МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ

ФЕДЕРАЦИИ

Федеральное государственное автономное

образовательное учреждение высшего образования

«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Межинститутская базовая кафедра

«Исследование поиска в ширину»

Отчет по лабораторной работе №2

по дисциплине «Конструирование программного обеспечения для систем искусственного интеллекта»

Выполнил студент группы ПИЖ-б-о-21-1
Ключникова Я. Д. « « 2024г.
Подпись студента_____
Работа защищена « «_____2024г.
Проверила Воронкин Р.А. _____
(подпись)

Ставрополь 2024

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

Задание:

1. Изучить теоретический материал работы.

2. Создать общедоступный репозиторий на GitHub, в котором будет использована лицензия MIT и язык программирования Python.

3. Выполните клонирование созданного репозитория.

4. Дополните файл .gitignore необходимыми правилами для работы с IDE PyCharm либо Visual Studio Code.

5. Создайте проект Python в папке репозитория.

6. Проработайте примеры лабораторной работы. Создайте для каждого примера отдельный модуль языка Python. Зафиксируйте изменения в репозитории.

7. Решите задания лабораторной работы с помощью языка программирования Python и элементов программного кода лабораторной работы 1 (имя файла начинается с PR.AI.001.). Проверьте правильность решения каждой задачи на приведенных тестовых примерах.

8. Для задачи «Расширенный подсчет количества островов в бинарной матрице» подготовить собственную матрицу, для которой с помощью разработанной в предыдущем пункте программы, подсчитать количество островов.

9. Для задачи «Поиск кратчайшего пути в лабиринте» подготовить собственную схему лабиринта, а также определить начальную и конечную позиции в лабиринте. Для данных найти минимальный путь в лабиринте от начальной к конечной позиции

10. Решить задачу о льющихся кувшинах

11. 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

12. Зафиксируйте сделанные изменения в репозитории.

13. Добавьте отчет по лабораторной работе в формате PDF в папку doc репозитория. Зафиксируйте изменения.

14. Выполните слияние ветки для разработки с веткой main / master.

15. Отправьте сделанные изменения на сервер GitHub.

16. Отправьте адрес репозитория GitHub на электронный адрес преподавателя

Ход работы:

Ссылка на GitHab: https://github.com/Yana-Kh/SAI-Lab-2

Выполнена задача расширенного подсчета количества островов в бинарной матрице.

Логика программы: была создана матрица visited для отслеживания посещённых клеток. Выполнен обход всех клеток матрицы, если найдена 1 и она ещё не посещена, запускается функция BFS. Она использует очередь для обхода всех соседних клеток, включая диагональные. Каждый раз, когда встречается новая клетка с 1, она добавляется в очередь и помечается как посещённая. Счётчик островов увеличивается каждый раз, когда запускается BFS для нового острова.

Листинг:

```
from collections import deque

# Вспомогательная функция для BFS
def bfs(r, c, visited, rows, cols):
    queue = deque([(r, c)])
    visited[r][c] = True

    # Все возможные направления движения (включая диагонали)
    directions = [
        (-1, 0), (1, 0), (0, -1), (0, 1),  # вверх, вниз, влево, вправо
        (-1, -1), (-1, 1), (1, -1), (1, 1)  # диагонали
    ]

    while queue:
        x, y = queue.popleft()

        # Проверяем все соседние клетки
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
```

```
            # Условие: внутри границ, это земля (1), и ещё не посещено
            if 0 <= nx < rows and 0 <= ny < cols and grid[nx][ny] == 1 and
not visited[nx][ny]:
                visited[nx][ny] = True
                queue.append((nx, ny))

def num_islands(grid):
    if not grid:
        return 0

    rows, cols = len(grid), len(grid[0])
    visited = [[False] * cols for _ in range(rows)]
    islands_count = 0


    # Основной цикл для перебора всех клеток матрицы
    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == 1 and not visited[i][j]:
                # Запускаем BFS для нового острова
                bfs(i, j, visited, rows, cols)
                islands_count += 1

    return islands_count

# Пример использования
grid = [
    [0, 1, 0, 0, 1],
    [0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0],
    [0, 1, 0, 1, 0],
    [1, 0, 1, 0, 1]
]

print(«Общее количество островов:», num_islands(grid))
```

```
C:\Users\User\Desktop\git\SAI-Lab-2\S
Общее количество островов: 3

Process finished with exit code 0
```

Рисунок 1 – Результат решения задачи о расширенном поиске островов

Выполнена задача поиска кратчайшего пути в лабиринте.

Логика программы: Очередь хранит тройки (x, y, steps) – текущие координаты и количество шагов до этой клетки. Множество visited используется для хранения посещённых клеток.

Пока очередь не пуста, берётся клетка из начала очереди. Если это целевая точка, возвращается количество шагов. Для всех четырёх направлений

проверяются границы и проходимость, если клетка доступна и не посещена, то она добавляется в очередь и помечается как посещённая.

Листинг:

```
from collections import deque


def shortest_path(maze, start, goal):
    rows, cols = len(maze), len(maze[0])

    # Проверка на границы и доступность начальной и конечной точки
    if maze[start[0]][start[1]] == 0 or maze[goal[0]][goal[1]] == 0:
        return -1

    # Вспомогательные структуры: очередь и множество посещённых клеток
    queue = deque([(start[0], start[1], 0)])  # (x, y, шаги)
    visited = set()
    visited.add((start[0], start[1]))

    # Возможные направления движения: вверх, вниз, влево, вправо
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    # Алгоритм BFS
    while queue:
        x, y, steps = queue.popleft()

        # Проверка на достижение цели
        if (x, y) == goal:
            return steps

        # Проход по всем возможным направлениям
        for dx, dy in directions:
            nx, ny = x + dx, y + dy

            # Проверка границ и доступности следующей клетки
            if 0 <= nx < rows and 0 <= ny < cols and maze[nx][ny] == 1 and
(nx, ny) not in visited:
                visited.add((nx, ny))
                queue.append((nx, ny, steps + 1))

    # Если цель недостижима
    return -1


# Пример использования
maze = [
    [1, 0, 1, 1, 1, 0, 1, 1, 0, 1],
    [1, 1, 0, 0, 1, 1, 0, 1, 1, 0],
    [0, 1, 1, 0, 0, 1, 1, 0, 1, 1],
    [1, 0, 1, 1, 0, 1, 0, 1, 1, 1],
    [0, 1, 0, 1, 1, 0, 1, 0, 0, 1],
    [1, 1, 1, 0, 1, 1, 1, 1, 0, 0],
    [0, 1, 0, 1, 0, 1, 0, 1, 1, 1],
    [1, 0, 1, 1, 1, 1, 1, 1, 0, 1],
    [1, 1, 0, 0, 1, 1, 1, 1, 1, 0],
    [0, 1, 1, 0, 0, 1, 0, 1, 1, 1],
]

# Начальная и целевая точка
initial = (0, 0)  # Стартовая позиция (верхний левый угол)
goal = (9, 9)     # Целевая позиция (правый нижний угол
```

```
# Запуск функции и вывод результата
result = shortest_path(maze, initial, goal)
print(f»Кратчайший путь: {result if result != -1 else 'Путь недостижим'}»)
```

```
C:\Users\User\Desktop\git\SAI-Lab-2\SAI-L
Кратчайший путь: 18

Process finished with exit code 0
```

Рисунок 2 – Результат решения задачи о поиске кратчайшего пути в лабиринте

Было разработано решение для задачи о льющихся кувшинах.

Логика программы: в очередь добавляется начальное состояние с пустым списком действий. Переменная visited используется для отслеживания уже рассмотренных состояний, чтобы избежать зацикливания.

Пока очередь не пуста, берётся текущее состояние и список действий. Если хотя бы в одном кувшине есть нужный объём воды, возвращается список действий и состояний, иначе выполняются все возможные действия:

- Наполнить кувшин.
- Опустошить кувшин.
- Перелить воду между двумя кувшинами.

Листинг:

```
from collections import deque

def water_jug_bfs(initial, goal, sizes):
    # Инициализируем очередь и множество для отслеживания посещённых
состояний
    queue = deque([(initial, [], [initial])])  # (текущее состояние,
действия, список состояний)
    visited = set()
    visited.add(initial)

    while queue:
        state, actions, path = queue.popleft()

        # Проверка, достигнута ли цель
        if goal in state:
            print(«Все состояния на пути к решению:»)
            for step, s in enumerate(path):
```

```
                print(f»Шаг {step}: {s}»)
            print(«\nПоследовательность действий:», actions)
            return  # Останавливаемся после первого найденного решения

        # Проходим по всем возможным действиям
        for i in range(len(sizes)):
            # 1. Наполнить i-й кувшин до верха
            new_state = list(state)
            new_state[i] = sizes[i]
            if tuple(new_state) not in visited:
                visited.add(tuple(new_state))
                queue.append((tuple(new_state), actions + [('Fill', i)], path
+ [tuple(new_state)]))

            # 2. Опустошить i-й кувшин
            new_state = list(state)
            new_state[i] = 0
            if tuple(new_state) not in visited:
                visited.add(tuple(new_state))
                queue.append((tuple(new_state), actions + [('Dump', i)], path
+ [tuple(new_state)]))

            # 3. Перелить из i-го в j-й кувшин
            for j in range(len(sizes)):
                if i != j:  # Переливаем только между разными кувшинами
                    new_state = list(state)
                    transfer = min(state[i], sizes[j] - state[j])  #
Максимально возможный объём перелива
                    new_state[i] -= transfer
                    new_state[j] += transfer
                    if tuple(new_state) not in visited:
                        visited.add(tuple(new_state))
                        queue.append((tuple(new_state), actions + [('Pour',
i, j)], path + [tuple(new_state)]))

    print(«Решение не найдено»)

# Исходные данные
initial = (1, 3, 2)
goal = 10
sizes = (5, 11, 21)

# Запуск функции
water_jug_bfs(initial, goal, sizes)
```

```
Все состояния на пути к решению:
Шаг 0: (1, 3, 2)
Шаг 1: (5, 3, 2)
Шаг 2: (0, 8, 2)
Шаг 3: (0, 0, 10)

Последовательность действий: [('Fill', 0), ('Pour', 0, 1), ('Pour', 1, 2)]

Process finished with exit code 0
```

Рисунок 3 – Результат решения задачи о льющихся кувшинах

Было разработано решение задачи коммивояжёра, используя поиск в глубину, при имеющейся таблице расстояний между городами. Изменениям подвергся только файл id.py. Добавлены функция breadth_first_search, которая выполняет поиск в ширину и функция run_search для запуска любого алгоритма поиска и вывода результатов.

Листинг id.py:

```
import time
import Queue
from collections import defaultdict
from Problem import Problem
from Node import Node, failure, path_states, expand
from Queue import PriorityQueue, FIFOQueue


# Список городов и расстояний между ними (без повторов)
distances = {
    ('Красный Маныч', 'Новокучерлинский'): 11.8,
    ('Красный Маныч', 'Сабан-Антуста'): 10.9,
    ('Красный Маныч', 'Голубиный'): 2.3,
    ('Красный Маныч', 'Каменная Балка'): 13.8,
    ('Новокучерлинский', 'Ясный'): 20.3,
    ('Сабан-Антуста', 'Каменная Балка'): 15.3,
    ('Сабан-Антуста', 'Кендже-Кулак'): 7.1,
    ('Голубиный', 'Каменная Балка'): 11.1,
    ('Кендже-Кулак', 'Шарахалсун'): 12.9,
    ('Шарахалсун', 'Кучерла'): 7.2,
    ('Кучерла', 'Мирное'): 14.9,
    ('Кучерла', 'Таврический'): 13.2,
    ('Куликовы Копани', 'Маштак-Кулак'): 14.9,
    ('Маштак-Кулак', 'Летняя Ставка'): 9.0,
    ('Летняя Ставка', 'Ясный'): 26.1,
    ('Летняя Ставка', 'Овощи'): 10.2,
    ('Чур', 'Овощи'): 12.8,
    ('Овощи', 'Горный'): 21.9,
    ('Камбулат', 'Малые Ягуры'): 9.8,
    ('Малые Ягуры', 'Казгулак'): 19.3,
    ('Казгулак', 'Ясный'): 40.9,
}


# Задача коммивояжера
class TSP(Problem):
    def __init__(self, start, finish):
        super().__init__(initial=start, goal=finish)
        self.graph = self.build_graph()

    def build_graph(self):
        graph = defaultdict(list)
        for (city1, city2), dist in distances.items():
            graph[city1].append((city2, dist))
            graph[city2].append((city1, dist))
        return graph

    def actions(self, state):
        return self.graph[state]

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

```
        return action[0]

    def action_cost(self, state, action, result):
        return action[1]

# Функция для поиска решения задачи коммивояжера
def search_TSP(problem):
    border = PriorityQueue([Node(problem.initial)])  # Очередь с приоритетом
    path = set()

    while border:
        node = border.pop()
        if problem.is_goal(node.state):
            return path_states(node), node.path_cost

        path.add(node.state)

        for city, cost in problem.actions(node.state):
            child = Node(city, node, path_cost=node.path_cost + cost)
            if child.state not in path:
                border.add(child)

    return failure


# Реализация поиска в ширину
def breadth_first_search(problem):
    node = Node(problem.initial)
    if problem.is_goal(problem.initial):
        return node
    frontier = FIFOQueue([node])
    reached = {problem.initial}
    while frontier:
        node = frontier.pop()
        for child in expand(problem, node):
            s = child.state
            if problem.is_goal(s):
                return child
            if s not in reached:
                reached.add(s)
                frontier.appendleft(child)
    return failure

# Запуск поиска и замер времени
def run_search(problem, search_fn):
    start_time = time.time()
    result = search_fn(problem)
    end_time = time.time()

    if result != failure:
        route = path_states(result)
        print(f»Маршрут: {' -> '.join(route)}»)
        print(f»Общее расстояние: {result.path_cost} км»)
    else:
        print(«Маршрут не найден.»)

    print(f»Время выполнения: {end_time - start_time:.4f} секунд»)

# Инициализация задачи
problem = TSP('Красный Маныч', 'Чур')

# Выбор поиска: BFS или поиск с приоритетами (A*)
print(«Результат BFS:»)
run_search(problem, breadth_first_search)
```

```
Результат BFS:
Маршрут: Красный Маныч -> Новокучерлинский -> Ясный -> Летняя Ставка -> Овощи -> Чур
Общее расстояние: 81.2 км
Время выполнения: 0.0010 секунд
```

Рисунок 6 – Результат решения задачи

Решение алгоритмом поиска в глубину совпадает с решением задачи в предыдущей лабораторной работе, а также с решением, полученным вручную.

Вывод: в ходе выполнения лабораторной работы было выполнено решение задачи коммивояжёра, используя метод поиска в ширину, получены навыки по работе с поиском в ширину с помощью языка программирования Python для решения задач о расширенном поиске островов, поиске выхода из лабиринта, о льющихся кувхинах

Вопросы:

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

В стратегии поиска в ширину используется тип очереди FIFO «первым пришел – первым ушел».

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

В стратегии поиска в ширину (BFS) новые узлы добавляются в конец очереди, чтобы гарантировать обход уровней в порядке их глубины. Это позволяет сначала обрабатывать ближайшие узлы и искать решения минимальным числом шагов.

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

Они будут извлечены и обработаны раньше.

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

Будет выбран узел следующего уровня, расширение проходит как правило слева направо.

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

Расширение узлов с наименьшей глубиной в BFS важно, чтобы гарантировать нахождение кратчайшего пути (в терминах количества шагов) от начального узла до целевого.

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

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

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

Она учитывает максимальное количество узлов, которые необходимо хранить в памяти одновременно. Пространственная сложность будет пропорциональна временной сложности, увеличенной в b раз.

8. В каких случаях поиск в ширину считается полным?

Если решение находится где-то в этом дереве, алгоритм обязательно его найдет, что делает его полным.

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

В контексте любого метода поиска в графе, который требует сохранения каждого расширяемого узла

10. В чем заключается оптимальность поиска в ширину?

Оптимальность поиска в ширину заключается в том, что он всегда находит кратчайший путь к цели при условии, что все переходы имеют одинаковый вес и путь существует.

11. Какую задачу решает функция breadth_first_search?

Функция breadth_first_search решает задачу поиска кратчайшего пути от начального состояния к целевому состоянию в заданной задаче, используя стратегию поиска в ширину.

Она проверяет каждое состояние, расширяя узлы с наименьшей глубиной, пока не найдёт целевое состояние или не исчерпает все возможности.

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

Он описывает задачу коммивояжера.

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

Создается начальный узел поиска, используя начальное состояние задачи.

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?

frontier – это объект типа FIFOQueue, который основан на deque (двусторонней очереди) из модуля collections. Очередь FIFO выбрана так, что

алгоритм исследует все узлы на текущем уровне (или глубине) перед тем, как перейти к следующему уровню.

16. Какую роль выполняет множество reached?

Оно отслеживает посещенные 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?

Он выполняется, пока в границе есть узлы для обработки

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

Он расширяется, находим новые узлы на более низком уровне

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

Раскрыть заданный узел (node) в дереве поиска, создавая и возвращая все дочерние узлы.

21. Как определяется, что состояние узла является целевым?

Состояние узла определяется как целевое с помощью метода is_goal(state) объекта problem. Этот метод сравнивает текущее состояние узла с заданным целевым состоянием (атрибут goal объекта problem). Если они совпадают, метод возвращает True, что указывает на то, что узел достиг целевой цели. Если состояния не совпадают, метод возвращает False

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

Если состояние узла не является целевым и не было ранее достигнуто, оно добавляется в множество reached и в очередь frontier для дальнейшей обработки. Это позволяет продолжить поиск, исследуя новые состояния.

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

Дочерний узел добавляется в начало очереди с помощью appendleft(child) для обеспечения обработки узлов на текущем уровне перед переходом к следующему, что соответствует логике поиска в ширину.

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

Если решение не найдено, функция breadth_first_search возвращает объект failure, который указывает на то, что алгоритм не смог найти решение.

25. Каково значение узла failure и когда он возвращается?

Значение узла failure представляет собой специальный узел, указывающий, что алгоритм не смог найти решение. Он возвращается, когда функция breadth_first_search завершает выполнение и не находит целевое состояние после проверки всех возможных узлов в очереди.