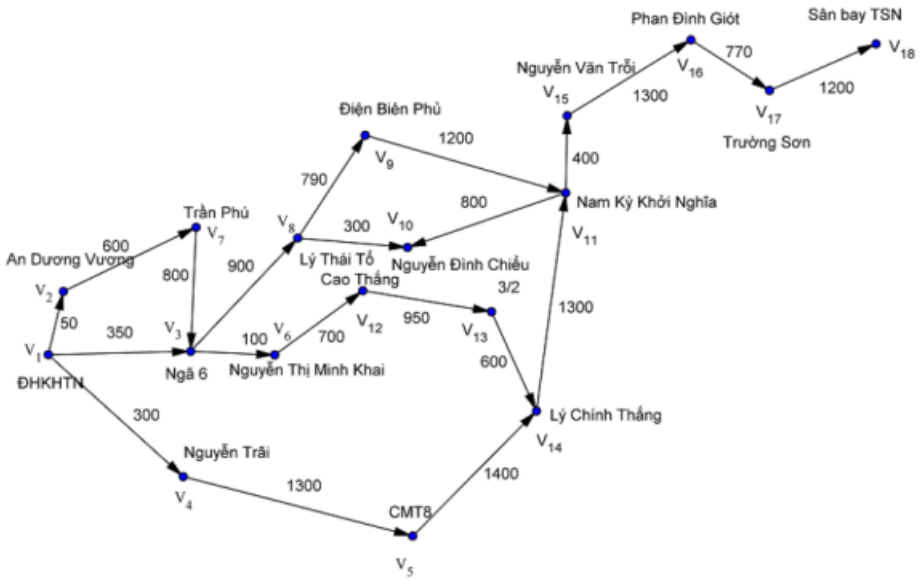
BÁO CÁO THỰC HÀNH NHẬP MÔN TRÍ TUỆ NHÂN TẠO

Tuần 1

Bài toán. Cho đồ thị như hình vẽ bên dưới

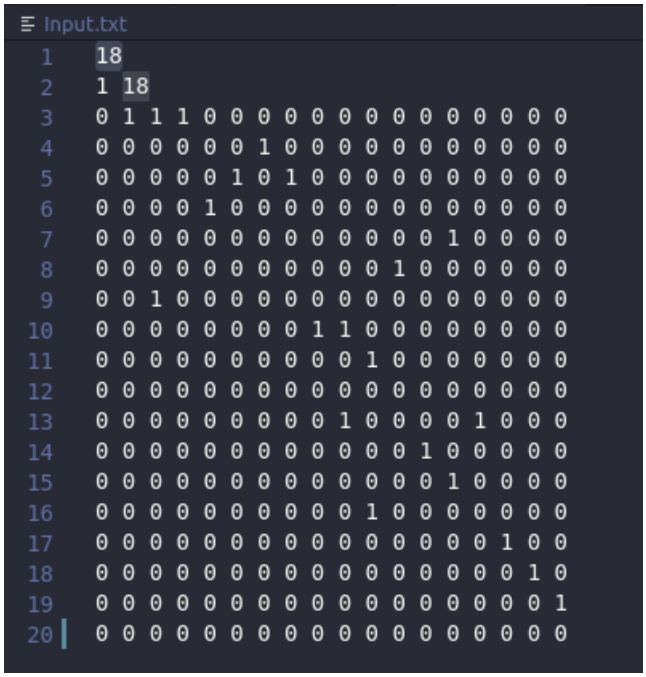


Tìm đường đi ngắn nhất từ trường Đại học Khoa học Tự nhiên (V_1) tới sân bay Tân Sơn Nhất (V_{18}) dùng các thuật toán sau:

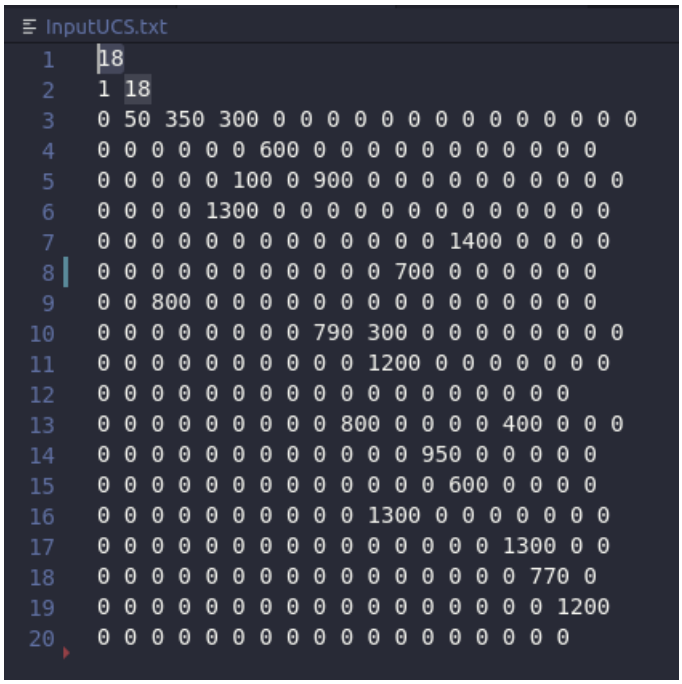
1. BFS
2. DFS
3. UCS

1 Dữ liệu đầu vào

1. Dữ liệu cho BFS và DFS



2. Dữ liệu cho UCS



Trong đó

- 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 đến 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 đến j.

2 Xử lý dữ liệu đầu vào

Đọc dữ liệu từ 2 file ở trên và chuyển dữ liệu kiểu *string* thành *int* và *list of int*.

```
107 def handle_input(name_file):
108     with open(name_file, 'r') as f:
109         vertices = int(f.readline())
110         start, goal = [int(num) for num in f.readline().split(' ')]
111         adj_matrix = [[int(num) for num in line.split(' ')] for line in f]
112         f.close()
113     return vertices, start, goal, adj_matrix
```

3 Xây dựng graph

Tạo **class Graph** có 2 thành phần là số đỉnh (**V**) và một danh sách chứa các cạnh của đồ thị (**graph**).

```
 5 class Graph:
 6     def __init__(self, vertices):
 7         self.V = vertices
 8         self.graph = defaultdict(list)
 9 
10     def add_edge(self, src, dest):
11         self.graph[src].append(dest)
12 
13     def add_edge_weight(self, src, dest, weight):
14         self.graph[src].append((weight, dest))
15 
16     def display_graph(self):
17         for node in self.graph:
18             print(node, "\t-->\t", self.graph[node])
```

Sau khi xử lý dữ liệu đầu vào, ta sẽ có số đỉnh và ma trận kề. Từ bộ dữ liệu này, ta có thể xây dựng được graph thông qua hàm **create_graph**. Hàm này sẽ chạy trên ma trận kề, ứng với giá trị 1 tại vị trí (i, j) sẽ tạo một đường nối từ i đến j và lưu vào **graph**.

```
115 def create_graph(vertices, adj_matrix):
116     temp_graph = Graph(vertices)
117     for i in range(vertices):
118         for j in range(vertices):
119             if adj_matrix[i][j] != 0:
120                 temp_graph.add_edge(i, j)
121     return temp_graph
```

Kết quả sau khi xây dựng class graph không có chi phí đường đi

```
0       -->     [1, 2, 3]
1       -->     [6]
2       -->     [5, 7]
3       -->     [4]
4       -->     [13]
5       -->     [11]
6       -->     [2]
7       -->     [8, 9]
8       -->     [10]
10      -->     [9, 14]
11      -->     [12]
12      -->     [13]
13      -->     [10]
14      -->     [15]
15      -->     [16]
16      -->     [17]
```

Tương tự ta sẽ xây dựng được hàm tạo graph có chi phí đường đi giữa 2 node, với giá trị khác 0 tại vị trí (i, j) sẽ là chi phí đường đi từ i đến j.

```
123 def create_graph_weight(vertices, adj_matrix):
124     temp_graph = Graph(vertices)
125     for i in range(vertices):
126         for j in range(vertices):
127             if adj_matrix[i][j] != 0:
128                 temp_graph.add_edge_weight(i, j, adj_matrix[i][j])
129     return temp_graph
```

Kết quả sau khi xây dựng class graph có chi phí đường đi

```
0       -->     [(50, 1), (350, 2), (300, 3)]
1       -->     [(600, 6)]
2       -->     [(100, 5), (900, 7)]
3       -->     [(1300, 4)]
4       -->     [(1400, 13)]
5       -->     [(700, 11)]
6       -->     [(800, 2)]
7       -->     [(790, 8), (300, 9)]
8       -->     [(1200, 10)]
10      -->     [(800, 9), (400, 14)]
11      -->     [(950, 12)]
12      -->     [(600, 13)]
13      -->     [(1300, 10)]
14      -->     [(1300, 15)]
15      -->     [(770, 16)]
16      -->     [(1200, 17)]
```

4 Thuật toán Breadth First Search

4.1 Ý tưởng

Từ trạng thái gốc ban đầu, xác định và duyệt qua các trạng thái kề xung quanh trạng thái gốc vừa xét. Tiếp tục quá trình duyệt qua các trạng thái kề trạng thái vừa xét cho đến khi đạt được kết quả cần tìm hoặc duyệt qua tất cả các trạng thái. Tại mỗi bước chọn trạng thái để phát triển (trạng thái được sinh ra trước các trạng thái chờ phát triển khác), thêm trạng thái được chọn vào danh sách đã duyệt để đánh dấu. Danh sách này được xử lý như hàng đợi **Queue**.

4.2 Khởi tạo

- Một hàng đợi **frontier** có một phần tử là giá trị **start**,
- List **explored** để kiểm soát các node đã duyệt qua để tránh bị trùng lắp. Tuy nhiên, vẫn sẽ có trường hợp trùng trên **explored**, nhưng sẽ có ảnh hưởng không nhiều đến kết quả,
- Dictionary **father** để lưu vị trí node cha của node đang xét,
- List **path** chứa kết quả đường đi từ start đến goal nhưng có chiều ngược lại.

4.3 Các bước thực hiện

- Bước 1: Tập **frontier** chứa node gốc chờ được xét.
- Bước 2: Kiểm tra tập **frontier** có rỗng không
  - Nếu tập **frontier** không rỗng, lấy một node ra khỏi tập **frontier** làm node đang xét **current_node**. Nếu **current_node** là node **goal** cần tìm, chuyển sang bước 4.
  - Nếu tập **frontier** rỗng, thông báo lỗi, không tìm được đường đi.
- Bước 3: Đưa **current_node** vào **explored**, sau đó xác định các node kề với **current_node** vừa xét. Nếu các node kề không thuộc **explored**, đưa chúng vào cuối tập **frontier**. Nếu node kề chưa có trong **father**, thêm **current_node** vào **father** tại node kề dang xét (ta chỉ lấy giá trị node cha đầu tiên xuất hiện khi duyệt). Quay lại bước 2.
- Bước 4: Duyệt trên tập **father**, bắt đầu tại node **goal**. Mỗi lần duyệt, thêm giá trị node cha của node đang xét vào **path**.
- Bước 5: Đảo ngược **path** ta sẽ có kết quả đường đi cần tìm.

4.4 Cài đặt hàm BFS

```python
    def bfs(self, start, goal):
        frontier = Queue()
        frontier.put(start)
        explored = []
        father = {}
        path = [goal]
        while True:
            if frontier.empty():
                raise Exception("No way Exception")
            current_node = frontier.get()
            explored.append(current_node)
            if current_node == goal:
                key = goal
                while key in father.keys():
                    value = father.pop(key)
                    path.append(value)
                    key = value
                    if key == start:
                        break
                path.reverse()
                return path
            if current_node not in self.graph:
                continue
            for node in self.graph[current_node]:
                if node not in explored:
                    frontier.put(node)
                    if node not in father.keys():
                        father[node] = current_node
```

4.5 Kết quả

```
Path from 0 to 17 using BFS: [0, 2, 7, 8, 10, 14, 15, 16, 17]
```

5 Thuật toán Depth First Search

5.1 Ý tưởng

Từ trạng thái gốc ban đầu, ta duyệt đi xa nhất theo từng nhánh. Khi nhánh đã duyệt hết, lùi về từng trạng thái để tìm và duyệt những nhánh tiếp theo. Quá trình duyệt chỉ dừng lại khi tìm thấy trạng thái cần tìm hoặc tất cả trảng thái đều đã được duyệt qua. Tại mỗi bước trạng thái được chọn để phát triển (trạng thái được sinh ra sau cùng trong số các trạng thái chờ phát triện), thêm trạng thái được chọn vào danh sách đã duyệt để đánh dấu. Danh sách này được xử lý như ngăn xếp **Stack**.

5.2 Cách cài đặt thuật toán DFS

Cài đặt thuật toán DFS tương tự khi cài đặt thuật toán BFS, tuy nhiên ở BFS tập **frontier** là hàng đợi **Queue** còn ở DFS là ngăn xếp **Stack**. Ở thuật toán DFS, những node được

thêm vào **frontier** sau cùng sẽ được lấy ra để duyệt trước.

5.3 Cài đặt hàm DFS

```
48      def dfs(self, start, goal):
49          frontier = LifoQueue()
50          frontier.put(start)
51          explored = []
52          father = {}
53          path = [goal]
54          while True:
55              if frontier.empty():
56                  raise Exception("No way Exception")
57              current_node = frontier.get()
58              explored.append(current_node)
59              if current_node == goal:
60                  key = goal
61                  while key in father.keys():
62                      value = father.pop(key)
63                      path.append(value)
64                      key = value
65                      if key == start:
66                          break
67                  path.reverse()
68                  return path
69              if current_node not in self.graph:
70                  continue
71              for node in self.graph[current_node]:
72                  if node not in explored:
73                      frontier.put(node)
74                      if node not in father.keys():
75                          father[node] = current_node
```

5.4 Kết quả

```
Path from 0 to 17 using DFS: [0, 3, 4, 13, 10, 14, 15, 16, 17]
```

6 Thuật toán Uniform-Cost Search

6.1 Ý tưởng

Từ trạng thái gốc ban đầu, 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.

6.2 Khởi tạo

- Một hàng đợi ưu tiên (priority queue) **frontier** có một phần tử là (0, **start**),
- List **explored** để kiểm soát các node đã duyệt qua để tránh bị trùng lắp. Tuy nhiên, vẫn sẽ có trường hợp trùng trên **explored**, nhưng sẽ có ảnh hưởng không nhiều đến kết quả,

- Dictionary **father** để lưu vị trí node cha của node đang xét,
- List **path** chứa kết quả đường đi từ start đến goal nhưng có chiều ngược lại.

6.3 Các bước thực hiện

- Bước 1: Tập **frontier** chứa node gốc chờ được xét.
- Bước 2: Kiểm tra tập **frontier** có rỗng không
  - Nếu tập **frontier** không rỗng, lấy một node ra khỏi tập **frontier** làm node đang xét **current_node** và chi phí đến node đang xét **current_w**. Nếu **current_node** là node **goal** cần tìm, chuyển sang bước 4.
  - Nếu tập **frontier** rỗng, thông báo lỗi, không tìm được đường đi.
- Bước 3: Đưa **current_node** vào **explored**, sau đó xác định các node kề với **current_node** vừa xét. Ta lấy ra 2 giá trị là **weight** chi phí từ node đang xét đến node kề chờ xét và **node** vị trí node chờ xét. Nếu các node kề không thuộc **explored**, đưa chúng vào cuối tập **frontier** cùng với chi phí cộng dồn từ node gốc đến node kề đang xét. Nếu node kề chưa có trong **father**, thêm **current_node** vào **father** tại node kề dang xét (ta chỉ lấy giá trị node cha đầu tiên xuất hiện khi duyệt). Quay lại bước 2.
- Bước 4: Duyệt trên tập **father**, bắt đầu tại node **goal**. Mỗi lần duyệt, thêm giá trị node cha của node đang xét vào **path**.
- Bước 5: Đảo ngược **path** ta sẽ có kết quả đường đi cần tìm. Và **current_w** là chi phí cho đường đi tối ưu từ **start** đến **goal**.

6.4 Cài đặt hàm UCS

```
77      def ucs(self, start, goal):
78          frontier = PriorityQueue()
79          frontier.put((0, start))
80          explored = []
81          father = {}
82          path = [goal]
83          while True:
84              if frontier.empty():
85                  raise Exception("No way Exception")
86              current_w, current_node = frontier.get()
87              explored.append(current_node)
88              if current_node == goal:
89                  key = goal
90                  while key in father.keys():
91                      value = father.pop(key)
92                      path.append(value)
93                      key = value
94                      if key == start:
95                          break
96                  path.reverse()
97                  return current_w, path
98              if current_node not in self.graph:
99                  continue
100             for vex in self.graph[current_node]:
101                 weight, node = vex
102                 if node not in explored:
103                     frontier.put((current_w + weight, node))
104                     if node not in father.keys():
105                         father[node] = current_node
```

6.5 Kết quả

```
The shortest path from 0 to 17 using UCS: [0, 2, 7, 8, 10, 14, 15, 16, 17] - cost: 6910
```