

Zadanie 2

Traveling Salesman Problem

Umelá inteligencia

Frederik Duvač
3. semester
7.10. 2023

Obsah

1. Teoretické východiská	2
1.1. Technické pozadie	2
1.2. Zadanie	2
1.3. Zakázané prehľadávanie (Tabu search)	2
1.4. Simulované žíhanie (Simulated annealing)	2
2. Implementácia	3
2.1. Reprezentácia údajov	3
2.2. Zakázané prehľadávanie	4
2.3. Simulované žíhanie	5
2.4. Používateľské rozhranie	6
3. Testovanie	8
4. Zhodnotenie	10

1. Teoretické východiská

1.1. Technické pozadie

Hardvér, na ktorom bol skompilovaný zdrojový kód a testovaný program, je notebook Lenovo Legion S7, 12. generácia Intel® Core™ i5-12500H, 16 GB RAM. Celý projekt je implementovaný v programovacom jazyku Python 3.

1.2. Zadanie

Obchodný cestujúci má za úlohu navštíviť daný počet miest (napr. 20 - 40) reprezentovaných súradnicami na mape. Jeho cieľom je minimalizovať celkové cestovné náklady, pričom cena prepravy medzi mestami sa určuje podľa Euklidovej vzdialenosti. Ide o uzavretú trasu, kde musí navštíviť každé mesto práve raz a vrátiť sa do východzieho mesta. Cieľom úlohy je nájsť optimálne poradie navštívenia miest (permutáciu), ktoré minimalizuje celkovú dĺžku cesty. Riešenie je potrebné nájsť pomocou algoritmov Zakázaného prehľadávania a Simulovaného žihania.

1.3. Zakázané prehľadávanie (Tabu search)

Algoritmus Zakázané prehľadávanie, často nazývaný Tabu Search, patrí do skupiny algoritmov, ktoré využívajú na hľadanie riešenia v priestore možných stavov lokálne vylepšovanie (optimalizáciu).

Postupnosť algoritmu je nasledovná:

1. **Inicializácia** - Algoritmus začína s počiatočnou permutáciou, ktorá je náhodne vygenerovaná.
2. **Nájdenie susedov** - Tabu Search generuje susedné riešenia aktuálneho stavu tým, že vykonáva rôzne operácie, ako sú výmeny, inverzie alebo iné transformácie v rámci problému.
3. **Výber najlepšieho suseda** - Z vygenerovaného susedstva riešení je vybrané to, ktoré má najlepšiu cenu cesty.
4. **Zakázané riešenia (Tabu List)** - Algoritmus udržiava zoznam zakázaných riešení, ktoré nemôžu byť znovu vybrané po určitú dobu. Čo pomáha zabrániť uviaznutiu v lokálnom extréme. Riešenia sú do zoznamu zakázaných pridávané a odstraňované po dosiahnutí stanovenej veľkosti tabu listu.
5. **Kritériá ukončenia** - Algoritmus Tabu Search má určité stanovené kritériá na ukončenie, napríklad maximálny počet iterácií alebo maximálny počet neoptimalizovanej trasy. Po ich splnení algoritmus končí a vracia nájdene riešenie.

1.4. Simulované žihanie (Simulated annealing)

Simulované žihanie je algoritmus inšpirovaný procesom ochladzovania kovu. Používa sa taktiež na hľadanie riešenia v priestore možných stavov pomocou lokálneho vylepšovania. Simulované žihanie a jeho schopnosť vyhýbať sa uväzneniu v miestnych optimách ho robí silným nástrojom na nájdene globálneho optima.

Kroky algoritmu sú nasledovné:

1. **Inicializácia teploty** - Začína na vysokej teplote, čo znamená, že je vyššia pravdepodobnosť na presun do stavu, ktorý má horšie ohodnotenie riešenia.
2. **Generovanie susedného riešenia** - Z aktuálneho riešenia sa vytvorí susedné riešenie prostredníctvom rôznych operácií, ako sú výmeny, inverzie, transformácie atď.
3. **Výpočet rozdielu ceny** - Vypočíta sa rozdiel medzi hodnotou aktuálneho riešenia a hodnotou susedného riešenia.
4. **Výber suseda** - Rozhoduje sa, či sa nové riešenie prijme alebo odmietne. Horšie riešenia môžu byť prijaté s určitou pravdepodobnosťou, ktorá klesá so zostupujúcou teplotou.
5. **Ochladzovanie**: Teplota postupne klesá podľa stanovenej hodnoty. Klesajúca teplota znamená, že algoritmus bude menej pravdepodobne prijímať horšie riešenia a tým sa bude približovať optimálnemu riešeniu.
6. **Kritériá ukončenia**: Algoritmus pokračuje v iteráciách, kým nie sú splnené stanovené kritériá ukončenia, akými môžu byť dosiahnutie maximálneho počtu iterácií alebo dosiahnutie stanoveného cieľového stavu.

2. Implementácia

2.1. Reprezentácia údajov

Celý problém obchodného cestujúceho je rozdelený do viacerých častí. Trieda World predstavuje svet s náhodne vygenerovanými mestami, po spustení programu. Ďalej táto trieda obsahuje ďalšie funkcionality, ktorými sú počítanie vzdialenosti medzi jednotlivými mestami, výpočet celkovej vzdialenosti cesty a algoritmus najbližšieho suseda.

```
class World:
    places: list
    distance_matrix: list[list]
    nearest_neighbour_permutation: list

    def __init__(self, amount):
        self.places = self.generate_places(int(amount))
        self.distance_matrix = self.fill_distance_matrix()
        self.nearest_neighbour_permutation =
self.get_nearest_neighbour()
```

Po vygenerovaní náhodných miest, ktoré sú reprezentované ako python dictionary obsahujúci informácie uid, lat a long, sa vypočíta matica vzdialeností týchto miest. Následne prebehne algoritmus najbližšieho suseda, ktorý slúži ako pomôcka pri výpočte fitness funkcie potrebnej pre oba algoritmy.

Funkcia main() primárne zabezpečuje konzolové ale aj grafické používateľské rozhranie. V konzolovom rozhraní používateľ nastaví počet miest a algoritmus na vygenerovanie optimalnej trasy. V grafickom rozhraní sa zobrazí iba výsledok.

Dôležitou súčasťou celého projektu sú oba algoritmy implementované v prislúchajúcich triedach TabuSearch a SimulatedAnnealing, kde každá trieda obsahuje potrebné funkcionality či už na generovanie susedných stavov, fitness funkciu a samotný algoritmus.

2.2. Zakázané prehľadávanie

Algoritmus je reprezentovaný ako samostatná trieda TabuSearch.

```
class TabuSearch:
    world: World
    final_permutation: list
    nearest_neighbour_value: float
    max_iterations = 1000
    max_tabu_list_size = 50
    max_none_improve = 100

    def __init__(self, world):
        self.world = world
        self.nearest_neighbour_value = world.get_permutation_value(
            world.nearest_neighbour_permutation)
        self.final_permutation = self.algo(world.places)

    def algo(self, permutation):
        ...
    def fitness(self, permutation):
        ...
    def get_neighborhood(self, permutation):
        ...
    @staticmethod
    def generate_neighborhood(permutation):
        ...
    @staticmethod
    def two_opt_neighborhood(permutation):
        ...
```

Po zavolaní algoritmu sa inicializujú maximálne hodnoty počtu iterácií, veľkosť tabu listu a počet nevylepšených riešení, čo znamená, po koľkých iteráciách má program skončiť v prípade, že sa už optimálna trasa viac nevylepšila.

Funkcia `algo()` rieši samotný algoritmus, ktorého kroky sú podrobnejšie popísané v kapitole [1.3](#).

Fitness funkcia vracia ohodnotenie ako veľmi je zvolené poradie optimálne v pomere celkovej ceny cesty pre nearest neighbour algoritmus, ktorý bol zvolený ako benchmark, aj keď tento algoritmus nie vždy vráti najoptimálnejšiu cestu a celkovej ceny cesty pre aktuálnu cestu.

Dôležitou časťou je generovanie susedstiev, a preto algoritmus pre ešte väčšie optimalizovanie výsledku využíva na generovanie susedstva dve rôzne funkcie generovanie susedstva a generovanie susedstva pomocou 2-opt heuristiky, medzi ktorými si program náhodne vyberie zavolaním funkcie `get_neighborhood()`.

2.3. Simulované žíhanie

Algoritmus je reprezentovaný ako samostatná trieda `SimulatedAnnealing`.

```
class SimulatedAnnealing:
    world: World
    final_permutation: list
    nearest_neighbour_value: float
    max_none_improve: int
    alpha = 0.999
    temperature = 1

    def __init__(self, world):
        self.world = world
        self.nearest_neighbour_value = world.get_permutation_value(
world.nearest_neighbour_permutation)
        self.max_none_improve = len(world.places) *
int(10 * math.pi)
        self.final_permutation = self.algo(world.places)

    def algo(self, permutation):
        ...
    def fitness(self, permutation):
        ...
    def get_neighbor(self, permutation):
        ...
    @staticmethod
    def swap(permutation):
        ...
    @staticmethod
    def insertion(permutation):
        ...
```

```

@staticmethod
def reordering(permutation):
    ...
@staticmethod
def reverse(permutation):
    ...
@staticmethod
def transport(permutation):
    ...

```

Taktiež ako v prípade Zakázaného prehľadávania sa po zavolaní Simulovaného žihania musia inicializovať dôležité hodnoty ako teplota, alfa čo reprezentuje hodnotu ako sa bude teplota zmenšovať a opäť maximálny počet nevylepšených riešení, kde sa počas testovania stanovil inicializačný vzorec pomocou konštanty π .

Funkcia `algo()` rieši samotný algoritmus, ktorého kroky sú podrobnejšie popísané v kapitole [1.4](#).


Rovnako ako pri Tabu search fitness funkcia vracia pomer celkovej ceny cesty pre nearest neighbour algoritmus a pre aktuálnu cestu.

Oproti Zakázanému prehľadávaniu, kde nám išlo o nájdenie najlepšieho spomedzi susedov pri Simulovanom žihaní je dôležitou časťou generovanie náhodného suseda a následne s použitím pravdepodobnosti a teploty si tohoto suseda môžeme vybrať po splnení podmienky aj v prípade ak má horšiu cenu.

Pre optimalizovanie výsledku, algoritmus využíva na generovanie susedov päť rôznych funkcií generovania náhodného suseda, ktorými sú výmena dvoch miest, výber a vloženie náhodného miesta k inému miestu, náhodné zamena poradia určitej podskupiny, opačné poradie podskupiny a premiestnenie podskupiny na inú pozíciu. Medzi týmito funkciami si program náhodne vyberie zavolaním funkcie `get_neighbor()`.

2.4. Používateľské rozhranie

Pre jednoduché nastavenie počtu miest a výber algoritmu bolo implementované konzolové rozhranie, kde používateľ zadá počet náhodných miest, ktoré je potrebné vygenerovať. Pre pohodlnejšiu prácu a porovnávanie oboch algoritmov bola implementovaná možnosť seedu pre vygenerovanie rovnakého sveta. Ďalej máme na výber algoritmus pre riešenie Problému obchodného cestujúceho. Počas behu algoritmu sme informovaný o aktuálnom stave, zobrazuje sa počet aktuálnych iterácií a dĺžka cesty (prípadne teplota). Na záver je vypísaná výsledná dĺžka optimálnej trasy.

Vizuálne riešenie celej trasy spolu s mestami je zobrazené po skončení programu a otvorené v novom okne Travelling Salesman Problem .

```

Amount of places 🌐 >> 30
Input world seed 🌱 >> pekngsvet123
🌐 World has been successfully created with 30 places 🌐

Choose algorithm for Travelling Salesman Problem 🚶
Both -> 0 | Tabu Search -> 1 | Simulated Annealing -> 2 >> 2

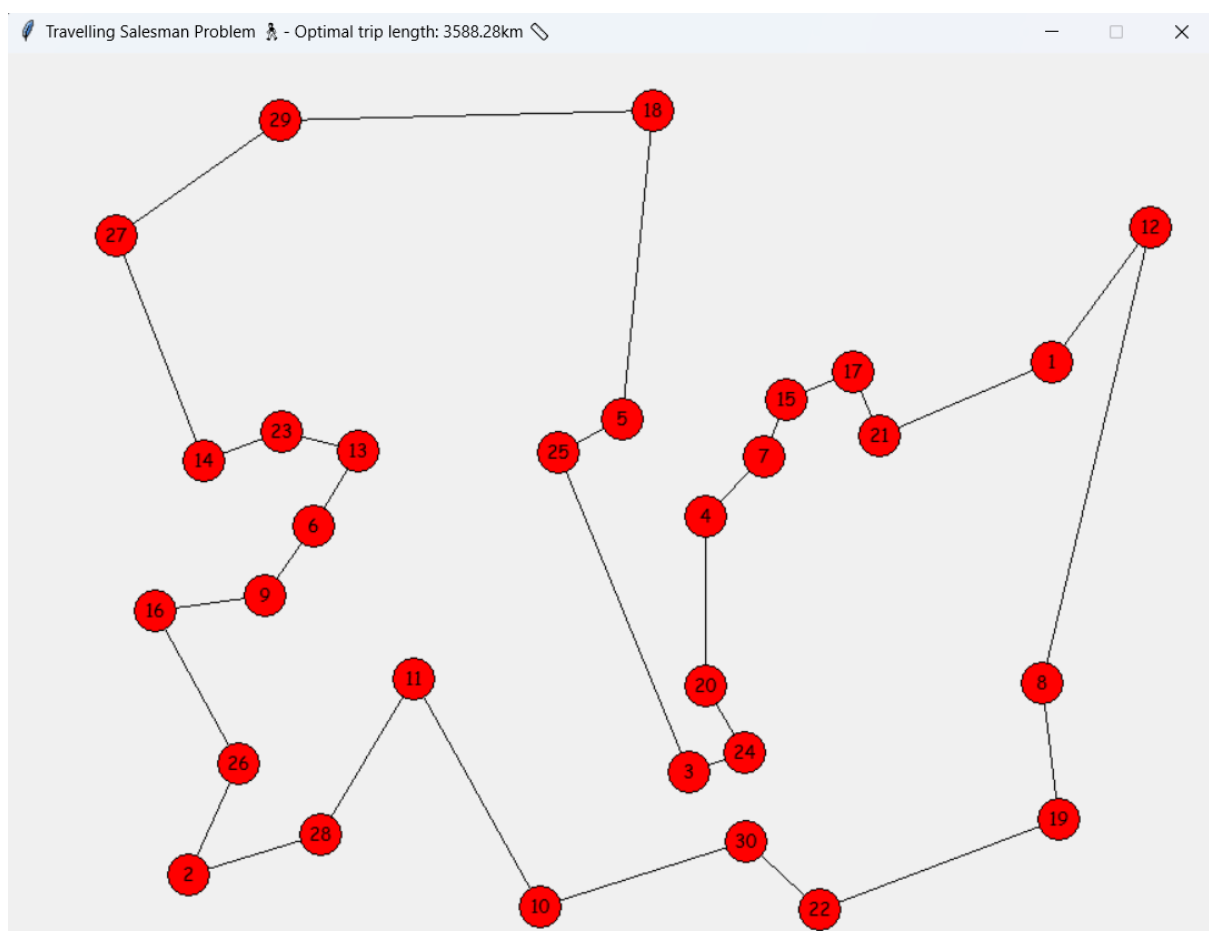
Iteration: 17348 🌱 Trip length: 3588.28km 📏 Temperature: 2.9008432113938494e-08 🌡️

Optimal trip length: 3588.28km 📏
Execution time: 904.00 ms ⚡

📄 INFO: RESULT IS DISPLAYED IN WINDOW
  - Travelling Salesman Problem 🚶

```

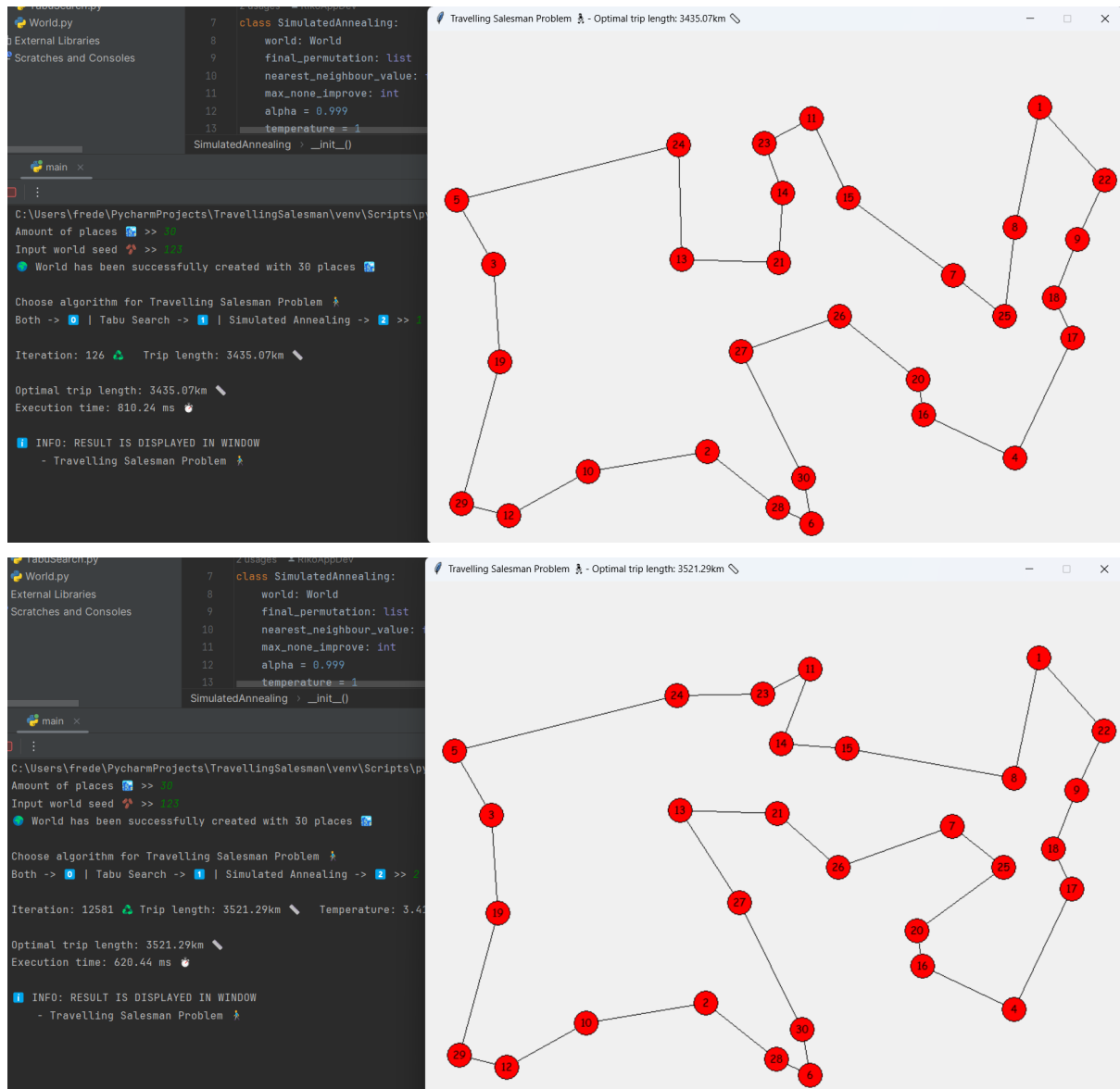
Obr. 1. Příklad konzolového výstupu



Obr. 2. Příklad možného výsledku GUI

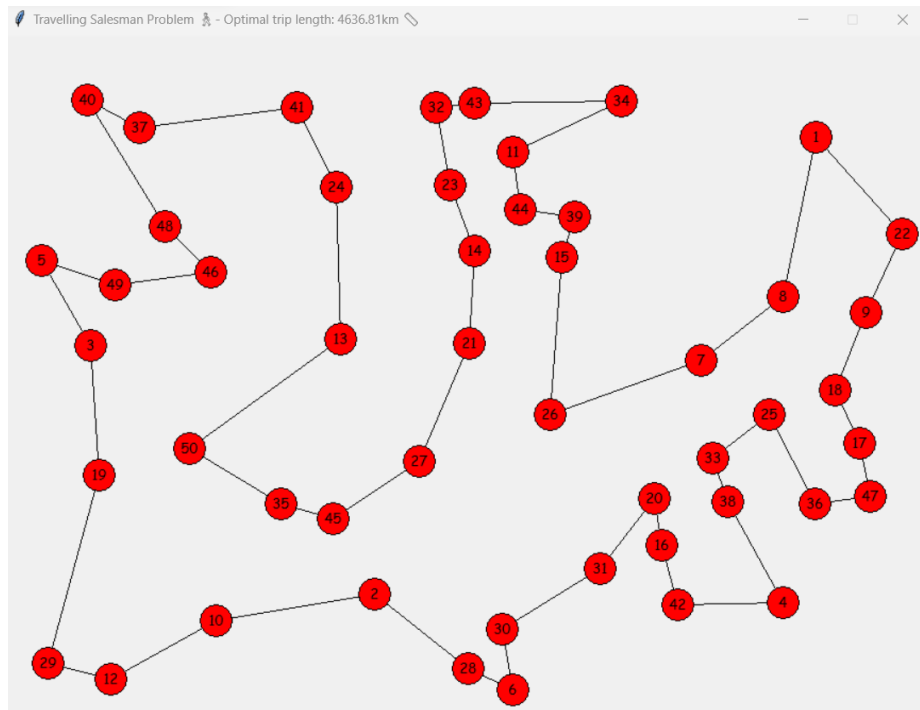
3. Testovanie

Algoritmy boli porovnávané medzi sebou pre rôzne veľkosti miest a rozne seedy. Počas testovania bolo zistené, že simulované žihanie sa rýchlejšie dostane k optimálnemu výsledku, avšak nebolo zistené, ktorý algoritmus poskytuje výslednú hodnotu lepšiu. Oba algoritmy sú závislé v určitých situáciách na náhode a preto sa nedá určiť ktorý je najoptimálnejší.

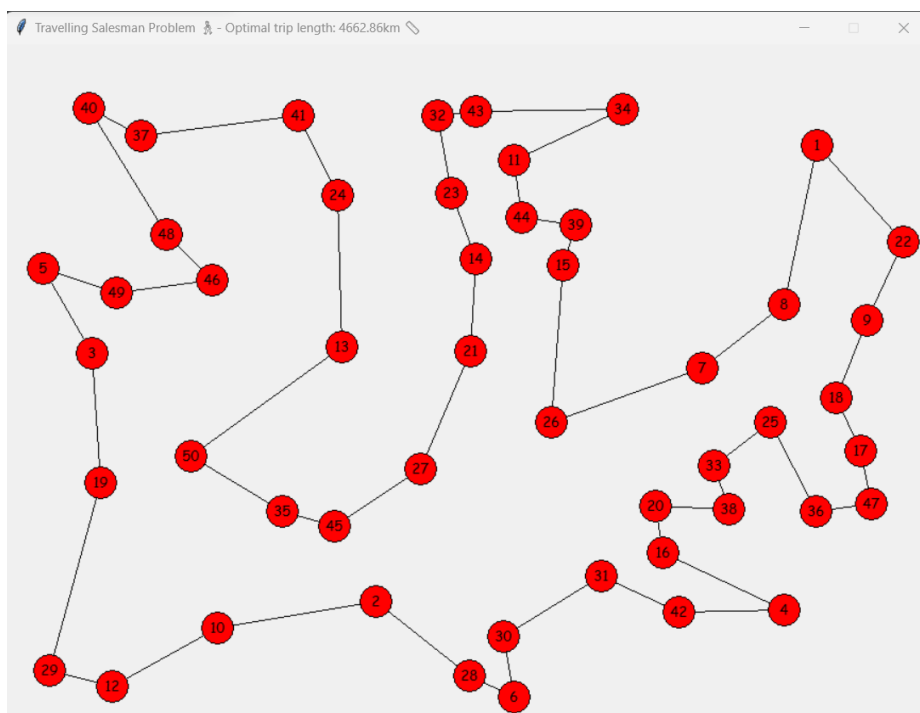


Obr. 3. Príklad porovnania algoritmov

Pre algoritmus Tabu Search bola hľadaná optimálna hodnota veľkosti tabu listu. Nasledujúce obrázky zobrazujú výsledok pre veľkosť tabu listu 5, ktorý iba v tomto prípade našiel o kúsok lepšiu trasu inak sa trasa nemenila pri iných veľkostiach tabu listu a pri iných testovaných vygenerovaných svetoch.



Obr. 4. Výsledok Tabu Search - tabu list veľkosť = 5



Obr. 5. Výsledok Tabu Search - tabu list veľkosť = 50, 10, 30, 100

Pri algoritme Simulated Annealing bolo testovaných viacero teplôt pre $\alpha = 0.999$ dosiahnuté výsledky boli odlišné ale neboli priamo úmerné so zmenou teploty.

Najrýchlejšie sa dostal k optimálnemu výsledku pri teplote 1 a preto bola zvolená.

Počet miest:	Seed:	Teplota:	Dĺžka trasy:
100	123	1	6891.61km
100	123	10	6968.53km
100	123	100	6851.79km
100	123	1000	7089.22km

4. Zhodnotenie

Riešenie Problému obchodného cestujúceho (Traveling Salesman Problem) pomocou zakázaného prehľadávania a simulovaného žihania poskytuje efektívne riešenie tohto známeho problému. Jeho úspešnosť a rýchlosť závisia od správneho nastavenia a ladenia parametrov týchto algoritmov.

Na záver projektu môžeme konštatovať:

- Simulované žihanie (Simulated Annealing) a zakázané prehľadávanie (Tabu Search) sú účinné metódy na riešenie náročných optimalizačných problémov, ako je TSP.
- Zakázané prehľadávanie pridáva do procesu obmedzenia vo výbere susedných riešení a udržiava históriu zakázaných ťahov. To pomáha algoritmu preskúmať rôzne časti stavového priestoru a prekonať lokálne optimá.
- Simulované žihanie umožňuje algoritmu preskúmať rôzne časti priestoru riešení a vyhnúť sa uviaznutiu v lokálnom minime. Postupné zníženie teploty umožňuje postupné zlepšovanie kvality nájdeného riešenia.