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 №2

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 ширину.

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

Ход работы:

Проработка примеров лабораторной работы:

Был проработан приведенный в лабораторной работе код алгоритма поиска в ширину. Код данного алгоритма:

```
#!/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


def breadth_first_search(problem):
    """
    Функция алгоритма поиска в ширину.

    :param problem: объект формальной задачи.
    :return: решение задачи (узел) или неудачу.
    """

    node = Node(problem.initial)
    if problem.is_goal(problem.initial):
        return node
    frontier = FIFOQueue([node])
```

```
    reached = {problem.initial}
    while frontier:
        node = frontier.pop()
        for child in expand(problem, node):
            s = child.state
            if problem.is_goal(s):
                return child
            if s not in reached:
                reached.add(s)
                frontier.appendleft(child)
    return failure
```

Выполнение индивидуальных заданий:

Поиск в ширину

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

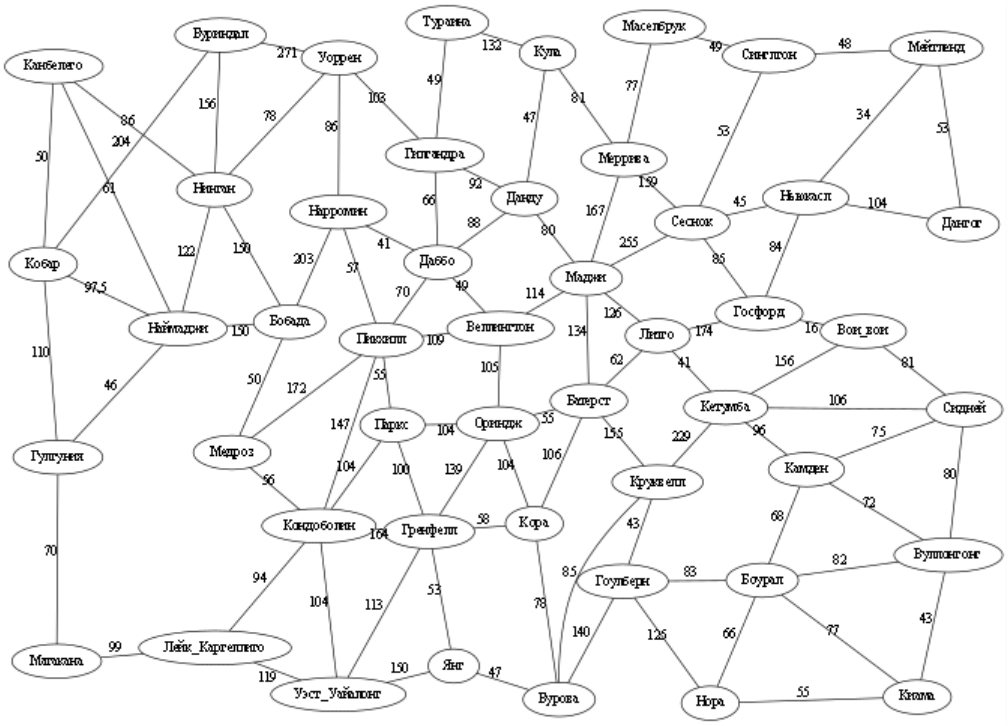

Рисунок 1 – Граф городов Австралии на языке DOT

Граф в программе был описан в виде списка ребер графа, который в дальнейшем преобразуется в список смежности графа. В алгоритме поиска в

ширину также учитываются веса ребер графа. Код программы нахождения кратчайшего пути в графе:

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


import heapq
from collections import defaultdict


class GraphProblem:
    """
    Задача поиска кратчайшего пути в графе с учётом весов рёбер.
    """
    def __init__(self, graph_structure, start_node, goal_node):
        self.graph_structure = graph_structure  # Словарь смежности, включая
веса рёбер
        self.start_node = start_node  # Начальный узел
        self.goal_node = goal_node  # Целевой узел

    def actions(self, current_node):
        return self.graph_structure[current_node]  # Возвращает список
смежных узлов с весами

    def is_goal(self, current_node):
        return current_node == self.goal_node


class Node:
    """
    Узел в дереве поиска.
    """
    def __init__(self, state_name, parent_node=None, total_cost=0):
        self.state_name = state_name
        self.parent_node = parent_node
        self.total_cost = total_cost

    def path(self):
        """
        Восстановление пути от начального состояния до текущего узла.
        """
        current_node, path_nodes = self, []
        while current_node:
            path_nodes.append(current_node.state_name)
            current_node = current_node.parent_node
        return list(reversed(path_nodes))


def bfs_search(problem_instance):
    """
    Поиск кратчайшего пути, реализованный как поиск в ширину с учетом весов
ребер графа.
    """
    # Очередь с приоритетом, где каждый элемент - кортеж (стоимость,
идентификатор, узел)
    priority_queue = []
    counter = 0  # Уникальный идентификатор для узлов
    heapq.heappush(priority_queue, (0, counter,
Node(problem_instance.start_node)))

    # Словарь для отслеживания минимальной стоимости до каждого узла
    visited_costs = {}
```

```
    while priority_queue:
        # Извлекаем узел с наименьшей стоимостью
        current_cost, _, current_node = heapq.heappop(priority_queue)

        # Если этот узел уже был обработан с меньшей или равной стоимостью -
пропускаем
        if current_node.state_name in visited_costs and
visited_costs[current_node.state_name] <= current_cost:
            continue

        # Обновляем минимальную стоимость до текущего узла
        visited_costs[current_node.state_name] = current_cost

        # Проверяем, достигли ли цели
        if problem_instance.is_goal(current_node.state_name):
            return current_node.path(), current_cost

        # Обрабатываем соседей
        for neighbor_name, edge_cost in
problem_instance.actions(current_node.state_name):
            new_path_cost = current_cost + edge_cost
            # Добавляем в очередь, если новый путь короче
            if neighbor_name not in visited_costs or
visited_costs[neighbor_name] > new_path_cost:
                counter += 1  # Увеличиваем идентификатор
                heapq.heappush(priority_queue,
                               (new_path_cost, counter, Node(neighbor_name,
current_node, new_path_cost)))

    return None, float('inf')  # Если путь не найден


if __name__ == "__main__":
    # Список рёбер графа
    graph_edges = [
        # Город Буриндал
        ("Буриндал", "Уоррен", 271),
        ("Буриндал", "Нинган", 156),
        ("Буриндал", "Кобар", 204),

        # Город Кобар
        ("Кобар", "Канбелего", 50),
        ("Кобар", "Наймаджи", 97.5),
        ("Кобар", "Гулгуния", 110),

        # Город Гулгуния
        ("Гулгуния", "Наймаджи", 46),
        ("Гулгуния", "Матакана", 70),

        # Город Наймаджи
        ("Наймаджи", "Канбелего", 61),
        ("Наймаджи", "Нинган", 122),
        ("Наймаджи", "Бобада", 150),

        # Город Нинган
        ("Нинган", "Канбелего", 86),
        ("Нинган", "Уоррен", 78),
        ("Нинган", "Бобада", 150),

        # Город Уоррен
        ("Уоррен", "Гилгандра", 103),
        ("Уоррен", "Нарромин", 86),
```

```
        # Город Нарромин
        ("Нарромин", "Бобада", 203),
        ("Нарромин", "Пикхилл", 57),
        ("Нарромин", "Даббо", 41),

        # Город Пикхилл
        ("Пикхилл", "Даббо", 70),
        ("Пикхилл", "Веллингтон", 109),
        ("Пикхилл", "Медроз", 172),
        ("Пикхилл", "Кондоболин", 147),
        ("Пикхилл", "Паркс", 55),

        # Город Даббо
        ("Даббо", "Гилгандра", 66),
        ("Даббо", "Данду", 88),
        ("Даббо", "Веллингтон", 49),

        # Город Бобада
        ("Бобада", "Медроз", 50),

        # Город Медроз
        ("Медроз", "Кондоболин", 56),

        # Город Кондоболин
        ("Кондоболин", "Паркс", 104),
        ("Кондоболин", "Гренфелл", 164),
        ("Кондоболин", "Уэст_Уайалонг", 104),
        ("Кондоболин", "Лейк_Каргеллиго", 94),

        # Город Матакана
        ("Матакана", "Лейк_Каргеллиго", 99),

        # Город Лейк_Каргеллиго
        ("Лейк_Каргеллиго", "Уэст_Уайалонг", 119),

        # Город Уэст_Уайалонг
        ("Уэст_Уайалонг", "Гренфелл", 113),
        ("Уэст_Уайалонг", "Янг", 150),

        # Город Гренфелл
        ("Гренфелл", "Паркс", 100),
        ("Гренфелл", "Ориндж", 139),
        ("Гренфелл", "Кора", 58),
        ("Гренфелл", "Янг", 53),

        # Город Ориндж
        ("Ориндж", "Кора", 104),
        ("Ориндж", "Паркс", 104),
        ("Ориндж", "Веллингтон", 105),
        ("Ориндж", "Батерст", 55),

        # Город Веллингтон
        ("Веллингтон", "Маджи", 114),

        # Город Маджи
        ("Маджи", "Данду", 80),
        ("Маджи", "Меррива", 167),
        ("Маджи", "Сеснок", 255),
        ("Маджи", "Батерст", 134),
        ("Маджи", "Литго", 126),

        # Город Батерст
        ("Батерст", "Литго", 62),
        ("Батерст", "Кора", 106),
```

```
        ("Батерст", "Круквелл", 155),

        # Город Бурова
        ("Бурова", "Янг", 47),
        ("Бурова", "Кора", 78),
        ("Бурова", "Круквелл", 85),
        ("Бурова", "Гоулберн", 140),

        # Город Круквелл
        ("Круквелл", "Кетумба", 229),
        ("Круквелл", "Гоулберн", 43),

        # Город Гоулберн
        ("Гоулберн", "Нора", 125),
        ("Гоулберн", "Боурал", 83),

        # Город Боурал
        ("Боурал", "Нора", 66),
        ("Боурал", "Киама", 77),
        ("Боурал", "Вуллонгонг", 82),
        ("Боурал", "Камден", 68),

        # Город Киама
        ("Киама", "Нора", 55),
        ("Киама", "Вуллонгонг", 43),

        # Город Вуллонгонг
        ("Вуллонгонг", "Камден", 72),
        ("Вуллонгонг", "Сидней", 80),

        # Город Камден
        ("Камден", "Кетумба", 96),
        ("Камден", "Сидней", 75),

        # Город Сидней
        ("Сидней", "Кетумба", 106),
        ("Сидней", "Вои_вои", 81),

        # Город Вои_вои
        ("Вои_вои", "Кетумба", 156),
        ("Вои_вои", "Госфорд", 16),

        # Город Госфорд
        ("Госфорд", "Ньюкасл", 84),
        ("Госфорд", "Сеснок", 85),
        ("Госфорд", "Литго", 174),

        # Город Сеснок
        ("Сеснок", "Меррива", 159),
        ("Сеснок", "Синглтон", 53),
        ("Сеснок", "Ньюкасл", 45),

        # Город Ньюкасл
        ("Ньюкасл", "Мейтленд", 34),
        ("Ньюкасл", "Дангог", 104),

        # Город Мейтленд
        ("Мейтленд", "Дангог", 53),
        ("Мейтленд", "Синглтон", 48),

        # Город Синглтон
        ("Синглтон", "Маселбрук", 49),

        # Город Меррива
```

```
        ("Меррива", "Маселбрук", 77),
        ("Меррива", "Кула", 81),

        # Город Кула
        ("Кула", "Данду", 47),
        ("Кула", "Тураина", 132),

        # Город Гилгандра
        ("Гилгандра", "Тураина", 49),
        ("Гилгандра", "Данду", 92),

        # Город Кетумба
        ("Кетумба", "Литго", 41)
    ]

    # Преобразование в словарь смежности для неориентированного графа
    adjacency_list = defaultdict(list)
    for start, end, edge_costs in graph_edges:
        adjacency_list[start].append((end, edge_costs))
        adjacency_list[end].append((start, edge_costs))

    # Определение задачи
    search_task = GraphProblem(adjacency_list, "Буриндал", "Сидней")  # граф,
начальное состояние, целевое состояние

    # Поиск кратчайшего пути
    shortest_path, total_cost_way = bfs_search(search_task)

    # Вывод результата
    if shortest_path:
        print("Кратчайший путь:", " -> ".join(shortest_path))
        print("Общая стоимость:", total_cost_way)
    else:
        print("Путь не найден.")
```

В результате выполнения данного кода находится кратчайший путь из города Буриндал в город Сидней (рис. 2).

```
C:\Users\Andrey\anaconda3\envs\AI_lab_1\python.exe C:\Users\Andrey\Desktop\ИИ\Лабораторная_работа_2\Artificial_Intelligence_laborato
Кратчайший путь: Буриндал -> Нинган -> Уоррен -> Нарромин -> Даббо -> Веллингтон -> Ориндж -> Батерст -> Литго -> Кетумба -> Сидней
Общая стоимость: 779

Process finished with exit code 0
```

Рисунок 2 – Результаты работы программы нахождения кратчайшего пути

Найденный путь полностью совпадает с найденным вручную в лабораторной работе 1 (рис. 3).

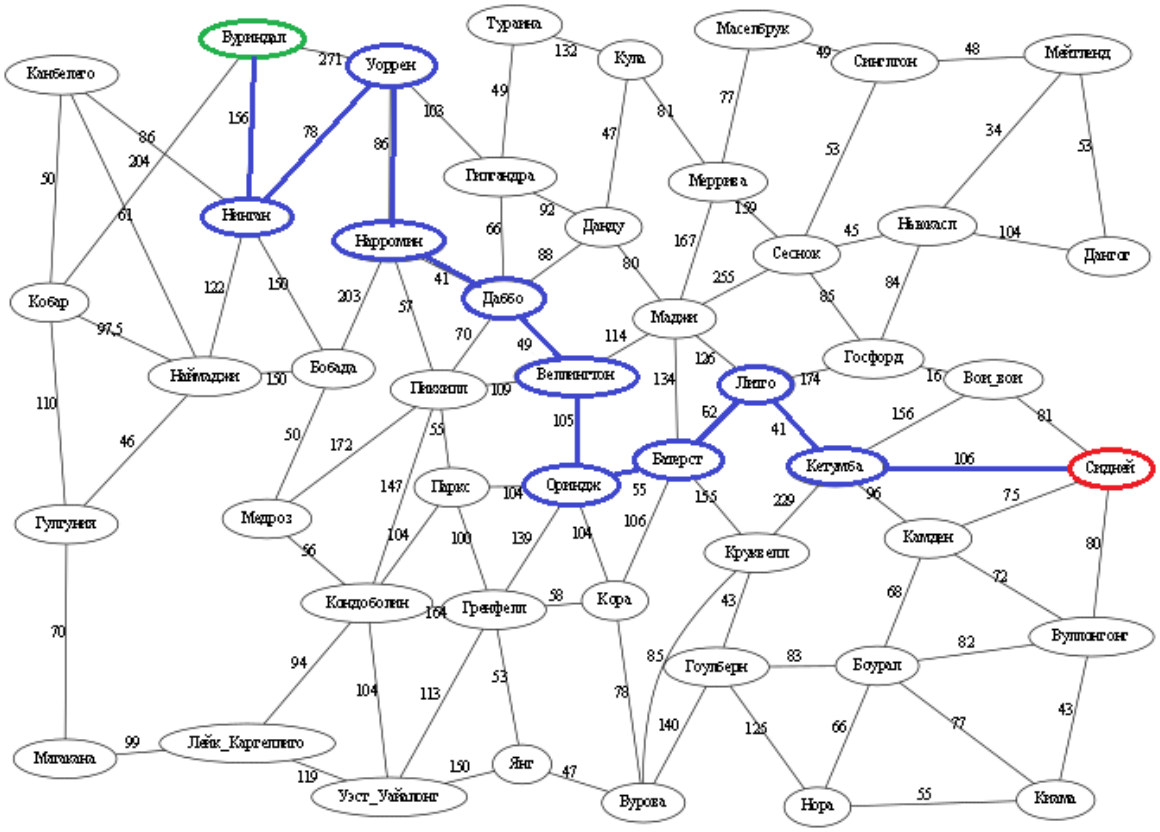

Рисунок 3 – Кратчайший путь на графе

Расширенный подсчет количества островов в бинарной матрице

Дана бинарная матрица, где 0 представляет собой воду, а 1 представляет собой землю. Связанные единицы по 4 стороны или по диагонали формируют остров. Необходимо подсчитать общее количество островов в данной матрице.

Код решения данной задачи с применением алгоритма поиска в ширину:

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


# Дана бинарная матрица, где 0 представляет собой воду, а 1 представляет
собой землю.
# Связанные единицы по 4 стороны или по диагонали формируют остров.
# Необходимо подсчитать общее количество островов в данной матрице.


from collections import deque


def allow_move(matrix, x, y, processed):
    """
    Функция для проверки перехода в позицию следующей клетки с текущей
клетки.
    Функция возвращает false, если заданы недействительные матричные
координаты (выход за пределы)
    или клетка представляет воду или позиция клетки уже обработана.

    :param matrix: бинарная матрица;
    :param x: координата x потерциального перехода;
```

```
    :param y: координата y потенциального перехода;
    :param processed: обработанный узел.
    :return: разрешение на переход.
    """

    rows = len(processed)
    cols = len(processed[0])
    return (
        0 <= x < rows and
        0 <= y < cols and
        matrix[x][y] == 1 and
        not processed[x][y]
    )


def search_in_width(matrix, processed, i, j, row, col):
    """
    Реализация алгоритма поиска в ширину.

    :param matrix: бинарная матрица;
    :param processed: позиция;
    :param i: координаты x;
    :param j: координаты y;
    :param row: перемещения по x;
    :param col: перемещения по y.
    """

    # создает пустую queue и ставит в queue исходный узел
    q = deque()
    q.append((i, j))

    # пометить исходный узел как обработанный
    processed[i][j] = True

    # Цикл # до тех пор, пока queue не станет пустой
    while q:
        # удаляет передний узел из очереди и обрабатывает его
        x, y = q.popleft()

        # проверяет все восемь возможных перемещений из текущей ячейки
        # и ставить в queue каждое допустимое движение
        for k in range(len(row)):
            # пропустить, если локация недействительна, уже обработана или
содержит воду
            if allow_move(matrix, x + row[k], y + col[k], processed):
                # пропустить, если местоположение неверно или уже
                # обработан или состоит из воды
                processed[x + row[k]][y + col[k]] = True
                q.append((x + row[k], y + col[k]))


def counting_islands(matrix, row, col):
    """
    Функция подсчета островов.

    :param matrix: бинарная матрица;
    :param row: перемещение по x;
    :param col: перемещение по y.
    :return: количество островов.
    """

    # Если матрица пустая
    if not matrix or not len(matrix):
        return 0
```

```
    # Расчет размерности матрицы M × N
    (M, N) = (len(matrix), len(matrix[0]))

    # запоминаем, обработана ячейка или нет
    processed = [[False for x in range(N)] for y in range(M)]

    island = 0
    for i in range(M):
        for j in range(N):
            # запускает BFS с каждого необработанного узла и увеличивает
количество островов
            if matrix[i][j] == 1 and not processed[i][j]:
                search_in_width(matrix, processed, i, j, row, col)
                island = island + 1

    return island


if __name__ == '__main__':
    # Укажем, что перемещаться можно как по сторанам света, так и по смежным
сторонам света
    # север, юг, запад, восток, северо-запад, северо-восток, юго-запад, юго-
восток
    o_x = [-1, -1, -1, 0, 1, 0, 1, 1]
    o_y = [-1, 1, 0, -1, -1, 1, 0, 1]

    grid = [
        [1, 1, 0, 0, 0],
        [0, 1, 0, 0, 1],
        [1, 0, 0, 1, 1],
        [0, 0, 0, 0, 0],
        [1, 0, 1, 0, 1]
    ]

    print("Общее количество островов:", counting_islands(grid, o_x, o_y))
```

Результаты работы программы для бинарной матрицы из примера лабораторной работы (рис. 4). Результаты работы программы для исходных данных совпали с результатами примера лабораторной работы.

```
C:\Users\Andrey\anaconda3\envs\AI_lab_1\python.exe C:\U
Общее количество островов: 5

Process finished with exit code 0
```

Рисунок 4 – Соответствующий вывод

Для расширенного подсчета количества островов в бинарной матрице была подготовлена собственная матрица со своими островами (рис. 5). Количество островов для данной матрицы равно 5, размерность матрицы 7 на 7.

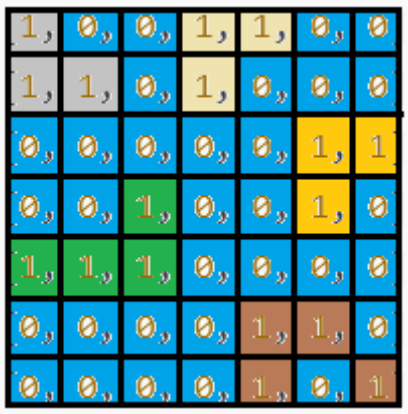

Рисунок 5 – Собственная бинарная матрица

Данная матрица была представлена в виде списка:

```
grid = [
    [1, 0, 0, 1, 1, 0, 0],
    [1, 1, 0, 1, 0, 0, 0],
    [0, 0, 0, 0, 0, 1, 1],
    [0, 0, 1, 0, 0, 1, 0],
    [1, 1, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 1, 0],
    [0, 0, 0, 0, 1, 0, 1]
]
```

Результаты работы программы для данной бинарной матрицы (рис. 6).

```
C:\Users\Andrey\anaconda3\envs\AI_lab_1\pytho
Общее количество островов: 5

Process finished with exit code 0
```

Рисунок 6 – Количество островов в матрице

Поиск кратчайшего пути в лабиринте

Необходимо решить задачу поиска кратчайшего пути через лабиринт, используя алгоритм поиска в ширину. Лабиринт представлен в виде бинарной матрицы, где 1 – это проход, а 0 – это стена.

Код решения данной задачи с применением алгоритма поиска в ширину:

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


# Необходимо решить задачу поиска кратчайшего пути через лабиринт, используя
алгоритм поиска в ширину.
# Лабиринт представлен в виде бинарной матрицы, где 1 - это проход, а 0 - это
стена.
```

```python
from collections import deque


def search_in_width(maze, start, end):
    """
    Реализация поиска в ширину для нахождения кратчайшего пути в лабиринте.

    :param maze: бинарная матрица лабиринта;
    :param start: начальное положение;
    :param end: выход из лабиринта.
    :return: список клеток лабиринта кратчайшего пути и длина этого пути.
    """

    # Правила перемещения по лабиринту: вверх, вниз, влево и вправо
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    queue = deque([start])
    visited = {start: None}  # Словарь для отслеживания предков узлов

    while queue:
        current = queue.popleft()

        # Если достигли цели, строим путь
        if current == end:
            way = []
            while current is not None:
                way.append(current)
                current = visited[current]
            return way[::-1]  # возврат пути в обратном порядке

        # Проверка соседних клеток
        for d in directions:
            neighbor = (current[0] + d[0], current[1] + d[1])
            if (0 <= neighbor[0] < len(maze) and
                    0 <= neighbor[1] < len(maze[0]) and
                    maze[neighbor[0]][neighbor[1]] == 1 and
                    neighbor not in visited):
                visited[neighbor] = current
                queue.append(neighbor)

    return None  # Если путь не найден


if __name__ == '__main__':
    labyrinth = [
        [1, 1, 1, 1, 1, 0, 0, 1, 1, 1],
        [0, 1, 1, 1, 1, 1, 0, 1, 0, 1],
        [0, 0, 1, 0, 1, 1, 1, 0, 0, 1],
        [1, 0, 1, 1, 1, 0, 1, 1, 0, 1],
        [0, 0, 0, 1, 0, 0, 0, 1, 0, 1],
        [1, 0, 1, 1, 1, 0, 0, 1, 1, 0],
        [0, 0, 0, 0, 1, 0, 0, 1, 0, 1],
        [0, 1, 1, 1, 1, 1, 1, 1, 0, 0],
        [1, 1, 1, 1, 1, 0, 0, 1, 1, 1],
        [0, 0, 1, 0, 0, 1, 1, 0, 0, 1],
    ]

    initial = (0, 0)
    goal = (7, 5)

    # Поиск кратчайшего пути в лабиринте
    path = search_in_width(labyrinth, initial, goal)

    if path:
```

```
        print("Длина пути:", len(path) - 1)
    else:
        print("Путь не найден!")
```

Для проверки работоспособности программы был добавлен пример лабиринта из примера лабораторной работы:

```
labyrinth = [
    [1, 1, 1, 1, 1, 0, 0, 1, 1, 1],
    [0, 1, 1, 1, 1, 1, 0, 1, 0, 1],
    [0, 0, 1, 0, 1, 1, 1, 0, 0, 1],
    [1, 0, 1, 1, 1, 0, 1, 1, 0, 1],
    [0, 0, 0, 1, 0, 0, 0, 1, 0, 1],
    [1, 0, 1, 1, 1, 0, 0, 1, 1, 0],
    [0, 0, 0, 0, 1, 0, 0, 1, 0, 1],
    [0, 1, 1, 1, 1, 1, 1, 1, 0, 0],
    [1, 1, 1, 1, 1, 0, 0, 1, 1, 1],
    [0, 0, 1, 0, 0, 1, 1, 0, 0, 1],
]

initial = (0, 0)
goal = (7, 5)
```

Результаты работы данного кода совпали с результатами ожидаемого вывода (рис. 7).

```
C:\Users\Andrey\anaconda3\envs\AI_lab_1\pyt
Длина пути: 12

Process finished with exit code 0
```

Рисунок 7 – Результаты работы программы в известном лабиринте

Для поиска кратчайшего пути в лабиринте была подготовлена собственная схема лабиринта (рис. 8). Начальная позиция в лабиринте – (0; 0), а выход из лабиринта – (11; 11).

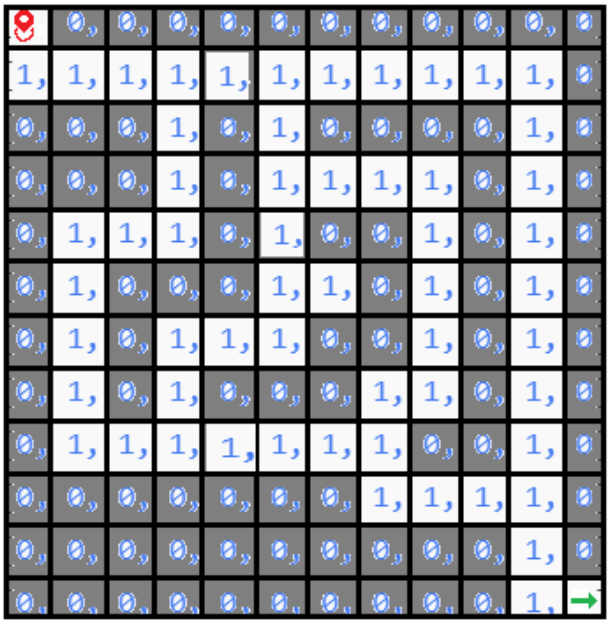

Рисунок 8 – Схема лабиринта

Данный лабиринт был представлен в виде бинарной матрицы:

```
labyrinth = [
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
    [0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0],
    [0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0],
    [0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0],
    [0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0],
    [0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0],
    [0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0],
    [0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0],
    [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1]
]
```

Результаты работы программы, вывод длины минимального пути в лабиринте (рис. 9).

```
C:\Users\Andrey\anaconda3\envs\AI_l
Длина пути: 22

Process finished with exit code 0
```

Рисунок 9 – Кратчайший путь в лабиринте

Задача о льющихся кувшинах

Необходимо реализовать алгоритм поиска в ширину для решения задачи о льющихся кувшинах, где цель состоит в том, чтобы получить заданный объем воды в одном из кувшинов.

Существует набор кувшинов, каждый из которых имеет размер (емкость) в литрах, текущий уровень воды. Задача состоит в том, чтобы отмерить определенный объем воды. Он может оказаться в любом из кувшинов. Состояние представлено кортежем текущих уровней воды, а доступными действиями являются:

– (Fill, i): наполнить i-й кувшин до самого верха (из крана с неограниченным количеством воды);

– (Dump, i): вылить всю воду из i-го кувшина;

– (Pour, i, j): перелить воду из i-го кувшина в j-й кувшин, пока либо кувшин i не опустеет, либо кувшин j не наполнится, в зависимости от того, что наступит раньше.

Код программы решения данной задачи о льющихся кувшинах:

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


# Необходимо реализовать алгоритм поиска в ширину для решения задачи о
льющихся кувшинах,
# где цель состоит в том, чтобы получить заданный объем воды в одном из
кувшинов.
# Существует набор кувшинов, каждый из которых имеет размер (емкость) в
литрах, текущий уровень воды.
# Задача состоит в том, чтобы отмерить определенный объем воды. Он может
оказаться в любом из кувшинов.
# Состояние представлено кортежем текущих уровней воды, а доступными
действиями являются:
# - (Fill, i): наполнить i-й кувшин до самого верха (из крана с
неограниченным количеством воды);
# - (Dump, i): вылить всю воду из i-го кувшина;
# - (Pour, i, j): перелить воду из i-го кувшина в j-й кувшин, пока либо
кувшин i не опустеет,
# либо кувшин j не наполнится, в зависимости от того, что наступит раньше.


from collections import deque


def search_in_width(initial_p, goal_p, sizes_p):
    """
    Реализация алгоритма поиска в ширину для задачи о льющихся кувшинах.

    :param initial_p: начальное состояние кувшинов;
    :param goal_p: целевой объем воды;
    :param sizes_p: ёмкости кувшинов.
```

```
    :return: последовательность действий и объемы кувшинов.
    """

    # Хранение состояния и пути к нему
    queue = deque([(initial_p, [])])  # (состояние, действия)
    visited = set()
    visited.add(initial_p)

    while queue:
        state, path_p = queue.popleft()

        # Если найдено целевое состояние
        if goal_p in state:
            return path_p, state

        # Генерация всех возможных действий
        for i in range(len(sizes_p)):
            # Наполнить i-й кувшин
            new_state = list(state)
            new_state[i] = sizes_p[i]
            if tuple(new_state) not in visited:
                visited.add(tuple(new_state))
                queue.append((tuple(new_state), path_p + [('Fill', i)]))

            # Опустошить i-й кувшин
            new_state = list(state)
            new_state[i] = 0
            if tuple(new_state) not in visited:
                visited.add(tuple(new_state))
                queue.append((tuple(new_state), path_p + [('Dump', i)]))

            # Перелить из i-го в j-й
            for j in range(len(sizes_p)):
                if i != j:
                    new_state = list(state)
                    transfer = min(state[i], sizes_p[j] - state[j])
                    new_state[i] -= transfer
                    new_state[j] += transfer
                    if tuple(new_state) not in visited:
                        visited.add(tuple(new_state))
                        queue.append((tuple(new_state), path_p + [('Pour', i,
j)]))

    return None, None  # Если решения нет


if __name__ == '__main__':
    # начальное состояние (количество воды в каждом кувшине)
    initial = (1, 1, 1)
    # goal: целевое количество воды
    goal = 13
    # размеры кувшинов
    sizes = (2, 16, 32)

    # Решение задачи
    path, final_state = search_in_width(initial, goal, sizes)

    # Вывод результатов
    if path:
        print("Решение найдено!")
        print("Действия:", path)
        print("Состояния кувшинов:", final_state)
    else:
        print("Решение невозможно.")
```

Результаты работы программы с предложенными в примерах условиями (рис. 10):

```
# начальное состояние (количество воды в каждом кувшине)
initial = (1, 1, 1)
# goal: целевое количество воды
goal = 13
# размеры кувшинов
sizes = (2, 16, 32)
```

```
C:\Users\Andrey\anaconda3\envs\AI_lab_1\python.exe C:\Users\Andrey\De
Решение найдено!
Действия: [('Fill', 1), ('Pour', 1, 0), ('Dump', 0), ('Pour', 1, 0)]
Состояния кувшинов: (2, 13, 1)

Process finished with exit code 0
```

Рисунок 10 – Результаты работы программы по примеру

Решение задачи полностью совпало с предложенным в лабораторной работе. Пусть теперь кувшинов будет 5, они будут разных объемов и необходимо будет отмерить 7 литров:

```
initial = (0, 0, 0, 0, 0)  # начальное состояние
goal = 7  # целевой объем
sizes = (5, 7, 10, 15, 20)  # размеры кувшинов
```

Результаты работы программы при данных условиях задачи (рис. 11).

```
C:\Users\Andrey\anaconda3\envs\AI_lab_1
Решение найдено!
Действия: [('Fill', 1)]
Состояния кувшинов: (0, 7, 0, 0, 0)

Process finished with exit code 0
```

Рисунок 11 – Отмеренные 7 литров в кувшинах

Ссылка на репозиторий данной лабораторной работы: https://github.com/AndreyPust/Artificial_Intelligence_laboratory_work_2.git

Ответы на контрольные вопросы:

1. Какой тип очереди используется в стратегии поиска в ширину?

В поиске в ширину используется очередь FIFO (First In, First Out), где узлы извлекаются в том порядке, в котором были добавлены.

2. Почему новые узлы в стратегии поиска в ширину добавляются в конец очереди?

Это позволяет гарантировать, что узлы будут расширяться в порядке их глубины, т.е., сначала обрабатываются более близкие к корню узлы, затем более удаленные. Это является основной стратегией поиска в ширину.

3. Что происходит с узлами, которые дольше всего находятся в очереди в стратегии поиска в ширину?

Узлы, которые дольше находятся в очереди, будут извлекаться и расширяться первыми, так как очередь FIFO гарантирует, что первым выходит узел, который был добавлен раньше всех.

4. Какой узел будет расширен следующим после корневого узла, если используются правила поиска в ширину?

Следующими будут расширены узлы, которые непосредственно связаны с корневым узлом, то есть узлы на глубине 1.

5. Почему важно расширять узлы с наименьшей глубиной в поиске в ширину?

Это гарантирует, что первое найденное решение является оптимальным (самым коротким путём) в терминах количества шагов от корня до цели.

6. Как временная сложность алгоритма поиска в ширину зависит от коэффициента разветвления и глубины?

Временная сложность поиска в ширину зависит от двух факторов: коэффициента разветвления (то есть количества потомков у каждого узла) и глубины целевого узла (то есть минимального числа шагов до цели). Поиск в ширину проходит все узлы уровня за уровнем, начиная с корня, поэтому на каждом новом уровне количество узлов для обработки резко возрастает. Чем больше потомков у каждого узла и чем глубже находится целевое состояние,

тем больше узлов нужно обработать. Это приводит к экспоненциальному росту времени выполнения при увеличении этих двух параметров.

7. Каков основной фактор, определяющий пространственную сложность алгоритма поиска в ширину?

Основной фактор, влияющий на объем памяти, который требует поиск в ширину, — это количество узлов, которые нужно сохранить на самом нижнем уровне поиска. Так как алгоритм должен хранить в памяти все узлы на каждом уровне, пока они не будут обработаны, наибольшее количество узлов накапливается на последнем уровне. Чем больше у узлов потомков и чем

глубже находится целевое состояние, тем больше узлов нужно хранить одновременно, и это сильно увеличивает потребность в памяти.

8. В каких случаях поиск в ширину считается полным?

Поиск в ширину считается полным, если пространство состояний конечно или если решение существует на конечной глубине, т.е. если есть гарантии достижения цели.

9. Объясните, почему поиск в ширину может быть неэффективен с точки зрения памяти.

Поскольку поиск в ширину хранит в памяти все узлы на каждом уровне, он требует много памяти, особенно при высоком коэффициенте разветвления и большой глубине .

10. В чем заключается оптимальность поиска в ширину?

Поиск в ширину является оптимальным по количеству шагов, если все шаги имеют одинаковую длину, так как он первым находит кратчайший путь от начального состояния к целевому.

11. Какую задачу решает функция breadth_first_search?

Breadth_first_search решает задачу поиска пути от начального состояния к целевому состоянию, используя алгоритм поиска в ширину.

12. Что представляет собой объект problem, который передается в функцию?

Problem представляет собой объект задачи, который содержит начальное состояние, целевое состояние, а также методы для определения допустимых действий и проверки достижения цели.

13. Для чего используется узел Node(problem.initial) в начале функции?

Node(problem.initial) создаёт корневой узел дерева поиска, представляющий начальное состояние задачи, с которого начинается процесс поиска.

14. Что произойдет, если начальное состояние задачи уже является целевым?

Если начальное состояние уже является целевым, функция breadth_first_search немедленно вернет этот узел, завершая поиск.

15. Какую структуру данных использует frontier и почему выбрана именно очередь FIFO?

Frontier использует очередь FIFO для обеспечения расширения узлов в порядке их глубины, что соответствует стратегии поиска в ширину.

16. Какую роль выполняет множество reached?

Множество reached хранит состояния, которые уже были достигнуты, чтобы избежать повторного расширения одного и того же состояния и предотвратить зацикливание.

17. Почему важно проверять, находится ли состояние в множестве reached?

Это предотвращает повторное расширение одного и того же состояния, экономя время и память.

18. Какую функцию выполняет цикл while frontier?

Цикл while frontier продолжает процесс поиска, пока остаются узлы для расширения. Он завершится, когда либо будет найдено решение, либо будут исчерпаны все узлы.

19. Что происходит с узлом, который извлекается из очереди в строке node = rontier.pop()?

Узел извлекается из очереди для дальнейшего расширения, и его дочерние узлы (возможные новые состояния) будут добавлены в очередь.

20. Какова цель функции expand(problem, node)?

Функция expand генерирует дочерние узлы для данного узла, используя допустимые действия и правила перехода в задаче problem.

21. Как определяется, что состояние узла является целевым?

Целевое состояние определяется с помощью метода is_goal объекта problem, который проверяет, соответствует ли текущее состояние целевому.

22. Что происходит, если состояние узла не является целевым, но также не было ранее достигнуто?

Если состояние узла не является целевым и не было достигнуто ранее, оно добавляется в множество reached и очередь frontier для дальнейшего расширения.

23. Почему дочерний узел добавляется в начало очереди с помощью appendleft(child)?

В алгоритме поиска в ширину дочерний узел добавляется в конец очереди, а не в начало, чтобы соблюсти принцип FIFO (очередь с извлечением элементов в порядке их поступления). Это гарантирует, что узлы будут обрабатываться по мере их добавления в очередь, начиная с узлов, расположенных ближе к корневому, и заканчивая узлами на более глубоких уровнях. Использование метода appendleft(child) применимо, скорее, для алгоритма поиска в глубину, который следует стратегии LIFO (стек), где узлы обрабатываются в порядке последнего добавления.

24. Что возвращает функция breadth_first_search , если решение не найдено?

Если решение не найдено, функция возвращает специальное значение failure, показывающее, что достижение цели невозможно.

25. Каково значение узла failure и когда он возвращается?

Узел failure обычно имеет состояние None или «неудача» и длина пути бесконечность. Он возвращается, если поиск завершился, но не было найдено решения.

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