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

Институт перспективной инженерии
Департамент цифровых, робототехнических систем и электроники

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

Выполнил:
Арзютов Иван Владиславович

3 курс, группа ИТС-б-о-22-1,

11.03.02 «Инфокоммуникационные технологии и системы связи, очная форма обучения

_____
(подпись)

Руководитель практики:

Воронкин Р.А., канд. тех. наук, доцент, доцент кафедры инфокоммуникаций

_____
(подпись)

Отчет защищен с оценкой _____ Дата защиты_____

Ставрополь, 2024 г.

Исследование методов поиска в пространстве состояний

Ссылка на репозиторий: https://github.com/Ivanuschka/1-LB

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

Порядок выполнения работы

Задание 1

Создал общедоступный репозиторий на GitHub, в котором использована лицензия MIT.

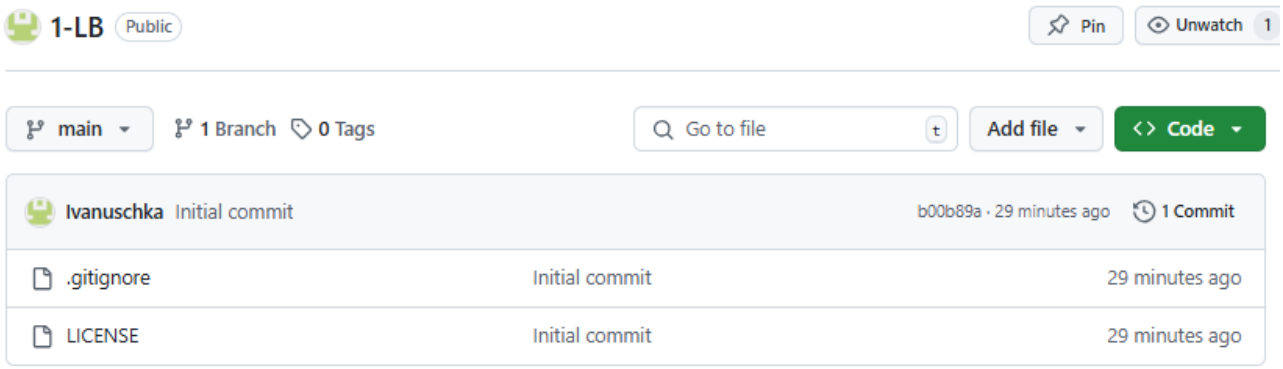


Рисунок 1. Создан новый репозиторий

Задание 2

Клонирование репозитория на свой компьютер.

```
C:\Users\ivana\OneDrive\Рабочий стол\ИИ\1 Лабораторная работа>git clone https://github.com/Ivanuschka/1-LB.git
Cloning into '1-LB'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (4/4), done.
```

Рисунок 2. Клонирование репозитория

Задание 3

Проработка примеров лабораторной работы

```python
import random
import heapq
import math
import sys
from collections import defaultdict, deque, Counter
from itertools import combinations
class Problem:
    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)
class Node:
    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
# Переносим после объявления класса
failure = Node('failure', path_cost=math.inf)
cutoff = Node('cutoff', path_cost=math.inf)
```

```python
cutoff = Node('cutoff', path_cost=math.inf) #Указывает на то, что поиск с итеративным углублением был прерван."
def expand(problem, node):
    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):
    if node.parent is None:
        return []
    return path_actions(node.parent) + [node.action]
def path_states(node):
    if node in (cutoff, failure, None):
        return []
    return path_states(node.parent) + [node.state]
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)."""
```

```python
            return heapq.heappop(self.items)[1]
        def top(self): return self.items[0][1]
        def __len__(self): return len(self.items)
```

Рисунок 3. Выполнение примера

Задание 4

Создание модулей для примеров

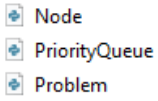


Рисунок 4. Модули в папке репозитория

Задание 5

Определить минимальный маршрут между городом E и I, который должен проходить через три промежуточных населённых пункта. Покажите данный путь на построенном графе.

```python
import networkx as nx
import matplotlib.pyplot as plt

# Создаем граф
G = nx.Graph()

# Список населенных пунктов
towns = ["Город A", "Город B", "Город C", "Город D", "Город E",
         "Город F", "Город G", "Город H", "Город I", "Город J"]

G.add_nodes_from(towns)

# Список дорог с расстояниями (весами рёбер)
roads = [("Город A", "Город B", 15), ("Город A", "Город C", 10),
         ("Город B", "Город D", 12), ("Город C", "Город D", 8),
         ("Город C", "Город E", 20), ("Город D", "Город E", 5),
         ("Город E", "Город F", 18), ("Город F", "Город G", 7),
         ("Город G", "Город H", 10), ("Город H", "Город I", 6),
         ("Город I", "Город J", 8), ("Город J", "Город A", 14),
         ("Город D", "Город G", 22), ("Город B", "Город H", 25)]

# Добавляем рёбра с расстояниями
for town1, town2, distance in roads:
    G.add_edge(town1, town2, weight=distance)

# Оптимальный путь (выделенный маршрут)
shortest_path = [("Город E", "Город D"), ("Город D", "Город G"),
                 ("Город G", "Город H"), ("Город H", "Город I")]

# Визуализация графа
plt.figure(figsize=(8, 6))
pos = nx.spring_layout(G, seed=42)  # Расположение узлов
```

```python

# Рисуем узлы и рёбра
nx.draw(G, pos, with_labels=True, node_color='lightblue', edge_color='gray',
        node_size=2000, font_size=10, font_weight='bold')

# Подписываем рёбра (расстояния)
edge_labels = {(town1, town2): f"{distance} км" for town1, town2, distance in roads}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=8)

# Выделяем кратчайший путь красным цветом
nx.draw_networkx_edges(G, pos, edgelist=shortest_path, edge_color='red', width=2.5)

plt.title("Минимальный маршрут между Городом E и Городом I")
plt.show()
```

Рисунок 4. Код для определения и показа оптимального маршрута

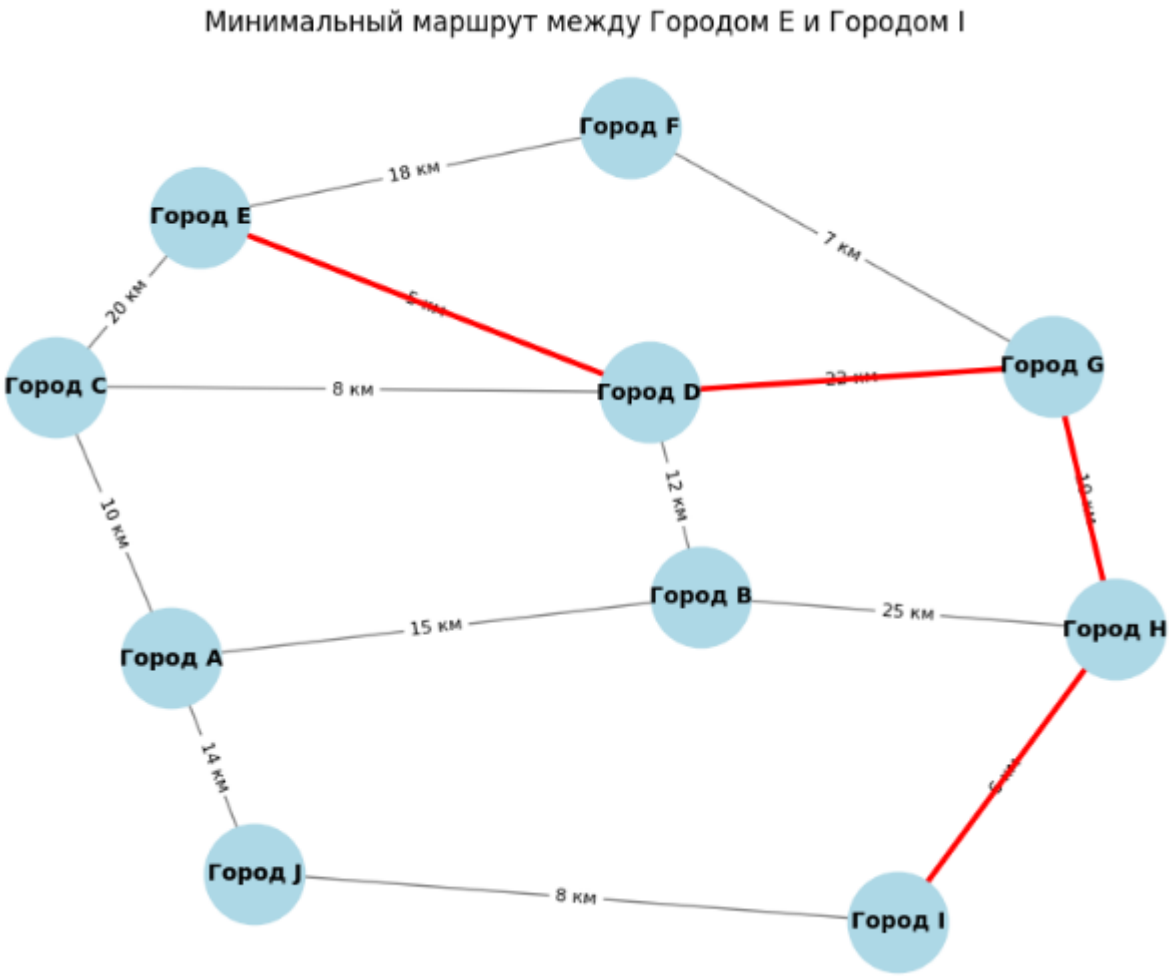


Рисунок 5. Построенный граф

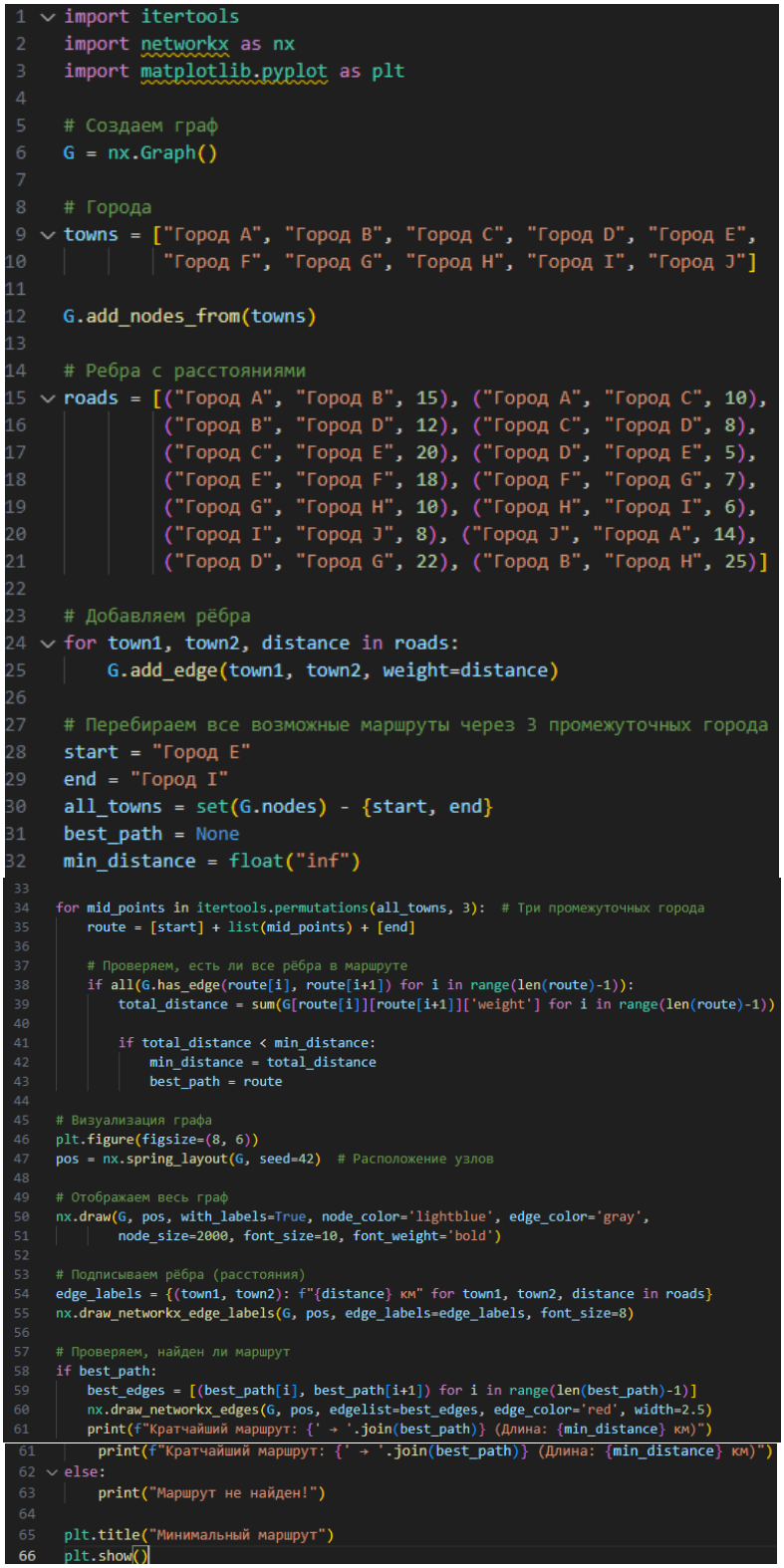
Задание 5.

Решим методом полного перебора задачу коммивояжёра.

Метод полного перебора реализован для решения задачи коммивояжёра.

Программа:

1. Перебирает все возможные маршруты, начиная и заканчивая в первом узле.
2. Вычисляет длину каждого маршрута.
3. Определяет маршрут с минимальной длиной.



```python
import itertools
import networkx as nx
import matplotlib.pyplot as plt

# Создаем граф
G = nx.Graph()

# Города
towns = ["Город A", "Город B", "Город C", "Город D", "Город E",
         "Город F", "Город G", "Город H", "Город I", "Город J"]

G.add_nodes_from(towns)

# Ребра с расстояниями
roads = [("Город A", "Город B", 15), ("Город A", "Город C", 10),
         ("Город B", "Город D", 12), ("Город C", "Город D", 8),
         ("Город C", "Город E", 20), ("Город D", "Город E", 5),
         ("Город E", "Город F", 18), ("Город F", "Город G", 7),
         ("Город G", "Город H", 10), ("Город H", "Город I", 6),
         ("Город I", "Город J", 8), ("Город J", "Город A", 14),
         ("Город D", "Город G", 22), ("Город B", "Город H", 25)]

# Добавляем рёбра
for town1, town2, distance in roads:
    G.add_edge(town1, town2, weight=distance)

# Перебираем все возможные маршруты через 3 промежуточных города
start = "Город E"
end = "Город I"
all_towns = set(G.nodes) - {start, end}
best_path = None
min_distance = float("inf")

for mid_points in itertools.permutations(all_towns, 3):  # Три промежуточных города
    route = [start] + list(mid_points) + [end]

    # Проверяем, есть ли все рёбра в маршруте
    if all(G.has_edge(route[i], route[i+1]) for i in range(len(route)-1)):
        total_distance = sum(G[route[i]][route[i+1]]['weight'] for i in range(len(route)-1))

        if total_distance < min_distance:
            min_distance = total_distance
            best_path = route

# Визуализация графа
plt.figure(figsize=(8, 6))
pos = nx.spring_layout(G, seed=42)  # Расположение узлов

# Отображаем весь граф
nx.draw(G, pos, with_labels=True, node_color='lightblue', edge_color='gray',
        node_size=2000, font_size=10, font_weight='bold')

# Подписываем рёбра (расстояния)
edge_labels = {(town1, town2): f"{distance} км" for town1, town2, distance in roads}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=8)

# Проверяем, найден ли маршрут
if best_path:
    best_edges = [(best_path[i], best_path[i+1]) for i in range(len(best_path)-1)]
    nx.draw_networkx_edges(G, pos, edgelist=best_edges, edge_color='red', width=2.5)
    print(f"Кратчайший маршрут: {' → '.join(best_path)} (Длина: {min_distance} км)")
else:
    print("Маршрут не найден!")

plt.title("Минимальный маршрут")
plt.show()
```

Рисунок 6. Код для решения задачи Коммивояжёра

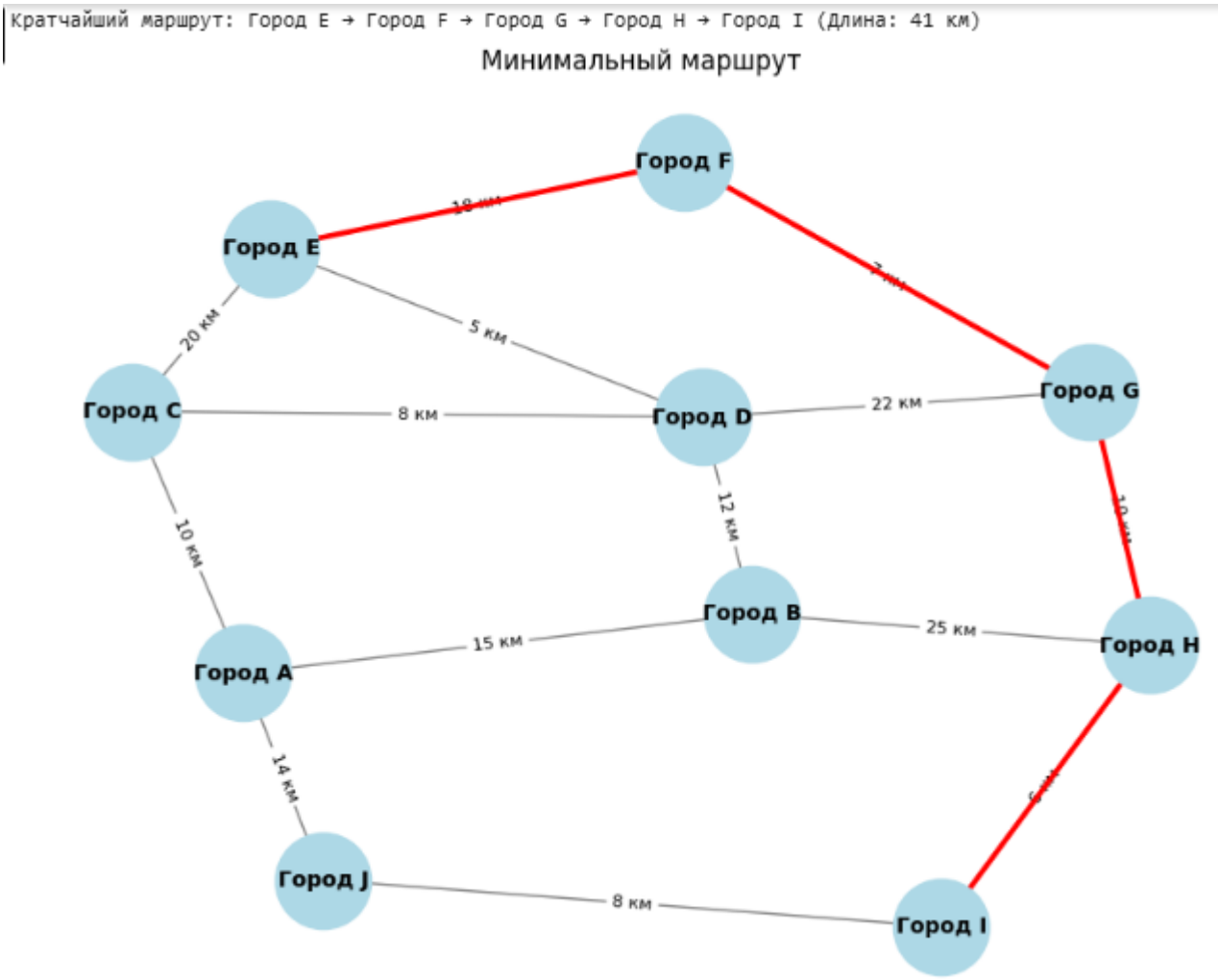


Рисунок 7. Графи решения задачи коммивояжера

Вывод по лабораторной работе: в ходе выполнения лабораторной работы были приобретены навыки по работе с методами поиска в пространстве состояний помощью языка программирования Python версии 3.x

Ответы на контрольные вопросы

1. Метод "слепого поиска" в искусственном интеллекте — это подход к поиску, при котором не используются никакие дополнительные знания о проблеме (эвристики), а только информация о состоянии текущего узла и его дочерних узлах. Этот метод включает в себя такие алгоритмы, как поиски в ширину и в глубину.

2. Эвристический поиск отличается от слепого поиска тем, что использует дополнительные знания о проблеме для оценки, насколько близко текущее состояние к цели. Он применяет эвристические функции для

приоритизации узлов, что позволяет более эффективно направлять поиск в сторону, которая, вероятнее всего, приведет к решению.

3. Эвристика в процессе поиска играет важную роль в оптимизации поиска, позволяя алгоритмам оценивать, какие состояния более перспективны, и сокращать пространство поиска. Эвристика помогает выбрать более обоснованные действия, что увеличивает вероятность нахождения решения быстрее.

4. Пример применения эвристического поиска в реальной задаче: Поиск на графе для нахождения кратчайшего пути на карте, например, алгоритм A, который использует эвристику для оценки расстояния до цели, позволяя находить маршруты более эффективно, чем алгоритмы слепого поиска.

5. Полное исследование всех возможных ходов в шахматах затруднительно для ИИ из-за огромного количества возможных комбинаций ходов. В шахматах число всех возможных позиций может достигать 10^120, что делает полное исследование неосуществимым в разумные сроки.

6. Факторы, ограничивающие создание идеального шахматного ИИ: Ограниченные ресурсы (вычислительная мощность и время), сложность заведомо неполной информации о стратегии, необходимость учитывать огромное количество возможных позиций за несколько ходов вперед и неэффективность генерирования всех позиций.

7. Основная задача ИИ при выборе ходов в шахматах заключается в оценке текущих позиций, предсказании результатов возможных ходов и выборе наиболее перспективного из них. Это требует поиска среди множества возможных будущих позиций.

8. Алгоритмы ИИ балансируют между скоростью вычислений и нахождением оптимальных решений с помощью техник, таких как обрезка альфа-бета, чтобы минимизировать количество положений, которые нужно анализировать. Они используют оценочные функции и оптимизации, чтобы сосредоточиться на наиболее перспективных ходах.

9. Основные элементы задачи поиска маршрута по карте включают стартовую и целевую точки, возможные маршруты или пути, оценки стоимости

движения по маршрутам и алгоритмы для поиска оптимального или кратчайшего пути.

10. Оптимальность решения задачи маршрутизации на карте Румынии можно оценить по критериям минимизации расстояния, времени проезда или затрат, а также использованию алгоритмов, например, A или Dijkstra.

11. Исходное состояние дерева поиска в задаче маршрутизации по карте — это узел, представляющий стартовую точку на карте, из которой начинается поиск. Этот узел соответствует начальному состоянию, с которого начинается исследование возможных маршрутов.

12. Листовые узлы в контексте алгоритма поиска по дереву — это узлы, которые не имеют дочерних узлов, то есть состояния, достигшие конца своей ветки поиска. Эти узлы представляют собой конечные состояния поиска. На этапе расширения узла в дереве поиска создаются новые узлы- дочерние, которые соответствуют возможным действиям, которые могут быть выполнены из текущего узла, таким образом расширяя пространство поиска.

13. Из города Арада в задаче поиска по карте можно посетить ближайшие города, если они соседние (например, другие города, соединенные дорогами с Арадом). Это зависит от конкретной конфигурации маршрутов на карте.

14. Целевое состояние в алгоритме поиска по дереву определяется как узел или состояние, удовлетворяющее условиям завершения задачи или достижения цели, например, достижение финальной точки на карте.

15. Основные шаги алгоритма поиска по дереву включают инициализацию, расширение узлов, оценку состояний, проверку на достижение целевого состояния и возврат решения, если цель достигнута.

16. Состояния в дереве поиска — это конкретные конфигурации системы, а узлы представляют собой абстракции этих состояний в структуре дерева. Узел может содержать информацию о состоянии, а также метаданные, такие как отец и стоимость.

17. Функция преемника — это функция, которая генерирует все возможные действия, которые могут быть выполнены из текущего состояния.

Она используется в алгоритме поиска, чтобы переходить от одного состояния к следующему, формируя дерево поиска.

18. Параметры b (разветвление), d (глубина решения) и m (максимальная глубина) влияют на сложность поиска. Чем выше b и d, тем больше узлов нужно исследовать, а m ограничивает максимальную глубину, которую может достигнуть поиск.

19. Алгоритмы поиска по дереву оцениваются по следующим критериям:

- Полнота: возможность найти решение, если оно существует.
- Временная сложность: общее время, необходимое для выполнения алгоритма.
- Пространственная сложность: объем памяти, необходимый для выполнения алгоритма.
- Оптимальность: способность находить лучшее решение, если оно существует.

20. Класс Problem обычно выполняет роль абстракции задачи в контексте поиска. Он представляет описание задачи, включая состояние, действия и цели, и служит основой для конкретных алгоритмов поиска.

21. Методы, которые необходимо переопределить при наследовании класса Problem, могут включать метод для определения цели (isgoal), функцию возврата стоимости действия (actioncost), генерацию возможных действий (successors), и другие методы, относящиеся к конкретной задаче.

22. Метод isgoal** в классе Problem проверяет, достигнута ли цель, т.е. является ли текущее состояние целевым состоянием. Если да, он может вернуть значение True, иначе — False.

23. **Метод actioncost в классе Problem используется для определения стоимости выполнения действий. Это может быть использовано для алгоритмов поиска, чтобы оценить себестоимость переходов между состояниями.

24. Класс Node в алгоритмах поиска представляет отдельный узел в дереве поиска, который описывает конкретное состояние, его родителей, стоимость, глубину и другие данные, необходимые для алгоритма.

25. Конструктор класса Node принимает такие параметры, как текущее состояние, родительский узел (предыдущее состояние), стоимость, глубина узла и, возможно, другие атрибуты, которые будут полезны для восприятия данных об узле.

26. Специальный узел failure может представлять состояние, в котором невозможно найти дальнейшие преемники, или состояние, которое не привело к успешному результату. Это необходимо для обработки ошибок или недопустимых состояний в ходе поиска.

27. Функция expand в коде используется для генерации дочерних узлов текущего узла, что позволяет развивать дерево поиска, добавляя новые состояния, которые могут быть достигнуты из текущего состояния.

28. Функция pathactions** генерирует последовательность действий (или ходов), которые были выполнены для достижения конечного состояния от начального. Она помогает понять, как было достигнуто решение.

29. **Функция pathstates возвращает последовательность состояний (или узлов) от начального состояния до целевого. Это может включать полный путь с указаниями на промежуточные этапы.

30. Тип данных, используемый для реализации FIFOQueue, обычно представляет собой очередь, такие как списки или двусвязные списки, которые обеспечивают порядок «первый пришел — первый вышел».

31. Очередь FIFOQueue (First In, First Out) отличается от LIFOQueue (Last In, First Out) тем, что в FIFO первый элемент, добавленный в очередь, будет первым, который будет извлечен, тогда как в LIFO последний элемент, добавленный в стек, будет первым, который будет извлечен.

32. Метод add в классе PriorityQueue добавляет элемент в очередь с учетом его приоритета, обеспечивая правильное размещение элементов так, чтобы элемент с наивысшим приоритетом всегда находился в начале очереди.