

TPHCM, ngày 24 tháng 10 năm 2024

**BÁO CÁO THỰC HÀNH – NHẬP MÔN TRÍ TUỆ NHÂN TẠO – LẦN 1**

Họ và tên: Nguyễn Thanh Kiên.  
MSSV: 22110092.

**I. Nhắc lại Lý thuyết.**

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

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.

```

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

```

UCS: 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.

```

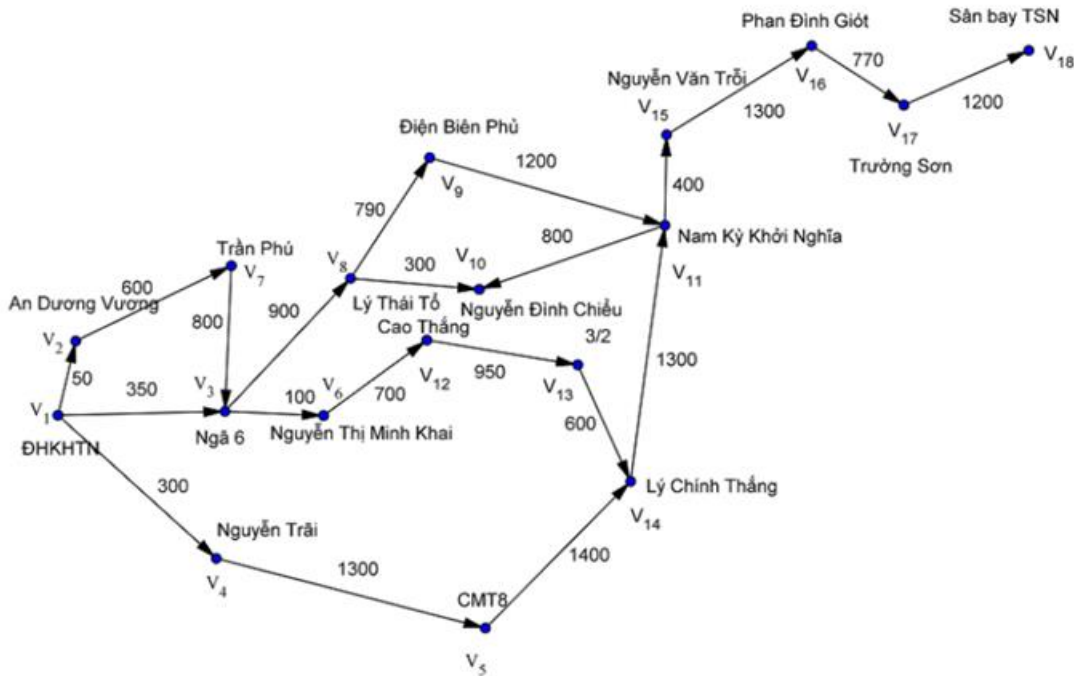
function Tìm_kiểm_UCS(bài_toán, ngăn_chứa) return lời_giải hoặc thất_bại.
ngăn_chứa ← Tạo_Hàng_Đội_Rỗng()
ngăn_chứa ← Thêm(TAO_NÚT(Trạng_Thái_Đầu[bài_toán]), ngăn_chứa)
loop do
  if Là_Rỗng(ngăn_chứa) then return thất_bại.
  nút ← Lấy_Chi_phí_Nhỏ_nhất(ngăn_chứa)
  if Kiểm_tra_Câu_hỏi_đích[bài_toán] trên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

```

## II. Thực hiện bài tập.

### 1. Chạy tay thuật toán BFS, DFS và UCS.

Cho đồ thị như hình vẽ bên dưới Tìm đường đi ngắn nhất từ trường Đại học Khoa học Tự nhiên (V1) tới sân bay Tân Sơn Nhất (V18) dùng các thuật toán sau: BFS – DFS -UCS.



#### . BFS

1.  $L = [V_1]$  (trạng thái ban đầu).
2.  $Node = V_1, L = [V_2, V_3, V_4], father[V_2, V_3, V_4] = V_1$ .
3.  $Node = V_2, L = [V_3, V_4, V_7], father[V_7] = V_2$ .
4.  $Node = V_3, L = [V_4, V_7, V_6, V_8], father[V_6, V_8] = V_3$ .
5.  $Node = V_4, L = [V_7, V_6, V_8, V_5], father[V_5] = V_4$ .
6.  $Node = V_7, L = [V_6, V_8, V_5]$ . ( $V_3$  kề với  $V_7$  nhưng đã tồn tại trong  $L$  nên không thêm vào),  $father[V_3] = V_7$ .
7.  $Node = V_6, L = [V_8, V_5, V_{12}], father[V_{12}] = V_6$ .
8.  $Node = V_8, L = [V_5, V_{12}, V_9, V_{10}], father[V_9, V_{10}] = V_8$ .
9.  $Node = V_5, L = [V_{12}, V_9, V_{10}, V_{14}], father[V_{14}] = V_5$ .
10.  $Node = V_{12}, L = [V_9, V_{10}, V_{14}, V_{13}], father[V_{13}] = V_{12}$ .
11.  $Node = V_9, L = [V_{10}, V_{14}, V_{13}, V_{11}], father[V_{11}] = V_9$ .

12.  $Node = V_{10}$ ,  $L = [V_{14}, V_{13}, V_{11}]$ ,  $father[V_{11}] = V_9$  (không có đường đi từ  $V_{10}$  tới bất kỳ cạnh nào khác,  $father$  vẫn giữ nguyên như cũ).
13.  $Node = V_{14}$ ,  $L = [V_{13}, V_{11}]$ . ( $V_{11}$  kề với  $V_{14}$  nhưng đã tồn tại trong  $L$  nên không thêm vào),  $father[V_{11}] = V_{14}$ .
14.  $Node = V_{13}$ ,  $L = [V_{11}]$ . ( $V_{14}$  kề với  $V_{13}$  nhưng đã tồn tại trong  $L$  nên không thêm vào),  $father[V_{14}] = V_{13}$ .
15.  $Node = V_{11}$ ,  $L = [V_{15}, V_{10}]$ ,  $father[V_{15}, V_{10}] = V_{11}$ .
16.  $Node = V_{15}$ ,  $L = [V_{10}, V_{16}]$ ,  $father[V_{16}] = V_{15}$ .
17.  $Node = V_{10}$ ,  $L = [V_{16}]$ ,  $father[V_{16}] = V_{15}$  (không có đường đi từ  $V_{10}$  tới bất kỳ cạnh nào khác,  $father$  vẫn giữ nguyên như cũ).
18.  $Node = V_{16}$ ,  $L = [V_{17}]$ ,  $father[V_{17}] = V_{16}$ .
19.  $Node = V_{17}$ ,  $L = [V_{18}]$ ,  $father[V_{18}] = V_{17}$ .
20.  $Node = V_{18}$  (trạng thái kết thúc)  $\rightarrow$  Dừng.

**Vậy đường đi từ đỉnh  $V_1$  tới đỉnh  $V_{18}$  là:**

Đường đi 1: $V_1 \rightarrow V_4 \rightarrow V_5 \rightarrow V_{14} \rightarrow V_{11} \rightarrow V_{15} \rightarrow V_{16} \rightarrow V_{17} \rightarrow V_{18}$ .	Đường đi 1: ĐHKHTN $\rightarrow$ Nguyễn Trãi $\rightarrow$ CMT8 $\rightarrow$ Lý Chính Thắng $\rightarrow$ Nam kỳ Khởi nghĩa $\rightarrow$ Nguyễn Văn Trỗi $\rightarrow$ Phan Đình Giót $\rightarrow$ Trường Sơn $\rightarrow$ Sân bay TSN.
Đường đi 2: $V_1 \rightarrow V_3 \rightarrow V_6 \rightarrow V_{12} \rightarrow V_{13} \rightarrow V_{14} \rightarrow V_{11} \rightarrow V_{15} \rightarrow V_{16} \rightarrow V_{17} \rightarrow V_{18}$ .	Đường đi 2: ĐHKHTN $\rightarrow$ Ngã 6 $\rightarrow$ Nguyễn Thị Minh Khai $\rightarrow$ Cao Thắng $\rightarrow$ 3/2 $\rightarrow$ Lý Chính Thắng $\rightarrow$ Nam kỳ Khởi nghĩa $\rightarrow$ Nguyễn Văn Trỗi $\rightarrow$ Phan Đình Giót $\rightarrow$ Trường Sơn $\rightarrow$ Sân bay TSN.
Đường đi 3: $V_1 \rightarrow V_2 \rightarrow V_7 \rightarrow V_3 \rightarrow V_8 \rightarrow V_9 \rightarrow V_{11} \rightarrow V_{15} \rightarrow V_{16} \rightarrow V_{17} \rightarrow V_{18}$ .	Đường đi 3: ĐHKHTN $\rightarrow$ An Dương Vương $\rightarrow$ Trần Phú $\rightarrow$ Ngã 6 $\rightarrow$ Lý Thái Tổ $\rightarrow$ Điện Biên Phủ $\rightarrow$ Nam kỳ Khởi nghĩa $\rightarrow$ Nguyễn Văn Trỗi $\rightarrow$ Phan Đình Giót $\rightarrow$ Trường Sơn $\rightarrow$ Sân bay TSN.
Đường đi 4: $V_1 \rightarrow V_2 \rightarrow V_7 \rightarrow V_3 \rightarrow V_6 \rightarrow V_{12}$	Đường đi 4: ĐHKHTN $\rightarrow$ An Dương

-> $V_{13} \rightarrow V_{14} \rightarrow V_{11} \rightarrow V_{15} \rightarrow V_{16} \rightarrow V_{17} \rightarrow V_{18}$ .	Vương -> Trần Phú -> Ngã 6 -> Nguyễn Thị Minh Khai -> Cao Thắng -> 3/2 -> Lý Chính Thắng -> Nam kỳ Khởi nghĩa -> Nguyễn Văn Trỗi -> Phan Đình Giót -> Trường Sơn -> Sân bay TSN.
Đường đi 5: $V_1 \rightarrow V_3 \rightarrow V_8 \rightarrow V_9 \rightarrow V_{11} \rightarrow V_{15} \rightarrow V_{16} \rightarrow V_{17} \rightarrow V_{18}$ .	Đường đi 5: ĐHKHTN -> Ngã 6 -> Lý Thái Tổ -> Điện Biên Phủ -> Nam kỳ Khởi nghĩa -> Nguyễn Văn Trỗi -> Phan Đình Giót -> Trường Sơn -> Sân bay TSN.

### .DFS

1.  $L = [V_1]$  (trạng thái ban đầu).
2.  $Node = V_1, L = [V_4, V_3, V_2], father[V_4, V_3, V_2] = V_1$ .
3.  $Node = V_4, L = [V_5, V_3, V_2], father[V_5] = V_4$ .
4.  $Node = V_5, L = [V_{14}, V_3, V_2], father[V_{14}] = V_5$ .
5.  $Node = V_{14}, L = [V_{11}, V_3, V_2], father[V_{11}] = V_{14}$ .
6.  $Node = V_{11}, L = [V_{15}, V_3, V_2], father[V_{15}] = V_{11}$ .
7.  $Node = V_{15}, L = [V_{16}, V_3, V_2], father[V_{16}] = V_{15}$ .
8.  $Node = V_{16}, L = [V_{17}, V_3, V_2], father[V_{17}] = V_{16}$ .
9.  $Node = V_{17}, L = [V_{18}, V_3, V_2], father[V_{18}] = V_{17}$ .
10.  $Node = V_{18}$  (trạng thái kết thúc) -> Dừng.

**Vậy đường đi từ đỉnh  $V_1$  tới đỉnh  $V_{18}$  là:**

$V_1 \rightarrow V_4 \rightarrow V_5 \rightarrow V_{14} \rightarrow V_{11} \rightarrow V_{15} \rightarrow V_{16} \rightarrow V_{17} \rightarrow V_{18}$  hay ĐHKHTN -> Nguyễn Trãi -> CMT8 -> Lý Chính Thắng -> Nam kỳ Khởi nghĩa -> Nguyễn Văn Trỗi -> Phan Đình Giót -> Trường Sơn -> Sân bay TSN.

### . UCS

1.  $PQ = \{(V_1, 0)\}$ . (PQ là Priority Queue).
2.  $PQ = \{(V_2, 50), (V_3, 350), (V_4, 300)\}$ .
3.  $PQ = \{(V_3, 350), (V_4, 300), (V_7, 650)\}$ .
4.  $PQ = \{(V_4, 300), (V_6, 450), (V_7, 650), (V_8, 1250)\}$ .

5.  $PQ = \{(V_6, 450), (V_7, 650), (V_8, 1250), (V_5, 1600)\}$ .
6.  $PQ = \{(V_7, 650), (V_{12}, 1150), (V_8, 1250), (V_5, 1600)\}$ .
7.  $PQ = \{(V_8, 1250), (V_5, 1600), (V_{12}, 2110)\}$ .
8.  $PQ = \{(V_{10}, 1550), (V_5, 1600), \{(V_9, 2040), (V_{12}, 2110)\}\}$ .
9.  $PQ = \{(V_5, 1600), (V_9, 2040), (V_{12}, 2110)\}$ .
10.  $PQ = \{(V_9, 2040), (V_{12}, 2110), (V_{14}, 3000)\}$ .
11.  $PQ = \{(V_{12}, 2110), (V_{14}, 3000), (V_{11}, 3240)\}$ .
12.  $PQ = \{(V_{14}, 3000), (V_{13}, 3060), (V_{11}, 3240)\}$ .
13.  $PQ = \{(V_{14}, 3000), (V_{13}, 3060), (V_{11}, 3240)\}$ .
14.  $PQ = \{(V_{13}, 3060), (V_{11}, 3240)\}$ .
15.  $PQ = \{(V_{11}, 3240)\}$ .
16.  $PQ = \{(V_{15}, 3640)\}$ .
17.  $PQ = \{(V_{16}, 4940)\}$ .
18.  $PQ = \{(V_{17}, 5710)\}$ .
19.  $PQ = \{(V_{18}, 6910)\}$ .

Vậy đường đi ngắn nhất từ  $V_1$  tới  $V_{18}$  hay đường đi ngắn nhất từ ĐHKHTN tới Sân Bay TSN là:  $V_1 \rightarrow V_3 \rightarrow V_8 \rightarrow V_9 \rightarrow V_{11} \rightarrow V_{15} \rightarrow V_{16} \rightarrow V_{17} \rightarrow V_{18}$  hay ĐHKHTN  $\rightarrow$  Ngã 6  $\rightarrow$  Lý Thái Tổ  $\rightarrow$  Điện Biên Phủ  $\rightarrow$  Nam kỳ Khởi nghĩa  $\rightarrow$  Nguyễn Văn Trỗi  $\rightarrow$  Phan Đình Giót  $\rightarrow$  Trường Sơn  $\rightarrow$  Sân bay TSN với chi phí là 6910 (đvtt).

## 2. Tính đúng đắn của các thuật toán đã cho sẵn code như trên.

### a. Tài nguyên:

Đồ thị vô hướng:

```
graph = {
    'V1': ['V2', 'V3', 'V4'],
    'V2': ['V1', 'V7'],
    'V3': ['V1', 'V7', 'V8', 'V6'],
    'V4': ['V1', 'V5'],
    'V5': ['V4', 'V14'],
    'V6': ['V3', 'V12'],
    'V7': ['V2', 'V3'],
    'V8': ['V3', 'V9', 'V10'],
    'V9': ['V8', 'V11'],
    'V10': ['V8', 'V11'],
    'V11': ['V10', 'V15'],
    'V12': ['V6', 'V13'],
    'V13': ['V12', 'V14'],
    'V14': ['V5', 'V11'],
    'V15': ['V11', 'V16'],
    'V16': ['V15', 'V17'],
    'V17': ['V16', 'V18'],
    'V18': ['V17']}
print(graph)
```

Tuy vậy, đồ thị từ Trường ĐHKHTN tới Sân bay TSN lại là đồ thị có hướng và trọng số. Vì vậy, ta sẽ dùng đồ thị có hướng và trọng số = 0 để sử dụng cho 2 thuật toán BFS và DFS.

```
# BFS + DFS (no pathcost)
graphBFSDFS = {
    'V1': {'V2': 0, 'V3': 0, 'V4': 0},
    'V2': {'V7': 0},
    'V3': {'V8': 0, 'V6': 0},
    'V4': {'V5': 0},
    'V5': {'V14': 0},
    'V6': {'V12': 0},
    'V7': {'V3': 0},
    'V8': {'V9': 0, 'V10': 0},
    'V9': {'V11': 0},
    'V11': {'V10': 0, 'V15': 0},
    'V12': {'V13': 0},
    'V13': {'V14': 0},
    'V14': {'V11': 0},
    'V15': {'V16': 0},
    'V16': {'V17': 0},
    'V17': {'V18': 0},
    'V18': {}
}
print(graphBFSDFS)
```

```
{'V1': {'V2': 0, 'V3': 0, 'V4': 0}, 'V2': {'V7': 0}, 'V3': {'V8': 0, 'V6': 0}, 'V4': {'V5': 0}, 'V5': {'V14': 0}, 'V6': {'V12': 0}, 'V7': {'V3': 0}, 'V8': {'V9': 0, 'V10': 0}, 'V9': {'V11': 0}, 'V11': {'V10': 0, 'V15': 0}, 'V12': {'V13': 0}, 'V13': {'V14': 0}, 'V14': {'V11': 0}, 'V15': {'V16': 0}, 'V16': {'V17': 0}, 'V17': {'V18': 0}, 'V18': {}}
```

Và đồ thị có hướng, có trọng số để sử dụng cho UCS là:

```
graphUCS = {
    'V1': {'V2': 50, 'V3': 350, 'V4': 300},
    'V2': {'V7': 600},
    'V3': {'V8': 900, 'V6': 100},
    'V4': {'V5': 1300},
    'V5': {'V14': 1400},
    'V6': {'V12': 700},
    'V7': {'V3': 800},
    'V8': {'V9': 790, 'V10': 300},
    'V9': {'V11': 1200},
    'V11': {'V10': 800, 'V15': 400},
    'V12': {'V13': 950},
    'V13': {'V14': 600},
    'V14': {'V11': 1300},
    'V15': {'V16': 1300},
    'V16': {'V17': 770},
    'V17': {'V18': 1200},
    'V18': {}
}
print(graphUCS)
```

```
{'V1': {'V2': 50, 'V3': 350, 'V4': 300}, 'V2': {'V7': 600}, 'V3': {'V8': 900, 'V6': 100}, 'V4': {'V5': 1300}, 'V5': {'V14': 1400}, 'V6': {'V12': 700}, 'V7': {'V3': 800}, 'V8': {'V9': 790, 'V10': 300}, 'V9': {'V11': 1200}, 'V11': {'V10': 800, 'V15': 400}, 'V12': {'V13': 950}, 'V13': {'V14': 600}, 'V14': {'V11': 1300}, 'V15': {'V16': 1300}, 'V16': {'V17': 770}, 'V17': {'V18': 1200}, 'V18': {}}
```

Dữ liệu txt cho BFS và DFS

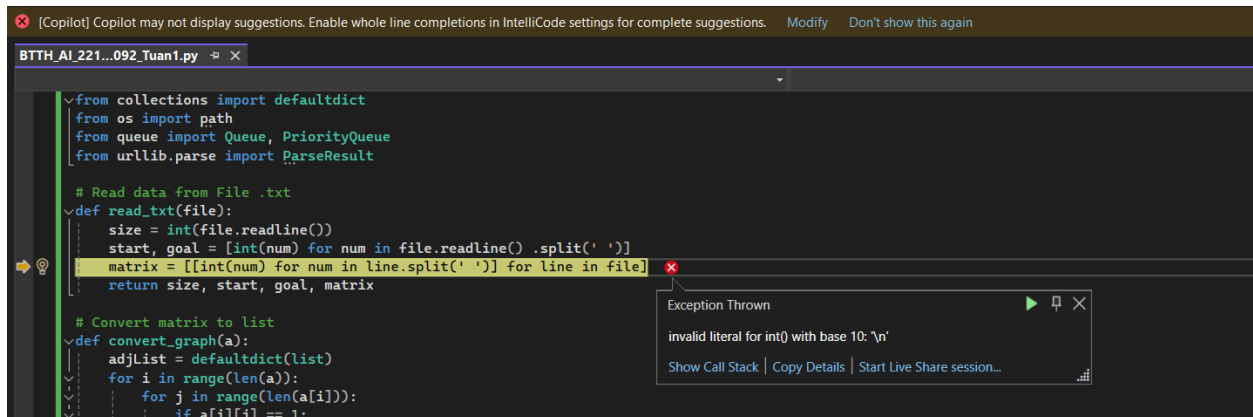
```
18
0 17
0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0
0 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 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 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 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
```

Dữ liệu txt cho DFS.

```
18
0 17
0 50 350 300 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 600 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 100 0 900 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1300 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 1400 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 700 0 0 0 0 0 0 0
0 800 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 790 300 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1200 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
0 0 0 0 0 0 800 0 0 0 0 400 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 950 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 600 0 0 0 0 0
0 0 0 0 0 0 0 0 1300 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1300 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 770 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1200 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

## b. Cài đặt và chạy chương trình.

Sau khi đã cài đặt chương trình giống như trong file mẫu của giáo viên hướng dẫn, chương trình không chạy được và bị báo lỗi ở line 10.

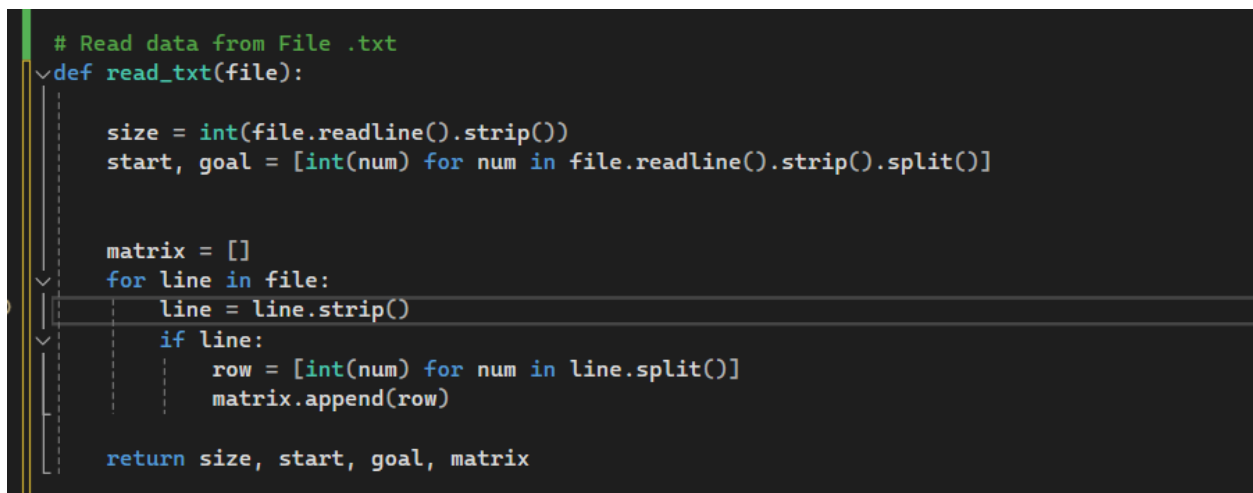


```
from collections import defaultdict
from os import path
from queue import Queue, PriorityQueue
from urllib.parse import ParseResult

# Read data from 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

# Convert matrix to list
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:
```

Ở đây, đã có 1 Exception đã được Throw ra: invalid literal for int() with base 10: '\n' Điều này có nghĩa là, có một dòng trống hoặc dòng có ký tự xuống dòng đã được đọc, và python không thể chuyển đổi nó thành số nguyên. Vì vậy, ta phải sử dụng cấu trúc .strip() để loại bỏ khoảng trắng và kiểm tra dòng trước khi chuyển đổi thành số nguyên



```
# Read data from File .txt
def read_txt(file):

    size = int(file.readline().strip())
    start, goal = [int(num) for num in file.readline().strip().split()]

    matrix = []
    for line in file:
        line = line.strip()
        if line:
            row = [int(num) for num in line.split()]
            matrix.append(row)

    return size, start, goal, matrix
```

ở đây:

line = line.strip(): loại bỏ khoảng trắng ở đầu dòng và cuối dòng.

Điều kiện if line: chỉ xử lý dòng nếu không rỗng.



Như vậy ta đã hoàn thành việc đọc data.

Các phần chạy chương trình đều không xuất hiện lỗi.

### Tính đúng đắn của các thuật toán:

1. BFS là thuật toán tìm kiếm theo chiều rộng, phù hợp với việc tìm các đường đi không đảm bảo ngắn nhất trong đồ thị phi trọng số hay đồ thị không có chi phí. Trong đoạn code, BFS sử dụng 1 hàng đợi Queue cho các Node chờ duyệt với một từ điển Parent để theo dõi cha của mỗi Node trên đường đi. Tuy nhiên, đoạn mã hiện tại đang thêm các node vào Visited trước khi kiểm tra, nên mỗi Node đưa vào hàng đợi nhiều lần và duyệt qua nhiều lần. Vì vậy, ta loại bỏ dòng `visited.append(current_node)` trong vòng lặp `while` và chỉ thêm node vào `visited` khi duyệt các Node con, giúp tránh trùng lặp và tối ưu bài toán.
2. DFS là thuật toán tìm kiếm theo chiều sâu, cũng không đảm bảo tìm thấy đường đi ngắn nhất như BFS. Trong đoạn code, BFS sử dụng một ngăn xếp frontier cho các Node chờ duyệt và cũng sử dụng từ điển Parent giúp xây dựng đường đi. Giống như BFS, ta có thể loại bỏ dòng `visited.append(current_node)` trong vòng lặp `While` và chỉ thêm vào các node con để tránh trùng lặp và tối ưu bài toán.
3. UCS là thuật toán tìm kiếm cho đồ thị có trọng số, dùng để tìm kiếm đường đi có chi phí thấp nhất, với việc sử dụng PriorityQueue cho các Node chờ duyệt và parent cùng `cost_so_far` để lưu trữ chi phí thấp nhất cho mỗi node đến thời điểm được duyệt. Visited không cần thiết trong đoạn code này vì `cost_so_far` sẽ đảm bảo chỉ thêm node nếu nó có chi phí thấp hơn chi phí đã ghi nhận.
4. Ta có thể xây dựng hàm để tối ưu code với hàm `build_path` giúp tìm kiếm đường đi từ node Start đến Node End dựa vào từ điển parent.

```
def BFS(graph, start, end):
    frontier = Queue()
    frontier.put(start)
    parent = {start: None}

    while not frontier.empty():
        current_node = frontier.get()

        if current_node == end:
            break

        for neighbor in graph[current_node]:
            if neighbor not in parent:
                parent[neighbor] = current_node
                frontier.put(neighbor)

    return build_path(parent, start, end)

def DFS(graph, start, end):
    frontier = [start]
    parent = {start: None}

    while frontier:
        current_node = frontier.pop()

        if current_node == end:
            break

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

    return build_path(parent, start, end)
```

```

def UCS(graph, start, end):
    frontier = PriorityQueue()
    frontier.put((0, start))
    parent = {start: None}
    cost_so_far = {start: 0}

    while not frontier.empty():
        current_cost, current_node = frontier.get()

        if current_node == end:
            break

        for neighbor, weight in graph[current_node]:
            new_cost = current_cost + weight
            if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
                cost_so_far[neighbor] = new_cost
                parent[neighbor] = current_node
                frontier.put((new_cost, neighbor))

    path = build_path(parent, start, end)
    return cost_so_far.get(end, float('inf')), path

```

```

def build_path(parent, start, end):
    path = []
    while end is not None:
        path.append(end)
        end = parent[end]
    path.reverse()
    return path if path[0] == start else []

```

### c. Kết quả sau khi chạy:

Sau khi chạy chương trình, kết quả ta thu được như sau:

```

Result using BFS:
[0, 2, 7, 8, 10, 14, 15, 16, 17]
Result using DFS:
[0, 3, 4, 13, 10, 14, 15, 16, 17]
Result using UCS:
[0, 2, 7, 8, 10, 14, 15, 16, 17] Path Costs: 6910
Press any key to continue . . .

```

Khi sử dụng code và chạy tay, ta thấy kết quả DFS và UCS đều giống nhau. Tuy vậy, đoạn code chỉ in ra 1 đường đi theo thuật toán BFS thay vì 5 đường đi như khi ta chạy tay.

Lí do là vì, khi triển khai thuật toán BFS cho code, ta chỉ tìm kiếm 1 đường đi nông nhất từ đỉnh bắt đầu đến đỉnh kết thúc, bởi vì khi tìm được đường đi đến đỉnh End, thuật toán lập tức thoát khỏi vòng lặp (if current\_node == end:

break) và trả về đường đi này.

Vì vậy để in được 5 đường đi cho BFS, ta sửa lại đoạn code như sau.

```
def BFS_all_paths(graph, start, end):
    all_paths = []
    frontier = deque([[start]])

    while frontier:
        path = frontier.popleft()
        current_node = path[-1]

        if current_node == end:
            all_paths.append(path)
        else:
            for neighbor in graph[current_node]:
                if neighbor not in path:
                    new_path = list(path)
                    new_path.append(neighbor)
                    frontier.append(new_path)

    return all_paths
```

### Kết quả

```
Result using ABFS:
[[0, 2, 7, 8, 10, 14, 15, 16, 17], [0, 3, 4, 13, 10, 14, 15, 16, 17], [0, 1, 6, 2, 7, 8, 10, 14, 15, 16, 17], [0, 2, 5,
11, 12, 13, 10, 14, 15, 16, 17], [0, 1, 6, 2, 5, 11, 12, 13, 10, 14, 15, 16, 17]]
Press any key to continue . . .
```

Hơn nữa, ta xây dựng lại tất cả thuật toán bằng cách sử dụng lập trình hướng đối tượng, sử dụng các thuộc tính kế thừa, đa hình, ... , tạo các lớp...

```

from collections import defaultdict
from queue import Queue, PriorityQueue

class Graph:
    def __init__(self, matrix):
        self.adj_list = self.convert_graph(matrix)

    def convert_graph(self, a):
        adj_list = defaultdict(list)
        for i in range(len(a)):
            for j in range(len(a[i])):
                if a[i][j] == 1:
                    adj_list[i].append(j)
        return adj_list

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

```

```

class SearchAlgorithm:
    def __init__(self, graph):
        self.graph = graph

    def build_path(self, parent, start, end):
        path = []
        while end is not None:
            path.append(end)
            end = parent[end]
        path.reverse()
        return path if path[0] == start else []

```

```

class BFS(SearchAlgorithm):
    def search(self, start, end):
        frontier = Queue()
        frontier.put(start)
        parent = {start: None}

        while not frontier.empty():
            current_node = frontier.get()

            if current_node == end:
                break

            for neighbor in self.graph.adj_list[current_node]:
                if neighbor not in parent:
                    parent[neighbor] = current_node
                    frontier.put(neighbor)

        return self.build_path(parent, start, end)

```

```

class DFS(SearchAlgorithm):
    def search(self, start, end):
        frontier = [start]
        parent = {start: None}

        while frontier:
            current_node = frontier.pop()

            if current_node == end:
                break

            for neighbor in self.graph.adj_list[current_node]:
                if neighbor not in parent:
                    parent[neighbor] = current_node
                    frontier.append(neighbor)

        return self.build_path(parent, start, end)

```

```

class UCS(SearchAlgorithm):
    def search(self, start, end):
        frontier = PriorityQueue()
        frontier.put((0, start))
        parent = {start: None}
        cost_so_far = {start: 0}

        while not frontier.empty():
            current_cost, current_node = frontier.get()

            if current_node == end:
                break

            for neighbor, weight in self.graph.adj_list[current_node]:
                new_cost = current_cost + weight
                if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
                    cost_so_far[neighbor] = new_cost
                    parent[neighbor] = current_node
                    frontier.put((new_cost, neighbor))

        path = self.build_path(parent, start, end)
        return cost_so_far.get(end, float('inf')), path

```

```

def read_txt(file):
    size = int(file.readline().strip())
    start, goal = [int(num) for num in file.readline().strip().split()]

    matrix = []
    for line in file:
        line = line.strip()
        if line:
            row = [int(num) for num in line.split()]
            matrix.append(row)

    return size, start, goal, matrix

```

```

if __name__ == "__main__":
    # Read file Input.txt and InputUCS.txt
    with open("C:/Users/T K/OneDrive - VNU-HCMUS/Desktop/Input.txt", "r") as file_1:
        size_1, start_1, goal_1, matrix_1 = read_txt(file_1)

    with open("C:/Users/T K/OneDrive - VNU-HCMUS/Desktop/InputUCS.txt", "r") as file_2:
        size_2, start_2, goal_2, matrix_2 = read_txt(file_2)

    graph_1 = Graph(matrix_1)
    graph_2 = Graph(matrix_2)
    graph_2.adj_list = graph_2.convert_graph_weight(matrix_2)

    # Run BFS
    bfs = BFS(graph_1)
    result_bfs = bfs.search(start_1, goal_1)
    print("Result using BFS: \n", result_bfs)

    # Run DFS
    dfs = DFS(graph_1)
    result_dfs = dfs.search(start_1, goal_1)
    print("Result using DFS: \n", result_dfs)

    # Run UCS
    ucs = UCS(graph_2)
    cost, result_ucs = ucs.search(start_2, goal_2)
    print("Result using UCS: \n", result_ucs, "Path Costs: ", cost)

```

Người báo cáo

Nguyễn Thanh Kiên