

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 1

Выполнил:
Бабенко Артём Тимофеевич
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:
Ассистент департамента цифровых,
робототехнических систем и
электроники Богданов С.С

(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:

Задание 1. Расширенный подсчет островов в бинарной матрице.

Результат работы программы:

```
D:\Gitlab\AI\AI-in-prof\AI-in-prof2\.venv\Scripts\python.exe D:\Gitlab\AI\AI-in-prof\AI-in-prof2\1.py
Количество островов: 3
Process finished with exit code 0
```

Рисунок 1 – Результат работы программы

Код программы:

```
def count_islands(grid):
    if not grid or not grid[0]:
        return 0

    rows, cols = len(grid), len(grid[0])
    visited = [[False for _ in range(cols)] for _ in range(rows)]
    directions = [(-1, -1), (-1, 0), (-1, 1),
                  (0, -1), (0, 1),
                  (1, -1), (1, 0), (1, 1)] # Все 8 направлений

    def dfs(r, c):
        """Обход в глубину для отметки всех частей одного острова."""
        stack = [(r, c)]
        while stack:
            row, col = stack.pop()
            if 0 <= row < rows and 0 <= col < cols and not visited[row][col] and grid[row][col] == 1:
                visited[row][col] = True
                for dr, dc in directions:
                    stack.append((row + dr, col + dc))

    island_count = 0
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == 1 and not visited[r][c]:
                island_count += 1
                dfs(r, c)

    return island_count

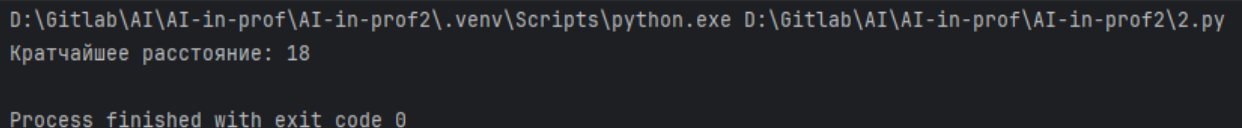
# Пример использования
grid = [
    [0, 1, 0, 0, 1],
```

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

```
result = count_islands(grid)  
print("Количество островов:", result)
```

Задание 2. Поиск кратчайшего пути в лабиринте

Результат работы программы:



```
D:\Gitlab\AI\AI-in-prof\AI-in-prof2\.venv\Scripts\python.exe D:\Gitlab\AI\AI-in-prof\AI-in-prof2\2.py  
Кратчайшее расстояние: 18  
  
Process finished with exit code 0
```

Рисунок 2 – Результат работы программы

Код программы:

```
from collections import deque  
  
def shortest_path(matrix, start, end):  
    # Проверяем корректность входных данных  
    rows, cols = len(matrix), len(matrix[0])  
    if not (0 <= start[0] < rows and 0 <= start[1] < cols):  
        return -1 # Начальная точка вне границ  
    if not (0 <= end[0] < rows and 0 <= end[1] < cols):  
        return -1 # Конечная точка вне границ  
    if matrix[start[0]][start[1]] == 0 or matrix[end[0]][end[1]] == 0:  
        return -1 # Начальная или конечная точка недоступна  
  
    # Направления движения (вверх, вниз, влево, вправо)  
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  
  
    # Матрица для отслеживания посещенных клеток  
    visited = [[False for _ in range(cols)] for _ in range(rows)]  
  
    # Инициализируем очередь для BFS  
    queue = deque()  
    queue.append((start[0], start[1], 0)) # (row, col, distance)  
    visited[start[0]][start[1]] = True  
  
    while queue:  
        row, col, dist = queue.popleft()  
  
        # Если достигли конечной точки, возвращаем расстояние  
        if (row, col) == end:  
            return dist  
  
        # Проверяем всех соседей  
        for dr, dc in directions:
```

```

new_row, new_col = row + dr, col + dc

# Проверяем, что соседняя клетка находится внутри матрицы
if 0 <= new_row < rows and 0 <= new_col < cols:
    if not visited[new_row][new_col] and matrix[new_row][new_col] == 1:
        visited[new_row][new_col] = True
        queue.append((new_row, new_col, dist + 1))

# Если конечная точка недостижима
return -1

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

start = (0, 0) # Начальная точка
end = (9, 9) # Конечная точка

result = shortest_path(matrix, start, end)
if result != -1:
    print(f"Кратчайшее расстояние: {result}")
else:
    print("Путь невозможен")

```

Задание 3. Задача о льющихся кувшинах

Результат работы программы:

```

Начальное состояние: [1, 1, 1]
Размеры кувшинов: [2, 16, 32]
Цель: 13
[[1, 1, 1], [1, 16, 1], [2, 15, 1], [0, 15, 1], [2, 13, 1]]
[('Fill', 1), ('Pour', 1, 0), ('Dump', 0), ('Pour', 1, 0)]

```

Рисунок 3 – Результат работы программы

Код программы:

```
from collections import defaultdict, deque, Counter
from itertools import combinations
from collections import deque

class Node:
    "Узел в дереве поиска"
    def __init__(self, state = [], size_list = [], action=[0,0,0], sizes = [0,0,0]):
        self.__dict__.update(state=state, size_list = size_list, action=action, sizes = sizes
        )
    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

def expand(node):
    "Раскрываем узел, создав дочерние узлы."
    st = node.state
    a = node.action
    sz_list = node.size_list
    sizes = node.sizes
    nodes = []
    #Действие Fill
    for i in range(0,3):
        temp_st = st.copy()
        temp_st.append(("Fill",i))
        temp_a = a.copy()
        temp_a[i] = sizes[i]
        if temp_a not in sz_list:
            temp_sz_list = sz_list.copy()
            temp_sz_list.append(temp_a)
            temp_node = Node(temp_st,temp_sz_list,temp_a,sizes)
            #print("Fill: ",temp_node.action)
            nodes.append(temp_node)
    #Действие Dump
    for i in range(0,3):
        temp_st = st.copy()
        temp_st.append(("Dump",i))
        temp_a = a.copy()
        temp_a[i] = 0
        if temp_a not in sz_list:
            temp_sz_list = sz_list.copy()
            temp_sz_list.append(temp_a)
            temp_node = Node(temp_st,temp_sz_list,temp_a,sizes)
            #print("Dump: ",temp_node.action)
            nodes.append(temp_node)
    #Действие Pour
    for i in range(0,3):
        for j in range(0,3):
            if i != j:
                temp_st = st.copy()
                temp_st.append(("Pour",i,j))
                temp_a = a.copy()
```

```

temp_sz_list = sz_list.copy()

if temp_a[j]+temp_a[i] <= sizes[j]:
    temp_a[j] += temp_a[i]
    temp_a[i] = 0
else:
    temp_a[i] = temp_a[i] - (sizes[j] - temp_a[j])
    temp_a[j] = sizes[j]
if temp_a not in sz_list:
    temp_sz_list.append(temp_a)
    temp_node = Node(temp_st,temp_sz_list,temp_a,sizes)
    #print("Pour: ",temp_node.action, i, j)
    nodes.append(temp_node)

return nodes
initial = [1, 1, 1]
# goal: целевое количество воды
goal = 13
# размеры кувшинов
sizes = [2, 16, 32]
n = Node([], [initial], initial, sizes)
q = deque()
q.append(n)
print(f"Начальное состояние: {initial}\nРазмеры кувшинов: {sizes}\nЦель: {goal}")

final_nodes = []
counter = 0
while q and counter == 0:
    q_node = q.popleft()
    nodes = expand(q_node)
    for item in nodes:
        if goal in item.action:
            #print(item.state, item.path_cost)
            #paths.append(item.state)
            final_nodes.append(item)
            print(item.size_list)
            print(item.state)
            counter += 1
            break
    else:
        q.append(item)

```

Задание 4. 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.

Результат работы программы:

```
D:\Gitlab\AI\AI-in-prof\AI-in-prof2\.venv\Scripts\python.exe D:\Gitlab\AI\AI-in-prof\AI-in-prof2\3.py
Минимальное расстояние между Москва и Екатеринбург: 2000 км
Расстояние, вычисленное вручную: 2450 км
Разница между автоматическим и ручным решением: 450 км

Process finished with exit code 0
```

Рисунок 4 – Результат работы программы

Код программы:

```
import networkx as nx
import matplotlib.pyplot as plt

# Шаг 1: Создание списка населённых пунктов и расстояний между ними
places = [
    "Москва", "Санкт-Петербург", "Новосибирск", "Екатеринбург", "Казань",
    "Нижний Новгород", "Челябинск", "Омск", "Самара", "Ростов-на-Дону",
    "Уфа", "Красноярск", "Воронеж", "Пермь", "Волгоград", "Краснодар",
    "Саратов", "Тюмень", "Тольятти", "Ижевск", "Барнаул"
]

# Матрица расстояний
distances = {
    ("Москва", "Санкт-Петербург"): 650,
    ("Москва", "Казань"): 800,
    ("Москва", "Нижний Новгород"): 450,
    ("Санкт-Петербург", "Новосибирск"): 3200,
    ("Санкт-Петербург", "Казань"): 1000,
    ("Новосибирск", "Екатеринбург"): 1700,
    ("Екатеринбург", "Казань"): 1200,
    ("Екатеринбург", "Челябинск"): 300,
    ("Екатеринбург", "Омск"): 900,
    ("Казань", "Нижний Новгород"): 200,
    ("Казань", "Самара"): 600,
    ("Нижний Новгород", "Челябинск"): 1500,
    ("Челябинск", "Омск"): 800,
    ("Омск", "Красноярск"): 1200,
    ("Самара", "Ростов-на-Дону"): 1100,
    ("Самара", "Уфа"): 400,
    ("Ростов-на-Дону", "Краснодар"): 400,
    ("Уфа", "Красноярск"): 2000,
    ("Уфа", "Пермь"): 500,
    ("Уфа", "Екатеринбург"): 500,
    ("Казань", "Екатеринбург"): 1200,
    ("Красноярск", "Воронеж"): 3000,
    ("Пермь", "Волгоград"): 1000,
    ("Волгоград", "Краснодар"): 500,
    ("Краснодар", "Саратов"): 700,
    ("Саратов", "Тюмень"): 1200,
    ("Тюмень", "Тольятти"): 1300,
    ("Тольятти", "Ижевск"): 400,
```

```
    ("Ижевск", "Барнаул"): 2000,  
}
```

Шаг 2: Создание графа

```
G = nx.Graph()
```

Добавление узлов (населённых пунктов)

```
G.add_nodes_from(places)
```

Добавление рёбер с весами (расстояниями)

```
for (place1, place2), distance in distances.items():
```

```
    G.add_edge(place1, place2, weight=distance)
```

Шаг 3: Алгоритм поиска в ширину (BFS) для нахождения минимального расстояния

```
def bfs_shortest_path(graph, start, end):
```

```
    # Проверка, существуют ли стартовый и конечный узлы в графе
```

```
    if not graph.has_node(start) or not graph.has_node(end):
```

```
        return None, float('inf')
```

```
    # Инициализация очереди и словаря расстояний
```

```
    queue = [(start, 0)] # Очередь содержит пары (узел, текущее расстояние)
```

```
    visited = set()      # Множество посещённых узлов
```

```
    while queue:
```

```
        current_node, current_distance = queue.pop(0) # Извлекаем первый элемент из очереди
```

```
        # Если достигли целевой узел, возвращаем расстояние
```

```
        if current_node == end:
```

```
            return current_node, current_distance
```

```
        # Если узел уже посещён, пропускаем его
```

```
        if current_node in visited:
```

```
            continue
```

```
        # Помечаем узел как посещённый
```

```
        visited.add(current_node)
```

```
        # Добавляем соседей в очередь
```

```
        for neighbor in graph.neighbors(current_node):
```

```
            if neighbor not in visited:
```

```
                edge_weight = graph[current_node][neighbor]['weight']
```

```
                queue.append((neighbor, current_distance + edge_weight))
```

```
        # Если путь не найден
```

```
        return None, float('inf')
```

Шаг 4: Выбор начального и конечного пунктов

```
start_city = "Москва"
```

```
end_city = "Екатеринбург"
```

Поиск минимального расстояния с помощью BFS

```
destination, min_distance = bfs_shortest_path(G, start_city, end_city)
```



```

# Вывод результатов
if destination is None:
    print(f"Путь между {start_city} и {end_city} не найден.")
else:
    print(f"Минимальное расстояние между {start_city} и {end_city}: {min_distance} км")

# Сравнение с решением, полученным вручную
# Предположим, что вручную вычисленное расстояние равно 2450 км
manual_distance = 2450
print(f"Расстояние, вычисленное вручную: {manual_distance} км")
print(f"Разница между автоматическим и ручным решением: {abs(min_distance - manual_distance)} км")

# Визуализация графа и маршрута
pos = nx.spring_layout(G, seed=42)
nx.draw(G, pos, with_labels=True, node_size=500, node_color="lightblue", font_size=10, font_weight="bold")

# Выделение маршрута (если он найден)
try:
    shortest_path = nx.shortest_path(G, source=start_city, target=end_city, weight='weight')
    route_edges = [(shortest_path[i], shortest_path[i + 1]) for i in range(len(shortest_path) - 1)]
    nx.draw_networkx_edges(G, pos, edgelist=route_edges, edge_color='red', width=2)
except nx.NetworkXNoPath:
    print(f"Маршрут между {start_city} и {end_city} не существует.")

plt.title("Граф дорог между населёнными пунктами")
plt.show()

```

Ответы на контрольные вопросы:

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

Используется очередь типа FIFO.

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

Это необходимо для того, чтобы обходить граф слой за слоем, где слоями являются подмножества вершин графа, находящиеся от начальной точки на одинаковом расстоянии.

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

Они обрабатываются в последнюю очередь.

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

Будет расширен первый подходящий дочерний узел корневого узла.

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

Это необходимо для того, чтобы не углубляться, пока не расширены все узлы на текущей глубине.

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

Сложность связана с данными коэффициентами следующим образом: $O(b^{d-1})$.

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

Основной фактор – d (глубина наименее дорогого решения).

8. В каких случаях поиск в ширину считается полным?

В случаях, когда b – конечное значение.

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

Хранение всех узлов в памяти одновременно невозможно на практике из-за их огромного количества. Это делает алгоритм как времязатратным, так и пространственно неэффективным.

10. В чем заключается оптимальность поиска в ширину?

Оптимальность подразумевает способность алгоритма находить решение с наименьшей стоимостью среди всех возможных.

Если длины рёбер графа равны между собой, поиск в ширину является оптимальным.

11. Какую задачу решает функция `breadth_first_search` ?

Данная функция реализует поиск в ширину.

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

Данный объект описывает задачу поиска.

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?

Инициализируется очередь `frontier` типа FIFO (First-In-First-Out), содержащая начальный узел. Эта очередь будет использоваться для хранения узлов, которые нужно расширить.

16. Какую роль выполняет множество `reached` ?

Множество `reached` используется для отслеживания посещенных состояний, чтобы избежать повторного посещения одного и того же состояния.

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

Для корректной работы программы.

18. Какую функцию выполняет цикл `while frontier`?

Он работает, пока очередь не пуста, что позволяет осуществить поиск в ширину.

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

Данный узел удаляется из очереди.

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

Данная функция позволяет расширить узел.

21. Как определяется, что состояние узла является целевым?

Оно сравнивается с целевым состоянием.

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

Данный узел добавляется во множество `reached` и расширяется.

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

Таким образом реализуется очередь типа FIFO.

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

Возвращается специальный узел `failure`.

25. Каково значение узла `failure` и когда он возвращается.

Значение данного узла устанавливается при написании программы, а используется он как возвращаемое значение в случае неудачного поиска.

Вывод: в ходе выполнения работы изучен алгоритм поиска в ширину.