

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»
Вариант ____

Выполнил:
Епифанов Алексей Александрович
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 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:

1. Создал новый репозиторий, клонировал его, в нем создал ветку developer и перешел на нее.

Репозиторий: https://github.com/alexeiepif/ii_3.git

2. Выполнил задание: Алгоритм заливки

Flood fill (также известный как seed fill) - это алгоритм, определяющий область, связанную с заданным узлом в многомерном массиве.

Он используется в инструменте заливки "ведро" в программе рисования для заполнения соединенных одинаково окрашенных областей другим цветом, а также в таких играх, как Go и Minesweeper, для определения того, какие фигуры очищены. Когда заливка применяется на изображении для заполнения цветом определенной ограниченной области, она также известна как заливка границ.

Алгоритм заливки принимает три параметра: начальный узел, целевой цвет и цвет замены.

```
> p fill
X-->C
['Y', 'Y', 'Y', 'G', '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', 'G', 'G', 'G', 'C']
['W', 'R', 'R', 'R', 'R', 'R', 'G', 'C', 'C', 'C']
['W', 'W', 'W', 'R', 'R', 'G', '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', 'B', 'C', 'C']
['W', 'B', 'B', 'C', 'C', 'C', 'C', 'C', 'C', 'C']
```

```
G-->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', 'V', 'V', 'V', 'X']
['W', 'R', 'R', 'R', 'R', 'R', 'V', 'X', 'X', 'X']
['W', 'W', 'W', 'R', 'R', 'V', 'V', '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', 'B', 'X', 'X']
['W', 'B', 'B', 'X', 'X', 'X', 'X', 'X', 'X', 'X']
```

Windows | ~/Desktop/./././code/ii_3 | ii-3-py3.12

Рисунок 1. Замена цветов

```
from copy import deepcopy
```

```
from tree import Problem
```

```
from tree import depth_first_recursive_search as dfs
```

```
class FillProblem(Problem):
```

```
    def __init__(self, initial, goal, matrix, target_color, replacement_color):
```

```
        super().__init__(initial, goal)
```

```
        self.matrix = deepcopy(matrix)
```

```
        self.target_color = target_color
```

```
        self.replacement_color = replacement_color
```

```
    def actions(self, state):
```

```
        r, c = state
```

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

```

for dr, dc in directions:
    if (
        0 <= r + dr < len(self.matrix)
        and 0 <= c + dc < len(self.matrix[0])
        and self.matrix[r + dr][c + dc] == self.target_color
    ):
        yield (r + dr, c + dc)

def result(self, state, action):
    self.matrix[action[0]][action[1]] = self.replacement_color
    return action

def solve(initial, goal, matrix, target_color, replacement_color):
    problem = FillProblem(
        initial, goal, matrix, target_color, replacement_color
    )
    dfs(problem)
    return problem.matrix

if __name__ == "__main__":
    matrix = [
        ["Y", "Y", "Y", "G", "G", "G", "G", "G", "G"],
        ["Y", "Y", "Y", "Y", "Y", "Y", "G", "X", "X", "X"],
        ["G", "G", "G", "G", "G", "G", "G", "X", "X", "X"],
        ["W", "W", "W", "W", "W", "G", "G", "G", "G", "X"],
        ["W", "R", "R", "R", "R", "R", "G", "X", "X", "X"],
        ["W", "W", "W", "R", "R", "G", "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", "B", "X", "X"],
        ["W", "B", "B", "X", "X", "X", "X", "X", "X", "X"],
    ]
    print("X-->C")
    start_node = (3, 9)
    target_color = "X"
    replacement_color = "C"

    new_matrix = solve(
        start_node, None, matrix, target_color, replacement_color
    )
    for row in new_matrix:
        print(row)
    print("\nG-->V")
    start_node = (0, 3)
    target_color = "G"
    replacement_color = "V"

    new_matrix = solve(
        start_node, None, matrix, target_color, replacement_color
    )
    for row in new_matrix:
        print(row)

```

3. Выполнил задание: 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.

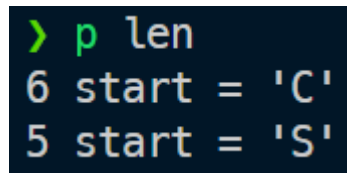


Рисунок 2. Длина наибольшего пути

```
from tree import Problem
from tree import depth_first_recursive_search as dfs

class LenProblem(Problem):
    def __init__(self, initial, goal, matrix, start):
        super().__init__(initial, goal)
        self.matrix = matrix
        self.max_len = 0
        self.start = start

    def actions(self, state):
        sw = 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:
            if (
                0 <= r + dr < len(self.matrix)
                and 0 <= c + dc < len(self.matrix[0])
                and (
                    ord(self.matrix[r + dr][c + dc]) - ord(self.matrix[r][c])
                    == 1
                )
            ):
                yield (r + dr, c + dc)
                sw = True

        if not sw:
            k = ord(self.matrix[r][c]) - ord(self.start)
            if k > self.max_len:
                self.max_len = k

    def result(self, state, action):
```

```

        return action

def solve(start, matrix):
    problem = LenProblem(None, None, matrix, start)

    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            if matrix[i][j] == start:
                problem.initial = (i, j)
                dfs(problem)
    return problem.max_len + 1

if __name__ == "__main__":
    matrix = [
        ["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"],
    ]

    start = "C"
    res = solve(start, matrix)
    print(res, "start = 'C'")

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

    start = "S"
    res = solve(start, matrix)
    print(res, "start = 'S'")

```

4. Выполнил задание: 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.

```

> p words
{'ТОН', 'МАРС'}

{'ФЛОТ', 'ШТОК', 'ДОЛЯ', 'РУКА'}

```

Рисунок 3. Найденные слова

```

from tree import Problem
from tree import depth_first_recursive_search as dfs

```

```

class WordsProblem(Problem):
    def __init__(self, initial, goal, board, word):
        super().__init__(initial, goal)
        self.board = board
        self.word = word
        self.visited = set()

    def actions(self, state):
        r, c = state[0]
        letters = state[1]
        directions = [
            (-1, 0),
            (1, 0),
            (0, -1),
            (0, 1),
            (-1, -1),
            (-1, 1),
            (1, -1),
            (1, 1),
        ]
        for dr, dc in directions:
            if (
                0 <= r + dr < len(self.board)
                and 0 <= c + dc < len(self.board[0])
                and (self.board[r + dr][c + dc] == self.word[len(letters)])
                and (r + dr, c + dc) not in self.visited
            ):
                self.visited.add((r + dr, c + dc))
                yield (r + dr, c + dc)

    def result(self, state, action):
        r, c = action
        n_state = (action, state[1] + self.board[r][c])
        return n_state

    def is_goal(self, state):
        return state[1] == self.word

def solve(board, dictionary):
    words = set()
    problem = WordsProblem(None, None, board, None)
    for word in dictionary:
        problem.visited = set()
        initial_letter = word[0]
        goal = word

```

```

for i, row in enumerate(board):
    for j, cell in enumerate(row):
        if cell == initial_letter:
            initial = ((i, j), initial_letter)
            problem.initial = initial
            problem.goal = goal
            problem.word = word
            word = dfs(problem)
            if word:
                words.add(word.state[1])
return words

if __name__ == "__main__":
    board = [["M", "C", "E"], ["P", "A", "T"], ["Л", "О", "Н"]]
    dictionary = ["МАРС", "СОН", "ЛЕТО", "ТОН"]
    words = solve(board, dictionary)
    print(words)

    print("\n")
    board = [
        ["Д", "О", "М", "У", "К"],
        ["Е", "Л", "Я", "Р", "А"],
        ["Ш", "К", "А", "Ф", "Т"],
        ["С", "Т", "О", "Л", "Ы"],
    ]
    dictionary = ["ДОЛЯ", "ШТОК", "РУКА", "ФЛОТ", "ДЫМ", "СТУЛ"]
    words = solve(board, dictionary)
    print(words)

```

5. Выполнил задание: 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.

```

> p min_path
Длина кратчайшего пути: 1898.4
Кратчайший путь: [('15', 'Липецк'), ('12', 'Тамбов'), ('1', 'Пенза'), ('2', 'Городище'), ('3', 'Чаадаевка'), ('4', 'Кузнецк'), ('5', 'Никольск'), ('6', 'Саранск'), ('9', 'Ульяновск'), ('7', 'Сызрань'), ('8', 'Саратов'), ('11', 'Самара')]

```

Рисунок 4. Результат

```

import json

from tree import Problem
from tree import depth_first_recursive_search as dfs
from tree import path_states

class MinPathProblem(Problem):
    def __init__(self, initial, goal, nodes, edges):
        super().__init__(initial, goal)
        self.nodes = nodes
        self.edges = edges

    def actions(self, state):

```



```

    for edge in self.edges:
        if (
            edge["data"]["source"] == state[0]
            or edge["data"]["target"] == state[0]
        ):
            yield edge["data"]

def result(self, state, action):
    if state[0] == action["source"]:
        target = action["target"]
    else:
        target = action["source"]
    for node in self.nodes:
        if node["data"]["id"] == target:
            return (node["data"]["id"], node["data"]["label"])

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

def action_cost(self, s, a, s1):
    return a["weight"]

def load_elems(path):
    with open(path) as f:
        elems = json.load(f)

    for i, elem in enumerate(elems):
        if not elem.get("position", None):
            nodes_index = i
            break

    nodes = elems[:nodes_index]
    edges = elems[nodes_index:]
    return nodes, edges

def search(problem: Problem):
    b = dfs(problem)
    length = b.path_cost
    path = path_states(b)
    return length, path

def solve(init, goal, nodes, edges):
    problem = MinPathProblem(init, goal, nodes, edges)
    return search(problem)

if __name__ == "__main__":
    nodes, edges = load_elems("elem_full.json")
    for node in nodes:
        if node["data"]["label"] == "Липецк":
            initial = (node["data"]["id"], node["data"]["label"])
        elif node["data"]["label"] == "Самара":
            goal = (node["data"]["id"], node["data"]["label"])
    length, path = solve(initial, goal, nodes, edges)
    print("Длина кратчайшего пути:", length)
    print("Кратчайший путь:", path)

```

Полученный путь не оптимален, так как есть путь короче - 885.2км.

Так получилось, потому что поиск в глубину не предназначен для поиска минимального пути, он находит случайный путь, и сразу возвращает его.

Ответы на 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, временная сложность, пространственная сложность и оптимальность.

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

При расширении узла в поиске в глубину его дочерние узлы помещаются в очередь.

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

Чтобы пройти по ветви до конца, и уже потом переходить к следующим ветвям.

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

Поиск в глубину предоставляет возможность освобождать узлы из памяти, когда достигается конец ветви и не находится цель, переходя к следующей ветви. Когда ветвь полностью исследована, нет необходимости сохранять её в памяти.

Поиск в глубину обладает возможностью освобождения узлов из памяти, чего нет в поиске в ширину. Поиск в ширину не может освобождать узлы из памяти, поскольку ему еще предстоит достигнуть дна дерева, и узлы на его краю могут потребоваться для генерации новых ветвей. В то время как

поиск в глубину, пройдя по всей ветви и достигнув ее конца без нахождения цели, позволяет удалять эту ветвь из памяти.

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

В памяти остаются все узлы, что и были, кроме узлов после последнего разветвления.

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

В бесконечном дереве поиск в глубину может "застрять" в бесконечной ветви, таким образом, не рассмотрев другие потенциальные решения.

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

Если b — коэффициент ветвления, а m — максимальная глубина дерева, то общее количество узлов, сгенерированных поиском в глубину, составит b^m .

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

Поиск в глубину возвращает первое попавшееся решение, и оно далеко не всегда оптимально.

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

Когда невозможно обеспечить необходимый объем памяти для поиска в ширину

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

Данная функция реализовывает рекурсивный поиск в глубину, принимает объект проблемы, и начальный узел.

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

Проверяет, задан ли начальный узел.

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

В любом случае функция возвращает узел, но если решение было найдено, возвращается найденный узел, иначе – узел `failure`.

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

Для предотвращения заикливания (когда алгоритм постоянно возвращается к уже посещенным узлам) используется проверка `is_cycle(node)`. Если текущий узел создает цикл, функция возвращает специальное значение `failure`, указывающее на неудачу в поиске пути.

15. Что возвращает функция при обнаружении цикла?

Если текущий узел создает цикл, функция возвращает специальное значение `failure`, указывающее на неудачу в поиске пути.

16. Как функция обрабатывает дочерние узлы текущего узла?

Если текущий узел не является целевым и не создает цикл, функция генерирует всех дочерних узлов (`child`) путем расширения текущего узла (`expand(problem, node)`). Затем она рекурсивно вызывает себя для каждого дочернего узла. Если рекурсивный вызов возвращает не `failure`, то найдено решение, и оно возвращается.

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

Рекурсивный поиск в глубину.

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

В этом случае функция возвращает `failure`, указывая на то, что решение не найдено.

19. Почему функция рекурсивно вызывает саму себя внутри цикла?

Она рекурсивно вызывает себя для каждого дочернего узла.

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

Она его расширяет.

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

Проверка на заикливание.

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

Чтобы не завершить работу поиска при завершении неправильной ветви, когда возвращается `failure`.

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

Только если решения нет.

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

При рекурсивной реализации углубление происходит вызовом функции самой себя для дочерних узлов, а при итеративной используется очередь LIFO для хранения узлов и углубления в граф.

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

Алгоритм может углубляться в ветвь, которая изначально не содержит решения, и завершиться только при переполнении стека.

Вывод: выполняя данную работу были получены навыки по работе с поиском в глубину с помощью языка программирования Python версии 3.x