

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

Институт перспективной инженерии
Департамент цифровых, робототехнических систем и электроники

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ №3
дисциплины
«Искусственный интеллект в профессиональной сфере»
Вариант 1

Выполнил:
Бабенко Артём Тимофеевич
3 курс, группа ИВТ-б-о-22-1,
09.03.01 «Информатика и
вычислительная техника»,
направленность (профиль)
«Программное обеспечение средств
вычислительной техники и
автоматизированных систем», очная
форма обучения

(подпись)

Проверил:
Ассистент департамента цифровых,
робототехнических систем и
электроники Богданов С.С

(подпись)

Отчет защищен с оценкой _____ Дата защиты _____

Ставрополь, 2024 г.

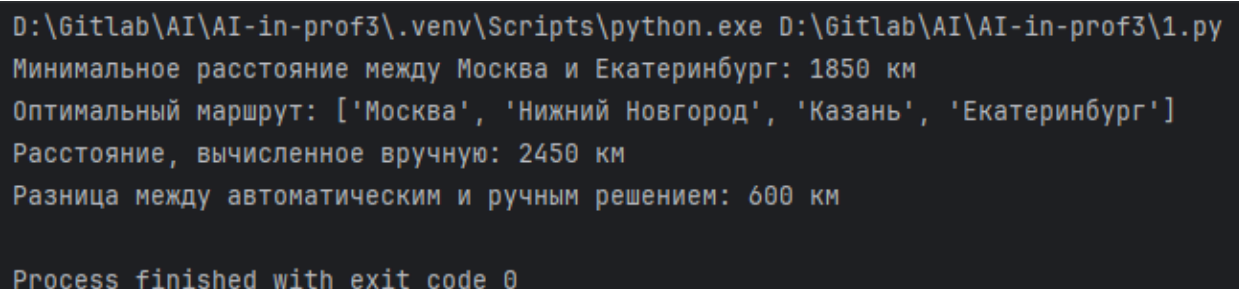
Тема: Исследование поиска в глубину

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

Порядок выполнения работы:

Задание 1. Для построенного графа лабораторной работы 1 (имя файла начинается с PR.AI.001.) напишите программу на языке программирования Python, которая с помощью алгоритма поиска в глубину находит минимальное расстояние между начальным и конечным пунктами. Сравните найденное решение с решением, полученным вручную.

Результат работы программы:



```
D:\Gitlab\AI\AI-in-prof3\.venv\Scripts\python.exe D:\Gitlab\AI\AI-in-prof3\1.py
Минимальное расстояние между Москва и Екатеринбург: 1850 км
Оптимальный маршрут: ['Москва', 'Нижний Новгород', 'Казань', 'Екатеринбург']
Расстояние, вычисленное вручную: 2450 км
Разница между автоматическим и ручным решением: 600 км

Process finished with exit code 0
```

Рисунок 1 – Результат работы программы

Код программы:

```
import networkx as nx
import matplotlib.pyplot as plt

# Шаг 1: Создание списка населённых пунктов и расстояний между ними
places = [
    "Москва", "Санкт-Петербург", "Новосибирск", "Екатеринбург", "Казань",
    "Нижний Новгород", "Челябинск", "Омск", "Самара", "Ростов-на-Дону",
    "Уфа", "Красноярск", "Воронеж", "Пермь", "Волгоград", "Краснодар",
    "Саратов", "Тюмень", "Тольятти", "Ижевск", "Барнаул"
]

# Матрица расстояний
distances = {
    ("Москва", "Санкт-Петербург"): 650,
    ("Москва", "Казань"): 800,
    ("Москва", "Нижний Новгород"): 450,
    ("Санкт-Петербург", "Новосибирск"): 3200,
    ("Санкт-Петербург", "Казань"): 1000,
    ("Новосибирск", "Екатеринбург"): 1700,
    ("Екатеринбург", "Казань"): 1200,
    ("Екатеринбург", "Челябинск"): 300,
    ("Екатеринбург", "Омск"): 900,
```

```

("Казань", "Нижний Новгород"): 200,
("Казань", "Самара"): 600,
("Нижний Новгород", "Челябинск"): 1500,
("Челябинск", "Омск"): 800,
("Омск", "Красноярск"): 1200,
("Самара", "Ростов-на-Дону"): 1100,
("Самара", "Уфа"): 400,
("Ростов-на-Дону", "Краснодар"): 400,
("Уфа", "Красноярск"): 2000,
("Уфа", "Пермь"): 500,
("Уфа", "Екатеринбург"): 500,
("Казань", "Екатеринбург"): 1200,
("Красноярск", "Воронеж"): 3000,
("Пермь", "Волгоград"): 1000,
("Волгоград", "Краснодар"): 500,
("Краснодар", "Саратов"): 700,
("Саратов", "Тюмень"): 1200,
("Тюмень", "Тольятти"): 1300,
("Тольятти", "Ижевск"): 400,
("Ижевск", "Барнаул"): 2000,
}

```

Шаг 2: Создание графа

```
G = nx.Graph()
```

Добавление узлов (населённых пунктов)

```
G.add_nodes_from(places)
```

Добавление рёбер с весами (расстояниями)

```
for (place1, place2), distance in distances.items():
```

```
    G.add_edge(place1, place2, weight=distance)
```

Шаг 3: Алгоритм поиска в глубину (DFS) для нахождения минимального расстояния

```
def dfs_shortest_path(graph, start, end):
```

```
    # Проверка, существуют ли стартовый и конечный узлы в графе
```

```
    if not graph.has_node(start) or not graph.has_node(end):
```

```
        return None, float('inf')
```

```
    # Инициализация переменных
```

```
    min_distance = float('inf')
```

```
    best_path = None
```

```
def dfs(current_node, current_path, current_distance):
```

```
    nonlocal min_distance, best_path
```

```
    # Если достигли целевой узел
```

```
    if current_node == end:
```

```
        if current_distance < min_distance:
```

```
            min_distance = current_distance
```

```
            best_path = current_path[:]
```

```
    return
```

```

# Исследуем соседей
for neighbor in graph.neighbors(current_node):
    if neighbor not in current_path: # Избегаем циклов
        edge_weight = graph[current_node][neighbor]['weight']
        current_path.append(neighbor)
        dfs(neighbor, current_path, current_distance + edge_weight)
        current_path.pop() # Возвращаемся назад (backtracking)

# Запуск DFS
dfs(start, [start], 0)

return best_path, min_distance

# Шаг 4: Выбор начального и конечного пунктов
start_city = "Москва"
end_city = "Екатеринбург"

# Поиск минимального расстояния с помощью DFS
best_path, min_distance = dfs_shortest_path(G, start_city, end_city)

# Вывод результатов
if best_path is None:
    print(f"Путь между {start_city} и {end_city} не найден.")
else:
    print(f"Минимальное расстояние между {start_city} и {end_city}: {min_distance} км")
    print(f"Оптимальный маршрут: {best_path}")

# Сравнение с решением, полученным вручную
# Предположим, что вручную вычисленное расстояние равно 2450 км
manual_distance = 2450
print(f"Расстояние, вычисленное вручную: {manual_distance} км")
print(f"Разница между автоматическим и ручным решением: {abs(min_distance - manual_distance)} км")

# Визуализация графа и маршрута
pos = nx.spring_layout(G, seed=42)
nx.draw(G, pos, with_labels=True, node_size=500, node_color="lightblue", font_size=10,
font_weight="bold")

# Выделение маршрута (если он найден)
if best_path:
    route_edges = [(best_path[i], best_path[i + 1]) for i in range(len(best_path) - 1)]
    nx.draw_networkx_edges(G, pos, edgelist=route_edges, edge_color='red', width=2)

plt.title("Граф дорог между населёнными пунктами")
plt.show()

```

Задание 2. Алгоритм заливки.

Результат работы программы:

```

D:\Gitlab\AI\AI-in-prof3\.venv\Scripts\python.exe D:\Gitlab\AI\AI-in-prof3\2.py
Исходная матрица:
J V R L K K Z
R F P J C C L
R B V S W N Y
Q Q S N U P A
K R R Y W L M
I I T E C V S
K P G A Z J G

Матрица после заливки:
J V R L K K Z
R F P J C C L
R B V S W N Y
Q Q S X U P A
K R R Y W L M
I I T E C V S
K P G A Z J G

Process finished with exit code 0

```

Рисунок 2 – Результат работы программы

Код программы:

```

def flood_fill(matrix, x, y, new_letter):
    # Получаем старую букву в точке (x, y)
    old_letter = matrix[x][y]

    # Если старая буква равна новой, ничего делать не нужно
    if old_letter == new_letter:
        return matrix

    # Рекурсивная функция для заливки
    def fill(x, y):
        # Проверяем границы матрицы
        if x < 0 or x >= len(matrix) or y < 0 or y >= len(matrix[0]):
            return

        # Если текущая буква не равна старой, выходим
        if matrix[x][y] != old_letter:
            return

        # Заменяем текущую букву на новую
        matrix[x][y] = new_letter

```

```

    # Рекурсивно вызываем функцию для соседних ячеек
    fill(x + 1, y) # Вниз
    fill(x - 1, y) # Вверх
    fill(x, y + 1) # Вправо
    fill(x, y - 1) # Влево

# Запускаем заливку
fill(x, y)
return matrix

# Создаем матрицу размером 7x7 с буквами
import random

letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
matrix = [[random.choice(letters) for _ in range(7)] for _ in range(7)]

# Вывод исходной матрицы
print("Исходная матрица:")
for row in matrix:
    print(' '.join(row))

# Начальная точка и новая буква
x, y = 3, 3 # Начальная точка
new_letter = 'X' # Новая буква

# Выполняем заливку
result = flood_fill(matrix, x, y, new_letter)

# Вывод результата
print("\nМатрица после заливки:")
for row in result:
    print(' '.join(row))

```

Задание 3. Поиск самого длинного пути в матрице.

Результат работы программы:

```
D:\Gitlab\AI\AI-in-prof3\.venv\Scripts\python.exe D:\Gitlab\AI\AI-in-prof3\3.py
Длина самого длинного пути: 5

Process finished with exit code 0
```

Рисунок 3 – Результат работы программы

Код программы:

```
def longest_alphabetical_path(matrix, start_char):
    # Получаем размер матрицы
    rows, cols = len(matrix), len(matrix[0])

    # Проверяем, находится ли координата внутри матрицы
    def is_valid(x, y):
        return 0 <= x < rows and 0 <= y < cols

    # Функция для получения следующего символа в алфавите
    def next_char(c):
        if c == 'Z':
            return None # После 'Z' нет следующего символа
        return chr(ord(c) + 1)

    # Рекурсивная функция для поиска пути
    def dfs(x, y, current_char, memo):
        # Если результат уже вычислен, используем его
        if memo[x][y] != -1:
            return memo[x][y]

        max_length = 1 # Минимальная длина пути — это сам символ

        # Проверяем все восемь направлений
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1, 1), (1, -1), (1, 1)]
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
```

```

        if is_valid(nx, ny): # Проверяем, что новая координата внутри матрицы
            if matrix[nx][ny] == next_char(current_char): # Проверяем
последовательность

                length = 1 + dfs(nx, ny, matrix[nx][ny], memo) # Продолжаем путь
                max_length = max(max_length, length)

# Сохраняем результат в мемоизацию
memo[x][y] = max_length
return max_length

# Ищем все возможные стартовые точки для заданного символа
max_path_length = 0
memo = [[-1 for _ in range(cols)] for _ in range(rows)] # Мемоизация

for i in range(rows):
    for j in range(cols):
        if matrix[i][j] == start_char:
            path_length = dfs(i, j, start_char, memo)
            max_path_length = max(max_path_length, path_length)

return max_path_length

# Пример использования
matrix = [
    ['A', 'B', 'C', 'D', 'E'],
    ['F', 'G', 'H', 'I', 'J'],
    ['K', 'L', 'M', 'N', 'O'],
    ['P', 'Q', 'R', 'S', 'T'],
    ['U', 'V', 'W', 'X', 'Y']
]

start_char = 'A' # Начинаем с символа 'A'

result = longest_alphabetical_path(matrix, start_char)
print(f"Длина самого длинного пути: {result}")

```


Задание 4. Генерирование возможных слов из матрицы символов

Результат работы программы:

```
D:\Gitlab\AI\AI-in-prof3\.venv\Scripts\python.exe D:\Gitlab\AI\AI-in-prof3\4.py
Найденные осмысленные слова:
['ЛЕС', 'ЛЬ', 'СЕН', 'КАТ', 'ЛО']

Process finished with exit code 0
```

Рисунок 4 – Результат работы программы

Код программы:

```
def load_russian_dictionary():
    return set([
        "КАТЕ", "ЛО", "ЛЬ", "СЕН", "СЕНО", "КАТ",
        "ДОМ", "СТОЛ", "КНИГА", "ГОРКА", "ЛЕС"
    ])

# Функция поиска осмысленных слов
def find_meaningful_words(matrix, dictionary):
    # Получаем размер матрицы
    rows, cols = len(matrix), len(matrix[0])

    # Проверяем, находится ли координата внутри матрицы
    def is_valid(x, y, visited):
        return 0 <= x < rows and 0 <= y < cols and not visited[x][y]

    # Рекурсивная функция для поиска слов
    def dfs(x, y, path, visited, words):
        # Добавляем текущую ячейку в путь
        path.append(matrix[x][y])
        current_word = ".".join(path)

        # Если текущее слово есть в словаре, добавляем его в результат
        if current_word in dictionary:
            words.add(current_word)
```

```

# Проверяем все восемь направлений
directions = [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1, 1), (1, -1), (1, 1)]
for dx, dy in directions:
    nx, ny = x + dx, y + dy
    if is_valid(nx, ny, visited): # Если новая координата допустима
        visited[nx][ny] = True
        dfs(nx, ny, path, visited, words)
        visited[nx][ny] = False # Снимаем метку посещения

# Убираем текущую ячейку из пути (backtracking)
path.pop()

# Множество для хранения всех найденных слов
all_words = set()

# Перебираем каждую ячейку как начальную точку
for i in range(rows):
    for j in range(cols):
        visited = [[False for _ in range(cols)] for _ in range(rows)] # Матрица
        посещенных ячеек
        visited[i][j] = True
        dfs(i, j, [], visited, all_words)

return list(all_words)

# Пример использования
matrix = [
    ['K', 'A', 'T'],
    ['O', 'L', 'B'],
    ['C', 'E', 'H']
]

# Загружаем словарь русских слов (можно заменить на реальный файл)

```

```
dictionary = load_russian_dictionary()
```

```
result = find_meaningful_words(matrix, dictionary)
```

```
print("Найденные осмысленные слова:")
```

```
print(result)
```

Ответы на контрольные вопросы:

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

Алгоритм поиска в глубину использует стек вместо очереди.

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

Полнота, временная сложность, пространственная сложность, глубина.

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

В начало стека добавляются дочерние узлы.

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

Это необходимо для обхода ветвей в глубину.

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

В любой момент времени алгоритму необходимо хранить информацию только о том пути, по которому он идет, и о тех узлах, которые еще предстоит исследовать на этом пути.

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

После достижения максимальной глубины остаются узлы текущего пути (стек) до максимальной глубины. Все остальные узлы, находящиеся на других ветвях, не сохраняются в памяти.

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

Поиск в глубину может застрять в следующих случаях:

Циклы: Если граф содержит циклы и не реализовано отслеживание посещенных узлов.

Бесконечные ветви: Если есть бесконечные ветви, которые не приводят к решению.

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

Временная сложность поиска в глубину составляет $O(b^d)$, где b — степень ветвления, а d — максимальная глубина дерева. Это означает, что с увеличением глубины количество узлов, которые нужно исследовать, экспоненциально возрастает.

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

Поиск в глубину не гарантирует нахождение оптимального решения, потому что он может найти решение, которое является локальным минимумом, не исследуя все возможные пути. Он не учитывает стоимость путей и может остановиться на более длинном или менее эффективном решении.

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

Поиск в глубину предпочтителен в следующих ситуациях:

Небольшие пространства поиска: Когда пространство поиска невелико и можно легко проверить все узлы.

Проблемы с ограничениями по памяти: Когда память ограничена, так как поиск в глубину требует меньше памяти по сравнению с другими алгоритмами.

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

Данная функция выполняет рекурсивный поиск в глубину. Принимаемые параметры: `problem`, `node`.

12. Какую задачу решает проверка `if node is None`?

Данная проверка выявляет отсутствие начального узла.

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

В случаях, когда узел успешно найден.

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

Чтобы не возник бесконечный цикл.

15. Что возвращает функция при обнаружении цикла?

При обнаружении цикла будет возвращено значение failure.

16. Как функция обрабатывает дочерние узлы текущего узла?

Данная функция принимает дочерние узлы в качестве параметра.

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

В данной реализации используется рекурсия.

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

Будет возвращено значение failure.

19. Почему функция рекурсивно вызывает саму себя внутри цикла?

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

Данная функция получает дочерние узлы текущего узла.

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

Данная функция проверяет наличие циклов.

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

Эта проверка позволяет вернуть решение в случае его наличия.

23. В каких ситуациях алгоритм может вернуть failure ?

В случаях, когда целевой элемент не найден или обнаружен цикл.

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

При итеративном поиске в глубину используется стек.

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

Может возникнуть бесконечный цикл.

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