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 №3

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 поиска в глубину

Цель: приобрести навыки по работе с поиском в глубину с помощью языка программирования Python версии 3.x.

Ход работы:

Поиск в глубину

Для построенного графа городов Австралии (рис. 1) лабораторной работы 1 была написана программа на языке программирования Python, которая с помощью алгоритма поиска в глубину находит минимальное расстояние между начальным и конечным пунктами (для лабораторной работы 1 начальным пунктом являлся город Буриндал, а конечный город Сидней).

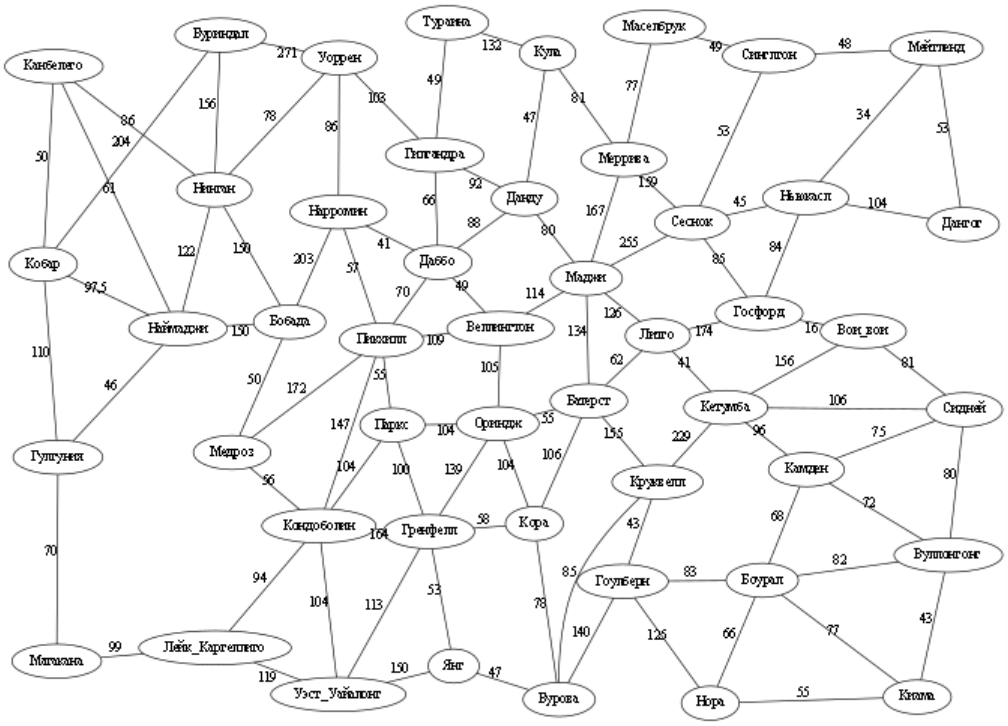

Рисунок 1 – Граф городов Австралии на языке DOT

Граф в программе был описан в виде словаря словарей с узлами и ребрами графа. Код программы нахождения кратчайшего пути в графе:

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

import math
from abc import ABC, abstractmethod


class Problem(ABC):
    """
```

```
    Абстрактный класс для формальной постановки задачи.
    Новый домен (конкретная задача) должен специализировать этот класс,
    переопределяя методы actions и result, а при необходимости
    action_cost, h и is_goal.
    """

    def __init__(self, initial=None, goal=None, **kwargs):
        self.initial = initial
        self.goal = goal
        # Сохраняем все остальные переданные параметры (при желании).
        for k, v in kwargs.items():
            setattr(self, k, v)

    @abstractmethod
    def actions(self, state):
        """Вернуть доступные действия (операторы) из данного состояния."""
        pass

    @abstractmethod
    def result(self, state, action):
        """Вернуть результат применения действия к состоянию."""
        pass

    def is_goal(self, state):
        """Проверка, является ли состояние целевым."""
        return state == self.goal

    def action_cost(self, s, a, s1):
        """
        Возвращает стоимость применения действия a,
        переводящего состояние s в состояние s1.
        По умолчанию = 1.
        """
        return 1

    def h(self, node):
        """Эвристическая функция; по умолчанию = 0."""
        return 0

    def __str__(self):
        return f"{type(self).__name__}({self.initial!r}, {self.goal!r})"


class Node:
    """Узел в дереве поиска."""
    def __init__(self, state, parent=None, action=None, path_cost=0.0):
        self.state = state       # Текущее состояние
        self.parent = parent     # Родительский узел
        self.action = action     # Действие, которое привело к этому узлу
        self.path_cost = path_cost  # Стоимость пути от начального узла

    def __repr__(self):
        return f"<Node {self.state}>"

    def __lt__(self, other):
        """Сравнение узлов (для приоритетных очередей), здесь
необязательно."""
        return self.path_cost < other.path_cost

    # Глубина узла — длина пути от корня (получаем рекурсивно)
    def __len__(self):
        if self.parent is None:
            return 0
        else:
```

```
            return 1 + len(self.parent)


# Специальные «сигнальные» узлы
failure = Node("failure", path_cost=math.inf)
cutoff = Node("cutoff",  path_cost=math.inf)


def expand(problem, node):
    """
    Генерируем (расширяем) дочерние узлы, применяя все действия к состоянию
node.state.
    """
    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(state=s1, parent=node, action=action, path_cost=cost)


def path_actions(node):
    """
    Восстанавливаем последовательность действий от корня до данного узла.
    """
    if node.parent is None:
        return []
    return path_actions(node.parent) + [node.action]


def path_states(node):
    """
    Восстанавливаем последовательность состояний (городов)
    от корня до данного узла.
    """
    if node.parent is None:
        return [node.state]
    return path_states(node.parent) + [node.state]


def is_cycle(node):
    """
    Функция, которая проверяет на зацикливание.
    :param node: Узел.
    :return: Результат проверки.
    """

    s = node.state
    p = node.parent
    while p is not None:
        if p.state == s:
            return True
        p = p.parent
    return False


def depth_first_recursive_search(problem, node=None):
    """
    Рекурсивный поиск в глубину.
    Если решение найдено, возвращаем узел.
    Если зациклились или не нашли решения по всем ветвям, возвращаем failure.
    """

    if node is None:
        node = Node(problem.initial)
```

```
    if problem.is_goal(node.state):
        return node

    # Проверка на цикл
    if is_cycle(node):
        return failure

    for child in expand(problem, node):
        result = depth_first_recursive_search(problem, child)
        if result:  # нашли решение
            return result

    # Если все потомки безрезультатны
    return failure


class MapProblem(Problem):
    """
    Дочерний класс для постановки задачи:
    Найти путь из одного города в другой по графу.
    """

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

    def actions(self, state):
        """
        Все соседние города, куда есть ребро state->neighbor.
        """
        return list(self.graph[state].keys())

    def result(self, state, action):
        """
        Перейти в город 'action'.
        """
        return action

    def action_cost(self, s, a, s1):
        """
        Стоимость (расстояние) из s в s1.
        """
        return self.graph[s][s1]


def main():
    """
    Главная функция программы.
    """

    # Граф городов Австралии
    graph = {
        "Буриндал": {"Уоррен": 271, "Нинган": 156, "Кобар": 204},
        "Уоррен": {"Буриндал": 271, "Нинган": 78, "Гилгандра": 103,
"Нарромин": 86},
        "Нинган": {"Буриндал": 156, "Канбелего": 86, "Уоррен": 78, "Бобада":
150, "Наймаджи": 122},
        "Кобар": {"Буриндал": 204, "Канбелего": 50, "Наймаджи": 97.5,
"Гулгуния": 110},
        "Канбелего": {"Кобар": 50, "Наймаджи": 61, "Нинган": 86},
        "Наймаджи": {"Канбелего": 61, "Нинган": 122, "Бобада": 150, "Кобар":
97.5, "Гулгуния": 46},
        "Гулгуния": {"Кобар": 110, "Наймаджи": 46, "Матакана": 70},
```

```
        "Матакана": {"Гулгуния": 70, "Лейк_Каргеллиго": 99},
        "Бобада": {"Наймаджи": 150, "Нинган": 150, "Нарромин": 203, "Медроз":
50},
        "Нарромин": {"Уоррен": 86, "Бобада": 203, "Пикхилл": 57, "Даббо":
41},
        "Пикхилл": {"Нарромин": 57, "Даббо": 70, "Веллингтон": 109, "Медроз":
172, "Кондоболин": 147, "Паркс": 55},
        "Даббо": {"Нарромин": 41, "Гилгандра": 66, "Данду": 88, "Веллингтон":
49, "Пикхилл": 70},
        "Гилгандра": {"Уоррен": 103, "Даббо": 66, "Тураина": 49, "Данду":
92},
        "Данду": {"Даббо": 88, "Маджи": 80, "Гилгандра": 92, "Кула": 47},
        "Медроз": {"Бобада": 50, "Кондоболин": 56, "Пикхилл": 172},
        "Кондоболин": {
            "Медроз": 56,
            "Паркс": 104,
            "Гренфелл": 164,
            "Уэст_Уайалонг": 104,
            "Лейк_Каргеллиго": 94,
            "Пикхилл": 147,
        },
        "Паркс": {"Пикхилл": 55, "Кондоболин": 104, "Гренфелл": 100,
"Ориндж": 104},
        "Гренфелл": {"Паркс": 100, "Ориндж": 139, "Кора": 58, "Янг": 53,
"Кондоболин": 164, "Уэст_Уайалонг": 113},
        "Ориндж": {"Кора": 104, "Паркс": 104, "Веллингтон": 105, "Батерст":
55, "Гренфелл": 139},
        "Веллингтон": {"Маджи": 114, "Пикхилл": 109, "Даббо": 49, "Ориндж":
105},
        "Маджи": {"Веллингтон": 114, "Данду": 80, "Меррива": 167, "Сеснок":
255, "Батерст": 134, "Литго": 126},
        "Батерст": {"Литго": 62, "Кора": 106, "Круквелл": 155, "Ориндж": 55,
"Маджи": 134},
        "Литго": {"Батерст": 62, "Кетумба": 41, "Маджи": 126, "Госфорд":
174},
        "Кора": {"Гренфелл": 58, "Ориндж": 104, "Батерст": 106, "Бурова":
78},
        "Бурова": {"Янг": 47, "Кора": 78, "Круквелл": 85, "Гоулберн": 140},
        "Круквелл": {"Батерст": 155, "Кетумба": 229, "Гоулберн": 43,
"Бурова": 85},
        "Гоулберн": {"Бурова": 140, "Круквелл": 43, "Нора": 125, "Боурал":
83},
        "Нора": {"Гоулберн": 125, "Боурал": 66, "Киама": 55},
        "Боурал": {"Нора": 66, "Киама": 77, "Вуллонгонг": 82, "Камден": 68,
"Гоулберн": 83},
        "Киама": {"Нора": 55, "Вуллонгонг": 43, "Боурал": 77},
        "Вуллонгонг": {"Киама": 43, "Боурал": 82, "Камден": 72, "Сидней":
80},
        "Камден": {"Боурал": 68, "Вуллонгонг": 72, "Кетумба": 96, "Сидней":
75},
        "Сидней": {"Вуллонгонг": 80, "Камден": 75, "Кетумба": 106, "Вои_вои":
81},
        "Кетумба": {"Литго": 41, "Сидней": 106, "Камден": 96, "Вои_вои": 156,
"Круквелл": 229},
        "Вои_вои": {"Сидней": 81, "Кетумба": 156, "Госфорд": 16},
        "Госфорд": {"Вои_вои": 16, "Ньюкасл": 84, "Сеснок": 85, "Литго":
174},
        "Ньюкасл": {"Госфорд": 84, "Сеснок": 45, "Мейтленд": 34, "Дангог":
104},
        "Сеснок": {"Госфорд": 85, "Ньюкасл": 45, "Меррива": 159, "Синглтон":
53, "Маджи": 255},
        "Мейтленд": {"Ньюкасл": 34, "Дангог": 53, "Синглтон": 48},
        "Дангог": {"Ньюкасл": 104, "Мейтленд": 53},
        "Синглтон": {"Сеснок": 53, "Мейтленд": 48, "Маселбрук": 49},
```

```
        "Маселбрук": {"Синглтон": 49, "Меррива": 77},
        "Меррива": {"Сеснок": 159, "Маселбрук": 77, "Кула": 81, "Маджи":
167},
        "Кула": {"Меррива": 81, "Данду": 47, "Тураина": 132},
        "Тураина": {"Кула": 132, "Гилгандра": 49},
        "Лейк_Каргеллиго": {"Матакана": 99, "Кондоболин": 94,
"Уэст_Уайалонг": 119},
        "Уэст_Уайалонг": {"Лейк_Каргеллиго": 119, "Кондоболин": 104,
"Гренфелл": 113, "Янг": 150},
        "Янг": {"Уэст_Уайалонг": 150, "Гренфелл": 53, "Бурова": 47},
    }

    problem = MapProblem("Буриндал", "Сидней", graph)

    solution_node = depth_first_recursive_search(problem)

    if solution_node is None or solution_node is failure:
        print("Путь не найден!")
    else:
        # Восстановим последовательность городов
        route = path_states(solution_node)
        print("Маршрут:", route)

        # Восстановим последовательность действий
        acts = path_actions(solution_node)
        print("Последовательность действий:", acts)
        print("Суммарная стоимость:", solution_node.path_cost)


if __name__ == "__main__":
    main()
```

Путь от начального города «Буриндал» до конечного «Сидней» был найден, но он не совпал с кратчайшим путем, полученным с помощью поиска в ширину. Таким образом алгоритм нашел путь, но он не является кратчайшим. Путь на графе, полученный с помощью поиска в ширину (оптимальный путь, кратчайший) (рис. 2). Таким образом, поиск в глубину не гарантирует нахождения наиоптимальнейшего решения.

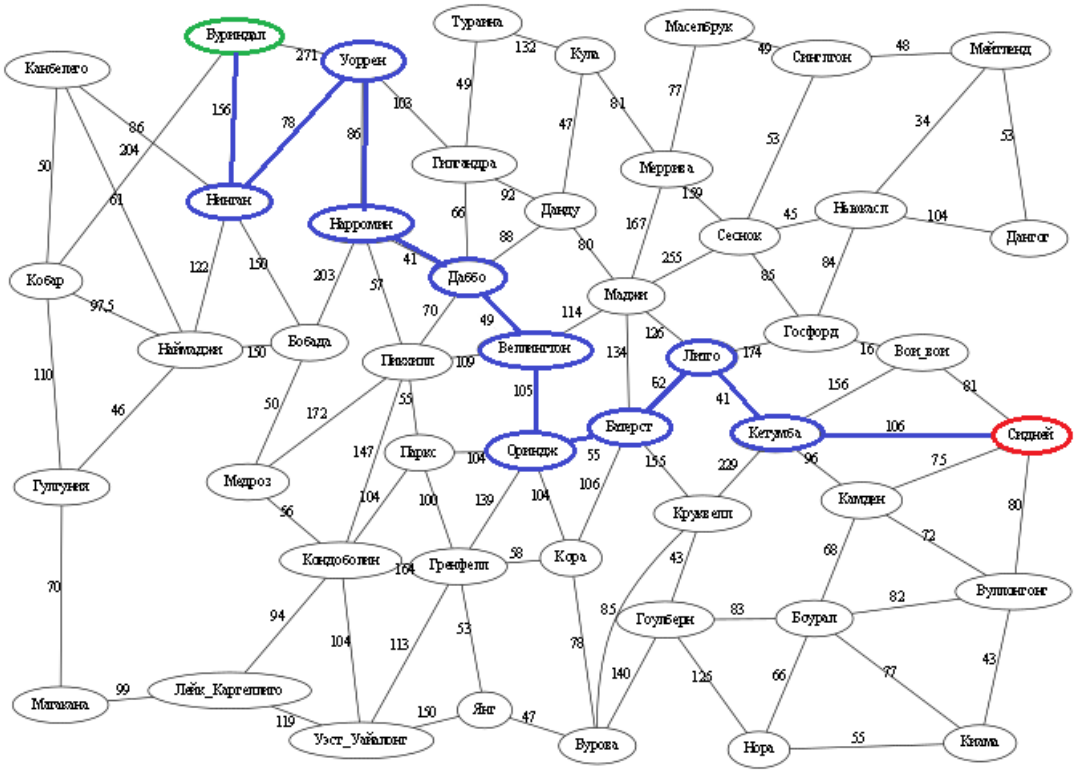

Рисунок 2 – Кратчайший путь на графе

Решение, полученное с помощью поиска в глубину (рис. 3) и его изображение на графе (рис. 4).

```
C:\Users\Andrey\AppData\Local\pypoetry\Cache\virtualenvs\ai-lab-2-dtOMEYOu-py3.12\Scripts\python.exe C:\Users\Andrey\Desktop\ИИ\Ла
Маршрут: ['Буриндал', 'Уоррен', 'Нинган', 'Канбелего', 'Кобар', 'Наймаджи', 'Бобада', 'Нарромин', 'Пикхилл', 'Даббо', 'Гилгандра',
Суммарная стоимость: 1731.5

Process finished with exit code 0
```

Рисунок 3 – Результат работы программы с поиском в глубину на графе городов Австралии

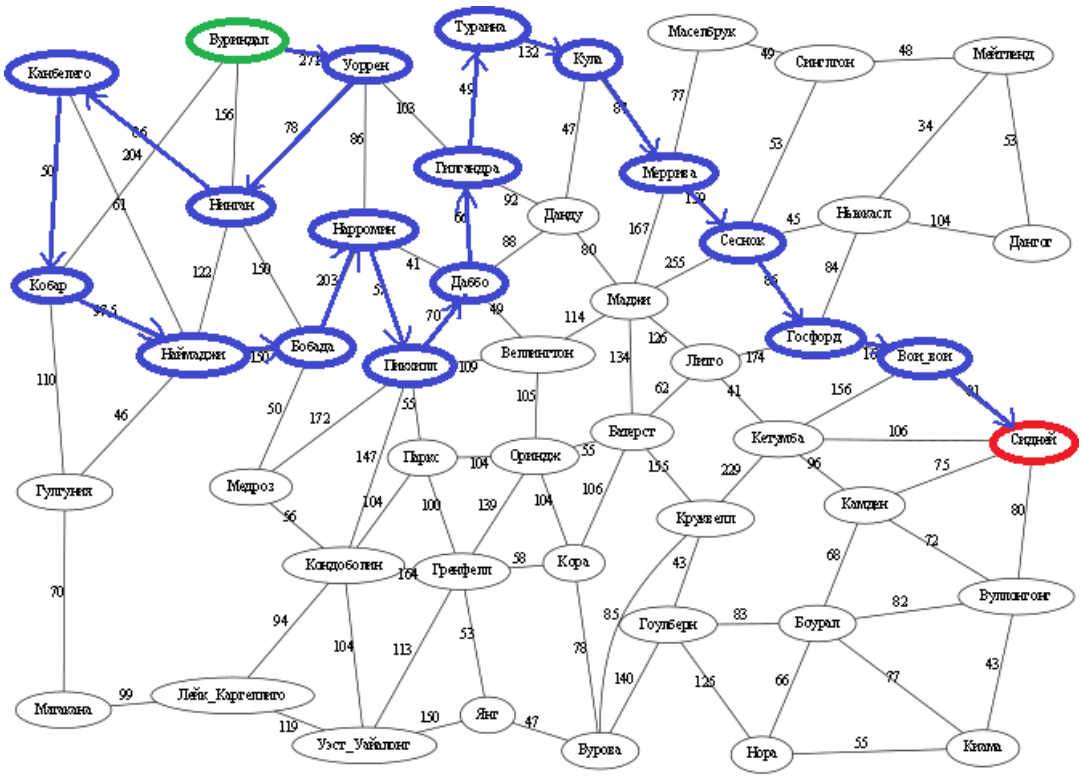

Рисунок 2 – Решение, найденное с помощью поиска в глубину

Алгоритм заливки

Необходимо создать программу с реализацией алгоритма заливки. Задача заключается в том, чтобы, зная узел заливки и цвета замены, закрасить на указанный цвет все узлы, соседствующие с узлом заливки и имеющие такой же цвет. Код программы с алгоритмом заливки:

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

import math
from abc import ABC, abstractmethod

"""
Необходимо создать программу с реализацией алгоритма заливки.
Задача заключается в том, чтобы, зная узел заливки и цвета замены,
закрасить на указанный цвет все узлы, соседствующие с узлом
заливки и имеющие такой же цвет.
"""


class Problem(ABC):
    """
    Абстрактный класс для формальной постановки задачи.
    Новый домен (конкретная задача) должен специализировать этот класс,
    переопределяя методы actions и result, а при необходимости
    action_cost, h и is_goal.
    """

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

```
        for k, v in kwargs.items():
            setattr(self, k, v)

    @abstractmethod
    def actions(self, state):
        """
        Вернуть доступные действия (операторы) из данного состояния.
        """
        pass

    @abstractmethod
    def result(self, state, action):
        """
        Вернуть результат применения действия к состоянию.
        """
        pass

    def is_goal(self, state):
        """
        Проверка на достижение цели.
        """
        return False

    def action_cost(self, s, a, s1):
        """
        Стоимость перехода; для заливки несущественно, оставим 1.
        """
        return 1

    def h(self, node):
        """
        Эвристическая функция, по умолчанию 0.
        """
        return 0


class Node:
    def __init__(self, state, parent=None, action=None, path_cost=0.0):
        self.state = state
        self.parent = parent
        self.action = action
        self.path_cost = path_cost

    def __repr__(self):
        return f"<Node {self.state}>"

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


# Специальные «сигнальные» узлы
failure = Node('failure', path_cost=math.inf)
cutoff = Node('cutoff',  path_cost=math.inf)


def expand(problem, node):
    """
    Генерация дочерних узлов, применяя actions к node.state.
    """

    s = node.state
    for action in problem.actions(s):
        s_next = problem.result(s, action)
        yield Node(state=s_next, parent=node, action=action)
```

```python
class FloodFillProblem(Problem):
    """
    Задача алгоритма заливки:
    matrix: двумерный список (rows x cols),
    start: начальная координата (row, col),
    target_color: цвет, который нужно заменить,
    replacement_color: цвет, на который заменяем.
    """

    def __init__(self, matrix, start, target_color, replacement_color):
        super().__init__(initial=start)
        self.matrix = matrix
        self.rows = len(matrix)
        self.cols = len(matrix[0]) if self.rows > 0 else 0

        self.target_color = target_color
        self.replacement_color = replacement_color

    def actions(self, state):
        """
        Выдаём список соседей по 4-м направлениям (вверх, вниз, влево,
вправо),
        которые ещё имеют цвет target_color.
        """

        (r, c) = state
        neighbors = []
        deltas = [(0, 1), (0, -1), (1, 0), (-1, 0)]
        for dr, dc in deltas:
            nr, nc = r + dr, c + dc
            if 0 <= nr < self.rows and 0 <= nc < self.cols:
                if self.matrix[nr][nc] == self.target_color:
                    neighbors.append((nr, nc))
        return neighbors

    def result(self, state, action):
        (nr, nc) = action
        self.matrix[nr][nc] = self.replacement_color
        return action

    def is_goal(self, state):
        """
        В задаче заливки нет одной цели.
        """
        return False


def flood_fill_dfs(problem):
    """
    Алгоритм заливки по принципу поиска в глубину.
    Начинаем с problem.initial, затем красим начальную клетку
    (если она имеет target_color), затем рекурсивно обходим соседей.
    """

    start_state = problem.initial
    r, c = start_state

    # Если клетка уже залита
    if problem.matrix[r][c] != problem.target_color:
        return

    # Красим начальную клетку
```

```
    problem.matrix[r][c] = problem.replacement_color

    # Реализация поиска в глубину
    stack = [(r, c)]
    while stack:
        (cr, cc) = stack.pop()

        for (nr, nc) in problem.actions((cr, cc)):
            problem.result((cr, cc), (nr, nc))
            stack.append((nr, nc))


def main():
    """
    Главная функция программы.
    """

    # Матрица для заливки
    matrix = [
        ["Y", "Y", "Y", "G", "G", "G", "G", "G", "G", "G"],
        ["Y", "Y", "Y", "Y", "Y", "Y", "G", "X", "X", "X"],
        ["G", "G", "G", "G", "G", "G", "G", "X", "X", "X"],
        ["W", "W", "W", "W", "W", "G", "G", "G", "G", "X"],
        ["W", "R", "R", "R", "R", "R", "G", "X", "X", "X"],
        ["W", "W", "W", "R", "R", "G", "G", "X", "X", "X"],
        ["W", "B", "W", "R", "R", "R", "R", "R", "R", "X"],
        ["W", "B", "B", "B", "B", "R", "R", "X", "X", "X"],
        ["W", "B", "B", "X", "B", "B", "B", "B", "X", "X"],
        ["W", "B", "B", "X", "X", "X", "X", "X", "X", "X"],
    ]
    print("Изначальная матрица: ")
    for row in matrix:
        print(row)

    # Узел заливки
    start_node = (3, 9)

    # Цвет, который нужно поменять
    target_color = "X"

    # Цвет, на который нужно поменять
    replacement_color = "C"

    problem = FloodFillProblem(matrix, start_node, target_color,
replacement_color)

    flood_fill_dfs(problem)

    # Выводим результат после заливки
    print("Закрашенная матрица: ")
    for row in matrix:
        print(row)


if __name__ == "__main__":
    main()
```

Результаты работы программы при заданной матрице, заданном узле заливки и цветах заливки (рис. 3).

```
Изначальная матрица:
['Y', 'Y', 'Y', 'G', 'G', 'G', 'G', 'G', 'G', 'G']
['Y', 'Y', 'Y', 'Y', 'Y', 'Y', 'G', 'X', 'X', 'X']
['G', 'G', 'G', 'G', 'G', 'G', 'G', 'X', 'X', 'X']
['W', 'W', 'W', 'W', 'W', 'G', 'G', 'G', 'G', 'X']
['W', 'R', 'R', 'R', 'R', 'R', 'G', 'X', 'X', 'X']
['W', 'W', 'W', 'R', 'R', 'G', 'G', 'X', 'X', 'X']
['W', 'B', 'W', 'R', 'R', 'R', 'R', 'R', 'R', 'X']
['W', 'B', 'B', 'B', 'B', 'R', 'R', 'X', 'X', 'X']
['W', 'B', 'B', 'X', 'B', 'B', 'B', 'B', 'X', 'X']
['W', 'B', 'B', 'X', 'X', 'X', 'X', 'X', 'X', 'X']
Узел заливки:  (3, 9)
Цвет, который нужно поменять:  X
Цвет, на который нужно поменять:  C
Закрашенная матрица:
['Y', 'Y', 'Y', 'G', 'G', 'G', 'G', 'G', 'G', 'G']
['Y', 'Y', 'Y', 'Y', 'Y', 'Y', 'G', 'C', 'C', 'C']
['G', 'G', 'G', 'G', 'G', 'G', 'G', 'C', 'C', 'C']
['W', 'W', 'W', 'W', 'W', 'G', 'G', 'G', 'G', 'C']
['W', 'R', 'R', 'R', 'R', 'R', 'G', 'C', 'C', 'C']
['W', 'W', 'W', 'R', 'R', 'G', 'G', 'C', 'C', 'C']
['W', 'B', 'W', 'R', 'R', 'R', 'R', 'R', 'R', 'C']
['W', 'B', 'B', 'B', 'B', 'R', 'R', 'C', 'C', 'C']
['W', 'B', 'B', 'C', 'B', 'B', 'B', 'B', 'C', 'C']
['W', 'B', 'B', 'C', 'C', 'C', 'C', 'C', 'C', 'C']
```

Рисунок 3 – Результаты работы программы алгоритма заливки

Поиск самого длинного пути в матрице

Дана матрица символов размером M×N. Необходимо найти длину самого длинного пути в матрице, начиная с заданного символа. Каждый следующий символ в пути должен алфавитно следовать за предыдущим без пропусков. Поиск возможен во всех восьми направлениях.

Код программы поиска самого длинного пути в матрице:

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

from abc import ABC, abstractmethod


class Problem(ABC):
    """
    Абстрактный класс для формальной постановки задачи.
    Новый домен (конкретная задача) должен специализировать этот класс,
    переопределяя методы actions и result, а при необходимости
    action_cost, h и is_goal.
    """

    def __init__(self, initial=None, goal=None, **kwargs):
        self.initial = initial
        self.goal = goal
        for k, v in kwargs.items():
```

```
            setattr(self, k, v)

    @abstractmethod
    def actions(self, state):
        """
        Вернуть доступные действия (операторы) из данного состояния.
        """
        pass

    @abstractmethod
    def result(self, state, action):
        """
        Вернуть результат применения действия к состоянию.
        """
        pass

    def is_goal(self, state):
        """
        Проверка на достижение цели.
        """
        return False

    def action_cost(self, s, a, s1):
        """
        Стоимость перехода; для заливки несущественно, оставим 1.
        """
        return 1

    def h(self, node):
        """
        Эвристическая функция, по умолчанию 0.
        """
        return 0


class LongestConsecutivePathProblem(Problem):
    """
    Задача поиска самого длинного пути в матрице.
    """

    def __init__(self, matrix, start_char):
        """
        :param matrix: двумерный массив символов (список списков).
        :param start_char: символ, с которого начинается путь.
        """

        super().__init__(initial=start_char)
        self.matrix = matrix
        self.rows = len(matrix)
        self.cols = len(matrix[0]) if self.rows > 0 else 0
        self.start_char = start_char

    def actions(self, state):
        """
        Возвращает список соседей (nr, nc) по 8 направлениям,
        у которых символ == chr(ord(matrix[r][c]) + 1).
        """

        (r, c) = state
        curr_char = self.matrix[r][c]
        next_char = chr(ord(curr_char) + 1)

        neighbors = []
```

```
        # Направления
        deltas = [
            (-1, -1), (-1, 0), (-1, 1),
            (0, -1),           (0, 1),
            (1, -1),  (1, 0),  (1, 1),
        ]

        for dr, dc in deltas:
            nr, nc = r + dr, c + dc
            if 0 <= nr < self.rows and 0 <= nc < self.cols:
                if self.matrix[nr][nc] == next_char:
                    neighbors.append((nr, nc))
        return neighbors

    def result(self, state, action):
        """
        Просто переходим в соседнюю ячейку (action).
        """
        return action


def dfs_longest(problem, r, c):
    """
    Поиск всех соседей, у которых символ = current_char + 1 по алфавиту
    Для каждого соседа вызывается dfs_longest, и берется максимум
    Возвращается 1 + max(...) или 1, если нет соседей
    """

    children = problem.actions((r, c))
    best_len = 1  # хотя бы текущая клетка
    for (nr, nc) in children:
        length_child = 1 + dfs_longest(problem, nr, nc)
        if length_child > best_len:
            best_len = length_child
    return best_len


def find_longest_consecutive_path(problem):
    """
    Ищет максимальную длину цепочки, начинающейся с problem.start_char.
    Перебирает все ячейки, где стоит start_char,
    и берёт максимум результата поиска.
    """

    matrix = problem.matrix
    rows, cols = problem.rows, problem.cols
    start_char = problem.start_char

    longest_path = 0
    for r in range(rows):
        for c in range(cols):
            if matrix[r][c] == start_char:
                length = dfs_longest(problem, r, c)
                if length > longest_path:
                    longest_path = length
    return longest_path


def main():
    """
    Главная функция программы.
    """

    # Исходная матрица
```

```
    matrix = [
        ["K", "L", "M", "N", "O", "P", "Q"],
        ["J", "A", "B", "C", "D", "E", "R"],
        ["I", "Z", "Y", "X", "W", "F", "S"],
        ["H", "G", "T", "U", "V", "G", "T"],
        ["G", "F", "E", "D", "C", "B", "U"],
        ["F", "E", "D", "C", "B", "A", "V"],
        ["E", "D", "C", "B", "A", "Z", "W"],
    ]

    start_char = 'C'
    problem = LongestConsecutivePathProblem(matrix, start_char)
    answer = find_longest_consecutive_path(problem)
    print(f"Длина самого длинного пути, начиная с символа '{start_char}':
{answer}")


if __name__ == "__main__":
    main()
```

Результаты работы программы для собственной матрицы символов для символа «C» (рис. 4).

```
Исходная матрица:
['K', 'L', 'M', 'N', 'O', 'P', 'Q']
['J', 'A', 'B', 'C', 'D', 'E', 'R']
['I', 'Z', 'Y', 'X', 'W', 'F', 'S']
['H', 'G', 'T', 'U', 'V', 'G', 'T']
['G', 'F', 'E', 'D', 'C', 'B', 'U']
['F', 'E', 'D', 'C', 'B', 'A', 'V']
['E', 'D', 'C', 'B', 'A', 'Z', 'W']
Длина самого длинного пути, начиная с символа 'C': 21
```

Рисунок 4 – Самый длинный путь в матрице символов

Генерирование списка возможных слов из матрицы символов

Дана матрица символов размером M×N. Задача – найти и вывести список всех возможных слов, которые могут быть сформированы из последовательности соседних символов в этой матрице. При этом слово может формироваться во всех восьми возможных направлениях (север, юг, восток, запад, северо-восток, северо-запад, юго-восток, юго-запад), и каждая клетка может быть использована в слове только один раз.

Код программы составления всех возможных слов из матрицы символов:

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

"""
Дана матрица символов размером M×N. Задача - найти и вывести список
всех возможных слов, которые могут быть сформированы из последовательности
соседних символов в этой матрице. При этом слово может формироваться
во всех восьми возможных направлениях (север, юг, восток, запад,
северо-восток, северо-запад, юго-восток, юго-запад), и каждая клетка
может быть использована в слове только один раз.
"""

from abc import ABC, abstractmethod


class Problem(ABC):
    """
    Абстрактный класс для формальной постановки задачи.
    Новый домен (конкретная задача) должен специализировать этот класс,
    переопределяя методы actions и result, а при необходимости
    action_cost, h и is_goal.
    """

    def __init__(self, initial=None, goal=None, **kwargs):
        self.initial = initial
        self.goal = goal
        for k, v in kwargs.items():
            setattr(self, k, v)

    @abstractmethod
    def actions(self, state):
        """
        Вернуть доступные действия (операторы) из данного состояния.
        """
        pass

    @abstractmethod
    def result(self, state, action):
        """
        Вернуть результат применения действия к состоянию.
        """
        pass

    def is_goal(self, state):
        """
        Проверка на достижение цели.
        """
        return False

    def action_cost(self, s, a, s1):
        """
        Стоимость перехода; для заливки несущественно, оставим 1.
        """
        return 1

    def h(self, node):
        """
        Эвристическая функция, по умолчанию 0.
        """
        return 0


class WordSearchProblem(Problem):
    """
```

```
    Описывает задачу поиска всех слов из dictionary
    в матрице board (список списков символов).
    """

    def __init__(self, board, dictionary):
        super().__init__(board=board, dictionary=dictionary)
        self.rows = len(board)
        self.cols = len(board[0]) if self.rows > 0 else 0

    def actions(self, state):
        return []

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


def can_form_word_dfs(board, word):
    """
    Функция проверяющая можно ли составить слово в матрице в 8-ми
направлениях.
    Возвращает True, если слово word можно найти в матрице символов,
    двигаясь во все 8 направлений, используя каждую клетку не более одного
раза.
    """

    rows = len(board)
    cols = len(board[0]) if rows > 0 else 0

    # Смещения по 8 направлениям
    deltas = [
        (-1, -1), (-1, 0), (-1, 1),
        (0, -1),           (0, 1),
        (1, -1),  (1, 0),  (1, 1),
    ]

    def dfs(r, c, index, visited):
        """
        Поиск слова начиная с клетки (r,c).
        """
        if index == len(word):
            return True  # все буквы совпали

        if not (0 <= r < rows and 0 <= c < cols):
            return False
        if board[r][c] != word[index]:
            return False
        if (r, c) in visited:
            return False

        # Помечаем текущую клетку как использованную
        visited.add((r, c))

        # Переходим к следующей букве
        for (dr, dc) in deltas:
            nr, nc = r + dr, c + dc
            if dfs(nr, nc, index + 1, visited):
                return True

        # Если не получилось найти дальше, снимаем отметку и идём другим
путём
        visited.remove((r, c))
        return False

    # Ищем все вхождения первой буквы во всей матрице
```

```
    first_char = word[0]
    for r in range(rows):
        for c in range(cols):
            if board[r][c] == first_char:
                # Пробуем начать DFS с (r,c)
                if dfs(r, c, 0, set()):
                    return True

    return False


def find_all_words(problem):
    """
    Возвращает множество слов, которые можно сформировать
    в матрице символов из словаря слов.
    """
    result = set()
    for w in problem.dictionary:
        if can_form_word_dfs(problem.board, w):
            result.add(w)
    return result


def main():
    """
    Главаная функция программы.
    """

    board = [
        ['М', 'И', 'Р', 'У', 'П'],
        ['А', 'П', 'А', 'П', 'А'],
        ['О', 'Р', 'А', 'Г', 'Д'],
        ['Л', 'Е', 'Т', 'О', 'М']
    ]
    print("Исходная матрица символов: ")
    for row in board:
        print(row)

    dictionary = [
        "МИР",
        "ЛЕТО",
        "УРАЛ",
        "ПАРОГ",
        "МАРТ",
        "ПИР",
    ]
    print("Доступный словарь слов: ", dictionary)

    problem = WordSearchProblem(board, dictionary)

    found_words = find_all_words(problem)

    print("Найденные слова:", found_words)


if __name__ == "__main__":
    main()
```

Результаты работы программы, поиск слов в матрице символов по заданном словарю слов (рис. 5)

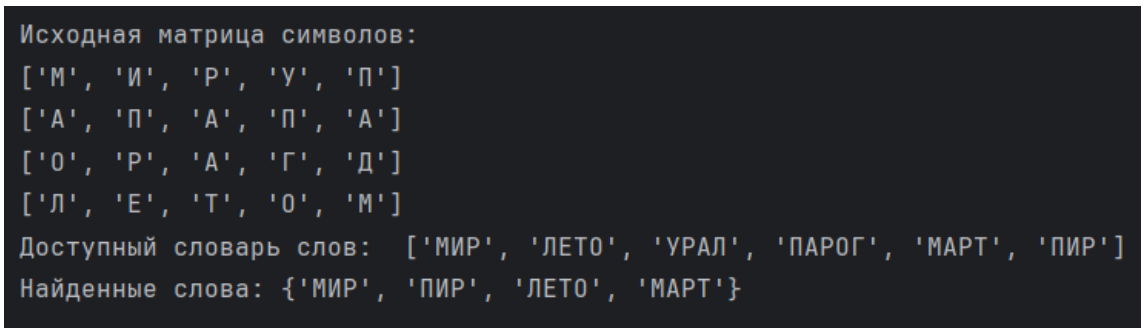

Рисунок 5 – Найденные слова в матрице символов

Ссылка на репозиторий данной лабораторной работы:
https://github.com/AndreyPust/Artificial_Intelligence_laboratory_work_3.git

Ответы на контрольные вопросы:

1. В чем ключевое отличие поиска в глубину от поиска в ширину?

Ключевое отличие состоит в стратегии обхода. Поиск в глубину (DFS) исследует одну ветвь дерева/графа до самого глубокого уровня, прежде чем перейти к следующей ветви. Поиск в ширину (BFS) исследует все узлы на одном уровне глубины перед переходом к следующему уровню.

2. Какие четыре критерия качества поиска обсуждаются в тексте для оценки алгоритмов?

– полнота – гарантирует ли алгоритм нахождение решения, если оно существует;

– оптимальность – находит ли алгоритм лучшее (минимальное по стоимости) решение;

– временная сложность – сколько времени требуется для нахождения решения;

– пространственная сложность – сколько памяти использует алгоритм.

3. Что происходит при расширении узла в поиске в глубину?

При расширении узла генерируются его дочерние узлы. Это включает создание новых узлов на основе соседей текущего узла, которые затем добавляются в стек или передаются в рекурсию.

4. Почему поиск в глубину использует очередь типа "последним пришел – первым ушел" (LIFO)?

Очередь LIFO (стек) обеспечивает приоритетное исследование недавно добавленных узлов, что позволяет алгоритму «углубляться» в ветви графа или дерева.

5. Как поиск в глубину справляется с удалением узлов из памяти, и почему это преимущество перед поиском в ширину?

Поиск в глубину удаляет узлы из памяти, как только они обработаны и нет необходимости возвращаться к ним. Это снижает объем памяти по сравнению с поиском в ширину, который должен хранить все узлы на текущем уровне.

6. Какие узлы остаются в памяти после того, как достигнута максимальная глубина дерева?

В памяти остаются узлы текущего пути от корня до текущего узла и узлы, которые еще не были исследованы.

7. В каких случаях поиск в глубину может "застрять" и не найти решение?

Если граф или дерево бесконечно глубокие. Если существует цикл, и алгоритм не имеет проверки на циклы.

8. Как временная сложность поиска в глубину зависит от максимальной глубины дерева?

Временная сложность DFS составляет O(bm), где b – фактор ветвления, а m – максимальная глубина дерева. Она растет экспоненциально с глубиной дерева.

9. Почему поиск в глубину не гарантирует нахождение оптимального решения?

DFS не рассматривает все пути одновременно, поэтому может найти не самый короткий путь, если решение обнаружено до исследования более выгодного варианта.

10. В каких ситуациях предпочтительно использовать поиск в глубину, несмотря на его недостатки?

Когда пространство поиска ограничено и важно минимизировать потребление памяти. Если известна приблизительная глубина решения. Когда нужно найти любое решение быстро, а не обязательно оптимальное.

11. Что делает функция depth_first_recursive_search, и какие параметры она принимает?

Функция выполняет рекурсивный поиск в глубину для нахождения решения. Она принимает:

– problem – задачу, содержащую начальный узел и цель;
– graph – граф для обхода;
– node – текущий узел.

12. Какую задачу решает проверка if node is None?

Она задает начальный узел, если он не был передан в качестве параметра.

13. В каком случае функция возвращает узел как решение задачи?

Когда состояние узла совпадает с целевым состоянием задачи (problem.is_goal(node.state)).

14. Почему важна проверка на циклы в алгоритме рекурсивного поиска в глубину?

Проверка на циклы предотвращает бесконечный возврат к ранее посещенным узлам, особенно в графах с циклическими структурами.

15. Что возвращает функция при обнаружении цикла?

Она возвращает None (или failure), указывая, что цикл был обнаружен и продолжение поиска по этому пути невозможно.

16. Как функция обрабатывает дочерние узлы текущего узла?

Функция генерирует дочерние узлы через «expand» и рекурсивно вызывает саму себя для каждого из них.

17. Какой механизм используется для обхода дерева поиска в этой реализации?

Используется рекурсия для перехода между узлами, а стек вызовов автоматически сохраняет текущий путь.

18. Что произойдет, если не будет найдено решение в ходе рекурсии?

Функция вернет failure, что указывает на отсутствие пути к цели.

19. Почему функция рекурсивно вызывает саму себя внутри цикла?

Это позволяет исследовать все ветви графа/дерева, начиная с текущего узла.

20. Как функция expand(problem, node) взаимодействует с текущим узлом?

Она генерирует список дочерних узлов текущего узла на основе графа и его соседей.

21. Какова роль функции is_cycle(node) в этом алгоритме?

Она проверяет, встречался ли текущий узел ранее в пути, предотвращая зацикливание.

22. Почему проверка if result в рекурсивном вызове важна для корректной работы алгоритма?

Эта проверка определяет, было ли найдено решение по данному пути, и завершает дальнейший поиск, если оно найдено.

23. В каких ситуациях алгоритм может вернуть failure?

Если узел не может быть расширен (нет дочерних узлов). Если все пути исследованы, но цель не достигнута.

24. Как рекурсивная реализация отличается от итеративного поиска в глубину?

В рекурсивной реализации используется стек вызовов, управляемый автоматически, в то время как в итеративной используется явный стек для хранения состояния узлов.

25. Какие потенциальные проблемы могут возникнуть при использовании этого алгоритма для поиска в бесконечных деревьях?

Бесконечная рекурсия при отсутствии проверки на глубину или циклы. Переполнение стека вызовов, что приведет к ошибке сегментации (stack overflow).

Вывод: в ходе выполнения лабораторной работы были приобретены навыки по работе с поиском в глубину с помощью языка программирования Python версии 3.x.