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:

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).

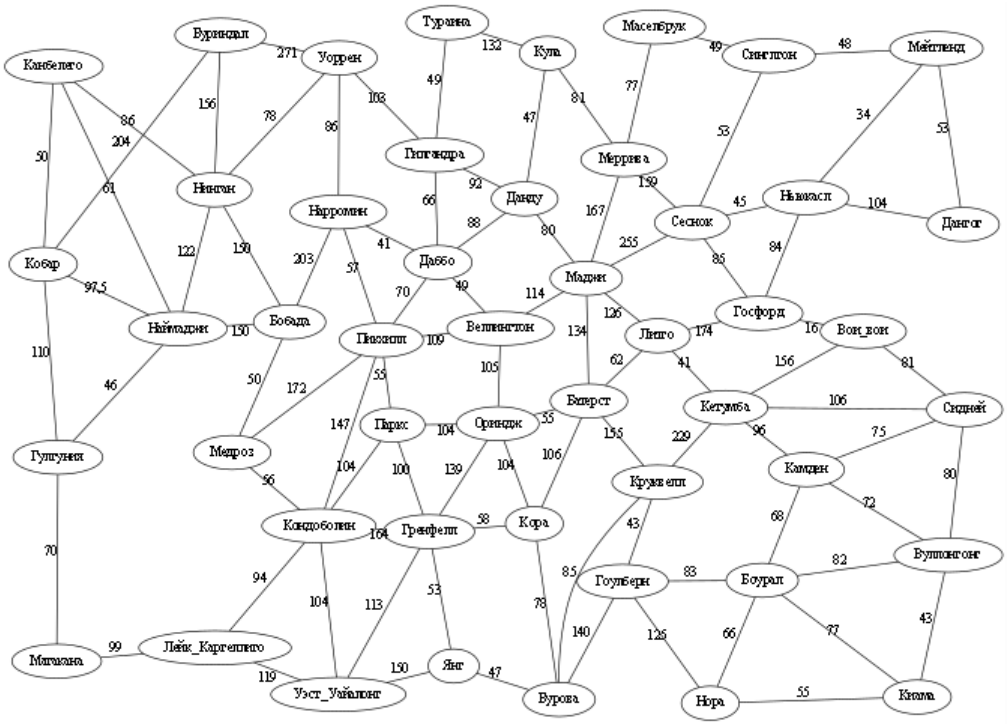

Рисунок 1 – Граф городов Австралии на языке DOT

Граф в программе был описан в виде словаря словарей с узлами и ребрами 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
import math
from abc import ABC, abstractmethod
from collections import deque
```

```python
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}>"

    # Позволяет сравнивать узлы по стоимости пути (для приоритетных очередей)
    def __lt__(self, other):
        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):
    """Раскрываем (расширяем) узел, генерируя все 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(state=s1, parent=node, action=action, path_cost=cost)


def path_actions(node):
    """Последовательность действий, чтобы добраться от корня до данного
узла."""
    if node.parent is None:
        return []
    return path_actions(node.parent) + [node.action]


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


FIFOQueue = deque  # Для поиска в ширину (очередь FIFO)


class PriorityQueue:
    """Очередь с приоритетом, где элемент с минимальным значением key(item)
извлекается первым."""

    def __init__(self, items=(), key=lambda x: x):
        self.key = key
        self.items = []  # внутри храним (priority, item)
        for item in items:
            self.add(item)

    def add(self, item):
        heapq.heappush(self.items, (self.key(item), item))

    def pop(self):
        return heapq.heappop(self.items)[1]

    def top(self):
        return self.items[0][1]

    def __len__(self):
        return len(self.items)


class MapProblem(Problem):
```

```
    """
    Описание конкретной задачи.
    """

    def __init__(self, initial, goal, graph):
        """
        Конструктор класса MapProblem, описывающего конкретную задачу.

        :param initial: Начальный город;
        :param goal: Конечный город.
        :param graph: Граф городов.
        """

        super().__init__(initial=initial, goal=goal)
        self.graph = graph  # словарь словарей для весов рёбер

    def actions(self, state):
        """Возвращаем всех соседей из данного города."""
        return list(self.graph[state].keys())

    def result(self, state, action):
        """Результат перехода: если из 'state' пойти в 'action', то окажемся
в 'action'."""
        return action

    def action_cost(self, s, a, s1):
        """Стоимость рёбра s->s1 (здесь a == s1)."""
        return self.graph[s][s1]


def breadth_first_search(problem):
    """
    Поиск в ширину с учётом весов рёбер.
    Возвращает узел с целевым состоянием или failure, если решения нет.
    Порядок выбора узла из frontier определяется его path_cost (минимальная
сумма весов).
    """

    # Создаём начальный узел
    start_node = Node(problem.initial, path_cost=0)
    # Помещаем его в приоритетную очередь (ключ = path_cost)
    frontier = PriorityQueue(key=lambda node: node.path_cost)
    frontier.add(start_node)

    # Пока есть узлы для расширения
    while len(frontier) > 0:
        # Извлекаем узел с наименьшим path_cost
        node = frontier.pop()

        # Проверяем, не достигли ли мы цели
        if problem.is_goal(node.state):
            return node

        # Расширяем узел и добавляем дочерние узлы в очередь
        for child in expand(problem, node):
            frontier.add(child)

    # Если очередь опустела, а цель не достигнута
    return failure


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

```
    """

    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},
    }

    # Найдем кратчайший путь из города Буриндал в город Сидней:
    problem = MapProblem(initial="Буриндал", goal="Сидней", graph=graph)

    solution_node = breadth_first_search(problem)
    if solution_node is failure:
        print("Путь не найден!")
    else:
        # Восстанавливаем маршрут
        route = path_states(solution_node)
        print("Маршрут:", " -> ".join(route))
        print("Суммарная стоимость:", solution_node.path_cost)


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

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

```
C:\Users\Andrey\AppData\Local\Microsoft\WindowsApps\python3.12.exe C:\Users\Andrey\Desktop\ИИ\Лабораторная_работа_2\Artificial
Маршрут: Буриндал -> Нинган -> Уоррен -> Нарромин -> Даббо -> Веллингтон -> Ориндж -> Батерст -> Литго -> Кетумба -> Сидней
Суммарная 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).

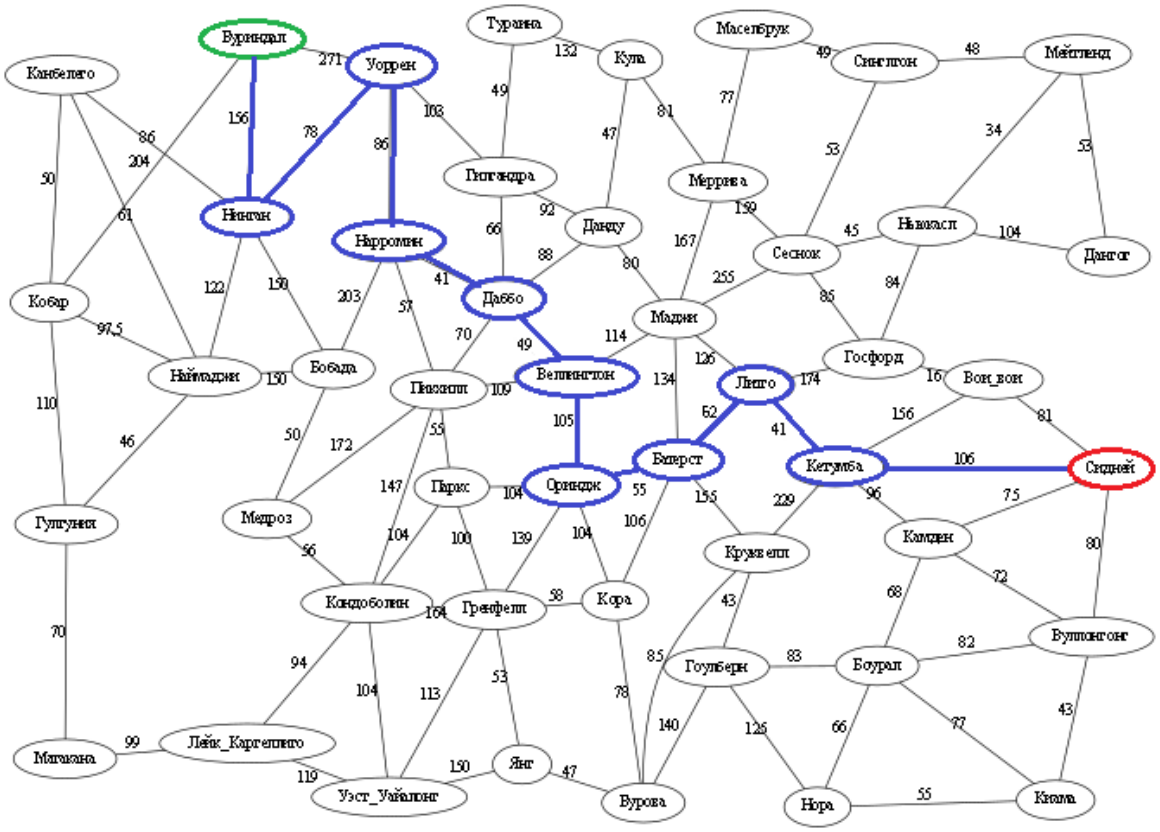

Рисунок 3 – Кратчайший путь на графе

Выполнение индивидуальных заданий:

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 воду, а 1 представляет землю.
Связанные единицы формируют остров. Необходимо подсчитать общее
количество островов в данной матрице. Острова могут соединяться как по
вертикали и горизонтали, так и по диагонали.
"""

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
        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."""
        # Для задачи «островов» конкретного целевого
        # состояния нет поэтому обычно всегда False.
        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


class Node:
    """Узел в дереве/графе поиска."""

    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}>"


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


class IslandsProblem(Problem):
    """
    Дочерний класс Problem: поиск «островов» в бинарной матрице.
    Задача: найти все клетки с 1 (земля) и считать их соединённость
    по 8 направлениям (горизонталь, вертикаль, диагональ).
    """

    def __init__(self, grid):
        super().__init__()
        self.grid = grid
```

```
        self.rows = len(grid)
        self.cols = len(grid[0]) if self.rows > 0 else 0

    def actions(self, state):
        """
        Отдаём все соседние клетки (r2, c2), которые внутри матрицы,
        grid[r2][c2] == 1, не выходят за границы (0 <= r2 < rows, 0 <= c2 <
cols).
        """

        r, c = state
        neighbors = []
        # смещения по 8 направлениям (включая диагонали)
        deltas = [
            (-1, -1),
            (-1, 0),
            (-1, 1),
            (0, -1),
            (0, 1),
            (1, -1),
            (1, 0),
            (1, 1),
        ]
        for dr, dc in deltas:
            r2, c2 = r + dr, c + dc
            if 0 <= r2 < self.rows and 0 <= c2 < self.cols:
                if self.grid[r2][c2] == 1:
                    neighbors.append((r2, c2))
        return neighbors

    def result(self, state, action):
        """Переход в соседнюю клетку (action)."""
        return action


def islands_bfs(problem, start, visited):
    """
    Поиск в ширину (BFS) для обхода одного острова
    Обходит (BFS) все клетки, достижимые из 'start' (т.е. один остров).
    'visited' - множество, куда добавляем все достигнутые клетки.
    """

    from collections import deque

    frontier = deque()
    frontier.append(start)
    visited.add(start)

    while frontier:
        current = frontier.popleft()
        # Выполним расширение (expand)
        for act in problem.actions(current):
            next_state = problem.result(current, act)
            if next_state not in visited:
                visited.add(next_state)
                frontier.append(next_state)


def count_islands_bfs(grid):
    """
    Функция подсчёта количества островов.

    :param grid: Бинарная матрица
    :return: Количество островов.
```

```
    """
    problem = IslandsProblem(grid)
    visited = set()
    count_islands = 0

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

    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == 1:
                # Если это земля и мы ещё не посещали этот участок
                if (r, c) not in visited:
                    # Обходим весь остров BFS
                    islands_bfs(problem, (r, c), visited)
                    count_islands += 1

    return count_islands


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

    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],
    ]

    number_of_islands = count_islands_bfs(grid)
    print("Общее количество островов = ", number_of_islands)


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

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.

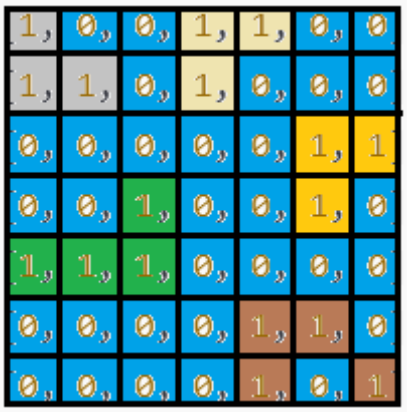

Рисунок 5 – Собственная бинарная матрица

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.
"""

from abc import ABC, abstractmethod
from collections import deque


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):
        self.initial = initial
        self.goal = goal

    @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):
        """Определить, достигнуто ли целевое состояние."""
        return state == self.goal


class LabyrinthProblem(Problem):
    """
    Лабиринт задан бинарной матрицей:
    1 – проход,
    0 – стена.
    Нужно найти путь от initial до goal.
    """

    def __init__(self, labyrinth, initial, goal):
        super().__init__(initial=initial, goal=goal)
        self.labyrinth = labyrinth
        self.rows = len(labyrinth)
        self.cols = len(labyrinth[0]) if labyrinth else 0

    def actions(self, state):
        """Список смежных координат, куда можно перейти по четырём
направлениям (если там 1)."""
        (r, c) = state
        moves = []
        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.labyrinth[nr][nc] == 1:
                    moves.append((nr, nc))
        return moves

    def result(self, state, action):
        """Переход в соседнюю клетку."""
```

```
        return action


def bfs_labyrinth(problem):
    """
    Ищет кратчайший путь (по числу шагов) от problem.initial до problem.goal
    с помощью алгоритма поиска в ширину (BFS).
    Возвращает длину пути или None, если путь не найден.
    """

    start = problem.initial
    goal = problem.goal

    # Если начальное состояние совпадает с целевым
    if start == goal:
        return 0

    queue = deque([(start, 0)])

    # Посещенные состояния
    visited = set()
    visited.add(start)

    while queue:
        (current, dist) = queue.popleft()

        for action in problem.actions(current):
            next_state = problem.result(current, action)
            if next_state not in visited:
                visited.add(next_state)
                # Проверяем, не является ли новая клетка целевым состоянием
                if problem.is_goal(next_state):
                    return dist + 1
                # Если нет, добавляем в очередь с расстоянием dist+1
                queue.append((next_state, dist + 1))

    # Если очередь опустела и целевое состояние не достигнуто — пути нет
    return None


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

    problem = LabyrinthProblem(labyrinth, initial, goal)
    distance = bfs_labyrinth(problem)

    if distance is None:
```

```
        print("Путь не найден.")
    else:
        print("Длина пути: ", distance)


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

Для проверки работоспособности программы был добавлен пример лабиринта из 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).

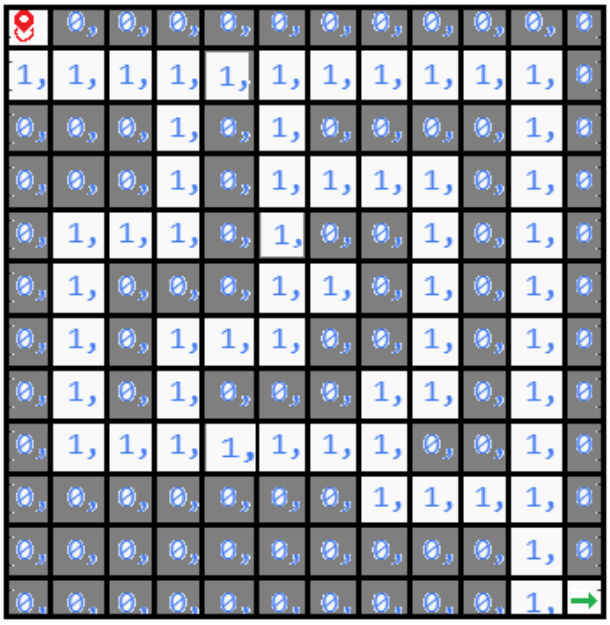

Рисунок 8 – Схема лабиринта

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).

```
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.
"""

import math
from collections import deque
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, является ли состояние целевым."""
        return state == self.goal

    def action_cost(self, s, a, s1):
        """Стоимость шага, по умолчанию 1."""
        return 1

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


class Node:

    def __init__(self, state, parent=None, action=None, path_cost=0.0):
        self.state = state       # текущее состояние (кортеж объёмов)
        self.parent = parent     # родительский узел
        self.action = action     # действие, которое привело к этому узлу
        self.path_cost = path_cost

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

    def __lt__(self, other):
        """Для приоритетных очередей."""
        return self.path_cost < other.path_cost


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


def expand(problem, node):
    """
    Генерируем (расширяем) дочерние узлы, применяя все действия к state.
    """

    s = node.state
    for action in problem.actions(s):
        s_next = problem.result(s, action)
        cost = node.path_cost + problem.action_cost(s, action, s_next)
        yield Node(state=s_next, 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]
```

```python
def path_states(node):
    """
    Восстанавливаем последовательность состояний (кувшины) от корня до узла.
    """

    if node.parent is None:
        return [node.state]
    return path_states(node.parent) + [node.state]


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

    # Создаём начальный узел
    node = Node(problem.initial)
    # Проверка: если начальное состояние уже цель
    if problem.is_goal(node.state):
        return node

    # Очередь (FIFO)
    frontier = deque([node])

    # Набор посещённых состояний
    explored = set()
    explored.add(node.state)

    # Пока очередь не пуста
    while frontier:
        current = frontier.popleft()

        # Расширяем текущий узел
        for child in expand(problem, current):
            if child.state not in explored:
                # Если это целевое состояние — возвращаем
                if problem.is_goal(child.state):
                    return child
                # Иначе помечаем как посещённое и добавляем в очередь
                explored.add(child.state)
                frontier.append(child)

    # Если очередь опустела, решения нет
    return failure


class WaterJugProblem(Problem):
    """
    Класс для задачи о льющихся кувшинах.
    Размер кувшинов - sizes
    Начальное состояние initial (кортеж),
    цель (целевой объём "goal").
    """

    def __init__(self, initial, goal, sizes):
        super().__init__(initial=initial, goal=goal)
        self.sizes = sizes  # кортеж допустимых объёмов

    def is_goal(self, state):
        """
        Цель достигается, если в одном из кувшинов есть объём == self.goal.
        """
```

```
        return any(volume == self.goal for volume in state)

    def actions(self, state):
        """
        Действия:
        (Fill, i): Наполнить i-й кувшин
        (Dump, i): Вылить i-й кувшин
        (Pour, i, j): Перелить из i-го в j-й
        """

        actions_list = []
        n = len(self.sizes)

        for i in range(n):
            (wi, si) = (state[i], self.sizes[i])

            # Fill i
            if wi < si:
                actions_list.append(("Fill", i))

            # Dump i
            if wi > 0:
                actions_list.append(("Dump", i))

            # Pour i -> j
            if wi > 0:
                for j in range(n):
                    if i != j:
                        wj, sj = state[j], self.sizes[j]
                        # Есть смысл переливать, если j не полон
                        if wj < sj:
                            actions_list.append(("Pour", i, j))

        return actions_list

    def result(self, state, action):
        """
        Применение действия к состоянию.
        """

        state = list(state)  # переводим в список для изменения
        a = action

        if a[0] == "Fill":
            i = a[1]
            state[i] = self.sizes[i]

        elif a[0] == "Dump":
            i = a[1]
            state[i] = 0

        elif a[0] == "Pour":
            i, j = a[1], a[2]
            amount_i = state[i]
            amount_j = state[j]
            capacity_j = self.sizes[j]

            # Сколько можем перелить: либо всё из i, либо пока j не
наполнится
            can_pour = min(amount_i, capacity_j - amount_j)
            state[i] -= can_pour
            state[j] += can_pour

        return tuple(state)  # возвращаем неизменяемый кортеж
```

```python
def main():
    """
    Главная функция программы.
    """
    initial = (1, 1, 1)
    goal = 13
    sizes = (2, 16, 32)

    problem = WaterJugProblem(initial, goal, sizes)
    solution_node = bfs(problem)

    if solution_node is failure:
        print("Решение не найдено!")
    else:
        # Восстановим и выведем последовательность действий
        acts = path_actions(solution_node)
        print("Последовательность действий:", acts)

        # Восстановим и выведем последовательность состояний
        state_sequence = path_states(solution_node)
        print("Последовательность состояний:", state_sequence)


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

Результаты работы программы с предложенным в примере условиям (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\AppData\Local\pypoetry\Cache\virtualenvs\ai-lab-2-dtOMEYOu-py3.12\Scripts\
Последовательность действий: [('Fill', 1), ('Pour', 1, 0), ('Dump', 0), ('Pour', 1, 0)]
Последовательность состояний: [(1, 1, 1), (1, 16, 1), (2, 15, 1), (0, 15, 1), (2, 13, 1)]
```

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, 6, 10, 15, 20)  # размеры кувшинов
```

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

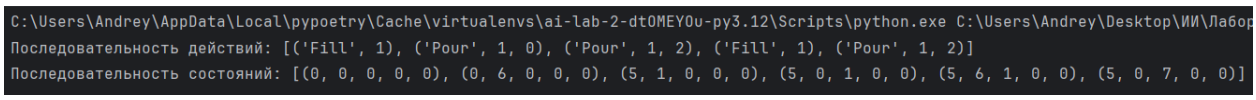

Рисунок 11 – Отмеренные 7 литров в кувшинах

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.