Johanna Tilešová AIS ID: 127293

Dokumentácia

Travelling Salesman Problem (TSP)

Zadanie 1c

Obsah

Johanna Tilešová AIS ID: 127293

Zadanie

Problém obchodného cestujúceho sa zaoberá plánovaním trasy pre cestujúceho, ktorý musí navštíviť niekoľko miest. Cestujúci sa snaží, aby ho cesta vyšla, čo najlacnejšie, a tak potrebuje minimalizovať náklady na cestovanie, pričom cena prepravy závisí od dĺžky trasy.

Cieľ zadania

Cieľom je navštíviť každé mesto presne raz a na konci sa vrátiť do mesta, kde začal. Pre lepšie vypočítanie vzdialenosti sú mestá reprezentované ako body na rovine s náhodne generovanými súradnicami. Vzdialenosť medzi mestami sa počítajú podľa Euklidovej vzdialenosti, čo umožňuje určiť, ako ďaleko sú od seba.

Očakávaný výstup

Výstupom úlohy je nájsť také poradie miest, ktoré vedie najkratšou možnou trasou.

Riešenie

Úlohou bolo spracovať problém obchodného cestujúceho pomocou dvoch zadaných algoritmov – tabu search a simulated annealing.

1. Zakázané vyhľadávanie (Tabu Search)

Zakázané prehľadávanie patrí medzi algoritmy, ktoré riešia problémy **lokálnou optimalizáciou**. Algoritmus sa vždy presunie do stavu s lepším ohodnotením. Ak taký stav neexistuje a sme v lokálnom extréme, vyberie horší stav a uloží aktuálny do tzv. tabu listu, aby sa nevrátil späť do extrému a nevytvoril nekonečný cyklus. Tabu list je krátky, aby kontrola netrvala dlho, a keď sa naplní, najstarší stav sa vymaže.

Problém je reprezentovaný ako vektor miest, nasledovníci vznikajú výmenou miest či otočením časti cesty. Dĺžka tabu listu musí byť správne nastavená, aby sa predišlo častým návratom do lokálnych extrémov alebo zbytočnému predlžovaniu riešenia.

Dôležitým prvkom algoritmu je **dĺžka tabu listu**. Ak je príliš krátky, algoritmus bude často prechádzať medzi niekoľkými lokálnymi extrémami, no ak je príliš dlhý, zbytočne predĺži čas riešenia, pretože kontrola zakázaných stavov bude trvať príliš dlho. Preto je potrebné nájsť vhodnú dĺžku tabu listu.

2. Simulované žíhanie (Simulated annealing)

Simulované žíhanie je algoritmus, ktorý **využíva lokálne vylepšovanie** a zároveň sa snaží **vyhnúť uviaznutiu v lokálnom extréme**. Z aktuálneho uzla si vytvorí nasledovníkov a jedného z nich vyberie. Ak má lepšie ohodnotenie, algoritmus doň vždy prejde. Ak má horšie ohodnotenie, môže doň prejsť s určitou pravdepodobnosťou, ktorá klesá počas behu algoritmu. Ak nevyberie žiadneho nasledovníka, algoritmus skončí a aktuálny uzol je riešením. Kľúčovým faktorom je rozvrh zmeny pravdepodobnosti prechodu do horšieho stavu, ktorý musí byť správne nastavený.

Problém je opäť reprezentovaný vektorom miest (permutáciou), pričom nasledovníci vznikajú výmenou miest alebo otočením časti cesty. Dôležitý je správny rozvrh pravdepodobnosti – ak klesá príliš rýchlo, algoritmus môže uviaznuť v lokálnom extréme, ak príliš pomaly, riešenie bude trvať dlho.

Reprezentácia údajov

Tieto funkcie sú základom programu a sú použité pri oboch algoritmoch, kedy sa vygenerujú náhodné mestá pomocou funkcie generuj_mesta, následne funkcia vypocet_vzdialenosti na výpočet vzdialenosti medzi mestami a funkcia celkova_dlzka, ktorá počíta celkovú dĺžku trasy.

```python
#generovanie nahodnych mesteciek
def generuj_mesta(pocet_miest=POCET_MIEST):  1 usage
    return[(random.randint( a: 50, canvas_width - 50), random.randint( a: 50, canvas_height - 50)) for i in range(pocet_miest)]

#vypocet vzdialenosti medzi vsetkymi mestami
def vypocet_vzdialenost(mesta):  1 usage
    n = len(mesta)
    vzdialenosti = [[0] * n for i in range(n)]
    for i in range(n):
        for j in range(n):
            vzdialenosti[i][j] = math.sqrt((mesta[i][0] - mesta[j][0]) ** 2 + (mesta[i][1] - mesta[j][1]) ** 2)
    return vzdialenosti

#vypocet celkovej dlzky trasy
def celkova_dlzka(trasa, vzdialenosti):  3 usages
    return sum(vzdialenosti[trasa[i]][trasa[i+1]]
               for i in range (len(trasa) - 1)) + vzdialenosti[trasa[-1]][trasa[0]]
```

Mestá sú reprezentované ako dvojice súradníc (x, y), ktoré určujú ich polohu na plátne. Tieto súradnice sú náhodne generované v danom rozsahu plátna. **Trasa** je reprezentovaná ako zoznam celých čísel, kde každé číslo predstavuje index mesta v zozname miest. Trasa teda určuje poradie, v akom sa mestá navštevujú. **Vzdialenosti medzi mestami** sú uložené v dvojrozmernom zozname, kde každá položka [i][j] obsahuje Euklidovskú vzdialenosť medzi mestami i a j.

Oba algoritmy používajú rovnakú funkciu kresli na vykresľovanie a aktualizáciu grafu, ktorý spája mestá (reprezentované štvorcami) na plátne.

```python
#aktualizacia vykreslovania
def kresli(mesta, trasa):  1 usage
    canvas.delete("cesta")
    for i in range(len(trasa) - 1):
        canvas.create_line(mesta[trasa[i]][0], mesta[trasa[i]][1],
                           mesta[trasa[i + 1]][0], mesta[trasa[i + 1]][1],
                           fill="blue", width=2, tags="cesta")
    canvas.create_line(mesta[trasa[-1]][0], mesta[trasa[-1]][1],
                       mesta[trasa[0]][0], mesta[trasa[0]][1],
                       fill="blue", width=2, tags="cesta")

    for index, mesto in enumerate(mesta):
        x, y = mesto
        canvas.create_text(x, y - 15, text=f"{chr(65 + index)}")
    root.update()
```

Pre optimalizáciu trasy v probléme obchodného cestujúceho som použila funkciu two_opt_swap, ktorá implementuje 2_opt algoritmus na výmenu miest. Táto funkcia mení poradie miest (susedov) v trase, aby sa zvýšili šance pre nájdenie kratšej cesty. Funguje na princípe, že sa trasa rozdelí na tri časti, kedy sa druhá časť prevráti a následne sa spoja všetky naspäť s tým, že druhá je prevrátená.

```python
#vymena miest v trase
def two_opt_swap(trasa, i, j):  1 usage
    nova_trasa = trasa[:i] + trasa[i:j+1][::-1] + trasa[j+1:]
    return nova_trasa
```

Tabu search *(zakázané prehľadávanie)*

Implementácia algoritmu

Úlohou algoritmu tabu search je nájsť optimálnu cestu pre TSP. Na začiatku si vygeneruje počiatočnú náhodnú trasu medzi mestami a následne skúša jej vylepšenia pomocou funkcie two_opt_swap , tak, že vymení susedov resp. poradie miest v trase. Riešenia, ktoré už boli vyskúšané a nevyhovujú sa ukladajú do tabu_mnozina , aby neboli opakovane použité. Počas iterácií algoritmus uchováva najlepšie nájdené riešenie, ale umožňuje aj prechod na horšie riešenia, aby sa vyhol uviaznutiu v lokálnych minimách. Tento proces sa opakuje, až kým sa nevyčerpajú iterácie, nenájde sa lepšia trasa alebo sa dosiahne limit bez zlepšenia.

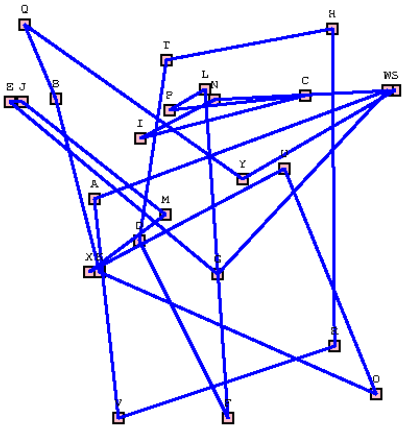
```python
#tabu search
def tabu_search(mesta, max_iteracii = ITERACIE, tabu_velkost = TABU_SIZE, pauza = CAS, iteracie_bez_zmeny = 1):  1 usage
    vzialenosti = vypocet_vzdialenost(mesta)
    aktualne_riesenie = list(range(len(mesta)))
    random.shuffle(aktualne_riesenie)
    najlepsie_riesenie = deepcopy(aktualne_riesenie)
    najlepsia_cena = celkova_dlzka(najlepsie_riesenie, vzialenosti)
    tabu_mnozina = set()

    pocet_bez_zmeny = 0

    for iteracia in range(max_iteracii):
        susedia = []

        #generovanie susednych miest pomocou 2-opt swap
        for i in range(len(aktualne_riesenie)):
            for j in range(i+1, len(aktualne_riesenie)):
                sused = two_opt_swap(aktualne_riesenie, i, j)
                cena = celkova_dlzka(sused, vzialenosti)
                if (i,j) not in tabu_mnozina or cena < najlepsia_cena:
                    susedia.append((sused, cena))

        if not susedia:
            break

        #vyber najlepsieho suseda
        najlepsi_sused = min(susedia, key = lambda x: x[1])
        aktualne_riesenie = najlepsi_sused[0]

        #aktualizácia tabu zoznamu
        tabu_mnozina.add((aktualne_riesenie.index(najlepsi_sused[0][0]), aktualne_riesenie.index(najlepsi_sused[0][1])))
        if len(tabu_mnozina) > tabu_velkost:
            tabu_mnozina.pop()

        aktualna_cena = najlepsi_sused[1]
        if aktualna_cena < najlepsia_cena:
            najlepsie_riesenie = deepcopy(aktualne_riesenie)
            najlepsia_cena = aktualna_cena
            pocet_bez_zmeny = 0
        else:
            pocet_bez_zmeny += 1

        #koniec ak sa ta ista iteracia X-ty krat opakuje
        if pocet_bez_zmeny >= iteracie_bez_zmeny:
            break

        #zobrazovanie priebeznych vysledkov
        print(f"iteracia {iteracia + 1}: dlzka trasy = {aktualna_cena:.2f} km")
        kresli(mesta, aktualne_riesenie)
        time.sleep(pauza)

    return najlepsie_riesenie, najlepsia_cena
```

Riešenie TSP pomocou Tabu Search

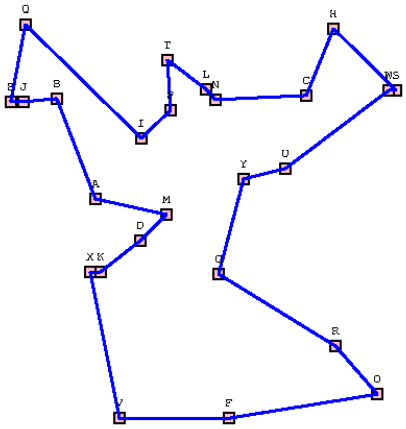
Táto funkcia slúži na riešenie TSP pomocou Tabu Search algoritmu. Ak sú vygenerované mestá, tak sa zavolá tabu search na vyhľadanie najlepšej (najlacnejšej) cesty. Priebežne sa vypisujú iterácie a dĺžka aktuálnej trasy, po uplynutí iterácií/nájdení cesty, sa vypíše aj najlepšia možná cesta.

```python
# TSP pomocou tabu search
def tsp_tabu(mesta):  1 usage
    if not mesta:
        return
    najlepsia_trasa, najlepsia_vzdialenost = tabu_search(mesta)
    print(f"\nKonečná dĺžka trasy {najlepsia_vzdialenost:.2f} km")

    # Výpis súradníc
    print("\nTrasa ide cez mestá:")
    for i in range(len(najlepsia_trasa)):
        mesto_index = najlepsia_trasa[i]
        dalsie_mesto_index = najlepsia_trasa[(i + 1) % len(najlepsia_trasa)]
        mesto = mesta[mesto_index]
        dalsie_mesto = mesta[dalsie_mesto_index]
        vzdialenost = math.sqrt((mesto[0] - dalsie_mesto[0]) ** 2 + (mesto[1] - dalsie_mesto[1]) ** 2)

        # Získanie písmenka pre aktuálne mesto a ďalšie mesto
        mesto_pismeno = chr(65 + mesto_index)  # A, B, C, ...
        dalsie_mesto_pismeno = chr(65 + dalsie_mesto_index)  # A, B, C, ...

        print(f"{mesto_pismeno} {mesto} --> {dalsie_mesto_pismeno} {dalsie_mesto} = {vzdialenost:.2f} km")

    # Zobrazenie trasy na plátne
    for i in range(len(najlepsia_trasa) - 1):
        canvas.create_line(mesta[najlepsia_trasa[i]][0], mesta[najlepsia_trasa[i]][1],
                           mesta[najlepsia_trasa[i + 1]][0], mesta[najlepsia_trasa[i + 1]][1],
                           fill="blue", width=2)

    # Prepojenie späť na začiatok
    canvas.create_line(mesta[najlepsia_trasa[-1]][0], mesta[najlepsia_trasa[-1]][1],
                       mesta[najlepsia_trasa[0]][0], mesta[najlepsia_trasa[0]][1], fill="blue", width=2)
```

Spustenie programu:



Obrázok 1: pôvodná trasa pre 25 miest



Obrázok 2: finálna trasa pre 25 miest po 21 iteráciách

Simulated annealing *(simulované žíhanie)*

Implementácia algoritmu

Algoritmus začína náhodným riešením, ktoré predstavuje poradie, v akom sa navštívia mestá. Na začiatku sa vypočíta celková dĺžka trasy a každá iterácia sa pokúša toto riešenie vylepšiť. Použitím techniky **2-opt swap** sa náhodne vyberú dve mestá a ich poradie sa v trase vymení. Ak je nové riešenie lepšie, automaticky sa prijme; ak je horšie, prijatie závisí od aktuálnej teploty a rozdielu medzi riešeniami. Teplota sa postupne znižuje v každej iterácii, čím sa znižuje pravdepodobnosť prijatia horších riešení, čím algoritmus simuluje proces ochladzovania kovu. Algoritmus sa zastaví po dosiahnutí stanoveného počtu iterácií alebo ak sa riešenie dlhšiu dobu nezlepšuje.

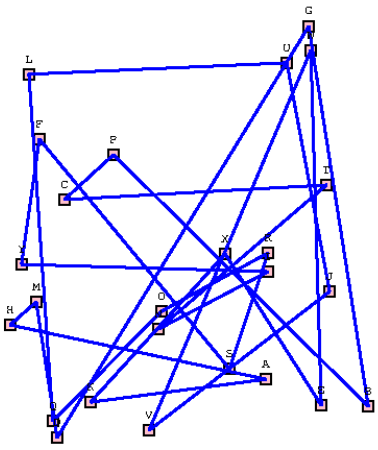
```python
# Simulated Annealing algoritmus
def simulated_annealing(mesta, start_temp=START_TEMP, end_temp=END_TEMP, alpha=ALPHA, max_iter=ITERACIE):  1 usage
    vzdialenosti = vypocet_vzdialenost(mesta)
    aktualne_riesenie = list(range(len(mesta)))
    random.shuffle(aktualne_riesenie)
    najlepsie_riesenie = aktualne_riesenie[:]
    najlepsia_cena = celkova_dlzka(najlepsie_riesenie, vzdialenosti)

    teplota = start_temp
    aktualna_cena = celkova_dlzka(aktualne_riesenie, vzdialenosti)
    max_iteracie_bez_zlepsenia = 1000
    iteracie_bez_zlepsenia = 0

    for iteracia in range(max_iter):
        # Generovanie susedného riešenia
        i, j = random.sample(range(len(aktualne_riesenie)),  k: 2)

        # Použitie 2-opt swap
        novy_sused = two_opt_swap(aktualne_riesenie, i, j)
        nova_cena = celkova_dlzka(novy_sused, vzdialenosti)

        # Ak je nové riešenie lepšie, akceptujeme ho
        if nova_cena < aktualna_cena or random.random() < math.exp((aktualna_cena - nova_cena) / teplota):
            aktualne_riesenie = novy_sused
            aktualna_cena = nova_cena
            if aktualna_cena < najlepsia_cena:
                najlepsie_riesenie = aktualne_riesenie[:]
                najlepsia_cena = aktualna_cena
                iteracie_bez_zlepsenia = 0
            else:
                iteracie_bez_zlepsenia += 1
        else:
            iteracie_bez_zlepsenia += 1

        # Znižovanie teploty
        teplota *= alpha

        # Zobrazovanie priebežných výsledkov
        if iteracia % 10 == 0:  # Zobrazujeme každých 10 iterácií
            print(f"Iterácia {iteracia + 1}: dĺžka trasy = {najlepsia_cena:.2f} km, teplota: {teplota:.2f}")
            kresli(mesta, aktualne_riesenie)
            time.sleep(CAS)

        # zastavi sa, ak iteracie dosiahnu maximum bez zlepsenia
        if iteracie_bez_zlepsenia >= max_iteracie_bez_zlepsenia:
            break

    return najlepsie_riesenie, najlepsia_cena
```

Riešenie TSP pomocou Simulated Annealing

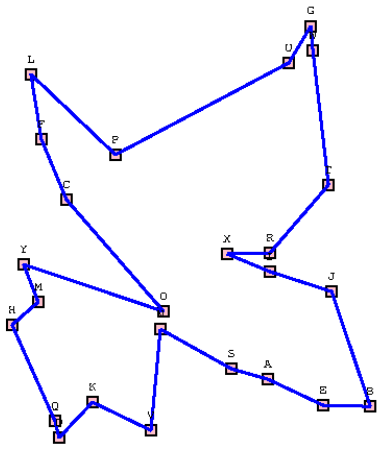
Táto funkcia implementuje algoritmus simulovaného žíhania na riešenie TSP. Ak sú vygenerované mestá, tak sa zavolá SA na vyhľadanie najlepšej cesty. Priebežne sa vypisuje dĺžka trasy a teplota, po dosiahnutí minimálnej teploty a uplynutí iterácií sa vypíše a vykreslí najlepšia možná cesta.

```python
# TSP s pouzitim simulated annealing
def tsp_simulated_annealing(mesta):  1 usage
    if not mesta:
        return
    najlepsia_trasa, najlepsia_vzdialenost = simulated_annealing(mesta)
    print(f"\nKonečná dĺžka trasy {najlepsia_vzdialenost:.2f} km")

    # Výpis súradníc
    print("\nTrasa ide cez mestá:")
    for i in range(len(najlepsia_trasa)):
        mesto_index = najlepsia_trasa[i]
        dalsie_mesto_index = najlepsia_trasa[(i + 1) % len(najlepsia_trasa)]
        mesto = mesta[mesto_index]
        dalsie_mesto = mesta[dalsie_mesto_index]
        vzdialenost = math.sqrt((mesto[0] - dalsie_mesto[0]) ** 2 + (mesto[1] - dalsie_mesto[1]) ** 2)

        mesto_pismeno = chr(65 + mesto_index)
        dalsie_mesto_pismeno = chr(65 + dalsie_mesto_index)

        print(f"{mesto_pismeno} {mesto} --> {dalsie_mesto_pismeno} {dalsie_mesto} = {vzdialenost:.2f} km")

    # Zobrazenie trasy na plátne
    for i in range(len(najlepsia_trasa) - 1):
        canvas.create_line(mesta[najlepsia_trasa[i]][0], mesta[najlepsia_trasa[i]][1],
                           mesta[najlepsia_trasa[i + 1]][0], mesta[najlepsia_trasa[i + 1]][1],
                           fill="blue", width=2)

    # Prepojenie späť na začiatok
    canvas.create_line(mesta[najlepsia_trasa[-1]][0], mesta[najlepsia_trasa[-1]][1],
                       mesta[najlepsia_trasa[0]][0], mesta[najlepsia_trasa[0]][1], fill="blue", width=2)
```

Spustenie programu:



Obrázok 3:trasa na začiatku



Obrázok 4: trasa na konci

Testovanie

Obe riešenia som testovala s rôznymi počtami miest a s rôznym počtom iterácií. Vytvorené riešenia fungujú optimálne pre oblasti menšie alebo rovné ako 30 miest a potom treba zvýšiť iterácie, aby sme dospeli k ideálnemu výsledku. Tiež som pri testovaní zistila, že zakázané prehľadávanie potrebuje oveľa menej iterácií ako simulované žíhanie.