

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

Дисциплины «Искусственный интеллект в профессиональной сфере»

Выполнил:

Пустяков Андрей Сергеевич

3 курс, группа ИВТ-б-о-22-1,

09.03.01 «Информатика и
вычислительная техника (профиль)
«Программное обеспечение средств
вычислительной
техники и автоматизированных
систем», очная форма обучения

(подпись)

Руководитель практики:

Воронкин Р. А., доцент департамента
цифровых и робототехнических
систем и электроники и института
перспективной инженерии

(подпись)

Ставрополь, 2024 г.

Тема: Исследование методов поиска в пространстве состояний.

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

Ход работы:

Для создания графа с некоторыми весами ребер был использован сервис «Яндекс Карты», были выбраны более 20 населенных пунктов, связанных между собой дорогами. В качестве городов на графе были выбраны города Австралии (рис. 1).

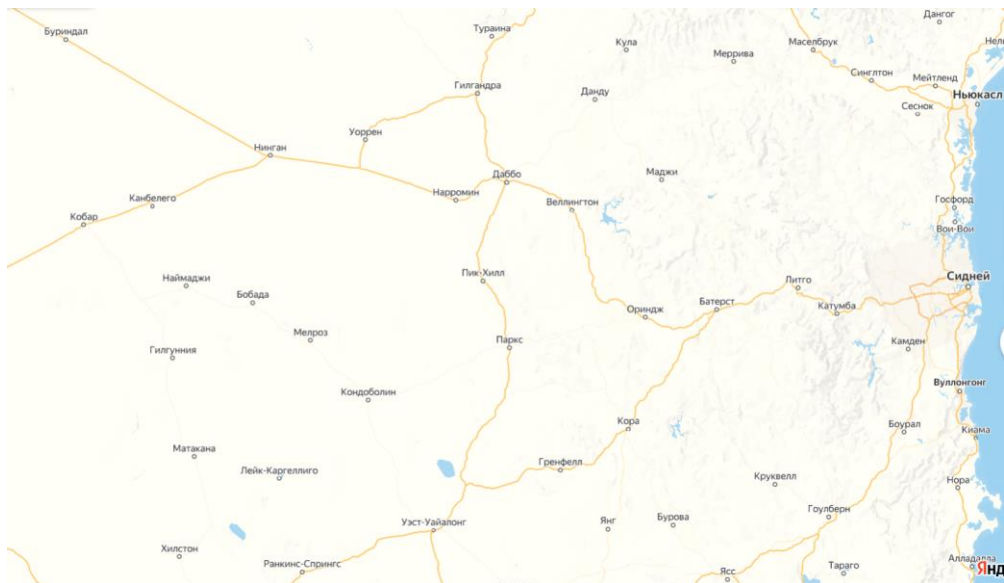


Рисунок 1 – Населенные пункты Австралии, недалеко от города «Сидней»

Был построен граф, в котором узлами выступают города Австралии, а ребрами – дороги, соединяющие эти города (рис. 2).

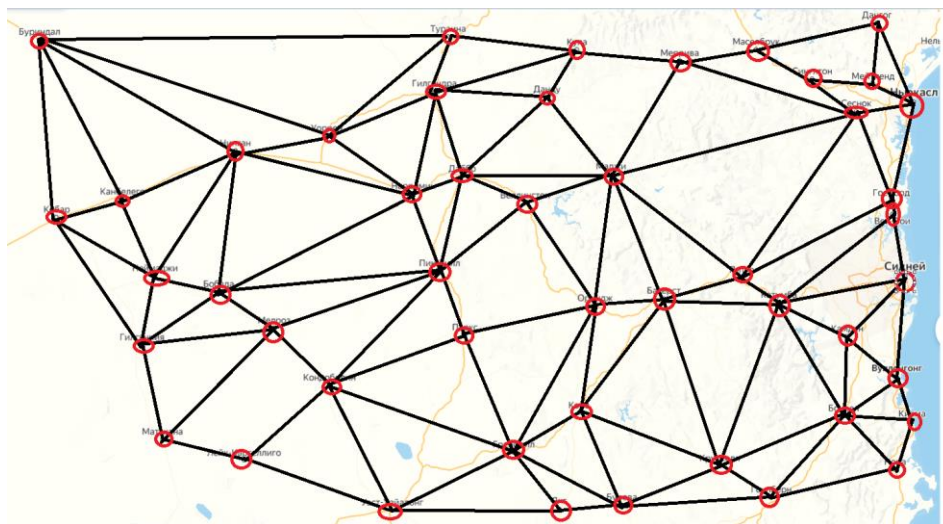


Рисунок 2 – Граф городов и дорог Австралии

Были построены на графе веса узлов. Вес каждого узла соответствует расстоянию дороги, ведущей к тому или иному городу, расстояния указаны в километрах (рис. 3).

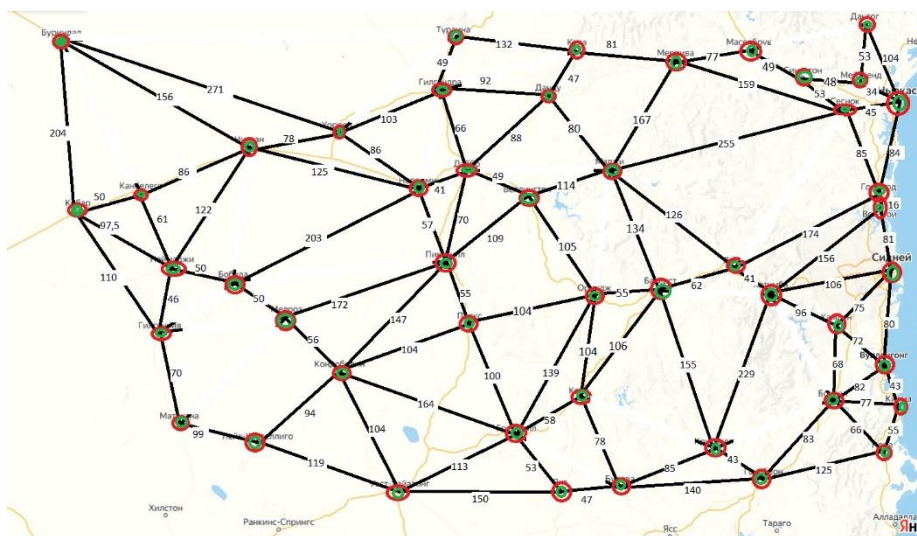


Рисунок 3 – Граф городов с расстоянием дорог в качестве весов

Допустим начальный населенный пункт это населенный пункт Буринда, а конечный – город Сидней, тогда кратчайший путь проходит через следующие узлы: Буринда → Нинган → Уоррен → Нарромин → Даббо → Веллингтон → Ориндж → Батерст → Литго → Кетумба → Сидней. Общая длина пути составила 779 км (исходя из весов соответствующих ребер). Данный путь был обозначен на графе (рис. 4).

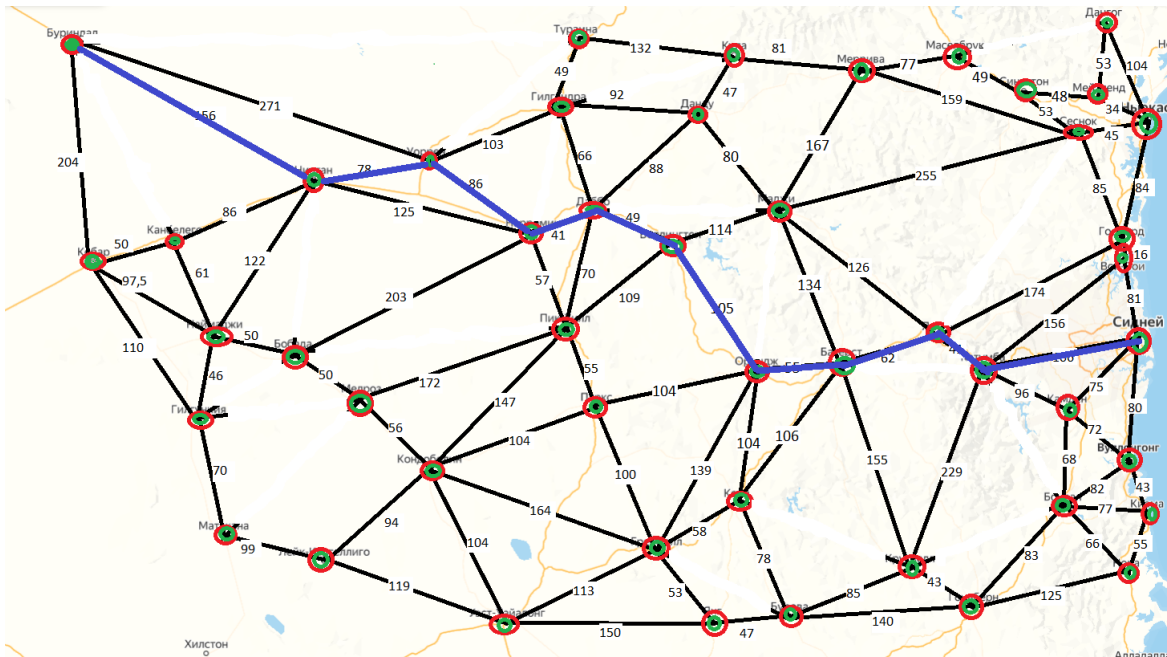


Рисунок 4 – Кратчайший путь от Буриндала до Сиднея

Данный граф был построен с помощью инструмента визуализации графов «Graphviz», для этого был написан код на языке «DOT»:

```
graph G {
    layout="neato";    // Используем алгоритм neato
    size="15,5!";      // Размер графа
    overlap=false;     // Убираем перекрытия
    splines=true;      // Плавные рёбра
    nodesep=0.6;       // Расстояние между узлами
    ranksep=1.0;       // Расстояние между уровнями
    concentrate=true;  // Объединение рёбер

    // Город Буриндал
    Буриндал -- Уоррен [label="271"];
    Буриндал -- Нинган [label="156"];
    Буриндал -- Кобар [label="204"];

    // Город Кобар
    Кобар -- Канбелого [label="50"];
    Кобар -- Наймаджи [label="97,5"];
    Кобар -- Гулгуния [label="110"];
```

// Город Гулгуния

Гулгуния -- Наймаджи [label="46"];

Гулгуния -- Матакана [label="70"];

// Город Наймаджи

Наймаджи -- Канбелого [label="61"];

Наймаджи -- Нинган [label="122"];

Наймаджи -- Бобада [label="150"];

// Город Нинган

Нинган -- Канбелого [label="86"];

Нинган -- Уоррен [label="78"];

Нинган -- Бобада [label="150"];

// Город Уоррен

Уоррен -- Гилгандра [label="103"];

Уоррен -- Нарромин [label="86"];

// Город Нарромин

Нарромин -- Бобада [label="203"];

Нарромин -- Пикхилл [label="57"];

Нарромин -- Даббо [label="41"];

// Город Пикхилл

Пикхилл -- Даббо [label="70"];

Пикхилл -- Веллингтон [label="109"];

Пикхилл -- Медроз [label="172"];

Пикхилл -- Кондоболин [label="147"];

Пикхилл -- Паркс [label="55"];

// Город Даббо

Даббо -- Гилгандра [label="66"];

Даббо -- Данду [label="88"];

Даббо -- Веллингтон [label="49"];

// Город Бобада

Бобада -- Медроз [label="50"];

// Город Медроз

Медроз -- Кондоболин [label="56"];

// Город Кондоболин

Кондоболин -- Паркс [label="104"];

Кондоболин -- Гренфелл [label="164"];

Кондоболин -- Уэст_Уайалонг [label="104"];

Кондоболин -- Лейк_Каргеллиго [label="94"];

// Город Матакана

Матакана -- Лейк_Каргеллиго [label="99"];

// Город Лейк_Каргеллиго

Лейк_Каргеллиго -- Уэст_Уайалонг [label="119"];

// Город Уэст_Уайалонг

Уэст_Уайалонг -- Гренфелл [label="113"];

Уэст_Уайалонг -- Янг [label="150"];

// Город Гренфелл

Гренфелл -- Паркс [label="100"];

Гренфелл -- Ориндж [label="139"];

Гренфелл -- Кора [label="58"];

Гренфелл -- Янг [label="53"];

// Город Ориндж

Ориндж -- Кора [label="104"];

Ориндж -- Паркс [label="104"];

Ориндж -- Веллингтон [label="105"];

Ориндж -- Батерст [label="55"];

// Город Веллингтон

Веллингтон -- Маджи [label="114"];

// Город Маджи

Маджи -- Данду [label="80"];

Маджи -- Меррива [label="167"];

Маджи -- Сеснок [label="255"];

Маджи -- Батерст [label="134"];

Маджи -- Литго [label="126"];

// Город Батерст

Батерст -- Литго [label="62"];

Батерст -- Кора [label="106"];

Батерст -- Круквелл [label="155"];

// Город Бурова

Бурова -- Янг [label="47"];

Бурова -- Кора [label="78"];

Бурова -- Круквелл [label="85"];

Бурова -- Гоулберн [label="140"];

// Город Круквелл

Круквелл -- Кетумба [label="229"];

Круквелл -- Гоулберн [label="43"];

// Город Гоулберн

Гоулберн -- Нора [label="125"];

Гоулберн -- Боурал [label="83"];

// Город Боурал

Боурал -- Нора [label="66"];

Боурал -- Киама [label="77"];

Боурал -- Вуллонгонг [label="82"];

Боурал -- Камден [label="68"];

// Город Киама

Киама -- Нора [label="55"];

Киама -- Вуллонгонг [label="43"];

// Город Вуллонгонг

Вуллонгонг -- Камден [label="72"];

Вуллонгонг -- Сидней [label="80"];

// Город Камден

Камден -- Кетумба [label="96"];

Камден -- Сидней [label="75"];

// Город Сидней

Сидней -- Кетумба [label="106"];

Сидней -- Вои_вои [label="81"];

// Город Вои_вои

Вои_вои -- Кетумба [label="156"];

Вои_вои -- Госфорд [label="16"];

// Город Госфорд

Госфорд -- Ньюкасл [label="84"];

Госфорд -- Сеснок [label="85"];

Госфорд -- Литго [label="174"];

// Город Сеснок

Сеснок -- Меррива [label="159"];

Сеснок -- Синглтон [label="53"];

Сеснок -- Ньюкасл [label="45"];

// Город Ньюкасл

Ньюкасл -- Мейтленд [label="34"];

Ньюкасл -- Дангог [label="104"];

// Город Мейтленд


```

Мейтленд -- Дангог [label="53"];
Мейтленд -- Синглтон [label="48"];

// Город Синглтон
Синглтон -- Маселбрук [label="49"];

// Город Меррива
Меррива -- Маселбрук [label="77"];
Меррива -- Кула [label="81"];

// Город Кула
Кула -- Данду [label="47"];
Кула -- Тураина [label="132"];

// Город Гилгандра
Гилгандра -- Тураина [label="49"];
Гилгандра -- Данду [label="92"];

// Город Кетумба
Кетумба -- Литго [label="41"];

// Настройка внешнего вида узлов
node [shape=circle, style=filled, color=green, fontcolor=black];
edge [color=black, penwidth=3.0];
}

```

После выполнения команды «dot -Tpng graph.dot -o graph.png» в командной строке был построен граф (рис. 5).

Был создан общедоступный репозиторий на «GitHub», клонирование репозитория, редактирование файла «.gitignore», организация репозитория согласно модели ветвления «git flow» (рис. 7).

```
C:\Program Files\Git>cd C:\Users\Andrey\Desktop\ИИ\Лабораторная_работа_1

C:\Users\Andrey\Desktop\ИИ\Лабораторная_работа_1>git clone https://github.com/AndreyPust/Artificial_Intelligence_laboratory_work_1
Cloning into 'Artificial_Intelligence_laboratory_work_1'...
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 5 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (5/5), done.

C:\Users\Andrey\Desktop\ИИ\Лабораторная_работа_1>cd C:\Users\Andrey\Desktop\ИИ\Лабораторная_работа_1\Artificial_Intelligence_laboratory_work_1
C:\Users\Andrey\Desktop\ИИ\Лабораторная_работа_1\Artificial_Intelligence_laboratory_work_1>git flow init
Which branch should be used for bringing forth production releases?
- main

C:\Users\Andrey\Desktop\ИИ\Лабораторная_работа_1\Artificial_Intelligence_laboratory_work_1>git flow feature start 1.0
Switched to a new branch 'feature/1.0'

Summary of actions:
- A new branch 'feature/1.0' was created, based on 'develop'
- You are now on branch 'feature/1.0'

Now, start committing on your feature. When done, use:

    git flow feature finish 1.0

C:\Users\Andrey\Desktop\ИИ\Лабораторная_работа_1\Artificial_Intelligence_laboratory_work_1>git branch
  develop
* feature/1.0
  main
```

Рисунок 7 – Создание репозитория

Проработка примеров лабораторной работы:

Для примеров лабораторной работы был создан отдельный модуль языка Python. Классы и методы примеров были проанализированы.

Абстрактный класс «Problem» будет использован для создания шаблонов конкретных задач. Каждая конкретная задача будет наследовать этот класс и переопределять его методы. В конструкторе этого класса под «initial» понимается начальное состояние, а под «goal» конечное состояние. Остальные методы являются абстрактными и меняются в зависимости от конкретной задачи. Методы «action_cost» и «h» предоставляют стандартные реализации для стоимости действия и эвристической функции соответственно.

Класс «Node» представляет конкретный узел в графе. Конструктор класса «Node» принимает текущее состояние «state», ссылку на родительский узел «parent», действие, которое привело к этому узлу «action», и стоимость пути «path_cost». Далее создаются специальные узлы для обозначения неудачи в поиске «failure» и обрезания поиска «cutoff» в алгоритмах, таких как итеративное углубление.

Функция «expand» расширяет узел генерируя дочерние узлы.

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

Функция «path_states» возвращает последовательность состояний, ведущих к данному узлу.

Код данных классов, функций, а также класс очереди «PriorityQueue»:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

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)

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]

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)

```

Выполнение индивидуального задания:

Необходимо методом полного перебора решить задачу «коммивояжера» для начального населенного пункта для данного графа (города Австралии), то есть необходимо посетить все города ровно один раз и вернуться в исходный город по минимальному (по длине) маршруту, то есть найти гамильтонов цикл, суммарный вес которого минимален.

Так как исходя из источника информации, задача «коммивояжера» методом полного перебора относится к трансвычислительным, то граф из 49

узлов для решения данной задачи не подходит, по крайней мере задача не будет решаться за приемлемое время. Граф городов Австралии был сокращен до 11 узлов, код описания графа на языке DOT:

```
graph G {
    layout="neato";
    size="15,5!";
    overlap=false;
    splines=true;
    nodesep=0.6;
    ranksep=1.0;
    concentrate=true;

    "Буриндал" -- "Уоррен" [label="271"];
    "Буриндал" -- "Нинган" [label="156"];
    "Буриндал" -- "Кобар" [label="204"];

    "Буриндал" -- "Нарромин" [label="41"];

    "Нарромин" -- "Гилгандра" [label="100"];

    "Гилгандра" -- "Нинган" [label="99"];

    "Уоррен" -- "Гилгандра" [label="103"];
    "Уоррен" -- "Нарромин" [label="86"];
    "Уоррен" -- "Нинган" [label="78"];

    "Нинган" -- "Кобар" [label="86"];
    "Нинган" -- "Наймаджи" [label="122"];
    "Нинган" -- "Канбелого" [label="86"];

    "Кобар" -- "Канбелого" [label="50"];
    "Кобар" -- "Наймаджи" [label="97.5"];
    "Кобар" -- "Гулгуния" [label="110"];
```

```
"Канбелего" -- "Наймаджи" [label="61"];
```

```
"Наймаджи" -- "Гулгуния" [label="46"];
```

```
}
```

После построения данного графа, сокращенная версия предоставлена на рисунке 8.



Рисунок 8 – Граф городов Австралии из 11 узлов

Для данного графа была решена задача «коммивояжера» методом полного перебора. Код программы решения данной задачи на языке Python:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import math
from itertools import permutations

class Problem:
    """
    Формальное описание задачи, задаем узлы графа и связи между ними.
    """

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

    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

failure = Node('failure', path_cost=math.inf)
cutoff = Node('cutoff', path_cost=math.inf)

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]

class TSPPProblem(Problem):
    """
    Наследует Problem, задавая методы для действий, переходов и стоимости на
    основе структуры графа.
    """

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

    def actions(self, state):
        return [neighbor for neighbor in self.graph[state]]

```

```

def result(self, state, action):
    return action

def action_cost(self, s, a, sl):
    return self.graph[s][sl]

def solve_tsp(problem):
    """
    Функция реализующая метод полного перебора задачи коммивояжера.
    :param problem: задача.
    :return: список городов по порядку и длина пути.
    """

    nodes = list(problem.graph.keys())
    nodes.remove(problem.initial)
    min_path_cost = math.inf
    min_path = None

    for perm in permutations(nodes):
        path = [problem.initial] + list(perm) + [problem.initial]
        cost = sum(problem.action_cost(path[i], None, path[i + 1]) for i in
range(len(path) - 1))

        if cost < min_path_cost:
            min_path_cost = cost
            min_path = path

    return min_path, min_path_cost

if __name__ == "__main__":

    # Задаем граф городов Австралии
    graph = {
        "Буриндал": {"Уоррен": 271, "Нинган": 156, "Кобар": 204, "Нарромин":
41, "Гилгандра": float('inf'),
            "Канбелего": float('inf'), "Наймаджи": float('inf'),
        "Гулгуния": float('inf')},
        "Уоррен": {"Буриндал": 271, "Гилгандра": 103, "Нарромин": 86,
        "Нинган": 78, "Кобар": float('inf'),
            "Канбелего": float('inf'), "Наймаджи": float('inf'),
        "Гулгуния": float('inf')},
        "Нинган": {"Буриндал": 156, "Кобар": 86, "Наймаджи": 122, "Уоррен":
78, "Канбелего": 86, "Гилгандра": 99,
            "Нарромин": float('inf'), "Гулгуния": float('inf')},
        "Кобар": {"Буриндал": 204, "Канбелего": 50, "Наймаджи": 97.5,
        "Гулгуния": 110, "Нинган": 86,
            "Гилгандра": float('inf'), "Нарромин": float('inf'),
        "Уоррен": float('inf')},
        "Канбелего": {"Кобар": 50, "Наймаджи": 61, "Нинган": 86, "Буриндал":
float('inf'), "Гилгандра": float('inf'),
            "Нарромин": float('inf'), "Уоррен": float('inf'),
        "Гулгуния": float('inf')},
        "Наймаджи": {"Кобар": 97.5, "Нинган": 122, "Гулгуния": 46,
        "Канбелего": 61, "Буриндал": float('inf'),
            "Гилгандра": float('inf'), "Нарромин": float('inf'),
        "Уоррен": float('inf')},
        "Гулгуния": {"Кобар": 110, "Наймаджи": 46, "Буриндал": float('inf'),
        "Гилгандра": float('inf'),
            "Нарромин": float('inf'), "Уоррен": float('inf'),
        "Канбелего": float('inf'),
            "Нинган": float('inf')},
        "Гилгандра": {"Нарромин": 100, "Уоррен": 103, "Нинган": 99,

```

```

"Буриндал": float('inf'), "Канбелего": float('inf'),
    "Наймаджи": float('inf'), "Гулгуния": float('inf'),
"Кобар": float('inf')},
    "Нарромин": {"Буриндал": 41, "Уоррен": 86, "Гилгандра": 100,
"Канбелего": float('inf'),
    "Наймаджи": float('inf'), "Гулгуния": float('inf'),
"Нинган": float('inf'), "Кобар": float('inf')}
}

# Создаем объект задачи и находим решение
tsp_problem = TSPProblem("Буриндал", "Буриндал", graph)
solution_path, solution_cost = solve_tsp(tsp_problem)

print("Минимальный путь для задачи коммивояжера:", solution_path)
print("Стоимость минимального пути:", solution_cost)

```

Заданный граф описывается в виде словаря, если между городами нет дороги, то вес ребра задается в виде бесконечности.

В результате выполнения данного кода был найден кратчайший путь, проходящий через все города по одному разу, и длина этого пути (рис. 9). Кратчайший путь состоит из городов: Буриндал, Кобар, Гулгуния, Наймаджи, Канбелего, Нинган, Уоррен, Гилгандра, Нарромин, Буриндал. Длина данного пути составила 829 км.

```

C:\Users\Анна\OneDrive\Documents\envs\AI_lab_1\python.exe C:\Users\Анна\OneDrive\Documents\AI_lab_1\main.py
Минимальный путь для задачи коммивояжера: ['Буриндал', 'Кобар', 'Гулгуния',
Стоимость минимального пути: 829

Process finished with exit code 0

```

Рисунок 9 – Результаты работы программы

Данный путь был обозначен на построенном графе (рис. 10).

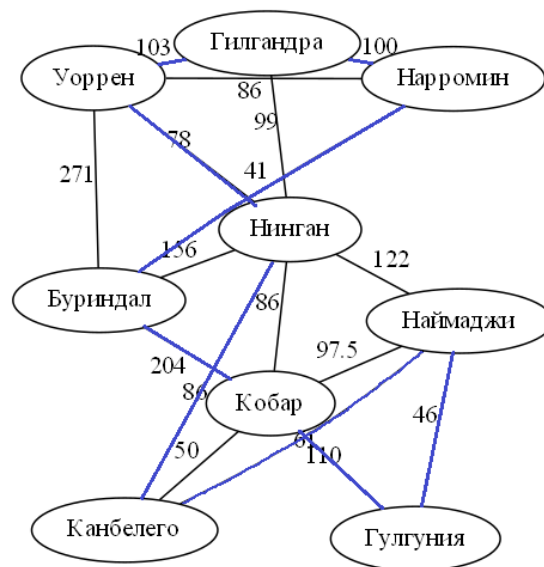


Рисунок 10 – Гамильтонов цикл

Ссылка на репозиторий лабораторной работы на GitHub:

https://github.com/AndreyPust/Artificial_Intelligence_laboratory_work_1.git

t

Ответы на контрольные вопросы:

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

Метод "слепого поиска" в ИИ — это метод, который не использует дополнительную информацию о целевом состоянии для оптимизации поиска.

Примеры включают поиск в ширину и в глубину, где каждый узел проверяется без учета его близости к цели.

2. Как отличается эвристический поиск от слепого поиска?

Эвристический поиск отличается тем, что использует эвристику — оценочную функцию, которая позволяет алгоритму более эффективно выбирать, какие узлы проверять, основываясь на предполагаемом расстоянии до цели. Эвристический поиск может находить путь быстрее, чем слепой поиск.

3. Какую роль играет эвристика в процессе поиска?

Эвристика играет роль оценки "расстояния" или "стоимости" от текущего состояния до целевого, помогая алгоритму сосредоточиться на более перспективных путях и, таким образом, ускоряя процесс поиска.

4. Приведите пример применения эвристического поиска в реальной задаче.

Пример применения эвристического поиска — навигация GPS, где алгоритм находит кратчайший или оптимальный маршрут на основе данных о расстояниях и текущей дорожной обстановке.

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

Полное исследование всех возможных ходов в шахматах затруднительно из-за огромного количества возможных комбинаций, что делает перебор всех ходов непрактичным даже для мощных компьютеров.

6. Какие факторы ограничивают создание идеального шахматного ИИ?

Факторы, ограничивающие создание идеального шахматного ИИ, включают ограничения вычислительных мощностей и времени, а также сложность оценки позиций, где требуется глубина понимания, приближенная к человеческой интуиции.

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

Основная задача ИИ при выборе ходов в шахматах — это определение оптимальных ходов, которые приводят к преимуществу в позиции, с учетом ограничений времени и вычислительных ресурсов.

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

Балансировка между скоростью вычислений и нахождением оптимальных решений достигается за счет эвристик и ограничений глубины поиска, которые позволяют ИИ быстро принимать приемлемые, но не всегда идеальные решения.

9. Каковы основные элементы задачи поиска маршрута по карте?

Основные элементы задачи поиска маршрута по карте — это начальная точка, целевая точка, граф дорог (или маршрутов), а также возможные ограничения, такие как расстояние и доступность дорог.

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

Оптимальность решения задачи маршрутизации на карте Румынии оценивается по следующим критериям:

Длина маршрута — кратчайшее расстояние между начальным и конечным пунктами.

Стоимость маршрута — учет дополнительных факторов, таких как время и расход топлива.

Эвристическая оценка — использование функции, которая помогает находить путь быстрее и ближе к оптимальному.

Эти критерии позволяют выбрать наилучший маршрут в зависимости от задачи.

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

Исходное состояние дерева поиска в задаче маршрутизации — это город, из которого начинается поиск, например, Арад, если цель — найти путь в Бухарест.

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

Листовые узлы — это узлы, не имеющие потомков в дереве, что означает, что для них больше нет доступных действий для расширения.

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

Этап расширения узла включает создание дочерних узлов, представляющих возможные состояния, достижимые из текущего узла с помощью доступных действий.

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

Города, которые можно посетить из Арада за одно действие: Зеринд, Сибиу и Тимишоара.

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

Целевое состояние в алгоритме поиска определяется как состояние, в котором выполнено условие задачи, например, достижение города Бухарест.

16. Какие основные шаги выполняет алгоритм поиска по дереву?

Основные шаги алгоритма поиска по дереву включают выбор узла для расширения, проверку, является ли он целевым состоянием, и, если нет, создание дочерних узлов.

17. Чем различаются состояния и узлы в дереве поиска?

Состояния и узлы различаются тем, что узел — это представление состояния в дереве поиска, которое включает дополнительную информацию, такую как путь и затраты.

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

Функция преемника генерирует возможные состояния, достижимые из текущего состояния, и используется для расширения узлов в дереве.

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

Параметры b (разветвление), d (глубина решения) и m (максимальная глубина) влияют на время и ресурсы, необходимые для поиска. Увеличение b и d значительно увеличивает количество узлов для проверки.

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

Алгоритмы поиска оцениваются по критериям полноты (способности найти решение, если оно существует), временной и пространственной сложности, а также оптимальности (нахождения наилучшего решения).

21. Какую роль выполняет класс `Problem` в приведенном коде?

Класс `Problem` в приведенном коде задает структуру задачи поиска, включая начальное состояние, целевое состояние и методы для выполнения действий и проверки цели.

22. Какие методы необходимо переопределить при наследовании класса `Problem`?

Методы, которые нужно переопределить в классе `Problem` при наследовании: `actions`, `result`, возможно, `is_goal` и `action_cost`.

23. Что делает метод `is_goal` в классе `Problem`?

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

24. Для чего используется метод `action_cost` в классе `Problem`?

Метод `action_cost` возвращает стоимость выполнения действия между состояниями и используется для расчета общей стоимости пути.

25. Какую задачу выполняет класс Node в алгоритмах поиска?

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

26. Какие параметры принимает конструктор класса Node?

Конструктор класса Node принимает параметры: state (состояние узла), parent (родительский узел), action (действие, которое привело к узлу), path_cost (стоимость пути до узла).

27. Что представляет собой специальный узел failure?

Специальный узел failure указывает, что алгоритм не смог найти решение для задачи.

28. Для чего используется функция expand в коде?

Функция expand создает дочерние узлы для текущего узла, основываясь на возможных действиях и результатах.

29. Какая последовательность действий генерируется с помощью функции path_actions?

Функция path_actions генерирует последовательность действий для достижения данного узла от начального состояния.

30. Чем отличается функция path_states от функции path_actions?

Функция path_states возвращает последовательность состояний, в то время как path_actions — последовательность действий.

31. Какой тип данных используется для реализации FIFOQueue?

Тип данных для реализации FIFOQueue — это deque из библиотеки collections, обеспечивающая быструю вставку и удаление с обоих концов.

32. Чем отличается очередь FIFOQueue от LIFOQueue?

Очередь `FIFOQueue` (первый пришел, первый ушел) работает по принципу очереди, в то время как `LIFOQueue` (последний пришел, первый ушел) — по принципу стека.

33. Как работает метод `add` в классе `PriorityQueue`?

Метод `add` в классе `PriorityQueue` добавляет элемент с приоритетом, чтобы обеспечить доступ к элементам с наименьшим значением приоритетной функции.

34. В каких ситуациях применяются очереди с приоритетом?

Очереди с приоритетом используются, когда элементы должны обрабатываться в порядке важности, например, в алгоритмах поиска по наименьшей стоимости.

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

Функция `heappop` извлекает элемент с наименьшим значением приоритетной функции и помогает в реализации очереди с приоритетом с минимальным значением на вершине.

Вывод: в ходе выполнения лабораторной работы были приобретены навыки по работе с методами поиска в пространстве состояний с помощью языка программирования Python, а также изучена суть задачи «коммивояжера» и метод полного перебора для ее решения. Метод полного перебора для решения задачи является малоэффективным, так как при сложном графе с количеством узлов больше 20-ти скорость выполнения алгоритма начинает расти экспоненциально.