

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 №3
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 глубину»

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

Порядок выполнения работы:

Задача 1. Реализация алгоритма заливки.

Листинг 1 – Код программы `general_fill.py`

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

from copy import deepcopy
from typing import Generator

from tree import Problem
from tree import depth_first_recursive_search as dfs

class FloodFill(Problem):
    def __init__(
        self, start: tuple[int, int], matrix: list[list[str]],
        target: str, fill: str
    ) -> None:
        super().__init__(start, None)
        self.grid = deepcopy(matrix)
        self.target = target
        self.fill = fill

    def actions(self, position: tuple[int, int]) -> Generator[tuple[int, int], None, None]:
        row, col = position
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        for dr, dc in directions:
            nr, nc = row + dr, col + dc
            if (
                0 <= nr < len(self.grid)
                and 0 <= nc < len(self.grid[0])
                and self.grid[nr][nc] == self.target
            ):
                yield nr, nc

    def result(self, state: tuple[int, int], action: tuple[int, int]) -> tuple[int, int]:
        self.grid[action[0]][action[1]] = self.fill
        return action

def flood_fill(
    start: tuple[int, int], matrix: list[list[str]],
    target: str, fill: str
)
```

```

) -> list[list[str]]:
    problem = FloodFill(start, matrix, target, fill)
    dfs(problem)
    return problem.grid

if __name__ == "__main__":
    data = [
        list("YYYGGGGGGG"),
        list("YYYYYYGXXX"),
        list("GGGGGGGXXX"),
        list("WWWWWGXXGXX"),
        list("WRRRRRGXXX"),
        list("WWWGRRGXXX"),
        list("WBWRRRRRRX"),
        list("WBBBBRRXXX"),
        list("WBBXBBBX.X"),
        list("WBBXXXXXXXX"),
    ]

    print("Заменяем 'X' на 'C':")
    filled_grid = flood_fill((3, 9), data, "X", "C")
    for line in filled_grid:
        print(" ".join(line))

    print("\nЗаменяем 'G' на 'V':")
    filled_grid = flood_fill((0, 3), data, "G", "V")
    for line in filled_grid:
        print(" ".join(line))

```

```

Заменяем 'X' на 'C':
Y Y Y G G G G G G
Y Y Y Y Y Y G C C C
G G G G G G G C C C
W W W W W G X G C C
W R R R R R G C C C
W W W G R R G C C C
W B W R R R R R R C
W B B B B R R C C C
W B B C B B B C . C
W B B C C C C C C C

Заменяем 'G' на 'V':
Y Y Y V V V V V V V
Y Y Y Y Y Y V X X X
V V V V V V V X X X
W W W W W V X G X X
W R R R R R G X X X
W W W G R R G X X X
W B W R R R R R R X
W B B B B R R X X X
W B B X B B B X . X
W B B X X X X X X X

```

Рисунок 1.1 – Вывод результата

Задача 2. Реализация алгоритма поиска самого длинного пути в матрице. Дана матрица символов размером $M \times N$. Необходимо найти длину самого длинного пути в матрице, начиная с заданного символа. Каждый следующий символ в пути должен алфавитно следовать за предыдущим без пропусков.

Листинг 2 – Код программы `general_len.py`

```

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

from typing import Generator

from tree import Problem
from tree import depth_first_recursive_search as dfs

class LenProblem(Problem):
    def __init__(
        self,
        initial: tuple[int, int] | None,

```

```

    goal: tuple[int, int] | None,
    matrix: list[list[str]],
    start: str,
) -> None:
    super().__init__(initial, goal)
    self.matrix = matrix
    self.max_len = 0
    self.start = start

def actions(self, state: tuple[int, int]) -> Generator[tuple[int, int], None, None]:
    found = False
    r, c = state
    directions = [
        (-1, 0), (1, 0), (0, -1), (0, 1),
        (-1, -1), (-1, 1), (1, -1), (1, 1)
    ]
    for dr, dc in directions:
        nr, nc = r + dr, c + dc
        if (
            0 <= nr < len(self.matrix)
            and 0 <= nc < len(self.matrix[0])
            and ord(self.matrix[nr][nc]) - ord(self.matrix[r][c]) == 1
        ):
            yield nr, nc
            found = True

    if not found:
        length = ord(self.matrix[r][c]) - ord(self.start)
        self.max_len = max(self.max_len, length)

def result(
    self, state: tuple[int, int], action: tuple[int, int]
) -> tuple[int, int]:
    return action

def solve(start: str, matrix: list[list[str]]) -> int:
    problem = LenProblem(None, None, matrix, start)

    for i, row in enumerate(matrix):
        for j, value in enumerate(row):
            if value == start:
                problem.initial = (i, j)
                dfs(problem)
    return problem.max_len + 1

if __name__ == "__main__":
    matrix1 = [
        ["D", "E", "H", "X", "B"],
        ["A", "O", "G", "P", "E"],
        ["D", "D", "C", "F", "D"],
    ]

```

```

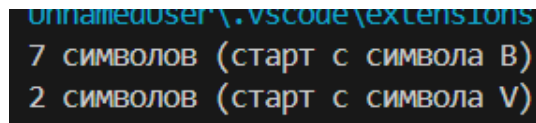
["E", "B", "E", "A", "S"],
["C", "D", "Y", "E", "N"],
]

print(solve("B", matrix1), "символов (старт с символа B)")

matrix2 = [
    ["A", "B", "C", "H", "E", "F"],
    ["P", "Q", "A", "S", "T", "G"],
    ["L", "B", "W", "V", "U", "H"],
    ["N", "M", "L", "K", "K", "I"],
]

print(solve("V", matrix2), "символов (старт с символа V)")

```



```

7 символов (старт с символа B)
2 символов (старт с символа V)

```

Рисунок 2.1 – Результат вывода программы

Задача 3. Генерирование списка возможных слов из матрицы символов. Вам дана матрица символов размером $M \times N$. 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.

Листинг 3 – Код программы `general_find_word.py`

```

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

from typing import Generator

from tree import Problem
from tree import depth_first_recursive_search as dfs

# Вам дана матрица символов размером  $M \times N$ . 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.

```
class WordsProblem(Problem):
    def __init__(
        self,
        initial: tuple[tuple[int, int], str] | None,
        goal: str | None,
        board: list[list[str]],
        word: str | None,
    ) -> None:
        super().__init__(initial, goal)
        self.board = board
        self.word = word
        self.visited: set[tuple[int, int]] = set()

    def actions(
        self, state: tuple[tuple[int, int], str]
    ) -> Generator[tuple[int, int], None, None]:
        r, c = state[0]
        directions = [
            (-1, 0), (1, 0), (0, -1), (0, 1),
            (-1, -1), (-1, 1), (1, -1), (1, 1)
        ]
        for dr, dc in directions:
            nr, nc = r + dr, c + dc
            if (
                0 <= nr < len(self.board)
                and 0 <= nc < len(self.board[0])
                and self.word and len(state[1]) < len(self.word)
                and self.board[nr][nc] == self.word[len(state[1])]
                and (nr, nc) not in self.visited
            ):
                self.visited.add((nr, nc))
                yield nr, nc

    def result(
        self, state: tuple[tuple[int, int], str], action: tuple[int, int]
    ) -> tuple[tuple[int, int], str]:
        r, c = action
        return (action, state[1] + self.board[r][c])

    def is_goal(self, state: tuple[tuple[int, int], str]) -> bool:
        return self.word is not None and state[1] == self.word

    def solve(board: list[list[str]], dictionary: list[str]) -> set[str]:
        words = set()
        for word in dictionary:
            for i, row in enumerate(board):
                for j, cell in enumerate(row):
                    if cell == word[0]:
```

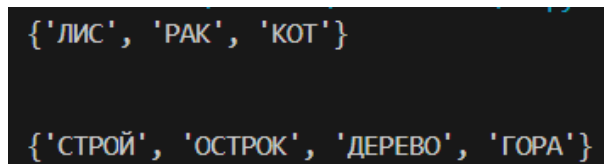
```

        problem = WordsProblem(((i, j), word[0]), word, board, word)
        problem.visited = {(i, j)}
        node = dfs(problem)
        if node:
            words.add(node.state[1])
    return words

if __name__ == "__main__":
    board = [
        ["K", "O", "T", "A"],
        ["P", "A", "K", "T"],
        ["Л", "И", "С", "Ы"],
        ["М", "Е", "Д", "В"]
    ]
    dictionary = ["КОТ", "ЛИС", "РАК", "МЕДВЕДЬ"]
    words = solve(board, dictionary)
    print(words)

    print("\n")
    board = [
        ["Д", "Е", "Р", "Е", "В"],
        ["О", "С", "Т", "О", "К"],
        ["Г", "О", "Р", "А", "Л"],
        ["С", "Т", "Р", "О", "Й"]
    ]
    dictionary = ["ДЕРЕВО", "СТРОЙ", "ГОРА", "ОСТРОК"]
    words = solve(board, dictionary)
    print(words)

```



```

{'ЛИС', 'РАК', 'КОТ'}

{'СТРОЙ', 'ОСТРОК', 'ДЕРЕВО', 'ГОРА'}

```

Рисунок 3.1 – Вывод программы

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

Листинг 4 – Код программы individual.py

```

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

```



```

import networkx as nx
import matplotlib.pyplot as plt

# 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 = 'Гарденс' # 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 маршрута с учётом расстояния с использованием поиска в глубину (DFS)
def dfs_shortest_path(cities, start, end):
    visited = set()
    shortest_path = None
    shortest_distance = float('inf')

    def dfs(current, path, current_distance):
        nonlocal shortest_path, shortest_distance
        if current == end:
            if current_distance < shortest_distance:
                shortest_path = path
                shortest_distance = current_distance
            return
        visited.add(current)
        for neighbor, distance in cities.get(current, { }).items():
            if neighbor not in visited:
                dfs(neighbor, path + [neighbor], current_distance + distance)
        visited.remove(current)

    dfs(start, [start], 0)
    return shortest_path, shortest_distance

# Построение графа и отображение маршрутов
def plot_graph(cities, routes, 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, shortest_distance = dfs_shortest_path(symmetric_cities, start_city, end_city)

# Вывод самого короткого маршрута
if shortest_route:
    print(f"\nСамый короткий маршрут: {' -> '.join(shortest_route)}, Расстояние: {round(shortest_distance, 1)} км")
    plot_graph(symmetric_cities, [shortest_route], shortest_route)
else:
    print("Путь не найден")

```

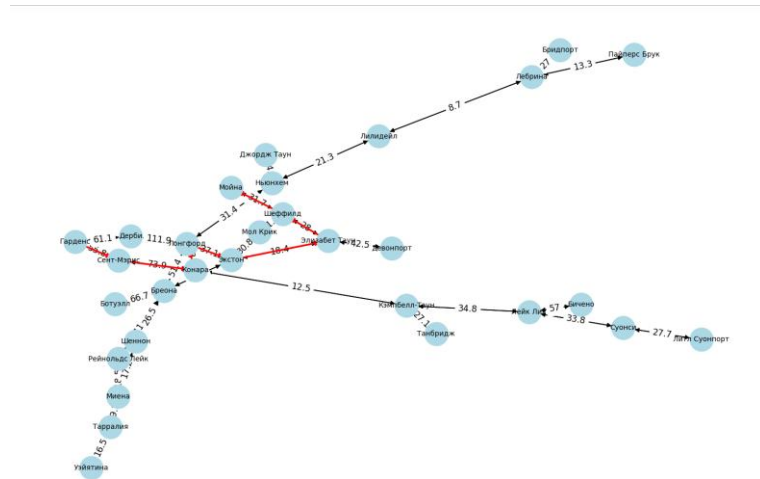


Рисунок 4.1 – Созданный программой граф

Самый короткий маршрут: Гарденс -> Сент-Мэри -> Конара -> Лонгфорд -> Экстон -> Элизабет Таун -> Шеффилд -> Мойна, Расстояние: 288.1 км
 PS: C:\Users\Unamed\Documents\skop\Урок_3\ИИ\AI_Элементы

Рисунок 4.2 – Результат вывода программы

Результат, который вывела программа поиска в глубину, совпадает с результатом программы, написанной в рамках лабораторной работы №2, осуществляющая поиск в ширину.

Ответы на контрольные вопросы:

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

Ключевое отличие поиска в глубину от поиска в ширину заключается в том, что поиск в глубину исследует все возможные пути до максимальной глубины (или решения) перед переходом к следующему уровню, тогда как поиск в ширину рассматривает все узлы на текущем уровне перед тем, как перейти к следующему уровню.

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

Четыре критерия качества поиска, обсуждаемые в тексте, это полнота, оптимальность, время работы и использование памяти.

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

При расширении узла в поиске в глубину алгоритм исследует все дочерние узлы текущего узла до тех пор, пока не будет достигнут конец пути или найдено решение.

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

Поиск в глубину использует очередь типа LIFO (последним пришел — первым ушел), потому что алгоритм всегда расширяет последний добавленный узел, погружаясь в глубину дерева, прежде чем исследовать другие узлы.

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

Поиск в глубину эффективно удаляет узлы из памяти, так как после возврата из рекурсии узлы, которые не ведут к решению, исчезают, что помогает сократить использование памяти. Это является преимуществом перед поиском в ширину, где все узлы на одном уровне хранятся в памяти одновременно.

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

После достижения максимальной глубины дерева в памяти остаются только узлы, которые соответствуют пути от корня до текущего узла, а также те узлы, которые должны быть исследованы для расширения поиска.

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

Поиск в глубину может "застрять", если он попадает в циклические или очень глубокие области дерева, не ведущие к решению.

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

Временная сложность поиска в глубину зависит от максимальной глубины дерева, поскольку количество операций возрастает экспоненциально с увеличением глубины ($O(b^d)$, где b — это ветвление, а d — максимальная глубина).

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

Поиск в глубину не гарантирует нахождение оптимального решения, так как может найти решение в одном из глубочайших ветвей, не проверяя более короткие пути на других уровнях дерева.

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

Поиск в глубину предпочтительно использовать, когда пространство поиска очень большое, и нам нужно исследовать его глубоко в поисках решения, или когда решение может быть найдено на глубоком уровне, и мы не заботимся о его оптимальности.

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

Функция `depth_first_recursive_search` реализует рекурсивный поиск в глубину, принимая текущую задачу (`problem`) и узел (`node`) как параметры. Она исследует все пути, начиная с данного узла.

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

Проверка `if node is None` используется для определения, является ли текущий узел пустым, что может означать конец поиска или некорректный путь.

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

Функция возвращает узел как решение задачи, если найдено решение или если достигнут конец поиска и узел является решением.

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

Проверка на циклы важна для того, чтобы предотвратить заикливание в поиске и избыточное исследование тех же путей, что может привести к бесконечному циклу.

15. Что возвращает функция при обнаружении цикла?

При обнаружении цикла функция возвращает "неудачу" или прекращает дальнейший поиск по этому пути, предотвращая повторное исследование одного и того же узла.

16. Как функция обрабатывает дочерние узлы текущего узла?

Функция обрабатывает дочерние узлы текущего узла, вызывая рекурсию для каждого дочернего узла, чтобы исследовать возможные пути.

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

Механизм обхода дерева поиска в этой реализации использует рекурсивный вызов самой себя для каждого дочернего узла, что позволяет глубже исследовать дерево.

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

Если решение не найдено в ходе рекурсии, функция возвращает "неудачу" или сигнализирует о том, что поиск не привел к успешному результату.

19. Почему функция рекурсивно вызывает саму себя внутри цикла?

Функция рекурсивно вызывает саму себя внутри цикла, чтобы глубже исследовать все возможные пути для каждого дочернего узла, что позволяет охватить все возможные пути в дереве.

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

Функция `expand(problem, node)` взаимодействует с текущим узлом, расширяя его, добавляя дочерние узлы, которые могут быть исследованы дальше.

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

Функция `is_cycle(node)` проверяет, является ли текущий узел частью цикла, чтобы избежать заикливания и повторного посещения узлов.

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

Проверка `if result` в рекурсивном вызове важна для того, чтобы убедиться, что поиск не привел к неудаче и что был найден путь, который решает задачу.

23. В каких ситуациях алгоритм может вернуть `failure`?

Алгоритм может вернуть "неудачу" в случае, если не удалось найти решение, если путь не существует или если поиск застрял в цикле.

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

Рекурсивная реализация отличается от итеративного поиска в глубину тем, что в итеративной версии используется явная стековая структура данных, тогда как рекурсивная версия использует стек вызовов системы.

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

Потенциальные проблемы при использовании алгоритма для поиска в бесконечных деревьях могут включать заикливание или чрезмерное использование памяти, так как глубина поиска может быть бесконечной, что делает поиск невозможным или неэффективным.

Выводы: В процессе выполнения лабораторной работы были приобретены навыки по работе с поиском в глубину с помощью языка программирования Python версии 3.x, проработаны примеры и выполнены индивидуальные задания.

Ссылка на репозиторий:

[IUnnamedUserI/AI_3: Искусственный интеллект в профессиональной сфере. Лабораторная работа №3](#)