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:
Быковская Стефания Станиславовна
3 курс, группа ИТС-б-о-22-1,
11.03.02 «Инфокоммуникационные технологии и системы связи»

_____
(подпись)

Проверил: доцент департамента цифровых, робототехнических систем и электроники института перспективной инженерии
Воронкин Роман Александрович

_____
(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:

Решите задания лабораторной работы с помощью языка программирования Python и элементов программного кода лабораторной работы 1. Проверьте правильность решения каждой задачи на приведенных тестовых примерах.

Алгоритм заливки:

```python
class Problem:
    def __init__(self, matrix):
        self.matrix = matrix
        self.rows = len(matrix)
        self.cols = len(matrix[0]) if self.rows > 0 else 0

    def actions(self, state):
        # Возвращает возможные действия (соседние клетки)
        row, col = state
        directions = [
            (-1, 0),  # вверх
            (1, 0),   # вниз
            (0, -1),  # влево
            (0, 1)    # вправо
        ]
        for dr, dc in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < self.rows and 0 <= new_col < self.cols:
                yield (new_row, new_col)
```

Рисунок 1. Класс Problem

```python
def flood_fill(problem, start, target_color, replacement_color):
    """Выполняет заливку области, начиная с заданного узла."""
    if problem.matrix[start[0]][start[1]] != target_color:
        return

    queue = [(start[0], start[1])]
    problem.matrix[start[0]][start[1]] = replacement_color

    while queue:
        current_row, current_col = queue.pop(0)

        for action in problem.actions((current_row, current_col)):
            new_row, new_col = action
            if problem.matrix[new_row][new_col] == target_color:
                problem.matrix[new_row][new_col] = replacement_color
                queue.append((new_row, new_col))
```

Рисунок 2. Алгоритм заливки

Поиск самого длинного пути в матрице:

```python
class Problem:
    def __init__(self, matrix):
        self.matrix = matrix
        self.rows = len(matrix)
        self.cols = len(matrix[0]) if self.rows > 0 else 0

    def actions(self, state):
        """Возвращает соседние клетки по всем восьми направлениям."""
        row, col = state
        directions = [
            (-1, -1), (-1, 0), (-1, 1),   # вверх влево, вверх, вверх вправо
            (0, -1),          (0, 1),     # влево, вправо
            (1, -1), (1, 0), (1, 1)       # вниз влево, вниз, вниз вправо
        ]
        for dr, dc in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < self.rows and 0 <= new_col < self.cols:
                yield (new_row, new_col)
```

Рисунок 3. Класс Problem

```python
def find_longest_alphabetic_path(problem, start_char):
    """Находит длину самого длинного алфавитного пути в матрице, начиная с заданного символа."""
    # Ищем первую позицию символа start_char в матрице
    start_positions = [(i, j) for i in range(problem.rows) for j in range(problem.cols) if problem.matrix[i][j] == start_char]
    if not start_positions:
        return 0  # Если символ не найден, возвращаем 0

    longest_path = [0]  # Используем список, чтобы передать по ссылке
    visited = set()

    # Запускаем поиск для каждого начального символа 'start_char'
    for start_row, start_col in start_positions:
        _dfs(problem, start_row, start_col, start_char, 1, visited, longest_path)

    return longest_path[0]
```

Рисунок 4. Поиск самого длинного пути в матрице

Генерирование списка возможных слов из матрицы символов:

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

class Problem:
    def __init__(self, matrix):
        self.matrix = matrix
        self.rows = len(matrix)
        self.cols = len(matrix[0]) if self.rows > 0 else 0

    def actions(self, state):
        """Возвращает соседние клетки по всем восьми направлениям."""
        row, col = state
        directions = [
            (-1, -1), (-1, 0), (-1, 1),   # вверх влево, вверх, вверх вправо
            (0, -1),          (0, 1),     # влево, вправо
            (1, -1), (1, 0), (1, 1)       # вниз влево, вниз, вниз вправо
        ]
        for dr, dc in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < self.rows and 0 <= new_col < self.cols:
                yield (new_row, new_col)
```

Рисунок 5. Класс Problem

```python
def find_words(matrix, word_list):
    found_words = set()
    problem = Problem(matrix)

    for word in word_list:
        visited = [[False] * problem.cols for _ in range(problem.rows)]
        for i in range(problem.rows):
            for j in range(problem.cols):
                if dfs(matrix, (i, j), visited, word, 0):
                    found_words.add(word)
                    break  # Если слово найдено, выходим из цикла
            if word in found_words:
                break  # Если слово найдено, выходим из внешнего цикла
    return found_words
```

Рисунок 6. Поиск слов в матрице

Для задачи "Поиск самого длинного пути в матрице" подготовить собственную матрицу, для которой с помощью разработанной в предыдущем пункте программы, подсчитать самый длинный путь.

```python
if __name__ == '__main__':
    # Входная матрица
    matrix = [
        ["X", "E", "H", "A", "B"],
        ["A", "A", "G", "C", "E"],
        ["D", "F", "T", "F", "D"],
        ["E", "B", "E", "G", "H"],
        ["C", "M", "Y", "K", "L"],
    ]
    start_char = 'A'

    problem = Problem(matrix)
    longest_path_length = find_longest_alphabetic_path(problem, start_char)
    print("Длина самого длинного пути:", longest_path_length)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

C:\Users\Лео\Desktop\ИИ\3>python .\prog\2.py
Длина самого длинного пути: 8
```

Рисунок 7. Собственная матрица и результат

Для задачи "Генерирование списка возможных слов из матрицы символов" подготовить собственную матрицу для генерирования списка возможных слов с помощью разработанной программы.

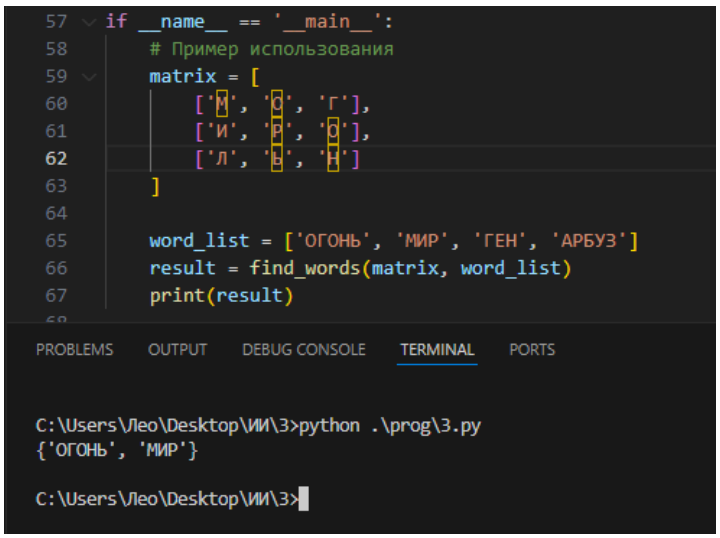

Рисунок 8. Собственная матрица и результат

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 не отличается от поиска в глубину

Контрольные вопросы:

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

Отличие поиска в глубину от поиска в ширину заключается в порядке расширения узлов, хотя оба метода представляют собой вариации поиска по дереву, этот процесс реализуется с использованием очереди типа "последним пришел - первым ушел" (LIFO).

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

Полнота: Этот критерий отражает способность алгоритма находить решение, если оно существует.

Временная сложность: Она измеряется количеством узлов, которые алгоритм должен сгенерировать.

Пространственная сложность поиска в глубину относится к количеству памяти, необходимой для хранения информации о структуре дерева и состоянии поиска.

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

В отличие от поиска в ширину, который требует хранения информации обо всех узлах на текущем уровне и всех их потомках, поиск в глубину может ограничиваться меньшим количеством узлов. Это количество узлов равно максимальной глубине дерева (m), умноженной на коэффициент ветвления (b), но в реальности часто требуется хранить гораздо меньше информации, так как узлы удаляются из памяти, как только они полностью исследованы.

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

Поиск в глубину (DFS) использует стек (структуру LIFO), так как он исследует ветви графа или дерева до самого конца, прежде чем возвращаться обратно к предшествующим узлам. LIFO обеспечивает возвращение к последнему посещенному узлу, что позволяет продолжить поиск вглубь и затем откатиться к предыдущим узлам по мере необходимости.

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

Поиск в глубину хранит только узлы текущей ветви, удаляя узлы, как только возвращается к предшествующим уровням, что экономит память. Это важно, так как в глубоком дереве или графе DFS хранит намного меньше узлов в памяти, чем поиск в ширину (BFS), где хранятся узлы всех уровней текущей глубины.

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

В памяти остаются узлы текущего пути от корня до узла на максимальной глубине, так как поиск в глубину возвращается к предшествующим узлам только после полного завершения текущей ветви.

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

DFS может "застрять" в случаях, когда: Граф или дерево бесконечно или содержит циклы (алгоритм уходит в бесконечный цикл). Решение находится на более высоком уровне, чем текущая глубина поиска. Не реализована проверка на уже посещенные узлы для предотвращения циклов.

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

Временная сложность DFS пропорциональна максимальной глубине дерева d и количеству дочерних узлов b в каждом узле, что выражается как $O(b^d)$.

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

DFS не гарантирует нахождение кратчайшего пути, так как он выбирает первый найденный узел, удовлетворяющий условиям, а не узел с минимальным числом переходов или наименьшим "весом". Поэтому решение может быть неоптимальным по расстоянию или стоимости.

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

DFS предпочтителен, если:

Глубина решения известна или ограничена. Требуется минимальное использование памяти. Нужно быстро получить хотя бы одно решение, а оптимальность не важна. Дерево/граф не слишком глубокий или не содержит циклов.

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

Функция depth_first_recursive_search рекурсивно ищет решение в графе или дереве методом поиска в глубину. Она обычно принимает:

Узел (node), который анализируется. Проблему (problem), которая описывает, как определяются состояния, решения и расширения узлов.

12. Какую задачу решает проверка if node is None?

Эта проверка служит для обработки случаев, когда узел отсутствует или путь к узлу прерывается. Например, если расширение узла не приводит к новым состояниям, функция может вернуть None, показывая, что нет решения.

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

Функция возвращает узел как решение, если узел удовлетворяет целевому состоянию задачи (например, проверка с goal_test(node)).

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

Без проверки на циклы алгоритм может зациклиться, что приведет к бесконечной рекурсии и переполнению стека вызовов, особенно если дерево бесконечно или граф содержит циклы.

15. Что возвращает функция при обнаружении цикла?

При обнаружении цикла функция обычно завершает текущий путь и возвращает значение, указывающее, что решение не найдено по этому пути, чтобы избежать бесконечного зацикливания.

16. Как функция обрабатывает дочерние узлы текущего узла?

Функция рекурсивно вызывает себя для каждого дочернего узла, таким образом, углубляясь в каждый путь. Это позволяет исследовать текущий путь до конца перед переходом к другим ветвям.

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

Используется рекурсивный вызов функции для углубления в ветвь, сохраняя текущий путь в стеке вызовов.

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

Если решение не будет найдено, функция вернет специальное значение, указывающее на неуспех (failure), или None, если это предусмотрено.

19. Почему функция рекурсивно вызывает саму себя внутри цикла?

Рекурсивный вызов функции в цикле по дочерним узлам позволяет исследовать все возможные пути от текущего узла вглубь, пока не достигнется решение или не будет пройден весь граф.

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

Функция expand создает или возвращает дочерние узлы текущего узла в соответствии с правилами, заданными в problem, что позволяет двигаться глубже по каждому пути.

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

Функция is_cycle(node) проверяет, был ли узел уже посещен на текущем пути, предотвращая зацикливание.

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

Эта проверка помогает определить, достиг ли алгоритм успешного решения по текущему пути. Если result содержит узел-решение, то он возвращается вверх по стеку, завершая дальнейшие вызовы.

23. В каких ситуациях алгоритм может вернуть failure?

Алгоритм возвращает failure, если он не находит решения ни по одному пути, исследовав все возможные варианты.

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

Рекурсивный DFS сохраняет состояние пути в стеке вызовов, тогда как итеративный DFS использует явный стек. Рекурсивный метод проще в реализации, но может привести к переполнению стека, особенно в глубоком или бесконечном дереве.

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

При бесконечных деревьях или графах рекурсивный DFS может уйти в бесконечный цикл или переполнить стек вызовов

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