VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

PROBLÉM OBCHODNÍHO CESTUJÍCÍHO

TSP - TRAVELLING SALESMAN PROBLEM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **JAN ŘEZNÍČEK**
AUTHOR

VEDOUCÍ PRÁCE **doc. Ing. FRANTIŠEK ZBOŘIL, CSc.**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

146207

Ústav:	Ústav inteligentních systémů (UITS)
Student:	**Řezníček Jan**
Program:	Informační technologie
Specializace:	Informační technologie
Název:	**Problém obchodního cestujícího**
Kategorie:	Umělá inteligence
Akademický rok:	2022/23

Zadání:

1. Prostudujte zadanou literaturu a seznamte se s různými přístupy k řešení problému obchodního cestujícího (TSP - Travelling Salesman Problem).
2. Vyberte si alespoň dva různé přístupy.
3. Navrhněte programy pro řešení TSP těmito vybranými přístupy.
4. Navržené programy implementujte.
5. Proveďte experimenty s různými počty míst (příklady naleznete v posledních dvou odkazech na literaturu).
6. Zhodnoťte dosažené výsledky (časy řešení a získané délky cest).

Literatura:

- Blum, Ch.: Ant colony optimization: Introduction and recent trends, Physics of Life Reviews Elsevier, 2005
- Chandra, A., Natalia, Ch., Naro, A.: Comparative Solutions of Exact and Approximate Methods for Traveling Salesman Problem, Lámosakos, Medelín-Colombia, 2021
- Chauhan, Ch., Gupta, R. Kshitij, P.: Survey of Methods of Solving TSP along with its Implementation using Dynamic Programming Approach, International Journal of Computer Applications, 2012
- Ngassa, J. L., Kierkegaard J: ACO and TSP, Roskilde University, 2007
- Reilly, R., Tchimev, P.: Neural Network Approach to Solving the Traveling Salesman Problem, JSCS 19, 1, 2003

Při obhajobě semestrální části projektu je požadováno:
První dva body zadání

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

Vedoucí práce:	**Zbořil František, doc. Ing., CSc.**
Vedoucí ústavu:	Hanáček Petr, doc. Dr. Ing.
Datum zadání:	1.11.2022
Termín pro odevzdání:	10.5.2023
Datum schválení:	5.4.2023

Abstrakt

Práce se zaměřuje na implementaci algoritmů, které řeší problém obchodního cestujícího. Součástí je i uživatelské rozhraní s mapou pro import míst. Hlavní algoritmy, které jsou součástí práce jsou ACO a mnou vymyšlený a implementovaný algoritmus. ACO optimalizace, které zlepšují výsledky, jako jsou nastavení počátečních feromonů pomocí algoritmu nejbližších sousedů. Můj algoritmus funguje na principu postupného vylepšování cesty.

Abstract

The work focuses on implementing algorithms that solve the Traveling Salesman Problem. It also includes a user interface with a map for importing locations. The main algorithms that are part of the work are ACO and an algorithm that I have devised and implemented. ACO optimization improves results, such as setting initial pheromone levels using the nearest neighbor algorithm. My algorithm works on the principle of gradually improving the path.

Klíčová slova

ACO, optimalizace mravenčí kolonií, TSP, Problém obchodního cestujícího, 2-opt, 3-opt, λ-opt, Algoritmus nejbližších sousedů, Lin–Kernighan heuristika, Lin–Kernighan Helsgaun heuristika, LKH, Hladový algoritmus, Genetický algoritmus, Globální zámek interpretu, GIL

Keywords

ACO, Ant colony optimization, TSP, Travelling salesman problem, 2-opt, 3-opt, λ-opt, Nearest neighbor algorithm, Lin–Kernighan heuristic, Lin–Kernighan Helsgaun heuristic, LKH, Greedy algorithm, Genetic Algorithm, Global interpret lock, GIL

Citace

ŘEZNÍČEK, Jan. *Problém obchodního cestujícího*. Brno, 2023. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. František Zbořil, CSc.

Problém obchodního cestujícího

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doc. Ing. Františka V. Zbořila, Csc. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Jan Řezníček
5. května 2023

Poděkování

Chtěl bych poděkovat vedoucímu mé bakalářské práce, doc. Ing. Františku V. Zbořilovi, CSc. za odborné vedení a rady při zpracování této práce.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Problém obchodního cestujícího je obtížný diskrétní optimalizační problém, a pro více míst je v rozumném čase neřešitelný, pokud by se procházely všechny možnosti. V kapitole 2 popíši problém obchodního cestujícího. V kapitole 3 se seznámíme s algoritmy a heuristikami, které se využívají k řešení tohoto problému. Asi nejúspěšnější z nich je **Lin-Kernighan heuristic**. V kapitole 4 popíši základní teorii algoritmů, které jsem si vybral. Jedním z nich je **ACO**, a druhým je **MyAlgo**. V kapitole 5 se zaměřím na konkrétní implementaci těchto dvou algoritmů. V kapitole 6 se zaměřím na program jako celek s všemi pomocnými soubory. V kapitole 7 jsou popsány funkce uživatelského rozhraní, jako je například import míst z mapy nebo popsané jednotlivé funkce widgetů. V kapitole 8 se zaměřím na implementaci ACO a MyAlgo. Oba algoritmy budou testovány až do 1500 míst, a od 1500 míst se zaměřím pouze na MyAlgo. Testy jsem prováděl až do 22775 míst.

Kapitola 2

Problém obchodního cestujícího

Problém byl poprvé matematicky formulován v 19. století dvěma matematiky, z nichž jeden byl Sir William Rowam Hamilton a druhý Thomas Penyngton Kirkman. Nicméně předpokládá se, že obecnou podobu TSP poprvé studoval Karl Menger. Slovní definice problému může znít následovně: Je dána množina měst, neboli míst. Pro přesun z jednoho místa do druhého je třeba využít cestu, která má určitou vzdálenost. Každé místo má cestu ke všem ostatním místům. Podstatou problému je najít nejlepší možný způsob, jak navštívit všechna místa a vrátit se do výchozího místa, a to minimalizací cestovních nákladů nebo cestovní vzdálenosti. Matematicky to můžeme vyjádřit tak, že hledáme v ohodnoceném úplném grafu nejkratší hamiltonovskou kružnici. S rostoucím počtem míst roste počet potenciálních řešení, které je třeba prohledat. Celkový počet možných tras pokrývajících všechna města, která je třeba navštívit (lze je zapsat jako množinu proveditelných řešení TSP), roste s počtem měst n a je dán jako:

$$O = \frac{(n-1)!}{2}$$

Například pro 20 míst máme zhruba $1.2*10^{18}$ tras, které mohou být potenciálně optimálním řešením, a to je pouze pro 20 míst. Pro tak velký počet potenciálních řešení je nemožné prohledat všechny trasy v rozumném čase, i pro superpočítače. Proto se při řešení problému používají algoritmy, které neprohledávají všechny trasy, ale pouze ty více "atraktivní"pro algoritmus. Tyto algoritmy mohou skončit po určitém počtu iterací nebo po splnění nějaké podmínky, například 3-opt, když se řešení nezlepší. Tuto informaci jsem převzal z [2].

Kapitola 3

Přehled algoritmů pro řešení problému obchodního cestujícího

3.1 Nearest Neighbor Algorithm

Jde asi o nejjednodušší algoritmus pro řešení problému obchodního cestujícího. Časová složitost algoritmu je $O(n^2)$. Algoritmus by mohl vypadat následovně: Vytvořím si datovou strukturu, která bude ukládat navštívená místa. Zvolím si místo, ve kterém budu začínat, například náhodně. Uložím ho jako aktuální místo, ve kterém jsem, a vložím ho do struktury navštívených míst. Zvolím cestu do následujícího místa takovou, která je nejkratší možná a zároveň se místo, kam cesta vede, nenachází v navštívených místech. Následující místo zvolím jako aktuální a vložím jej do struktury pro navštívená místa. To opakuji, dokud nenavštívím všechna místa.

3.2 Greedy Algorithm

Je to jednoduchý algoritmus, který se používá v různých oblastech, jako je například optimalizace a teorie grafů. Jeho princip spočívá v tom, že v každém kroku algoritmu se vybere nejlepší (nebo nejhorší) možnost na základě aktuální situace a nevrací se zpět, aby se nějaká dřívější volba změnila. Tento postup může vést k rychlému a jednoduchému řešení problému, ale v některých případech může být neefektivní a vést k špatnému výsledku.

Jedním z příkladů použití Greedy algoritmu může být problém hledání nejkratší cesty v grafu. Algoritmus by se mohl rozhodnout v každém kroku pro hranu s nejmenší váhou a pokračovat tímto způsobem, dokud není nalezena cesta mezi dvěma uzly. Tento postup ale nemusí vést k nejkratší cestě. Dalším příkladem aplikace algoritmu může být problém obchodního cestujícího, kdy se postupně volí nejkratší cesty, které lze spojit a postupně spojují místa, dokud nejsou navštívena všechna místa.

3.3 2-opt

Algoritmus 2-opt je speciálním případem algoritmu λ-opt, kde v každém kroku jsou nahrazeny λ odkazy aktuální trasy novými λ odkazy takovým způsobem, aby byla dosažena kratší trasa. Jinými slovy, v každém kroku se dosáhne kratší trasy tím, že jsou odstraněny λ odkazy a výsledné cesty jsou spojeny novým způsobem, případně některé z nich jsou obráceny. Algoritmus λ-opt je založen na konceptu λ-optimality: Trasa je označena jako

λ-optimální (nebo jednoduše λ-opt), pokud není možné dosáhnout kratší trasy nahrazením jakýchkoli λ odkazů jinou sadou λ odkazů [6]. Praktické využití je popsáno v 3.6.5.

3.4 Genetic Algorithm

Genetický algoritmus je metoda pro řešení omezených i neomezených optimalizačních problémů, která je založena na přírodním výběru, procesu, který řídí biologickou evoluci. Genetický algoritmus opakovaně modifikuje populaci individuálních řešení. V každém kroku genetický algoritmus vybírá jedince z aktuální populace, kteří se stanou rodiči a používá je k vytvoření potomků pro další generaci. V průběhu následujících generací se populace "vyvíjí"směrem k optimálnímu řešení. Genetický algoritmus lze aplikovat k řešení různých optimalizačních problémů, které nejsou dobře vhodné pro standardní optimalizační algoritmy, včetně problémů, ve kterých je cílová funkce nespojitá, nediferencovatelná, stochastická nebo silně nelineární. Genetický algoritmus může řešit problémy kombinovaného celočíselného programování, kde jsou některé komponenty omezeny být celočíselné. [10]

3.5 Lin-Kernighan Heuristic

Jedná se asi o nejlepší algoritmus pro řešení problému obchodního cestujícího. Poprvé byl publikován v roce 1973 v [9]. Existují různé varianty a vylepšení tohoto algoritmu, které jsou velmi efektivní i při velkém počtu měst. Jak je zmíněno v [6], celková implementace a návrh algoritmu jsou velmi složité. Algoritmus Lin-Kernighan je úzce spjat s algoritmem λ-opt.

Algoritmus Lin-Kernighan patří do třídy tzv. lokálních optimalizačních algoritmů. Je specifikován pomocí výměn (nebo tahů), které umožňují převést jednu trasu na jinou. Po získání proveditelné trasy algoritmus opakovaně provádí výměny, které snižují délku aktuální trasy, dokud není dosaženo trasy, pro kterou žádná výměna neposkytuje vylepšení. Tento proces může být opakován mnohokrát z různých počátečních tras generovaných nějakým náhodným způsobem.

3.6 Ant Colony Optimization Algorithm (ACO)

Mravenci projevují složité sociální chování, které již dlouho přitahuje pozornost lidí. Pravděpodobně jedním z nejzřetelnějších chování, které můžeme pozorovat, je tvorba takzvaných mravenišť. Když jsme byli mladí, mnozí z nás možná šlápli na takovou mravenčí dálnici nebo překážku na její cestě jen proto, abychom viděli, jak mravenci na takové rušení zareagují. Možná jsme se také ptali, kam tato mraveniště vedou nebo jak vůbec vznikají. Pro většinu z nás se takové otázky mohou stát méně naléhavými, jakmile dospějeme a začneme studovat jiné předměty, jako jsou počítačové vědy, matematika a podobně. Nicméně existuje značný počet výzkumníků, hlavně biologů, kteří studují chování mravenců detailně.

Jedním z nejzajímavějších chování, které mravenci projevují, je schopnost určitých druhů mravenců najít to, co počítačoví vědci nazývají nejkratšími cestami. Biologové experimentálně prokázali, že toto je možné díky využití komunikace založené pouze na feromonech, pachové chemické látky, kterou mravenci mohou ukládat a cítit. Právě toto chování inspirovalo počítačové vědce k vývoji algoritmů pro řešení optimalizačních problémů. První pokusy v této oblasti se objevily v 90. letech a mohou být považovány za spíše "hrací"ukázky, i když důležité pro naznačení obecné platnosti přístupu. Od té doby tyto a podobné myšlenky při-

tahují stále větší množství výzkumu a optimalizace mravenčí kolonií (ACO) je jedním z výsledků těchto úsilí. Ve skutečnosti jsou ACO algoritmy nejúspěšnější a nejuznávanější algoritmické techniky založené na mravenčím chování. Kolonie mravenců a obecně společnosti sociálních hmyzu jsou distribuované systémy, které navzdory jednoduchosti svých jednotlivců prezentují vysoce strukturovanou sociální organizaci. Díky této organizaci mohou kolonie mravenců dokázat složité úkoly, které v některých případech dalece převyšují individuální schopnosti jednoho mravence.

Obor algoritmů mravenců zkoumá modely odvozené z pozorování skutečného chování mravenců a používá tyto modely jako zdroj inspirace pro návrh nových algoritmů pro řešení optimalizačních a distribuovaných řídicích problémů. Hlavní myšlenkou je, že samoorganizační principy, které umožňují vysoce koordinované chování skutečných mravenců, lze využít k souřadění populací umělých agentů, kteří spolupracují na řešení výpočetních problémů.

Několik různých aspektů chování kolonií mravenců inspirovalo různé druhy algoritmů mravenců. Příklady jsou hledání potravy, rozdělení práce, třídění potomstva a kooperativní přeprava. V těchto případech koordinují mravenci své aktivity prostřednictvím stigmergie, formy nepřímé komunikace prostřednictvím modifikací prostředí. Biologové ukázali, že mnoho chování na úrovni kolonie pozorovaných u sociálního hmyzu lze vysvětlit poměrně jednoduchými modely, ve kterých je přítomna pouze stigmergická komunikace. Jinými slovy, biologové ukázali, že často stačí zvážit stigmergickou, nepřímou komunikaci k vysvětlení toho, jak mohou sociální hmyz dosáhnout samoorganizace. Myšlenkou za algoritmy mravenců je tedy využít formu umělé stigmergie k souřadění společností umělých agentů. Jedním z nejlepších příkladů algoritmů mravenců je známý jako optimalizace kolonií mravenců nebo ACO, který se zaměřuje na diskrétní optimalizační problémy [4].

3.6.1 Rozdíly skutečných a simulovaných mravenců

- Umělí mravenci jsou vytvořeni tak, aby spolupracovali a využívali informace, které zanechávají ostatní mravenci. Ve skutečnosti se jednotliví mravenci snaží získat co nejvíce potravy pro sebe.
- V ACO algoritmu jsou feromony inicializovány na nulovou hodnotu, zatímco v přírodě mravenci používají feromony, které už byly zanechány jinými mravenci.
- Skuteční mravenci vypouští feromon neustále, umělí mravenci značkují cestu „feromonem“ pouze při návratu do mraveniště. [18]
- Umělí mravenci mají tendenci procházet kratší cestami, zatímco v přírodě mohou být cesty dlouhé, ale mohou obsahovat více potravy.
- Umělí mravenci v ACO algoritmu se vypořádávají s problémy s paralelizací a koordinací, které nejsou relevantní pro skutečné mravence.

3.6.2 Popis algoritmu

ACO je metaheuristický algoritmus inspirovaný chováním mravenců. Jeho principem je vytvoření umělého mraveniště, které obsahuje počáteční řešení optimalizačního problému, a postupné procházení řešení umělými mravenci. Mravenci vytvářejí feromonové stopy, které informují ostatní mravence o kvalitě cesty, kterou prošli. Tímto způsobem mravenci postupně hledají nejlepší cestu řešení daného problému. Princip algoritmu spočívá v iterativním procházení umělého mraveniště několikrát za sebou, přičemž v každé iteraci se aktualizují feromonové stopy a vybírají se nejlepší cesty. Výsledkem algoritmu je nejlepší nalezené

řešení daného problému. Algoritmus se využívá k řešení optimalizačních problémů v různých oblastech, jako je například logistika, plánování dopravy nebo hledání nejlepších tras v sítích.

3.6.3 Aplikace algoritmu na problém obchodního cestujícího

1. Nastaví se počáteční stav intenzit feromonů na $\tau_{ij} = \alpha$. α by mělo být kladné číslo blízké 0, obvykle v rozmezí (0.01,0.5), pro všechny cesty mezi místy i a j kde ($i = 1...N, j = 1...N, i \neq j, N$ je celkový počet míst).

2. Dále se vytvoří zvolený počet mravenců.

3. V každém mravenci se začne iterovat přes $y = (0...Y - 1)$, kde Y je počet iterací, který jste nastavili.

4. V každé iteraci v rámci jednoho mravence se vybere počáteční místo.

5. Následně v každé iteraci v rámci mravence se vybírá každé další místo, které mravenec ještě nenavštívil na základě vztahu:

$$p_{ij} = \frac{[\tau_{ij}]^{\alpha} [\eta_{ij}]^{\beta}}{\sum_{l=1}^{N} [\tau_{il}]^{\alpha} [\eta_{il}]^{\beta}}$$

   kde p_{ij} je pravděpodobnost cesty z místa i do místa j, $\eta_{ij} = 1/d_{ij}$ a α a β jsou empirické konstanty. Pravděpodobnost cesty p_{ij} se vypočítá pro každé nenavštívené místo z aktuálního místa. Na základě těchto pravděpodobností se rozhodne, které místo bude další. Tento proces se opakuje, dokud mravenec nenavštíví všechna místa v rámci jedné iterace. Poté se v každé iteraci aktualizují feromony. Pro tento účel se nejprve uzamkne zámek, protože feromony jsou sdílené pro každého mravence. $\Delta\tau$ představuje přírůstek feromonů na všech cestách, které mravenec navštívil v rámci jedné iterace:

$$\Delta\tau = \frac{Q}{L}$$

   Q je zvolená konstanta, která určuje, kolik feromonů mravenec vyloučí během jedné iterace. L je délka cesty.

   Následně se v každé iteraci v rámci vlákna aplikuje postupné vyprchávání feromonů na všechny cesty, nikoliv pouze na ty, které mravenec v rámci iterace navštívil.

$$\tau = \tau * (1 - \rho)$$

6. Dále se porovná, zda právě dokončená cesta není nejlepší možná. Zde je také potřeba použít zámek. Pokud je cesta lepší než všechny ostatní cesty v rámci všech mravenců, uloží se tato cesta. Tento proces probíhá v každé iteraci v rámci každého vlákna.

7. Nakonec se v rámci mravence inkrementuje čítač y a vrátí se na bod 3.

3.6.4 Mnou modifikovaný algoritmus s využitím dalších algoritmů

Jsou zde tři hlavní rozdíly: větší využití Q0 v bodě 1, použití dvou dalších algoritmů (Nearest Neighbor Algorithm) pro nastavení počátečních feromonů a více využití 2-opt k přeskupování cest, aby vzniklo lepší řešení (viz sekce 3.6.5).

1. Nastaví se počáteční stav intenzity feromonů na $\tau_{ij} = \alpha$, kde α by mělo být kladné číslo blízké nule, obvykle v rozmezí (0,01 až 0,5). To se provede pro všechny cesty mezi místy i a j, kde $(i = 1...N, j = 1...N, i \neq j)$ a N je celkový počet míst.

2. Začne se iterovat přes $z = (0...Z - 1)$, kde Z je počet iterací, kolikrát se má použít Nearest Neighbor Algorithm.

3. V rámci iterace se zvolí počáteční místo m, které ještě nebylo v žádné iteraci zvoleno.

4. Použije se v rámci iterace Nearest Neighbor Algorithm, ale pouze pro omezený počet míst F, který je obvykle nastaven na 2 až 4 místa.

5. V rámci iterace se inkrementuje τ_{ij} na cestách, které byly v rámci Nearest Neighbor Algorithm použity, pomocí vztahu:

$$\Delta\tau = \frac{Q}{L}$$

   Konstanta Q je zvolena jako množství feromonů, které mravenec vyloučí. Konstanta L reprezentuje délku cesty.

6. Inkrementuje se čítač z a program se vrací na bod 2.

7. Následně se vytvoří požadovaný počet mravenců (agentů), které reprezentují v mé implementaci vlákna.

8. V každém vláknu se iteruje přes y = (0...Y-1), kde Y je počet iterací, který byl nastaven.

9. V každé iteraci v rámci vlákna se vybere počáteční místo.

10. Poté se v každé iteraci v rámci vlákna vybírá každé další místo, které mravenec ještě nenavštívil na základě vztahu:

$$p_{ij} = \frac{[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}}{\sum_{1}^{N}[\tau_{il}]^{\alpha}[\eta_{il}]^{\beta}}$$

    kde p_{ij} je pravděpodobnost cesty z místa i do místa j a $\eta_{ij} = 1/d_{ij}$. Pravděpodobnost cesty p_{ij} se vypočítá pro každé nenavštívené místo z aktuální pozice. Dle pravděpodobností se rozhodne, které místo bude následující. Toto se opakuje, dokud mravenec (vlákno) v rámci jedné iterace nenavštíví všechna místa.

11. V algoritmu figuruje také volba na základě konstanty Q0. Pokud pseudonáhodně vygenerované číslo v rozmezí (0,1) je menší než konstanta Q0, zvolí se cesta s největší pravděpodobností, tj. "nejvýhodnější".

12. Poté v každé iteraci v rámci vlákna se aktualizují feromony, aby k tomu mohlo dojít je nejprve zamknout zámek, jelikož jsou feromony pro každého mravence sdílené. $\Delta\tau$ je přírůstek feromonů na všech cestách, které mravenec v rámci jedné iterace navštívil.

$$\Delta\tau = \frac{Q}{L}$$

Q je zvolená konstanta kolik mravenec vyloučí feromonů za jednu iteraci. L je délka cesty.

13. V každé iteraci v rámci vlákna se aplikuje i postupné vyprchávání feromonů a to již pro všechny cesty ne pouze ty, které mravenec v rámci iterace navštívil.

$$\tau = \tau * (1 - \rho)$$

14. Dále se porovná, jestli cesta právě dokončená není nejlepší možná. Zde je potřeba taky použít zámek. Pokud je cesta lepší než všechny ostatní cesty v rámci všech mravenců cesta se uloží. Toto probíhá taky v každé iteraci v rámci každého vlákna.

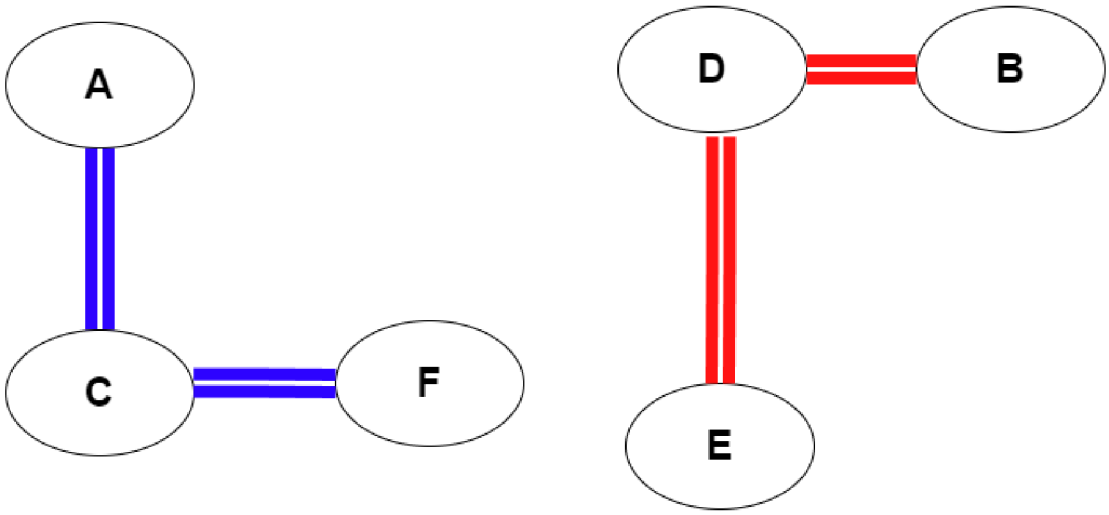
15. V rámci vlákna se inkrementuje čítač y a vrátí se se na bod 8.

16. Pro nejlepší cestu kterou jsme našli bude vstup 2-opt algoritmu, který pomocí přeskládávání cest mezi dvěma místy. Vrátí lepší nebo stejně dobrou cestu.

3.6.5 Využití Nearest Neighbor Algorithm a 2-opt v ACO

Nearest Neighbor Algorithm slouží k tomu, aby se hned v prvních iteracích zlepšila cesta. Pokud volíme pouze malý počet míst, pro který tento algoritmus chceme použít, je velmi velká šance, že patří k optimálnímu řešení. Následně to pomůže ACO jako zdroj počátečních feromonů. Jde to pěkně vidět na obrázku 3.1. Tento obrázek zobrazuje použití Nearest Neighbor Algorithm pro 2 iterace a počet navštívených míst 3. V první iteraci se náhodně vygenerovalo, že budeme začínat v místě **A**. Následně jsme se rozhodli jít do nejbližšího místa **C** pomocí cesty (A,C), na kterou se připočetla hodnota $\Delta\tau$ k τ_{AC}. Poté jsme se z místa C rozhodli jít do místa F, které je nejbližší nenavštívené místo, a to pomocí cesty (C,F), na kterou se také připočetla hodnota $\Delta\tau$ k τ_{CF}. Stejný postup platí i pro červenou cestu, pouze s počátečním místem **B**.

2-opt slouží k optimalizaci řešení ACO tím, že se snaží prohodit místo B za C. Myšlenka je jednoduchá máme místa A,B,C,D kde z místa A jdeme do B a z C do D. A snažíme se udělat výměnu v podobě toho, že přepojíme místa, aby se šlo z A do C a z B do D, pokud je takto přepojená cesta výhodnější, tak přepojení uskutečníme. A dokud se cesta vylepšuje zkoušíme takové výměny pro všechny místa dokola.



Obrázek 3.1: Nejbližší sousedi

3.6.6 Volba parametrů

Volba parametrů je velmi důležitou částí ACO algoritmu. Správnou volbou parametrů můžete urychlit nalezení kvalitního řešení. Je to poměrně kritická část ACO, jelikož parametry mají velký vliv na řešení. Špatnou volbou parametrů můžete dokonce z funkčního algoritmu udělat velmi špatný algoritmus. Můj program již má předem nastavené parametry, které jsem vyladil dlouhým testováním programu. Přednastavené parametry jsou uvedeny na obrázku 6.8 v levé části. Zde se zaměříme na každý z nich a popíšeme, co dělá, jaká by měla být optimální hodnota a jaké jsou možné hodnoty. Jednotlivé informace o parametrech jsou čerpány z [8].

- α – Reprezentuje důležitost vyloučeného feromonu na cestě. Pokud by byl parametr nastaven jako příliš velké číslo, mravenci by měli tendenci volit ty samé cesty jako předchozí, což by vedlo k silnější spolupráci mezi mravenci. Na druhou stranu, to nemusí být nutně výhoda, protože algoritmus by mohl zůstat pouze v lokálním řešení a nenajít žádné lepší. Pokud by ale hodnota byla příliš malá, rychlost konvergence ACO by se zpomalila, bez ohledu na to, že lze globální vyhledávací schopnost algoritmu zlepšit. Optimálně nastavená hodnota je podle mě (0.7,2.0).

- β – Reprezentuje faktor viditelnosti na cestě. Pokud je hodnota příliš velká, algoritmus se bude blížit řešení pomocí Nearest Neighbour, protože se budou vybírat nejlepší cesty na základě viditelnosti, která je přímo úměrná vzdálenosti mezi místy. To může vést k uvíznutí v lokálním řešení. Na druhou stranu, pokud je hodnota příliš nízká, cesty s velkou vzdáleností budou stejně atraktivní jako ty s malou, což může vést k stagnaci algoritmu. Podle mého názoru je optimální rozsah hodnot (1.0,35.0).

- ρ – Koeficient odpařování feromonu vyjadřuje míru odpařování feromonu a odráží míru vzájemného ovlivňování mezi mravenci. Obecně by měla být hodnota dostatečně vysoká, aby účinně bránila nekonečnému hromadění feromonu. Pokud je hodnota příliš malá, může se snížit globální vyhledávací schopnost ACO. Naopak, pokud je příliš vysoká, může se zlepšit globální vyhledávací schopnost ACO, ale rychlost konvergence

bude pomalá. Optimální rozsah se pohybuje někde mezi (0.1,0.3). Hodnota by neměla být větší než 1 a menší než 0.

- **Q** – Intenzita feromonu představuje celkové množství feromonu na jednotlivých hranách grafu a ovlivňuje rychlost konvergence ACO. Pokud je hodnota příliš vysoká, koncentrace feromonu bude příliš vysoká a algoritmus může upadnout do lokálního optima. Naopak, pokud je hodnota příliš nízká, rychlost optimalizace bude pomalá. Podle mého názoru je optimální rozsah mezi (2000, 50000).

- **q0** – Vyjadřuje konstantu, s jakou pravděpodobností se zvolí ta nejpravděpodobnější cesta. Tento prvek je důležitý, protože pokud by byl nastaven na velkou hodnotu, naše řešení by se blížilo k řešení pomocí Nearest Neighbour. Nicméně, tento parametr v ACO nehraje klíčovou roli a může být klidně nastaven na hodnotu 0, pokud ho nechcete použít. Jedná se pouze o vylepšení, které pomáhá najít lepší řešení rychleji. Optimální rozsah pro tuto konstantu je někde mezi 0 a 0.5. Minimální hodnota je 0 a maximální hodnota je 1. Funguje to následovně: Nejprve zvolíme konstantu q0, například 0.4. Poté náhodně vygenerujeme číslo q v rozsahu (0,1). Pokud je q menší než $q0$, zvolíme nejpravděpodobnější cestu. Pokud je q větší než $q0$, volíme cesty podle pravděpodobností.

3.7 MyAlgo

Rozhodl jsem se navrhnout vlastní algoritmus, jelikož ACO byla prakticky nepoužitelná pro více než 1000 míst a kvalita řešení nebyla zrovna uspokojivá, i s využitím 2-opt algoritmu pro optimalizaci řešení. Původně jsem chtěl zvolit Artifical Atom Algorithm jako druhý hlavní algoritmus, ale po shlédnutí výsledků, i když byl algoritmus schopný nalézt většinou optimální řešení do 50 míst, trvalo to asi 100 sekund. Což bylo ještě pomalejší než ACO. Původní myšlenky, které mě napadly, byly, že bych mohl implementovat přehazování míst na základě pravděpodobnosti, která by fungovala podobně jako v ACO (závisela by na délce celkové cesty a délce cesty od místa A do místa B), ale pouze by se mohla vylepšovat a přehodil by se do lepšího řešení. Zde jsem si uvědomil, že by se algoritmus mohl hodně rychle zaseknout a nenalézt lepší cestu, takže jsem ani nezačal s implementací.

3.7.1 Myšlenka

Základní myšlenku jsem převzal asi z nejlepšího algoritmu na řešení problému obchodního cestujícího, konkrétně z algoritmu **lin–kernighan heuristic**. Tento algoritmus se snaží vylepšit řešení pomocí prohození míst. Mě napadlo, že by bylo dobré vygenerovat optimální řešení pro nejbližších x míst a poté je spojit dohromady s použitím hranic. Avšak, protože bychom mohli uváznout v lokálním optimu, rozhodl jsem se nebrat v úvahu všechna místa, ale vždy vybrat určité místo jako začátek a přidávat další místa k němu postupně, abychom procházeli různé možnosti se stejnými místy. Z toho mi vyplynulo, co bych potřeboval implementovat. Vždy zvolím místo, které jsem nezvolil jako začátek, a přidám k němu několik nejbližších sousedů. Poté vypočítám "optimální řešení"mezi nimi. Pokud je nové řešení lepší než původní, přepojím body. I když je myšlenka jednoduchá, implementace byla poměrně obtížná, zejména v počítání vzdáleností v přepojovaných místech a volbě nových hranic. Také jsem si uvědomil, že pokud například počítám vylepšení pro 5 míst, mohu přepojit 5 míst, ale optimální cesta pro 4 místa by mohla být vylepšena, ale pro 5 míst ne. Proto přidávám místa postupně a vybírám to nejlepší možné vylepšení z nich.

3.7.2 Popis algoritmu

1. Nastaví se čítač **s** na 0 a uloží se počáteční řešení.
2. Nastaví se aktuální řešení problému na počáteční řešení.
3. Dokud se dá dosáhnout zlepšení, volá se jádro algoritmu s počáteční hodnotou sousedů + **s**.
4. Pokud se zlepšení nedá dosáhnout, volá se jádro algoritmu s prostřední hodnotou sousedů.
5. Aplikuje se na problém 2-opt algoritmus pro optimalizaci.
6. Uloží se trasa a aktuální délka trasy.
7. Pokud je čítač **s** větší nebo roven počtu počátečních řešení, pokračuje se na bod **7**. Pokud je menší, inkrementuje se čítač **s** a vrátí se na bod 2.
8. Vybere se nejlepší počáteční trasa.
9. Pro nejlepší vybranou počáteční trasu se volá jádro algoritmu s posledním počtem sousedů, dokud se zlepšuje.

3.7.3 Popis jádra algoritmu

1. Začne se iterovat nad všemi místy tak, aby se každé prošlo právě 1x.
2. Pro aktuální místo se zjistí levá a pravá hranice.
3. Vypočítá se vzdálenost místa po odpojení od jeho hranic.
4. Uloží se aktuální místo z bodu **1** jako počáteční.
5. Vytvoří se struktura, která ukládá novou část cesty, a vloží se tam počáteční místo.
6. Nastaví se čítač **z** na 0.
7. Zvolí se nejbližší soused, buď nejbližšího k počátečnímu místu, nebo místu aktuálnímu. Souseda nastavíme jako aktuální místo.
8. Kontrola, jestli místo není hranicí. Pokud je, musí se vypočítat vzdálenost po přepojení hranic.
9. Pokud místo není hranice, musí se vypočítat vzdálenost po odpojení od jeho hranic.
10. Odstraní se aktuální místo.
11. Do struktury pro novou část cesty se vloží aktuální hranice a aktuální místo.
12. Na novou část cesty se aplikuje 2-opt algoritmus.
13. Z struktury pro novou část cesty se oddělají hranice, ale zároveň se nechají ve výsledku pro část nové cesty, kterou vrátí 2-opt algoritmus.
14. Spočítá se vzdálenost nové cesty a uloží se vzdálenost a nová část cesty.

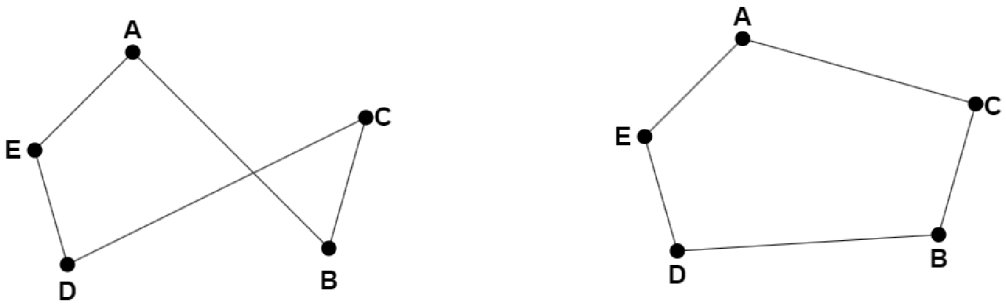
15. Pokud je čítač **z** větší nebo roven počtu nastavených sousedů, pokračuje se na další bod. Pokud ne, inkrementuje se čítač **z** a vrátí se na bod **7**.

16. Vybere se nejlepší cesta dle vzdálenosti.

17. Z té se vytvoří nová cesta, která se vytváří z cesty, ve které je odpojeno určité množství míst, a ten samý počet stejných míst tvoří novou část cesty.

3.7.4 Přepojování míst

Jednou z klíčových částí algoritmu je přepojování míst a výpočet vzdáleností po přepojení. Počítání celkové vzdálenosti a provádění změn v poli je totiž v porovnání s tímto způsobem extrémně výpočetně náročné. Abychom mohli přepojování míst použít pro všechna místa s použitím hranic, musíme vytvořit cyklus z pole míst, kde místo na indexu 0 bude mít jako levého souseda místo na posledním indexu a místo na posledním indexu bude mít jako pravého souseda místo na nultém indexu. Prvním krokem je zvolit nějaké místo, které ještě nebylo zvoleno, řekněme místo A. Předpokládejme, že máme množinu míst [A, B, C, D, E]. Pro místo **A** musíme zvolit levého souseda (místo **E**) a pravého souseda (místo **B**). Poté musíme spočítat vzdálenost odpojení místa **A** od **E** a **B**. To zahrnuje cestu z místa **E** do **A** a cestu z místa **A** do **B**. Tuto vzdálenost nazýváme například **vzdálenost hranic**. Poté vybereme nejbližší místo k místu **A**, což v tomto případě bude místo **E**. Jelikož místo **E** je zároveň levá hranice posunu levou hranici na místo **D**. Tím že je místo **E** je levá hranice znamená, že nemá pravého souseda toho už jsme odpojili u vzdálenosti hranic. Proto jediné co musíme spočítat je vzdálenost po odpojení místa **E** od jeho levého souseda místa **D**, což bude vzdálenost z místa **E** do **D**. Pokračuji dále, tím že vyberu další místo nejblíže místu **A**, které jsem ještě nevybral, což bude místo **C**. Jelikož místo **C** není pravá ani levá hranice, musím jeho vzdálenost po odpojení spočítat tak, že vezmu rozdíl vzdáleností přepojení pravé hranice místa C do levé hranice místa C a odpojení místa **C** od hranic místa C. To je rovno cesta z **B** do **D** - cesta z **B** do **C** - cesta z **C** do **D**. Poté s použitím hranic vytvořím nejlepší řešení přeskládáváním míst [A,C,E]. Řešení, které dostanu bude vypadat například jako [E,A,C]. Vypočítáme vzdálenost po připojení, což bude vzdálenost levé hranice **D** do místa E + vzdálenost řešení + vzdálenost místa C do pravé hranice **B**. Vzdálenost řešení je vzdálenost z místa **E** do **A** + vzdálenost z **A** do **C**. Na takto malém příkladu se to může zdát počítat jako zbytečné, ale když máte pro tisíce míst spočítat celkovou vzdálenost je to časově náročnější než pouze vzdálenost po přepojení. Ještě by se museli dělat změny v poli kterým se vyhneme, pokud přepojování míst nevylepší řešení. Na obrázku 3.2 můžeme vidět přesně to co jsem zde napsal, jak se z levého grafu stane ten pravý.



Obrázek 3.2: Přepojení míst

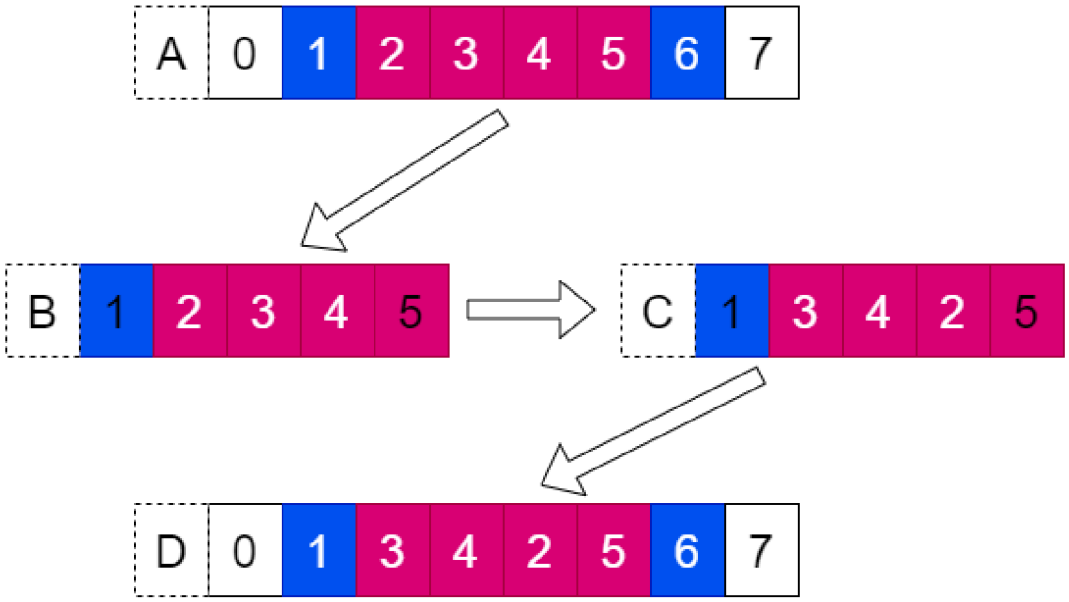
3.7.5 Problémy s uváznutím v lokálním optimu

Jelikož je jádro algoritmu poměrně rychlé, pro 4000 míst za 10 vteřin zvládne vyřešit problém s 10 nejbližšími sousedy. Pro 1 souseda a 4000 míst je to pod vteřinu. Pokud použijeme více sousedů, řešení se většinou zlepší. Jenže jsem si všiml jedné věci, algoritmus je hodně závislý na prvotním generovaném řešení. Když je špatné, uvízne a už se nezlepší. Proto jsem se rozhodl aplikovat způsob postupného vylepšování cesty postupným zvyšováním počtu sousedů, což opravdu má velký vliv na kvalitu řešení. Místo toho, abych hned aplikoval například 60 sousedů, aplikuji nejdříve 1 souseda, pak 2, a poté 3. Můžeme tomu říkat **první nejbližší sousedi**, protože většinou platí, že čím víc sousedů, tím lepší řešení. Musím pro každé řešení prvního nejbližšího souseda nějak dorovnat řešení "aby to bylo fér". Proto každé řešení prvního nejbližšího souseda vezmu a vylepším stejným počtem sousedů, například 10, poté na každé řešení aplikují 2-opt algoritmus a vyberu to nejlepší. Nakonec se snažíme co nejvíce vylepšit řešení, takže zvýšíme hodnotu sousedů, například na 60.

3.7.6 Rozdělení problému na segmenty

Všiml jsem si, že pokud náhodně generuji počáteční cestu a následně aplikují MyAlgo, je možné dostat lepší řešení. Abych využil tuto vlastnost a zároveň neřešil celý problém najednou, rozhodl jsem se rozdělit ho na menší podproblémy a ty postupně zlepšovat. Když místa v podproblému náhodně zamíchám a aplikují MyAlgo, pokud je aktuální část řešení horší než nově vygenerovaná, nahradím ji. I když tato úprava zní jednoduše, celé jádro algoritmu a všechny pomocné funkce se musely předělat. Jádro algoritmu je na bázi cyklu, jak je zmíněno v 3.7.4, kde v podstatě neexistuje nejpravější a nejlevější prvek. Ale pokud vezmeme jen určitou část problému, musíme vzít i nějaké hranice segmentu. Po překonání velikosti segmentu (myslím tím, když chceme dostat pravý prvek od nejpravějšího prvku v segmentu nebo levý od nejlevějšího), jsou tyto hranice nutné. Pokud tam tyto hranice nebyly, v podstatě by to znamenalo, že řešení, které nám jádro vrátí, nemusí vůbec zapadat do celkového problému, protože bychom tam nepočítali vzdálenost při napojení do celkového řešení. Všechny funkce se musely předělat, například funkce pro vytváření nové cesty, pokud je zlepšení vkládala novou cestu vždy po levé hranici. Jenže pokud použijeme jen určitý segment a naše levá hranice je zároveň hranice segmentu, kterou nemohu měnit, nemůžu vložit řešení v rámci podproblému, kde není hranice segmentu. Pokud je levá hranice zároveň hranice levého segmentu, tak na začátek, pokud pravého, tak na konec. Pro představu je to vyobrazeno na obrázku 3.3. Kde písmena značí označení, v reálném programu se nepracuje

s tak malým počtem míst a segment náhodně zamícháváme pouze pro představu. Písmeno A značí původní spojení míst. Představme si, že pro toto spojení je segment, který chceme upravovat, vyobrazen růžově a jeho hranice modře. Písmeno B označuje skupinu bodů, kterou dostaneme v rámci jádra algoritmu. Levá a pravá hranice jsou vybarveny černou barvou a jsou to místa 1 a 5. Zároveň si můžeme všimnout, že levá hranice je zároveň hranicí segmentu. Poté aktivujeme 2-opt. Jak vidíte, nepřehazuje se hraniční body a dostaneme řešení C. Nakonec vyměníme segment v celkovém řešení, pokud dojde ke zlepšení, a to je reprezentováno řešením D.



Obrázek 3.3: Segmenty

3.7.7 Náměty na zlepšení algoritmu

První možností, která by určitě fungovala, je **využití vláken**. Nabízí se zde několik způsobů, jak vlákna využít. Jak je zmíněno v sekci 3.7.5, generuji nějaký počet prvotních řešení, ze kterých vybírám to nejlepší a až poté aplikuji větší rozsah sousedů. Skupina prvotních řešení není na sobě nijak závislá, jen se vybírá to nejlepší. Zde se přímo nabízí využití paralelizace pro každé z prvotních řešení.

Další možností by mohlo být využití při segmentech, jelikož jednotlivé segmenty nejsou na sobě závislé. Tato myšlenka napadne asi každého. Další způsob využití vláken by mohl být například v jádru algoritmu, kde používám pro volbu kandidátů pouze nejbližšího souseda k aktuálnímu místu a k počátečnímu. Například jedno vlákno by mohlo využívat tuto techniku a druhé nějakou pokročilejší techniku, například volbu mezi nejbližšími sousedy na základě pravděpodobností. Poté by se použil vždy ten lepší výsledek. S tím souvisí i další nápad, který mě napadl, a to je v podstatě vracení řešení na nějaký určitý záchytný bod, pokud dojde k uváznutí a není způsob, jak se zlepšit. Následně by se řešení generovalo jiným způsobem, například metodou vkládání místa nebo skupiny míst na lepší pozici, což je v podstatě opačný způsob fungování algoritmu. Dokonce jsem ji již pro 1 místo implementoval. Způsobů, jak zlepšit aktuální řešení, je určitě hodně. Kvalita řešení by se určitě

zlepšila i využitím více opt algoritmu, například pro jednotlivé segmenty, ale implementace například 5-opt algoritmu není jednoduchá, aby byla zároveň efektivní a netrvala příliš dlouho. Jedinou verzi, kterou jsem našel, již implementovanou byla součástí LKH algoritmu a je popsaná zde: [7].

Kapitola 4

Návrh a implementace algoritmů

4.1 Ant Colony Optimization Algorithm

Původně jsem se rozhodl algoritmus implementovat v jazyce Python, ale kvůli absenci paralérního zpracování vláken v jazyce jsem algoritmus přepsal do jazyka C++ a ještě ho nějak vylepšil. Implementaci v jazyce Python zde popisovat nebudu, jelikož ji v programu nevyužívám.

4.1.1 Zpracování vstupního souboru

Vstupem algoritmu je soubor algo_inputN.txt kde N je číslo jádra na kterém algoritmus běží. Prvních 11 řádků souboru je nastavení algoritmu neboli parametry, které jsme zvolili. Řádky které následují reprezentují místa. Místo si můžeme představit jako nějakou strukturu, která má nějaké unikátní označení **ID**. Dále místo obsahuje souřadnice **X** a **Y**. A list vzdálenosti do ostatních míst. Přesně v tomto pořadí se nachází i na řádcích v souboru algo_inputN.txt. Takže funkce **parse_file_to_places()** přečte soubor a zpracuje ho do proměnných reprezentující parametry ACO. A vytvoří dynamické pole s místy, které naplní konkretním počtem míst co je v souboru.

4.1.2 Přednastavení feromonů

Struktura místa bude kromě dynamického pole pro vzdálenosti do ostatních míst obsahovat i dynamické pole pro jejich τ množství vyloučených feromonů na cestě do ostatních míst. Počáteční nastavení hodnoty na zvolené množství $\tau 0$ se uskuteční v funkci **parse_file_to_places()**. Dále, aby neměly všechny cesty stejné množství τ a zrychlila se tím i rychlost algoritmu, použije se funkce **setup_nn_tau()**. Funkce nastaví počáteční feromony pomocí Nearest Neighbour algoritmu. Konkrétně se zvolí počáteční místo, které ještě nebylo vybráno. A pomocí nastavených iterací a délky míst v Nearest Neighbour se pomocí tohoto algoritmu přenastaví τ. Oproti normálnímu přenastavování τ zde není žádné vyprchávání feromonů, pouze se pomocí funkce **setup_tau_start_NN()** přičte $\Delta\tau$ k τ, kde $\Delta\tau = \frac{Q}{trasa}$. Q je předem nastavený parametr a trasa je celková trasa Nearest Neighbour algoritmu.

4.1.3 Vytvoření agentů

Pomocí funkce **setup_threads()** jsou vytvořena jednotlivá vlákna pro agenty s počtem vláken, který byl nastaven. Poté se v funkci **created_agent()**, která běží jako samostatné

vlákno, začne iterovat v počtu nastavených iterací. V každé iteraci jsou volány 3 další funkce.

- **agent_chose_path** – Lze říct, že funkce je srdcem ACO, neboť se v ní odehrává samotné procházení míst algoritmem. Nejprve se náhodně zvolí začínající místo. Poté se začne iterovat od 0 do celkového počtu míst - 1. První krok spočívá výpočtu součtu všech viditelností a hodnoty τ z aktuálního místa do ostatních míst za pomoci funkce **setup_visibility_tau**. Konkrétně se jedná o tuto část vzorce:

$$SUM = \sum_0^N [\tau_{il}]^\alpha [\eta_{il}]^\beta$$

  Poté se aktuální místo odstraní z pole nenavštívených míst a začne se iterovat od 0 do velikosti pole nenavštívených míst -1. V každé iteraci se spočítá viditelnost a hodnota τ pro konkrétní trasu a vypočítá se pravděpodobnost cesty z aktuálního místa i do dalšího nenavštíveného místa. Toto vyplývá z následujícího vzorce:

$$p_{ij} = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{SUM}$$

  Poté, kdy skončí iterování v nenavštívených místech a máme uložené všechny potřebné pravděpodobnosti tras, náhodně vybereme číslo q z rozmezí (0,1). Pokud je q menší než q0, nastavíme aktuální místo na to s nejvyšší pravděpodobností. V opačném případě vybereme aktuální místo z nenavštívených míst na základě pravděpodobnosti. Poté se přičte vzdálenost z posledního navštíveného místa do aktuálního místa. Tímto končí cyklus, který probíhal až do počtu míst - 1. Nakonec se ještě přičte vzdálenost zpět do prvního místa a první místo se vloží do seznamu navštívených míst. Funkce vrátí strukturu, kde je uložena trasa navštívených míst a celková vzdálenost.

- **agent_update_tau** – Funkce slouží k aktualizaci sdílených feromonů mezi vlákny. Nejprve se použije zámek, protože τ je sdílená proměnná pro všechny vlákna a přístup k proměnné více vlákny najednou by mohl způsobit problémy. Navíc by hodnoty feromonů nebyly aktualizovány s aktuálními hodnotami. Poté se vypočítá $\delta\tau = \frac{Q}{D}$. Kde **D** je vzdálenost cesty v aktuálním vláknu a v rámci iterace. **Q** je konstanta, kterou jsme nastavili. Poté se přičte $\delta\tau$ ke všem cestám, které jsme v aktuálním vláknu v rámci iterace navštívili, a zámek se odemkne.

- **agent_check_best_distance** – Funkce slouží pouze k udržení nejlepší cesty a vzdálenosti. Zase zde musíme použít zámek, protože nejlepší cesta a vzdálenost jsou globální proměnné. Funkce pouze kontroluje, zda je aktuální cesta lepší než nejlepší cesta; pokud ano, uloží se jako nová nejlepší cesta.

- **pheromone_evaporation** – Funkci volá pouze jeden agent, a to ten poslední. Slouží k vypařování feromonů. Je jedno, zda jdu z místa [A] do [B] nebo z [B] do [A], protože ve funkci neprocházíme všechny cesty. Navíc je cesta z [A] do [A] irelevantní. Iterujeme pouze od determinantu matice a kopírujeme hodnoty na druhou stranu matice, kde jsou hodnoty stejné ([A][B] == [B][A]). Vypařování se počítá jako $\tau = \tau(1 - \rho)$, kde ρ je zvolená konstanta.

4.1.4 Aktivace 2-opt

Pro optimalizaci cesty a zlepšení řešení používám 2-opt algoritmus, který většinou odstraní křížení cest v řešení. Tento algoritmus se volá s nejlepší nalezenou cestou v ACO a pokouší se ji vylepšit.

4.1.5 Zapsaní nejlepších výsledku do souboru

Poslední věcí, která se odehraje, je zapsání nejlepší vzdálenosti do souboru **best_pathN.txt**, kde **N** je číslo jádra, na kterém algoritmus běží. Soubor se otevře a zapíše se na jeden řádek navštívená místa oddělená mezerami a na druhém řádku bude vzdálenost cesty.

4.2 MyAlgo

MyAlgo je taky pro zlepšení rychlosti implementovaný v jazyce C++. Skládá se v podstatě z dvou hlavních funkcí **start_algo**, která koordinuje algoritmus. A funkce **core**, která je v podstatě jádro algoritmu. Tyto funkce využívají různých vedlejších funkcí. Ještě se zde nachází funkce, která používá místo celého řešení pouze segmenty **core_for_segment**. Dále se v algoritmu nachází i funkce **start_core_insertion**, kterou nevyužívám, protože nezlepší řešení. Rychlost a princip algoritmu spočívá v myšlence, pokud je přepojení lepší, tak teprve potom udělej změny v poli. Funguje zde stejně načítání míst 4.1.1 a zápis nejlepšího řešení 4.1.5 jako v ACO. S jediným rozdílem a to, že zde nepoužíváme jádra procesu. Popíši zde hlavní funkce algoritmu například nebudu zde popisovat funkce pro segmenty, jelikož fungují na stejném principu jako ty normální a rozdíl je už vysvětlený v teorii a taky funkce, které nepoužívám je jich tam hodně zkoušel jsem, jak bych v algoritmu fungovali, ale neosvědčili se mi. Na druhou stranu jdou zlepšit, tak jsem je nechtěl mazat například best5 může být použita pro víc prvků ne jen pro 5.

4.2.1 start_algo

Funkce funguje jako koodrinátor algoritmu. Má 3 hlavní úkoly:

- **Vygenerovat určitý počet počátečních řešení** - Nejprve se zavolá jádro algoritmu s počátečním počtem sousedů. Pokud se řešení zlepšuje, stále se volá jádro algoritmu s tímto počtem sousedů. Poté se opakuje tento postup s prostředním počtem sousedů a opět se hledá zlepšení. Nakonec se pokusí zlepšit celé řešení pomocí 2-opt algoritmu. Tento postup se opakuje s postupně větším počtem počátečních sousedů, až se dosáhne nastaveného počtu.

- **Vybrat nejlepší z počátečních řešení a zlepšit ho** - nejlepší z počátečních řešení se vybere a pokusí se zlepšit voláním jádra algoritmu s posledním počtem sousedů. Zase se používá strategie, dokud se řešení zlepšuje voláme jádro s tímto počtem sousedů.

- **Použití segmentů** Zde se volá jádro pro segmenty, je potřeba si i představit, jak část řešení bude fungovat například pokud se volá segment velikosti 120 je zbytečné volat segment velikosti 60. Po testovaní jsem přišel na docela pěkné velikosti segmentů, které skoro vždy zlepší výsledek od 120 do 102 vždy o 3 místa, aby se co nejvíc kryly.

4.2.2 core

Funkce je srdcem algoritmu, iteruje přes všechny místa co máme. V podstatě se snaží v každé iteraci vybírat za pomocí nejbližších míst kandidáty, které vloží vedle sebe a s nimi za pomoci 2-opt algoritmu zlepší řešení. Funkce v každé iteraci pro aktuální místo dělá následující kroky:

- Za pomocí funkce cycle zvolí index levé a pravé hranice.
- Indexy použije v aktuální cestě a zjistí levé a pravé hranice.
- Spočítá vzdálenost po odpojení místa nad kterým iterujeme v původním spojení.
- Začíná iterovat nad zvoleným počtem nejbližších sousedů.
- Vybere z postupného iterovaní nad nejbližšími sousedy nejlepší výsledek.
- Pokud je nejlepší výsledek horší jak aktuální cesta použije se aktuální cesta, jinak se vytvoří nová cesta s nejlepším výsledkem za pomocí funkce construct_new_path.

Jak je zmíněno v čtvrtém bodě, je zde důležitá věc, a to, že se začíná iterovat nad zvoleným počtem sousedů. Zde jsou důležité kroky, které se musí v iteraci odehrát:

- Vybere se místo, buď nejbližší nevybraný soused od předchozího vybraného místa, nebo od místa, nad kterým iterujeme.
- Vždy musíme zkontrolovat, jestli místo vybrané pomocí nejbližšího souseda není levá nebo pravá hranice, a to za pomocí funkce check_border. Zde je třeba si představit, jak to bude fungovat a proč tady vlastně odlišujeme hranice, jak je zmíněno v předchozím seznamu v bodě 3. Spočítáme vzdálenost aktuálního místa po odpojení, což znamená, že pravá a levá hranice nejsou z jedné strany napojené. Takže pro hranice jen odečteme jednu stranu a musíme hranice nahradit novou hranicí. Pokud místo není hranice, musíme od jejího levého a pravého místa odpojit a tyto místa spojit k sobě, a k tomu využíváme funkci calculate_distance_after_remove_point.
- Musíme zkonstruovat cestu mezi místy, které jsme už odpojili, a to včetně levé a pravé hranice. Zde si musíme představit, že řešení vkládáme mezi ně.
- Nakonec spočítáme vzdálenosti mezi místy, které nám vrátí 2-opt algoritmus, a zjistíme, jestli není v iteraci sousedů nejlepší. Pokud je, uložíme si ji.

4.2.3 cycle

Jednoduchá pomocná funkce, která z pole vytvoří cyklus. Volá se tak, že ji předáme aktuální index, informaci o tom, jestli chceme pravý nebo levý index od aktuálního indexu a velikost cesty. Pokud chceme získat pravý index a zároveň je aktuální index poslední v poli, vrátíme první index. Stejně tak, pokud chceme levý index od prvního indexu, vrátíme ten poslední v poli. V ostatních případech získáme pravý index tak, že přičteme k aktuálnímu indexu 1 a levý index tak, že od aktuálního indexu odečteme 1.

4.2.4 calculate_distance_after_remove_point

Funkce, která má spočítat vzdálenost po odebrání místa od jeho levé a pravé hranice a spojení hranic k sobě. K určení pravé a levé hranice použijeme funkci cycle. Poté jen spočítáme vzdálenost jako součet vzdálenosti od levé hranice ke zvolenému místu, vzdálenosti od zvoleného místa k pravé hranici a odečteme vzdálenost od levé hranice k pravé hranici.

4.2.5 construct_new_path

Funkce, která vytvoří novou cestu z části cesty hranic a staré cesty. Nejprve si vytvoříme cestu takovou, která neobsahuje novou část cesty ze staré cesty, kromě levé hranice, která je uložena na nultém indexu. Poté za levou hranici vložíme novou část cesty a vrátíme novou cestu.

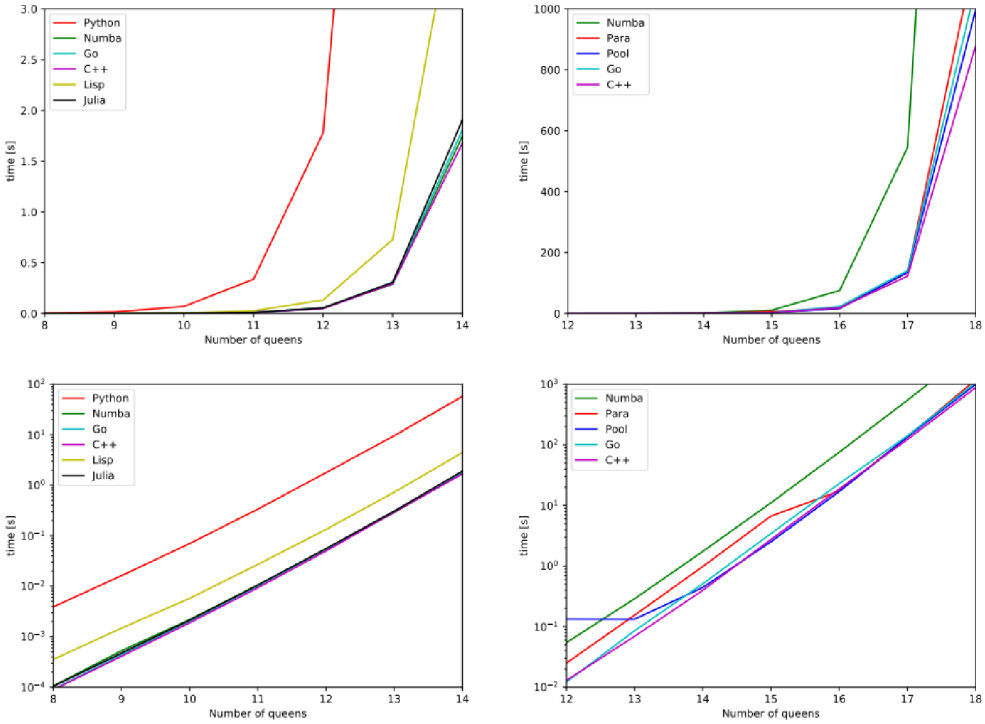
Kapitola 5

Struktura a implementace programu

5.1 Výběr jazyků

Pro svou jednoduchost a rychlost implementace jsem se rozhodl použít jazyk Python, který jsem použil na části programu, kde rychlost nebyla hlavním faktorem, například pro grafické rozhraní a skriptu, který slouží k využití všech jader procesoru. Pro druhou část programu, kde byla rychlost velmi důležitým faktorem, jsem se rozhodl použít jazyk C++. Také jsem zvažoval, zda použít knihovnu Numba pro implementaci algoritmů. Numba je just-in-time kompilátor pro jazyk Python, který nejlépe funguje s kódem, který využívá pole a funkce v NumPy, a s cykly. Nejběžnějším způsobem použití Numba je pomocí dekorátorů, které lze aplikovat na funkce, aby se Numba naučila, jak je optimalizovat. Pokud je poté zavolána funkce s dekorátorem, je přeložena do strojového kódu pro okamžité vykonání a celý nebo část kódu může následně běžet rychlostí nativního strojového kódu [3]. Rozhodování, jestli zvolím jazyk Python společně s knihovnou Numba nebo jazyk C++, bylo složité, ale nakonec jsem zvolil jazyk C++, i díky srovnání rychlosti při paralelizaci, které je vidět na pravé straně grafu v obrázku 5.1, převzatém z [11]. Tento graf ukazuje rychlost jednotlivých jazyků při řešení problému N dám.

Implementoval jsem také Ant Colony Optimization Algorithm a Nearest Neighbor Algorithm v jazyce Python. Jedním z problémů při použití Pythonu je problém s využíváním vláken. Globální zámek interpretu Pythonu, zvaný GIL, je mutex (nebo zámek), který umožňuje pouze jednomu vláknu ovládat interpret Pythonu. To znamená, že pouze jedno vlákno může být v režimu spuštění v jakémkoli bodě v čase. Dopad GIL není viditelný pro vývojáře, kteří spouštějí jednovláknové programy, ale může být úzkým místem výkonu v programování vázaném na procesor a vícevláknovém kódu. Informace převzatá z [12]. Celkově to zabraňuje paralelizaci jednotlivých úloh na úrovni vláken. Jelikož v **Ant Colony Optimization Algorithm** každého mravence reprezentuje vlákno, tak nedává smysl Python využívat.



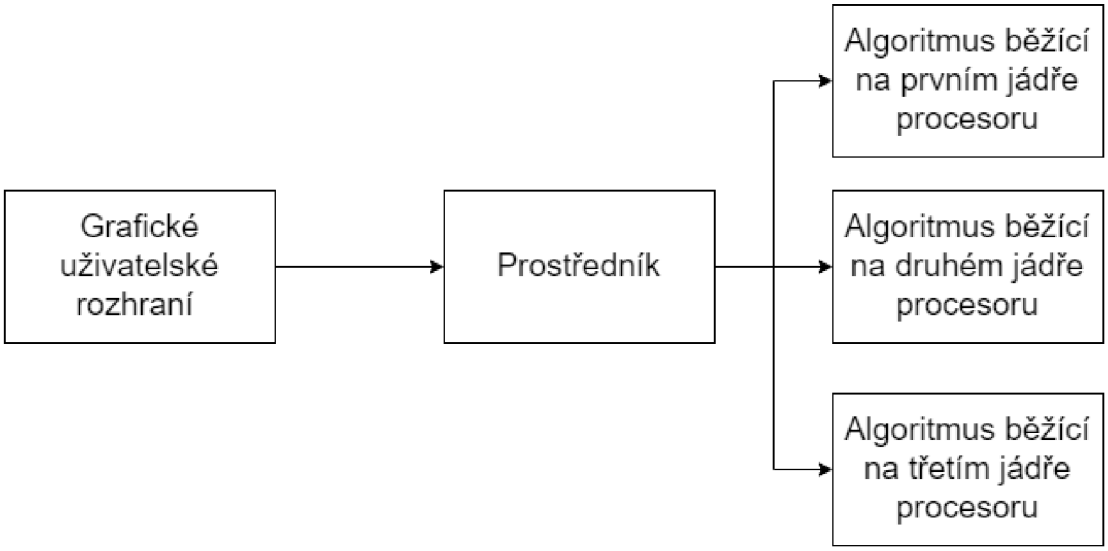
(a) (b)

Figure 2: Run times as a function of the board size. Linear scale at the top and log scale at the bottom. (a) Sequential. (b) Parallel.

Obrázek 5.1: C++ vs Numba

5.2 Struktura aplikace

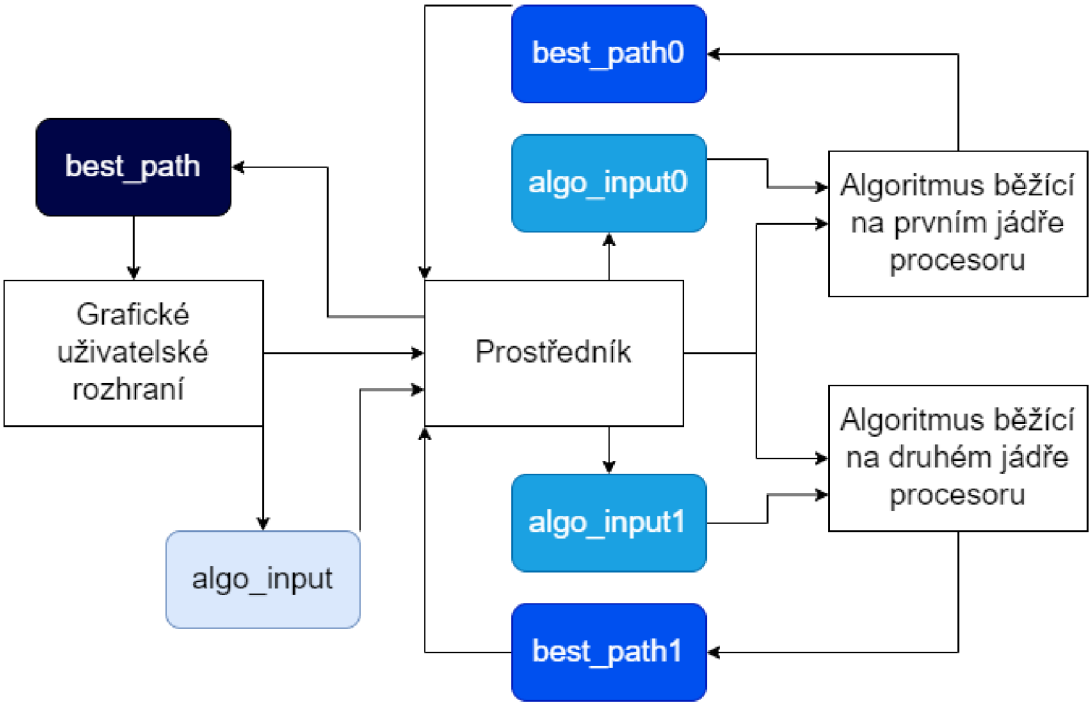
Jednotlivé části aplikace jsou zobrazeny na obrázku 5.2. Po spuštění aplikace se zobrazí **grafické uživatelské rozhraní**, které umožňuje zobrazovat místa a jejich cesty a dokáže místa i generovat. Po stisknutí tlačítka **Solve** se grafické uživatelské rozhraní rozdělí na 2 vlákna. V jednom bude běžet grafické uživatelské rozhraní a v druhém se za pomocí python interpretu spustí **Prostředník**. Prostředník nejprve přeloží algoritmus za pomocí optimalizačního přepínače **O3**. O3 instruuje kompilátor, aby optimalizoval výkon generovaného kódu a ignoroval velikost generovaného kódu, což může vést k většímu objemu kódu [1]. Následně prostředník zjistí počet jader systému a poté ze souboru algo_input přečte, kolik jader uživatel chce použít. Pokud je počet jader, které chce uživatel použít, větší než reálný počet jader procesoru, prostředník sníží zvolený počet na maximální možný. Poté se na každém jádru spustí algoritmus. Na obrázku 5.2 je vidět, jak by to fungovalo, pokud by byl počet zvolených jader 3 (samozřejmě by ty jádra musela být k dispozici).



Obrázek 5.2: Struktura aplikace

5.3 Způsob komunikace

Pro svou jednoduchost a také díky tomu, že mě řešení hned napadlo, jsem se rozhodl použít ke komunikaci mezi jednotlivými částmi aplikace, neboli procesy, komunikaci pomocí souborů. Další výhodou je nezávislost na platformě a to, že soubory mohou fungovat nezávisle na platformě. Na následujícím obrázku 5.3 je vidět vztah při vytváření souborů a následném zapisování do nich. Vytváření souborů probíhá od nejsvětlejšího k nejtmavšímu. Po kliknutí na tlačítko Solve v grafickém uživatelském rozhraní se vytvoří soubor algo_input, ve kterém je uložené nastavení algoritmu společně s místy a jejich vzdálenostmi mezi nimi. Následně tento soubor vezme prostředník a nakopíruje ho pro každé jádro, přidá k tomu číslo na obrázku - algo_input0 a algo_input1. Kopie se vytvářejí kvůli tomu, aby nedocházelo k chybám při čtení a zapisování na více procesech najednou. Počet zvolených jader prostředník přečte z algo_input. Následně je každému algoritmu při exekuci na STDIN přiděleno ID, podle toho pozná, ze kterého souboru má číst. Na obrázku 5.3 algo_input0 slouží k nahraní nastavení algoritmu a míst společně s vzdálenostmi do algoritmu běžícího na prvním jádře procesoru. To samé platí pro algo_input1 a algoritmus běžící na druhém jádře procesoru. Po dokončení všech iterací algoritmu rozhodne algoritmus, která cesta byla nejkratší. Vzdálenost společně s navštívenými místy je uložena do souboru best_path + ID. Prostředník poté načte všechny soubory best_path + ID a vyhodnotí, která vzdálenost je nejkratší. Tu pak uloží do souboru best_path, který pak grafické uživatelské rozhraní přečte a zobrazí.



Obrázek 5.3: Komunikace aplikace

Kapitola 6

Uživatelské rozhraní

6.1 Informace o řešení a struktura GUI

Grafické uživatelské rozhraní je implementováno pomocí modulu **Tkinter** v jazyce **Python**. K tomu, aby bylo grafické uživatelské rozhraní pro uživatele přívětivé a snadno použitelné, zvolil jsem jednoduchý a moderní design s pomocí knihovny **CustomTkinter**. Snažil jsem se také zvolit vhodné názvy pro jednotlivé popisky k prvkům, aby bylo jasné, co vlastně dělají. Dále jsem dodržoval konzistenci mezi jednotlivými prvky, včetně jejich stylu a chování, aby se zlepšila celková použitelnost grafického rozhraní a zabránilo se zmatení uživatele. Aplikace využívá chybové a informativní hlášky pro uživatele, aby byl uživatel informován, co dělá špatně. Grafické uživatelské rozhraní se skládá ze tří oken. Hlavní okno aplikace obsahuje plochu pro vygenerovaná místa a tlačítka pro ovládání nebo otvírání vedlejších oken. Další okno je určeno pro import míst z mapy, kde se nachází mapa a tlačítka pro ovládání. Poslední okno slouží k nastavení konkrétních algoritmů. Implementačně jsem jednotlivé okna rozdělil na třídy, každá v jiném souboru. Celý programový cyklus běží v hlavním okně a nepředává se nijak kontrola jiným oknům. Po skončení aplikace se, pro korektní ukončení aplikace, zjistí, jestli neběží jedno z vedlejších oken. Pokud vedlejší okno běží, tak se ukončí.

6.2 Hlavní okno

Hlavní účel hlavního okna je zobrazení míst a cest mezi nimi. Plocha pro zobrazení míst se nachází uprostřed obrazovky na obrázku 6.1. Jednotlivé místa jsou reprezentovány bílými tečkami 6.2, cesty mezi nimi pomocí černých čar 6.3.

6.2.1 Widgety pro ovládání hlavního okna

Nachází se v levé části programu, jak je vidět na kterémkoliv z obrázků 6.1 6.2 6.3. Zde je jejich seznam a k čemu slouží.

- **Place generation mode** *výběrové menu* – Pro zvolení možnosti jakou chceme generovat místa náhodně, z souboru nebo z mapy.
- **Algorithm types** *výběrové menu* – Pro zvolení jaký konkrétní algoritmus chceme použít pro řešení problému.
- **Solve** *tlačítko* – Začne řešit problém obchodního cestujícího mezi vygenerovanými místy zvoleným algoritmem v **Algorithm types**.

- **Stop** *tlačítko* – Přestane řešit problém obchodního cestujícího nebo přestane generovat místa. Slouží k tomu, aby uživatel nemusel ukončovat aplikaci nebo čekat na vyřešení problému, pokud se rozhodl v průběhu, že ho vlastně nechce řešit, nebo zjistil, že by to trvalo příliš dlouho. Neslouží k zastavení a k následnému spuštění. Sice se po zmáčknutí tlačítka práce nesmaže. Ale to nastane až po dalším kliknutí na tlačítko **Generate** nebo **Solve**.
- **Clear roads** *tlačítko* – Slouží k smazaní cest mezi jednotlivými místy.
- **Algorithm settings** *tlačítko* – Otevření nastavení zvoleného algoritmu v **Algorithm types**
- **Appearance mode** *výběrové menu* – Slouží k přepínání mezi tmavým 6.2 a světlým módem 6.1.
- **UI scaling** *výběrové menu* – Možnost měnit velikost widgetů od 80% do 150%

Dále se zde nacházejí widgety závisle na zvoleném **Place generation mode** Při zvolení možnosti **Random** 6.2. Generujeme místa do plochy pseudonáhodně. A zobrazí se pouze následující widgety:

- **Number of places** *textové pole* a *posuvník* – Textové pole určuje, kolik míst chceme vygenerovat, a je propojené s posuvníkem. Ten nabývá hodnot od 0 do 1000. Je zde možnost zadaní i větší hodnoty, a to ručním vepsáním hodnoty do textového pole. Pokud je špatný formát, například když tam napíšete nenumerický znak, vyskočí na vás chyba.
- **Generate** *tlačítko* - vygeneruje zvolený počet míst v textovém poli do modré plochy uprostřed obrazovky.

Při zvolení možnosti **From Map** 6.1 chceme použít reálnou mapu pro generování míst. A zobrazí se následující widgety.

- **Add places** *tlačítko* – Přidá možnost přidat další místa k těm aktuálním. Spustí nové okno které složí k importu míst z reálné mapy, ale předchozí vygenerovaná místa se nesmažou.
- **Import places** *tlačítko* – Dělá úplně to samé jako tlačítko **Add places** s tím rozdílem, že se předchozí místa smažou.

Poslední možností je import souboru **From file** 6.3 v kterém jsou uložené místa a jejich souřadnice. Jak můžeme vidět na obrázku Pro změnu se nám zobrazí zase:

- **Select file** *tlačítko* – Po stisknutí tlačítka se zobrazí dialog pro výběr souboru. Zvolíme soubor s místy jejich souřadnicemi. A následně se nám zobrazí v modré ploše pro místa.

6.2.2 Pravidla pro používání tlačítek

Pravidla pro používání tlačítek nejsou složité, ale asi by bylo dobré je zmínit. Pokud stisknete tlačítko **Generate** nebo **Solve** vytvoří se nové vlákno, kde běží generování míst nebo řešení algoritmu. To slouží k tomu, aby Tkinter aplikace nezamrzla. Další věc, která nastane po stisknutí tlačítek je, že se nastaví flag, který říká, že něco běží na pozadí na True. Následně pokud chceme zase stisknut některé z tlačítek, nebude nám to umožněno, dokud se aktuální činnost nedokončí. Poté se flag nastaví zpět na False.

6.2.3 Vyskakovací informativní a chybové hlášky

Existují celkem 4 informativní a 1 chybová hláška.Příklad chybové hlášky
Informativní 6.5 jsou:

- **You didn´t generate places before running the solution**

  - Hláška se vás snaží upozornit na to, že jste chtěl zapnout řešení algoritmu předtím než jste vygeneroval nějakým způsobem místa. Proto aby se vám nezobrazovala můžete například náhodně vygenerovat místa nebo je nahrát ze souboru, či z mapy.

- **You can´t start new solve while still solving**

  - Tato hláška vás upozorňuje na to, že jste se pokusil nové řešení když už nějaké probíhá. Proto aby se vám nezobrazovala musíte počkat na to než se dokončí aktuální řešení. A teprvé spustit nové.

- **You can´t generate new places while still solving**

  - Tato hláška je velmi podobná té předchozí s tím rozdílem, že se snažíte vygenerovat nové místa, když stále probíhá řešení aktuálního problému. A zase musíte počkat k dokončení aktuálního problému před generováním nových míst.

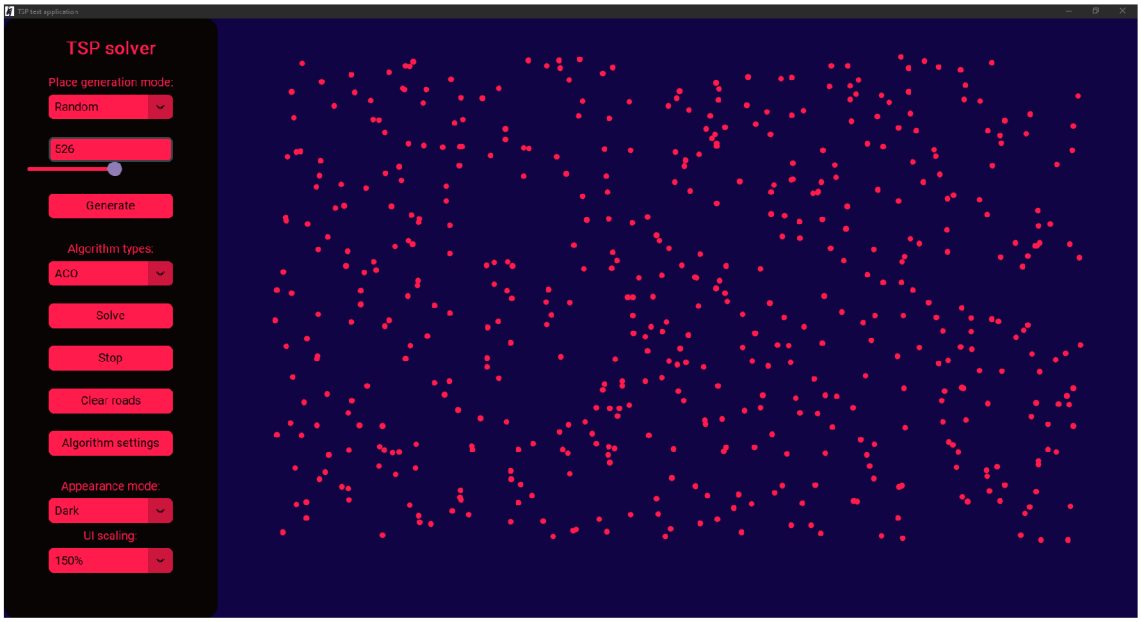
- **Distance: [distance] Count of visisted places: [count of visited places]**

  - Poslední z informativních hlášek vám po dokončení řešení problému obchodního cestujícího vypíše celkovou vzdálenost, která se zobrazí i v dolní části uprostřed obrazovky. Ještě zobrazí počet míst, které navštívil.

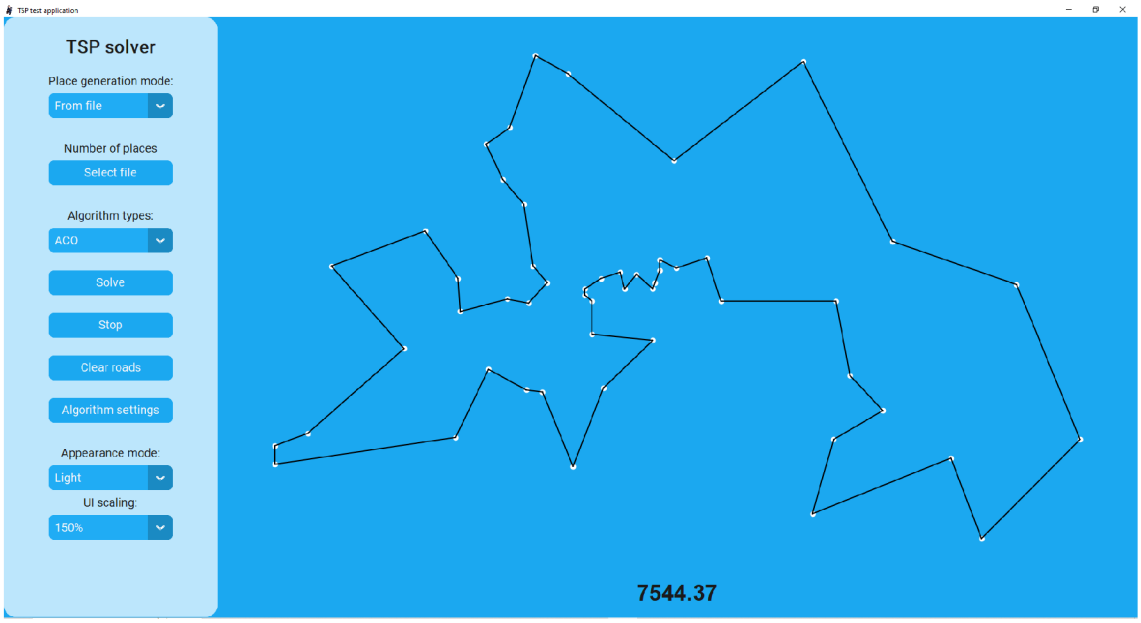
Chybová 6.4 je:

- **Wrong format of places count Right format is [int] > 1** - Hláška se vás snaží upozornit, že nemůžete vygenerovat místa, jelikož jste do textového pole pod **Number of places** zadal hodnotu, která není číslo, které jde převést na číslo typu integer. Nebo je číslo menší jak 2 v tom případě nemá smysl problém obchodního cestujícího pro tolik míst ani řešit.

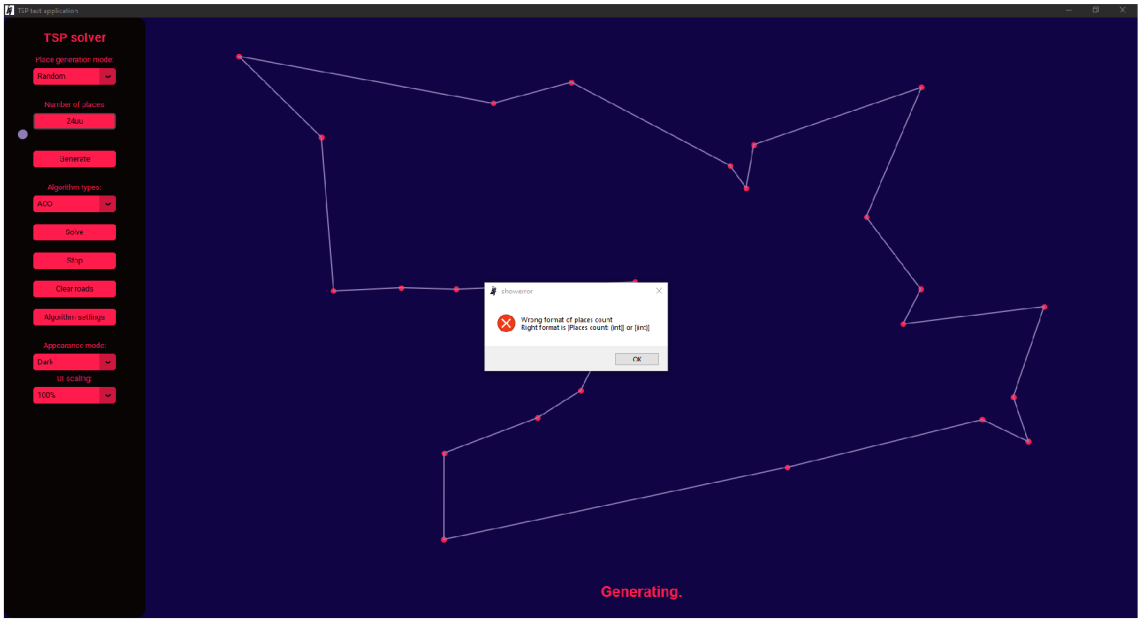
Obrázek 6.1: Hlavní okno aplikace



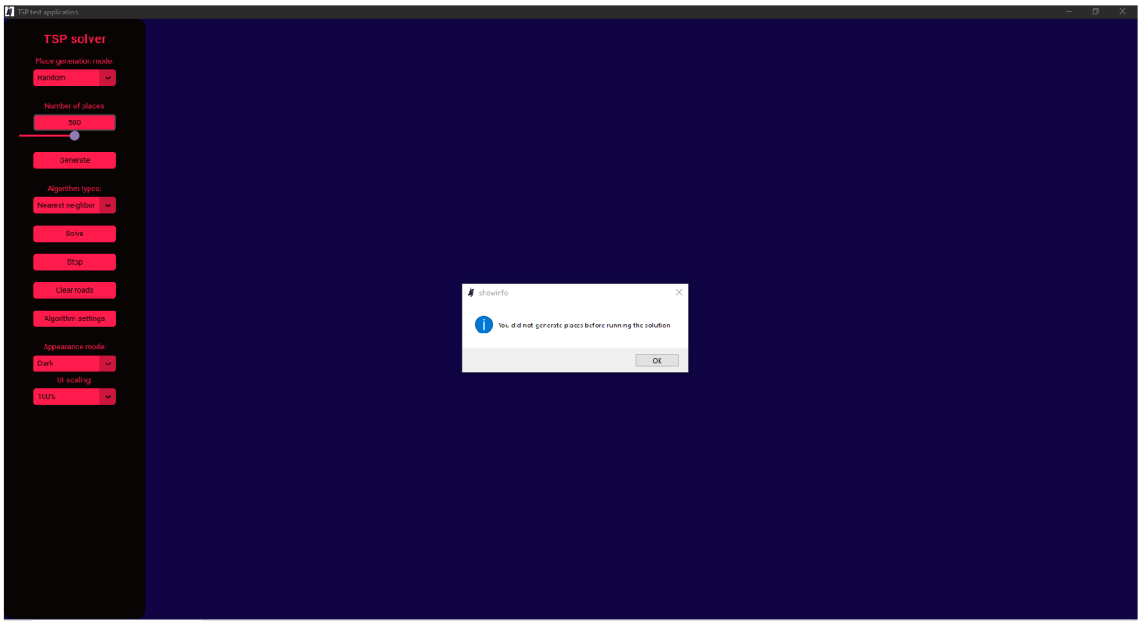
Obrázek 6.2: Hlavní okno aplikace dark mode a vygenerované místa



Obrázek 6.3: Hlavní okno aplikace s vyřešeným problémem obchodního cestujícího(berlin52)



Obrázek 6.4: Chybová vyskakovací hláška



Obrázek 6.5: Informativní vyskakovací hláška

6.3 Okno pro import míst z mapy

Další okno aplikace do kterého se můžete dostat slouží k importu míst z reálné mapy. Implementačně jsem to zpracoval pomocí knihovny od stejného autora jako **CustomTkinter**, takže zde není nějaké problematické spojení těchto dvou knihoven dohromady. Dokonce jsou na githubu konkrétní příklady spojení těchto knihoven dohromady[13]. Autorem je Tom Schimansky. Obrazovku pro import míst z mapy můžeme vidět na obrázku 6.6.

6.3.1 Ovládání

Po levé straně se nacházejí **widgety** pro ovládání. Zde je jejich seznam:

- **Remove last marker** *tlačítko* – Smaže poslední značku tu s největším číslem.
- **Remove all marks** *tlačítko* – Smaže všechny značky, pokud nějaké jsou.
- **Import marks** *tlačítko* – Zavře okno a nahraje značky, které jsme zvolili do hlavního okna.
- **Tile server** *výběrové menu* – Slouží k tomu jaký server pro mapy chceme použít celkově to mění design mapy.

Celkové ovládání mapy je velmi jednoduché a intuitivní. Kliknutím levého tlačítka myši kam chceme, se objeví značka, tj. místo, které chceme nahrát do hlavní obrazovky. Obrazovku můžeme posunout pomocí stisknutí levého tlačítka a hýbání s ní. Pokud chceme přiblížit nebo oddálit, slouží nám + a - v levé horní části obrazovky či kolečko na myši.

6.3.2 Z mapy do hlavního okna

Jak jsem již zmínil, po stisknutí tlačítka **Import marks** se jednotlivé značky nahrají do hlavního okna. Aby to fungovalo, musejí se udělat dvě důležité věci. První je spočítání

vzdálenosti mezi dvěma místy na planetě. Konkrétně používám funkci **distance.geodesic** z knihovny geopy[5]. Jelikož Země není koule, ale geoid, funkce využívá **Vincentyho formuli**. Původně jsem si myslel, že použiji Haversinovu formuli, která počítá vzdálenost dvou bodů na sféře. Vzorec Vincentyho formule je docela složitý a nemá s projektem moc společného, tak ho tady nebudu uvádět, ale můžete ho najít například na [14].

Druhou důležitou věcí je celková normalizace, jelikož plocha která zobrazuje místa je v rozsahu 0-1 pro x i pro y. Je to kvůli tomu, aby se nemusela pro nové místa plocha zbytečně mazat. Vzorec pro normalizaci je následovný [16]. Předpokládejme, že x je list prvků x_i.

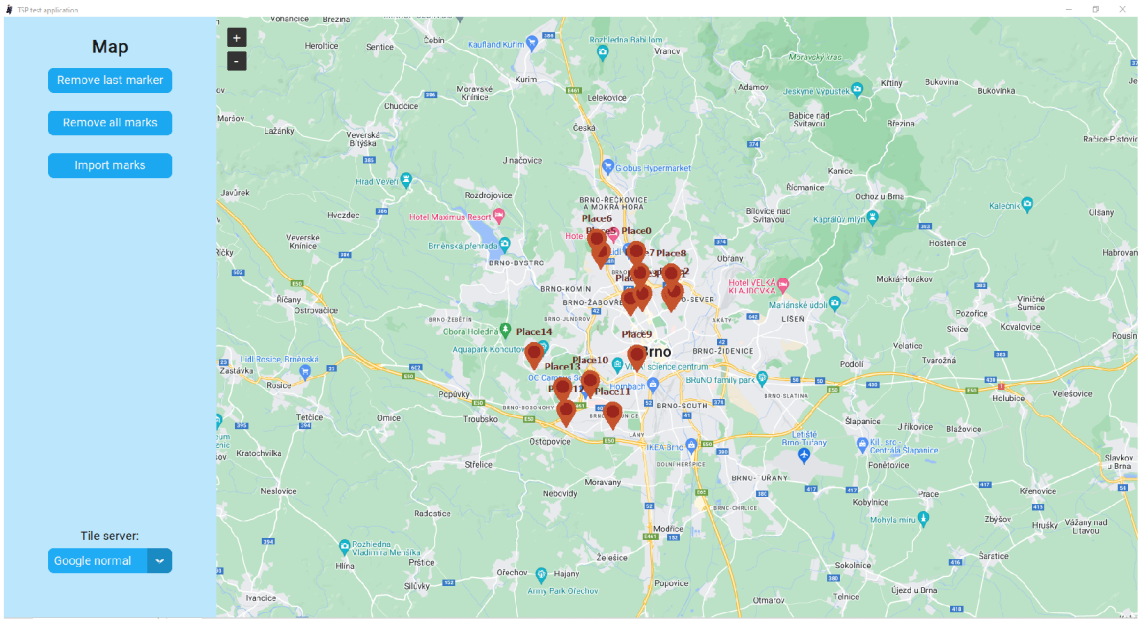
$$normalized\_x_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

Já využívám souřadnice x i y proto jsem provedl následující úpravy:

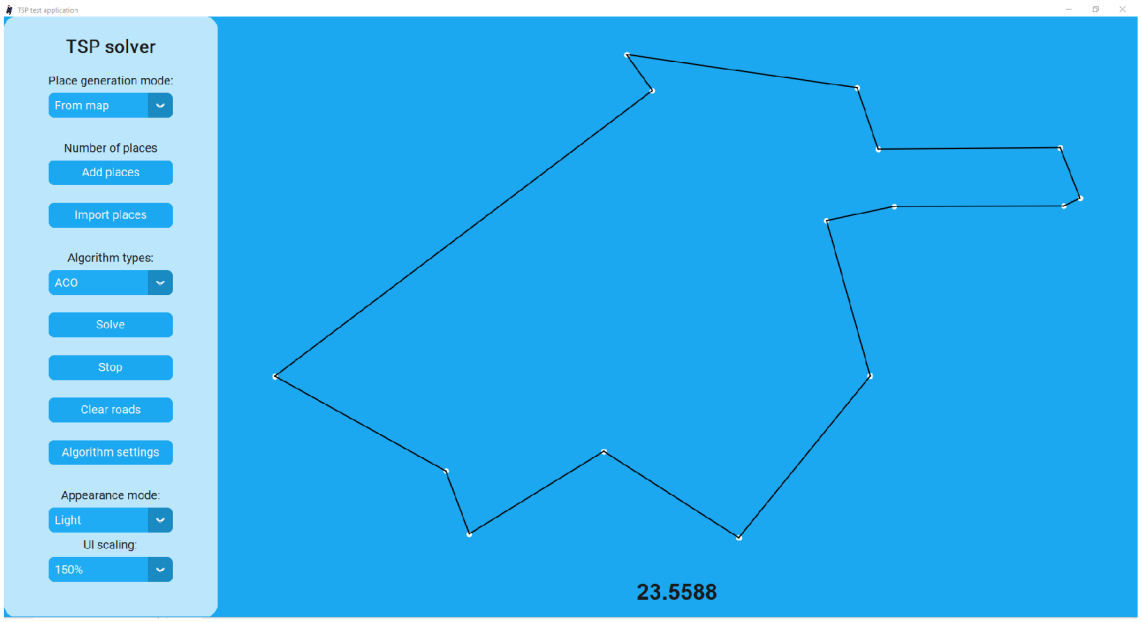
$$xdiv = \max(x) - \min(x)$$
$$ydiv = \max(y) - \min(y)$$
$$\text{div} = \text{xdiv if xdiv} > \text{ydiv else ydiv}$$
$$normalized\_x_i = \frac{x_i - \min(x)}{div}$$
$$normalized\_y_i = \frac{y_i - \min(y)}{div}$$

Celkově to slouží k tomu, aby se zachoval poměr mezi x a y souřadnicemi na mapě. Pokud bychom zvolili dvě místa v libovolném poměru, vždy bude jedno místo v levém horním rohu a druhé v pravém dolním rohu, což platí, pokud pro x a y využijeme první vzorec. Druhý vzorec tomu zabraňuje tím, že dělitel je vždy stejný pro x i y. Samozřejmě se zde uchovává i poměr mezi x a y souřadnicemi, ale jelikož Země není plochá, ale má geoidový tvar, nebude to zcela přesné, ale slouží to pouze k zobrazení míst. Máme již vypočítané celkové vzdálenosti celkem přesně, takže to není velký problém.

Na obrázku 6.6 vidíme, že jsme zvolili celkem 15 míst, a to jsou některé školy v Brně, ale ne všechny. Abychom mohli vypočítat nejlepší cestu mezi nimi, musíme stisknout tlačítko pro **Import marks**. Následně se okno pro mapu vypne a na hlavním okně se zobrazí místa na ploše podobně jako na obrázku 6.7, s tím rozdílem, že tam ještě nebudou černé čáry mezi místy. Vybereme algoritmus, jakým chceme problém řešit, a klikneme na tlačítko **Solve**. Poté bude hlavní okno vypadat přesně jako na obrázku 6.7. V dolní části obrazovky se zobrazí celková vzdálenost mezi všemi místy, která je 23.5588. Černé čáry mezi místy značí cesty mezi nimi. Toto je nejkratší možná cesta, kterou algoritmus našel.



Obrázek 6.6: Okno pro import míst z mapy s značkami



Obrázek 6.7: Hlavní okno s nahranými značkami z mapy

6.4 Okno pro nastavení Ant Colony Optimization

První okno sloužící pro nastavení konkrétně pro Ant Colony Optimization je na obrázku 6.8. Okno je velmi jednoduché pouze název parametru, který můžete měnit a textové pole, pro jeho zadání. Hodnoty už jsou přednastavené. Pokud najedete na nějaké textové pole zobrazí se vám informace z jakého rozsahu je můžete vybírat a jaký by měl být optimální. Okno kontroluje pouze typ čísla int či float a ne rozsah. A jestli je není číslo menší než 0.

Ale nekontroluje se, jestli zadáte například víc jader než máte. Předpokládám, že v tom případě by to Python script pro paralerní spouštění na více jádrech. Nespustil paralérně, ale vždy by čekal na uvolnění některého z jader. Silně nedoporučuji využívat všechny jádra zvlášť pro velké problémy a ještě v kombinaci s moc agenty tedy vlákny. Může se stát, že váš počítač bude neovladatelný, kvůli vytížení procesoru. Může se to jevit jako problém, ale já v tom problém nevidím, jelikož uživatel aplikace by měl zvážit co by mělo být pro jeho problém optimální nastavení. Osobně doporučuji využívat u větších problémů 500 míst a víc. okolo 25% jader procesoru a okolo 50 agentů. Pro menší problémy do 50 míst můžete využít klidně všechny jádra procesoru a 100 agentů.

6.4.1 Jednotlivé parametry v nastavení

Popíši zde pouze pravou část nastavení, jelikož k levé části je už víc v 3.6.6

- **Iterations** *int* – Počet iterací ACO algoritmu, pouze pro jednoho agenta. Počet iterací musí být větší jak 0. A optimální je tak od 100 iterací do 1000 iterací, hodně záleži na počtu agentů. Všiml jsem si, že od nějakého počtu iterací ACO už lepší cestu najde jen velmi nepravděpodobně. Celkový počet iterací algoritmu je roven vztahu:

$$\text{Count of agents} * \text{Number of cores} * \text{Iterations}$$

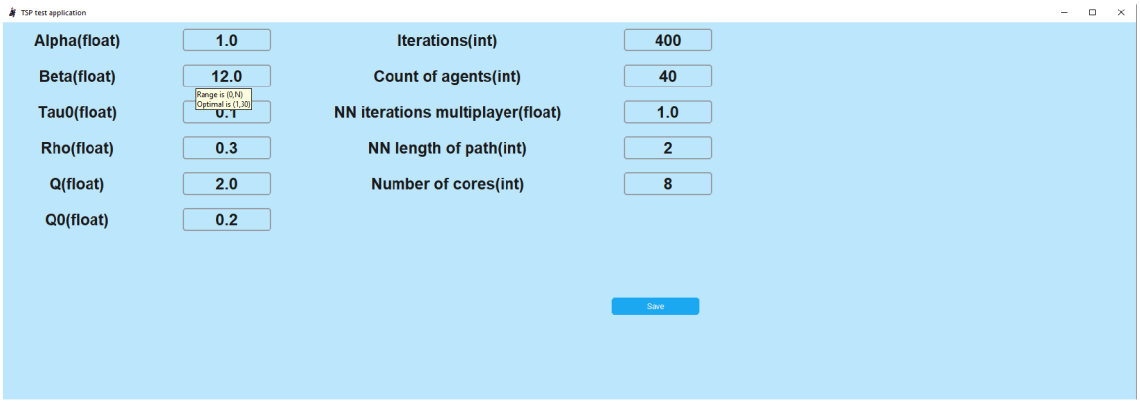
- **Count of agents** *int* – Počet agentů(vláken) nebo chcete-li mravenců. Kteří procházejí každý svými iteracemi a aktualizují globální feromony společné pro všechny mravence. Počet agentů by měl být větší než 0, což je samozřejmé. Doporučuji tak od 5 agentů do 100 agentů, zase záleží na počtu iterací počtu využitých jader a velikosti problému.

- **NN iterations multiplayer** *float* – Tento parametr slouží k vynásobení celkového počtu míst tímto parametrem a získání z toho počet iterací Nearest Neighbor Algorithm. Pro nastavení prvotních hodnot τ. Ideální hodnota je tak okolo 1, ale reálně nastavení τ pomocí Nearest Neighbor Algorithm nebude mít velký vliv na řešení problému. Jediné k čemu slouží je, aby bylo prvotní cesta lepší než čistě přes ACO, tím se sníží počet iterací. Zde je vzorec:

$$\text{Počet iterací NN} = \text{NN iterations multiplayer} * \text{Počet míst}$$

- **NN lenght of path** *int* – Tento parametr taky slouží k prvotnímu nastavení τ. Je to celková délka cesty. Měl by být větší jak 0, ale pokud nastavíte 0, tak se Nearest Neighbor Algorithm prostě nepoužije. Není dobré ani nastavovat moc velkou délku cestu. Protože algoritmus už bude mít všechny výhodné místa spojené tím pádem připadnou méně atraktivní cesty. Ideální je tak okolo 2. Když si to zkusíte představit většinou to budou cesty, které se budou využívat pro optimální řešení samozřejmě ne vždy.

- **Number of cores** *int* – Zde se určuje počet jader, na kterých bude algoritmus spuštěn nezávisle na ostatních. Popis jak funguje komunikace a vybírání cesty je detailněji popsaný už v 5.3.



Obrázek 6.8: Okno pro nastavení Ant Colony Optimization

6.5 Okno pro nastavení mého algoritmu

Toto další okno slouží k nastavení mých algoritmů. Abyste se k němu dostali, stačí změnit **Algorithm types** na **MyAlgo**. Provádí jednoduché kontroly, zda jsou zadaná čísla typu celého čísla větší než 0. Dále zde naleznete 2 přepínače.

6.5.1 Jednotlivé parametry v nastavení

- **Start NN size** – Jedná se o velikost, do které se mohou přidávat sousedící prvky, s kterými se tvoří lepší řešení pomocí 2-opt algoritmu.
- **Middle NN size** – Jedná se o střední velikost, do které se mohou přidávat sousedící prvky, s kterými se tvoří lepší řešení pomocí 2-opt algoritmu.
- **Last NN size** – Jedná se o poslední velikost, do které se mohou přidávat sousedící prvky, s kterými se tvoří lepší řešení pomocí 2-opt algoritmu.
- **Start NN count** – Počet prvotně vygenerovaných řešení, na které nebyla aplikována **Last NN size**, a vybírá se vždy nejlepší z nich.
- **Rounding** – Zaokrouhlování vzdáleností, které používají lidé s kterými jsem porovnával výsledky. Nejspíš kvůli tomu, aby nemuseli vzdálenosti ukládat, jako desetinné číslo, ale jako celé číslo.
- **Segments** – Použití optimalizace tím, že se řešení rozdělí na segmenty a v nich se náhodně generuje řešení a aplikovává se stejný postup, jako u normálního řešení. Tímto se snažíme segmenty zlepšit.

Kapitola 7

Testování

Testování jsem prováděl na počítači s 64bitovým operačním systémem Windows 10 Education. S procesorem AMD Ryzen 7 3750H with Radeon Vega Mobile Gfx 2.30 GHz a 5,93 GB RAM paměti. Do 1500 míst jsou testovány oba algoritmy od 1500 míst, jelikož ACO začíná být neefektivní a ne tak účinné jako můj algoritmu, provádím testy pouze mého algoritmu. **Potom co jsem provedl většinu testů jsem si uvědomil, že nepoužívají vzdálenost typu double, ale int, což mi reálně zhoršilo výsledky**. Nepřepsal jsem algoritmus k tomu, aby pracoval s inty i když by to určitě urychlilo výpočet, ale bylo by zde problematické pracování s náhodně generovanými místy, jelikož ty jsou jen v 0-1 rozsahu. Pouze je možnost zapnout zaokrouhlování. **Zaokrouhlování je potřeba zapnout před generováním míst tedy počítání vzdáleností mezi nimi** . Pro některé sady to bude mít minimální vliv, protože například, jestli bude vzdálenost 400000 nebo 400001 je irelevantní změna a výslednou cestu to nijak nezmění. Sady na, které ovšem tato skutečnost bude mít vliv jsou místa generovaná uměle, jelikož jejich vzdálenosti jsou o dost menší než ty z reálných map.

7.1 Popis tabulky mého algoritmu

- **Délka cesty** – Délka nejlepší cesty, kterou algoritmus našel.
- **Doba běhu** – Doba běhu algoritmu i s načítáním míst z souboru a zápisem nejlepší cesty do souboru.
- **Bez souboru** – Doba běhu algoritmu s už načtenými místy a bez zápisu nejlepší cesty do souboru.
- **Chybovost** – Procentuální chyba řešení.
- **SNN** – Počáteční počet sousedů v nastavení je reprezentovaná jako *Start NN size*.
- **MNN** – Střední počet sousedů v nastavení je reprezentovaná jako *Middle NN size*.
- **LNN** – Poslední počet sousedů v nastavení je reprezentovaná jako *Last NN size*.
- **SC** – Počet prvotních řešení v nastavení je reprezentovaná jako *Start NN count*.
- **Segmenty** – Využití rozdělení na segmenty pro zlepšení řešení.

7.2 Popis tabulky ACO

- **Délka cesty** – Délka nejlepší cesty, kterou algoritmus našel.
- **Doba běhu** – Doba běhu algoritmu i s načítáním míst z souboru a zápisem nejlepší cesty do souboru.
- **Chybovost** – Procentuální chyba řešení.
- α – Reprezentuje důležitost vyloučeného feromonu na cestě.
- β – Reprezentuje faktor viditelnosti na cestě.
- ρ – Koeficient odpařování feromonu.
- **Q** – Intenzita feromonu, kterou představuje celkový feromon.
- **Q0** – Vyjadřuje s jakou pravděpodobností se zvolí nejlepší cesta podle aktuálního τ.
- **Ag** – Počet agentů.
- **It** – Celkový počet iterací
- **C** – Počet jader na kterých algoritmus běží.

7.3 Testovací sada berlin52

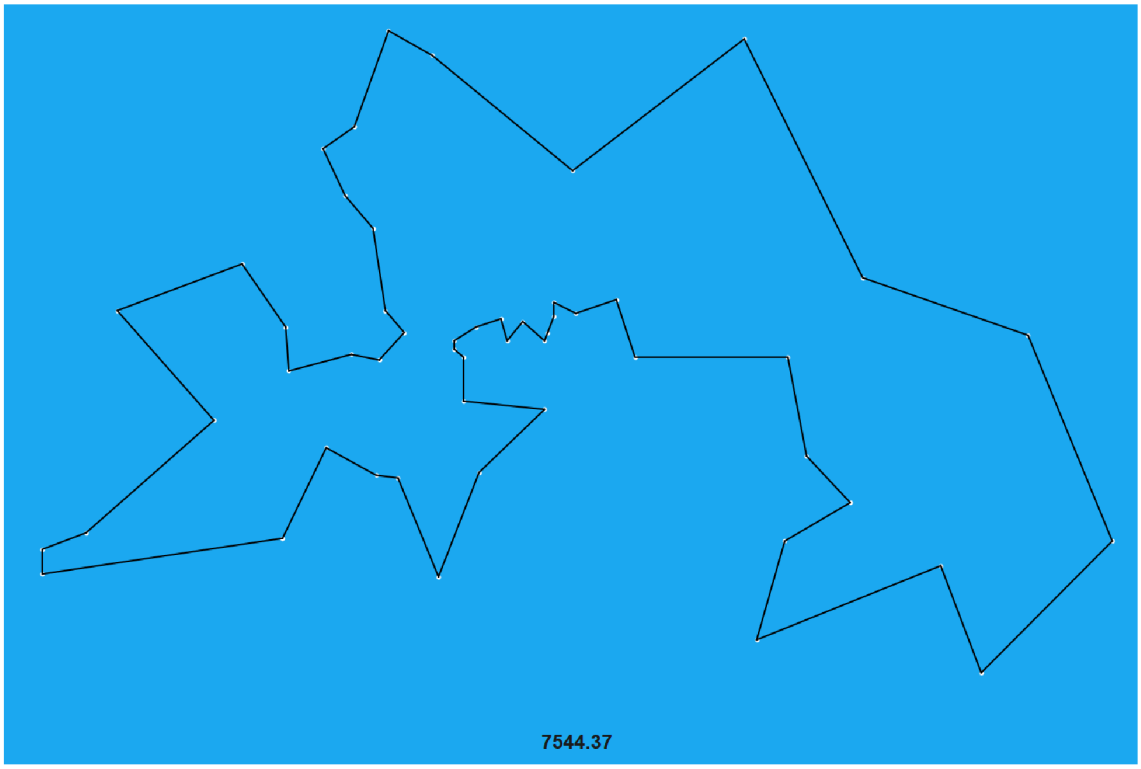
Pro testovací sadu s 52 místy berlin52 se oběma algoritmům povedlo nalézt optimální řešení. Které je dlouhé **7544.37**. Snímek optimálního řešení nalezeno oběma algoritmy je k dispozici: 7.1.

Tabulka 7.1: berlin52-ACO

Délka cesty	**Doba běhu**	**Chybovost**	α	β	ρ	**Q**	**Q0**	**Ag**	**It**	**C**
7544.37	6s	0%	1	3	0.1	200000	0.1	150	50	4
7826.0	6s	3.733%	1	3	0.1	200000	0.1	20	20	4
7658	6s	1.5062%	1	3	0.1	200000	0.1	20	100	4
7606.95	6s	0.8295%	1	13	0.1	200000	0.1	150	50	4

Tabulka 7.2: berlin52-MyAlgo

Délka cesty	**Doba běhu**	**Bez souboru**	**Chybovost**	**SNN**	**MNN**	**LNN**	**SC**	**Segmenty**
7544.37	6s	23ms	0%	1	12	33	4	Ne
7544.37	6s	14ms	0%	1	12	33	1	Ne
7544.37	6s	3ms	0%	1	4	11	1	Ne
7716.69	6s	3ms	2.2841%	5	6	11	1	Ne
7544.37	6s	2ms	0%	1	2	3	1	Ne



Obrázek 7.1: berlin52

7.4 Testovací sada QA194

Z testovací sady s 194 místy s optimální délkou **9352** se mi nepodařilo najít optimální řešení. Pokud však na tuto sadu aplikujeme náhodné přeskupování míst před spuštěním algoritmu, dokáže ho MyAlgo poměrně rychle najít. Tuto metodu jsem však nezkoušel otestovat na větším počtu míst, protože není praktická, a chtěl jsem sjednotit testy. Nicméně MyAlgo je v porovnání s ACO lepší ve všech aspektech. ACO využívá při použití více agentů nebo jader téměř 100% výkonu procesoru, zatímco MyAlgo nepoužívá více než 20%. Přesto se mu podaří najít mnohem rychleji lepší řešení. Oba algoritmy využívají 2-opt algoritmus pro zlepšení řešení.

Tabulka 7.3: QA194 – ACO

Délka cesty	**Doba běhu**	**Chybovost**	α	β	ρ	**Q**	**Q0**	**Ag**	**It**	**C**
9976.41	7s	6.6768%	1	3	0.1	200000	0.1	100	50	1
9690.9	3m 20s	3.6238%	1	5	0.1	200000	0.1	300	150	6
9982.26	1m 10s	6.7393%	0.5	2	0.1	2000	0.1	50	400	6

Tabulka 7.4: QA194 – MyAlgo

Délka cesty	**Doba běhu**	**Bez souboru**	**Chybovost**	**SNN**	**MNN**	**LNN**	**SC**	**Segmenty**
9376.65	20s	15s	0.2636%	1	12	45	4	Ano
9409.45	6s	42ms	0.6143%	2	12	23	1	Ne
9461.1	6s	27ms	1.1666%	2	5	12	1	Ne
9509.82	6s	205ms	1.6876%	12	25	33	1	Ne
9998.15	6s	14ms	6.9092%	1	2	3	1	Ne
9391.19	18s	13s	0.4191%	1	2	3	10	Ano
9376.65	1m 21s	1m 15s	0.2636%	1	2	3	100	Ano

7.5 Testovací sada PKB411

Jedná se o uměle vygenerovanou sadu, takže je zde velký rozdíl v zaokrouhlování vzdálenosti. Po přepočtu na desetinná čísla by měla optimální cesta vzdálenost 1365,4. Problém je, že se to tak brát nedá, jelikož oni už ze začátku počítali s zaokrouhlenými vzdálenostmi. A optimální vzdáleností po zaokrouhlení je tedy vzdálenost **1343**. Jak zde vidíme mému algoritmu se podařilo už za 6 sekund bez optimalizací pomocí segmentů najít mnohem lepší výsledek jak ACO. Které muselo běžet 1 minutu a 17 sekund, aby našlo řešení s chybovostí skoro 7% oproti tomu MyAlgo běžel asi 6 sekund bez překladu a generování míst je to 282ms a našel řešení s chybovostí 3.5%. Nejlepší mnou nalezené řešení nalezl MyAlgo za 33 sekund s chybovostí zhruba 0.6%.

Tabulka 7.5: PKB411 – MyAlgo bez zaokrouhlení

Délka cesty	**Doba běhu**	**Bez souboru**	**Chybovost**	**SNN**	**MNN**	**LNN**	**SC**	**Segmenty**
1361.08	32s	25s	-	1	12	23	4	Ano
1380.17	6s	703ms	-	1	12	23	4	Ne
1397.52	5s	197ms	-	7	12	23	1	Ne
1361.08	35s	29s	-	1	12	60	4	Ano
1367.72	27s	20s	-	1	5	5	1	Ano
1448.49	5s	96ms	-	1	5	5	1	Ne
1628.38	5s	19ms	-	1	1	1	1	Ne

Tabulka 7.6: PKB411 – ACO s zaokrouhlením

Délka cesty	**Doba běhu**	**Chybovost**	α	β	ρ	**Q**	**Q0**	**Ag**	**It**	**C**
1499	21s	11.6158%	1	3	0.1	200000	0.1	100	50	1
1440	1m 3s	7.2226%	1	13	0.1	200000	0.1	200	70	4
1436	1m 17s	6.9248%	0.5	30	0.1	200000	0.1	200	140	4

Tabulka 7.7: PKB411 – MyAlgo s s zaokrouhlením

Délka cesty	Doba běhu	Bez souboru	Chybovost	SNN	MNN	LNN	SC	Segmenty
1355	33s	26s	0.8935%	1	12	33	4	Ano
1351	33s	26s	0.5957%	5	12	23	1	Ano
1390	6s	282ms	3.4996%	5	12	23	1	Ne

7.6 Testovací sada DJA1436

Jedná se o uměle vygenerovanou sadu s 1436 místy. Zde budu testovat pouze testy s zaokrouhlením, jelikož je to umělá sada. Její optimální řešení má délku **5257**. Jak si můžeme všimnout mému algoritmu se oproti ACO podařilo najít rychleji mnohem kvalitnější cestu. Najít nejlepší cestu od ACO trvalo 41 minut a chybovost je poměrně velká 8,1% oproti tomu mému algoritmu trvalo najít cestu s chybovostí pouze 3,5% 8 sekund.

Tabulka 7.8: DJA1436 – ACO

Délka cesty	**Doba běhu**	**Chybovost**	α	β	ρ	**Q**	**Q0**	**Ag**	**It**	**C**
5827	36s	10.8427%	1	3	0.1	200000	0.5	40	30	1
5694	20m 8s	8.3127%	1	13	0.1	200000	0.1	200	70	4
5683	41m 23s	8.1035%	1	30	0.1	200000	0.1	200	140	4

Tabulka 7.9: DJA1436 – MyAlgo

Délka cesty	**Doba běhu**	**Bez souboru**	**Chybovost**	**SNN**	**MNN**	**LNN**	**SC**	**Segmenty**
5366	1m 41s	1m 35s	2.0734%	1	12	33	4	Ano
5453	10s	3s	3.7284%	5	12	44	1	Ne
5443	8s	1s 793ms	3.5381%	1	12	23	1	Ne

7.7 Testovací sada FQM5087

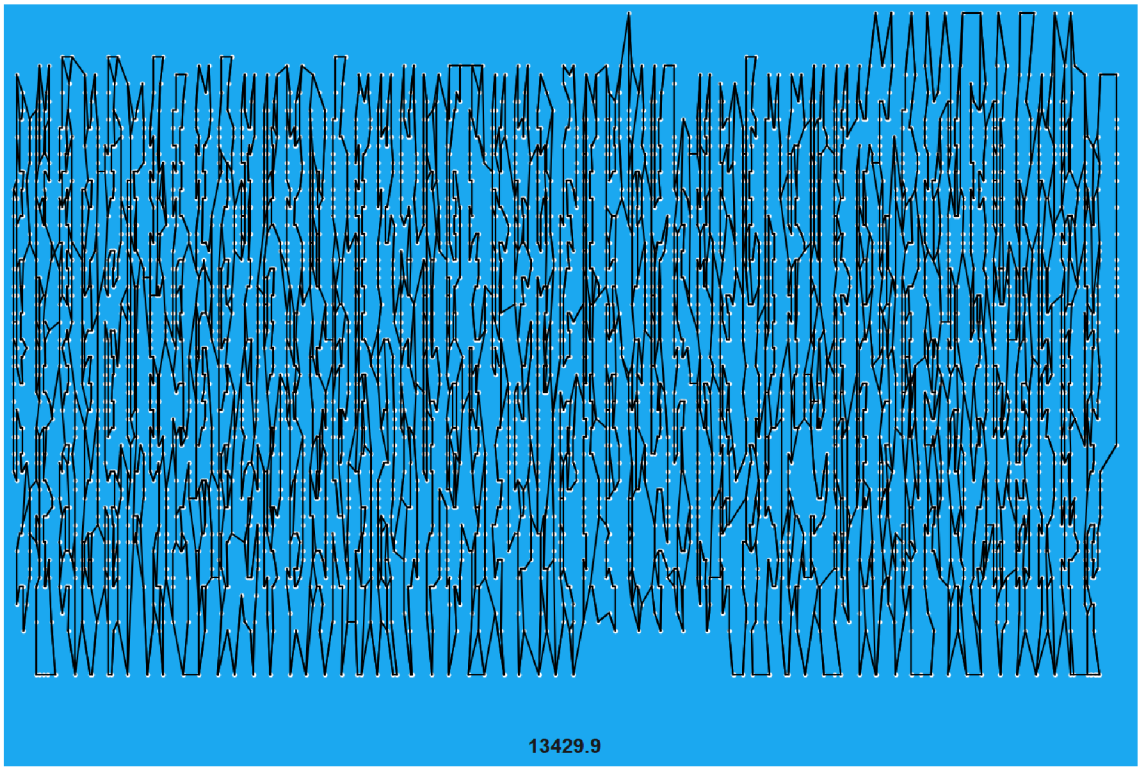
Je to uměle vygenerovaná sada s 5087 místy. Její optimální řešení má délku **13029**. Nejkratší cesta, kterou se mi povedlo za pomoci algoritmu najít má délku **13296**. Doba běhu programu trvá i s využitím segmentů maximálně do 10 minut. A nejrychlejší řešení se algoritmu povedlo najít za 35 sekund. Moje nejlepší nalezená cesta je vyobrazena na obrázku 7.2.

Tabulka 7.10: FQM5087 – Bez zaokrouhlení

Délka cesty	**Doba běhu**	**Bez souboru**	**Chybovost**	**SNN**	**MNN**	**LNN**	**SC**	**Segmenty**
13436.1	9m a 42s	8m a 27s	-	1	12	23	4	Ano
13626.4	3m a 1 s	1m a 5s	-	1	12	23	4	Ne
13627	1m a 50 s	35s	-	7	12	23	1	Ne
13429.9	7m a 42s	6m a 25s	-	5	12	40	1	Ano

Tabulka 7.11: FQM5087 – S zaokrouhlením

Délka cesty	**Doba běhu**	**Bez souboru**	**Chybovost**	**SNN**	**MNN**	**LNN**	**SC**	**Segmenty**
13296	7m 20s	6m 50s	2.0493%	1	12	33	4	Ano
13336	6m	5m 2s	2.3563%	4	12	23	1	Ano



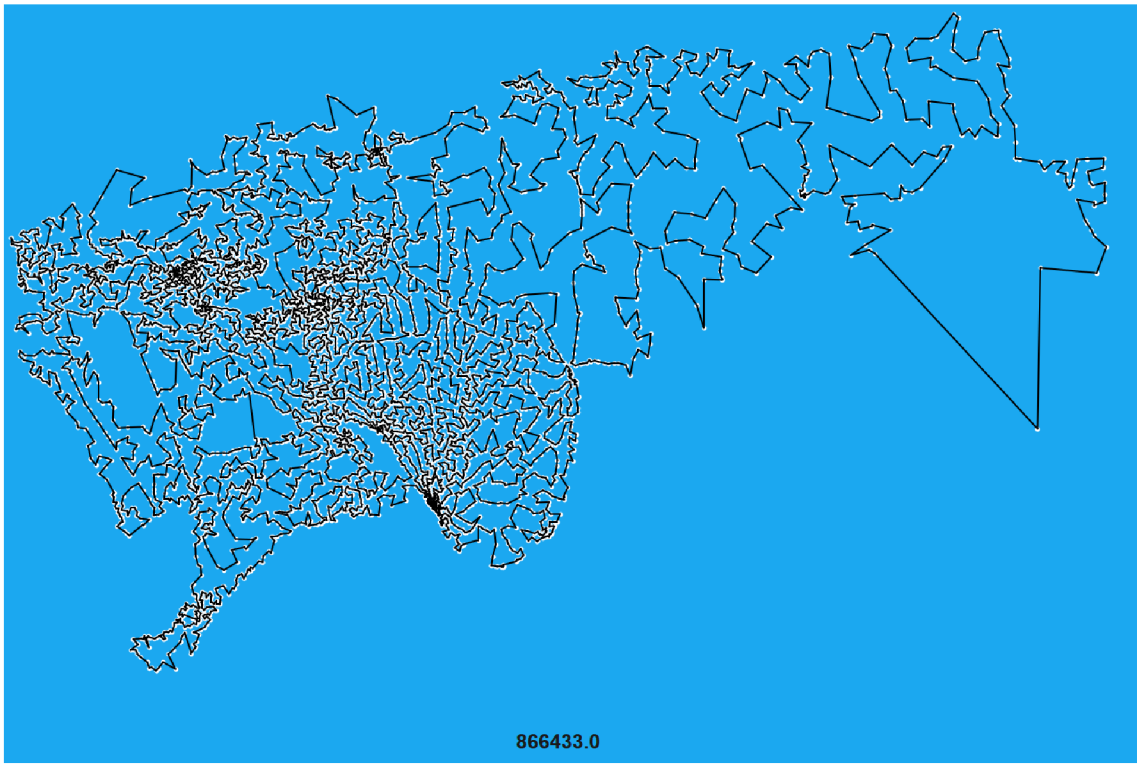
Obrázek 7.2: FQM5087

7.8 Testovací sada AR9152

Jedná se o města v státě Argentina jejich počet je 9152 a optimální cesta má délku **837479**. Mě se podařilo najít cestu dlouhou **866433**, která je vyobrazena na obrázku 7.3.

Tabulka 7.12: AR9152

Délka cesty	Doba běhu	Bez souboru	Chybovost	SNN	MNN	LNN	SC	Segmenty
866433	18m 42s	13m 20s	3.4573%	5	12	23	1	Ano
881675	10m 20s	4m 30s	5.2773%	5	12	23	1	Ne



Obrázek 7.3: AR9152

7.9 Testovací sada xmc10150

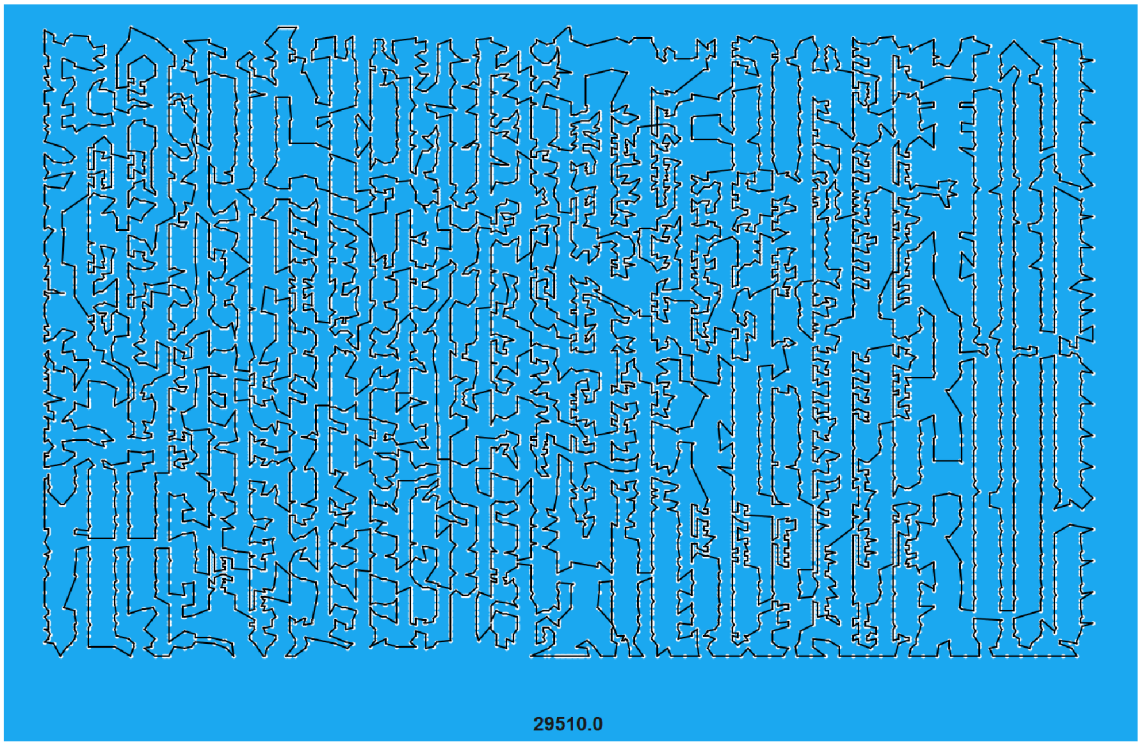
Jedná se o uměle vygenerovanou sadu s 10150 místy. Nejlepší nalezené řešení je s délkou **28387**, které nalezl hybrid genetic algorithm od **Hung Dinh Nguyen**. Toto řešení ovšem vycházelo z již vygenerovaného řešení s délkou 28388, které nalezl **LKH** [15]. Na obrázku 7.4 je zobrazeno nejlepší mnou nalezené řešení.

Tabulka 7.13: xmc10150 - Bez zaokrouhlováním

Délka cesty	Doba běhu	Bez souboru	Chybovost	SNN	MNN	LNN	SC	Segmenty
29994.6	42m 5s	35m 3s	-	1	12	33	4	Ano
30422.1	15m 6s	8m 24s	-	7	12	23	1	Ne
29959.8	38m 2s	21m 12s	-	1	12	40	1	Ano

Tabulka 7.14: xmc10150 - S zaokrouhlováním

Délka cesty	Doba běhu	Bez souboru	Chybovost	SNN	MNN	LNN	SC	Segmenty
29718	20m 1s	18m 20s	4.6888%	5	12	33	1	Ano
29510	19m	17m 30s	3.956%	1	12	23	4	Ano



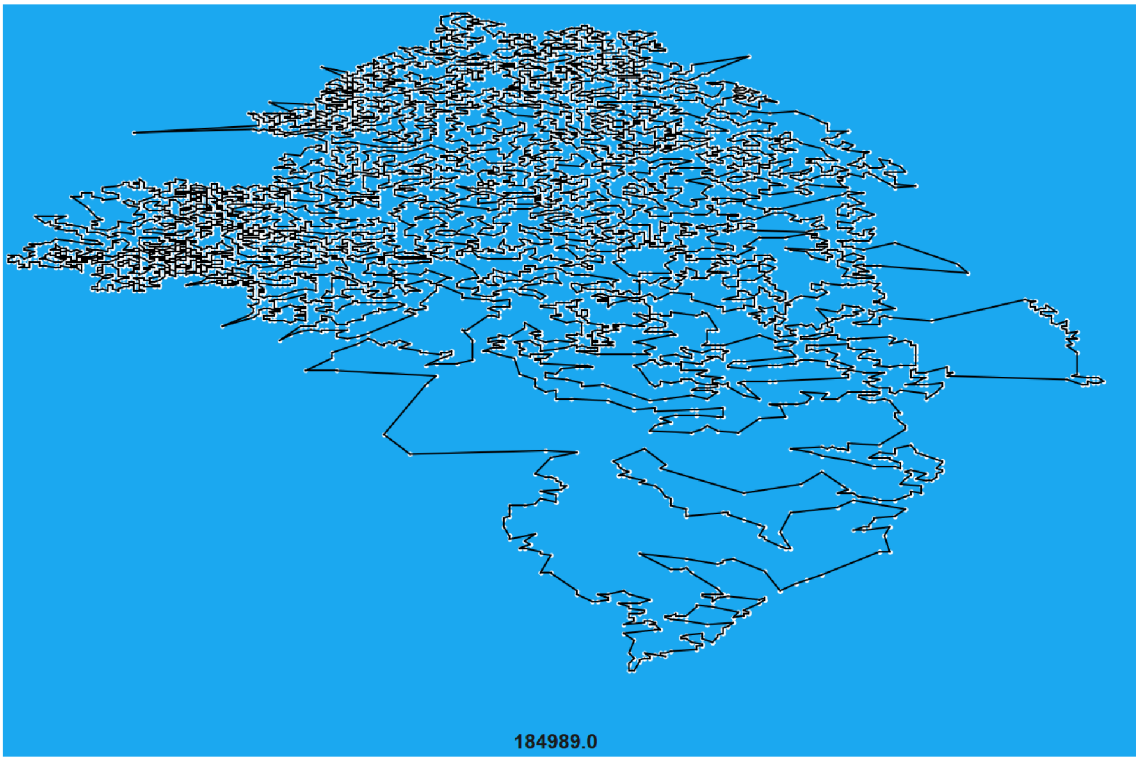
Obrázek 7.4: xmc10150

7.10 Testovací sada HO14473

Jedná se o sadu s místy v státě Honduras s 14473 místy. A nejlepší nalezenou cestou 177,092, která ovšem už vychází z předtím nalezených cest. Pro představu **LKH** běžel asi 381 hodin, aby nalezl cestu dlouhou **177,405** [17]. Oproti tomu MyAlgo za 16 minut nalezl řešení dlouhé 184989. A i s generováním 4 počátečních řešení mu netrvalo najít cestu dlouhou **183034** ani hodinu. Zde se začíná projevovat pomalé načítání míst do paměti, které by šlo zrychlit přidáním vláken. Povedlo se mi neuložit si soubor s řešením s délkou 183034, tak pouze přikládám řešení s délkou 184989 zde: 7.5.

Tabulka 7.15: HO14473

Délka cesty	Doba běhu	Bez souboru	Chybovost	SNN	MNN	LNN	SC	Segmenty
183034	1h 14m	50m 6s	3.3553%	1	12	33	4	Ano
184989	52m 13s	16m 12s	4.4593%	1	12	23	1	Ne



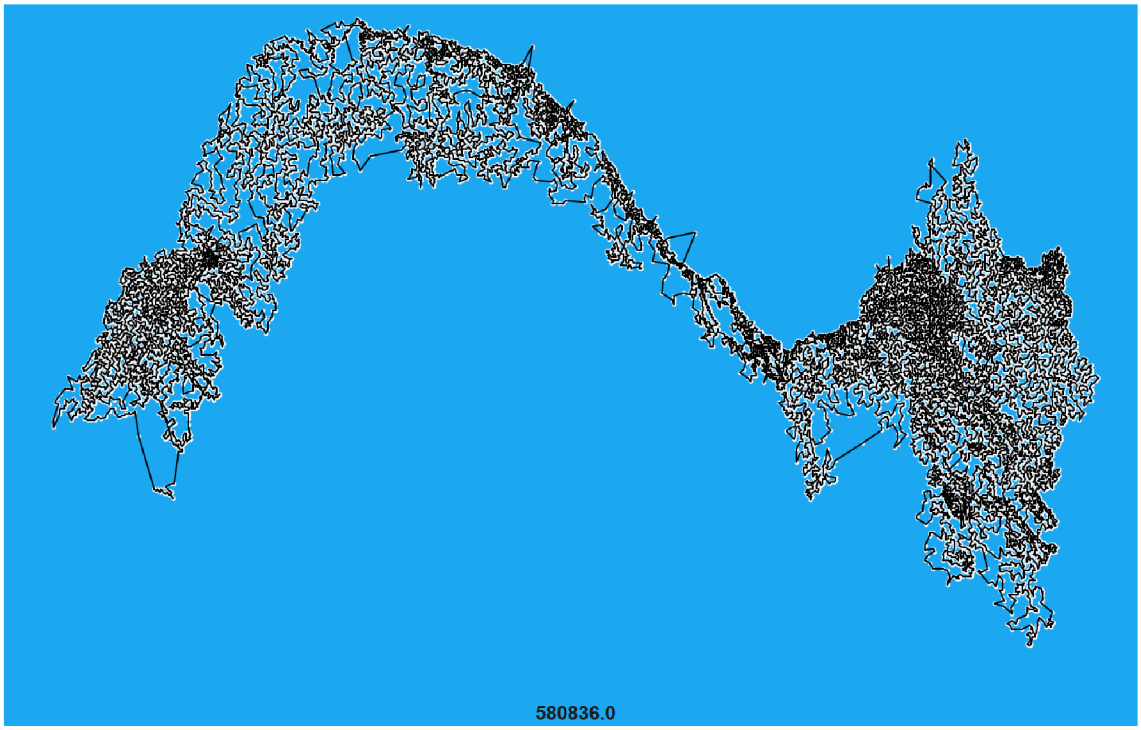
Obrázek 7.5: HO14473

7.11 Testovací sada VM22775

Zde se začínají projevovat hardwarové nedostatky a přesněji to, že mám k dispozici necelých 6GB operační paměti a velikost souboru s vzdálenostmi je skoro 8GB i když je rychlost algoritmu stále relativně rychlá. Program většinu času tráví v čtení a zápisu do souboru celkově 3 hodiny a 5 minut z celkové doby běhu 4 hodin a 52 minut. Je to nejspíše způsobeno tím, že operační systém musí vytvořit na pevném disku virtuální prostor pro RAM a s tím poté pracuje. Pro tuto sadu se mi povedlo najít relativně dobrý výsledek i přes to, že je zde 22775 míst. Zvýšil jsem velikost posledních sousedů a taky prvních oproti předchozím testováním. Pokud bych zde přidal víc segmentů domnívám se, že by program mohl najít řešení s chybovostí klidně okolo 1.5%, ale prodloužil by se čas řešení. Nejlepším nalezeným řešením pro tuto sadu je **569288** a mé výsledné řešení s délkou **580836** je zde 7.6

Tabulka 7.16: VM22775

Délka cesty	Doba běhu	Bez souboru	Chybovost	SNN	MNN	LNN	SC	Segmenty
580836	4h 52m	1h 47m	2.0285%	5	12	35	1	Ano
595972	4h 7m	1h 17m	4.6873%	1	12	23	1	Ne



Obrázek 7.6: VM22775

Kapitola 8

Návod k spuštění

Aplikace byla vyvíjena a testována na systému 64 bitovém Windows. Ale doporučuji pro spuštění použít například Ubuntu kvůli jednoduššímu spouštění binárních souborů, jelikož ty potřebují některé dll knihovny.

8.1 Požadavky

8.1.1 Python 3.8+

- Nainstalujte Python 3.8 nebo vyšší na svůj počítač. K dispozici jsou různé zdroje, včetně oficiálního webu Pythonu (https://www.python.org/downloads/).
- Ověřte, zda máte nainstalovanou správnou verzi Pythonu příkazem "python –version"v příkazovém řádku nebo terminálu. Pokud je nainstalován správně, měla by se vám zobrazit verze Pythonu.
- Zkontrolujte, zda máte přístup k Pythonu z jakéhokoli adresáře v příkazovém řádku nebo terminálu. Pokud ne musíte přidat cestu k Pythonu do systémové proměnné PATH.
- Celkově, aby program fungoval musí fungovat příkaz "python", který spustí mezi script.

8.1.2 GCC

- Ověřte, zda je g++ nainstalován na vašem počítači. To můžete provést spuštěním následujícího příkazu v příkazovém řádku nebo terminálu:

  ```
  g++ --version
  ```

  Pokud máte g++ nainstalován, zobrazí se vám informace o verzi.

- Pokud nemáte g++ nainstalován, budete si ho muset nainstalovat. Existuje několik způsobů, jak nainstalovat g++ v závislosti na vašem operačním systému. Například pro Ubuntu a další linuxové distribuce můžete použít následující příkaz v terminálu:

  ```
  sudo apt-get install g++
  ```

- Pokud jste na systému Windows program nepotřebuje překladač, stačí mu pouze, aby se ve složce System32 ve vašem počítači nacházeli potřebné dll soubory. Soubory, které se tam musí nacházet se nachází ve složce dll, k které se dostanete po rozbalení zipu. Popřípadě pokud toto nebude fungovat doporučuji nainstalovat například MinGW-w64, ale musí být k dispozici i knihovna s vlákny, které využívá ACO.

8.2 Windows

- Stáhněte a rozbalte zip, který se nachází v externích přílohách
- Otevřete složku TSPSolverWindows
- Spustě MainWindow.exe

8.3 Linux

- Stáhněte a rozbalte zip, který se nachází v externích přílohách
- Otevřete složku TSPSolverLinux
- Spustě MainWindow například v termínálu pomocí ./MainWindow

8.4 Ovládání programu

Jednotlivé funkce tlačítek jsou již popsány v sekci 6.2. Zmíním pouze to, že pro spuštění nastavení jiného algoritmu je nutné přepnout **Algorithm types** na ten který chceme nastavit a následně stisknout tlačítko **Algorithm settings**. Algoritmus následně spustíte pomocí tlačítka **Solve**, ale předtím musí být vygenerována místa. Například, pokud jsou místa vybrána náhodně, použijte tlačítko **Generate**. Aktuální trasu smažete stiskem tlačítka **Clear roads**. Pro použití segmentů v MyAlgo je potřeba mít minimálně 130 míst, protože jsou tam pevně stanovené velikosti. Při použití **roundingu** je nutné zaškrtnout tuto volbu před generováním míst, protože v té době se počítají vzdálenosti.

Kapitola 9

Závěr

Cílem mé práce bylo vytvořit alespoň dva algoritmy pro řešení problému obchodního cestujícího. Hlavními algoritmy, které jsem implementoval, jsou MyAlgo a ACO. Tyto algoritmy využívají pro optimalizaci nebo například u ACO k nastavení původních feromonů i jiné algoritmy, konkrétně se jedná o Nearest Neighbor algoritmus, 3-opt a 2-opt algoritmy. Dále jsem implementoval uživatelské rozhraní pro testování a vytváření míst. U ACO se zde nachází paralelní výpočty, jak v podobě jader, tak i vláken. I přes tuto skutečnost se mi podařilo navrhnout algoritmus, který je o mnoho lepší a nepoužívá ani jádra ani vlákna. To považuji za úspěch. Jádro mého algoritmu je relativně rychlé, jelikož vždy prvně počítám vzdálenost a pokud je lepší, až poté dělám změny v poli.

Celkově si myslím, že by se dala navrhnout lepší metoda na optimalizaci mého algoritmu v podobě prohledávání více typů řešení, protože vždy existuje možnost, že aktuální řešení pomocí mého algoritmu nelze zlepšit, i když lepší řešení existuje. Nebo se zde mohla použít víc k-opt metoda pro generování například prvotních sousedů. Jenže implementace je poměrně problematická.

Metoda ACO je efektivní tak do 60 míst a dává většinou optimální výsledky v poměrně krátkém čase. Pro více než 1000 míst dává už hodně špatné výsledky s chybovostí okolo 10 % a dobou řešení až 40 minut. Oproti tomu MyAlgo v žádné sadě nepřekonal chybovost 5 % a i pro 22775 míst byla chybovost pouze 2 %. Pokud srovnáme dobu běhu, ACO běží podstatně déle, proto jsem ji pro více než 1500 míst netestoval. Například k tomu, aby dosáhla chybovosti 8 % v sadě s 1436 místy, trvalo ACO 41 minut a 23 sekund. Oproti tomu MyAlgo našel řešení s chybovostí 3.583 % za pouhých 8 sekund a řešení s chybovostí 2 % za 1 minutu a 35 sekund. Je možné, že pro ACO existují lepší parametry než jsem nastavil, ale po experimentování jsem lepší nastavení nenašel. V porovnání s ostatními implementacemi ACO by chybovost odpovídala.

Literatura

[1] ARM. *Using common compiler options - Selecting optimization options* [online]. 2019 [cit. 2023-03-03]. Dostupné z: https://developer.arm.com/documentation/100748/0612/using-common-compiler-options/selecting-optimization-options.

[2] Davendra, D. *The Traveling Salesman Problem: Theory and Applications*. 1. vyd. InTech, 2010. ISBN 978-953-307-426-9. Dostupné z: http://www.exatas.ufpr.br/portal/docs_degraf/paulo/Traveling_Salesman_Problem__Theory_and_Applications.pdf.

[3] developers, N. *Numba 5 Minute Guide* [online]. 2020 [cit. 2023-03-03]. Dostupné z: https://numba.readthedocs.io/en/stable/user/5minguide.html.

[4] Dorigo, M. a Stützle, T. *Ant Colony¨Optimization*. 1. vyd. Massachusetts Institute of Technology, 2004. ISBN 0-262-04219-3.

[5] Geopy. *Distance* [GitHub repository]. 2022 [cit. 2023-03-16]. Dostupné z: https://github.com/geopy/geopy/blob/master/geopy/distance.py.

[6] Helsgaun, K. An effective implementation of the Lin-Kernighan traveling salesman heuristic. *European Journal of Operational Research*. 1. vyd. Elsevier. 2000, sv. 126, č. 1, s. 106–130.

[7] Helsgaun, K. An effective implementation of K-opt moves for the Lin-Kernighan TSP heuristic. *European Journal of Operational Research*. 1. vyd. Duben 2023, č. 1.

[8] Li, P. a Zhu, H. Parameter Selection for Ant Colony Algorithm Based on Bacterial Foraging Algorithm. *Mathematical Problems in Engineering*. 1. vyd. Hindawi Publishing Corporation. Dec 2016, sv. 2016, č. 1, s. 6469721. DOI: 10.1155/2016/6469721. ISSN 1024-123X. Dostupné z: https://doi.org/10.1155/2016/6469721.

[9] Lin, S. a Kernighan, B. W. An Effective Heuristic Algorithm for the Traveling-Salesman Problem. *Operations Research*. 1. vyd. 1973, sv. 21, č. 2, s. 498–516. DOI: 10.1287/opre.21.2.498.

[10] MathWorks. *What Is the Genetic Algorithm?* [online]. 2023 [cit. 2023-04-09]. Dostupné z: https://www.mathworks.com/help/gads/what-is-the-genetic-algorithm.html.

[11] Pascal Fua, K. L. *Comparing Python, Go, and C++ on the N-Queens Problem* [online]. 2020 [cit. 2023-03-03]. Dostupné z: https://arxiv.org/pdf/2001.02491.pdf.

[12] Python, R. *Python Global Interpreter Lock (GIL): What It Is and How It Works.* 2018 [cit. 2023-03-02]. Dostupné z: https://realpython.com/python-gil/.

[13] Schimansky, T. *Map_with_customtkinter* [GitHub repository]. 2021 [cit. 2023-03-16]. Dostupné z: https://github.com/TomSchimansky/TkinterMapView/blob/main/examples/map_with_customtkinter.py.

[14] Scripts, M. T. *Vincenty formulae for distance between two Latitude/Longitude points* [online]. 2022 [cit. 2023-03-16]. Dostupné z: http://www.movable-type.co.uk/scripts/latlong-vincenty.html.

[15] University of Waterloo. *XMC10150 - Computation Log* [online]. 2017 [cit. 2023-04-18]. Dostupné z: https://www.math.uwaterloo.ca/tsp/vlsi/xmc10150.log.html.

[16] user25658. *Re: How to normalize data to 0-1 range?* [online]. 2013 [cit. 2023-03-16]. Dostupné z: https://stats.stackexchange.com/questions/70801/how-to-normalize-data-to-0-1-range/70810#70810.

[17] Waterloo, T. U. of. *HO14473 - 14473 point benchmark instance for the geometric traveling salesman problem* [online]. 2006. Dostupné z: https://www.math.uwaterloo.ca/tsp/vlsi/xmc10150.log.html.

[18] ZBOŘIL, F. a ZBOŘIL jr., F. *Základy umělé inteligence.* FIT VUT v Brně, 2012. Skripta do předmětu IZU.