BỘ GIÁO DỤC VÀ ĐÀO TẠO

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN – ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH

KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO ĐỒ ÁN MÔN HỌC

Đồ án:

XÂY DỰNG CÁC THUẬT TOÁN TÌM KIẾM

MÔN HỌC: CƠ SỞ TRÍ TUỆ NHÂN TẠO

LỚP: 18_22

GVHD: HOÀNG XUÂN TRƯỜNG

HỌ TÊN SINH VIÊN: HOÀNG TRUNG NAM

MSSV: 18120466

MỤC LỤC

PHẦN I: GIỚI THIỆU ĐỒ ÁN

CHƯƠNG I: GIỚI THIỆU ĐỒ ÁN

1.1. Mục tiêu đồ án.

- Nghiên cứu, cài đặt và trình bày các thuật toán tìm kiếm trên đồ thị.

1.2. Yêu cầu đồ án

- Project được thực hiện theo cá nhân. Thời gian và cách thức nộp, xem trên Moodle. Nội dung cần nộp: - Báo cáo trình bày trong file pdf chứa:
  • Thông tin sinh viên: họ tên, MSSV…
  • Mức độ hoàn thành của mỗi mức yêu cầu. Tự đánh giá đồ án trên thang điểm 10.
  • Trình bày lý thuyết cơ bản (ý tưởng, độ phức tạp, tính chất,...) của mỗi thuật toán cài đặt.
  • Trình bày điểm khác biệt giữa UCS và A*.
  • Điểm cộng nếu có thêm thuật toán Tìm kiếm khác ngoài 4 thuật toán BFS, DFS, UCS(Uniform Cost Search), A*.
  • Chi tiết mỗi thuật toán cài đặt:
    - Đưa ra các trường hợp(testcase) khác nhau của thuật toán, xét trường hợp đặc biệt (nếu có).
    - Có hình minh họa cho mỗi testcase
    - Nhận xét mỗi thuật toán. Đối với thuật toán A*, phải đưa ra heuristic, giải thích và nhận xét tại sao chọn hàm heuristic đó. –
- Source code - Video quay lại màn hình cho mỗi thuật toán của một testcase đã cài đặt được. Ví dụ: testcase 1 có ít nhất 4 video bfs, dfs, ucs, a_star.

CHƯƠNG II: MÔ TẢ YÊU CẦU

1.1. Một số yêu cầu khi làm bài

- Viết code của mỗi thuật toán vào các hàm tương ứng đã được định nghĩa.
- Không được chỉnh sửa tham số đầu vào của các hàm (BFS, DFS, UCS, AStar)
- Các màu thể hiện cho các trạng thái của đồ thị được ghi tại file node_color.py như sau:
  • Màu xanh dương: đỉnh đã duyệt qua
  • Màu đỏ: đỉnh vừa duyệt tới
  • Màu đen: đỉnh chưa được duyệt

- Màu vàng: đỉnh hiện tại
- Màu tím: đỉnh đích
- Màu cam: đỉnh bắt đầu
- Màu xanh lá: cạnh của đường đi tìm kiếm được
- Màu xám: đỉnh hoặc cạnh chưa duyệt qua
- Màu trắng: cạnh đã duyệt qua hoặc viền của đỉnh

PHẦN II: QUY TRÌNH THỰC HIỆN

CHƯƠNG I: CƠ SỞ LÝ THUYẾT CÁC THUẬT TOÁN

- Một số kí hiệu:
  - N: Số trạng thái trong bài toán
  - B: Thừa số phân nhánh trung bình (Số con trung bình) (B>1)
  - L: Độ dài đường đi từ đỉnh bắt đầu(start) đến đỉnh đích(goal) với số bước di chuyển ít nhất
  - LMAX: Độ dài đường đi không chu trình dài nhất từ start đến bất cứ đâu
  - Q: Kích cỡ hàng đời ưu tiên trung bình

1.1.Thuật toán Breadth First Search (BFS)

Ý tưởng	Tính chất	Độ phức tạp	Ưu điểm	Khuyết điểm
Sử dụng 2 stack. Stack1(queueNode) để lưu lại những node đã được duyệt đến ở mỗi lần duyệt, stack2 (queue) lưu danh sách đỉnh chờ duyệt vào stack2 chỉ lấy những giá trị chưa xuất hiện ở stack1(tránh lặp vô hạn). Duyệt lần lượt các đỉnh đầu trong stack2 (sau mỗi lần duyệt thì xóa đã duyệt) đến khi tìm được node đích hoặc stack2 rỗng thì dừng. Về đường đi khởi tạo một mảng markpath (mọi phần tử = -1). Giá trị ở vị trí thứ k(vertex) của markpath là đỉnh cha của đỉnh k(adjacency_node).	Tìm kiếm đường đi từ một đỉnh gốc cho trước tới một đỉnh đích, và tìm kiếm đường đi từ đỉnh gốc tới tất cả các đỉnh khác.	Thời gian: O(min(N, B^L)) Không gian: O(min(N, B^L))	Xét duyệt tất cả các đỉnh để trả về kết quả. Nếu số đỉnh là hữu hạn, thuật toán chắc chắn tìm ra kết quả.	Mang tính chất mù quáng, duyệt tất cả đỉnh, không chú ý đến thông tin trong các đỉnh để duyệt hiệu quả, dẫn đến duyệt qua các đỉnh không cần thiết.

1.2.Thuật toán Depth First Search (DFS)

Ý tưởng	Tính chất	Độ phức tạp	Ưu điểm	Khuyết điểm
Sử dụng 2 stack. Stack1(queueNode) để lưu lại những node đã được duyệt đến ở mỗi lần duyệt, stack2 (queue) lưu danh sách đỉnh chờ duyệt vào stack2 chỉ lấy những giá trị chưa xuất hiện ở stack1(tránh lặp vô hạn). Duyệt lần lượt các đỉnh cuối trong stack2 (sau mỗi lần duyệt thì xóa đã duyệt) đến khi tìm được node đích hoặc stack2 rỗng thì dừng. Về đường đi khởi tạo một mảng markpath(mọi phần tử = -1). Giá trị ở vị trí thứ k(vertex) của markpath là đỉnh cha của đỉnh k(adjacency_node).	Thuật toán khởi đầu tại gốc (hoặc chọn một đỉnh nào đó coi như gốc) và phát triển xa nhất có thể theo mỗi nhánh.	Thời gian: O(B^{LMAX}) Không gian: O(LMAX)	Xét duyệt tất cả các đỉnh để trả về kết quả. Nếu số đỉnh là hữu hạn, thuật toán chắc chắn tìm ra kết quả.	Mang tính chất mù quáng, duyệt tất cả đỉnh, không chú ý đến thông tin trong các đỉnh để duyệt hiệu quả, dẫn đến duyệt qua các đỉnh không cần thiết.

1.3.Thuật toán Uniform Cost Search (UCS)

Ý tưởng	Tính chất	Độ phức tạp	Ưu điểm	Khuyết điểm
Lấy trọng số của 2 đỉnh là khoảng cách của 2 đỉnh trong mặt phẳng tọa độ. Sử dụng 2 stack. Stack1 (queueNode) để lưu lại những node đã được duyệt đến ở mỗi lần duyệt, stack2 (queue) lưu bộ(tuple) gồm danh sách đỉnh chờ duyệt và chi phí từ đỉnh gốc đến đỉnh đó stack2 chỉ lấy những giá trị chưa xuất hiện ở stack1(tránh lặp vô hạn). Sau mỗi lần duyệt xong các node kề ta sắp xếp lại stack theo thứ tự tăng dần vào. Duyệt lần lượt các đỉnh đầu trong stack2 (sau mỗi lần duyệt thì xóa đã duyệt) đến khi stack2 rỗng thì dừng. Về đường đi khởi tạo một mảng markpath (mọi phần tử = -1). Giá trị ở vị trí thứ k(vertex) của markpath là đỉnh cha của đỉnh k(adjacency_node).	Thuật toán sẽ trở thành thuật toán BFS nếu giá trị trọng số ở mỗi cạnh là như nhau, thuật toán UCS có trọng số không tìm đường đi có chi phí nhỏ nhất nhưng là thuật toán tìm số bước đi ít nhất.	Thời gian: O(log(Q) * min(N, B^L)) Không gian: O(min(N, B^L))	Tìm số bước đi ít nhất từ một đỉnh tới một đỉnh bất kì.	Không tìm ra đường đi có chi phí nhỏ nhất đến đích.

1.4.Thuật toán A*

Ý tưởng	Tính chất	Độ phức tạp	Ưu điểm	Khuyết điểm
Lấy trọng số của 2 đỉnh là khoảng cách của 2 đỉnh trong mặt phẳng tọa độ. Sử dụng 2 stack. Stack1 (queueNode) để lưu lại những node đã được duyệt đến ở mỗi lần duyệt, stack2 (queue) lưu bộ(tuple) gồm danh sách đỉnh chờ duyệt và chi phí từ đỉnh gốc đến đỉnh đó + chi phí từ đỉnh đó đến đỉnh gốc theo đường "chim bay"(Hàm heuristic) stack2 chỉ lấy những giá trị chưa xuất hiện ở stack1(tránh lặp vô hạn). Sau mỗi lần duyệt xong các node kề ta sắp xếp lại stack theo thứ tự tăng dần vào. Duyệt lần lượt các đỉnh đầu trong stack2 (sau mỗi lần duyệt thì xóa đã duyệt) đến khi stack2 rỗng thì dừng. Về đường đi khởi tạo một mảng markpath (mọi phần tử = -1). Giá trị ở vị trí thứ k(vertex) của markpath là đỉnh cha của đỉnh k(adjacency_node).	Thuật toán được biết đến rộng rãi để tìm chi phí nhỏ nhất với một Heuristic phù hợp. Với từng bài toán ta cần có một Heuristic xử lí riêng.	Phụ thuộc vào hàm Heuristic	Một thuật giải linh động, tổng quát, trong đó hàm chứa cả tìm kiếm chiều sâu, tìm kiếm chiều rộng và những nguyên lý Heuristic khác. Nhanh chóng tìm đến lời giải với sự định hướng của hàm Heuristic. Chính vì thế mà người ta thường nói A* chính là thuật giải tiêu biểu cho Heuristic.	A* rất linh động nhưng vẫn gặp một khuyết điểm cơ bản giống như chiến lược tìm kiếm chiều rộng, đó là tốn khá nhiều bộ nhớ để lưu lại những trạng thái đã đi qua. Mỗi bài toán cần phải có từng Heuristic phù hợp riêng để xử lý. Nếu không phù hợp sẽ không giải quyết được bài toán

CHƯƠNG II: CHI TIẾT MỖI THUẬT TOÁN CÀI ĐẶT

1.1. Thuật toán Breadth First Search (BFS)

```
def BFS(graph, edges, edge_id, start, goal):
    """
    BFS search
    """
    # TODO: your code
    k = 0
    mark_path = [] #mảng lấy vết đường đi
    initVertexCrossed(graph, mark_path) # hàm khởi tạo mả_path = -1
    queue = [] # stack2: lưu đỉnh chờ duyệt
    queueNode = [] #stack1: lưu những đỉnh đã duyệt qua
    queue.append(start)
    while(len(queue) !=0):
        vertex = queue[0]
        queue.remove(queue[0]) # Xóa đỉnh đầu sau mỗi lần duyệt
        queueNode.append(vertex)
        graph[vertex][3] = yellow
        node_1 = graph[vertex]
        for adjacency_node in node_1[1]:
            if adjacency_node not in queueNode: #queueNode kiểm tra đỉnh đã có trong danh sách chưa?
                queue.append(adjacency_node) #Thêm vào nếu chưa có trong danh sách
                graph[adjacency_node][3] = red # Tô màu đỉnh duyệt đến
                mark_path[adjacency_node] = vertex #Đánh dấu dường đi của các đỉnh
                edges[edge_id(vertex, adjacency_node)][1] = white #Tô màu cạnh đã duyệt qua

            queueNode.append(adjacency_node)
                        graphUI.updateUI()
            time.sleep(.1)
            if goal in queue:
                k = 1
                break
        graphUI.updateUI()
        time.sleep(1)
        graph[vertex][3] = blue
        if k == 1:
            break
    fillNode(graph, vertex, start, goal) # Hàm tô màu đỉnh sau khi kết thúc duyệt
    path = []
    colorPath(mark_path, path, edges, edge_id, start, goal) #Tô mày đường đi
    print("Implement BFS algorithm.")
    pass
```

1.2. Thuật toán Depth First Search (DFS)

```
def DFS(graph, edges, edge_id, start, goal):
    """
    DFS search
    """
    # TODO: your code
    pos = -1

    k = 0
    mark_path = []
    initVertexCrossed(graph, mark_path)
    stack = []
    queueNode = []
    stack.append(start)
    while (stack):
        vertex = stack.pop() #Lấy đỉnh cuối ra để duyệt đồng thời xóa khỏi stack
        queueNode.append(vertex)
        graph[vertex][3] = yellow
        node_1 = graph[vertex]
        time.sleep(.3)
        for adjacency_node in node_1[1]:
            if adjacency_node not in queueNode:
                stack.append(adjacency_node)
                graph[adjacency_node][3] = red
                mark_path[adjacency_node] = vertex
                edges[edge_id(vertex, adjacency_node)][1] = white
            queueNode.append(adjacency_node)
            graphUI.updateUI()
            if goal in stack:
                k = 1
                break
        time.sleep(1)
        graph[vertex][3] = blue
        if k == 1:
            break
    fillNode(graph, vertex, start, goal)
    path = []
    colorPath(mark_path, path, edges, edge_id, start, goal)
    print("Implement DFS algorithm.")
    pass
```

1.3. Thuật toán Uniform Cost Search (UFS)

```
def UCS(graph, edges, edge_id, start, goal):
    """
    Uniform Cost Search search
    """
    # TODO: your code
    k = 0
    mark_path = []
    initVertexCrossed(graph, mark_path)
    queue = []
    queueNode = []
    a = (start, 0)
    queue.append(a)
    while (len(queue) != 0): #Kiểm tra stack rỗng thì ngưng
        vertex = queue[0]
        queue.remove(queue[0])
        queueNode.append(vertex[0])
        graph[vertex[0]][3] = yellow
        node_1 = graph[vertex[0]]
        for adjacency_node in node_1[1]:
            if adjacency_node not in queueNode:
                a = int(getWeight(graph, vertex[0], adjacency_node))#Chi phí từ đỉnh gốc tới đỉnh kề đang xét
                queue.append((adjacency_node, a + vertex[1]))#Bộ đỉnh, chi phí đường đi giữa 2 đỉnh
                graph[adjacency_node][3] = red
                mark_path[adjacency_node] = vertex[0]
                edges[edge_id(vertex[0], adjacency_node)][1] = white
            sortTuple(queue)
            queueNode.append(adjacency_node)
                        graphUI.updateUI()
            time.sleep(.1)
            if goal == adjacency_node:
                k = 1
                break
        graphUI.updateUI()
        time.sleep(1)
        graph[vertex[0]][3] = blue
        if k == 1:
            break
    fillNode(graph, vertex[0], start, goal)
    path = []
    colorPath(mark_path, path, edges, edge_id, start, goal)
    print("Implement Uniform Cost Search algorithm.")
    pass
```

1.4. Thuật toán A*

```
def AStar(graph, edges, edge_id, start, goal):
    """
    A star search
    """
    # TODO: your code

    weight = []
    wei1 = 0
    initWeighAF(graph, weight, goal)
    k = 0
    mark_path = []
    initVertexCrossed(graph, mark_path)
    queue = []
    queueNode = []
    a = (start, 0)
    queue.append(a)
    while (len(queue) != 0):
        vertex = queue[0]
        queue.remove(queue[0])
        queueNode.append(vertex[0])
        graph[vertex[0]][3] = yellow
        node_1 = graph[vertex[0]]
        for adjacency_node in node_1[1]:
            if adjacency_node not in queueNode:
                a = int(getWeight(graph, vertex[0], adjacency_node))#Chi phí từ đỉnh gốc tới đỉnh kề đang xét
                b = int(weight[adjacency_node]) # Heuristic chi phí từ đỉnh kề đang xét tới đích
                # Bộ đỉnh, chi phí đường đi giữa 2 đỉnh và chi phí từ đỉnh đó tới đích
                queue.append((adjacency_node, a + b + vertex[1]))
                graph[adjacency_node][3] = red
                mark_path[adjacency_node] = vertex[0]
                edges[edge_id(vertex[0], adjacency_node)][1] = white
            sortTuple(queue)
            queueNode.append(adjacency_node)
                        graphUI.updateUI()

            time.sleep(.1)
            if goal == adjacency_node:
                k = 1
                break
        graphUI.updateUI()
        time.sleep(1)
        graph[vertex[0]][3] = blue
        if k == 1:
            break
    fillNode(graph, vertex[0], start, goal)
    path = []
    colorPath(mark_path, path, edges, edge_id, start, goal)
    print("Implement A* algorithm.")
    pass
```

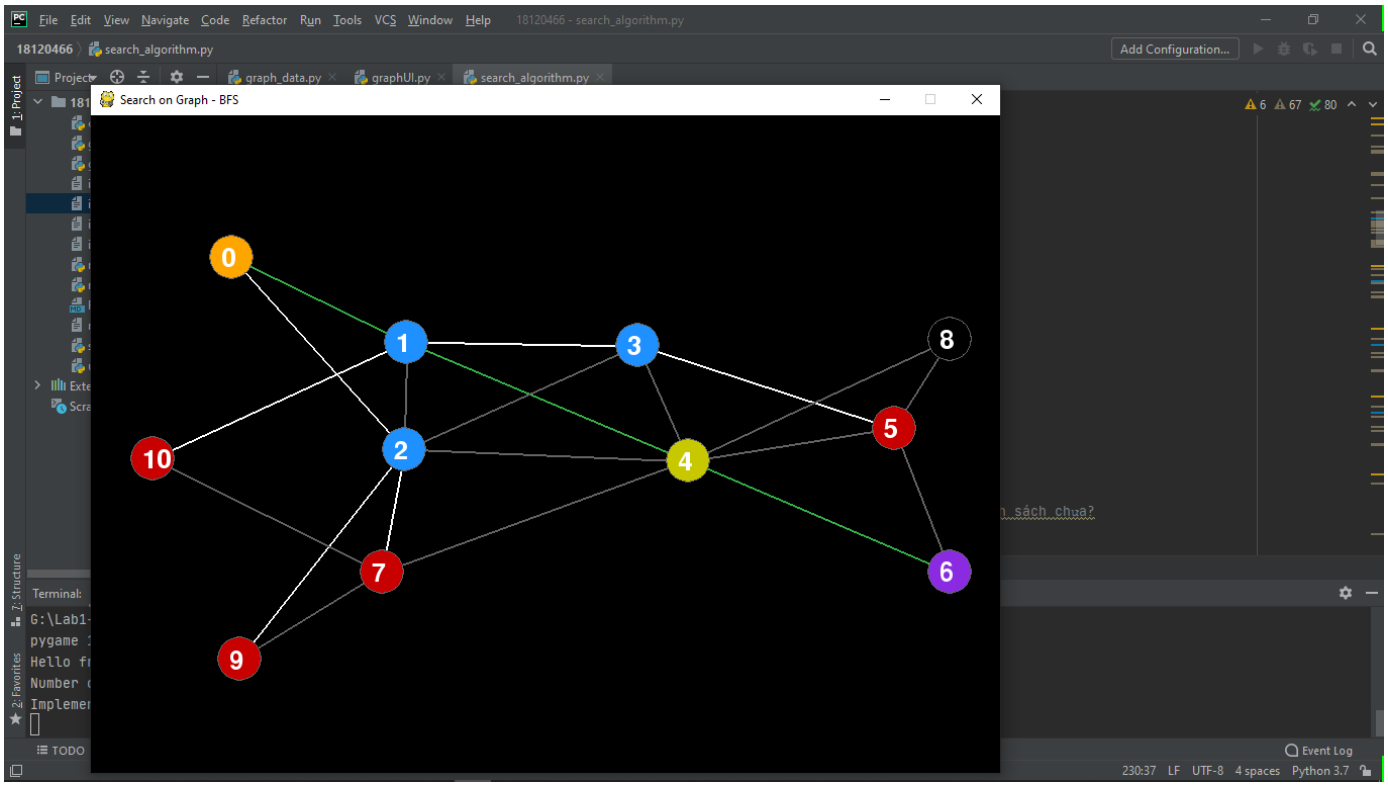
1.5. Một số hàm thêm để tính toán

```
# Hàm tô màu các đỉnh khi chương trình kết thúc
def fillNode(graph, vertex, start, goal):
    graph[vertex][3] = yellow
    graph[start][3] = orange
    graph[goal][3] = purple
    graphUI.updateUI()

# Hàm tạo mảng mark_path = -1
def initVertexCrossed(graph, path):
    for i in range(len(graph)):
        path.append(-1)
#Hàm tô màu đường đi
def colorPath(mark_path, path, edges, edge_id, start, goal):
    for i in range(len(mark_path)):
        for j in range(len(mark_path)):
            if j == goal:
                path.append(goal)
                goal = mark_path[j]
                break
        if goal == start:
            path.append(start)
            break
    for i in range(len(path) - 1):
        edges[edge_id(path[i], path[i + 1])][1] = green
    graphUI.updateUI()

# Hàm heuristic chi phí tới đỉnh gốc của mỗi đỉnh theo đường "chim bay"
def getWeight(graph, a, b):
    (x1, y1) = graph[a][0]
    (x2, y2) = graph[b][0]
    dist = math.sqrt(pow(x1-x2, 2) + pow(y1-y2, 2))
    return dist
    pass
# Sắp xếp bộ đỉnh, đường đi từ gốc tới đỉnh tăng dần
def sortTuple(tup):
    for i in range(len(tup)):
        for j in range(len(tup)):
            if tup[i][1] < tup[j][1]:
                temp = tup[i]
                tup[i] = tup[j]
                tup[j] = temp
def initWeighAF(graph, weight, goal):
    for i in range(len(graph)):
        weight.append(i)
    for i in range(len(graph)):
        if i != goal:
            weight[i] = getWeight(graph, i, goal)
```

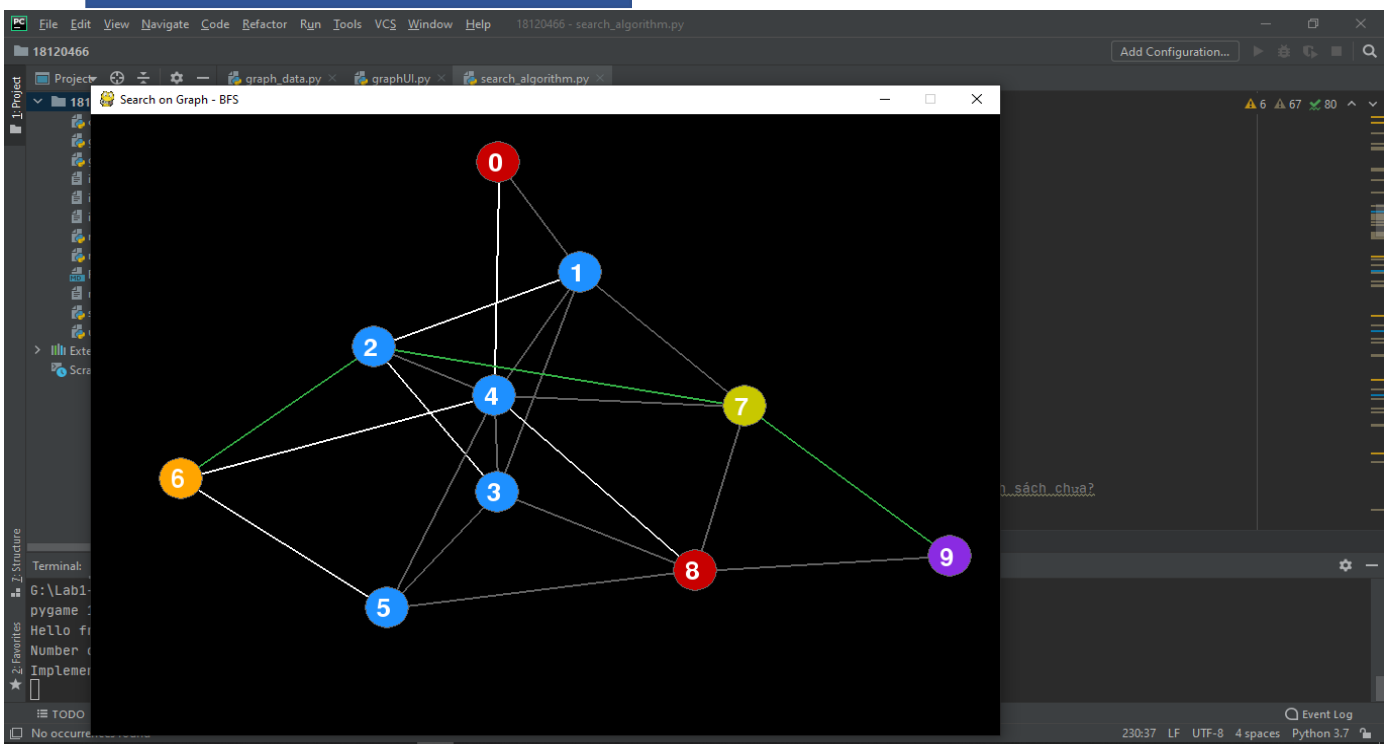
CHƯƠNG III: CÁC TESTCASE TRONG CÁC THUẬT TOÁN

1.1.Thuật toán Breadth First Search (BFS)

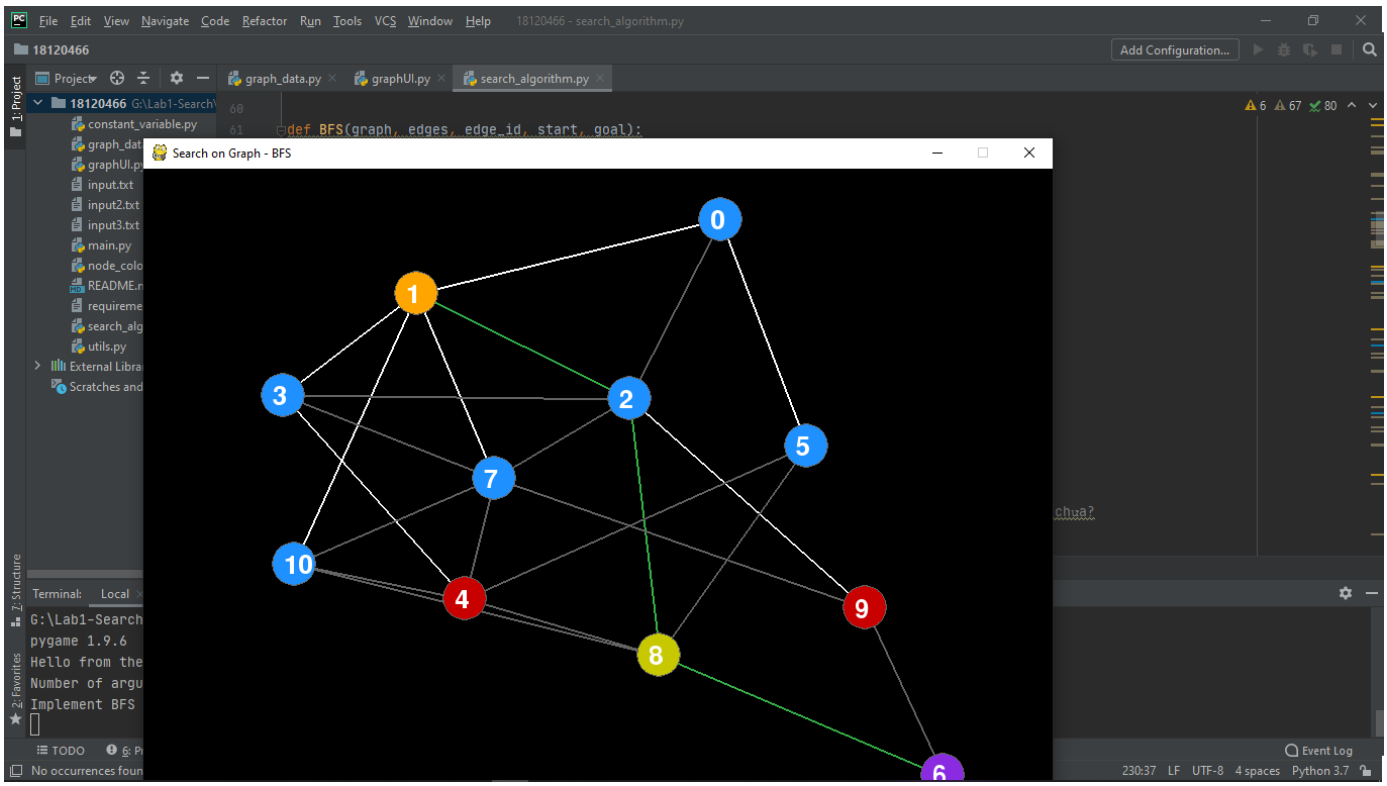
- **Testcase 1: file input.txt**
  - **Đỉnh bắt đầu: 0**
  - **Đỉnh kết thúc: 6**



- **Testcase 2: file input2.txt**
  - **Đỉnh bắt đầu: 6**
  - **Đỉnh kết thúc: 9**

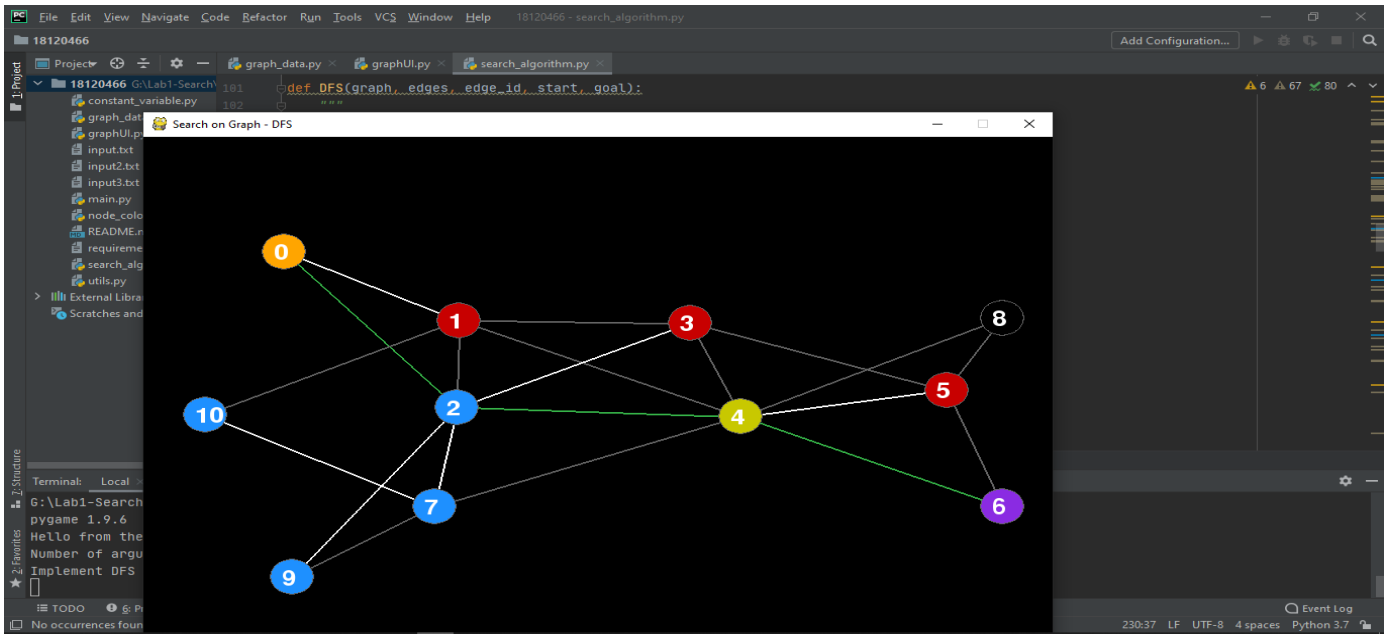


- **Testcase 3: file input3.txt**
  - **Đỉnh bắt đầu: 1**
  - **Đỉnh kết thúc: 6**

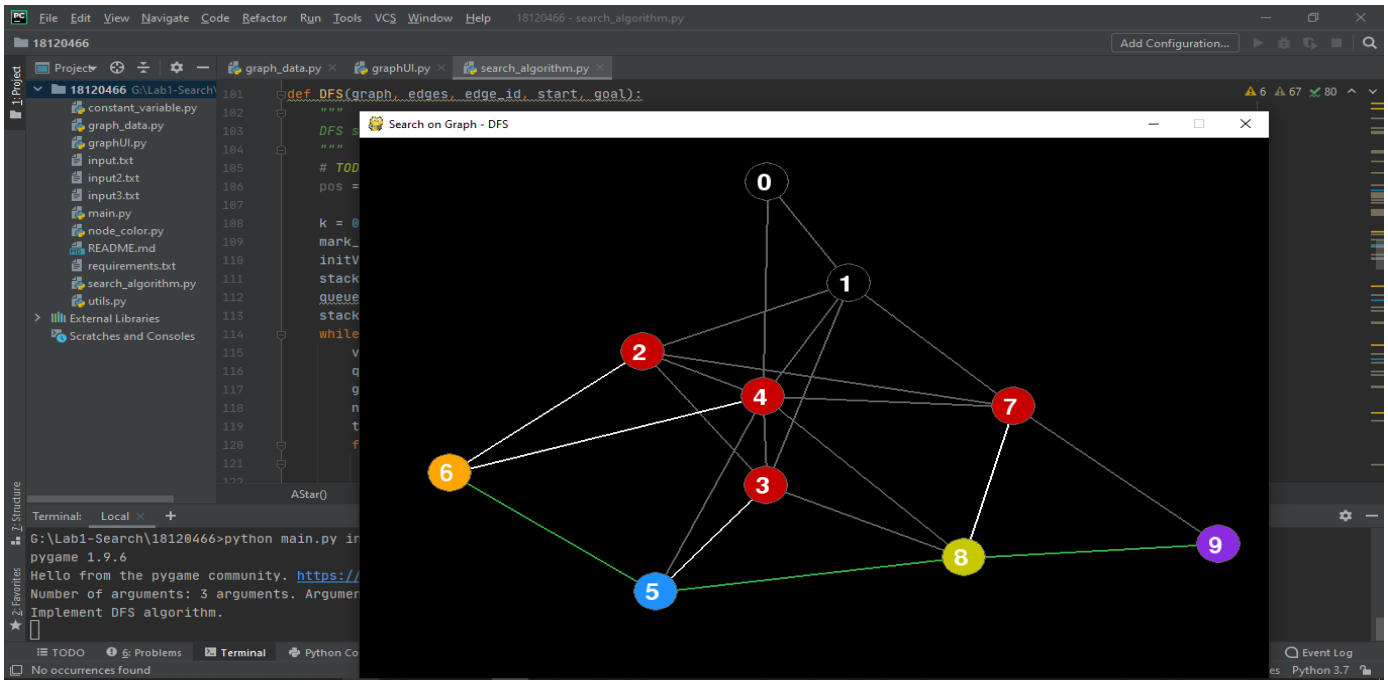


1.2.Thuật toán Depth First Search (DFS)

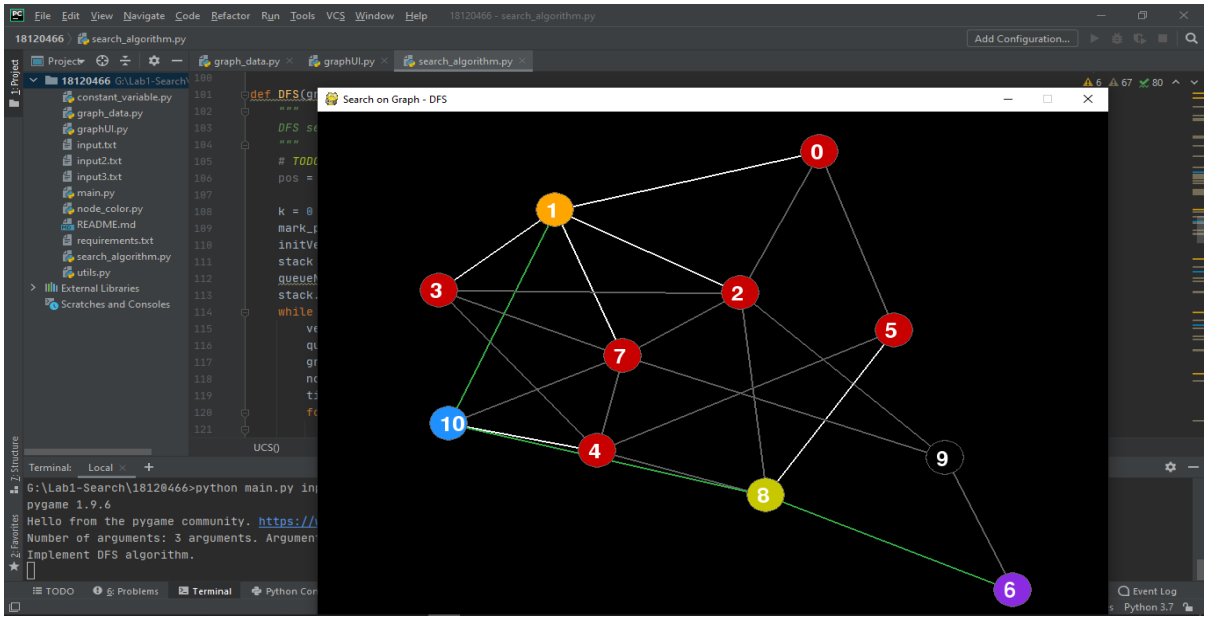
- **Testcase 1: file input.txt**
  - **Đỉnh bắt đầu: 0**
  - **Đỉnh kết thúc: 6**



- **Testcase 2: file input2.txt**
  - **Đỉnh bắt đầu: 6**
  - **Đỉnh kết thúc: 9**

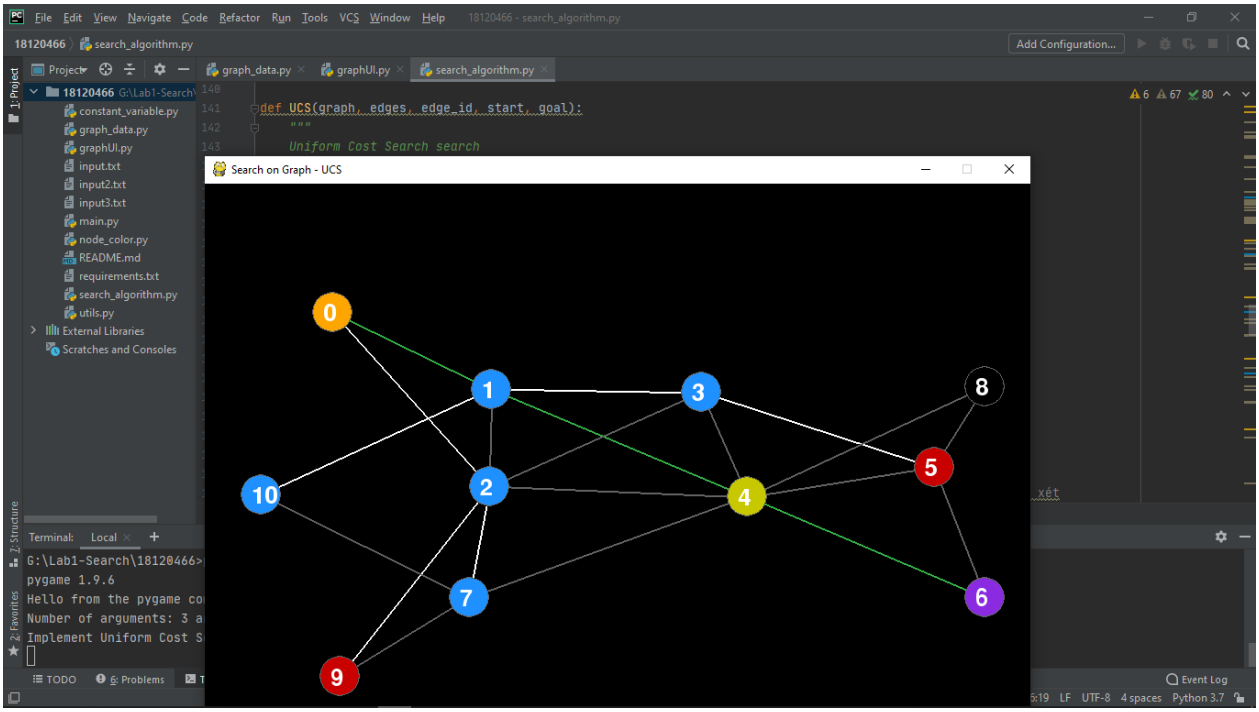


- **Testcase 3: file input3.txt**
  - **Đỉnh bắt đầu: 1**
  - **Đỉnh kết thúc: 6**

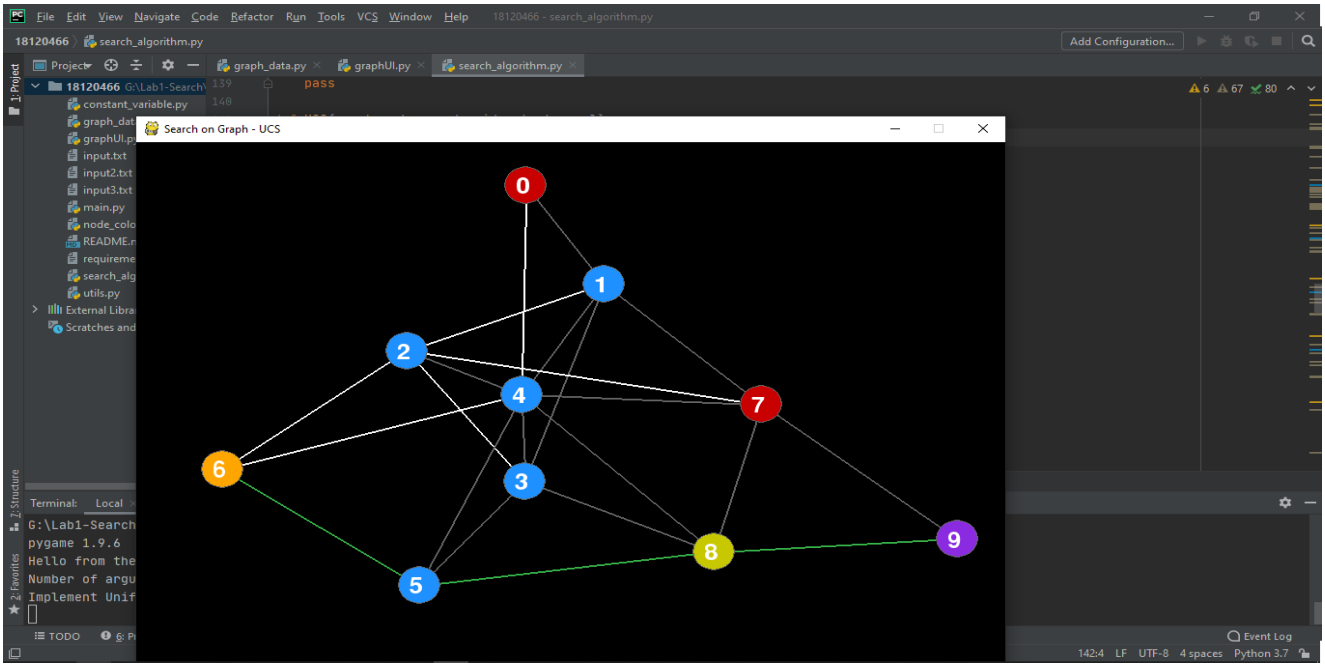


1.3.Thuật toán Uniform Cost Search (UCS)

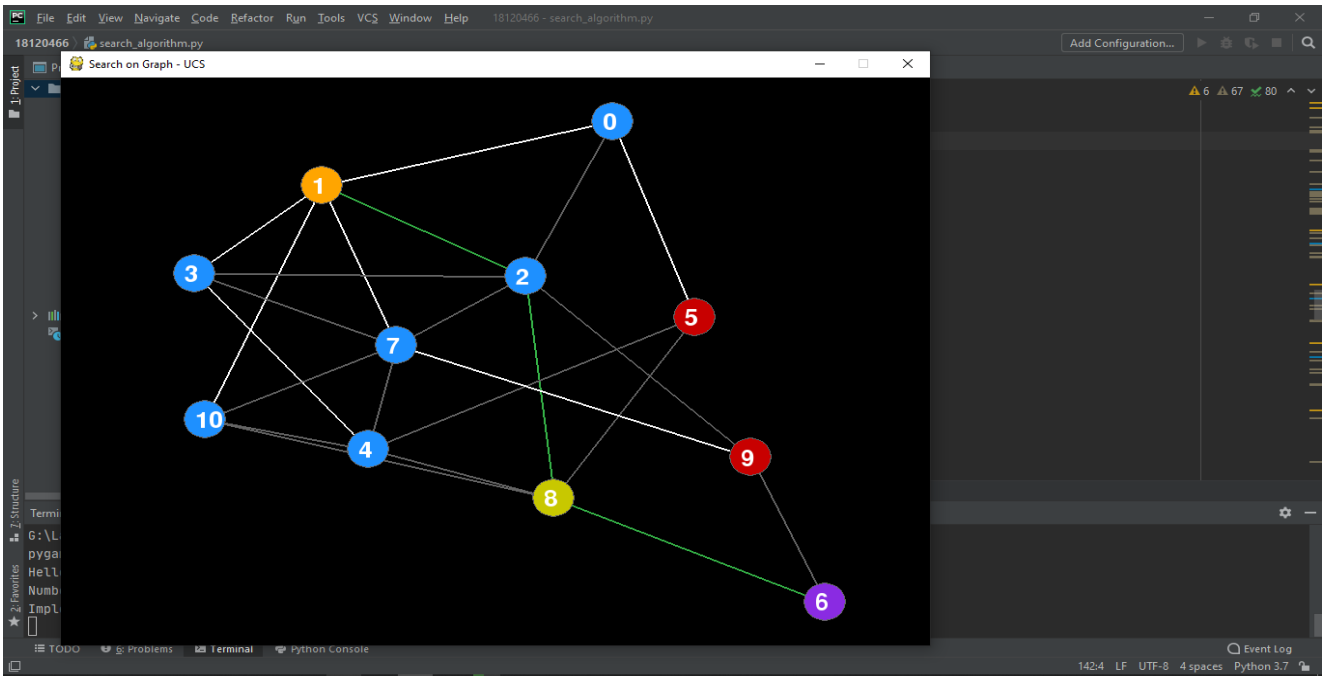
- **Testcase 1: file input.txt**
  - **Đỉnh bắt đầu: 0**
  - **Đỉnh kết thúc: 6**



- **Testcase 2: file input2.txt**
  - **Đỉnh bắt đầu: 6**
  - **Đỉnh kết thúc: 9**

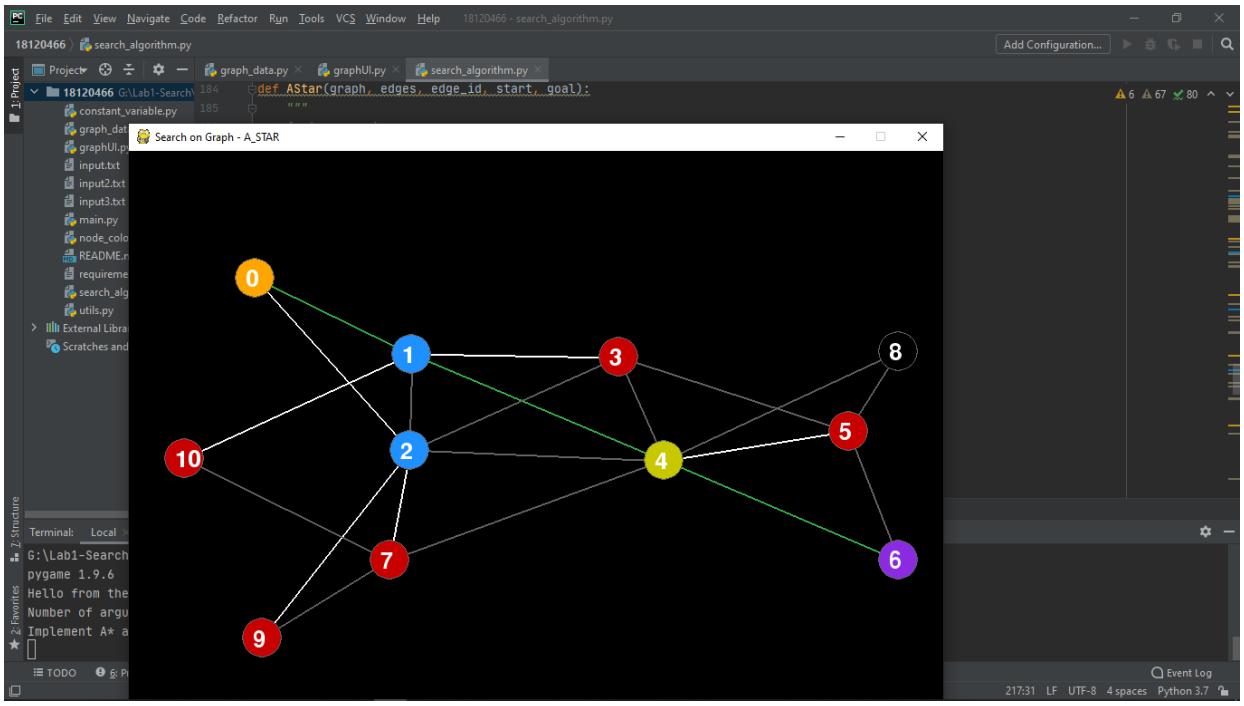


- **Testcase 3: file input3.txt**
  - **Đỉnh bắt đầu: 1**
  - **Đỉnh kết thúc: 6**

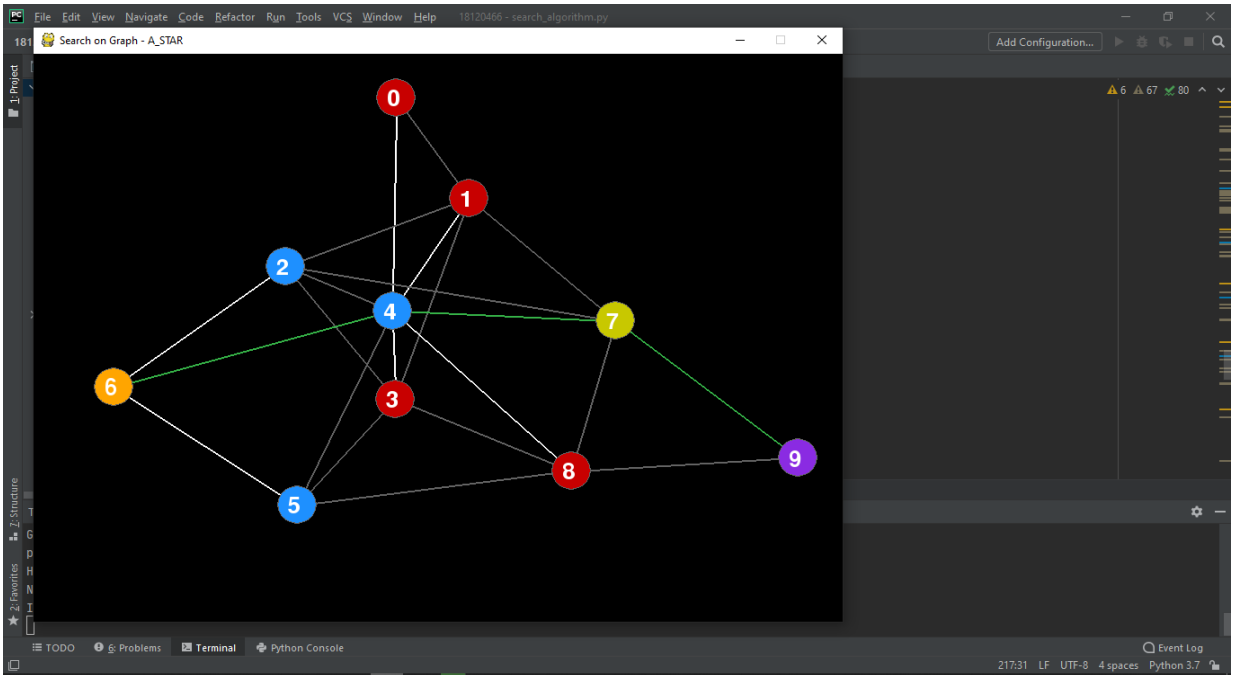


1.4.Thuật toán A*

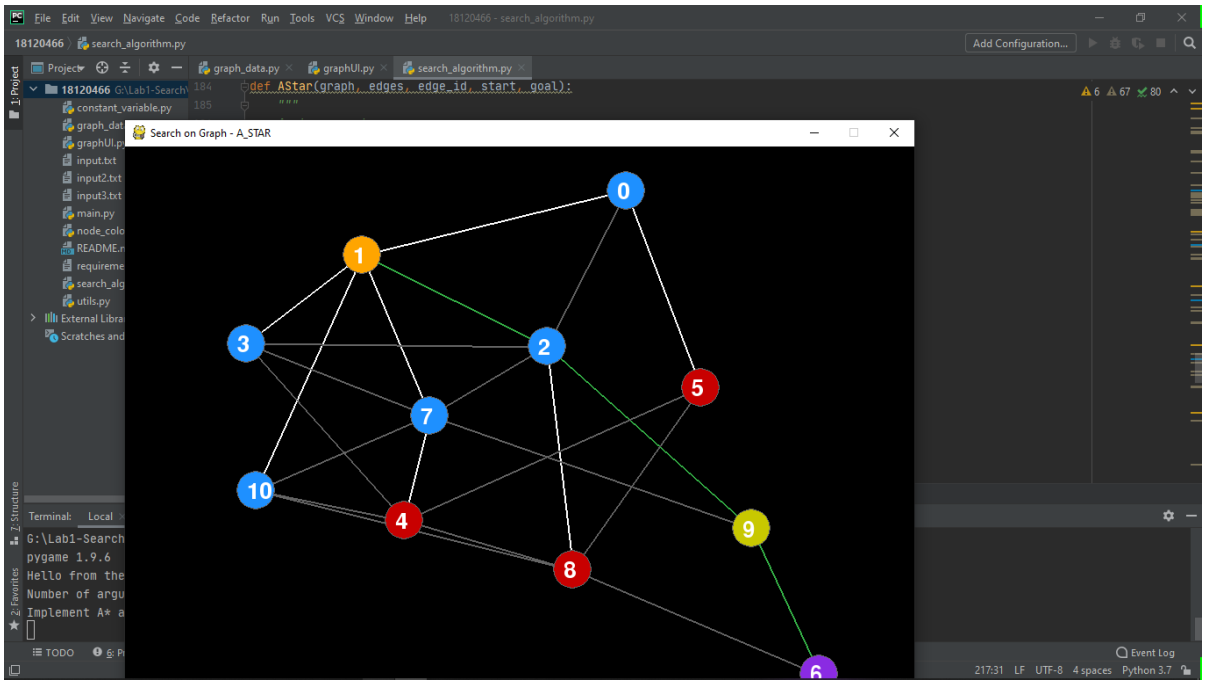
- **Testcase 1: file input.txt**
  - **Đỉnh bắt đầu: 0**
  - **Đỉnh kết thúc: 6**



- **Testcase 2: file input2.txt**
  - **Đỉnh bắt đầu: 6**
  - **Đỉnh kết thúc: 9**



- **Testcase 3: file input3.txt**
  - **Đỉnh bắt đầu: 1**
  - **Đỉnh kết thúc: 6**



PHẦN III: NHẬN XÉT - KẾT LUẬN

CHƯƠNG I: ĐÁNH GIÁ BÀI LÀM

1.1.Tự đánh giá đồ án trên thang điểm 10

- **Hoàn thành hầu hết yêu cầu + thêm thuật toán Greedy Best First Search: 10điểm**

1.2.So sánh UCS và A*

UCS	A*
Tìm số bước đi ít nhất nhưng không tìm ra đường đi có chi phí nhỏ nhất đến đích.	Tìm ra đường đi có chi phí nhỏ nhất nhưng không tìm được số bước đi ít nhất đến đích.
Xử lí khá tốt trên nhiều bài toán tìm kiếm. Vì là một thuật toán tìm kiếm tổng quát.	Cần có một hàm Heuristic phù hợp cho mỗi bài toán khác nhau.

1.3.Lý do chọn Heuristic ở A*

- Em chọn Heuristic ở hàm A* là vì tọa đang xử lí trên mặt phẳng tọa độ không gian O(x, y) thì khoảng cách giữu 2 đỉnh là trọng số phù hợp cho 2 đỉnh. Vì thê Heuristic đánh giá khoảng cách theo đường "chim bay" giữa từ mỗi đỉnh tới đỉnh gốc phù hợp cho bài toán tìm kiếm đường đi này do mọi thông số đều dựa trên miền không gian O(x, y).

CHƯƠNG II: TÀI LIỆU THAM KHẢO

1.1.Liên kết

- BFS: https://medium.com/@manhtandthu/thu%E1%BA%ADt-to%C3%A1n-t%C3%ACm-ki%E1%BA%BFm-theo-chi%E1%BB%81u-r%E1%BB%99ng-73f6f17c06eb
- DFS: https://medium.com/@manhtandthu/thu%E1%BA%ADt-to%C3%A1n-dfs-c-python-df109b51e129
- A*: https://www.stdio.vn/giai-thuat-lap-trinh/thuat-giai-a-DVnHj

1.2.Sách: Cơ sở trí tuệ nhân tạo(Lê Hoài Bắc – Tô Hoài Việt

CHƯƠNG III: THUẬT TOÁN THÊM

1.1. Greedy Best-First Search

1.1.1. Cơ sở lý thuyết

Ý tưởng	Tính chất	Độ phức tạp	Ưu điểm	Khuyết điểm
Lấy trọng số của 2 đỉnh là khoảng cách của 2 đỉnh trong mặt phẳng tọa độ. Sử dụng 2 stack. Stack1 (queueNode) để lưu lại những node đã được duyệt đến ở mỗi lần duyệt, stack2 (queue) lưu bộ(tuple) gồm danh sách đỉnh chờ duyệt và chi phí từ đỉnh đó đến đỉnh gốc theo đường "chim bay"(Hàm heuristic) stack2 chỉ lấy những giá trị chưa xuất hiện ở stack1(tránh lặp vô hạn). Sau mỗi lần duyệt xong các	Thuật toán được biết đến rộng rãi để tìm chi phí nhỏ nhất với một Heuristic phù hợp. Với từng bài toán ta cần có một Heuristic xử lí riêng.	Phụ thuộc vào hàm Heuristic	Một thuật giải linh động, tổng quát, trong đó hàm chứa cả tìm kiếm chiều sâu, tìm kiếm chiều rộng và những nguyên lý Heuristic khác. Nhanh chóng tìm đến lời giải với sự định hướng của hàm Heuristic. Chính vì thế mà người ta thường nói A* chính là thuật giải tiêu biểu cho Heuristic.	Khuyết điểm cơ bản giống như chiến lược tìm kiếm chiều rộng, đó là tốn khá nhiều bộ nhớ để lưu lại những trạng thái đã đi qua. Mỗi bài toán cần phải có từng Heuristic phù hợp riêng để xử lý. Nếu không phù hợp sẽ không giải quyết được bài toán.

node kề ta sắp xếp lại stack theo thứ tự tăng dần vào. Duyệt lần lượt các đỉnh đầu trong stack2 (sau mỗi lần duyệt thì xóa đã duyệt) đến khi stack2 rỗng thì dừng. Về đường đi khởi tạo một mảng markpath (mọi phần tử = -1). Giá trị ở vị trí thứ k(vertex) của markpath là đỉnh cha của đỉnh k(adjacency_node).				

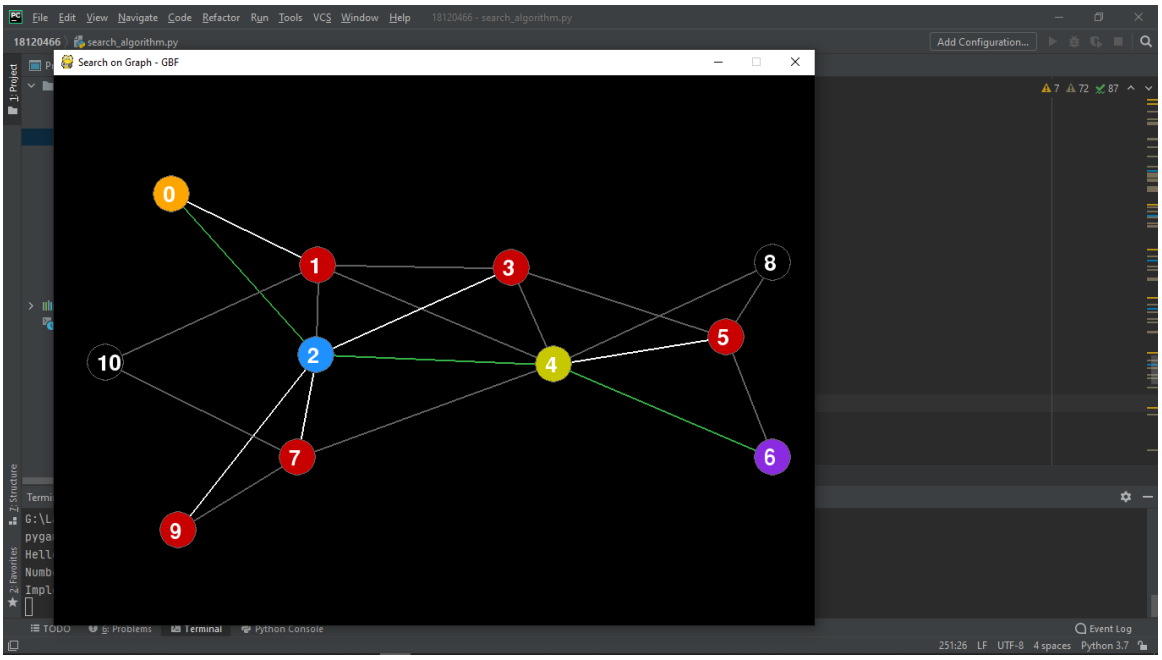
1.1.2. Chi tiết cài đặt

```
2.  def GBF(graph, edges, edge_id, start, goal):
        """
        A star search
        """
        # TODO: your code

        weight = []
        wei1 = 0
        initWeighAF(graph, weight, goal)
        k = 0
        mark_path = []
        initVertexCrossed(graph, mark_path)
        queue = []
        queueNode = []
        a = (start, 0)
        queue.append(a)
        while (len(queue) != 0):
            vertex = queue[0]
            queue.remove(queue[0])
            queueNode.append(vertex[0])
            graph[vertex[0]][3] = yellow
            node_1 = graph[vertex[0]]
            for adjacency_node in node_1[1]:
                if adjacency_node not in queueNode:
                    b = int(weight[adjacency_node]) # Heuristic chi phí từ đỉnh kề đang xét tới đích
                    # Bộ đỉnh, chi phí đường đi từ đỉnh đó tới đích
                    queue.append((adjacency_node, b))
                    graph[adjacency_node][3] = red
                    mark_path[adjacency_node] = vertex[0]
                    edges[edge_id(vertex[0], adjacency_node)][1] = white
                sortTuple(queue)
```

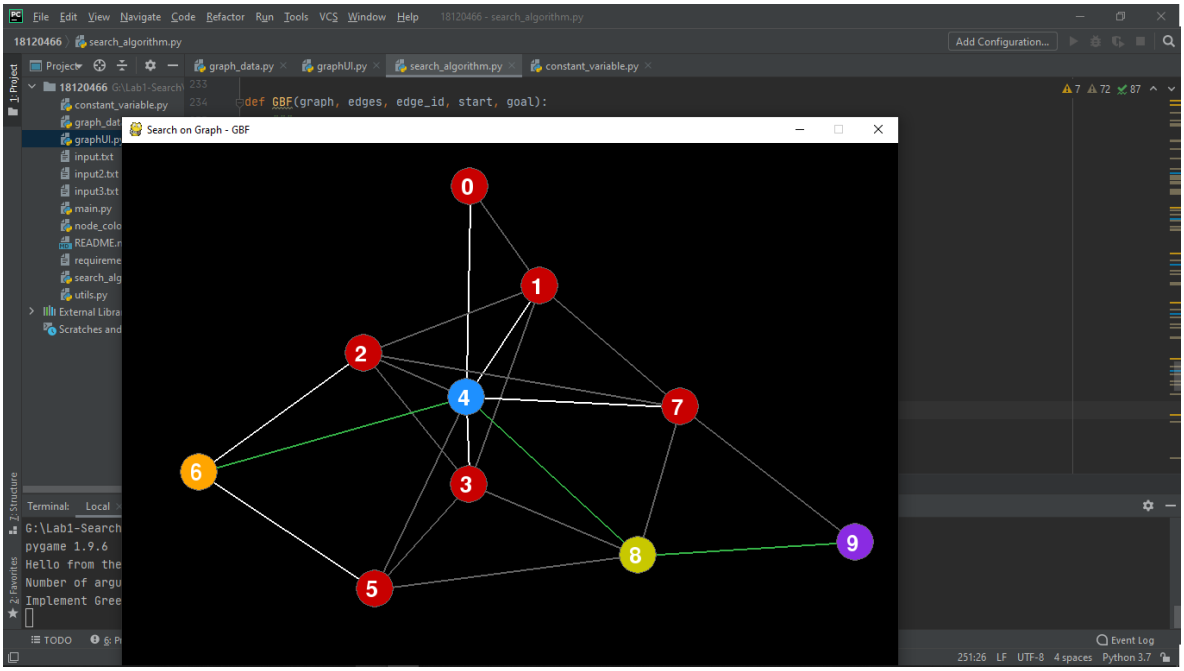
```
            queueNode.append(adjacency_node)
            graphUI.updateUI()
            time.sleep(.1)
            if goal == adjacency_node:
                k = 1
                break
        graphUI.updateUI()
        time.sleep(1)
        graph[vertex[0]][3] = blue
        if k == 1:
            break
    fillNode(graph, vertex[0], start, goal)
    path = []
    colorPath(mark_path, path, edges, edge_id, start, goal)
    print("Implement Greedy Best-First Search algorithm.")
    pass
```

2.1.1. Testcase

- **Testcase 1: file input.txt**
  - **Đỉnh bắt đầu: 0**
  - **Đỉnh kết thúc: 6**



- **Testcase 2: file input.txt**
  - **Đỉnh bắt đầu: 6**
  - **Đỉnh kết thúc: 9**



- **Testcase 3: file input.txt**
  - **Đỉnh bắt đầu: 1**
  - **Đỉnh kết thúc: 6**