualizujeme pozici částice vektorovým součtem vypočítané rychlosti a současné pozice částice podle formule 3.2.

$$X_i^{t+1} = X_i^t + V_i^{t+1} \tag{3.2}$$

Po dokončení výpočtu nové pozice částice je ohodnocena fitness funkcí. Ta se liší podle problému, který řešíme. Pokud je pomocí fitness funkce nalezena lepší hodnota, ať už minimalizací nebo maximalizací, tak aktualizujeme současnou nejlepší hodnotu. Na níže uvedených formulích 3.3 a 3.4 je uvedeno pravidlo aktualizace nejlepších hodnot, za předpokladu, že se snažíme nalézt co nejmenší hodnotu fitness funkce. Pravidlo 3.3 představuje aktualizaci nejlepší pozice nalezené částicí, zatímco pravidlo 3.4 představuje globální aktualizaci.

$$\text{Pokud } f(X_i^{t+1}) < f(P_i^t), \text{ pak } P_i^{t+1} = X_i^{t+1} \tag{3.3}$$

$$\text{Pokud } f(X_i^{t+1}) < f(G^t), \text{ pak } G^{t+1} = X_i^{t+1} \tag{3.4}$$



Obrázek 3.3: Ukázka stavu částic po provedení většího počtu iterací PSO. V ideálním případě budou seskupeny v optimálním řešení, nebo se zaseknou v lokálním optimu. Převzato z https://towardsdatascience.com

Po provedení aktualizací končí iterace a začíná nová, nebo dojde k ukončení algoritmu. K tomu může dojít například dosáhnutím maximálního počtu iterací, nalezením požadované hodnoty, stagnace hodnoty nejlepšího nalezeného výsledku nebo nalezením dostatečně dobrého řešení.

3.2.2 Diskrétní PSO

Klasické PSO pracuje s reálnými čísly. Prostor řešení je spojitý a vzdálenost mezi řešeními se většinou měří pomocí euklidovské vzdálenosti. V diskrétní optimalizaci jsou řešení v prostoru reprezentované celočíselnými proměnnými. Pokud mezi jednotlivými řešeními neexistuje žádné omezení kontinuity (tj. každý bod v prostoru řešení), vzdálenost mezi jednotlivými řešeními lze vyjádřit mnoha způsoby a různé metody měření vzdálenosti budou odpovídat různým strukturám prostorů řešení. Při diskrétní optimalizaci jsou tedy hodnoty bodů v prostoru a jejich fitness funkce určeny podle problému, který se snažíme vyřešit. Současně platí, že vztah měření vzdálenosti mezi body v prostor je nejistý, takže změna hodnoty fitness je také nejistá. [4]

Mezi diskrétní problémy patří například problém barvení grafů a dobře známý problém obchodního cestujícího, který je v zásadě velmi špatný pro tento druh heuristik. Výsledky ukazují, že diskrétní PSO rozhodně není výkonné jako specifické algoritmy pro daný problém, ale na druhou stranu může být upravené pro jakýkoli diskrétní/kombinatorický problém, pro který není žádný efektivní specializovaný algoritmus. Verzí diskrétního PSO je více a můžou se lišit jak průběhem, tak reprezentací stavového prostoru. [6] Přesné popisy diskrétních variant algoritmů PSO, které budu využívat nebo na ně odkazovat, jsou uvedeny v následující kapitole 4.

3.3 Optimalizace založená na chování včelího roje

Algoritmus včelího roje, také označovaný jako ABC, je meta-heuristický algoritmus. Jeho autorem je Karboga, který ho zavedl v roce 2005 pro optimalizaci numerických problémů [15]. V této sekci je demonstrována základní verze tohoto algoritmu.

Jedná se o simulaci zjednodušeného sociálního systému inspirovaným chování včel. Hlavním úkolem včel je shánění potravy pro kolonii, proto musí zajistit nepřerušené zásobování od zdrojů potravy. Aby toho dosáhli, tak mezi sebou komunikují pomocí kolébavého tance. Pokud skautka našla bohatý zdroj potravy, tak prostřednictvím tohoto tance informuje ostatní včely o směru a vzdálenosti k nově objevenému zdroji potravy. Ke kterému se vydají sběratelky, aby ho zanesly do kolonie. [30]

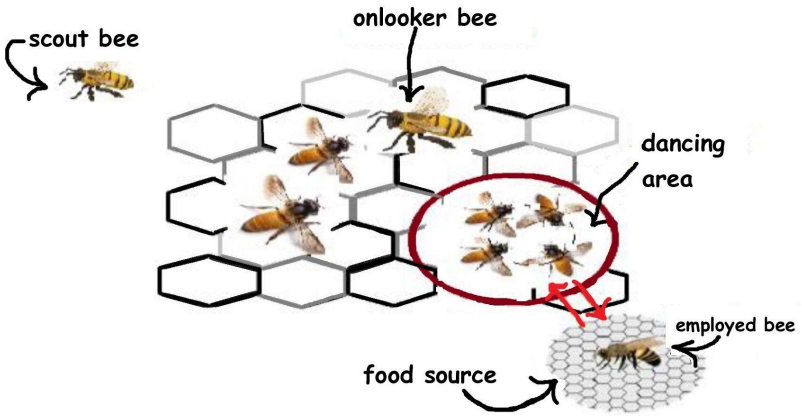
Na tomto principu funguje ABC algoritmus, kde skautka vygeneruje a ohodnotí zdroj potravy. Následně se přepne na sběratelku, která jej bude udržovat. Přihlížející včely si pravděpodobnostně vyberou zdroj potravy podle jeho ohodnocení a budou jej přenášet do kolonie. Pokud se to nebude dařit, pak se sběratelka přepne na skautku, aby našla nový zdroj potravy. Ten se bude opět spolu s přihlížejícími včelami snažit přenést do kolonie.

3.3.1 Formulace ABC

Algoritmus ABC se skládá z populace včel, kde každá včela hledá, nebo modifikuje řešení ve vyhledávacím prostoru. V algoritmu jsou řešení analogické s pozicemi zdrojů potravy. Včely lze rozdělit do tří rolí: sběratelky, přihlížející včely a skauti. Během inicializace je důležité zvolení správného počtu každého typu včel pro dosažení rychlé konvergence. Dále je

sběratelkám přiděleno náhodně vygenerované řešení. Tato řešení jsou ohodnocena a nejlepší z nich se nastaví jako nejlepší globální řešení (gBest).

Po inicializaci populace včel je řešení modifikováno v jednotlivých iteracích, dokud není splněna ukončující podmínka. Každá iterace algoritmu se skládá ze tří po sobě následujících fází, jmenovitě sběratelská fáze, přihlížející fáze a skautská fáze. Ve sběratelské fázi se každá sběratelka snaží zlepšit svou pozici pomocí specifikovaného pravidla. Po sběratelské fázi se spustí přihlížející fáze. V té si každá přihlížející včela vybere jedno z řešení a snaží se ho zlepšit. Každé řešení je spojeno s počítadlem neúspěšných pokusů o zlepšení. Pokud se během modifikace nepodaří zlepšit řešení sběratelkou nebo přihlížející včelou, pak se její počítadlo zvýší o jedničku. Ve skautské fázi jsou všechna řešení zkontrolována. Pokud se zjistí, že řešení není vylepšeno v předem definovaném počtu iterací, pak je řešení považováno za stagnující a je přiděleno skautce. Skautka náhodně generuje řešení a nastaví mu hodnotu počítadla neúspěšných pokusů o zlepšení na nulu. [18]



Obrázek 3.4: Znázornění včel v algoritmu. Převzato z https://www.baeldung.com/cs/artificial-bee-colony

Kroky algoritmu

- 1. Ve fázi inicializace populace se, podle nastavených hodnot, danému počtu včel přiřadí role sběratelek, přihlížejících včel nebo skautek. Také je sběratelkám přiřazena náhodná výchozí pozice. Pozice jsou ohodnoceny a nejlepší z nich je zvoleno jako gBest.
- 2. Sběratelská fáze: Pro každou sběratelku vylepši současnou pozici patřičnou operací, kterou může být například prohození náhodných pozic v permutaci. Pokud je nové řešení lepší než předchozí, tak jej nahraď.
- 3. Přihlížející fáze: Pro každou přihlížející včelu zkopíruj pozici některé sběratelky a tu modifikují. Sběratelky se zvolí pravděpodobnostně podle hodnoty jejich řešení.
- 4. Skautská fáze: Skauti budou prohledávat prostor, tím může být například vygenerování řešení a přepnutí na sběratelku.
- 5. Kontrola ukončující podmínky, kterou je maximum počtu iterací nebo minimální zlepšení nejlepšího nalezeného řešení. Pokud není splněna, pokračuje se krokem 2.

Inicializace

Každé částici je přiřazena náhodně vygenerovaná pozice X_i, pro $i = 1, 2, ..., N$, kde N představuje počet částic. Pozice X_i reprezentuje vektor $X_i = (x_0, x_1, ..., x_m)$, kde m je počet dimenzí prohledávaného prostoru. Dále je každé částici přiřazena role. Role se může přiřazovat rozložením nebo zvolením procenta z celkové velikosti populace pro určitou roli. Také se zvolí limit neúspěšných pokusů o zlepšení, po kterých je sběratelka udržující řešení přepnuta na skautku. Nakonec se nastaví nejlepší řešení.

Průběh algoritmu

Každá iterace je rozdělena do tří fází. V první, tedy sběratelské fázi, se pro každou sběratelku provede operace nad současným řešením pro vygenerování nového blízkého řešení. Pokud je jeho ohodnocení pomocí fitness funkce lepší, pak se nahradí podle následující formule 3.5:

$$\text{Pokud } f(X_i^{t+1}) < f(X_i^t), \text{ pak } X_i = X_i^{t+1}, \text{ jinak } X_i = X_i^t \tag{3.5}$$

V přihlížející fázi si každá přihlížející včela vybere jedno z řešení a snaží se ho zlepšit. Každé řešení je spojeno s počítadlem pokusů o zlepšení. Pokud ke zlepšení dojde, pak je provedena úprava podle vzorce výše a resetuje se počítadlo pokusů. Pokud řešení nezlepší sběratelka nebo přihlížející včela, pak se její počítadlo zvýší o jedničku. Pokud se zjistí, že řešení není vylepšeno v předem definovaném počtu iterací, pak je řešení považováno za stagnující a je přiděleno skautce. Přihlížející včely mohou pro zlepšení provádět stejnou operaci jako sběratelky, ale většinou se způsob prohledávání liší.

Na začátku skautské fáze jsou všechna řešení zkontrolována. Pokud se zjistí, že řešení není vylepšeno v předem definovaném počtu iterací, pak je řešení považováno za stagnující a sběratelka s tímto řešením je přepnuta na skautku. Přidělení může být realizováno procentuálním poměrem nejhorších sběratelek, ty se pak stanou skauty. Další možností je třeba přidělení s pravděpodobností odvozené od počitadla neúspěšných pokusů. Skautka náhodně generuje řešení a nastaví mu hodnotu počítadla na nulu. Následně se přepne na sběratelku udržující toto řešení.

Po ukončení skautské fáze začíná nová iterace sběratelskou fází, nebo dojde k ukončení algoritmu. Princip ukončení algoritmu je stejný jako u PSO. Dosáhnutím maximálního počtu iterací, nalezení požadované hodnoty, stagnace hodnot nejlepšího nalezeného výsledku nebo nalezení dostatečně dobrého řešení.

3.4 Diverzita populace

Tato sekce je převzatá ze studie [5]. Diverzita populace PSO a ABC je důležitá pro měření a dynamické nastavování schopnosti algoritmu prozkoumat a vykořisťovat. Jinými slovy, rozložení a rychlost částic má vliv na tendenci prohledávání velkého vyhledávacího prostoru nebo zpřesnění v místní oblasti. ABC nemá rychlost částic, ale zpřesnění v místní oblasti je ovlivněno počítadlem nepovedených pokusů o vylepšení. Definice diverzity by se daly rozdělit na tři druhy: diverzita pozice, diverzita rychlosti a kognitivní diverzita.

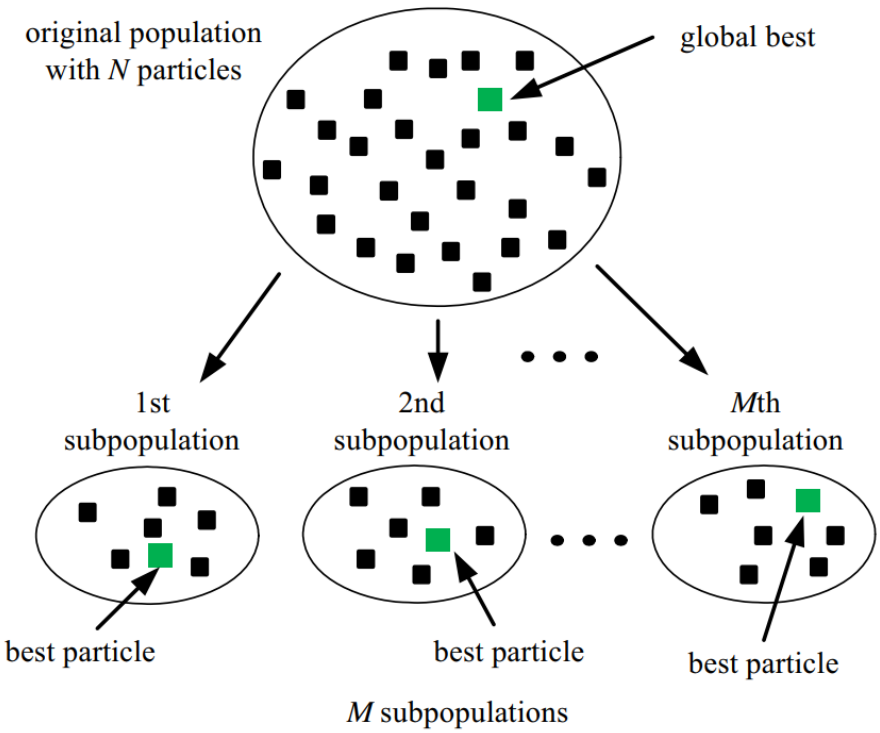
Diverzita pozice měří a poskytuje informaci o rozložení částic v prostoru. Zda budou částice divergovat nebo konvergovat by se mohlo od tohoto měření odrazit. Diverzitu pozice lze měřit po jednotlivých prvcích nebo dimenzionálně. Měření po prvcích bere všechny částice a všechny rozměry jako celek pro výpočet diverzity populace. Zatímco dimenzionální bere každou dimenzi zvlášť.

Diverzita rychlosti dává dynamickou informaci o rozložení rychlosti částic a různorodosti směru pohybu částic.

Kognitivní diverzita představuje rozložení všech řešení nalezených částicemi. Měření kognitivní diverzity je stejné jako měření diverzity pozic, s rozdílem, že je použita aktuální nejlepší nalezená pozice částicí místo její aktuální pozice. Proto je také analýza diverzity pozic aplikovatelná na kognitivní diverzitu.

3.4.1 Více hejn

Optimalizace více hejn je založená na použití topologie složené z více hejn místo jednoho standardního hejna. Obecný přístup k optimalizaci je takový, že každé z hejn se zaměřuje na určitou oblast, zatímco metoda diverzifikace rozhoduje o tom, kde a kdy vypustit další hejno. To znamená, že se algoritmus snaží vypouštět nová hejna v řídce osídlených oblastech pro dobré prozkoumání prostoru. Výpočet vzdálenosti by byl v diskrétním PSO proveden pomocí Hammingovy vzdálenosti. Na tomto přístupu, akorát pro reálné PSO, je například založená metoda podpopulací v PSO uvedená v [3]. Kde každá z podpopulací má vlastní nejlepší hodnotu, podle toho je i upraven výpočet rychlosti.



Obrázek 3.5: Znázorněná metoda podpopulací. [3]

Kapitola 4

Optimalizace TSP pomocí metod KI

Pro optimalizaci TSP je potřeba diskretizovat zvolenou metodu KI, pokud pracuje s reálnými čísly. Dále se musí upravit některé kroky algoritmu a v případě PSO také předefinovat reprezentaci pozice a rychlost částice. Tato kapitola se bude zabývat obecnou aplikací KI na problém TSP pomocí metod PSO a ABC. Také zde budou představeny algoritmy, které byli převzaty ze studií nebo jimi inspirovány. Cílem bylo vyzkoušet více různých přístupů k řešení problému a reprezentace cest. Z nich budou vybrány nejlepší a ty se pokusím vylepšit. Mnou navržené algoritmy budou představeny v následující kapitole 5.

4.1 Optimalizace pomocí PSO

Pro řešení TSP je nutné algoritmus PSO diskretizovat. Níže uvedené algoritmy se liší diskrétní reprezentací řešení a výpočty pozice. Cílem této práce je porovnat jednotlivé reprezentace a jejich výsledky. Zjistit jestli neexistuje nějaká lepší reprezentace, než ty co se standartně používají.

4.1.1 Swap PSO

Clerc navrhl diskrétní PSO pro TSP, ve kterém pozice každé částice je definována jako permutace všech měst a rychlost je definována jako sekvence transpozic, které budou s určitou pravděpodobností provedeny. Transpozicí je zde myšlena dvojice pozic měst, jejichž prohozením se vytvoří nová cesta. [6] V experimentech je tato metoda označená jako PSO_S.

Vektor představující pozici každé částice je permutací měst TSP. Pozice částice je definována jako $X = (X_1, X_2, ..., X_n)$. Řada čísel $X_1, X_2, ..., X_n$ zde představuje cestu skrz n měst v pořadí od X_1 do X_n a následně návrat do výchozího města. Jedná se o nejzákladnější a nejpřímočařejší reprezentaci.

Diskretizace PSO upravuje pouze vzorce výpočtu rychlosti, pozice a fitness funkce. Rychlost je definována jako sekvence transpozic jednotlivých měst v cestě, které mají být aplikovány na současný vektor reprezentující uspořádání měst pro tuto částici. Rychlost je vypočtena podle vzorce 4.1. Operace $-$ představuje vytvoření sekvence transpozic rozdílem mezi cestami, $+$ představuje disjunkci sekvence transpozic. Transpozice je aplikována podle šance stanovené c a r. Funkce koeficientů c a náhodných čísel r se od PSO s reálnými hodnotami neliší.

$$V_i^{t+1} = V_i^t + c_1 * r_1 * (P_i^t - X_i^t) + c_2 * r_2 * (G^t - X_i^t) \tag{4.1}$$

Rychlost z předchozího kroku je symbolizována V_i^t. Následně se aktualizuje pozice pomocí formule 4.2, která představuje aplikaci transpozic na současnou řadu měst.

$$X_i^{t+1} = X_i^t + V_i^{t+1} \tag{4.2}$$

Kvalita získaného řešení se ohodnotí pomocí fitness funkce. Cílem je minimalizovat hodnotu celkové délky cesty definované formulí 4.3:

$$f(X_i) = \sum_{k=1}^{n-1} d_{k,k+1} + d_{n1} \tag{4.3}$$

Získaná hodnota se porovná s lokálním a globálním nejlepším řešením. Pokud je nově nalezené řešení lépe ohodnocené než stávající pBest/gBest, potom jej nahradí. A tak se pokračuje, dokud není splněna ukončovací podmínka. Schéma algoritmu je ilustrováno 1.

Algorithm 1 Swap PSO

```
for each částici do
    inicializuj částici
end for
for each iteraci do
    for each částici do
        aktualizuj rychlost podle 4.1
        aktualizuj pozici podle 4.2
        aplikuj k-opt pokud ho algoritmus využívá
        ohodnoť pozici pomocí 4.3
        if f(X_i) < f(P_i) then
            P_i = X_i
        end if
        if f(X_i) < f(G) then
            G = X_i
        end if
    end for
end for
navrať nejlepší výsledek
```

4.1.2 Binary PSO

Tento algoritmus byl původně navržen pro řešení UC (unit commitment) problému. Studie [2] tento přístup převádí pro řešení TSP. V kapitole s experimenty bude označené jak PSO_B.

Zde se nejedná o permutaci měst, místo toho je i-tá pozice částice je definována jako $X_i = (X_1, X_2, ..., X_{nc})$, kde se tato řada čísel skládá z binárních hodnot, tedy 0 a 1. Cesta byla zakódována tak, že n představuje počet měst a c je konstanta, pro kterou platí $2^c \geq n$.

Kostra algoritmu je stejná jako 1, ale vzorce pro výpočet se liší. Rychlost se skládá ze stejného vektoru jako pozice. Níže uvedený vzorec 4.4 znázorňuje její výpočet.

$$V_i^{t+1} = w_1 \otimes (P_i^t \oplus X_i^t) + w_2 \otimes (G^t \oplus X_i^t) \tag{4.4}$$

Ve vzorci 4.4 znak $\otimes$ představuje operaci AND, $\oplus$ představuje operaci XOR a $+$ představuje OR.

Pozice se vypočítá podle vzorce 4.5.

$$X_i^{t+1} = X_i^t \oplus V_i^{t+1} \tag{4.5}$$

Pro ohodnocení fitness funkcí se musí pozice dekódovat tak, že se vezme c znaků a jejich hodnota se převede do desítkové soustavy. Následně je pozici přiřazeno nejbližší číslo města, které zatím přiřazeno nebylo. Zbytek algoritmu je nezměněn.

4.1.3 Real switch PSO

V rámci této práce byl navrhnout nový algoritmus zvaný jako real switch PSO. V experimentech bude jeho zkratka PSO_R. Permutace měst je zde opět definována jako $X_i = (X_1, X_2, ..., X_n)$. Rychlost je definována jako řada reálných čísel $R_i = (R_1, R_2, ..., R_n)$. Pozice měst a rychlosti jsou spolu spárovány, tedy každé město má přiřazenou rychlost, představovanou reálným číslem. Níže uvedený vzorec 4.6 znázorňuje výpočet rychlosti.

$$V_i^{t+1} = S(w * R_i^t + c_1 * r_1 * (P_i^t - R_i^t) + c_2 * r_2 * (G^t - R_i^t)) \tag{4.6}$$

Kde $P_i^t - R_i^t$ znázorňuje rozdíl rychlosti nejlepšího reálného vektoru částice a rychlosti současného reálného vektoru. $G^t - R_i^t$ znázorňuje to samé, pouze pro rychlost nejlepší nalezené cesty. Tyto rozdíly jsou sečteny s hodnou rychlosti z předchozí iterace. $S()$ zde představuje funkci, která seřadí vektor tak, že všechna reálná čísla jsou seřazena od nejmenšího po největší. Města jsou seřazena tak, že město je v permutaci vloženo na pozici, na kterou byla vložena jeho hodnota rychlosti v řadě reálných čísel.

Algoritmus nyní funguje podle stejného principu jako předchozí. Permutace měst je ohodnocena fitness funkcí a pokud je lepší než pBest/gBest, tak jej nahradí, ale zároveň je uložena i rychlost. Zbytek algoritmu je nezměněn.

4.1.4 Enhanced Swap PSO

Algoritmus je převzatý ze studie [8]. V experimentech bude označen zkratkou PSO_E. Rozšiřuje diskrétní swap PSO o adaptivní koeficienty, zapomínání, elitismus a vliv sousedních částic. Algoritmus tu bude znázorněn jen zjednodušeně, protože kvůli časové složitosti jeho operací jej nepovažuji za vhodný pro řešení rozsáhlých instancí TSP. Ve studii se výsledky ukazují pouze na velikostech instance do 50 měst. V pozdějších kapitolách na něj budu odkazovat, protože při návrhu vlastních algoritmů jsem převzal několik myšlenek. Přesněji jsem se inspiroval sebeadaptivními koeficienty a výpočtem vzdálenosti. Ty jsem samozřejmě zjednodušil. Pokud bych se řídil jejich podobou ve studii, pak by byly příliš časově náročné. Zrychlení se v tomto algoritmu vypočítá podle vzorce 4.7.

$$V_i^{t+1} = V_i^t + cp * r_1 * (P_i^t - X_i^t) + cl * r_2 * ((1 - f_i)L_i^t - X_i^t) + cg * r_3 * ((1 - f_i)G^t - X_i^t) \tag{4.7}$$

Kde cp, cl a cg jsou sebeadaptivní koeficienty pro pBest, lBest a gBest. Na rozdíl od základního algoritmu PSO jsou tyto koeficienty zrychlení jedinečné pro každou částici a jsou

adaptivně upraveny na základě historických znalostí elitních částic v populaci. Za elitní částice jsou považovány ty, které dosáhli dobře ohodnocených pozic.

Zde musíme rozlišovat mezi pBest a lBest, kde pBest představuje nejlepší nalezenou hodnotu částicí a lBest představuje nejlepší nalezenou hodnotu sousedy této částice, kteří jsou vybráni náhodně. Pokud po delší dobu nedojde ke zlepšení nejlepší nalezené hodnoty, pak částice zapomene některé informace ze svých sociálních znalostí. Toto diverzifikuje chování obyvatelstva, které následně zvyšuje jeho průzkum. Koeficient zapomínání je vyjádřen ve vzorci 4.7 jako f_i. Výpočet je závislý na vzdálenosti mezi částicí a gBest nebo lBest. Pokud nedojde ke zlepšení po určitý počet iterací, pak dojde k přepočítání sebeadaptivních koeficientů, zapomínání a sousedů částic. Algoritmus je zajímavý, ale doba průběhu algoritmu je větší než u swap PSO. Tato doba navíc roste rychleji s velikostí instance. To jej dělá nevhodným pro veliké instance TSP.

4.2 Optimalizace pomocí ABC

Pro řešení TSP je nutné využít diskrétní ABC. Níže uvedené algoritmy se liší diskrétní reprezentací řešení a výpočty pozice. Protože jsem pro ABC nenašel studie zabývající se různou reprezentací, tak tato sekce obsahuje reprezentaci permutací a můj návrh binární reprezentace pro ABC.

4.2.1 Swap ABC

Algoritmus je ilustrovaný na následující straně 2. Pozice každé částice dána permutací měst jako v diskrétním PSO. Algoritmus využívá nejjednodušší možnou modifikaci nalezeného řešení, tedy prohození pořadí dvou měst. Po provedení modifikace cesty částicí je ohodnocena fitness získané cesty. Pokud se jedná o zlepšení, pak se modifikovaná cesta uloží a počítadlo neúspěšných pokusů o zlepšení C se vynuluje. Tento způsob vylepšování prohozením je základní přístup k řešení TSP pomocí ABC. Nakonec je X nejhorších sběratelek nastaveno na skauty, kde X je nastavený počet skautů před spuštěním algoritmu. V přihlížející fázi si každá přihlížející včela podle fitness rulety vybere sběratelku, jejíž řešení bude vylepšovat.

$$p_i = \frac{\frac{1}{f_i}}{\sum_{k=0}^{n} \frac{1}{f_n}} \tag{4.8}$$

Ve vzorci 4.8 p_i představuje pravděpodobnost zvolení, f_i představuje ohodnocení fitness cesty, která je udržována sběratelkou. Cílem je, aby sběratelka s nejlepší hodnotou měla největší pravděpodobnost zvolení, která se odvíjí od velikosti její fitness v porovnání s ostatními. Po zvolení sběratelky je aplikována operace prohození a ohodnocena fitness. V modifikacích se zvolené operace můžou mezi fázemi lišit. Ve skautské fázi se generuje nová náhodná cesta. A počítadlo neúspěšných pokusů o zlepšení C se vynuluje.

Ohodnocení fitness a porovnání je stejné jako v PSO. Je ale nutné brát ohled na to, že fitness se ohodnocuje v každé fázi, pro každou včelu po modifikaci. Protože se sběratelka po během sběratelské fáze může přepnout na skautku, pak může během iterace modifikovat své řešení dvakrát. Jednou ve sběratelské fázi, kde se po neúspěšném řešení přepne na skautku a poté ve skautské fázi po vygenerování nové cesty. Pokud je nové řešení lepší, tak nahradíme předchozí nalezené řešení. To se opakuje, dokud není splněna ukončovací podmínka.

Algorithm 2 ABC

for each částici **do**
  inicializuj částici
end for
for each iteraci **do**
  for each sběratelku **do**
    modifikuj své řešení
    aplikuj k-opt na nejlepší řešení, pokud ho algoritmus využívá
    ohodnoť pozici opět pomocí 4.3
    if $f(X_i) < f(G)$ **then**
      $G = X_i$
      $C = 0$
    else
      $C = C + 1$ a uprav roli na skautku, pokud překročil hranici
    end if
  end for
  nejhorší sběratelky se nastaví na skautky
  for each přihlížecí **do**
    zvol jedno z řešení podle fitness
    modifikuj řešení
    ohodnoť pozici opět pomocí 4.3
    if $f(X_i) < f(G)$ **then**
      $G = X_i$
      $C = 0$
    else
      $C = C + 1$ a uprav roli držitelky řešení na skautku, pokud překročil hranici
    end if
  end for
  for each skautku **do**
    vygeneruj řešení a uprav roli na sběratelku
    ohodnoť pozici opět pomocí 4.3
    if $f(X_i) < f(G)$ **then**
      $G = X_i$
    end if
    $C = 0$
  end for
end for
navrať nejlepší výsledek

4.2.2 Binary ABC

V rámci této práce je potřeba zmínit pokus o návrh nového algoritmu binary ABC. Snahou bylo využít principy binárního PSO v ABC algoritmu. V experimentech označený ABC_B. Princip a reprezentace jsou převzané z binárního PSO. Tedy zakóduje cestu do binárních hodnot a provádí nad nimi operace. Modifikace cesty se provede pomocí 4.9.

$$X_i^{t+1} = w \otimes (G^t \oplus X_i^t) \tag{4.9}$$

Ve vzorci 4.9 znak $\otimes$ představuje operaci AND a $\oplus$ představuje operaci XOR. U sběratelek je w nastavené na 0.7 a u přihlížejících na 0.95. U přihlížejících včel se ale vygeneruje náhodná cesta, která má malou šanci na vygenerování 1. Pro přihlížející včely je ve vzorci G^t nahrazeno touto náhodnou cestou.

Pro ohodnocení fitness ji zase převede do desítkové soustavy podle stejných pravidel jako binární PSO. Kostra algoritmu je stejná jako u předchozího ABC algoritmu 2. K-opt pro tento algoritmus, nebyl implementován.

Tento algoritmus byl navržen na začátku práce, kde jsem experimentoval s různými reprezentacemi cest. Během těchto předběžných experimentů se binární reprezentace podle výsledků zdály jako nejlepší. Já jsem ale nemohl přijít na to proč, dokud jsem pořádně neprozkoumal výsledné cesty. Zjistil jsem, že určité instance, na kterých jsem testoval, měli nejkratší cestu seřazenou podle pořadí měst ve vstupních datech. Binární algoritmy mají tendenci cesty řadit. To je dáno tím, že pokud při překladu z binární reprezentace je pro tuto hodnotu město už přiřazeno, tak se vezme nejbližší hodnota. Znovu provedené experimenty a jejich popis bude uveden v kapitole 6.

Kapitola 5

Implementace algoritmů a návrh vlastních

Tato kapitola se zaměří na implementaci. Začne zvoleným programovacím jazykem. Nastíní funkčnost programu a data se kterými pracuje. Dále budou navrženým algoritmům věnovány vlastní sekce. Těmito algoritmy jsou diverzní PSO a ABC. Nakonec obsahuje sekce s důležitými metodami, které tyto algoritmy využívají a rozšíření více hejn/rojů pro oba algoritmy.

5.1 Implementace a funkce programu

Pro implementaci jsem využil programovací jazyk Rust. Jedná se o víceparadigmatický jazyk, kladoucí důraz na výkon a bezpečnost. Rust byl ovlivněn nápady z funkcionálního programování, včetně neměnnosti, funkcí vyššího řádu a algebraickými datovými typy. Pokud bych měl Rust porovnat s C++, tak Rust je navržen, aby byl bezpečnější jazyk než C++. Naopak C++ je výkonnější a flexibilnější jazyk než Rust a má k dispozici více knihoven. Což dává smysl, protože existuje mnohem déle než Rust. Rust je relativně nový jazyk, který roste na popularitě. [26] Zvolil jsem ho pro implementaci algoritmů oproti C++ kvůli přehlednosti a bezpečnosti. I když je možné dosáhnou o něco větší rychlosti v C++ a podobných jazycích, Rust je dost výkonný na to, aby nebyl moc pozadu. Navíc mi byl doporučen a chtěl jsem se naučit něco nového.

Rust obsahuje vlastního správce balíčků `Cargo`, ten se stará o spoustu úkolů, jako je překlad kódu nebo stahování knihoven, na kterých kód závisí. Program používá jeho standardizovaný systém správy závislostí. Proto úplný seznam knihoven třetích stran, které jsou používané v aplikaci lze nalézt v souboru `Cargo.toml` v kořenovém adresáři projektu.

5.1.1 Nastavení parametrů

Protože program obsahuje více algoritmů a ty mají větší množství nastavitelných koeficientů, tak se program spouští bez argumentů. Podstatné parametry jsou umístěny v souboru params.yaml, jinak by počet argumentů při spuštění byl příliš veliký. Soubor obsahuje nastavení všech koeficientů pro algoritmy založené na PSO nebo ABC. Na obrázku 5.1 jsou zobrazeny první řádky. Ty obsahují spuštěný algoritmus, počet běhů, iterací, velikost populace. Na dalším řádku je cesta k souboru, kde se nachází data. Pod ní je cesta, kam uložit soubor s výsledky. Dále obsahuje parametr pro vytvoření `svg` zobrazující výslednou cestu. Následují parametry pro algoritmy s více hejny. Tedy počet iterací, po kterých nedojde ke

zlepšení, pak dojde k oddělení části hejna. Na dalším řádku se nachází počet iterací odděleného hejna, než dojde ke spojení. Dále jsou parametry určující, zda se použije k-opt a zda se pro prohození využije tabulka nejbližších sousedů a s jakou pravděpodobností. Následují parametry jednotlivých algoritmů, ze kterých je zde pouze znázorněn pouze PSO_I. Ostatní parametry algoritmů nejsou ukázány v obrázku. Poslední je generování částic, jestli je náhodné, nebo se vygenerují určité vzory, například seřazení.

```yaml
---
- !Setup
  algorithm: pso_s # jednotlive algoritmy, nebo vsechny pomoci "all"
  runs: 10 # pocet behu
  iterations: 5000 # pocet iteraci
  population_size: 100 # velikost populace
  coordinates_file_path: data/berlin52.tsp # cesta k souboru tsp, nebo
  output_file_path: results/compareParams # cesta kam se ulozi vystupni
  plot_svg: true # zda se ma vyplotit vysledne svg (pouze pro svg)

- !MultipleSwarms # pro algoritmy s vice hejny/roji pso_mss, abc_mss
  split_stagnations: 100 # pocet stagnaci pro oddeleni
  split_iterations: 250 # pocet iteraci oddeleneho hejna

- !LocalSearch # pro pso_s, pso_i, abc, abc_i
  k_opt: 1 # 2, nebo 3 (jinak se neprovede)
  use_neighbour_table: true # vyuziti tabulky nejblizsich sousedu
  use_neighbour_chance: 0.95 # sance na jeji vyuziti pri vymene

- !PsoImp # pso_i
  pso_i_alfa: 0.9 # koeficient pro gbest
  pso_i_beta: 0.9 # koeficient pro pbest
  pso_i_gamma: 0.9 # koeficient pro nahodnou cestu
  pso_i_max_random_swaps: 3
  pso_i_insert_chance: 0.001
  pso_i_difference_mod: 1000.0
  pso_i_max_stagnation: 50 # koeficient poctu iteraci po kterych se nez
```

Obrázek 5.1: Znázornění prvních řádků souboru s parametry, které obsahují parametry běhu a PSO_I.

5.1.2 Vstupní data reprezentující TSP

Data představující problém TSP, tedy pozice jednotlivých měst, jsou převzata ze souboru. Je možné číst soubory typu `tsp` nebo `svg`. Pokud se data v následující kapitole s experimenty čtou ze souborů `tsp`, tak budou převážně z TSPLIB [1]. Jedná se o soubor velikého množství instancí TSP. Je ale nutné vzít v potaz, že většina instancí, které obsahuje, jsou menší velikosti. Data ze souboru formátu `svg` je možné například vygenerovat pomocí StippleGen2 [2], sloužící pro generování artTSP. Cestu k souboru je možné definovat pomocí parametru `coordinates_file_path`.

[1] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/
[2] https://github.com/evil-mad/stipplegen

Příklad vstupního souboru je znázorněn na obrázku 5.2. Hlavička v souboru, obsahující tagy pro jméno, komentář, dimenze, atd..., není z veliké části podstatná. Jediný podstatný tag, který obsahuje, je `EDGE_WEIGHT_TYPE`. Ten rozhodne, jak budou vypočítány hrany v grafu. Podporovány jsou typy `GEO`, `CEIL_2D`, `EUC_2D`. Ostatní tagy můžou sice obsahovat důležitá data pro některé experimenty, zde ale nebudou třeba a program je nebere v potaz. Pro program je podstatná sekce začínající tagem `NODE_COORD_SECTION`, kde jsou pozice měst. Města a pozice jsou čtena jako na obrázku, tedy označení, souřadnice x a y. Označení není bráno v potaz, aby se předešlo situacím, kde budou dvě místa označena stejně. Souřadnicím jsou přiřazena sestupně pořadí. Souřadnice musí obsahovat pouze čísla, jinak bude program ukončen s chybou. Toto je podstatné, protože v TSPLIB existují soubory obsahující souřadnice s exponentem. Pro přečtení těchto souborů je nutné odstranit exponenty ze souřadnic. V přiložené složce obsahující instance TSP problému se tyto případy nenachází. Nakonec je také potřeba vložit tag `EOF` značící ukončení. Oba tagy pro rozpoznání začátku i konce sekce jsou podstatné, jinak nebude možné data přečíst.

pso_for_tsp > data > burma14.tsp

```
NAME: burma14
TYPE: TSP
COMMENT: 14-Staedte in Burma (Zaw Win)
DIMENSION: 14
EDGE_WEIGHT_TYPE: GEO
EDGE_WEIGHT_FORMAT: FUNCTION
DISPLAY_DATA_TYPE: COORD_DISPLAY
NODE_COORD_SECTION
    1  16.47       96.10
    2  16.47       94.44
    3  20.09       92.54
    4  22.39       93.37
    5  25.23       97.24
    6  22.00       96.05
    7  20.47       97.02
    8  17.20       96.29
    9  16.30       97.38
   10  14.05       98.12
   11  16.53       97.38
   12  21.52       95.59
   13  19.41       97.13
   14  20.09       94.55
EOF
```

Obrázek 5.2: Příklad vstupního souboru ve formátu `tsp`.

Pokud čte data z `svg` souboru, pak hledá element s atributy cx a cy, ze kterých získá souřadnice x a y. Na názvu tohoto elementu nezáleží. Níže je uveden příklad tohoto elementu ze souboru vygenerovaného pomocí StippleGen2. Ostatní elementy nejsou podstatné.

- <circle cx="1418.9117"cy="513.9206"r="3.7573528"/>

5.1.3 Výstup řešení TSP

Výsledky běhů jsou pak uloženy do souboru v adresáři na cestě. Tu je možné definovat parametrem `output_file_path`. Pokud cesta k adresáři neexistuje, pak je adresář spolu s ostatnímu adresáři na cestě vytvořen. Vytvořený soubor nese název algoritmu, který byl pro experiment zavolán. První řádek souboru obsahuje název algoritmu. Druhý obsahuje

pole všech ohodnocení výsledných cest běhů v experimentu. Následuje řádek s celkovými časy jednotlivých běhů. Nakonec obsahuje řádky s délkou nejlepší cesty v každé iteraci. Každý běh je zvlášť uložen na řádku. Pokud už soubor se stejným jménem existuje, pak je přepsán.

Vykreslené `svg` soubory jsou uloženy v kořenovém adresáři. Zda se mají vykreslit se nastavuje parametrem `plot_svg`. Kostra souboru odpovídá `svg` souborům generovaným pomocí StippleGen2.

5.1.4 Skript pro vykreslení boxplotů a výpis výsledků

Skript `plot.py` pro vizualizaci dat je umístěn v kořenovém adresáři. Byl napsán v programovacím jazyce Python. Jeho spuštěním je možné vykreslit boxplot a konvergenční křivky z výstupních souborů. Při spuštění je potřeba zadat argument představující mód spuštění. V případě 1 se vykreslí boxploty a konvergenční křivky. V případě 2 vypíše do konzole minimum, průměr a maximum dosažených hodnot ke každému souboru v adresáři. Další argument představuje cestu do adresáře s výstupními soubory jednotlivých experimentů. Z těchto dat vykreslí jeden boxplot pro celou složku a konvergenční křivky pro každý experiment. Poslední argument je volitelný a představuje násobení hodnot. Slouží k převedení výsledků do jiných jednotek, pokud je zadán. Příklad spuštění skriptu je uveden níže.

- python plot.py <mód> <cesta> [násobek]

5.2 Vlastní návrh PSO algoritmu

V experimentech a další kapitole bude označován jako PSO_I. Reprezentace cesty algoritmu je permutace měst. Důvodem je, že z prozkoumaných reprezentací v předchozí sekci má swap PSO jasně definovaný posun částic v prostoru, kterým je prohození pozic měst. To představuje pohyb směrem k pBest/gBest. Posuny ostatních algoritmů nejsou jasně definované, tzn například u binárního PSO se v jedné iteraci může pouze prohodit pozice dvou měst v řešení, nebo se řešení může změnit celé. Také dával nejlepší výsledky při porovnání algoritmů.

To ale neznamená, že PSO nemá podstatný problém a tím je diverzita. Fakt, že se v prostoru částice pohybují pouze prohozením rozdílů oproti pBest/gBest, redukuje schopnost prohledat prostor a vylepšit současné nejlepší řešení. Okolím budu zde považovat všechna řešení vzdálená od současného jedním prohozením dvou měst. Při zvýšení velikosti grafu o jedno město přidáváme novou dimenzi do řešení, tím zvětšujeme velikost okolí řešení exponenciálně. Pokud máme velikou cestu, za kterou zde budu považovat 1000 měst a více. Hejno částic sice doletí ke gBest, ale pokud jich není dostatek, tak okolí gBest nebude vůbec prozkoumané. A protože se okolí prohledá jen málo, pak existuje menší pravděpodobnost získání lepšího gBest během letu. Pokud lepší řešení nebude nalezeno, tak po doletu k němu už částice už nebudou mít co změnit, nebo budou provádět změny z předchozí iterace, tedy posun zpět a následně k gBest. Navíc částice na pozici gBest nebude schopna provést žádné operace, protože nemá žádné rozdíly proti gBest, což ji dělá zbytečnou, dokud další částice nenalezne lepší řešení.

Další problém spočívá v rychlosti. Ve spojitém prostoru rychlost upravuje velikost posunu v iteraci. Ve swap PSO se normálně aplikuje s určitou vahou rychlost z předchozí iterace. To znamená, že aplikuje některé z výměn provedených v předchozí iteraci znovu. Ale pokud prohodíme města na pozicích například 1 a 3, a toto prohození provedeme v další

iteraci znovu, tak akorát vrátíme města na původní pozice. Nejedná se o zrychlení konvergence, ani příliš nepomáhá s prohledáním prostoru.

Posledním problémem je vytvoření sekvence transpozic a následně jejich provedení s určitou pravděpodobností. Prvně vytvoříme kopii současné cesty, pak se iteruje přes celou permutaci měst a porovnává s nejlepším řešením, kde každé město, které není na správné pozici, je nutné prohodit a prohození uložit do sekvence transpozic. To se provede jak pro gBest, tak pBest. Po získání celé sekvence transpozic se aplikují s určitou pravděpodobností na současnou permutaci cesty. Prohazujeme tedy dvakrát, jednou při vytváření sekvence transpozic a následně při její aplikaci. Navíc musíme vytvořit kopii současné cesty. Pro menší instance to není problém, ale pro instance s velikostí nad 1 000, to už začne být znát.

5.2.1 Původní návrh

Cílem bylo zmenšení počtu operací, aby se místo vytváření sekvence transpozic, jednotlivá prohození provedla rovnou. Pro zrychlení se omezí počet prohození, navíc pokud částice zlepšuje nalezené řešení, tak se zvýší počítadlo prováděných prohození. Pokud se částice bude pohybovat správným směrem, tak bude zrychlovat. Pokud nebude docházet ke zlepšení, tak bude snižovat počet prohození a důkladně prohledávat prostor. Nakonec se přidají náhodná prohození pro dosažení diverzity. Tím se částice nebudou pohybovat pouze po prohození rozdílů s nejlepším řešením, ale také náhodně prohledávat prostor. Cíl byl provádět náhodná prohození více, pokud je řešení podobné gBest, aby se prohledal okolní prostor. Jinak jako u klasického PSO budou částice provádět prohození rozdílných měst směrem k nejlepším pozicím. Na první pohled se neliší od diskrétního PSO algoritmu 1 uvedeného v předchozí kapitole. Základní rozdíl je existence rychlosti V, koeficientu D a výpočtu pozice, které budou vysvětleny níže.

Algorithm 3 Původní návrh vlastního algoritmu PSO

for each částici **do**
    inicializuj částici
end for
for each iteraci **do**
    for each částici **do**
        aktualizuj koeficient D podle 5.2
        aktualizuj pozici podle 5.1
        proveď k-opt pokud je nastaven a jedná se o nejlepší částici
        ohodnoť pozici pomocí standartního výpočtu fitness
        if $f(X_i) < f(P_i)$ **then**
            $P_i = X_i$
            $V = V * 2$
        else
            $V = V/2$
        end if
        if $f(X_i) < f(G)$ **then**
            $G = X_i$
        end if
    end for
end for
navrať nejlepší výsledek

V tomto algoritmu 3 je rychlost definována proměnnou V, která představuje počet prohození. Bude nastavena na dvojnásobek předchozí hodnoty s každým úspěšným zlepšení pozice, naopak pokud po změně pozice nedojde ke zlepšení, pak se nastaví na polovinu předchozí hodnoty, ale ne méně jak 1. Při inicializaci se nastaví na 1.

Výpočet změny pozice bude proveden podle následujícího vzorce 5.1.

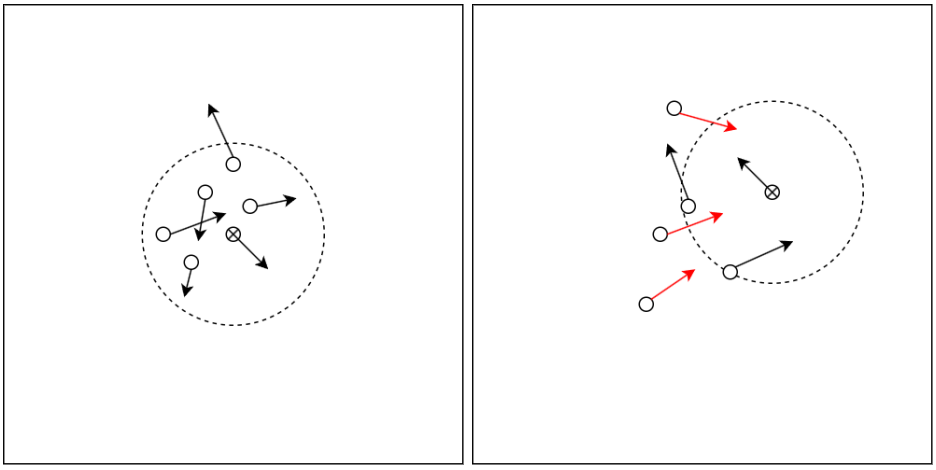
$$X_i^{t+1} = V \otimes (D * r_1 * (P_{is}^t - X_{im}^t) + D * r_2 * (G_r^t - X_{in}^t) + (1 - D) * r_3 * \text{random\_swap}()) \quad (5.1)$$

Pokud $(P_i^t - X_i^t)$ a $(G^t - X_i^t)$ představují představuje vytvoření sekvence transpozic rozdílem mezi cestami, pak $(P_{is}^t - X_{im}^t)$ a $(G_r^t - X_{in}^t)$ představují pouze jedno z těchto prohození, které je v současném řešení označené m/n pro pBest/gBest. Na rozdíl od původního přístupu by se při počítání rozdílů mezi současnou cestou a pBest/gBest, kde se prochází celé sekvence a vytváří sekvence transpozic, začalo v náhodné části sekvence a pokračovalo, dokud se nenarazí na pozici s rozdílným znakem a ta se změnila s určitou pravděpodobností a tak se opakovalo, dokud se neprovede V prohození. Funkce random_swap() představuje jedno náhodné prohození. Rychlost V představuje počet prohození. Šance provedení bude spočívat v sebeadaptivním koeficientu D a náhodné hodnotě r. Ta se bude pohybovat mezi hodnotami c a 1.0, kde c je číslo mezi 0 a 1.0, nastavené před spuštěním algoritmu. Částice tedy provádí pouze dvě prohození v iteraci, proto se místo vynásobení nastavuje minimální náhodně vygenerovaná hodnota, aby byla větší šance na provedení operace než ve swap PSO. Sebeadaptivní koeficient D představuje šanci prohození, která se zvyšuje podle toho, zda se současná fitness funkce $f(P_i^t)$ blíží hodnotě fitness nejlepší nalezené pozice $f(G^t)$. Bude vypočítán podle vzorce 5.2.

$$D = min(\frac{f(G^t) - f(X_i^t)}{f(G^t)} * k, 0.9) \quad (5.2)$$

Funkce $min()$ slouží k tomu, aby šance prohození byla maximálně 0.9, díky tomu nebude garantované prohození a šance náhodného prohození bude minimálně 0.1. Pomocí hraničního koeficientu k se může nastavit, při jakém procentu rozdílu fitness dojde ke změně a začnou se více provádět náhodná prohození pro prohledání okolí, než konvergovat k nejlepší nalezené pozici. Protože se zde bude pracovat s většími instancemi, tedy s velikými hodnotami fitness, proto k nastavuji na 100 až 1 000. Díky tomu částice začne mnohem více prohledávat prostor okolo nejlepší pozice. Cílem je mít rozdíl v řádu několika prohození. Pokud je částice na nejlepší známé pozici gBest, pak $D = 0$ a pouze náhodně prohledává prostor.

Obrázek 5.3 znázorňuje hranici, za kterou se částice přibližují pBest/gBest. Její velikost se nastavuje koeficientem D. Cílem algoritmu je vytvořit hejno, které tímto prostorem prolítne tak, že se postupně bude přibližovat nejlepšímu nalezenému řešení. Jakmile k němu částice dolétnou, tak začnou náhodně prohledávat okolní prostor. Pokud se posunou za hranici, pak se začnou posunovat směrem k řešení. Důvod, proč jsem zvolil tento způsob výpočtu vzdálenosti 5.2, a ne jednodušší přístup jako rozdíl v počtu prohození, je časová náročnost ve velikých instancích. Pro menší instance by ale nebyl problém upravit tento vzorec nastavením pravděpodobnosti podle Hammingovy vzdálenosti nebo počtu prohození mezi jednotlivými sekvencemi. Nevýhodou je, že je třeba být opatrný s nastavením koeficientu. Pokud bude ilustrovaná na obrázku příliš malá, tak k náhodnému prohledávání nebude skoro docházet. Pokud bude naopak moc veliká, k němu bude docházet příliš často a částice nebudou tolik dolétávat k nejlepší nalezené hodnotě.



Obrázek 5.3: Na obrázku je znázorněno hejno 5 částic. Nalevo je předchozí iterace a napravo současná. Pokud se částice nachází na nejlepší nalezené pozici, tak je označená křížem. Přerušovaný kruh zobrazuje okolí, ve kterém je vzdálenost cest menší než hranice, která se určuje pomocí koeficientu k. Uvnitř mají větší pravděpodobnost náhodného prohledání okolí. Mimo mají větší pravděpodobnost cestovat k nejlepší nalezené lokaci.

Náhodné prohození random_swap() zde představuje diverzitu rychlosti. Existuje několik studií, které se zabývají její efektivitou a to například [9], ale zde se neřeší TSP, ale RVTP. Zarazilo mě, že jsem nebyl schopen najít studie zabývající se diverzitou PSO pro TSP. Naopak pro klasické PSO existují studie věnující se tomuto problému a to například [12].

Mimo těchto změn k jiným změnám nedochází. Podle změn ve vzorcích lze poznat, že částice budou k nejlepšímu řešení konvergovat pomaleji než u swap PSO. Výhodou je silnější prozkoumávání prostoru pomocí diverzity rychlosti, která roste s přiblížením k hodnotě gBest.

5.2.2 Finální verze

Ve finální verzi byl upraven výpočet pozice. V původní verzi byl náhodný výběr pozice v permutaci, od kterého se určitý počet měst vyměnil. To sice vedlo k podstatnému zvýšení rychlosti, ale zanedbávalo rozdíly na začátku cest. Problém byl i s počtem prohození V, ten byl zvýšen, pokud se zlepšila fitness hodnota. Tento přístup způsoboval, že v nepřesně definovaném prostoru se tato hodnota zvýšila spíš výjimečně. To mělo za následek, že se měnilo velmi málo znaků. Nový výpočet pozice je znázorněn vzorcem 5.3.

$$X_i^{t+1} = D * r_2 * (G^t - D * r_1 * (P_i^t - X_i^t)) + V * (1 - D) * r_3 * \text{random\_swap}() \qquad (5.3)$$

Oproti původnímu vzorci 5.1 se více podobá vzorci v diskrétním swap PSO. Na rozdíl od něj se však nevytvoří sekvence transpozic. Prvně se s pravděpodobností $D * r_1$ provede prohození pozic odlišných proti pBest a tato nová permutace se následně porovná s gBest, kde se také s určitou pravděpodobností $D * r_2$ prohodí rozdílné pozice. Nakonec se na této instanci s pravděpodobností $(1 - D) * r_3$ provede V náhodných prohození, kde V je koeficient, který se nastaví před spuštěním algoritmu. Nemusí se jednat jen o náhodná prohození, ale může se také s velmi malou pravděpodobností jednat o vložení prvku na danou pozici. To znamená, že je město přesunuto na danou pozici a všechna následující

města jsou posunuta o jednu pozici dále. Malá pravděpodobnost na vložení je nastavena, protože pokud se provádí často, tak pak musí částice provádět spoustu prohození, aby se vrátila k hejnu, nebo hejno dolétlo za ní. Náhodná prohození nebo vložení mohou využít tabulku nejbližších sousedů. Podle té budou provádět prohození tak, že po náhodném zvolení pozice se zkontroluje pozice následující a její město. Ze sousedů tohoto města v tabulce se jedno vybere a jejich pozice se prohodí. Tato tabulka bude také popsána v nadcházející sekci 5.4. Koeficienty r_m určují šanci na prohození stejně jako ve swap PSO, oproti původnímu návrhu, který nastavoval minimální šanci na prohození kvůli menšímu počtu prohození. K jiným změnám nedochází.

Ze změn je poznat, že se jedná o ústupek z hlediska rychlosti proti původnímu návrhu. Přináší ale přesnější a spolehlivější konvergenci při průběhu. Stále však zrychluje původní PSO algoritmus odstraněním sekvence transpozic z výpočtu. Navíc řeší problém, když jsou oproti původní cestě X_i^t na nějaké pozici n rozdílné znaky jak v pBest a gBest, tak se toto prohození neprovede zbytečně dvakrát. Tím je myšleno, že v původním algoritmu pro diskrétní PSO by se prohození proti pBest uložilo do sekvence transpozic a následně to stejné prohození s gBest by se uložilo později. Nakonec při provádění prohození by se mohli provést obě a tím bychom opět získali stejné město na pozici n, které na ní bylo před aplikací sekvence transpozic. Tímto přístupem redukujeme počet zbytečných operací a zrychlujeme konvergenci.

5.3 Vlastní návrh ABC algoritmu

Algoritmus ABC je principiálně vhodnější k řešení TSP než PSO algoritmus. Neprochází se celá sekvence, aby se vytvořila tabulka transpozic. Provede se jednoduchá modifikace a popřípadě zahodí, pokud nedojde ke zlepšení. Tím se hodně šetří čas. Tento přístup také umožňuje provádět složitější mutace, protože to pak nebude komplikovat dolet částic.

V experimentech bude označen jako ABC_I. Algoritmus je upraven tak, že pokud je nastaveno provádění k-opt operace, pak ji sběratelky provedou, jinak provedou jedno náhodné prohození dvou pozic. Přihlížející včely vždy provedou jedno prohození a následně nastavený počet dodatečných náhodných prohození. Pokud se využije tabulka nejbližších sousedů a podle ní se budou provádět tato prohození, pak se podstatně zrychlí konvergence. Pro tato dodatečná prohození se místo prohození provede operace vložení s nastavenou pravděpodobností. Na rozdíl od PSO zde však není problém následného doletu k hejnu. Proto je tato operace vhodnější pro ABC než pro PSO. Na konci každé 100-té iterace se kontroluje diverzita a pokud jsou si řešení sběratelek podobné, pak se krom jedné přepnou na skauty. Ty v následující iteraci vygenerují nové řešení a začnou ho vylepšovat. Cílem této změny je neplýtvat iterace vylepšováním několika téměř identických řešení, což je problémem tohoto algoritmu. Mimo těchto změn kostra odpovídá popisu uvedeném v algoritmu 2.

Pro diversifikaci se na začátku získá pole obsahující počet výskytu měst na dané pozici v permutaci. Pole se skládá z dvojice město a počet výskytů. Ukládají se pouze města, která obsahuje alespoň jedna z instancí na dané pozici, aby nedocházelo ke zvětšení velikost pole pro větší instance. Součet všech výskytů je velikost podobnosti. Její výpočet je upřesněn ve vzorci 5.4. Pro včelu i, kde N představuje délka permutací a M počet sběratelek. C je 1 pokud se znak na pozici n včely m vyskytuje a nejedná se o včelu i, jinak 0.

$$X_i = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} C \tag{5.4}$$

Pokud tato hodnota přesahuje hranici stanovenou ve vzorci 5.5, pak je role včely změněna na skauta. Kde W je reálné číslo v rozmezí 0 a 1 představující míru podobnosti. V mém případě je toto číslo 0.5. M je opět počet sběratelek a N je délka permutací.

$$H = W * M * N \tag{5.5}$$

Tento způsob výpočtu byl zvolen, aby se zabránilo složitějším výpočtům a vytváření rozsáhlých tabulek. Zejména tabulek, kde by byla porovnávána každá cesta s ostatními, což by u velikých instancí trvalo příliš dlouho. Nejedná se sice o velmi přesný výpočet podobnosti jednotlivých permutací, ale jeho jednoduchost zajišťuje rozumnou dobu provedení výpočtu i pro rozsáhlejší instance řešené větším počtem částic.

5.4 Tabulka nejbližších sousedů a k-opt

Oba výše uvedené algoritmy také mohou využívat tabulku udržující 5 nejbližších sousedů. Algoritmus PSO ji využívá pouze při náhodných prohozeních. Při prohození je vybráno 5 sousedů následujícího města, který soused bude prohozen je zvoleno náhodně. Při náhodném prohození nebo vložení ji mohou s určitou pravděpodobností využít pro výběr prohozených měst. Tato pravděpodobnost se nastavuje v souboru s parametry. Využití tabulky zrychluje konvergenci algoritmů k optimálnímu řešení. Počet nejbližších měst je omezen na 5, protože jinak by tabulka s velikostí instance rostla exponenciálně a už by nebyla efektivní. S omezením počtu prvků roste její velikost s velikostí instance TSP lineárně. Na obrázku 5.4 je znázorněno, kolik paměti by tato matice zabírala, pokud by se neredukovala.

Instance Size	Pheromone Matrix	Restricted Matrix
100	39 KB	12.5 KB
1000	3.8 MB	125 KB
10,000	381.5 MB	1.22 MB
100,000	37.3 GB	12.2 MB

Obrázek 5.4: Na obrázku je vlevo velikost instance, uprostřed je matice obsahující všechny sousedy, vlevo je redukovaná matice sousedů. Zde se jedná o feromonovou matici pro mravenčí algoritmus. Převzato z [25]

Také je zde možné využít k-opt algoritmus nastavením v parametrech. Navržený algoritmus PSO využije k-opt, pokud po určitý počet iterací nedošlo ke zlepšení gBest. Poté se při každé iteraci a pro nejlepší částici nad získaným řešením provede k-opt a následně se výsledek ohodnotí fitness. Za nejlepší částici se považuje ta, která má ohodnocení nejlepší nalezené pozice rovné gBest. Pokud dojde ke zlepšení, počítadlo stagnujících iterací se resetuje. U swap PSO se k-opt provede v každé iteraci pro nejlepší částici za předpokladu, že je jeho provedení nastaveno. V případě ABC algoritmů je k-opt prováděn sběratelskými včelami. Implementované jsou 2-opt a 3-opt verze algoritmu. S lehkou úpravou pro 3-opt, kde je jedna z rozpojených hran vždy mezi prvním a posledním znakem cesty pro algoritmus PSO. Při provedení se pak neposune cesta, proto PSO algoritmus potom nemusí provádět během doletu tolik prohození.

5.5 Využití více hejn a rojů

Protože se jedná o diskrétní problém s nejasně definovaným prostorem, je vytváření topologií definujících rozložení v prostoru poměrně náročné. Jsou sice způsoby výpočtu vzdálenosti, ale upravit je pro výpočet fitness, aby byla jednotlivá hejna dostatečně vzdálená, zde nejde pouze pomocí jednoduchého výpočtu Euklidovy vzdálenosti. Nejjednodušší by byla Hammingova vzdálenost, ale i její výpočet zabírá určitou dobu zejména pro více hejn. Navíc neošetřuje posunutí o jeden znak vlevo a poslední na začátek, což je v podstatě stejná cesta, ale z pohledu Hammingovy vzdálenosti maximálně vzdálena. Proto se zde nezabývám složitými topologiemi, ale subpopulace se oddělí od současného hejna a nehlídá se, zda se vzdalují, nebo prohledávají v blízkém prostoru. Cílem je při dosažení stagnace oddělit část populace a upravit s náhodnými parametry algoritmu tak, aby bylo možné dosáhnou lepšího řešení. Například prováděním více prohození nebo vložení. Také je stanoven počet částic, při kterém se oddělí. Nemá smysl vytvářet hejno pro jedno nebo dvě částice, proto pokud současné hejno obsahuje méně jak 25% počtu částic, se kterými bylo první hejno inicializováno, pak se už nebude dělit. Hlavním cílem je tedy snaha o doladění řešení algoritmu posunem z lokálního minima a vytvářením více mutací.

Algorithm 4 Modifikace více hejn/rojů pro PSO/ABC

```
inicializuj originální hejno
for each iteraci do
    for each hejno do
        proveď iteraci navrženého algoritmu PSO/ABC a získej gBest_i^hejna
        if gBest_i^hejna < gBest_{i-1}^hejna then
            gBest_i^hejna = gBest_{i-1}^hejna
            S^hejna = 0
        else
            S^hejna = S^hejna + 1
        end if
        if S^hejna > S_limit a má dostatečně velkou populaci then
            rozděl hejno
            S^hejna = 0
        end if
        if i^hejna > i_limit^hejna then
            spoj hejno s originálním
        end if
    end for
end for
navrať nejlepší výsledek
```

V 4 pro zjednodušení popisu, může být hejnem také myšlen roj pro ABC_I. Kde i je současná iterace hejna a její limit je označen i_{limit}. Počet stagnujících iterací je označen S a limit pro oddělení S_{limit}. Při každém rozdělení je v případě ABC hlídán počet sběratelek a skautů, aby nové hejno mělo zhruba polovinu toho předchozího. Při spojení hejna se zkontrolují nejlepší hodnoty obou a zvolí se to nejlepší. Pro ABC_I se opět přepočítají počty sběratelek a skautů. Algoritmus PSO tyto přepočty provádět nemusí, protože jeho částice nemají přiřazenou roli.

Kapitola 6

Experimentální výsledky

Tato kapitola popisuje jednotlivé experimenty provedené s využitím implementovaných algoritmů. Je zde popsáno, se kterými parametry jednotlivé algoritmy pracují a za jakých podmínek jsou efektivnější než ostatní. Účelem těchto experimentů je porovnat implementované algoritmy podle výsledků, kterých dosahují. Vyberu nejvhodnější algoritmy a pokusím se tyto výsledky zlepšit. Prvně se porovnají výsledky jednotlivých algoritmů vzhledem k jejich parametrům a následně se budou provádět závěrečné experimenty s těmito parametry. Nakonec provedu experimenty s navrženými algoritmy a porovnám je se současnými a nejlepšími známými výsledky pro dané instance TSP.

6.1 Základní parametry testů

Vstupní data pro experiment mohou být ve formátu `tsp` nebo `svg`. Pokud nebude uvedeno jinak, pak budu v experimentech pracovat s daty získanými z TSPLIB, kde každá instance z TSPLIB má ve jménu i její velikost. Tedy instance pr136 má velikost 136 měst. Data pro artTSP jsou získána z formátu `svg`, do kterých byl obraz převeden pomocí StippleGen2. Základní parametry, se kterými pracuje každý algoritmus, jsou:

- velikost populace,
- počet iterací,
- počet běhů,
- způsob inicializace.

Všechny algoritmy s více hejny obsahují:

- počet stagnujících iterací nejlepší hodnoty pro oddělení,
- počet iterací nového hejna.

Všechny algoritmy ABC obsahují:

- počet sběratelek,
- počet skautů,
- počítadlo neúspěšných změn, po kterých se sběratelka přepne na skautku.

Všechny algoritmy PSO obsahují:

- koeficienty $c_{1..n}$ pro šanci výměny pozic.

Experimenty běžely na 2 zařízeních a to na mém zařízení s Intel(R) Core(TM) i7-9750HF CPU @ 2.60GHz a 16 GB RAM. Dalším zařízením byl server s Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz a 15 GB. Pokud budu porovnávat čas běhu, tak budu experiment provádět na mém zařízení. V případě porovnání získaných výsledků, nemá zařízení žádný vliv na výsledek.

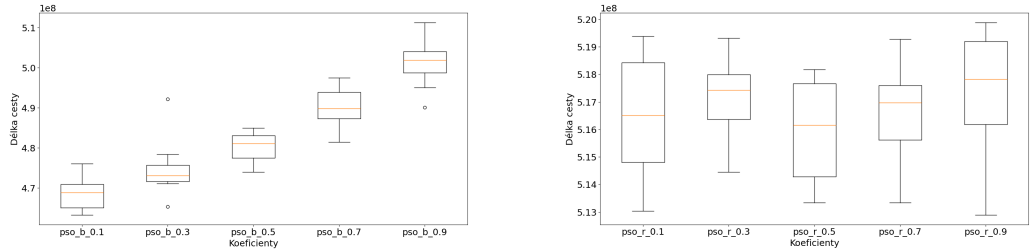
6.2 Předběžný experiment podle základních parametrů

Jedná se o znovu provedení experimentu, který měl v polovině práce rozhodnout na jaké algoritmy se zaměřit a jaké základní parametry jsou nejvhodnější. Důvod znovu provedení experimentu jsou úpravy programu a implementace. Dále umožňuje porovnat původní algoritmy s mnou navrženými. Uvedu zde pouze boxploty a tabulky průměrných hodnot. Konvergenční křivky hrály také roli při původním rozhodování, zejména pro binární algoritmy, které konvergují podstatně rychleji, ale nedosahují dobrých výsledků. Zde porovnáme pouze výsledky, proto konvergenční křivky nebudu uvádět.

Porovnával jsem výsledky jednotlivých algoritmů s různým nastavením parametrů. Algoritmy nevyužívaly k-opt. Velikost populace je konstantně 100, počet iterací 10 000, počet běhů 10, inicializace počátečních cest je generuje se vzory. Vzorem je zde myšleno seřazení všech měst podle jejich označení nebo seřazení pouze části měst. Počet částic generujících vzory je 10, zbytek je náhodně generován. Důvod využití vzorů v tomto experimentu jsou binární algoritmy, které mají tendenci řadit města za sebe podle jejich pořadí ve vstupních datech. Pro některé instance z TSPLIB je to výhoda a u jiných ne. Takhle všechny algoritmy začnou stejně a binární algoritmy nebudou konvergovat rychleji s již seřazenými cestami. Experiment byl proveden pro dsj1000 z TSPLIB. PSO_I měl také nastaven počet náhodných prohození V na 3 a koeficient k na 1 000, šanci na vložení 0.001. Pro ABC algoritmy je na začátku nastaveno maximální počítadlo nepovedených pokusů o vylepšení řešení na 4 000 a 1 skaut. ABC_I mělo nastavený počet dodatečných náhodných prohození na 2 s šancí na vložení 0.5. Algoritmy využívají tabulku nebližších sousedů s 95% pravděpodobností. V boxplotech jsou uvedeny ve formátu (zkratka názvu)_(hodnoty podstatných parametrů).

6.2.1 Základní parametry PSO

Na obrázku 6.1 jsou znázorněny jednotlivé algoritmy PSO s různými koeficienty. Cílem je získat rozsah ideálního nastavení. Počet iterací byl zvolen tak, aby všechny algoritmy s různými nastaveními dokonvergovaly k nejlepšímu řešení. Až na výjimky do 3 000-té iterace všechny algoritmy už stagnovaly. Z výsledků vyplývá, že některé algoritmy jsou závislejší na správném zvolení parametrů než jiné. Správný algoritmus nebo reprezentace cesty má ale větší vliv na výsledek než nastavení koeficientů. Dále z tohoto experimentu je možné odvodit rozmezí správného nastavení parametrů pro další experimenty. Musí se ovšem počítat, že s většími instancemi TSP se mohou ideální parametry změnit. Také mohou být ovlivněny počtem iterací, protože různé nastavení konvergují s různou rychlostí. Například algoritmus PSO_S s nastavenými koeficienty na 0.9 konverguje podstatně rychleji než 0.1.



(a) PSO_B

(b) PSO_R

(c) PSO_S

(d) PSO_I

Obrázek 6.1: Porovnání algoritmů PSO s různými inicializačními koeficienty $c_{1..n}$. Při větším počtu těchto koeficientů jsou všechny nastaveny na stejnou hodnotu, protože jejich kombinace by vyžadovali veliké množství testů.

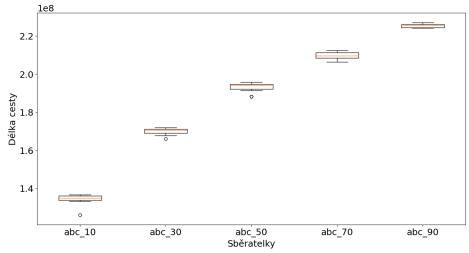
Podle 6.1 dosahuje nejlepších výsledků PSO_I pro hodnotu 0.9. Z ostatních algoritmů si vede nejlépe PSO_S s hodnotou 0.7. Pro PSO_B a PSO_R se jedná o hodnoty 0.1 a 0.5. Tyto algoritmy se ale pohybují v hodnotách horších než PSO_S. Je třeba ale vzít na vědomí, že toto nastavení má veliký rozptyl. Nejlepší parametry pro tyto algoritmy budou použity při dalších experimentech. Ostatní algoritmy nebudou použity v dalších experimentech, protože nedosahují dobrých výsledků, PSO_S bude v dalších experimentech sloužit pro porovnání s PSO_I.

	0.1	0.3	0.5	0.7	0.9
PSO_B	468975749.8	474821883.5	480164506.8	490104395.3	501203220.6
PSO_I	144867306.1	160684266.2	152860234.6	149928174.3	137811022.9
PSO_R	516382089.1	517109041.7	515976795.8	516630164.4	517247616.8
PSO_S	433761568.3	438781529.5	436989664.3	431124403.4	434475486.0

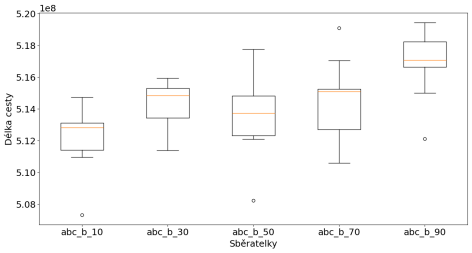
Tabulka 6.1: Porovnání průměrných výsledků podle hodnot koeficientů $c_{1..n}$ PSO z obrázku 6.1. V horním řádku máme hodnotu koeficientů $c_{1..n}$. Na levé straně tabulky jsou algoritmy PSO.

6.2.2 Základní parametry ABC

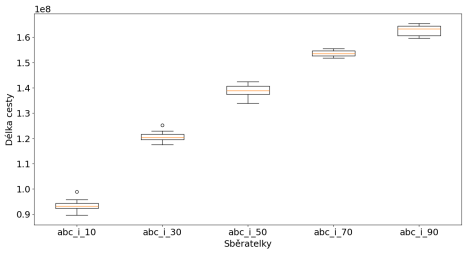
Také jsem porovnal ABC algoritmy se stejnými parametry. Rozdíl je v počtu sběratelek. Tedy včel, které udržují řešení. Menší počet sběratelek znamená, že populace obsahuje větší počet přihlížejících včel. Výsledky jsou zobrazeny v na obrázku 6.2.



(a) ABC_S



(b) ABC_B



(c) ABC_I

Obrázek 6.2: Porovnání algoritmů ABC s různými počty sběratelek. Počty sběratelek se pohybují od 10 do 90, kde s narůstajícím počtem se zhoršuje získané řešení pro všechny uvedené algoritmy.

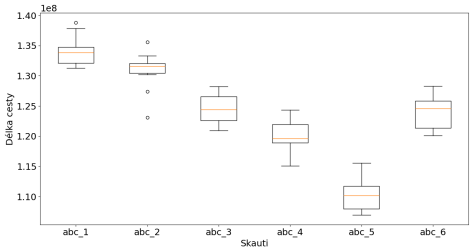
Od ABC_B jsem upustil během vývoje, nicméně je zde uveden pro finální porovnání. Z výsledků je poznat, že ABC_S algoritmus dává podstatně lepší výsledky než PSO_S pro řešení TSP problému. Nicméně rychlost konvergence je zde na rozdíl od PSO problém. Hlavním důvodem, proč dává menší počet sběratelek lepší výsledky je, že algoritmy nestihly dokonvergovat ani s 10 000 iteracemi a s menším počtem sběratelek konvergují o trochu rychleji. Sběratelky totiž udržují jedno z možných řešení. Více sběratelek tedy udržuje více řešení, které se paralelně vylepšují. Každé řešení pak má menší šanci, že jej přihlížející včela zvolí pro vylepšení.

	10	30	50	70	90
ABC_S	134212400.3	169916510.7	193079131.0	209829896.9	225470303.3
ABC_B	512189543.0	514346108.7	513591828.7	514423738.8	516978290.8
ABC_I	93511194.7	120794709.7	138892483.0	153650493.5	162833502.9

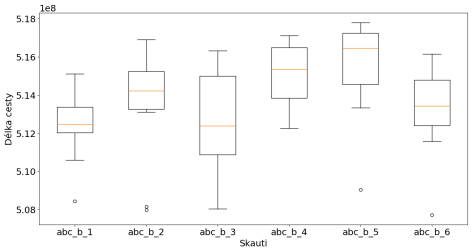
Tabulka 6.2: Porovnání průměrných hodnot dosažených cest ABC algoritmů dle sběratelek z boxplotů 6.2. V horním řádku máme nastavený počet sběratelek. Na levé straně tabulky jsou algoritmy ABC.

V dalších experimentech budu pracovat s 10 sběratelkami, které dávají nejlepší výsledky jak pro ABC_S i ABC_I. Dále jsem otestoval proměnný počet skautů s 10 sběratelkami.

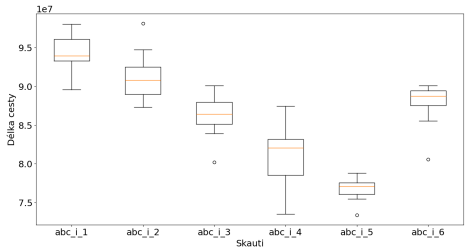
Z boxplotu 6.3 je vidět, že počet skautů nemá tak veliký vliv na výsledek, jako počet sběratelek. Nastavením 5 skautů se získají o trochu lepší výsledky. Rozdílem je binární algoritmus, který dává nejlepší výsledky pro 3 skauty. Počet skautů budu při 10 sběratelkách nastavovat na 5, tedy na polovinu sběratelek zaokrouhleno dolů.



(a) ABC_S



(b) ABC_B



(c) ABC_I

Obrázek 6.3: Porovnání algoritmů ABC s různými počty skautů při 10 sběratelkách

	1	2	3	4	5	6
ABC_S	134063331.8	130795045.8	124398937.1	119826404.5	110246178.5	123880177.4
ABC_B	512422872.7	513511540.4	512671205.9	515124971.9	515456428.2	513247739.2
ABC_I	94368644.7	91343944.7	86170019.6	80923710.3	76817123.7	87766599.7

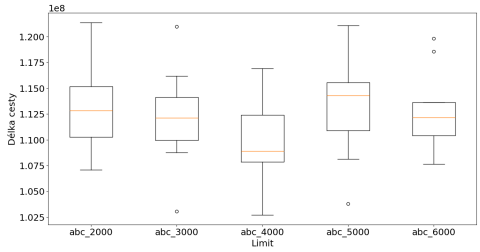
Tabulka 6.3: Porovnání průměrných hodnot cest ABC algoritmů dle skautů z boxplotů 6.3. V horním řádku máme nastavený počet skautů. Na levé straně tabulky jsou algoritmy ABC.

Následuje experiment pro správné nastavení limitu počítadla nepovedených pokusů o vylepšení řešení. To udává, po jakém počtu modifikací, které nezlepší řešení, bude sběratelka přepnuta na skautku a vygeneruje nové řešení. Experiment bude proveden s aktualizovanými parametry podle předchozího experimentu, takže s 5 skauty.

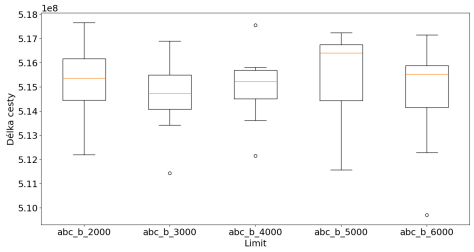
Z boxplotu 6.4 se zdá, že limit počítadla nepovedených pokusů o vylepšení řešení je ideální nastavit na 4 000. Přesnější data jsou uvedená v tabulce 6.4, ze kterých je vidět, že v případě ABC_S je tato hodnota celkem jasná, ale v ABC_I by se také dalo zvolit 6 000. Obecně nemá počítadlo nepovedených pokusů zásadní vliv na výsledek.

	2000	3000	4000	5000	6000
ABC_S	113316106.2	112093697.2	109981495.2	113261365.3	112781019.8
ABC_B	515165649.9	514714854.3	514994570.0	515469256.4	514764214.6
ABC_I	78307744.5	76675567.8	74463462.6	75592742.5	74599119.2

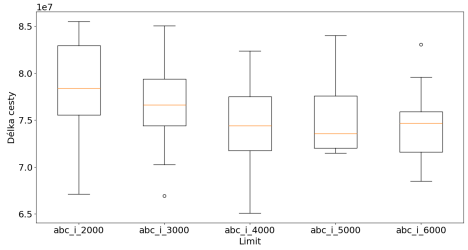
Tabulka 6.4: Porovnání průměrných hodnot dosažených cest ABC algoritmů dle nastavení počítadla nepovedených pokusů z boxplotů 6.4. V horním řádku máme nastavený počet počítadla nepovedených pokusů. Na levé straně tabulky jsou algoritmy ABC.



(a) ABC_S



(b) ABC_B



(c) ABC_I

Obrázek 6.4: Porovnání algoritmů ABC s různými nastaveními počítadla nepovedených pokusů o vylepšení řešení.

V závěru budu volit pro PSO_S c_1 a c_2 0.7, pro PSO_I c_1, c_2 a c_3 0.9. Pro ABC algoritmy budou parametry stejné a to 10 sběratelek, 5 skautů a limit 4 000. Na parametrech pro binární algoritmy nezáleží, protože už s nimi nebudu pracovat. Déle nebudu testovat PSO_R, protože také nedosahuje dostatečně dobrých výsledků.

6.3 Experiment porovnání algoritmů s poměrem populace a iterací

V tomto experimentu již budu pouze porovnávat algoritmy PSO_I, PSO_S, ABC_I, ABC_S. Dále také PSO_I a ABC_I s více hejny označené jako PSO_MSS a ABC_MSS. Také zde porovnám výsledky algoritmů při využití k-optu. Důvod využití k-optu je, že jeho přítomnost by měla mít vliv na ideální poměr populace a iterací. Celý experiment se skládá ze čtyř menších. Algoritmy PSO a ABC jsem porovnal jednotlivě. Nejdříve budu experimentovat na menší instanci o velikost 100 měst, dále na větší o velikosti 4 461 měst a nakonec výsledky mezi instancemi porovnám. Tento experiment bude obsahovat hodně dat. Protože by boxploty spolu s tabulkami zabírali spoustu místa, tak zde uvedu pouze tabulky. Počet běhů je pro všechny experimenty s parametry stejný, tedy 10. Inicializace počátečních cest je náhodná. Parametry algoritmů jsou stejné jako nastavení pro nejlepší výsledky z předchozího experimentu, netestované se nemění. Populace a iterace jsou nastaveny tak, že jejich vynásobení dává 1 000 000, tedy počet provedení fitness funkce. Tabulky uvádějí pro danou populaci nejlepší dosaženou hodnotu, průměr a nejhorší dosaženou hodnotu z 10 běhů. Levá sekce uvádí výsledky s k-opt a pravá uvádí dosažené výsledky bez k-opt. PSO_I využije k-opt po 50 stagnujících iteracích.

6.3.1 Experiment s PSO algoritmy na menší instanci TSP

Tento experiment proběhl na problému dodaném vedoucím práce. Jedná se o problém se 100 městy. Protože se jedná o zeměpisné souřadnice, tak jsou výsledky uvedené v kilometrech. Nastavené parametry odpovídají nejlepším získaným hodnotám v předchozím experimentu a výše uvedeným hodnotám.

Populace	S 3-opt			Bez K-opt		
	Min	Průměr	Max	Min	Průměr	Max
20	6556.2	6724.3	6974.1	24453.4	28711.1	31214.0
50	6446.6	6766.5	7102.9	24860.5	25897.6	27360.6
100	6555.9	6859.7	7242.7	21958.9	24059.1	25299.6
200	6810.3	7276.1	7747.6	21067.0	22332.2	23644.8
400	7237.6	7769.6	8128.8	18963.1	19727.5	20572.5
500	7583.8	8198.8	8591.9	18900.3	19928.2	21029.2

Tabulka 6.5: PSO_S na 100 městech s hodnotami v km. V horním řádku je, zda byl využit k-opt a jaký, pod ním minimum, průměr, maximum. Na levé straně tabulky jsou inicializované velikosti populací tohoto algoritmu.

Z tabulky 6.5 je poznat veliký rozdíl mezi výsledky, pokud se využije k-opt. Navíc to způsobuje zajímavou situaci, kdy PSO_S bez k-opt dává lepší výsledky s větším počtem částic, protože tím lépe prohledá prostor. Naopak k-opt pracuje lépe s větším počtem iterací, protože může lépe prohledat prostor. Tyto dvě situace mají tedy opačnou tendenci z hlediska kvality výsledků v závislosti na populaci. Také lze poznat, že PSO_S dává špatné výsledky i s větším počtem částic. Pro dosažení výsledků podobným těm s využitím k-opt, je potřeba opravdu veliký počet částic.

Populace	S 3-opt			Bez K-opt		
	Min	Průměr	Max	Min	Průměr	Max
20	6512.4	6624.2	6751.1	8820.7	9736.5	11236.2
50	6632.6	6857.5	7107.0	8512.8	9998.2	11406.7
100	6489.7	6765.2	7092.8	8524.9	9854.8	11144.9
200	6476.6	6822.9	7120.2	8724.7	9615.8	11249.6
400	6617.3	6983.9	7298.8	8938.4	9607.0	10185.9
500	6810.0	7310.3	7917.6	8616.5	10009.6	11244.6

Tabulka 6.6: PSO_I na 100 městech s hodnotami v km. V horním řádku je, zda byl využit k-opt a jaký, pod ním minimum, průměr, maximum. Na levé straně tabulky jsou velikosti populace.

Z tabulky 6.6 lze poznat, že vylepšený algoritmus dává podstatně lepší výsledky bez využití k-opt oproti PSO_S, navíc výsledky nejsou tolik závislé na poměru populace a iterací. To je veliký rozdíl od předchozího experimentu, kde je veliký počet částic důležitý pro dobré výsledky algoritmu. Při využití 3-opt dává o něco lepší výsledky než bez něj. Rozdíl ale není tak veliký jako u předchozího experimentu. 3-opt dává lepší výsledky pro veliký počet iterací.

Populace	S 3-opt			Bez K-opt		
	Min	Průměr	Max	Min	Průměr	Max
20	6507.9	6625.4	6765.6	7949.2	8803.1	9404.3
50	6553.4	6689.2	6788.7	7934.0	8847.7	9488.1
100	6404.6	6644.7	6864.7	7214.8	8961.9	10477.2
200	6402.2	6686.9	6913.5	8381.4	9075.1	10093.3
400	6640.7	6962.9	7287.3	8480.2	9268.8	10232.7
500	6977.2	7280.6	7609.6	8950.8	9432.6	10034.3

Tabulka 6.7: PSO_MSS na 100 městech s hodnotami v km. V horním řádku je, zda byl využit k-opt a jaký, pod ním minimum, průměr, maximum. Na levé straně tabulky jsou velikosti populace.

Více hejn v 6.7 s využitím 3-opt nedosahuje lepších výsledků v porovnání s algoritmy bez nich. Bez využití k-opt jsou výsledky lepší a favorizují počet iterací, což dává smysl, protože se nové hejno s jinými parametry vytvoří pouze, pokud se stagnuje po určitou dobu. Je potřeba vzít v potaz, že parametry nejsou vyladěné. To může mít vliv na výsledky více hejn, protože při randomizaci parametrů se může nové hejno vytvořit s vhodnějším nastavením parametrů. Ve finálních experimentech nemusí dosahovat tak velikého zlepšení. Počet stagnujících iterací pro oddělení je 100 a počet iterací odděleného hejna je 250.

6.3.2 Experiment s ABC algoritmy na menší instanci TSP

Proměnné parametry algoritmů jsou opět stejné jako nastavení pro nejlepší výsledky z předchozích experimentů. Při porovnání výsledků je potřeba vzít v potaz, že ABC algoritmus má podstatně menší rychlost konvergence, zejména ABC_S. Pro menší instanci by to ale neměl být problém.

Populace	S 3-opt			Bez K-opt		
	Min	Průměr	Max	Min	Průměr	Max
20	7330.6	7855.2	8198.5	9284.8	9666.9	9975.4
50	7640.5	8427.2	9002.1	8735.5	9632.5	10072.0
100	8247.2	8845.4	9409.1	9098.4	9714.3	10330.1
200	8285.3	9224.9	9871.1	9051.2	9438.2	9873.8
400	9253.0	9592.5	10279.0	8884.1	9657.5	9992.0
500	8963.8	9415.3	9718.4	8818.9	9442.4	10073.5

Tabulka 6.8: ABC_S na 100 městech s hodnotami v km. V horním řádku je, zda byl využit k-opt a jaký, pod ním minimum, průměr, maximum. Na levé straně tabulky jsou velikosti populace.

V tabulce 6.8 je poznat, že pro PSO_I je malý rozdíl mezi algoritmem využívajícím 3-opt a bez něj. K-opt opět funguje lépe s více iteracemi. Pokud se podíváme na průměrné výsledky ABC_S bez k-optu, tak je rozdíl mezi velikostí populace malý a neobsahuje korelaci výsledků s velikostí populace.

Populace	S 3-opt			Bez K-opt		
	Min	Průměr	Max	Min	Průměr	Max
20	6430.7	6549.1	6674.5	8232.9	8665.8	9419.4
50	6388.1	6640.2	6868.8	7576.0	8322.8	8894.6
100	6732.5	6844.9	7049.9	7772.0	8174.0	8694.4
200	6653.9	6900.4	7035.7	7576.5	8026.0	8647.7
400	6963.7	7148.0	7449.6	7345.5	8062.3	8396.6
500	6898.2	7228.4	7416.4	7654.2	8140.8	8879.3

Tabulka 6.9: ABC_I na 100 městech s hodnotami v km. V horním řádku je, zda byl využit k-opt a jaký, pod ním minimum, průměr, maximum. Na levé straně tabulky jsou velikosti populace.

ABC_I v 6.9 dává lepší výsledky jak při využití 3-opt, tak bez něj. K-opt opět favorizuje veliký počet iterací s malou populací. Bez k-opt nejlepší průměrný výsledek dává populace o velikosti 200 jedinců. Rozdíl oproti ostatním ale není zásadní. ABC_I favorizuje velikost populace než počet iterací.

Populace	S 3-opt			Bez K-opt		
	Min	Průměr	Max	Min	Průměr	Max
20	6415.9	6468.6	6550.9	7177.4	8428.5	9005.0
50	6435.2	6529.1	6594.4	7532.5	8323.3	8907.7
100	6449.1	6628.0	6721.4	7335.5	8099.8	8527.3
200	6537.8	6751.0	6959.5	7830.5	8208.4	8518.7
400	6875.5	7062.3	7353.6	7774.5	8043.0	8365.8
500	6976.9	7238.9	7541.1	6976.5	7896.9	8371.5

Tabulka 6.10: ABC_MSS na 100 městech s hodnotami v km. V horním řádku je, zda byl využit k-opt a jaký, pod ním minimum, průměr, maximum. Na levé straně tabulky jsou velikosti populace.

V tabulce 6.10 je z dat vidět, že více hejn nemá stejně jako u PSO žádný podstatný vliv na výsledky pro 3-opt. Pro ABC_MSS bez k-optu se zdá, že dosahuje trochu lepších výsledků, ale jedná se o minimální rozdíl. Počet stagnujících iterací pro oddělení byl 1 000 a počet iterací odděleného hejna byl 500.

6.3.3 Experiment s PSO algoritmy na větší instanci TSP

Tentokrát se jedná o instanci o velikosti 4461 měst. Váha hran byla počítána pomocí Euklidovy vzdálenosti. Nejedná se tedy o zeměpisné souřadnice. Zde je třeba poznamenat, že z časových důvodů je počet iterací menší než ideální. Důvodem je, že algoritmus PSO_S je poměrně pomalý a víc iterací by zabralo moc času. Algoritmy se sice už ve všech případech podstatně nezlepšují, ale nestagnují. Což znamená, že více hejn nebude mít podstatný vliv, protože algoritmus nestagnoval dostatečný počet iterací. Parametry byly až na výjimky stejné, jak v předchozím experimentu. Pro oba algoritmy s více hejny byl počet stagnujících iterací pro oddělení 1 000 a počet iterací odděleného hejna byl 500. Na rozdíl od předchozího testu se pracuje s 2-opt a šance využití tabulky nejbližších sousedů je 75%.

Pop	S 2-opt			Bez K-opt		
	Min	Průměr	Max	Min	Průměr	Max
20	2762012.0	2794296.1	2832464.3	7639200.0	7698132.8	7750206.5
50	4094259.8	4135660.9	4208243.0	7422494.5	7490879.3	7538255.5
100	5176453.5	5219691.6	5292164.5	7217758.0	7357537.2	7438675.5
200	6151935.0	6228363.7	6282700.0	7160522.0	7218560.8	7269136.0
400	6980642.0	7013145.0	7063091.5	6955918.5	7081333.1	7217344.5
500	7150765.0	7194061.0	7241355.0	6934372.5	7001971.8	7085849.0

Tabulka 6.11: PSO_S na 4461 městech. V horním řádku je, zda byl využit k-opt a jaký, pod ním minimum, průměr, maximum. Na levé straně tabulky jsou velikosti populace.

Při použití algoritmu PSO_S na větší instanci je z dat v 6.11 vidět daleko větší rozdíl mezi získanou výslednou cestou bez a při použití 2-opt. Navíc bez opravdu velikého počtu částic není schopen rozumně prohledat prostor. Oproti spuštění na menší instanci se zvýšil rozdíl v kvalitě výsledků. To i v případě využití pouze 2-opt, to je způsobeno malou schopností PSO_S prohledat prostor.

Pop	S 2-opt			Bez K-opt		
	Min	Průměr	Max	Min	Průměr	Max
20	2237778.8	2549727.9	2746319.5	2305308.3	2565757.6	2772669.0
50	2521393.0	2816155.6	3172007.3	2547052.3	2891398.9	3216265.3
100	2673333.0	3068289.0	3658413.5	2837626.8	3119653.9	3508526.3
200	2529574.8	3240591.8	4318634.0	2219234.0	3219939.5	4127118.0
400	2576235.5	3674221.3	4514604.0	2554346.8	3547023.8	4474754.5
500	3012535.5	3624555.8	4851260.5	2437539.0	3418766.6	4811726.5

Tabulka 6.12: PSO_I na 4461 městech. V horním řádku je, zda byl využit k-opt a jaký, pod ním minimum, průměr, maximum. Na levé straně tabulky jsou velikosti populace.

Tabulka 6.12 ukazuje, že PSO_I algoritmus dává podobné výsledky v obou případech. Je ale potřeba vzít v úvahu, že tento algoritmus používá k-opt pouze po určitém počtu stagnujících iterací, v tomto případě k nim tolik nedošlo, proto mezi nimi není takový rozdíl. Dokonce dává lepší výsledky jak PSO_S s 2-opt. Doba běhu je tak menší.

Pop	S 2-opt			Bez K-opt		
	Min	Průměr	Max	Min	Průměr	Max
20	2279348.8	2582397.1	2734576.5	2344908.3	2583548.6	2718698.5
50	2550390.3	2857170.9	3155064.8	2606303.5	2932019.7	3139684.5
100	2813235.8	3293899.2	3727752.0	2425289.5	2991601.9	3667375.3
200	2521992.8	3036736.4	3763481.0	2082253.8	3238968.2	4208676.0
400	2841763.0	3393092.9	4136550.0	2576902.5	3521577.6	4650091.5
500	2507861.0	3774835.1	4720547.0	2463976.3	3441309.7	4687281.0

Tabulka 6.13: PSO_MSS na 4461 městech. V horním řádku je, zda byl využit k-opt a jaký, pod ním minimum, průměr, maximum. Na levé straně tabulky jsou velikosti populace běhu.

V 6.13 není podstatný rozdíl oproti výsledkům z 6.12 z důvodu malého využití hejn, protože se nejedná o kvalitní experiment z hlediska výsledku. Naopak je možné vidět rozdíly v konvergenci. Takže budu při rozhodování o nejlepším poměru populace a iterací pracovat převážně s výsledky získanými na menší instanci. Nicméně tento experiment na rozdíl od experimentu s malou velikostí instance dobře ukazuje rychlost konvergence, takže bude brán v potaz při volení ideálního poměru.

6.3.4 Test ABC algoritmů na větší instanci TSP

Parametry se opět neliší od předchozího testu. Dodržuji zde počet iterací pro konzistenci i když se ABC algoritmy provedou daleko rychleji v porovnání s PSO. Jejich konvergence je ale pomalejší.

Pop	S 2-opt			Bez K-opt		
	Min	Průměr	Max	Min	Průměr	Max
20	2293168.4	2323925.7	2359791.7	2524792.3	2570051.6	2615819.1
50	2223507.6	2258981.6	2286591.4	2315549.9	2354839.4	2392907.5
100	2200671.4	2244137.5	2265921.2	2257952.2	2298061.1	2353329.6
200	2215154.0	2251765.4	2284715.9	2256259.4	2282932.2	2314862.7
400	2215154.0	2251765.4	2284715.9	2204520.6	2257675.4	2307340.8
500	2213532.8	2243872.3	2288306.2	2209668.0	2264513.4	2293894.6

Tabulka 6.14: ABC_S na 4461 městech. V horním řádku je, zda byl využit k-opt a jaký, pod ním minimum, průměr, maximum. Na levé straně tabulky jsou velikosti populace běhu.

V tabulce 6.14 ABC_S opět dává solidní výsledky i v porovnání s PSO_I. Při využití 2-opt nebo bez něj, nejsou výsledné hodnoty silně závislé na poměru populace a iterací. Při využití k-opt dává trochu lepší výsledky při menší velikosti populace. Bez k-opt se naopak získají trochu lepší výsledky při větší populaci. Tyto výsledky mají podobný trend jako výsledky experimentu pro menší velikost instance. Oproti němu se ale zdá, že u větších instancí není tak veliká závislost výsledků na poměru populace a iterací. Je celkem zajímavé, že z hlediska konvergence se algoritmus bez 2-opt blíží algoritmu, který jej využívá.

Pop	S 2-opt			Bez K-opt		
	Min	Průměr	Max	Min	Průměr	Max
20	1139241.8	1159599.9	1169960.1	1821495.1	1884232.8	1917290.0
50	1182885.3	1224565.8	1255769.3	1634739.8	1691804.9	1758563.0
100	1293116.3	1322087.1	1348304.3	1589392.2	1625220.6	1657549.2
200	1304875.8	1398771.9	1462215.5	1559968.0	1602809.6	1633538.0
400	1418967.1	1460560.8	1490208.2	1518296.6	1580553.2	1641133.3
500	1450004.1	1473815.7	1504888.5	1530866.7	1576784.3	1622945.8

Tabulka 6.15: ABC_I na 4461 městech. V horním řádku je, zda byl využit k-opt a jaký, pod ním minimum, průměr, maximum. Na levé straně tabulky jsou velikosti populace běhu.

V 6.15 je vidět, že ABC_I algoritmus dosahuje podstatně lepších výsledků než ABC_S z předchozího experimentu. Toto může být samozřejmě z části způsobené rychlejší konvergencí ABC_I. Oproti algoritmu z předchozího experimentu má ABC_I větší závislost na správném zvolení poměru populace a iterací. Víc iterací je preferováno pro k-opt a větší populace bez něj. Zde je korelace velikosti populace a počtu iterací v obou případech zřejmá.

Pop	S 2-opt			Bez K-opt		
	Min	Průměr	Max	Min	Průměr	Max
20	1121190.5	1149266.2	1170534.4	1844083.3	1888226.4	1931455.5
50	1188730.0	1231590.7	1262036.8	1626337.9	1666245.1	1714473.8
100	1295279.2	1329874.8	1380353.0	1583809.5	1637452.9	1697930.3
200	1370557.0	1417659.4	1478135.6	1575737.1	1606578.8	1650229.6
400	1398580.6	1458727.0	1491126.1	1569699.0	1602735.2	1629910.4
500	1427301.8	1467326.0	1511453.3	1537987.2	1573505.1	1594942.3

Tabulka 6.16: ABC_MSS na 4461 městech. V horním řádku je, zda byl využit k-opt a jaký, pod ním minimum, průměr, maximum. Na levé straně tabulky jsou velikosti populace běhu.

V 6.16 se opět nejedná o podstatný rozdíl oproti ABC_I z důvodu malého využití hejn. Ty se tolik nevytváří, protože algoritmus nezačal dostatečně stagnovat.

6.3.5 Shrnutí poměru populace a iterací

První rozdíl při porovnání výsledků menších a větších instancí je ten, že PSO_I konverguje rychleji a dává lepší výsledky pro menší populaci a více iterací. ABC_I naopak dává lepší výsledky pro větší populaci. Malý vliv k-optu pro ABC_S při větší instanci byl celkem překvapivý. Experimenty na obou instancích ukazovaly, až na menší výjimky, podobný trend.

Z dat jsem zvolil poměry populace/počet iterací pro PSO:

- PSO_S - 1/4,
- PSO_S s k-opt - 1/2500,
- PSO_I - 1/2500,
- PSO_I s k-opt - 1/2500.

Pro ABC jsem zvolil poměry populace/počet iterací:

- ABC_S - 1/4,
- ABC_S s k-opt - 1/2500,
- ABC_I - 1/4,
- ABC_I s k-opt - 1/2500.

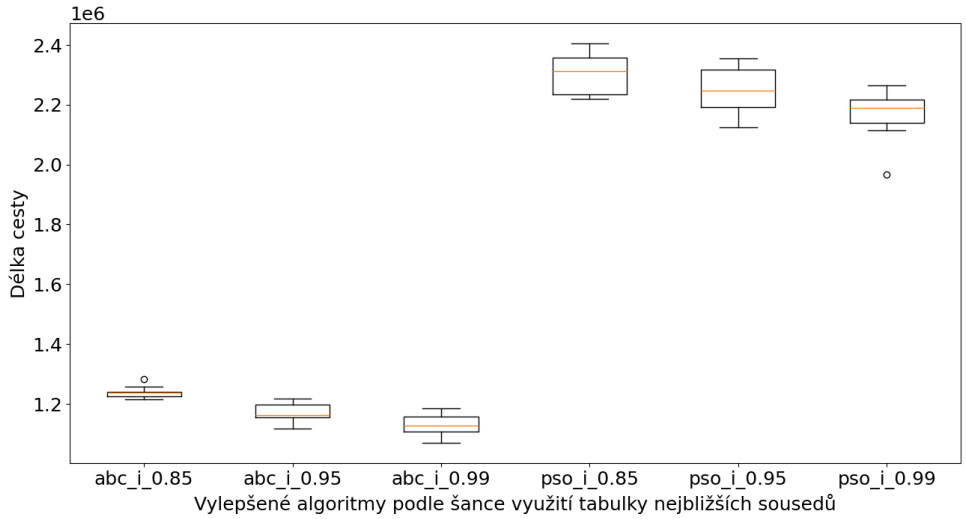
Pro algoritmy s více hejny se budu řídit stejnými parametry, jako pro originální algoritmy. V základu ukazovaly stejný trend. Poměry nebudu dodržovat přesně do částice. Vzhledem k tomu, že získané hodnoty také nejsou na částici přesné, takže by to nemělo

smysl. V následujících testech je budu převádět na blízká čísla. Cílem bude mít stejný počet provedení fitness funkce pro všechny algoritmy. Největší problém jsou algoritmy ABC s pomalejší konvergencí. Zde musím vzít v potaz, že pro současný test fnl4461 algoritmy nestihly dokonvergovat a výsledky se tedy mohou lišit, ale rychlost konvergence je taky metrika, kterou beru na vědomí při volení populace/iterace. Budu tedy volit daný poměr s tendencí zvyšovat počty iterací. Konvergence bude zejména důležitá pro veliké instance, kde je potřeba pro získání solidního výsledku jak hodně částic, tak i hodně iterací.

6.4 Experiment s pravděpodobností využití tabulky sousedů

Kromě základních parametrů máme ještě parametry specifické pro navržené algoritmy. Tyto experimenty budou provedeny pouze pro algoritmy bez využití více hejn. Více hejn by nemělo mít zásadní vliv na nastavení parametrů. Správné nastavení parametrů může ovlivnit rozdíl mezi algoritmem a výsledky při využití hejn. Důvodem je, že při správném nastavení parametrů, nevytvoří hejno s lepším nastavením. Tabulku nejbližších sousedů využívají PSO_I a ABC_I. Proto je tento experiment proveden pro oba algoritmy. Velikost populace je pro ABC_I algoritmus 625 s počtem iterací 2 800. Algoritmus PSO_I má populaci o velikosti 25 s počtem iterací 70 000. Parametry budou stejné jako u předchozího testu. Experiment je opět proveden pro instanci fnl4461.

V předchozí kapitole jsem uvedl princip tabulky nejbližších sousedů. Při prohození nebo vložení se tato tabulka využije s určitou pravděpodobností. Předpokládám, že při větší pravděpodobnosti využití tabulky, dostanu lepší výsledky.



Obrázek 6.5: Porovnání algoritmů dle šance využití tabulky nejbližších sousedů

V boxplotu 6.5 představuje číslo za algoritmem šanci využití tabulky. Čím je tato šance větší, tím lepší výsledky dostaneme v obou algoritmech. Pro udržení náhodnosti jsem maximální šanci nastavil na 99%. Má tedy stále 1% šanci na úplně náhodné prohození nebo vložení.

Výsledky z boxplotu 6.5 i tabulky 6.17 jsou celkem jednoznačné. Proto budu v následujících experimentech pracovat s šancí využití tabulky nastavenou na 99%.

	0.85%	0.95%	0.99%
ABC_I	1239214.8	1170879.6	1130084.6
PSO_I	2303615.1	2245131.2	2168330.2

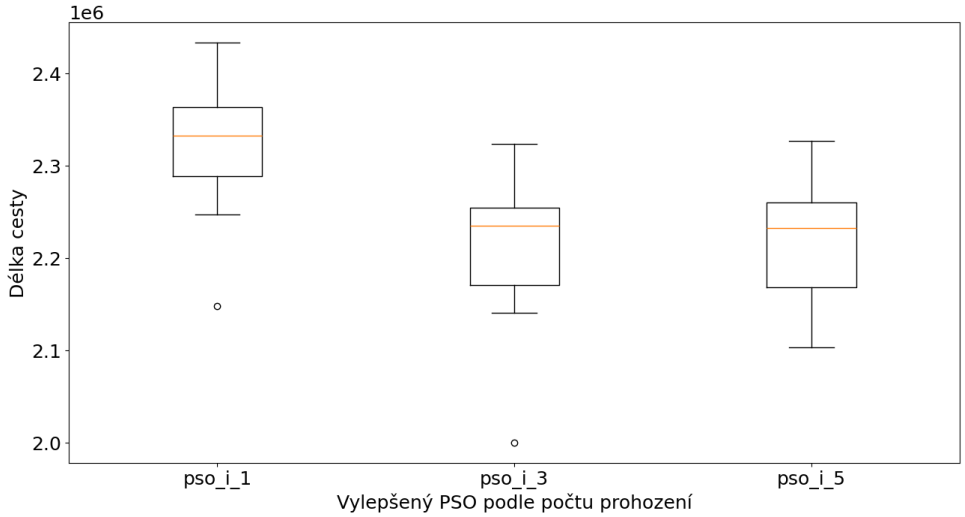
Tabulka 6.17: Porovnání průměrných hodnot dosažených cest algoritmů dle šance využití tabulky nejbližších sousedů při prohození z boxplotu 6.5. V horním řádku je šance využití této tabulky při prohození nebo vložení. Na levé straně tabulky jsou algoritmy.

6.5 Experimenty parametrů PSO_I

Experimenty jsou opět provedeny pro instanci fnl4461, pokud nebude uvedeno jinak. PSO_I má nastavenou populaci o velikosti 25 s počtem iterací 70 000, pokud nebude uvedeno jinak. Parametry jsou odvozeny z předchozích experimentů. Každý experiment zde bude mít vlastní podsekci a po jeho provedení budou jeho nejlepší parametry využity u následujících experimentů.

6.5.1 Experiment vlivu počtu prohození

Počtem prohození se v tomto případě myslí náhodná prohození, která se provádí převážně pokud se řešení blíží nejlepšímu. Dá se předpokládat, že větší počet zlepší konvergenci a umožní překonat některá lokální minima, ale zhorší schopnost vylepšit řešení na maximum. Tím je myšleno, že pokud máme řešení, které pro vylepšení potřebuje méně jednotlivých prohození, tak více prohození v jednom kroku bude nevýhodou. Proto se dá předpokládat, že se zvyšujícím počtem prohození se bude zlepšovat i kvalita řešení, dokud se nenarazí na hranici, od které začne klesat.



Obrázek 6.6: Porovnání PSO_I dle počtu prohození

V boxplotu 6.6 je počet prohození uveden za algoritmem. Je poznat, že $3-5$ prohození je podstatně lepší než pouze 1. Mezi výsledky více prohození ale není podstatný rozdíl. Dá se také předpokládat, že 5 prohození bude představovat hranici, za kterou už se bude kvalita řešení zhoršovat.

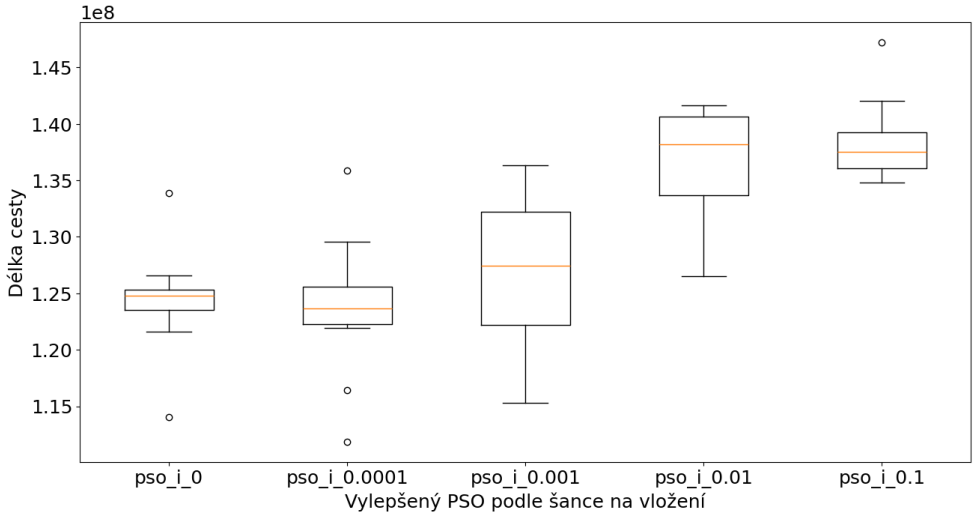
Nakonec jsem se rozhodl zvolit 3 prohození, protože dosahují trochu lepšího průměru 2206428.5. Dalo by se namítnou, že 5 prohození je skoro stejně dobrých, protože medián je stejný, akorát dosahuje většího rozptylu. Průměrná hodnota pro 5 prohození je 2216991.7.

6.5.2 Experiment vlivu šance na vložení

Experiment šance na vložení jsem pro PSO_I upravil. Z důvodu trochu neočekávaného výsledku při prvotním pokusu, jsem se rozhodl experiment provést znovu. Provedu ho dvakrát, jednou na větší instanci a poté na menší. Zde jsem žádné jiné parametry experimentu neměnil. Důvodem je, že chci také změřit a ukázat rychlost provedení algoritmu. Poté test znovu provedu na menší instanci, kde ukážu rozdíl výsledků. Tento experiment bude tedy detailnější než ostatní experimenty.

Experiment vlivu šance na vložení na větší instanci

První část experimentu bude provedena na instanci dsj1000. Jedná se o instanci menší než fnl4461, ale při porovnání s výsledky na ještě menší instanci, nastíní trend vlivu na výsledek při změně velikosti instance.



Obrázek 6.7: Porovnání PSO_I dle šance na vložení

Z boxplotu 6.7 je vidět, že s menší šancí na vložení dostáváme lepší výsledky. Mezi nulovou šancí a minimální není podstatný rozdíl. Je nutné poukázat, že malá šance na vložení zde dává trochu lepší výsledky, ale nic, podle čeho by se dalo říct, že vliv na výsledek je zásadní. Veliká šance na vložení už dává výsledky podstatně horší.

Průměrné hodnoty v 6.18 potvrzují výsledky z boxplotu a ukazují, že pro instanci o velikosti 1 000 je lepší nevkládat žádné prvky nebo nepodstatné množství. V základu to dává smysl, protože změní cestu částice na tolik, že buď bude delší dobu hejno cestovat k ní nebo ona k němu.

Z dob běhů v sekundách v druhém řádku tabulky 6.18 je poznat, že prohození má pro veliké instance veliký vliv nejen na výsledek, ale také na dobu běhu. V obou případech ale negativní. Hlavně vliv na rychlost běhu je velmi zásadní, 51-krát delší doba běhu je veliký

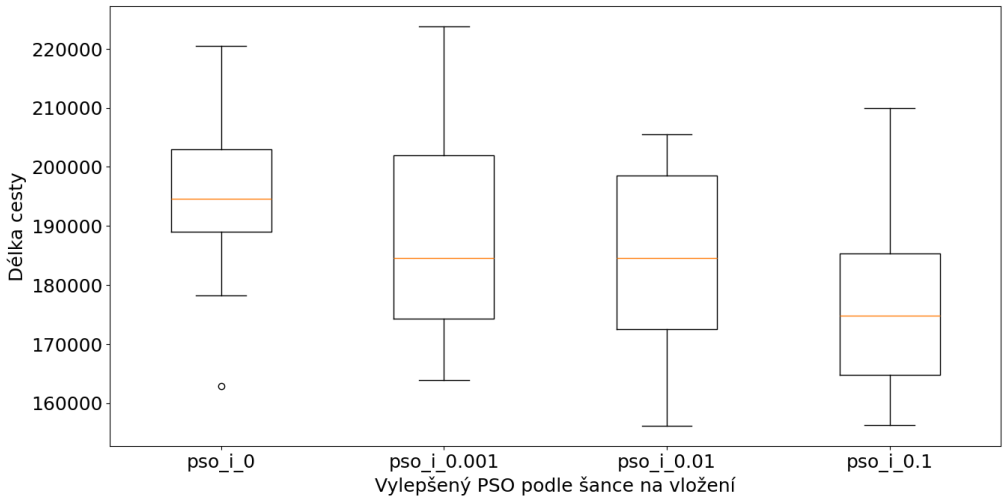
rozdíl, hlavně pokud nedojde ke zlepšení kvality výsledku. Dá se předpokládat, že pro větší instance se zhorší jak výsledek, tak doba běhu.

	0%	0.001%	0.001%	0.01%	0.1%
délka cesty	124399643.7	123666789.2	127235619.9	136488342.0	138526563.3
doba běhu	2.623s	23.157s	64.436s	119.878s	134.153s

Tabulka 6.18: Porovnání průměrných hodnot dosažených cest PSO_I dle šance na vložení z boxplotu 6.7. V horním řádku je šance vložení. Na levé straně tabulky jsou výsledné délky cest a doba běhu algoritmu.

Experiment vlivu šance na vložení na menší instanci

Pro dobrou ilustraci vlivu vložení na výsledek jsem tento experiment také provedl na menší instanci pr136 s populací 15 a počtem iterací 20 000. Důvodem je opačný trend vlivu šance vložení na výsledek pro menší instance.



Obrázek 6.8: Porovnání PSO_I dle šance na vložení

Z boxplotu 6.8 je vidět zmíněný opačný trend, kde větší šance na prohození dává lepší výsledky. Vzhledem k délkám nalezených cest je tento rozdíl poměrně zásadní. Proto je pro menší instance dobré mít nenulovou pravděpodobnost vložení prvku i pro algoritmus PSO, pokud chceme dosáhnou dobrého výsledku.

	0%	0.001%	0.01%	0.1%
délka cesty	193904.9	189068.8	183967.2	176903.3
doba běhu	0.238s	0.242s	0.626s	1.408s

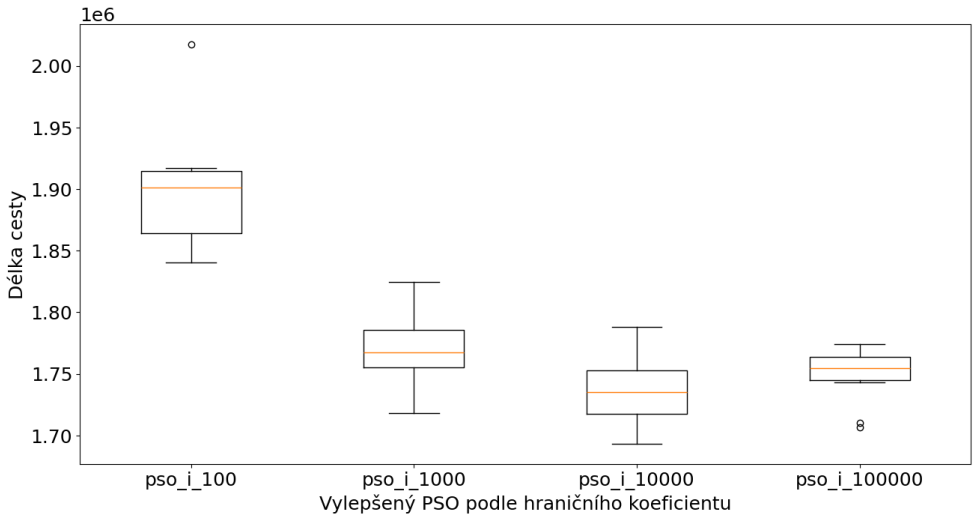
Tabulka 6.19: Porovnání průměrných hodnot dosažených cest PSO_I dle šance na vložení z boxplotu 6.8. V horním řádku je šance vložení. Na levé straně tabulky jsou výsledné délky cest a doba běhu algoritmu.

Tabulka 6.19 ukazuje stejný trend jako boxplot, kde je poznat, že se zvyšující šancí na vložení se zlepšuje kvalita výsledku.

Se zvětšenou šancí na vložení roste i časová náročnost. Neroste ale tolik jako při velikých instancích. To se dá očekávat, protože při provedení vložení se částice nevzdálí tolik, jako při větších instancích. Protože jsem před těmito experimenty lehce testoval parametry na meších iteracích, tak jsem předpokládal, že větší šance na vložení bude dobrá, ale z výsledků mohu usoudit, že pro větší instance je velmi nevhodná. Pro další experimenty bude nastavená na 0%.

6.5.3 Experiment pro nastavení ideální hodnoty hraničního koeficientu

Experiment pro nastavení ideální hodnoty hraničního koeficientu bude posledním experimentem pro správné nastavení parametrů PSO_I. Správným nastavením určíme hranici domény, ve které budou částice prozkoumávat prostor a mimo tuto doménu se budou blížit nejlepší nalezené hodnotě. Tento test je opět proveden na instanci fnl4461 s populací 25 a počtem iterací 70 000.



Obrázek 6.9: Porovnání PSO_I dle hraničního koeficientu

Z výsledků 6.9 se dá vyčíst, že nejlepší nastavení koeficientu je 10 000. Kde mezi hodnotami 1 000 a 100 000 není takový rozdíl, ale hodnota 100 dává podstatně horší výsledky. Koeficient pracuje s vysokými čísly kvůli veliké délce cesty.

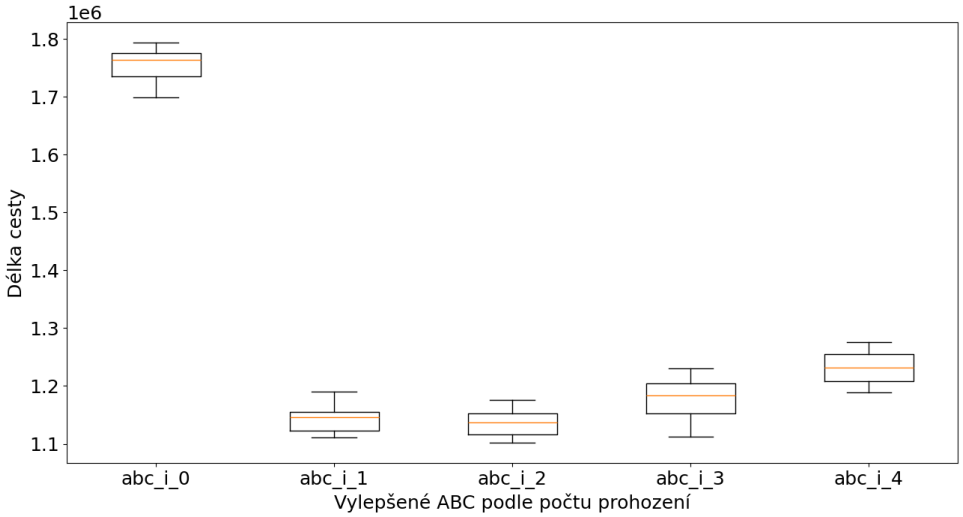
Při pohledu na trend v průměrných hodnotách. Hodnota 10 000 se ukazuje jako nejlepší s výslednou délkou cesty 1735642.2. Následoval by pak 100 000 výslednou délkou cesty 1749333.7. Proto s hodnotou 10 000 budu pracovat v následujících experimentech.

6.6 Experiment parametrů ABC_I

Všechny experimenty v této sekci jsou provedeny pro instanci fnl4461. Velikost populace pro ABC_I a ABC_MSS je 625 s počtem iterací 2800. Parametry budou stejné jako u předchozího testu. Každý experiment zde bude mít vlastní podsekci a po jeho provedení budou jeho nejlepší parametry využity u dalších experimentů.

6.6.1 Experiment vlivu počtu prohození

Stejně jako u výše uvedeného experimentu pro PSO algoritmus je cílem zjistit ideální počet prohození, při kterém se již nebude zhoršovat kvalita řešení. Opět je snahou najít správnou hranici, za kterou se výsledné řešení začne zhoršovat.



Obrázek 6.10: Porovnání ABC_I dle dodatečného počtu prohození

Z boxplotu 6.10 je poznat, že nejlepší hodnotu dávají 1 nebo 2 prohození, se zvyšujícím počtem se pak kvalita řešení snižuje. Zhoršení se zvyšujícím počtem prohození se dá očekávat, překvapivé je, jak rychle se kvalita řešení zhoršuje. Skok mezi 0 dodatečných prohození a 1 je trochu překvapivý, ale je pravděpodobně způsoben pomalou konvergencí.

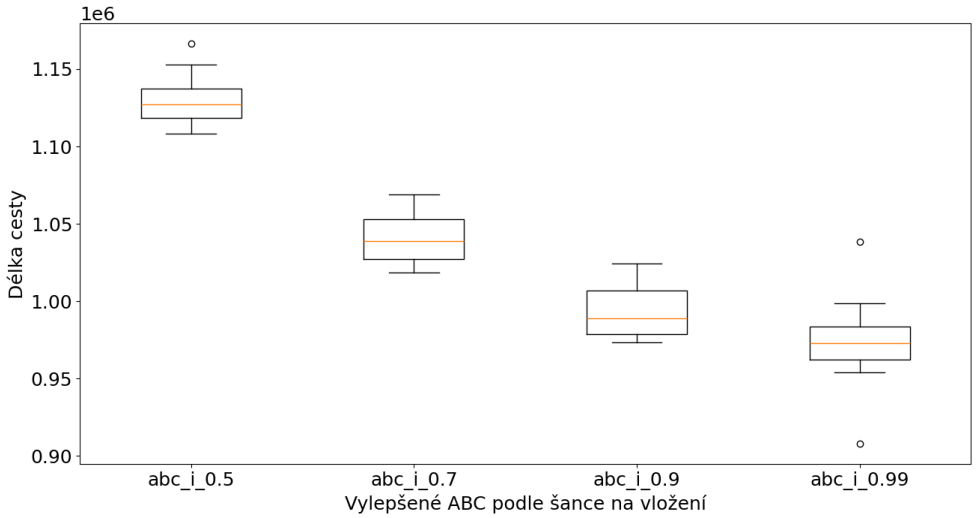
Z těchto výsledků jsem se rozhodl v následujících experimentech pracovat se 2 dodatečnými prohozeními, které dávají v průměru délku cesty 1134727.6. Nastavení 1 dodatečné prohození je jen o trochu horší s průměrnou délkou cesty 1143246.6.

6.6.2 Experiment s šancí na vložení

Oproti experimentu s PSO_I zde nerozděluji experiment na dva pod experimenty. Cílem je zjisti ideální šanci na vložení. U PSO algoritmů je ideálně nulová šance, ale ABC algoritmy pracují na principu vhodnějším pro tuto akci. Protože po vložení se nemusí ostatní částice posunovat směrem k nové hodnotě, ale včely s ní mohou rovnou pracovat.

Z boxplotu 6.10 je poznat, že čím větší šance na vložení, tím je lepší výsledek. Protože se okamžitě využijí upravená řešení, nemá operace vložení zásadní vliv na dobu trvání algoritmu a nepřináší negativní vliv na průběh. Vkládání je tedy vhodnější operací pro ABC algoritmy.

Nejlepší nastavení šance na vložení je 99% s průměrnou výslednou délkou cesty 973436.1. Nechávám procento šance na prohození pro udržení náhodnosti. S touto šancí na vložení budu pracovat v následujících algoritmech.



Obrázek 6.11: Porovnání ABC_I dle šance na vložení

6.7 Experiment s dodatečnými parametry algoritmů s více hejny

Tyto experimenty budou provedeny pouze pro algoritmy využívající více hejn. Tento experiment bude rozdělen pro oba algoritmy, jeden pro PSO_MSS a druhý pro ABC_MSS. Každý experiment pro jednotlivý algoritmus bude složen ze dvou menších experimentů. Nejdřív se otestuje počet stagnujících iterací pro oddělení hejna, kde bude počet iterací odděleného hejna konstantně nastavených na 500. Po zjištění ideálního počtu stagnací bude v dalším experimentu cílem zjistit ideální počet iterací oddělených hejn pro toto nastavení počtu stagnací. Ostatní parametry obou algoritmů vychází z předchozích testů. Všechny pod experimenty budou provedeny pro instanci fnl4461. Názvy algoritmů v boxplotech a tabulkách budou uvedeny ve formátu (zkratka názvu)_(počet stagnací)_(počet iterací).

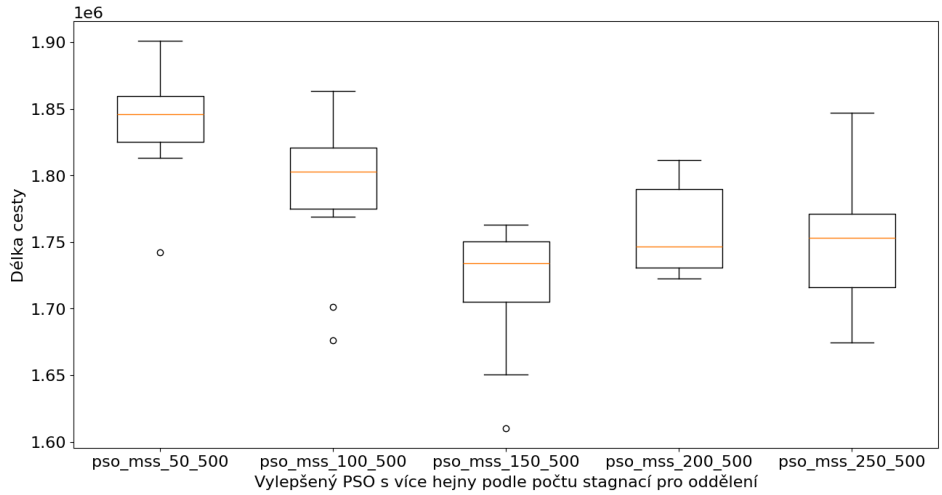
6.7.1 PSO_MSS

Algoritmus PSO_MSS má populaci o velikosti 25 s počtem iterací 70 000. Více hejn jsem navrhoval převážně pro algoritmus PSO, který by mohl podávat lepší výsledky díky rozdílu v mutacích různých hejn. Je sice možné, že se mutace u nového hejna sníží, to ale nemusí být nutně špatné, protože po dokonvergování k řešení je méně mutací vhodnějších pro konečná zlepšení. Zde předpokládám zásadnější vliv existence více hejn na výsledek.

Experiment počtu stagnací pro oddělení

Nejdříve se provede experiment s proměnným počtem stagnací algoritmu pro vytvoření nového hejna.

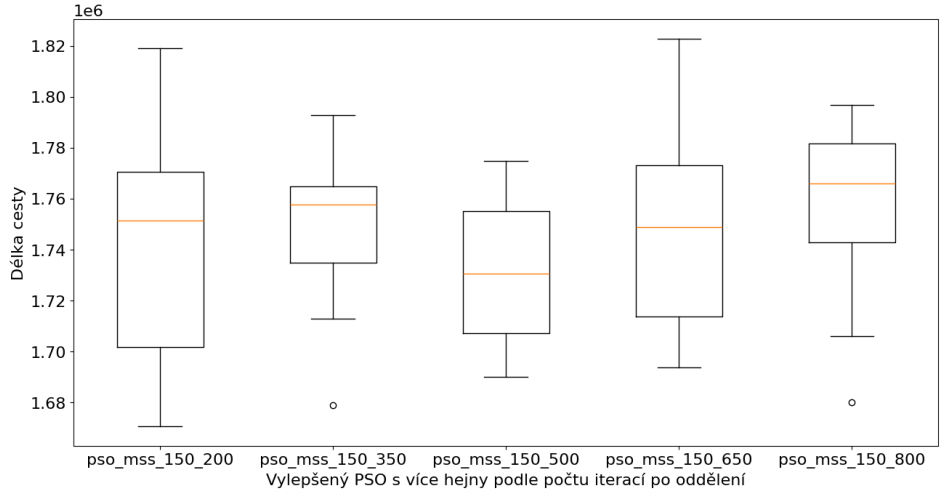
Z boxplotu 6.12 je poznat, že s větším počtem stagnací se výsledek zlepšuje. Když dosáhne hranice 150 stagnací pro oddělení, tak se začne mírně zhoršovat. Z hlediska mediánů se zdá, že dosažená délka výsledné cesty spíše stagnuje po hodnotě 150 stagnací. Tato hodnota dosahuje průměrné délky výsledné cesty 1716575.6. Stagnace může být způsobená vytvářením malého hejn. Pro další část experimentu budu volit 150 stagnací.



Obrázek 6.12: Porovnání PSO_MSS podle počtu stagnací pro vytvoření nového hejna ze stávajícího

Experiment počtu iterací po oddělení

Po získání ideálního počtu stagnací pro vytvoření nového hejna je potřeba zjistit ideální počet iterací odděleného hejna.



Obrázek 6.13: Porovnání PSO_MSS podle počtu iterací nového hejna po oddělení

Z boxplotu 6.13 se nezdá, že by počet iterací měl zásadní vliv na výsledek. Netestoval jsem počet iterací po oddělení menších než 200, protože by pak oddělené hejno nemohlo vytvářet nová, protože je tato hodnota menší než doba stagnování pro oddělení. Nicméně 500 iterací dává nejlepší medián. Nejlepší výsledek našlo 200 iterací, ačkoliv spolu s 650 má veliký rozptyl výsledných hodnot. Po hranici 500 iterací se zdá, že se výsledek mírně zhoršuje.

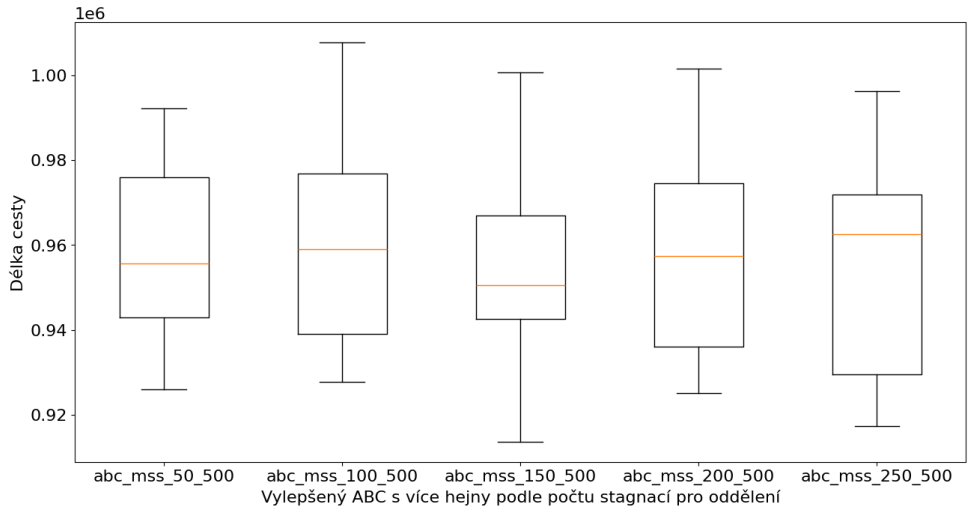
Nastavení 500 iterací má nejlepší průměrnou hodnotu výsledné cesty 1732359.0. Výsledky mě trochu překvapily, protože jsem očekával větší vliv počtu iterací na výsledek, kde by se s větším počtem výsledek zlepšoval, dokud by nedosáhl na hranici, od které by se začal zhoršovat. Je pravda, že tento trend na první pohled v tabulce a boxplotu existuje, ale rozdíly v hodnotách jsou velmi malé. Protože má nastavení 500 iterací nejlepší medián a průměrnou hodnotu, budu s ním ve finálních experimentech pracovat.

6.7.2 ABC_MSS

Velikost populace je 625 s počtem iterací 2 800. Na rozdíl od PSO neočekávám zásadní vliv více hejn na výsledek. Důvodem je, že ABC nemůže provádět velké množství mutací. To by s velikou pravděpodobností nezlepšilo cestu, ale naopak zhoršilo a včely by tedy zvolili předchozí cestu místo této nové. Více hejn by tedy nemělo dávat podstatně lepší výsledky u ABC algoritmu.

Experiment počtu stagnací pro oddělení

Nejprve provedeme experiment s proměnným počtem stagnací ABC_I pro vytvoření nového hejna.



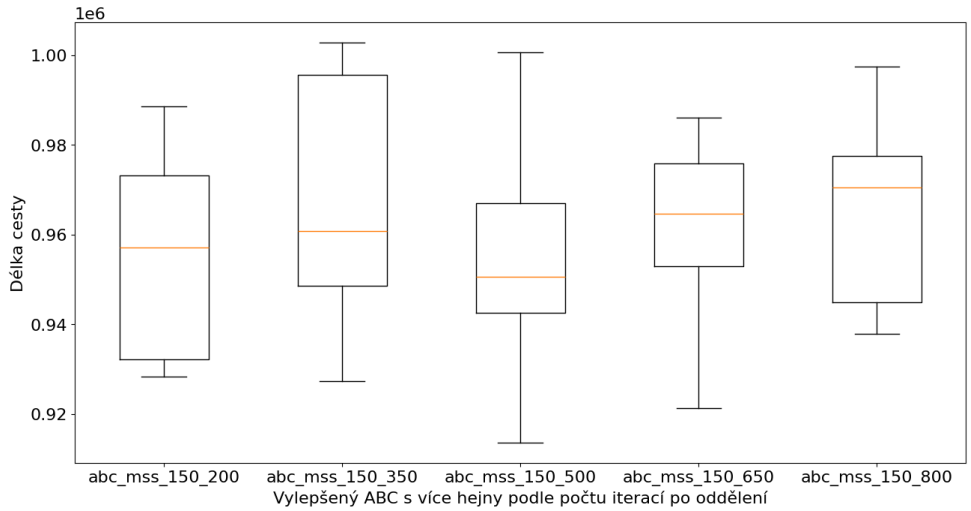
Obrázek 6.14: Porovnání ABC_MSS podle počtu stagnací pro vytvoření nového hejna ze stávajícího

Z boxplotu 6.14 je poznat, že počet stagnací pro oddělení nemá zásadní vliv na výsledek. Nastavení na 150 zde dává nejlepší medián a nachází nejlepší hodnotu, ale rozdíl je tak malý, že znovu provedení experimentu by mohlo podat jiné výsledky.

Nejlepší průměrnou cestu je možné dostat s nastavením na 250, která dosahuje 954609.5. Protože nastavení na 150 dosahuje nejlepšího výsledku a dává nejlepší medián, budu pracovat v dalších experimentech, i když nenachází nejlepší průměrnou cestu. Průměrná délka výsledné cesty pro 150 je 956000.0.

Experiment počtu iterací po oddělení

Po získání ideálního počtu stagnací pro vytvoření nového hejna je potřeba zjistit ideální počet iterací odděleného hejna. Netestoval jsem počet iterací po oddělení menších než 200 ze stejného důvodu jako u předchozího experimentu s PSO algoritmem. Tedy, že by se z oddělených hejn nevytvářelo další heno.



Obrázek 6.15: Porovnání ABC_MSS podle počtu iterací nového hejna po oddělení

Z boxplotu 6.15 je opět vidět minimální vliv na výsledky. Nejlepší hodnotu a medián dává 500 iterací. Má ale také největší rozptyl hodnot. Mediány všech nastavení byli velmi podobné, proto je těžké zvolit nejlepší nastavení.

	200	350	500	650	800
délka cesty	955095.2	968732.0	956000.0	960492.2	964748.5

Tabulka 6.20: Porovnání průměrných hodnot dosažených cest ABC_MSS podle počtu iterací po oddělení

Nejlepší průměrnou výslednou cestu 955095.2 je možné dostat s nastavením na 200, které je následované 500 iteracemi s hodnotou 956000.0. Velikost rozdílu se podobá předchozímu experimentu se stagnacemi. Protože nastavení na 500 dosahuje nejlepšího výsledku a dává nejlepší medián, tak s ním budu pracovat v dalších experimentech. Nicméně zvolit nastavení 200 iterací je také rozumná volba. Jak bylo uvedeno výše, nezdá se, že by různá nastavení měla vysoký vliv na výsledek pro ABC_MSS.

6.8 Porovnání dob běhů algoritmů

Jedním z cílů bylo také odstranit zbytečné operace z PSO, proto by bylo dobré porovnat doby průběhů algoritmů pro ujištění, jestli došlo ke zlepšení a jak je zásadní. K ABC byly naopak další operace přidány, proto by bylo dobré zkontrolovat, zda nedošlo k zásadnímu zhoršení. Porovnání proběhlo na dvou instancích, a to bier127 a fnl4461. Rychlost byla

měřena v sekundách se zaokrouhlením na 3 desetinná místa. Parametry byly odvozené z předchozích testů. Populace a počet iterací byly pro všechny algoritmy v obou instancích stejné a to 50 a 1000. Vzhledem k tomu, že se zde klade zaměření na doby běhů, tak není třeba porovnávat výsledky.

Populace	S 2-opt			Bez K-opt		
	Min	Průměr	Max	Min	Průměr	Max
PSO_S	0.669	0.682	0.712	0.087	0.129	0.175
PSO_I	0.039	0.042	0.048	0.038	0.039	0.041
PSO_MSS	0.039	0.041	0.044	0.039	0.048	0.066
ABC_S	0.045	0.048	0.06	0.026	0.026	0.027
ABC_I	0.048	0.05	0.061	0.04	0.041	0.043
ABC_MSS	0.047	0.049	0.054	0.04	0.041	0.042

Tabulka 6.21: Porovnání dob běhů na bier127. V horním řádku je, zda byl využit k-opt a jaký, pod ním minimum, průměr, maximum. Na levé straně tabulky jsou algoritmy.

V tabulce 6.21 je vidět, že na instanci o velikosti 127 proběhly všechny algoritmy pod sekundu, ačkoliv i tady je už poznat, že PSO zaostává za ABC. Trochu zajímavé je, že můj PSO_I s 2-optem dává lepší výsledky než ABC_I s 2-optem. To je dáno tím, že ABC ve všech algoritmech provádí více 2-opt operací. Na této malé instanci všechny algoritmy proběhly poměrně rychle, proto z časového hlediska není problém zvolit žádný z algoritmů.

Populace	S 2-opt			Bez K-opt		
	Min	Průměr	Max	Min	Průměr	Max
PSO_S	438.709	440.303	442.571	67.945	116.447	204.068
PSO_I	2.289	2.349	2.516	2.321	2.353	2.434
PSO_MSS	2.287	2.377	2.57	2.267	2.325	2.401
ABC_S	2.326	2.357	2.396	2.111	2.124	2.161
ABC_I	2.221	2.245	2.299	2.196	2.221	2.269
ABC_MSS	2.203	2.25	2.371	2.191	2.231	2.343

Tabulka 6.22: Porovnání dob běhů na fnl4461. V horním řádku je, zda byl využit k-opt a jaký, pod ním minimum, průměr, maximum. Na levé straně tabulky jsou algoritmy.

Na datech 6.22 z instance o velikosti 4461 je poznat, že se doby běhů zvýšily neúměrně. PSO_S vyskočilo na opravdu vysoké hodnoty. Zejména pro 2-opt, kde není problém jen operace, ale také dolet částic zpět k nejlepší nalezené cestě. Zatímco PSO_I má díky zjednodušení operací dobu průběhu podstatně menší. V tomto příkladu se ale nestihl aplikovat u PSO_I 2-opt algoritmus, proto je třeba počítat, že pro větší počet iterací by se výsledek podstatně zhoršil. Navíc má PSO_I o něco horší nárůst doby průběhu v porovnání s ABC algoritmy. Všechny ABC algoritmy dosahují velmi dobrých dob průběhu a není třeba zde nic zlepšovat. Dále je možné vidět, že i pro veliké instance je malý rozdíl mezi algoritmy s více hejny a bez nich, protože nedošlo k dostatečné stagnaci.

6.9 Experiment porovnání algoritmů na menších instancích TSP

Pro PSO_S a ABC algoritmy bez k-opt byl zvolen počet částic 160 na 625 iteracích. Pro PSO_I a při využití k-opt 10 částic na 10 000 iterací. Cílem bylo získat 100 000 zavolání fitness funkce. Podle poměrů by měl být rozdíl mezi částice a iteracemi větší, ale nechtěl jsem jít pod 10 částic. Zde by to způsobilo problém pro ABC algoritmy, kde se musí rozdělit role mezi včelami a s opravdu malým počtem částic je to problém. Dále algoritmy s více hejny nebudou už při 10 částicích tak efektivní při vytvoření nového hejna. Pokud tedy bude počet částic 10, pak nastavím počet sběratelek v ABC algoritmech na 3, jinak jich bude 10. Pro 3 sběratelky budou mít 1 skautku, jinak jich budou mít 5. Algoritmy mají nastavení podle nejlepších získaných hodnot z výše uvedených experimentů. Experimenty proběhnou na dvou instancích, a to berlin52 a pr136. Počet běhů v experimentu bude oproti předchozím experimentům větší, tedy 20 běhů. Nakonec budou výsledky porovnány se studií [28], která srovnávala několik algoritmů a jejich výsledky.

	S 3-opt			Bez K-opt		
	Min	Průměr	Max	Min	Průměr	Max
PSO_S	7747.5	8217.0	8677.1	12762.6	14576.2	16029.9
PSO_I	7544.4	8103.6	8989.3	8653.9	9807.9	11215.4
PSO_MSS	7544.4	8048.4	8624.3	8426.2	9530.5	11008.4
ABC_S	7890.2	8293.2	8583.6	9501.6	10070.3	10747.0
ABC_I	7544.4	7745.4	8041.4	8480.1	9206.	10057.3
ABC_MSS	7544.4	7698.2	8063.3	8146.6	9308.1	10678.8

Tabulka 6.23: Výsledky pro berlin52. V horním řádku je, zda byl využit k-opt a jaký, pod ním minimum, průměr, maximum. Na levé straně tabulky jsou algoritmy.

Z tabulky 6.23 je poznat, že algoritmy s více hejny nachází nejlepší výsledky. Pro PSO_I existence více hejn zlepšuje dosažené výsledky obecně. Zdá se, že ABC_MSS nachází lepší výsledek pro 20 běhů, ale má také větší rozptyl výsledných hodnot. Studie [28] a data z univerzity v Heidelbergu[1] udávají, že nejlepší cesta pro berlin52 má hodnotu 7 542, ke které se navržené algoritmy s 3-opt přiblížily. PSO_I s 3-opt nedávalo špatné výsledky a oproti PSO_S a ABC_S se zde dosáhlo dobrých výsledků. Pro tuto velikost instance se algoritmy s využitím 3-opt blíží dobrému výsledku. Bez něj se jim nedaří přesáhnou hranici 8 000.

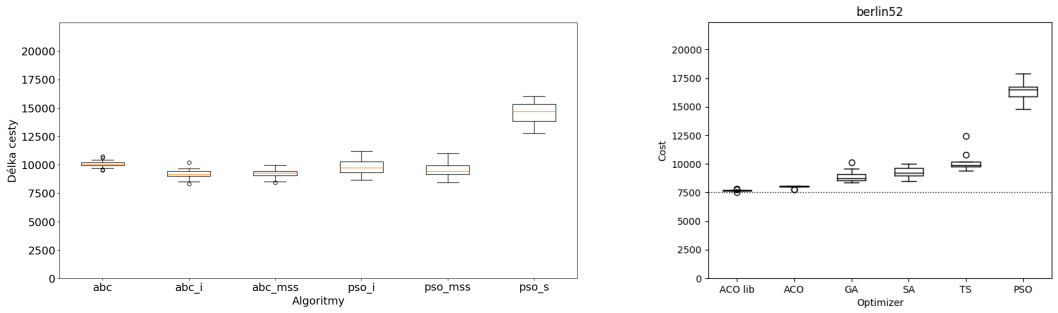
V tabulce 6.24 opět dávají algoritmy s více hejny lepší výsledky v obou případech bez 3-optu. Pro 3-opt PSO_MSS opět dosáhlo lepších výsledů. V případě ABC_MSS je trend při využití 3-opt stejný jako u předchozího experimentu. Bez něj se ale naopak zmenšil rozptyl a nenašlo nejlepší hodnotu. Nejlepší hodnotu v tomto případě našlo ABC_I. PSO_S dává velmi špatné výsledky bez 3-opt, ale s tím se dalo počítat. Problémy principu tohoto algoritmu jsem shrnul v předchozí sekci. ABC_S naopak bylo nejhorší při použití 3-opt v porovnání s ostatními algoritmy.

Nejlepší nalezená hodnota podle studie a univerzity v Heidelbergu [28] a byla 96772, od které se zde už výsledky vzdálily a dá se předpokládat, že tento ústup se bude s většími instancemi zvyšovat.

[1] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/STSP.html

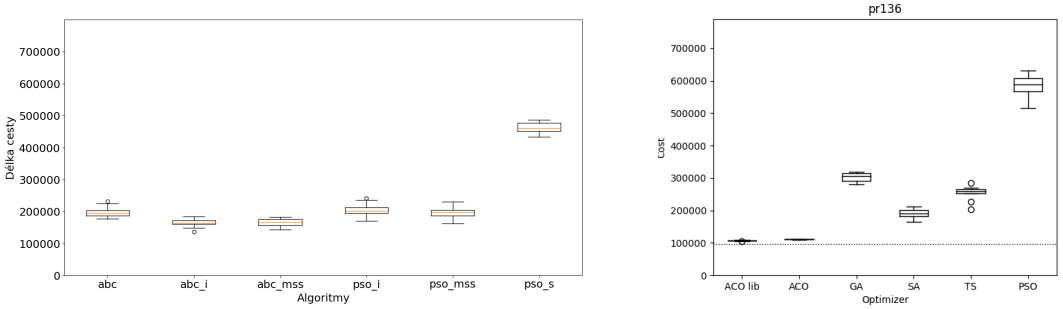
	S 3-opt			Bez K-opt		
	Min	Průměr	Max	Min	Průměr	Max
PSO_S	117083.7	122505.0	127139.9	432992.7	461859.2	486345.6
PSO_I	107737.5	111916.5	120929.7	170799.5	204541.1	241476.9
PSO_MSS	106441.0	114297.2	121918.8	162324.1	196449.8	230136.8
ABC_S	128984.6	139193.5	149531.6	177649.9	197909.6	231558.4
ABC_I	102470.6	106748.6	110812.3	136545.1	165061.5	184512.1
ABC_MSS	101397.8	107924.2	112097.0	143151.0	165014.3	182463.6

Tabulka 6.24: Výsledky pro pr136. V horním řádku je, zda byl využit 3-opt, pod ním minimum, průměr, maximum. Na levé straně tabulky jsou algoritmy.



Obrázek 6.16: Porovnání algoritmů pro berlin52 bez k-opt nalevo se studií napravo [28].

V porovnání 6.16, ACO představuje mravenčí algoritmy, GA genetické, SA simulované žíhání a TS tabu search. Z porovnání je poznat, že bez 3-optu si algoritmy stále nevedou dost dobře v porovnání s mravenčími algoritmy.



Obrázek 6.17: Porovnání algoritmů pro pr136 bez k-opt nalevo se studií napravo [28].

Ve porovnání 6.17, se rozdíl oproti mravenčím algoritmům trochu zhoršil. Samotné PSO_S si v porovnání se studií zde vede lépe díky využití více částic a méně iterací. Vylepšené algoritmy s více hejny si vedou podobně jak simulovaná žíhání pro tuto velikost instance, ale zdá se, že se víc oddalují od optimálního řešení při zvětšení velikosti instance.

6.10 Experiment porovnání algoritmů na velikých instancích TSP

Pro PSO_S a ABC algoritmy bez k-opt byl zvolen počet částic 640 na 3125 iteracích. Pro PSO_I a při využití k-opt 25 částic na 80 000 iterací. Cílem bylo získat 2 000 000 zavolání fitness funkce. Počet sběratelek a skautů v ABC algoritmech, který bylo nutné trochu upravit v předchozím experimentu, se nastaví podle nejlepších hodnot z předchozích experimentů na 10 a 5. Algoritmy mají nastavení ostatních parametrů podle nejlepších získaných hodnot z výše uvedených experimentů. Experimenty proběhnou na dvou instancích, a to dsj1000 a fnl4461. Počet běhů v experimentu bude opět 20. Protože výsledné cesty jsou moc dlouhé, tak je tabulka pro dsj1000 rozdělené na dvě.

	Min	Průměr	Max
PSO_S	60572846.0	63375487.8	68186494.0
PSO_I	63319134.0	67370861.0	71559101.0
PSO_MSS	65674765.0	69694353.9	75218619.0
ABC_S	74007001.0	78071988.8	81755556.0
ABC_I	27569920.0	29711804.0	32440306.0
ABC_MSS	30896336.0	33230405.3	36414025.0

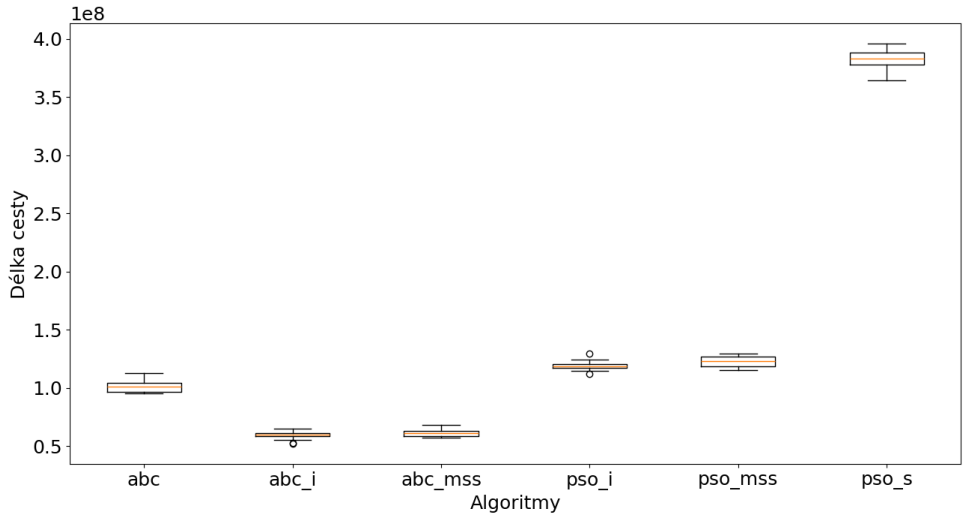
Tabulka 6.25: Výsledky pro dsj1000 s využitím 2-opt. V horním řádku je minimum, průměr, maximum. Na levé straně tabulky jsou algoritmy.

Z tabulky 6.25 je poznat, že algoritmy s více hejny už nenachází nejlepší výsledky. Důvodem je, že parametry, které jsem během experimentů získal jsou vhodnější pro větší instance. Menší randomizace parametrů nepomůže. Je to ale stále trochu překvapení zejména pro PSO_MSS, kde jsem očekával lepší výsledky. Důvodem je malá šance na vložení u které jsem čekal, že umožní hejnu se přesunout a získat lepší cestu. Zdá se, že pro větší instance zhoršuje nalezené řešení, protože pak hejno dlouho dolétává za částicí nebo ona k němu. Mezi tím se nenalézá lepší řešení v okolí. U ABC_MSS jsem zhoršení pro větší instance očekával. Nejlepší nalezená délka cesty z dat univerzity v Heidelbergu je 18660188. Nejblíže tomu je ABC_I a výsledek už při takovéto velikosti je poměrně špatný. Překvapivé je PSO_S, které v tomto jediném experimentu překonalo s výsledky PSO_I při využití 2-opt. Pro jistotu jsem test spustil dvakrát, abych se ujistil, že mám dobře nastavené parametry. Nejedná se o trend, ale PSO_S s 2-optem dává lepší výsledky specificky pro tuto instanci.

	Min	Průměr	Max
PSO_S	364334558.0	382396256.4	395980542.0
PSO_I	111679577.0	118886789.8	129601574.0
PSO_MSS	115134370.0	122448887.0	129471411.0
ABC_S	95387537.0	100984289.4	112525294.0
ABC_I	51645332.0	59148490.1	64751911.0
ABC_MSS	57119354.0	61150098.6	68279733.0

Tabulka 6.26: Výsledky pro dsj1000 bez k-opt. V horním řádku je minimum, průměr, maximum. Na levé straně tabulky jsou algoritmy.

V tabulce 6.26 a boxplot 6.18 je možné pozorovat stejný trend s horšími výsledky. I bez využití k-opt si více hejn nevede dobře. Nejblíže nejlepší známé cestě je ABC_I a to



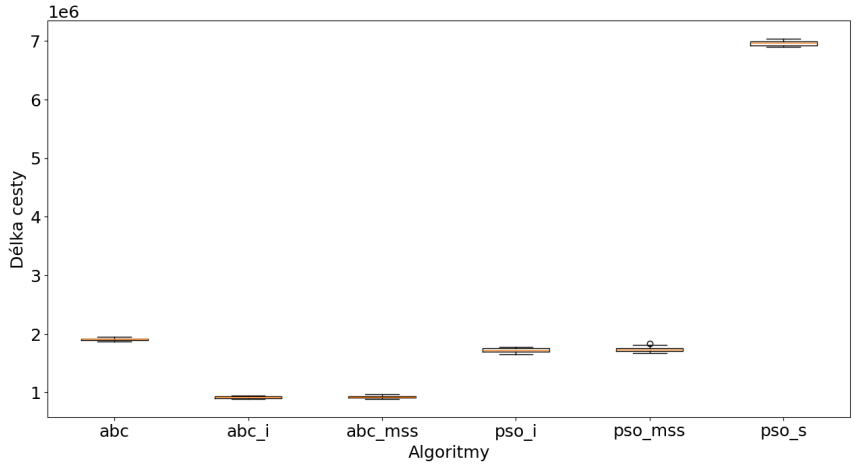
Obrázek 6.18: Porovnání algoritmů pro dsj1000 bez k-opt, pro data z tabulky 6.26

s dvakrát lepším výsledkem jak PSO_I. Ostatní algoritmy jsou na tom podstatně hůře. Navržené algoritmy si vedou lépe než základní. To je nejvíce znát na PSO_S, ale i takovéto zlepšení ani zdaleka nestačí pro dobré výsledky u velikých instancí. PSO_S už nepřekvapilo a dává horší výsledky i pro tuto instanci.

	S 2-opt			Bez K-opt		
	Min	Průměr	Max	Min	Průměr	Max
PSO_S	2239526.7	2265017.7	2302884.8	6897551.6	6964285.1	7039193.8
PSO_I	1624320.0	1659439.2	1752994.0	1651557.2	1725176.3	1775474.3
PSO_MSS	1583615.0	1667318.3	1751571.7	1678536.5	1738327.6	1831647.0
ABC_S	1811571.4	1854945.1	1902066.5	1865294.8	1907304.9	1955611.4
ABC_I	585664.0	606422.6	631797.5	884419.3	918637.8	948387.2
ABC_MSS	593710.8	610871.8	636179.2	894847.3	925088.9	968943.3

Tabulka 6.27: Výsledky pro fnl4461. V horním řádku je, zda byl využit 2-opt, pod ním minimum, průměr, maximum. Na levé straně tabulky jsou algoritmy.

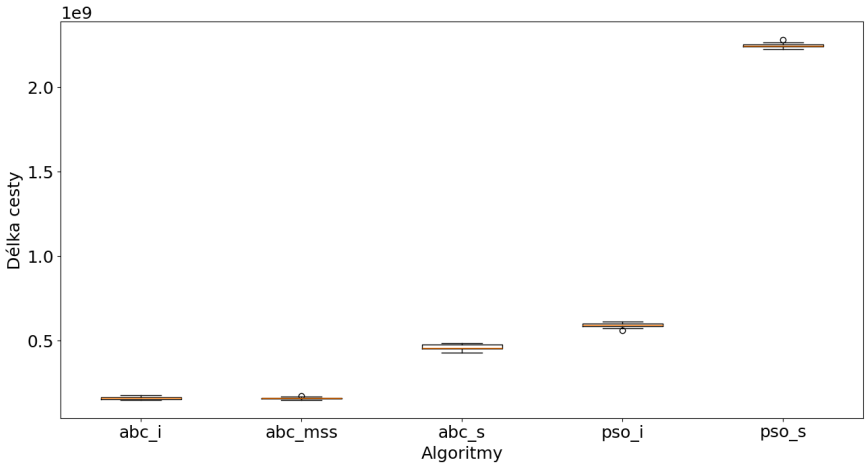
V tabulce 6.27 a boxplotu 6.19 překvapivě má PSO_MSS nejlepší výsledek z PSO algoritmů při využití 2-opt. To je ale jediný případ, kdy algoritmus s více hejny dosáhl lepšího výsledku. Výsledky se řídí očekávaným trendem, kdy kvalita výsledků PSO se vzdaluje více od jeho PSO_I. Stejný trend platí pro ABC_S, akorát se nevzdaluje tolik. To neplatí pro 2-opt, což může být způsobeno, že se z časových důvodů místo 3-optu využívá 2-opt. 3-opt by byl příliš časově náročný. Nejlepší nalezená délka cesty z dat univerzity v Heidelbergu je 182566. Nejlepší délka získaná z experimentů se více vzdálila od nejlepší známé cesty. Nejlepší délka v experimentu byla nalezena ABC_I.



Obrázek 6.19: Porovnání algoritmů pro fnl4461 bez k-opt, pro data z tabulky 6.27

6.11 Experiment porovnání algoritmů na rozsáhlých instancích TSP

Pro PSO_S a ABC algoritmy bez k-opt byl zvolen počet částic 1920 na 9375 iteracích. Pro PSO_I a při využití k-opt 75 částic na 240 000 iterací. Cílem bylo získat 18 000 000 zavolání fitness funkce. Parametry algoritmů jsou stejné jako v předchozím experimentu. Experimenty proběhnou na dvou instancích, a to pla7397 a usa13509.



Obrázek 6.20: Porovnání algoritmů pro pla7397 bez k-opt, pro data z tabulky 6.28

Protože výsledné cesty jsou příliš veliké, tak byla data rozdělena do dvou tabulek 6.28 a 6.29. Z časových limitů obsahují méně dat. Zde se už jednalo o rozsáhlé instance, proto už PSO algoritmy nevyužívaly k-opt, jinak by běžely příliš dlouho. PSO_S obsahuje jen 10 běhů, protože i těch 10 běhů počítalo zhruba 60 hodin. Zatímco PSO_I s 20 běhy netrvalo ani 10 hodin. Operace vložení se zde ukázala jako příliš náročná pro PSO algoritmy, proto PSO_MSS běželo moc dlouho, proto zde nejsou jeho výsledky.

	Min	Průměr	Max
PSO_S	2224589104.0	2246163614.2	2280997076.0
PSO_I	559368870.0	591414061.2	614521057.0
ABC_S	429468980.0	461243350.6	484609158.0
ABC_I	146852847.0	157788860.6	176516896.0
ABC_MSS	149837159.0	158671940.1	172730466.0

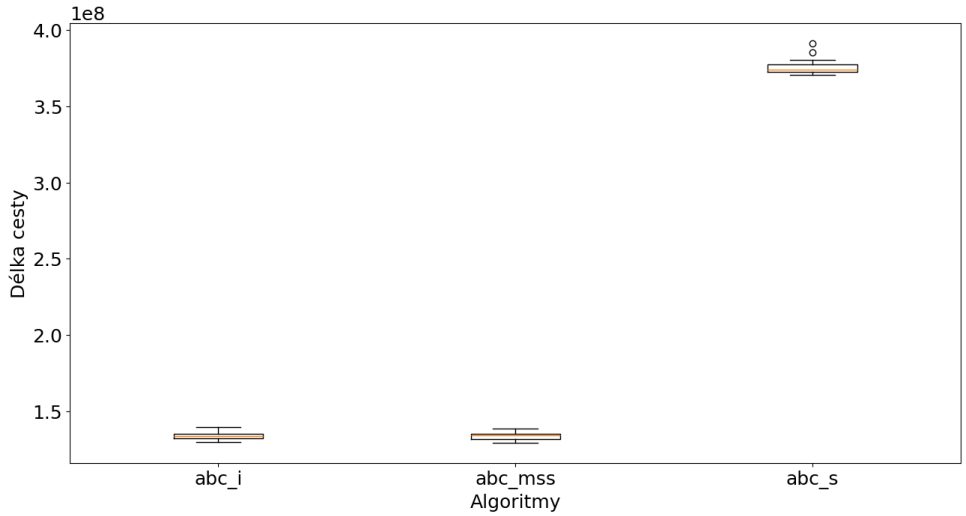
Tabulka 6.28: Výsledky pro pla7397 bez k-opt. V horním řádku je minimum, průměr, maximum. Na levé straně tabulky jsou algoritmy.

Tabulka 6.28 a boxplot 6.20 ukazuje zajímavý vývoj, kde PSO_I nyní podává horší výsledky jak ABC_S. ABC_MSS má sice horší průměr výsledků, ale rozptyl se zmenšil. Navíc se výsledky přiblížily ABC_I. Nejlepší nalezená délka cesty z dat univerzity v Heidelbergu je 23260728. Nejblíže tomu je ABC_I.

	Min	Průměr	Max
ABC_S	333711403.0	365882115.0	403955009.0
ABC_I	71516012.0	80779078.9	98199631.0
ABC_MSS	71226927.0	78394669.7	86126982.0

Tabulka 6.29: Výsledky pro pla7397 s využitím 2-opt. V horním řádku je minimum, průměr, maximum. Na levé straně tabulky jsou algoritmy.

Tabulka 6.29 ukazuje zajímavou změnu, kde ABC_MSS nyní podává lepší výsledky jak ABC_I. To může být dáno randomizací parametrů při vytváření nového hejna. Experimenty s parametry byly provedeny na instancích s menší velikostí a bez k-opt, proto nemusí být optimální pro rozsáhlé instance. Podobně to bylo u menších instancí, kde algoritmy s hejny byly schopné překonat algoritmy bez nich. ABC_MSS je pak schopné vytvořit hejno s optimálními parametry pro tyto velikosti.



Obrázek 6.21: Porovnání algoritmů pro usa13509 bez k-opt, pro data z tabulky 6.30

	Min	Průměr	Max
ABC_S	370706134.4	375864134.7	391347239.8
ABC_I	129860812.3	133811578.7	139551143.9
ABC_MSS	129034770.2	133598111.2	138661343.8

Tabulka 6.30: Výsledky pro usa13509 bez k-opt. V horním řádku je minimum, průměr, maximum. Na levé straně tabulky jsou algoritmy.

V tabulce 6.30 a boxplotu 6.21 jsou pouze ABC algoritmy, protože instance byla příliš veliká a zabralo by moc času otestovat PSO algoritmy. To samé platí pro 2-opt. Je vidět stejný trend jako u pla7397, kde více hejn podává kvalitnější výsledky. Při této velikosti není třeba využít 2-opt, pro získání lepších hodnot s ABC_MSS oproti ABC_I. Nejlepší nalezená délka cesty z dat univerzity v Heidelbergu je 19982859. Nejlepší délka v experimentu byla nalezena ABC_MSS.

6.12 Zhodnocení výsledků

V provedených experimentech se získaly ideální parametry algoritmů, poté se porovnaly časové náročnosti. Dále byly vyhodnoceny výsledky, kterých je schopen dosáhnout každý z algoritmů při využití k-opt nebo bez něj. Na závěr budou porovnány výsledky se známými nejlepšími cestami. Cílem je získat procentuální rozdíl průměrných hodnot dvaceti běhů od známého nejlepšího řešení. Pokud z časových důvodů nemám výsledky, pak je v tabulce místo hodnoty pomlčka. Výsledky jsou převzaty z předchozích experimentů a také jsou přidány výsledky, které nebyly uvedeny v předchozích experimentech.

Na výsledky v 6.31 se dá podívat dvěma způsoby. Pokud se hodnotí výsledky PSO_I pouze v kontextu algoritmů PSO, pak jsou výsledky velmi dobré. Například dosahuje lepších výsledků než ve studii [8] i pro menší instance a přitom redukuje časovou složitost. Pokud se ale vezme na vědomí existence mravenčích algoritmů, tak v porovnání s nimi jsou výsledky nedostatečné. To samé platí pro ABC_I akorát v menší míře, navíc pro malé instance dosahuje horších výsledků jak algoritmus ve studii [18]. Ten má ale větší časovou složitost. Výsledky pro menší instance nejsou špatné pro oba algoritmy a při optimalizaci operací pro tyto velikosti instancí by byly schopné podávat velmi dobré výsledky. V současném stavu podávají dobré výsledky pro instance do velikosti 50 měst, poté ale začne kvalita výsledků rychle klesat. Když jsem začínal s diplomovou prací, tak mě překvapil nedostatek studií zabývajícími se využitím PSO a ABC pro větší instance TSP. Například studie PSO [8] řeší pouze velikost instance do 50 měst a studie ABC [18] řeší pouze velikost do 200 měst. Pro tyto počty měst dávají dobré výsledky, ale nejsou aplikovatelné na větší instance, protože obsahují operace, které mají velikou časovou složitost. To nevadí u menších instancí, ale u větších je to problém. Navržené úpravy podstatně zlepšují slabiny těchto algoritmů a umožňují je využít pro veliké instance. Výsledky ale nejsou na úrovni vhodnějších mravenčích algoritmů. Problémy nejsou jen výsledky, ale pro PSO algoritmy také časová náročnost. To zejména platí pro PSO_S, které i bez 2-opt počítá u 20 běhů pro fnl4461 zhruba 11 hodin na serveru a s využitím 2-opt dokonce okolo 5 dnů. PSO_I zkracuje dobu trvání dost a to na 1 hodinu a s 2-opt na 40 hodin. Důvod proč PSO_I probíhá déle jak v sekci s naměřenými hodnotami je ten, že už využívá k-opt, protože stagnoval po určitou dobu iterací. Pak hejno začne provádět hodně prohození, aby se přesunulo na danou polohu

kvůli k-opt. Zde je ale potřeba vzít na vědomí, že na serveru může běžet více procesů, proto je potřeba brát tyto časy s rezervou. I takto veliké zlepšení ale nestačí u velikých instancí.

Dále jsem plánoval vykreslit výsledné cesty v kapitole s ArtTSP. Velikosti vykreslených instancí měli být okolo 10 000 měst. Tuto kapitolu jsem nakonec nepřidal, protože výsledky nebyly dostatečně dobré.

	52	136	1000	4461	7397	13509
PSO_S	93.3	377.3	1949.3	3714.7	9556.5	-
PSO_I	30.0	111.4	537.1	845.0	2442.5	-
PSO_MSS	26.4	103.0	556.2	852.2	-	-
ABC_S	33.5	104.5	441.2	944.7	1882.9	1780.9
ABC_I	22.1	70.6	217.0	403.2	578.3	569.6
ABC_MSS	23.4	70.5	227.7	406.7	582.1	568.6
PSO_S_2opt	11.3	37.6	239.6	1140.7	-	-
PSO_I_2opt	10.6	16.4	261.0	809.0	-	-
PSO_MSS_2opt	8.4	17.2	273.5	813.3	-	-
ABC_S_2opt	17.1	61.9	318.4	916.0	1473.0	-
ABC_I_2opt	5.3	14.5	59.2	232.2	247.3	-
ABC_MSS_2opt	4.4	21.3	78.1	234.6	237.0	-
PSO_S_3opt	8.9	26.6	-	-	-	-
PSO_I_3opt	7.4	15.6	-	-	-	-
PSO_MSS_3opt	6.7	18.1	-	-	-	-
ABC_S_3opt	10.0	43.8	232.7	779.3	-	-
ABC_I_3opt	2.7	10.3	38.1	139.8	-	-
ABC_MSS_3opt	2.1	11.5	47.2	138.2	-	-

Tabulka 6.31: Porovnání výsledků algoritmů s nejlepšími známými délkami cest. Nahoře je délka řešené instance. Nalevo jsou algoritmy, kde má každý u sebe uvedeno, zda využívá k-opt a jaký. Dále každé číslo je procentuální rozdíl průměru dvaceti běhů od známého nejlepšího řešení. Vzorec pro výpočet je (((Průměrné řešení z finálních experimentů)/(známého nejlepšího řešení))-1)*100.

V tabulce 6.31 je taky zajímavé, že ABC algoritmy po dosažení určitého rozdílu se přestanou zhoršovat. Bohužel pro 2-opt i bez něj se jedná o vysoká čísla.

6.13 Možná vylepšení

I přes vylepšení jsem podcenil časovou náročnost operací PSO. Jedno z možných vylepšení PSO by bylo provádět prohození oproti gBest/pBest v jednom běhu, pokud je rozdílné oproti jednomu, tak se s určitou pravděpodobností vymění. Pro zjištění pozice výměny by měla gBest prioritu před pBest. Je ale potřeba vzít na vědomí, že tohle vylepšení sice řeší jeden ze symptomů, ale úplně nevyřeší problém s principem, na kterém PSO založené. Tedy stále by se musela procházet celá cesta v částici a porovnávat jednotlivá města. Další z možných vylepšení by bylo zakomponování operace vložení do PSO. Během experimentů se tato akce ukázala, jako poměrně efektivní. V současném stavu ale algoritmus není schopen tuto operaci využít bez veliké časové náročnosti a zhoršení výsledků. Zhorší výsledek, protože částice pak dlouho prohazují města, aby přiletěla k nové nejlepší pozici. Jedním z řešení by bylo tuto operaci upravit tak, že po jejím provedení by se ohodnotila fitness částice. Pokud

by byla horší jak gBest, pak se celá operace vrátila zpět, aby částice nemusela prohozeními docestovat zpět k hejnu. Pokud by překonala gBest, pak by se provedla pro všechny částice v hejnu tak, aby se celé hejno přesunulo k okolí pozice. Tím je myšleno, že by se město vložilo u všech částic na tuto pozici. Tím by hejno nemuselo dolítat zpět pomocí velikého množství prohození. Tato vylepšení by měla umožnit PSO se přiblížit ABC jak v rychlosti, tak v kvalitě výsledku. V případě výsledku je dokonce možné ho překonat. Nevýhodou je, že to není dostatečné zlepšení pro řešení velikých instancí, aby se kvalita výsledků dostala na úroveň mravenčích algoritmů. Stejné vylepšení jako pro operaci vložení je možné využít i pro k-opt.

V případě ABC mě napadá operace vynucení mutace, kde by včela s určitou pravděpodobností provedla mutaci, i když nezlepšila řešení. Princip by byl podobný simulovanému žíhání. Tím by se vyřešil problém, kde není možné u ABC na cestu aplikovat složitější mutace pro vylepšení, protože je veliká pravděpodobnost, že nezlepší délku cesty a nebudou aplikovány. Tím by se u ABC s většími instancemi mohli využít komplikovanější mutace. Opět to ale nezlepší kvalitu výsledku na tolik aby dosahovali lepších výsledků než mravenčí algoritmy. Na to by bylo potřeba zkusit navrhnout hybridní algoritmu využívající princip ABC a ACO, ale to je za hranice této práce.

Kapitola 7

Závěr

V této práci jsem definoval problém obchodního cestujícího a náročnost řešení. Dále jsem ilustroval přístupy k řešení tohoto problému. Nakonec jsem popsal princip řešení pomocí kolektivní výpočetní inteligence. Z této oblasti jsem se zaměřil na optimalizace, které jsou založené na chování hejna a optimalizace, které jsou založené na chování včelího roje. V další kapitole jsem popsal jejich diskretizaci a aplikaci na TSP. Dále jsem shrnul princip diverzity a její problémy pro diskrétní prostor. Nakonec jsem navrhl a otestoval vlastní modifikace algoritmů, které by měli řešit hlavní problémy na které jsem narazil na začátku vývoje. Mezi tyto modifikace patří zrychlení algoritmu PSO, aby byl použitelný pro větší instance. Při modifikaci algoritmů jsem si dával pozor, abych nepřidal operace s velkou časovou složitostí, které by komplikovali jejich provedení pro veliké instance. Na algoritmech byly provedeny experimenty, které sloužily pro zjištění konvergence a kvalitu výsledků, kterých algoritmy dosahují. Z experimentů jsem vyvodil ideální nastavení parametrů pro jednotlivé algoritmy a jaký to má vliv na výsledky. Tyto zjištění byli využity při dalších experimentech. Nakonec jsem experimentoval se slibnými algoritmy na větších instancích. Výsledky těchto experimentů byly porovnány mezi jednotlivými algoritmy. Nakonec jsem tyto výsledky zhodnotil v rámci nejlepších známých výsledků nad instancemi, se kterými jsem experimentoval.

Z experimentů lze vyvodit, že navržené algoritmy ABC_I a PSO_I vedou daleko lépe. Při srovnání s nelepším známým řešení větších instancí je ale poznat, že samotné výsledky jsou stále nedostatečné. Navrhl jsem na závěr několik možných vylepšení pro oba algoritmy, které by jim umožnili se těmto výsledkům přiblížit. Věřím, že tyto vylepšení by zlepšili současné výsledky, ale nemyslím si, že by to stačilo pro získání výsledků na úrovni optimalizací mravenčích kolonií pro veliké instance.

Literatura

[1] Arman, N. Graph Representation: Comparative Study and Performance Evaluation. *Information Technology Journal*, Duben 2005, sv. 4.

[2] Bin, W.; Qinke, P.; Jing, Z. a Xiao, C. A binary particle swarm optimization algorithm inspired by multi-level organizational learning behavior. *European Journal of Operational Research*, 2012, sv. 219, č. 2, s. 224–233. ISSN 0377-2217. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0377221712000240.

[3] Chang, W.-D. A modified particle swarm optimization with multiple subpopulations for multimodal function optimization problems. *Applied Soft Computing*, 2015, sv. 33, s. 170–182. ISSN 1568-4946. Dostupné z: https://www.sciencedirect.com/science/article/pii/S1568494615002161.

[4] Chen, Q.; Peng, Y.; Zhang, M. a Yin, Q. Application Analysis on PSO Algorithm in the Discrete Optimization Problems. *Journal of Physics: Conference Series*. IOP Publishing, nov 2021, sv. 2078, č. 1, s. 012018. Dostupné z: https://dx.doi.org/10.1088/1742-6596/2078/1/012018.

[5] Cheng, S. a Shi, Y. Diversity control in particle swarm optimization. In: *2011 IEEE Symposium on Swarm Intelligence*. 2011, s. 1–9.

[6] Clerc, M. Discrete Particle Swarm Optimization, illustrated by the Traveling Salesman Problem, Leden 2004, sv. 47.

[7] Davendra, D. *Traveling Salesman Problem*. Rijeka: IntechOpen, Dec 2010. Dostupné z: https://doi.org/10.5772/547.

[8] Emambocus, B. A. S.; Jasser, M. B.; Hamzah, M.; Mustapha, A. a Amphawan, A. An Enhanced Swap Sequence-Based Particle Swarm Optimization Algorithm to Solve TSP. *IEEE Access*, 2021, sv. 9, s. 164820–164836.

[9] García Villoria, A. a Pastor, R. Introducing dynamic diversity into a discrete particle swarm optimization. *Computers & Operations Research*, 2009, sv. 36, č. 3, s. 951–966. ISSN 0305-0548. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0305054807002596.

[10] Gutin, G.; Yeo, A. a Zverovich, A. Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP. *Discrete Applied Mathematics*, 2002, sv. 117, č. 1, s. 81–86. ISSN 0166-218X. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0166218X01001950.

[11] Hu, J.; Bhowmick, P.; Jang, I.; Arvin, F. a Lanzon, A. A Decentralized Cluster Formation Containment Framework for Multirobot Systems. *IEEE Transactions on Robotics*, 2021, sv. 37, č. 6, s. 1936–1955.

[12] Jana, N. D.; Sil, J. a Das, S. Particle Swarm Optimization with population adaptation. In: *2014 IEEE Congress on Evolutionary Computation (CEC)*. 2014, s. 573–578.

[13] Jünger, M.; Reinelt, G. a Rinaldi, G. Chapter 4 The traveling salesman problem. In: *Network Models*. Elsevier, 1995, sv. 7, s. 225–330. Handbooks in Operations Research and Management Science. ISSN 0927-0507.

[14] Kahng, A. B. a Reda, S. Match twice and stitch: a new TSP tour construction heuristic. *Operations Research Letters*, 2004, sv. 32, č. 6, s. 499–509. ISSN 0167-6377. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0167637704000471.

[15] Karaboga, D. et al. *An idea based on honey bee swarm for numerical optimization*. Technical report-tr06, Erciyes university, engineering faculty, computer ..., 2005.

[16] Kennedy, J. a Eberhart, R. Particle swarm optimization. In: *Proceedings of ICNN'95 - International Conference on Neural Networks*. 1995, sv. 4, s. 1942–1948 vol.4.

[17] Kernighan, B. W. a Lin, S. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 1970, sv. 49, č. 2, s. 291–307.

[18] Khan, I. a Maiti, M. K. A swap sequence based Artificial Bee Colony algorithm for Traveling Salesman Problem. *Swarm and Evolutionary Computation*, 2019, sv. 44, s. 428–438. ISSN 2210-6502. Dostupné z: https://www.sciencedirect.com/science/article/pii/S2210650216304588.

[19] Laporte, G. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 1992, sv. 59, č. 2, s. 231–247. ISSN 0377-2217. Dostupné z: https://www.sciencedirect.com/science/article/pii/037722179290138Y.

[20] Lewis, M. a Bekey, G. The Behavioral Self-organization Of Nanorobots Using Local Rules. In:. Srpen 1992, sv. 2, s. 1333–1338. ISBN 0-7803-0737-2.

[21] Luu, Q. T. *Traveling Salesman Problem: Exact Solutions vs. Heuristic vs. Approximation Algorithms* online. Červen 2023. Dostupné z: https://www.baeldung.com/cs/tsp-exact-solutions-vs-heuristic-vs-approximation-algorithms. [cit. 2023-11-16].

[22] Ma, Z.; Liu, L. a Sukhatme, G. S. An adaptive k-opt method for solving traveling salesman problem. In: *2016 IEEE 55th Conference on Decision and Control (CDC)*. 2016, s. 6537–6543.

[23] Matai, R.; Singh, S. a Mittal, M. Traveling Salesman Problem: an Overview of Applications, Formulations, and Solution Approaches. In:. Listopad 2010. ISBN 978-953-307-426-9.

[24] Moraglio, A. a Poli, R. Topological crossover for the permutation representation. In:. červen 2005, s. 332–338.

[25] Peake, J.; Amos, M.; Yiapanis, P. a Lloyd, H. Scaling techniques for parallel ant colony optimization on large problem instances. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. New York, NY, USA: Association for Computing Machinery, 2019, s. 47–54. GECCO '19. ISBN 9781450361118. Dostupné z: https://doi.org/10.1145/3321707.3321832.

[26] Pratap, D. A. *Explore the differences between Rust and C++: A comparison of features, syntax, and applications* online. Leden 2023. Dostupné z: https://medium.com/programming-concepts/explore-the-differences-between-rust-and-c-a-comparison-of-features-syntax-and-applications-edce2f93271c. [cit. 2023-12-27].

[27] Sawik, T. A note on the Miller-Tucker-Zemlin model for the asymmetric traveling salesman problem. *Bulletin of the Polish Academy of Sciences: Technical Sciences*, 2016, s. 517–520.

[28] Tosoni, D.; Galli, C.; Hanne, T. a Dornberger, R. Benchmarking Metaheuristic Optimization Algorithms on Travelling Salesman Problems. In:. New York, NY, USA: Association for Computing Machinery, 2022, s. 20–25. ICSLT '22. ISBN 9781450396660. Dostupné z: https://doi.org/10.1145/3545922.3545926.

[29] Williamson, D. P. a Shmoys, D. B. *The Design of Approximation Algorithms*. 1st. USA: Cambridge University Press, 2011. ISBN 0521195276.

[30] Wong, L.-P.; Low, M. Y. H. a Chong, C. S. An efficient Bee Colony Optimization algorithm for Traveling Salesman Problem using frequency-based pruning. In: *2009 7th IEEE International Conference on Industrial Informatics*. 2009, s. 775–782.

Příloha A

Instalační manuál

V této příloze je popis instalace programu. Prerekvizitou je řetěz nástrojů Rust, který je možné stáhnout na odkazu https://www.rust-lang.org/learn/get-started. Zahrnuje nástroje jako `rustc`, `cargo` a `rustup`.

Překlad

Všechny závislosti a knihovny třetích stran jsou spravovány pomocí `cargo`, takže překlad je poměrně přímočarý. Finální spustitelný soubor je staticky propojen se všemi jeho závislostmi.

Příkaz pro sestavení ladicí verze spustitelného souboru:

- `cargo build`

Příkaz pro sestavení verze k vydání spustitelného souboru:

- `cargo build -release`

Spuštění

Příkaz pro spuštění spustitelného souboru přímo:

- `cargo run [-release]`

Příloha B

Struktura obsahu přiloženého média

Přiložené médium obsahuje:

- `/`
  - `data` ..... Repozitář s tsp instancemi
  - `results` ..... Repozitář s výsledky testů
  - `src` ..... Repozitář s kódem
    - `algorithms` ..... Repozitář s jednotlivými algoritmy
    - `algorithms.rs` ..... Obsahuje funkci pro výběr a spuštění algoritmů
    - `graph.rs` ..... Obsahuje strukturu grafu
    - `local_search.rs` ..... Obsahuje k-opt implementaci
    - `main.rs`
    - `params.rs` ..... Soubor obsahující strukturu parametry
    - `permutation_utils.rs` ..... Obsahuje pomocné funkce
    - `result_writer.rs` ..... Obsahuje funkce pro výpis výsledků
    - `svg_reader_writer.rs` ..... Obsahuje funkce pro čtení a výpis svg
    - `tsp_reader.rs` ..... Obsahuje funkce pro čtení tsp souborů
    - `yaml_reader.rs` ..... Obsahuje funkce pro čtení yaml souboru s parametry
  - `target` ..... Repozitář s verzí sestavenou na edesign.fit.vutbr.cz
  - `cargo.lock`
  - `cargo.toml` ..... Knihovny a nastavení pro cargo
  - `params.yaml` ..... Soubor s upravitelnými parametry běhu a algoritmů
  - `plot.py` ..... Script pro vykreslení boxplotů a konvergenčních křivek
  - `readme.md`
  - `xfrane16.pdf` ..... Text diplomové práce