

# BÀI 1: CÁC THUẬT TOÁN TÌM KIẾM: BFS, DFS VÀ UCS

## I. MỤC TIÊU:

Sau khi thực hành xong, sinh viên nắm được:

- Thuật toán tìm kiếm BFS, DFS và UCS.
- Cài đặt được các thuật toán này trên máy tính.

## II. TÓM TẮT LÝ THUYẾT:

### 1. Thuật toán BFS:

BFS (Breadth First Search) là thuật toán duyệt đồ thị ưu tiên chiều rộng để tìm kiếm đường đi ngắn nhất từ một đỉnh gốc tới tất cả các đỉnh khác.

★ **Ý tưởng:** tại mỗi bước chọn trạng thái để phát triển là trạng thái được sinh ra trước các trạng thái chờ phát triển khác. Danh sách  $L$  được sử lý như hàng đợi (Queue).

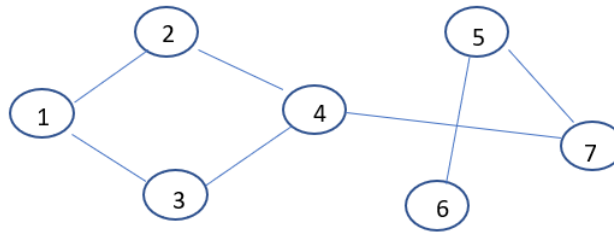
#### **Procedure** *Breadth\_Search*

##### **Begin**

1. Khởi tạo danh sách  $L$  chứa trạng thái ban đầu;
2. **While** (1)
  - 2.1 **if**  $L$  rỗng **then**  
{  
    Thông báo tìm kiếm thất bại;  
    **stop**;  
}
  - 2.2 Loại trạng thái  $u$  ở đầu danh sách  $L$ ;
  - 2.3 **if**  $u$  là trạng thái kết thúc **then**  
{  
    Thông báo tìm kiếm thành công;  
    **stop**;  
}
  - 2.4 Lấy các trạng thái  $v$  kề với  $u$  và thêm vào cuối danh sách  $L$ ;  
    **for** mỗi trạng thái  $v$  kề  $u$  **do**  
         $\text{father}(v) = u$ ;

**end**

★ **Ví dụ:** Tìm đường đi từ đỉnh 1 tới đỉnh 7



Các bước thực hiện của thuật toán BFS:

1.  $L = [1]$  (trạng thái ban đầu)
2.  $Node = 1, L = [2, 3], father[2, 3] = 1$
3.  $Node = 2, L = [3, 4], father[4] = 2$
4.  $Node = 3, L = [4]$  (đỉnh 4 kề với đỉnh 3 nhưng đã tồn tại trong  $L$  nên không thêm vào),  $father[4] = 3$ .
5.  $Node = 4, L = [7], father[7] = 4$ .
6.  $Node = 7$  (trạng thái kết thúc)  $\rightarrow$  dừng.

$\Rightarrow$  Đường đi từ đỉnh 1 tới đỉnh 7 là:  $1 \rightarrow 2 \rightarrow 4 \rightarrow 7$  hoặc  $1 \rightarrow 3 \rightarrow 4 \rightarrow 7$ .

## 2. Thuật toán DFS:

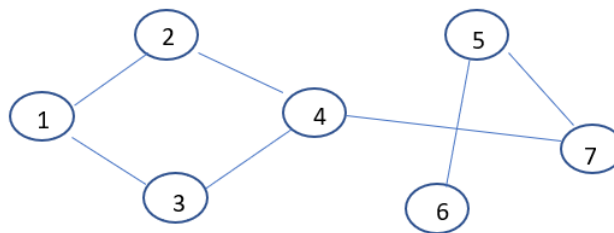
DFS (Depth First Search) là thuật toán tìm kiếm theo chiều sâu xuất phát từ một đỉnh gốc và đi xa nhất có thể theo một nhánh trước khi quay lui.

★ **Ý tưởng:** tại mỗi bước trạng thái được chọn để phát triển là trạng thái được sinh ra sau cùng trong số các trạng thái chờ phát triển. Danh sách  $L$  được xử lý như ngăn xếp (**Stack**).

**Procedure** *Depth\_Search***Begin**

1. Khởi tạo danh sách L chứa trạng thái ban đầu;
  2. **While** (1)
    - 2.1 **if** L rỗng **then**  
{  
    Thông báo tìm kiếm thất bại;  
    **stop**;  
}
    - 2.2 Loại trạng thái u ở đầu danh sách L;
    - 2.3 **if** u là trạng thái kết thúc **then**  
{  
    Thông báo tìm kiếm thành công;  
    **stop**;  
}
    - 2.4 Lấy các trạng thái v kề với u và thêm vào đầu danh sách L;  
    **for** mỗi trạng thái v kề u **do**  
        father(v) = u;
- end**

★ **Ví dụ:** Tìm đường đi từ đỉnh 1 tới đỉnh 7



Các bước thực hiện của thuật toán DFS:

1.  $L = [1]$
2.  $Node = 1, L = [3, 2], father[3, 2] = 1$
3.  $Node = 3, L = [4, 2], father[4] = 3$
4.  $Node = 4, L = [7, 2], father[7] = 4$
5.  $Node = 7$  (trạng thái kết thúc)  $\rightarrow$  dừng.

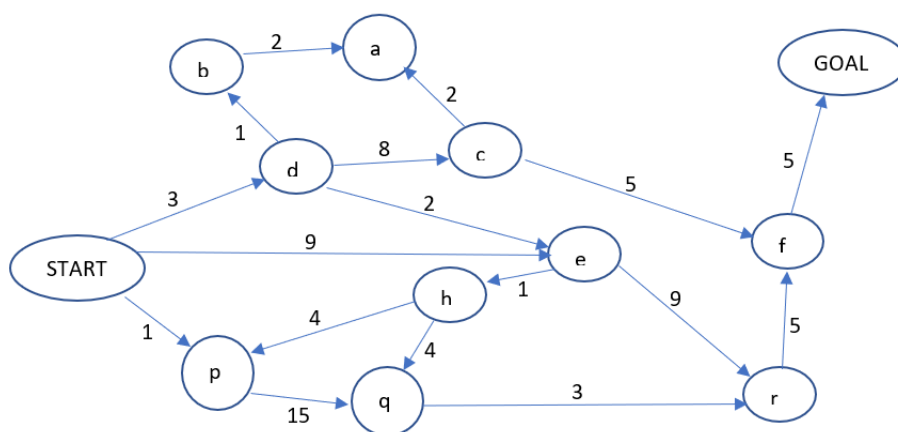
⇒ Đường đi từ đỉnh 1 tới đỉnh 7 là:  $1 \rightarrow 3 \rightarrow 4 \rightarrow 7$ .

### 3. Thuật toán UCS:

★ **Ý tưởng:** Thuật toán UCS là một thuật toán tìm kiếm trên một cấu trúc cây hoặc đồ thị có trọng số (chi phí). Việc tìm kiếm bắt đầu tại nút gốc và tiếp tục bằng cách duyệt các nút tiếp theo với trọng số hay chi phí thấp nhất tính từ nút gốc. UCS sử dụng một hàng đợi ưu tiên (**Priority Queue – PQ**) để lưu trữ và duyệt các trạng thái trên đường đi.

```
function Tìm_kiểm_UCS(bài_toán, ngăn_chứa) return lời_giải hoặc thất_bại.  
ngăn_chứa ← Tạo_Hàng_Đội_Rỗng()  
ngăn_chứa ← Thêm(TẠO_NÚT(Trạng_Thái_Đầu[bài_toán]), ngăn_chứa)  
loop do  
    if Là_Rỗng(ngăn_chứa) then return thất_bại.  
    nút ← Lấy_Chi_phí_Nhỏ_nhất(ngăn_chứa)  
    if Kiểm_tra_Câu_hỏi_đích[bài_toán] trên Trạng_thái[nút] đúng.  
        then return Lời_giải(nút).  
lg ← Mở(nút, bài_toán) //lg tập các nút con mới  
ngăn_chứa ← Thêm_Tất_cả(lg, ngăn_chứa)
```

★ **Ví dụ:** Cho đồ thị như hình



Các bước thực hiện của thuật toán Tìm\_kiểm\_UCS:

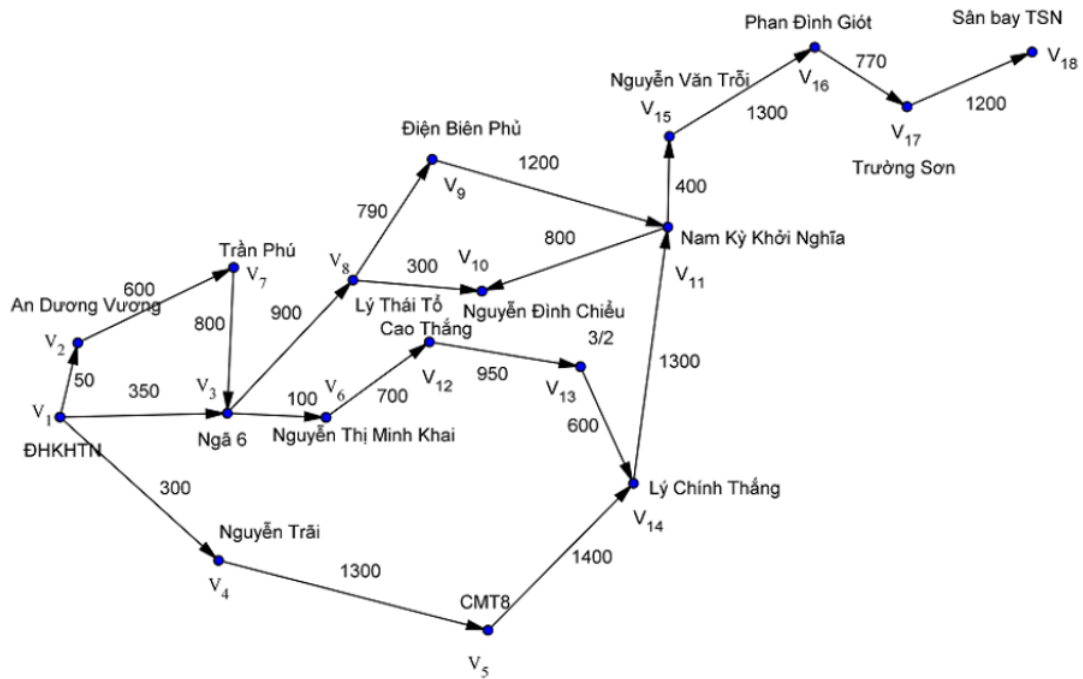
1.  $PQ = \{(START, 0)\}$ . (PQ là Priority Queue)
2.  $PQ = \{(p, 1), (d, 3), (e, 9)\}$

3.  $PQ = \{(d, 3), (e, 9), (q, 16)\}$
4.  $PQ = \{(b, 4), (e, 5), (c, 11), (q, 16)\}$
5.  $PQ = \{(e, 5), (a, 6), (c, 11), (q, 16)\}$
6.  $PQ = \{(a, 6), (h, 6), (c, 11), (r, 14), (q, 16)\}$
7.  $PQ = \{(h, 6), (c, 11), (r, 14), (q, 16)\}$
8.  $PQ = \{(q, 10), (c, 11), (r, 14)\}$
9.  $PQ = \{(c, 11), (r, 13)\}$
10.  $PQ = \{(r, 13), (f, 16)\}$
11.  $PQ = \{(f, 16)\}$
12.  $PQ = \{(GOAL, 21)\}$

$\Rightarrow$  Đường đi ngắn nhất từ START tới GOAL là:  $START \rightarrow d \rightarrow c \rightarrow f \rightarrow GOAL$   
với chi phí là 21.

### III. NỘI DUNG THỰC HÀNH:

Cho đồ thị như hình vẽ bên dưới Tìm đường đi ngắn nhất từ trường Đại học Khoa học



Tự nhiên ( $V_1$ ) tới sân bay Tân Sơn Nhất ( $V_{18}$ ) dùng các thuật toán sau:

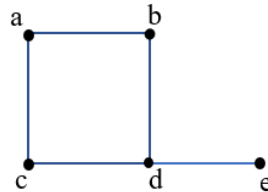
- BFS
- DFS
- UCS

## 1. Graph, Queue, Stack và Priority Queue trong Python:

### a. Graph:

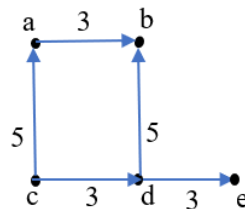
**Đồ thị**  $G = (V, E)$  trong Python được biểu diễn như là kiểu dữ liệu từ điển (dictionary).

**Ví dụ 1:** Cho đồ thị vô hướng  $G = (V, E)$  như hình. Ta có:  $V = \{a, b, c, d, e\}$ ,  $E = \{ab, ac, bd, cd, de\}$ .



```
graph = {
    'a': ['b', 'c'],
    'b': ['a', 'd'],
    'c': ['a', 'd'],
    'd': ['b', 'c', 'e'],
    'e': ['d']}
print(graph)
{'a': ['b', 'c'], 'b': ['a', 'd'], 'c': ['a', 'd'], 'd': ['b', 'c', 'e'], 'e': ['d']}
graph['a']
['b', 'c']
```

**Ví dụ 2:** Cho đồ thị có hướng  $G = (V, E)$  có trọng số như hình.



```
graph = {
    'a': {'b': 3},
    'b': {},
    'c': {'a': 5, 'd': 3},
    'd': {'b': 5, 'e': 3},
    'e': {}
}
print(graph)
{'a': {'b': 3}, 'b': {}, 'c': {'a': 5, 'd': 3}, 'd': {'b': 5, 'e': 3}, 'e': {}}
graph['a']['b']
3
```

**b. Stack:** phần tử cuối cùng thêm vào là phần tử bị loại ra đầu tiên (Last In First Out). Để thêm một phần tử vào stack, ta sử dụng `append()`. Để loại bỏ 1 phần tử của stack, ta sử dụng `pop()`.

c. **Queue:** phần tử đầu tiên thêm vào là phần tử đầu tiên bị loại (First In First Out).

Để thêm 1 phần tử vào queue, ta sử dụng `put()`. Để loại bỏ 1 phần tử của queue, ta sử dụng `get()`.

d. **Priority Queue:** là cấu trúc dữ liệu lưu trữ các phần tử cùng với độ ưu tiên của nó và khi lấy phần tử ra khỏi hàng đợi sẽ căn cứ vào độ ưu tiên nhỏ nhất.

## 2. Dữ liệu đầu vào và dữ liệu đầu ra:

### - INPUT:

❖ Dòng 1: Số node trên đồ thị

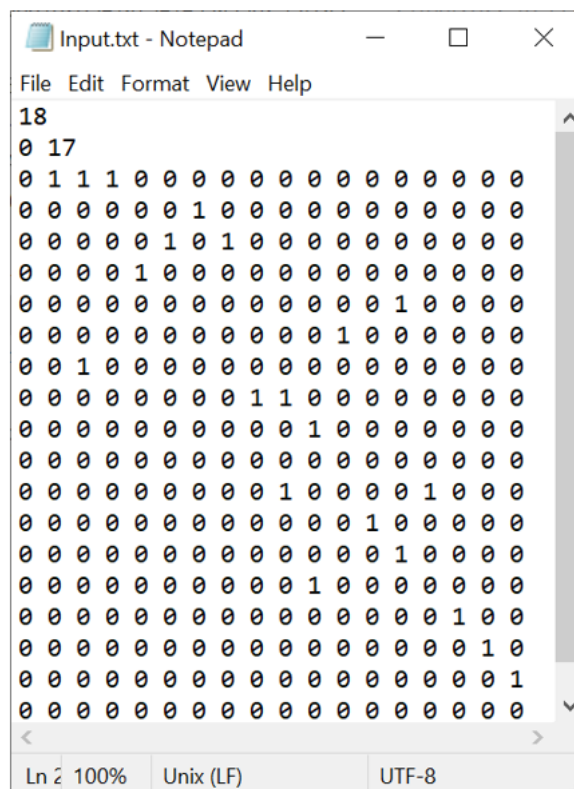
❖ Dòng 2: node xuất phát và node đích

❖ Những dòng tiếp theo: ma trận kề  $M$  của đồ thị với quy ước:

◆  $M[i][j] = 1$ : có đường nối trực tiếp từ  $i$  tới  $j$  ( $M[i][j] = w$ : có đường nối trực tiếp từ  $i$  đến  $j$  với chi phí là  $w$  ( $w > 0$ ) cho thuật toán UCS).

◆  $M[i][j] = 0$ : không có đường nối trực tiếp từ  $i$  tới  $j$ .

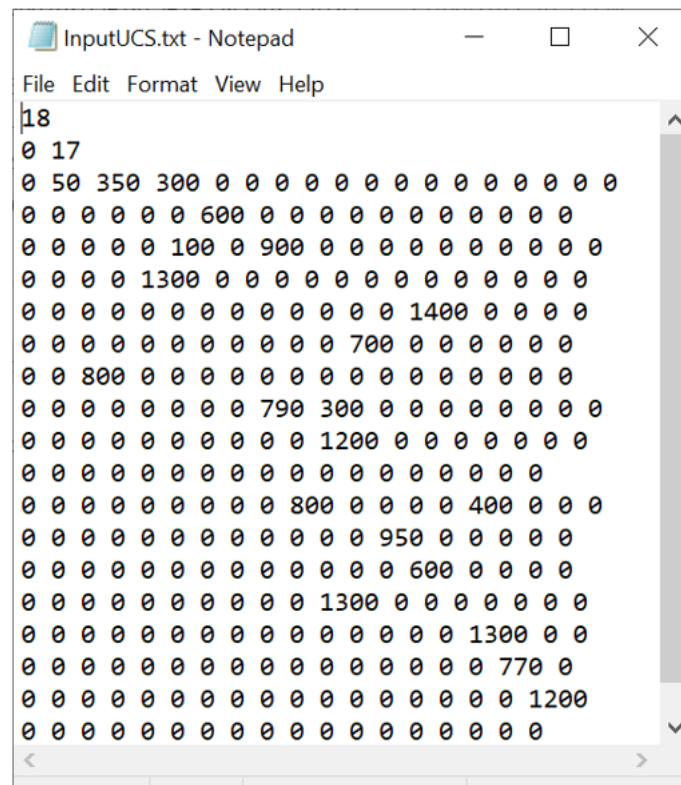
❁ Dữ liệu cho BFS và DFS



```
Input.txt - Notepad
File Edit Format View Help
18
0 17
0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
```



## ❁ Dữ liệu cho UCS



⇒ lưu 2 file như hình với định dạng \*.txt ⇒ đọc dữ liệu từ file \*.txt

## ❁ Đọc dữ liệu từ file trong Python:

**open (file, mode)**

trong đó:

✧ **file**: đường dẫn và tên của file

✧ **mode**: “r” – mở file để đọc (read), “w” – mở file để viết (write)

**read()**: trả về 1 chuỗi

**readline()**: trả về 1 dòng

**readlines()**: trả về danh sách các dòng

**write()**: viết 1 chuỗi vào file

**writelines()**: viết danh sách các chuỗi vào file

**close()**: đóng file

**string.split(separator, maxsplit)**: chuyển chuỗi thành list (separator: dấu ngăn cách để tách chuỗi (mặc định là khoảng trắng), maxsplit: mặc định là -1 (tất cả các lần xuất hiện))

⇒ xây dựng Graph ???

```

from collections import defaultdict
from queue import Queue, PriorityQueue

#đọc dữ liệu từ file txt
def read_txt(file):
    size = int(file.readline())
    start, goal = [int(num) for num in file.readline().split(' ')]
    matrix = [[int(num) for num in line.split(' ')] for line in file]
    return size,start,goal,matrix
#chuyển ma trận kề thành danh sách kề
def convert_graph(a):
    adjList=defaultdict(list)
    for i in range (len(a)):
        for j in range(len(a[i])):
            if a[i][j]==1:
                adjList[i].append(j)
    return adjList

def convert_graph_weight(a):
    adjList = defaultdict(list)
    for i in range (len(a)):
        for j in range(len(a[i])):
            if a[i][j]!= 0:
                adjList[i].append((j,a[i][j]))
    return adjList

```

## - OUTPUT:

- ❖ Nếu tồn tại đường đi: xuất ra màn hình thứ tự đường đi từ  $V_1$  tới  $V_{18}$ .
- ❖ Nếu không tồn tại đường đi: thông báo không có đường đi.

```

if __name__ == "__main__":
    # Đọc file Input.txt và InputUCS.txt
    file_1 = open("Input.txt","r")
    file_2 = open("InputUCS.txt","r")
    size_1, start_1,goal_1,matrix_1 = read_txt(file_1)
    size_2, start_2,goal_2,matrix_2 = read_txt(file_2)
    file_1.close()
    file_2.close()
    graph_1 = convert_graph(matrix_1)
    graph_2 = convert_graph_weight(matrix_2)

    # Thực thi thuật toán BFS
    result_bfs = BFS(graph_1,start_1,goal_1)
    print("Kết quả sử dụng thuật toán BFS: \n",result_bfs)
    #Thực thi thuật toán DFS
    result_dfs = DFS(graph_1,start_1,goal_1)
    print("Kết quả sử dụng thuật toán DFS: \n",result_dfs)
    #Thực thi thuật toán UCS
    cost, result_ucs = UCS(graph_2,start_2,goal_2)
    print("Kết quả sử dụng thuật toán UCS: \n",result_ucs, "với tổng chi phí là", cost)

```

### 3. Cài đặt thuật toán BFS:

```
def BFS(graph, start, end):  
  
    visited = []  
    frontier = Queue()  
  
    #thêm node start vào frontier và visited  
    frontier.put(start)  
    visited.append(start)  
  
    #start không có node cha  
    parent = dict()  
    parent[start] = None  
  
    path_found = False  
  
    while True:  
        if frontier.empty():  
            raise Exception("No way Exception")  
        current_node = frontier.get()  
        visited.append(current_node)  
  
        # Kiểm tra current_node có là end hay không  
        if current_node == end:  
            path_found = True  
            break  
  
        for node in graph[current_node]:  
            if node not in visited:  
                frontier.put(node)  
                parent[node] = current_node  
                visited.append(node)  
  
    # Xây dựng đường đi  
    path = []  
    if path_found:  
        path.append(end)  
        while parent[end] is not None:  
            path.append(parent[end])  
            end = parent[end]  
        path.reverse()  
    return path
```

#### 4. Cài đặt thuật toán DFS:

```
def DFS(graph, start, end):
    visited = []
    frontier = []

    #thêm node start vào frontier và visited
    frontier.append(start)
    visited.append(start)

    #start không có node cha
    parent = dict()
    parent[start] = None

    path_found = False

    while True:
        if frontier == []:
            raise Exception("No way Exception")
        current_node = frontier.pop()
        visited.append(current_node)

        # Kiểm tra current_node có là end hay không
        if current_node == end:
            path_found = True
            break

        for node in graph[current_node]:
            if node not in visited:
                frontier.append(node)
                parent[node] = current_node
                visited.append(node)

    # Xây dựng đường đi
    path = []
    if path_found:
        path.append(end)
        while parent[end] is not None:
            path.append(parent[end])
            end = parent[end]
        path.reverse()
    return path
```

## 5. Cài đặt thuật toán UCS:

```
def UCS(graph, start, end):  
    visited = []  
    frontier = PriorityQueue()  
    #thêm node start vào frontier và visited  
    frontier.put((0, start))  
    visited.append(start)  
    #start không có node cha  
    parent = dict()  
    parent[start] = None  
    path_found = False  
    while True:  
        if frontier.empty():  
            raise Exception("No way Exception")  
        current_w, current_node = frontier.get()  
        visited.append(current_node)  
  
        # Kiểm tra current_node có là end hay không  
        if current_node == end:  
            path_found = True  
            break  
  
        for nodei in graph[current_node]:  
            node, weight = nodei  
            if node not in visited:  
                frontier.put((current_w + weight, node))  
                parent[node] = current_node  
                visited.append(node)  
  
    # Xây dựng đường đi  
    path = []  
    if path_found:  
        path.append(end)  
        while parent[end] is not None:  
            path.append(parent[end])  
            end = parent[end]  
        path.reverse()  
    return current_w, path
```

## 6. Yêu cầu:

- Cài đặt và thực thi chương trình. Nếu chương trình bị báo lỗi thì lỗi ở dòng nào và sửa lại như thế nào?
- Viết báo cáo trình bày:
  - ✿ Chạy tay thuật toán BFS, DFS và UCS.
  - ✿ Kiểm tra tính đúng đắn của các thuật toán đã cho sẵn code như trên. Nếu chưa đúng thì em sửa lại như thế nào cho phù hợp?
  - ✿ Từ đó, em có nhận xét gì về kết quả chạy tay với kết quả chạy trên máy tính.