**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №2**

**по дисциплине «Построение и анализ алгоритмов»**

Тема: Коммивояжёр

|  |  |  |
| --- | --- | --- |
| Студент гр. 3344 |  | Тукалкин В.А. |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург

2025

**Цель работы**

Решить задачу коммивояжёра.

**Задание**

Решить ЗК двумя методами в соответствии с вариантом: 1) Методом ВиГ. 2) Приближённым методом.

Дано: матрица весов графа, все веса неотрицательны; стартовая вершина.

Найти: путь коммивояжёра (последовательность вершин) и его стоимость.

**Вариант 3**

3. МВиГ: последовательный рост пути + использование для отсечения двух нижних оценок ве́са оставшегося пути: 1) полусуммы весов двух легчайших рёбер по всем кускам; 2) веса МОД. Эвристика выбора дуги — поиск в глубину с учётом веса добавляемой дуги и нижней оценки веса остатка пути.

Приближённый алгоритм: АМР.

Замечание к варианту 3. Начинать МВиГ со стартовой вершины.

Независимо от варианта, при сдаче работы должна быть возможность генерировать матрицу весов (произвольную или симметричную; для варианта 6 — симметричную), сохранять её в файл и использовать в качестве входных данных.

**Описание алгоритма**

Задача коммивояжёра (TSP) формулируется следующим образом:

Дан полный взвешенный граф G=(V,E), где V — множество вершин (городов), а E — множество ребер, каждому из которых сопоставлен вес (расстояние или стоимость перемещения между городами).

Цель задачи — найти такой гамильтонов цикл (маршрут, проходящий через каждую вершину ровно один раз и возвращающийся в исходную), который имеет минимальный суммарный вес.

Задача коммивояжёра является NP-полной, что означает, что не существует известного алгоритма, который может решить её за полиномиальное время для всех случаев.

Количество возможных маршрутов в полном графе с n городами равно (n-1)! / 2 (для симметричной матрицы расстояний).

В худшем случае рекурсивный перебор всех вариантов расширения цепочки, соответствует перебору всех перестановок вершин, что имеет сложность O(n!) по времени и O(n!) по памяти, где n — количество вершин графа.

Однако, на практике применение нижних оценок и отсечений значительно сокращает число исследуемых вариантов.

Отсечение ветвей:

При каждом выборе следующей вершины вычисляется нижняя оценка остатка пути. Если сумма уже накопленного веса и нижней оценки не лучше найденного решения, ветвь отсекается. Это позволяет не рассматривать факториальное число вариантов и существенно уменьшает число рекурсивных вызовов.

При вычислении нижних оценок учитываются только такие дуг, которые потенциально могут быть добавлены к решению (допустимые). Нижняя оценка считается на основе:

1) МОД в качестве "вершин" при построении МОД используются куски, при этом в это МОД могут входить только допустимые рёбра (соответствующие допустимым дугам). МОД строится по алгоритму Краскала.

2) Полусуммы весов двух легчайших рёбер: для каждого куска берётся вес легчайшей допустимой дуги, входящей в начальную вершину куска, и вес легчайшей допустимой дуги, исходящей из конечной вершины куска. Таким образом, для m кусков получаем 2m чисел. Все эти веса складываются, сумма делится на 2 — это и будет нижняя оценка на основе полусуммы весов двух легчайших рёбер по всем кускам.

Эвристический выбор кандидатов:

Кандидаты сортируются по сумме s+L (где s — вес дуги, а L — нижняя оценка остатка пути), что позволяет в первую очередь рассматривать более перспективные варианты и, как следствие, находить хорошее начальное решение для последующих отсечений.

Таким образом, в теории, в худшем случае алгоритм имеет факториальную сложность O(n!) по времени и O(n!) по памяти, однако на практике жадные эвристики и эффективные отсечения позволяют добиться значительно меньшей средней сложности, что делает алгоритм применимым для умеренных значений n.

Приближённый алгоритм (АМР):

Алгоритм модификации решения перебирает перестановки вершин в найденном маршруте. Число таких перестановок зависит от n и в худшем случае составляет O(n2) по времени и O(n2) по памяти, что по сравнению с O(n!) не является основным фактором, также применяется простая эвристика выбор городов по вкладу в стоимость. Для каждого города вычисляется сумма весов входящего и исходящего рёбер. Города сортируются по убыванию этой суммы. Это позволяет сначала оптимизировать наиболее «дорогие» участки маршрута.

Основной процесс:

1) Пользователь выбирает режим работы: ручной ввод матрицы или автоматическая генерация.

2) При генерации матрицы весов пользователь может выбрать симметричный или произвольный вариант, а также сохранить матрицу в файл.

3) При необходимости пользователь может загрузить матрицу весов из файла.

4) Создаётся объект класса tspSolver, которому передаётся матрица весов. Вычисляются два минимальных ребра для каждой вершины, для дальнейшей оценки в методе lowerBound.

5) Вызывается метод solve, который:

* Запускает рекурсивный поиск с возвратом (метод extendChain), используя эвристику s+L для выбора следующей вершины.
* Применяет метод lowerBound, в котором считается нижняя оценка на основе полусуммы минимальных двух рёбер вершин и МОД.
* Применяет отсечения ветвей, если сумма текущего веса и нижней оценки не лучше найденного маршрута.

6) После получения первого решения выполняется приближённый алгоритм (АМР) для получения второго решения:

* Создаётся объект класса tspAMR, которому передаётся матрица весов.
* На каждой итерации вычисляется вклад каждого города в общий вес. Города сортируются по убыванию вклада, ищется решение перебором городов по убыванию вклада. Лучшее улучшение применяется в конце итерации. Вклад = сумма весов входящего и исходящего рёбер.
* Останавливается, если было достигнуто максимальное число итераций или не найдено улучшений на текущей итерации.

7) Итоговые решения (наименьшие найденные маршруты) выводятся на экран вместе с их весами.

**Описание функций и структур данных**

Структура класса tspAMR:

Атрибуты:

* weightMatrix – матрица смежности графа.
* n – количество вершин.
* bestSolution – лучшая найденная цепочка (маршрут).
* bestWeight – вес лучшего маршрута (изначально бесконечность).

Методы:

* calcSolutionWeight(sol) – вычисляет суммарный вес заданной цепочки.

Параметры:

sol - список вершин, представляющий полный маршрут (с возвратом в начальную вершину) (List[int]).

Возвращает: int.

* \_calculateCityContribution(solution) – Рассчитывает вклад каждого города в общий вес маршрута. Вклад = сумма весов входящего и исходящего рёбер.

Параметры:

solution - список вершин, представляющий полный маршрут (с возвратом в начальную вершину) (List[int]).

Возвращает: Dict(int: int)

* amrModification(start, solution) – приближенный алгоритм модификации решения (АМР) с эвристикой для ускорения.

Параметры:

start - индекс стартовой вершины (по умолчанию 0) (int).

solution – начальное решение (если не указано, генерируется автоматически) (List[int]).

Возвращает: tuple(List[int], int).

Структура класса tspSolver:

Атрибуты:

* weightMatrix – матрица смежности графа.
* n – количество вершин.
* bestSolution – лучшая найденная цепочка (маршрут).
* bestWeight – вес лучшего маршрута (изначально бесконечность).
* minTwo – список пар двух наименьших весов исходящих дуг для каждой вершины (используется при вычислении нижней оценки).

Методы:

* solve(start) – основной метод, который запускает рекурсивное расширение пути.

Параметры:

start - индекс стартовой вершины (по умолчанию 0) (int).

Возвращает: tuple(List[int], int).

* extendChain(chain, currentWeight, visited) – рекурсивная функция, которая расширяет текущий путь, добавляя к нему непосещённые вершины. Перед переходом проверяется нижняя оценка остатка пути. Если сумма текущего веса и оценки превышает лучший найденный результат, происходит отсечение ветви.

Параметры:

chain – текущая цепочка (список индексов вершин) (List[int])

currentWeight – суммарный вес текущей цепочки (int)

visited – список, где True означает, что вершина уже добавлена в цепочку (List[int])

Возвращает: None

* markVisited(visited, v) – возвращает копию списка visited, где вершина v отмечена как посещённая. Используется для вычисления нижней оценки без изменения текущего состояния visited.

Параметры:

visited – список посещенных вершин (List[int])

v – индекс вершины, которая должна быть посещена (int)

Возвращает: List[int]

* lowerBound(chain, currentWeight, visited) – вычисляет нижнюю оценку для непосещённых вершин с использованием двух методов (полусумма минимальных дуг и МОД через алгоритм Прима (графа считается неориентированным и берётся минимальный вес пути)).

Параметры:

chain – текущая цепочка (список индексов вершин) (List[int])

currentWeight – суммарный вес текущей цепочки (int)

visited – список посещенных вершин (List[int])

Возвращает: int

* mstWeight(chain, visited) – вычисляет вес минимального остовного дерева для подмножества непосещённых вершин (алгоритм Краскала).

Параметры:

chain – текущая цепочка (список индексов вершин) (List[int])

vertices - список вершин, для которых строим МОД (List[int])

Возвращает: int

Функции для работы с матрицей весов:

* generateWeightMatrix(n, symmetric) – генерирует матрицу весов для графа с заданным числом вершин. При выборе симметричного режима матрица заполняется одинаковыми значениями для пар вершин (i,j) и (j,i).

Параметры:

n – количество вершин графа (int)

symmetric – флаг симметричности (bool)

Возвращает: List[List[int]]

* saveMatrixToFile(matrix, filename) – сохраняет матрицу весов в файл в формате JSON.

Параметры:

matrix – двумерный массив, в котором записан граф (List[List[int]])

filename – имя файла куда будет сохранено (str)

* loadMatrixFromFile(filename) – загружает матрицу весов из файла.

Параметры:

filename – имя файла откуда будет загружено (str)

Возвращает: List[List[int]]

**Тестирование**

Таблица 1 – Результаты тестирования для МВиГ

|  |  |  |
| --- | --- | --- |
| № | Входные данные | Выходные данные |
| 1 | 3  -1 1 3  3 -1 1  1 2 -1 | Лучшее найденное решение: 0 1 2 0  Вес решения: 3 |
| 2 | 4  -1 3 4 1  1 -1 3 4  9 2 -1 4  8 9 2 -1 | Лучшее найденное решение: 0 3 2 1 0  Вес решения: 6 |
| 3 | 5  0 13 5 2 8  13 0 13 11 17  5 13 0 8 20  2 11 8 0 19  8 17 20 19 0 | Лучшее найденное решение: 0 3 2 1 4 0  Вес решения: 48 |
| 4 | 10  0 14 9 19 11 8 8 14 5 13  13 0 13 7 13 6 5 16 11 17  4 5 0 19 6 15 19 10 2 15  20 5 9 0 12 17 9 3 5 13  7 4 20 4 0 11 18 7 16 16  11 3 20 2 17 0 12 18 12 18  20 3 20 4 16 1 0 3 14 7  14 17 6 12 12 5 11 0 17 20  13 14 18 1 10 16 18 11 0 2  3 12 5 12 15 16 13 14 12 0 | Лучшее найденное решение: 0 4 1 6 5 3 7 2 8 9 0  Вес решения: 39 |

Таблица 2 – Результаты тестирования для АМР

|  |  |  |
| --- | --- | --- |
| № | Входные данные | Выходные данные |
| 1 | 3  -1 1 3  3 -1 1  1 2 -1 | Лучшее найденное решение для АМР: 0 1 2 0  Вес решения для АМР: 3 |
| 2 | 4  -1 3 4 1  1 -1 3 4  9 2 -1 4  8 9 2 -1 | Лучшее найденное решение для АМР: 0 1 2 3 0  Вес решения для АМР: 18 |
| 3 | 5  0 13 5 2 8  13 0 13 11 17  5 13 0 8 20  2 11 8 0 19  8 17 20 19 0 | Лучшее найденное решение для АМР: 0 3 2 1 4 0  Вес решения для АМР: 48 |
| 4 | 10  0 14 9 19 11 8 8 14 5 13  13 0 13 7 13 6 5 16 11 17  4 5 0 19 6 15 19 10 2 15  20 5 9 0 12 17 9 3 5 13  7 4 20 4 0 11 18 7 16 16  11 3 20 2 17 0 12 18 12 18  20 3 20 4 16 1 0 3 14 7  14 17 6 12 12 5 11 0 17 20  13 14 18 1 10 16 18 11 0 2  3 12 5 12 15 16 13 14 12 0 | Лучшее найденное решение для АМР: 0 2 4 1 5 3 7 6 8 9 0  Вес решения для АМР: 60 |

**Вывод**

Была решена задача коммивояжёра с использованием методов ВиГ и АМР.

**Приложение А**

**ИСХОДНЫЙ КОД ПРОГРАММЫ**

Имя файла: Коммивояжёр.py

import random

import json

from PIL import Image, ImageDraw

import time

class tspAMR:

def \_\_init\_\_(self, weightMatrix):

"""

Конструктор класса.

weightMatrix - матрица смежности, где weightMatrix[i][j] — вес дуги из вершины i в вершину j.

"""

self.weightMatrix = weightMatrix # Матрица весов графа

self.n = len(weightMatrix) # Число вершин

self.bestSolution = None # Лучшая найденная цепочка (решение)

self.bestWeight = float('inf') # Вес лучшего решения (изначально бесконечность)

def calcSolutionWeight(self, sol):

"""

Вычисляет суммарный вес заданной цепочки sol.

sol - список вершин, представляющий полный маршрут (с возвратом в начальную вершину).

"""

total = 0

for i in range(len(sol) - 1):

total += self.weightMatrix[sol[i]][sol[i + 1]]

return total

def amrModification(self, start):

"""

Приближенный алгоритм модификации решения (АМР) с эвристикой для ускорения.

Цель: найти решение быстрее чем основной алгоритм.

"""

maxModifications = self.n

self.bestSolution = [x for x in range(self.n)] + [0]

self.bestWeight = self.calcSolutionWeight(self.bestSolution)

improved = True # Флаг, указывающий на наличие улучшения

modifications = 0 # Счетчик выполненных модификаций

while improved and modifications < maxModifications:

improved = False

currentSol = self.bestSolution[:-1] # Текущая цепочка без повторной стартовой вершины

print(f"\nАМР: текущая цепочка {currentSol} с весом {self.bestWeight}")

# Эвристика: рассматриваем только ближайшие соседние вершины

for i in range(1, self.n):

for delta in [-1, 1, 2]:

j = i + delta

if j < 1 or j >= self.n:

continue

newSol = currentSol.copy() # Копия текущего решения

city = newSol.pop(i) # Удаляем город с позиции i

newSol.insert(j, city) # Вставляем его в позицию j

newSolComplete = newSol + [newSol[0]] # Полная цепочка с возвратом в начало

newWeight = self.calcSolutionWeight(newSolComplete)

print(

f"Пробуем переместить город {city} с позиции {i} на позицию {j} -> {newSolComplete} с весом {newWeight}")

if newWeight < self.bestWeight:

print(f"\n>>> АМР: улучшение найдено! Новая цепочка: {newSolComplete} с весом {newWeight}")

self.bestSolution = newSolComplete

self.bestWeight = newWeight

improved = True

modifications += 1

break

if improved:

break

if start != 0:

ind = self.bestSolution.index(start)

self.bestSolution = self.bestSolution[ind:] + self.bestSolution[1:ind] + [start]

return self.bestSolution, self.bestWeight

class tspSolver:

def \_\_init\_\_(self, weightMatrix):

"""

Конструктор класса.

weightMatrix - матрица смежности, где weightMatrix[i][j] — вес дуги из вершины i в вершину j.

"""

n = len(weightMatrix)

self.weightMatrix = [[0] \* n for \_ in range(n)] # Матрица весов графа

for i in range(n):

for j in range(i + 1, n):

self.weightMatrix[i][j] = min(weightMatrix[i][j], weightMatrix[j][i])

self.weightMatrix[j][i] = min(weightMatrix[i][j], weightMatrix[j][i])

#self.weightMatrix=weightMatrix

self.n = n # Число вершин

self.bestSolution = None # Лучшая найденная цепочка (решение)

self.bestWeight = float('inf') # Вес лучшего решения (изначально бесконечность)

# Предварительный этап: для каждой вершины вычисляем два наименьших исходящих ребра.

# Это будет использоваться при расчёте нижней оценки.

self.minTwo = []

for i in range(self.n):

# Создаем список весов дуг из вершины i во все остальные (исключая петли i->i)

outs = [self.weightMatrix[i][j] for j in range(self.n) if i != j]

outs.sort() # Сортируем веса для нахождения двух минимальных

if len(outs) >= 2:

self.minTwo.append((outs[0], outs[1])) # Сохраняем два минимальных веса

elif outs:

self.minTwo.append((outs[0], outs[0])) # Если есть только одна дуга, используем её дважды

else:

self.minTwo.append((0, 0)) # На случай, если вершина не имеет исходящих дуг

print(f"Инициализация завершена. Минимальные ребра для каждой вершины (0 ... {self.n - 1}):", self.minTwo)

def solve(self, start=0):

"""

Основной метод решения задачи.

start - индекс стартовой вершины (по умолчанию 0).

"""

print("\nЗапуск алгоритма из стартовой вершины:", start)

# Создаем список для отслеживания посещённых вершин

visited = [False] \* self.n

visited[start] = True # Стартовая вершина считается посещённой

# Запускаем рекурсивное расширение цепочки, начиная со стартовой вершины

self.extendChain([start], 0, visited)

return self.bestSolution, self.bestWeight

def extendChain(self, chain, currentWeight, visited):

"""

Рекурсивная функция, которая расширяет цепочку, добавляя по одной вершине.

chain - текущая цепочка (список индексов вершин)

currentWeight - суммарный вес текущей цепочки

visited - список, где True означает, что вершина уже добавлена в цепочку

"""

print(f"\nРассматриваем цепочку {chain} с весом {currentWeight}")

# Если цепочка содержит все вершины, необходимо добавить дугу возврата в стартовую вершину

if len(chain) == self.n:

totalWeight = currentWeight + self.weightMatrix[chain[-1]][chain[0]]

print(f"Получена полная цепочка {chain + [chain[0]]} с общим весом {totalWeight}")

if totalWeight < self.bestWeight:

self.bestWeight = totalWeight

self.bestSolution = chain + [chain[0]]

print(f"\n\n>>>Новое лучшее решение: {self.bestSolution} с весом {self.bestWeight}")

return

# Вычисляем нижнюю оценку для завершения цепочки (осталось добавить)

lowerBound = self.lowerBound(chain, visited)

print(f"Нижняя оценка для цепочки {chain}: {lowerBound}, сумма с текущим весом: {currentWeight + lowerBound}")

if currentWeight + lowerBound >= self.bestWeight:

print("Отсечение ветви, так как сумма текущего веса и нижней оценки не лучше лучшего решения.")

return

# Из последней вершины текущей цепочки пробуем добавить каждую непосещённую вершину

last = chain[-1]

candidates = []

for v in range(self.n):

if not visited[v]:

s = self.weightMatrix[last][v] # Вес дуги от последней вершины к кандидату v

newChain = chain + [v] # Создаем новую цепочку с добавлением v

newWeight = currentWeight + s # Обновляем суммарный вес цепочки

lbNew = self.lowerBound(newChain, self.markVisited(visited, v))

candidates.append((s + lbNew, v, s))

print(f"Кандидат {v}: вес дуги = {s}, нижняя оценка после добавления = {lbNew}, сумма = {s + lbNew}")

# Сортируем кандидатов по сумме (s + lbNew), чтобы сначала рассмотреть более перспективные ветви

candidates.sort(key=lambda x: x[0])

for total, v, s in candidates:

print(f"Переход от {last} к {v} (вес дуги {s}, суммарная оценка {total})")

visited[v] = True # Отмечаем вершину v как посещённую

self.extendChain(chain + [v], currentWeight + s, visited) # Рекурсивно расширяем цепочку

visited[v] = False # После возврата отменяем посещение для других вариантов

def markVisited(self, visited, v):

"""

Возвращает копию списка visited, где вершина v отмечена как посещённая.

Используется для вычисления нижней оценки без изменения текущего состояния visited.

"""

newVisited = visited.copy()

newVisited[v] = True

return newVisited

def lowerBound(self, chain, visited):

"""

Вычисляет нижнюю оценку остатка пути (L) как максимум из двух оценок:

1. Оценка на основе полусуммы минимальных дуг для всех непосещённых вершин

(и для соединения с началом и концом цепочки).

2. Оценка на основе веса минимального остовного дерева (MST) для непосещённых вершин

с учетом подключения цепочки.

"""

lb1 = 0

for v in range(self.n):

if not visited[v]:

m1, m2 = self.minTwo[v]

lb1 += (m1 + m2)

# Для каждой непосещённой вершины находим минимальные допустимые дуги

for v in range(self.n):

if not visited[v]:

# Минимальная дуга, исходящая из вершины v

minOut = min([self.weightMatrix[v][j] for j in range(self.n) if not visited[j]] or [0])

# Минимальная дуга, входящая в вершину v

minIn = min([self.weightMatrix[i][v] for i in range(self.n) if not visited[i]] or [0])

lb1 += (minOut + minIn)

# Если цепочка не пуста, учитываем дуги для соединения с началом и концом цепочки

if chain:

start = chain[0]

end = chain[-1]

# Минимальная дуга, исходящая из конца цепочки

minFromEnd = min([self.weightMatrix[end][v] for v in range(self.n) if not visited[v]] or [0])

# Минимальная дуга, входящая в начало цепочки

minToStart = min([self.weightMatrix[v][start] for v in range(self.n) if not visited[v]] or [0])

lb1 += (minFromEnd + minToStart)

# Делим на 2, чтобы избежать двойного учёта

lb1 = lb1 / 2

unvisited = [v for v in range(self.n) if not visited[v]]

lb2 = 0

if unvisited:

lb2 = self.mstWeight(unvisited)

lb = max(lb1, lb2)

print(

f"\nОценка на основе полусуммы минимальных {lb1}\nОценка на основе веса минимального остовного дерева (MST): {lb2}\nМаксимальная оценка: {lb}")

return lb1

def mstWeight(self, vertices):

"""

Вычисляет вес минимального остовного дерева (MST) для подмножества вершин с использованием алгоритма Прима.

vertices - список вершин, для которых строим MST.

"""

if not vertices:

return 0

inMst = {vertices[0]}

total = 0

remaining = set(vertices[1:])

while remaining:

minEdge = float('inf')

vSelected = None

for u in inMst:

for v in remaining:

if self.weightMatrix[u][v] < minEdge:

minEdge = self.weightMatrix[u][v]

vSelected = v

if vSelected is None:

break

inMst.add(vSelected)

remaining.remove(vSelected)

total += minEdge

return total

# Функции для генерации, сохранения и загрузки матрицы весов

def generateWeightMatrix(n, symmetric):

"""

Генерирует матрицу весов для графа с n вершинами.

symmetric - если True, генерируется симметричная матрица (для неориентированного графа).

Возвращает матрицу в виде списка списков.

"""

minWeight = 1

maxWeight = 20

matrix = [[0] \* n for \_ in range(n)]

for i in range(n):

for j in range(n):

if i == j:

matrix[i][j] = 0

else:

weight = random.randint(minWeight, maxWeight)

matrix[i][j] = weight

if symmetric:

matrix[j][i] = weight

return matrix

def saveMatrixToFile(matrix, filename):

"""

Сохраняет матрицу весов в файл в формате JSON.

"""

with open(filename, "w") as f:

json.dump(matrix, f)

print(f"Матрица весов сохранена в файл {filename}")

def loadMatrixFromFile(filename):

"""

Загружает матрицу весов из файла в формате JSON.

Возвращает матрицу в виде списка списков.

"""

with open(filename, "r") as f:

matrix = json.load(f)

return matrix

# Функции визуализации

def printMatrix(matrix):

"""

Выводит матрицу весов в удобном для чтения формате.

"""

for row in matrix:

print(" ".join(str(x) for x in row))

def visualizeTSP(solution):

"""

Визуализирует найденное решение задачи коммивояжёра.

"""

size = 500

margin = 50

if not solution or len(solution) < 2:

print("Ошибка: Невозможно визуализировать пустой маршрут!")

return

n = len(solution) - 1 # Количество вершин (без учёта возврата в начальную)

random.seed(42) # Фиксируем случайность для одинаковых результатов

# Генерируем случайные координаты для каждой вершины

positions = {i: (random.randint(margin, size - margin), random.randint(margin, size - margin)) for i in range(n)}

# Создаём изображение

img = Image.new("RGB", (size, size), "white")

draw = ImageDraw.Draw(img)

# Рисуем рёбра маршрута

for i in range(n):

start = positions[solution[i]]

end = positions[solution[i + 1]]

draw.line([start, end], fill="blue", width=3)

# Рисуем вершины

for i, (x, y) in positions.items():

draw.ellipse([x - 5, y - 5, x + 5, y + 5], fill="red", outline="black")

draw.text((x + 8, y - 8), str(i), fill="black")

# Отображаем изображение

img.show()

def test(matrix):

arr = []

for start in range(len(matrix)):

solver = tspSolver(matrix)

bestPath, bestCost = solver.solve(start)

arr.append((bestPath, bestCost))

for i in range(len(arr)): print(f"{i}: {arr[i]}")

# Основной блок программы

if \_\_name\_\_ == "\_\_main\_\_":

print("Выберите режим работы:")

print("1 - Ввести матрицу весов вручную")

print("2 - Сгенерировать матрицу весов автоматически")

print("3 - Использовать матрицу из файла weightMatrix.json")

mode = input("Ваш выбор (1-3): ")

if mode.strip() == "1":

n = int(input("Введите число вершин: "))

print("Введите матрицу весов (по одной строке, элементы через пробел):")

graph = []

for i in range(n):

row = [int(x) for x in input().split()]

graph.append(row)

elif mode.strip() == "2":

n = int(input("Введите число вершин: "))

symmetricInput = input("Генерировать симметричную матрицу? (y/n): ").strip().lower()

symmetric = symmetricInput == "y"

graph = generateWeightMatrix(n, symmetric)

print("\nСгенерированная матрица весов:")

printMatrix(graph)

saveChoice = input("Сохранить матрицу в файл? (y/n): ").strip().lower()

if saveChoice == "y":

saveMatrixToFile(graph, "weightMatrix.json")

elif mode.strip() == "3":

graph = loadMatrixFromFile("weightMatrix.json")

print("\nЗагруженная матрица весов:")

printMatrix(graph)

else:

print("Неверный выбор. Завершение программы.")

exit(1)

#test(graph)

start = int(input("Введите стартовую вершину: "))

# Решение задачи TSP

print("\n\n>>>Начало решения задачи коммивояжера<<<")

t = time.time()

solver = tspSolver(graph)

bestPath, bestCost = solver.solve(start)

# После нахождения первого решения переходим к фазе модификации (АМР)

print("\n\n>>>Начало фазы модификации (АМР)<<<")

t1 = time.time()

solver2 = tspAMR(graph)

bestPathAmr, bestCostAmr = solver2.amrModification(start)

print("\nЛучшее найденное решение:", \*bestPath)

print("Вес решения:", bestCost)

print("\nЛучшее найденное решение для АМР:", \*bestPathAmr)

print("Вес решения для АМР:", bestCostAmr)

print(f"Время работы МВиГ, АМР: {time.time() - t}, {time.time() - t1}")

visualizeTSP(bestPath)