Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Департамент цифровых, робототехнических
систем и электроники института перспективной инженерии

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ №1
дисциплины «Искусственный интеллект в профессиональной сфере»

Выполнил:
Плугатырев Владислав Алексеевич
3 курс, группа ПИЖ-б-о-22-1,
09.03.04 «Программная инженерия»,
направленность (профиль) «Разработка и
сопровождение программного
обеспечения», очная форма обучения

_____
(подпись)

Доцент кафедры инфокоммуникаций:
Воронкин Роман Александрович

_____
(подпись)

Отчет защищен с оценкой _____ Дата защиты_____

Ставрополь, 2024 г.

Тема: Исследование методов поиска в пространстве состояний.

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

Ход выполнения работы

1. Создание репозитория.

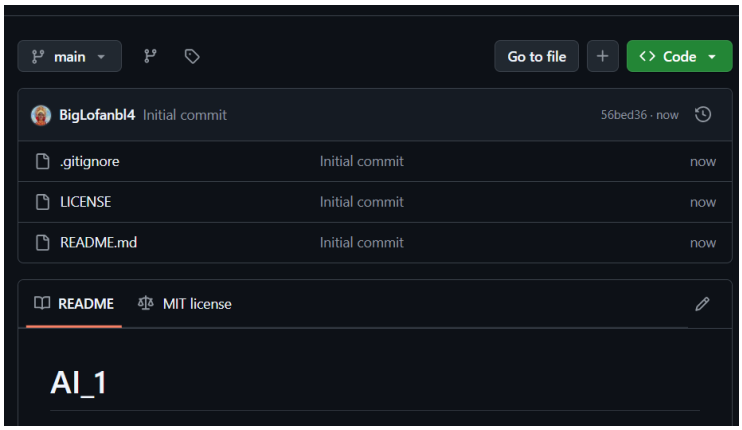

Рисунок 1.1 – Создание репозитория

2. Построение графа и нахождение минимального маршрута.

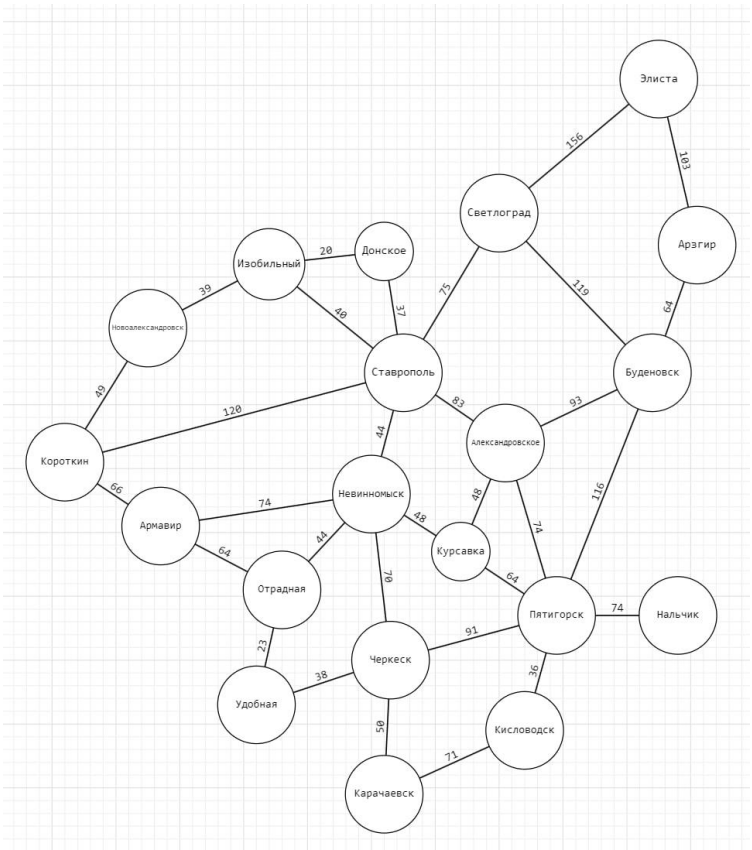

Рисунок 1.2 – Построенный граф

```python
def find_path(graph, start, end):
    min_cost = math.inf
    best_path = None

    nodes = list(graph.keys())
    nodes.remove(start)
    nodes.remove(end)

    for combination in combinations(nodes, 3):
        path = [start] + list(combination) + [end]
        cost = 0

        for index, city in enumerate(path):
            if (len(path) == index + 1):
                break
            cost += graph[city].get(path[index + 1], math.inf)

        if cost < min_cost:
            min_cost = cost
            best_path = path

    return best_path, min_cost

if __name__ == "__main__":
    graph = {
        'Ставрополь': {'Невинномыск': 44, 'Светлоград': 75, 'Александровское': 83, 'Изобильный': 40, 'Донское': 37, 'Короткин': 120},
        'Невинномыск': {'Ставрополь': 44, 'Армавир': 74, 'Черкеск': 70, 'Отрадная': 44, 'Курсавка': 48},
        'Светлоград': {'Ставрополь': 75, 'Элиста': 156, 'Буденовск': 119},
        'Элиста': {'Светлоград': 156, 'Арзгир': 103},
        'Пятигорск': {'Нальчик': 74, 'Буденовск': 116, 'Александровское': 74, 'Черкеск': 91, 'Курсавка': 64, 'Кисловодск': 36},
        'Нальчик': {'Пятигорск': 74},
        'Армавир': {'Невинномыск': 74, 'Короткин': 66, 'Отрадная': 64},
        'Короткин': {'Армавир': 66, 'Ставрополь': 120, 'Новоалександровск': 49},
        'Буденовск': {'Светлоград': 119, 'Пятигорск': 116, 'Арзгир': 64, 'Александровское': 93},
        'Арзгир': {'Буденовск': 64, 'Элиста': 103},
        'Александровское': {'Ставрополь': 83, 'Пятигорск': 74, 'Буденовск': 93, 'Курсавка': 48},
        'Черкеск': {'Невинномыск': 70, 'Пятигорск': 91, 'Удобная': 38, 'Карачаевск': 50},
        'Изобильный': {'Ставрополь': 40, 'Донское': 20, 'Новоалександровск': 39},
        'Донское': {'Ставрополь': 37, 'Изобильный': 20},
        'Новоалександровск': {'Изобильный': 39, 'Короткин': 49},
        'Отрадная': {'Армавир': 64, 'Невинномыск': 44, 'Удобная': 23},
        'Удобная': {'Отрадная': 23, 'Черкеск': 38},
        'Курсавка': {'Невинномыск': 48, 'Пятигорск': 64, 'Александровское': 48},
        'Карачаевск': {'Черкеск': 50, 'Кисловодск': 71},
        'Кисловодск': {'Карачаевск': 71, 'Пятигорск': 36}
    }

    start = 'Ставрополь'
    end = 'Пятигорск'


    best_path, best_cost = find_path(graph, start, end)

    print("Лучший путь:", best_path)
    print("Лучшая цена:", best_cost)
```

Рисунок 1.3 – Программа

```
Лучший путь: ['Ставрополь', 'Невинномыск', 'Курсавка', 'Александровское', 'Пятигорск']
Лучшая цена: 214
```

Рисунок 1.4 – Результат

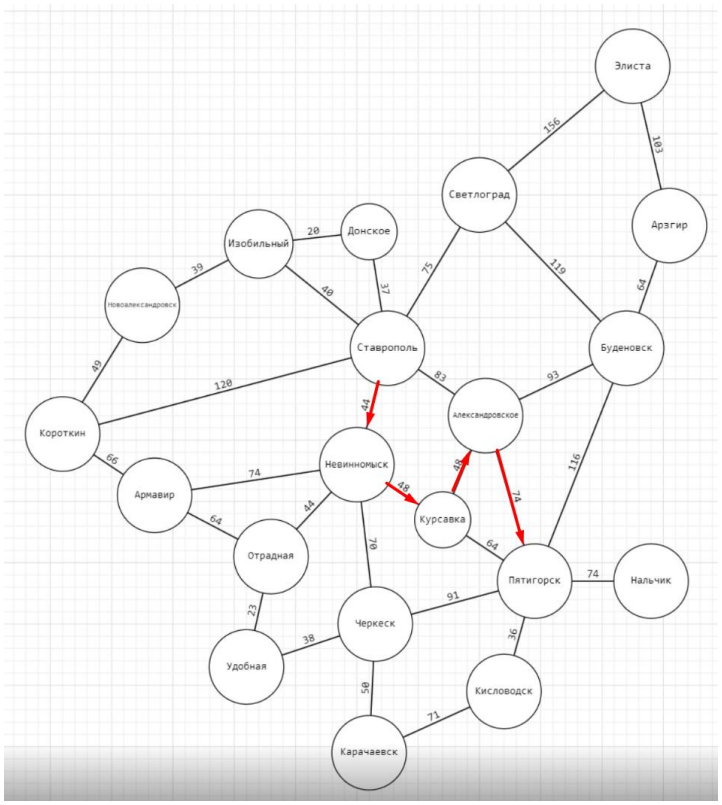

Рисунок 1.5 – Полученный путь

3. Решение задачи коммивояжера: уменьшил количество узлов до 7, сделав граф, полностью связным, как того требует задач коммивояжера.

```
graph = {
'Ставрополь': {
    'Александровское': 83,
    'Невинномыск': 44,
    'Изобильный': 40,
    'Пятигорск': 61,
    'Черкеск': 91,
    'Кисловодск': 56,
},
'Александровское': {
    'Ставрополь': 83,
    'Пятигорск': 74,
    'Черкеск': 67,
    'Невинномыск': 50,
    'Кисловодск': 90,
    'Изобильный': 81,
},
'Пятигорск': {
    'Александровское': 74,
    'Черкеск': 91,
    'Кисловодск': 36,
    'Ставрополь': 61,
    'Невинномыск': 78,
    'Изобильный': 82,
},
'Черкеск': {
    'Пятигорск': 91,
    'Невинномыск': 70,
    'Ставрополь': 84,
    'Александровское': 67,
    'Кисловодск': 55,
    'Изобильный': 63,
},
'Кисловодск': {
    'Пятигорск': 36,
    'Ставрополь': 56,
    'Александровское': 90,
    'Черкеск': 55,
    'Невинномыск': 48,
    'Изобильный': 71,
},
'Невинномыск': {
    'Ставрополь': 44,
    'Черкеск': 70,
    'Александровское': 50,
    'Пятигорск': 78,
    'Кисловодск': 48,
    'Изобильный': 39,
},
'Изобильный': {
    'Ставрополь': 40,
    'Александровское': 81,
    'Пятигорск': 82,
    'Черкеск': 63,
    'Кисловодск': 71,
    'Невинномыск': 39,
},
}
```

Рисунок 1.6 – Вид графа

```python
class TSP(Problem):
    # state - список уже пройденных городов (конфигурация системы)
    # action - строка, которая говорит какой город будет следующим

    def __init__(self, initial, graph):
        super().__init__(initial, goal=None)
        self.graph = graph
        self.cities_num = len(graph)

    def actions(self, state):
        # Список возможных городов для посещения
        return [
            city
            for city in self.graph[state[-1]]
            if city not in state or (len(state) == self.cities_num and city == state[0])
        ]

    def result(self, state, action):
        # Вернет новое состояние
        new_state = [i for i in state]
        new_state.append(action)
        return new_state

    def is_goal(self, state):
        # Посещены ли все города
        solved = True
        for city in list(self.graph.keys()):
            if city not in state:
                solved = False

        # Вернулись ли мы в начальный город
        return solved and state[-1] == state[0]

    def action_cost(self, s, a, s1):
        # Расстояние между последним городом и новым
        return self.graph[s[-1]][a]

failure = Node("failure", path_cost=math.inf)
cutoff = Node("cutoff", path_cost=math.inf)
# Для вывода оптимального пути
def path_actions(node):
    if node in (cutoff, failure, None):
        return []
    # у самого первого узла action = None, заменим этот action на стартовый город
    action = node.action if node.action is not None else node.state[0]
    return path_actions(node.parent) + [action]
```

Рисунок 1.7 – Класс TSP

```python
def tsp_solution(initial, graph):
    problem = TSP(initial, graph)
    best_node = None
    min_cost = math.inf
    frontier = [
        Node(
            [
                initial,
            ]
        )
    ]

    while frontier:
        curr_node = frontier.pop()

        if problem.is_goal(curr_node.state):
            if curr_node.path_cost < min_cost:
                best_node = curr_node
                min_cost = curr_node.path_cost
            continue
        elif curr_node.path_cost > min_cost:
            continue

        frontier.extend(expand(problem, curr_node))


    return best_node, min_cost
```

Рисунок 1.8 – Функция решения задачи

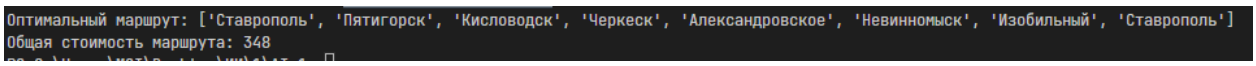

Рисунок 1.9 – Результат

Ответы на контрольные вопросы

1. Метод слепого поиска представляет собой поиск, который не использует никакой дополнительной информации о проблеме, кроме данных о начальном и целевом состояниях.

2. Эвристический поиск отличается от слепого поиска тем, что использует эвристику, которая направляет поиск, опираясь на некоторую оценку приближенности к цели.

3. Эвристика играет роль направляющей силы в процессе поиска, помогая быстрее достичь цели за счет использования информации о проблеме.

4. Примером применения эвристического поиска является задача нахождения кратчайшего пути в навигационной системе, где используются оценки расстояний до цели.

5. Полное исследование всех возможных ходов в шахматах затруднительно для ИИ из-за экспоненциального роста числа возможных состояний игры с каждым новым ходом.

6. Факторы, ограничивающие создание идеального шахматного ИИ, включают ограниченные вычислительные ресурсы и сложность поиска оптимального хода в разумные сроки.

7. Основная задача искусственного интеллекта при выборе ходов в шахматах заключается в нахождении наилучшего возможного хода, который приводит к победе или улучшению позиции.

8. Алгоритмы ИИ балансируют между скоростью вычислений и нахождением оптимальных решений за счет использования эвристик и сокращения числа исследуемых вариантов.

9. Основными элементами задачи поиска маршрута по карте являются исходное местоположение, целевое местоположение, возможные маршруты и их стоимость.

10. Оптимальность решения задачи маршрутизации на карте Румынии можно оценить по минимальной общей стоимости пути, ведущего к цели.

11. Исходное состояние дерева поиска в задаче маршрутизации по карте Румынии — это точка старта, от которой начинается исследование всех возможных маршрутов.

12. Листовыми узлами в контексте алгоритма поиска по дереву называются узлы, которые не имеют потомков и являются конечными состояниями.

13. На этапе расширения узла в дереве поиска генерируются все возможные действия и состояния, которые могут быть достигнуты из данного узла.

14. Одним действием из Арада можно посетить города Зеринд, Сибиу или Тимишоара.

15. Целевое состояние в алгоритме поиска по дереву определяется состоянием, которое соответствует заранее заданной цели.

16. Основные шаги, выполняемые алгоритмом поиска по дереву, включают инициализацию, расширение узлов и проверку достижения целевого состояния.

17. Состояния в дереве поиска — это конфигурации системы, а узлы — это структуры, которые содержат информацию о состоянии, родительских узлах и стоимости пути.

18. Функция преемника используется для получения списка возможных следующих состояний для текущего состояния.

19. Параметры b, d и m влияют на производительность поиска, определяя, сколько узлов исследуется, на какой глубине находится решение и насколько глубоко может исследоваться дерево.

20. Алгоритмы поиска по дереву оцениваются по полноте, временной и пространственной сложности, а также по оптимальности на основе их способности найти решение, затрачиваемого времени и памяти.

21. Класс Problem выполняет роль абстрактного представления задачи, определяя начальное состояние, целевое состояние, возможные действия и правила переходов.

22. При наследовании класса Problem необходимо переопределить методы для определения начального состояния, целевого состояния, возможных действий и их стоимости.

23. Метод is_goal в классе Problem проверяет, достигнуто ли целевое состояние для текущего состояния.

24. Метод action_cost в классе Problem используется для расчета стоимости выполнения действия и перехода в новое состояние.

25. Класс Node в алгоритмах поиска представляет узел дерева поиска и хранит информацию о состоянии, родителе, действии и стоимости пути.

26. Конструктор класса Node принимает параметры состояния, родителя, действия и стоимости пути для создания нового узла.

27. Специальный узел failure представляет собой обозначение неудачи в поиске, используемое для указания того, что решение не найдено.

28. Функция expand в коде генерирует все возможные следующие узлы, расширяя текущее состояние.

29. Последовательность действий, генерируемая с помощью функции path_actions, представляет собой набор шагов, которые привели к целевому состоянию.

30. Функция path_states возвращает последовательность состояний, а функция path_actions возвращает последовательность действий, которые привели к целевому состоянию.

31. Для реализации FIFOQueue используется тип данных очередь, которая поддерживает порядок первого пришел — первый ушел.

32. Очередь FIFOQueue отличается от LIFOQueue тем, что в FIFO элементы обрабатываются в порядке поступления, а в LIFO — в обратном порядке.

33. Метод add в классе PriorityQueue добавляет элемент в очередь с указанием его приоритета, сортируя элементы по возрастанию приоритета.

34. Очереди с приоритетом применяются в ситуациях, где важно учитывать порядок выполнения задач на основе их важности.

35. Функция heappop помогает в реализации очереди с приоритетом, извлекая элемент с наивысшим приоритетом из структуры данных.