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

Искусственный интеллект в профессиональной сфере

Отчет по лабораторной работе №3

Исследование поиска в глубину

Выполнил:
Тихоненко Борис Витальевич
3 курс, группа ИТС-б-з-22-1,
11.03.02
«Инфокоммуникационные технологии и системы связи»,
заочная форма обучения

_____
(подпись)

Проверил: доцент департамента цифровых робототехнических систем и электроники
Воронкин Роман Александрович

_____
(подпись)

Отчет защищен с оценкой _____ Дата защиты_____

Ставрополь, 2025 г.

Тема: исследование поиска в глубину.

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

Ход работы:

Задание 1. Создал проект в папке репозитория. Приступил к работе с примером. Добавил новый файл lab3-prim.py.

```
def expand(problem, node):
    # Логика расширения узлов
    pass


failure = None  # Определите значение failure, если это нужно


def 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 1. Выполнение lab3-prim.py

Задание 2. Необходимо для задачи "Поиск самого длинного пути в матрице" подготовить собственную матрицу, для которой с помощью разработанной в предыдущем пункте программы, подсчитать самый длинный путь. Разработаем матрицу:

```
new_matrix = [
    ["A", "B", "Y", "U", "I", "F", "T"],
    ["Z", "C", "D", "E", "F", "G", "H"],
    ["I", "J", "K", "L", "M", "I", "S"],
    ["R", "Q", "P", "O", "N", "K", "T"],
    ["S", "T", "U", "V", "W", "L", "Y"],
    ["Z", "A", "B", "C", "D", "M", "F"],
    ["G", "H", "I", "J", "K", "P", "M"],
]
```

Рисунок 2. Создание матрицы.

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. Разработать функцию поиска самого длинного пути в матрицесимволов, начиная с заданного символа. Символы в пути должны следовать в алфавитном порядке и быть последовательными. Поиск возможен во всех восьми направлениях. Напишем программу:

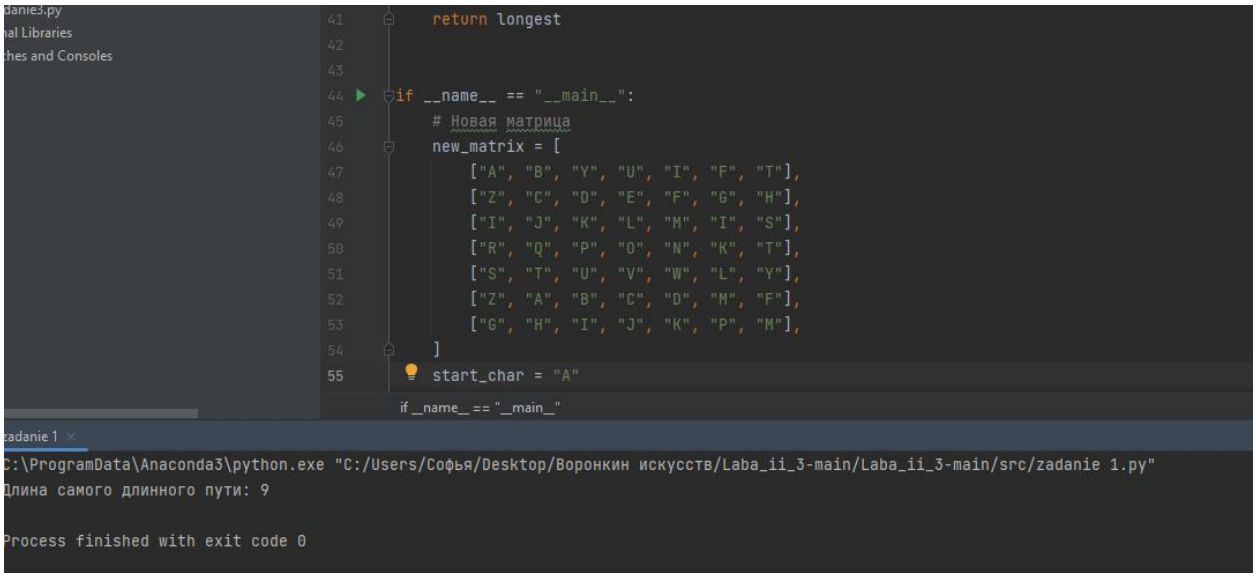


Рисунок 3. Разработанная программа

Вывод программы показал результат 3, что соответствует выбранной нами матрице с островами.

Задание 4. Необходимо для задачи "Генерирование списка возможных слов из матрицы символов" подготовить собственную матрицу для генерирования списка возможных слов с помощью разработанной программы.

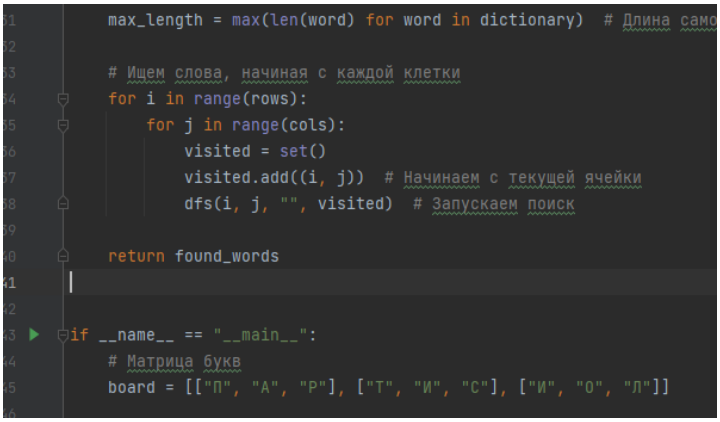


Рисунок 4. Матрица для второго задания

Наша 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, юговосток, юго-запад), и каждая клетка может быть использована в слове только один раз.

Напишем программу:

```
def find_words(board, dictionary):
    found_words = set()
    rows, cols = len(board), len(board[0])

    # Направления для перемещения (восемь направлений)
    directions = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]

    # Рекурсивная функция для поиска слов
    def dfs(x, y, path, visited):
        path += board[x][y]  # Добавляем текущий символ к пути
        if path in dictionary:
            found_words.add(path)  # Если слово найдено, добавляем его в набор

        if len(path) > max_length:
            return

        # Проверка всех направлений
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < rows and 0 <= new_y < cols and (new_x, new_y) not in visited:
                visited.add((new_x, new_y))  # Добавляем в посещенные
                dfs(new_x, new_y, path, visited)
                visited.remove((new_x, new_y))  # Убираем из посещенных

    max_length = max(len(word) for word in dictionary)  # Длина самого длинного слова

    # Ищем слова, начиная с каждой клетки
```

Рисунок 5. Выполнение написанной программы

Задание 6. Необходимо 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. Сравним найденное решение с решением, полученным вручную. Найдем минимальное расстояние между городами Калязин и Кострома.

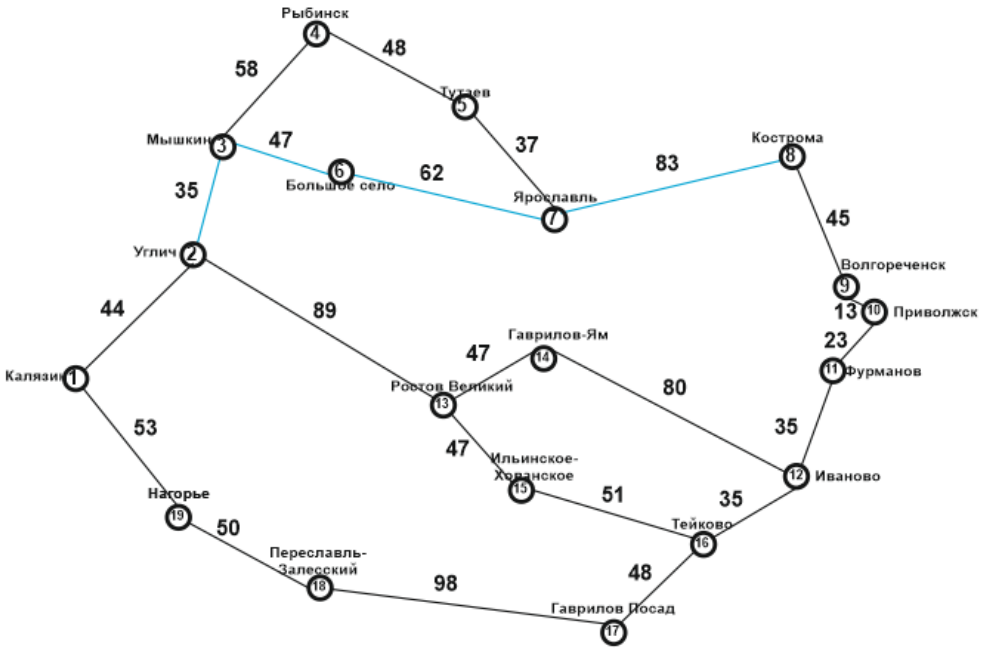
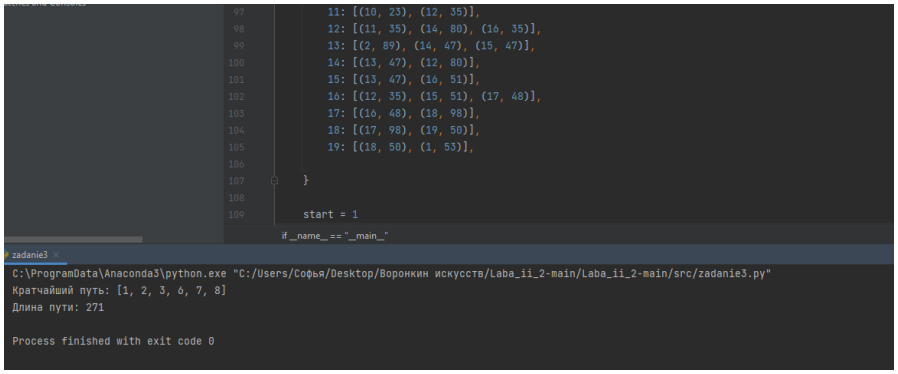


Рисунок 5. Граф из лабораторной работы №1.

Если считать вручную, то минимальное расстояние составляет 271 км. Далее составим программу и проверим:



Результат программы вывел так же 271 км.

Ответы на контрольные вопросы:

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

Ключевое отличие состоит в стратегии обхода. Поиск в глубину (DFS) исследует одну ветвь дерева/графа до самого глубокого уровня, прежде чем перейти к следующей ветви. Поиск в ширину (BFS) исследует все узлы на одном уровне глубины перед переходом к следующему уровню.

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

• Полнота — гарантирует ли алгоритм нахождение решения, если оно существует.

• Оптимальность — находит ли алгоритм лучшее (минимальное по стоимости) решение.

• Временная сложность — сколько времени требуется для нахождения решения.

• Пространственная сложность — сколько памяти использует алгоритм.

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

При расширении узла генерируются его дочерние узлы. Это включает создание новых узлов на основе соседей текущего узла, которые затем добавляются в стек или передаются в рекурсию.

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

Очередь LIFO (стек) обеспечивает приоритетное исследование недавно добавленных узлов, что позволяет алгоритму "углубляться" в ветви графа или дерева.

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

Поиск в глубину удаляет узлы из памяти, как только они обработаны и нет необходимости возвращаться к ним. Это снижает объем памяти по сравнению с поиском в ширину, который должен хранить все узлы на текущем уровне.

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

В памяти остаются узлы текущего пути от корня до текущего узла и узлы, которые еще не были исследованы.

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

• Если граф или дерево бесконечно глубокие.

• Если существует цикл, и алгоритм не имеет проверки на циклы.

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

Временная сложность DFS составляет $O(b_m)$, где b — фактор ветвления, а m — максимальная глубина дерева. Она растет экспоненциально с глубиной дерева.

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

DFS не рассматривает все пути одновременно, поэтому может найти не самый короткий путь, если решение обнаружено до исследования более выгодного варианта.

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

• Когда пространство поиска ограничено и важно минимизировать потребление памяти.

• Если известна приблизительная глубина решения.

• Когда нужно найти любое решение быстро, а не обязательно оптимальное.

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

Функция выполняет рекурсивный поиск в глубину для нахождения решения. Она принимает:

• problem — задачу, содержащую начальный узел и цель.

• graph — граф для обхода.

• node (опционально) — текущий узел.

Дополнительно может передаваться текущая минимальная длина пути и путь.

12. Какую задачу решает проверка if node is None?

Она задает начальный узел, если он не был передан в качестве параметра.

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

Когда состояние узла совпадает с целевым состоянием задачи (problem.is_goal(node.state)).

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

Проверка на циклы предотвращает бесконечный возврат к ранее посещенным узлам, особенно в графах с циклическими структурами.

15. Что возвращает функция при обнаружении цикла?

Она возвращает None (или failure), указывая, что цикл был обнаружен и продолжение поиска по этому пути невозможно.

16. Как функция обрабатывает дочерние узлы текущего узла?

Функция генерирует дочерние узлы через expand и рекурсивно вызывает саму себя для каждого из них.

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

Используется рекурсия для перехода между узлами, а стек вызовов автоматически сохраняет текущий путь.

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

Функция вернет failure, что указывает на отсутствие пути к цели.

19. Почему функция рекурсивно вызывает саму себя внутри цикла?

Это позволяет исследовать все ветви графа/дерева, начиная с текущего узла.

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

Она генерирует список дочерних узлов текущего узла на основе графа и его соседей.

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

Она проверяет, встречался ли текущий узел ранее в пути, предотвращая зацикливание.

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

Эта проверка определяет, было ли найдено решение по данному пути, и завершает дальнейший поиск, если оно найдено.

23. В каких ситуациях алгоритм может вернуть failure?

• Если узел не может быть расширен (нет дочерних узлов).

• Если все пути исследованы, но цель не достигнута.

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

В рекурсивной реализации используется стек вызовов, управляемый автоматически, в то время как в итеративной используется явный стек для хранения состояния узлов.

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

• Бесконечная рекурсия при отсутствии проверки на глубину или циклы.

• Переполнение стека вызовов, что приведет к ошибке сегментации (stack overflow).

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