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»
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 ширину

Цель: приобретение навыков по работе с поиском в ширину 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_2.git

2. Выполнил задание: Расширенный подсчет количества островов в бинарной матрице.

Вам дана бинарная матрица, где 0 представляет воду, а 1 представляет землю. Связанные единицы формируют остров. Необходимо подсчитать общее количество островов в данной матрице. Острова могут соединяться как по вертикали и горизонтали, так и по диагонали.

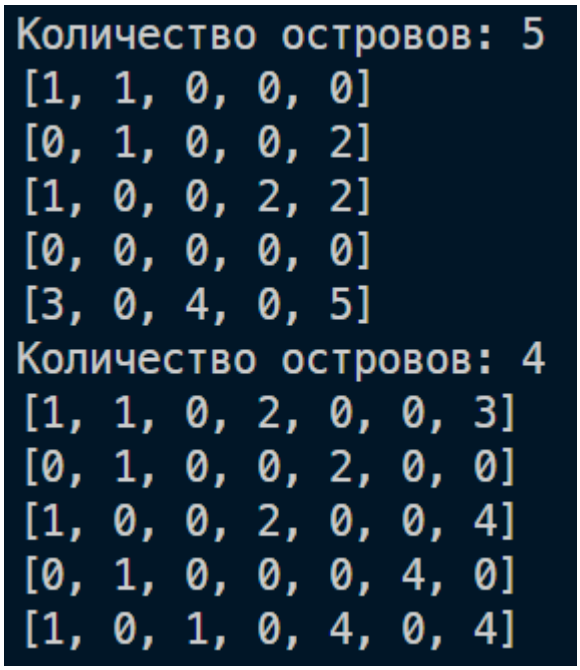

Рисунок 1. Подсчет и отображение островов

```
from tree import FIFOQueue, Problem


class IslandProblem(Problem):
    def __init__(self, grid):
        super().__init__(initial=None)
        self.grid = grid
        self.rows = len(grid)
        self.cols = len(grid[0]) if self.rows > 0 else 0

    def actions(self, state):
```

```
        r, c = state
        directions = [
            (-1, 0),
            (1, 0),
            (0, -1),
            (0, 1),  # Вверх, вниз, влево, вправо
            (-1, -1),
            (-1, 1),
            (1, -1),
            (1, 1),
        ]  # Диагонали
        return [
            (r + dr, c + dc)
            for dr, dc in directions
            if 0 <= r + dr < self.rows and 0 <= c + dc < self.cols
        ]

    def is_land(self, state):

        r, c = state
        return self.grid[r][c] == 1


def bfs_islands(problem, start, visited, new_grid, island_count):
    """Запускает поиск в ширину для поиска всех клеток одного острова."""
    if problem.is_land(start):
        new_grid[start[0]][start[1]] = island_count + 1
    frontier = FIFOQueue([start])
    visited.add(start)

    while frontier:
        node = frontier.pop()
        for action in problem.actions(node):
            if action not in visited and problem.is_land(action):
                visited.add(action)
                new_grid[action[0]][action[1]] = island_count + 1
                frontier.appendleft(action)
    return new_grid


def count_islands(grid):
    problem = IslandProblem(grid)
    visited = set()
    island_count = 0
    new_grid = grid.copy()

    for r in range(problem.rows):
        for c in range(problem.cols):
            state = (r, c)
            if state not in visited and problem.is_land(state):
                new_grid = bfs_islands(
                    problem, state, visited, new_grid, island_count
                )
                island_count += 1

    return island_count, new_grid


if __name__ == "__main__":
    # Пример использования
    grid = [
        [1, 1, 0, 0, 0],
        [0, 1, 0, 0, 1],
```

```
    [1, 0, 0, 1, 1],
    [0, 0, 0, 0, 0],
    [1, 0, 1, 0, 1],
]

island_count, new_grid = count_islands(grid)
print("Количество островов:", island_count)
for row in new_grid:
    print(row)

# еще одна матрица 5 на 7
grid2 = [
    [1, 1, 0, 1, 0, 0, 1],
    [0, 1, 0, 0, 1, 0, 0],
    [1, 0, 0, 1, 0, 0, 1],
    [0, 1, 0, 0, 0, 1, 0],
    [1, 0, 1, 0, 1, 0, 1],
]

island_count, new_grid = count_islands(grid2)
print("Количество островов:", island_count)
for row in new_grid:
    print(row)
```

3. Выполнил задание: поиск кратчайшего пути в лабиринте.

Вам предоставлен код для поиска кратчайшего пути через лабиринт, используя алгоритм поиска в ширину (BFS). Лабиринт представлен в виде бинарной матрицы, где 1 обозначает проход, а 0 — стену. Необходимо модифицировать и дополнить код, чтобы реализовать полный функционал поиска пути.

```
❯ p maze
Длина кратчайшего пути: 12
Кратчайший путь: [(0, 0), (0, 1), (1, 1), (1, 2), (2, 2), (3, 2), (3, 3), (4, 3), (5, 3), (5, 4), (6, 4)
, (7, 4), (7, 5)]
Еще один лабиринт
Длина кратчайшего пути: 40
Кратчайший путь: [(9, 9), (8, 9), (7, 9), (7, 8), (7, 7), (7, 6), (7, 5), (7, 4), (7, 3), (8, 3), (9, 3)
, (9, 2), (9, 1), (8, 1), (7, 1), (6, 1), (5, 1), (4, 1), (4, 0), (3, 0), (2, 0), (2, 1), (2, 2), (2, 3)
, (2, 4), (2, 5), (2, 6), (2, 7), (2, 8), (2, 9), (1, 9), (0, 9), (0, 8), (0, 7), (0, 6), (0, 5), (0, 4)
, (0, 3), (0, 2), (0, 1), (0, 0)]
```

Рисунок 2. Длина кратчайшего пути и узлы, в него входящие

```
from tree import Problem, breadth_first_search, path_states


class MazeProblem(Problem):
    def __init__(self, maze, initial, goal):
        super().__init__(initial=initial, goal=goal)
        self.maze = maze
        self.rows = len(maze)
        self.cols = len(maze[0]) if self.rows > 0 else 0

    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 < self.rows
                and 0 <= c + dc < self.cols
                and self.maze[r + dr][c + dc] != 0
            ):
                yield (r + dr, c + dc)

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

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

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


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


def solve(maze, start, goal):
    problem = MazeProblem(maze, start, goal)
    return search(problem)


if __name__ == "__main__":
    maze = [
        [1, 1, 1, 1, 1, 0, 0, 1, 1, 1],
        [0, 1, 1, 1, 1, 1, 0, 1, 0, 1],
        [0, 0, 1, 0, 1, 1, 1, 0, 0, 1],
        [1, 0, 1, 1, 1, 0, 1, 1, 0, 1],
        [0, 0, 0, 1, 0, 0, 0, 1, 0, 1],
        [1, 0, 1, 1, 1, 0, 0, 1, 1, 0],
        [0, 0, 0, 0, 1, 0, 0, 1, 0, 1],
        [0, 1, 1, 1, 1, 1, 1, 1, 0, 0],
        [1, 1, 1, 1, 1, 0, 0, 1, 1, 1],
        [0, 0, 1, 0, 0, 1, 1, 0, 0, 1],
    ]
    initial = (0, 0)
    goal = (7, 5)
    length, path = solve(maze, initial, goal)
    print("Длина кратчайшего пути:", length)
    print("Кратчайший путь:", path)

    # еще одна матрица 10 на 10
    print("Еще один лабиринт")
    maze2 = [
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [1, 1, 0, 1, 1, 1, 1, 1, 1, 1],
        [1, 1, 0, 1, 1, 1, 1, 1, 1, 1],
        [1, 1, 0, 1, 1, 1, 1, 1, 1, 1],
        [1, 1, 0, 1, 1, 1, 1, 1, 1, 1],
        [1, 1, 0, 1, 0, 0, 0, 0, 0, 1],
        [1, 1, 1, 1, 1, 1, 1, 1, 0, 1],
    ]

    initial = (9, 9)
```

```
goal = (0, 0)

length, path = solve(maze2, initial, goal)
print("Длина кратчайшего пути:", length)
print("Кратчайший путь:", path)
```

4. Выполнил задание: задача о льющихся кувшинах

Реализуйте алгоритм поиска в ширину (BFS) для решения задачи о льющихся кувшинах, где цель состоит в том, чтобы получить заданный объем воды в одном из кувшинов.

```
❯ p pour
Длина наименьшего решения: 4
Кратчайшее решение: [(1, 1, 1), (1, 16, 1), (2, 15, 1), (0, 15, 1), (2, 13, 1)]
Действия: [('Fill', 1), ('Pour', 1, 0), ('Dump', 0), ('Pour', 1, 0)]
```

Рисунок 3. Кратчайшее решение

```
from tree import Problem, breadth_first_search, path_actions, path_states


class PourProblem(Problem):
    def __init__(self, initial: tuple, goal: int, sizes: tuple):
        super().__init__(initial, goal)
        self.sizes = sizes

    def actions(self, _):
        pour = (0, 1, 2)
        for i in pour:
            yield ("Fill", i)
            yield ("Dump", i)
            for j in pour:
                if i != j:
                    yield ("Pour", i, j)

    def result(self, state: tuple, action: tuple):
        list_state = list(state)
        match action:
            case ("Fill", i):
                list_state[i] = self.sizes[i]
            case ("Dump", i):
                list_state[i] = 0
            case ("Pour", i, j):
                diff = list_state[i] - (self.sizes[j] - list_state[j])
                if diff > 0:
                    list_state[i] = diff
                    list_state[j] = self.sizes[j]
                else:
                    list_state[j] += list_state[i]
                    list_state[i] = 0
        return tuple(list_state)

    def is_goal(self, state: tuple):
        return self.goal in state

    def action_cost(self, *_):
        return 1
```

```
def search(problem: Problem):
    b = breadth_first_search(problem)
    length = b.path_cost
    path = path_states(b)
    actions = path_actions(b)
    return length, path, actions


def solve(init: tuple, goal: int, sizes: tuple):
    problem = PourProblem(init, goal, sizes)
    return search(problem)


if __name__ == "__main__":
    initial = (1, 1, 1)
    goal = 13
    sizes = (2, 16, 32)
    length, path, actions = solve(initial, goal, sizes)
    print("Длина наименьшего решения:", length)
    print("Кратчайшее решение:", path)
    print("Действия:", actions)
```

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 ширину находит минимальное расстояние между начальным и конечным пунктами. Сравните найденное решение с решением, полученным вручную.

```
❯ p min_path
Длина кратчайшего пути: 944
Кратчайший путь: [('15', 'Липецк'), ('12', 'Тамбов'), ('8', 'Саратов'), ('11', 'Самара')]
```

Рисунок 4. Результат

```
import json

from tree import Problem, breadth_first_search, 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 = breadth_first_search(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 ширину не учитывает веса ребер, он ищет кратчайший с точки зрения количества узлов путь.

Ответы на контрольные вопросы:

1. Какой тип очереди используется в стратегии поиска в ширину?

В стратегии поиска в ширину первым делом расширяется наименее глубокий из нераскрытых узлов. Этот процесс реализуется с использованием очереди типа "первым пришел - первым ушел" (FIFO).

2. Почему новые узлы в стратегии поиска в ширину добавляются в конец очереди?

Новые узлы помещаются в конец очереди, тогда как те, которые ожидают дольше всех, находятся впереди и обрабатываются в первую очередь. Так, когда мы расширяем узел A, находящийся в корне дерева, остальные узлы B, C, D, E, F и G пока не раскрыты и не сформированы.

3. Что происходит с узлами, которые дольше всего находятся в очереди в стратегии поиска в ширину?

Он расшииряется.

4. Какой узел будет расширен следующим после корневого узла, если используются правила поиска в ширину?

Любой дочерний корневому узел.

5. Почему важно расширять узлы с наименьшей глубиной в поиске в ширину?

Расширение узлов с наименьшей глубиной в поиске в ширину важно, потому что это позволяет осуществлять поиск по всем доступным узлам на текущем уровне перед переходом к следующему уровню.

6. Как временная сложность алгоритма поиска в ширину зависит от коэффициента разветвления и глубины?

$$b + b^2 + b^3 + \cdots + b^d = \frac{b^{d+1} - 1}{b - 1} = O(b^{d+1}).$$

Где b (максимальный коэффициент разветвления) - показывает, сколько дочерних узлов может иметь один узел; d (глубина наименее дорогого решения) - определяет, насколько далеко нужно спуститься по дереву для нахождения оптимального решения.

7. Каков основной фактор, определяющий пространственную сложность алгоритма поиска в ширину?

Пространственная Сложность: $O(b^{d-1})$.

8. В каких случаях поиск в ширину считается полным?

Полнота алгоритма означает его способность находить решение, если оно существует. Поиск в ширину, выполняясь достаточно долго, исследует все дерево поиска. Следовательно, если решение находится где-то в этом дереве, алгоритм обязательно его найдет, что делает его полным. Однако, стоит учесть, что если функция преемника указывает на бесконечное количество возможных действий, достижение следующего уровня дерева станет невозможным. В таком случае, мы оказываемся застрявшими на одном уровне с бесконечным количеством узлов, в то время как решение может быть на более глубоком уровне. Поэтому, при условии, что коэффициент ветвления конечен (что обычно и бывает), поиск в ширину остается полным.

9. Объясните, почему поиск в ширину может быть неэффективен с точки зрения памяти.

Главная проблема поиска в ширину – его высокая требовательность к памяти. Хранение всех узлов в памяти одновременно невозможно на практике из-за их огромного количества. Это делает алгоритм как времязатратным, так и пространственно неэффективным, хотя основная проблема заключается именно в необходимости большого объема памяти.

10. В чем заключается оптимальность поиска в ширину?

Оптимальность подразумевает способность алгоритма находить решение с наименьшей стоимостью среди всех возможных. Оптимальный алгоритм всегда предоставляет решение с минимальными затратами. Это не означает, что алгоритм самый быстрый или экономичный по памяти, но он гарантирует нахождение решения с наименьшей стоимостью.

Оптимальность: да (если стоимость шага = 1).

11. Какую задачу решает функция breadth_first_search?

Данная функция решает задачу поиска в ширину.

12. Что представляет собой объект problem, который передается в функцию?

Объект problem это конкретная задача, реализованная классом наследником от Problem.

13. Для чего используется узел Node(problem.initial) в начале функции?

Создается начальный узел поиска, используя начальное состояние задачи problem.initial.

14. Что произойдет, если начальное состояние задачи уже является целевым?

Возвращается начальный узел как решение.

15. Какую структуру данных использует frontier и почему выбрана именно очередь FIFO?

Очередь frontier типа FIFO (First-In-First-Out), содержащая начальный узел. Эта очередь будет использоваться для хранения узлов, которые нужно расширить. FIFO выбрана, так как именно эта очередь используется в поиске в ширину.

16. Какую роль выполняет множество reached?

Создается множество reached для отслеживания посещенных состояний, чтобы избежать повторного посещения одного и того же состояния.

17. Почему важно проверять, находится ли состояние в множестве reached?

Чтобы избежать повторного посещения одного и того же состояния.

18. Какую функцию выполняет цикл while frontier?

Запускается цикл, который продолжается, пока в границе есть узлы для обработки.

19. Что происходит с узлом, который извлекается из очереди в строке node = frontier.pop()?

Извлекается узел из очереди для расширения.

20. Какова цель функции expand(problem, node)?

Расширение узла node.

21. Как определяется, что состояние узла является целевым?

if problem.is_goal(s):

22. Что происходит, если состояние узла не является целевым, но также не было ранее достигнуто?

Добавляем состояние в множество достигнутых и дочерний узел в начало очереди frontier .

23. Почему дочерний узел добавляется в начало очереди с помощью appendleft(child)?

Чтобы он встал в конец очереди.

24. Что возвращает функция breadth_first_search, если решение не найдено?

Если решение не найдено, возвращается специальный узел failure.

25. Каково значение узла failure и когда он возвращается?

Если решение не найдено, возвращается специальный узел failure.

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