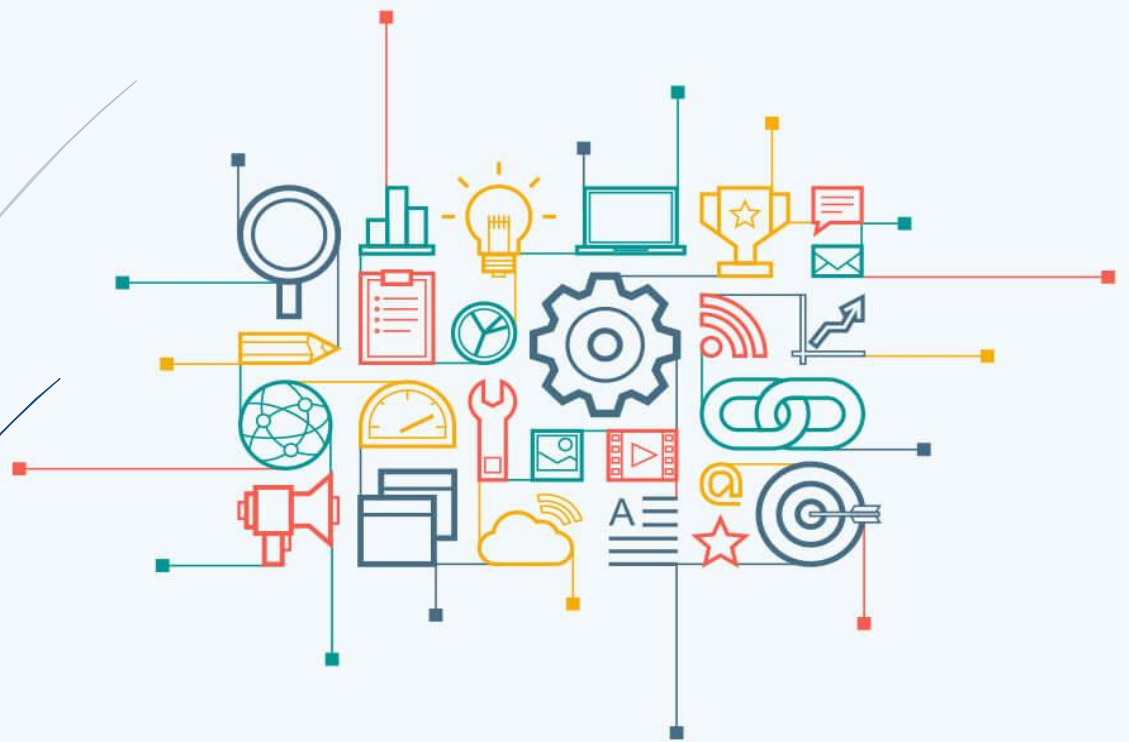


Data Structures and Algorithms



ĐH KHOA HỌC TỰ NHIÊN TP HCM

Đồ án cuối kì

Graph

Tài liệu này ghi nhận lại nội dung các thuật toán tìm kiếm trong đồ thị bao gồm:

1. DFS
2. BFS
3. UCS

MỤC LỤC

I.Thông tin chung	1
II.Nội dung	2
A. DFS.....	2
a. Ý tưởng chung:.....	2
b. Mã giả.....	3
c. Đánh giá thuật toán:.....	4
d. Ví dụ thuật toán:	5
B. BFS	10
a. Ý tưởng chung:.....	10
b. Mã giả.....	12
c. Đánh giá thuật toán:.....	13
d. Ví dụ thuật toán:	14
C. UCS.....	18
1. Ý tưởng chung.....	18
2. Mã giả:	19
3. Đánh giá thuật toán	20
4. Ví dụ thuật toán	20
D. So sánh giữa các thuật toán:.....	24
III. Kết quả cài đặt.....	27
IV.Bảng phân công công việc	31

I. Thông tin chung

Các thành viên tham dự:

STT	MSSV	Họ và tên	Email
1	21280115	Trần Đức Trung	21280115@student.hcmus.edu.vn
2	21280085	Lê Hồ Hoàng Anh	21280085@student.hcmus.edu.vn

Mục tiêu:

1. Tìm hiểu và trình bày các thuật toán tìm kiếm đường đi trên đồ thị
2. So sánh các thuật toán với nhau
3. Cài đặt được thuật toán tìm kiếm đường đi

Thời gian bắt đầu:29/11/2022..... Thời gian kết thúc:11/12/2022.....

II. Nội dung

A. DFS

a. Ý tưởng chung:

- Tìm kiếm ưu tiên chiều sâu hay tìm kiếm theo chiều sâu (tiếng Anh: *Depth-first search* - DFS) là một thuật toán duyệt hoặc tìm kiếm trên một cây hoặc một đồ thị. 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.
- Thông thường, DFS là một dạng tìm kiếm thông tin không đầy đủ mà quá trình tìm kiếm được phát triển tới đỉnh con đầu tiên của nút đang tìm kiếm cho tới khi gặp được đỉnh cần tìm hoặc tới một nút không có con. Khi đó giải thuật quay lui về đỉnh vừa mới tìm kiếm ở bước trước. Trong dạng không đệ quy, tất cả các đỉnh chờ được phát triển được bổ sung vào một ngăn xếp LIFO

- Ý tưởng DFS:

- Chọn một đỉnh tùy ý của đồ thị làm gốc
- Xây dựng đường đi từ đỉnh này bằng cách lần lượt ghép các cạnh sao cho mỗi cách mới ghép sẽ nối đỉnh cuối cùng trên đường đi với một đỉnh còn chưa thuộc đường đi. Tiếp tục ghép thêm cạnh vào đường đi chừng nào không thể thêm được nữa
- Nếu đường đi qua tất cả các đỉnh của đồ thị thì cây do đường này tạo nên là cây khung
- Nếu chưa thì lùi lại đỉnh trước đỉnh cuối cùng của đường đi và xây dựng đường đi mới xuất phát từ đỉnh này đi qua các đỉnh còn chưa thuộc đường đi.

Nếu điều đó không thể làm được thì lùi thêm một đỉnh nữa trên đường đi và thử xây dựng đường đi mới. Tiếp tục quá trình như vậy cho đến khi tất cả các đỉnh của đồ thị được ghép vào cây. Cây T có được là cây khung của đồ thị.

b. Mã giả

```

def DFS(graph g):
    open_set = [g.start]
    closed_set = []
    father = [-1]* g.get_len()
    while(open_set not empty):
        if(open_set[-1] in closed_set):
            open_set.pop(-1)
            break
        current=open_set.pop(-1)
        if ( current is g.goal ):
            findFather(g, current, father)
            break
        list_neighbors=findNeighbor(current)
        for neighbor in list_neighbors:
            if neighbor not in closed_set:
                open_set.append(neighbor)
                father[neighbor]=current
        end for
        closed_set.append(current)
    end while
end DFS

```

```

//hàm để in ra đường đi dựa vào list father
Function findFather(graph g, node current, node father):
    If current = g.start:
        Print(g.start)
    Print(current)
    Return findFather(g, father[neighbor], father)
End findFather

```

c. Đánh giá thuật toán:

- **Tính tối ưu:** DFS không có tính tối ưu, vì cần di chuyển nhiều lần hoặc nhiều chi phí để tìm đến đích.

- **Tính đầy đủ:** DFS có tính đầy đủ trong không gian cây tìm kiếm hữu hạn, vì nó sẽ di chuyển đến mọi vị trí trong cây để tìm kiếm đích cần tìm.

- **Độ phức tạp :** $O(V+E)$ khi đồ thị được biểu diễn ở dạng danh sách kề
 $O(V^2)$ khi đồ thị được biểu diễn ở dạng ma trận kề

Với V là số đỉnh, E là số cạnh trong đồ thị

- Thuận lợi:

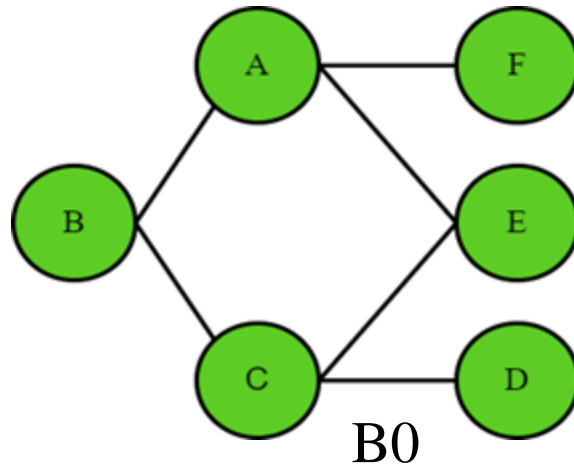
- DFS yêu cầu ít bộ nhớ vì nó chỉ cần lưu trữ một stack các node trên đường đi từ node gốc đến node hiện tại
- DFS mất ít thời gian hơn để đến node đích so với BFS (nếu nó đi đúng đường)

- Khó khăn:

- DFS algorithm tìm kiếm theo chiều sâu nên một số trường hợp có thể đi đến vô hạn

d. Ví dụ thuật toán:

Graph được xét:



Chú thích:

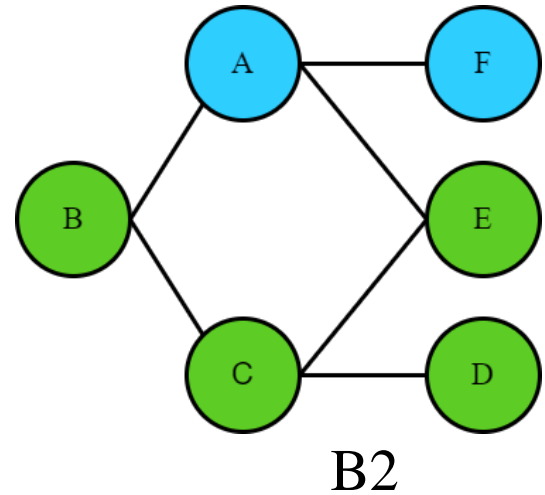
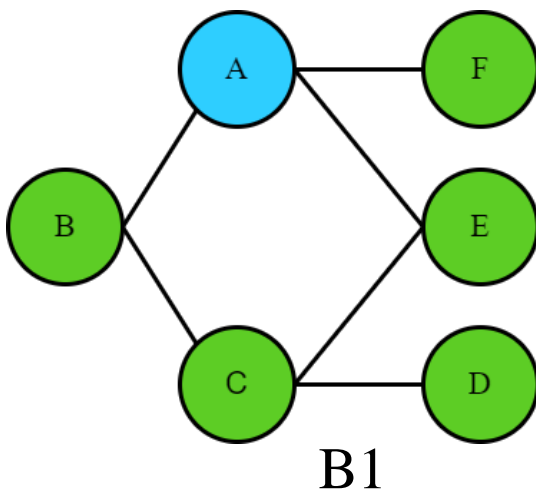


:Node chưa được duyệt trong graph



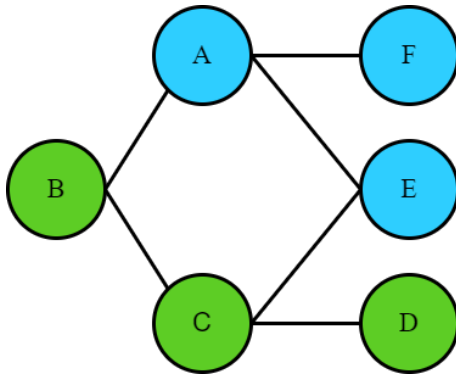
:Node đã được duyệt trong graph

- Đường đi theo các node màu xanh dương từ node bắt đầu đến node kết thúc là đường đi mà thuật toán DFS duyệt.

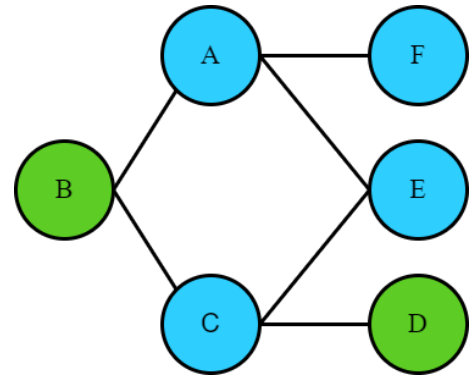


- Xét node ban đầu là A. Tại A các node liên kết được đưa vào **opened set** là {B, E, F} với **opened set** được lưu dưới dạng stack. Sau đó đưa A thêm vào **closed set**

Bước	Current	Opened set (Stack)	Closed set
B0		A	
B1	A	[B, E, F]	A
B2	F	[B, E]	F



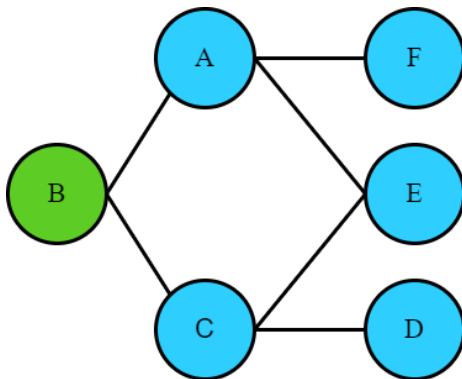
B3



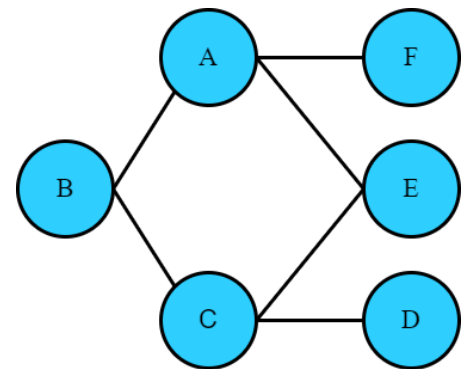
B4

- Node F được đẩy khỏi stack và tại F không node nào được liên kết. Nên ta đẩy node kế tiếp trong stack ra do đó node được xét tiếp theo trong **opened set** là E.
- Tại node E, **opened set** được cập nhật là {B, C} do E có 2 node liên kết là A và C nhưng A đã nằm trong **closed set** do đã được duyệt. Và node được chọn kế tiếp là C

	Current	Opened set (Stack)	Closed set
B0		A	
B1	A	[B, E, F]	A
B2	F	[B, E]	F
B3	E	[B, C]	E
B4	C	[B, D]	C



B5



B6

- Tại node C các node trong **opened set** được cập nhật thành {B, B, D} do liên kết với C còn 2 node chưa nằm trong **closed set**. Và theo stack, node được duyệt tiếp theo là D. **Closed set** thêm C.
- Tại node D không còn node liên kết nào để duyệt, **opened set** được cập nhật là {B, B}.
- Và node được chọn kế tiếp trong stack là B. Đẩy B ra khỏi stack, tại B không còn node nào để duyệt, thêm B vào closed set. Nhưng trong **opened set** vẫn còn giá trị {B} nên thuật toán tiếp tục thực hiện xét node B và khi này do trong **closed set** đã có node B nên **opened set** chỉ việc đẩy node này ra và kết thúc khi **opened set** đã rỗng

	Current	Opened set (Stack)	Closed set
B0		A	
B1	A	[B, E, F]	A
B2	F	[B, E]	F
B3	E	[B, C]	E
B4	C	[B, B, D]	C
B5	D	[B, B]	D
B6	B	[B]	B
B7		[]	

B. BFS

a. Ý tưởng chung:

- Trong lý thuyết đồ thị, tìm kiếm theo chiều rộng (BFS) là một thuật toán tìm kiếm trên đồ thị trong đó việc tìm kiếm chỉ bao gồm 2 thao tác: (a) cho trước một đỉnh của đồ thị; (b) thêm các đỉnh kề với đỉnh vừa cho vào danh sách có thể hướng tới tiếp theo.
- Có thể sử dụng thuật toán tìm kiếm theo chiều rộng cho hai mục đích: 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. Trong đồ thị không có trọng số, thuật toán tìm kiếm theo chiều rộng luôn tìm ra đường đi ngắn nhất có thể.
- Thuật toán BFS bắt đầu từ đỉnh gốc và lần lượt nhìn các đỉnh kề với đỉnh gốc. Sau đó, với mỗi đỉnh trong số đó, thuật toán lại lần lượt nhìn trước các đỉnh kề với nó mà chưa được quan sát trước đó và lặp lại. Xem thêm thuật toán tìm kiếm theo chiều sâu, trong đó cũng sử dụng 2 thao tác trên nhưng có trình tự quan sát các đỉnh khác với thuật toán tìm kiếm theo chiều rộng.
- Ta có thể sử dụng cấu trúc dữ liệu hàng đợi (queue) để cài đặt thuật toán này.

- Ý tưởng BFS:

Cho G là đồ thị liên thông với tập đỉnh $\{v_1, v_2, \dots, v_n\}$

- Thêm v_1 như là gốc của cây rỗng
- Thêm vào các đỉnh kề v_1 và các cạnh nối v_1 với chúng. Những đỉnh này là đỉnh mức 1 trong cây
- Đối với mọi đỉnh v mức 1, thêm vào các cạnh kề với v vào cây sao cho không tạo nên chu trình. Ta thu được các đỉnh mức 2

Tiếp tục quá trình này cho tới khi tất cả các đỉnh của đồ thị được ghép vào cây. Cây T có được là cây khung của đồ thị.

b. Mã giả

```
function BFS(graph g):
    opened_set=[g.start] # queue
    closed_set=[]        # list of visited node
    father=[-1]*g.get_len() #list of node's father

    while(opened_set not empty):
        current=opened_set.pop(0)      # take first element out
        if (current is g.goal ):
            findFather(g,current,father)
            break
        list_neighbors=find_neighbor(current)
        for neighbor in list_neighbors:
            if neighbor not in closed_set && neighbor not in opened_set:
                open_set.append(neighbor)
                father[neighbor] = current
        end for
        closed_set.append(current)
    end while
end BFS
```

```
//hàm để in ra đường đi dựa vào list father
Function findFather(graph g, node current, node father):
    If current = g.start:
        Print(g.start)
    Print(current)
    Return findFather(g, father[neighbor], father)
End findFather
```

c. Đánh giá thuật toán:

- **Tính tối ưu:** Đối với đồ thị không trọng số, BFS sẽ tối ưu hơn so với DFS. BFS tối ưu nếu chi phí đường đi là một hàm không tăng của độ sâu $d(\text{depth})$. Thông thường, BFS được áp dụng khi tất cả mọi hành động có cùng chi phí đường đi.
- **Tính đầy đủ:** BFS đầy đủ, điều đó có nghĩa là nếu node đích ở độ sâu nhất định thì BFS sẽ tìm ra giải pháp đường đi
- **Độ phức tạp:** $O(V+E)$ khi đồ thị được biểu diễn ở dạng danh sách kề

$O(V^2)$ khi đồ thị được biểu diễn ở dạng ma trận kề

Với V là số đỉnh, E là số cạnh trong đồ thị

- Thuận lợi:

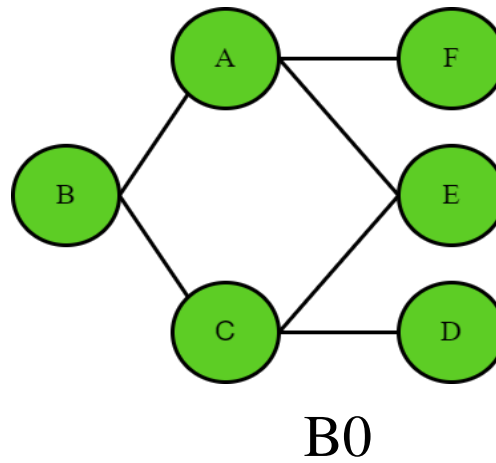
- BFS sẽ cung cấp một giải pháp nếu có bất kỳ giải pháp tồn tại
- Nếu có nhiều hơn một phương pháp cho graph thì BFS sẽ đưa ra giải pháp tiêu hao ít chi phí nhất

- Khó khăn:


- BFS cần nhiều bộ nhớ mỗi khi lưu tầng tầng của tree để mở rộng thêm tầng tiếp theo.
- BFS cần nhiều thời gian nếu node đích ở xa so với node gốc cần di chuyển.


d. Ví dụ thuật toán:

Graph được xét:

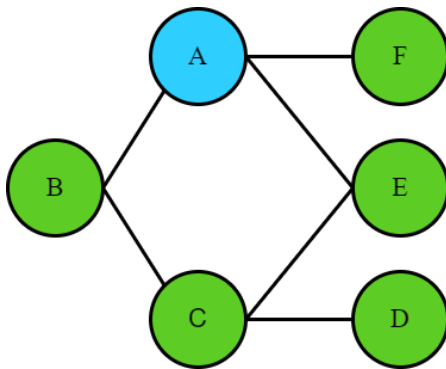


Chú thích:

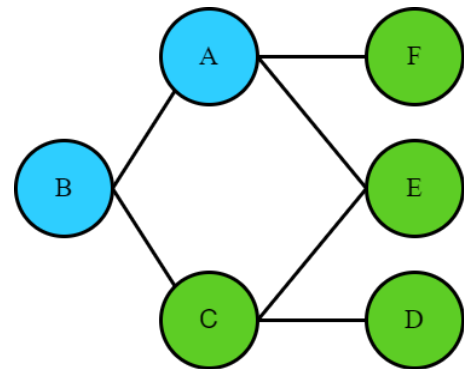
 :Node chưa được duyệt trong graph

 :Node đã được duyệt trong graph

- Đường đi theo các node màu xanh dương từ node bắt đầu đến node kết thúc là đường đi mà thuật toán BFS duyệt.



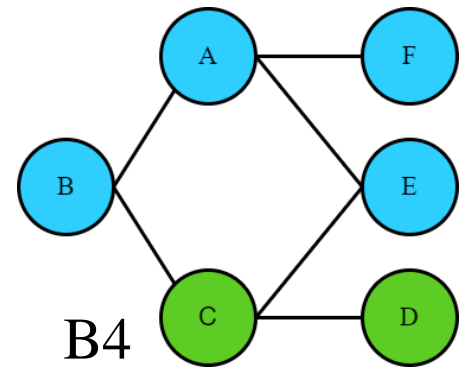
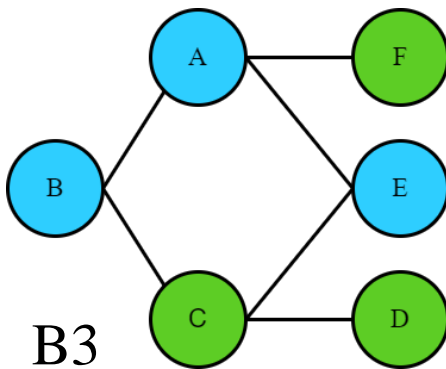
B1



B2

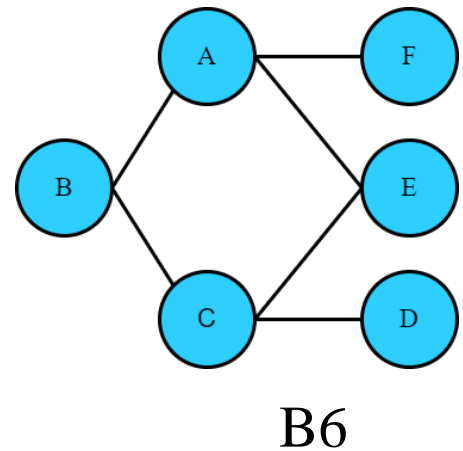
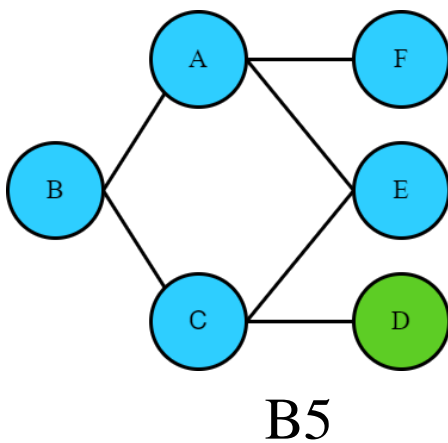
- Xét node ban đầu là A. Tại A các node liên kết được thêm vào **opened set** là {B, E, F} với **opened set** được sử dụng như 1 queue. Xong A được thêm vào **closed set**
- Và xét node tiếp theo được lấy ra từ **opened set** là B.
- Đẩy B ra khỏi **opened set**, tại node B, node liên kết C được thêm vào **opened set** là {E, F, C}, thêm B vào **closed set**. Và theo **opened set**, node được duyệt tiếp theo là E.

	Current	Opened set	Closed set
B0		[A]	
B1	A	[B,E,F]	A
B2	B	[E,F,C]	B



- Đẩy node E khỏi **opened set**, **opened set** được cập nhật thành {F, C} do node liền kề với E là C và A nhưng A đã được duyệt, còn C thì đã được thêm vào **opened set** khi duyệt node B nên không cần thiết thêm nữa, cho E vào **closed set**. Và node được chọn kế là F.

	Current	Opened set	Closed set
B0		[A]	
B1	A	[B,E,F]	A
B2	B	[E,F,C]	B
B3	E	[F,C]	E
B4	F	[C]	F



- Đẩy F khỏi **opened set**, các node trong **opened set** lúc này {C} vì tại F không còn duyệt thêm được node nào. Thêm F vào **closed set**. Và theo **opened set** tiếp theo là C.
- Đẩy node C ra khỏi **opened set** và cập nhật thêm node liên kề chưa duyệt và cũng chưa có trong **opened set** {D}, thêm C vào closed set . Và node cuối trong **opened set** là D, đẩy D ra khỏi **opened set**, **opened set** rỗng kết thúc thuật toán.

	Current	Opened set	Closed set
B0		[A]	
B1	A	[B,E,F]	A
B2	B	[E,F,C]	B
B3	E	[F,C]	E
B4	F	[C]	F
B5	C	[D]	C
B6	D	[]	D

C. UCS

1. Ý tưởng chung

- Tìm kiếm chi phí đều (hay còn gọi là *tìm kiếm chi phí cực tiểu* hoặc *tìm kiếm theo giá thành thống nhất*, viết tắt tiếng Anh là UCS) là một cách duyệt cây dùng cho việc duyệt hay tìm kiếm một cây, cấu trúc cây, hoặc đồ thị có trọng lượng (chi phí). Việc tìm kiếm bắt đầu tại nút gốc.^[1] Việc tìm kiếm tiếp tục bằng cách duyệt các nút tiếp theo với trọng lượng hay chi phí thấp nhất tính từ nút gốc. Các nút được duyệt tiếp tục cho đến khi đến được nút đích cần đến.

Nội dung:

- **Opened set** sẽ dựa vào hàng đợi ưu tiên (**priority queue**). Mỗi node mới sẽ được thêm vào dựa trên ưu tiên cho đường đi tiêu tốn ít chi phí nhất
- Node ở vị trí trên cùng của **opened set** sẽ được đẩy ra khỏi danh sách, điều đó có nghĩa là node này sẽ được xem xét ở bước kế tiếp. Nó sẽ không lặp lại. Nếu đã đi qua node, sẽ bỏ qua node đó bằng cách đưa node đó vào **closed set**.

Thuật toán:

1. Thêm node bắt đầu start vào **opened set** với chi phí đường đi là $\text{cost}[\text{start}] = 0$
2. Đẩy node có chi phí thấp nhất lúc này ra khỏi **opened set** và gọi node này là current. Kiểm tra current có là goal hay không nếu là goal thì kết thúc, không thì tiếp tục
3. Thêm node mới là các node liên kết với current vào **opened set** (các node này đều chưa được duyệt tức không nằm trong **closed set**). Các node này được thêm vào với thứ tự ưu tiên cho các node có chi phí thấp. Nếu node được thêm đã tồn tại trong **opened set** ta sẽ giữ node có chi phí thấp hơn và bỏ node còn lại.
4. Đồng thời cập nhật lại giá trị trong father và cost
5. Thêm current tức node vừa xét xong vào **closed set** (các node đã duyệt)

6. Kiểm tra opened set có rỗng không. Rỗng thì kết thúc, không thì lặp lại bước 2.

2. Mã giả:

```
Function UCS(g):
    opened_set = {} // được khởi tạo kiểu dictionary ("key": "value")
    opened_set[g.start] = 0
    closed_set = []
    father = [-1]*g.get_len()
    cost = [100000]*g.get_len()
    cost[g.start] = 0

    while(opened_set not empty):
        current = min cost node in opened_set
        opened_set.pop(current)
        if (current is g.goal):
            findFather(g,current,father) // Hàm đã được dùng ở phần BFS
            break
        list_neighbors=find_neighbor(current)
        for neighbor in list_neighbors:
            if neighbor not in closed_set:
                distance = findDistance(neighbor, current)
                if (cost[current]+distance < cost[neighbor]):
                    //cập nhật giá trị của node neighbor này trong cost
                    cost[neighbor] = cost[current]+distance
                    //thêm/cập nhật cặp "key": "value" trong opened_set
                    opened_set[neighbor] = cost[current]+distance
                    father[neighbor]=current
            end for
        closed_set.append(current)
    end while
end UCS
```

3. Đánh giá thuật toán

- **Tính đầy đủ:** Vì với mọi đồ thị trừ đồ thị có trọng số âm, UCS sẽ luôn tìm được đường đi ngắn nhất từ đỉnh này tới đỉnh khác

- **Tính tối ưu:** UCS luôn có tính tối ưu, vì UCS chỉ chọn đường đi với chi phí thấp nhất

- **Độ phức tạp:** $O(m^{(1+\text{floor}(L/e))})$

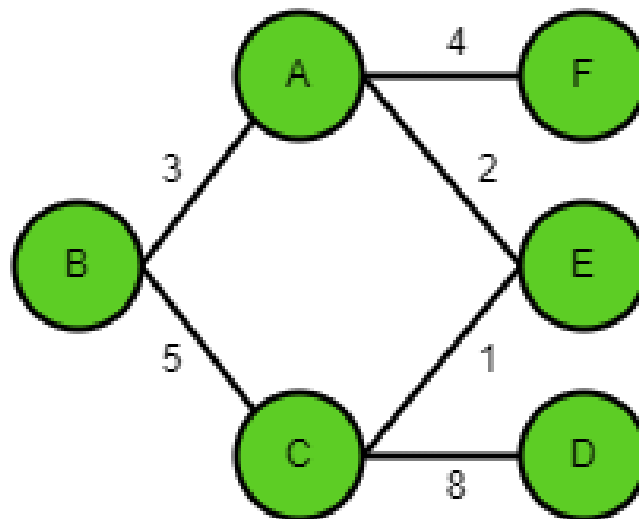
Với : m là số liên kết tới đã 1 node có

L là chiều dài đường đi ngắn nhất tới đích

e là chi phí thấp nhất của 1 cạnh

4. Ví dụ thuật toán

- Xét graph đường đi từ A đến D:

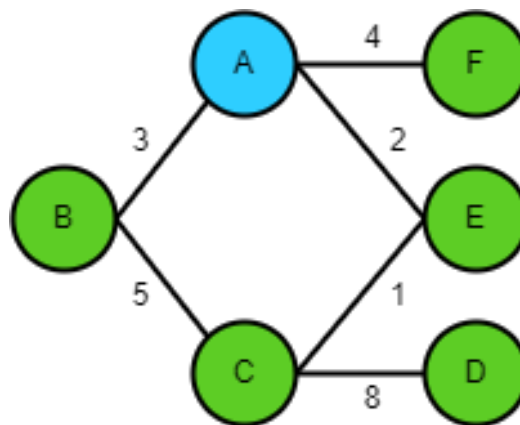


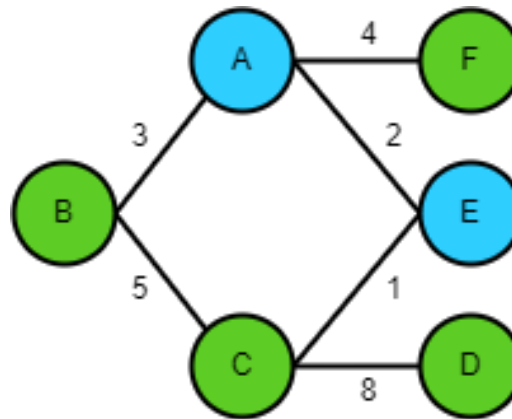
Current	Opened set (priority queue)	Closed set
	A(A,0)	
A (A, 0)	E(A, 2), B(A, 3), F(A, 4)	A
E (A, 2)	B(A, 3), C(E, 3), F(A, 4)	E
B (A, 3)	C(E, 3), F(A, 4)	B
C (E, 3)	F(A, 4), D(C, 11)	C
F(A,4)	D(C,11)	F
D(C,11)		D

-Trình bày dưới dạng bảng ta được:

	A	B	C	D	E	F
B0	0*	∞	∞	∞	∞	∞
B1	-	(A, 3)	∞	∞	(A, 2)*	(A, 4)
B2	-	(A, 3)*	(E, 3)	∞	-	(A, 4)
B3	-	-	(E, 3)*	∞	-	(A, 4)
B4	-	-	-	(C, 11)	-	(A, 4)*
B5	-	-	-	(C, 11)*	-	-

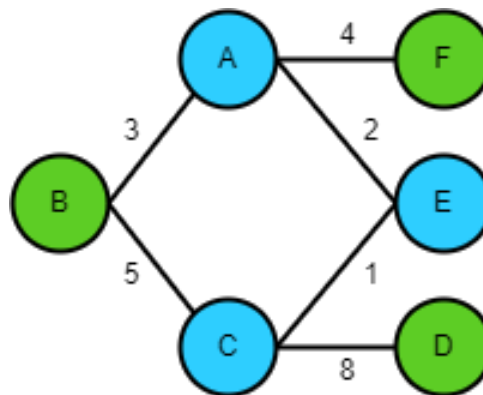
- B1: Chọn A làm node gốc. Với các node xung quanh liên kết với A là B, E, F. Vì **opened set** là priority queue theo chi phí nên thứ tự được thêm vào **opened set** là [E(A,2), B(A,3), F(A,4)].





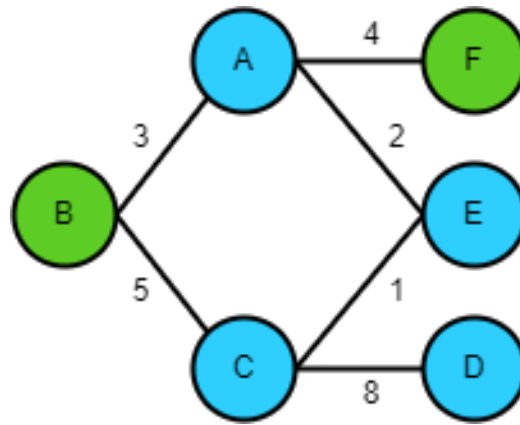
-B2: Node E(A,2) được đẩy ra khỏi **opened set** và chọn làm current, ta được các node kề chưa duyệt với E là C(E,3), khi này **opened set** được cập nhật thành [B(A, 3), C(E, 3), F(A, 4)]. Closed set bổ sung thêm E.

-B3: Current lúc này là B(A,3), ta được các node kề chưa duyệt với B là C, nhưng chi phí đi qua đường này để tới C là 8 nên C(E,3) trong opened set không đổi. **Opened set** được cập nhật thành [C(E, 3), F(A, 4)]. Closed set bổ sung thêm B.



- B4: Current là C(E,3) theo **opened set**, ta có node liền kề chưa duyệt là D tính chi phí của D theo C(E,3) ta được D(C,11). **Opened set** được cập nhật thành [F(A, 4), D(C,11)]. Closed set bổ sung thêm C.

- B5: Current là $F(A,4)$ theo **opened set**, do không còn node nào duyệt được. **Opened set** được cập nhật thành $[D(C,11)]$. Closed set bổ sung thêm F.



- B6: Current là $D(C,11)$ theo **opened set**. D là đích đến.
- B7: Khi tìm được D, thuật toán dừng lại và duyệt ngược để dò father của từng node bắt đầu từ goal để có được đường đi ngắn nhất là A-E-C-D. Kết thúc UCS

D. So sánh giữa các thuật toán:*a. Giữa DFS và BFS*

Tiêu chí	DFS	BFS
CTDL được sử dụng	Stack (Last in First out)	Queue (First in Last out)
Định nghĩa	Tiến hành duyệt các nodes xa nhất có thể cho đến khi không còn node xung quanh chưa được duyệt	Duyệt toàn bộ nodes trên cùng một mức trước khi di chuyển tới mức tiếp theo
Kỹ thuật	Phải duyệt qua nhiều cạnh để duyệt đến đỉnh cần tìm từ một đỉnh nguồn	Có thể sử dụng để tìm kiếm đường đi ngắn nhất trong đồ thị không trọng số. Vì trong BFS, có thể duyệt đến một đỉnh với số cạnh nhỏ nhất từ một đỉnh
Sự khác biệt về ý tưởng	DFS được xây dựng dựa trên tree sub-tree by sub-tree	BFS được xây dựng dựa trên cấp độ của cây
Thời gian	$O(V+E)$: Khi là một danh sách kề $O(V^2)$: Khi là ma trận kề Với V cạnh và E đỉnh	$O(V+E)$: Khi là một danh sách kề $O(V^2)$: Khi là ma trận kề Với V cạnh và E đỉnh
Bộ nhớ	DFS sử dụng ít bộ nhớ hơn	BFS sử dụng nhiều bộ nhớ hơn
Space complexity	DFS có ít hơn vì trong cùng một thời gian DFS chỉ lưu trữ một đường đi đơn từ gốc đến node lá	Nhiều hơn
Tốc độ	DFS nhanh hơn so với BFS	BFS chậm hơn so với DFS
Đồ thị tìm kiếm phù hợp	DFS phù hợp khi node đích ở xa so với node gốc	BFS phù hợp cho việc tìm kiếm đỉnh ở các vị trí gần node gốc bắt đầu
Backtracking	Thuật toán DFS là một hàm đệ quy có thể dựa trên ý tưởng của backtracking	BFS không có ý tưởng nào từ backtracking

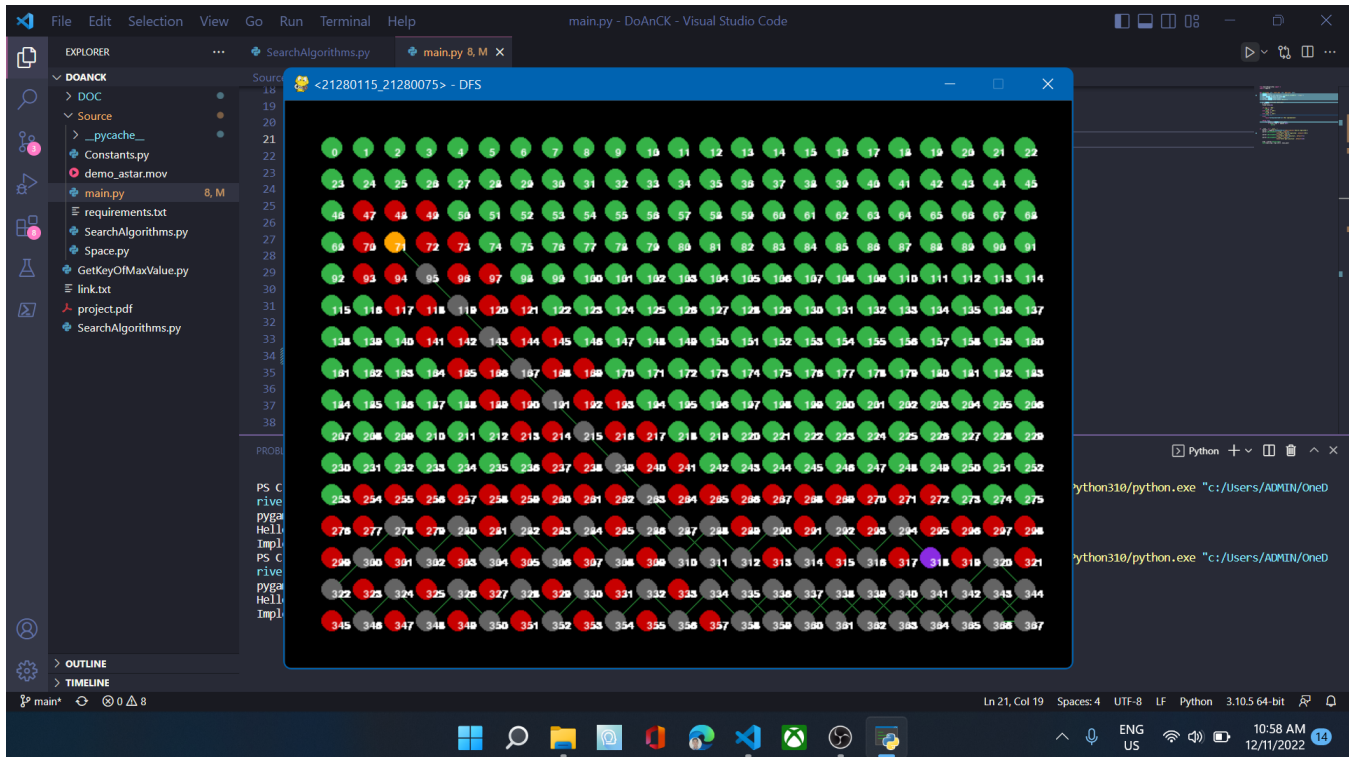
Khả năng tối ưu	DFS không tối ưu cho việc tìm đường đi ngắn nhất	BFS tối ưu với cho việc tìm kiếm đường đi ngắn nhất
Việc duyệt các node con	Ở đây, thuật toán DFS duyệt các node con trước khi duyệt các node gốc	Các node cha sẽ được duyệt trước, sau đó mới tới các node con được duyệt
Cách loại bỏ các node đã đi qua	Các node đã được duyệt sẽ được thêm vào stack và sau đó xóa cho đến khi không còn node nào để duyệt	Các node đã được duyệt qua nhiều lần sẽ bị xóa khỏi hàng đợi
Sử dụng khi nào?	DFS được sử dụng khi node đích ở khoảng cách xa so với node bắt đầu, khi đó DFS sẽ phù hợp hơn	BFS được sử dụng khi node đích ở các vị trí gần hoặc lân cận so với node bắt đầu, khi đó BFS sẽ thể hiện việc tìm kiếm đường đi tốt hơn

b. Giữa Dijkstra và Uniform-cost search

Tiêu chí	Uniform-cost search	Dijkstra
Cấu trúc dữ liệu được sử dụng	Priority queue (hàng đợi ưu tiên)	
Khởi tạo	Tập hợp chỉ có một đỉnh nguồn	Tập hợp tất cả các đỉnh trong G
Memory requirement	Chỉ cần lưu trữ các đỉnh cần thiết	Cần phải lưu trữ một graph khác sau mỗi lần duyệt qua mỗi đỉnh
Khả năng tối ưu	UCS tối ưu hơn cho việc tìm kiếm đường đi ngắn nhất từ 2 đỉnh có sẵn	Dijkstra tối ưu cho việc tìm kiếm đường đi ngắn nhất từ đỉnh bắt đầu tới tất cả các đỉnh có trong graph với chi phí thấp nhất
Thời gian chạy	Nhanh hơn vì cần ít bộ nhớ cần thiết	Chậm hơn vì cần sử dụng nhiều bộ nhớ
Application	Implicit and explicit graphs	Explicit graphs only
Điểm dừng	Khi tìm được đường đi tối ưu cho điểm đích đề ra	Khi tìm được đường đi tối ưu cho tất cả các điểm

III. Kết quả cài đặt

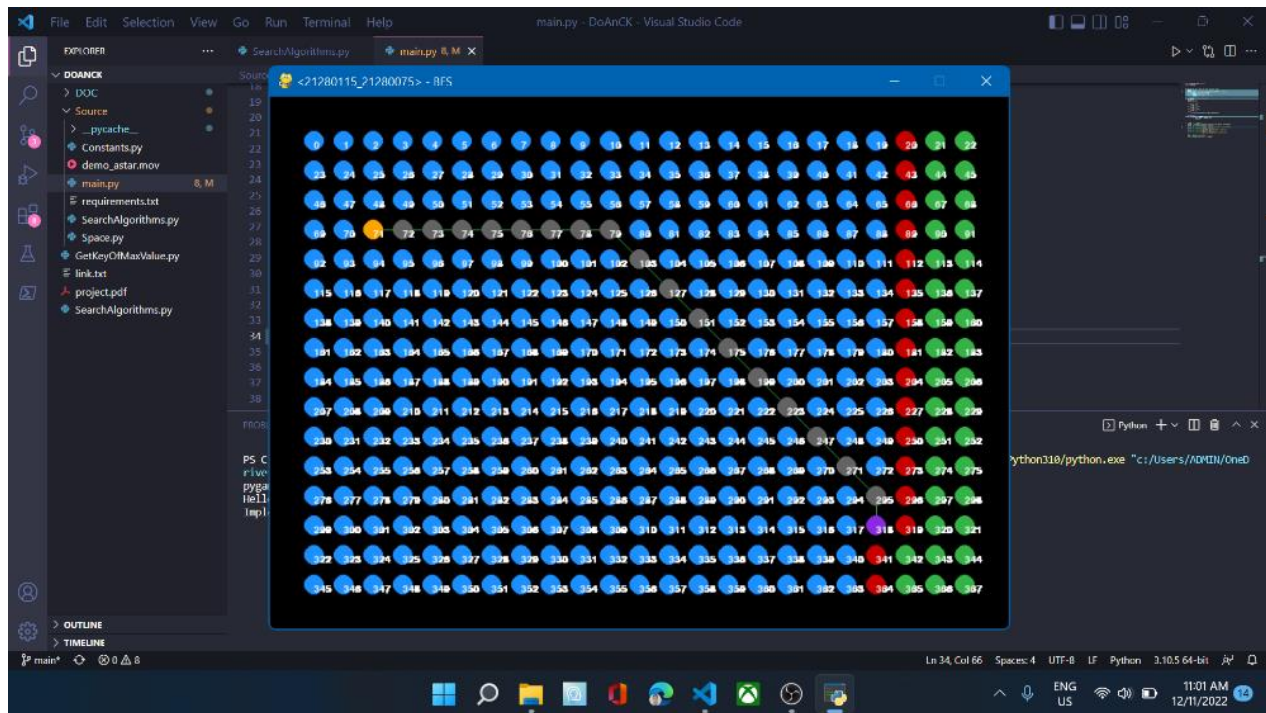
1. DFS



-Mô tả: Từ điểm start (màu vàng) DFS mở rộng ra tìm các node neighbor của mình và đưa vào stack theo thứ tự (trên, dưới, trái, phải, trái trên, phải trên, trái dưới, phải dưới) do đó node được ưu tiên duyệt kết tiếp sẽ ngược lại với thứ tự thêm bên trên. Cứ xét như vậy với từng node trong stack, xét xong node thì đẩy nó ra khỏi stack và xét tiếp, đến khi gặp node goal hay không còn node nào để duyệt

-Nhận xét: đường đi của DFS chưa tối ưu, chưa phải đường đi ngắn nhất, tuy nhiên hao phí bộ nhớ lại không quá nhiều ít hơn hẳn so với 2 thuật toán còn lại và thời gian chạy cũng nhanh hơn tương đối.

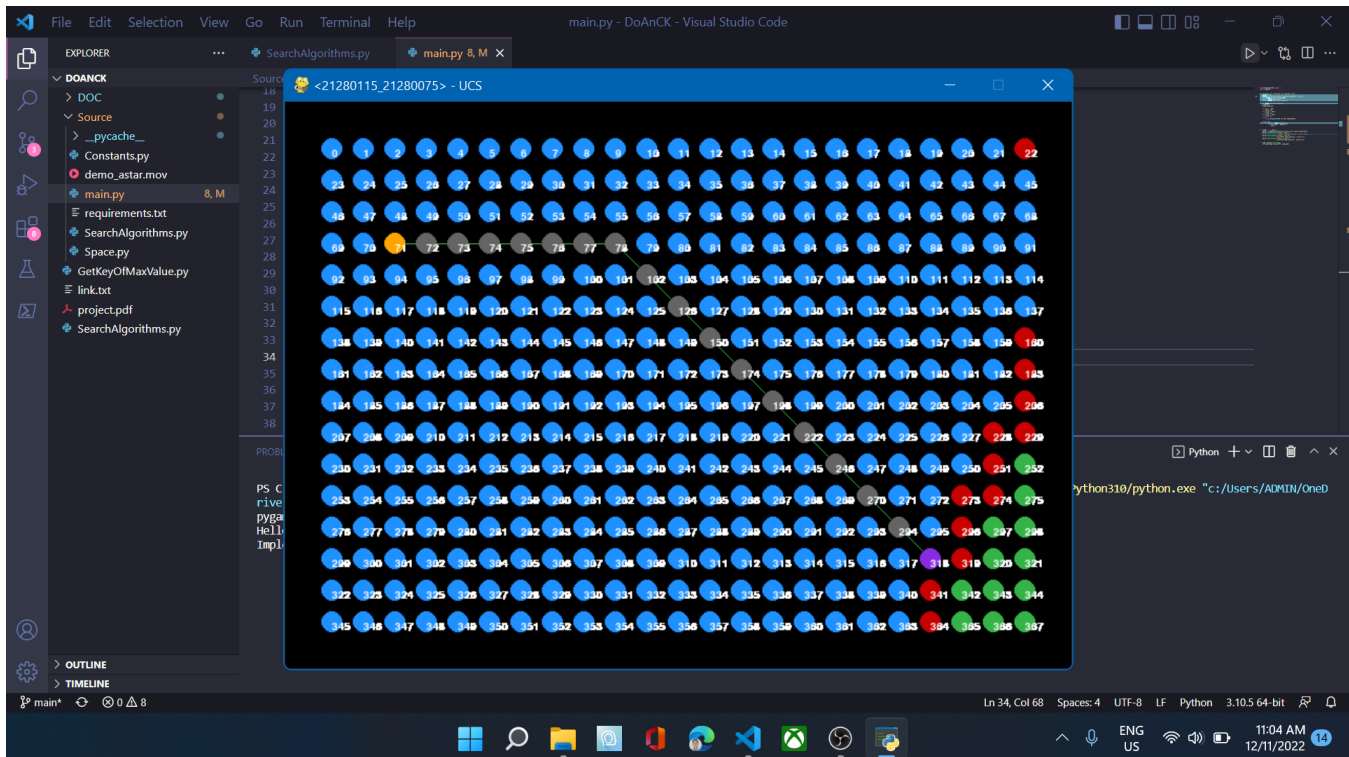
2.BFS



-Mô tả: từ điểm ban đầu start (màu vàng) thuật toán BFS sẽ tìm các neighbor của node đang xét và đưa vào queue theo thứ tự (trên, dưới, trái, phải, trái trên, phải trên, trái dưới, phải dưới) do đó node được ưu tiên duyệt (và thêm các neighbor của node đó) kết tiếp sẽ theo thứ tự bên trên. Nên khi chạy ra sẽ thấy thuật toán sẽ mở rộng tỏa dần ra. Đến khi đạt được goal (màu tím) hoặc không còn node để duyệt

-Nhận xét: Đường đi của BFS tối ưu hơn so với DFS nhưng để tìm được đường đi này nó đã phải duyệt rất nhiều node do đó chi phí bộ nhớ khá lớn (số node đã duyệt gần như là tất cả so với DFS chỉ khoảng 1/3) và thời gian duyệt lâu hơn rất nhiều

3.UCS



-Mô tả: từ điểm ban đầu start (màu vàng) thuật toán UCS sẽ tìm các neighbor của node đang xét và đưa vào priority queue theo thứ tự ưu tiên node nào có chi phí thấp nhất. Khi chạy ta thấy thuật toán cũng sẽ mở rộng tỏa dần ra xung quanh tuy nhiên không hoàn toàn giống BFS mà nó sẽ đi tới nhưng node có chi phí di chuyển thấp nhất tức có quãng đường đi gần nhất. Đến khi đạt được goal (màu tím) hoặc không còn node để duyệt

-Nhận xét: Đường đi của UCS là 1 đường đi tối ưu nhưng để tìm được đường đi này nó đã phải duyệt rất nhiều node do đó chi phí bộ nhớ cũng khá lớn (số node đã duyệt gần như là tất cả so với DFS chỉ khoảng 1/3) và thời gian duyệt lâu hơn nhiều. Ngoài ra UCS không chỉ ghi nhận lại đường đi tối ưu cho mình node GOAL mà còn cả những node còn lại

Tài liệu tham khảo:

<https://www.javatpoint.com/ai-uninformed-search-algorithms>

<https://www.baeldung.com/cs/uniform-cost-search-vs-dijkstras>

<https://www.educative.io/answers/what-is-a-uniform-cost-search-algorithm>

[GeeksforGeeks | A computer science portal for geeks](#)

2014-Data abstraction and Problem Solving 6th-Book

7. [Graph - Google Drive](#)

IV. Bảng phân công công việc

STT	Người phụ trách	Mô tả nội dung công việc	Bắt đầu	Kết thúc
1	Trần Đức Trung	-Tìm hiểu, cài đặt thuật toán DFS, BFS, UCS -Viết mã giả 3 thuật toán -Minh họa BFS, DFS, UCS	29/11/2022	10/12/2022
2	Lê Hồ Hoàng Anh	-Tìm ý tưởng chung của của DFS, BFS, UCS -Đánh giá về 3 thuật toán -So sánh DFS – BFS và UCS - Dijkstra	1/12/2022	10/12/2022