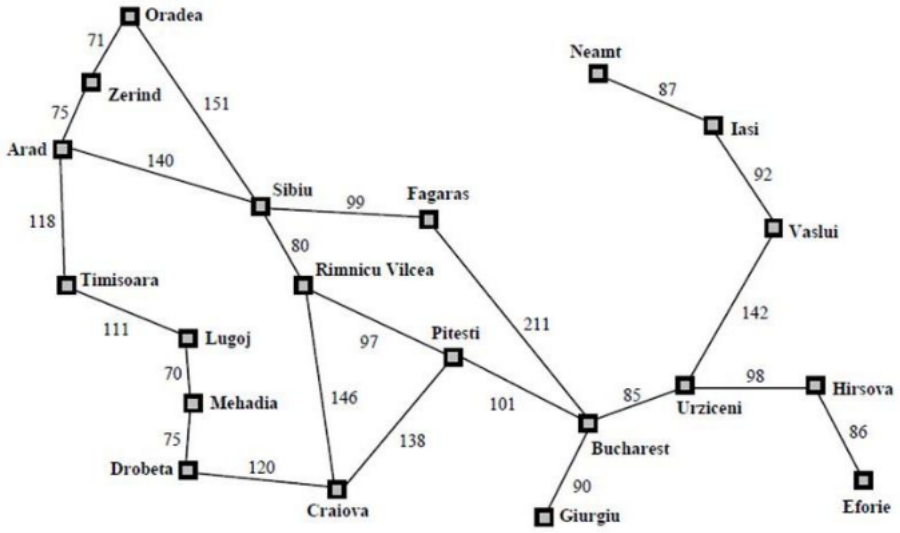
BÁO CÁO THỰC HÀNH NHẬP MÔN TRÍ TUỆ NHÂN TẠO

Tuần 3

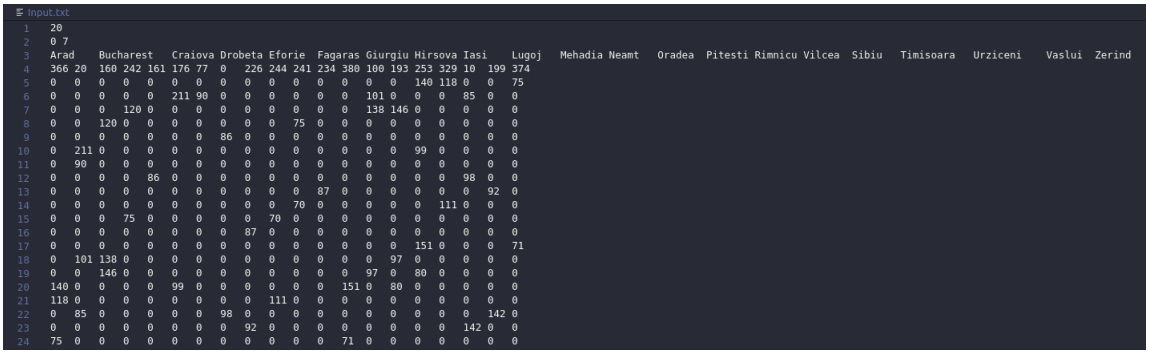
Bài toán 1. Cài đặt thuật toán Greedy - Best - First search để tìm đường đi từ Arad tới Hirsova như hình với h(n) được xác định như sau:

h(Arad) = 366	h(Hirsova) = 0	h(Rimnicu Vilcea) = 193
h(Bucharest) = 20	h(Iasi) = 226	h(Sibiu) = 253
h(Craiova) = 160	h(Lugoj) = 244	h(Timisoara) = 329
h(Drobeta) = 242	h(Mehadia) = 241	h(Urziceni) = 10
h(Eforie) = 161	h(Neamt) = 234	h(Vaslui) = 199
h(Fagaras) = 176	h(Oradea) = 380	h(Zerind) = 374
h(Giurgiu) = 77	h(Pitesti) = 100	

Bài toán 2. Cài đặt thuật toán A* để tìm đường đi ngắn nhất từ Arad tới Hirsova như hình với hàm h(n) được xác định như trong bài tập 1 và g(n) là khoảng cách giữa 2 thành phố.



1 Dữ liệu đầu vào



Trong đó

- Dòng 1: Số node trên đồ thị.
- Dòng 2: Node xuất phát và node đích.
- Dòng 3: Tên của từng node.
- Dòng 4: Heuristic của từng node.
- Những dòng tiếp theo: Ma trận kề M của đồ thị với quy ước
  - $M[i][j] = w$: có đường nối trực tiếp từ i đến j với khoảng cách là w ($w > 0$).
  - $M[i][j] = 0$: không có đường nối trực tiếp từ i đến j.

2 Xử lý dữ liệu đầu vào

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

```
117  def handle_input(name_file):
118      with open(name_file, 'r') as f:
119          vertices = int(f.readline())
120          start, goal = [int(num) for num in f.readline().split(' ')]
121          name_cities = [name for name in f.readline()[:-1].split('\t')]
122          heuristic = [int(num) for num in f.readline().split('\t')]
123          adj_matrix = [[int(num) for num in line.split('\t')] for line in f]
124          f.close()
125      return vertices, start, goal, name_cities, heuristic, adj_matrix
```

3 Xây dựng graph

Tạo **class Graph** có 4 thành phần là số đỉnh (**V**), một danh sách chứa các cạnh của đồ thị (**graph**), danh sách chứa các giá trị heuristic của từng node (**heuristics**) và một danh sách chứa các tên của từng node (**name_cities**).

```python
class Graph:
    def __init__(self, vertices, heuristics, name_cities):
        self.V = vertices
        self.graph = defaultdict(list)
        self.heuristics = heuristics
        self.name_cities = name_cities

    def add_edge(self, src, dest, name, heuristic, weight):
        self.graph[src].append((dest, name, heuristic, weight))

    def display_graph(self):
        for node in self.graph:
            print((node, self.name_cities[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 khác 0 tại vị trí (i, j) sẽ tạo một đường nối từ i đến j và lưu vào **graph**.

```python
def create_graph(vertices, heuristic, name_cities, adj_matrix):
    temp_graph = Graph(vertices, heuristic, name_cities)
    for i in range(vertices):
        for j in range(vertices):
            if adj_matrix[i][j] != 0:
                temp_graph.add_edge(i, j, name_cities[j], heuristic[j] ,adj_matrix[i][j])
    return temp_graph
```

Kết quả sau khi xây dựng class graph ta được

```
(0, 'Arad')     -->     [(15, 'Sibiu', 253, 140), (16, 'Timisoara', 329, 118), (19, 'Zerind', 374, 75)]
(1, 'Bucharest')        -->     [(5, 'Fagaras', 176, 211), (6, 'Giurgiu', 77, 90), (13, 'Pitesti', 100, 101), (17, 'Urziceni', 10, 85)]
(2, 'Craiova')  -->     [(3, 'Drobeta', 242, 120), (13, 'Pitesti', 100, 138), (14, 'Rimnicu Vilcea', 193, 146)]
(3, 'Drobeta')  -->     [(2, 'Craiova', 160, 120), (10, 'Mehadia', 241, 75)]
(4, 'Eforie')   -->     [(7, 'Hirsova', 0, 86)]
(5, 'Fagaras')  -->     [(1, 'Bucharest', 20, 211), (15, 'Sibiu', 253, 99)]
(6, 'Giurgiu')  -->     [(1, 'Bucharest', 20, 90)]
(7, 'Hirsova')  -->     [(4, 'Eforie', 161, 86), (17, 'Urziceni', 10, 98)]
(8, 'Iasi')     -->     [(11, 'Neamt', 234, 87), (18, 'Vaslui', 199, 92)]
(9, 'Lugoj')    -->     [(10, 'Mehadia', 241, 70), (16, 'Timisoara', 329, 111)]
(10, 'Mehadia')         -->     [(3, 'Drobeta', 242, 75), (9, 'Lugoj', 244, 70)]
(11, 'Neamt')   -->     [(8, 'Iasi', 226, 87)]
(12, 'Oradea')  -->     [(15, 'Sibiu', 253, 151), (19, 'Zerind', 374, 71)]
(13, 'Pitesti')         -->     [(1, 'Bucharest', 20, 101), (2, 'Craiova', 160, 138), (14, 'Rimnicu Vilcea', 193, 97)]
(14, 'Rimnicu Vilcea')  -->     [(2, 'Craiova', 160, 146), (13, 'Pitesti', 100, 97), (15, 'Sibiu', 253, 80)]
(15, 'Sibiu')   -->     [(0, 'Arad', 366, 140), (5, 'Fagaras', 176, 99), (12, 'Oradea', 380, 151), (14, 'Rimnicu Vilcea', 193, 80)]
(16, 'Timisoara')       -->     [(0, 'Arad', 366, 118), (9, 'Lugoj', 244, 111)]
(17, 'Urziceni')        -->     [(1, 'Bucharest', 20, 85), (7, 'Hirsova', 0, 98), (18, 'Vaslui', 199, 142)]
(18, 'Vaslui')  -->     [(8, 'Iasi', 226, 92), (17, 'Urziceni', 10, 142)]
(19, 'Zerind')  -->     [(0, 'Arad', 366, 75), (12, 'Oradea', 380, 71)]
```

4 Thuật toán Greedy-Best-First Search

4.1 Ý tưởng

GBFS mở rộng nút gần đích nhất với hi vọng cách làm này sẽ dẫn đến lời giải một cách nhanh nhất. Đánh giá chi phí của các nút chỉ dựa trên hàm heuristic:

$$f(n) = h(n)$$

4.2 Khởi tạo

- Một hàng đợi ưu tiên (priority queue) **frontier** có một phần tử là (heuristic[start], 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** và heuristic node đang xét **current_h**. 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 **heur** heuristic của 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 heuristic của 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.

4.4 Cài đặt hàm GBFS

```
26      def GBFS(self, start, goal):
27          frontier = PriorityQueue()
28          root = (self.heuristics[start], start)
29          frontier.put(root)
30          explored = []
31          father = {}
32          path = [goal]
33          while True:
34              if frontier.empty():
35                  raise Exception("No way Exception")
36              current_h, current_node = frontier.get()
37              explored.append(current_node)
38              if current_node == goal:
39                  key = goal
40                  while key in father.keys():
41                      value = father.pop(key)
42                      path.append(value)
43                      key = value
44                      if key == start:
45                          break
46                  path.reverse()
47                  return path
48              if current_node not in self.graph:
49                  continue
50              for vex in self.graph[current_node]:
51                  heur, node = vex[2], vex[0]
52                  if node not in explored:
53                      frontier.put((heur, node))
54                      if node not in father.keys():
55                          father[node] = current_node
```

5 Thuật toán A^* Search

5.1 Ý tưởng

Thuật toán đánh giá 1 nút dựa trên chi phí đi từ nút gốc đến nút đó (g(n)) cộng với chi phí từ nút đó đến nút đích (h(n)).

$$f(n) = g(n) + h(n)$$

Hàm h(n) được gọi là chấp nhận được nếu với mọi trạng thái n, h(n) $\leq$ độ dài đường đi ngắn nhất thực tế từ u tới trạng thái đích.

5.2 Cách cài đặt thuật toán DFS

Thuật toán A^* sử dụng 2 tập hợp sau đây

- OPEN: tập chứa các trạng thái đã được sinh ra nhưng chưa được xét đến $\Rightarrow$ OPEN là 1 hàng đợi ưu tiên (priority queue) mà trong đó, phần tử có độ ưu tiên cao nhất là phần tử tốt nhất.

- CLOSE: tập chứa các trạng thái đã được xét đến. Ta cần lưu trữ những trạng thái này trong bộ nhớ để đề phòng khi một trạng thái mới được tạo ra lại trùng với 1 trạng thái mà ta đã xét đến trước đó. Trong trường hợp không gian tìm kiếm có dạng cây thì không cần dùng tập này.

Khi xét đến một trạng thái T_i bên cạnh việc lưu trữ 3 giá trị cơ bản $g(T_i)$, $h(T_i)$, $f(T_i)$ để phản ánh độ tốt của trạng thái đó, A^* còn lưu trữ thêm 2 thông số sau:

- *Trạng thái cha của trạng thái T_i (Father(T_i).* Trong trường hợp có nhiều trạng thái dẫn đến trạng thái T_i thì chọn Father(T_i) sao cho chi phí đi từ trạng thái khởi đầu đến T_i là thấp nhất, nghĩa là $g(T_i) = g(T_{father} + cost(T_{father}, T_i))$ là thấp nhất.

- *Danh sách các trạng thái kế tiếp của T_i.* Danh sách này lưu trữ các trạng thái kế tiếp T_k của T_i sao cho chi phí đến T_k thông qua T_i từ trạng thái ban đầu là thấp nhất $\Rightarrow$ danh sách này được tính từ thuộc tính Cha của các trạng thái được lưu trữ.

Procedure Astar-Search

Begin

1. Đặt OPEN chỉ chứa T_0. Đặt $g(T_0) = 0, h(T_0) = 0$ và $f(T_0) = 0$. Đặt CLOSE là tập rỗng.

2. Lặp lại các bước cho đến khi gặp điều kiện dừng.

   2.a. Nếu OPEN rỗng: bài toán vô nghiệm, thoát.

   2.b. Ngược lại, chọn T_max trong OPEN sao cho $f(T_{max})$ là nhỏ nhất.

      2.b.1. Lấy T_{max} ra khỏi OPEN và đưa T_{max} vào CLOSE.

      2.b.2. Nếu T_{max} là T_G (trạng thái đích) thì thoát và thông báo lời giải là T_{max}.

      2.b.3. Nếu T_{max} không phải là T_G. Tạo ra danh sách tất cả các trạng thái kế tiếp của T_{max}. Gọi một trạng thái này T_k. Với mỗi T_k, làm các bước sau:

         2.b.3.1. Tính $g(T_k) = g(T_{max}) + cost(T_{max}, T_k)$.

         2.b.3.2. Nếu tồn tại $T_{k'}$ trong OPEN trùng T_k.
         Nếu $g(T_k) < g(T_{k'}$ thì
         Đặt $g(T_{k'}) = g(T_k)$
         Tính lại $f(T_{k'}$
         Đặt $Cha(T_{k'}) = T_{max}$

         2.b.3.3. Nếu tồn tại $T_{k'}$ trong CLOSE trùng với T_k
         Nếu $g(T_k) < g(T_{k'})$ thì

Đặt $g(T_{k'}) = g(T_k)$

Tính lại $f(T_{k'}$

Đặt $Cha(T_{k'}) = T_{max}$

Lan truyền sự thay đổi giá trị g, f cho tất cả các trạng thái tiếp theo của T_i (ở tất cả các cấp) đã được lưu trữ trong CLOSE và OPEN

2.b.3.4. Nếu T_k chưa xuất hiện trong cả OPEN và CLOSE thì

Thêm T_k và OPEN

Tính: $f(T_k) = g(T_k) + h(T_k)$

Xây dựng lại đường đi từ T_0 tới T_G: ta lần ngược theo thuộc tính Cha của các trạng thái đã được lưu trữ trong CLOSE cho đến khi đạt đến T_0.

5.3 Cài đặt hàm A^*

```
57      def AStar(self, start, goal):
58          openset = []
59          openset_n = []
60          root = (start, 0, 0, 0)
61          openset.append(root)
62          openset_n.append(root[0])
63          closedset = []
64          closedset_n = []
65          father = {}
66          path = [goal]
67          while openset:
68              if len(openset) == 0:
69                  raise Exception("No way Exception")
70              current = get_min(openset)
71              current_n, current_h, current_g, current_f = current
72              openset.remove(current)
73              openset_n.remove(current_n)
74              closedset.append(current)
75              closedset_n.append(current_n)
76              if current_n == goal:
77                  key = goal
78                  while key in father.keys():
79                      value = father.pop(key)
80                      path.append(value)
81                      key = value
82                      if key == start:
83                          break
84                  return path[::-1]
85              for node in self.graph[current[0]]:
86                  node_n, node_h, node_g = node[0], node[2], node[-1]
87                  new_g = current_g + node_g
88                  new_f = new_g + node_h
89                  new_node = [node_n, node_h, new_g, new_f]
90                  if (node_n not in openset_n) & (node_n not in closedset_n) :
91                      openset.append(new_node)
92                      openset_n.append(node_n)
93                      father[node_n] = current_n
94                  else:
95                      if node_n in openset_n:
96                          if new_g < openset[openset_n.index(node_n)][2]:
97                              openset[openset_n.index(node_n)][2] = new_g
98                              openset[openset_n.index(node_n)][3] = new_f
99                              father[node_n] = current_n
100                     else:
101                         if node_n in closedset_n:
102                             if new_g < closedset[closedset_n.index(node_n)][2]:
103                                 closedset[closedset_n.index(node_n)][2] = new_g
104                                 closedset[closedset_n.index(node_n)][3] = new_f
105                                 father[node_n] = current_n
```

6 Kết quả

```
********************************************************************************
>> Using Greedy Best First Search
Path from Arad to Hirsova:
Arad  --> Sibiu  --> Fagaras  --> Bucharest  --> Urziceni  --> Hirsova
********************************************************************************
>> Using A* Search
Path from Arad to Hirsova:
Arad  --> Sibiu  --> Rimnicu Vilcea  --> Pitesti  --> Bucharest  --> Urziceni  --> Hirsova
********************************************************************************
```