

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

тут перспективной инженерии
Департамент цифровых, робототехнических систем и электроники

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ №2
дисциплины «Искусственный интеллект в профессиональной сфере»
Вариант 2

Выполнил:
Иващенко Олег Андреевич
3 курс, группа ИВТ-б-о-22-1,
09.03.02 «Информационные и
вычислительные машины»,
направленность (профиль)
«Программное обеспечение средств
вычислительной техники и
автоматизированных систем»

(подпись)

Руководитель практики:
Воронкин Роман Александрович,
доцент департамента цифровых,
робототехнических систем и
электроники

(подпись)

Отчет защищен с оценкой _____ Дата защиты _____

Ставрополь, 2024 г.

Тема: «Исследование поиска в ширину»

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

Порядок выполнения работы:

Задание 1. 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, так и по диагонали.

Листинг 1 – Код программы общего задания 1 (general_1.py)

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

def count_islands(grid):
    """
    Метод подсчёта количества островов при помощи метода поиска в ширину.

    Аргументы:
        grid (list) - бинарная матрица, где 1 - остров, 0 - вода

    Возвращает:
        int - количество островов
    """

    if not grid:
        return 0

    rows = len(grid)
    cols = len(grid[0])
    island_count = 0

    def bfs(r, c):
        queue = [(r, c)]
        while queue:
            row, col = queue.pop(0)

            for dr in [-1, 0, 1]:
                for dc in [-1, 0, 1]:
                    if dr == 0 and dc == 0:
                        continue
                    nr, nc = row + dr, col + dc
                    if 0 <= nr < rows and 0 <= nc < cols and grid[nr][nc] == 1:
                        grid[nr][nc] = 0
                        queue.append((nr, nc))

    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == 1:
                bfs(r, c)
                island_count += 1

    return island_count
```

```

for i in range(rows):
    for j in range(cols):
        if grid[i][j] == 1:
            island_count += 1
            grid[i][j] = 0
            bfs(i, j)

return island_count

def main():
    """
    Основная функция программы.
    """
    # Исходные данные
    grid = [
        [1, 0, 1, 0, 1, 1, 0, 1, 0, 1],
        [0, 1, 0, 0, 0, 0, 0, 0, 1, 0],
        [1, 0, 0, 0, 1, 1, 0, 0, 0, 1],
        [0, 0, 0, 1, 0, 0, 1, 0, 0, 0],
        [1, 0, 0, 0, 1, 0, 0, 1, 0, 1],
        [0, 1, 0, 0, 0, 0, 1, 0, 0, 0],
        [1, 0, 1, 0, 0, 1, 0, 1, 0, 1],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [1, 0, 1, 0, 1, 0, 0, 1, 0, 0],
        [0, 1, 0, 1, 0, 0, 1, 0, 1, 0],
    ]

    island_count = count_islands(grid)
    print(f"Общее количество островов: {island_count}")

if __name__ == "__main__":
    main()

```

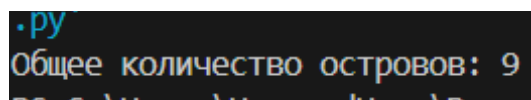


Рисунок 1 – Вывод программы

Задание 2. Необходимо организовать поиск кратчайшего пути через лабиринт, используя алгоритм в ширину (BFS). Лабиринт представлен в виде бинарной матрицы, где 1 обозначает проход, а 0 – стену.

Листинг 2 – Код программы общего задания 2 (general_2.py)

```

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

```

```

def shortest_path(maze, initial, goal):
    """
    Находит кратчайший путь в лабиринте от начальной точки до целевой.

    Аргументы:
        maze (list) - Бинарная матрица, представляющая лабиринт, где
            1 обозначает проход;
            0 обозначает стену.

        initial (tuple) - Координаты начальной точки (строка, столбец).
        goal (tuple) - Координаты целевой точки (строка, столбец).

    Возвращает:
        int - Длина кратчайшего пути. Если путь не найден, возвращает -1.
    """

    rows = len(maze)
    cols = len(maze[0])
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    if maze[initial[0]][initial[1]] != 1 or maze[goal[0]][goal[1]] != 1:
        return -1

    queue = [(initial[0], initial[1], 0)]

    visited = set()
    visited.add((initial[0], initial[1]))

    while queue:
        row, col, distance = queue.pop(0)

        if (row, col) == goal:
            return distance

        for dr, dc in directions:
            nr, nc = row + dr, col + dc
            if 0 <= nr < rows and 0 <= nc < cols and maze[nr][nc] == 1 and (nr, nc) not in visited:
                visited.add((nr, nc))
                queue.append((nr, nc, distance + 1))

    return -1

def main():
    """
    Основная функция программы.
    """
    # Пример лабиринта
    maze = [
        [1, 0, 1, 1, 1, 0, 1, 1, 1, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 0, 1],
        [1, 1, 1, 0, 1, 1, 1, 0, 0, 1],
    ]

```

```

[0, 0, 0, 0, 0, 0, 1, 0, 0, 1],
[1, 1, 1, 1, 1, 1, 1, 0, 0, 1],
[1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
]
initial = (0, 0) # Начальная точка
goal = (9, 9) # Целевая точка

path_length = shortest_path(maze, initial, goal)

if path_length != -1:
    print(f"Длина кратчайшего пути: {path_length}")
else:
    print("Путь не найден.")

if __name__ == "__main__":
    main()

```

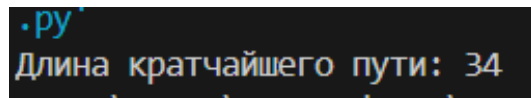


Рисунок 2 – Результат вывода программы

Задание 3. Реализовать алгоритм поиска в ширину (BFS) для решения задачи о льющихся кувшинах, где цель состоит в том, чтобы получить заданный объём воды в одном из кувшинов.

Листинг 3 – Код программы общего задания (general_3.py)

```

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

def bfs_jugs(initial, goal, sizes):
    """
    Решает задачу о льющихся кувшинах с использованием BFS.

    Параметры:
    initial (tuple) - Начальное состояние (количество воды в каждом кувшине).
    goal (int) - Целевой объем воды.
    sizes (tuple) - Размеры (емкости) кувшинов.

    Возвращает:
    list - Последовательность действий для достижения цели.
    list - Последовательность состояний, соответствующих действиям.
    """

```

```

"""
from collections import deque

if goal in initial:
    return [], [initial]

queue = deque([(initial, [], [initial])])

visited = set()
visited.add(initial)

while queue:
    state, actions, states = queue.popleft()

    for action in get_possible_actions(state, sizes):
        new_state = apply_action(state, action, sizes)
        if new_state not in visited:
            new_actions = actions + [action]
            new_states = states + [new_state]

            if goal in new_state:
                return new_actions, new_states
            visited.add(new_state)
            queue.append((new_state, new_actions, new_states))

return [], []

def get_possible_actions(state, sizes):
    """
    Возвращает список возможных действий для текущего состояния.

    Аргументы:
        state (tuple) - Текущее состояние (количество воды в каждом кувшине).
        sizes (tuple) - Размеры (емкости) кувшинов.

    Возвращает:
        list - Список возможных действий.
    """
    actions = []
    num_jugs = len(state)

    for i in range(num_jugs):
        if state[i] < sizes[i]:
            actions.append(('Fill', i))
        if state[i] > 0:
            actions.append(('Dump', i))

    for i in range(num_jugs):
        for j in range(num_jugs):
            if i != j and state[i] > 0 and state[j] < sizes[j]:
                actions.append(('Pour', i, j))

```

```
return actions
```

```
def apply_action(state, action, sizes):
```

```
    """
```

Применяет действие к текущему состоянию и возвращает новое состояние.

Аргументы:

state (tuple) - Текущее состояние (количество воды в каждом кувшине).

action (tuple) - Действие (Fill, Dump или Pour).

sizes (tuple) - Размеры (емкости) кувшинов.

Возвращает:

tuple - Новое состояние после применения действия.

```
    """
```

```
    state = list(state)
```

```
    action_type = action[0]
```

```
    if action_type == 'Fill':
```

```
        i = action[1]
```

```
        state[i] = sizes[i]
```

```
    elif action_type == 'Dump':
```

```
        i = action[1]
```

```
        state[i] = 0
```

```
    elif action_type == 'Pour':
```

```
        i, j = action[1], action[2]
```

```
        amount = min(state[i], sizes[j] - state[j])
```

```
        state[i] -= amount
```

```
        state[j] += amount
```

```
    return tuple(state)
```

```
def main():
```

```
    """
```

Основная функция программы.

```
    """
```

```
    # Входные данные
```

```
    initial = (0, 0, 0) # Изначальное состояние
```

```
    goal = 6 # Необходимая цель
```

```
    sizes = (3, 5, 8) # Объём кувшинов
```

```
    actions, states = bfs_jugs(initial, goal, sizes)
```

```
    if actions:
```

```
        print("Последовательность действий:", actions)
```

```
        print("Последовательность состояний:", states)
```

```
    else:
```

```
        print("Решение не найдено.")
```

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

```
Последовательность действий: [('Fill', 0), ('Fill', 2), ('Pour', 0, 1), ('Pour', 2, 1)]  
Последовательность состояний: [(0, 0, 0), (3, 0, 0), (3, 0, 8), (0, 3, 8), (0, 5, 6)]  
PS C:\Users\UnnamedUser\Documents\СКФУ\Курс 3\ИИ\AI 2\exes>
```

Рисунок 3 – Результат вывода программы

Индивидуальное задание. Для построенного графа лабораторной работы 1 напишите программу на языке программирования Python, которая с помощью алгоритма поиска в ширину находит минимальное расстояние между начальным и конечным пунктами. Сравните найденное решение с решением, полученным вручную.

Листинг 4 – Код программы индивидуального задания

```
#!/usr/bin/env python3  
# -*- coding: utf-8 -*-  
  
import itertools  
import networkx as nx  
import matplotlib.pyplot as plt  
from collections import deque  
  
# Пример входных данных (можно заменить своими)  
cities = {  
    'Лонгфорд': {'Ньюнхем': 31.4, 'Экстон': 37.1, 'Бреона': 51.4, 'Конара': 43.2, 'Дерби': 111.9},  
    'Конара': {'Сент-Мэрис': 73.9, 'Кэмпбелл-Таун': 12.5},  
    'Кэмпбелл-Таун': {'Танбридж': 27.1, 'Лейк Лик': 34.8},  
    'Лейк Лик': {'Бичено': 57, 'Суонси': 33.8},  
    'Ньюнхем': {'Джордж Таун': 44.3, 'Лилидейл': 21.3},  
    'Джордж Таун': {},  
    'Лилидейл': {'Лебрина': 8.7},  
    'Лебрина': {'Пайперс Брук': 13.3, 'Бридпорт': 27},  
    'Экстон': {'Элизабет Таун': 18.4, 'Мол Крик': 30.8, 'Бреона': 38.4},  
    'Элизабет Таун': {'Шеффилд': 28, 'Девонпорт': 42.5},  
    'Девонпорт': {},  
    'Шеффилд': {'Мойна': 31.7},  
    'Мойна': {},  
    'Бреона': {'Рейнольдс Лейк': 11.2, 'Шеннон': 26.5, 'Ботуэлл': 66.7},  
    'Рейнольдс Лейк': {'Миена': 18.5},  
    'Мол Крик': {'Шеффилд': 51.5},  
    'Миена': {'Тарралия': 59.2},  
    'Шеннон': {'Миена': 17.2},  
    'Тарралия': {'Уэйятина': 16.5},  
    'Уэйятина': {},  
}
```



```

'Ботуэлл': {},
'Танбридж': {},
'Литл Суонпорт': {},
'Суонси': {'Литл Суонпорт': 27.7},
'Сент-Мэрис': {'Гарденс': 55.8},
'Гарденс': {'Дерби': 61.1},
'Дерби': {},
'Пайперс Брук': {},
'Бридпорт': {},
}

start_city = 'Сент-Мэрис' # Исходный город
end_city = 'Мойна' # Целевой город

# Создание симметричного графа
def create_symmetric_graph(cities):
    symmetric_cities = {}
    for city, neighbors in cities.items():
        if city not in symmetric_cities:
            symmetric_cities[city] = {}
        for neighbor, distance in neighbors.items():
            symmetric_cities[city][neighbor] = distance
            if neighbor not in symmetric_cities:
                symmetric_cities[neighbor] = {}
            symmetric_cities[neighbor][city] = distance
    return symmetric_cities

symmetric_cities = create_symmetric_graph(cities)

# Поиск кратчайшего пути с использованием BFS
def bfs_shortest_path(graph, start, end):
    queue = deque([(start, [start])]) # Очередь: (текущий город, путь)
    visited = set()

    while queue:
        current_city, path = queue.popleft()
        if current_city == end:
            return path
        if current_city not in visited:
            visited.add(current_city)
            for neighbor in graph.get(current_city, {}):
                if neighbor not in visited:
                    queue.append((neighbor, path + [neighbor]))
    return None

# Вычисление длины маршрута
def calculate_distance(route, graph):
    distance = 0
    for i in range(len(route) - 1):
        distance += graph[route[i]][route[i + 1]]
    return distance

```

```

# Построение графа и отображение маршрутов
def plot_graph(cities, shortest_route):
    G = nx.DiGraph()

    # Добавление рёбер с весами
    for city, neighbors in cities.items():
        for neighbor, weight in neighbors.items():
            G.add_edge(city, neighbor, weight=weight)

    pos = nx.spring_layout(G) # Позиционирование узлов

    # Отображение графа
    plt.figure(figsize=(12, 8))
    edge_labels = nx.get_edge_attributes(G, 'weight')
    nx.draw(G, pos, with_labels=True, node_size=700, node_color='lightblue', font_size=8)
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)

    # Подсветка самого короткого маршрута
    if shortest_route:
        shortest_edges = [(shortest_route[i], shortest_route[i + 1]) for i in range(len(shortest_route)
- 1)]
        nx.draw_networkx_edges(G, pos, edgelist=shortest_edges, edge_color='red', width=2)

    plt.title("Граф маршрутов")
    plt.show()

# Основной код
shortest_route = bfs_shortest_path(symmetrical_cities, start_city, end_city)

if shortest_route:
    shortest_distance = calculate_distance(shortest_route, symmetrical_cities)
    print(f"Самый короткий маршрут: {' -> '.join(shortest_route)}, Расстояние:
{round(shortest_distance, 1)} км")
    plot_graph(symmetrical_cities, shortest_route)
else:
    print("Маршрут не найден.")

```

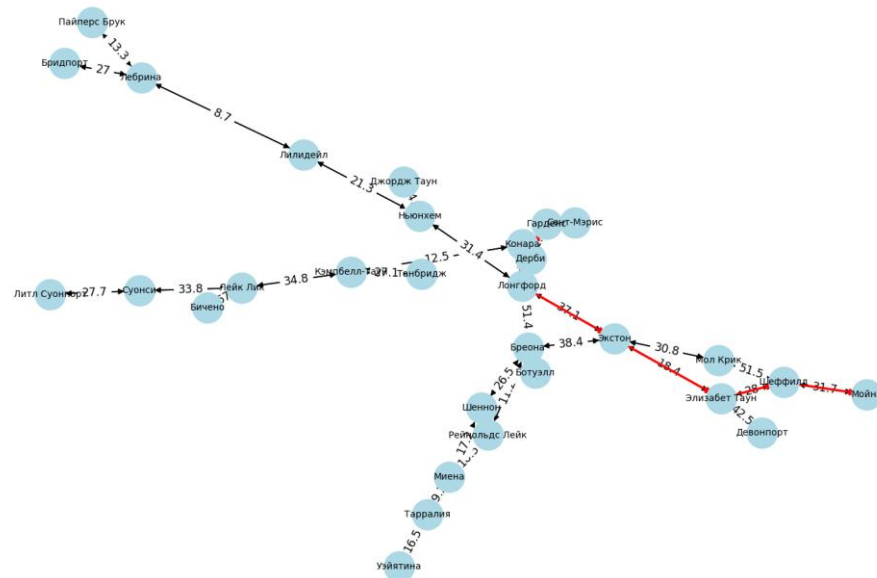


Рисунок 4 – Построенный граф (красным выделен кратчайший маршрут)

```

C:\Users\Unnam...
Python 3.10.11 (tags/v3.10.11:2023-01-11, 11:02:00) [AMD64]
...
Самый короткий маршрут: Сент-Мэрис -> Конара -> Лонгфорд -> Экстон -> Элизабет Таун -> Шеффилд -> Мойна, Расстояние: 232.3 км

```

Рисунок 5 – Вывод кратчайшего маршрута в консоли

Ответы на контрольные вопросы:

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

В стратегии поиска в ширину первым делом расширяется наименее глубокий из нераскрытых узлов. Этот процесс реализуется с использованием очереди типа «первый пришёл – первый ушёл» (FIFO).

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

Это обеспечивает обработку узлов по уровням глубины, начиная с корневого.

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

Они обрабатываются первыми, так как находятся в начале очереди.

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

Узлы первого уровня (дочерние узлы корневого узла).

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

Это гарантирует нахождение кратчайшего пути (оптимального по глубине).

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

Зависит от коэффициента разветвления (b) и глубины (d): $O(b^d)$.

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

Определяется необходимостью хранения всех узлов текущего уровня: $O(b^d)$.

8. В каких случаях поиск в ширину считается полным?

Поиск считается полным, если коэффициент разветвления конечен и решение существует.

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

BFS хранит все узлы текущего уровня, что требует много памяти.

10. В чём заключается оптимальность поиска в ширину?

Находит кратчайший путь в графе или дереве (оптимален по глубине).

11. Какую задачу решает функция `breadth_first_search`?

Решает задачу поиска пути или решения с использованием BFS.

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

Описывает задачу, включая начальное состояние, цель и возможные действия.

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

Создаёт начальный узел на основе начального состояния задачи.

14. Что произойдёт, если начальное состояние задачи уже является целеным?

Функция сразу возвращает этот узел как решение.

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

Очередь FIFO, чтобы узлы обрабатывались в порядке их добавления.

16. Какую роль выполняет множество `reached`?

Хранит достигнутые состояния для избежания повторной обработки.

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

Чтобы избежать циклов и избыточных вычисления.

18. Какую функцию выполняет цикл `while frontier`?

Обрабатывает узлы до тех пор, пока очередь не опустеет или не найдётся решение.

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

Узел извлекается для проверки и расширения (поиска дочерних узлов).

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

Генерирует дочерние узлы на основе текущего узла.

21. Как определяется, что состояние узла является целевым?

Сравнением состояния узла с целевым состоянием задачи.

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

Узел добавляется в очередь для дальнейшего исследования.

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

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

Функция возвращает значение, указывающее на отсутствие решения (например, `None`).

25. Каково значение узла `failure` и когда он возвращается?

Указывает на неудачи поиска и возвращается, если решение не найдено.

Выводы: В процессе выполнения лабораторной работы были приобретены навыки по работе с поиском в ширину с помощью языка

программирования Python версии 3.x, проработаны примеры и выполнены индивидуальные задания.

Ссылка на репозиторий:

[IUnnameUserI/AI_2: Искусственный интеллект в профессиональной сфере.](#)
[Лабораторная работа 2](#)