Martin Hric
ID: 111696

Slovenská technická univerzita v Bratislave

Fakulta Informatiky a Informačných Technológii

Zadanie 3

Travelling Salesman Problem

17.11.2021

2021/2022

Martin Hric
ID: 111696

OBSAH

Martin Hric
ID: 111696

1. Znenie zadania:

Obchodný cestujúci má navštíviť viacero miest. V jeho záujme je minimalizovať cestovné náklady a cena prepravy je úmerná dĺžke cesty, preto snaží sa nájsť najkratšiu možnú cestu tak, aby každé mesto navštívil práve raz. Keďže sa nakoniec musí vrátiť do mesta z ktorého vychádza, jeho cesta je uzavretá krivka.

Je daných aspoň 20 miest (20 – 40) a každé má určené súradnice ako celé čísla X a Y. Tieto súradnice sú náhodne vygenerované. (Rozmer mapy môže byť napríklad 200 * 200 km.) Cena cesty medzi dvoma mestami zodpovedá Euklidovej vzdialenosti – vypočíta sa pomocou Pytagorovej vety.

Celková dĺžka trasy je daná nejakou permutáciou (poradím) miest. Cieľom je nájsť takú permutáciu (poradie), ktorá bude mať celkovú vzdialenosť čo najmenšiu.

Výstupom je poradie miest a dĺžka zodpovedajúcej cesty.

Martin Hric
ID: 111696

2. Úvod

Mne bola pridelená skupina E – Travelling salesman problem (TSP) s použitím kombinácie algoritmov Tabu search a Simulated Annealing .

Tabu search:

Zakázané prehľadávanie patrí do skupiny algoritmov, ktoré využívajú na hľadanie riešenia v priestore možných stavov lokálne vylepšovanie (optimalizáciu). To znamená, že z aktuálneho stavu si vytvorí nasledovníkov a presunie sa do takého, ktorý má lepšie ohodnotenie (najlepšieho takého). Ak neexistuje nasledovník s lepším ohodnotením (a nenašli sme dostatočne dobré riešenie), tak sme v lokálnom extréme a je potrebné sa z neho dostať. Tento algoritmus si teda vyberie horšieho nasledovníka a zároveň si uloží aktuálny stav do tzv. zoznamu zakázaných stavov (tabu list). Je to nevyhnutné, aby sme sa z toho horšieho nasledovníka znovu nedostali do tohto lokálneho extrému a nevytvorili tak nekonečný cyklus. Zoznam zakázaných stavov je pomerne krátky, aby nám netrvala dlho jeho kontrola. Keď sa zaplní a je doň potrebné vložiť nový stav, tak ten najstarší sa zahodí.

Problém je opäť reprezentovaný vektorom, ktorý obsahuje index každého mesta v nejakom poradí (nejaká permutácia miest). Nasledovníci sú vektory, v ktorých je vymenené poradie niektorej dvojice susedných uzlov.

Dôležitým parametrom tohto algoritmu je dĺžka zoznamu zakázaných stavov. Príliš krátky zoznam spôsobí, že algoritmus bude často pendlovať medzi niekoľkými lokálnymi extrémami, príliš dlhý zoznam natiahne čas riešenia, lebo bude dlho trvať kontrola každého stavu, či nie je zakázaný. Je potrebné nájsť jeho vhodnú dĺžku.

Simulated Annealing:

Simulované žíhanie patrí do skupiny algoritmov, ktoré využívajú na hľadanie riešenia v priestore možných stavov lokálne vylepšovanie. Zároveň sa algoritmus snaží zabrániť uviaznutiu v lokálnom extréme. Z aktuálneho uzla si algoritmus klasicky vytvorí nasledovníkov. Potom si jedného vyberie. Ak má zvolený nasledovník lepšie ohodnotenie, tak doň na 100% prejde. Ak má nasledovník horšie ohodnotenie, môže doň prejsť, ale len s pravdepodobnosťou menšou ako 100%. Ak ho odmietne, tak skúša ďalšieho nasledovníka. Ak sa mu nepodarí prejsť do žiadneho z nich, algoritmus končí a aktuálny uzol je riešením. Pre nájdenie globálneho extrému je dôležitý správny rozvrh zmeny pravdepodobnosti výberu horšieho nasledovníka. Pravdepodobnosť je spočiatku relatívne vysoká a postupne klesá k nule.

Problém je opäť reprezentovaný vektorom, ktorý obsahuje index každého mesta v nejakom poradí (nejaká permutácia miest). Nasledovníci sú vektory, v ktorých je vymenené poradie niektorej dvojice susedných uzlov.

Dôležitým parametrom tohto algoritmu je rozvrh zmeny pravdepodobnosti výberu horšieho nasledovníka. Príliš krátky (rýchly) rozvrh spôsobí, že algoritmus nestihne obísť lokálne extrémy, príliš dlhý rozvrh natiahne čas riešenia, lebo bude obiehať okolo optimálneho riešenia. Je potrebné nájsť vhodný rozvrh.

Martin Hric
ID: 111696

3. Opisy konrétne použitých algoritmov

Nastavenie globálnych premenných:

Na začiatku kódu si nastavím dané premenné, ktoré majú dopad na celkový chod algoritmov.

Nastavujem tam počet miest, koľko iterácii sa ma vykonať v tabu search, veľkosť mapy pre prípad keby som chcel plátno zväčšiť/zmenšiť, začiatočnú a konečnú teplotu pre SA ako aj ALPHA o koľko sa má znižovať teplota .

```
CITIES_COUNT = 20
ITERATIONS = 10000
TABU_SIZE = 50
MAP_SIZE = 500
INITIAL_TEMP = 90
FINAL_TEMP = 0.1
ALPHA = 0.01
```

Tabu search:

Algoritmus na tabu search som použil z prednášky, ktorý som upravil len do jazyka Python a implementoval som k nemu potrebné ďalšie funkcie.

```
def tabu_search(cities,solution):
    sBest = solution
    bestCandidate = solution
    tabuList = [solution]
    distances = []
    values = open("values_tabu.csv",'w', newline='')
    writer = csv.writer(values)
    for i in range(ITERATIONS):
        sNeighborhood = permutacie(bestCandidate)
        bestCandidate = sNeighborhood[0]
        for sCandidate in sNeighborhood:
            if (sCandidate not in tabuList) and (distance(cities, sCandidate) < distance(cities,bestCandidate)):
                bestCandidate = sCandidate

        if distance(cities,bestCandidate) < distance(cities,sBest):
            sBest = bestCandidate

        #distances.append(int(distance(cities,bestCandidate)))
        tabuList.append(bestCandidate)
        if len(tabuList) > TABU_SIZE:
            tabuList.remove(tabuList[0])
    #writer.writerow(distances)
    values.close()
    return sBest
```

Martin Hric
ID: 111696

Nachádzajú sa tam momentálne aj zakomentované časti kódu, ktoré tam boli len kvôli posielaniu údajov do Excelu ,kde som vytvoril grafy v testovaní.

Tento algoritmus dostane ako parameter náhodne vygenerované riešenie, spolu aj s mestami, ktoré sú uložené v dict a tak viem len číslo mesta vďaka čomu viem pomocou dict aké súradnice má, čo využívam ďalej vo väčšine funkcií, na zistenie dĺžok ako aj na vygenerovanie grafu ako vyzerá výsledné riešenie.

Simulated Annealing:

```
def simulated_annealing(cities,initial_state):

    current_temp = INITIAL_TEMP

    current_state = initial_state.copy()
    solution = current_state
    values = open("values_annealing.csv", 'w', newline='')
    writer = csv.writer(values)
    distances = []

    while current_temp > FINAL_TEMP:

        bestneighbor = random.choice(permutacie(solution))
        cost_diff = distance(cities, bestneighbor) - distance(cities, solution)
        #distances.append(int(distance(cities, solution)))

        if cost_diff < 0:
            solution = bestneighbor

        elif random.uniform(0, 1) < math.exp(-cost_diff / current_temp):
            solution = bestneighbor

        current_temp -= ALPHA
    #writer.writerow(distances)
    values.close()
    return solution
```

Martin Hric
ID: 111696

V tomto algoritme taktiež mam zakomentované časti kódu kvôli tvorbe grafu aby to nezhoršovalo výkonnosť.

Do algoritmu taktiež vstupuje náhodne vygenerované riešenie, kde vo while cykle kým teplota nieje nižšia ako mnou predom nastavená konečná teplota prebieha vytvorenie permutácií a náhodný výber jedného z nich. Vypočítaju sa vzdialenosti a ak je vzdialenosť nižšia tak sa mu priradí solution , ale ak nieje nižšia tak vo vzorci

```
random.uniform(0, 1) < math.exp(-cost_diff / current_temp):
```

sa vyberie číslo od 0 – 1 , a ak je nižšie ako číslo na základe teploty (čím vyššia current_temp tým vyššie číslo to bude v rozmedzí 0- 1 , pretože som použil funkciu math.exp , ktorá mi vracia exponent.)

Nakoniec sa teplota zníži o hodnotu ALPHA.

Náhodné súradnice a náhodné riešenie:

```
def random_coordinates(count):
    array={}
    for x in range(count):
        array[x]=[random.randrange(50,MAP_SIZE - 50),random.randrange(100,MAP_SIZE - 50)]

    return array
```

Náhodné súradnice generujem v rozmedzí 50 – veľkosť mapy-50 , kvôli tomu aby vo vyobrazení to dobre vyzeralo. Generujem ich vo funkcii main a parameter dostáva počet miest koľko ma vygenerovať , ktoré zapisuje do dict.

```
def randomSolution(cities):
    solution=[]
    listofcities=list(range(len(cities)))
    for i in range(CITIES_COUNT):
        randomcity=listofcities[random.randint(0,len(listofcities)-1)]
        solution.append(randomcity)
        listofcities.remove(randomcity)
    return solution
```

Náhodné riešenie si vyberie random číslo z listu kde je uložený počet všetkých miest, pridá ho do solution a odoberie z listu.

Martin Hric
ID: 111696

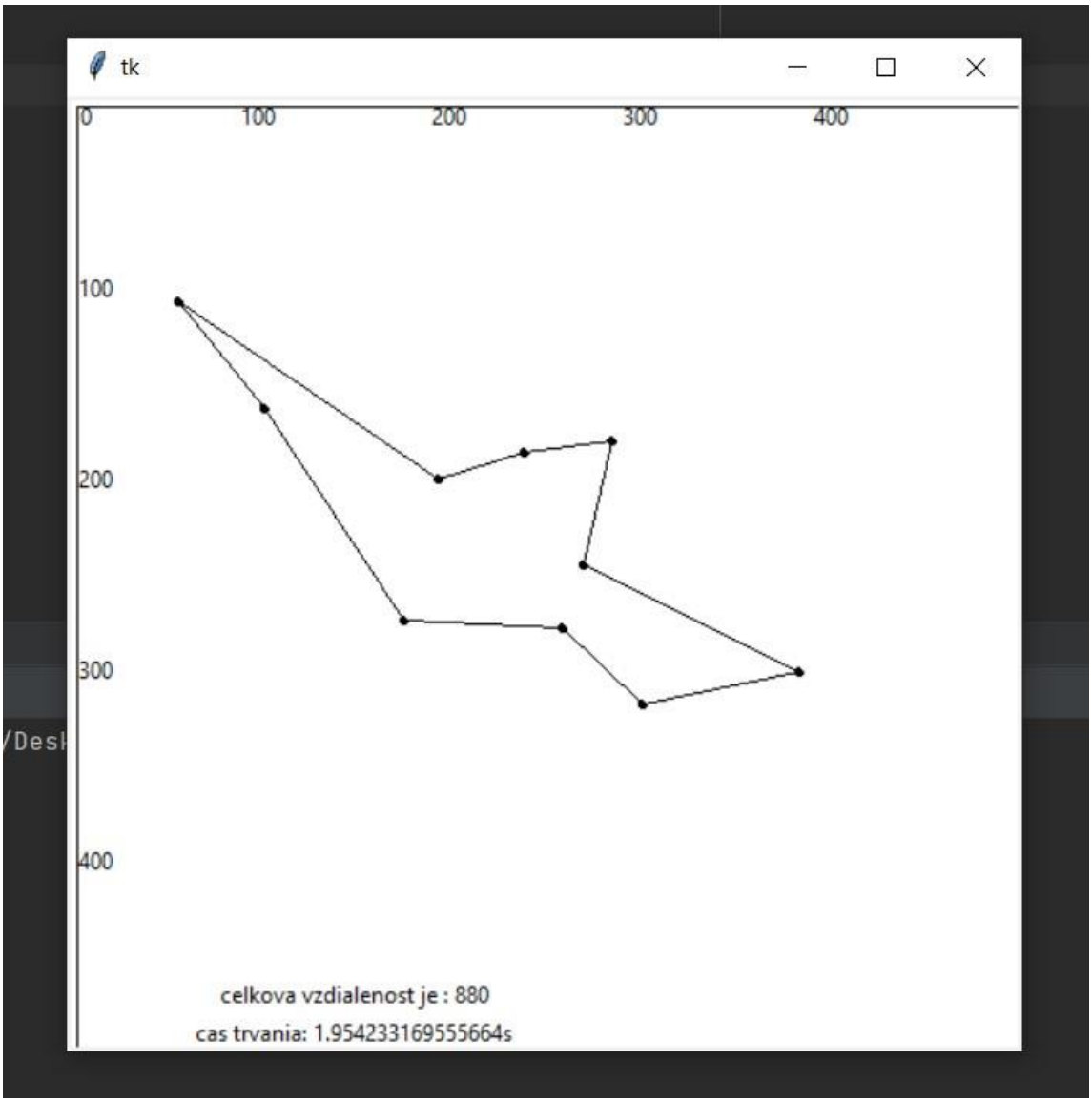
Vyobrazenie riešenia:

Na vyobrazenie riešenia som sa rozhodol si naimplementovať vlastnú funkciu, kde som konečne využil poznatky z gymnázia, kde sme si len kreslili na plátno obrázky v pythone.

Nechcel som používať nejaké externé knižnice na vyobrazenie, keď mám dané súradnice a spájať čiarky na plátne je už jednoduché.

Mám aj po krajoch napísané súradnice aby sme približne videli, kde sú vygenerované mestá.

Nakonci mi to aj vypíše celkovú vzdialenosť uzavretej krivky a čas trvania.



Martin Hric
ID: 111696

Fitness:

Riešenie fitness hodnôt, ktoré sa používajú v algoritmoch na nájdenie lepšieho riešenia som samozrejme implementoval vďaka súčtu dĺžok , vo funkcii distance() .

```
def distance(cities, solution):
    total_distance = 0
    for i in range(CITIES_COUNT - 1):
        distance1 = math.sqrt((abs(cities[solution[i]][0] - cities[solution[i+1]][0]) ** 2) + (abs(cities[solution[i]][1] - cities[solution[i+1]][1]) ** 2))

        total_distance += distance1

    distance2 = math.sqrt((abs(cities[solution[0]][0] - cities[solution[CITIES_COUNT-1]][0]) ** 2) + (abs(cities[solution[0]][1] - cities[solution[CITIES_COUNT-1]][1]) ** 2))
    total_distance += distance2

    return total_distance
```

Snažil som sa túto funkciu implentovať čo najefektívnejšie, keďže volá sa v algoritmoch vela krát, preto som nepoužil žiadne pomocné premenné.

Susedstvo:

Susedov generujem vo funkcii permutacie() , ktorá si vyberie random index a vymení ho na každej pozícii .

```
def permutacie(solution):
    neighbors = []
    randomIndex = random.randint(0,CITIES_COUNT-1)
    for i in range(CITIES_COUNT):
        if i == randomIndex:
            continue
        replace = solution.copy()
        replace[i] = solution[randomIndex]
        replace[randomIndex]= solution[i]
        neighbors.append(replace)
    return neighbors
```

Pri tejto funkcií sa mi stávali aj problémy, keďže na začiatku som ju mal zle implementovanú a do algoritmov som posielal veľmi vela susedstiev, čo vyústilo v to, že mi to trvalo pri tabu search okolo 2 minúty.

Martin Hric
ID: 111696

4. Testovanie

Testoval som časy vykonania a dĺžku uzavretej krivky riešenia.

Testovanie som vykonával na:

počet miest – 10 ,20

Tabu search: Iterácie – 1000 a 10 000

Veľkosť tabu listu – 10 a 50

S. Annealing: zač. teplota : 90

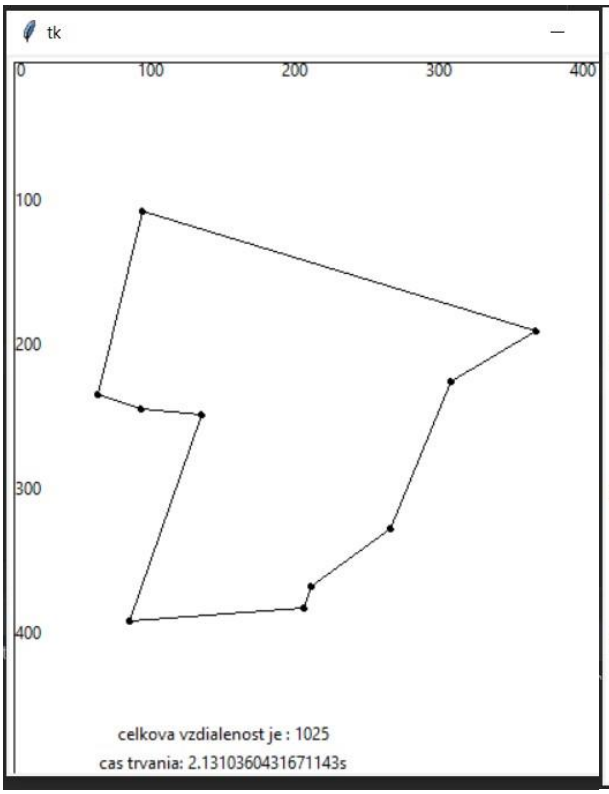
ALPHA – 0.01 a 0.5

Martin Hric
ID: 111696

MESTÁ 10:

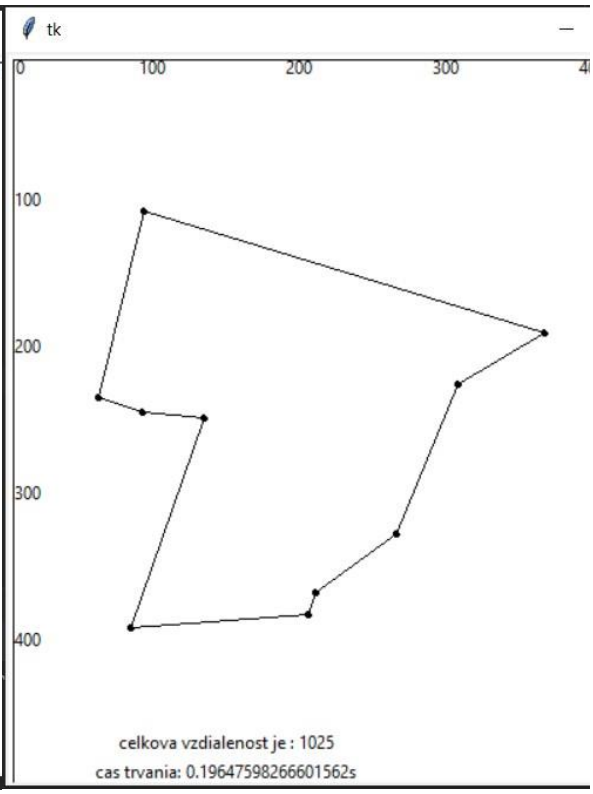
TABU SEARCH:

10 000 iterácií , 50 tabu size



Simulated annealing:

teplota 90, alpha 0.01

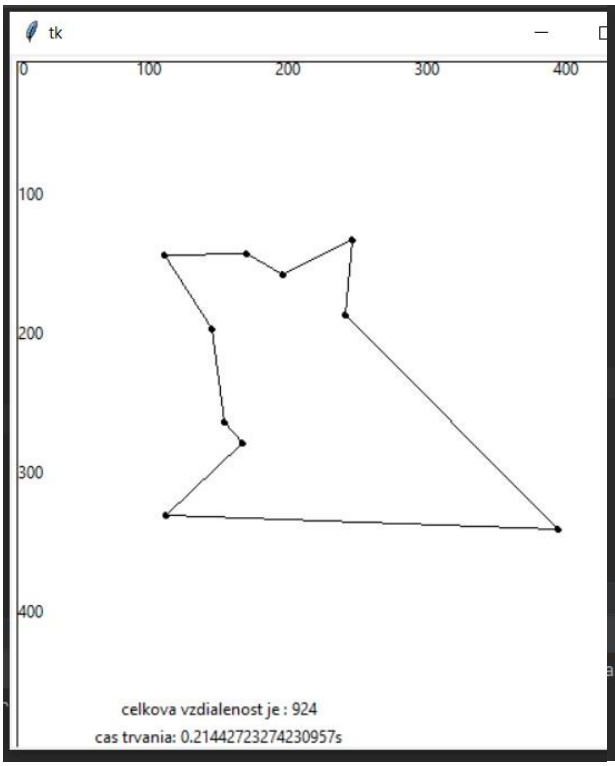


Našlo rovnaké riešenie , SA omnoho rýchlejší.

Martin Hric
ID: 111696

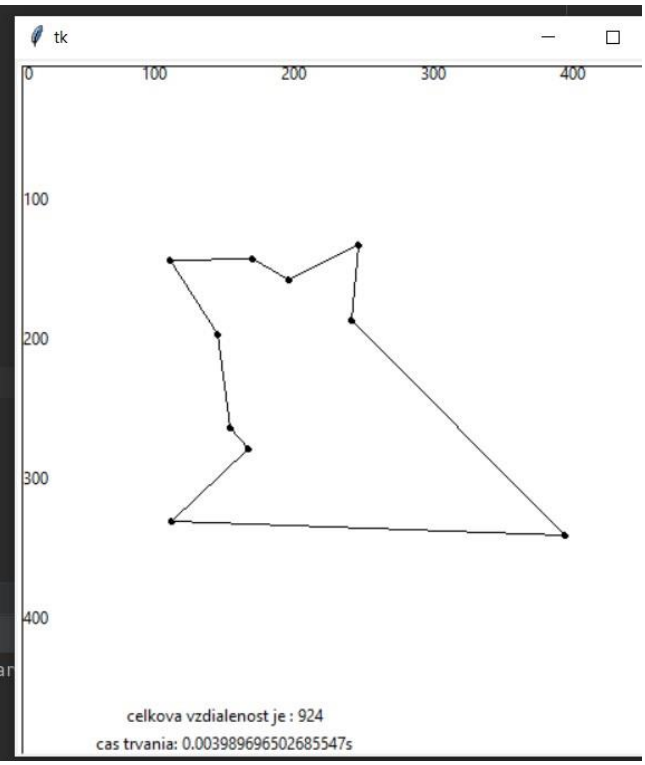
TABU SEARCH:

1 000 iterácií , 10 tabu size



Simulated annealing:

teplota 90, alpha 0.5



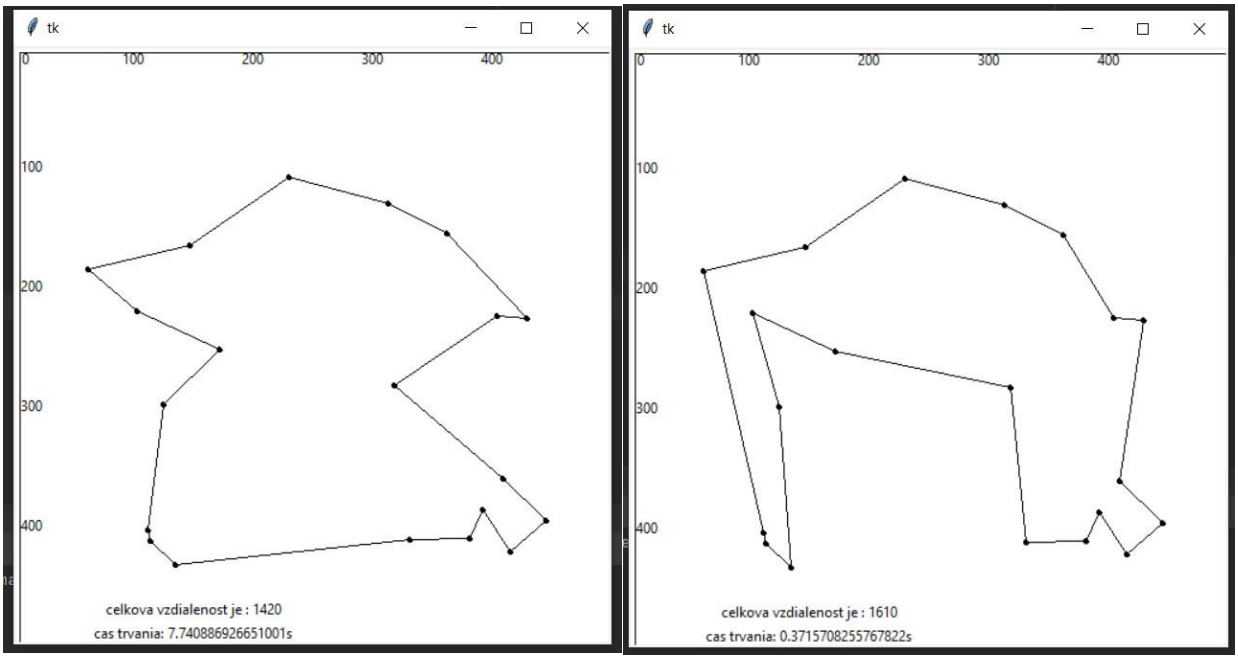
Opäť rovnaké riešenie , aj po znížení tabu size sa nezvyšujú časy , čo je zvláštne.

Martin Hric
ID: 111696

MESTÁ 20:

TABU SEARCH: Simulated annealing:

10 000 iterácií , 50 tabu size teplota 90, alpha 0.01

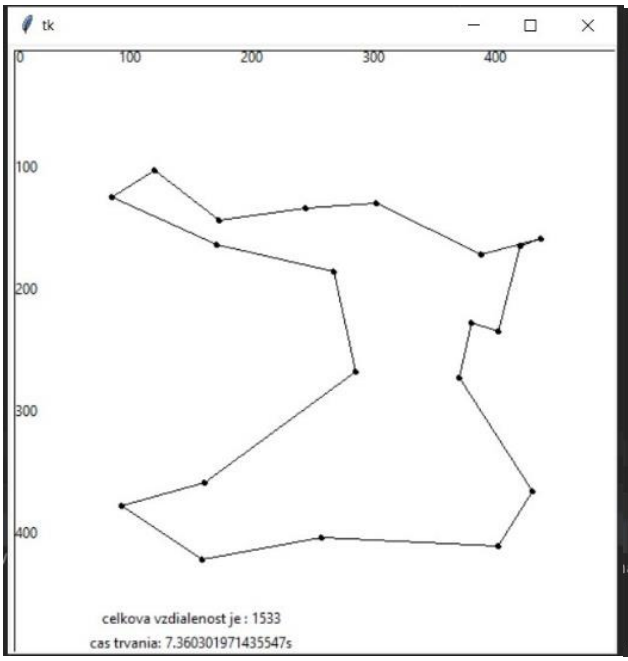


SA našlo horšie riešenie pri väčšom počte miest, stále lepší čas.

Martin Hric
ID: 111696

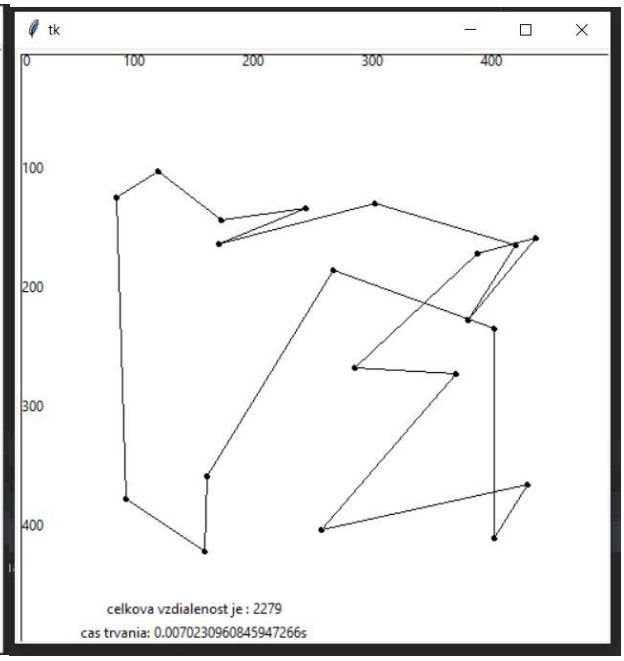
TABU SEARCH:

10 000 iterácií , 10 tabu size



Simulated annealing:

teplota 90, alpha 0.5



SA je veľmi rýchle kvôli alpha 0.5 , ale kvôli tomu aj neprehľadá dostatočne všetko a našlo nie dobré riešenie v porovnaní s TS , kde aj pri znížení tabu size čas nestúpa.

Martin Hric
ID: 111696

5. Zhodnotenie testovania

Pri znižovaní tabu size sa čas nezvyšol, čo je veľmi prekvapivé, pravdepodobne tam budem mať nejakú implementačnú chybičku.

Tabu search mal vždy omnoho horší čas oproti SA , čo môže byť aj implementáciou, keďže v TS prehľadávam všetky susedstvá , kde to pri SA si náhodne zvolí 1 a z toho ďalej robí permutácie.

Najväčší dopad na funkčnosť TS má počet iterácii, koľkokrát sa vykoná cyklus , keďže ako stoppingCondition() sú nastavené len iterácie a pri menšom počte iterácii je výsledok totožný ak pri SA zvýšime hodnotu ALPHA , čiže hodnoty o koľko sa bude teplota postupne znižovať.

Čas rapídne narastá pri zvyšujúcom sa počte miest. Často nájdu rovnaké riešenie oba algoritmy.

So zvyšujúcim sa počtom miest je potrebné zvýšiť aj iterácie / znížiť ALPHA hodnotu , keďže potrebujeme viacero susedstiev prekontrolovať v prípade lepšieho výsledku vzdialeností, za cenu horšieho času.

Martin Hric
ID: 111696

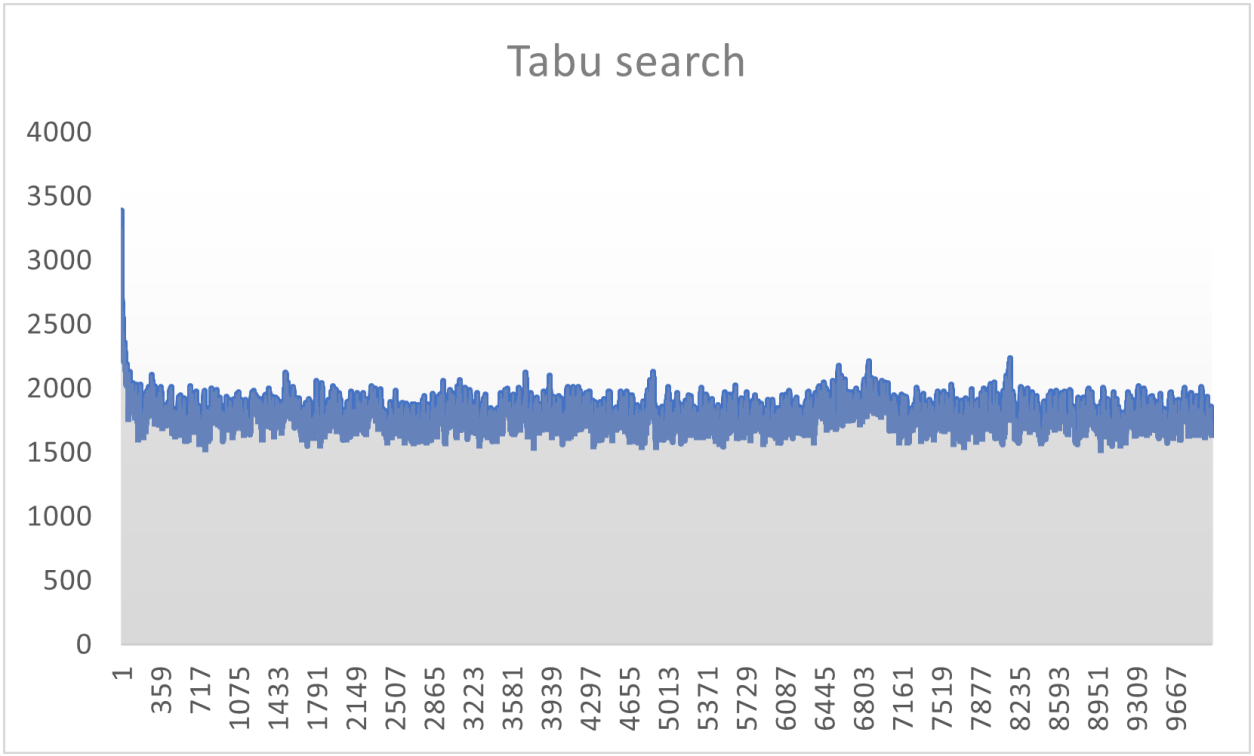
GRAFY:

Vytvoril som 2 grafy , pre TS a SA, kde som do Excelu posielal pre TS bestCandidate a SA solution .

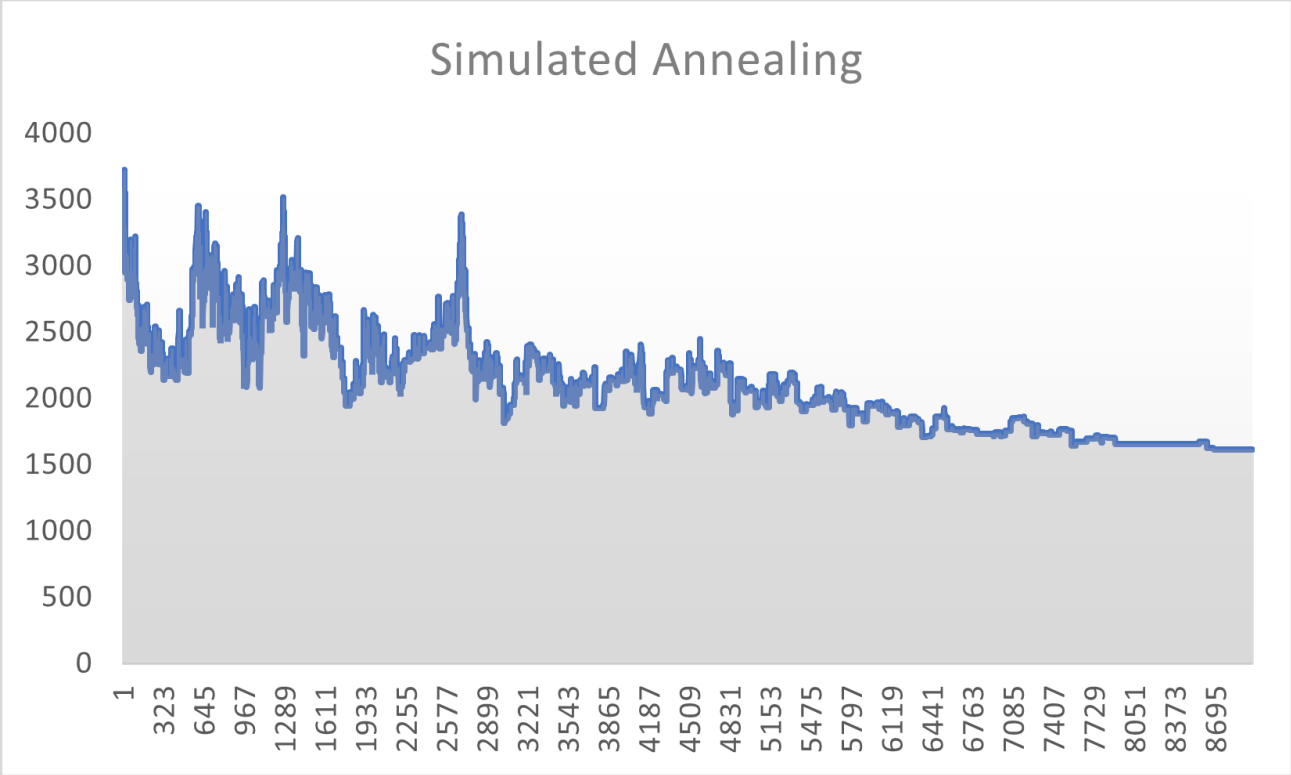
V týchto grafov je veľmi pekne vidieť krivku, keď sa začne cykliť v lokálnom maxime a príjmu aj horšie riešenie aby sa dostali postupne ku globálnemu maximu.

Bolo to vykonené pre nasledovné hodnoty:

```
CITIES_COUNT = 20
ITERATIONS = 10000
TABU_SIZE = 50
MAP_SIZE = 500
INITIAL_TEMP = 90
FINAL_TEMP = 0.1
ALPHA = 0.01
```



Martin Hric
ID: 111696



V grafoch sú znázornené na ľavej osi dĺžky , a dole je postupne ako sa menili.

Pri SA je vidieť vačšie krivky , čiže prijalo to na začiatku keď teplota bola vysoká omnoho horšie riešenia a postupne ako chladlo, prestávalo.

Martin Hric
ID: 111696