Vysoká škola ekonomická v Praze
Fakulta informatiky a statistiky


# 3D problém obchodního cestujícího – heuristiky a metaheuristiky

DIPLOMOVÁ PRÁCE

Studijní program: Ekonometrie a operační výzkum

Autor: Bc. Vojtěch Vávra

Vedoucí práce: prof. Ing. Josef Jablonský, CSc.

Praha, květen 2022

## Prohlášení

Prohlašuji, že jsem diplomovou práci *3D problém obchodního cestujícího – heuristiky a meta-heuristiky* vypracoval samostatně za použití v práci uvedených pramenů a literatury.

V Praze dne 2. května 2022 .....

Podpis studenta

## Poděkování

Rád bych poděkoval prof. Ing. Josefovi Jablonskému, CSc. za všechny rady a podněty, které mi pomohly při vzniku práce včetně zapůjčení knihy. Dále bych rád poděkoval rodině za trpělivost a podporu během studia.

## Abstrakt

Název práce: 3D problém obchodního cestujícího – heuristiky a metaheuristiky
Autor: Bc. Vojtěch Vávra
Katedra: Katedra ekonometrie
Vedoucí práce: prof. Ing. Josef Jablonský, CSc.

Tato práce se zabývá především problémem obchodního cestujícího ve 3D prostoru. V úvodní části je uvedeno definování problému a rešerše literatury. V teoretické části jsou definovány metody a algoritmy, kterými lze řešit formulovaný problém. Při řešení problému jsou uvedeny metody exaktní, heuristické a metaheuristické. V experimentální části byly implementovány vybrané heuristické a metaheuristické metody v prostředí jazyka R i s několika doplňky, jako je například vizualizace řešení. V rámci vyhodnocení jsou metody porovnávány mezi sebou a nejlepší doporučeny podle výpočetního času i hodnoty účelové funkce k řešení uvedeného problému.



### Klíčová slova

TSP, problém obchodního cestujícího, heuristiky, metaheuristiky, jazyk R, mezihvězdné cestování

### JEL klasifikace

C44, C61, C63

# Abstract

Title: 3D travelling salesman problem – heuristics and metaheuristics
Author: Bc. Vojtěch Vávra
Department: Department of Econometrics
Supervisor: prof. Ing. Josef Jablonský, CSc.

This thesis deals primarily with travelling salesmen problem in 3D. In introductory part we deliver formulation of the proposed problem and literature review. In the theoretical part we define exact, heuristic and metaheuristic methods for solving the proposed problem. In the experimental part were implemented selected heuristic and metaheuristic methods in the environment of language R with several addition for example visualization of the solution. In conclusion of the thesis all used models are compared and best recommended according to computational time and value of the objective function for solving mentioned problem.



## Keywords

TSP, travelling salesman problem, heuristics, metaheuristics, language R, star tour

## JEL classification

C44, C61, C63

# Obsah

**Úvod** **15**

**1 Formulace problému a rešerše literatury** **17**
1.1 Problém obchodního cestujícího – TSP . . . 17
1.2 Data – hvězdná mapa vesmíru . . . 17
1.3 Rešerše přístupů k řešení TSP . . . 18

**2 Metody pro řešení okružních úloh** **21**
2.1 Exaktní metody . . . 21
2.1.1 Matematický model lineárního programování TSP . . . 21
2.1.2 Metoda hrubé síly – Brute force . . . 22
2.1.3 Metoda větvení a mezí – Branch and bound . . . 23
2.2 Heuristiky . . . 23
2.2.1 Nejbližší soused . . . 24
2.2.2 Hladový algoritmus . . . 24
2.2.3 Metoda výhodnostních čísel – Clark, Wright . . . 25
2.2.4 Metoda vkládací . . . 26
2.2.5 Metoda konvexního obalu . . . 26
2.2.6 Metoda minimální kostry . . . 27
2.2.7 Christofidova metoda . . . 28
2.2.8 Metoda zatřiďovací . . . 28
2.2.9 Metoda výměn – Lin, Kerninghen . . . 29
2.3 Metaheuristiky . . . 30
2.3.1 Metoda lokálního hledání . . . 30
2.3.2 Tabu vyhledávání . . . 31
2.3.3 Metoda prahové akceptace . . . 31
2.3.4 Metoda simulovaného žíhání oceli . . . 32
2.3.5 Genetický algoritmus . . . 33
2.3.6 Optimalizace mravenčí kolonií . . . 37
2.3.7 Optimalizace včelí kolonií . . . 38
2.3.8 Algoritmus světlušek . . . 41
2.3.9 Netopýří algoritmus . . . 43
2.3.10 Optimalizace fyziologie stromů . . . 45
2.3.11 Další metaheuristiky . . . 47

**3 Numerické experimenty a jejich vyhodnocení** **49**
3.1 Prostředí R . . . 49
3.2 Vzdálenosti v R . . . 49
3.3 Grafické zobrazení řešení TSP ve 3D v R . . . 51
3.3.1 Balíček scatterplot3d . . . 51

3.3.2 Balíček rgl . . . 54
3.4 TSP funkce implementované v R . . . 55
3.5 Pomocné funkce pro řešení TSP v R . . . 56
3.6 Heuristiky v R . . . 57
3.6.1 Nejbližší soused v R . . . 57
3.6.2 Hladový algoritmus v R . . . 58
3.6.3 Výhodnostní čísla v R . . . 58
3.6.4 Metoda minimální kostry v R . . . 59
3.6.5 2-opt Lin Kerninghen v R . . . 60
3.7 Metaheuristiky v R . . . 61
3.7.1 Metoda prahové akceptace v R . . . 62
3.7.2 Metoda simulovaného žíhání v R . . . 64
3.7.3 Genetický algoritmus v R . . . 65
3.7.4 Optimalizace mravenčí kolonií v R . . . 70
3.7.5 Optimalizace včelí kolonií v R . . . 72
3.8 Ladění parametrů . . . 73
3.9 Výsledky . . . 75
3.9.1 Porovnání všech implementovaných metod . . . 75
3.9.2 Podrobnější porovnání heuristik . . . 79
3.9.3 Podrobnější porovnání metaheuristik . . . 79
3.10 Zhodnocení výpočetních experimentů . . . 80

**Závěr** **83**

**Seznam použité literatury** **85**

**A Ukázka skriptů** **93**
A.1 Kód hladového algoritmu . . . 93
A.2 Kód metody výhodnostních čísel . . . 94
A.3 Kód genetického algoritmu . . . 96
A.4 Kód křížení PMX . . . 97
A.5 Kód optimalizace včelí kolonie . . . 98

**B Tabulky výpočtů procedur** **101**

**C Obrázky porovnání procedur** **105**

# Seznam algoritmů


1 Metoda hrubé síly . . . 22
2 Metoda větvení a mezí . . . 23
3 Nejbližší soused . . . 24
4 Hladový algoritmus . . . 25
5 Vyčerpávající smyčka: pro zamezení tvoření podcyklů . . . 25
6 Metoda výhodnostních čísel . . . 26
7 Metoda vkládací . . . 26
8 Metoda konvexního obalu . . . 27
9 Metoda minimální kostry . . . 27
10 Převod Eulerova cyklu na Hamiltonův cyklus . . . 27
11 Christofidova metoda . . . 28
12 Metoda zatřiďovací . . . 29
13 Metoda výměn Lin–Kerninghen 2-opt . . . 29
14 Metoda lokálního hledání . . . 31
15 Tabu vyhledávání . . . 31
16 Metoda prahové akceptace . . . 32
17 Metoda SIAM . . . 33
18 Genetický algoritmus . . . 37
19 Optimalizace mravenčí kolonií . . . 39
20 Optimalizace včelí kolonií . . . 41
21 Algoritmus světlušek . . . 43
22 Netopýří algoritmus . . . 45
23 Optimalizace fyziologie stromů – TPO . . . 46

# Seznam použitých zkratek

**ABC** Artificial bee colony (umělá kolonie včel)
**ACO** Ant colony optimization (optimalizace mravenčí kolonií)
**AIS** Artificial immune systems (umělé imunitní systémy)
**B&B** Branch and bound (metoda větví a mezí)
**BA** Bat algorithm (netopýří algoritmus)
**BB** Best Bound (nejlepší vazba)
**BCO** Bee colony optimization (optimalizace včelí kolonií)
**BFA** Bacterial foraging algorithm (algoritmus bakterií vyhledávajících potravu)
**CP** Cutting planes (metoda sečných nadrovin)
**CS** Cuckoo search (cuckoovo prohledávání)
**DE** Differential evolution (diferenciální vývoj)
**EM** Electromagnetism-like method (metoda elektromagnetismu)
**FA, FFA** Firefly algorithm (algoritmus světlušek)
**FOA** Fruit fly optimization algorithm (optimalizace mušky octomilky)
**GA** Genetic algorithm (genetický algoritmus)
**GP** Genetic programming (genetické programování)
**HJ** Hooke–Jeeves
**HS** Harmony search (vyhledávání harmonie)
**IWD** Intelligent water drops (inteligentní kapky vody)
**LK, LKH** Lin-Kernighan heuristika (metoda výměn)
**LP** Linear programming (lineární programování)
**MIP** Mixed integer programming (smíšené celočíselné programování)
**MST** Minimum spanning tree (minimální kostra grafu)
**NP** nedeterministicky polynomiální
**OX1** Ordered crossover (pořadové křížení 1)
**P** polynomiální
**PMX** Partially mapped crossover (částečně zmapované křížení)
**PSO** Particle swarm optimization (optimalizace roje částic)
**RK** Random key (náhodný klíč)
**SA, SIAM** Simulated annealing (simulace žíhání oceli)
**SEC** Subtour elimination constraints (eliminační omezení podcyklů)
**TA** Threshold acceptance (metoda prahové akceptace)
**TPO** Tree physiology optimization (optimalizace fyziologie stromů)
**TSP** Travelling salesman problem
(problém obchodního cestujícího, okružní dopravní problém)
**WOA** Whale optimization algorithm (algoritmus optimalizace velryb)

# Úvod

V posledních letech se opět zvedl zájem o zkoumání vesmíru a pomalu dochází k pokrokům v objevování. Je již otázkou času, než nalezneme efektivní způsob, jak cestovat mezi planetami nebo dokonce hvězdami. V této práci se podíváme na problém obecně známý, ovšem k řešení poměrně problematický. Jedná se o problém obchodního cestujícího (TSP). Nebudeme však cestovat mezi městy, jak je tomu běžně zvykem, ale mezi hvězdami. Ty jsou zpravidla středem jednotlivých hvězdných soustav, jako je například Slunce středem naší sluneční soustavy. Existují hvězdné soustavy, které mohou obsahovat i více než jednu hvězdu.

Při řešení těchto problémů se vyhneme exaktním metodám, které pro úplnost práce zmíníme v teoretické části. Důvod je ten, že výpočetní čas těchto metod je s rostoucím se množstvím hvězd mnohonásobně náročnější, neboť TSP je NP-těžká úloha, kterou nelze vyřešit do optimality rychlým přístupem. V experimentální části se zaměříme na metody výpočetně jednodušší než exaktní metody, jako jsou heuristiky a metaheuristiky. V teoretické části se snažíme vytvořit kompletní seznam těchto přístupů a popsat jejich aplikaci podrobněji. Tyto metody lze řešit v rozumném čase a některé z nich lze i kombinovat. Příkladem těchto kombinací je genetický algoritmus v kombinaci s libovolnou metodou, která najde přípustné řešení.

Cílem práce je ověřit kvality a nedostatky jednotlivých heuristik a metaheuristik na datech trojrozměrně definovaných bodů (hvězd). To budeme zjišťovat pomocí implementace v prostředí jazyka R. Ten je pověstný svým pomalejším zpracováváním cyklů, takže jednotlivé přístupy budou časově penalizovány o trochu více za svoji složitost. Otázkou tedy je, zda bude viditelný rozdíl ve výpočetním čase oproti implementovaným metodám v R, ale napsaných na základě jiného jazyka, např. jazyka C. Představíme problémy, s kterými jsme se potýkali při implementaci metod, a dodáme návod k vytvoření skriptů pro řešení a vizualizaci problému výše uvedeného, který se v odborné literatuře označuje jako „Star tour“.

Práce se zaměřuje více na heuristiky, které dokáží nalézt přípustné dobré řešení, a metaheuristiky, které jsou inspirovány přírodou. Zároveň obsahuje kombinaci těchto metod a porovnání výsledků mezi nimi.

Jelikož na VŠE na Katedře ekonometrie dominuje používání jazyka R, vyvstává otázka, zda by bylo možné jednotlivé heuristiky a metaheuristiky vyučovat v prostředí tohoto jazyka, nebo by bylo vhodnější použít jazyk jiný. Způsob jejich aplikace jsme totiž ve výuce postrádali a myslíme si, že by bylo vhodné studenty blíže seznámit s některými přístupy, které zde uvádíme, než jen pomocí tužky a papíru.

# 1. Formulace problému a rešerše literatury

V nadcházejících letech nastane situace, kdy lidstvo bude cestovat mezi planetami. Jelikož je trendem, že matematika je vždy napřed před realizací, bylo by vhodné se podívat na hledání nejkratší cesty mezi různými planetami. Poněvadž však vzdálenosti planet jsou poměrně zanedbatelné vzhledem k vzdálenostem mezihvězdného prostoru, lze si tento problém zjednodušit a hledat nejkratší vzdálenosti mezi hvězdami. Tím se nám může mnohonásobně zjednodušit hledání řešení při zanedbatelné ceně nepřesnosti řešení.

## 1.1 Problém obchodního cestujícího – TSP

Vyhledáváme takové pořadí návštěv jednotlivých hvězd, při kterém urazíme nejkratší vzdálenost tak, abychom každou hvězdu navštívili právě jednou a vrátili se do startovní pozice. Hledáme tedy minimální Hamiltonův cyklus mezi hvězdami a řešíme problém obchodního cestujícího, též známý jako okružní dopravní problém (TSP, travelling salesman problem).[1] Pokud bychom se nechtěli vracet do startovního bodu, pak bychom tento problém pojmenovali jako hledání minimální Hamiltonovy cesty.

Dle teorie grafů (Pelikán, 2001) je TSP model definován na grafu $G = (V,E)$, ve kterém uzly (nodes, případně vrcholy, $V$, vertex) představují místa, která mají být navštívena, a hrany ($E$, edges) znázorňují komunikační síť mezi všemi uzly.

Pro TSP hledáme Hamiltonův cyklus na grafu $G$ tak, aby přesun po hranách $E$, které jsou cenově $c_{ij}$ ohodnoceny (vzdálenost, přepravní náklady, doba přejezdu...), byl minimální.

Hamiltonův cyklus lze definovat jako uzavřenou cestu na grafu, ta obsahuje všechny uzly grafu právě jednou.

Pro řešení TSP úlohy předpokládejme místa $V = \{1, 2, \ldots, n\}$ k navštívení a cenově ohodnocené hrany $\forall i \neq j : E = \{(i,j)\}$, kde $i,j \in V$. Proměnná $n$ označuje celkové množství míst, která mají být navštívena, včetně startovního uzlu.

## 1.2 Data – hvězdná mapa vesmíru

Hlavním úkolem mise Gaia Evropské kosmické agentury (ESA, European Space Agency) je udělat co nejrozsáhlejší a nejpřesnější 3D model naší galaxie a celkově vesmíru. Dělá to pomocí

[1] V literatuře se běžně používá jak travelling, tak traveling.

prohledávání bezprecedentního 1 % 100 miliard hvězd (ESA, 2019). Již vydala 3 datasety DR1, DR2 a DR3, které jsou k nalezení na internetových stránkách agentury (ESA, 2022).

Dataset DR2, ze kterého pochází data v této práci, publikován viz (Bailer-Jones et al., 2018), obsahuje na 1,33 miliard hvězd. Dalším zdrojem dat je (Nash, 2022), který byl základním kamenem pro vytváření menších datasetů.

Vědecký tým ve složení David Applegate, Robert Bixby, Vašek Chvátal, Wiliam Cook, Daniel Espinoza, Marcos Goycoolea a Keld Helsgaun se pustil do řešení „Star TSP“ pro výše uvedené zdroje dat. Data přizpůsobili k řešení TSP a výsledky jejich práce jsou veřejně k dispozici na internetových stránkách (Cook, 2022b). Lze zde i stáhnout datasety s pozicemi hvězd uložené ve formátu o třech proměnných $X$,$Y$,$Z$, se kterými budeme pracovat v kapitole 3. Vzdálenosti se počítají v jednotkách parsek (pc). Jeden pc je definován jako vzdálenost, z níž má jedna astronomická jednotka (au) úhlový rozměr jedné vteřiny, tedy:

$$1\,\mathrm{pc} = \frac{1\,\mathrm{au}}{\tan 1''} \approx 206\,265\,\mathrm{au} \approx 3{,}262\,\mathrm{ly} \approx 3{,}086 \cdot 10^{16}\,\mathrm{m}$$

Naše Slunce je pro naše data středem $\{X = 0, Y = 0, Z = 0\}$ z toho důvodu, že z naší planety Země pozorujeme všechna ostatní tělesa ve vesmíru. Nezapomeňme, že $X$,$Y$,$Z \in \mathbb{R}$.

## 1.3 Rešerše přístupů k řešení TSP

Práce volně přebírá myšlenky z knihy (Applegate et al., 2006). Autoři zde rozebírají program CONCORDE, který je navržen k řešení TSP úloh a obsahuje celou řadu způsobů, jak vyřešit TSP. Program je napsán v jazyce ANSI C. Lze jej stáhnout ze stránky (Cook, 2022a) nebo vyzkoušet jeho internetovou verzi viz (Mittelmann, 2022). Zprovoznit se nám však program nepodařilo z důvodu nekompatibility.

Řešení TSP lze hledat pomocí optimalizace matematického lineární programování již od publikace (Miller et al., 1960). Nemusíme však hledat nutně optimální řešení, ale řešení blízké optimálnímu pomocí heuristik a metaheuristik.

S každým dalším uzlem se výpočetní čas exaktních metod navyšuje obrovskou rychlostí, protože TSP patří do skupiny NP-obtížných úloh. V této práci se právě odchýlíme od exaktních metod k jednotlivým přístupům heuristik a metaheuristik, které sice nemusí poskytovat optimální řešení, ale blízké optimu. Rozdíl mezi nimi je jednoduchý. Heuristiky lze použít jen na daný problém a metaheuristiky lze použít obecně na libovolný problém.

Z oblasti heuristik uvádíme algoritmus nejbližšího souseda pro jeho jednoduchost a rychlost, která je ovšem kompenzována nepřesností, která může být zvýšena podle (Alemayehu et al., 2017), metodu konvexního obalu pro vysokou úspěšnost podle (Renaud et al., 1996), metodu minimální kostry pro historický význam (Held et al., 1970) a LKH heuristiku, na níž se specializuje Keld Heslgaun např. v práci (Tinós et al., 2018), kde odkazuje na vysokou efektivitu při hledání nejlepšího řešení. Dnes je tato heuristika považována za jednu z nejefektivnějších.

Část zabývající se metaheuristikami lze rozdělit do několika kategorií. V první kategorii inspirované přírodou se objevují prvky evolučních algoritmů. Příkladem toho může být genetický algoritmus (GA), např. i doplněný o preselekci viz (Razali et al., 2011).

Další skupinou algoritmů v kategorii inspirovaných přírodou jsou algoritmy založené na roji zvířat. Příkladem jsou optimalizace mravenčí kolonií (ACO) a optimalizace včelí kolonií (BCO). Porovnání těchto algoritmů nalezneme v práci (Jasser et al., 2014), kde závěrem bylo zjištěno, že ACO je při výpočtu rychlejší pro menší množství uzlů a pro více uzlů jsou výpočetní časy stejně dlouhé. Dalším příkladem optimalizace rojem je optimalizace pomocí roje částic (PSO) navrženém v práci (Shi et al., 2007). PSO dokáže řešit efektivně i velké problémy, ovšem je navrženo ke spojitým problémům.

Nakonec do kategorie inspirované přírodou spadá algoritmus založený na rostlinách, např. optimalizace fyziologií stromů (TPO). Ten byl vyvinut v práci (Halim et al., 2013) a následně upraven pro TSP v článku (Halim et al., 2019). Jedná se o poměrně nový přístup k řešení TSP.

V druhé kategorii nalezneme metaheuristiky inspirované fyzikálními zákony. Příkladem jsou vyhledávání harmonie (HS) z práce (Geem et al., 2001), kde autor uvádí nový přístup k optimalizaci v dnešní době běžného problému maximalizace zisku při minimálních nákladech ovšem v nelineárním světě. Dalším takovým algoritmem je simulace žíhání oceli (SIAM, SA), která je podle (Zhan et al., 2016) jednoduchá k aplikování pro výpočet TSP.

V práci (Haxhimusa et al., 2011) se zabývali rozdílem mezi 2D a 3D TSP. Součástí práce byl i sociální experiment, který nechal skupinu lidí řešit TSP úlohy jak ve 2D, tak ve 3D. Došli k závěru, že TSP ve 3D vede k 5% větší chybovosti oproti 2D TSP. Řešitelé velmi často používali minimální kostru k hledání řešení, na které pak autoři uvádí algoritmus pyramidového modelu minimální kostry grafu.

# 2. Metody pro řešení okružních úloh

Pro řešení TSP lze použít celou řadu přístupů a mnohé z nich lze i vzájemně kombinovat.

První skupinu, kterou uvádíme, označujeme jako exaktní metody. Jedná se o způsoby řešení, které vyžadují pro velké množství uzlů významné množství času k řešení, proto je v části numerických experimentů vynecháme.

Druhou skupinu uvádíme z oblasti heuristik. Jedná se o poměrně jednoduché algoritmy, které využívají logický přístup k řešení. Některé algoritmy již mohou být i náročnější na pochopení, ale většinou se drží jednoduché základní myšlenky.

Třetí uvedenou skupinou jsou metaheuristiky. Ty lze rozdělovat do dalších podskupin, ovšem jednou ze společných vlastností je inspirace v reálném světě, ať už se jedná o zvířecí, nebo třeba o fyzikální svět.

Do všech uvedených přístupů je vstupem matice vzdáleností $D$, která je dána prvky $d_{ij}$, které udávají vzdálenost mezi uzly $i$ a $j$. Vzdálenost $d_{ii}$ je zpravidla nulová a nepřípustná k použití, ale některé přístupy (např. zatřiďovací metoda) mohou vyžadovat jejich existenci. Některé přístupy zase naopak vyžadují jejich odstranění nebo vysokou hodnotu (např. nekonečno), díky které nebudou vybrány do optimální cesty.

## 2.1 Exaktní metody

Jedná se postupy, které prohledávají celou množinu přípustných řešení a vyhledávají to nejlepší, které existuje. Výpočetní, a tudíž i časová náročnost je vysoká a nepříliš vhodná pro instance s velkým množstvím uzlů. Řešení úloh může teoreticky trvat i několik let. Jejich přesnost k nalezení optima je však vysoká.

### 2.1.1 Matematický model lineárního programování TSP

Nalezení přesného optimálního řešení TSP lze pomocí lineárního programování (LP). V našem případě se jedná o úlohu smíšeného celočíselného programování (MIP). Pro model TSP si nejprve definujme následující symboly:

- $Z \ldots$ hodnota účelové funkce, uražená vzdálenost
- $n \ldots$ počet uzlů na grafu G,
- $i, j = 1, \ldots, n \ldots$ indexy jednotlivých uzlů,
- $d_{ij} \ldots$ kladná vzdálenost mezi uzly $i$ a $j$,
- $x_{ij} \in \{0,1\} \ldots$ indikátor použití (=1) cesty z uzlu $i$ přímo do uzlu $j$,
- $u_i \ldots$ pořadové číslo navštívení uzlu $i$.

Základní model TSP pro celočíselné lineární programování podle (Miller et al., 1960) lze zapsat následovně

$$Z = \sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}x_{ij} \longrightarrow MIN, \quad i \neq j, \tag{2.1}$$

$$\sum_{j=1}^{n} x_{ij} = 1, \quad \forall i, \tag{2.2}$$

$$\sum_{i=1}^{n} x_{ij} = 1, \quad \forall j, \tag{2.3}$$

$$u_i - u_j + nx_{ij} \leq n-1, \quad i \neq j, i = 1,\ldots,n,\ j = 2,\ldots,n, \tag{2.4}$$

$$x_{ij} \in \{0,1\}, \quad \forall i, \forall j, \tag{2.5}$$

$$u_i \in \mathbb{Z}^+, \quad \forall i, \tag{2.6}$$

kde účelová funkce (2.1) minimalizuje celkovou ujetou vzdálenost za podmínek:

- ▷ (2.2) – z každého uzlu $i$ se vyjede právě jednou,
- ▷ (2.3) – každý uzel $j$ může být navštíven pouze jednou,
- ▷ (2.4) – zamezuje tvoření nežádoucích podcyklů navíc mezi uzly,
- ▷ (2.5) – binarita proměnných $x_{ij}$,
- ▷ (2.6) – $u_i$ patří do množiny kladných celých čísel.

Podmínku (2.4) nazýváme antismyčkovou (anticyklickou) podmínkou, označovanou jako SEC (subtour elimination constraints). V tomto znění označujeme podmínku MTZ podle autorů Miller, Tucker a Zemlin. Poslední podmínka (2.6) zaručuje, že v proměnné $u_i$ bude uložena celočíselná cesta, která tvoří optimální hamiltonovský cyklus v podobě seřazených uzlů podle $i = 1,2,\ldots,n$.

### 2.1.2 Metoda hrubé síly – Brute force

Jedná se o velmi problematický postup k hledání řešení. Sice je jisté, že najdeme nejlepší řešení, ovšem při rostoucím množství uzlů na grafu se výpočetní doba významně prodlužuje. Složitost tohoto algoritmu je $\mathcal{O}(\frac{(n-1)!}{2})$ pod podmínkou symetričnosti vzdáleností. S touto složitostí však nelze operovat vzhledem k množství hvězd, které naše galaxie, Mléčná dráha, obsahuje. Popis algoritmu dle (Abid et al., 2015) viz algoritmus 1.

**Algoritmus 1** Metoda hrubé síly

1: Spočtěme celkové množství možných řešení.
2: Vypišme všechna možná řešení (všechny permutace). ▷ Lze vynechat symetrická řešení.
3: Spočtěme délky tras pro všechna řešení.
4: Vybereme nejkratší trasu a označme ji jako optimální řešení.

### 2.1.3 Metoda větvení a mezí – Branch and bound

Metoda větvení a mezí (B&B) se používá pro řešení velkých úloh. Skládá se ze tří hlavních složek: výběru zpracovávaného uzlu, výpočtu vazeb a větvení podle (Coutinho et al., 2016). S drobnou úpravou se také používá pro řešení dalších celočíselných problémů a je nedílnou součástí silných výpočetních softwarů typu Gurobi a Concorde.

Hlavní myšlenkou je dělení množiny přípustných řešení, díky kterému se velikost problému postupně zmenšuje. V těchto podmnožinách je již mnohem jednodušší prohledávat hodnoty účelových funkcí a hledání optimální hodnoty. Používá se také k určení dolní meze hodnoty optimální minimalizační účelové funkce pomocí relaxace. Díky tomu můžeme zrychlit postup řešení úlohy a počítat, jak daleko jsme od nejlepšího možného řešení.

V praxi se řešení interpretuje pomocí prohledávání stromu, které může být provedeno jak do šířky pro vyšší kvalitu řešení, tak do hloubky pro rychlejší prohledávání uzlů díky stanovení dolní meze (lower bound). Zjednodušený algoritmus 2 inspirovaný (Balas et al., 1983).

Jako další alternativa k této metodě je Metoda větvení a řezů.

**Algoritmus 2** Metoda větvení a mezí

1: Inicializace horní meze účelové funkce $U$ (nekonečno).
2: Vyberme libovolný startovní uzel a vytvořme list nevyřešených podproblémů *seznam*.
3: **while** *seznam* $\notin \{\}$ **do**
4: Vyberme podproblém $i$ ze *seznamu*, odstraňme ho a ohodnoťme účelovou funkcí $Z_i$.
5: **if** $Z_i < U$ **then**
6: Řešení je validní a uložme $U \leftarrow Z_i$.
7: **else**
8: Řešení bude horší nebo stejné.
9: Další iterace while cyklu.
10: **end if**
11: Aplikujme pravidlo (např. do hloubky) větvení problému, generujme další větve k řešení a aktualizujme *seznam*.
12: **end while**
13: Hodnota účelové funkce uložena v $U$.

## 2.2 Heuristiky

Heuristiky jsou algoritmy, které se používají pro řešení daného problému. Ve většině případů je jejich výpočetní složitost nízká, což snižuje i výpočetní dobu, ale šance nalezení optimálního řešení je mizivá. Používají se tedy k nalezení přípustného řešení blízkého optimálnímu řešení, též označovanému jako suboptimální řešení.

Heuristiky lze rozdělit do několika podskupin:

- ▷ konstruktivní heuristiky (NN, Greedy, ...),
- ▷ zatřiďovací heuristiky (vkládací, zatřiďovací, ...),
- ▷ zlepšující heuristiky (LKH, ...).

Vstupem pro výpočet heuristik je zpravidla matice vzdálenosti $D$ mezi jednotlivými uzly daná rozměry $n \times n$. Ve speciálních případech mohou pomoci i souřadnice jednotlivých uzlů (metoda konvexního obalu).

### 2.2.1 Nejbližší soused

Algoritmus nejbližšího souseda (Nearest neighbor, NN) se často zaměňuje s hladovým algoritmem popsaným v kapitole 2.2.2, i když jeho hamižnost je významná a je zařazován do třídy hladových algoritmů. Jeho krása spočívá v jednoduchosti a počáteční efektivitě. Ta je ovšem ke konci algoritmu penalizována používáním delších a delších hran, které vedou k neefektivitě algoritmu. Smysl algoritmu spočívá v nalezení nejkratší hrany, kam se můžeme přesunout, a její následné použití k přesunu. Jak uvádí (Kizilateş et al., 2013), algoritmus je popsán viz algoritmus 3. Složitost výpočtu je $\mathcal{O}(n^2)$.

**Algoritmus 3** Nejbližší soused

1: Navštivme náhodný uzel. ▷ Většinou první.
2: **while** Bylo navštíveno méně než $n$ uzlů. **do**
3: Vyberme nejbližší nenavštívený uzel k poslednímu navštívenému a navštivme ho.
4: **end while**
5: Vraťme se do náhodně vybraného uzlu z kroku 1, uzavřeme Hamiltonův cyklus a spočtěme uraženou vzdálenost.

### 2.2.2 Hladový algoritmus

Hladový algoritmus (Greedy algorithm) se podobá algoritmu nejbližšího souseda z kapitoly 2.2.1. Hlavním rozdílem je, že nevybíráme pouze z hran, kam se můžeme přesunout z posledního navštíveného bodu, ale vždy vybíráme nejkratší přípustnou hranu na grafu. Zde se komplikuje algoritmus tím, že se musí kontrolovat, zda nedojde k vytvoření podcyklu, což by znemožnilo vytvoření Hamiltonova cyklu. Algoritmus 5 popisuje jednu z možných metod, tedy vyčerpávající smyčku (Jackovich, 2019). Ta funguje na principu procházení jednotlivých nalezených sledů a hledání, zda nevytvoří cyklus. Složitost algoritmu je $\mathcal{O}(n^2 \log n)$. Obecný postup dle (Pelikán, 2001), viz algoritmus 4.

**Algoritmus 4** Hladový algoritmus

Setřiďme všechny hrany od nejkratší po nejdelší.
**while** Máme méně než $n-1$ hran v cestě. **do**
    Vyberme nejkratší hranu tak, aby nebyly porušeny podmínky pro splnění Hamiltovova cyklu viz algoritmus 5, a vložme ji do cesty.
**end while**
Přidejme poslední možnou hranu tak, abychom vytvořili Hamiltonův cyklus.

**Algoritmus 5** Vyčerpávající smyčka: pro zamezení tvoření podcyklů

Vstupuje zvolená hrana daná indexy $i$,$j$ a *cesta* obsahující již použité hrany.
Inicializace kontrolních proměnných $continue, continue2 = TRUE$
**if** Uzly zvolené hrany jsou oba stupně 2 **then**
    $ocas \leftarrow i$
    $hlava \leftarrow j$
    **while** $continue = TRUE$ **do**
        **if** $hlava \in cesta.$ **then**
            $next\_node$ = daný uzel, do kterého vede cesta z hlavy ▷ Informace bude uložena v *cesta*.
            **if** $next\_node = ocas$ **then**
                Vyřaďme hranu ze zásobníku.
                Vraťme stupně uzlu $i$,$j$ na původní hodnotu.
                $continue = FALSE$ ▷ Ukončeme hledání hlavy a ocasu.
                $continue2 = FALSE$ ▷ Přeskočme zápis hrany a jděme na další hranu.
            **else**
                $continue = TRUE$
                $hlava = next\_node$ ▷ Posuňme hlavičku dopředu na další uzel.
            **end if**
        **else**
            $continue = FALSE$
        **end if**
    **end while**
**end if**
**if** $continue2 = FALSE$ **then**
    Vraťme se, vyberme novou hranu.
**end if**
Nyní při použití zvolené hrany nevytvoříme podcyklus.

### 2.2.3 Metoda výhodnostních čísel – Clark, Wright

V metodě výhodnostních čísel (Clarke, Wright's savings method) se počítá s maticí $S = s_{ij}$, která je složena z nejkratších hran mezi vrcholy v grafu. Vzorec pro výpočet matice $S$ je dán

vzorcem:

$$s_{ij} = d_{i1} + d_{1j} - d_{ij}, \quad i,j = 2,3,\ldots,n, \quad i \neq j. \tag{2.7}$$

Celkem tedy bude možno vybírat z $n^2 - 3 \cdot n + 2$ možných hran, protože nepočítáme s hlavní diagonálou a prvním řádkem s prvním sloupcem matice S. Složitost algoritmu je $\mathcal{O}(n^2 \log n)$. Pokud se objevují ve vstupní matici vzdáleností duplikátní hodnoty, může algoritmus 6 volit mezi více možnostmi najednou, a to může způsobit nalezení jiných řešení pro různé softwary, záleží totiž na způsobu utřídění pořadí. Dle (Pelikán, 2001) viz algoritmus 6.

**Algoritmus 6** Metoda výhodnostních čísel

1: Spočtěme matici výhodnostních čísel $S$ z matice vzdáleností dle vzorce (2.7).
2: Setřiďme hodnoty $s_{ij}$ sestupně.
3: **while** Vybraných cest není celkem $n - 1$. **do**
4: Zvolme hranu s nejvyšší hodnotou $s_{ij}$, tak aby nebyla porušena podmínka pro vznik Hamiltonova cyklu podle algoritmu 5 a vložme ji do cesty.
5: **end while**
6: Přidejme poslední možnou hranu tak, abychom vytvořili Hamiltonův cyklus.

### 2.2.4 Metoda vkládací

Metoda vkládací (insertion method) je založena na principu postupného přidávání uzlů a hran do cyklu. Obtížnost je poměrně jednoduchá, tedy $\mathcal{O}(n^2)$. Podrobně popsaný algoritmus viz 7 dle (Pelikán, 2001).

**Algoritmus 7** Metoda vkládací

1: Určeme výchozí uzel. Označíme číslem 1.
2: Najděme nejvzdálenější uzel od uzlu 1 pomocí vzorce $d_{1s} = \max(d_{1j})$. Vytvořme výchozí uzavřenou cestu $C : 1 - s - 1$. Uložme uzel a hrany do množiny již použitých uzlů $V'$ a hran $E'$.
3: **while** $V \neq V'$ **do** ▷ Dokud cesta neobsahuje všechny uzly.
4: Najděme uzel $k$ neležící ve $V'$, který je nejblíže k uzlům již zařazeným do $V'$.
5: Nalezněme hranu se souřadnicemi $i,j$ z $V'$, pro kterou je minimální $d_{ik} + d_{kj} - d_{ij}$.
6: Vložme uzel $k$ mezi uzly $i$ a $j$ do množiny $V'$.
7: Množinu $E'$ doplňme o hrany $(i,k)$ a $(k,j)$.
8: Vyřaďme z množiny $E'$ hranu $(i,j)$.
9: **end while**
10: Výsledný Hamiltonův cyklus je uložen v hranách $E'$ a uzlově ve $V'$.

### 2.2.5 Metoda konvexního obalu

Metoda konvexního obalu (Convex hull method) se obecně používá pro vzdálenosti, které jsou dány Euklidovsky. Důležitým poznatkem je, že v pořadí, jak je tvořen konvexní obal, tak se ve

stejném pořadí jednotlivé uzly objevují v optimální cestě. Pro body zadané ve 3D však může docházet ke komplikacím oproti 2D zobrazení. Zvyšuje se výpočetní náročnost a snižuje se efektivita (nelze správně určit hrany, které jsou stěžejní pro konvexní obal). Obecně je tento přístup vhodný pro 2D problémy.

Použití této heuristiky tedy vyloučíme, ale pro úplnost práce zmíníme algoritmus 8 dle (Golden et al., 1980). Včetně matice vzdáleností $D$ vstupují do algoritmu i souřadnice uzlů.

**Algoritmus 8** Metoda konvexního obalu

1: Vytvořme konvexní obal, který bude obsahovat všechny uzly $V$ z grafu $G$.
2: Uzly v pořadí vytvoření obalu zařaďme do cesty $C$ jako výchozí cyklus.
3: **while** Nemáme využity všechny uzly $V$ v cestě $C$. **do**
4: Pro každý uzel $k$, který neleží v $C$ nalezněme hranu $(i,j)$, pro kterou je minimální délka zajížďky dle vzorce $d_{ik} + d_{kj} - d_{ij}$.
5: Vyberme takový uzel $k^*$ z $k$, pro který je minimální hodnota z předpisu

$$\frac{d_{i^*k^*} + d_{k^*j^*}}{d_{i^*j^*}}.$$

6: Uzel $k^*$ vložme mezi uzly $(i^*,j^*)$ v cyklu $C$.
7: **end while**

### 2.2.6 Metoda minimální kostry

Metoda minimální kostry (minimum spanning tree method, metoda MST) slouží pro symetrické úlohy a její složitost je $\mathcal{O}(n^2)$. Byla vyvinuta v práci (Held et al., 1970), ze které vychází algoritmus 9. Výhodou tohoto algoritmu je, že vytváří menší skupinky uzlů, které spojí nejkratší cestou. Problém ovšem nastává, když začne propojovat tyto skupinky. Kvůli tomu dochází k neefektivitě řešení. Tato metoda je ovšem díky pevnému základu vhodná pro následné úpravy pomocí algoritmů zlepšujících řešení. Součástí algoritmu je i úloha převodu Eulerova cyklu na Hamiltonův cyklus. Algoritmus 10 tuto úlohu právě řeší a je inspirován Hierholzerovým algoritmem.

**Algoritmus 9** Metoda minimální kostry

1: Nalezněme $T$ minimální kostru grafu $G$.
2: Pomocí zdvojení hran kostry $T$ získejme Eulerův cyklus.
3: Transformace Eulerova cyklu na Hamiltonův cyklus viz algoritmus 10.
4: Spočtěme hodnotu cesty $C$.

**Algoritmus 10** Převod Eulerova cyklu na Hamiltonův cyklus

1: Vytvořme posloupnost uzlů ležících na Eulerově cyklu.
2: Z této posloupnosti vyřaďme duplikátní uzly (první výskyt ponecháme) a vytvořme výslednou cestu $C$.
3: Ujistěme se, že cesta $C$ začíná a končí ve stejném uzlu (zde se jediná duplicita ponechá).

### 2.2.7 Christofidova metoda

V Christofidově metodě (Christofides algorithm) se stejně jako v metodě z kapitoly 2.2.6 využívá minimální kostry grafu, ovšem zpravidla dosahuje lepších výsledků. Tentokrát však propojujeme pouze ty uzly, které mají lichý stupeň. Díky tomu získáme propojení vzdálenějších oblastí v místě větvení a koncových uzlů. Každý uzel pak je sudého stupně. Christofidova metoda by měla zpravidla dosahovat lepších výsledků než metoda minimální kostry.

K řešení se používá dále metody perfektního párování (perfect matching) s minimálními náklady, která hledá minimální náklady mezi páry vybraných uzlů tak, aby mezi uzly nezbyl žádný osamocený. Lze použít model LP nebo Edmondsův algoritmus, který byl v posledních letech vylepšován. Aktuálně nejlepší verze se nazývá „Blossom V“ viz (Kolmogorov, 2009).

Nejhorší případ složitosti algoritmu je $\mathcal{O}(n^3)$, hlavně kvůli užití metody perfektního párování, která ovšem může počítat s méně než $n$ uzly. Představujeme algoritmus 11 dle (Nilsson, 2003).

**Algoritmus 11** Christofidova metoda

1: Nalezněme $T$ minimální kostru grafu $G$.
2: Vyberme uzly lichého stupně v $T$ a spojme je hranami $PM$ metodou perfektního párování s minimálními náklady (Edmondsův algoritmus).
3: Nové hrany $PM$ přidejme do minimální kostry grafu $T$.
4: Transformujme Eulerův cyklus z $T$ na Hamiltonův cyklus, např. užitím algoritmu 10.

### 2.2.8 Metoda zatřiďovací

V metodě zatřiďovací pracujeme s několika cykly, které obsahují všechny uzly na grafu $G$, a snažíme se je postupně pospojovat tak, aby to bylo co nejvýhodnější. To provedeme tak, že hledáme mezi všemi cykly takové 2 cykly $A$ a $B$, kde pomocí hran $c_{ij} \in A$ a $c_{kl} \in B$ najdeme minimální hodnotu $value$:

$$value = c_{il} + c_{kj} - c_{ij} - c_{kl}. \tag{2.8}$$

Jakmile hrany $c_{ij}$ a $c_{kl}$ nahradíme hranami $c_{il}$ a $c_{kj}$, získáme z cyklů $A$ a $B$ jeden cyklus a opakujeme, pokud máme stále více než jeden cyklus. Postup výpočtu dle (Pelikán, 2001), viz algoritmus 12. Složitost algoritmu je $\mathcal{O}(n^2)$.

Běžně se v prvním kroku metody vytvoří pro každý uzel cyklus. Logicky se nabízí použít pro inicializaci metody algoritmus, který umí rozdělit data (jednotlivé uzly) do několika oblastí (cyklů). Příkladem takové metody může být shluková analýza (clustering). Zároveň se zatřiďovací metodou mohou řešit problémy jiných přístupů, při kterých mohou vzniknout podcykly a ty touto metodou postupně spojit v jeden cyklus.

**Algoritmus 12** Metoda zatřiďovací

1: Pro každý uzel vytvořme elementární cyklus. ▷ Bude jich celkem $n$.
2: **while** Počet cyklů $\neq 1$ **do**
3: V množině všech cyklů hledejme takové 2 cykly, které mají minimální vzdálenost dle vzorce (2.8) a označme je jako $A$ a $B$.
4: Cykly $A$ a $B$ spojme hranami $c_{ik}$,$c_{kj}$ použitými pro výpočet minimální hodnoty.
5: Odstraňme hrany $c_{ij}$ a $c_{kl}$. ▷ Nyní máme ze dvou cyklů jeden.
6: **end while**
7: Výsledný cyklus je Hamiltonův cyklus.

### 2.2.9 Metoda výměn – Lin, Kerninghen

Metoda výměn (Lin Kerninghen method) je určena pro zlepšování již nalezeného řešení. V praxi se velmi častou používá v kombinaci s metodou nejbližšího souseda 2.2.1 nebo hladového algoritmu 2.2.2.

Vylepšení řešení je dosaženo pomocí výměny 2 hran za jiné 2 hrany (2-opt). Lze provést i výměny 3 a 3 hran (3-opt). Pro výpočet tedy již předpokládáme uspořádaný Hamiltonův cyklus $C$ popsaný jednotlivými uzly a množinou použitých hran $E$.

Pomocí této metody se dnes řeší nejrozsáhlejší úlohy, které byly kdy vytvořeny s poměrně vysokou úspěšností. Je však třeba dodat, že k řešení je použito i mnoho dalších přístupů. Zjednodušený postup podle (Helsgaun, 2000) viz algoritmus 13.

Metoda končí, pokud již byly použity všechny dvojice hran a nebylo dosaženo lepšího řešení. Nebo lze také použít omezený počet výměn.

**Algoritmus 13** Metoda výměn Lin–Kerninghen 2-opt

1: **while** Ukončovací kritérium není splněno. **do**
2: Vyberme 2 takové hrany, které nemají společný vrchol. Označme je $(v_i,v_{i+1})$ a $(v_j,v_{j+1})$ kde $i+1<j$.
3: Vyměňme hrany $(v_i,v_{i+1})$ a $(v_j,v_{j+1})$ za hrany $(v_i,v_j)$ a $(v_{i+1},v_{j+1})$.
4: Ohodnocení Hamiltonova cyklu $C$ se změní o $\delta$.
5: **if** $\delta < 0$ **then**
6: Cyklus změňme na nový a vraťme se do kroku 1.
7: **end if**
8: **end while**

Ukončovací kriterium lze nadefinovat tak, aby výpočet končil po určitém počtu iterací, nebo když nedokáže algoritmus nalézt výměnu dvou hran, které by nezlepšily řešení.

## 2.3 Metaheuristiky

Metaheuristiky jsou přístupy, které se používají pro řešení obecných problémů. Po drobných úpravách je lze použít na dané problémy jako je například diskrétní TSP. Optimální řešení mohou nalézt, ale spíše naleznou řešení blížící se optimálnímu (suboptimální). V porovnání s exaktními metodami jsou metaheuristiky rychlejší, ale méně přesné.

Pro následující podkapitoly si zavedeme hromadnou symboliku:

- ▷ $X \ldots$ množina přípustných řešení obecného optimalizačního problému,
- ▷ $x_i \ldots$ řešení $i$ v podobě permutace (cesty),
- ▷ $f(x) \ldots$ hodnota účelové funkce řešení $x$,
- ▷ $x^* \ldots$ (dočasné) optimální řešení dané účelovou funkcí $f(x^*)$,
- ▷ $N(x) \subset X \ldots$ okolí bodu $x$, platí $x \in N(x)$.

Do okolí řešení $N(x)$ se lze dostat řadou způsobů:

- ▷ prohozením návštěvy dvou po sobě jdoucích uzlů v Hamiltonově cyklu,
- ▷ inverzí cesty na její určité části,
- ▷ jeden krok LK heuristiky 2-opt nebo 3-opt viz 2.2.9,
- ▷ libovolnou drobnou změnou, která řešitele úlohy napadne.

Ukončovací kritéria pro hlavní cykly jednotlivých metaheuristik lze obecně definovat:

- ▷ Bylo dosaženo předem určeného množství iterací $R$.
- ▷ Po předem určeném počtu iterací $R$ nedošlo ke zlepšení řešení.
- ▷ Neexistuje další klíč, podle kterého můžeme pokračovat dále ve výpočtu.

Časově jsou metaheuristiky náročnější než heuristiky, ale zpravidla dosahují lepších výsledků. Některé metaheuristiky dokonce počítají s implementací některých heuristik pro nalezení dobrého přípustného řešení. Při aplikaci těchto metod často pracuje náhoda, která může při opakování metod dávat jiná řešení.

### 2.3.1 Metoda lokálního hledání

Metoda lokálního hledání (Local search method) pracuje se zlepšováním již nalezeného přípustného řešení, které vylepšuje. Lze zvolit několik možností, jak se přesunout do okolního řešení, např. LK heuristiku 2-opt. Metoda hledá pouze lokální minimum, nikoliv globální, kterého ovšem může ve specifických případech být dosaženo. Je tedy zřejmé, že problémem této metaheuristiky je, že se nedokáže odpoutat od lokálního minima. Jednoduchý zápis algoritmu dle (Pelikán, 2001) viz 14.

**Algoritmus 14** Metoda lokálního hledání

1: Nalezněme libovolné $x \in X$.
2: **for** Pro všechna $x' \in N(x)$ **do**
3: **if** $f(x') < f(x)$ **then** ▷ Pokud je hodnota nové účelové funkce nižší...
4: $x \leftarrow x'$ ▷ Přesuňme se z uzlu $x$ do uzlu $x'$.
5: Pokračujme další iterací smyčky.
6: **else**
7: Výpočet končí a $x$ je řešením.
8: **end if**
9: **end for**

### 2.3.2 Tabu vyhledávání

S drobnou úpravou metody 2.3.1 lze prohledávat i vyšší hodnoty účelové funkce než aktuální nejnižší nalezené řešení, abychom se dokázali odvrátit od lokálního minima a měli vyšší šanci nalézt globální minimum. Tabu vyhledávání (Tabu search) je určeno k prohledávání iteračních problémů. Již prohledané varianty se přidávají do tabu listu ($TL$) a poskytují seznam, který již není třeba prohledávat. Množina $\overline{N}(x)$ obsahuje ta přípustná řešení z $N(x)$, která nejsou v seznamu $TL$. Jelikož se $TL$ rozšiřuje a zabírá mnoho paměti, je záhodno ho postupně redukovat od nejstarších zápisů. Naopak nejnovější zápisy jsou nejdůležitější a je potřeba je ponechat. Algoritmus dle (Pelikán, 2001) a (Gupta, 2013) nalezneme zde, viz 15.

Mezi ukončovací kritéria lze zahrnout další a to, že bude dosaženo $\overline{N}(x) = \{\}$ nebo $N(x) = \{\}$.

**Algoritmus 15** Tabu vyhledávání

1: Nalezněme výchozí řešení $x \in X$.
2: Označme ho jako dosud nejlepší řešení $x^* \leftarrow x$.
3: **while** Ukončovací kritérium nebylo splněno. **do**
4: Nalezněme $x' \in \overline{N}(x) \subset N(x)$ takové, aby účelová funkce $f$ byla minimální na $\overline{N}(x)$.
5: Zapišme řešení $x$ do $TL$ a dosaďme $x \leftarrow x'$. ▷ Již se neobjeví v $\overline{N}(x)$.
6: **if** $f(x) < f(x^*)$ **then**
7: $x^* \leftarrow x$.
8: **end if**
9: Redukce $TL$. ▷ Volitelné.
10: **end while**
11: Řešením je $x^*$.

### 2.3.3 Metoda prahové akceptace

Metoda prahové akceptace (Threshold acceptance, TA) vzniká opět drobnou úpravou metody z kapitoly 2.3.1. V této metodě se lepší hodnota účelové funkce hned neoznačí jako nejlepší, nýbrž pouze tehdy, pokud bude vylepšeno ($f(x') - f(x)$) o předem stanovenou hodnotu. Tuto hodnotu označujeme jako práh $T$ (threshold). Pokud budeme tento práh postupně

snižovat parametrem $r \in (0,1)$, pak zabráníme zacyklení a povolíme přechod do lokálního minima. Pokud však bude $r = 0$, jedná se o lokální hledání. Opět však není zaručeno nalezení globálního minima. Algoritmus podle (Pelikán, 2001) a (Reinelt, 2003)[kap. 9] viz 16.

**Algoritmus 16** Metoda prahové akceptace

1: Nalezněme výchozí řešení $x$.
2: Stanovme výchozí hodnotu prahu $T$ a jeho koeficient snižování $r \in (0,1)$.
3: Označme $x^* \leftarrow x$.
4: **while** Není splněno ukončovací kritérium. **do**
5: Náhodně vyberme řešení z okolí řešení $x$. ▷ Jeden krok 2-opt, prohození dvou uzlů...
6: **if** $f(x') - f(x) < T$ **then**
7: Dosaďme $x \leftarrow x'$.
8: **if** $f(x) < f(x^*)$ **then**
9: Změňme $x^* \leftarrow x$.
10: **end if**
11: **end if**
12: Upravme prahovou hodnotu $T \leftarrow T \cdot r$.
13: **end while**

### 2.3.4 Metoda simulovaného žíhání oceli

Metoda simulovaného žíhání oceli (simulated anneling method, SA, SIAM) rozšiřuje metodu lokálního hledání z kapitoly 2.3.1 o stochastickou složku. Je založena na fyzikálních principech. Laicky řečeno „kováme, dokud je železo žhavé“. Zchladlá část oceli se již netvaruje a postupně celá vychladne.

Potřebujeme stanovit hodnotu teploty $T > 0$, při které stále kováme. Dalším zmíněným prvkem je ochlazování dané $r \in (0,1)$. Určíme také dobu $R$, po kterou budeme ocel žíhat. Přidáme stochastickou složku v podobě pravděpodobnosti přechodu (úspěšného kutí) danou vzorcem

$$P = \exp \frac{-(f(x') - f(x))}{T}. \tag{2.9}$$

Rozdíly v hodnotách dvou účelových funkcí lze definovat $\delta = f(x') - f(x)$ a můžeme říci, že pro hodnoty $\delta$ je pravděpodobnost přechodu určena:

- ▷ malé $\delta$, $x'$ je jen o málo horší – přechod s vysokou pravděpodobností,
- ▷ velké $\delta$, pravděpodobnost přechodu je nízká,
- ▷ pokud $\delta < 0$, pak pravděpodobnost přechodu je 100 %.

Algoritmus dle (Skiscim et al., 1983) viz 17.

**Algoritmus 17** Metoda SIAM

1: Nalezněme výchozí řešení $x$. Zvolme parametry metody $T, r, R$.
2: **while** Není splněno ukončovací kritérium. **do**
3: Zvolme řešení $x'$ z okolí $N(x)$. ▷ 1 krok 2-opt nebo náhodně.
4: **if** $f(x') < f(x)$ **then** ▷ Lepší je nové než jeho předchůdce.
5: Dosaďme $x \leftarrow x'$.
6: **else** ▷ Nové je horší než předchůdce.
7: Dosaďme $x \leftarrow x'$ s pravděpodobností $P$ podle (2.9).
8: **end if**
9: **if** $f(x^*) > f(x)$ **then** ▷ Lepší než dosud nejlepší?
10: Dosaďme $x^* \leftarrow x$
11: **end if**
12: Přepočítejme teplotu $T \leftarrow r \cdot T$. ▷ Ochlazování.
13: **end while**

### 2.3.5 Genetický algoritmus

Genetický algoritmus (Genetic algorithm, GA) je postaven na základních principech evoluce – přirozeného výběru. Přežijí silní a slabí se vytratí. Genetická informace jednotlivce je uchovávána v chromozomu. Jednotlivci tvoří populaci, která obsahuje celkem $N$ jedinců. Ti jsou každou generací lepší a nestane se, že by slabší jedinec nahradil silnějšího jedince z populace. Platí však, že silnější jedinec nahradí slabšího v populaci.

Nejlepší jedinec v populaci se označuje jako fitness. To, jak je který jedinec silný, se vypočítá pomocí účelové funkce, kterou pro případ TSP minimalizujeme.

Populace prochází postupně generacemi (počet generací dán parametrem $G$). Noví potomci vznikají několika možnostmi:

- ▷ křížení matky a otce (případně i více jedinců),
- ▷ mutace (inverze),
- ▷ reprodukce.

Výběr otce a matky lze provést mnoha způsoby:

- ▷ podle hodnoty účelové funkce,
- ▷ ruletové pravidlo,
- ▷ stochastické univerzální vzorkování,
- ▷ turnajový výběr,
- ▷ elitářství,
- ▷ náhodný výběr.

Cestu lze podle (Larranaga et al., 1999) reprezentovat dvěma způsoby. Buď permutací jed-

notlivých neopakujících se uzlů nebo binárním zápisem jednotlivých uzlů např.

$$i = 1 \ldots i = 0001,$$
$$i = 2 \ldots i = 0010,$$
$$i = 3 \ldots i = 0011.$$
$$\ldots$$

Pro binární zápis bychom získali vhodný vstup pro GA, ovšem dosahuje horších výsledků než použití permutačního typu zápisu, jak v závěru uvádí (Larranaga et al., 1999). Jak též představuje (Applegate et al., 2006), je vhodnější používat permutační typ zápisu a budeme se ho tedy držet i pro tuto metodu.

#### Výběr rodičů

Jedním z prvků algoritmu GA je výběr rodičů. My si uvedeme 2 příklady, další způsoby byly uvedeny v práci (Chudasama et al., 2011), která obsahuje i jejich porovnání.

Výběr pomocí ruletového kola je chápán jako náhodný výběr. Nicméně není úplně náhodný, protože čím má jedinec v populaci lepší účelovou funkci, tím větší část kola nepřímo úměrně zabírá. Výběr tedy probíhá následovně. Přiřadíme jedincům z populace interval, který jim bude patřit. Následně vygenerujeme náhodné číslo a to vybereme z populace prvního rodiče.

Pro turnajový výběr se vytvoří náhodně v populaci páry. Lepší v páru postupuje do dalšího kola eliminace a poražený vypadává. Kol je tolik, dokud není dosažený určený počet jedinců, ze kterých se následně stanou rodiče. Ideálně doporučujeme použít počet jedinců v populaci $2^q$, kde $q \in \mathbb{N}$ a $q$ dostatečně velké na to, aby byl přiměřený počet rodičů (neuvažujme cca. $q < 5$). Důvodem je, aby byl vždy sudý počet rodičů a vždy se dobře vytvořil pár. Zároveň lze snáze naplánovat počet rodičů.

Obecně lze vytvořit $K$ turnajový výběr, kde $K$ určuje počet kandidátů, kteří se uchází o rodičovství. Nejlepší z množství $K$ uchazečů pak je vybrán jako jeden z rodičů. Výše uvedený případ pak popisuje $K = 2$ případ, kdy uvažujeme výběr rodiče jen jednou.

#### Křížení

Při křížení se přenáší část genetické informace otce ($x_o$) i matky ($x_m$) na potomka. Lze křížit s jednou změnou, dvěma změnami nebo s rovnoměrným křížením. Jednotlivé druhy křížení a jejich modifikace viz (Hussain et al., 2017). Jako příklad k ukázkám křížení uvažujme:

$$x_o = (1, 2, 3, 4, 5, 6),$$
$$x_m = (1, 5, 6, 3, 4, 2).$$

Jednobodové křížení v polovině předpisu chromozomu pak vyprodukuje potomky s částí kódu otce i matky následovně:

$$x_1 = (1, 2, 3, 3, 4, 2),$$
$$x_2 = (1, 5, 6, 4, 5, 6).$$

Dvoubodové křížení (po 2. a 4. uzlu, takto budeme uvažovat i v nadcházejících příkladech) pak může vytvořit potomky s dvěma částmi kódu otce a jednou částí matky a naopak. Výsledek vypadá následovně:

$$x_1 = (1, 2, 6, 3, 5, 6),$$
$$x_2 = (1, 5, 3, 4, 4, 2).$$

Můžeme aplikovat podobně i vícebodové křížení.

Jak vidíme, v těchto jednoduchých zápisech se však objevují některé uzly víckrát, což způsobuje porušení podmínky, že můžeme každý uzel v TSP navštívit pouze jednou. Proto uvádíme následující metody, které již jsou vhodné pro řešení TSP.

Podle Davisova pořadového křížení (ordered crossover, OX1) lze použít dvoubodové křížení obdobně, kde se použije „střední část“ (jádro) chromozomů jednoho z rodičů pro potomky následovně:

$$x_1 = (\_, \_, 3, 4, \_, \_),$$
$$x_2 = (\_, \_, 6, 3, \_, \_).$$

Doplní je druhý z rodičů náhodnými uzly tak, aby byly použity jen ty, které ještě nejsou v chromozomu potomka např.:

$$x_1 = (1, 2, 3, 4, 6, 5),$$
$$x_2 = (1, 4, 6, 3, 5, 2).$$

Nyní vidíme, že každý uzel je pro oba potomky uveden v zápisu chromozomu pouze jednou, což nám dává přípustné řešení TSP.

Částečně zmapované křížení (Partially mapped crossover, PMX) můžeme také použít pro TSP. Opět použijeme dvojí křížení a prohodíme jádra obou rodičů. Pokud vznikne duplikace některých uzlů, změníme je pomocí mapování tak, aby mimo jádro byl uzel změněn zpět, ukázka zisku jádra:

$$x_1 = (\_, \_, 6, 3, \_, \_),$$
$$x_2 = (\_, \_, 3, 4, \_, \_).$$

Nyní opíšeme pro $x_1$ chybějící uzly z otce a obdobně pro $x_2$ z matky.

$$x_1 = (1, 2, 6, 3, 5, 6),$$
$$x_2 = (1, 5, 3, 4, 4, 2).$$

Vidíme, že máme v chromozomech potomků duplikáty. Pro $x_1$ to je uzel 6. Při výměně jader uzel 6 nahradil uzel 3, takže prohodíme tuto hodnotu mimo jádro $x_1$. Obdobně opět prohodíme uzel 3 na stejné pozici za uzel 4, protože byl také vyměněn jádrem. Pro $x_2$ tu je uzel 4 duplikátní. Uzel 4 opět vyměnil v jádru uzel 3, takže provedeme náhradu mimo jádro. Bohužel však uzel 3 již máme v chromozomu a opět se podíváme, jaký uzel byl nahrazen novým uzlem 3. Byl to uzel 6 a opět provedeme záměnu mimo jádro. Provedeme následovně, nejprve:

$$x_1 = (1, 2, 6, 3, 5, 3), \qquad 6 \leftarrow 3,$$
$$x_2 = (1, 5, 3, 4, 3, 2), \qquad 4 \leftarrow 3.$$

A následně provedeme další korekci.

$$x_1 = (1, 2, 6, 3, 5, 4), \qquad 3 \leftarrow 4,$$
$$x_2 = (1, 5, 3, 4, 6, 2), \qquad 3 \leftarrow 6.$$

Běžně se provádí v jednom kroku, ale pro lepší přehlednost rozepisujeme do kroků dvou.

Existují další způsoby křížení:

- ▷ pořadové křížení 2 (OX2),
- ▷ náhodné křížení (Shuffle crossover),
- ▷ cyklické křížení (Cycle crossover).

Jejich rozebrání však, jako mnohé další metody, tato práce nezahrnuje.

#### Mutace

Při mutaci se prohodí bity (pro TSP uzly) v chromozomu potomka. Existují 3 možnosti mutace.

1. Prohození (Swap) změní v chromozomu pozici dvou uzlů.
2. Náhodná (Scramble) mutace prohodí v permutaci v určeném úseku náhodně všechny uzly.
3. Inverzní mutace otočí celé označené pořadí permutace, např. $x_1$ = (1,2,3,4,5,6,7,8,9) pak může mít po inverzní mutaci na 3.–7. uzlu předpis $x_1'$ = (1,2,7,6,5,4,3,8,9).

#### Reprodukce

Při reprodukci zůstane jedinec takový jaký byl v předchozí generaci, tzn. $x_1 \leftarrow x_1$.

#### Další kroky

Šance, že bude použit právě daný typ je dán pravděpodobnostními parametry. Po výpočtu poslední generace je ve fitness funkci uložen (sub)optimální jedinec, který je nejlepším

nalezeným řešením.

Populaci inicializujeme náhodnými přípustnými řešeními, ale můžeme mezi ně přidat i řešení nalezená pomocí jednoduchých přístupů. Pro případ TSP tomu mohou být například nejbližší soused, hladový algoritmus nebo třeba metoda minimální kostry. GA dle (Potvin, 1996) lze zapsat viz algoritmus 18.

Ukončovací kritérium lze nastavit mnoha způsoby. Například po provedení určeného množství iterací, po určitém množství iterací nebylo nalezeno lepší řešení nebo třeba dosažené určité hodnoty účelové funkce.

**Algoritmus 18** Genetický algoritmus

1: Zvolme $N$ jako libovolné kladné celé číslo.
2: Výchozí populace – vygenerujme $N$ přípustných řešení (chromozomů). Případně vložme jedno řešení podle libovolné heuristiky nalézající přípustné řešení.
3: Hodnotu nejlepší účelové funkce $f(x)$ uložme $f(x^*) \leftarrow f(x)$.
4: **while** Není splněno ukončovací kritérium. **do**
5: Vyberme rodiče pomocí některého výběrů z podkapitoly 2.3.5 a vytvořme páry.
6: Proveďme křížení párů pro vznik dětí. Ty případně mutujme nebo ponechme jaké jsou.
7: Nové prvky vložme do populace a seřaďme vzestupně podle hodnoty účelových funkcí.
8: Odeberme takový počet prvků z populace, aby bylo v populaci přesně $N$ řešení. Z populace odstraňme vždy ta nejhorší řešení (pro TSP nejvyšší hodnoty).
9: **if** $f(x) < f(x^*)$ **then**
10: Dosaďme $f(x^*) \leftarrow f(x)$.
11: **end if**
12: **end while**
13: Výpočet ukončen, výsledkem je řešení $x^*$.

### 2.3.6 Optimalizace mravenčí kolonií

Optimalizace mravenčí kolonií (Ant colony optimization, ACO) vychází z myšlenky spolupráce v rámci roje. Mravenci (artificial ants = umělí mravenci, agenti) si při cestě za potravou předávají informace pomocí feromonu. Ten mravenci řekne, kudy lze dojít k potravě a které cesty již využili ostatní mravenci. Mravenec se pak sám rozhodne, jestli použije cestu, kterou už použil jiný mravenec, nebo třeba použije cesty od dvou předchůdců a zkombinuje je. Také se může rozhodnout najít úplně novou cestu. Vše se rozhoduje podle síly feromonu (čím více mravenců půjde jednou cestou, tím bude feromon silnější) a nastavením parametrů. Feromon však časem vyprchává, což dává příležitost mravenci nalézt nové neprozkoumané cesty. Díky tomu můžeme překonat lokální minima a nalézt lepší řešení, ne však nutně optimální řešení.

Algoritmus 19 se snaží simulovat chování mravenců, jak prezentuje (Chitty, 2017), a lze jej přizpůsobit pro větší problémy, protože ACO má 2 zásadní problémy. Za prvé má vysoké

nároky na paměť ukládání hodnot feromonů. A za druhé přepočet, který uzel má být navštíven jako další, je výpočetně náročný.

Speciální symbolika pro ACO:

- ▷ $\tau_{ij}$ ... hladina feromonu uloženého na hraně vedoucím z uzlu $i$ do uzlu $j$,
- ▷ $\eta_{ij}$ ... heuristická informace, která se skládá ze vzdálenosti mezi uzlem $i$ a uzlem $j$ nastavenou na $\eta_{ij} = \frac{1}{d_{ij}}$,
- ▷ $\alpha$,$\beta$ ... ladící parametry, které řídí relativní vliv feromonového ložiska $\tau_{il}$ a heuristické informace $\eta_{il}$,
- ▷ $l \in V^k$ ... uzly $l$, které jsou dostupné pro mravence $k$ na množině uzlů $V$ tak, aby se nezacyklil a vrátil do startovního uzlu. Předpokládá se, že vychází z uzlu $i$,
- ▷ $\rho$ ... rychlost odpařování, nabývá hodnot $\rho \in (0,1)$,
- ▷ $x^k$ ... trasa mravence $k$,
- ▷ $Z^k$ ... délka cesty mravence $k$ na trase $T^k$,
- ▷ $m$ ... počet mravenců k řešení na grafu $G$.

Pravděpodobnost, že mravenec $k$ přejde z uzlu $i$ do uzlu $j$ je dána vzorcem

$$P_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in V^k} [\tau_{il}]^\alpha [\eta_{il}]^\beta}. \tag{2.10}$$

Globální odpařování feromonu na každé hraně grafu představuje vzorec

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij}. \tag{2.11}$$

Každý mravenec $k$ doručí lokálně feromon do hrany na základě kvality objevené trasy

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^k. \tag{2.12}$$

Obnos doneseného feromonu na hranu je definován tak, aby zanechával více feromonů na kratších hranách následovně

$$\Delta\tau_{ij}^k = \begin{cases} \frac{1}{Z^k}, & \text{když hrana } i,j \text{ patří do } x^k, \\ 0, & \text{jinak.} \end{cases} \tag{2.13}$$

Ladění jednotlivých parametrů viz (Zhang et al., 2018).

### 2.3.7 Optimalizace včelí kolonií

Optimalizace včelí kolonií (Bee colony optimization, BCO) vychází opět z chování zvířat a opět půjde o způsob sběru potravy. Včela když objeví nový zdroj potravy, přiletí zpět k úlu a tzv. včelím tanečkem (waggle dance) v podobě osmičky sděluje úlu, kterým směrem a jak daleko se nová potrava nachází. Díky tomu se k potravě dostanou další včely. Potravu pro úlohu TSP představuje hodnota účelové funkce v podobě celkové uražené vzdálenosti. Čím je trasa kratší, tím dokáže včela přinést více nektaru. Algoritmus podle (Teodorović, 2009), (Wong et al., 2008) a (Wong et al., 2009) je dán předpisem 20.

**Algoritmus 19** Optimalizace mravenčí kolonií

1: Inicializace feromonových tras a dalších parametrů.
2: **while** Nejsou splněna kritéria ukončení. **do**
3: **for** $k = 1 : m$ **do** ▷ Pro každého mravence $k$.
4: Mravenci $k$ náhodně přiřaďme startovní uzel $i$.
5: Inicializace seznamu nenavštívených uzlů.
6: **while** Mravenec nenavštívil $n$ uzlů. **do**
7: Přesuňme mravence z uzlu $i$ do ještě nenavštíveného uzlu $j$ pomocí (2.10).
8: Aktualizujme seznam nenavštívených uzlů.
9: Uložme uzel $j$ do trasy $x^k$.
10: $i \leftarrow j$.
11: **end while**
12: Aktualizace nejlepší trasy.
13: Zanechání feromonu mravencem $k$ na trase podle (2.12) a (2.13).
14: **end for**
15: Odpařování feromonu pro další iteraci podle (2.11).
16: **end while**
17: Ukončeme algoritmus a vraťme nejlepší řešení.

Speciální symbolika pro BCO:

- ▷ $P_{ij}^t \ldots$ pravděpodobnost přechodu z uzlu $i$ do uzlu $j$ po $t$ přechodech, je dle (2.14),
- ▷ $\rho_{ij}^t \ldots$ označení vhodnosti (fitness) hodnoty hrany mezi uzly $i$ a $j$ po $t$ přechodech,
- ▷ $A_i^t \ldots$ seznam dosud nenavštívených uzlů včelou po $t$ přechodech v uzlu $i$,
- ▷ $\alpha \ldots$ relativní významnost vhodnosti hrany dle $\theta$,
- ▷ $\beta \ldots$ relativní významnost vhodnosti hrany dle matice vzdáleností,
- ▷ $\theta \ldots$ preferovaná vybraná cesta doporučená tančící včelkou,
- ▷ $\lambda \ldots$ představuje pravděpodobnost následování uzlu v $\theta$,
- ▷ $F_i^t \ldots$ uzel, který je doporučován pro navštívení z $i$ dle doporučované cesty $\theta$ po $t$ přechodech.
- ▷ $D_i \ldots$ délka tance včely $i$ v jednotkách iterací při nalezení potravy, dána vzorcem (2.16),
- ▷ $K \ldots$ uživatelem nastavený parametr k délce tance,
- ▷ $Pf_i \ldots$ výnosnost potravy včely $i$, hodnota účelové funkce, je dána vzorcem (2.17),
- ▷ $Pf_{colony} \ldots$ průměrná výnosnost potrav celého úlu (kolonie), hodnota účelové funkce, je dána vzorcem (2.18).
- ▷ $N_{Bee} \ldots$ počet včel vykonávajících včelí taneček po nalezení potravy.

Pravděpodobnost, že včela přejde z uzlu $i$ do uzlu $j$ po $t$ přechodech je dána vzorcem

$$P_{ij}^t = \frac{[\rho_{ij}^t]^\alpha [\frac{1}{d_{ij}}]^\beta}{\sum_{j \in A_i^t} [\rho_{ij}^t]^\alpha [\frac{1}{d_{ij}}]^\beta}. \tag{2.14}$$

Včelí taneček může včela provést pouze tehdy, pokud její trasa byla lepší než v posledním řešení. Každá včela si tedy musí pamatovat hodnotu své nejlepší účelové funkce během výpočtu.

Včela, která tančí, doporučuje svoji cestu ostatním včelkám.

Nyní lze vzít v úvahu více možností. Včela buď hledá trasu sama (náhodně, úprava své nejlepší trasy), nebo se nechá zlákat tanečkem jiné včely. Cestu, kterou si nechala včela poradit, označujeme jako $\theta$. Slouží tedy jako vodítko k nalezení trasy pro první případ ($j \in F_i^t, |A_i^t| > 1$) předpisu pro vhodnost hrany

$$\rho_{ij}^t = \begin{cases} \lambda, & \text{když } j \in F_i^t, |A_i^t| > 1, \\ \frac{1-\lambda|A_i^t \cap F_i^t|}{|A_i^t \setminus F_i^t|}, & \text{když } j \notin F_i^t, |A_i^t| > 1, \\ 1, & \text{když } |A_i^t| = 1. \end{cases} \tag{2.15}$$

První případ udává, že seznam nenavštívených uzlů obsahuje víc než jeden uzel a zároveň následující uzel $j$ může být stále poraděn z $\theta$. Druhý případ udává to samé, jen uzel $j$ už nemůže být poraděn. Poslední případ je nejjednodušší. Pokud zbývá navštívit poslední uzel, jednoduše ho navštívíme, protože nemáme jinou možnost.

Délka doby tance včely $i$ na přivolání včel v jednotkách počtu iterací se vypočítá vzorcem:

$$D_i = K \frac{Pf_i}{Pf_{colony}}. \tag{2.16}$$

Během tance je včela nahrazena jinou včelou z úlu a je jí přiřazen stejný index $i$. To je z toho důvodu, aby pro každou iteraci proběhl výpočet trasy pro stejné množství včel.

Výnosnost nově nalezené potravy včelou $i$ je počítána z hodnoty účelové funkce. Nicméně algoritmus vyžaduje maximalizaci potravy a účelová funkce v našem případě minimalizaci, tudíž bude pracovat s převrácenou hodnotou následovně:

$$Pf_i = \frac{1}{f(x_i)}, \tag{2.17}$$

kde $f(x_i)$ je délka včelou $i$ uražené cesty neboli hodnota účelové funkce včely $i$.

Průměrná výnosnost potravy tančících včel nalézané celým úlem je přepisována vždy po dokončení trasy každou tančící včelou a je dána předpisem

$$Pf_{colony} = \frac{1}{N_{Bee}} \sum_{i=1}^{N_{Bee}} Pf_i. \tag{2.18}$$

**Algoritmus 20** Optimalizace včelí kolonií

Inicializace populace $x_i$ pro $i = 1,2,\ldots,n$. ▷ Ideálně nejbližší soused viz algoritmus 3.
Inicializace $t \leftarrow 0$ a nejlepšího řešení $x^*$.
Inicializace tancujících včel. ▷ Volitelné.
**while** Není splněno ukončovací kritérium. **do**
    $t \leftarrow t + 1$
    **for** Každá sběrná včela $i = 1 : n$ **do**
        Cesta $x_i \leftarrow \{\}$.
        Včela $i$ hledá a vybírá cestu pomocí vzorců (2.15) a (2.14).
        Vylepšení cesty $x_i$ pomocí jednoho kroku 2-opt. ▷ Volitelné.
        **if** $f(x_i) < f(x_i^*)$ **then** ▷ Současné $i$ řešení je lepší než dosud nejlepší včely $i$.
            Aktualizujme nejlepší řešení včely $i$ pomocí zápisu $x_i^* \leftarrow x_i$.
            Včela $i$ vykonává Waggle dance podle vzorců (2.16), (2.17) a (2.18).
        **end if**
        **if** $f(x_i) < f(x^*)$ **then** ▷ Současné je lepší než nejlepší nalezené.
            Aktualizujme nejlepší řešení celého úlu pomocí vzorce $x^* \leftarrow x_i$.
        **end if**
    **end for**
**end while**

### 2.3.8 Algoritmus světlušek

Světluška je dalším zvířetem, které inspirovalo metaheuristické výpočty. V případě algoritmu světlušek (Firefly algorithm, FA, FFA) se základní myšlenka opírá o svítivost světlušek a jejich vzájemné přitažlivosti. Samčí světlušky totiž rytmicky svítí, aby přilákaly samičky. V tomto algoritmu podle (Kumbharana et al., 2013) se však uvažují bezpohlavní světlušky a přilákat se mohou libovolné světlušky. Přitažlivost se zakládá na síle vyzařovaného světla, a to je definováno pomocí hodnoty účelové funkce. FA lze zapsat viz algoritmus 21.

Speciální symbolika pro FA:

- ▷ $I_i \ldots$ svítivost světlušky $i$, kde platí $I_i \propto f(x_i)$, tzn. $I_i = \kappa \cdot f(x_i)$, kde $\kappa$ je konstanta,
- ▷ $r \ldots$ vzdálenost mezi dvěma řešeními dána HD nebo SD
- ▷ $r_{ij} \ldots$ vzdálenost mezi světluškou $i$ a světluškou $j$,
- ▷ $I_s \ldots$ intenzita zdroje světla,
- ▷ $I(r) \ldots$ světelná intenzita,
- ▷ $\gamma \ldots$ konstantní parametr koeficientu absorpce světla,
- ▷ $\beta(r) \ldots$ atraktivita mezi dvěma světluškami.

Mezi vstupní řešení je doporučeno vložit aspoň jedno efektivní řešení pomocí metody nejbližšího souseda nebo hladového algoritmu pro zrychlení výpočtu. Má to však i nevýhodu toho, že prostor přípustných řešení nemusí být kompletně prohledán.

Aproximace absorpce světla danou vzdáleností $r$ je dána gaussovskou formou

$$I(r) = I_0 \exp(-\gamma r^2), \tag{2.19}$$

kde $I_0$ je původní světelná intenzita.

Atraktivita světlušky je dána pomocí světelné intenzity okolních světlušek

$$\beta(r) = \beta_0 \exp(-\gamma r^2), \tag{2.20}$$

kde $\beta_0$ je atraktivita při vzdálenosti $r = 0$.

Jelikož se v postupu porovnávají dvě permutace pro zisk vzdálenosti $r$, je vhodné uvést, jak se vzdálenost mezi dvěma permutacemi získá. V obecném algoritmu se počítá s eulerovou vzdáleností, která zde však není vhodná. My použijeme následující dva přístupy k výpočtu vzdálenosti $r_{ij}$ na příkladu:

$$x_1 = (1, 2, 3, 4, 5, 6)$$
$$x_2 = (1, 2, 4, 3, 6, 5)$$

1. Hammingova vzdálenost (HD) nám dává informaci, jak moc se od sebe 2 řešení liší co se týče cesty a pořadí navštívení jednotlivých uzlů. Vzdálenost pro uvedený příklad je HD($x_1$,$x_2$) = 4, protože 3. až 6. pozice $x_1$ neodpovídá stejné pozici $x_2$. Pozice na 1. až 2. pozici je pak shodná.
2. Vzdálenost prohození tzv. „SwapDistance“ (SD) definujeme jako – kolik je potřeba provést prohození dvou sousedních uzlů v jedné cestě, aby byla stejná jako druhá. Vzdálenost pro uvedený příklad je SD($x_1$,$x_2$) = 2, protože pro $x_2$ stačí prohodit dvojice (4,3) a (6,5) a bude dosaženo stejné řešení jako $x_1$.

Čím menší jsou vzdálenosti mezi dvěma řešeními (světluškami), tím spíše se přesune horší z těchto řešení blíže k druhému lepšímu řešení (zářivější světlušce). Počet okopírovaných uzlů ze světlušky $j$ světluškou $i$ lze matematicky vyjádřit následovně

$$jadro_i \leftarrow random(2, r_{ij}), \tag{2.21}$$

kde $r_{ij}$ je vzdálenost mezi světluškou $i$ a světluškou $j$ dána pomocí HD nebo SD. Délka pohybu světlušky pak bude náhodně zvolena mezi 2 a hodnotou $r_{ij}$. Přesun světlušky se pak provede pomocí inverzní mutace, viz kapitola 2.3.5, kde délka jádra pro včelu $i$ je $jadro_i$ a doplnění náhodně dalšími uzly nebo zvolením pravidla pro opsání původního předpisu světlušky $i$.

**Algoritmus 21** Algoritmus světlušek

Inicializace parametrů, proměnných.
Inicializace populace náhodnými řešení $x_i (i = 1,2,\ldots,n)$.
Definování světelné intenzity $I_i$ v $x_i$ definována pomocí (2.19).
Definice koeficientu absorpce světla $\gamma$.
**while** $t < R$ **do**
    **for** $i = 1 : n$ světlušek **do**
        **for** $j = 1 : n$ světlušek **do**
            **if** $I_i < I_j$ **then**
                Přesuňme světlušku $i$ k světlušce $j$ podle (2.21).
            **end if**
            Atraktivita se vzdáleností $r$ pomocí (2.20).
            Vyhodnoťme nová řešení a aktualizujme světelnou intenzitu vzorcem (2.19).
        **end for**
    **end for**
    Ohodnoťme světlušky podle efektivity řešení. Nejlepší uložme jako $x^*$.
**end while**
Vraťme nejlepší nalezené řešení $x^*$.

### 2.3.9 Netopýří algoritmus

Netopýry inspirovaný algoritmus (Bat algorithm, BA) patří mezi algoritmy inspirované přírodou. Tentokrát je metaheuristika inspirována echolokačním systémem netopýrů. Netopýři při létání vysílají ultrazvukové vlny, aby lokalizovali překážky a potravu. Rozdíl rozeznají sami od sebe. Zároveň dokáže netopýr lokalizovat sám sebe a dokonce i další netopýry, kteří mohou lovit na úrodném místě. BA se tedy snaží napodobovat echolokační systém netopýrů a užít ho k prohledávání řešení. Původně byl navržen k řešení spojitých optimalizačních problémů, ale TSP je kombinatorický optimalizační problém, a BA tedy bude vyžadovat několik úprav.

V BA každý netopýr umí rozeznávat vzdálenosti $d_{ij}$ a má následující vlastnosti popsané speciální symbolikou:

- $v_b \ldots$ rychlost letu netopýra $b$,
- $A_b \in (A_{min},A_0) \ldots$ hlasitost netopýra $b$ k hledání kořisti, kde minimální hodnoty dosahuje v $A_{min} > 0$ a maximální vysoké nastavené hodnoty parametru $A_0 > 0$,
- $r_b \in (0,1) \ldots$ rychlost vysílání impulzů netopýra $b$,
- $\beta \in (0,1) \ldots$ náhodně generované číslo na uvedeném intervalu,
- $\square^t \ldots$ iterace libovolné proměnné v čase $t$,
- $x_b \ldots$ cesta reprezentovaná netopýrem $b$,
- $x^* \ldots$ dosud nejlepší nalezená cesta,
- $f(x_b) \ldots$ hodnota účelové funkce cesty $x$ netopýra $b$,
- $\alpha \ldots$ nastavitelný parametr upravující hlasitost $A_b$,
- $\gamma \ldots$ nastavitelný parametr upravující rychlost vysílání impulzů $r_b$.

Každý netopýr reprezentuje jedno přípustné řešení problému. Parametr frekvence $f_b$ pro náš diskrétní problém vyřadíme. Pokud by měl čtenář zájem o algoritmus, který frekvenci pro diskrétní problém obsahuje, pak viz (Saji et al., 2016).

Aktualizace rychlosti netopýra se provede za pomocí znalosti nejlepšího a současného řešení, mezi kterými se hledá Hammingova vzdálenost (funkce $HD$). Následně vybíráme náhodné číslo na intervalu od 1 do hodnoty Hammingovy vzdálenosti následovně:

$$v_b^t = Random[1,HD(x_b^t,x^*)] \tag{2.22}$$

Pohyb netopýra pro TSP předefinujeme na prohledávací heuristiku. Jak doporučují (Osaba et al., 2016) lze použít LK heuristiku buď 2-opt nebo 3-opt. Ty jsou uvedeny v kapitole 2.2.9. Pohyb netopýra (změna cesty) lze zapsat jako

$$x_b^t \leftarrow \text{2-opt}(x_b^{t-1},v_b^t), \tag{2.23}$$

$$x_b^t \leftarrow \text{3-opt}(x_b^{t-1},v_b^t), \tag{2.24}$$

kde 2–*opt* je LK heuristika 2-opt a obdobně 3-opt. V každé generaci $t$ netopýr prozkoumá $v_b$ sousedů a nejlepšího vybere jako aktuální pohyb. Tyto vzorce se používají i pro výpočet nového blízkého řešení k nejlepšímu nalezenému. Aby se rozlišilo použití obou přístupů, pak uvažujme $\frac{n}{2}$ a pokud $v_b^t$ je nižší, vykonáme menší změnu 2-opt a v opačném případě větší změnu 3-opt.

Při splnění podmínky $rand < A_b$ a $f(x_b) < f(x^*)$ ve for cyklu je potřeba aktualizovat parametry $A_b^{t+1}$ a $r_b^{t+1}$ následovně:

$$A_b^{t+1} = \alpha A_b^{t+1} \tag{2.25}$$

$$r_b^{t+1} = r_b^0[1 - \exp -\gamma t] \tag{2.26}$$

Abychom dokonvergovali k vhodným hodnotám pro $t \to \infty$, je vhodné nastavit parametry $0 < \alpha < 1$ a $0 < \gamma$. Tím získáme hlasitost $A_b^t \to 0$ a $r_b^t \to r_b^0$. Pro zjednodušení lze nastavit parametry $\alpha = \gamma$. Jak uvádí (Osaba et al., 2016) je vhodné zvolit parametry $\alpha = \gamma = (0{,}90; 0{,}99)$ a sami ve své práci došli k nejlépe fungující hodnotě parametru $\alpha = \gamma = 0{,}98$. Též uvádí upravený BA algoritmus 22 na diskrétní TSP problém.

V podmínkách na řádcích 13 a 17 se objevuje $\beta$. To je náhodně generované číslo mezi 0 a 1. Tím je zaručeno, že se netopýři chovají náhodně. Můžou a nemusí hledat lepší řešení. Zároveň se při pozdějších iteracích šance na změnu řešení snižuje.

**Algoritmus 22** Netopýří algoritmus

Definujme účelovou funkci a počet netopýrů $n$.
Inicializujme populaci netopýrů $X = x_1,x_2,\ldots,x_n$
**for** $b = 1 : n$ **do** ▷ Pro každého netopýra.
    Definujme rychlost impulzů $r_b$, rychlost netopýra $v_b$, hlasitost $A_b$.
**end for**
**while** Není splněno ukončovací kritérium. **do**
    **for** $b = 1 : n$ **do** ▷ Pro každého netopýra v populaci.
        Vygenerujme nové náhodné číslo $\beta$.
        Spočítejme novou rychlost $v_b^t$ pomocí (2.22).
        **if** $v_b^t < \frac{n}{2}$ **then** ▷ Malý krok.
            Generujme nové řešení pomocí 2-opt (2.23)..
        **else**
            Generujme nové řešení pomocí 3-opt (2.24).
        **end if**
        **if** $\beta > r_b$ **then**
            Nalezněme nejlepší řešení $f(x^*)$.
            Generujme nové řešení okolo $f(x^*)$ pomocí 2-opt nebo 3-opt.
        **end if**
        **if** $\beta < A_b$ a zároveň $f(x_b) < f(x^*)$ **then**
            Přijměme nové řešení.
            Snižme $A_b$ pomocí (2.25) a zvyšme $r_b$ pomocí (2.26).
        **end if**
    **end for**
**end while**

### 2.3.10 Optimalizace fyziologie stromů

Optimalizace fyziologie stromů (Tree Physiology Optimization, TPO) je založena na fyziologickém systému růstu rostlin a je určena k prohledávání spojitých problémů. TSP není spojitý problém, ale diskrétní, tudíž vyžaduje několik úprav. Takový algoritmus pak můžeme podle (Halim et al., 2013) a (Halim et al., 2019) zapsat následovně.

Uvažujme kmen, na který navazujeme výhonky (shoots, $s$), které jsou pro připodobnění k metaheuristikám založeným na bázi roje zvířat agenty (mravenci, včely, apod.). Pro prohledávání po celém spektru přípustných řešení se používají větve (branch, $br$), abychom se mohli vyhnout lokálnímu minimu. Každá větev obsahuje určený počet výhonků.

Výhonky můžeme klasifikovat do několika kategorií:

- ▷ kmeny,
- ▷ větve,
- ▷ listy.

Základem každé rostliny jsou kořeny (roots), na kterých mohou výhonky narůstat. Jejich hlavním cílem je získávat vodu a další živiny pro rostlinu. To lze definovat jako náhodné vyhledávání řízené účelovou funkcí.

Speciální symbolika pro TPO:

- ▷ $s_i \dots$ hodnota výhonku $i$, permutace cesty pro Hamiltonův cyklus,
- ▷ $s_{gbest} \dots$ globální nejlepší výhonek (řešení),
- ▷ $N_j \dots$ nutriční hodnoty řešení $j$
- ▷ $C \dots$ uhlík (carbon), živiny, které získá rostlina navíc danými výhonky,

Jako první výpočet v diskrétním TPO je prodlužování výhonků směrem ke světlu. Každý nový list je brán jako nové řešení a větve jako jednotlivé populační výsledky. Výhonky $S_i$ definujeme jako

$$S_i = \begin{cases} s_{i_j}, & s_{i_j} \neq s_{gbest_j}, s_{i_j} = N_j, \\ s_{gbest}, & s_j = s_{gbest_j}, \end{cases} \tag{2.27}$$

kde $i$ představuje současnou populaci a $j$ současný uzel.

Vyhodnocení jednotlivých výhonků se počítá pomocí počítadla, které se používá na zjištění celkového zisku uhlíku (carbon, $C$) následovně:

$$C = n\lfloor \%count \rfloor \tag{2.28}$$

To je následně použito k přepočtu prodloužení kořenů.

$$r_{ij} = \begin{cases} s_{i_j}(SL(C+1;n)), & i \geq C \\ s_{i_j}, & i < C, \end{cases} \tag{2.29}$$

kde $SL$ je funkce jednokrokového „slide“ operátora. Např. pro $C = 5$ a $n = 10$ operátor $SL$ říká, že je 2% rozdíl mezi současným a globálním optimem.

**Algoritmus 23** Optimalizace fyziologie stromů – TPO

1: Inicializace populace $n$, větve $br$.
2: **for** $i = 1 : R$ **do**
3: Vyhodnoťme celkovou vzdálenost.
4: Určeme nejlepší řešení $S_{gbest}$
5: **for** $p = 1 : br$ **do**
6: Vyhodnoťme nejlepší řešení pro každou větev $S_{popbest}$.
7: Porovnejme $S_{popbest}$ pro nalezení nejlepšího řešení, $S_{gbest}$.
8: Rozšiřující výhonky podle vzorců (2.27) a (2.28).
9: Prodlužování kořenů podle (2.29).
10: flip $r_{ij}$ pomocí $C$.
11: **end for**
12: **end for**

### 2.3.11 Další metaheuristiky

Optimalizace roje částic (Partical swarm optimization, PSO) je inspirována pozorováním rojů zvířat. Jedná se například o ptáky, ryby nebo dokonce i lidské sociální chování. Základní myšlenkou je roj částic a prohledávání přípustných řešení. Chaotické prohledávání přípustných řešení pak řeší algoritmus chaotické optimalizace (Chaotic optimization algorithm, CAO). Pro výpočet je klíčová rychlost a pozice. Základní algoritmus PSO řeší spojité problémy a pro aplikaci na TSP je potřeba převést na diskrétní problém.

Jako další zajímavý algoritmus představujeme inteligentní kapky vody (Intelligent water drops, IWD). Spadá do kategorie metaheuristik inspirovaných přírodou. Inspirace spočívá v přírodních systémech řek, jezer, moří a oceánů a zároveň zákonech akce a reakce mezi jednotlivými kapkami vody. Jak je přirozené, voda teče z vyšších míst do níže položených, kde tvoří řeky a z nich větší uskupení vod (jezera etc.). Jak řeka teče, prohledává terén a nalézá nejvhodnější místa toku, což vytváří zatáčky, meandry nebo i ostrovy. Cílem řeky však není nalézt nejkratší tok, ale nejvhodnější místa, kudy téct. Časem se tok řek také mění. V optimalizaci se pak počítá s rychlostí řeky, která je při dokončení trasy nulová a obsahem zeminy, kterou přinesla s sebou do cílové destinace (např. moře).

Myšlenka vyhledávání harmonie (Harmony search, HS) pochází z oblasti umění – hudby. Snaha o dosažení harmonie se dokládá až do doby starého Řecka. Hudebníci při ladění různých nástrojů hledají optimální výšku a tóninu, což se v rámci této metody označuje jako fantastická harmonie. Důležitým prvkem pak je paměť harmonie a improvizace, která umožňuje prohledat větší škálu řešení právě díky paměti o předchozích řešení. Víceúčelové užití algoritmu viz (Geem et al., 2001).

Metoda elektromagnetismu (Electromagnetism-like method, EM) dle (Birbil et al., 2004) funguje na principech fyzikálních zákonů. Hlavním prvkem je síla přitahování a odpuzování, jako tomu je u běžných magnetů založených na Coulombově zákonu. Využívá také lokálního hledání pro optimalizování řešení.

Předposlední předloženou metaheuristikou je algoritmus umělých imunitních systémů (Artificial immune systems, AIS), který zrekapitulovali (Bonyadi et al., 2008). AIS je založen na základním lidském štítu proti chorobám, vetřelcům a změnám v těle – imunitě. Imunita pracuje systematicky na základních třech krocích – rozpoznání, kategorizaci a obraně. Při rozpoznání rozezná, o jaký druh problému se jedná, zařadí ho do kategorie a podle toho nastolí obranu (léčbu). Zároveň si imunita pamatuje vetřelce, se kterými se již vypořádala, a nebo může být připravena na vetřelce díky očkování. Algoritmus jako takový pak velmi připomíná GA, protože při nastolení léčby se využívá genetických pravidel. Jednotlivé operace se však provádí jinak.

Závěrem bychom rádi zmínili algoritmus náhodného klíče (Random key, RK), který lze použít pro transformaci spojitých metaheuristik na diskrétní metaheuristiky. Hlavní myšlenka spočívá v hledání náhodných vzorů v již předložených řešeních. Používá se například v kombinaci

s GA viz (Snyder et al., 2006).

Následuje výčet dalších metaheuristik použitelných pro výpočet TSP. Již je pouze zmíníme pro snahu vytvořit kompletní výčet metaheuristických procedur (a algoritmů).

- ▷ Optimalizace mušky octomilky (Fruit fly Optimization Algorithm, FOA)
- ▷ Algoritmus optimalizace velryb (Whale Optimization Algorithm, WOA)
- ▷ Cuckoovo prohledávání (Cuckoo search, CS)
- ▷ Algoritmus bakterií vyhledávajících potravu (Bacterial foraging algorithm, BFA)
- ▷ Umělá kolonie včel (Artificial bee colony, ABC) $\approx$ BCO
- ▷ Genetické programování (Genetic programming, GP)
- ▷ Diferenciální vývoj (Differential evolution, DE) – v kombinaci s GA
- ▷ Hooke–Jeeves (HJ)

# 3. Numerické experimenty a jejich vyhodnocení

V praktické části se zabýváme aplikací jednotlivých heuristik a metaheuristik v prostředí jazyku R. Zajímá nás aplikovatelnost jednotlivých přístupů v jazyce R. Dalším aspektem k pozorování je efektivita jednotlivých přístupů v řešení TSP. Nakonec budeme ověřovat vhodnost metod na řešení 3D TSP.

## 3.1 Prostředí R

Jazyk R byl vyvinut z jazyka S. Je používán převážně pro statistickou analýzu a grafickou vizualizaci, ale lze ho použít univerzálně. Je dostupný pro širokou veřejnost, stejně tak jako RStudio, které lze použít jako grafické rozhraní pro práci s jazykem R. Jednotlivá rozšíření jsou v podobě balíčků ukládána na CRAN (Comprehensive R Archive Network), což je veřejně dostupná databáze, ze které lze volně stahovat vytvořená rozšíření pro jazyk R. Existuje i provázanost s jazykově univerzálním úložištěm GitHub. Více informací lze dočíst viz (Ihaka et al., 1996).

Jazyk R má své nevýhody a jednou z nich je pomalejší zpracovávání smyček (cyklů). Nicméně pokud je importovaná funkce napsána na základě jiného jazyka (např. C++), nemusí tímto problémem trpět. Zároveň můžeme některé výpočty provádět na více jádrech počítače a tím urychlit řešení úloh pomocí balíčku „parallel“.

Jednou z využitých výhod je i v této práci provázanost s LaTeXem, ve kterém je tato práce psána. Lze si vygenerovat kódový předpis tabulek, který pak už jen stačí vložit do souboru „.tex“ nebo vytváří přímo latexové soubory.

Komentáře se v jazyce R aktivují symbolem „#“ a jakýkoli příkaz nebo text za tímto symbolem se na dané řádce nevykoná.

Oblíbenými alternativami pro jazyk R jsou Python, C++, Java, Pascal nebo MATLAB.

## 3.2 Vzdálenosti v R

K výpočtu vzdálenosti ve 3D lze přistoupit několika způsoby podobně jako ve 2D. Nejjednodušší a nejlogičtější se zdá být použití Euklidovy metriky, která měří vzdálenost jako přímku mezi dvěma body. Podobně uvažovali v práci (Cook, 2022b) a doporučili vzorec

$$c_{ij} = \sqrt{(X_i - X_j)^2 + (Y_i - Y_j)^2 + (Z_i - Z_j)^2}, \tag{3.1}$$

který nyní stačí pouze zaokrouhlit na celá čísla pro úsporu paměti (použije se po zápis hodnot $c_{ij}$ integer, nikoliv double).

V prostředí R lze buď vypočítat vzorcem (3.1) nebo vestavěnou funkcí „dist“ viz kód,

```
dist(data, method = "euclidean", diag = TRUE, upper = TRUE, p = 2)
```

kde `data` představují souřadnice jednotlivých hvězd. Zde je potřeba dát si pozor, protože funkce `dist` vrací objekt třídy dist. Detailnější popis viz dokumentace (RDocumntation, 2022).

Funkce pro výpočet vzorce (3.1) se dá zapsat v jazyce R následovně,

```
vzdalenost <- function(data){
  distance <- data.frame()
  n <- as.numeric(nrow(data))
  for(i in 1:n){
    for(j in 1:n){
      x = data[i,1] - data[j,1]
      y = data[i,2] - data[j,2]
      z = data[i,3] - data[j,3]
      distance[i,j] <- round(sqrt(x^2 + y^2 + z^2))
    }
  }
  rownames(distance) <- c(1:n)
  colnames(distance) <- c(1:n)
  return(distance)
}
```

kde vstupní položka `data` je typu `data.frame()` a obsahuje 3 sloupce se zápisy jednotlivých souřadnic po řádcích, jak je uvedeno např. v tabulce 3.1. Do `distance` je pak vložen data frame, který je postupně pomocí `for` cyklů plněn hodnotami vzdáleností zaokrouhlenými na celá čísla. Nakonec pak jsou ještě přiřazena jména řádků a sloupců pořadovými čísly. Výstup funkce pak z hvězd uvedených v tabulce 3.1 vytvoří matici vzdáleností jako je v tabulce 3.2.

Máme k dispozici celkem 3 datasety,

- „star100_xyz.txt“ se 100 hvězd (malý dataset),
- „star1k_xyz.txt“ s 1 000 hvězd (středně velký dataset),
- „star10k_xyz.txt“ s 10 000 hvězd (velký dataset).

ze kterých si budeme vybírat množství hvězd dle potřeby.

Tabulka 3.1: Ukázka souřadnic prvních šesti hvězd malého datasetu.

| i | X | Y | Z |
|---|---|---|---|
| 1 | 0,00000 | 0,00000 | 0,00000 |
| 2 | −4,95181 | −4,13973 | −11,56674 |
| 3 | −4,95203 | −4,14084 | −11,56625 |
| 4 | −4,72264 | −3,61451 | −11,51219 |
| 5 | −0,17373 | −18,16613 | 1,49123 |
| 6 | −19,98000 | 5,04305 | 14,95504 |

Tabulka 3.2: Ukázka matice vzdáleností prvních šesti hvězd malého datasetu.

| i | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 13 | 13 | 13 | 18 | 25 |
| 2 | 13 | 0 | 0 | 1 | 20 | 32 |
| 3 | 13 | 0 | 0 | 1 | 20 | 32 |
| 4 | 13 | 1 | 1 | 0 | 20 | 32 |
| 5 | 18 | 20 | 20 | 20 | 0 | 33 |
| 6 | 25 | 32 | 32 | 32 | 33 | 0 |

## 3.3 Grafické zobrazení řešení TSP ve 3D v R

Pro implementaci grafického zobrazení cest mezi hvězdami ve 3D prostoru máme na výběr ze dvou balíčků:

1. scatterplot3d
2. rgl

Výhodou balíčku scatterplot3d je jednodušší vykreslování a vkládání do výsledné práce. Výraznou nevýhodou je, že jakmile dojde ke křížení cest ve 3D prostoru, pak ve 2D pohledu je výsledek velmi matoucí. Naopak právě balíček rgl se špatně vkládá do výsledné práce a vykreslovací funkce zanechává diagonály na stěnách jednotlivých prostorů, což nepůsobí dobře. Naopak si dokáže velmi dobře poradit s křížením cest a velkým množstvím bodů. V rámci rozhraní R pak působí velmi dobře a přehledně se s ním kontroluje vykreslování jednotlivých řešení TSP.

### 3.3.1 Balíček scatterplot3d

Balíček „scatterplot3d“ viz (Ligges et al., 2003) převádí 3D grafy do 2D grafů. Graf je statický a nelze si ho otočit. Nicméně je vhodný pro porovnání cest mezi sebou, protože vždy bere

stejný úhel pohledu na graf.

Použité funkce pro vykreslování grafu

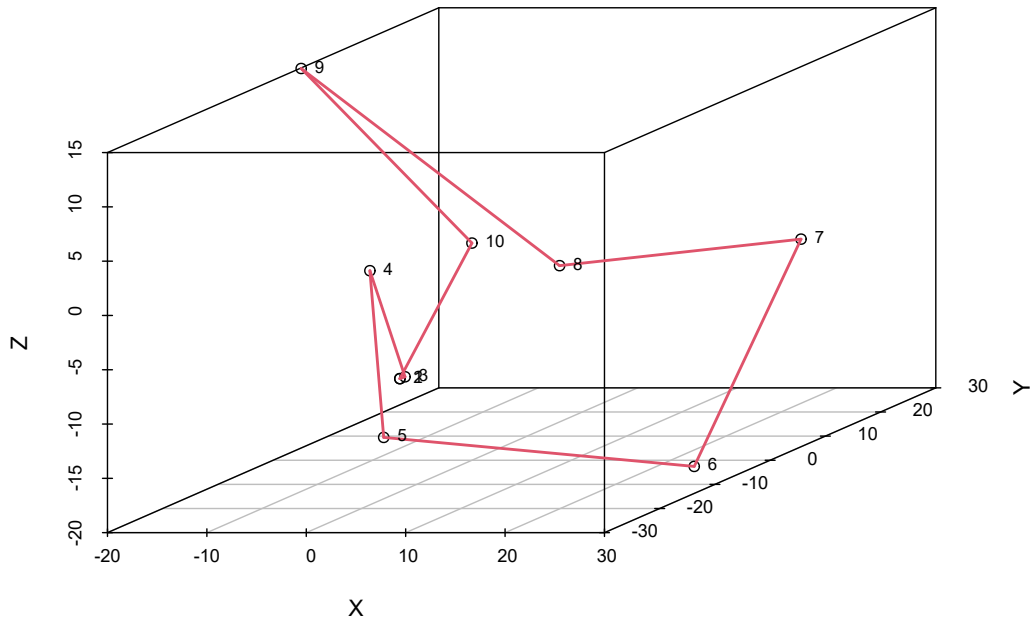
- ▷ `scatterplot3d()` ... zobrazí celý graf i s jednotlivými body,
- ▷ `xyz.convert()` ... převede 3D rozměry na 2D rozměry,
- ▷ `segments()` ... přidá úsečku mezi 2 body (cestu z $i$ do $j$ uzlu),
- ▷ `text()` ... přidá popisek k bodům nebo segmentům na grafu.

Funkce v jazyce R, která vykresluje 3D cestu TSP problému, lze zapsat následovně. V rámci přiložených R skriptů lze dohledat i další implementace.

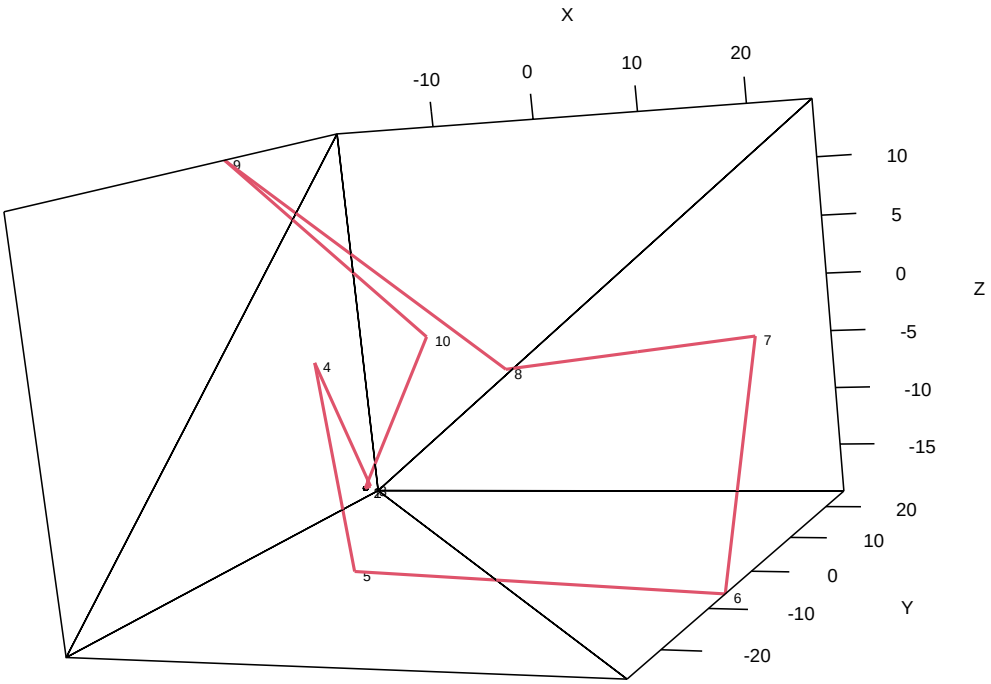
```
graf_staticky3 <- function(data, tour){
  n <- as.numeric(nrow(data))
  data2 <- data[tour,]
  obraz <- scatterplot3d(data2,
                         xlab = "X", ylab = "Y", zlab = "Z")
  body <- sapply(1:(length(tour)),
                 function(i){obraz$xyz.convert(data2[i,])})
  body2 <- data.frame(x = unlist(body["x",]),
                      y = unlist(body["y",]))
  t1 <- body2[1:(length(tour)),]
  t2 <- body2[c(2:(length(tour)),1),]
  segments(t1$x, t1$y,
           t2$x, t2$y,
           lwd=2,col=2)
  text(t1$x,
       t1$y,
       labels = (1:n),
       cex = .5,
       pos = 4)
}
```

Do `n` byl uložen rozměr úlohy. Do `obraz` se ukládá graf. Do `data2` ukládáme vstupní soubor dat seřazený podle navštívení dle dané cesty (tour). Do `body` je uložená konverze 3D bodů do 2D prostoru. Do `body2` je pak otevřen zápis `body`. Do `t1` se vkládá seznam bodů, ze kterých se vychází v rámci cesty, a obdobně do `t2` vkládáme doby, do kterých se vchází v rámci cesty. Kombinace `t1_i` a `t2_i` pak vytváří přechod z hvězdy $i$ do hvězdy $j$. Funkcí `segments()` pak právě pospojujeme body spárované body $i$ a $j$ (na stejné řádce) úsečkou. Posledním příkazem s `text()` vytváříme popisky pro jednotlivé hvězdy a to tak, že je očíslujeme podle pořadí, jak byly navštíveny.

Názorná ukázka zpracovaného výsledku TSP (červená) pomocí balíčku scatterplot3d viz obrázek 3.1.




Obrázek 3.1: Ukázkové grafické zobrazení po konverzi pro TSP z 3D do 2D prostoru pomocí balíčku scatterplot3d.




Obrázek 3.2: Ukázkový grafické zobrazení 3D grafu po drobném natočení podle balíčku rgl.

### 3.3.2 Balíček rgl

Balíček „rgl“ vit (Murdoch et al., 2022) zobrazí 3D graf v samostatném 3D okně. Graf je také statický, ale lze si s ním otáčet pro jiné úhly pohledu, což umožňuje rozeznat komplikovanější zobrazení. Pokud však chceme grafy rozeznávat mezi sebou, měli bychom nastavit stejný úhel pohledu.

Použité funkce k vykreslování grafu

- ▷ `plot3d()` ... zobrazí celý graf i s jednotlivými body,
- ▷ `open3d()` ... otevře nový prázdný graf,
- ▷ `points3d()` ... zobrazí jednotlivé body na grafu,
- ▷ `segments3d()` ... přidá úsečku mezi 2 body (cestu z $i$ do $j$ uzlu),
- ▷ `texts3d()` ... přidá popisek k bodům nebo segmentům na grafu,
- ▷ `axes3d()` ... přidá popisky os jednotlivých dimenzí.

Na zápis funkce, která vykresluje graf TSP ve 3D, v jazyce R viz kód níže,

```
graf_dynamicky2 <- function(data, tour){
  n <- as.numeric(nrow(data))
  data2 <- data[tour,]
  obraz <- open3d
  points3d(x = data2[,1],
           y = data2[,2],
           z = data2[,3])
  for(i in 1:(n-1)){
    segments3d(x=as.vector(t(data2[c(i,i+1),1])),
               y=as.vector(t(data2[c(i,i+1),2])),
               z=as.vector(t(data2[c(i,i+1),3])),
               col=2,lwd=2)
  }
  segments3d(x=as.vector(t(data2[c(n,1),1])),
             y=as.vector(t(data2[c(n,1),2])),
             z=as.vector(t(data2[c(n,1),3])),
             col=2,lwd=2)
  texts3d(x = data2[,1],y = data2[,2], z = data2[,3],
          text = (1:n), cex = .75, pos = 4)
  axes3d()
  title3d(xlab="X", ylab="Y", zlab="Z")
}
```

kde do `n` ukládáme rozměr úlohy. Do `data2` ukládáme data seřazená podle toho, jak jsou navštěvována řešením TSP. Do `obraz` se ukládá graf. Dále užíváme funkce `points3d()` k vykreslení všech bodů. Ve for cyklu pak vykreslujeme jednotlivé úsečky (kromě poslední) pomocí funkce `segments3d()`. Mimo cyklus užijeme stejnou funkci pro návrat z posledního do star-

tovního uzlu. Nyní již jen přidáme popisky jednotlivých hvězd v pořadí jejich navštívení pomocí `texts3d()`, popisky os pomocí `axes3d()` a přejmenujeme je díky funkci `title3d()`.

Ukázka výstupu TSP řešení (červená) pomocí balíčku rgl viz obrázek 3.2.

## 3.4 TSP funkce implementované v R

V rámci jazyka R a databáze CRANu existuje balíček TSP viz (Hahsler et al., 2022). S jeho pomocí lze řešit úlohy TSP, ovšem paleta možných řešení je poněkud omezená. Jako argument metody k řešení lze vybrat z následujících:

- "identity"... vrátí cestu v pořadí jak dle matice vzdáleností,
- "random"... vrátí náhodně vytvořenou cestu
- "nearest_insertion"... nejbližší vkládání,
- "cheapest_insertion"... nejlevnější vkládání,
- "farthest_insertion"... nejvzdálenější vkládání,
- "arbitrary_insertion"... libovolné vkládání,
- "nn"... algoritmus nejbližšího souseda,
- "repetitive_nn"... algoritmus nejbližšího souseda, který je vykonán pro každý uzel jako startovní a z nich je vybráno nejlepší řešení,
- "two_opt"... metoda 2-opt Lin–Kerninghen

Vkládací (insertion) verze algoritmů pracují s výpočetním vzorcem (3.2), kde hledáme uzel $k$ mezi uzly $i$ a $j$ tak, aby byl splněn požadavek dle dané verze. Vzorec pochází z myšlenek kapitoly 2.2.8 a zní

$$d(i,k) + d(k,j) - d(i,j). \tag{3.2}$$

Použití balíčku TSP na vstupní matici vzdáleností deseti hvězd viz kód níže.

```
etsp10  <- as.ETSP(dist10)
tour10  <- solve_TSP(etsp10, method = "nn")
tour10 <- list(Z = tour_length(tour10),
               cesta = as.integer(names(tour10)))
```

Nejprve vyžaduje funkce `solve_TSP` konverzi vložených vzdáleností do třídy ETSP pomocí funkce `as.ETSP`. Nyní můžeme problém vyřešit pomocí `solve_TSP`. Jako metodu jsme zvolili například `"nn"`, což je metoda nejbližšího souseda. Po výpočtu následně pomocí funkcí `tour_length()` a `names()` vytáhneme informace o délce trasy TSP a výslednou Hamiltonovu cestu ve formě permutace uzlů. Tu ukládáme jako list, abychom s výsledkem mohli pracovat ve stejné podobě jako s řešeními, které aplikujeme v nadcházejících kapitolách.

## 3.5 Pomocné funkce pro řešení TSP v R

Jelikož se v některých heuristikách a metaheuristikách opakují některé námi vytvořené funkce, rozhodli jsme se pro jejich vysvětlení před jejich použitími. Jedná se o dvě funkce.

Pro výpočet náhodného řešení používáme funkce `NahodReseni`. Vstupem je matice vzdáleností jednotlivých uzlů `vzdalenosti`.

```
NahodReseni <- function(vzdalenosti){
  n <- dim(vzdalenosti)[1]
  cesta <- sample(1:n,n)
  Z <- 0
  for(i in 1:(n-1)){
    Z <- Z + vzdalenosti[cesta[i],cesta[i+1]]
  }
  Z <- Z + vzdalenosti[cesta[n],cesta[1]]
  return(list(Z = Z, cesta = cesta))
}
```

Náhodný výběr jednotlivých uzlů pak provádíme pomocí funkce `sample()`, kterou náhodně (rovnoměrné šance) vybereme všechny uzly. Nakonec spočítáme hodnotu účelové funkce `Z` ve `for` cyklu a vracíme řešení TSP.

Druhá užitečná funkce je `VypoctiZtka`. Do ní vstupuje seznam řešení libovolného množství `cesty` a matice vzdálenosti jednotlivých uzlů `vzdalenosti`.

```
VypoctiZtka <- function(cesty, vzdalenosti){
  n <- dim(vzdalenosti)[2]
  cesty <- matrix(cesty, ncol = n)
  Ztka <- numeric(dim(cesty)[1])
  for(j in 1:dim(cesty)[1]){
    cesta <- cesty[j,]
    Ztka[j] = 0
    for(i in 1:(n-1)){
      Ztka[j] <- Ztka[j] + vzdalenosti[cesta[i],cesta[i+1]]
    }
    Ztka[j] <- Ztka[j] + vzdalenosti[cesta[n],cesta[1]]
  }
  return(Ztka)
}
```

Funkce je napsána tak, že spočítá řešení pro libovolné množství cest. Proto si i cesty upravíme na matici, aby se nám s ní lépe pracovalo. Pomocí funkce `dim()[1]` určujeme, kolik řešení jsme vlastně k výpočtu poslali, a počítáme tolik hodnot účelových funkcí. Jakmile máme hotovo, vracíme číselné hodnoty `Ztka`.

## 3.6 Heuristiky v R

Z teoretické části jsme k implementaci vybrali následující heuristiky – nejbližší soused, hladový algoritmus, algoritmus výhodnostních čísel, algoritmus minimální kostry a 2-opt LK heuristiku. První 4 zmíněné nalézají přípustné řešení, kdežto poslední zmíněná prohledává okolí vloženého řešení a snaží se nalézt lepší řešení.

### 3.6.1 Nejbližší soused v R

Heuristický postup metody nejbližšího souseda jsme implementovali následovně.

```
NejSoused <- function(vzdalenost, start = 1){
  n <- dim(vzdalenost)[1]
  zbyva <- setdiff(1:n, start)
  cesta <- NULL
  cesta <- start
  Z <- 0
  i <- 1
  pozice <- start

  while(i < n){
    i <- i + 1
    vzd <- vzdalenost[pozice, zbyva]
    Z <- Z + min(vzd)
    novapozice <- zbyva[which.min(vzd)]
    pozice <- novapozice
    cesta <- c(cesta, novapozice)
    zbyva <- setdiff(zbyva, pozice)
  }
  Z <- Z + vzdalenost[pozice, start]
  return(list(Z = Z, cesta = cesta))
}
```

Vstupem funkce `NejSoused` jsou matice vzdáleností `vzdalenost` a druhotný parametr `start`, který udává v jakém uzlu má algoritmus začít. Pokud není uvedeno jinak, pak je nastaven na první uzel. Do `n` ukládáme celkové množství uzlů k navštívení. Do parametru `zbyva` ukládáme pomocí funkce `setdiff()` uzly, které ještě nebyly navštíveny. To provádíme i v hlavním cyklu. Inicializujeme proměnné `cesta` (posloupnost navštívených uzlů), `Z` (hodnota účelové funkce) a `i` (index současného uzlu k navštívení). Do `pozice` ukládáme poslední navštívený uzel. Jinými slovy, v další iteraci se jedná o uzel, ze kterého budeme vycházet.

V hlavním cyklu pak procházíme jednotlivé inicializované proměnné a měníme jejich hodnoty. Navyšujeme hodnotu `i`. Nalézáme minimální hodnotu mezi vzdálenostmi, kterou přidáváme do hodnoty účelové funkce `Z`. Pomocí funkce `which.min()` nalézáme index minimální hrany,

kterou máme k dispozici pro vytvoření cesty, a uložíme ho do `novapozice`. Aktualizujeme `pozice` pro další iteraci (v této iteraci po provedení cesty z $i$ do $j$ se aktualizuje index $i$ na hodnotu $j$ pro další iteraci). Do `cesta` přidáváme nově navštívený uzel `novapozice` v současné iteraci. Nyní musíme oříznout `zbyva` o uzel, který již nesmí být navštíven, a to je uzel, který jsme právě navštívili.

Po ukončení hlavního cyklu nezapomeneme do hodnoty účelové funkce přidat návrat na startovní pozici a vrátíme výsledky výpočtu.

### 3.6.2 Hladový algoritmus v R

Pro délku Hladového algoritmu uvádíme kód v příloze A v kapitole A.1. Pro jeho použití potřebujeme aktivovat balíček reshape2 pomocí příkazu `library(reshape2)`.

V první části kódu připravujeme inicializaci algoritmu. Tentokrát do `cesta` však ukládáme hrany a ne jednotlivé uzly. Do `vzdal_sort` ukládáme všechny dostupné hrany, které můžeme použít pro výběr hrany. Tento seznam je seřazen od nejnižší vzdálenosti po nejvyšší pomocí funkce `order()`. Pro inicializaci úlohy máme k dispozici celkem $n^2 - n$ hran.

V hlavním `while()` cyklu vybíráme nejkratší hranu k dispozici a označujeme ji jako `mini`. To děláme dokud nepoužijeme celkem $n$ hran. Z ní si vytáhneme indexy `i` a `j`. S jejich pomocí kontrolujeme, zda je hrana přípustná pro řešení či nikoliv. To určujeme pomocí stupňů uzlů. Pokud `stupen[i]` a zároveň `stupen[j]` jsou rovny dvěma, pak je nová hrana navázána z obou stran na další hrany, což může vytvořit parciální cyklus a vyžaduje kontrolu.

V dílčím `while()` cyklu pak kontrolujeme hlavu `head` a ocas `tail`. Po navázaných hranách se totiž posouváme po hlavičce dále po cestě. Pokud se stane, že hlava narazí na ocas, pak byl vytvořen parciální cyklus a musíme danou hranu vyřadit ze `vzdal_sort`. To kontrolujeme pomocí `continue`. Pokud je pravda (`TRUE`), pak nebyla propojena hlavička s ocasem. Druhá kontrolní proměnná `continue2` s hodnotou `TRUE` umožňuje zápis hrany do `cesta`.

Pokud však hrana nevytvořila parciální cyklus, vkládáme ji do cesty a aktualizujeme hodnotu účelové funkce. Zároveň zkrátíme seznam hran `vzdal_sort` o některé nepoužitelné hrany, abychom ušetřili výpočetní čas při marných pokusech hledání vhodné hrany.

Nakonec už jen seřadíme hrany od startovního po konečný uzel a nezapomeneme přičíst do účelové funkce hranu s návratem do startovního uzlu.

### 3.6.3 Výhodnostní čísla v R

Kód algoritmu výhodnostních čísel uvádíme pro jeho délku v příloze A. Velmi se podobá kódu hladového algoritmu – uvedeme tedy změny, které obsahuje.

Místo vzdáleností se pracuje s výhodnostními čísly, které jsou dané maticí `S`, kterou počítáme

vzorcem (2.7) a pokud nemá být vypočítána, dáváme hodnotu $-\infty$, podle které vyřazujeme neexistující hodnoty. Podle této matice pak jsou řazeny hrany sestupně (od nejvýhodnější po nejméně výhodnou). Hrany jsou pak uloženy do seznamu `S_sort`, který je alternativou `vzdal_sort` z hladového algoritmu. Má však méně hran – celkem $n^2 - 3n + 2$ hran. Vybíráme hranu od nejvýhodnějších (s nejvyšší hodnotou) a tu ukládáme do `maxi`. Oproti hladovému algoritmu však počítáme v hlavním cyklu o jednu iteraci méně, $n - 1$ krát, protože pro první uzel nepočítáme výhodnostní čísla a přiřazujeme ho do cesty až po ukončení hlavní smyčky.

### 3.6.4 Metoda minimální kostry v R

Kód vyžaduje aktivaci balíčků `library(igraph)` pro nalezení minimální kostry grafu a pro vytvoření seznamu hran a vzdáleností z matice vzdáleností balíček `library(reshape2)`. Dále potřebujeme načíst funkce `SeradEuler` ze souboru „SeradEuler.R“ a funkci `VypoctiZtka` ze souboru „Vypocti_Z.R“.

```
MetMinKostra <- function(vzdalenosti){
  n <- dim(vzdalenosti)[1]
  pomoc <- data.frame(as.matrix(vzdalenosti) + diag(Inf, nrow = n, ncol = n))
  pomoc[upper.tri(pomoc)] <- Inf
  colnames(pomoc) <- 1:n
  vzdal_sort <- subset(melt(as.matrix(pomoc)), value!=Inf)
  vzdal_sort <- vzdal_sort[order(vzdal_sort$value),1:3]
  rownames(vzdal_sort) <- 1:((n^2-n)/2)

  grafik <- graph_from_data_frame(vzdal_sort)
  MST <- as_data_frame(minimum.spanning.tree(grafik, weights = vzdal_sort$value))
  MST2 <- MST[,c(2,1,3)]
  colnames(MST2) <- colnames(MST)
  double_hrany <- rbind(MST, MST2)

  euler <- SeradEuler(double_hrany)
  cesta <- as.numeric(unique(euler$from))
  Z <- VypoctiZtka(cesta, vzdalenosti)
  return(list(Z = Z, cesta = cesta))
}
```

V první části kódu vyřazujeme horní trojúhelník matice vzdáleností i s diagonálou pro tvorbu matice `pomoc`. Matici `pomoc` pak přepisujeme na tabulku `vzdal_sort`, která obsahuje seznam hran i se vzdáleností. K tomu používáme funkce `melt` z „reshape2“.

Hrany `vzdal_sort` pak vstupují do funkce `graph_from_data_frame` pro vytvoření grafu `grafik`. Ten pak vstupuje jako argument funkce `minimum.spanning.tree()` pro vytvoření minimální kostry grafu. Do funkce výpočtu minimální kostry vkládáme vážené hrany a me-

toda volí použití Jarníkova algoritmu (Prim's algorithm). Hrany z minimální kostry pak zdvojíme do proměnné `double_hrany`.

Ze zdvojených hran vytváříme Eulerův cyklus pomocí námi vytvořené funkce `SeradEuler()`. Vrátil se nám seznam hran, ze kterého již však vyřadíme všechny duplicitní uzly kromě prvních uvedených v seznamu. K tomu používáme funkce `unique()`. Nyní už jen musíme spočítat hodnotu účelové funkce pomocí námi vytvořené funkce `VypoctiZtka`.

### 3.6.5 2-opt Lin Kerninghen v R

V námi provedené implementaci heuristiky LK 2-opt používáme drobné úpravy původního algoritmu. Místo toho, abychom posouvali indexy, posouváme vždy zápis cesty o jedna vlevo a z první pozice na pozici poslední. Tzn. ze zápisu cesty $x = (1,2,3,4,5)$ vytvoříme cestu $x' = (2,3,4,5,1)$.

```
LK_2_opt <- function(vzdal, cesta, Z){
  i = 1
  j = 3
  neobjel_z5 = TRUE
  posunuta_cesta = cesta
  posl_i <- 1
  posl_j <- 3
  n <- dim(vzdal)[1]

  while(neobjel_z5){
    delta <- vzdal[posunuta_cesta[1],posunuta_cesta[j]] +
      vzdal[posunuta_cesta[2],posunuta_cesta[j+1]] -
      vzdal[posunuta_cesta[1],posunuta_cesta[2]] -
      vzdal[posunuta_cesta[j],posunuta_cesta[j+1]]
    if(delta<0){
      zac <- 1
      kon <- (j+1):n
      posunuta_cesta <- c(posunuta_cesta[zac],
                          rev(posunuta_cesta[2:j]),
                          posunuta_cesta[kon])
      Z <- Z + delta
      posl_i <- i
      posl_j <- j
      ############################ break
    }

    if(j == n-1){
      if(i == n){
        i = 1
        j = 3
      }else{
```

```
        i = i+1
        j = 3
      }
      posunuta_cesta <- c(posunuta_cesta[2:n], posunuta_cesta[1])
    }else{j = j+1}
    neobjel_z5 = !((i==posl_i) && (j==posl_j))
  } # end while
  return(list(Z = Z, cesta = posunuta_cesta))
}
```

Do funkce vstupují matice vzdáleností, zápis cesty a hodnota její účelové funkce. Zafixujeme startovní pozice indexů. Zavedeme proměnnou `posunuta_cesta` pro posouvání cesty. Pokud se cesta vrátí na svoji startovní pozici a nebylo nalezeno žádné další možné vylepšení cesty, pak ukončujeme hlavní cyklus výpočtu. To určujeme podle proměnné `neobjel_z5`. Do proměnné `delta` ukládáme v každé iteraci změnu, kterou uděláme při použití změny páru hran. Pokud bylo `delta` menší než 0, změníme `posunuta_cesta` podle nalezené lepší cesty a přepočteme hodnotu účelové funkce `Z`. Pokud zde aktivujeme příkaz `break`, který je zakomentovaný symboly #, pak použijeme pouze jeden krok výpočtu 2-opt, který se nám bude hodit pro některé metaheuristiky.

Následuje zápis, který umožňuje prohledávání celé cesty (přičítání +1 pro indexy `i` a `j`) a provádí i její posunutí v případě prohledání všech dvojic hran pro daný uzel `i`. Po uzavření hlavního cyklu už jen vracíme výslednou cestu a hodnotu účelové funkce.

Pro porovnání s ostatními metodami a jednodušší prací se všemi skripty dohromady jsme vytvořili ještě třetí variantu funkce `LK_2_opt_nahoda()`. Do té vstupuje už pouze matice vzdáleností. Místo vkládání cesty a hodnoty účelové funkce se tyto hodnoty inicializují pomocí nalezení náhodného přípustného řešení.

## 3.7 Metaheuristiky v R

Z metaheuristik jsme k implementaci vybrali metody prahové akceptace, simulované žíhání oceli, genetický algoritmus, optimalizaci mravenčí kolonií a optimalizaci včelí kolonií. První dvě zmíněné metaheuristiky (TA, SIAM) prohledávají okolní řešení z postupně zlepšované cesty. Třetí zmíněná (GA) prohledává populaci, ze které se snaží křížením a mutacemi nalézt nejlepší řešení. Poslední dvě (ACO, BCO) vytváří konstrukčně cesty pomocí z paměti mapované cesty, která je ovlivněna i maticí vzdáleností. Přechod mezi uzly je pak určen s vypočítanou pravděpodobností.

### 3.7.1 Metoda prahové akceptace v R

Pro spuštění funkce `PrahAkcept`, ve které je aplikována metoda prahové akceptace, budeme potřebovat spustit skripty „NahodReseni.R“, „vypocti_Z.R“, „H2_opt_1_step.R“ a v neposlední řadě „M_GA_fce.R“. Funkce, které jsou použity z jednotlivých skriptů, jsou vysvětleny v kapitolách 3.4, 3.6.5 a 3.7.3.

Pro vstup do funkce používáme matici vzdálenosti `vzdalenost`. Dále `procento`, které určuje jaké procento z náhodně vygenerované hodnoty účelové funkce bude použito pro práh (threshodl) $T$, kde procento $\in (0,1)$. Dále vkládáme pravděpodobnostní rozložení metod k prohledávání okolí řešení danou jako `p`, kde jsou uloženy tři hodnoty. Dalším vstupním parametrem je redukce prahu $T$ o parametr `r`. Je vhodné pro něj použít interval $(0{,}8; 1)$. Další parametr `parR` určuje počet opakování iterací jednotlivých výpočtů podle vzorce $\texttt{R} = n \cdot \texttt{parR}$. Nakonec vstupuje parametr `strop_nezmeny`, který udává, jaké maximální množství iterací v hlavním cyklu může být provedeno, aniž by nebyla vylepšena nejlepší hodnota účelové funkce.

```
PrahAkcept <- function(vzdalenosti, procento = 0.05, p = c(0.3,0.4,0.3),
                       r = 0.95, parR = 200, strop_nezmeny = 1300){
  n <- dim(vzdalenosti)[1]
  nahoda <- NahodReseni(vzdalenosti)
  cesta <- nahoda$cesta
  cesta_nej <- cesta
  Z <- nahoda$Z
  Z_nej <- Z
  Praha <- Z_nej * procento
  R = parR * n
  iter = 0
  kdy_naposled <- 0
  while(iter < R && kdy_naposled < strop_nezmeny){
    modifikace <- sample(1:3, 1, prob = p)
    nove <- switch(modifikace,
                   "1" = Swap(cesta,n),
                   "2" = Inverze(cesta,n),
                   "3" = LK_2_opt_1_step(vzdalenosti, cesta, Z)
    )
    if(class(nove)=="list"){
      cesta_nova <- nove$cesta
      Z_nova <- nove$Z
    } else{
      cesta_nova <- nove
      Z_nova <- VypoctiZtka(cesta_nova, vzdalenosti)
      }
    if(Z_nova - Z < Praha){
      cesta <- cesta_nova
      Z <- Z_nova
      if(Z < Z_nej){
        cesta_nej <- cesta
```

```
        Z_nej <- Z
        kdy_naposled <- 0
        Praha <- Z_nej * procento
      }
    }
    Praha <- Praha*r
    iter <- iter + 1
    kdy_naposled <- kdy_naposled + 1
  } # end while
  return(list(Z = Z_nej, cesta = cesta_nej))
}
```

Kód začínáme inicializací jednotlivých parametrů počínaje `n`, které odpovídá počtu uzlů na grafu $G$, tedy $n$. K tomu používáme funkci `dim()`, která vrací dimenzi vloženého objektu. Na první pozici [1] je umístěn počet řádků a na druhé pozici [2] počet sloupců.

Řešení inicializujeme nalezením náhodného řešení pomocí naší funkce `NahodReseni()`. Určíme hodnotu účelové funkce a inicializujeme nejlepší nalezené řešení pomocí nalezeného řešení do proměnných `cesta_nej` a `Z_nej`. Následuje inicializace prahu $T$, který označujeme jako `Praha`, abychom se vyhnuli označení `T`. V R se totiž `T` označuje (obarvuje) jako hodnota `TRUE`.

V hlavním `while` cyklu, který ukončíme po `R` iteracích nebo pokud byla naposledy změněna účelová funkce před `strop_nezmeny` iteracích. První změnou řešení v cyklu je, že vybíráme jednu ze 3 možností změny cesty. K tomu používáme funkce `switch()`. Vybrána je metoda podle vah pravděpodobností `p`. Cestu můžeme změnit pomocí `Swap()`, tedy prohozením dvou indexů. Druhou možností je `Inverze()`, což je inverze náhodné délky části vstupní cesty. Jako třetí a poslední možnost se nabízí ta nejdůležitější, díky které se bude řešení zlepšovat. K tomu používáme funkce `LK_2_opt_1_step`. Ta provede jedno zlepšující prohození 2 hran pomocí jednoho kroku 2-opt heuristiky. Předchozí 2 možnosti slouží hlavně k rozsáhlejšímu prohledávání množiny přípustných řešení. Takto provedená metoda má tedy potenciál být lepší než obyčejná 2-opt heuristika.

Dále v hlavním cyklu následuje spočítání hodnoty účelové funkce a drobného ošetření obsahu nového řešení, které označujeme jako `nove`. Podstatné to je proto, že jednotlivé funkce jsou napsány i pro jiné metody a jednotlivé výstupy funkcí nejsou totožné.

Nakonec v hlavním cyklu kontrolujeme, zda jsme překonali práh `Praha`. Pokud ano, aktualizujeme cestu, kterou používáme jako vzor pro hledání nových řešení. Tu označujeme jako `cesta`. Pokud byl překonán práh, pak ještě máme možnost zlepšit nejlepší dosud nalezenou hodnotu účelové funkce `Z_nej`. Pokud je naše řešení lepší, pak aktualizaci provedeme. V každé iteraci smyčky pomalu snižujeme práh `Praha` pomocí `r` podle vzorce $\texttt{Praha} = \texttt{Praha} \cdot \texttt{r}$.

Jakmile je hlavní smyčka ukončena, už jen vracíme nejlepší nalezené řešení.

### 3.7.2 Metoda simulovaného žíhání v R

Simulace žíhání oceli je aplikována ve funkci `SIAM()`. Opět jako hlavní argument vstupuje matice vzdáleností `vzdalenosti`. Následuje sled přednastavených parametrů, které lze při volání funkce upravit. Obdobně jako v metodě prahové akceptace určujeme počáteční teplotu $T$ parametrem `teplota` a pravděpodobnost nalezení okolních řešení pomocí použitých funkcí určuje tříhodnotový parametr `prav`. K ochlazování používáme parametr `r`. K ukončení hlavního cyklu používáme vstupního parametru `parR`, kterým počítáme `R`, což je maximální počet iterací. Ještě můžeme hlavní cyklus ukončit po `strop_nezmeny` itrací, během kterých nedošlo ke změně řešení.

Inicializaci řešení, ze kterého odvozujeme následující okolní řešení, hledáme pomocí funkce `NahodReseni()`. Jedná se tedy o náhodně nalezené přípustné řešení. Inicializujeme další parametry pro výpočet hlavního cyklu.

V hlavním cyklu, který ukončíme po `R` iteracích nebo po dobu `strop_nezmeny` nezměnění hodnoty nejlepší účelové funkce, provádíme operace. Obdobně jako v metodě prahové akceptace hledáme okolní řešení pomocí funkce `Swap()`, `Inverze()` a `LK_2_opt_1_step`. Který z těchto přístupů použijeme, určíme pomocí váženého náhodného výběru. Váhy určuje parametr `prav`. Pokud jsme se přesunuli do lepšího řešení než je současné prohledávané, aktualizujeme prohledávané řešení. Pokud ne, můžeme aktualizovat prohledávané řešení pod vypočítanou pravděpodobností `P` – vzorec viz kapitola 2.3.4. Pokud prohledávaná cesta (po aktualizaci) dosahuje lepší hodnoty účelové funkce než dosud nejlepší nalezené, pak též aktualizujeme. Hlavní cyklus ukončuje přepočítání teploty a pomocných parametrů. Jakmile je hlavní cyklus ukončen, vracíme nejlepší nalezené řešení.

```
SIAM <- function(vzdalenosti, procento = 0.05, prav = c(0.3,0.4,0.3),
                         r = 0.95, parR = 200, strop_nezmeny = 1200){
  n <- dim(vzdalenosti)[1]
  nahoda <- NahodReseni(vzdalenosti)
  cesta <- nahoda$cesta
  cesta_nej <- cesta
  Z <- nahoda$Z
  Z_nej <- Z
  teplota <- Z_nej * procento
  iter = 0
  kdy_naposled <- 0
  R = parR * n
  while(iter < R && kdy_naposled < strop_nezmeny){
    modifikace <- sample(1:3, 1, prob = prav)
    nove <- switch(modifikace,
                   "1" = Swap(cesta,n),
                   "2" = Inverze(cesta,n),
                   "3" = LK_2_opt_1_step(vzdalenosti, cesta, Z)
    )
    if(class(nove)=="list"){
```

```
      cesta_nova <- nove$cesta
      Z_nova <- nove$Z
    } else{
      cesta_nova <- nove
      Z_nova <- VypoctiZtka(cesta_nova, vzdalenosti)
    }
    if(Z_nova < Z){
      cesta <- cesta_nova
      Z <- Z_nova
    } else{
        P = -(Z_nova-Z)
        P = exp(P/teplota)
        pom <- sample(1:2, 1, prob = c(P,(1-P)))
        cesta <- switch(pom,
                        "1" = cesta_nova,
                        "2" = cesta
        )
        Z <- switch(pom,
                    "1" = Z_nova,
                    "2" = Z
        )
    } # end zmena s pravdepodobnosti
    if(Z < Z_nej){
      cesta_nej <- cesta
      Z_nej <- Z
      kdy_naposled <- 0
      teplota <- Z_nej * procento
    }
    teplota <- teplota*r
    iter <- iter + 1
    kdy_naposled <- kdy_naposled + 1
  } # end while
  return(list(Z = Z_nej, cesta = cesta_nej))
}
```

### 3.7.3 Genetický algoritmus v R

Kód genetického algoritmu byl vložen do přílohy A.3 pro svoji délku. Pro spuštění algoritmu je potřeba aktivovat scripty „M_GA_fce.R“, „NahodReseni.R“, „H_greedy.R“ a „vypocti_Z.R“. GA patří k programátorsky náročnějším algoritmům a vyžaduje mnoho dílčích funkcí. Zároveň se jedná o zajímavý přístup, který lze vylepšovat velkým množstvím způsobů a používáním nejrůznějších přístupů.

Do funkce vstupuje jako hlavní prvek matice vzdáleností `vzdalenost`. Druhotné vstupy mají svoji inicializovanou hodnotu, ale lze ji nastavit při spouštění funkce. Do `ex1` se vkládá parametr pro velikost populace, který je dán vzorcem $N = 2^{ex1}$. Do `ex2` se vkládá výsledný

počet rodičů daný vzorcem $2^{ex2}$. Pokud nastane situace, kdy $x1 < x2$, bude uživatel upozorněn, že to není možné a funkce nebude spuštěna. Parametr `strop_nezmeny` slouží pro ukončovací podmínku a pokud po tuto danou dobu iterací nebude změněna nejlepší hodnota účelové funkce, bude hlavní cyklus ukončen. Do `coefmin` a `coefmax` se vkládají parametry pro PMX křížení, které pomáhají udávat velikost jádra. Dalším vstupním parametrem je `prst`, který udává pravděpodobnost toho, zda bude provedena mutace u dětí nebo ne. Pokud bude vložena hodnota mimo interval (0,1), uživatel bude upozorněn, že to není možné a výpočet neproběhne. Jako poslední parametr vstupují `vahy_mutaci`, které udávají pravděpodobnost použití mutace „swap“, „inverze“ a „scramble“. Jedná se tedy o 3 hodnoty, které lze nastavit libovolně, ale čím vyšší hodnotu dostane, tím bude větší šance užití daného typu mutace.

V první části kódu dochází k inicializaci populace. Je vloženo $N - 1$ náhodně vygenerovaných řešení a jedno, které je spočítáno pomocí heuristiky hladového algoritmu. Inicializace je ukončena nalezením nejlepšího „fitness“ řešení.

Hlavní cyklus bude ukončen, jakmile dosáhneme daného množství iterací bez změny hodnoty účelové funkce. V tomto cyklu se počítá s celou populací zároveň. Jako první jsou spočítáni rodiče pomocí námi vytvořené funkce `VyberTurnaj`, který vybírá turnajovým pravidlem rodiče. Následuje křížení rodičů pomocí funkce `krizeniPMX`. Pokud bude pravděpodobnost nakloněna, pak ještě proběhne mutace dětí pomocí funkce `MutujCestu`. Následuje přepočet hodnoty účelové funkce pomocí `VypoctiZtka`. Následně je celá populace (původní s dětmi) seřazena od nejnižší hodnoty účelové funkce po nejvyšší hodnotu účelové funkce a je vybráno pouze $N$ nejlepších řešení. Pokud dojde k překonání nejlepšího řešení, pak dojde k jeho přepsání.

### Výběr rodičů turnajovým pravidlem v R

Pro výběr rodičů jsme vybrali turnajové pravidlo. Do funkce vstupují hodnoty jednotlivých účelových funkcí `Ztka` populace, velikost populace `N` a parametr `ex2` pro výpočet množství rodičů.

```
VyberTurnaj <- function(Ztka, N, ex2){
  pocet <- 2^(ex2)
  aktualni_pocet <- N
  postup <- 1:N
  while(pocet < aktualni_pocet){
    pom <- postup
    vitezove <- NULL
    for(z in 1:(aktualni_pocet/2)){
      a <- sample(pom, 1)
      pom <- setdiff(pom, a)
      if(length(pom) == 1){b <- pom
        }else{
        b <- sample(pom, 1)
```

```
        pom <- setdiff(pom,b)
      }
      if(Ztka[a] < Ztka[b]){vitezove <- c(vitezove, a)
        }else{vitezove <- c(vitezove, b)}
    }
    postup <- vitezove
    aktualni_pocet <- length(postup)
  } # end while
  vzor <- sample(postup, length(postup)/2)
  otec <- as.data.frame(vzor)
  postup <- postup[!(postup %in% vzor)]
  matka <- as.data.frame(sample(postup, length(postup)/2))
  rodice <- data.frame(otec,matka)
  colnames(rodice) <- c("otec", "matka")
  return(rodice)
}
```

Inicializujeme parametry před hlavním cyklem, který bude ukončen tehdy, když bude vybráno méně rodičů `aktualni_pocet` oproti požadovanému množství rodičů `pocet`. V rámci hlavního cyklu pak používáme dílčí `for` cyklus, který použijeme tolikrát, kolik máme k dispozici kandidátů na rodiče děleno dvěma. V dílčím cyklu pak vybíráme náhodně 2 kandidáty `a` a `b` pomocí funkce `sample()`. Ten s lepší hodnotou pak bude uložen mezi vítěze a postupující. Zároveň kandidáti `a` a `b` již nesmí být znovu vybráni, a tak jsou ihned po vybrání odstraněni z možnosti výběru `pom`. Po ukončení hlavního cyklu jsou rodiče rozděleni na otce a matky a vráceni funkcí v páru jako data frame `rodice`.

### Křížení PMX v R

Křížení rodičů jsme zhotovili pomocí pravidla PMX. Kód viz kapitola A.4. Vstupem do funkce jsou `cesty`, `rodice` a `n`. Název udává jasně, že se jedná o jednotlivé cesty, spárované rodiče (otce a matky) a index velikost úlohy $n$ udávající počet uzlů.

Další dva druhotné parametry `coefmin` a `coefmax` udávají do jaké maximální šířky lze manipulovat s jádrem pro výpočet PMX. Tyto hodnoty jsou pak vypočteny a uloženy do `min_jadro` a `max_jadro`. Jednotlivé krajní indexy jsou pak vybrány náhodně a uloženy do `j1` a `j2`. Celá tato procedura již probíhá ve `for` cyklu, aby každý pár potomků byl originálně zkřížen.

Cyklus pokračuje taháním jader `jadro_o` a `jadro_m` z cesty rodičů. Ty doplňujeme druhým z rodičů a vytváříme děti `dite_1` a `dite_2`. Požíváme pak trik se změnou typu indexů na charaktery, aby se nám lépe mapovaly vyměněné indexy pomocí funkcí `names()` a `as.character()`. V dílčích dvou `while` cyklech (pro každé dítě jeden) pak mapujeme a nahrazujeme druhotné indexy v cestách dětí. Jakmile se žádný index v cestě neopakuje, přichází na řadu další iterace páru rodičů.

Po použití všech párů rodičů vracíme `deti` dané jednotlivými cestami.

### Mutace swap v R

Po náročnějších úkonech přichází jednodušší, a tou jsou mutace. Jednoduché prohození 2 náhodně vybraných uzlů, jak je mutace swap (prohození) definována, pomocí funkce `sample()` v cestě lze provést v R následovně.

```
Swap <- function(cesticka, n){
  s <- sample(n,2)
  novacesta <- cesticka
  novacesta[s] <- novacesta[rev(s)]
  return(novacesta)
}
```

Používáme pro to funkce `rev()`, která inverzně prohodí vloženou číselnou čadu.

### Mutace inverze v R

Do inverzní mutace vstupuje `cesticka` určující permutaci cesty a hodnota `n`, která určuje délku cesty.

```
Inverze <- function(cesticka, n){
  dvojice <- sample(n,2)
  i1 <- min(dvojice)
  i2 <- max(dvojice)

  if(i1 == 1){zac <- c()
    }else{zac <- 1:(i1-1)}

  if(i2 == n){kon <- c()
    }else{kon <- (i2+1):n}

  novacesta <- c(cesticka[zac],rev(cesticka[i1:i2]), cesticka[kon])
  return(novacesta)
}
```

Opět začínáme výběrem dvou indexů pomocí funkce `sample()` a uložíme je do `dvojice`. Tentokrát však vybereme ten, jehož index je menší a označíme ho `i1`. Pro vyšší index použijeme označení `i2`. Pokud je některý ze zvolených indexů startovní (poslední), pak není potřeba indexy předcházející (následující) ukládat do nově vzniklé cesty. Pokud však ne, je potřeba si jednotlivé uzly k okopírování uložit. Do nové cesty pak vkládáme první část cesty `zac`,

kterou opíšeme. Do druhé části vkládáme inverzi cestu mezi indexy `i1` a `i2` převrácenou pomocí funkce `rev()`. Do poslední části vkopírujeme poslední část cesty `kon`. Závěrem vracíme hodnotu nové cesty.

### Mutace scramble v R

Mutace scramble (náhodná) funguje velmi podobně jako swap. Jediný rozdíl je, že místo inverzního otočení jádra dojde k náhodnému proházení cesty v úseku jádra.

```
Scramble <- function(cesticka, n){
  dvojice <- sample(1:n,2)
  sc1 <- min(dvojice)
  sc2 <- max(dvojice)
  l <- sc2-sc1+1
  if(sc1 == 1){zac <- c()
    }else{
    zac <- 1:(sc1-1)}
  if(sc2 == n){kon <- c()
    }else{
    kon <- (sc2+1):n}
  novacesta <- c(cesticka[zac],sample(cesticka[sc1:sc2],l), cesticka[kon])
  return(novacesta)
}
```

Opět vybíráme 2 indexy `sc1` (nižší) a `sc2` (vyšší), které určují pozici jádra. Délku jádra, tedy části cesty k „zamíchání“ si označíme jako `l`. Následuje stejná procedura jako u inverze. Nyní však místo funkce `rev()` použijeme funkci `sample`. S její pomocí vybereme náhodně pořadí jednotlivých uzlů v jádru.

### Výběr mutace

Pro výběr mutace a aplikace na všechny děti používáme funkce `MutujCestu`. Do vstupního parametru `cesty` tedy vkládáme `deti`. Další vstupní parametr `n` opět označuje délku cesty k řešení TSP. Poslední vstupní parametr `p` označuje pravděpodobnosti použití jednotlivých tří typů mutací.

```
MutujCestu <- function(cesty, n, p = rep(1/3, 3)){
  mutovane_deti <- cesty
  for(i in 1:dim(cesty)[1]){
    modifikace <- sample(1:3, 1, prob = p)
    mutovane_deti[i,] <- switch(modifikace,
                        "1" = Swap(cesty[i,],n),
```

```
                        "2" = Inverze(cesty[i,],n),
                        "3" = Scramble(cesty[i,],n)
                        )
  }
  return(mutovane_deti)
}
```

V hlavním `for` cyklu pak vybíráme pomocí výběrové funkce `switch()`, kterou z náhodně vybraných (pomocí `sample()` mutací použijeme k mutaci. Pod jednotlivými variantami 1 až 3 pak aplikujeme mutační funkce z předchozích kapitol. Po mutování všech dětí pak vracíme `mutovane_deti`.

### 3.7.4 Optimalizace mravenčí kolonií v R

Pro metodu ACO jsme vytvořili funkci `ACO()`. Opět do ní vstupuje matice vzdáleností a několik druhotných parametrů, které jsou přednastaveny, ale mohou být uživatelem změněny. Parametr `coefm` udává množství mravenců $m$, které máme k dispozici. To následně určujeme jako $m = coefm \cdot n$. Jelikož dělení nulou není úplně přípustné a v rámci vstupních dat se objevují vzdálenosti hvězd, které jsou nulové (dvojhvězdí, trojhvězdí), je potřeba drobného opatření, ke kterému používáme parametru `eps`. To udává, jaká nízká hodnota se má přičíst k matici vzdáleností, abychom se vyhnuli dělení nulou. Parametry `alpha, beta` a `rho` pak odpovídají parametrům $\alpha, \beta$ a $\rho$ z kapitoly 2.3.6. Nakonec vstupují parametry `R` a `nezlepseno`, které pomáhají určit konec hlavní smyčky a ukončit výpočet.

Počet uzlů na grafu ukládáme jako vždy do `n`. Počet mravenců do `m`. Pokud by jich bylo k řešení málo, uživatel na to bude upozorněn a výpočet se neprovede. Dále inicializujeme `eta` z matice vzdáleností a parametru `eps` pro vyhnutí se dělení nulou. Připravíme si feromon na hrany do proměnné matice `feromon`. Po celé matici rozmístíme jedničky kromě hlavní diagonály, tam vložíme nuly. Dále před hlavní smyčkou inicializujeme potřebné parametry pro výpočet.

V hlavním `while` cyklu pak kontrolujeme ukončovací podmínky. Ty jsou dány určeným množstvím iterací `R`. Druhá možnost, kdy může být výpočet ukončen, je, že za posledních `nezlepseno` iterací nebyla vylepšena hodnota účelové funkce. V dílčím `for` cyklu pak hledáme cestu pro každého mravence pomocí feromonu a informace z matice vzdáleností. To zda se přesune z uzlu $i$ do uzlu $j$ je dáno pravděpodobnostně, takže každý výpočet může přinést jiné řešení stejné úlohy. K práci s pravděpodobností pak používáme funkce `sample()` a seznam uzlů `povol_seznam`, kam se ještě může mravenec přesunout. Ten aktualizujeme pomocí funkce `setdiff()`. Pro výpočet trasy používáme ještě `while` cyklus, který zajišťuje projití celé trasy.

Jakmile máme pro mravence cestu, zjistíme hodnotu účelové funkce pomocí naší funkce `VypoctiZtka()`. Pokud bylo nalezeno lepší řešení než dosud nejlepší nalezené, parametry

určující nejlepší řešení aktualizujeme. Následně provedeme lokální aktualizaci feromonu na trasách prošlých mravencem. Jedná se tedy o feromon, který mravenec uvolnil.

Když všech $m$ mravenců najde svoji cestu, pak aktualizujeme globálně feromon, což je dáno vypařováním a snížením této hodnoty. Následně zkontrolujeme, zda došlo ke změně pomocí parametru `doslo_ke_zmene` a případně aktualizujeme parametr určující ukončení hlavního cyklu.

Jakmile bude ukončen výpočet, funkce vrátí trasu nejlepší nalezené cesty a její hodnotu účelové funkce.

```
ACO <- function(vzdalenosti, coefm = 1, eps = 1e-10,
                alpha = 1, beta = 2, rho = 0.15,
                R = 100, nezlepseno = 10){
  n <- dim(vzdalenosti)[1]
  m <- floor(n*coefm)
  if (m<4){stop("Malo mravencu, zvys coefm")}
  eta <- 1/(vzdalenosti+eps)
  feromon <- matrix(1, nrow = n, ncol = n)
  diag(feromon) <- 0
  cesta_nej <- c()
  Z_nej <- Inf
  r <- 1
  aktualizace <- 0
  while(r < R && aktualizace < nezlepseno){
    doslo_ke_zmene <- FALSE
    for(k in 1:m){
      i <- sample(1:n,1)
      povol_seznam <- setdiff(1:n,i)
      cesta <- i
      while(length(cesta)<n-1){
        prsti <- (feromon[i,povol_seznam])^alpha * (eta[i,povol_seznam])^beta
        j <- sample(povol_seznam, size = 1, prob = prsti)
        povol_seznam <- setdiff(povol_seznam,j)
        cesta <- c(cesta,j)
        i <- j
      } # end while cesty
      cesta <- c(cesta, povol_seznam)
      Z <- VypoctiZtka(cesta,vzdalenosti)
      if (Z < Z_nej){
        Z_nej <- Z
        cesta_nej <- cesta
        doslo_ke_zmene = TRUE
      }
      hrany_cesty <- matrix(c(cesta, cesta[2:n], cesta[1]), nrow = n)
      feromon[hrany_cesty] <- feromon[hrany_cesty]+1/Z

    } # end for mravence
```

```
    feromon <- (1-rho) * feromon
    if (doslo_ke_zmene){aktualizace <- 0
    } else {aktualizace <- aktualizace + 1}
    r <- r + 1
  } # end while
  return(list(Z = Z_nej, cesta = cesta_nej))
}
```

### 3.7.5 Optimalizace včelí kolonií v R

Pro použití algoritmu BCO jsme využili námi vytvořené funkce, které je potřeba spustit ze souborů „H_2-opt_1_step.R“, „H_nearest_neighbor.R“ a „vypocti_Z.R“. Pro délku kódu algoritmu jej uvádíme v příloze A.5.

Hlavním vstupním parametrem pro funkci `BCO()` je matice vzdáleností. Stejně jako pro algoritmus ACO, tak i zde inicializujeme parametry `eps`, `alpha`, `beta`, `R` a `nezlepseno`. Nově však vkládáme parametr `howtheta`, který určuje jak má být zvolena včelka $\theta$, kterou se má včela $i$ na startu inspirovat v podnikání cesty. Na výběr jsou `Zweighted` a `uniform`. První zmíněná možnost vybírá $\theta$ podle váhy převrácené hodnoty účelové funkce $(1/Z)$ a druhá možnost vybírá $\theta$ náhodně. Dalším parametrem je `inic_W`, což je doba v jednotkách iterací, po kterou budou inicializační včely radit své trasy ostatním včelám. Do parametru `K` vkládáme hodnotu, která udává, jak dlouho bude včelka radit svoji nalezenou trasu. Následuje vstupní parametr `povolit_2opt`, který lze zvolit jako pravdivý (`TRUE`) nebo nepravdivý (`FALSE`). Říká pak, jestli má nebo nemá být použit jeden krok 2-opt Lin Kernighen heuristiky pro vylepšení nalezené trasy. Poslední nezmíněný parametr je `lambda`, který odpovídá $\lambda$ z algoritmu BCO. Jedná se o pravděpodobnost následování dalšího uzlu podle rady včely $\theta$. Je vhodné ho inicializovat na intervalu $(0{,}8; 1)$.

Začínáme inicializací počtu uzlů v úloze `n`, přejmeme myšlenku z ACO a heuristickou informaci o cestě uložíme do `eta` a seznamu tancujících včelek `waggle` s informací celé cesty, který index včela měla při nalezení cesty, hodnotu účelové funkce a dobu, po jakou bude včelka tancovat. Tancující včelky pak plníme jednotlivými včelami ve `for` cyklu a inicializujeme je algoritmem nejbližšího souseda implementovaném jako `NejSoused()`. Zároveň uložíme tyto hodnoty pro každou jednotlivou včelu $i$ jako její nejlepší dosažené řešení `vcely_nej` s hodnotami účelových funkcí `Zvcely_nej`. Nakonec následuje příprava parametrů výpočtu pro hlavní cyklus.

V hlavním výpočetním `while` cyklu máme 3 možnosti jeho ukončení. Buď dosáhneme daného množství iterací `R`, nebo nebylo nalezeno lepší řešení po dobu `nezlepseno` iteraci, a nebo `waggle_neprazdny` nabude nepravdivé hodnoty (bude tedy prázdný) a včelky by se neměly čím inspirovat, takže bude výpočet ukončen. Zde lze aplikovat algoritmus i jiným způsobem a povolit počítání dále, i když už nebude radit žádná včelka, ale je to potřeba ošetřit během výpočtu.

V hlavním cyklu pak jeho podstatnou část zabírá `for` cyklus, který vytváří cestu pro každou včelku, která jde hledat trasu. Aktuální včelka na cestě je označována indexem $k$ a bude posláno `n` včel. Vždy je potřeba zkontrolovat, zda zrovna některá z $k$ včel nepřestala tancovat a případně ji odstranit ze seznamu `waggle`. Pokud se stane, že bude odstraněna poslední včela ze seznamu `waggle`, pak bude algoritmus ukončen pomocí `waggle_neprazdny <- FALSE` a příkazu `break`. Nyní volíme včelu $\theta$ ze seznamu `waggle`, kterou označíme jako `theta` a její cestu `theta_cesta`.

Nyní ve `while` cyklu volíme jednotlivé uzly, která má včela $k$ navštívit. Začíná v uzlu $i = k$ a vždy hledá uzel $j$ podle toho, zda jí ho může včela $\theta$ poradit (v tom případě radí uzel označovaný jako `F_i`) nebo ne. Zároveň sama dokáže uvážit, jestli nepoužít hranu podle její délky. Následující uzel $j$ je vždy vybrán podle pravděpodobnosti a pomocí funkce `sample()`, která vybírá podle vah. Vždy je pak aktualizován seznam `povol_seznam`, který říká, které uzly ještě nebyly navštíveny.

Jakmile je vytvořena cesta, uzavíráme ji do Hamiltonova cyklu a počítáme hodnotu účelové funkce. Pokud je to povoleno, aplikujeme na nalezenou cestu jeden krok 2-opt Lin Kerninghen heuristiky pomocí funkce `LK_2_opt_1_step`. Následuje aktualizace nejlepší cesty $k$ pro včelu $k$, pokud bylo dosaženo lepšího řešení. Pokud ano, pak včelka půjde tancovat a radit ostatním včelkám do `waggle` seznamu. Použité vzorce již byly vysvětleny v kapitole 2.3.7. Následně ověříme, zda nebylo nalezeno lepší řešení než nejlepší řešení celého úlu a pokud ano, aktualizujeme hodnoty `Z_nej` a `cesta_nej`. Pokud došlo ke změně, aktualizujeme parametr `aktualizace`, aby mohlo vyhledávání lepší cesty pokračovat i v dalších iteracích.

Následuje ukončení `for` a `while` cyklu a vrácení nejlepšího nalezeného řešení.

## 3.8 Ladění parametrů

Jelikož ladění parametrů není hlavním cílem práce, zmiňujeme se stroze o použitých parametrech a navrhujeme je k bedlivějšímu výzkumu. Dostupná literatura není jednotná v nastavení použitých metod.

Pro heuristiky není třeba ladit žádné parametry, ovšem pro metaheuristiky je tomu jinak. Vzhledem k ladění parametrů i hlavních myšlenek metod lze aplikované metaheuristiky rozdělit do tří skupin následovně:

- ▷ prohledávání extrémů pomocí blízkých řešení - TA, SIAM,
- ▷ evoluční prohledávání - GA,
- ▷ postupná konstrukce cesty inspirovaná přírodou - ACO, BCO.

Pro nastavení jednotlivých hodnot jsme použili metody pokusu a omylu v kombinaci s doporučenými nastaveními z citovaných článků v rámci jednotlivých teoretických částí. Při studiu ladění parametrů jsme narazili na nejednotnost, tudíž jsme používali ty parametry, které vykazovaly lepší řešení. Pokud řešení nedávalo smysl (nebylo suboptimální), pak jsme dané

nastavení zavrhli. Pro některé metaheuristiky nastavené parametry dosahovaly dobrých výsledků pro malé úlohy, ale pro větší úlohy už však nastávaly problémy v podobě neefektivity. Snažili jsme se tedy nalézt rovnováhu v jejich ladění.

Uvádíme jednotlivé použité parametry a jejich vhodné a námi použité hodnoty v zápisu jazyka R. Popisy daných parametrů jsou uvedeny v rámci kapitoly 3.7.

Pro prohledávání extrémů (TA a SIAM) ladíme následující parametry.

```
procento = 0.05
p = c(0.3,0.4,0.3)
r = 0.95
parR = 200
strop_nezmeny = 1200
```

Pro evoluční prohledávání (GA) používáme následující parametry.

```
ex1 = 8
ex2 = 6
coefmin = 1/7
coefmax = 1/2
prst = 0.5
vahy_mutaci = c(0.6,0.2,0.2)
parR = 100
strop_nezmeny = 200
```

Nakonec pro konstrukce jednotlivých přechodů mezi uzly inspirované přírodou (ACO, BCO) používáme tyto parametry.

```
eps = 1e-10
alpha = 1
R = 100
```

Jelikož jsou tyto přístupy komplikovanější, pak pro upřesnění metody ACO používáme další parametry.

```
coefm = 1
beta = 2
rho = 0.15
nezlepseno = 10
```

Takto použijeme $n$ mravenců, tedy stejně jako počet uzlů.

Stejně tomu je i v případě metody BCO, které přidáváme následující parametry.

```
howtheta = "Zweighted"
inic_W = 30
beta = 8
K = 20
povolit_2opt = TRUE
lambda = 0.9
nezlepseno = 20
```

## 3.9 Výsledky

Veškeré výpočty jsme implementovali ve skriptu „vypocty_tabulka.R“ za pomoci skriptu „TSPdanouMetodou.R“. Nahráváme potřebné balíčky, námi vytvořené kódy a data. Abychom nemuseli opakovat výpočty vzdáleností a znovu spouštět funkce, ukládáme provedené operace do „.RData“ záznamů, pro opětovné použití. Ty následně načítáme v R a provádíme na nich další výpočetní experimenty.

Doporučujeme se vyhnout výpočtům vzdáleností velkých datasetů ($n \geq 1000$), které mohou trvat i několik hodin. Místo toho můžeme načítat již námi vypočtené vzdálenosti.

Výpočty jsme rozdělili na heuristiky a metaheuristiky z důvodu časové náročnosti výpočtů především u ACO a BCO.

Během experimentů doporučujeme zabránit počítači v přechodu do režimu spánku a samovolným aktualizacím. Některé metody vyžadují více času k řešení a výpočty se mohou protáhnout na několik desítek hodin. Zároveň doporučujeme použít vícejádrový počítač, který dokáže dělat několik výpočtů najednou v kombinaci s balíčkem „parallel“.

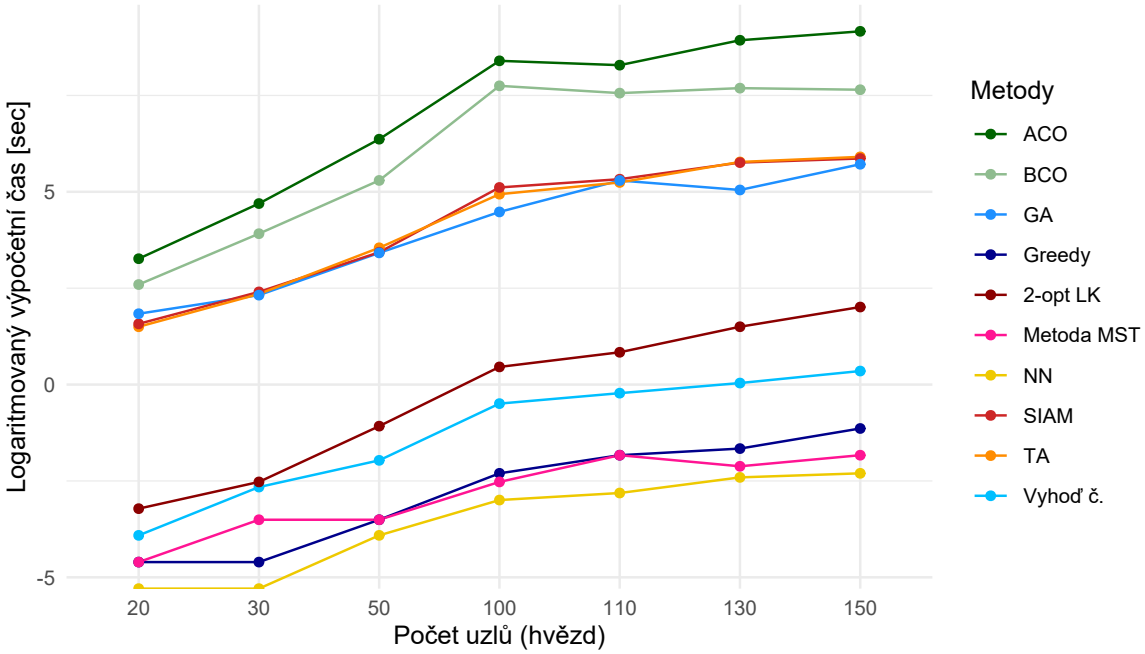
Výpočet metaheuristik jsme opakovali třikrát a vybrali jsme vždy nejlepší hodnoty účelových funkcí a k nim adekvátní časové údaje.

### 3.9.1 Porovnání všech implementovaných metod

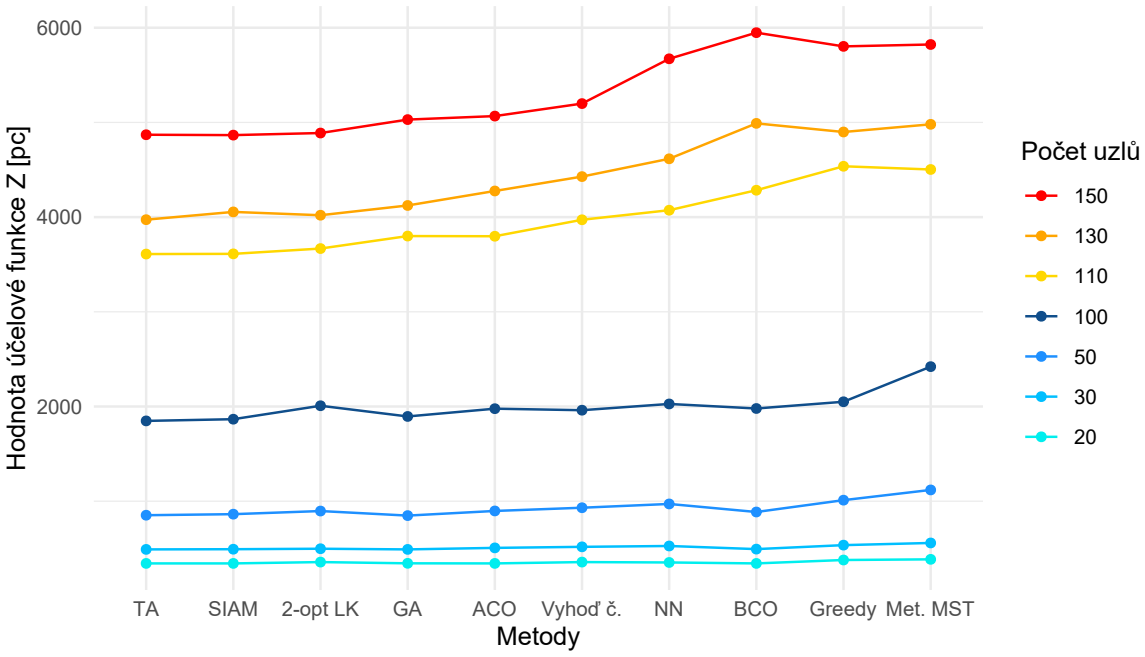
Porovnání potřebného času k vypočtení všech metod je v závislosti na počtu uzlů vyobrazeno na obrázku 3.3.

Nejrychlejší byla metoda nejbližšího souseda následována metodou minimální kostry, hladovým algoritmem, metodou výhodnostních čísel a 2-opt LK heuristikou. Heuristiky jsou tedy znatelně rychlejší než metaheuristiky. Nejrychlejší z metaheuristik byl genetický algoritmus, následovaný metodou prahové akceptace a simulací žíhání oceli. Mezi těmito metodami byl už jen drobný časový rozdíl. Časově nejhůře dopadly optimalizace včelí a mravenčí kolonií.

Ve výsledcích je patrné, že i drobný výpočet přináší časové zpoždění. To můžeme porovnat na




Obrázek 3.3: Výpočetní čas v závislosti na počtu uzlů pro každou metodu.




Obrázek 3.4: Hodnoty účelových funkcí v závislosti na jednotlivých metodách.

hladovém algoritmu a metodě výhodnostních čísel, kde pro druhou metodu počítáme navíc výhodnostní čísla.

Zajímavá je také rychlost metody minimální kostry a hladového algoritmu. Pro méně uzlů byla lepší metoda hladového algoritmu, ale s přibývajícími uzly byla rychlejší metoda minimální kostry.

Porovnání hodnot účelových funkcí metod zobrazujeme na obrázku 3.4.

Celková efektivita metod podle hodnot účelových funkcí je seřazena od nejlepších po nejhorší zleva doprava na obrázku 3.4. Nejefektivnější byly prahová akceptace a metoda simulovaného žíhání oceli. Následovala nejlepší heuristika 2-opt LK. Další v pořadí jsou metody genetického algoritmu a optimalizace mravenčí kolonií. Další v pořadí jsou metoda výhodnostních čísel a nejbližšího souseda. Nyní přichází na řadu nejhorší umístěná metaheuristika optimalizace včelí kolonií, která by vyžadovala úpravu parametrů pro nalezení lepších řešení. Nutno však podotknout, že v sobě obsahuje i 2-opt LK heuristiku, která by měla zařídit zlepšování řešení. Nejhůře dopadly hladový algoritmus a metoda minimální kostry.

Přehled jednotlivých hodnot pro vykreslení obrázků viz tabulky 3.3 a 3.4.

Tabulka 3.3: Porovnání metod na úlohách pro $n \leq 100$.

| $n$ | Metoda | Čas [s] | $Z$ [pc] | $n$ | Metoda | Čas [s] | $Z$ [pc] |
|---|---|---|---|---|---|---|---|
| 20 | NN | $< 0{,}01$ | 353 | 50 | NN | 0,02 | 971 |
| | Greedy | 0,01 | 379 | | Greedy | 0,03 | 1011 |
| | Výhod. č. | 0,02 | 357 | | Výhod. č. | 0,14 | 931 |
| | Metoda MST | $< 0{,}01$ | 386 | | Metoda MST | 0,03 | 1119 |
| | 2-opt LK | 0,04 | 357 | | 2-opt LK | 0,34 | 896 |
| | TA | 4,48 | 343 | | TA | 34,73 | 852 |
| | SIAM | 4,83 | 343 | | SIAM | 30,92 | 863 |
| | GA | 6,28 | 344 | | GA | 30,43 | 848 |
| | ACO | 26,17 | 343 | | ACO | 581,06 | 897 |
| | BCO | 13,39 | 343 | | BCO | 199,33 | 886 |
| 30 | NN | $< 0{,}01$ | 527 | 100 | NN | 0,05 | 2027 |
| | Greedy | 0,01 | 536 | | Greedy | 0,10 | 2050 |
| | Výhod. č. | 0,07 | 518 | | Výhod. č. | 0,61 | 1961 |
| | Metoda MST | 0,03 | 559 | | Metoda MST | 0,08 | 2421 |
| | 2-opt LK | 0,08 | 498 | | 2-opt LK | 1,58 | 2008 |
| | TA | 10,47 | 491 | | TA | 139,50 | 1848 |
| | SIAM | 11,06 | 493 | | SIAM | 166,17 | 1866 |
| | GA | 10,17 | 491 | | GA | 88,06 | 1895 |
| | ACO | 109,43 | 507 | | ACO | 4436,77 | 1977 |
| | BCO | 50,08 | 494 | | BCO | 2318,81 | 1979 |

Tabulka 3.4: Porovnání metod na úlohách pro $100 < n \leq 150$.

| $n$ | Metoda | Čas [s] | $Z$ [pc] |
|---|---|---|---|
| 110 | NN | 0,06 | 4073 |
| | Greedy | 0,16 | 4537 |
| | Výhod. č. | 0,80 | 3972 |
| | Metoda MST | 0,16 | 4503 |
| | 2-opt LK | 2,31 | 3668 |
| | TA | 189,19 | 3610 |
| | SIAM | 205,90 | 3612 |
| | GA | 199,00 | 3800 |
| | ACO | 3965,07 | 3798 |
| | BCO | 1919,89 | 4284 |
| 130 | NN | 0,09 | 4616 |
| | Greedy | 0,19 | 4899 |
| | Výhod. č. | 1,04 | 4428 |
| | Metoda MST | 0,12 | 4980 |
| | 2-opt LK | 4,48 | 4020 |
| | TA | 321,95 | 3973 |
| | SIAM | 316,63 | 4055 |
| | GA | 155,64 | 4123 |
| | ACO | 7563,94 | 4276 |
| | BCO | 2184,67 | 4990 |
| 150 | NN | 0,10 | 5672 |
| | Greedy | 0,32 | 5803 |
| | Výhod. č. | 1,42 | 5199 |
| | Metoda MST | 0,16 | 5823 |
| | 2-opt LK | 7,47 | 4888 |
| | TA | 367,42 | 4870 |
| | SIAM | 352,14 | 4866 |
| | GA | 303,33 | 5030 |
| | ACO | 9534,37 | 5067 |
| | BCO | 2096,13 | 5948 |

### 3.9.2 Podrobnější porovnání heuristik

K této kapitole patří tabulky a obrázky uvedené v přílohách B a C.

Jak vidíme na obrázku C.1, jednotlivé metody ještě nejsou řádně ustálené, ale výpočetně nejnáročnější je heuristika 2-opt LK, což se během následujících experimentů potvrdí. Na obrázku C.3 již vidíme ustálení jednotlivých procedur, které se už na obrázku C.5 nemění. Nejdéle trval výpočet 2-opt LK heuristiky, která trvala necelých 17 hodin.

Jednoznačně nejrychlejší byla metoda minimální kostry, pro kterou jsme použili zabudovanou funkci výpočtu minimální kostry, což mohlo urychlit celkovou výpočetní dobu. Na druhém místě skončila metoda nejbližšího souseda, která též dokázala problémy řešit v řádech vteřin. Hladový algoritmus byl třetí nejrychlejší následovaný metodou výhodnostních čísel.

Jak vidíme na obrázcích C.2, C.4 a C.6, efektivita jednotlivých heuristik se nemění a zůstává neměnná pro různě velká $n$. Pořadí je opět seřazeno od nejlepší po nejhorší směrem zleva doprava, tedy: 2-opt LK, metoda výhodnostních čísel, nejbližší soused, hladový algoritmus a metoda minimální kostry.

Překvapením je, že metoda nejbližšího souseda dosahovala lepších výsledků než hladový algoritmus. Zřejmě 3D prostředí a rozmístění hvězd svědčí této heuristice.

### 3.9.3 Podrobnější porovnání metaheuristik

K této kapitole patří tabulky a obrázky uvedené v přílohách B a C.

Z metaheuristik vychází nejhůře výpočetní čas procedur ACO a vzápětí BCO. Zároveň dosahují i horších hodnot účelových funkcí. Hlavně u BCO se nepodařilo nastavit parametry tak, aby metody dokázaly konkurovat ostatním implementovaným metaheuristikám. Na druhou stranu vysoký výpočetní čas indikuje neefektivitu procedur. Ta spočívá v konstrukci cest index po indexu, kdežto ostatní metaheuristiky prohledávají okolní přípustná řešení již nalezené dobré cesty.

Časově nejlépe vychází GA následovaný TA a SIAM. Nejlepších hodnot účelových funkcí však dosahuje SIAM následovaná TA a GA. Těžko tedy volit, který z těchto přístupů je lepší. I pro tyto metody je prostor ke zlepšení v podobě úpravy parametrů na každou úlohu zvlášť.

Pro malé úlohy byl nejlepší GA, který získal oproti ostatním metodám pro počet uzlů $n = 50$ nevýrazný náskok.

Na základě výzkumu můžeme říci, že prohledávání okolních řešení je pro 3D úlohy efektivnější než postupná konstrukce cest. Budoucí výzkum by tedy bylo vhodné směřovat touto cestou nebo prohledáváním efektivity nastavení parametrů.

## 3.10 Zhodnocení výpočetních experimentů

Pořadí efektivity (výpočetní čas, hodnota účelové funkce) heuristik mezi sebou a metaheuristik mezi sebou uvádíme v tabulce 3.5.

Překvapivě dobrých výsledků dosáhla heuristika nejbližšího souseda. Nejlepších výsledků však dosáhla 2-opt LK heuristika s nejvyšší časovou náročností. Dosáhla tak dobrých řešení, že nakonec předčila i genetický algoritmus, který však pro menší úlohy dával lepší řešení.

Velmi rychlá, avšak špatných hodnot dosahující metoda minimální kostry by mohla být vylepšena použitím, v teoretické části zmíněné, Christofidovy metody, která by ovšem rapidně navýšila výpočetní čas.

Metoda výhodnostních čísel dopadla vždy lépe, co se týče hodnoty účelové funkce, než hladový algoritmus. Obě tyto metody jsou založeny na podobném principu. Nicméně metoda výhodnostních čísel je výpočetně náročnější a výpočet jí trvá déle.

Metaheuristiky ACO a BCO nedosahovaly dobrých výsledků. Pro metodu ACO by šlo rychlost řešení vylepšit vyšší koncentrací feromonu na hranách použitých některou z heuristik. Pro metodu BCO jsme již aplikovali zlepšování pomocí jednoho kroku 2-opt heuristiky a inicializací tancujících včel, jejichž cesty byly dány heuristikou nejbližšího souseda. Mohli bychom řešení vylepšit aplikací 3-opt LK heuristiky.

Nejlepší mezi metaheuristikami byly TA a SIAM, které dosahovaly velmi podobných výsledků. Metoda SIAM dosahuje nejlepších řešení. Pro tuto metodu bychom mohli dále implementovat 3-opt LK heuristiku a pro její zrychlení inicializovat řešení libovolnou rychlou heuristikou. Obdobné platí i pro TA metaheuristiku. Pro GA bychom mohli vytvořit další přístupy a podle nich volit okolní řešení pro větší škálu řešení.

Porovnání pořadí všech implementovaných metod mezi sebou uvádíme v tabulce 3.6. Vycházíme z obrázků uvedených v kapitole 3.9.1 a příloze C.

Tabulka 3.5: Pořadí heuristik a metaheuristik podle času a hodnoty účelové funkce ($Z$).

| Heuristiky | | | Metaheuristiky | | |
|---|---|---|---|---|---|
| Metoda | Čas | $Z$ | Metoda | Čas | $Z$ |
| Nejbližší soused | 2. | 3. | TA | 2. | 2. |
| Hladový algoritmus | 3. | 4. | SIAM | 3. | 1. |
| Výhodnostní čísla | 4. | 2. | GA | 1. | 3. |
| Metoda MST | 1. | 5. | ACO | 5. | 4. |
| 2-opt LK | 5. | 1. | BCO | 4. | 5. |

Tabulka 3.6: Pořadí implementovaných metod podle času a hodnoty účelové funkce ($Z$).

| Metoda | Čas | $Z$ |
|---|---|---|
| Nejbližší soused | 2. | 7. |
| Hladový algoritmus | 3. | 9. |
| Výhodnostní čísla | 4. | 6. |
| Metoda MST | 1. | 10. |
| 2-opt LK | 5. | 3. |
| TA | 7. | 2. |
| SIAM | 8. | 1. |
| GA | 6. | 4. |
| ACO | 10. | 5. |
| BCO | 9. | 8. |

# Závěr

Práci jsme zahájili definováním problému obchodního cestujícího se všemi náležitostmi pomocí teorie grafů. Uvádíme též zdroj dat, tedy hvězdnou mapu.

V teoretické části uvádíme okrajově 3 exaktní metody, 9 heuristik a 10 metaheuristik. Dále uvádíme už jen výčet základních myšlenek pěti metaheuritik následovaný výčtem několika dalších.

V praktické části představujeme prostředí jazyka R, které používáme pro implementaci vybraných metod z teoretické části. Jazyk R byl vybrán v návaznosti na výuku katedry ekonometrie, kde dominuje výuka tohoto jazyka. Zpětně však zpochybňujeme volbu jazyka a volili bychom vzhledem k náročnosti výpočtů spíše rychlejší jazyk MATLAB. Nicméně pro představení jednotlivých metod na příkazové řádce jazyka R shledáváme pro výuku jako dostačující a vhodný, pokud nebudou řešeny velké problémy. Mnohé aplikované přístupy nalézají řešení s rozumnou rychlostí.

Podařilo se nám zpracovat 3D data a zhotovit funkce pro práci s takto upraveným problémem obchodního cestujícího. Uvádíme dva přístupy vykreslování grafů, kde oba mají své výhody a nevýhody. Pomocí balíčku „scatterplot3d“ převádíme 3D rozměr na 2D rozměr, což umožňuje bezproblémové vložení do papírového dokumentu. Pomocí balíčku „rgl“ již není tak jednoduché a přehledné vytvořit graf vhodný pro papírový dokument, ovšem v prostředí RStudia lze mnohem lépe rozpoznat používané hrany pro přechody pomocí otáčení.

Uvádíme též další funkce pro zpříjemnění práce s jednotlivými heuristikami a metaheuristikami. Z heuristik jsme implementovali:

- metodu nejbližšího souseda,
- hladový algoritmus,
- metodu výhodnostních čísel,
- metodu minimální kostry,
- metodu výměn Lin – Kerninghen 2-opt.

Z metaheuristik jsme k implementaci vybrali následující metody:

- prahovou akceptaci,
- simulaci žíhání oceli,
- genetický algoritmus,
- optimalizaci mravenčí kolonií,
- optimalizaci včelí kolonií.

Jelikož efektivita metaheuristik záleží na vstupních parametrech, věnovali jsme se v rámci práce také jejich ladění se smíšenými výsledky. Věříme, že při hlubším prozkoumání parametrů bychom mohli dosáhnout lepších výsledků. Navrhujeme tedy k budoucímu hlubšímu

prozkoumání rozsáhlejší simulační studií.

V části popisující výsledky výpočetních experimentů jsme porovnali všechny implementované metody mezi sebou i podle kategorií heuristik a metaheuristik. Pro heuristiky jsme použili několik datasetů a největší z nich čítal na 5000 hvězd. Pro metaheuristiky jsme z důvodu výpočetních časů a opakování výpočtů metaheuristik (často se aplikuje náhoda, kterou bylo třeba ošetřit více výpočty) zvolili maximální počet hvězd 150.

Nejlepší heuristika, co do hodnoty účelové funkce, byla vyhodnocena metoda výměny hran 2-opt Lin Kerninghen. Byla však i výpočetně nejnáročnější a trvala déle. Velmi dobře si vedla metoda nejbližšího souseda, které vyhovovalo 3D rozmístění uzlů.

Z implementovaných metaheuristik oceňujeme metaheuristiky prohledávají extrémy okolního řešení – metodu prahové akceptace a metodu simulovaného žíhání. Genetický algoritmus si zase vedl nejlépe s časem výpočtu a dosahoval nejlepších řešení pro malé úlohy. Optimalizace mravenčí a včelí kolonií však oproti těmto přístupům zůstávaly pozadu.

Při porovnání optimalizace mravenčí a včelí kolonií byla první zmíněná metoda lepší, ale zato pomalejší. Časové výpočty pro mravenčí kolonii však byly neúnosné. Pro budoucí aplikaci by mohlo být vhodné snížit množství mravenců na procházení grafu pro urychlení výpočtu v kombinaci s inicializací feromonu na hranách určených pomocí některé z heuristik.

Dále navrhujeme k implementaci další metaheuritiky a jejich hlubší prozkoumání.

# Seznam použité literatury


ABID, Malik Muneeb; MUHAMMAD, Iqbal, 2015.
Heuristic approaches to solve traveling salesman problem.
*TELKOMNIKA Indonesian Journal of Electrical Engineering*. Roč. 15, č. 2, s. 390–396.

ALEMAYEHU, Temesgen Seyoum; KIM, Jai-Hoon, 2017. Efficient nearest neighbor heuristic TSP algorithms for reducing data acquisition latency of UAV relay WSN. *Wireless Personal Communications*. Roč. 95, č. 3, s. 3271–3285.

APPLEGATE, David L; BIXBY, Robert E; CHVÁTAL, Vašek; COOK, William J, 2006. *The traveling salesman problem A Computational Study*. Princeton university press.

BAILER-JONES, CAL; RYBIZKI, J; FOUESNEAU, M; MANTELET, G; ANDRAE, R, 2018. Estimating distance from parallaxes. IV. Distances to 1.33 billion stars in Gaia data release 2. *The Astronomical Journal*. Roč. 156, č. 2, s. 58.

BALAS, Egon; TOTH, Paolo, 1983.
Branch and bound methods for the traveling salesman problem.

BIRBIL, Ş İlker; FANG, Shu-Cherng; SHEU, Ruey-Lin, 2004.
On the convergence of a population-based global optimization algorithm.
*Journal of global optimization*. Roč. 30, č. 2, s. 301–318.

BONYADI, Mohammad Reza; AZGHADI, Mostafa Rahimi; SHAH-HOSSEINI, Hamed, 2008.
Population-based optimization algorithms for solving the travelling salesman problem.
*Traveling Salesman Problem. In-Tech. Croatia*, s. 001–034.

COOK, William J., 2022a. *CONCORDE*.
Dostupné také z: `https://www.math.uwaterloo.ca/tsp/concorde.html`.

COOK, William J., 2022b. *TSP Star Tours*.
Dostupné také z: `http://www.math.uwaterloo.ca/tsp/star/index.html`.

COUTINHO, Walton Pereira; NASCIMENTO, Roberto Quirino do; PESSOA, Artur Alves; SUBRAMANIAN, Anand, 2016.
A branch-and-bound algorithm for the close-enough traveling salesman problem.
*INFORMS Journal on Computing*. Roč. 28, č. 4, s. 752–765.

ESA, 2019. *Gaia summary*. 2019-09.
Dostupné také z: `https://sci.esa.int/web/gaia/-/28820-summary`.

ESA, 2022. *Gaia release dataset*. 2022-04.
Dostupné také z: `https://www.cosmos.esa.int/web/gaia`.

GEEM, Zong Woo; KIM, Joong Hoon; LOGANATHAN, Gobichettipalayam Vasudevan, 2001. A new heuristic optimization algorithm: harmony search. *simulation*.
Roč. 76, č. 2, s. 60–68.

GOLDEN, Bruce; BODIN, Lawrence; DOYLE, T; STEWART JR, W, 1980.
Approximate traveling salesman algorithms. *Operations research*.
Roč. 28, č. 3-part-ii, s. 694–711.

GUPTA, Divya, 2013. Solving tsp using various meta-heuristic algorithms.
*International Journal of Recent Contributions from Engineering, Science & IT (iJES)*.
Roč. 1, č. 2, s. 22–26.

HAHSLER, Michael; HORNIK, Kurt, 2022. *TSP: Traveling Salesperson Problem (TSP)*.
Dostupné také z: `https://CRAN.R-project.org/package=TSP`.
R package version 1.2-0.

HALIM, A Hanif; ISMAIL, Idris, 2013.
Nonlinear plant modeling using neuro-fuzzy system with Tree Physiology Optimization.
In: *2013 IEEE Student Conference on Research and Developement*, s. 295–300.

HALIM, A Hanif; ISMAIL, Idris, 2019.
Tree Physiology Optimization in Benchmark Function and Traveling Salesman Problem.
*Journal of Intelligent Systems*. Roč. 28, č. 5, s. 849–871.

HAXHIMUSA, Yll; CARPENTER, Edward; CATRAMBONE, Joseph; FOLDES, David; STEFANOV, Emil; ARNS, Laura; PIZLO, Zygmunt, 2011.
2D and 3D traveling salesman problem. *The Journal of Problem Solving*.
Roč. 3, č. 2, s. 8.

HELD, Michael; KARP, Richard M, 1970.
The traveling-salesman problem and minimum spanning trees. *Operations Research*.
Roč. 18, č. 6, s. 1138–1162.

HELSGAUN, Keld, 2000.
An effective implementation of the Lin–Kernighan traveling salesman heuristic.
*European journal of operational research*. Roč. 126, č. 1, s. 106–130.

HUSSAIN, Abid; MUHAMMAD, Yousaf Shad; NAUMAN SAJID, M; HUSSAIN, Ijaz; MOHAMD SHOUKRY, Alaa; GANI, Showkat, 2017. Genetic algorithm for traveling salesman problem with modified cycle crossover operator.
*Computational intelligence and neuroscience*. Roč. 2017.

CHITTY, Darren M, 2017. Applying ACO to large scale TSP instances. In:
*UK Workshop on Computational Intelligence*, s. 104–118.

CHUDASAMA, Chetan; SHAH, SM; PANCHAL, Mahesh, 2011.
Comparison of parents selection methods of genetic algorithm for TSP. In:
*International Conference on Computer Communication and Networks CSI-COMNET-2011, Proceedings*, s. 85–87.

IHAKA, Ross; GENTLEMAN, Robert, 1996. R: a language for data analysis and graphics.
*Journal of computational and graphical statistics*. Roč. 5, č. 3, s. 299–314.

JACKOVICH, Petar D, 2019.
Solving the Traveling Salesman Problem Using Ordered-Lists.

JASSER, Muhammed Basheer; SARMINI, Mohamad; YASEEN, Rauf, 2014.
Ant Colony Optimization (ACO) and a Variation of Bee Colony Optimization (BCO) in Solving TSP Problem: A Comparative Study.
*International Journal of Computer Applications*. Roč. 96, č. 9.

KIZILATEŞ, Gözde; NURIYEVA, Fidan, 2013.
On the nearest neighbor algorithms for the traveling salesman problem. In:
*Advances in Computational Science, Engineering and Information Technology*.
Springer, s. 111–118.

KOLMOGOROV, Vladimir, 2009.
Blossom V: a new implementation of a minimum cost perfect matching algorithm.
*Mathematical Programming Computation*. Roč. 1, č. 1, s. 43–67.

KUMBHARANA, Sharad N; PANDEY, Gopal M, 2013.
Solving travelling salesman problem using firefly algorithm.
*International Journal for Research in science & advanced Technologies*.
Roč. 2, č. 2, s. 53–57.

LARRANAGA, Pedro; KUIJPERS, Cindy M. H.; MURGA, Roberto H.; INZA, Inaki; DIZDAREVIC, Sejla, 1999. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial intelligence review*.
Roč. 13, č. 2, s. 129–170.

LIGGES, Uwe; MÄCHLER, Martin, 2003.
Scatterplot3d - an R Package for Visualizing Multivariate Data.
*Journal of Statistical Software*. Roč. 8, č. 11, s. 1–20.

MILLER, Clair E.; TUCKER, Albert W.; ZEMLIN, Richard A., 1960.
Integer programming formulation of traveling salesman problems.
*Journal of the ACM (JACM)*. Roč. 7, č. 4, s. 326–329.
Dostupné také z: `https://dl.acm.org/doi/pdf/10.1145/321043.321046`.

MITTELMANN, Hans, 2022. *Neos server*. 2022-03.
Dostupné také z: `https://neos-server.org/neos/solvers/co:concorde/TSP.html`.

MURDOCH, Duncan; ADLER, Daniel, 2022.
*RGL - 3D visualization device system for R using OpenGL*. 2022-04. Dostupné také z: `https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/dist`.

NASH, David, 2022. *The Astronomy Nexus: HYG database*. 2022-03.
Dostupné také z: `http://www.astronexus.com/hyg`.

NILSSON, Christian, 2003. Heuristics for the traveling salesman problem.
*Linkoping University*. Roč. 38, s. 00085–9.

OSABA, Eneko; YANG, Xin-She; DIAZ, Fernando; LOPEZ-GARCIA, Pedro; CARBALLEDO, Roberto, 2016. An improved discrete bat algorithm for symmetric and asymmetric traveling salesman problems.
*Engineering Applications of Artificial Intelligence*. Roč. 48, s. 59–71.

PELIKÁN, Jan, 2001. *Diskrétní modely v operačním výzkumu.* Praha: Professional Publishing.

POTVIN, Jean-Yves, 1996. Genetic algorithms for the traveling salesman problem. *Annals of Operations Research.* Roč. 63, č. 3, s. 337–370.

RAZALI, Noraini Mohd; GERAGHTY, John et al., 2011. Genetic algorithm performance with different selection strategies in solving TSP. In: *Proceedings of the world congress on engineering.* Sv. 2, s. 1–6. Č. 1.

RDOCUMNTATION, 2022. *dist: Distance Matrix Computation.* Dostupné také z: `https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/dist`.

REINELT, Gerhard, 2003. *The traveling salesman: computational solutions for TSP applications.* Springer.

RENAUD, Jacques; BOCTOR, Fayez F; LAPORTE, Gilbert, 1996. A fast composite heuristic for the symmetric traveling salesman problem. *INFORMS Journal on computing.* Roč. 8, č. 2, s. 134–143.

SAJI, Yassine; RIFFI, Mohammed Essaid, 2016. A novel discrete bat algorithm for solving the travelling salesman problem. *Neural Computing and Applications.* Roč. 27, č. 7, s. 1853–1866.

SHI, Xiaohu H; LIANG, Yanchun Chun; LEE, Heow Pueh; LU, C; WANG, QX, 2007. Particle swarm optimization-based algorithms for TSP and generalized TSP. *Information processing letters.* Roč. 103, č. 5, s. 169–176.

SKISCIM, Christopher C; GOLDEN, Bruce L, 1983. *Optimization by simulated annealing: A preliminary computational study for the tsp.* Tech. zpr. Institute of Electrical a Electronics Engineers (IEEE).

SNYDER, Lawrence V; DASKIN, Mark S, 2006. A random-key genetic algorithm for the generalized traveling salesman problem. *European journal of operational research.* Roč. 174, č. 1, s. 38–53.

TEODOROVIĆ, Dušan, 2009. Bee colony optimization (BCO). In: *Innovations in swarm intelligence.* Springer, s. 39–60.

TINÓS, Renato; HELSGAUN, Keld; WHITLEY, Darrell, 2018. Efficient recombination in the Lin-Kernighan-Helsgaun traveling salesman heuristic. In: *International Conference on Parallel Problem Solving from Nature*, s. 95–107.

WONG, Li-Pei; LOW, Malcolm Yoke Hean; CHONG, Chin Soon, 2008. A bee colony optimization algorithm for traveling salesman problem. In: *2008 Second Asia International Conference on Modelling & Simulation (AMS)*, s. 818–823.

WONG, Li-Pei; LOW, Malcolm Yoke Hean; CHONG, Chin Soon, 2009. An efficient bee colony optimization algorithm for traveling salesman problem using frequency-based pruning. In: *2009 7th IEEE International Conference on Industrial Informatics*, s. 775–782.

ZHAN, Shi-hua; LIN, Juan; ZHANG, Ze-jun; ZHONG, Yi-wen, 2016. List-based simulated annealing algorithm for traveling salesman problem. *Computational intelligence and neuroscience*. Roč. 2016.

ZHANG, Sicheng; WONG, TN, 2018. Integrated process planning and scheduling: an enhanced ant colony optimization heuristic with parameter tuning. *Journal of Intelligent Manufacturing*. Roč. 29, č. 3, s. 585–601.

# Přílohy

# A. Ukázka skriptů

## A.1 Kód hladového algoritmu

```
Hladovy <- function(vzdalenost){
  n <- dim(vzdalenost)[1]
  Z <- 0
  k <- 0
  i <- 0
  j <- 0
  cesta <- data.frame(0,0,0)
  colnames(cesta) <- c("z", "do", "cena")
  stupen = rep(0,n)
  vzdalenost <- data.frame(as.matrix(vzdalenost) + diag(Inf, nrow = n, ncol = n))
  colnames(vzdalenost) <- 1:n
  vzdal_sort <- subset(melt(as.matrix(vzdalenost)), value!=Inf)
  vzdal_sort <- vzdal_sort[order(vzdal_sort$value),1:3]
  rownames(vzdal_sort) <- 1:(n^2-n)

  while (k < (n-1)){
    mini <- vzdal_sort[1,]
    i <- as.numeric(mini[1,1])
    j <- as.numeric(mini[1,2])
    stupen[i] = stupen[i] + 1
    stupen[j] = stupen[j] + 1
    continue = TRUE
    continue2 = TRUE
    if((stupen[i] == 2) && (stupen[j] == 2)){
      tail <- i
      head <- j
      while(continue == TRUE){
        if(head %in% cesta[,1]){
          next_node = cesta[which(cesta[,1] == head),2]
          if(next_node == tail){
            vzdal_sort <- vzdal_sort[-1,]
            stupen[i] = stupen[i] - 1
            stupen[j] = stupen[j] - 1
            continue = FALSE
            continue2 = FALSE
          }
          else{
            continue = TRUE
            head = next_node
          }
        }
        else{continue = FALSE}
```

```
      } # end while
    }
    if (continue2 == FALSE){next}
    k <- k+1
    cesta[k,1] <- i
    cesta[k,2] <- j
    cesta[k,3] <- vzdalenost[i,j]
    Z = Z + vzdalenost[i,j]
    vzdal_sort <- vzdal_sort[-1,]
    vzdal_sort <- vzdal_sort[-which(vzdal_sort[,1] == i),]
    vzdal_sort <- vzdal_sort[-which(vzdal_sort[,2] == j),]
  } # end while

  start = setdiff(cesta$z, cesta$do)
  ted = start
  konec = setdiff(cesta$do, cesta$z)
  cesta2 = ted
  while(ted != konec){
    ted = cesta[which(cesta$z == ted),2]
    cesta2 = c(cesta2,ted)
  }
  Z = Z + vzdalenost[konec,start]
  return(list(Z = Z, cesta = cesta2))
}
```

## A.2 Kód metody výhodnostních čísel

```
VyhodCisla <- function(vzdalenost){
  n <- dim(vzdalenost)[1]
  S <- data.frame()
  k = 0
  Z = 0
  i = 0
  j = 0
  cesta <- data.frame(0,0,0)
  colnames(cesta) <- c("z", "do", "cena")
  stupen = rep(0,n)
  for (i in 1:n){
    for (j in 1:n){
      if (i==j || i == 1 || j == 1){
        S[i,j] = -Inf
        next
      }
      S[i,j] = vzdalenost[i,1] + vzdalenost[1,j] - vzdalenost[i,j]
    }
  }
```

```
colnames(S) <- 1:n
S_sort <- subset(melt(as.matrix(S)), value!=-Inf)
S_sort <- S_sort[order(S_sort$value, decreasing = TRUE),1:3]
rownames(S_sort) <- 1:(n^2-3*n+2)
while (k < (n-2)){
  maxi <- S_sort[1,]
  i <- as.numeric(maxi[1,1])
  j <- as.numeric(maxi[1,2])
  stupen[i] = stupen[i] + 1
  stupen[j] = stupen[j] + 1
  continue = TRUE
  continue2 = TRUE
  if((stupen[i] == 2) && (stupen[j] == 2)){
    tail <- i
    head <- j
    while(continue == TRUE){
      if(head %in% cesta[,1]){
        next_node = cesta[which(cesta[,1] == head),2]
        if(next_node == tail){
          S_sort <- S_sort[-1,]
          stupen[i] = stupen[i] - 1
          stupen[j] = stupen[j] - 1
          continue = FALSE
          continue2 = FALSE
        }else{
          continue = TRUE
          head = next_node
        }
      }else{continue = FALSE}
    } # end while
  }
  if (continue2 == FALSE){
    next
  }
  k <- k+1
  cesta[k,1] <- i
  cesta[k,2] <- j
  cesta[k,3] <- vzdalenost[i,j]
  Z = Z + vzdalenost[i,j]
  S_sort <- S_sort[-1,]
  S_sort <- S_sort[-which(S_sort[,1] == i),]
  S_sort <- S_sort[-which(S_sort[,2] == j),]
}
start = setdiff(cesta$z, cesta$do)
ted = start
konec = setdiff(cesta$do, cesta$z)
cesta2 = ted
while(ted != konec){
  ted = cesta[which(cesta$z == ted),2]
```

```
    cesta2 = c(cesta2,ted)
  }
  cesta2 <- c(1,cesta2)
  Z = Z + vzdalenost[konec,1] + vzdalenost[1,start]
  return(list(Z = Z, cesta = cesta2))
}
```

## A.3 Kód genetického algoritmu

```
GeneticAlg <- function(vzdalenost, ex1 = 8, ex2 = 6,
                       coefmin = 1/7, coefmax = 1/2,
                       prst = 0.5, vahy_mutaci = c(0.6,0.2,0.2),
                       parR = 100, strop_nezmeny = 200){
  if(ex1 < ex2){stop("ex1 musi byt vyssi nez ex2")}
  if(prst > 1 || prst < 0){
    stop("prst je pravdepodobnost provedeni mutace v intervalu (0,1)")
  }
  n <- dim(vzdalenost)[1]
  N <- 2^ex1
  Ztka <- numeric(N)
  cesty <- matrix(0,nrow = N, ncol = n)
  for (i in 1:(N-1)){
    reseni <- NahodReseni(vzdalenost)
    Ztka[i] <- reseni$Z
    cesty[i,] <- reseni$cesta
  }
  greedy <- Hladovy(vzdalenost)
  Ztka[N] <- greedy$Z
  cesty[N,] <- greedy$cesta
  fitness <- which.min(Ztka)
  Z_nej <- Ztka[fitness]
  cesta_nej <- cesty[fitness,]
  R <- n * parR
  r <- 0
  kdy_naposled <- 0

  while(r < R && kdy_naposled < strop_nezmeny){
    rodice <- VyberTurnaj(Ztka, N, ex2)
    deti <- KrizeniPMX(cesty, rodice, n, coefmin, coefmax)
    mam_mutovat <- rbinom(1,1,prst)
    if(mam_mutovat){
      deti <- MutujCestu(cesty = deti, n, p = vahy_mutaci)
    }
    Z_deti <- VypoctiZtka(deti,vzdalenost)
    cesty <- rbind(cesty, deti)
    Ztka <- c(Ztka, Z_deti)
```

```
    poradi <- order(Ztka)
    cesty <- cesty[poradi[1:N],]
    Ztka <- Ztka[poradi[1:N]]
    Z_novenej <- Ztka[1]
    cesta_novanej <- cesty[1,]
    if(Z_novenej == Z_nej){
        kdy_naposled <- kdy_naposled + 1
    }else{
      Z_nej <- Z_novenej
      cesta_nej <- cesta_novanej
      kdy_naposled <- 0
    }
    r <- r + 1
  }
  return(list(Z = Z_nej, cesta = cesta_nej))
}
```

## A.4 Kód křížení PMX

```
KrizeniPMX <- function(cesty, rodice, n, coefmin = 1/7, coefmax = 1/2){
  deti <- matrix(ncol = n)
  deti <- deti[-1,]

  for(i in 1:dim(rodice)[1]){
    cesty[rodice[i,"otec"],]
    cesty[rodice[i,"matka"],]

    min_jadro <- max(floor(n*coefmin),2)
    max_jadro <- ceiling(n*coefmax)

    j1 <- sample(1:(n-min_jadro),1)
    if((j1+min_jadro) == min(n,j1+max_jadro)){
      j2 = n
    }
    else{
      j2 <- sample((j1+min_jadro):min(n,j1+max_jadro),1)
    }

    jadro <- j1:j2
    nejadro <- setdiff(1:n, jadro)

    jadro_o <- cesty[rodice[i, "otec"],j1:j2]
    jadro_m <- cesty[rodice[i,"matka"],j1:j2]
    names(jadro_o) <- as.character(jadro_m)
    names(jadro_m) <- as.character(jadro_o)
```

```
    if(j1 == 1){zac <- c()
    }else{zac <- 1:(j1-1)}

    if(j2 == n){kon <- c()
    }else{kon <- (j2+1):n}

    dite_1 <-  c(cesty[rodice[i,"matka"],zac],
                 cesty[rodice[i, "otec"],j1:j2],
                 cesty[rodice[i,"matka"],kon])
    dite_2 <-  c(cesty[rodice[i, "otec"],zac],
                 cesty[rodice[i,"matka"],j1:j2],
                 cesty[rodice[i, "otec"],kon])
    while(length(unique(dite_1)) < n){
      ind <- nejadro[which(is.element(dite_1[nejadro], dite_1[jadro]))]
      dite_1[ind] <- jadro_m[as.character(dite_1[ind])]
    }
    while(length(unique(dite_2)) < n){
      ind <- nejadro[which(is.element(dite_2[nejadro], dite_2[jadro]))]
      dite_2[ind] <- jadro_o[as.character(dite_2[ind])]
    }
    deti <- rbind(deti, dite_1, dite_2)
  }
  rownames(deti) <- NULL
  return(deti)
}
```

## A.5 Kód optimalizace včelí kolonie

```
BCO <- function(vzdalenosti, eps = 1e-10,
                howtheta = "Zweighted", inic_W = 30,
                K = 20, povolit_2opt = TRUE,
                alpha = 1, beta = 8, lambda = 0.9,
                R = 100, nezlepseno = 10){
  n <- dim(vzdalenosti)[1]
  eta <- 1/(vzdalenosti+eps)
  waggle <- as.data.frame(matrix(1:(n+3), nrow = 1))
  colnames(waggle) <- c(paste0("cesta", 1:n), "tanecnik", "Z", "D")
  vcely_nej <- matrix(0, nrow = n, ncol = n)
  Zvcely_nej <- rep(Inf, n)
  for(i in 1:n){
    NN <- NejSoused(vzdalenosti, start = i)
    waggle <- rbind(waggle, c(NN$cesta, i, NN$Z, inic_W))
    vcely_nej[i, ] <- NN$cesta
    Zvcely_nej[i] <- NN$Z
  }
  waggle <- waggle[-1,]
```

```
cesta_nej <- c()
Z_nej <- Inf
r <- 1
aktualizace <- 0
waggle_neprazdny <- TRUE
while(r < R && aktualizace < nezlepseno && waggle_neprazdny){
  doslo_ke_zmene <- FALSE
  for(k in 1:n){
    indexy <- which(waggle$tanecnik==k)
    waggle[indexy, "D"] <- waggle[indexy, "D"]-1
    odstran <- indexy[waggle[indexy,"D"] <= 0]
    if(length(odstran) >0){
      waggle <- waggle[-odstran,]
    }
    i <- k
    povol_seznam <- setdiff(1:n,i)
    cesta <- i
    if(dim(waggle)[1] == 0){
      waggle_neprazdny <- FALSE
      break
    }
    theta <- switch(howtheta,
          "uniform" = sample(1:dim(waggle)[1], size = 1),
          "Zweighted" = sample(1:dim(waggle)[1], size = 1, prob = 1/waggle$Z))
    theta_cesta <- waggle[theta, 1:n]
    while(length(cesta)<n-1){
      pozice <- which(theta_cesta == i) # kde je i
      F_i <- theta_cesta[(pozice %% n) + 1] # co nasleduje po i
      F_je_v_povol <- is.element(F_i,povol_seznam)
      rho_ij <- (1-lambda*F_je_v_povol)/length(setdiff(povol_seznam,F_i))
      if(F_je_v_povol){rho_ij[which(povol_seznam==F_i)] <- lambda}
      prsti <- (rho_ij)^alpha * (eta[i,povol_seznam])^beta
      j <- sample(povol_seznam, size = 1, prob = prsti)
      povol_seznam <- setdiff(povol_seznam,j)
      cesta <- c(cesta,j)
      i <- j
    } # end while cesty
    cesta <- c(cesta, povol_seznam)
    Z <- VypoctiZtka(cesta,vzdalenosti)
    if(povolit_2opt == TRUE){
      nove <- LK_2_opt_1_step(vzdalenosti, cesta, Z)
      cesta <- nove$cesta
      Z <- nove$Z
    }
    if (Z < Zvcely_nej[k]){
      Zvcely_nej[k] <- Z
      vcely_nej[k,] <- cesta
      Pfcolony <- mean(1/waggle$Z)
      Pfi <- 1/Z
```

```
        Di <- K * Pfi/Pfcolony
        waggle <- rbind(waggle,
                        c(cesta, k, Z, Di))
      }
      if (Z < Z_nej){
        Z_nej <- Z
        cesta_nej <- cesta
        doslo_ke_zmene = TRUE
      }
    } # end for včelek
    if (doslo_ke_zmene){
      aktualizace <- 0
    } else {aktualizace <- aktualizace + 1}
    r <- r + 1
  } # end while hlavniho cyklu
  return(list(Z = Z_nej, cesta = cesta_nej))
}
```

# B. Tabulky výpočtů procedur

Tabulka B.1: Porovnání výpočetních časů a dosažených hodnot účelových funkcí ($Z$) heuristik na malém datasetu (vlevo) a na středně velkém datasetu (vpravo).

| $n$ | Metoda | Čas [s] | $Z$ [pc] |
|---|---|---|---|
| 20 | NN | < 0,01 | 353 |
| | Greedy | 0,01 | 379 |
| | Výhod. č. | 0,02 | 357 |
| | Metoda MST | < 0,01 | 386 |
| | 2-opt LK | 0,04 | 357 |
| 30 | NN | < 0,01 | 527 |
| | Greedy | 0,01 | 536 |
| | Výhod. č. | 0,07 | 518 |
| | Metoda MST | 0,03 | 559 |
| | 2-opt LK | 0,08 | 498 |
| 50 | NN | 0,02 | 971 |
| | Greedy | 0,03 | 1011 |
| | Výhod. č. | 0,14 | 931 |
| | Metoda MST | 0,03 | 1119 |
| | 2-opt LK | 0,34 | 896 |
| 75 | NN | 0,03 | 1456 |
| | Greedy | 0,04 | 1494 |
| | Výhod. č. | 0,34 | 1411 |
| | Metoda MST | 0,07 | 1739 |
| | 2-opt LK | 0,97 | 1366 |
| 100 | NN | 0,05 | 2027 |
| | Greedy | 0,10 | 2050 |
| | Výhod. č. | 0,61 | 1961 |
| | Metoda MST | 0,08 | 2421 |
| | 2-opt LK | 1,58 | 2008 |

| $n$ | Metoda | Čas [s] | $Z$ [pc] |
|---|---|---|---|
| 200 | NN | 0,21 | 7014 |
| | Greedy | 0,86 | 7341 |
| | Výhod. č. | 2,75 | 6629 |
| | Metoda MST | 0,15 | 7752 |
| | 2-opt LK | 14,17 | 6336 |
| 300 | NN | 0,39 | 10238 |
| | Greedy | 3,75 | 10810 |
| | Výhod. č. | 8,44 | 9838 |
| | Metoda MST | 0,25 | 11384 |
| | 2-opt LK | 23,76 | 9243 |
| 500 | NN | 1,05 | 15692 |
| | Greedy | 10,47 | 16837 |
| | Výhod. č. | 27,87 | 15576 |
| | Metoda MST | 0,57 | 18271 |
| | 2-opt LK | 104,73 | 15134 |
| 750 | NN | 2,54 | 20930 |
| | Greedy | 20,64 | 21447 |
| | Výhod. č. | 61,46 | 20347 |
| | Metoda MST | 0,92 | 24152 |
| | 2-opt LK | 313,00 | 19944 |
| 1000 | NN | 4,16 | 25583 |
| | Greedy | 56,51 | 25771 |
| | Výhod. č. | 138,31 | 24741 |
| | Metoda MST | 1,84 | 29218 |
| | 2-opt LK | 897,54 | 23814 |

Tabulka B.2: Porovnání heuristik na velkých úlohách z velkého datasetu.

| $n$ | Metoda | Čas [s] | $Z$ [pc] |
|---|---|---|---|
| 2000 | NN | 17,15 | 72080 |
| | Greedy | 417,53 | 76594 |
| | Výhod. č. | 816,90 | 70417 |
| | Metoda MST | 6,73 | 81015 |
| | 2-opt LK | 4096,14 | 67816 |
| 3000 | NN | 38,97 | 107393 |
| | Greedy | 1775,17 | 114508 |
| | Výhod. č. | 2760,44 | 104753 |
| | Metoda MST | 15,58 | 121395 |
| | 2-opt LK | 16348,10 | 101139 |
| 5000 | NN | 109,72 | 175568 |
| | Greedy | 8700,75 | 190527 |
| | Výhod. č. | 12458,41 | 173713 |
| | Metoda MST | 41,38 | 202559 |
| | 2-opt LK | 58912,39 | 165492 |

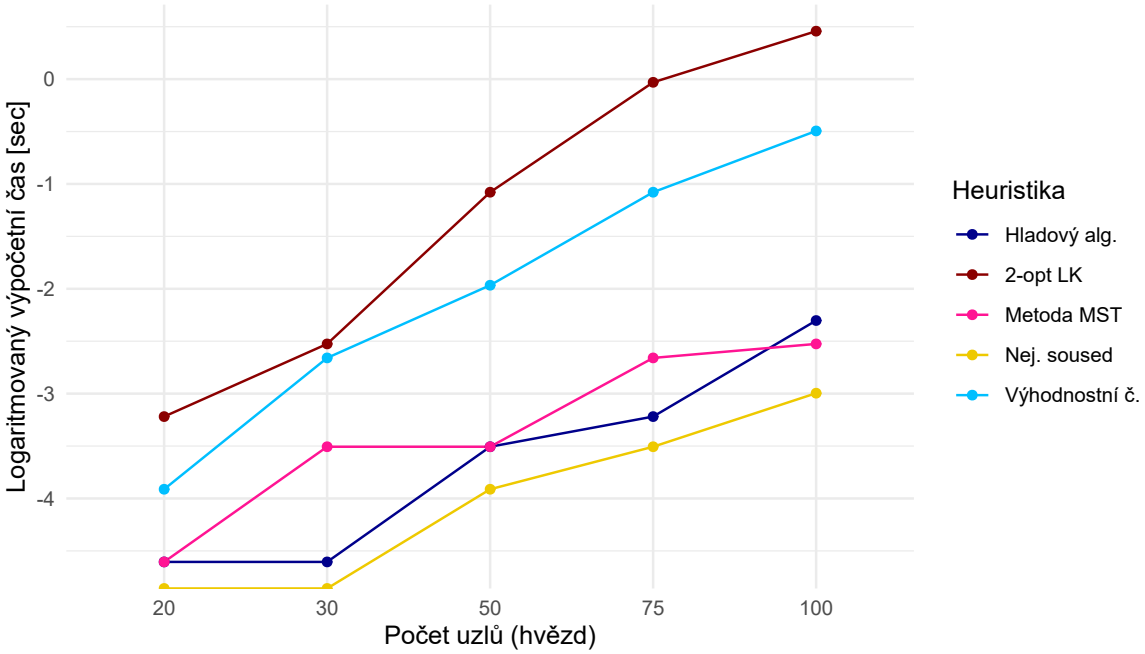
Tabulka B.3: Porovnání výpočetních časů a dosažených hodnot účelových funkcí ($Z$) metaheuristik na malých úlohách (vlevo) a na větších úlohách (vpravo) z malého datasetu.

| $n$ | Metoda | Čas [s] | $Z$ [pc] | $n$ | Metoda | Čas [s] | $Z$ [pc] |
|---|---|---|---|---|---|---|---|
| | TA | 1,18 | 198 | | TA | 47,37 | 1058 |
| | SIAM | 1,24 | 198 | | SIAM | 50,44 | 1071 |
| 10 | GA | 4,09 | 198 | 60 | GA | 45,11 | 1048 |
| | ACO | 2,81 | 198 | | ACO | 929,64 | 1100 |
| | BCO | 2,16 | 198 | | BCO | 461,38 | 1117 |
| | TA | 4,48 | 343 | | TA | 69,41 | 1224 |
| | SIAM | 4,83 | 343 | | SIAM | 68,58 | 1199 |
| 20 | GA | 6,28 | 344 | 70 | GA | 54,38 | 1163 |
| | ACO | 26,17 | 343 | | ACO | 1067,95 | 1187 |
| | BCO | 13,39 | 343 | | BCO | 397,66 | 1279 |
| | TA | 10,47 | 491 | | TA | 94,25 | 1470 |
| | SIAM | 11,06 | 493 | | SIAM | 92,72 | 1413 |
| 30 | GA | 10,17 | 491 | 80 | GA | 56,31 | 1440 |
| | ACO | 109,43 | 507 | | ACO | 2060,95 | 1502 |
| | BCO | 50,08 | 494 | | BCO | 708,30 | 1530 |
| | TA | 19,72 | 677 | | TA | 119,49 | 1625 |
| | SIAM | 20,69 | 672 | | SIAM | 112,73 | 1624 |
| 40 | GA | 23,65 | 671 | 90 | GA | 73,93 | 1686 |
| | ACO | 260,64 | 696 | | ACO | 3198,77 | 1720 |
| | BCO | 81,18 | 698 | | BCO | 1560,09 | 1710 |
| | TA | 34,73 | 852 | | TA | 139,50 | 1848 |
| | SIAM | 30,92 | 863 | | SIAM | 166,17 | 1866 |
| 50 | GA | 30,43 | 848 | 100 | GA | 88,06 | 1895 |
| | ACO | 581,06 | 897 | | ACO | 4436,77 | 1977 |
| | BCO | 199,33 | 886 | | BCO | 2318,81 | 1979 |

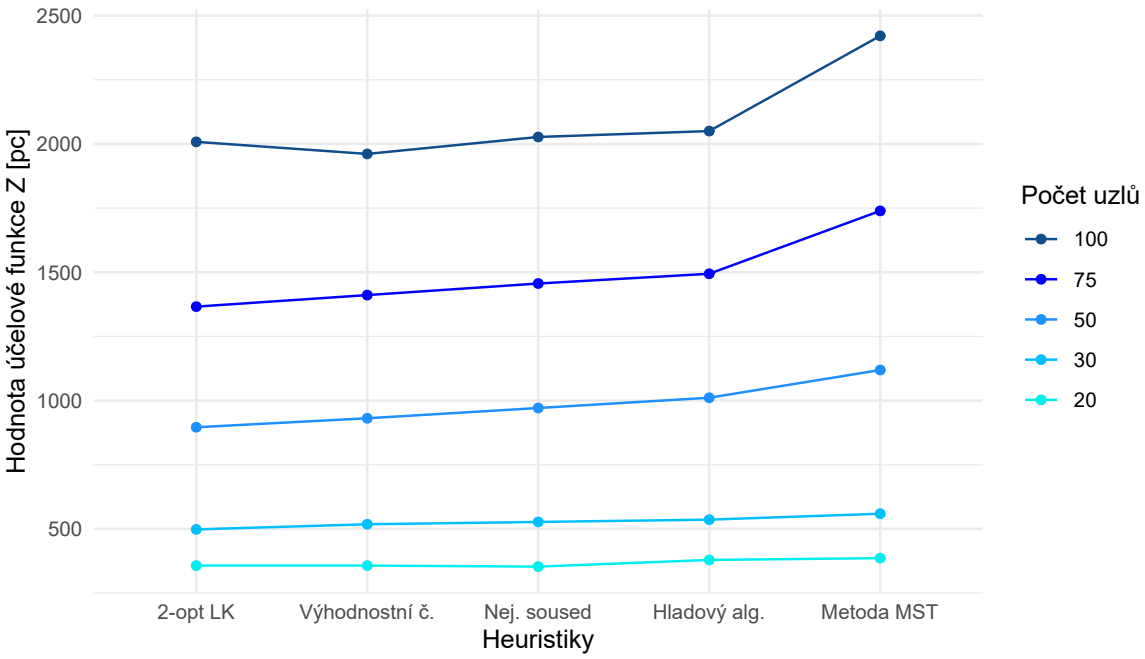
Tabulka B.4: Porovnání metaheuristik na velkých úlohách ze středně velkého datasetu.

| $n$ | Metoda | Čas [s] | $Z$ [pc] |
|---|---|---|---|
| 110 | TA | 189,19 | 3610 |
| | SIAM | 205,90 | 3612 |
| | GA | 199,00 | 3800 |
| | ACO | 3965,07 | 3798 |
| | BCO | 1919,89 | 4284 |
| 120 | TA | 294,17 | 3889 |
| | SIAM | 280,77 | 3767 |
| | GA | 220,78 | 3947 |
| | ACO | 4850,92 | 4069 |
| | BCO | 2003,23 | 4661 |
| 130 | TA | 321,95 | 3973 |
| | SIAM | 316,63 | 4055 |
| | GA | 155,64 | 4123 |
| | ACO | 7563,94 | 4276 |
| | BCO | 2184,67 | 4990 |
| 140 | TA | 330,27 | 4371 |
| | SIAM | 314,72 | 4385 |
| | GA | 236,97 | 4431 |
| | ACO | 8621,59 | 4566 |
| | BCO | 2851,30 | 5366 |
| 150 | TA | 367,42 | 4870 |
| | SIAM | 352,14 | 4866 |
| | GA | 303,33 | 5030 |
| | ACO | 9534,37 | 5067 |
| | BCO | 2096,13 | 5948 |

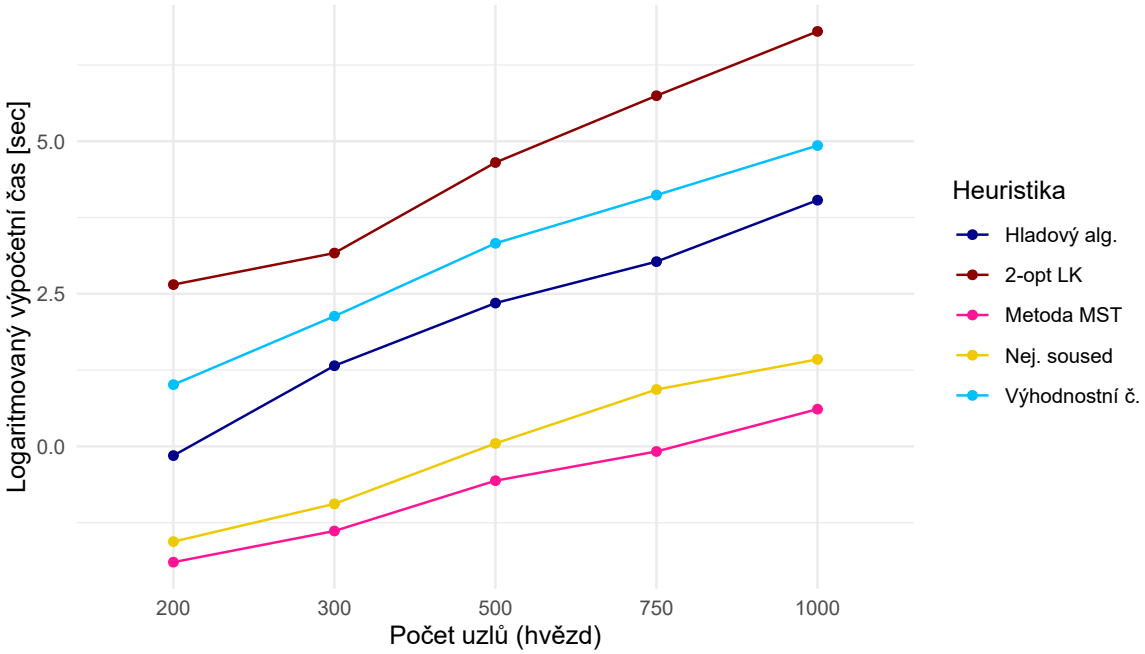
# C. Obrázky porovnání procedur



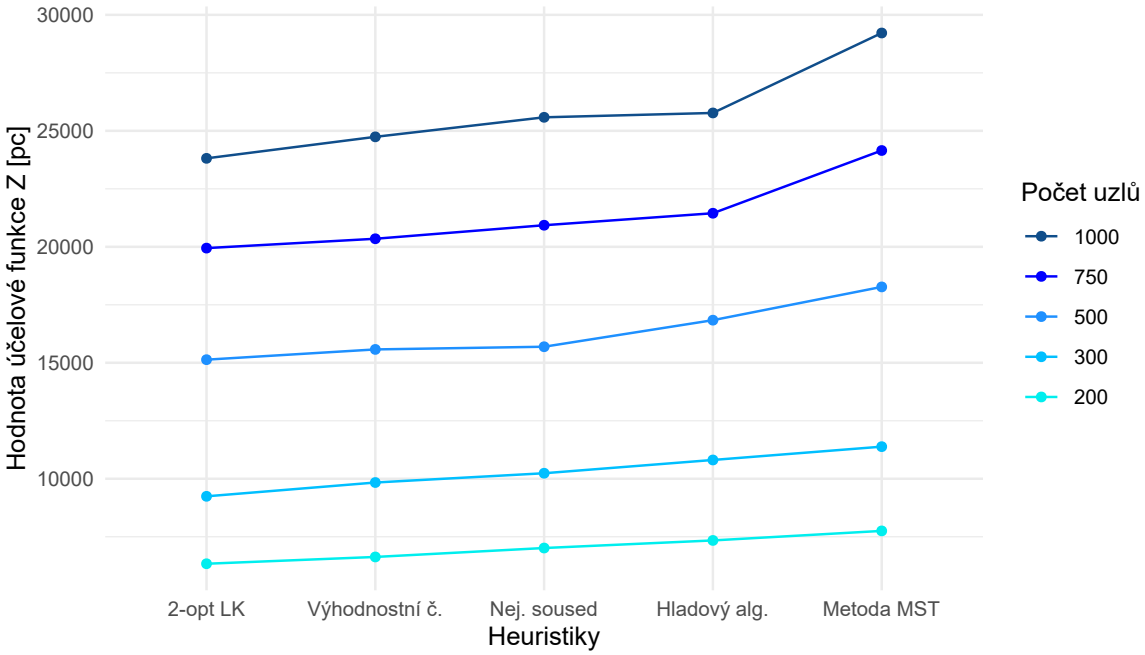

Obrázek C.1: Výpočetní čas v závislosti na počtu uzlů pro každou heuristiku zvlášť na malém datasetu.



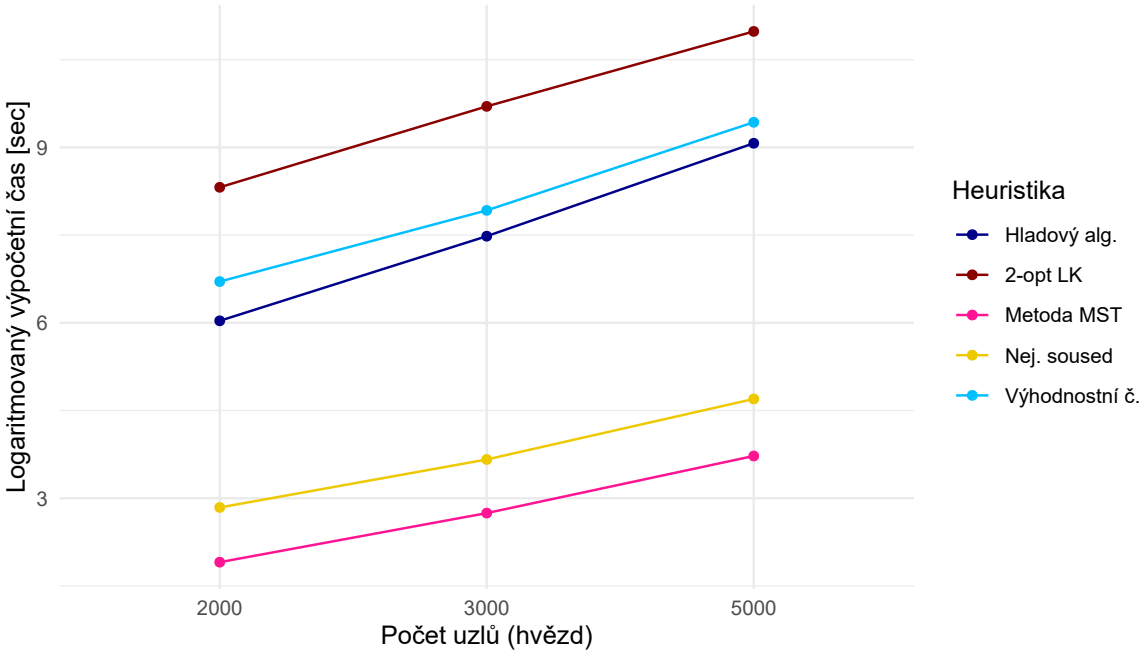

Obrázek C.2: Hodnoty účelových funkcí v závislosti na jednotlivých heuristikách pro každou sadu počtu uzlů z malého datasetu.



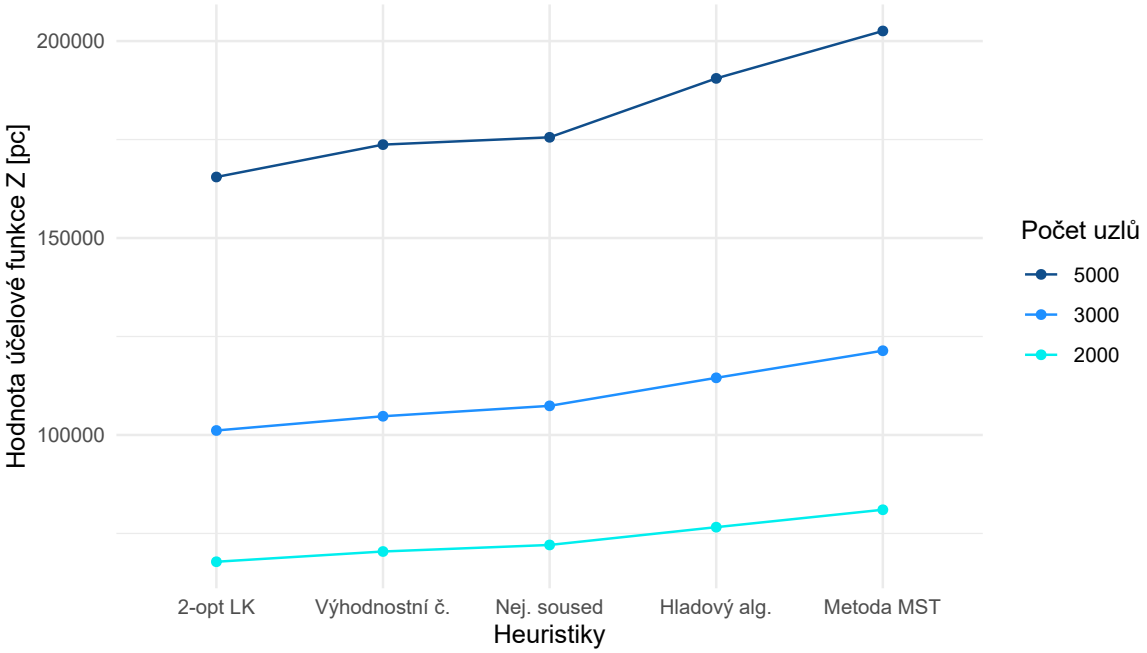

Obrázek C.3: Výpočetní čas v závislosti na počtu uzlů pro každou heuristiku zvlášť na středně velkých datasetech.



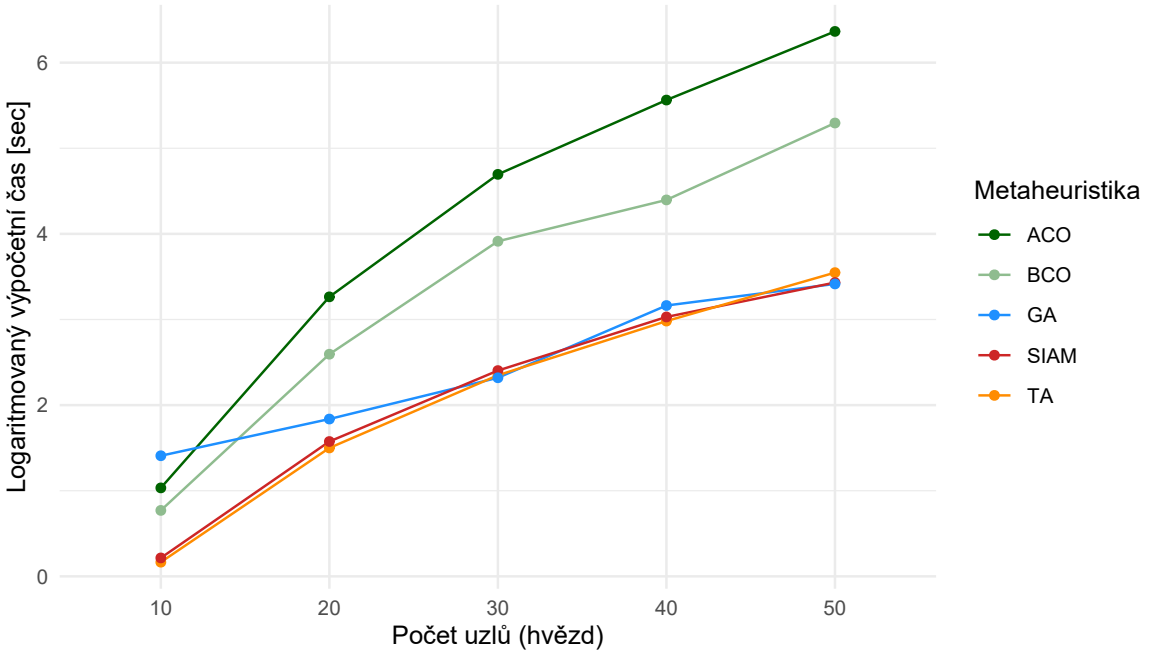

Obrázek C.4: Hodnoty účelových funkcí v závislosti na jednotlivých heuristikách pro každou sadu počtu uzlů ze středně velkého datasetu.



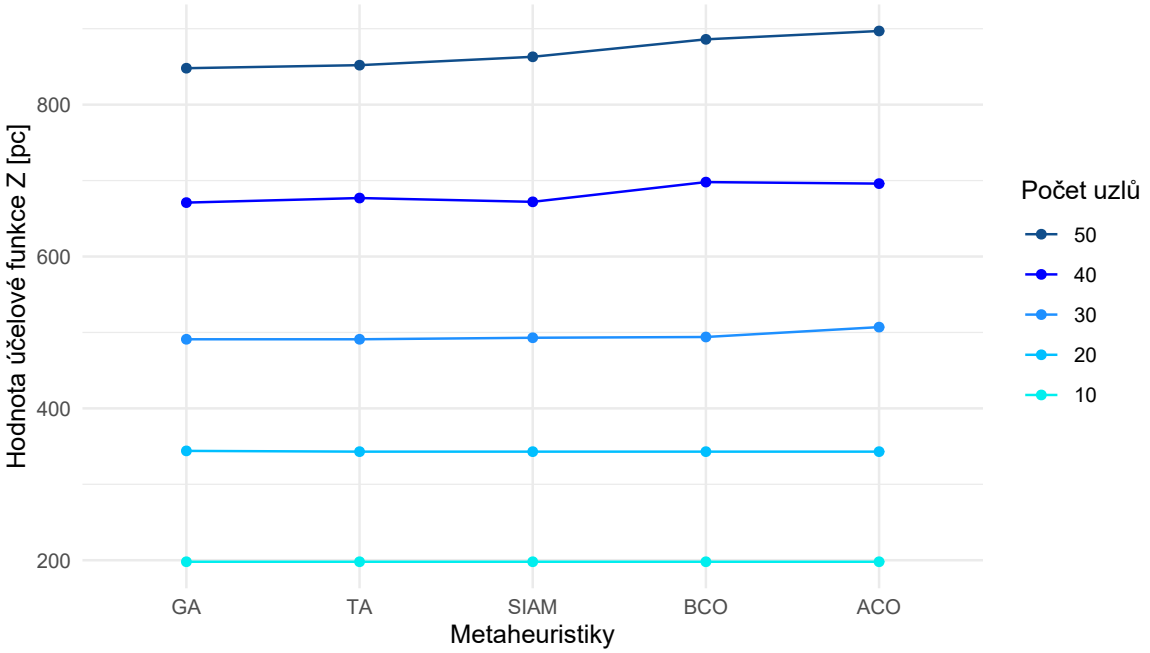

Obrázek C.5: Výpočetní čas v závislosti na menší počet uzlů pro každou heuristiku zvlášť na velkém datasetu.



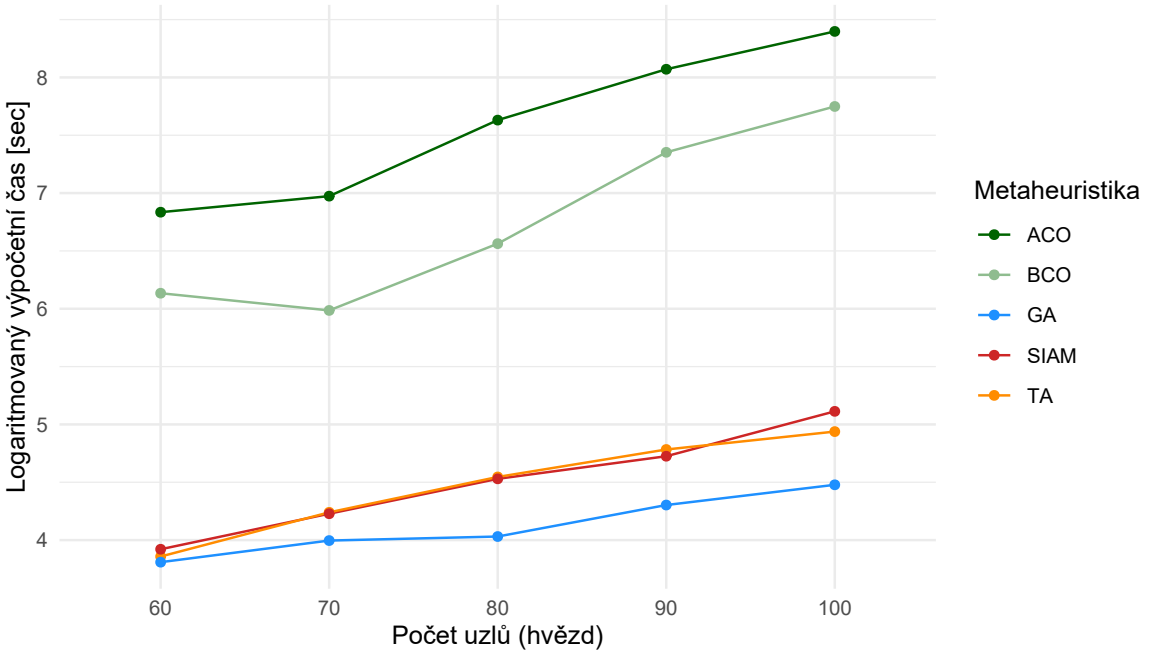

Obrázek C.6: Hodnoty účelových funkcí v závislosti na jednotlivých heuristikách pro menší sadu počtu uzlů z velkého datasetu.



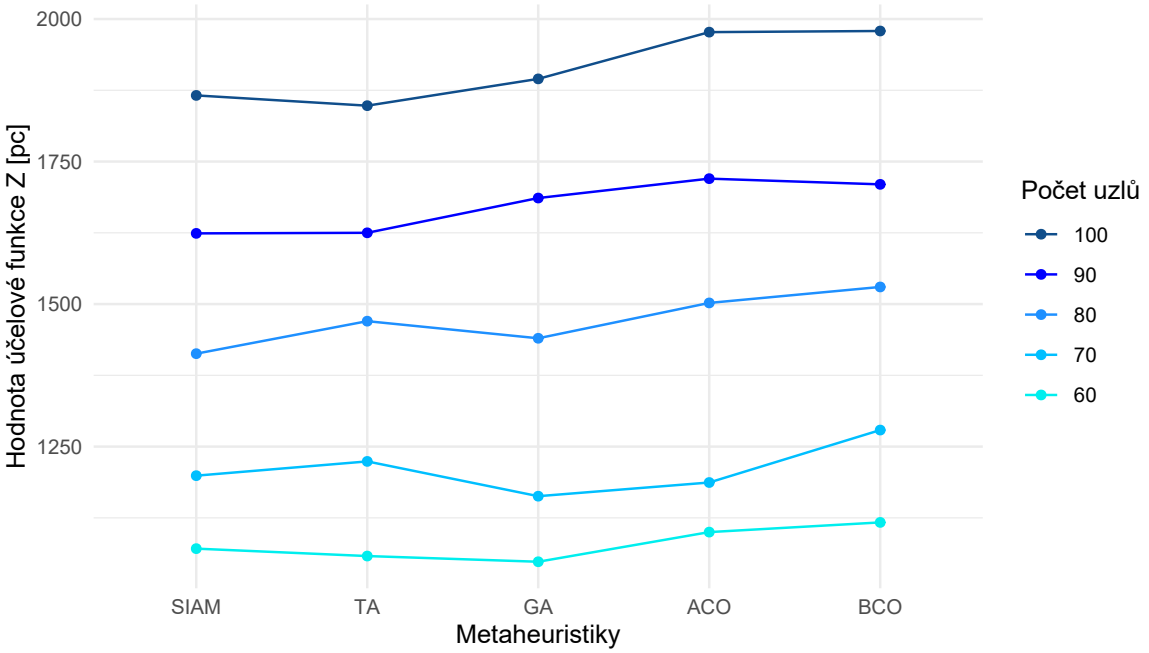

Obrázek C.7: Výpočetní čas v závislosti na malém počtu uzlů pro každou metaheuristiku zvlášť na malém datasetu.



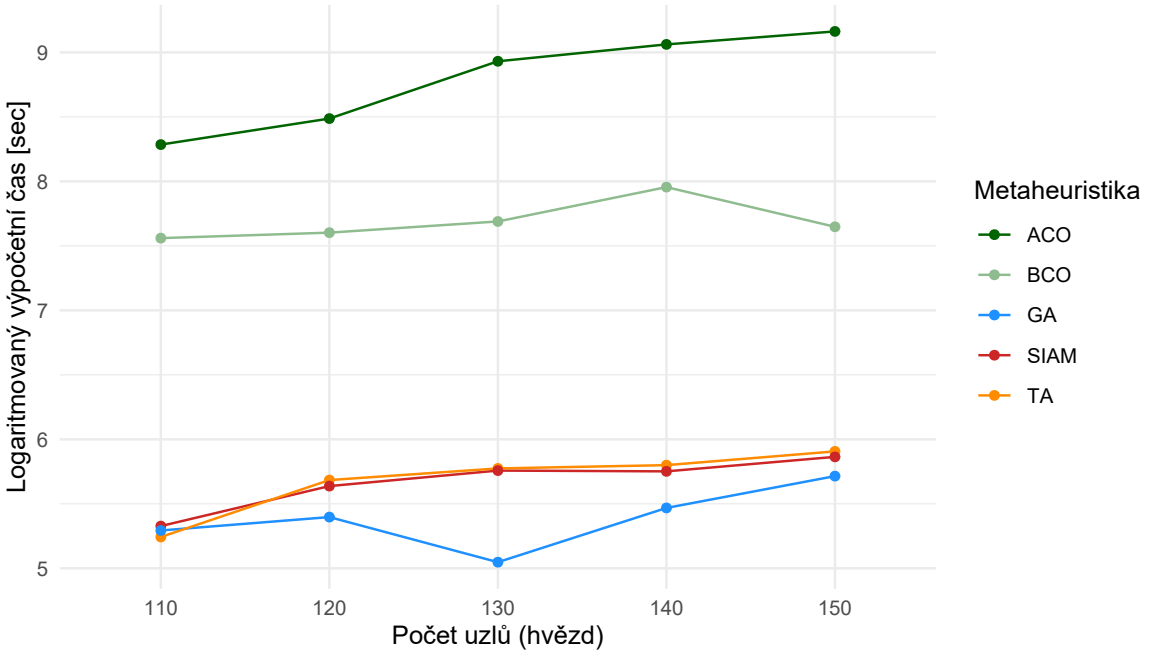

Obrázek C.8: Hodnoty účelových funkcí v závislosti na jednotlivých metaheuristikách pro malou sadu počtu uzlů z malého datasetu.



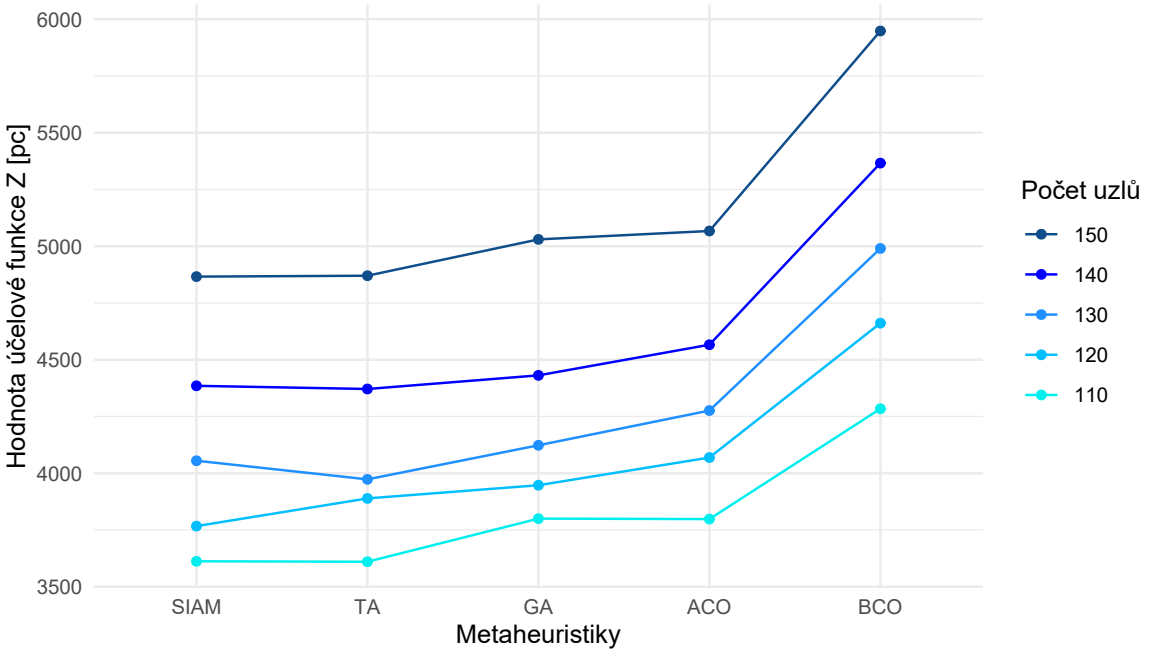

Obrázek C.9: Výpočetní čas v závislosti na větším počtu uzlů pro každou metaheuristiku zvlášť z malého datasetu.




Obrázek C.10: Hodnoty účelových funkcí v závislosti na jednotlivých metaheuristikách pro větší sadu počtu uzlů z malého datasetu.




Obrázek C.11: Výpočetní čas v závislosti na počtu uzlů pro každou metaheuristiku zvlášť ze středně velkého datasetu.




Obrázek C.12: Hodnoty účelových funkcí v závislosti na jednotlivých metaheuristikách pro každou sadu počtu uzlů ze středně velkého datasetu.