SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE

Fakulta informatiky a informačných technológii

Umelá Inteligencia

Zadanie č.3:

Problém obchodného cestujúceho - Tabu search a simulované žíhanie

Tomáš Socha

ID: 110896

Akademický rok 2021/2022

1) Zadanie

Je daných aspoň 20 miest a každé má určené súradnice ako celé čísla X a Y. Tieto súradnice sú náhodne vygenerované. Cena cesty medzi dvoma mestami zodpovedá Euklidovej vzdialenosti – vypočíta sa pomocou Pytagorovej vety. Celková dĺžka trasy je daná nejakou permutáciou (poradím) miest, pričom každé je navštívené práve raz. Cieľom je nájsť takú permutáciu (poradie), ktorá bude mať celkovú vzdialenosť čo najmenšiu.

Mojím cieľom bolo aplikovať a porovnať algoritmy tabu search a simulovaného žíhania(simulated annealing) na problém obchodného cestujúceho(traveling salesman problem), a nastaviť vstupné parametre tak, aby našli čo najmenšiu celkovú vzdialenosť.

Príklad trasy medzi náhodne zvolenými mestami

Príklad rozloženia miest (20 miest):

```
(60, 200), (180, 200), (100, 180), (140, 180), (20, 160), (80, 160), (200, 160), (140, 140), (40, 120), (120, 120),
(180, 100), (60, 80), (100, 80), (180, 60), (20, 40), (100, 40), (200, 40), (20, 20), (60, 20), (160, 20)
```

[1]

Obrázok 1

[1] http://www2.fiit.stuba.sk/~kapustik/obchodny_cestujuci.html

2) Implementácia

Vo vlastnej implementácii som na vyriešenie problému obchodného cestujúceho použil postupne hill climbing algoritmus, tabu search a simulované žíhanie, a primárne sledoval veľkosti nájdených ciest a čas potrebný na nájdenie riešenia. Na jej realizáciu bol použitý jazyk Python vo verzii 3.9, hlavne kvôli jednoduchšej práce s ním oproti jazyku C a taktiež množstvu vstavaných funkcii, ktoré jazyk C nepodporuje.

a) Reprezentácia nájdeného riešenia

Na reprezentáciu aktuálneho nájdeného riešenia, najlepšieho v okolí a najlepšieho celkovo som vytvoril triedu *Vektor*, ktorá obsahuje 3 atribúty:

- **stav**- čiže poradie navštívenia jednotlivých miest
- **vzdialenosť**- celková cena cesty pri navštívení všetkých miest
- **iterácia** – uloženie čísla iterácie, v ktorej sa našlo najlepšie riešenie

```
class Vektor:
    stav = None
    vzdialenost = None
    iteracia = None
```

Stav bol reprezentovaný ako jeden list obsahujúci listy, každý reprezentujúci x a y – ovú súradnicu mesta. **Veľkosť mapy** som nastavil na pevný rozmer 200x200km. **Fitness** funkcia bola v tomto prípade prevrátená hodnota vzdialenosti, a tak som vo vlastnej implementácii pracoval len s premennou vzdialenosť a nie fitnes, kvôli priamočiarejšiemu interpretovaniu.

b) Globálne premenné

Taktiež som použil **globálne premenné** slúžiace ako parametre pre algoritmy, ktoré je možné meniť v prípade testovania. Spoločné parametre boli počet miest a iterácii a veľkosť vygenerovaného okolia. Pri tabu search išlo o veľkosť tabu listu a pri simulovanom žíhaní o teplotu a faktor ochladzovania(cooling factor).

c) Tabu search

Ide o hill-climbing algoritmus s lokálným vylepšením, ktorým je zavedenie tabu listu. Princíp fungovania je relatívne jednoduchý. Z aktuálneho stavu je vytvorený určitý počet nasledovníkov vymenením ľubovoľnej dvojice miest.(nazývaných okolie). Z okolia sa následne vyberie stav s najlepším hodnotením a presunie sa doň. Zároveň uloží aktuálny stav do tabu listu. Takže pri ďalšom generovaní okolia už nebude možné prejsť do predchádzajúceho stavu, čo zabezpečí vyhnutiu sa zacyklenia alebo zaseknutia v lokálnom extréme. Taktiež umožní akceptovať aj stav s horším ohodnotením.
Na uchovanie hodnoty lokálneho maxima slúži inštancia triedy typu *Vektor* s názvom *global_best.* Po Zaplnení tabu listu na maximálnu zvolenú veľkosť sa zahodí najstarší stav, a vloží sa nový.

d) Simulované žíhanie

Ide o hill-climbing algoritmus s iným lokálným vylepšením ako tabu search. Z aktuálneho stavu je vytvorený určitý počet nasledovníkov z ktorých si vyberie jedného náhodného. Ak má zvolený nasledovník lepšie ohodnotenie, tak doň prejde na 100%. V prípade, že má nasledovník horšie ohodnotenie, tak doň prejde len s pravdepodobnosťou menšou ako 100%. V prípade odmietnutia skúša ďalšieho ľubovoľne vybraného nasledovníka. Pravdepodobnosť je pri horších ohodnoteniach spočiatku relatívne vysoká a postupne klesá k nule.

3) Opis riešenia

Chod programu je zabezpečený vďaka funkcii *search_algorithm()*, ktorá sa spúšta v *main* funkcii. Najskôr sa vygenerujú pozície jednotlivých miest na mape, a ich počiatočné poradie pomocou funkcie *generator().*

Následne funkcia obsahuje cyklus na zabezpečenie vykonania určitého prednastaveného počtu iterácii. V cykle sa na základe argumentu *typ_algoritmu* zavolá príslušná funkcia zodpovedajúca danému algoritmu.

Funkcia *hill_climbing()* pozostáva z jednoduchého while cyklu, ktorý zabezpečí vygenerovanie rôznych potomkov. Postupne sú ukladaný do dict *okolie,* ktorý sa po skončení cyklu zoradí a vyberie sa z neho najlepší potomok. Ak je potomok lepší ako jeho predchodca, tak sa prejde do toho stavu.

Funkcia *tabu_search()* funguje na rovnakom princípe ako funkcia *hill_climbing()*, no ešte pred vkladaním do dict sa skontroluje, či sa daný potomok už nenachádza v tabu liste. Po skončení cyklu sa potom najlepší potomok vloží do tabu listu. Ak už je tabu_list plný, tak sa najskôr vyberie najstarší prvok, a až potom vloží nový. Taktiež sa v prípade nájdenia zatiaľ najlepšieho riešenia upraví hodnota premennej global_best.

Funkcia *simulated_annealing()* taktiež používa while cyklus na vygenerovanie okolia potrebnej veľkosti. Následne obsahuje ďalší cyklus, ktorý zabezpečí náhodné vyberanie potomkov z okolia, pokiaľ nebude nejaký akceptovaný alebo kým sa nevyprázdni list. Ak je ohodnotenie potomka lepšie, tak bude na 100% akceptovaný. Ak je horšie, tak bude akceptovaný, ak je jeho pravdepodobnosť akceptovania >= ako náhodná pravdepodobnosť od 0 po 1. Pravdepodobnosť je počítaná pomocou vzorca[2] P = $e^{-\frac{\Delta E}{k*t}}$. ΔE je absolútna hodnota rozdielu vzdialenosti aktuálneho stavu a potomka. t je teplota a k je konštanta pomocou ktorej sa bude t znižovať pri každej iterácii. Teplota je znižovaná geometricky, čiže pomocou vynásobenia konštanty menšej ako 1.

Po vykonaní všetkých iterácii sa vypíše začiatočná a koncová cesta a ich vzdialenosti a vykreslia sa grafy zobrazujúce vývoj hľadania optimálneho riešenia a cesty. Taktiež sa vypíše celkový zmeraný čas vykonania jednotlivých hľadaní.

Program taktiež obsahuje funkciu na výpočet celkovej vzdialenosti pre daný stav pomocou pytagorovej vety. Tiež sa v ňom nachádza aj funkcia *novy_stav_gen()*, ktorá vygeneruje nový stav vymenením ľubovoľných 2 miest medzi sebou.

Na vykreslenie grafu a cesty po vykonaní jednotlivých algoritmov slúžia funkcie s príslušnými názvami.

[2] https://towardsdatascience.com/optimization-techniques-simulated-annealing-d6a4785a1de7

4) Testovanie

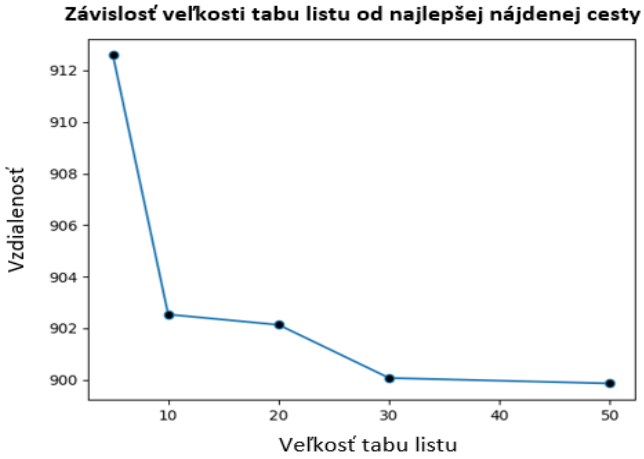
a) Tabu search rôzne veľkosti Tabu listu

Pri testovaní som sa ako prvé zameral na testovanie rôznych veľkostí tabu listu. Chcel som zistiť prípadný súvis medzi časom, vzdialenosťou a veľkosťou tabu listu.

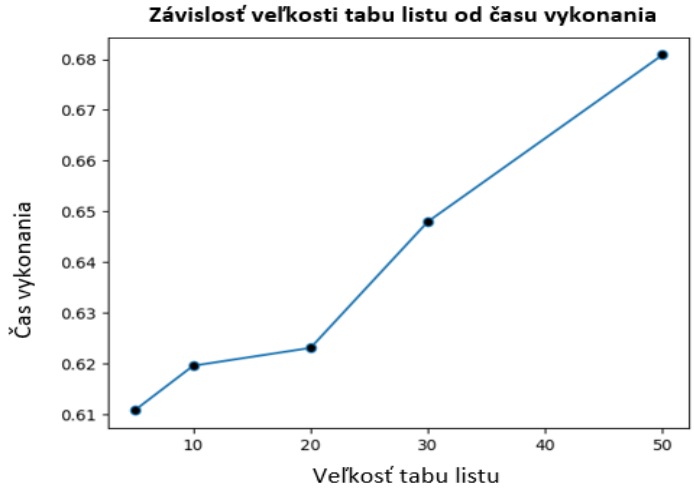
Meranie bolo vykonané na vzorke 20 spustení algoritmu pre každú veľkosť tabu listu. Počet miest bol 20, veľkosť okolia 19 a počet iteracií bol nastavený na 1000 pri každom spustení. Jednotlivé hodnoty v grafoch predstavujú priemer z 20 meraní.

Z obidvoch grafoch môžeme sledovať miernu závislosť, ktorá je však zanedbateľná. Pri viacnásobnom vykonaní rovnakého testu vychádzali rôzne hodnoty, preto nemožno jednoznačne tvrdiť že lepšie riešenie nájdeme vždy pri väčšom tabu liste. Môže to byť spôsobené hlavne tým, že pri generovaní nových stavov sa vymení ľubovoľná dvojica miest, a nie susedné, čo môže prispieť k jednoduchšiemu vyslobodeniu z lokálneho extrému a aj minimalizácii dosiahnutia lokálneho extrému.

Veľkosť tabu listu preto postačí pre testované parametre nastaviť na veľkosť 10, aby bolo nájdené aspoň dostatočné riešenie, a v niektorých prípadoch aj najlepšie.



Pri pozorovaní zmeny času tiež vidno miernu závislosť, ktorá avšak nieje nijak veľka, pretože zisťovanie prítomnosti prvku v tabu liste prebieha pomocou indexu, čiže v konštantnom čase a nie je treba zakaždým prechádzať celý list



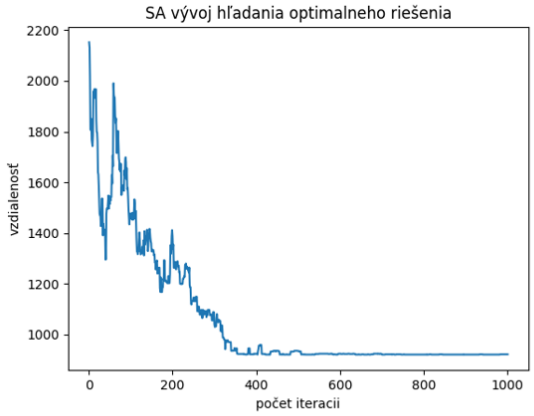
b) Simulované žíhanie testovanie zmeny teploty a faktoru ochladzovania

Ako ďalšie som testoval rôzne hodnoty parametrov pre algoritmus simulovaného žíhania. Tak isto ako v prvom prípade som pracoval s 20 mestami, veľkosť okolia bolo nastavene na 19, a počet iterácii bolo 1000.

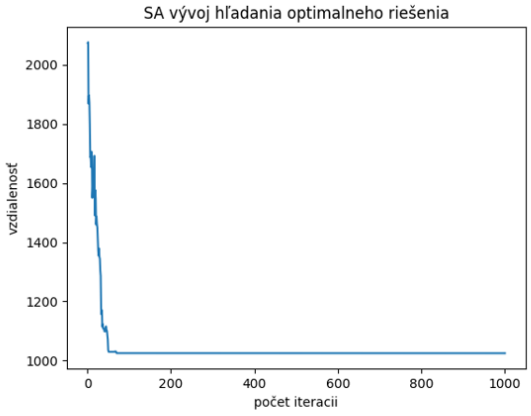
Použil som 4 rôzne konfigurácie. Každá mala iné hodnoty teploty alebo faktoru ochladzovania. Konfiguracie:

(1) Teplota = 50 , Faktor ochladzovania = 0.9951
(2) teplota = 50 , Faktor ochladzovania = 0.95
(3) teplota = 50 , Faktor ochladzovania = 0.999
(4) teplota = 5000 , Faktor ochladzovania = 0.981

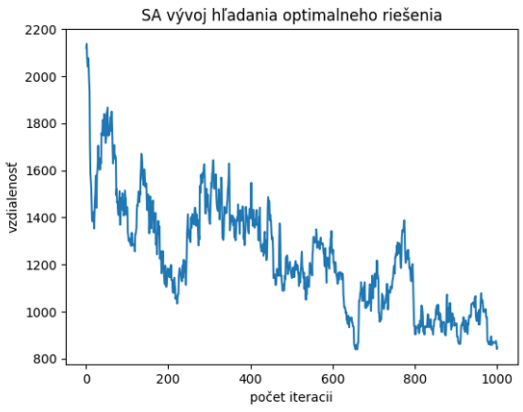
Nasledujúce grafy znázorňujú vývoj hľadania optimálneho riešenia pre jednotlivé konfigurácie. Konfigurácie boli zvolené tak, aby poukazovali na príliš rýchle a pomalé ochladzovanie, a následne podobný vývoj pri hľadaní riešenia, avšak s 10 násobne vyššou teplotou a tým pádom aj príslušne vyšším faktorom ochladzovania.



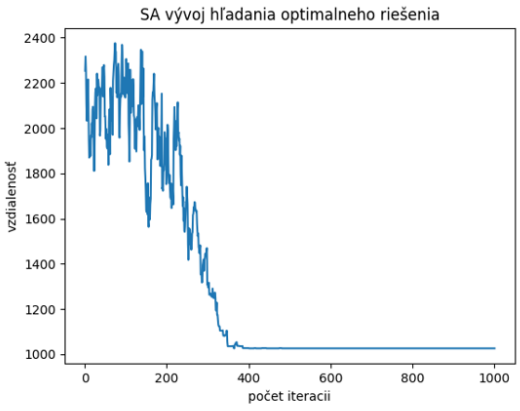
konfigurácia 1



konfigurácia 2



konfigurácia 3



konfigurácia 4

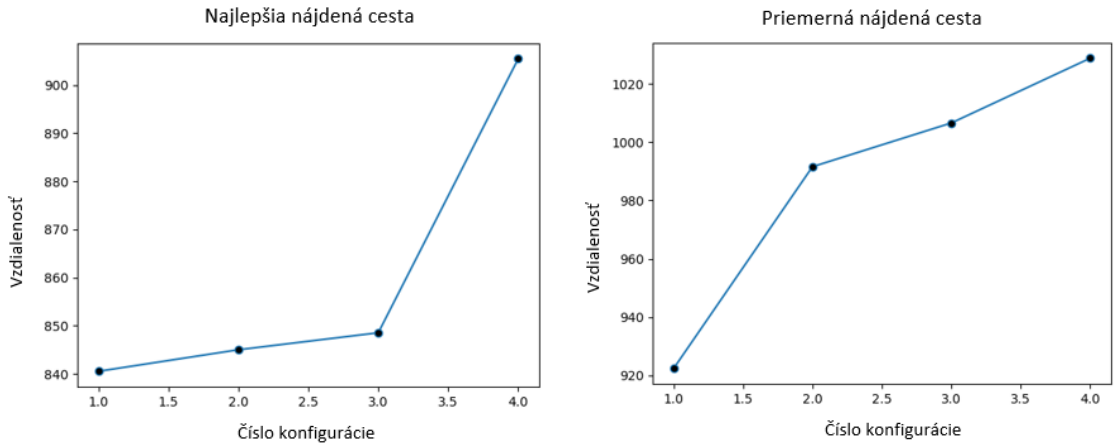
Nad každou konfiguráciou bolo vykonaných 5 opakovaní zakaždým s rovnakým vstupom a boli zachytávané hodnoty najlepšej nájdenej cesty, ako aj priemernej z 5 meraní. Najnižšie hodnoty pre najlepšie nájdené riešenie, ako aj priemerné sme dostali pri konfigurácii 1, ktorá síce našla najlepšie riešenie až v úplne posledných iteráciách, no už okolo 400. iterácii sa jej podarilo nájsť riešenie ktoré sa od najlepšieho líšilo len v jednotkách.

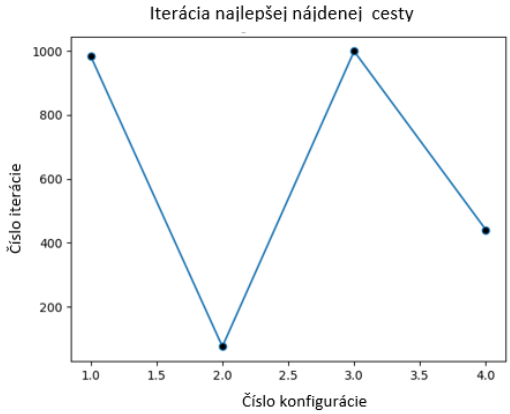
Toto nájdené riešenie bolo aj skutočné najlepšie pre dané rozloženie miest, takže nešlo len o ďalšie lokálne minimum, ale už globálne. Na druhej strane, pri ostatných konfiguráciách sa nám podarilo nájsť len niektoré z lokálneho minima.

Pri konfigurácii 2 bol príliš veľký faktor ochladzovania, no aj tak sa nám podarilo nájsť skoro optimálne riešenie už po vykonaní necelej 50 iterácií. Takéto dobré hodnoty sa však nepodarí pri danej konfigurácii dosiahnuť vždy, čo svedčí aj hodnota z grafu priemernej nájdenej cesty, kde sa priemerná hodnota pohybuje skôr vo vyššej polovici grafu.

Pri konfigurácii 3 bol faktor ochladzovania zase príliš malý, kvôli čomu sa nedokázalo v 1000 iteráciách nájsť už optimálne riešenie, ale zbytočne dlho sa obiehalo okolo neho.

4 konfigurácia mala podobný priebeh ako 1. no počiatočná teplota bola 100 krát väčšia. Napriek tomu program pomocou nej našiel najhoršie cesty spomedzi ostatných konfiguracií. Z tohto zistenia možno potvrdiť, že úspešnosť algoritmu simulovaného žíhania nezávisí len na faktore ochladzovania, ale do určitej miery prispieva aj rozumné nastavenie vstupnej teploty.

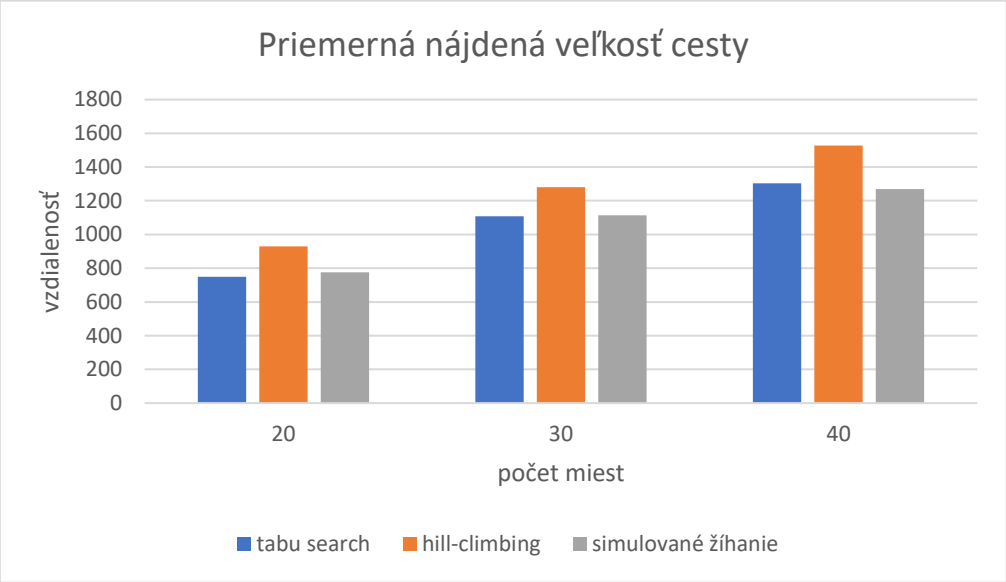




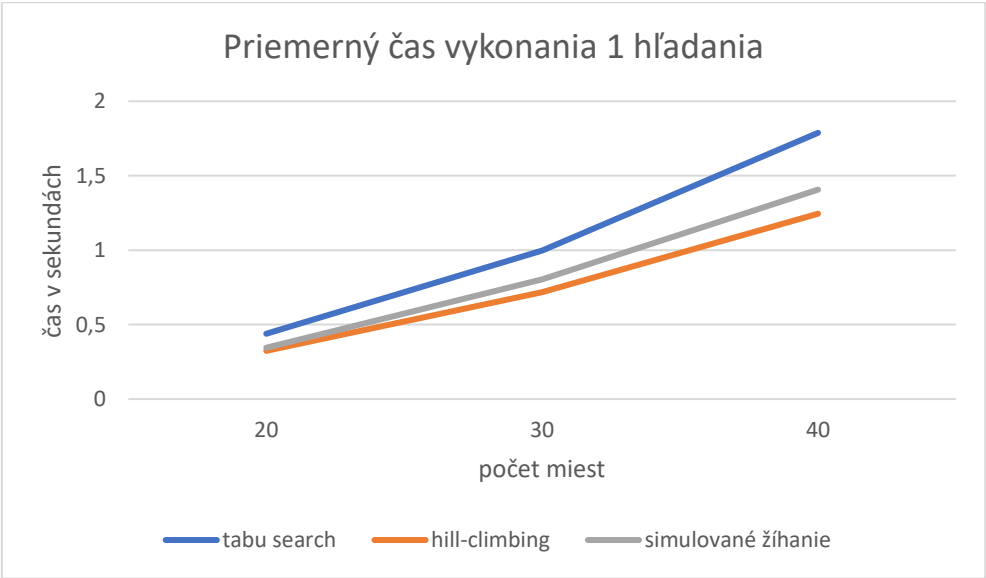
c) Porovnanie hill-climbing, tabu search a simulated annealing

Ako posledné som porovnal rôzne algoritmy pre rovnaké vstupy a rôznu veľkosť miest, a sledoval vývoj ich časov a veľkosti nájdenej cesty. Veľkosť okolia bolo nastavené na hodnotu veľkosť miest-1 a počet iterácii bolo 1000. Veľkosť tabu listu bola 10 a hodnoty parametrov pre simulované žíhanie boli nastavené na konfiguráciu 1 z predchádzajúceho testovania. Hodnoty zobrazené v grafoch predstavujú vždy priemer z 5 meraní.

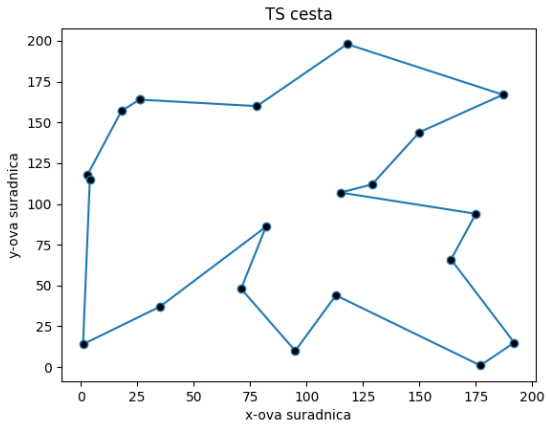
Z grafu možno jednoznačne vidieť, že tabu search aj simulované žíhanie dosiahli vo všetkých prípadoch lepšie nájdené riešenia ako hill-climbing. Rozdiely v nájdených cestách medzi týmito 2 algoritmami sú minimálne, kde tabu search bol na tom lepšie pri 20 a 30 mestách, a simulované žíhanie zase pri 40 mestách.

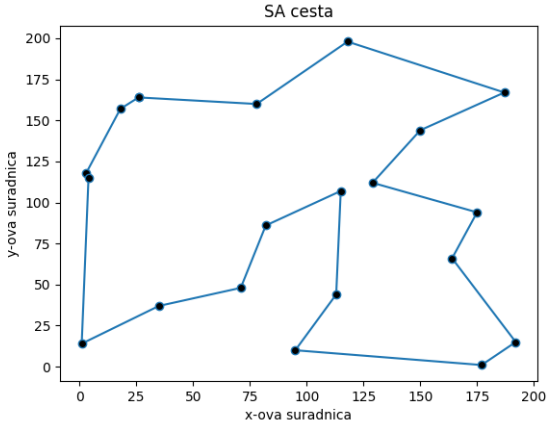


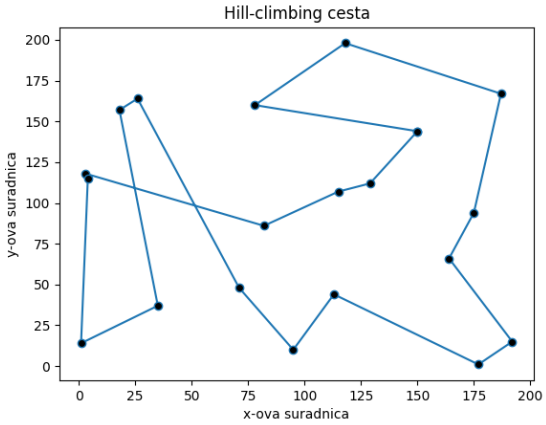
Pri Porovnaní časov je na tom najlepší hill-climbing, ale za cenu nájdenia najhoršieho riešenia. Teda v kombinácii času a nájdeného riešenia je na tom najlepšie simulované žíhanie. Tabu search dokáže nájsť vo väčšine prípadom o kúsok lepšie riešenia ako simulované žíhanie, ale za cenu citeľného zvýšenia potrebného času pre vykonanie.



Na záver pridávam ešte ukážku najlepších nájdených ciest jednotlivých algoritmov pri 20 mestách. Tabu search ako aj simulované žíhanie dosiahli uspokojivé riešenia. TS sa podarilo nájsť o 5km menšiu cestu ako SA. Najhoršie dopadol hill-climbing, ktorý našiel cestu dlhšiu o 200km ako predchádzajúce.Kvôli čomu je pri ňom možné vidieť aj prekríženie ciest, takže nemožno dané riešenie považovať ako uspokojivé.







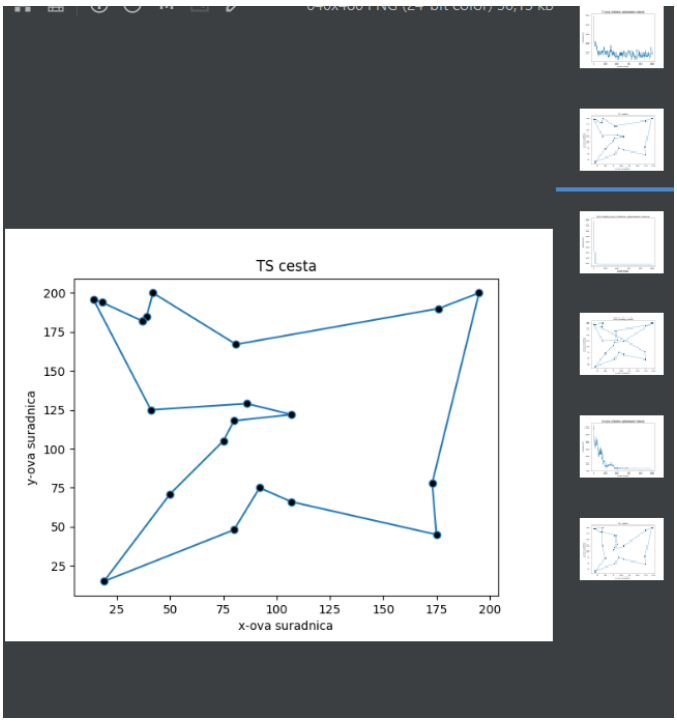
5) Používateľské prostredie

Môj program neobsahuje interaktívne prostredie, no je možné meniť jeho správanie na základe globálnych premenných umiestnených na začiatku programu. Je možné si nastaviť počet miest a iterácii, ako aj veľkosť okolia. Ďalej je možné zmeniť veľkosť tabu listu pri Tabu search a teplotu a faktor ochladzovania pri simulovanom žíhaní.

```
pocet_miest = 20
pocet_iteracii = 1000
velkost_okolia = pocet_miest-1

tabu_list_velkost = 10

teplota = 50
faktor_ochladzovania = 0.9951
```

Po spustení programu sa zobrazí nasledovný výpis a vykreslenie grafov cesty a vývoju riešenia pre každý algoritmus. Začiatočná a koncová cesta na obrázku nie sú zobrazené kompletné, lebo sa naň nezmestili.

```
TS algoritmus
------------------------------
Zaciatocna cesta: [[173, 78], [92, 75]
vzdialenost: 1963.25
Koncova cesta: [[37, 182], [18, 194],
vzdialenost: 851.45
0.4227 sekund

Hill-climbing algoritmus
------------------------------
Zaciatocna cesta: [[173, 78], [92, 75]
vzdialenost: 1963.25
Koncova cesta: [[37, 182], [173, 78],
vzdialenost: 983.65
0.2908 sekund

SA algoritmus
------------------------------
Zaciatocna cesta: [[173, 78], [92, 75]
vzdialenost: 1963.25
Koncova cesta: [[42, 200], [14, 196],
vzdialenost: 856.07
0.3505 sekund
```



6) Rozšíriteľnosť a optimalizácia

Môj program sa mi oproti 1.verzii podarilo zlepšiť a optimalizovať vo viacerých oblastiach.

Jednou zo zmien bolo generovanie náhodných indexov miest, ktoré si budú vymieňať pozície. Najprv som používal na generovanie náhodného čísla funkciu random.randint(), ktorá však nebola najrýchlejšia a tak som ju vymenil za rýchlejšiu alternatívu, a to random.getrandbits(7), ktorá vráti taktiež integer, avšak pracuje s bitmi a vdaka tomu pracuje rýchlejšie. Výsledná hodnota indexu potom bol zvyšok po delení počtu miest. Výsledný čas sa mi tak podarilo zmenšiť o nejakých 7% čo nie je nejaké obrovské číslo, no vzhľadom na to akou malou zmenou sa to podarilo zrealizovať tak ide o nezanedbateľnú hodnotu.

Ďalšou zmenou, už viacej komplexnejšou bolo zmena systému generovania okolia pri tabu search a hill-climbing. Pôvodná verzia generovala vždy vopred stanovený počet potomkov, a po každom vygenerovaní ho porovnala s zatiaľ najlepším nájdeným z daného okolia a v prípade najlepšia lepšieho ho vymenila. Toto riešenie bolo pamäťovo efektívnejšie, pretože som si uchovával len najlepší vektor z okolia, no v prípade vygenerovania 2 rovnakých potomkov, ich bralo ako 2 rôznych, čiže napríklad pri generovaní okolia veľkosti 20 sa vygenerovalo 20 potomkov, no z toho mohli byť napríklad 3 rovnaké , a tak bola skutočná veľkosť okolia len 18 a nie 20. taktiež porovnávanie každého vygenerovaného potomka nebolo úplne optimálne. Toto som vyriešil zavedením while cyklu, ktorý generoval nové jedinečné vektory pre okolie a dict, ktorý je rýchlejší ako list a ukladal do neho okolie, ktoré bolo následne zoradené podľa kľúča a vybral sa najlepší vektor. Toto riešenie je taktiež o niečo rýchlejšie ako pôvodné a to aj napriek while cyklu.

Ďalšie vylepšenia by sa dali realizovať pri počítaní vzdialenosti, kde by sa nepočítala odznova celá dĺžka cesty pri každom vygenerovanom novom vektore, ale by sa len zmenili hodnoty pri vymenených 2 miestach, čo by zlepšilo výsledný čas

Program som testoval aj pri počte miest 50, kde už však je potrebné nastaviť iné parametre pre algoritmy, pretože tie určené pre 20 miest už pri tomto počte miest nedokážu nájsť najlepšiu cestu.

Z testovania som okrem vhodného nastavenia parametrov pre počet miest 20 aj 30 na nájdenia čo najlepšej cesty testoval aj jednotlivé algoritmy, kde v kombinácii potrebného času a nastavenej cesty najlepšie zvládol svoju úlohu algoritmus simulovaného žíhania. Avšak je potrebné mať dobre nastavené parametre, čo nie je úplne triviálne a zaberie to nejaký čas. Taktiež sa menia v závislosti od počtu miest, veľkosti okolia ako aj počtu iterácií.

Preto v prípade že nedokážete vhodne nastaviť parametre je jednoduchšie spoľahnúť sa na tabu search(kde mi prišlo nastavovanie vhodných parametrov omnoho jednoduchšie). Ktorý dokáže nájsť vo väčšine prípadov najlepšiu cestu, aj keď za cenu vyššieho času vykonávania.