Санкт-Петербургский Национальный Исследовательский Университет

Информационных Технологий, Механики и Оптики

Факультет инфокоммуникационных технологий

Лабораторная работа №4

Выполнили:

Лазуренко А.В., Маатук А.

Проверил

Мусаев А.А.

Санкт-Петербург,

2023

ЗАДАНИЕ 1

Для решения задания написали простой алгоритм.

lefts=[]  
rights=[]  
s=input("Enter string: ")  
  
flag=True  
  
for i in range(len(s)):  
 if s[i]=="(": lefts.append(i)  
 elif s[i]==")": rights.append(i)  
  
 if len(rights)>len(lefts):  
 print(rights[-1])  
 flag=False  
 break  
if len(lefts)>len(rights):  
 print(lefts[0])  
 flag=False  
if flag: print("Все отлично!")

*Рисунок 1 – Алгоритм определения ПСП*

Правильность закрывающих скобок проверяется динамически. Если Их количество при переборе превысит количество открывающих скобок, то последовательность перестает быть правильной.

Если же перебор дошел до конца, то в правильной последовательности должно быть одинаковое количество открывающих и закрывающих скобок. Сравниваем их и делаем выводы.

ЗАДАНИЕ 2

Для выполнения задания изучили алгоритмы поиска в графе в глубину и в ширину. Написали программы для поиска кратчайшего пути в неориентированном графе от точки А до точки Б

graph = {  
 'A': {'B':2, 'C':3, 'D':4},  
 'B': {'A':2, 'D':5, 'F':4},  
 'C': {'A':3, 'E':7, 'F':1 },  
 'D': {'A':4, 'B':5, 'E':1, 'F':4},  
 'E': {'D':1, 'C':7},  
 'F': {'B':4, 'C':1, 'D':4}  
}  
  
def dfs(graph, start, end) :  
 stack = [(start, [start], 0)]  
 while stack:  
 (node, path, distance) = stack.pop()  
  
 for neighbor in set(graph[node]) - set(path):  
 if neighbor == end:  
 yield path + [neighbor], distance+graph[node][end]  
 stack.append((neighbor, path+ [neighbor],distance + graph[node][neighbor]))  
  
def bfs(graph, start, end) :  
 queue = [(start, [start], 0)]  
 while queue:  
 (node, path, distance) = queue.pop()  
  
 for neighbor in set(graph[node]) - set(path):  
 if neighbor == end:  
 yield path + [neighbor], distance + graph[node][end]  
 queue.append((neighbor, path + [neighbor], distance + graph[node][neighbor]))  
  
start\_node = 'A'  
end\_node = 'E'  
dfs\_paths = list(dfs(graph, start\_node, end\_node))  
bfs\_paths = list(bfs(graph, start\_node, end\_node))  
  
print("DFS shortest path", min(dfs\_paths, key=lambda x:x[1]))  
print("BFS shortest path", min(bfs\_paths, key=lambda x:x[1]))

*Рисунок 2 – Алгоритмы DFS и BFS*

Алгоритм поиска кратчайшего пути в глубину (DFS) и алгоритм поиска кратчайшего пути в ширину (BFS) являются двумя основными алгоритмами для поиска пути в графе. Они используют разные подходы и имеют разные применения.

Алгоритм поиска в глубину (DFS) работает путем исследования графа до самого его конца, прежде чем начать возврат назад и исследование других вариантов. Он использует стек для хранения текущих вершин и применяется для исследования всех возможных вариантов до достижения целевой вершины. DFS обычно применяется для проверки наличия пути между двумя вершинами, обхода графа или поиска глубинных свойств графа, таких как циклы.

Алгоритм поиска в ширину (BFS) работает путем исследования всех соседних вершин на текущем уровне, прежде чем переходить на следующий уровень. Он использует очередь для хранения текущих вершин и применяется для нахождения кратчайшего пути между двумя вершинами или для поиска всех вершин на определенном расстоянии от заданной вершины.

Выводы:

1. DFS хорошо подходит для задач, связанных с исследованием и обходом графа, проверкой наличия пути между двумя вершинами, поиском циклов или глубинных свойств графа.

2. BFS хорошо подходит для задач, требующих поиска кратчайшего пути между двумя вершинами или поиска всех вершин на определенном расстоянии от исходной вершины.

ЗАДАНИЕ 3

Для решения задачи написали алгоритм поиска выхода из матрицы.

Алгоритм ищет выход сверху вниз.

def find\_exit(maze):  
 n = len(maze)  
 visited = [[False] \* n for \_ in range(n)]  
  
 def dfs(row, col):  
 if row < 0 or row >= n or col < 0 or col >= n:  
 return False  
  
 if maze[row][col] == 1 or visited[row][col]:  
 return False  
  
 visited[row][col] = True  
  
 if row == 0 or row == n - 1 or col == 0 or col == n - 1:  
 return True  
  
 if dfs(row - 1, col): # Вверх  
 return True  
  
 if dfs(row + 1, col): # Вниз  
 return True  
  
 if dfs(row, col - 1): # Влево  
 return True  
  
 if dfs(row, col + 1): # Вправо  
 return True  
  
 return False  
  
 # Начинаем поиск выхода с первой строки  
 for col in range(n):  
 if maze[0][col] == 0:  
 if dfs(0, col):  
 return True  
  
 return False  
  
  
# Пример использования  
maze = [  
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1],  
 [0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],  
 [0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],  
 [0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0],  
 [0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],  
 [0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],  
 [0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1],  
 [0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],  
 [0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0],  
 [0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0],  
 [0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0],  
 [0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],  
 [0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0],  
 [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0],  
 [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0]  
]  
  
if find\_exit(maze):  
 print("Выход найден")  
else:  
 print("Выход не найден")

*Рисунок 3 – Алгоритм поиска выхода*

В алгоритме для каждой клетки проверяем соседние. Используем метод поиска в глубину, поэтому алгоритм проверяет все пути – тупиковые и правильные, к выходу из рекурсии приводит только правильный путь.