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
По дисциплине
«Искусственный интеллект в профессиональной сфере»

Выполнил:
Арзютов Иван Владиславович

3 курс, группа ИТС-б-о-22-1,

11.03.02 «Инфокоммуникационные технологии и системы связи, очная форма обучения

_____
(подпись)

Проверил:

Воронкин Р.А., канд. тех. наук, доцент, доцент кафедры инфокоммуникаций

_____
(подпись)

Отчет защищен с оценкой _____ Дата защиты_____

Ставрополь, 2024 г.

Исследование поиска в глубину

Ссылка на репозиторий: https://github.com/Ivanuschka/3-LB

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

Порядок выполнения работы

Задание 1

Создал общедоступный репозиторий на GitHub, в котором использована лицензия MIT.

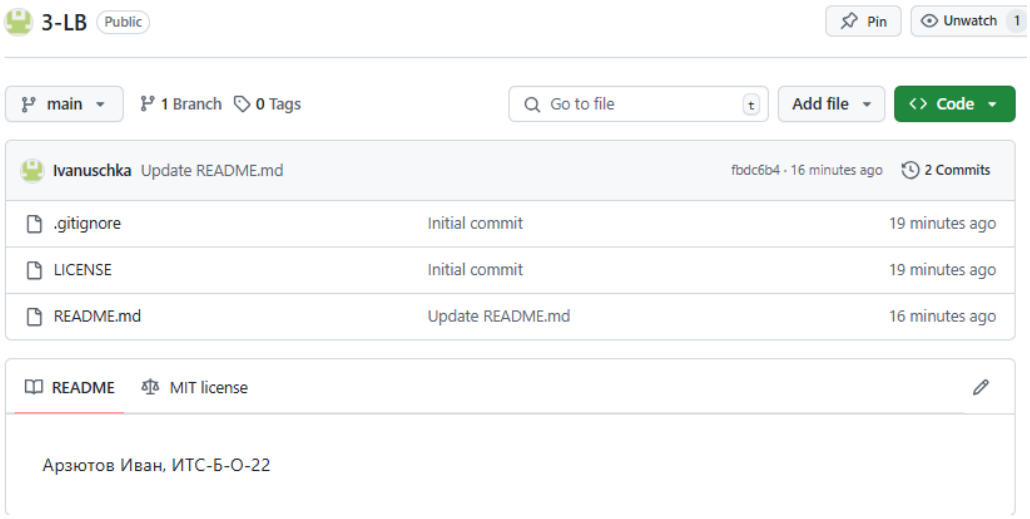

Рисунок 1. Создан новый репозиторий

Задание 2

Клонирование репозитория на свой компьютер.

```
C:\Users\ivana\OneDrive\Рабочий стол\ИИ\1 Лабораторная работа>git clone https://github.com/Ivanuschka/3-LB.git
Cloning into '3-LB'...
remote: Enumerating objects: 8, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 8 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (8/8), done.
Resolving deltas: 100% (1/1), done.

C:\Users\ivana\OneDrive\Рабочий стол\ИИ\1 Лабораторная работа>cd C:\Users\ivana\OneDrive\Рабочий стол\ИИ\1 Лабораторная
работа\3-LB

C:\Users\ivana\OneDrive\Рабочий стол\ИИ\1 Лабораторная работа\3-LB>
```

Рисунок 2. Клонирование репозитория

Задание 3

Проработка примера лабораторной работы

```
f depth_first_recursive_search(problem, node=None):
    if node is None:
        node = Node(problem.initial)
    if problem.is_goal(node.state):
        return node
    elif is_cycle(node):
        return failure
    else:
        for child in expand(problem, node):
            result = depth_first_recursive_search(problem, child)
            if result:
                return result
        return failure
```

Рисунок 3. Выполнение примера

Задание 4

Создание модулей для примера

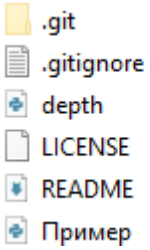

Рисунок 4. Модули в папке репозитория

Задание 5

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

```python
import numpy as np
import matplotlib.pyplot as plt

def is_valid_move(matrix, x, y, prev_char):
    return 0 <= x < len(matrix) and 0 <= y < len(matrix[0]) and ord(matrix[x][y]) == ord(prev_char) + 1

def longest_path(matrix, x, y, memo):
    if memo[x][y] is not None:
        return memo[x][y]

    directions = [(-1, -1), (-1, 0), (-1, 1),
                  (0, -1),         (0, 1),
                  (1, -1), (1, 0), (1, 1)]
    max_length = 1

    for dx, dy in directions:
        new_x, new_y = x + dx, y + dy
        if is_valid_move(matrix, new_x, new_y, matrix[x][y]):
            max_length = max(max_length, 1 + longest_path(matrix, new_x, new_y, memo))

    memo[x][y] = max_length
    return max_length

def find_longest_path(matrix, start_char):
    rows, cols = len(matrix), len(matrix[0])
    memo = [[None] * cols for _ in range(rows)]
    max_path = 0

    for i in range(rows):
        for j in range(cols):
            if matrix[i][j] == start_char:
                max_path = max(max_path, longest_path(matrix, i, j, memo))

    return max_path

# Создание случайной матрицы символов
np.random.seed(0)
letters = np.array([chr(i) for i in range(ord('A'), ord('Z') + 1)])
matrix = np.random.choice(letters, (5, 5))
print("Матрица символов:")
print(matrix)

start_char = 'A'
longest = find_longest_path(matrix, start_char)
print(f"Самый длинный путь в матрице, начиная с '{start_char}': {longest}")

# Визуализация матрицы
plt.figure(figsize=(6, 6))
plt.imshow(np.ones_like(matrix, dtype=int), cmap="coolwarm", interpolation="nearest")
for i in range(len(matrix)):
    for j in range(len(matrix[0])):
        plt.text(j, i, matrix[i, j], ha='center', va='center', color='black')
plt.title("Матрица символов")
plt.show()
```

Рисунок 4. Код для определения самого длинного пути

```
Матрица символов:
[['M' 'P' 'V' 'A' 'D']
 ['D' 'H' 'J' 'T' 'V']
 ['S' 'E' 'X' 'G' 'Y']
 ['Y' 'M' 'B' 'G' 'H']
 ['X' 'O' 'Y' 'R' 'F']]
Самый длинный путь в матрице, начиная с 'A': 1
```

Рисунок 5. Результат выполнения программы

Задание 6.

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

```python
import numpy as np
import matplotlib.pyplot as plt

def is_valid_move(matrix, x, y, visited):
    return 0 <= x < len(matrix) and 0 <= y < len(matrix[0]) and not visited[x][y]

def find_words(matrix, x, y, visited, current_word, word_list, dictionary):
    visited[x][y] = True
    current_word += matrix[x][y]

    if current_word in dictionary:
        word_list.add(current_word)

    directions = [(-1, -1), (-1, 0), (-1, 1),
                  (0, -1),         (0, 1),
                  (1, -1), (1, 0), (1, 1)]

    for dx, dy in directions:
        new_x, new_y = x + dx, y + dy
        if is_valid_move(matrix, new_x, new_y, visited):
            find_words(matrix, new_x, new_y, visited, current_word, word_list, dictionary)

    visited[x][y] = False

def find_all_words(matrix, dictionary):
    rows, cols = len(matrix), len(matrix[0])
    word_list = set()
    visited = [[False] * cols for _ in range(rows)]

    for i in range(rows):
        for j in range(cols):
            find_words(matrix, i, j, visited, "", word_list, dictionary)

    return word_list

# Создание матрицы, содержащей точно 10 слов из словаря
dictionary = {"CAT", "DOG", "BAT", "RAT", "CAR", "BAR", "ART", "TOOL", "TREE", "BIRD"}
matrix = np.array([
    ['C', 'A', 'T', 'O', 'X'],
    ['D', 'O', 'G', 'B', 'A'],
    ['B', 'A', 'T', 'R', 'A'],
    ['T', 'R', 'E', 'E', 'L'],
    ['B', 'I', 'R', 'D', 'Y']
])
print("Матрица символов:")
print(matrix)

found_words = find_all_words(matrix, dictionary)
print("Найденные слова:", found_words)

# Визуализация матрицы
plt.figure(figsize=(6, 6))
plt.imshow(np.ones_like(matrix, dtype=int), cmap="coolwarm", interpolation="nearest")
for i in range(len(matrix)):
    for j in range(len(matrix[0])):
        plt.text(j, i, matrix[i, j], ha='center', va='center', color='black')
plt.title("Матрица символов")
plt.show()
```

Рисунок 6. Код для слов в матрице

Задание 6.

Для построенного графа лабораторной работы 1 написать программу на языке программирования Python, которая с помощью алгоритма поиска в глубину находит минимальное расстояние между начальным и конечным пунктами.

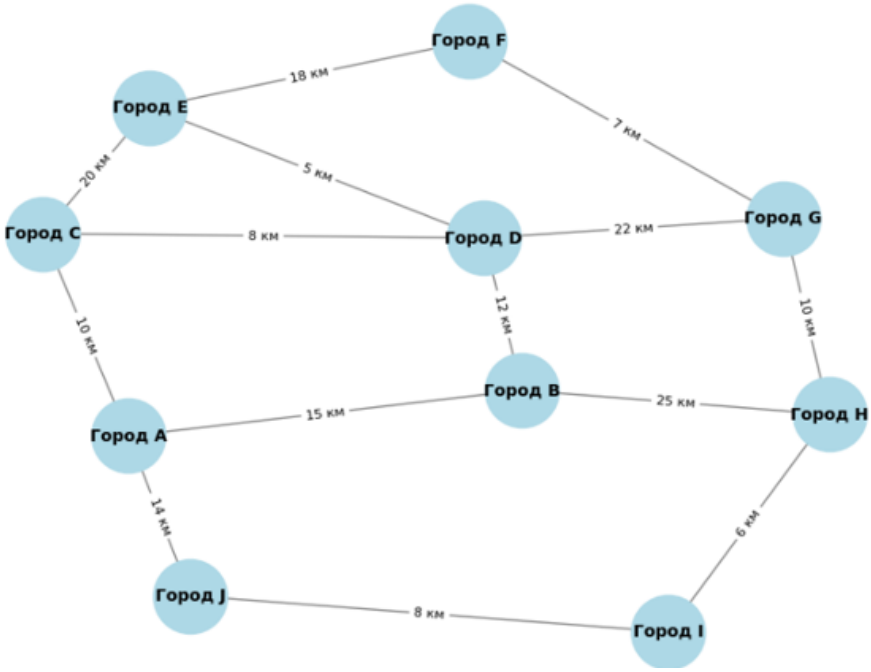

Рисунок 7. Граф городов

```python
import numpy as np
import matplotlib.pyplot as plt

def dfs(graph, current, target, visited, path, min_path, min_distance, current_distance):
    visited.add(current)
    path.append(current)

    if current == target:
        if current_distance < min_distance[0]:
            min_distance[0] = current_distance
            min_path.clear()
            min_path.extend(path)
    else:
        for neighbor, distance in graph[current]:
            if neighbor not in visited:
                dfs(graph, neighbor, target, visited, path, min_path, min_distance, current_distance + distance)

    path.pop()
    visited.remove(current)

def find_shortest_path_dfs(graph, start, end):
    visited = set()
    min_path = []
    min_distance = [float('inf')]
    dfs(graph, start, end, visited, [], min_path, min_distance, 0)
    return min_path, min_distance[0]

graph = {
    'Город A': [('Город B', 15), ('Город J', 14)],
    'Город B': [('Город A', 15), ('Город D', 12), ('Город H', 25)],
    'Город C': [('Город A', 10), ('Город D', 8), ('Город E', 20)],
    'Город D': [('Город C', 8), ('Город B', 12), ('Город E', 5), ('Город F', 5), ('Город G', 22)],
    'Город B': [('Город A', 15), ('Город D', 12), ('Город H', 25)],
    'Город C': [('Город A', 10), ('Город D', 8), ('Город E', 20)],
    'Город D': [('Город C', 8), ('Город B', 12), ('Город E', 5), ('Город F', 5), ('Город G', 22)],
    'Город E': [('Город C', 20), ('Город D', 5), ('Город F', 18)],
    'Город F': [('Город E', 18), ('Город D', 5), ('Город G', 7)],
    'Город G': [('Город D', 22), ('Город F', 7), ('Город H', 10)],
    'Город H': [('Город G', 10), ('Город B', 25), ('Город I', 6)],
    'Город I': [('Город H', 6), ('Город J', 8)],
    'Город J': [('Город I', 8), ('Город A', 14)]
}

start_city = 'Город E'
end_city = 'Город I'
shortest_path, shortest_distance = find_shortest_path_dfs(graph, start_city, end_city)

print(f'Кратчайший путь из {start_city} в {end_city}: {" -> ".join(shortest_path)}')
print(f'Общая длина пути: {shortest_distance} км')
```

Рисунок 8. Код программы для поиска в ширину

```
Кратчайший путь из Город E в Город I: Город E -> Город D -> Город F -> Город G -> Город H -> Город I
Общая длина пути: 33 км
```

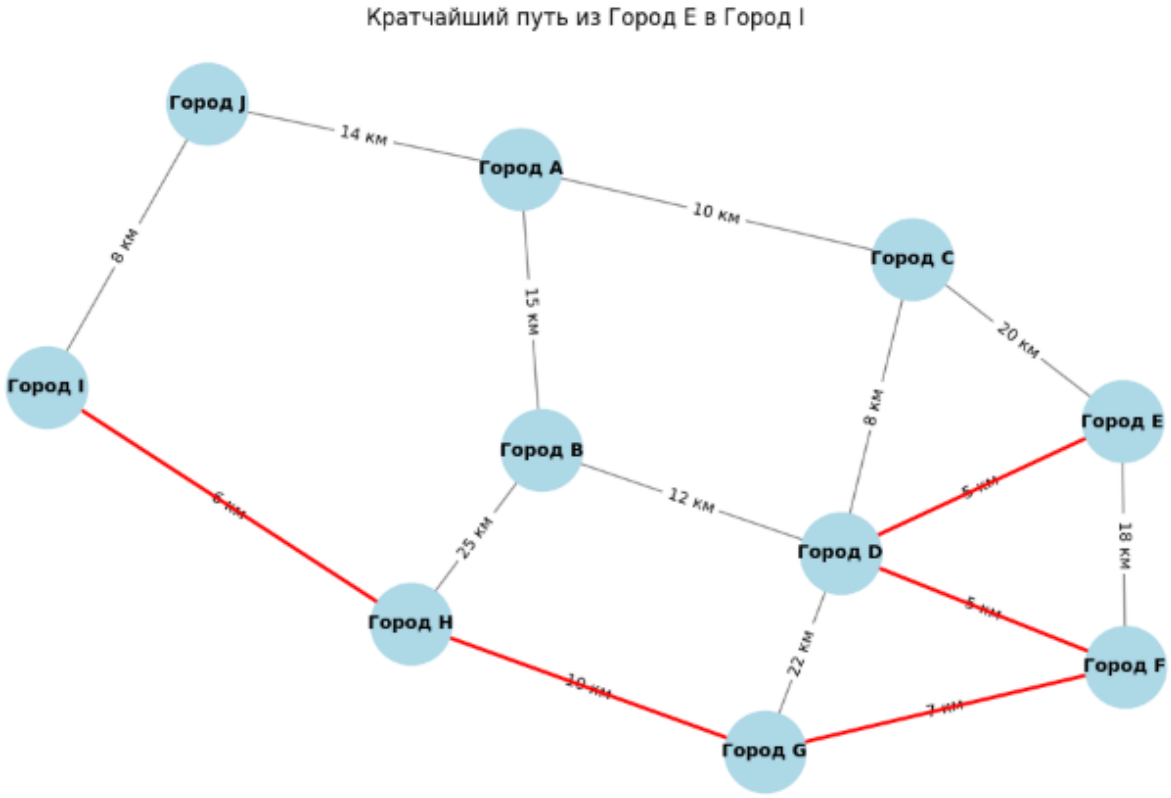

Рисунок 9. Визуализация кратчайшего пути при поиске в глубину

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

Ответы на контрольные вопросы

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

Ключевое отличие заключается в порядке обхода узлов. Поиск в глубину (DFS) сначала исследует как можно глубже каждую ветвь, прежде чем вернуться назад, тогда как поиск в ширину (BFS) исследует все соседние узлы текущего уровня перед тем, как перейти на следующий уровень.

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

Критерии качества поиска могут включать: корректность, полноту, оптимальность и эффективность (временная и пространственная сложность).

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

При расширении узла в DFS исследуются все его дочерние узлы, при этом они добавляются в стек для дальнейшего обследования.

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

LIFO-структура стека позволяет DFS сначала вернуться к последнему исследованному узлу, что соответствует подходу исследования глубины.

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

DFS может освобождать память, так как не хранит все узлы на одном уровне одновременно; вместо этого он хранит только текущий путь. Это значительно снижает потребление памяти по сравнению с BFS.

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

Узлы, находящиеся на текущем пути к листьям, остаются в памяти, однако все узлы выше по этому пути могут быть удалены.

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

DFS может застрять в бесконечных циклах или если дерево имеет очень глубокие ветви, не приводящие к решению.

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

Временная сложность DFS составляет O(b^d), где b — число ветвлений, а d — максимальная глубина. При увеличении глубины d время выполнения может экспоненциально увеличиваться.

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

Поскольку DFS не рассматривает все возможные пути до решения и возвращается к предыдущим узлам, он может не найти кратчайший путь к цели.

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

DFS предпочтителен при необходимости поиска в ограниченных пространствах, при серьезных ограничениях по памяти или когда требуется найти одно из возможных решений, а не оптимальное.

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

Эта функция выполняет рекурсивный поиск в глубину, принимая текущий узел и, возможно, другие параметры, такие как цель поиска или используемая структура данных для посещенных узлов.

12. Какую задачу решает проверка if node is None?

Эта проверка используется для определения, достигнут ли конец дерева или если текущий узел не существует. Это важно для предотвращения ошибок обращения к несуществующим узлам.

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

Функция возвращает узел как решение, когда найден узел, соответствующий условию цели.

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

Проверка на циклы предотвращает зацикливание алгоритма, которое может привести к бесконечной рекурсии и истощению памяти.

15. Что возвращает функция при обнаружении цикла?

Обычно такая функция возвращает индикатор или значение, указывающее на наличие цикла, чтобы прекратить дальнейшие действия.

16. Как функция обрабатывает дочерние узлы текущего узла?

Функция рекурсивно вызывает саму себя для каждого дочернего узла, передавая ему необходимые параметры.

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

Используется рекурсивный механизм, который вызывает функцию для

текущего узла и движется вниз по иерархии дерева.

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

В этом случае функция может вернуться к предыдущему узлу и продолжить поиск в других ветвях или вернуть значение, указывающее на отсутствие решения.

19. Почему функция рекурсивно вызывает саму себя внутри цикла?

Это делается для исследования каждого дочернего узла и дальнейшего населения стека вызовов по мере движения вниз по дереву.

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

Функция `expand` генерирует и возвращает возможные дочерние узлы (или состояния) текущего узла, основываясь на заданной проблеме.

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

Функция `is_cycle` проверяет, является ли текущий узел частью зацикленной последовательности, предотвращая зацикливание алгоритма.

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

Если `result` возвращает решение, это позволяет алгоритму сразу завершить выполнение и вернуть результат, избегая ненужных вычислений.

23. В каких ситуациях алгоритм может вернуть failure?

Алгоритм возвращает `failure`, если не нашел ни одного решения после проверки всех возможных узлов.

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

Рекурсивная реализация использует стек вызовов для хранения состояния, тогда как итеративная реализация явно управляет стеком в коде.

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

Основная проблема заключается в возможности бесконечной рекурсии и переполнения стека, что приводит к краху программы, поскольку алгоритм будет продолжать обход без нахождения решения.