

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»

Выполнил:
Кожуховский Виктор Андреевич
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. Изучил теоретический материал работы.
2. Создал общедоступный репозиторий на GitHub, в котором использована лицензия MIT и язык программирования Python.
3. Выполнил клонирование созданного репозитория.
4. Дополнил файл .gitignore необходимыми правилами для работы с IDE.
5. Организовал свой репозиторий в соответствии с моделью ветвления git-flow.
6. Создал проект в папке репозитория.
7. Проработал примеры лабораторной работы.
8. 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.

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.

```

159 def main():
160     grid = [
161         ["Y", "Y", "Y", "G", "G", "G", "G", "G", "G", "G"],
162         ["Y", "Y", "Y", "Y", "Y", "Y", "G", "X", "X", "X"],
163         ["G", "G", "G", "G", "G", "G", "G", "X", "X", "X"],
164         ["W", "W", "W", "W", "W", "G", "G", "G", "G", "X"],
165         ["W", "R", "R", "R", "R", "R", "G", "X", "X", "X"],
166         ["W", "W", "W", "R", "R", "G", "G", "X", "X", "X"],
167         ["W", "B", "W", "R", "R", "R", "R", "R", "R", "X"],
168         ["W", "B", "B", "B", "B", "R", "R", "X", "X", "X"],
169         ["W", "B", "B", "X", "B", "B", "B", "B", "X", "X"],
170         ["W", "B", "B", "X", "X", "X", "X", "X", "X", "X"],
171     ]
172
173     start_x, start_y = 3, 0 # Начальная позиция для заливки
174     target_color = "W" # Цвет, который нужно заменить
175     replacement_color = "G" # Новый цвет
176
177     problem = ProblemFloodFill(grid, start_x, start_y, target_color, replacement_color)
178     result_node = depth_first_recursive_search(problem)
179
180     for row in result_node.state:
181         print(" ".join(row))
182
183
184 if __name__ == "__main__":
185     main()
186

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS COMMENTS SQL HISTORY TASK MONITOR

```

PS C:\Users\viktor\Desktop\ncfu\ai\AI_3> & "C:/Program Files/Python311/python.exe" c:/Users/viktor/D
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
G G G G G G G G X
G R R R R R G X X X
G G G R R G G X X X
G B G R R R R R X
G B B B B R X X X
G B B X B B B X X
G B B X X X X X X

```

Рисунок 1. Решение задачи Flood fill

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import math
from collections import deque

```

```

class Problem:
    def __init__(self, initial=None, goal=None, **kwargs):
        self.__dict__.update(initial=initial, goal=goal, **kwargs)

    def actions(self, state):
        raise NotImplementedError

    def result(self, state, action):
        raise NotImplementedError

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

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

    def h(self, node):
        return 0

    def __str__(self):
        return "{}({!r}, {!r})".format(type(self).__name__, self.initial, self.goal)

```

```

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

    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

    @staticmethod
    def is_cycle(node):
        parent = node.parent
        while parent:
            if parent.state == node.state:
                return True
            parent = parent.parent
        return False

    failure = None
    cutoff = None

    @staticmethod
    def expand(problem, node):
        s = node.state
        for action in problem.actions(s):
            s1 = problem.result(s, action)
            cost = node.path_cost + problem.action_cost(s, action, s1)
            yield Node(s1, node, action, cost)

Node.failure = Node("failure", path_cost=math.inf)
Node.cutoff = Node("cutoff", path_cost=math.inf)

def depth_first_recursive_search(problem, node=None):
    if node is None:
        node = Node(problem.initial)

    if problem.is_goal(node.state):
        return node
    elif Node.is_cycle(node):
        return Node.failure
    else:
        for child in Node.expand(problem, node):
            result = depth_first_recursive_search(problem, child)
            if result is not Node.failure:
                return result

    return Node.failure

class ProblemFloodFill(Problem):
    def __init__(self, grid, start_x, start_y, target_color, replacement_color):
        initial = self.find_initial_state(grid)
        goal = None
        super().__init__(
            initial=initial,
            goal=goal,
            grid=grid,
            start_x=start_x,
            start_y=start_y,
            target_color=target_color,
            replacement_color=replacement_color,
        )

    def find_initial_state(self, grid):
        return tuple(tuple(row) for row in grid)

    def actions(self, state):
        return [
            (dx, dy)
            for dx, dy in (
                (1, 0),

```

```

        (-1, 0),
        (0, 1),
        (0, -1),
    )
]

def result(self, state, action):
    grid = [list(row) for row in state]
    x, y = self.start_x, self.start_y
    target_color = grid[x][y]

    if target_color == self.replacement_color:
        return state

    self.flood_fill(grid, x, y, target_color)

    return tuple(tuple(row) for row in grid)

def flood_fill(self, grid, x, y, target_color):
    queue = deque([(x, y)])
    visited = set()

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

        if (
            (x, y) in visited
            or x < 0
            or x >= len(grid)
            or y < 0
            or y >= len(grid[0])
            or grid[x][y] != target_color
        ):
            continue

        grid[x][y] = self.replacement_color
        visited.add((x, y))

        for dx, dy in self.actions(grid):
            nx, ny = x + dx, y + dy
            queue.append((nx, ny))

def is_goal(self, state):
    for row in state:
        for cell in row:
            if cell == self.target_color:
                return False
    return True

def main():
    grid = [
        ["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", "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"],
    ]

    start_x, start_y = 3, 0 # Начальная позиция для заливки
    target_color = "W" # Цвет, который нужно заменить
    replacement_color = "G" # Новый цвет

    problem = ProblemFloodFill(grid, start_x, start_y, target_color, replacement_color)
    result_node = depth_first_recursive_search(problem)

    for row in result_node.state:
        print(" ".join(row))

if __name__ == "__main__":
    main()

```

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.

Для задачи "Поиск самого длинного пути в матрице" подготовить собственную матрицу, для которой с помощью разработанной в предыдущем пункте программы, подсчитать самый длинный путь.

```

30 class ProblemLongestPath(Problem):
31     def __init__(self, matrix, start_char):
32         self.matrix = matrix
33         self.start_char = start_char
34         self.rows = len(matrix)
35         self.cols = len(matrix[0]) if self.rows > 0 else 0
36         self.max_path_length = 0
37         super().__init__(initial=None)
38
39     def actions(self, state):
40         x, y = state
41         possible_actions = []
42         directions = [
43             (-1, -1), (-1, 0), (-1, 1),
44             (0, -1), (0, 1),
45             (1, -1), (1, 0), (1, 1)
46         ]
47         path_length = ord(self.matrix[x][y]) - ord(self.start_char)
48         if path_length > self.max_path_length:
49             self.max_path_length = path_length
50
51         for dx, dy in directions:
52             nx, ny = x + dx, y + dy
53             if self.is_valid(nx, ny):
54                 if ord(self.matrix[nx][ny]) == ord(self.matrix[x][y]) + 1:
55                     possible_actions.append((nx, ny))
56
57         return possible_actions
58
59     def result(self, state, action):
60         return action
61
62     def is_valid(self, x, y):
63         return 0 <= x < self.rows and 0 <= y < self.cols
64
65     def longest_path(self):
66         max_length = 0
67         for i in range(self.rows):
68             for j in range(self.cols):
69                 if self.matrix[i][j] == self.start_char:
70                     self.initial = (i, j)
71                     self.max_path_length = 0
72                     depth_first_recursive_search(self)
73                     max_length = max(max_length, self.max_path_length)
74
75         return max_length + 1
76
77     def main():
78         matrix = [
79             ["a", "b", "c", "d", "e", "f", "g"],
80             ["h", "a", "i", "j", "k", "l", "m"],
81             ["n", "o", "p", "q", "r", "s", "t"],
82             ["u", "v", "w", "x", "y", "z", "A"]
83         ]
84         start_char = "a"
85         problem = ProblemLongestPath(matrix, start_char)
86         longest_path_length = problem.longest_path()
87
88         print(
89             f"The length of the longest path starting with '{start_char}' is: {longest_path_length}"
90         )
91
92     if __name__ == "__main__":
93         main()

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GIT LENS COMMENTS SQL HISTORY TASK MONITOR

The length of the longest path starting with 'a' is: 7

Рисунок 2. Решение задачи поиск самого длинного пути в матрице

```

class ProblemLongestPath(Problem):
    def __init__(self, matrix, start_char):
        self.matrix = matrix
        self.start_char = start_char
        self.rows = len(matrix)
        self.cols = len(matrix[0]) if self.rows > 0 else 0
        self.max_path_length = 0
        super().__init__(initial=None)

    def actions(self, state):

```

```

x, y = state
possible_actions = []
directions = [
    (-1, -1), (-1, 0), (-1, 1),
    (0, -1), (0, 1),
    (1, -1), (1, 0), (1, 1)
]

path_length = ord(self.matrix[x][y]) - ord(self.start_char)
if path_length > self.max_path_length:
    self.max_path_length = path_length

for dx, dy in directions:
    nx, ny = x + dx, y + dy
    if self.is_valid(nx, ny):
        if ord(self.matrix[nx][ny]) == ord(self.matrix[x][y]) + 1:
            possible_actions.append((nx, ny))

return possible_actions

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

def is_valid(self, x, y):
    return 0 <= x < self.rows and 0 <= y < self.cols

def longest_path(self):
    max_length = 0
    for i in range(self.rows):
        for j in range(self.cols):
            if self.matrix[i][j] == self.start_char:
                self.initial = (i, j)
                self.max_path_length = 0
                depth_first_recursive_search(self)
                max_length = max(max_length, self.max_path_length)

    return max_length + 1

def main():
    matrix = [
        ["a", "b", "c", "d", "e", "f", "g"],
        ["h", "a", "i", "j", "k", "l", "m"],
        ["n", "o", "p", "q", "r", "s", "t"],
        ["u", "v", "w", "x", "y", "z", "A"]
    ]
    start_char = "a"
    problem = ProblemLongestPath(matrix, start_char)
    longest_path_length = problem.longest_path()

    print(
        f"The length of the longest path starting with '{start_char}' is: {longest_path_length}"
    )

if __name__ == "__main__":
    main()

```

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.

Для задачи "Генерирование списка возможных слов из матрицы символов" подготовить собственную матрицу для генерирования списка возможных слов с помощью разработанной программы.

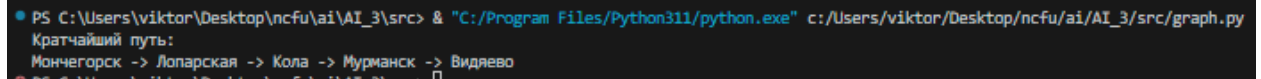
```
88 class WordGenerationProblem:
89     def __init__(self, board, dictionary):
90         self.board = board
91         self.dictionary = dictionary
92         self.rows = len(board)
93         self.cols = len(board[0])
94         self.found_words = []
95         self.directions = [
96             (-1, -1),
97             (-1, 0),
98             (-1, 1),
99             (0, -1),
100             (0, 1),
101             (1, -1),
102             (1, 0),
103             (1, 1),
104         ]
105
106     def is_valid(self, x, y, visited):
107         return 0 <= x < self.rows and 0 <= y < self.cols and (x, y) not in visited
108
109     def dfs(self, x, y, word, current_path, visited):
110         if current_path == word:
111             self.found_words.append(word)
112             return
113
114         if len(current_path) >= len(word):
115             return
116
117         if current_path != word[: len(current_path)]:
118             return
119
120         for dx, dy in self.directions:
121             nx, ny = x + dx, y + dy
122
123             if self.is_valid(nx, ny, visited):
124                 new_visited = visited.copy()
125                 new_visited.add((nx, ny))
126
127                 self.dfs(nx, ny, word, current_path + self.board[nx][ny], new_visited)
128
129     def solve(self):
130         for word in self.dictionary:
131             for x in range(self.rows):
132                 for y in range(self.cols):
133                     if self.board[x][y] == word[0]:
134                         initial_visited = {(x, y)}
135                         self.dfs(x, y, word, self.board[x][y], initial_visited)
136
137         return self.found_words
138
139
140 def main():
141     board = [
142         ["H", "B", "B"], ["B", "B", "B"], ["H", "B", "B"], ["H", "B", "B"]
143     ]
144     dictionary = ["MAP", "CO", "LETO", "TOH"]
145
146     problem = WordGenerationProblem(board, dictionary)
147     found_words = problem.solve()
148
149     print("Найденные слова:")
150     for word in found_words:
151         print(word)
152
153 if __name__ == "__main__":
154     main()
```

PROBLEMS 4 OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS COMMENTS SQL HISTORY TASK MONITOR

Найденные слова:
MAPC
TOH

Рисунок 3. Решение задачи генерирование списка возможных слов из матрицы символов

10. 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.



```
PS C:\Users\viktor\Desktop\ncfu\ai\AI_3\src> & "C:/Program Files/Python311/python.exe" c:/Users/viktor/Desktop/ncfu/ai/AI_3/src/graph.py
Кратчайший путь:
Мончегорск -> Лопарская -> Кола -> Мурманск -> Видяево
```

Рисунок 4. Код программы решения задания и его выполнение

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import math
import json
from collections import deque

class Problem:
    def __init__(self, initial=None, goal=None, **kwargs):
        self.__dict__.update(initial=initial, goal=goal, **kwargs)

    def actions(self, state):
        raise NotImplementedError

    def result(self, state, action):
        raise NotImplementedError

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

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

    def h(self, node):
        return 0

    def __str__(self):
        return "{ }({ !r}, { !r})".format(type(self).__name__, self.initial, self.goal)

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

    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
```

```

    @staticmethod
    def is_cycle(node):
        parent = node.parent
        while parent:
            if parent.state == node.state:
                return True
            parent = parent.parent
        return False

    failure = None
    cutoff = None

    @staticmethod
    def expand(problem, node):
        s = node.state
        for action in problem.actions(s):
            s1 = problem.result(s, action)
            cost = node.path_cost + problem.action_cost(s, action, s1)
            yield Node(s1, node, action, cost)

Node.failure = Node("failure", path_cost=math.inf)
Node.cutoff = Node("cutoff", path_cost=math.inf)

class CityProblem(Problem):
    def __init__(self, cities, distances, initial, goal):
        super().__init__(initial=initial, goal=goal)
        self.cities = cities
        self.distances = distances

    def actions(self, state):
        return [target for target in self.cities if (state, target) in self.distances]

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

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

def depth_first_recursive_search(problem, node=None):
    if node is None:
        node = Node(problem.initial)

    if problem.is_goal(node.state):
        return node
    elif Node.is_cycle(node):
        return Node.failure
    else:
        for child in Node.expand(problem, node):
            result = depth_first_recursive_search(problem, child)
            if result is not Node.failure:
                return result

    return Node.failure

```

```

def reconstruct_path(came_from, current):
    path = []
    while current is not None:
        path.append(current)
        current = came_from[current]
    path.reverse()
    return path

if __name__ == "__main__":
    with open("elem.json", "r", encoding="utf-8") as file:
        data = json.load(file)

    selected_ids = {"8", "9", "2", "15", "6", "1", "3", "7", "13", "18"}

    cities = {}
    distances = {}

    for item in data:
        if "label" in item["data"]:
            if item["data"]["id"] in selected_ids:
                cities[item["data"]["id"]] = item["data"]["label"]
        elif "source" in item["data"]:
            source = item["data"]["source"]
            target = item["data"]["target"]
            if source in selected_ids and target in selected_ids:
                weight = item["data"]["weight"]
                distances[(source, target)] = weight
                distances[(target, source)] = weight

    start_city = "8"
    goal_city = "15"

    problem = CityProblem(cities, distances, start_city, goal_city)
    solution = depth_first_recursive_search(problem)

    if solution is None:
        print("Решение не найдено.")
    else:
        path = []
        current = solution
        while current is not None:
            path.append(current.state)
            current = current.parent
        path.reverse()
        print("Кратчайший путь:")
        print(" -> ".join(cities[city] for city in path))

```

11. Зафиксировал сделанные изменения в репозитории.

12. Выполнил слияние ветки для разработки с веткой master/main.

13. Отправил сделанные изменения на сервер GitHub.

Ссылка: https://github.com/Viktorkozh/AI_3

Контрольные вопросы:

1. В чем ключевое отличие поиска в глубину от поиска в ширину?

Поиск в глубину расширяет самый глубокий из нерасширенных узлов, в то время как поиск в ширину возвращается к узлам на том же уровне.

2. Какие четыре критерия качества поиска обсуждаются в тексте для оценки алгоритмов?

Временная сложность, пространственная сложность, оптимальность и полнота.

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

При расширении узла в поиске в глубину узел становится серым, а его дочерние узлы добавляются в список на рассмотрение.

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

Он расширяет самый глубокий из нерасширенных узлов.

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

Поиск в глубину справляется с удалением узлов из памяти, освобождая узлы, когда достигает конца ветви и не находит цель. Это преимущество перед поиском в ширину, который не может освобождать узлы, так как ему нужно хранить информацию обо всех узлах на текущем уровне.

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

В памяти после достижения максимальной глубины дерева остаются только узлы, которые находятся на пути от корня до текущего узла, а также узлы, которые еще предстоит исследовать.

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

Поиск в глубину может "застрять" в бесконечной ветви, не рассматривая другие потенциальные решения, если он углубляется в бесконечное дерево.

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

Временная сложность поиска в глубину зависит от максимальной глубины дерева, так как общее количество узлов, сгенерированных поиском в глубину, составляет b^m , где b — коэффициент ветвления, а m — максимальная глубина дерева.

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

Поиск в глубину не гарантирует нахождение оптимального решения, так как он может найти решение на большей глубине, чем наименьшая по стоимости.

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

Поиск в глубину предпочтительно использовать в ситуациях, когда пространственная эффективность является приоритетной, и когда недостатки, связанные с неполнотой и потенциально высокой временной сложностью, являются приемлемыми рисками.

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

Функция `depth_first_recursive_search` решает задачу поиска в глубину и принимает два параметра: `problem`, представляющий задачу, и `node`, который является текущим узлом в процессе поиска.

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

Проверка `if node is None` создает начальный узел с использованием начального состояния задачи, если текущий узел не указан.

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

Функция возвращает узел как решение задачи, если состояние текущего узла соответствует целевому состоянию задачи.

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

Проверка на циклы важна в алгоритме рекурсивного поиска в глубину, чтобы предотвратить заикливание, когда алгоритм постоянно возвращается к уже посещенным узлам.

15. Что возвращает функция при обнаружении цикла?

При обнаружении цикла функция возвращает специальное значение `failure`, указывающее на неудачу в поиске пути.

16. Как функция обрабатывает дочерние узлы текущего узла?

Функция обрабатывает дочерние узлы текущего узла, генерируя их путем расширения текущего узла и рекурсивно вызывая себя для каждого дочернего узла.

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

Для обхода дерева поиска в этой реализации используется рекурсивный механизм.

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

Если не будет найдено решение в ходе рекурсии, функция возвращает `failure`, указывая на то, что решение не найдено.

19. Почему функция рекурсивно вызывает саму себя внутри цикла?

Функция рекурсивно вызывает саму себя внутри цикла для поиска решения среди дочерних узлов.

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

Функция `expand(problem, node)` генерирует всех дочерних узлов текущего узла, расширяя его.

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

Роль функции `is_cycle(node)` в этом алгоритме заключается в проверке, создает ли текущий узел цикл, чтобы предотвратить заикливание.

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

Проверка `if result` в рекурсивном вызове важна для корректной работы алгоритма, так как она определяет, найдено ли решение среди дочерних узлов.

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

Алгоритм может вернуть `failure` в ситуациях, когда ни один из дочерних узлов не привел к решению или если обнаружен цикл.

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

Рекурсивная реализация отличается от итеративного поиска в глубину тем, что в рекурсивной реализации используется стек вызовов для хранения информации о текущем пути, тогда как в итеративной реализации используется явный стек.

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

Потенциальные проблемы, которые могут возникнуть при использовании этого алгоритма для поиска в бесконечных деревьях, включают возможность застревания в бесконечных ветвях и высокую временную сложность.

Вывод: приобрел навыки по работе с поиском в глубину с помощью языка программирования Python версии 3.x