Захист <https://youtu.be/lcnZSi7yX_s>

**ЛАБОРАТОРНА РОБОТА №1**

**МЕТОДИ ПОШУКУ У ПРОСТОРІ СТАНІВ**

**Мета.** - Отримати теоретичні та практичні навички по роботі з методами пошуку розв‘язків задач.

**1.1 Теоретичн відомості**

Розрізняють локальний та системний підходи до подання інтелектуальних задач. Локальний або «задачний» підхід заснований на точці зору, що для кожної задачі, властивої творчій діяльності людини, можна знайти спосіб її вирішення на електронній обчислювальній машині (ЕОМ), який, будучи реалізований у вигляді програми, дає результат, або подібний результату, отриманому людиною, або навіть кращий.

Системний або заснований на знаннях підхід пов’язаний із уявленням про те, що розв’язання окремих творчих задач не вичерпує всієї проблематики ШІ. Природний інтелект людини здатний не лише розв’язувати творчі завдання, а за потреби навчається того чи іншого виду творчої діяльності. Тому і програми ШІ повинні бути орієнтовані не лише або не стільки на вирішення конкретних інтелектуальних задач, скільки на створення засобів, що дозволяють автоматично будувати програми вирішення інтелектуальних задач, коли в таких програмах виникне потреба.

У такому підході проблема створення інтелектуальних систем розглядається як частина загальної теорії програмування. При цьому підході для складання інтелектуальних програм використовуються звичайні програмні засоби, що дозволяють писати потрібні програми за описами задач професійною природною мовою. Усі метазасоби, що виникають при цьому на базі часткового аналізу природного інтелекту, розглядаються тут лише з точки зору створення інтелектуального програмного забезпечення, тобто комплексу засобів, що автоматизують діяльність самого програміста.

Оскільки поняттям знання про задачи можуть відповідати стани задачи, а правилам виведення - оператори переходу з одного стану в інший від задачи до підзадачі, то процедура пошуку рішення називається пошуком у просторі станів.

Пошук рішень у просторі станів зводиться до визначення послідовності операторів, які відображають початкові стани в цільові. Причому якщо така послідовність не одна й задано критерій опти- мальності, то пошук зводиться до визначення оптимальної послідовності операторів, які забезпечують оптимум заданого критерію оптимальності.

Методи пошуку рішень у просторі станів зручно розглянути, використовуючи дерево (граф) станів. На дереві станів пошук рішення зводиться до визначення шляху (оптимального, якщо задано критерій опти¬мальності) від кореня дерева до цільової вершини А, тобто до вершини, яка відповідає цільовому стану. Вершини В і С є вершинами, що розкриваються (обчислюваними, проміжними). Вершина D термінальна, тобто завершальна. Ребра, що з’єднують вершини, означають приєднані процедури, які необхідно виконати, щоб перейти до наступного стану.

**Хід роботи**

1. Завдання зображено в таблиці 1.1

Таблиця 1.1

|  |  |
| --- | --- |
| Варіант | Завдання |
| 14 | Реалізувати програму, що здійснює сліпий прямий пошук цільової вершини в глибину на неорієнтованому графі, (що містить 15 вершин) вглибину.Тестові графи створити самостійно. |

Для реалізації завдання було використано мову програмування Python

Створений граф зображений на рисунку 1.1. Реалізація графа в матриці суміжності зображено в лістингу 1.1

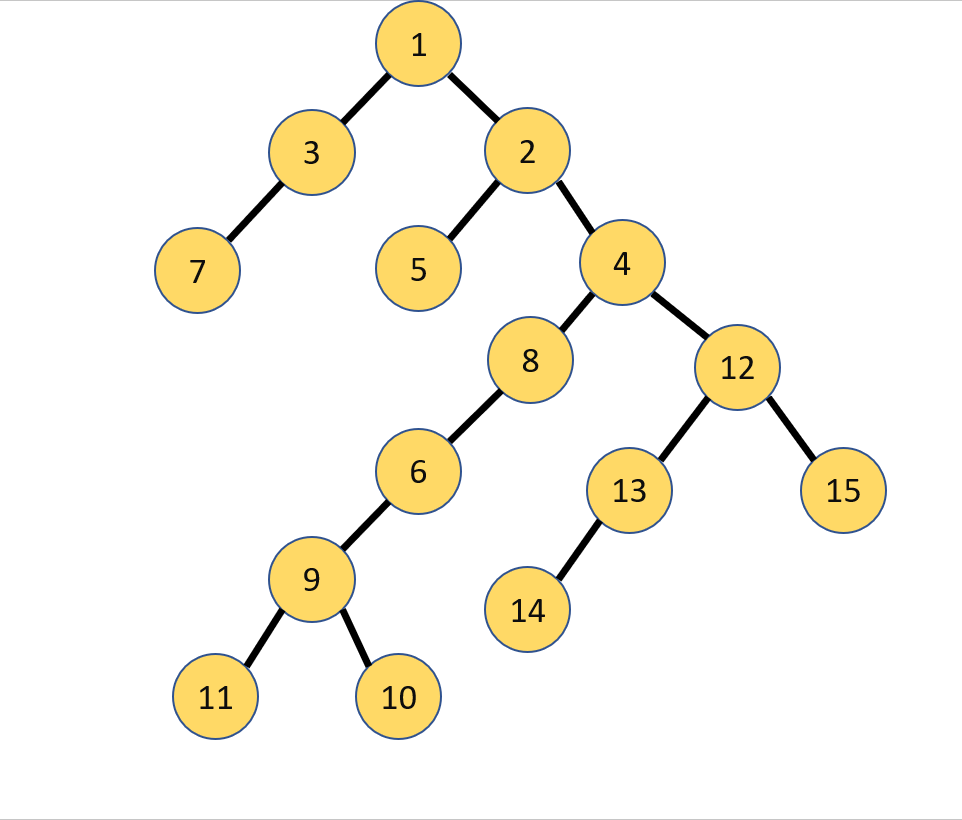


Рисунок 1.1 – Створений граф на 15 вершин

Лістинг 1.1

Adjacency\_Matrix = [

    # 1  2  3  4  5  6  7  8  9  10 11 12 13 14 15

    [0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  # 1

    [1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  # 2

    [1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],  # 3

    [0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0],  # 4

    [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  # 5

    [0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0],  # 6

    [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  # 7

    [0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],  # 8

    [0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0],  # 9

    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],  # 10

    [0, 0, 0, 0, 0, 0, 0, 0, 9, 0, 0, 0, 0, 0, 0],  # 11

    [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1],  # 12

    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0],  # 13

    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],  # 14

    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],  # 15

]

Програма що виконує реалізацію прямого глубинного пошуку зображено в лістингу 1.2. Результат виконання зображено на рисунку 1.2.

Лістинг 1.2

from collections import deque

from graphs import Adjacency\_Matrix

import threading

import time

def bfs(graph\_to\_search, start, end):

    queue = [[start]]  # Створюємо чергу вершин

    print(queue)

    visited = set()  # Створюємо список пройдених вершин на базі сету

    while queue:

        # Отримуємо перший елемент черги, список

        path = queue.pop(0)

        print(path)

        # Отримуємо останній елемент списку

        vertex = path[-1]

        print(vertex)

        # Перевіряємо і повертаємо шлях у разі успіху

        if vertex == end:

            return path

        # Перевіряємо чи не пройдений вже елемент

        elif vertex not in visited:

            # отримуємо нові вершини і додаємо до черги/ отримуємо по ключу

            for current\_neighbour in graph\_to\_search.get(vertex, []):

                print(current\_neighbour)

                new\_path = list(path)

                print(new\_path)

                new\_path.append(current\_neighbour)

                queue.append(new\_path)

                print(f'queue {queue}')

            # Додаємо до відвіданого

            visited.add(vertex)

# Recursive search direct in depth

def dfs\_matrix(matrix: list, startRow, endPoint, visited: set, find: bool, way: set):

    if find == False:

        #  Проходимо вершини графаs

        for node\_index, node\_list in enumerate(matrix):

            # Перевіяємо чи не знайдено вершину

            if startRow == endPoint:

                print(f'Знайдено точку пошуку {startRow}')

                way.add(endPoint)

                # print(visited)

                print(f'Шлях {way}')

                find = True

                exit()

            # Перевірка досягнення стартової вершини

            if (node\_index + 1) == startRow:

                visited.add(node\_index + 1)  # Помічаємо вершину як пройдену

                way.add(node\_index + 1)

                edge\_number = 0

                ind\_list = list()

                count\_end = 0

                count\_edge = 0

                # Перевіряємо індекси і їх значення в кожній вершині

                for index, edge in enumerate(node\_list):

                    if edge == 1:

                        edge\_number = edge\_number + 1

                        ind\_list.append(index + 1)

                        print(

                            f'Вузол {node\_index + 1} з’єднаний з {index + 1}')

                        if not (index + 1) in visited:

                            count\_edge += 1

                            # Запускаємо рекурсію

                            # Повертає 1 у разі кінечної вершини

                            if (dfs\_matrix(matrix, index + 1, endPoint, visited, find, way)) == 1:

                                count\_end += 1  # Сумуємо кількість кінечних вершин в проміжних вершинах

                if (count\_end == count\_edge):  # Якщо кількість не пройдених вершин рівна кінечним вершинам

                    # Вузол теж кінечний, видаляємо зі шляху

                    way.remove(node\_index + 1)

                    return 1

                # if edge\_number == 1 and (node\_index + 1) != endPoint:

                #     way.remove(node\_index + 1)

                #     return 1

def search\_bfs(start, graph, search\_queue, visited, end, middle\_list: list):

    search\_queue += graph[start]

    while search\_queue:

        # print(f'Search queue {search\_queue}')

        node = search\_queue.popleft()

        # print(f'Get node {node}')

        k = 0

        if not node in visited:

            # Перевірямо середні вершини, у разі пройдення зупиняємо

            for i in middle\_list:

                if node == i:

                    k += 1

                    print(f'{threading.current\_thread()} ,Middle tree {node}')

                    if k == 2:

                        return

            else:

                search\_queue += graph[node]

                visited += [node]

# Двонаправлений пошук

def bidirectional\_search\_bfs\_based(startPoint, endPoint, graph, middle\_list: list):

    # Створюємо черги

    searchQueue\_direct = deque()

    searchQueue\_backward = deque()

    visited\_dir = []

    visited\_back = []

    # Створюємо потоки для кожного направлення

    thread\_dir = threading.Thread(target=search\_bfs,

                                  args=(startPoint, graph, searchQueue\_direct, visited\_dir, endPoint, middle\_list,))

    thread\_back = threading.Thread(target=search\_bfs,

                                   args=(endPoint, graph, searchQueue\_backward, visited\_back, startPoint, middle\_list,))

    # Запускаємо потоки

    thread\_dir.start()

    thread\_back.start()

    # Перевіряємо списки на середні вершини

    while True:

        time.sleep(1)

        for i in middle\_list:

            if i in visited\_dir and i in visited\_back:

                print("Done")

                return

if (\_\_name\_\_ == "\_\_main\_\_"):

    dfs\_matrix(Adjacency\_Matrix, 1, 13, set(), False, set())

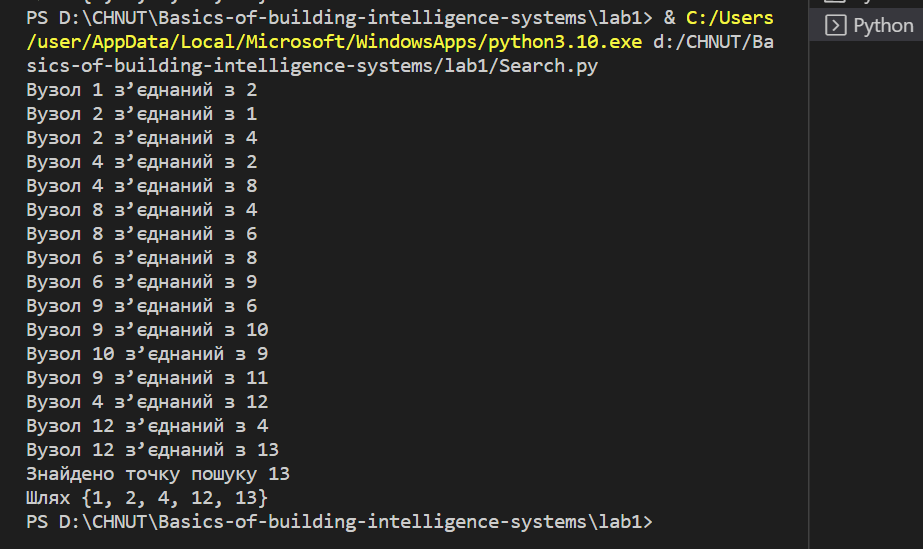


Рисунок 1.2 – Результат виконання програми

Висновок: на даній лабораторній роботі я отримати теоретичні та практичні навички по роботі з методами пошуку розв‘язків задач, розглянув метод глибокого пошуку.