

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»

Выполнила:
Михеева Елена Александровна
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:
Воронкин Р.А.-доцент департамента
цифровых, робототехнических систем и
электроники института перспективной
инженерии

(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:

Репозиторий: https://github.com/helendddd/AI_3.git

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

Пример матрицы:

M	N	O	P	Q
L	K	J	I	R
A	B	C	D	E
Z	Y	X	W	V
U	T	S	F	G

```

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

import math
from itertools import product

class Problem:
    """Абстрактный класс для формальной задачи."""
    def __init__(self, initial=None, goal=None, **kwargs):
        self.__dict__.update(initial=initial, goal=goal, **kwargs)

    def actions(self, state):
        raise NotImplementedError

    def result(self, state, action):
        raise NotImplementedError

    def is_goal(self, state):
        return state == self.goal

    def action_cost(self, s, a, s1):
        return 1

    def h(self, node):
        return 0

    def __str__(self):
        return '{0}({1r}, {1r})'.format(
            type(self).__name__, self.initial, self.goal)

class Node:
    """Узел в дереве поиска."""
    def __init__(self, state, parent=None, action=None, path_cost=0):
        self.__dict__.update(
            state=state, parent=parent, action=action, path_cost=path_cost)

    def __repr__(self):
        return '<{}>'.format(self.state)

    def __len__(self):
        return 0 if self.parent is None else (1 + len(self.parent))

    def __lt__(self, other):
        return self.path_cost < other.path_cost

failure = Node('failure', path_cost=math.inf)
cutoff = Node('cutoff', path_cost=math.inf)

def expand(problem, node):
    """Раскрываем узел, создав дочерние узлы."""
    s = node.state
    for action in problem.actions(s):
        s1 = problem.result(s, action)
        cost = node.path_cost + problem.action_cost(s, action, s1)
        yield Node(s1, node, action, cost)

def path_actions(node):
    """Последовательность действий, чтобы добраться до этого узла."""
    if node.parent is None:
        return []
    return path_actions(node.parent) + [node.action]

def path_states(node):
    """Последовательность состояний, чтобы добраться до этого узла."""
    if node in (cutoff, failure, None):
        return []
    return path_states(node.parent) + [node.state]

class MatrixPathProblem(Problem):
    """Задача поиска пути в матрице символов."""
    def __init__(self, matrix, start_char):
        self.matrix = matrix
        self.start_char = start_char
        self.rows = len(matrix)
        self.cols = len(matrix[0])
        self.initial = self.find_start_positions()
        self.goal = None # Целевое состояние не определено

    def find_start_positions(self):
        """Находим все позиции начального символа в матрице."""
        positions = []
        for r, c in product(range(self.rows), range(self.cols)):
            if self.matrix[r][c] == self.start_char:
                positions.append((r, c))
        return positions

    def actions(self, state):
        """Возвращаем список допустимых действий из состояния."""
        r, c = state
        possible_actions = []
        for (dr, dc) in product([-1, 0, 1], repeat=2):
            if (dr != 0 or dc != 0) and
                0 <= r + dr < self.rows and
                0 <= c + dc < self.cols and
                ord(self.matrix[r + dr][c + dc]) == ord(self.matrix[r][c]) + 1:
                possible_actions.append((dr, dc))
        return possible_actions

    def result(self, state, action):
        """Возвращаем новое состояние после применения действия."""
        r, c = state
        dr, dc = action
        return (r + dr, c + dc)

    def is_goal(self, state):
        """Переопределяем метод, но цель не определена явно."""
        return False

def depth_first_search(problem):
    """Поиск в глубину для нахождения самого длинного пути."""
    def recursive_dfs(node, visited):
        visited.add(node.state)
        max_length = 0
        for child in expand(problem, node):
            if child.state not in visited:
                length = recursive_dfs(child, visited)
                max_length = max(max_length, length)
        visited.remove(node.state)
        return 1 + max_length

    max_path_length = 0
    for start_state in problem.initial:
        node = Node(start_state)
        max_path_length = max(
            max_path_length, recursive_dfs(node, set()))
    return max_path_length

if __name__ == "__main__":
    matrix = [
        ['D', 'E', 'H', 'X', 'B'],
        ['A', 'O', 'G', 'P', 'E'],
        ['D', 'D', 'C', 'F', 'D'],
        ['E', 'D', 'E', 'A', 'S'],
        ['C', 'D', 'Y', 'E', 'N']
    ]
    start_char = 'C'
    problem = MatrixPathProblem(matrix, start_char)
    result = depth_first_search(problem)
    print(f"Длина самого длинного пути, начиная с символа '{start_char}': {result}")

```

Рисунок 2. Код программы

```

(venv) (base) elenamiheeva@MacBook-Pro-Elena AI_3 % python program/find_longest_path.py
Длина самого длинного пути, начиная с символа 'C': 6

```

Рисунок 3. Результат работы программы

Генерирование списка возможных слов из матрицы символов.

Дана матрица символов размером $M \times N$. Необходимо 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.

К	О	Т	И
А	Р	Т	О
И	Т	О	К

Словарь с возможными словами: ['КОТ', 'ТОК', 'КИТ', 'ТАК', 'АРТ', 'СОК'].

```

7 class Problem:
8     """Абстрактный класс для формальной задачи."""
9
10     def __init__(self, initial=None, goal=None, **kwargs):
11         self.__dict__.update(initial=initial, goal=goal, **kwargs)
12
13     def actions(self, state):
14         raise NotImplementedError
15
16     def result(self, state, action):
17         raise NotImplementedError
18
19     def is_goal(self, state):
20         return state == self.goal
21
22     def action_cost(self, s, a, s1):
23         return 1
24
25     def h(self, node):
26         return 0
27
28     def __str__(self):
29         return '{0}({1r}, {2r})'.format(
30             type(self).__name__, self.initial, self.goal)
31
32
33 class Node:
34     """Узел в дереве поиска."""
35
36     def __init__(self, state, parent=None, action=None, path_cost=0):
37         self.__dict__.update(
38             state=state, parent=parent, action=action, path_cost=path_cost)
39
40     def __repr__(self):
41         return '<{}>'.format(self.state)
42
43     def __len__(self):
44         return 0 if self.parent is None else (1 + len(self.parent))
45
46     def __lt__(self, other):
47         return self.path_cost < other.path_cost
48
49 failure = Node('failure', path_cost=math.inf)
50 cutoff = Node('cutoff', path_cost=math.inf)
51
52
53 def expand(problem, node):
54     """Раскрываем узел, создав дочерние узлы."""
55     s = node.state
56     for action in problem.actions(s):
57         s1 = problem.result(s, action)
58         cost = node.path_cost + problem.action_cost(s, action, s1)
59         yield Node(s1, node, action, cost)
60
61
62 def path_actions(node):
63     """Последовательность действий, чтобы добраться до этого узла."""
64     if node.parent is None:
65         return []
66     return path_actions(node.parent) + [node.action]
67
68
69 def path_states(node):
70     """Последовательность состояний, чтобы добраться до этого узла."""
71     if node in (cutoff, failure, None):
72         return []
73     return path_states(node.parent) + [node.state]
74
75
76 def is_valid_move(nx, ny, rows, cols, path):
77     """Проверяет, находится ли (nx, ny) в матрице и посещена ли она."""
78     return (
79         0 <= nx < rows and
80         0 <= ny < cols and
81         (nx, ny) not in path
82     )
83
84
85 def __init__(self, board, dictionary):
86     super().__init__(initial=None)
87     self.board = board
88     self.dictionary = set(dictionary)
89     self.rows = len(board)
90     self.cols = len(board[0]) if self.rows > 0 else 0
91     self.prefixes = self._build_prefix_set(dictionary)
92
93
94 def _build_prefix_set(self, words):
95     """Создает множество всех префиксов из списка слов."""
96     prefixes = set()
97     for word in words:
98         for i in range(len(word)):
99             prefixes.add(word[:i + 1])
100     return prefixes
101
102
103 def actions(self, state):
104     """Возвращает список допустимых действий из текущего состояния."""
105     (x, y, path) = state
106     directions = [(-1, -1), (-1, 0), (-1, 1),
107                  (0, -1), (0, 0), (0, 1),
108                  (1, -1), (1, 0), (1, 1)]
109     for dx, dy in directions:
110         nx, ny = x + dx, y + dy
111         if is_valid_move(nx, ny, self.rows, self.cols, path):
112             yield (nx, ny)
113
114
115 def result(self, state, action):
116     """Возвращает новое состояние после применения действия."""
117     (x, y, path) = state
118     (nx, ny) = action
119     new_path = path + [(nx, ny)]
120     return (nx, ny, new_path)
121
122
123 def is_goal(self, state):
124     """Проверяет, является ли текущее состояние целевым."""
125     (x, y, path) = state
126     word = ''.join(self.board[i][j] for i, j in path)
127     return word in self.dictionary
128
129
130 def find_words_from(self, x, y):
131     """Ищет все слова, начинающиеся с позиции (x, y)."""
132     found_words = set()
133     stack = [(x, y, [(x, y)])]
134     while stack:
135         state = stack.pop()
136         (cx, cy, path) = state
137         word = ''.join(self.board[i][j] for i, j in path)
138         if word in self.dictionary:
139             found_words.add(word)
140         if word in self.prefixes:
141             for action in self.actions(state):
142                 new_state = self.result(state, action)
143                 stack.append(new_state)
144     return found_words
145
146
147 def find_all_words(self):
148     """Ищет все слова в матрице."""
149     found_words = set()
150     for x, y in product(range(self.rows), range(self.cols)):
151         found_words.update(self.find_words_from(x, y))
152     return found_words
153
154
155 if __name__ == '__main__':
156     board = [
157         ['K', 'O', 'T', 'I'],
158         ['A', 'P', 'T', 'O'],
159         ['I', 'T', 'O', 'K'],
160     ]
161     dictionary = ['КОТ', 'ТОК', 'КИТ', 'ТАК', 'АРТ', 'СОК']
162     problem = MatrixWordSearchProblem(board, dictionary)
163     found_words = problem.find_all_words()
164     print(found_words)

```

Рисунок 4. Код программы

```

(venv) (base) elenamiheeva@MacBook-Pro-Elena AI_3 % python program/word_matrix.py
{'ТОК', 'КОТ', 'СОК', 'ТАК', 'АРТ'}

```

Рисунок 5. Результат работы программы

Нахождение минимального расстояния между начальными и конечными пунктами с использованием алгоритма поиска в глубину.

Был построен граф из 20 населенных пунктов Италии. Узлы данного графа представляют населённые пункты, а рёбра — дороги, соединяющие их. Вес каждого ребра соответствует расстоянию между этими пунктами. В качестве начального населенного пункта выбран Милан, а конечного — Алессандрия.

Поиск в глубину не гарантирует нахождения минимального пути, так как он углубляется в одно направление до конца, не оценивая стоимость путей. Поэтому полученный путь, представленный в результатах выполнения программы (рис. 8) не является оптимальным.

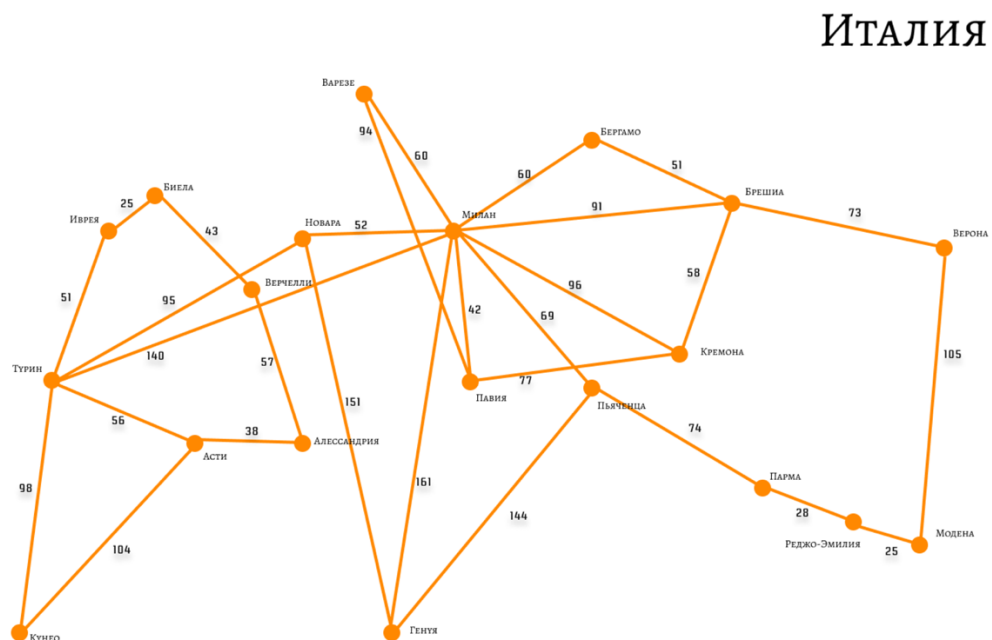


Рисунок 6. Граф из населенных пунктов

```

def __init__(self, initial, goal=None, heuristics=None):
    self.__dict__.update(initial, goal=goal, heuristics=heuristics)

def actions(self, state):
    """Возвращает возможные действия для данного состояния."""
    raise NotImplementedError

def result(self, state, action):
    """Возвращает результат применения действия к состоянию."""
    raise NotImplementedError

def is_goal(self, state):
    """Проверяет, является ли состояние целевым."""
    return state == self.goal

def action_cost(self, s, a, t1):
    """Возвращает стоимость действия."""
    return 1

def h(self, node):
    """Вспомогательная функция, по умолчанию 0."""
    return 0

def __str__(self):
    return f'({type(self).__name__}({self.initial}, {self.goal}))'

class Node:
    """Узел в дереве поиска."""
    def __init__(self, state, parent=None, action=None, path_cost=0):
        self.__dict__.update(
            state=state,
            parent=parent,
            action=action,
            path_cost=path_cost
        )

    def __repr__(self):
        return f'({self.state})'

    def __len__(self):
        return len(self.parent)

    def __lt__(self, other):
        if self.parent is None:
            return 0
        return 1 + (self.path_cost < other.path_cost)

failure = Node('failure', path_cost=math.inf) # Алгоритм не смог найти решение
costff = Node('costff', path_cost=math.inf) # Прерывание поиска

def expand(problem, node):
    """Расширяет узел, создание дочерних узлов."""
    s = node.state
    for action in problem.actions(s):
        cost = node.path_cost + problem.action_cost(s, action, s1)
        yield Node(s1, node, action, cost)

def path_actions(node):
    """Возвращает последовательность действий до узла."""
    if node.parent is None:
        return []
    return path_actions(node.parent) + [node.action]

def path_states(node):
    """Возвращает последовательность состояний до узла."""
    if node in (costff, failure, None):
        return []
    return path_states(node.parent) + [node.state]

class ItalyGraphProblem(Problem):
    """Проблема поиска пути в Италии"""
    def __init__(self, graph, initial, goal):
        super().__init__(initial, goal=goal)
        self.graph = graph

    def actions(self, state):
        """Возвращает соседей (действия) для данного состояния."""
        return self.graph.get(state, [])

    def result(self, state, action):
        """Возвращает результат применения действия (сосед)."""
        return action

    def action_cost(self, s, a, t1):
        """Возвращает стоимость перехода между двумя пунктами."""
        return self.graph[s][a]

    def is_goal(self, state):
        """Проверяет, является ли состояние целевым."""
        return state == self.goal

def depth_first_search(problem):
    """Поиск в глубину для нахождения минимального расстояния."""
    frontier = Node(problem.initial)
    explored = set()

    while frontier:
        node = frontier.pop()

        if node.state in explored:
            continue

        explored.add(node.state)

        if problem.is_goal(node.state):
            return path_states(node), node.path_cost

        for child in expand(problem, node):
            frontier.append(child)

    return None, math.inf

if __name__ == '__main__':
    graph = {
        "Турин": {"Милан": 51, "Венеция": 85, "Модена": 148, "Асти": 56, "Кунео": 98},
        "Милан": {"Турин": 51, "Венеция": 25},
        "Венеция": {"Милан": 25, "Бергамо": 43},
        "Новара": {"Турин": 85, "Милан": 52, "Генуя": 151, "Павия": 42},
        "Бергамо": {"Милан": 43, "Алессандрия": 57},
        "Брешиа": {"Милан": 68, "Павия": 94},
        "Монца": {},
        "Турин": 148,
        "Новара": 52,
        "Брешиа": 68,
        "Бергамо": 68,
        "Брешиа": 94,
        "Кремونا": 91,
        "Пьяченца": 69,
        "Генуя": 52,
        "Павия": 42,
    },
    problem = ItalyGraphProblem(graph, "Милан", "Алессандрия")
    path, cost = depth_first_search(problem)
    print("Путь:", path)
    print("Стоимость:", cost)

```

Рисунок 7. Код программы для задания

```

(venv) (venv) (base) elenaiheeva@MacBook-Pro-Elena AI_3 % python program/DFS.py
Введите начальный город: Милан
Введите конечный город: Алессандрия
Путь: ['Милан', 'Павия', 'Кремона', 'Брешиа', 'Верона', 'Модена', 'Реджо-Эмилия', 'Парма', 'Пьяченца', 'Генуя', 'Новара', 'Турин', 'Кунео', 'Асти', 'Алессандрия']
Стоимость: 1112

```

Рисунок 8. Результат работы программы

Ответы на контрольные вопросы:

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

Поиск в глубину исследует дерево до максимальной глубины перед переходом к соседям, тогда как поиск в ширину изучает узлы на каждом уровне перед углублением.

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

- Полнота;
- оптимальность;
- временная сложность;
- пространственная сложность.

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

При расширении создаются дочерние узлы текущего узла, соответствующие результатам возможных действий.

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

Очередь LIFO позволяет глубже исследовать одно направление, возвращаясь назад только после исчерпания вариантов.

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

Поиск в глубину хранит только текущий путь и соседние узлы, что снижает использование памяти по сравнению с поиском в ширину.

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

В памяти остаются узлы на текущем пути и дочерние узлы самой глубокой вершины.

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

Если дерево бесконечное или если узлы посещаются циклически.

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

Зависит от максимальной глубины дерева: $O(b^m)$, где b — фактор ветвления, m — максимальная глубина.

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

Поиск в глубину находит первое решение, а не самое короткое или дешевое.

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

Когда глубина решения мала, а пространство памяти ограничено.

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

Выполняет рекурсивный поиск в глубину; принимает узел, проблему и список посещенных состояний.

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

Убедиться, что узел существует и может быть обработан.

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

Когда узел соответствует целевому состоянию.

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

Чтобы избежать бесконечных повторений.

15. Что возвращает функция при обнаружении цикла?

Если цикл обнаружен, функция завершает ветвь поиска.

16. Как функция обрабатывает дочерние узлы текущего узла?

Дочерние узлы создаются через `expand` и проверяются на наличие циклов.

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

Используется рекурсивный вызов для обхода дерева.

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

Возвращается `"failure"` после проверки всех узлов.

19. Почему функция рекурсивно вызывает саму себя внутри цикла?

Для продолжения поиска в глубину в новом поддереве.

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

Создает дочерние узлы текущего узла, используя действия и результаты.

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

Проверяет, присутствует ли узел в текущем пути.

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

Останавливает рекурсию, если найдено решение.

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

Если все варианты исследованы и решения нет.

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

Рекурсивный удобнее записывать, но может вызвать переполнение стека.

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

Возможность заикливания или переполнения памяти.

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