FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ´

SLOVENSKÁ TECHNICKÁ UNIVERZITA

Ilkovičova 2, 842 16 Bratislava 4

2023/2024

Umelá Inteligencia

Traveling Salesman Problem (Zadanie 3. c)

Učiteľ: Ing. Juraj Vincúr, PhD. **Realizované:**

Svätopluk Puterka

Čas: Štvrtok 16:00 – 17:50 **AIS ID:** 120870

Úvod

Problém obchodného cestujúceho je jedna z najznámejších optimalizačných úloh. Je to aj vďaka veľkému množstvu jej praktických aplikácií.

Problém obchodného cestujúceho je jedna z najznámejších optimalizačných úloh. Je to aj vďaka veľkému množstvu jej praktických aplikácií.

Nech je daných N miest a nech je možné dostať sa z každého mesta do všetkých ostatných (buď priamo alebo cez niektoré iné mesto). Nech je každá cesta medzi dvomi navzájom prepojenými mestami ohodnotená číslom, ktoré môže vyjadrovať vzdialenosť, cenu, čas atď. Cieľom obchodného cestujúceho je vybrať sa z mesta, v ktorom sa práve nachádza, navštíviť každé mesto práve raz a vrátiť sa do počiatočného mesta. Pritom sa snaží túto cestu absolvovať tak, aby precestoval čo najmenšiu vzdialenosť, zaplatil čo najmenej peňazí a podobne

Nech je daných N miest a nech je možné dostať sa z každého mesta do všetkých ostatných (buď priamo alebo cez niektoré iné mesto). Nech je každá cesta medzi dvomi navzájom prepojenými mestami ohodnotená číslom, ktoré môže vyjadrovať vzdialenosť, cenu, čas atď. Cieľom obchodného cestujúceho je vybrať sa z mesta, v ktorom sa práve nachádza, navštíviť každé mesto práve raz a vrátiť sa do počiatočného mesta. Pritom sa snaží túto cestu absolvovať tak, aby precestoval čo najmenšiu vzdialenosť, zaplatil čo najmenej peňazí a podobne.

2. Zadanie

Pri mojej implementácii som implementoval 2 heuristiky:

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.

Poznámky k externým knižniciam.

Na vykreslovaní výstupných grafov porovnania algoritmov som použil

knižnicu **matplotlib**.
Pri odovzdávaní som odovzdal aj súbor **requirements.txt** ktorý treba nainštalovať aby sa daná závislosť nainštalovala.

Inštalácia programu:

1. **Vytvorenie a spustenie virtuálneho prostredia**

   Pre inštaláciu programu je potrebné si vytvoriť virtuálne prostredie príkazom:

   python -m venv env

   Následne si prestredie aktivujeme s príkazom:

   source env/bin/activate

2. **Nainštalovanie závislostí**

   Program má svoje 3 stranové knižnice, definované v súbore **requirements.txt**

   Pre inštaláciu závislostí použijeme príkaz:

   python -m pip install -r requirements.txt

Spustenie programu:

python main.py –help - vypisanie pouzivania programu.
python main.py -d -r -c40 --max-iterations=100 ts - pustenie tabu search s debug modom s 40 random vygenerovanymi mestami

python main.py -d -r -c40 --max-iterations=100 ts - pustenie simulated annealing s debug modom s 40 random vygenerovanymi mestami

Zakázané prehľadávanie (tabu search)

Moja implementácia:

```
def tabu_search(
        distance_matrix,
        initial_solution: Cities,
        cities: Cities,
        debug_mode: bool,
        draw_config: DrawConfig,
        iterations=100,
        tabu_size=100):
    """
    Perform tabu search with 2-opt heuristic for finding neighborhood
    """
    best_solution = initial_solution[:]
    best_cost = calculate_total_distance(distance_matrix, best_solution)
    tabu_list: List[Coordinate] = []

    iteration = 1
    while iteration < iterations or len(tabu_list) > tabu_size:

        neighborhood = []
        for neighb_a_index in range(1, len(best_solution) - 1):
            for neighb_b_index in range(neighb_a_index + 1, len(best_solution)):
                if (neighb_a_index, neighb_b_index) not in tabu_list:
                    new_route = swap_2opt(best_solution, neighb_a_index, neighb_b_index)
                    new_cost = calculate_total_distance(distance_matrix, new_route)
                    # optimize
                    neighborhood.append((new_route, new_cost, neighb_a_index, neighb_b_index))

        neighborhood.sort(key=lambda x: x[1])
        for new_route, new_cost, neighb_a_index, neighb_b_index in neighborhood:
            if new_cost < best_cost:
                best_solution, best_cost = new_route, new_cost

                if debug_mode:
                    draw_route(draw_config, cities, best_solution, iteration, iterations)


            tabu_list.append((neighb_a_index, neighb_b_index))
            if len(tabu_list) > tabu_size:
                tabu_list.pop(0)
            break

        iteration += 1
```

Pri implementácii som sa riadil pseudo kódom zo zdroja [wikipedia] s miernymi modifikáciami.

Na začiatku algoritmu som si vygeneroval náhodné riešenie, ktoré som zároveň aj označil aj ako iniciálne najlepšie riešenie.
Pre toto riešenie som vypočítal aj jeho cenu. Cenu cesty počítam ako súčet euklivoských vzdialeností medzi jednotlivými mestami.
Následne som v cykle vyhľadával ďalšie riešenia.

Riešenia vyhľadávané v cykle:

Na začiatku som vyhľadal všetky susedstvá, ktoré nie sú v tabu liste. Postupoval som tak že som spravil všetky možné kombinácie medzi mestami a a ak daná kombinácia nebola v tabu liste, tak som nad ňou vykonal algoritmus, ktorý sa nazýva 2-opt.
2-opt algoritmus odstráni z cesty hrany mestá a nahradí ich inými dvoma.
Postupu prebieha v troch krokoch:

1. zoberieme cestu z prvého mesta až do aktuálneho (v mojom algoritme parameter ***i***)
2. zoberieme cestu z aktuálneho mesta (i) až do druhého mesta (v mojom algoritme parameter ***k***) a pridáme ho v reverznom poradí do cesty z bodu č. 1.
3. zoberieme cestu z **k+1** mesta až do koncového mesta a pridáme ho do novej cesty.

Riešenie som pridal do poľa **neighborhood**.

```
def swap_2opt(route, i, k):
    """
    Swap the endpoints of two edges by reversing a section of nodes,
    typically between node i and node k.
    2-opt heuristic pseudo code: https://en.wikipedia.org/wiki/2-opt

    1. take route[0] to route[v1] and add them in order to new_route
    2. take route[v1+1] to route[v2] and add them in reverse order to new_route
    3. take route[v2+1] to route[start] and add them in order to new_route
    """
    new_route = route[0:i] + route[i:k + 1][::-1] + route[k + 1:]
    return new_route
```

Následne som toto pole zoradil podľa ceny cesty. Zoradenie som robil

preto,

že dĺžka tabu listu je obmedzená a tým by sa mohli niektoré dobré riešenia dostať do výberu a naopak niektoré zlé by sa tam zase dostali.

Potom som z tohto zoradeného zoznamu vybral najlepšie riešenie. Taktiež som všetkých vybratých susedov zaradil do tabu listu.

Takto som pokračoval v cykle opakoval všetky kroky, dovtedy, dokedy som buď nedosiaho maximálny počet povolených iterácii, alebo kým nebola dosiahnutá maximálne dĺžka tabu listu.

Simulované žíhanie (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.

Moja implementácia:

```python
for iteration in range(max_iterations):
    # cool the system down
    temperature *= alpha
    if temperature < stopping_temperature:
        break

    # pick two random indexes of cities and perform 2-opt swap
    i, k = sorted(random.sample(range(1, len(current_solution)), 2))
    new_solution = swap_2opt(current_solution, i, k)
    new_cost = calculate_total_distance(distance_matrix, new_solution)

    cost_diff = new_cost - current_cost
    # always accept better solutions and worse solutions accept only with
    # certain probability
    if cost_diff < 0 or math.exp(-cost_diff / temperature) > random.random():
        current_solution = new_solution
        current_cost = new_cost

        # update the best solution
        if new_cost < best_cost:
            best_solution = new_solution
            best_cost = new_cost

            if debug_mode:
                draw_route(draw_config, cities, best_solution, iteration, max_iterations)
```

Podobne ako pri Tabu Search som na začiatku inicializoval najlepšie riešenie na jedno náhodne vygenerované a následne som začal vykonávať jeho iterácie.

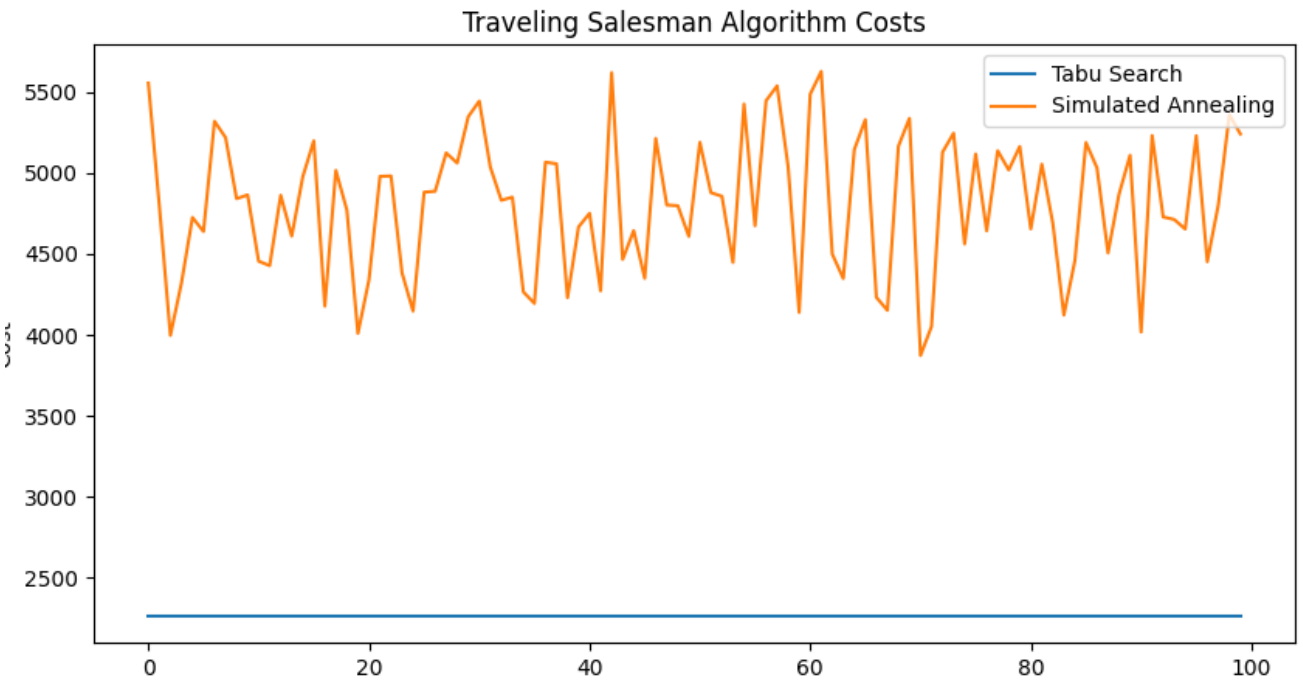
V cykle som vybral náhodne 2 mestá a vykonal nad nimi 2-opt swap. Následne som vypočítal cenu tejto vygenerovanej cesty.

Ak táto cena bola lepšia ako celková najlepšia cena tak ju nastav ako novú najlepšiu cestu.

Grafy

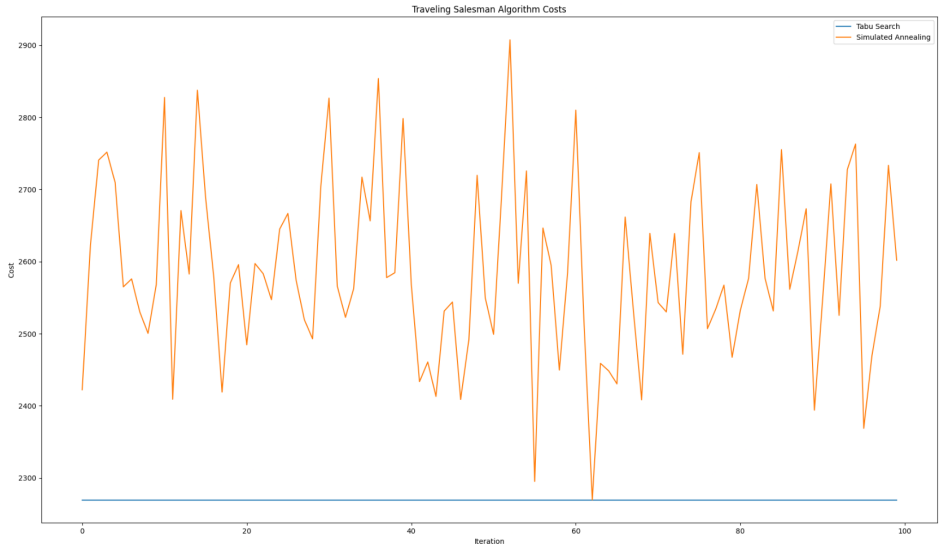
Porovnanie Tabu Search a Simulated Annealing:

porovnanie prebehlo na rovnakom nastavení algoritm pri max. počte iterácii.



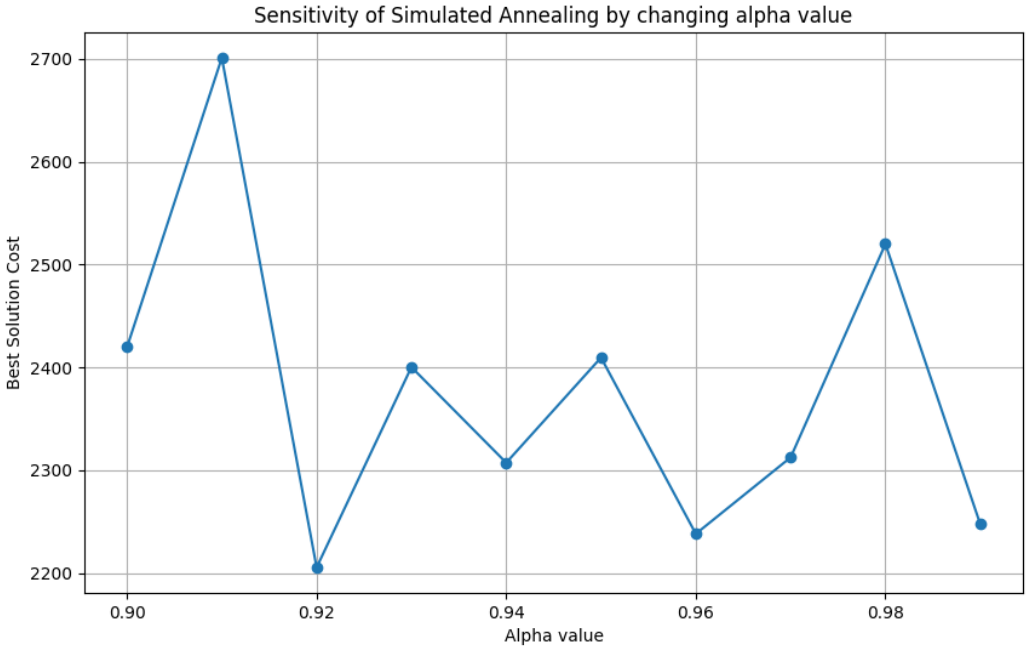
Porovnanie Tabu Search a Simulated Annealing:

porovnanie prebehlo pro rovnako nastavení iterácii v tomto prípade na 10 tisíc.



Analýza senzitivity pre SImulated annealing

Hodnota alpha bola postupne nastavená od 0.90 - 0.99 na rovnakom zadaní.



Analýza senzitivity pre Tabu Search

Pre tejto analýze som menil maximálnu veľkosť tabu listu.