BÀI TẬP THỰC HÀNH TUẦN 3

Mục lục

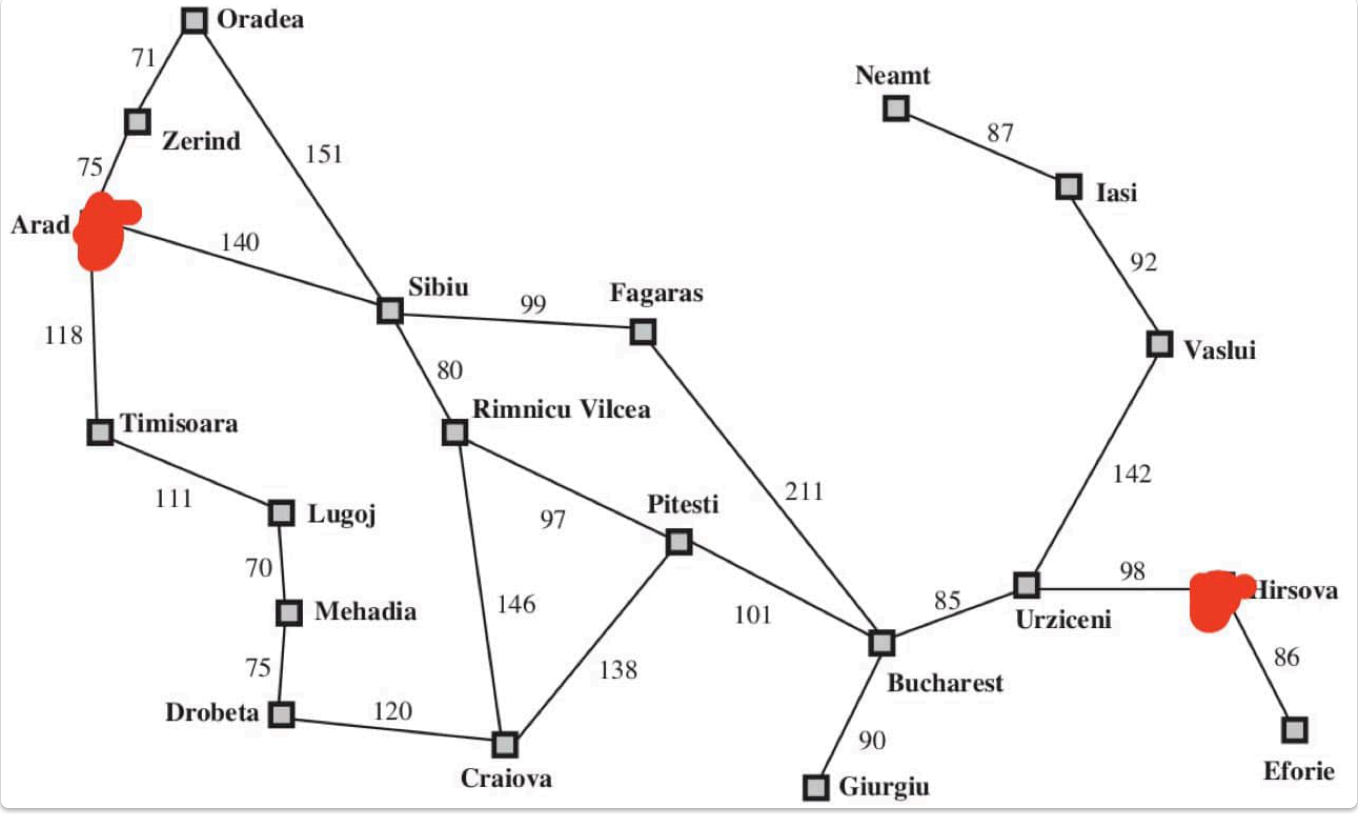
1. Cài đặt và thực thi chương trình

Sau khi em cài đặt code và chạy code thì không có lỗi nào xảy ra, chương trình chạy ổn.

2. Chạy tay GBFS và tiếp tục chạy tay xong thuật toán A star

2.1. GBFS

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	



Ta sẽ có cấu trúc hàng đợi ưu tiên là tập OPEN lưu trữ các nút mà có heuristic nhỏ nhất là ưu tiên, và trong đó có phần parent. Và tập CLOSE lưu lại các nút đã đi qua vì đây là Graph Search.

Tập OPEN và CLOSE ban đầu:

OPEN: {(366, Arad, Cha = None)}
CLOSE: {}

BẮT ĐẦU THUẬT TOÁN

Thăm Arad
OPEN: {(253, Sibiu, Cha=Arad), (329, Timisoara, Cha=Arad), (374, Zerind, Cha=Arad)}
CLOSE: {Arad}

- Đã thêm Zerind, Sibiu, Timisoara
- Sibiu là nút có H nhỏ nhất trong OPEN nên ta thăm nó

Thăm Sibiu
OPEN: {(176, Fagaras, Cha=Sibiu), (193, Riminicu Vilcea, Cha=Sibiu), (329, Timisoara, Cha=Arad), (374, Zerind, Cha=Arad), (380, Oradea, Cha=Sibiu)}
CLOSE: {Arad, Sibiu}

- Đã thêm Oradea, Fagaras, Rimnicu Vilcea
- Không thêm lại Arad vì nó đã được thăm trước đó

- Fagaras* là nút có H nhỏ nhất trong OPEN nên ta thăm nó

Thăm Fagaras

OPEN: {(20, Bucharest, Cha=Fagaras), (193, Riminicu Vilcea, Cha=Sibiu), (329, Timisoara, Cha=Arad), (374, Zerind, Cha=Arad), (380, Oradea, Cha=Sibiu)}
CLOSE: {Arad, Sibiu, Fagaras}

- Đã thêm Bucharest
- Không thêm lại Sibiu vì nó đã được thăm trước đó
- Bucharest là nút có H nhỏ nhất trong OPEN nên ta thăm nó

Thăm Bucharest

OPEN: {(10, Urziceni, Cha=Bucharest), (77, Giurgiu, Cha=Bucharest), (100, Pitesti, Cha=Bucharest), (193, Riminicu Vilcea, Cha=Sibiu), (329, Timisoara, Cha=Arad), (374, Zerind, Cha=Arad), (380, Oradea, Cha=Sibiu)}
CLOSE: {Arad, Sibiu, Fagaras, Bucharest}

- Đã thêm Pitesti, Urziceni, Giurgiu
- Không thêm lại Bucharest vì nó đã được thăm trước đó
- Urziceni là nút có H nhỏ nhất trong OPEN nên ta thăm nó

Thăm Urziceni

OPEN: {(0, Hirsova, Cha=Urziceni), (77, Giurgiu, Cha=Bucharest), (100, Pitesti, Cha=Bucharest), (193, Riminicu Vilcea, Cha=Sibiu), (199, Vaslui, Cha=Urziceni), (329, Timisoara, Cha=Arad), (374, Zerind, Cha=Arad), (380, Oradea, Cha=Sibiu)}
CLOSE: {Arad, Sibiu, Fagaras, Bucharest, Urziceni}

- Đã thêm Vaslui, Hirsova
- Không thêm lại Urziceni vì nó đã được thăm trước đó
- Hirsova là nút có H nhỏ nhất trong OPEN nên ta thăm nó

Thăm Hirsova

- Đây là điểm đến, dừng thuật toán

Vậy đường đi từ Arad đến Hirsova sử dụng GBFS là:

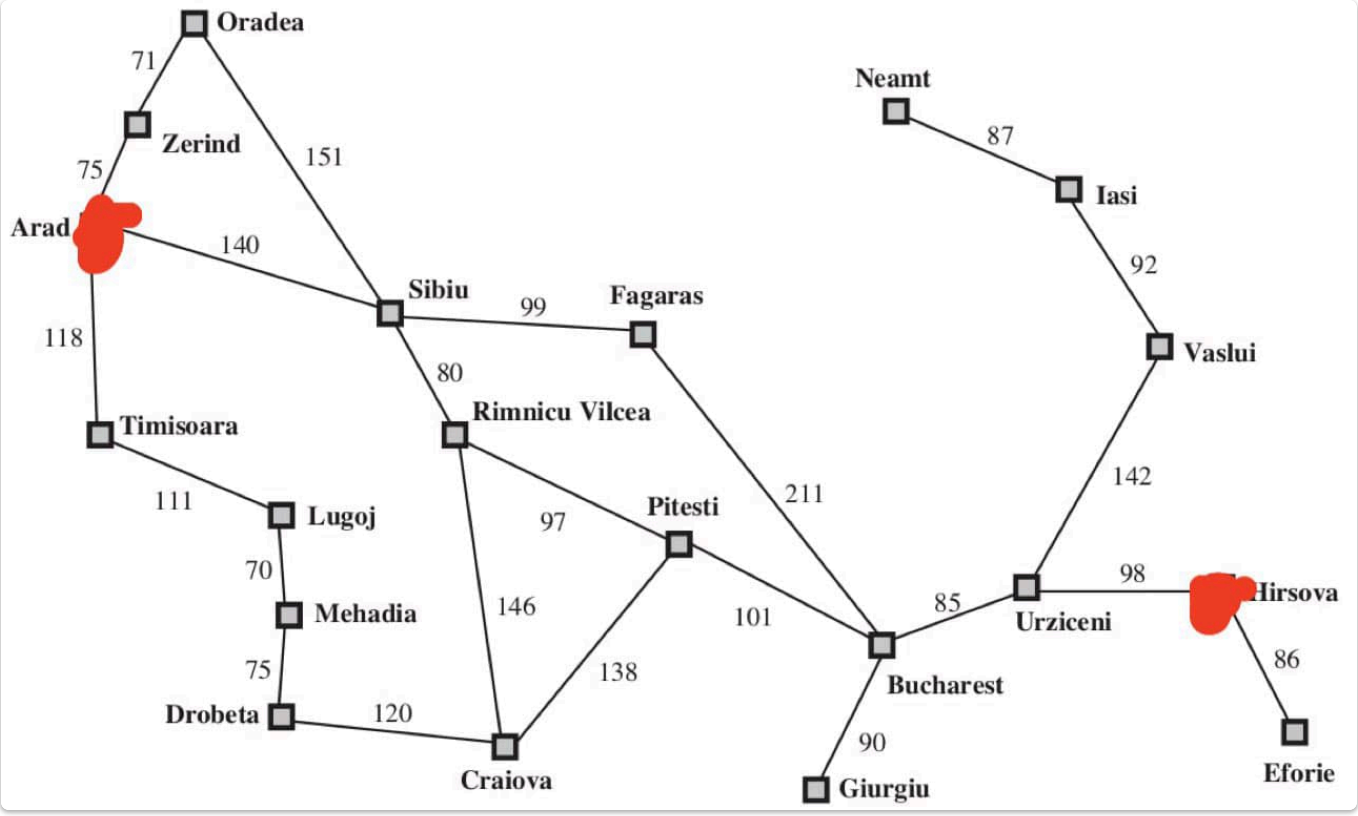
$$\text{Arad} \rightarrow \text{Sibiu} \rightarrow \text{Fagaras} \rightarrow \text{Bucharest} \rightarrow \text{Urziceni} \rightarrow \text{Hirsova}$$

Với chi phí tổng là:

$$\text{cost} = 140 + 99 + 211 + 85 + 98 = 633$$

2.2. A star Search

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	



Em sẽ chạy tiếp tục dựa trên quá trình chạy trước đó của cô:

Bước cuối cùng của cô chạy:

OPEN = {
(Timisoara, g = 118, h = 329, f = 447, Cha = Arad),
(Zerind, g = 75, h = 374, f = 449, Cha = Arad),
(Oradea, g = 291, h = 380, f = 617, Cha = Sibiu),
(Craiova, g = 366, h = 160, f = 526, Cha = R.Vilcea),
(Giurgiu, g = 508, h = 77, f = 585, Cha = Bucharest),
(Urziceni, g = 503, h = 10, f = 513, Cha = Bucharest)
}

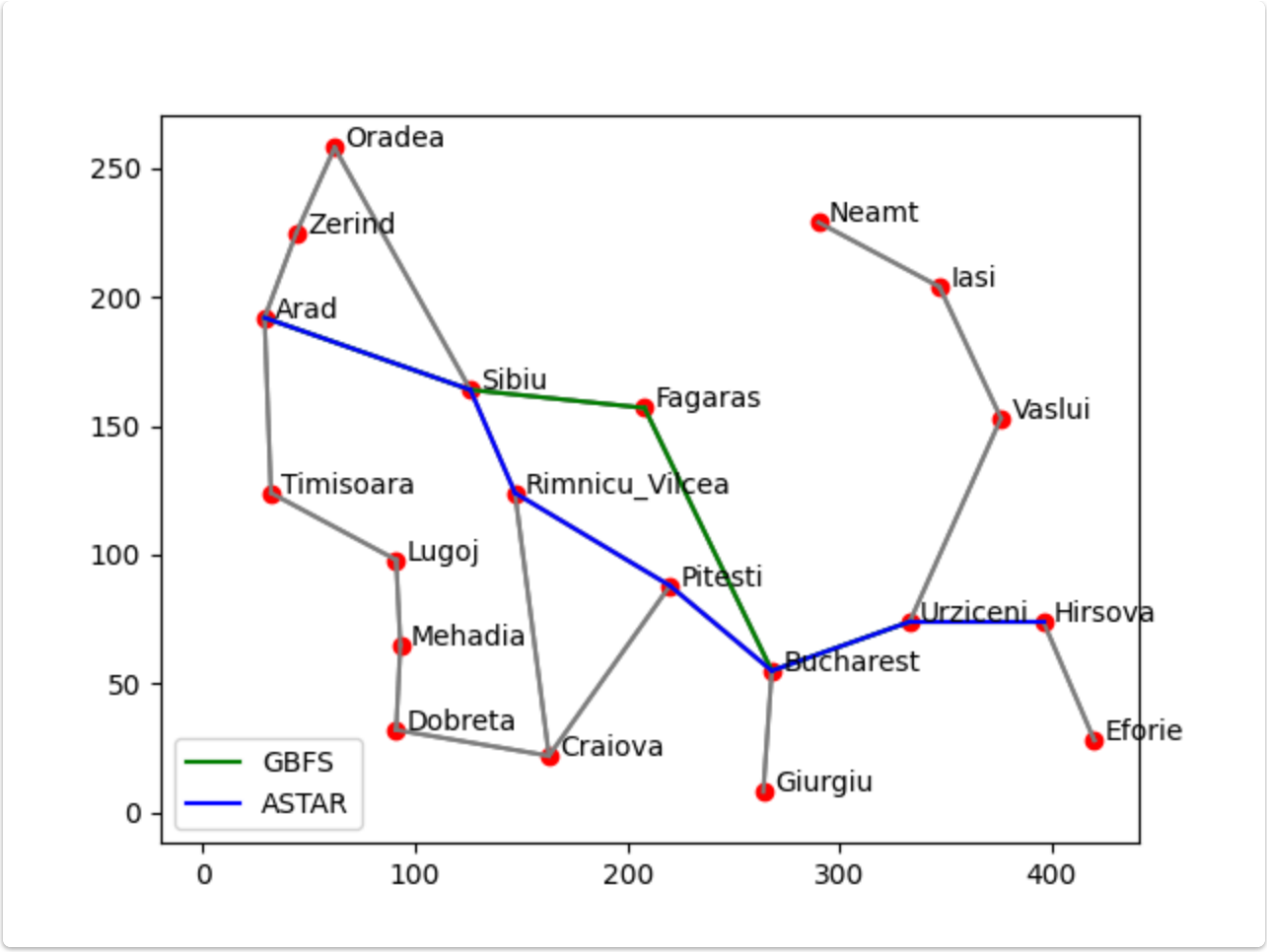
CLOSE = {
(Arad, g=0, h=0, f=0),
(Sibiu, g = 140, h = 253, f = 393, Cha = Arad),
(R.Vilcea, g = 220, h = 193, f = 413, Cha = Sibiu),
(Fagaras, g = 239, h = 176, f = 415, Cha = Sibiu),
(Pitesti, g = 317, h = 100, f = 417, Cha = R.Vilcea),
(Bucharest, g = 418, h = 20, f = 438, Cha = Pitesti)
}

Trong tập OPEN, Timisoara có giá trị f nhỏ nhất nên $T_{max} = \text{Timisoara}$. Từ Timisoara ta đi đến được Lugoj và Arad

```
h(Lugoj) = 244
g(Lugoj) = g(Timisoara)+cost(Timisoara, Lugoj)=118+111=229
f(Lugoj) = g(Lugoj)+h(Lugoj)=229+244=473
```

3. Kiểm tra tính đúng đắn của Code

```
Nhập đỉnh bắt đầu: 1
Nhập đỉnh kết thúc: 8
GBFS =>  ['Arad', 'Sibiu', 'Fagaras', 'Bucharest', 'Urziceni', 'Hirsova']
ASTAR =>  ['Arad', 'Sibiu', 'Rimnicu_Vilcea', 'Pitesti', 'Bucharest', 'Urziceni', 'Hirsova']
```



Thuật toán trong Code chạy đã ra kết quả đúng với khi em chạy tay, nhưng có nhiều điểm trong thuật toán mà em cảm thấy không đúng, và có thể sai trong vài ví dụ.

3.1. Đặt lại hàng đợi trong vòng While của cả hai thuật toán

Trong cả 2 thuật toán có một điểm chung là khởi tạo lại hàng đợi trong vòng lặp While như sau:

```
#### Thuật toán GBFS
while priorityQueue.empty() == False:
        current = priorityQueue.get()[1]
        path.append(current)

        if current == goalNode:
                break
        # khởi tạo lại hàng đợi
        priorityQueue = queue.PriorityQueue()

        for i in graph[current]:
                if i[0] not in path:
                        priorityQueue.put((heuristics[i[0]], i[0]))
```

```
#### Thuật toán A-star
while priorityQueue.empty() == False:
        current = priorityQueue.get()[1]
        path.append(current[0])
        distance += int(current[1])

        if current[0] == goalNode:
                break
        # khởi tạo lại hàng đợi
        priorityQueue = queue.PriorityQueue()

        for i in graph[current[0]]:
                if i[0] not in path:
                        priorityQueue.put((heuristics[i[0]] + int(i[1]) +
distance, i))
```

Việc khởi tạo lại hàng đợi trong vòng While như thế này có thể sẽ gặp các lỗi như sau:

- Làm mất các đỉnh đã được thêm vào trong hàng đợi nhưng chưa có thăm.
- Khiến thuật toán không xem xét hết tất cả đỉnh có thể tìm đến đích → có thể không tìm thấy lời giải.

Ví dụ về lỗi

Ví dụ trong `filecode.ipynb` có chỉ mục là `5.1. Đặt lại hàng đợi trong vòng While`

Với cách khởi tạo đồ thị và tính toán Heuristic tự động như sau:

```
def calculate_heuristics(city, endPoint):
    # hàm xây dựng nên Heuristic bằng cách tính thủ công khoảng cách L2
(đường chim bay)
    heuristics = {}
    endPointLocation = city.get(endPoint)
    for point in city.keys():
        location = city[point]
        distance = sqrt((location[0] - endPointLocation[0])**2 +
(location[1] - endPointLocation[1])**2)
        heuristics[point] = distance
    return heuristics

graph_test = {
    'A': [['B', 1], ['C', 2]],
    'B': [['A', 1]],
    'C': [['A', 1], ['E', 1]],
    'E': [['C', 1]] # E là điểm đích
```

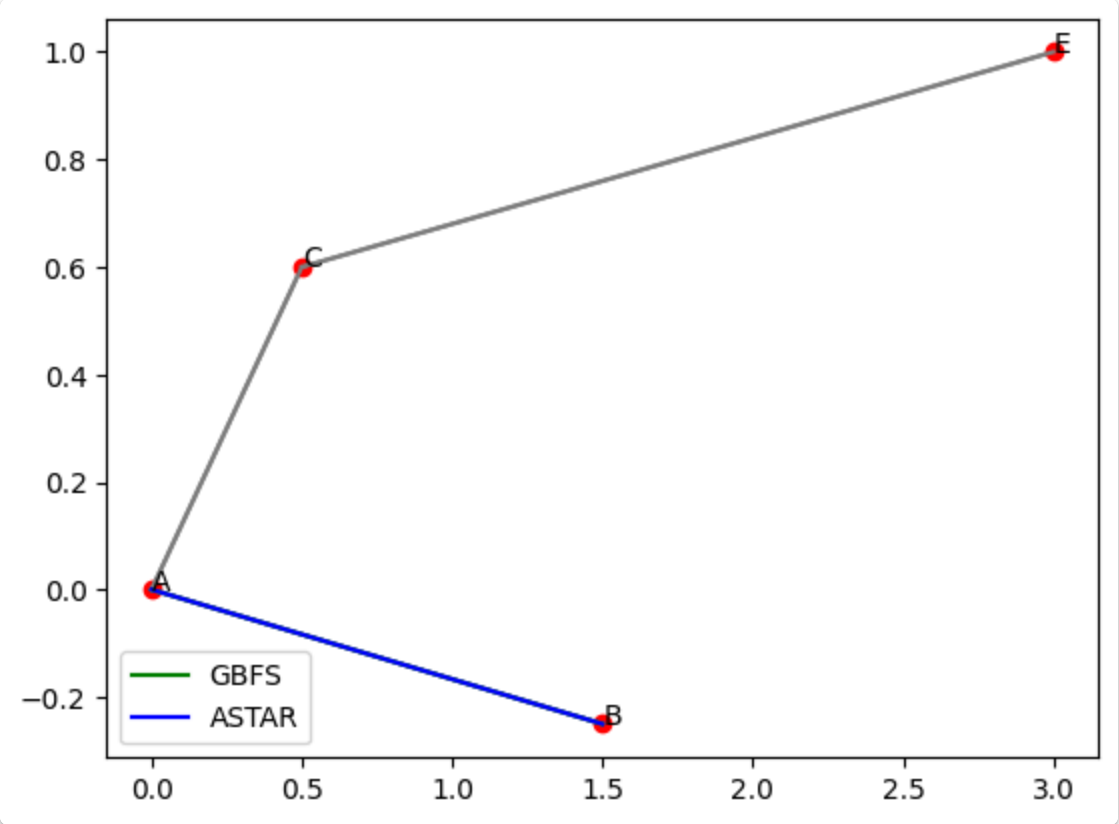
```
}

heuristics_test = calculate_heuristics(city_test, 'E')

city_test = {
    'A': [0, 0],
    'B': [1.5, -0.25],
    'C': [0.5, 0.6],
    'E': [3, 1] # E là điểm đích
}
```

Khi chạy 2 thuật toán sẽ xuất ra kết quả kèm hình vẽ như sau:

```
Caculated Heuristics: {
        'A': 3.1622776601683795,
        'B': 1.9525624189766635,
        'C': 2.5317977802344327,
        'E': 0.0
}
GBFS => ['A', 'B']
ASTAR => ['A', 'B']
```



Chỉ vì từ A sang B có Heuristic nhỏ hơn (đối với GBFS), hàm f nhỏ hơn (đối với A-star), và sau khi duyệt điểm B, nó đặt lại hàng đợi nên mất đi điểm C trong hàng đợi trước đó, dẫn đến không thể tìm được

đường đến đích là E.

Giải thích chi tiết:

Trong thuật toán GBFS

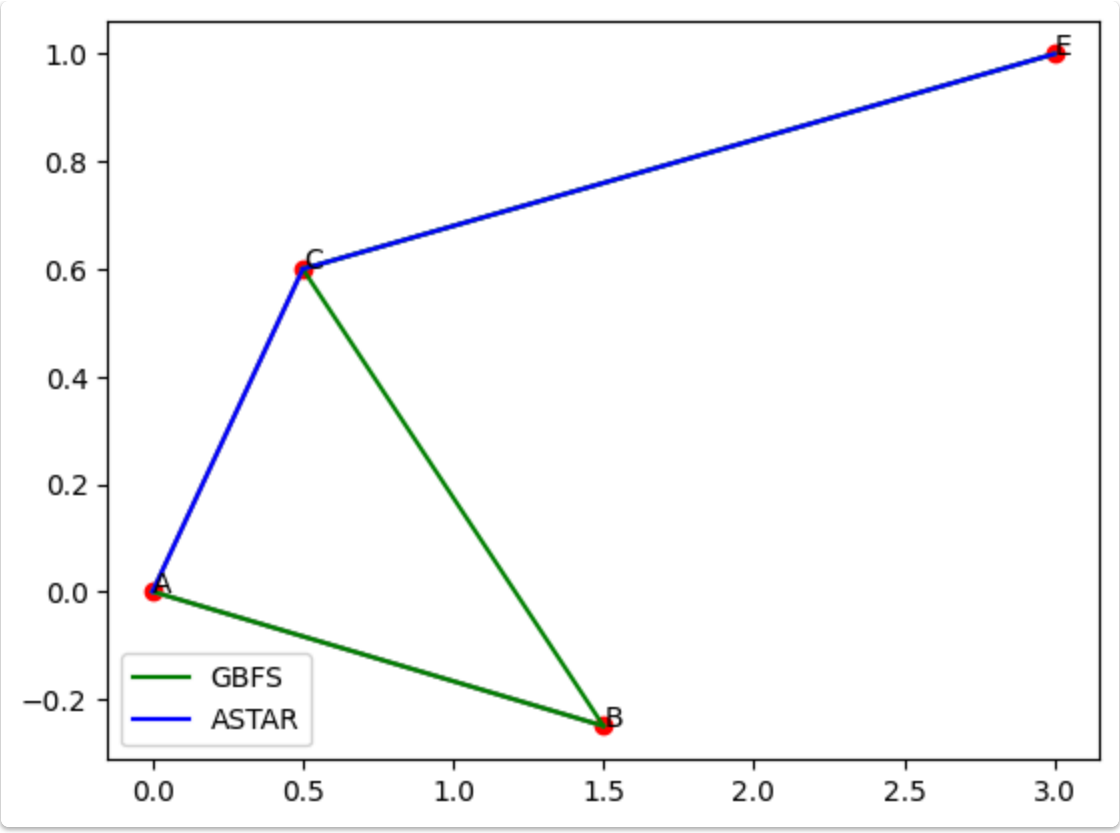
- **Khởi tạo:**
  - Hàng đợi ưu tiên chứa đỉnh 'A' với heuristic của nó.
  - `priorityQueue = [(heuristic['A'], 'A')]`
- **Vòng lặp đầu tiên:**
  - Lấy đỉnh 'A' ra khỏi hàng đợi (`current = 'A'`), thêm vào `path = ['A']`.
  - Đỉnh 'A' không phải là đỉnh mục tiêu ('E').
  - **Đặt lại hàng đợi ưu tiên:** `priorityQueue = queue.PriorityQueue()`
  - Thêm các đỉnh kề của 'A' vào hàng đợi nếu chưa có trong `path`.
    - Đỉnh 'B' (heuristic của 'B'), thêm vào hàng đợi.
    - Đỉnh 'C' (heuristic của 'C'), thêm vào hàng đợi.
  - Hàng đợi ưu tiên hiện tại chứa 'B' và 'C'.
- **Vòng lặp thứ hai:**
  - Lấy đỉnh có heuristic thấp nhất từ hàng đợi. Lúc này chính là B.
  - `current = 'B'`, thêm vào `path = ['A', 'B']`.
  - Đỉnh 'B' không phải là đỉnh mục tiêu.
  - **Đặt lại hàng đợi ưu tiên:** `priorityQueue = queue.PriorityQueue()`
  - Thêm các đỉnh kề của 'B' vào hàng đợi nếu chưa có trong `path`.
    - Đỉnh kề duy nhất của 'B' là 'A', nhưng 'A' đã có trong `path`, nên không thêm gì vào hàng đợi.
  - **Hàng đợi ưu tiên hiện tại trống.**
- **Vòng lặp kết thúc:**
  - Do hàng đợi ưu tiên trống, vòng lặp kết thúc.
  - Thuật toán trả về `path = ['A', 'B']`.

Và thuật toán A-star cũng được tính toán tương tự với f = g + cost, và dẫn đến kết quả mắc kẹt tại điểm B

Cách khắc phục

Bỏ đi dòng đặt lại hàng đợi trong thuật toán. Sau khi comment lại dòng đặt lại hàng đợi, kết quả đã tốt hơn:

```
GBFS => ['A', 'B', 'C', 'E']
ASTAR => ['A', 'C', 'E']
```



Hai thuật toán đã có thể tìm được đường đi tới điểm đích

3.2. Lỗi lưu trữ đường đi và không theo dõi chi phí tích lũy riêng cho từng đỉnh trong A-star

Trong phần này, em xin trình bày hai lỗi là:

- Lỗi lưu trữ đường đi và kiểm tra nút đã thăm bằng biến `path` trong GBFS và A-star.
- Lỗi cập nhật chi phí đường đi của thuật toán A-star

LƯU TRỮ ĐƯỜNG ĐI BẰNG BIẾN PATH

Em nhận thấy rằng biến `path` trong cả hai thuật toán có mục đích như sau: *lưu trữ đường đi (lưu các nút đã thăm theo thứ tự tạo thành đường đi từ nút bắt đầu tới nút hiện tại)* và *kiểm tra xem một nút đã được thăm hay chưa trước khi thêm vào hàng đợi*.

Em nghĩ nên tách riêng hai chức năng này ra thành 2 biến là biến `visited` để kiểm tra xem nút đã được thăm hay chưa trước khi thêm vào hàng đợi, và biến `path` để lưu trữ đường đi tối ưu bằng cách truy ngược cha từ nút đích đến.

Vậy code sau khi sửa của GBFS thầy/cô có thể tìm thấy trong notebook `filecode.ipynb` với chỉ mục thứ 6. Trong code đã tinh chỉnh lại vai trò của biến `path`, thêm biến `visited` và tái tạo đường đi từ `startNode` đến `endNode` bằng quan hệ cha-con.

Còn về code của A-star, em xin trình bày thêm lỗi sau và sửa một lần.

KHÔNG THEO DÕI CHI PHÍ TÍCH LŨY RIÊNG CHO TỪNG ĐỈNH TRONG A-STAR

Biến `distance` trong thuật toán A-star đã được sử dụng như sau:

```
distance = 0
...
distance += int(current[1])
```

Vấn đề ở đây là biến distance được cập nhật một cách toàn cục, cộng dồn chi phí mỗi khi thăm một node mới. Trong thuật toán A-star, mỗi node cần có một giá trị g riêng, đại diện cho chi phí tối ưu để tới được node đó kể từ node bắt đầu. Nên sử dụng distance một cách toàn cục như vậy thì không giải đúng với thuật toán.

GIẢI PHÁP: Em sẽ thêm một dictionary `gvalues` để lưu trữ giá trị g cho từng node trong đồ thị. Như vậy mỗi node sẽ có một giá trị g của riêng mình. Và khởi tạo nó với ban đầu với g của startNode = 0.

Và trong vòng for em sẽ cập nhật giá trị `g` của từng node như thế này:

```
for neighbor in graph[current[0]]:
        neighbor_node = neighbor[0]
        neighbor_cost = neighbor[1]

        tentative_g = g_values[current[0]] + int(neighbor_cost) # tính chi
phí từ current đến đỉnh kề của nó 1 cách tạm thời

        if neighbor_node in visited:
                # Nếu node đã được thăm ta bỏ qua, vì ta đã chứng minh được
rằng A-star sau khi thăm một đỉnh
                # thì đỉnh đó đã mang chi phí tối ưu nhất
                continue

        if neighbor_node not in g_values or tentative_g <
g_values[neighbor_node]:
                g_values[neighbor_node] = tentative_g
                f_value = tentative_g + heuristics[neighbor_node]
                priorityQueue.put((f_value, neighbor))
                parent[neighbor_node] = current[0] # thiết lập quan hệ cha
con
```

Toàn bộ code có thể được xem trong `filecode.ipynb`.

4. Nhận xét

Kết quả chạy tay và chạy bằng code của cả 2 thuật toán tương đồng nhau. Nhưng trong code của cô còn tiềm ẩn nhiều lỗi có thể làm cho thuật toán chạy sai.