TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

BÁO CÁO ĐỒ ÁN

MÔN : CƠ SỞ TRÍ TUỆ NHÂN TẠO

TÌM KIẾM ĐƯỜNG ĐI TRÊN ĐỒ THỊ

Họ tên : Vương Thị Ngọc Linh

MSSV : 18120195

GV hướng dẫn : thầy Lê Hoài Bắc

thầy Dương Nguyễn Thái Bảo

Tháng 10/2020

MỤC LỤC

PHẦN 1 – GIỚI THIỆU ĐỒ ÁN

Yêu cầu :

- Cài đặt 4 thuật toán tìm kiếm đã học để tìm đường đi trên đồ thị (BFS, DFS, UCS và A*).
- Ngôn ngữ cài đặt : *python 3*

Mức độ hoàn thành :

Tên thuật toán	Mức độ hoàn thành
Breadth First Search	100%
Depth First Search	100%
Uniform Cost Search	100%
A*	100%

Tự đánh giá đồ án : 8.5 / 10 điểm.

Chưa hoàn thành :

- Code chưa tối ưu nhất (Không sử dụng *Priority Queue* của python.)
- Chưa xử lý được trường hợp không đi được đến đích mà chỉ báo lỗi và thoát chương trình.

PHẦN 2 – BREADTH FIRST SEARCH

I. Lý thuyết cơ bản

Ý tưởng:

- Thuật toán tìm các đỉnh bằng cách duyệt theo chiều rộng trên đồ thị vô hướng không có trọng số.
- Xuất phát từ 1 đỉnh (start) và đi tới các đỉnh kề nó cho đến khi không còn đỉnh kề của đỉnh đó, tiếp tục duyệt đỉnh kề trái nhất của đỉnh đó.
- Trong quá trình đi đến đỉnh kề, tiến hành lưu lại đỉnh cha của đỉnh kề (dùng để làm đường đi).
- Duyệt cho đến đỉnh đích (goal), ta sẽ tìm được đường đi ngắn nhất từ start đến goal.

Độ phức tạp: O(|V| + |E|) (V là số đỉnh, E là số cạnh)

II. Chi tiết thuật toán

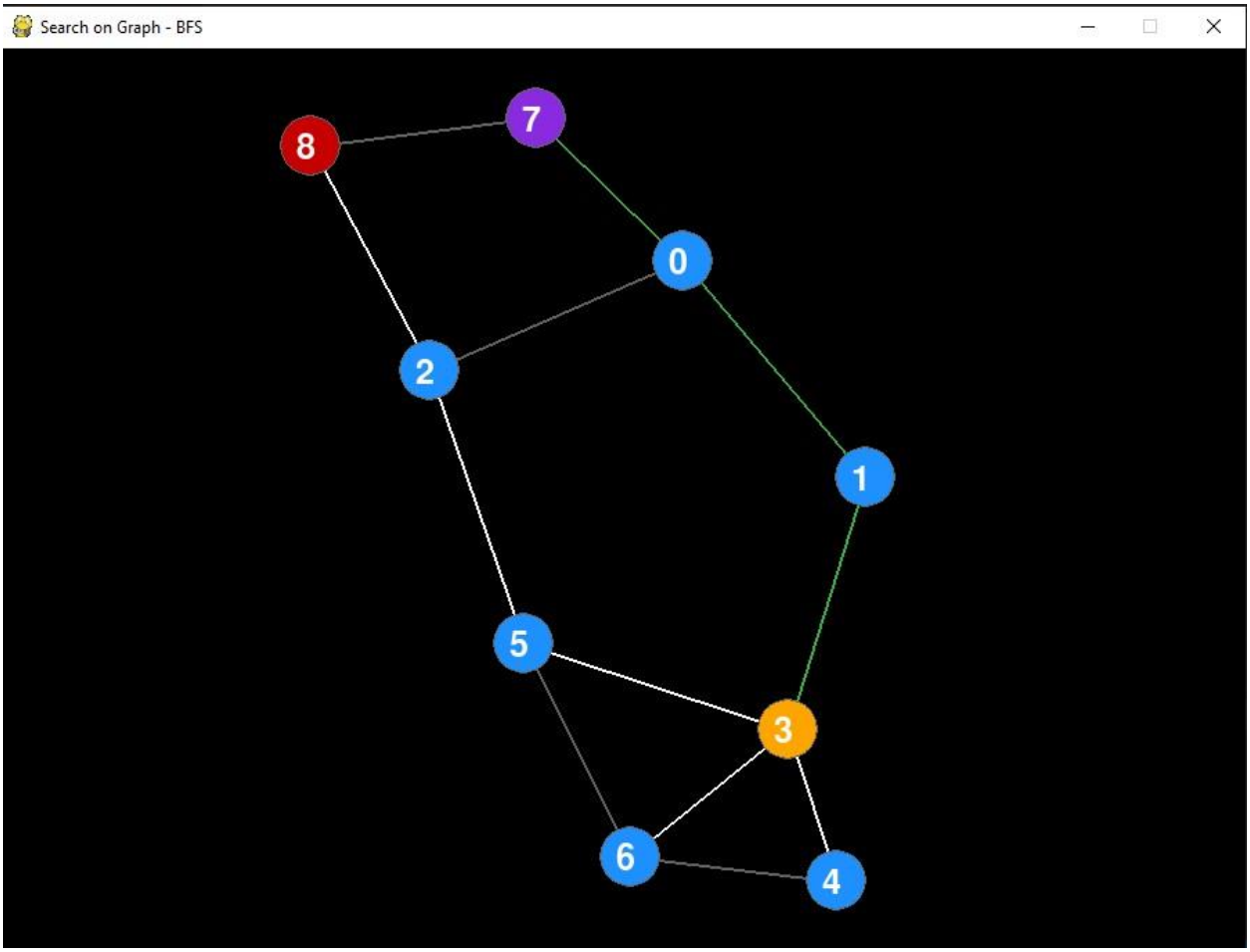
```
def BFS(graph, edges, edge_id, start, goal):
    """
    BFS search
    """
    openQueue = [] #tập đỉnh kề đang mở
    path = [] #đường đi
    visitedQueue = [] #tập đỉnh đã thăm

    for i in graph:
        path.append(-1)

    node = start
    graph[node][3]=orange #đỉnh start
    graphUI.updateUI()
    visitedQueue.append(node)

    while node != goal or len(openQueue) != 0:
        graph[node][3] = yellow #đỉnh đang duyệt
        graphUI.updateUI()
        for i in graph[node][1]:
            if i not in visitedQueue: #nếu đỉnh kề chưa có trong visited thì thêm đỉnh đó vào open và visited
                openQueue.append(i)
                visitedQueue.append(i)
                path[i] = node #lưu lại đỉnh trước đỉnh i
                edges[edge_id(node,i)][1] = white #tô màu cạnh kề
                graph[i][3] = red #tô màu đỉnh trong Queue
                graphUI.updateUI()

        graph[node][3] = blue #đỉnh đã duyệt qua
        graphUI.updateUI()
```

```
    node = openQueue.pop(0) #lấy khỏi open đỉnh đầu tiên để duyệt tiếp

    #nếu đỉnh đó là goal thì tô màu tím và dừng vòng lặp
    if node == goal:
        graph[node][3] = purple
        graphUI.updateUI()
        break

check = goal
graph[start][3] = orange
graphUI.updateUI()

#tô màu đường đi từ goal tới start
while check != start:
    edges[edge_id(path[check],check)][1] = green
    graphUI.updateUI()
    check = path[check]
    if check == start:
        break

print("Implement BFS algorithm.")
pass
```

Chương trình cài đặt thuật toán BFS

1. *Test case 1*

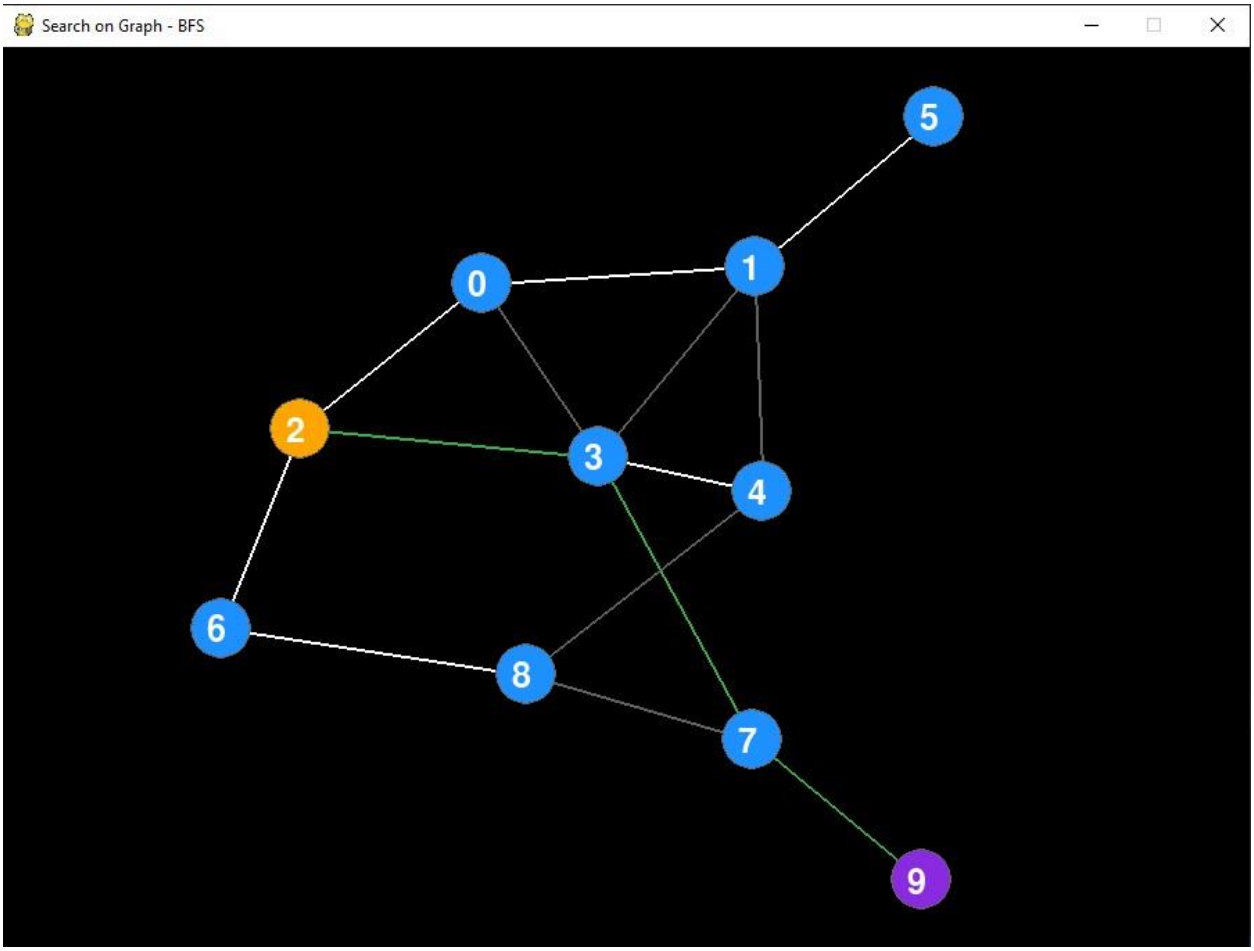
Input: testcase1.txt



Đường đi ngắn nhất (tô màu xanh) từ Start (đỉnh 3) tới Goal (đỉnh 7)

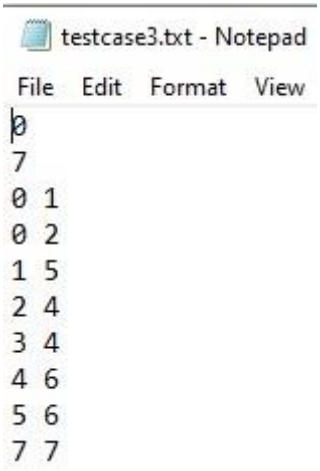
2. Test case 2

Input: testcase2.txt

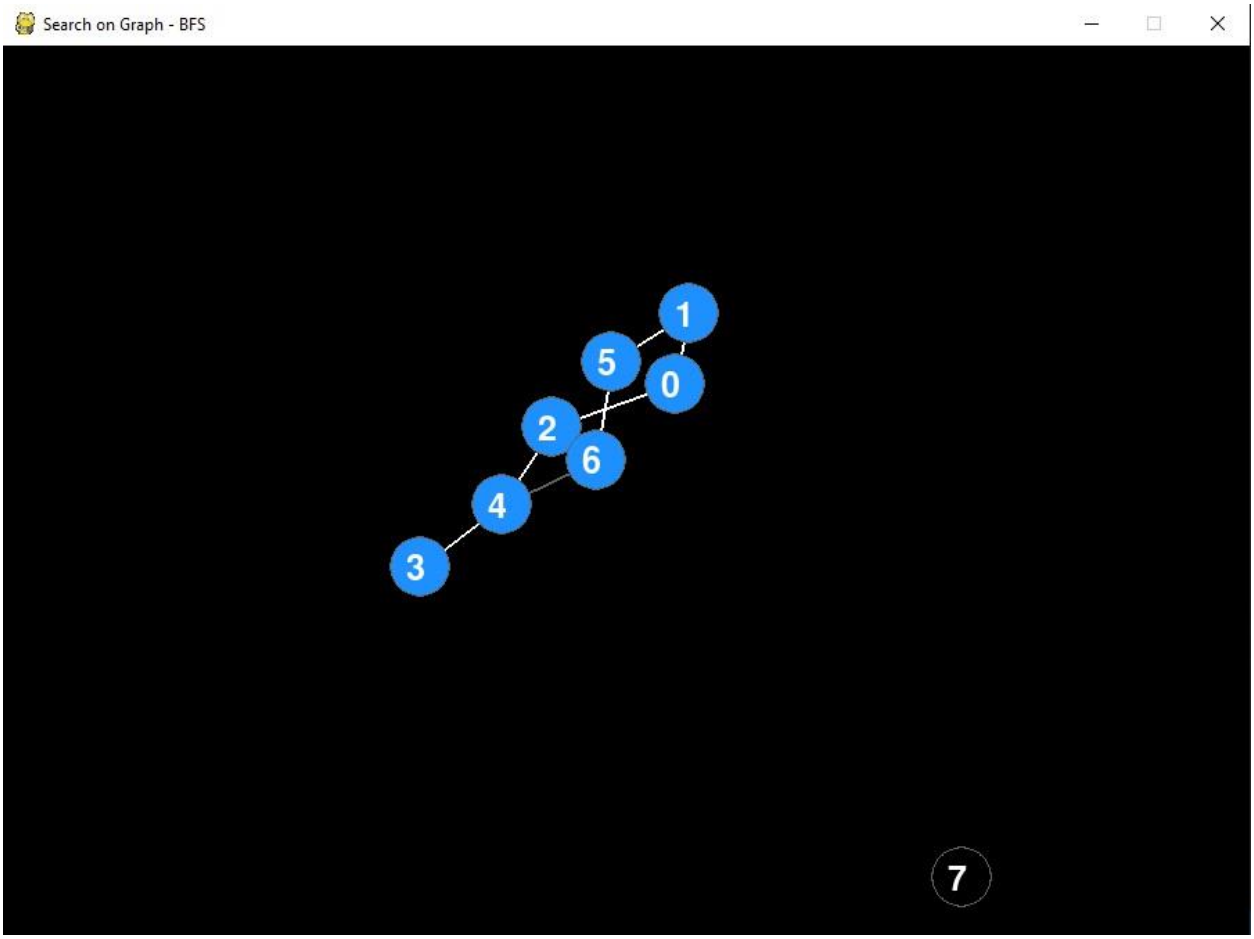


Đường đi ngắn nhất (tô màu xanh) từ Start (đỉnh 2) tới Goal (đỉnh 9)

3. Test case 3



Input: testcase3.txt



Trường hợp đặc biệt : đỉnh 7 (goal) cô lập => không thể đi tới đỉnh.

Nhận xét: Thuật toán BFS tìm ra đường đi ngắn nhất, nhưng thời gian thực hiện lâu vì phải thăm tất cả đỉnh con cho tới khi tìm thấy goal. Đồng thời tốn khá nhiều bộ nhớ để lưu lại trạng thái hàng đợi.

PHẦN 3 - DEPTH FIRST SEARCH

I. Lý thuyết cơ bản

Ý tưởng:

- Thuật toán tìm các đỉnh bằng cách duyệt theo chiều sâu trên đồ thị vô hướng không có trọng số.
- Xuất phát từ 1 đỉnh (start) và đi tới các đỉnh kề bên trái, kiểm tra nếu vẫn còn đỉnh kề thì thay đỉnh đang duyệt = đỉnh kề đi tiếp tới đỉnh kề sau, cho tới khi đỉnh đang duyệt không còn đỉnh kề thì quay lại đỉnh kề trước đó và thực hiện tương tự (duyệt đến đỉnh kề xa nhất).
- Trong quá trình đi đến đỉnh kề, tiến hành lưu lại đỉnh cha của đỉnh kề (dùng để làm đường đi).
- Dừng chương trình khi tìm được đỉnh goal hoặc tất cả các đỉnh đã được thăm (len (stack) == 0. Nếu tìm được đỉnh goal, ta sẽ có đường đi từ đỉnh start đến đỉnh goal.

Độ phức tạp: O(|V| + |E|) (V là số đỉnh. E là số cạnh)

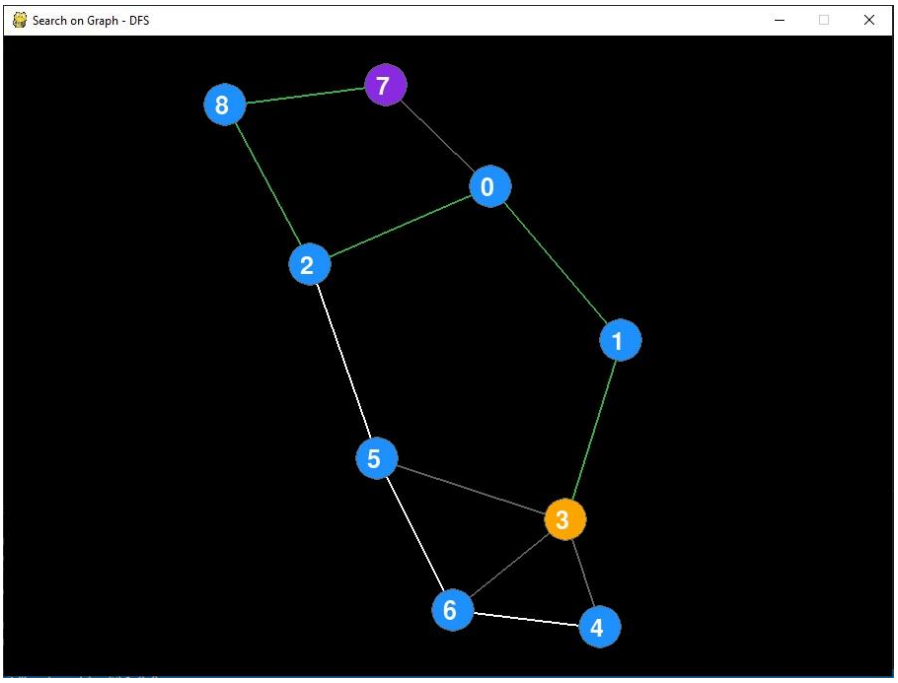
II. Chi tiết thuật toán

```
def DFS(graph, edges, edge_id, start, goal):
    """
    DFS search
    """
    nodeStack = [] #tập đỉnh kề đang mở
    path = [] #đường đi
    visited = [] #tập đỉnh đã thăm

    for i in graph:
        path.append(-1)

    node = start
    visited.append(node)
    nodeStack.append(node)
    graph[node][3]=orange
    graphUI.updateUI()

    while node != goal:
        find = False #đặt biến find để kiểm tra đỉnh kề,nếu đỉnh i có đỉnh kề chưa thăm thì find = True
        for i in graph[node][1]:
            if i not in visited:
                find = True
                nodeStack.append(i)
                visited.append(i)
                path[i] = node #lưu đường đi
                edges[edge_id(node,i)][1] = white
                graph[i][3] = red
                graphUI.updateUI()
                node = i #thay đỉnh đang duyệt thành đỉnh i, duyệt lại tập đỉnh kề của i
                graph[node][3] = yellow
                graphUI.updateUI()
                break
```

```
    if find == False: #nếu i không còn đỉnh kề chưa thăm thì pop đỉnh mới nhất trong stack ra để duyệt
        if len(nodeStack) == 0:
            break
        node = nodeStack.pop()

    graph[node][3] = blue
    graphUI.updateUI()
    if node == goal:
        graph[node][3] = purple
        graphUI.updateUI()

check = goal
graph[start][3] = orange
graphUI.updateUI()

#tô màu đường đi từ goal tới start bằng list path
while check != start:
    edges[edge_id(path[check],check)][1] = green
    graphUI.updateUI()
    check = path[check]
    if check == start:
        break

print("Implement DFS algorithm.")
pass
```

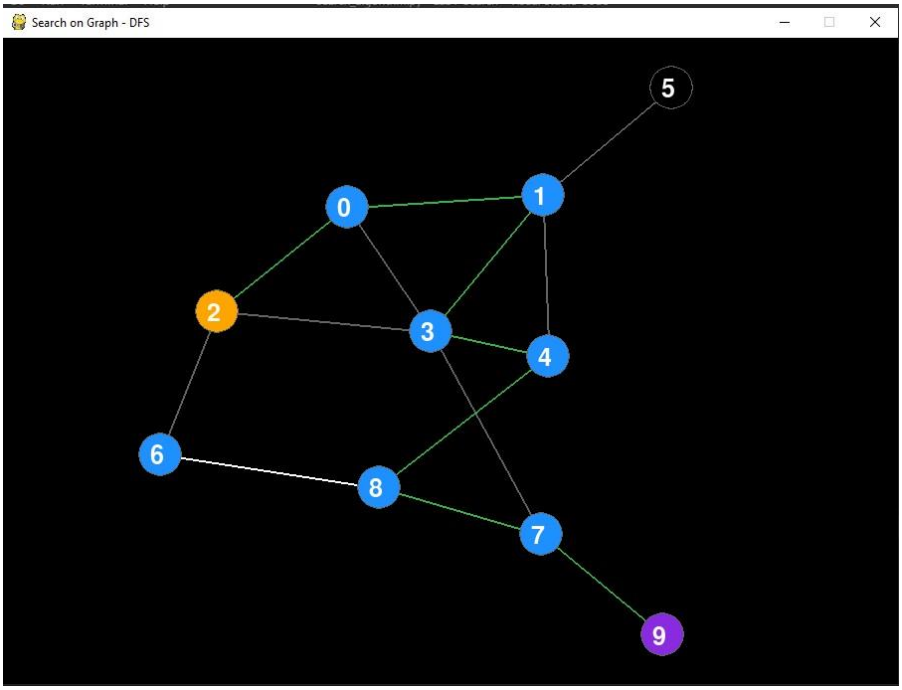
Chương trình cài đặt thuật toán DFS

1. Test case 1



Đường đi (tô màu xanh) từ Start (đỉnh 3) tới Goal (đỉnh 7) bằng thuật toán DFS

2. Test case 2



Đường đi (tô màu xanh) từ Start (đỉnh 2) tới Goal (đỉnh 9) bằng thuật toán DFS

3. *Test case 3*

Trường hợp đặc biệt : đỉnh 7 (goal) cô lập => không thể đi tới đỉnh.

Nhận xét:

- Ưu điểm:

  + Nếu số đỉnh là hữu hạn và không có đỉnh cô lập thì chắc chắn tìm được đỉnh goal.

- Khuyết điểm:

  + Không thể áp dụng cho đồ thị có quá nhiều đỉnh vì sẽ duyệt gần như tất cả các đỉnh, mất thời gian.

  + Mang tính chất mù quáng, duyệt tất cả đỉnh không quan tâm đến các giá trị để giúp duyệt nhanh (chi phí đường đi,…)

PHẦN 4 – UNIFORM COST SEARCH

I. Lý thuyết cơ bản

Ý tưởng:

- Sừ dụng hàng đợi ưu tiên (PQ) để lấy ra đỉnh cho chi phí đường đi thấp nhát trong tập đỉnh đang mở. Với chi phí là g(n).
- Duyệt cho tới khi tìm được goal hoặc queue rỗng, nếu tìm được goal, ta sẽ có đường đi từ đỉnh start tới đỉnh goal với chi phí thấp nhất.
- Ở đây, chọn chi phí = tổng khoảng cách toạ độ từ đỉnh start đến đỉnh con đang mở. Sau mỗi lần duyệt đến đỉnh i, cập nhật chi phí cho đỉnh i nếu đỉnh i chưa được thăm

Độ phức tạp: $O(b^{1+C^*/\varepsilon})$ (C* là chi phí tối ưu nhất, b là hệ số phân nhánh).

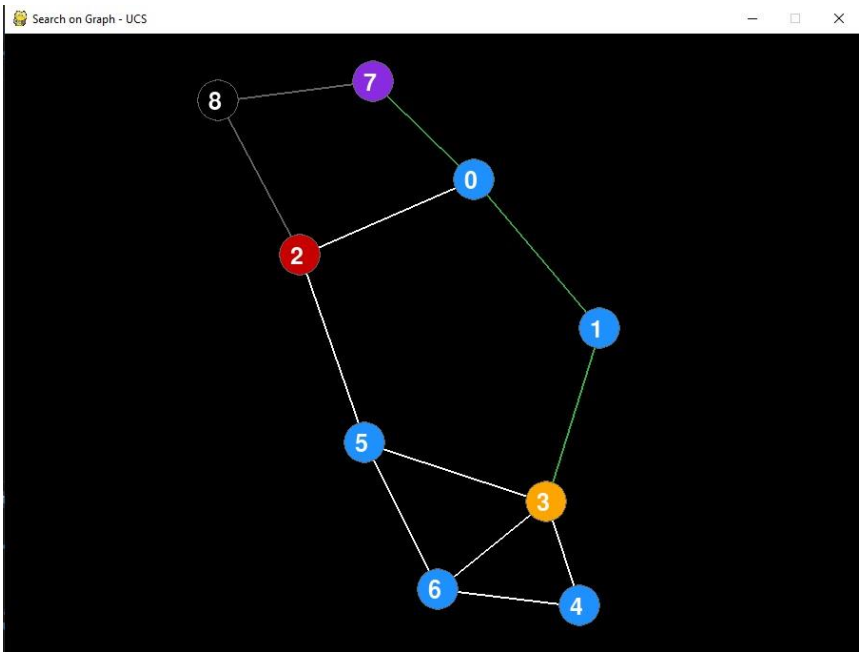
II. Chi tiết thuật toán

```
def UCS(graph, edges, edge_id, start, goal):
    """
    Uniform Cost Search search
    """
    nodeQueue = [] #tập đỉnh kề đang mở
    path = [] #đường đi
    visited = [] #tập đỉnh đã thăm
    cost = [] #chi phí

    for i in graph:
        cost.append([i,0])
        path.append(-1)

    node = start
    visited.append(node)
    graph[node][3]=orange
    graphUI.updateUI()

    while node != goal:

        for i in graph[node][1]:
            if i not in visited:
                #cập nht65 chi phí từ đỉnh Start đến đỉnh con i
                cost[i][1]=cost[node][1] + math.sqrt(pow((graph[i][0][0] - graph[node][0][0]),2) + pow((graph[i][0][1] - graph[node][0][1]),2))
                if i not in nodeQueue:
                    nodeQueue.append(i)
                    path[i] = node
                edges[edge_id(node,i)][1] = white
                graph[i][3] = red
                graphUI.updateUI()
```

```
        x = [] #xác định đỉnh có chi phí thấp nhất trong queue
        for i in nodeQueue:
            x.append(cost[i][1])
        minCost = min(x)

        for i in nodeQueue:
            if cost[i][1] == minCost: #lấy ra đỉnh có chi phí thấp nhất để duyệt tiếp
                node = i
                nodeQueue.remove(i)
                visited.append(i)
                graph[node][3] = yellow
                graphUI.updateUI()
                break

        graph[node][3] = blue
        graphUI.updateUI()
        if node == goal:
            graph[node][3] = purple
            graphUI.updateUI()
            break

    check = goal
    graph[start][3] = orange
    graphUI.updateUI()

    #tô màu đường đi từ goal tới start bằng list path
    while check != start:
        edges[edge_id(path[check],check)][1] = green
        graphUI.updateUI()
        check = path[check]
        if check == start:
            break
```

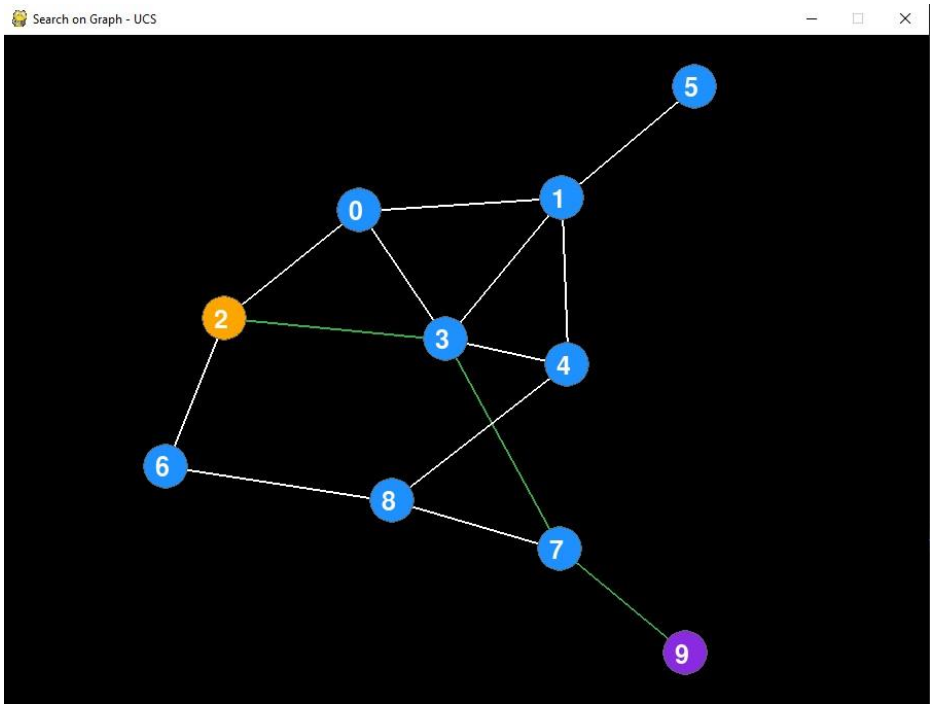
Chương trình cài đặt thuật toán UCS

1. Test case 1



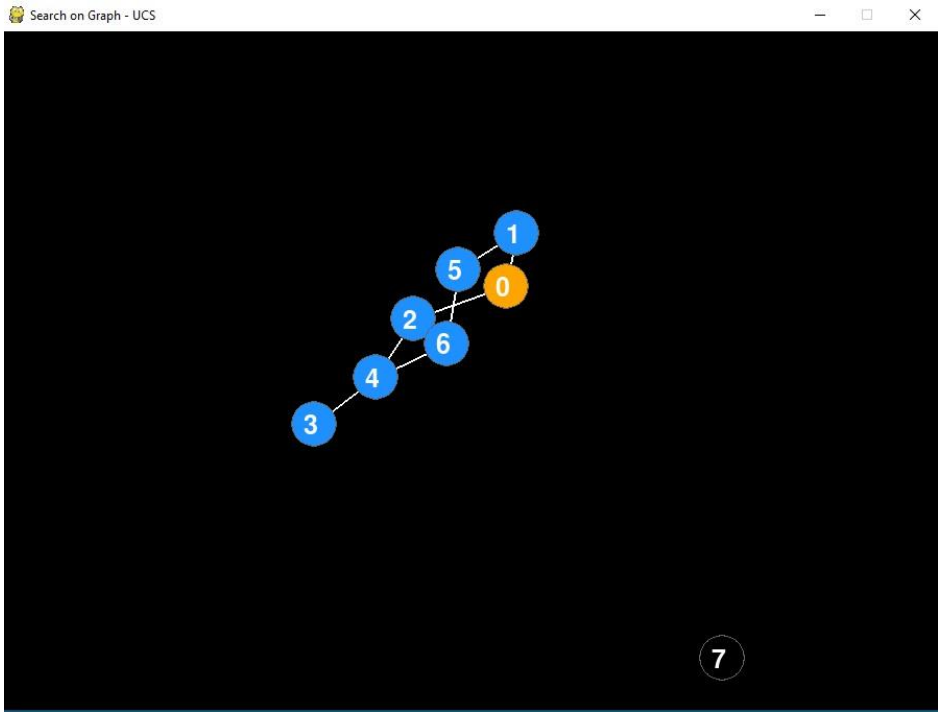
Đường đi (tô màu xanh) từ Start (đỉnh 3) tới Goal (đỉnh 7) với chi phí thấp nhất

2. Test case 2



Đường đi (tô màu xanh) từ Start (đỉnh 2) tới Goal (đỉnh 9) với chi phí thấp nhất

3. Test case 3



Trường hợp đặc biệt : đỉnh 7 (goal) cô lập => không thể đi tới đỉnh.

Nhận xét: UCS tìm được đường đi ngắn nhất thông qua chi phí đường đi, tuy nhiên nó phải thăm gần như toàn bộ đỉnh trước khi đi tới goal vì chỉ quan tâm chi phí đường đi mà không có bất kỳ thông tin nào về goal => chưa tối ưu.

PHẦN 5 – A*

I. Lý thuyết cơ bản

Ý tưởng:

- Thuật toán A* tìm ra đường đi ngắn nhát trong đồ thị bằng hàm *Heuristic.*
- Biểu thức tìm đường đi *: f(n) = g(n) + h(n). Với g(n) là chi phí đường đi được tính như UCS, h(n) là hàm Heuristic để đánh giá trạng thái hiện tại so với đích, từ đó tìm ra lựa chọn tối ưu nhất.*
- Ở bài này, chọn h(n) là khoảng cách toạ độ từ đỉnh n đến goal.
- *Lý do chọn:* Đồ thị mô phỏng trên mặt phẳng 2 chiều, tượng trưng cho đường đi trên thực tế nhìn từ trên. Trên thực tế, chúng ta sẽ chọn đi con đường tạo ra cảm giác khoảng cách của chúng ta tới đích là gần nhất.
- Sau mỗi lần đi đến đỉnh kề I, ta cập nhật lại f(i). Sau đó lấy ra khỏi queue đỉnh đang mở đỉnh n có giá trị f(n) thấp nhất và duyệt dỉnh n.
- Dừng chương trình khi tìm được goal hoặc queue rỗng.

Độ phức tạp: Phụ thuộc vào giá trị hàm Heuristic đã chọn. Công thức khái quát để tính độ phức tạp là **|h(x) – h*(x)| <= O(log h*(x))**

Trong đó h*(x) là hàm Heuristic tối ưu nhất từ đỉnh x tới đích.

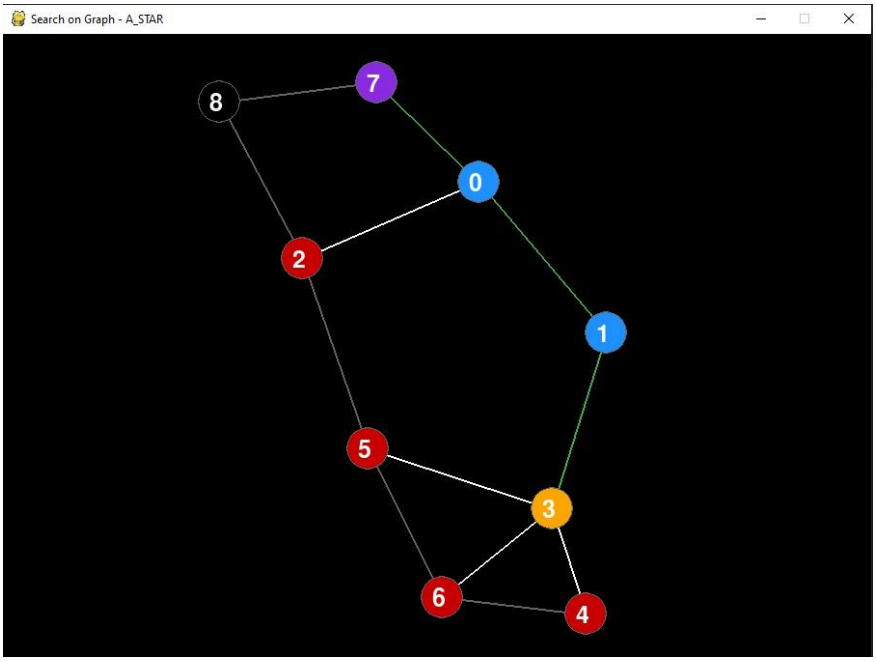
II. Chi tiết thuật toán

```
def AStar(graph, edges, edge_id, start, goal):
    """
    A star search
    """
    nodeQueue = []
    path = []
    visited = []
    cost = []
    f = []

    for i in graph:
        cost.append([i,0])
        f.append([i,0])
        path.append(-1)

    node = start
    visited.append(node)
    graph[node][3]=orange
    graphUI.updateUI()

    while node != goal:
        for i in graph[node][1]:
            if i not in visited:
                #cập nhật chi phí đường đi và f(n) của đỉnh i
                cost[i][1]=cost[node][1] + math.sqrt(pow((graph[i][0][0] - graph[node][0][0]),2) + pow((graph[i][0][1] - graph[node][0][1]),2))
                f[i][1] = cost[i][1] + math.sqrt(pow((graph[goal][0][0] - graph[i][0][0]),2) + pow((graph[goal][0][1] - graph[i][0][1]),2))
                if i not in nodeQueue:
                    nodeQueue.append(i)
                    path[i] = node
                edges[edge_id(node,i)][1] = white
                graph[i][3] = red
                graphUI.updateUI()
```

```
        x = []
        for i in nodeQueue:
            x.append(f[i][1])
        minf = min(x) #lấy giá trị f(n) thấp nhất trong hàng đợi

        for i in nodeQueue:
            if f[i][1] == minf:
                node = i #lấy ra khỏi hàng đợi đỉnh có f(n) thấp nhất để duyệt
                nodeQueue.remove(i)
                visited.append(i)
                graph[node][3] = yellow
                graphUI.updateUI()
                break

        graph[node][3] = blue
        graphUI.updateUI()
        if node == goal:
            graph[node][3] = purple
            graphUI.updateUI()
            break

    check = goal
    graph[start][3] = orange
    graphUI.updateUI()

    #tô màu đường đi từ goal tới start bằng list path
    while check != start:
        edges[edge_id(path[check],check)][1] = green
        graphUI.updateUI()
        check = path[check]
        if check == start:
            break
```

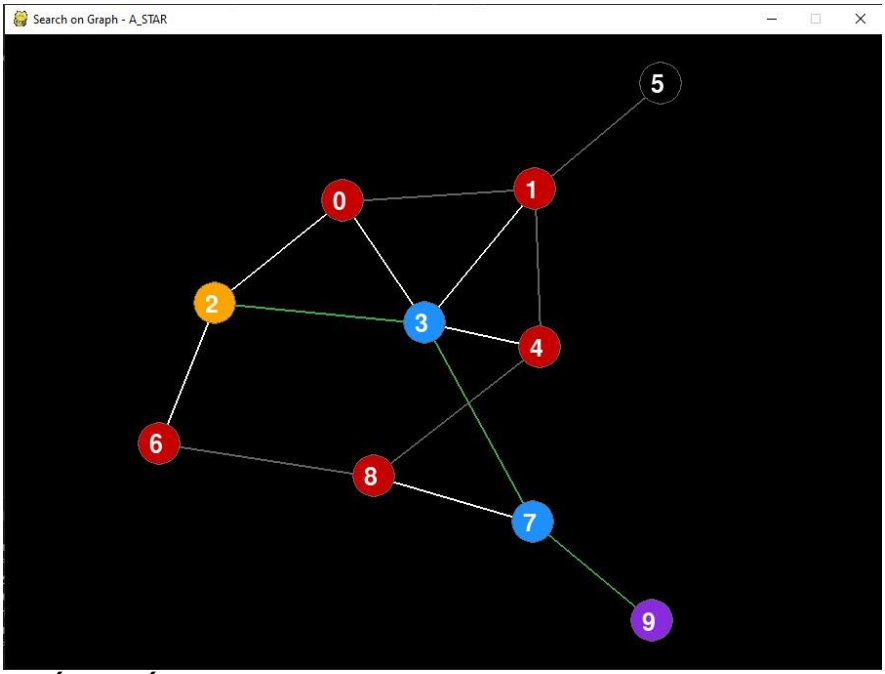
*Chương trình cài đặt thuật toán A**

1. *Test case 1*



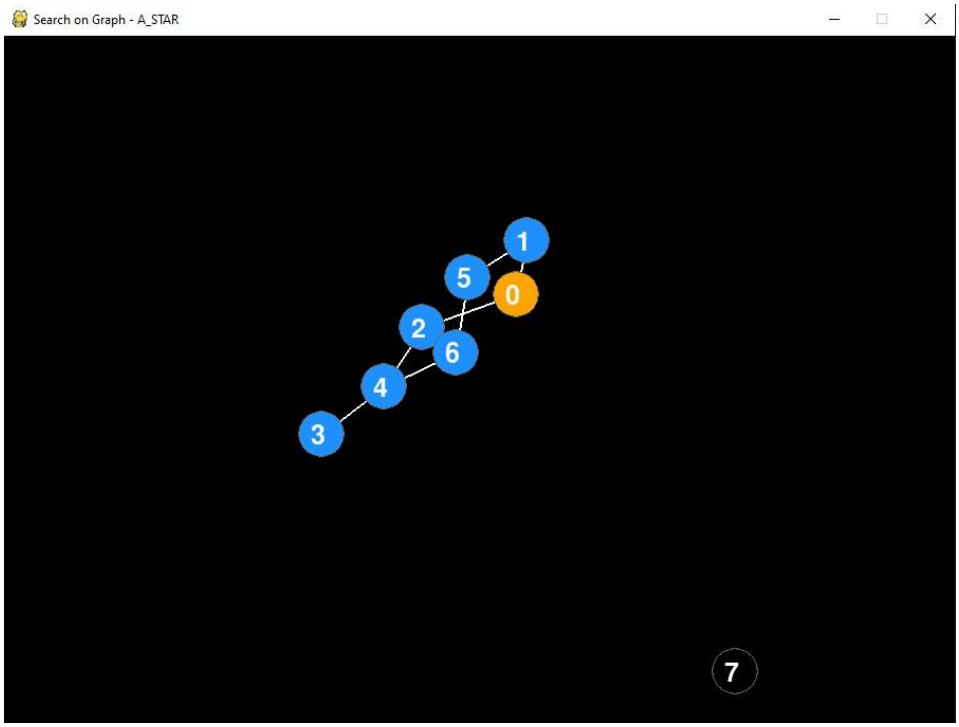
*Đường đi ngắn nhất (tô màu xanh) từ Start (đỉnh 3) tới Goal (đỉnh 7) bằng thuật toán A**

2. *Test case 2*



*Đường đi ngắn nhất (tô màu xanh) từ Start (đỉnh 2) tới Goal (đỉnh 9) bằng thuật toán A**

3. *Test case 3*



Trường hợp đặc biệt : đỉnh 7 (goal) cô lập => không thể đi tới đỉnh.

Nhận xét: Thuật toán giúp tìm kiếm đường đi ngắn nhất một cách nhanh chóng và linh hoạt với hàm Heuristic. Tuy nhiên, cũng giống như BFS, thuật toán A* chiếm khá nhiều tài nguyên bộ nhớ để lưu lại trạng thái hàng đợi.

III. So sánh thuật toán A* và thuật toán UCS

A* là thuật toán tìm kiếm có thông tin, nó dựa vào hàm Heuristic để đánh giá trạng thái hiện tại so với mục tiêu, từ đó lựa chọn con đường tối ưu nhất để đi tới đích ở hiện tại. Thuật toán A* giúp giảm số lượng đỉnh phải thăm.

UCS không được cung cấp thông tin về mục tiêu mà chỉ đơn thuần là chọn ra đường đi có chi phí thấp nhất từ điểm khởi đầu. Do đó, thông thường UCS sẽ thăm gần như toàn bộ đỉnh để đến được mục tiêu vì nó chỉ quan tâm đến chi phí đường đi thấp nhất.

TÀI LIỆU THAM KHẢO

Bài giảng môn Cơ Sở Trí Tuệ Nhân Tạo – thầy Lê Hoài Bắc

Giáo trình Cơ Sở Trí Tuệ Nhân Tạo

Python Tutorial – https://www.w3schools.com/

https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/

https://www.geeksforgeeks.org/a-search-algorithm/