МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ

ФЕДЕРАЦИИ

Федеральное государственное автономное

образовательное учреждение высшего образования

«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Межинститутская базовая кафедра

«Исследование методов поиска в пространстве состояний»

Отчет по лабораторной работе №1

по дисциплине «Конструирование программного обеспечения для систем искусственного интеллекта»

Выполнил студент группы ПИЖ-б-о-21-1
Ключникова Я. Д. « » 2024г.
Подпись студента_____
Работа защищена « »_____2024г.
Проверила Воронкин Р.А. _____
(подпись)

Ставрополь 2024

Цель: приобретение навыков по работе с методами поиска в пространстве состояний с помощью языка программирования Python версии 3.x

Задание:

1. Изучить теоретический материал работы.

2. Создать общедоступный репозиторий на GitHub, в котором будет использована лицензия MIT и язык программирования Python.

3. Выполните клонирование созданного репозитория.

4. Дополните файл .gitignore необходимыми правилами для работы с IDE PyCharm либо Visual Studio Code.

5. Создайте проект Python в папке репозитория.

6. Проработайте примеры лабораторной работы. Создайте для каждого примера отдельный модуль языка Python. Зафиксируйте изменения в репозитории.

7. Воспользуйтесь сервисом Google Maps или аналогичными картографическими приложениями, чтобы выбрать на карте более 20 населённых пунктов, связанных между собой дорогами. Постройте граф, где узлы будут представлять населённые пункты, а рёбра – дороги, соединяющие их. Вес каждого ребра соответствует расстоянию между этими пунктами. Выберите начальный и конечный пункты на графе. Определите минимальный маршрут между ними, который должен проходить через три промежуточных населённых пункта. Покажите данный путь на построенном графе.

8. Методом полного перебора решите [задачу коммивояжёра] для начального населенного пункта на построенном графе. Оцените время решения данной задачи, если оно окажется достаточно большим, уменьшите количество узлов и ребер графа. Покажите полученный путь на построенном графе.

9. Зафиксируйте сделанные изменения в репозитории.

10. Добавьте отчет по лабораторной работе в формате PDF в папку doc репозитория. Зафиксируйте изменения.

11. Выполните слияние ветки для разработки с веткой main / master.

12. Отправьте сделанные изменения на сервер GitHub.

13. Отправьте адрес репозитория GitHub на электронный адрес преподавателя

Ход работы:

Ссылка на GitHab: https://github.com/Yana-Kh/SAI-Lab-1

Было разработано решение задачи коммивояжера, при имеющейся матрице расстояний между городами, при условии, что есть пути связывающие каждые 2 отдельных города.

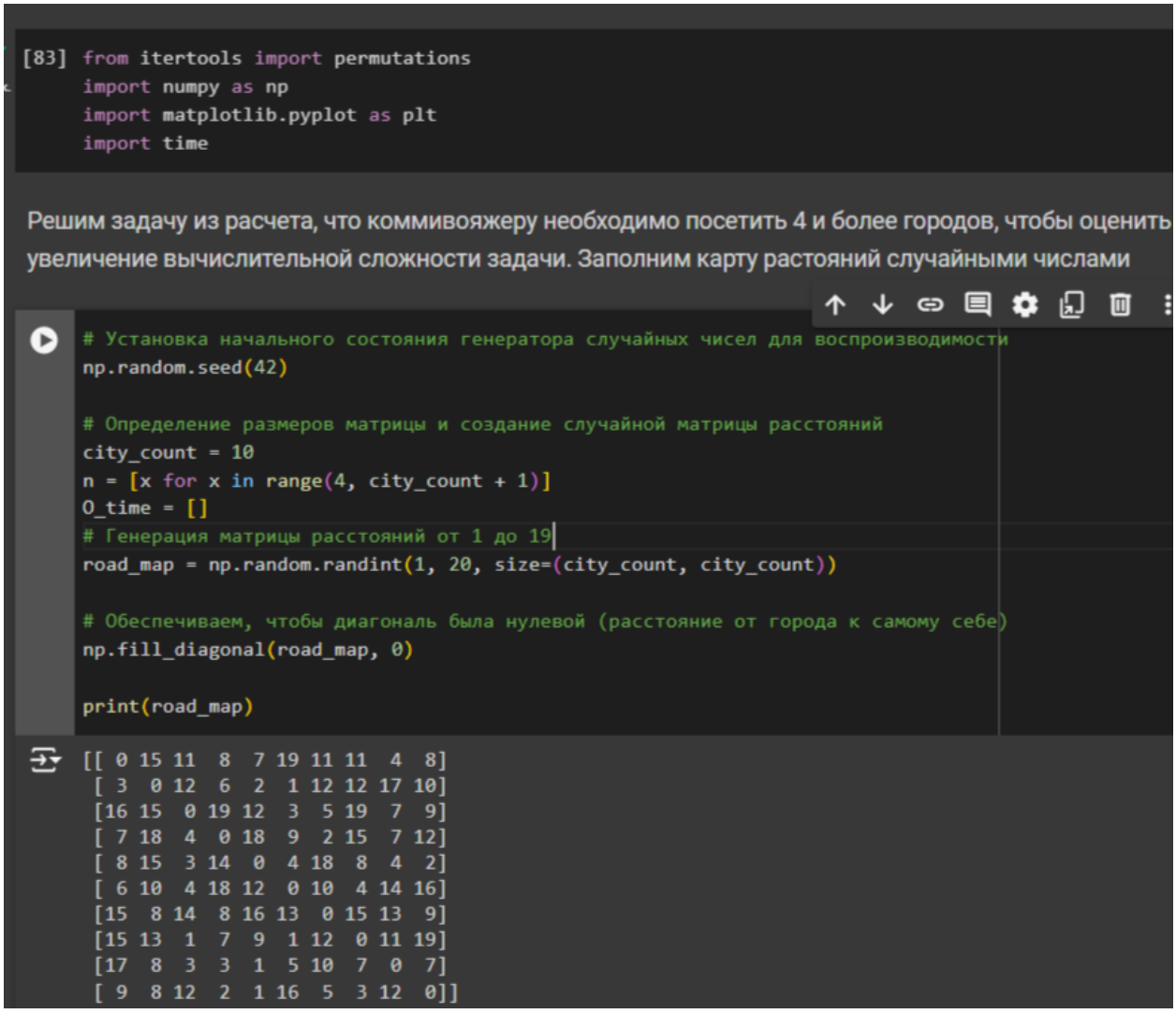


Рисунок 1 – Подготовительный этап

```
[57] #Функция, которая вычисляет длину дороги по заданному маршруту и карте
     def calculate_distance(route, distance_matrix):
         total_distance = 0
         for i in range(len(route) - 1):
             total_distance += distance_matrix[route[i], route[i + 1]]
         # добавляем расстояние обратно в стартовый город
         total_distance += distance_matrix[route[-1]][route[0]]
         return total_distance
```

```
     #Функция, которая ищет самый короткий маршрут по карте и количеству городов
     def traveling_salesman(distance_matrix, n):
         shortest_route = None
         min_distance = 100

         # Генерируем все возможные маршруты
         for perm in permutations(range(n)):
             #print("Perm: ", perm)
             current_distance = calculate_distance(perm, distance_matrix)
             if current_distance < min_distance:
                 min_distance = current_distance
                 shortest_route = perm

         return shortest_route, min_distance
```

Рисунок 2 – Блок функций

```
#Вывод информации о маршрутах
for i in n:
  start = time.time()
  route, distance = traveling_salesman(road_map, i)
  finish = time.time()
  execution_time = finish - start
  O_time.append(execution_time)
  print("При количестве городов:", i)
  print("Кратчайший маршрут:", route)
  print("Минимальное расстояние:", distance)
  print("Время выполнения:", finish - start)
  print("-------------------------")
```

Рисунок 3 – Блок вычислений

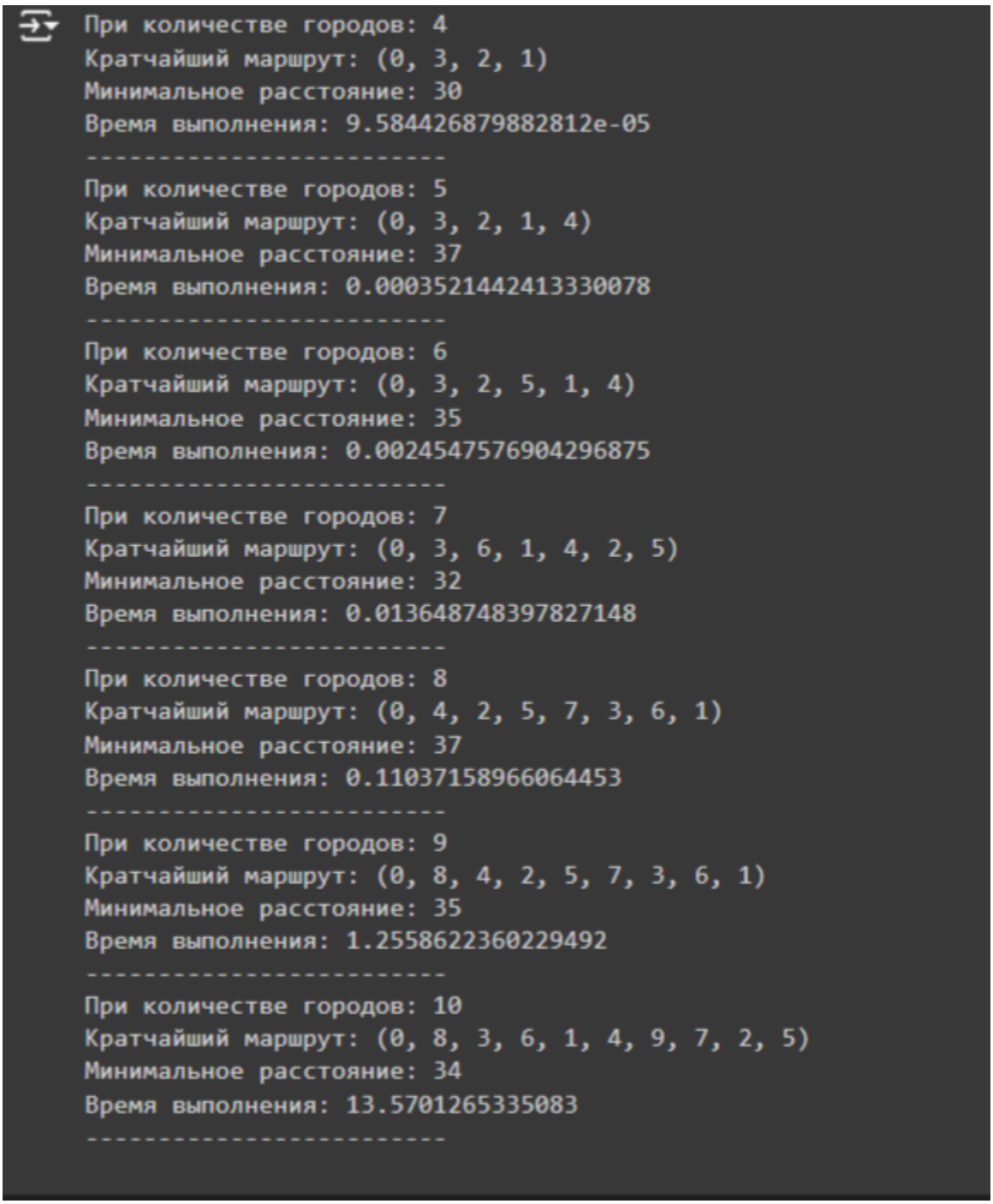
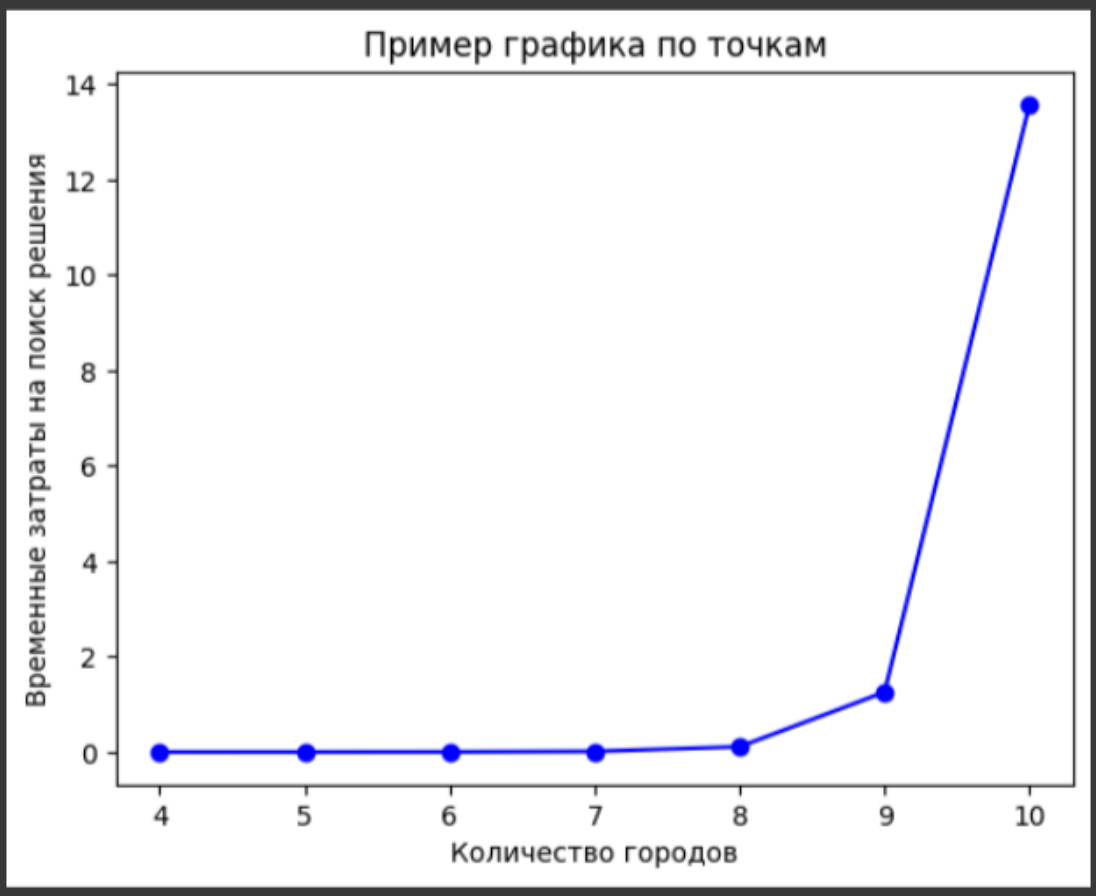


Рисунок 4 – Блок вывода результатов

```
# Построение графика
plt.plot(n, O_time, marker='o', color='b')  # 'o' - маркер точек
plt.title('Пример графика по точкам')
plt.xlabel('Количество городов')
plt.ylabel('Временные затраты на поиск решения')

plt.show()
```



Как видно из графика, время на поиск решения задачи растет экспоненциально

Рисунок 5 – Блок анализа результатов

Было выполнено построение таблицы расстояний между городами Туркменского района Ставропольского края (таблица 1). На основе которой будут проходить все измерения.

Таблица 1 – Населенные пункты

1		Красный Маныч	Новокучерлинский	Сабан-Антуста	Голубиный	Каменная Балка	Кендже-Кулак	Шарахалсун	Кучерла	Мирное	Таврический	Куликовы Копани	Маштак-Кулак	Владимировка	Летняя Ставка	Чур	Овощи	Горный	Камбулат	Малые Ягуры	Казгулак	Ясный
2	Красный Маныч	-	11,8	10,9	2,3	13,8																
3	Новокучерлинский	11,8	-																			20,3
4	Сабан-Антуста	10,9		-		15,3	7,1															
5	Голубиный	2,3			-	11,1																
6	Каменная Балка	13,8		15,3	11,1	-																
7	Кендже-Кулак			7,1			-	12,9														
8	Шарахалсун						12,9	-	7,2													
9	Кучерла							7,2	-	14,9	13,2	12,9	17,5									
10	Мирое								14,9	-												
11	Таврический								13,2		-	4,9										
12	Куликовы Копани								12,9		4,9	-	14,9									
13	Маштак-Кулак								17,5			14,9	-	1,5	9	4,5						
14	Владимировка												1,5	-	4,4	5,8						
15	Летняя Ставка												9	4,4	-	4,8	10,2					26,1
16	Чур												4,5	5,8	4,8	-	12,8					
17	Овощи														10,2	12,8	-	21,9	24,8			
18	Горный																21,9	-	12,7			
19	Камбулат																24,8	12,7	-	9,8		
20	Малые Ягуры																		9,8	-	19,3	
21	Казгулак																			19,3	-	40,9
22	Ясный		20,3												26,1						40,9	-

Были использованы вспомогательные абстрактные классы Node, Problem и Queue.

Листинг Node.py:

```
import random
import heapq
import math
import sys
from collections import defaultdict, deque, Counter
from itertools import combinations


class Node:
    "Узел в дереве поиска"
    def __init__(self, state, parent=None, action=None, path_cost=0):
        self.__dict__.update(state=state, parent=parent, action=action,
                             path_cost=path_cost)

    def __repr__(self): return '<{}>'.format(self.state)

    def __len__(self): return 0 if self.parent is None else (1 +

len(self.parent))

    def __lt__(self, other): return self.path_cost < other.path_cost

failure = Node('failure', path_cost=math.inf) # Алгоритм не смог найти
решение.
cutoff = Node('cutoff', path_cost=math.inf) # Указывает на то, что поиск с
итеративным углублением был прерван.
```

```
def expand(problem, node):
    "Раскрываем узел, создав дочерние узлы."
    s = node.state
    for action in problem.actions(s):
        s1 = problem.result(s, action)
        cost = node.path_cost + problem.action_cost(s, action, s1)
        yield Node(s1, node, action, cost)


def path_actions(node):
    "Последовательность действий, чтобы добраться до этого узла."
    if node.parent is None:
        return []
    return path_actions(node.parent) + [node.action]


def path_states(node):
    "Последовательность состояний, чтобы добраться до этого узла"
    if node in (cutoff, failure, None):
        return []
    return path_states(node.parent) + [node.state]
```

Листинг Problem.py:

```
import random
import heapq
import math
import sys
from collections import defaultdict, deque, Counter
from itertools import combinations


class Problem:
    """Абстрактный класс для формальной задачи. Новый домен
специализирует этот класс,
переопределяя `actions` и `results`, и, возможно, другие методы.
Эвристика по умолчанию равна 0, а стоимость действия по умолчанию
равна 1 для всех состояний.
Когда вы создаете экземпляр подкласса, укажите `начальное` и
`целевое` состояния
(или задайте метод `is_goal`) и, возможно, другие ключевые слова для
подкласса."""

    def __init__(self, initial=None, goal=None, **kwds):
        self.__dict__.update(initial=initial, goal=goal, **kwds)

    def actions(self, state):        raise NotImplementedError

    def result(self, state, action): raise NotImplementedError

    def is_goal(self, state):        return state == self.goal

    def action_cost(self, s, a, s1): return 1

    def h(self, node):               return 0

    def __str__(self):
        return '{}({!r}, {!r})'.format(
            type(self).__name__, self.initial, self.goal)
```

Листинг Queue.py:

```
import random
import heapq
import math
import sys
from collections import defaultdict, deque, Counter
from itertools import combinations


FIFOQueue = deque
LIFOQueue = list
class PriorityQueue:
    """Очередь, в которой элемент с минимальным значением f(item) всегда
    выгружается первым."""
    def __init__(self, items=(), key=lambda x: x):
        self.key = key
        self.items = [] # a heap of (score, item) pairs
        for item in items:
            self.add(item)

    def add(self, item):
        """Добавляем элемент в очередь."""
        pair = (self.key(item), item)
        heapq.heappush(self.items, pair)

    def pop(self):
        """Достаем и возвращаем элемент с минимальным значением
        f(item)."""
        return heapq.heappop(self.items)[1]

    def top(self): return self.items[0][1]

    def __len__(self): return len(self.items)
```

Помимо этого, был разработан класс TSP для решения задачи коммивояжера.

Листинг id.py:

```
import time
from collections import defaultdict
from Problem import Problem
from Node import Node, failure, path_states
from Queue import PriorityQueue


# Список городов и расстояний между ними (без повторов)
distances = {
    ('Красный Маныч', 'Новокучерлинский'): 11.8,
    ('Красный Маныч', 'Сабан-Антуста'): 10.9,
    ('Красный Маныч', 'Голубиный'): 2.3,
    ('Красный Маныч', 'Каменная Балка'): 13.8,
    ('Новокучерлинский', 'Ясный'): 20.3,
    ('Сабан-Антуста', 'Каменная Балка'): 15.3,
    ('Сабан-Антуста', 'Кендже-Кулак'): 7.1,
    ('Голубиный', 'Каменная Балка'): 11.1,
    ('Кендже-Кулак', 'Шарахалсун'): 12.9,
    ('Шарахалсун', 'Кучерла'): 7.2,
    ('Кучерла', 'Мирное'): 14.9,
    ('Кучерла', 'Таврический'): 13.2,
```

```
    ('Куликовы Копани', 'Маштак-Кулак'): 14.9,
    ('Маштак-Кулак', 'Летняя Ставка'): 9.0,
    ('Летняя Ставка', 'Ясный'): 26.1,
    ('Летняя Ставка', 'Овощи'): 10.2,
    ('Чур', 'Овощи'): 12.8,
    ('Овощи', 'Горный'): 21.9,
    ('Камбулат', 'Малые Ягуры'): 9.8,
    ('Малые Ягуры', 'Казгулак'): 19.3,
    ('Казгулак', 'Ясный'): 40.9,
}


class TSP(Problem):
    """Класс для решения задачи коммивояжера."""

    def __init__(self, start, finish):
        super().__init__(initial=start, goal=finish)
        self.graph = self.build_graph()

    def build_graph(self):
        """Построение графа городов на основе списка расстояний."""
        graph = defaultdict(list)
        for (city1, city2), dist in distances.items():
            graph[city1].append((city2, dist))
            graph[city2].append((city1, dist))  # Двусторонняя связь
        return graph

    def actions(self, state):
        """Возвращает соседние города и расстояние до них."""
        #print('action', self.graph[state], state)
        return self.graph[state]


    def result(self, state, action):
        """Переход в следующий город."""
        return action[0]

    def action_cost(self, state, action, result):
        """Возвращает стоимость перехода."""
        return action[1]


# Функция для поиска решения задачи коммивояжера
def search_TSP(problem):
    border = PriorityQueue([Node(problem.initial)])  # Очередь с приоритетом
    path = set()

    while border:
        node = border.pop()
        if problem.is_goal(node.state):
            return path_states(node), node.path_cost

        path.add(node.state)

        for city, cost in problem.actions(node.state):
            child = Node(city, node, path_cost=node.path_cost + cost)
            if child.state not in path:
                border.add(child)

    return failure


# Инициализация задачи
problem = TSP('Красный Маныч','Чур')
```

```python
# Запуск поиска и замер времени
start_time = time.time()
route, total_distance = search_TSP(problem)
end_time = time.time()

# Вывод результатов
if route:
    print(f"Минимальный маршрут: {' -> '.join(route)}")
    print(f"Общее расстояние: {total_distance} км")
else:
    print("Маршрут не найден.")

print(f"Время выполнения: {end_time - start_time:.4f} секунд")
```

```
C:\Users\User\Desktop\git\SAI-Lab-1\py\SAI-Lab1\.venv\Scripts\python.exe C:\Users\User\Desktop\git\SAI-Lab-1\py\SAI-Lab1\id.py
Минимальный маршрут: Красный Маныч -> Новокучерлинский -> Ясный -> Летняя Ставка -> Овощи -> Чур
Общее расстояние: 81.2 км
Время выполнения: 0.0000 секунд

Process finished with exit code 0
```

Рисунок 6 – Результат решения задачи

Вывод: в ходе выполнения лабораторной работы было выполнено решение задачи коммивояжёра, используя метод перебора. Основываясь на результатах поиска при разном размере массива, можно сделать вывод, что сложность задачи растет экспоненциально с увеличением количества городов.

Вопросы:

1. Что представляет собой метод «слепого поиска» в искусственном интеллекте?

Он исследует пространство возможных решений методом проб и ошибок, не обладая информацией о том, насколько близко каждое принятое решение к финальной цели.

2. Как отличается эвристический поиск от слепого поиска?

Эвристический поиск, в отличие от слепого, использует дополнительные знания или «эвристики» для направления процесса поиска.

3. Какую роль играет эвристика в процессе поиска?

Все эффективные методы (сокращающие полный перебор) – методы эвристические. В большинстве эвристических методов находится не самый эффективный маршрут, а приближённое решение. Зачастую востребованы так называемые any-time алгоритмы, то есть постепенно улучшающие некоторое текущее приближенное решение.

4. Приведите пример применения эвристического поиска в реальной задаче.

Шахматы представляют собой классическую арену, где можно применить эти стратегии поиска.

5. Почему полное исследование всех возможных ходов в шахматах затруднительно для ИИ?

Учитывая огромное количество возможных ходов в шахматах, полное исследование всех вариантов становится практически невозможным даже для современных суперкомпьютеров.

6. Какие факторы ограничивают создание идеального шахматного ИИ?

Современные технологии и компьютеры не располагают возможностями для анализа такого большого количества вариантов.

7. В чем заключается основная задача искусственного интеллекта при выборе ходов в шахматах?

Эффективное просеивание и анализ ходов, отсекая менее перспективные и сосредотачиваясь на более целесообразных.

8. Как алгоритмы ИИ балансируют между скоростью вычислений и нахождением оптимальных решений?

Жадные алгоритмы, эвристики, разделяй и властвуй, параллелизация, адаптивные методы, пороговые методы, обучение на примерах

9. Каковы основные элементы задачи поиска маршрута по карте?

Узлы, ребра, вес ребра, начало, конец, алгоритм поиска, маршрут

10. Как можно оценить оптимальность решения задачи маршрутизации на карте Румынии?

Оптимальность решения в данном случае предполагает нахождение маршрута с минимальной стоимостью.

11. Что представляет собой исходное состояние дерева поиска в задаче маршрутизации по карте Румынии?

Город Арад.

12. Какие узлы называются листовыми в контексте алгоритма поиска по дереву?

В контексте алгоритма поиска по дереву листовыми узлами называются узлы, которые не имеют дочерних узлов (или подузлов). Это конечные точки в структуре дерева, которые представляют собой завершенные решения или состояния, не требующие дальнейшего разветвления.

13. Что происходит на этапе расширения узла в дереве поиска?

Расширение подразумевает применение функции преемника, которая определяет все возможные действия, применимые к текущему состоянию.

14. Какие города можно посетить, совершив одно действие из Арада в примере задачи поиска по карте?

Сибиу, Тимишоара, Зеринд

15. Как определяется целевое состояние в алгоритме поиска по дереву?

Сравниваем с целью назначения, если не является таковым, продолжаем поиск.

16. Какие основные шаги выполняет алгоритм поиска по дереву?

Алгоритм действует следующим образом: если нет кандидатов для расширения, алгоритм возвращает неудачу; в противном случае выбирается листовой узел для расширения в соответствии со стратегией, определяющей, какой из листовых узлов будет расширен; если узел содержит целевое состояние, возвращается соответствующее решение – путь в дереве, приведший к этому листовому узлу с целевым состоянием; если нет, узел расширяется, и полученные узлы добавляются к дереву.

17. Чем различаются состояния и узлы в дереве поиска?

Состояния представляют собой конкретные конфигурации или условия задачи, описывающие текущее положение, например, положение в игре. Узлы – это элементы дерева поиска, которые содержат состояние и ссылки на дочерние узлы, организующие иерархию поиска.

18. Что такое функция преемника и как она используется в алгоритме поиска?

Функция преемника – это метод, который генерирует все возможные состояния (или узлы), которые могут быть достигнуты из текущего состояния в процессе поиска. Она используется в алгоритмах поиска для расширения узлов, позволяя исследовать новое состояние, переходя от одного узла к другому.

19. Какое влияние на поиск оказывают такие параметры, как b (разветвление), d (глубина решения) и m (максимальная глубина)?

1. b (максимальный коэффициент разветвления) - показывает, сколько дочерних узлов может иметь один узел.

2. d (глубина наименее дорогого решения) - определяет, насколько далеко нужно спуститься по дереву для нахождения оптимального решения.

3. m (максимальная глубина дерева) - показывает, насколько глубоко можно в принципе спуститься по дереву. В некоторых случаях это значение может быть бесконечным.

20. Как алгоритмы поиска по дереву оцениваются по критериям полноты, временной и пространственной сложности, а также оптимальности?

Полнота означает, что алгоритм находит решение, если оно существует. Отсутствие решения возможно только в случае, когда задача невыполнима.

Временная сложность измеряется количеством сгенерированных узлов, а не временем в секундах или циклах ЦПУ. Она пропорциональна общему количеству узлов.

Пространственная сложность относится к максимальному количеству узлов, которые нужно хранить в памяти в любой момент времени. В некоторых алгоритмах она совпадает с временной сложностью.

21. Какую роль выполняет класс Problem в приведенном коде?

Этот класс служит шаблоном для создания конкретных задач в различных предметных областях. Каждая конкретная задача будет наследовать этот класс и переопределять его методы

22. Какие методы необходимо переопределить при наследовании класса Problem?

Методы actions и result являются абстрактными и должны быть реализованы для каждой конкретной задачи.

23. Что делает метод is_goal в классе Problem?

Метод is_goal проверяет, достигнуто ли целевое состояние

24. Для чего используется метод action_cost в классе Problem?

Метод action_cost предоставляет стандартные реализации для стоимости действия.

25. Какую задачу выполняет класс Node в алгоритмах поиска?

Класс Node представляет узел в дереве поиска.

26. Какие параметры принимает конструктор класса Node?

Конструктор класса Node принимает текущее состояние (state), ссылку на родительский узел (parent), действие, которое привело к этому узлу (action), и стоимость пути (path_cost).

27. Что представляет собой специальный узел failure?

Специальный узел для обозначения неудачи в поиске.

28. Для чего используется функция expand в коде?

Функция expand , расширяет узел, генерируя дочерние узлы.

29. Какая последовательность действий генерируется с помощью функции path_actions?

Функция path_actions возвращает последовательность действий, которые привели к данному узлу.

30. Чем отличается функция path_states от функции path_actions?

Функция path_states возвращает последовательность **состояний**, ведущих к данному узлу.

31. Какой тип данных используется для реализации FIFOQueue?

Она реализуется с помощью deque из модуля collections. deque - это обобщенная версия стека и очереди, которая поддерживает добавление и удаление элементов с обоих концов.

32. Чем отличается очередь FIFOQueue от LIFOQueue?

FIFO – это очередь, где тот, кто первый зашел, тот первый и вышел. В LIFO зашедший первым выходит последним (стек).

33. Как работает метод add в классе PriorityQueue?

Метод для добавления элемента в очередь. Каждый элемент добавляется в кучу с его приоритетом, определенным функцией key.

34. В каких ситуациях применяются очереди с приоритетом?

Это полезно, когда нужно обработать наиболее важные элементы в первую очередь.

35. Как функция heappop помогает в реализации очереди с приоритетом?

Гарантирует извлечение элемента с наименьшим приоритетом