

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО «СЕВЕРО - КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»
ИНСТИТУТ ПЕРСПЕКТИВНОЙ ИНЖЕНЕРИИ**

**Дисциплина «Конструирование программного обеспечения для
систем искусственного интеллекта»**

**Отчет
по лабораторной работе №2
на тему: «Исследование поиска в ширину»**

Выполнил: студент группы ПИЖ-б-о-21-1
Трушева Вероника Олеговна

(подпись)

Проверил: доцент департамента цифровых,
робототехнических систем и электроники
института перспективной инженерии
Воронкин Роман Александрович

(подпись)

Ставрополь, 2024 г.

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

Ход работы

Ссылка на гитхаб: https://github.com/pynikcmd/2_DS_AI.git

1. Изучить теоретический материал работы.
2. Создать общедоступный репозиторий на GitHub, в котором будет использована лицензия MIT и язык программирования Python.

Рисунок 1 – Создание репозитория

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.

```
from Problem import Problem
from Node import Node, failure, expand, path_states
from collections import deque
import heapq

# Список городов и расстояний между ними
distances = {
    ('Ставрополь', 'Михайловск'): 14,
    ('Михайловск', 'Московское'): 25,
    ('Московское', 'Донское'): 20,
    ('Донское', 'Изобильный'): 22,
    ('Михайловск', 'Изобильный'): 46,
    ('Изобильный', 'Новотроицкая'): 14,
    ('Новотроицкая', 'Новоалександровск'): 17,
    ('Новоалександровск', 'Кропоткин'): 56,
    ('Кропоткин', 'Новоивановский'): 25,
    ('Кропоткин', 'Восточный'): 32,
    ('Новоивановский', 'Мирный'): 7,
    ('Мирный', 'Тбилисская'): 14,
    ('Тбилисская', 'Ладожская'): 25,
    ('Восточный', 'Ладожская'): 41,
    ('Ладожская', 'Двубратский'): 14,
    ('Двубратский', 'Васюринская'): 39,
    ('Васюринская', 'Знаменский'): 37,
    ('Васюринская', 'Старокорсунская'): 12,
    ('Старокорсунская', 'Хутор Ленина'): 10,
    ('Хутор Ленина', 'Краснодар'): 25,
    ('Знаменский', 'Краснодар'): 17,
}

# Функция для получения расстояния между городами
def get_distance(city1, city2):
    return distances.get((city1, city2)) or distances.get((city2, city1),
float('inf'))

class TSPPProblem(Problem):
    """Класс для решения задачи коммивояжёра."""

    def __init__(self, initial, goal):
        super().__init__(initial=initial, goal=goal)
        self.cities = list(distances.keys())

    def actions(self, state):
        """Возвращает возможные действия - соседние города."""
        return [city for city in distances.keys() if state in city]

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

```

        """Возвращает следующий город, в который переходит коммивояжер."""
        return action[1] if state == action[0] else action[0]

    def action_cost(self, state, action, result):
        """Возвращает стоимость перехода между городами."""
        return get_distance(state, result)

# Функция поиска в ширину

def breadth_first_search(problem):
    node = Node(problem.initial)
    if problem.is_goal(problem.initial):
        return path_states(node), 0, 1 # 1 сгенерированный узел (начальный)

    frontier = [(0, node)] # Приоритетная очередь: (стоимость, узел)
    reached = {problem.initial: 0} # Отслеживаем минимальную стоимость до
    каждой вершины
    node_count = 1 # Счётчик узлов (начальный узел уже сгенерирован)

    while frontier:
        cost, node = heapq.heappop(frontier) # Извлекаем узел с наименьшей
стоимостью

        # Если цель достигнута
        if problem.is_goal(node.state):
            return path_states(node), cost, node_count

        # Расширяем узлы
        for child in expand(problem, node):
            s = child.state
            new_cost = cost + problem.action_cost(node.state, None,
child.state)

            # Если это новый узел или более дешёвый путь
            if s not in reached or new_cost < reached[s]:
                reached[s] = new_cost
                heapq.heappush(frontier, (new_cost, child)) # Добавляем узел
с его стоимостью
            node_count += 1 # Увеличиваем счётчик узлов

    return failure, float('inf'), node_count # Если решение не найдено

# Инициализация задачи
problem = TSPPProblem(initial='Ставрополь', goal='Краснодар')

# Запуск поиска в ширину
route, distance, generated_nodes = breadth_first_search(problem)

print(f"Минимальный маршрут: {route}")
print(f"Расстояние: {distance} км")
print(f"Сгенерировано узлов: {generated_nodes}")

```



```
Run Main x
"D:\Program Files\Python\Python312\python.exe" "D:\4 курс\ИИ\2_DS_AI\Python\Main
Минимальный маршрут: ['Ставрополь', 'Михайловск', 'Изобильный', 'Новотроицкая',
Расстояние: 318 км
Сгенерировано узлов: 20
Process finished with exit code 0
```

Рисунок 2 – Результат работы программы

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.

```
from collections import deque

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

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

    def breadth_first_search(start_row, start_col):
        # Направления для перемещения (включая диагонали)
        directions = [
            (-1, -1), (-1, 0), (-1, 1), # Верхние строки
            (0, -1), (0, 1), # Горизонтально
            (1, -1), (1, 0), (1, 1) # Нижние строки
        ]

        queue = deque([(start_row, start_col)])
        visited[start_row][start_col] = True

        while queue:
            row, col = queue.popleft()
            for dr, dc in directions:
                new_row, new_col = row + dr, col + dc
                # Проверка границ и посещения
                if 0 <= new_row < rows and 0 <= new_col < cols and not
visited[new_row][new_col] and grid[new_row][
new_col] == 1:
                    visited[new_row][new_col] = True
                    queue.append((new_row, new_col))

    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == 1 and not visited[i][j]: # Найти новый остров
                breadth_first_search(i, j) # Запустить BFS для данного
острова

                islands_count += 1 # Увеличить счетчик островов
```

```

        return islands_count

# Пример ввода
grid1 = [
    [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]
]

grid2 = [
    [1, 0, 0, 1],
    [0, 0, 0, 0],
    [1, 1, 0, 1],
    [0, 0, 1, 1]
]

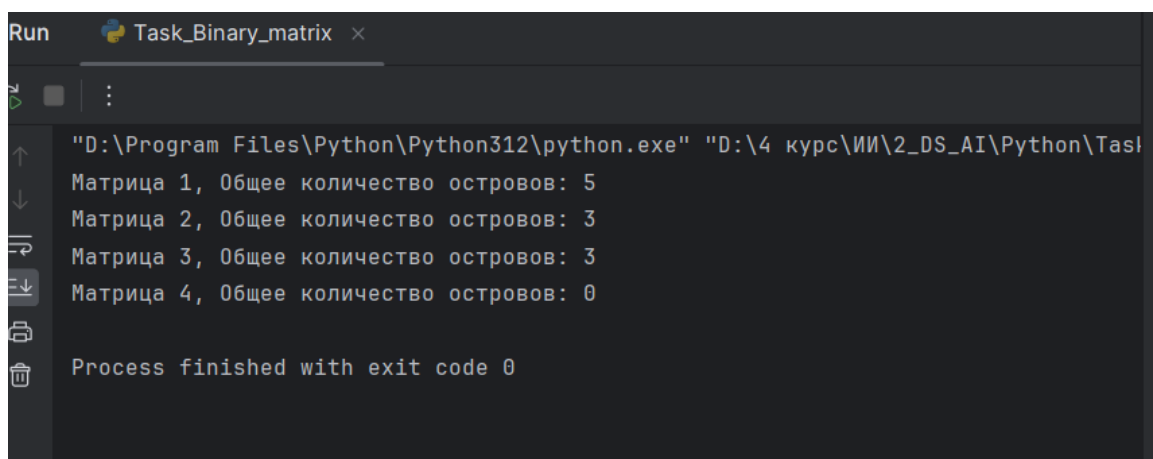
grid3 = [
    [1, 0, 0, 0],
    [1, 0, 1, 1],
    [0, 0, 0, 0],
    [0, 1, 1, 0],
    [1, 0, 0, 1]
]

grid4 = [
    [0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]
]

# Подсчет островов
# Тестирование различных матриц
grids = [grid1, grid2, grid3, grid4]

for index, grid in enumerate(grids, start=1):
    result = count_islands(grid)
    print(f'Матрица {index}, Общее количество островов: {result}')

```



```

Run Task_Binary_matrix x
:
"D:\Program Files\Python\Python312\python.exe" "D:\4 курс\ИИ\2_DS_AI\Python\Task_Binary_matrix.py"
Матрица 1, Общее количество островов: 5
Матрица 2, Общее количество островов: 3
Матрица 3, Общее количество островов: 3
Матрица 4, Общее количество островов: 0
Process finished with exit code 0

```

Рисунок 3 – Результат работы программы

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.

```
from collections import deque

class Problem:
    def __init__(self, initial, goal, maze):
        self.initial = initial
        self.goal = goal
        self.maze = maze

    def actions(self, state):
        row, col = state
        actions = []
        if row > 0 and self.maze[row - 1][col] == 1: # вверх
            actions.append(('UP', (row - 1, col)))
        if row < len(self.maze) - 1 and self.maze[row + 1][col] == 1: # вниз
            actions.append(('DOWN', (row + 1, col)))
        if col > 0 and self.maze[row][col - 1] == 1: # влево
            actions.append(('LEFT', (row, col - 1)))
        if col < len(self.maze[0]) - 1 and self.maze[row][col + 1] == 1: #
            actions.append(('RIGHT', (row, col + 1)))
        return actions

    def result(self, state, action):
        return action[1] # возвращает новое состояние после действия

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

class Node:
    def __init__(self, state, parent=None, action=None, path_cost=0):
        self.state = state
        self.parent = parent
        self.action = action
        self.path_cost = path_cost

    def __repr__(self):
        return f"<{self.state}>"

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

def breadth_first_search(problem):
    node = Node(problem.initial)
    if problem.is_goal(node.state):
        return node

    frontier = deque([node]) # очередь для BFS
    reached = {problem.initial}

    while frontier:
        node = frontier.popleft()
        for action, child_state in problem.actions(node.state):
            if child_state not in reached:
                child_node = Node(child_state, node, action)
                if problem.is_goal(child_state):
                    return child_node
                frontier.append(child_node)
                reached.add(child_state)
```

```

        return None # если решение не найдено

def path_actions(node):
    actions = []
    while node.parent is not None:
        actions.append(node.action)
        node = node.parent
    return actions[::-1]

# Пример лабиринта
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) # начальная позиция
goal = (7, 5) # целевая позиция

problem = Problem(initial, goal, maze)
solution = breadth_first_search(problem)

if solution:
    actions = path_actions(solution)
    print("Длина кратчайшего пути:", len(actions))
else:
    print("Путь не найден.")

```

```

69 # Пример лабиринта
70 maze = [
71     [1, 1, 1, 1, 1, 0, 0, 1, 1, 1],
72     [0, 1, 1, 1, 1, 1, 0, 1, 0, 1],
73     [0, 0, 1, 0, 1, 1, 1, 0, 0, 1],
74     [1, 0, 1, 1, 1, 0, 1, 1, 0, 1],
75     [0, 0, 0, 1, 0, 0, 0, 1, 0, 1],
76     [1, 0, 1, 1, 1, 0, 0, 1, 1, 0],
77     [0, 0, 0, 0, 1, 0, 0, 1, 0, 1],
78     [0, 1, 1, 1, 1, 1, 1, 1, 0, 0],
79     [1, 1, 1, 1, 1, 0, 0, 1, 1, 1],
80     [0, 0, 1, 0, 0, 1, 1, 0, 0, 1]
81 ]
82
83 initial = (0, 0) # начальная позиция
84 goal = (7, 5) # целевая позиция
85

```

Run Task_Labyrinth x



```

"D:\Program Files\Python\Python312\python.exe" "D:\4 курс\ИИ\2_DS_AI\Py
Длина кратчайшего пути: 12

```

Process finished with exit code 0

Рисунок 4 – Результат работы программы


```
69 # Пример лабиринта
70 maze = [
71     [1, 1, 0],
72     [0, 1, 0],
73     [1, 1, 1]
74 ]
75
76 initial = (0, 0) # начальная позиция
77 goal = (2, 2) # целевая позиция
78
79 problem = Problem(initial, goal, maze)
```

Run Task_Labyrinth x

"D:\Program Files\Python\Python312\python.exe" "D:\4 курс\ИИ\2_DS_AI\Py
Длина кратчайшего пути: 4

Process finished with exit code 0

Рисунок 5 – Результат работы программы

10. 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.

```
from collections import deque

class PourProblem:
    def __init__(self, start, goal, sizes):
        self.start = start # начальное состояние кувшинов (объемы воды)
        self.goal = goal # целевой объем воды
        self.sizes = sizes # максимальные размеры кувшинов

    def is_goal(self, state):
        return self.goal in state # проверка, содержит ли один из кувшинов
        целевой объем

    def actions(self, state):
        actions = []
        num_jugs = len(self.sizes)
        # Наполнение каждого кувшина до максимума
        for i in range(num_jugs):
            if state[i] < self.sizes[i]:
                actions.append(('Fill', i))

        # Выливание содержимого каждого кувшина
        for i in range(num_jugs):
            if state[i] > 0:
                actions.append(('Dump', i))

        # Переливание воды из одного кувшина в другой
        for i in range(num_jugs):
            for j in range(num_jugs):
                if i != j and state[i] > 0 and state[j] < self.sizes[j]:
                    actions.append(('Pour', i, j))

        return actions
```

```

def result(self, state, action):
    state = list(state) # создаем копию состояния
    if action[0] == 'Fill':
        i = action[1]
        state[i] = self.sizes[i] # наполняем кувшин до максимума
    elif action[0] == 'Dump':
        i = action[1]
        state[i] = 0 # опустошаем кувшин
    elif action[0] == 'Pour':
        i, j = action[1], action[2]
        transfer = min(state[i], self.sizes[j] - state[j])
        state[i] -= transfer # переливаем воду из i в j
        state[j] += transfer

    return tuple(state)

def bfs(problem):
    # Инициализируем начальный узел
    start_node = problem.start
    if problem.is_goal(start_node):
        return start_node, [] # Если стартовое состояние уже является
# целевым

    frontier = deque([start_node])
    explored = set([start_node])
    paths = {start_node: []}

    while frontier:
        current_state = frontier.popleft()

        for action in problem.actions(current_state):
            new_state = problem.result(current_state, action)

            if new_state not in explored:
                explored.add(new_state)
                frontier.append(new_state)
                paths[new_state] = paths[current_state] + [action]

            if problem.is_goal(new_state):
                return new_state, paths[new_state]

    return None, [] # если решение не найдено

# Примеры для проверки
test_cases = [
    ((1, 2, 1), 2, (3, 2, 5)), # Начальное состояние с доступным решением
    ((1, 0, 0), 2, (3, 5, 8)), # Начальное состояние с другим объемом
    ((1, 1, 1), 3, (3, 5, 7)), # Начальное состояние для получения 3
    ((1, 0, 1), 2, (2, 3, 5)) # Начальное состояние с 1 литром
]

# Выполняем все тесты
for start_state, goal, sizes in test_cases:
    problem = PourProblem(start_state, goal, sizes)
    final_state, solution_path = bfs(problem)

    print(f"Начальное состояние: {start_state}, Цель: {goal}, Размеры:
{sizes}")
    if final_state:
        print("Последовательность действий:", solution_path)
        current_state = start_state
        states = [current_state]
        for action in solution_path:

```

```
        current_state = problem.result(current_state, action)
        states.append(current_state)
        print("Состояния:", states)
    else:
        print("Решение не найдено.")
    print()

74
75 # Примеры для проверки
76 test_cases = [
77     ((1, 2, 1), 2, (3, 2, 5)), # Начальное состояние с доступным решением
78     ((1, 0, 0), 2, (3, 5, 8)), # Начальное состояние с другим объемом
79     ((1, 1, 1), 3, (3, 5, 7)), # Начальное состояние для получения 3
80     ((1, 0, 1), 2, (2, 3, 5)) # Начальное состояние с 1 литром
81 ]
82
83 # Выполняем все тесты
84 for start_state, goal, sizes in test_cases:
85     problem = PourProblem(start_state, goal, sizes)
86     final_state, solution_path = bfs(problem)
```

Run Task_Water x

```
"D:\Program Files\Python\Python312\python.exe" "D:\4 курс\ИИ\2_DS_AI\Python\Task_Water.py"
Начальное состояние: (1, 2, 1), Цель: 2, Размеры: (3, 2, 5)
Последовательность действий: []
Состояния: [(1, 2, 1)]

Начальное состояние: (1, 0, 0), Цель: 2, Размеры: (3, 5, 8)
Последовательность действий: [('Fill', 1), ('Dump', 0), ('Pour', 1, 0)]
Состояния: [(1, 0, 0), (1, 5, 0), (0, 5, 0), (3, 2, 0)]

Начальное состояние: (1, 1, 1), Цель: 3, Размеры: (3, 5, 7)
Последовательность действий: [('Fill', 0)]
```

Рисунок 6 – Результат работы программы

11. Зафиксируйте сделанные изменения в репозитории.
12. Добавьте отчет по лабораторной работе в формате PDF в папку doc репозитория. Зафиксируйте изменения.
13. Выполните слияние ветки для разработки с веткой main / master.
14. Отправьте сделанные изменения на сервер GitHub.
15. Отправьте адрес репозитория GitHub на электронный адрес преподавателя.

Вывод: в ходе выполнения лабораторной работы был изучен поиск в ширину с помощью языка программирования Python версии 3.x.

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?

Узлы, находящиеся в очереди дольше всего, будут обработаны первыми, так как стратегия BFS следует принципу FIFO, что гарантирует, что узлы с меньшей глубиной обрабатываются раньше.

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

Это зависит от выбранной стратегии, но в контексте поиска в ширину первыми расширяются наименее глубокие нераскрытые узлы.

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

Это позволяет гарантировать нахождение кратчайшего пути к целевому состоянию, так как все узлы на текущем уровне (глубине) будут обработаны перед переходом на следующий уровень.

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

Переходя к временной сложности, отметим, что она не измеряется в секундах или циклах процессора, а определяется количеством сгенерированных узлов. Следовательно, временная сложность прямо зависит от числа узлов, которые необходимо создать.

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

Она учитывает максимальное количество узлов, которые необходимо хранить в памяти одновременно. В контексте любого метода поиска в графе, который требует сохранения каждого расширяемого узла, пространственная сложность будет пропорциональна временной сложности, увеличенной в b раз. В частности, при поиске в ширину каждый сгенерированный узел остается в памяти.

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.

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.

11. Какую задачу решает функция `breadth_first_search`?

Функция `breadth_first_search` решает задачу поиска решения в графе состояний, используя алгоритм поиска в ширину.

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

Объект `problem` содержит начальное состояние, целевое состояние и методы для проверки достижения цели, генерации возможных действий и состояния.

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

Узел создается для представления начального состояния задачи в структуре дерева или графа для дальнейшего поиска.

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

Функция сразу вернет узел, представляющий начальное состояние, как решение.

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

`frontier` использует очередь FIFO (First-In-First-Out), чтобы обрабатывать узлы в порядке их добавления, обеспечивая поиск в ширину.

16. Какую роль выполняет множество `reached`?

Множество `reached` отслеживает уже посещенные состояния, чтобы избежать повторного анализа одних и тех же состояний.

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

Это важно для предотвращения бесконечных циклов и увеличения эффективности, так как повторное посещение одного и того же состояния не приведет к новому решению.

18. Какую функцию выполняет цикл `while frontier`?

Цикл продолжает выполняться, пока в `frontier` есть узлы, которые необходимо обработать.

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

Узел извлекается для расширения, и его состояние будет использоваться для генерации дочерних узлов.

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

Функция `expand` генерирует все возможные дочерние узлы (состояния) из текущего узла на основе доступных действий.

21. Как определяется, что состояние узла является целевым?

Состояние узла считается целевым, если оно соответствует критериям, определенным в методе `is_goal` объекта `problem`.

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

Если состояние новое, оно добавляется в множество `reached`, и дочерний узел добавляется в очередь `frontier` для дальнейшего анализа.

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

Добавление дочернего узла в начало очереди может быть связано с приоритетом обработки (в данном случае FIFO может быть изменено на LIFO), хотя в традиционном BFS это не требуется.

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

Если решение не найдено, функция возвращает специальное значение `failure`.

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

Узел `failure` представляет собой маркер, указывающий на то, что решение не было найдено после полного исследования всех возможных узлов. Он возвращается, когда очередь `frontier` пуста и не найдено целевое состояние.