ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA TOÁN - TIN HỌC

———— * ————


BÁO CÁO THỰC HÀNH ĐỒ ÁN

# NGHIÊN CỨU THỰC HIỆN CÁC THUẬT TOÁN TÌM KIẾM ĐƯỜNG ĐI

MÔN HỌC: TH Cấu trúc dữ liệu & giải thuật
GVHD: Thầy Nguyễn Bảo Long
LỚP: 21KDL1B

—o0o—

SVTH 1: Huỳnh Lưu Vĩnh Phong - 21280103
SVTH 2: Trần Thị Uyên Nhi - 21280125

TP. HỒ CHÍ MINH, 12/2022

# Lời mở đầu

Bài toán tìm đường đi tối ưu đề cập đến việc tìm kiếm con dường với chi phí tối thiểu giữa hai điểm. Đây là một vấn đề cơ bản trong lý thuyết đồ thị. Trong bài toán tìm đường đi tối ưu thông thường, các thông số (khoảng cách, thời gian...) giữa các nút khác nhau được giả định định rằng biết chính xác. Nhưng trong những tình huống thực tế đời sống, luôn luôn tồn tại sự không chắc chắn về thông số giữa các nút khác nhau.

Chính vì vậy đồ án này tạo ra nhằm thực hiện các mục tiêu sau:
- Nghiên cứu một số thuật toán tìm đường tối ưu
- Nghiên cứu một số thuật toán tìm dường ngắn nhất
- Cài đặt thử nghiệm thuật toán tìm đường đi tối ưu
- Đánh giá và so sánh các thuật toán

Nội dung trình bày dưới đây là những thuật toán tìm kiếm đường đi thông dụng nhất, đó cũng là những nội dung mà nhóm chúng em nghiên cứu trong đồ án này, bao gồm 2 nội dung chính:

**1. Các thuật toán tìm đường**
- Breadth First Search (BFS)
- Depth First Search (DFS)
- Uniform Cost Search (UCS)
- Dijkstra's algorithm
- A star algorithm

**2. So sánh**
- So sánh BFS và DFS
- So sánh UCS và DIJKSTRA
- So sánh A star và DIJKSTRA

**3. Tổng kết:** Kết quả nghiên cứu đồ án.

Để hoàn thành tốt đồ án này, nhóm chúng em rất biết ơn Thầy Nguyễn Bảo Long. – giảng viên bộ môn Thực hành Cấu trúc dữ liệu và giải thuật đã giúp đỡ chúng em nhiệt tình trong quá trình nghiên cứu và hoàn thành đề tài. Mặc dù nhóm chúng em đã rất cố gắng và nỗ lực để hoàn thành đồ án, nhưng vì kiến thức bản thân còn nhiều hạn chế, trong quá trình soạn bài và hoàn thiện bài báo cáo này chúng em không thể tránh khỏi những sai sót, kính mong nhận được những ý kiến đóng góp từ Thầy để hoàn thiện hơn. Một lần nữa, chúng em xin gửi lời cảm ơn chân thành nhất đến Thầy.

# Mục lục

# PHẦN 1

# PHÂN CÔNG ĐÁNH GIÁ

## 1.1 Bảng phân công nhiệm vụ từng thành viên

| Giai đoạn | Người thực hiện | Nội dung | Kết quả |
|---|---|---|---|
| **23-26/11** | Vĩnh Phong | • [x] Tiếp tục học python<br>• [x] Tìm hiểu anaconda, conda (môi trường chạy python)<br>• [x] Xem thuật toán, đọc các file .py | 100% |
| | Uyên Nhi | • [x] Hoàn thành Python Tutorial (nguồn lát gửi)<br>• [x] Tìm hiểu về conda, anaconda, môi trường chạy python | 100% |
| **27/11-1/12** | Vĩnh Phong | • [x] BFS<br>• [x] DFS | 100% |
| | Uyên Nhi | • [x] UCS<br>• [x]DIJKSTRA | 100% |
| **2/12** | Vĩnh Phong | • [x] So sánh BFS và DFS<br>• [x] Demo BFS, DFS<br>• [x] Tùy chỉnh tổng kết sprint | 100% |
| | Uyên Nhi | • [x] Xem lại thuật toán disktra<br>• [x] So sánh UCS và disktra<br>• [x] Hoàn thành phần demo discktra và UCS | 100% |
| **3-7/12** | Vĩnh Phong | • [x ] Cài đặt tất cả thuật toán và chạy thành công | 100% |
| | Uyên Nhi | • [x] Viết báo cáo, đảm bảo nội dung đã được tinh chỉnh, chọn lọc | 100% |
| **8-9/12** | Vĩnh Phong | • [x] Tiếp tục hoàn thiện Phần code bfs, dfs, ucs<br>• [x] Quay video, đăng link ytb, gửi hình kết quả chạy cho Nhi<br>• [x] Tìm hiểu cách code A* và chạy demo | 100% |
| | Uyên Nhi | • [x] Hoàn thiện đồ án bao gồm cả kết luận, phân công, lời mở đầu,… (chưa bao gồm thuật toán mới)<br>• [x] Tìm hiểu A* và báo cáo lại trễ nhất vào kì họp giám sát lần sau | 100% |
| **10/12** | Vĩnh Phong | • [x] Tổng kết tất cả và hoàn thành nộp | 100% |
| | Uyên Nhi | • [x] Tổng kết tất cả và hoàn thành nộp | 100% |

## 1.2 Bảng tự đánh giá đồ án

| Thứ tự | Tiêu chí | Thang điểm | Tự đánh giá | Nhận xét |
|---|---|---|---|---|
| 1 | Tìm hiểu và trình bày được các thuật toán tìm kiếm trên đồ thị | 4 | 4 | Có thể trình bày được các thuật toán cách đầy đủ |
| 2 | So sánh các thuật toán với nhau | 2 | 2 | Đã so sánh trên nhiều diện khác nhau để thấy rõ sự khác biệt |
| 3 | Cài đặt được các thuật toán kể trên | 3 | 3 | Hoàn thành tốt đúng với những quy định |
| 4 | Tìm hiểu thêm các thuật toán tìm kiếm ngoài yêu cầu(đảm bảo được 3 yếu tố) | 1 | 1 | Có tìm hiểu và cài đặt thêm A* và dijsktra, đáp ứng đủ các tiêu chí |
| | Tổng kết | 10 | 10 | |

# PHẦN 2

# THUẬT TOÁN TÌM KIẾM ĐƯỜNG ĐI

## 2.1 Giải thuật tìm đường

### 2.1.1 Breadth First Search (BFS)

#### Ý tưởng

Thuật toán duyệt ưu tiên các đỉnh gần đỉnh xuất phát hơn trước.

Đầu tiên ta thăm đỉnh nguồn a, đồng thời ta cũng thấy rằng có các đỉnh b1, b2, b3,... kề với a và gần a nhất. Việc thăm đỉnh b1 cũng mở ra các đỉnh khác u1, u2, u3,... kề với b1. Nhưng với thuật toán thì ta sẽ duyệt a rồi đến b1, b2, b3,... rồi mới đến các đỉnh xa hơn như u1, u2, u3,...

Với việc duyệt các đỉnh gần hơn trước mới đến các đỉnh xa, thuật toán sử dụng một hàng đợi để chưa những đỉnh đang chờ được duyệt. Ta lấy ra duyệt đỉnh gần nhất ở đầu hàng đợi, rồi xếp các đỉnh mới kề với đỉnh đang xét ở cuối danh sách.

Thuật toán có thể thực hiện với 2 mục đích: Tìm đường đi từ đỉnh bắt đầu đến đích và tìm đường đi ngắn nhát đến mọi đỉnh trong đồ thị.

#### Mã giả

```
def BFS(graph):
        open_set = [start_vertex] #queue ban dau chua 1 dinh start
        close_set = [] #Tap cac dinh da duoc danh dau (ban dau empty)

        # Thuc hien cho den khi open_set rong
        while open_set:
                current = open_set.pop(0) #Lay 1 vertex trong queue
                close_set.append(current) #Them vao tap da duyet

                #Kiem tra trang thai tung vertex neighbor cua current
                for vertex in neightbor:
                        if (vertex not in(open_set and close_set)):
                                open_set.append(vertex) #Them vao queue
                                father[vertex] = current #Luu lai cha
```

**Đánh giá thuật toán**

- **Tính đầy đủ**

  Là một giải pháp tìm kiếm cho mọi loại đồ thị bất kể đồ thị đó là gì.

- **Độ phức tạp**

  Cách biểu diễn đồ thị (n đỉnh và m cạnh) có ảnh hưởng lớn đến chi phí và thời gian thực hiện thuật toán:

  - Với đồ thị được biểu diễn theo danh sách kề:

  Khi thăm các đỉnh, ta sử dụng một mảng close.set [] để tránh việc duyệt một đỉnh nhiều lần. Việc này tốn chi phí O(n).

  Bất cứ khi nào một dỉnh được thăm thì mọi cạnh kề sẽ được duyệt và ta chỉ tốn O(1) cho 1 cạnh. Và theo nhận xét của Handshaking lemma [4], thì mỗi cạnh sẽ được duyệt một lần với đồ thị có hướng. Điều này tốn chi phí O(m) với m cạnh.

  Do đó, thuật toán này có độ phức tạp O(m + n)

  - Đồ thị biểu diễn bàng ma trận kề:

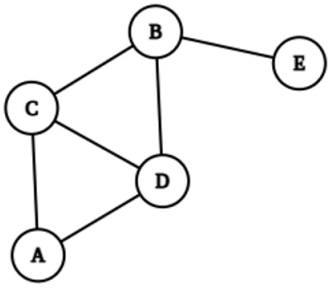
  Để duyệt qua tất cả các đỉnh ta vẫn sẽ mất độ phức tạp là O(n). Và trong khi duyệt 1 đỉnh, ta sẽ phải duyệt qua tất cả các đỉnh khác để kiểm tra đỉnh kề với nó xem có trùng không. Dẫn đến độ phức tạp cuối cùng là O($n^2$)

- **Tính tối ưu**

  Tối ưu trong đồ thị không trọng số.
  Tối ưu trong việc tìm ra đường đi ngắn nhất đến đích hoặc đến mọi điểm, không phải chi phí.
  Xây dựng cây rộng và ngắn.

**Mô phỏng thuật toán**

Ta có một đồ thị như sau:



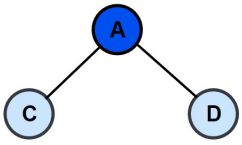

Ta sẽ duyệt qua tất cả các node với:

- Current Node: node đang xét hiên tại
- Open.set: Một hàng đợi (queue) chứa các node chờ được duyệt
- Close.set: Tập đóng chứa các node đã duyệt

Đầu tiên ta thêm vào open.set

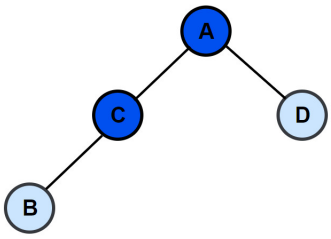

| Lần lặp | Current Node | Open set | Close set |
|---|---|---|---|
| 0 | | A | |

Ở lần lặp đầu tiên, Node A đến được C và D



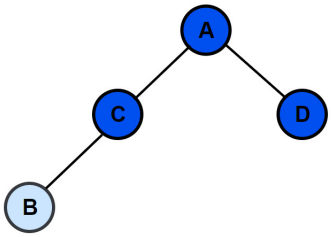
| Lần lặp | Current Node | Open set | Close set |
|---|---|---|---|
| 0 | | A | |
| 1 | A | C, D | A |

Duyệt C trước, ta lại tiếp tục thêm B vào queue và đánh dấu C đã duyệt trong close set



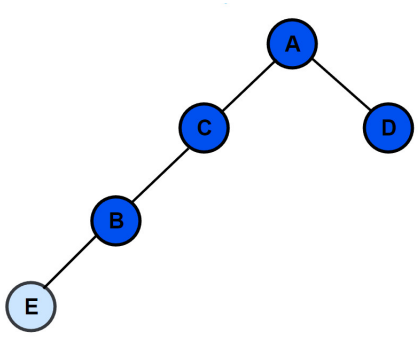

| Lần lặp | Current Node | Open set | Close set |
|---|---|---|---|
| 0 | | A | |
| 1 | A | C, D | A |
| 2 | C | D, B | A, C |

Kế đến, vẫn tiếp tục lấy 1 node trong queue là D để duyệt. Lần này thì D không thể đến node con nào (vì B trước đó đã được duyệt bởi C)



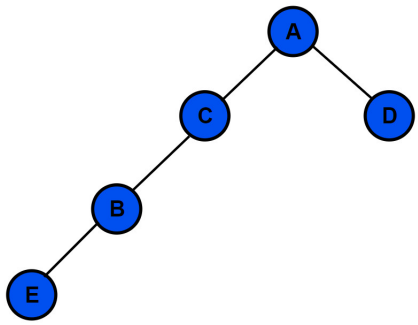

| Lần lặp | Current Node | Open set | Close set |
|---|---|---|---|
| 0 | | A | |
| 1 | A | C, D | A |
| 2 | C | D, B | A, C |
| 3 | D | B | A, C, D |

Sau đó ta tiếp tục duyệt sang B sau khi D vào mục đã xét




| Lần lặp | Current Node | Open set | Close set |
|---|---|---|---|
| 0 | | A | |
| 1 | A | C, D | A |
| 2 | C | D, B | A, C |
| 3 | D | B | A, C, D |
| 4 | B | E | A, C, D, B |

Cuối cùng kà duyệt node E. Đây là node cuối cùng và cũng là node lá




| Lần lặp | Current Node | Open set | Close set |
|---|---|---|---|
| 0 | | A | |
| 1 | A | C, D | A |
| 2 | C | D, B | A, C |
| 3 | D | B | A, C, D |
| 4 | B | E | A, C, D, B |
| 5 | E | | A, C, D, B, E |

Kết thúc khi open set rỗng.

### 2.1.2 Depth First Search (DFS)

**Ý tưởng**

Giải thuật tìm kiếm theo chiều sâu có ý tưởng kiểm tra từ đỉnh bắt đầu, rồi duyệt đến các đỉnh xa nhất có thể của mỗi nhánh.

Đi theo cạnh đến đỉnh kể của đỉnh vừa đánh dấu. Nếu đường đó đến 1 đỉnh mới, ta đánh dấu nó và tiếp tục dò. Nếu không thì quay lại đi đường khác.

Thuật toán quay lui lại vị trí đỉnh đã đánh dấu trước cho đến khi tìm thấy đỉnh mới chưa được đánh dấu (đường mới) và tiếp tục lặp lại các bước trên cho đến khi mọi đỉnh đều được đánh dấu.

Cuối cùng thuật toán kết thúc khi ta trở về xuất phát và đã đánh dấu hết tất cả các đỉnh.

Cũng như BFS, ta chỉ cần thay đổi tiến trình duyệt đỉnh từ hàng đợi (queue) sang (stack) để duyệt các đỉnh mới xa nhất trước.

**Mã giả**

```
#Thuat toan duyet BFS
def DFS(graph):
        open_set = [start_vertex] #queue ban dau chua 1 dinh start
        close_set = [] #Tap cac dinh da duoc danh dau (ban dau empty)

        # Thuc hien cho den khi open_set rong
        while open_set:
                current = open_set.pop() #Lay 1 vertex trong stack
                close_set.append(current) #Them vao tap da duyet

                #Kiem tra trang thai tung vertex neighbor cua current
                for vertex in neightbor:
                        if (vertex not in open_set and close_set):
                                open_set.append(vertex) # Them vao stack
                                father[vertex] = current #Luu lai cha
```

**Đánh giá thuật toán**

- **Tính đầy đủ**

  Là một dạng tìm kiếm thông tin không đầy đủ [5]

- **Độ phức tạp** [6]

  Với n đỉnh và m cạnh thuộc thành phần liên thông chưa đỉnh bắt đầu thì giải thuật có độ phức tạp O(n + m) khi:

  - Đồ thị biểu diễn bằng cấu trúc danh sách kề - Bằng cách đánh dấu các đỉnh đã thăm, ta có thể xét các cạnh kề với đỉnh hiện tại một cách hệ thống không qúa 1 lần.

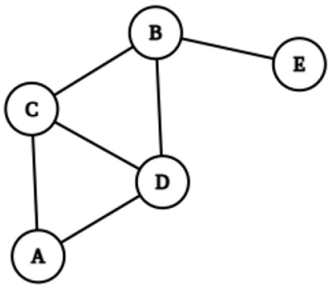
- **Tính tối ưu**

  Sử dụng vùng nhớ hiệu quả khi chiếm không gian tuyến tính lưu nhớ đường dẫn đơn với các đỉnh chưa được khám phá chứ không phải toàn bộ các đỉnh của đồ thị.

  Xây dựng cây hẹp và dài, dùng để kiểm tra các đồ thị theo chu kỳ và thứ tự topô.

**Mô phỏng thuật toán**

Ta có một đồ thị như sau:



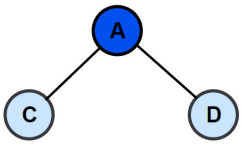

Ta sẽ duyệt qua tất cả các node với:

- Current Node: node đang xét hiên tại
- Open.set: Một ngăn xếp (stack) chứa các node chờ được duyệt
- Close.set: Tập đóng chứa các node đã duyệt

Đầu tiên ta thêm vào open.set

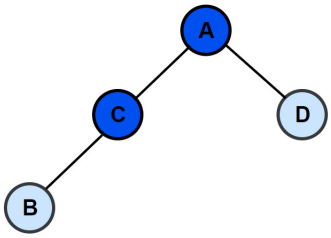

| Lần lặp | Current Node | Open set | Close set |
|---|---|---|---|
| 0 | | A | |

Ở lần lặp đầu tiên, Node A đến được C và D



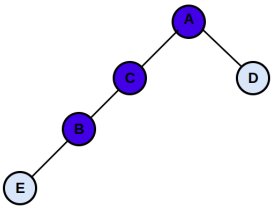

| Lần lặp | Current Node | Open set | Close set |
|---|---|---|---|
| 0 | | A | |
| 1 | A | C, D | A |

Duyệt C trước, ta tìm ra và thêm B vào stack, đánh dấu C đã duyệt trong close set



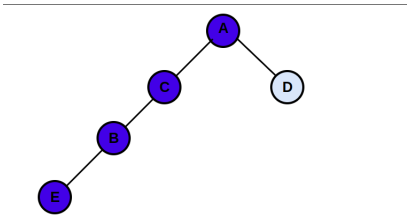

| Lần lặp | Current Node | Open set | Close set |
|---|---|---|---|
| 0 | | A | |
| 1 | A | C, D | A |
| 2 | C | B, D | A, C |

Tiếp tục duyệt B và tìm thấy E.



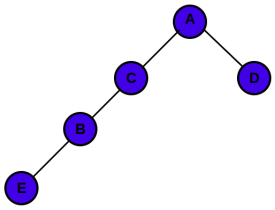
| Lần lặp | Current Node | Open set | Close set |
|---|---|---|---|
| 0 | | A | |
| 1 | A | C, D | A |
| 2 | C | B, D | A, C |
| 3 | B | E, D | A, C, B |

Duyệt đến E, nhận thấy không có node con nào để đi nữa. Ta lại quay trở lại D để tìm đường mới chưa khám phá




| Lần lặp | Current Node | Open set | Close set |
|---|---|---|---|
| 0 | | A | |
| 1 | A | C, D | A |
| 2 | C | B, D | A, C |
| 3 | B | E, D | A, C, B |
| 4 | E | D | |

Cuối cùng là duyệt node D. Đây là node cuối cùng vì không còn node con.




| Lần lặp | Current Node | Open set | Close set |
|---|---|---|---|
| 0 | | A | |
| 1 | A | C, D | A |
| 2 | C | B, D | A, C |
| 3 | B | E, D | A, C, B |
| 4 | E | D | A, C, B, E |
| 5 | D | | A, C, B, E, D |

Thuật toán kết thúc khi stack rỗng.

### 2.1.3 Uniform Cost Search (UCS)

**Ý tưởng**

UCS sử dụng hai danh sách, danh sách open và danh sách close. Danh sách đầu tiên chứa các nút có thể được chọn và danh sách đóng chứa các nút đã được chọn.

1. Cho đỉnh xuất phát vào danh sách open.
2. Nếu open rỗng thì tìm kiếm thất bại và kết thúc tìm kiếm.
3. Lấy đỉnh đầu trong open ra và cho vào danh sách close.
4. Nếu đỉnh đầu là đỉnh đích thì việc tìm kiếm thành công, kết thúc tìm kiếm.
5. Tìm tất cả các đỉnh con của đỉnh đầu ở bước 3 không thuộc danh sách open và close, cho vào open theo thứ tự tăng dần về khoảng cách từ đỉnh xuất phát.
6. Trở lại bước 2

**Mã giả**

```
#UCS
def uniform_cost_search(start, goal state):

    node = start #node bat dau
    open_set = priority queue containing node only #cap nhap hang doi
    close_set = empty list #tao mot danh sach rong
        #Run den khi queue null
    while open_set != NULL:
        if open_set is empty then
            return failure
        current = open_set.pop() #Lay 1 node trong queue

        if current is a goal state then   #neu tim duoc goal
            notice goal founded!
        close_set.add(current) #them vao danh sach dong

        for n in each of neighbors do: #xet nut con
            if (n is not in open_set and close_set) then:
                open_set.add(n) #Neu khong ton tai thi add open_set
            else if (n is already in open_set with higher cost)
                replace existing node by n #thay the bang nut con
```

**Đánh giá thuật toán**

- **Tính đầy đủ**

Tìm kiếm chi phí thống nhất hoàn tất , đầy đủ khi nếu có giải pháp nào khác thì UCS sẽ tìm thấy giải pháp đó.

- **Độ phức tạp**

Đặt C là Chi phí của giải pháp tối ưu và e là mỗi bước để tiến gần hơn đến nút mục tiêu. Sau đó, số bước là = $\frac{C}{e} + 1$. Ở đây chúng ta lấy +1 vì khi chúng ta bắt đầu là từ trạng thái 0 và kết thúc với $\frac{C}{e}$.

Do đó, độ phức tạp thời gian trong trường hợp xấu nhất của tìm kiếm chi phí đồng nhất là O($n^{1+\frac{C}{e}}$).[1]

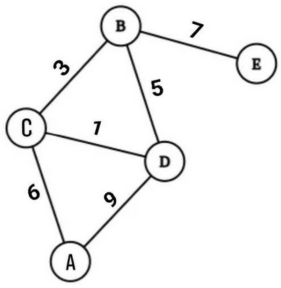
- **Tính tối ưu**

Tìm kiếm chi phí thống nhất luôn tối ưu vì nó chỉ chọn một đường dẫn có chi phí đường đi thấp nhất.

Xây dựng cây hẹp và dài, dùng để kiểm tra các đồ thị theo chu kỳ và thứ tự topô .

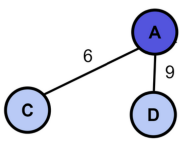
**Mô phỏng thuật toán**

Ta có đồ thị như sau:
Nút bắt đầu là nút A
Nút cần tìm là nút E

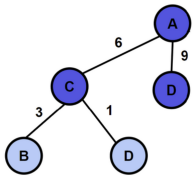



Đầu tiền duyệt A, cho A vào danh sách close.

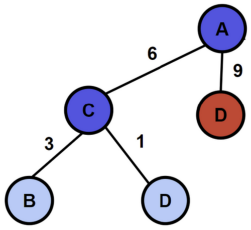
| Current | OpenSet | Close |
|---|---|---|
| A | C(6A), D(9A) | A |



A đi đến 2 nút là C và D với chi phí 6 và 9, thấy từ A đến C có chi phí nhỏ hơn nên tiếp tục cho C vào danh sách close.

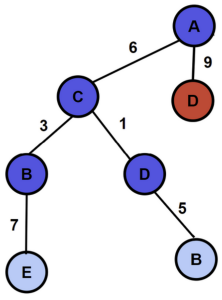



| Current | OpenSet | Close |
|---|---|---|
| A | C(6A), D(9A) | A |
| C | D(7C), B(9C) | C |



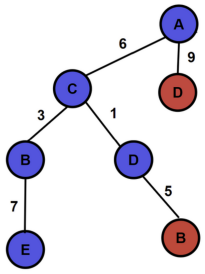

Thấy C đến 2 nút là B và D với chi phí là 9 là 7. Bây giờ ta có 3 nút cần kiểm tra là B9, D7 và D9, thấy D7 có chi phí nhỏ nhất nên bỏ D vào danh sách close.

| Current | OpenSet | Close |
|---|---|---|
| A | C(6A), D(9A) | A |
| C | D(7C), B(9C) | C |
| D | B(9C) | D |




Bây giờ ta có 3 nút cần xét là B9, B12, D9. Ta sẽ xét nút có chi phí nhỏ nhất là B9 và D9, và sẽ xét theo thứ tự chữ cái alpha, beta. Nên ta bỏ B9 và danh sách close để xét.

| Current | OpenSet | Close |
|---|---|---|
| A | C(6A), D(9A) | A |
| C | D(7C), B(9C) | C |
| D | B(9C) | D |
| B | E(16B) | B |




Tìm được E với chi phí là 16.

| Current | OpenSet | Close |
|---|---|---|
| A | C(6A), D(9A) | A |
| C | D(7C), B(9C) | C |
| D | B(9C) | D |
| B | E(16B) | B |
| E | | E |

Kết thúc thuật toán. Ta có đường đi đến E như sau: E <- B <- C <- A

Đây cũng là con đường đến E từ A với chi phí thấp nhất.

### 2.1.4 Dijkstra's algorithm

**Ý tưởng**

1. Đặt khoảng cách đỉnh nguồn là 0, khoảng cách tới mọi điểm trên đồ thị là vô cùng
2. Đặt đỉnh hienj tại là đỉnh xuất phát
3. Đánh dấu đỉnh vừa chọn đã được xét
4. Trong số các cạnh xuất phát từ đỉnh nguồn, ta chọn cạnh có độ dài nhỏ nhất, sau đó gán cho đỉnh đó bằng đỉnh A cộng với độ dài cạnh tương ứng.

5. Xét đến các đỉnh chưa được xét, ta lấy ra đỉnh có khoảng cách từ đỉnh nguồn là ngắn nhất, sau đó tiếp tục quay lại bước 3 là đánh dấu đỉnh đã được xét rồi cập nhập đường đi

6. lặp lại cho tới khi tất cả các đỉnh đều được xét hết

**Mã giả**

```
def Dijkstra(Graph, source):
       for each vertex v in Graph.Vertices:
           dist[v]     INFINITY
 #dist mang chua khoang cach tu nguon den dinh khac
           prev[v]     UNDEFINED
 #prev duong di ngan nhat tu nguon den dinh da cho
           add v to Q #Q la tap cac dinh u
      dist[source]     0
      while Q is not empty:
           u     vertex in Q with min dist[u]
           remove u from Q
           for each neighbor v of u still in Q:
               alt     dist[u] + Graph.Edges(u, v)
# graph.edges khoang cach giua hai nut lan can u va v
#alt la do dai cua duong dan tu nut goc den nut lan can
              if alt < dist[v]:
                  dist[v]     alt
                   prev[v]     u
      return dist[], prev[]
```

**Đánh giá thuật toán**

- **Tính đầy đủ**

  Nó luôn tìm ra giải pháp nếu có

- **Độ phức tạp**

  Độ phức tạp về thời gian của thuật toán Dijkstra phụ thuộc vào cách chúng ta triển khai Q.[3]

  N(V): số đỉnh

  M(E): số cạnh

  Mỗi lần xét N đỉnh sẽ mất O(logN) thời gian. Ngoài ra, chúng ta cần cập nhật hàng đợi ưu tiên khi chúng ta thay đổi giá trị khoảng cách của một đỉnh liền kề. Mỗilần như thế cần có O(logN) thời gian. Có nhiều nhất M các phép toán như vậy, vì có M cạnh.

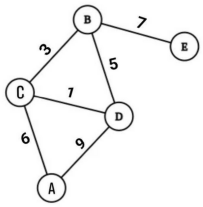
  Do đó, độ phức tạp thời gian chung của thuật toán Dijkstra là : O((N+M)logN).

- **Tính tối ưu**

  Nó luôn tìm ra giải pháp tốt nhất (ít tốn kém nhất).

**Mô phỏng thuật toán**

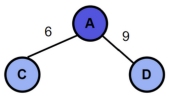
Ta có đồ thị như sau: Nút bắt đầu là nút A




Đầu tiên ta xuất phát từ A

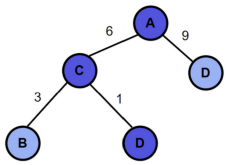

A duyệt đến 2 nút là C và D

| Current | Open Set | Close |
|---|---|---|
| A | C(6A), D(9A) | A |



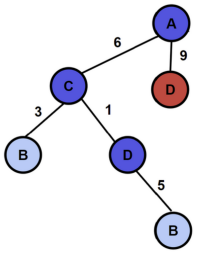

Thấy C có chi phí 6 < 9, tiếp tục chọn C

| Current | OpenSet | Close |
|---|---|---|
| A | C(6A), D(9A) | A |
| C | D(7C), B(9C) | C |



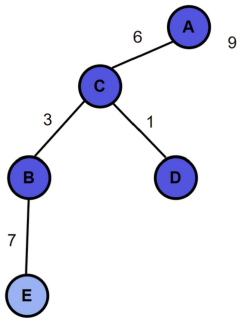

Thấy từ C đến D có chi phí nhỏ hơn từ C qua B, tiếp tục chọn D.

| Current | OpenSet | Close |
|---|---|---|
| A | C(6A), D(9A) | A |
| C | D(7C), B(9C) | C |
| D | B(12D) | D |




Từ D qua B có tổng chi phí là 12 lại lớn hơn từ C qua B, nên ta cập nhập lại cho B là 9.

| Current | OpenSet | Close |
|---|---|---|
| A | C(6A), D(9A) | A |
| C | D(7C), B(9C) | C |
| D | B(9C) | D |




Tiếp tục xét đến E với tổng chi phí đường đi ngắn nhất là 16.

| Current | OpenSet | Close |
|---|---|---|
| A | C(6A), D(9A) | A |
| C | D(7C), B(9C) | C |
| D | B(9C) | D |
| B | E(16B) | B |
| E | | E |

Hoàn thành thuật toán.

### 2.1.5 A star search

**Ý tưởng**

1.Cho đỉnh xuất phát vào open

2.Nếu open rỗng thì tìm kiếm thất bại, kết thúc việc tìm kiếm

3.Lấy đỉnh đầu trong open ra và gọi là O. Cho O vào closed

4.Nếu O là đỉnh đích thì việc tìm kiếm thành công, kết thúc việc tìm kiếm

1. TÌm tất cả các đỉnh con của O không thuộc closed à open theo thứ tự tăng dần với hàm f(x) = g(x) + h(x) 2. Trở lại bước 2

g(x): khoảng cách từ đỉnh đến X

h(x): khoảng cách ước lượng từ X đến đích

**Mã giả**

```
def Dijkstra(Graph, source):
       for each vertex v in Graph.Vertices:
           dist[v]     INFINITY
 #dist mang chua khoang cach tu nguon den dinh khac
           prev[v]     UNDEFINED
 #prev duong di ngan nhat tu nguon den dinh da cho
           add v to Q #Q la tap cac dinh u
      dist[source]     0
      while Q is not empty:
           u     vertex in Q with min dist[u]
           remove u from Q
           for each neighbor v of u still in Q:
               alt     dist[u] + Graph.Edges(u, v)
# graph.edges khoang cach giua hai nut lan can u va v
#alt la do dai cua duong dan tu nut goc den nut lan can
              if alt < dist[v]:
                  dist[v]     alt
                   prev[v]     u
      return dist[], prev[]
```

**Đánh giá thuật toán**

- **Tính đầy đủ**

  Thuật toán có tính chất đầy đủ nếu đánh giá heuristic của nó là thu nạp được (admissible). Đó là tìm kiếm đệ quy theo lựa chọn tốt nhất

- **Độ phức tạp**

  Độ phức tạp của A star phụ thuộc vào heuristic. Trong trường hợp xấu nhất của không gian tìm kiếm không giới hạn, số lượng nút được mở rộng theo cấp số nhân theo độ sâu

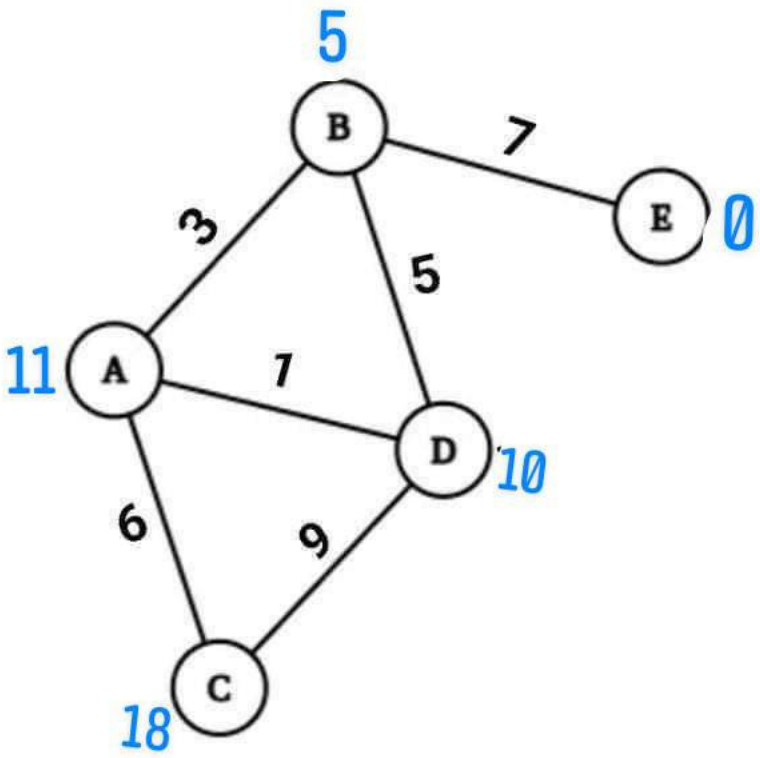
của giải pháp (đường đi ngắn nhất) d : O ($b^d$) , trong đó b là hệ số phân nhánh (số lượng nút kế tiếp trung bình trên mỗi trạng thái ).

- **Tính tối ưu**

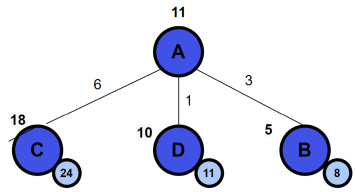
  Với một heuristic nhất quán, Astar được đảm bảo tìm ra đường đi tối ưu mà không cần xử lý bất kỳ nút nào nhiều hơn một lần và Astar tương đương với việc chạy Dijkstra với chi phí giảm [2].

**Mô phỏng thuật toán**

Ta có đồ thị như sau
chọn A là nút bắt đầu
E là nút goal

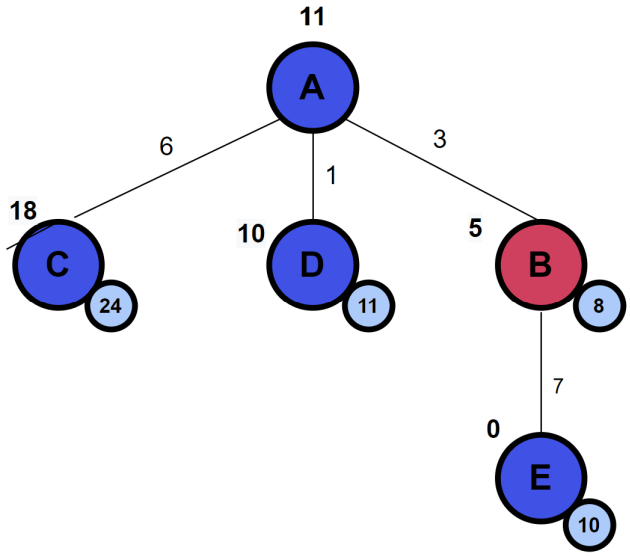



Xét A, A đi tới 3 nút B(3+5), D(1+10),C(6+18). Thấy B(3+5) nhỏ nhất, giữ nguyên D và C




| node | open | closed |
|---|---|---|
| A | C(6+18),B(3+5),D(1+10) | |

Tiếp tục xét B, B đến E(0+10)




| node | open | closed |
|---|---|---|
| A | C(6+18),B(3+5),D(1+10) | |
| B | E(0+10) | A |

Ta thấy đã đến nút goal, nên kết thúc thuật toán.

Tìm được đường đi là A→B→E với chi phí là 10.

Hoàn thành thuật toán.

# 2.2 SO SÁNH

## 2.2.1 So sánh BFS và DFS

| *TIÊU CHÍ* | **BFS** | **DFS** |
|---|---|---|
| *CƠ BẢN* | Thuật toán dựa trên đỉnh | Thuật toán dựa trên cạnh |
| *CẤU TRÚC ĐỂ LƯU CÁC NODE* | Queue (hàng đợi) | Stack (ngắn xếp) |
| *SỬ DỤNG BỘ NHỚ* | Không hiệu quả vì phải lưu giữ các node | Hiệu quả khi chỉ nhớ các đường dẫn đơn đến các node chưa khám phá |
| *CÂY ĐƯỢC XÂY DỰNG* | Rộng, ngắn | Hẹp, dài |
| *TRÌNH TỰ DUYỆT* | Các đỉnh gần nhát trước | Các đỉnh dọc theo cạnh xa nhất trước |
| *TÍNH TỐI ƯU* | Tối ưu cho việc tìm kiếm khoảng cách ngắn nhất (không phải chi phí nhỏ nhất) | Không tối ưu |
| *ỨNG DỤNG* | Kiểm tra biểu đồ lưỡng cực, tìm thành phần liên thông của đồ thị | Sắp xếp tô pô cho đồ thị, kiểm tra đồ thị phẳng |

### 2.2.2 So sánh UCS và DIJKSTRA

| *TIÊU CHÍ* | UCS | DIJKSTRA |
|---|---|---|
| *CƠ BẢN* | Bắt đầu với một đỉnh duy nhất và sau đó mở rộng dần sang các đỉnh khác | Chọn một đỉnh có khoảng cách đến nút gốc là nhỏ nhất trong mỗi bước mở rộng. |
| *MỤC TIÊU* | Dừng khi thấy nút cần tìm | Tiếp tục cho đến khi tất cả các nút bị xóa khỏi hàng đợi ưu tiên |
| *KHÔNG GIAN* | Không hiệu quả vì phải lưu giữ tất cả các node | Hiệu quả khi chỉ nhớ các đường dẫn đơn đến các node chưa khám phá |
| *THỜI GIAN* | Tốn ít hơn dijkstra | Tốn nhiều hơn UCS |
| *ÁP DỤNG* | Có thể áp dụng cho cả đồ thị rõ ràng và đồ thị ẩn | Chỉ được áp dụng với biểu đồ rõ ràng **khi biết tất cả các đỉnh và các cạnh.** |
| *BỘ NHỚ* | Không cần nhiều bộ nhớ vì chỉ lưu trữ đỉnh gốc và ngừng mở rộng khi chúng ta đến đỉnh đích | Yêu cầu nhiều bộ nhớ hơn vì phải lưu trữ toàn bộ biểu đồ trong bộ nhớ. |

### 2.2.3 So sánh A STAR và DIJKSTRA

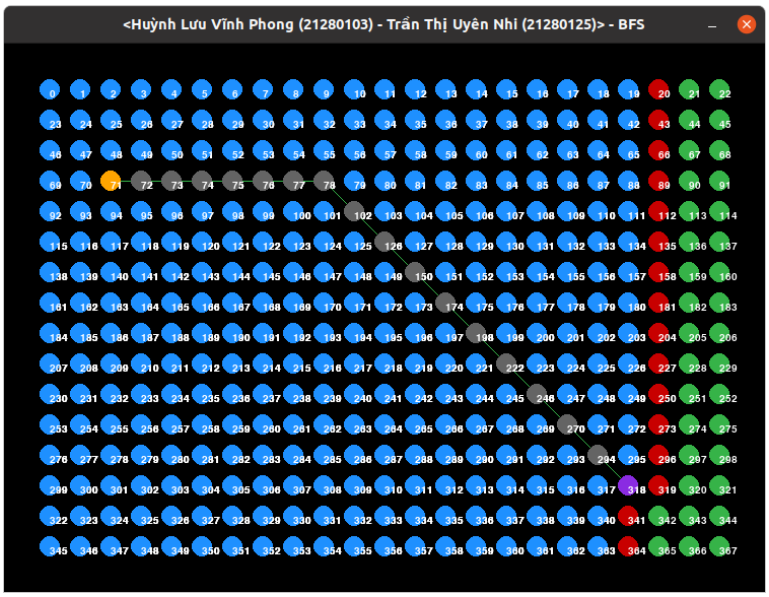
| ***TIÊU CHÍ*** | **Astar** | **Dijkstra** |
|---|---|---|
| *Chi phí* | Tìm thấy chi phí tối thiểu từ node bắt đầu đến node mục tiêu | Tìm thấy chi phí tối thiểu từ node bắt đầu cho tất cả các node khác |
| *Hiệu quả* | Tốt hơn | Kém hơn |
| *Cơ bản* | Là phiên bản nâng cao của Dijkstra | |
| *Độ ưu tiên* | Hàm của chi phí(chi phí các nút trước đó + chi phí để có được ở đây) | Chỉ bị ảnh hưởng bởi chi phí thực tế |
| *Cách duyệt* | Không xét một nút nhiều lần | Xét nhiều lần một nút |
| *Thời gian* | Rất nhanh | Tốn nhiều thời gian |
| *ỨNG DỤNG* | Phù hợp việc tìm đường | Chủ yếu để phát hiện nhiều node mục tiêu |

# PHẦN 3

# KẾT QUẢ NGHIÊN CỨU

Sau khi tìm hiểu và cài đặt các thuật toán , nhóm nghiên cứu đã thu được các kết quả thông qua hình demo như sau:

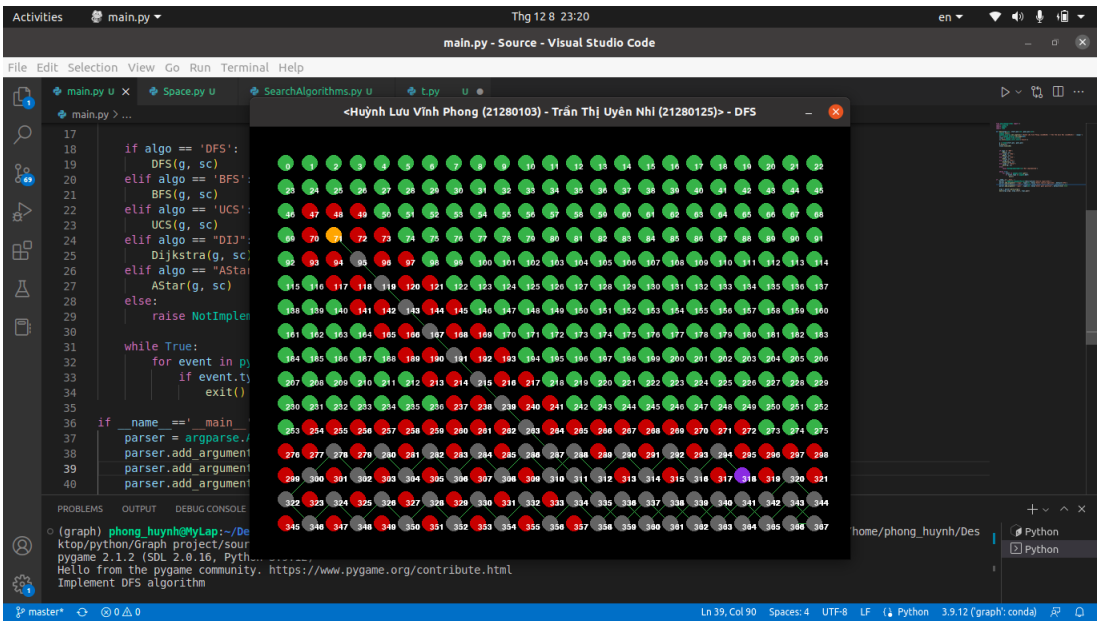
**1. BFS**




**Breadth First Search(BFS)**

Ta thấy rằng BFS thì nó sẽ lần theo các node gần nó và lần ra với các node xa hơn nếu các node gần đã được thăm hết. Các node được duyệt lan tỏa ra như một đám cháy từ 1 điểm đến toàn bộ graph cho đến khi tìm được goal. Thuật toán này trả ra đường đi ngắn nhất (không tính chi phí) đến đích với đồ thị không trọng số.

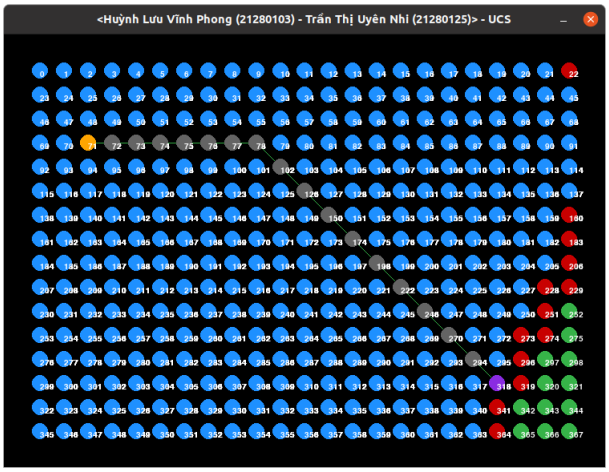
**2. DFS**




**Depth First Search**

DFS nó sẽ lần các node theo chiều sâu đồng nghĩa với việc nó sẽ lần các node ngoài vô trong đến khi đến đỉnh thì dừng và lần lại các dấu vết đã lưu ở father

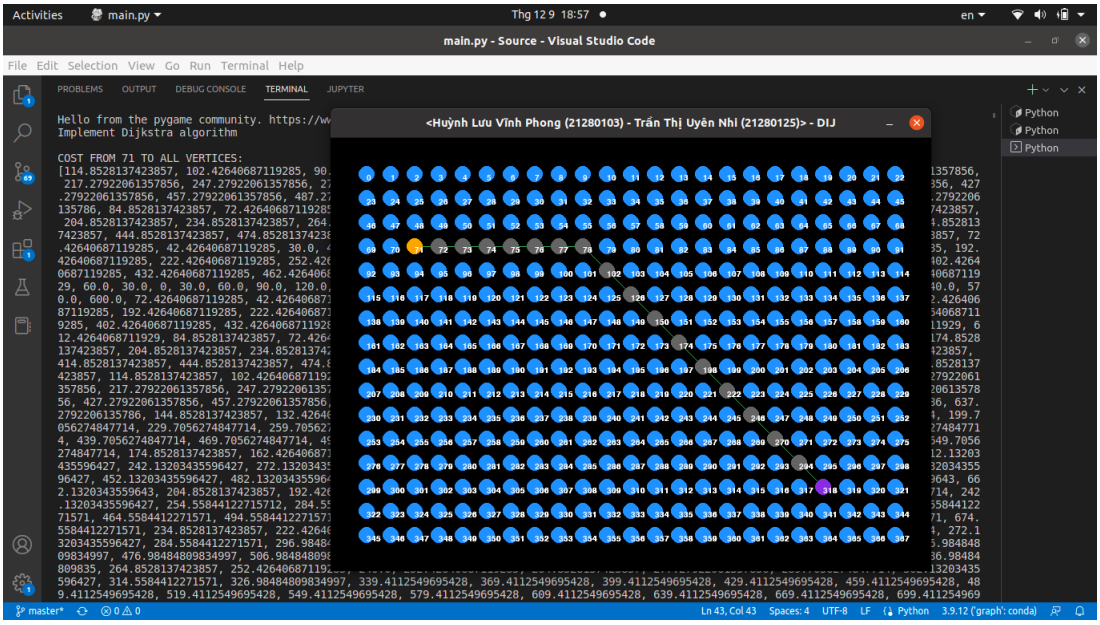
### 3. UCS




**Uniform Cost Search**

Thuật toán UCS luôn chọn con đường có chi phí thấp nhất để đi trước. Thuật toán lan rộng tỏa ra xung quanh và đi nhanh hơn ở hướng có chi phí thấp. Do đó nó luôn đảm bảo đường đến goal là đường có chi phí thấp nhất có thể.

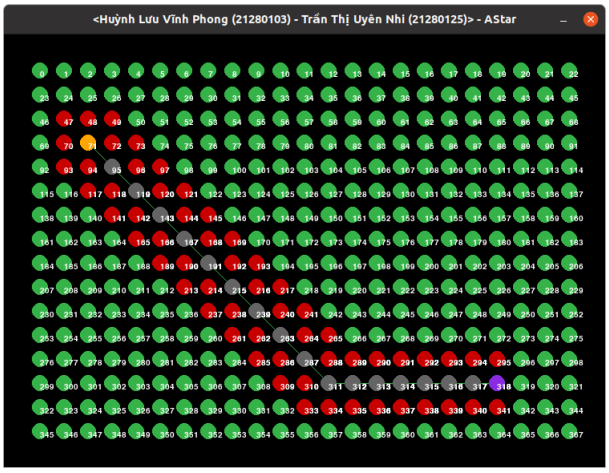
### 4. DIJKSTRA




**Dijkstra's Algortihm**

Ta thấy rằng Dijsktra với UCS là như nhau về đường đi nhưng UCS đến đỉnh goal thì dừng còn Dijsktra sẽ duyệt hết các node có trong đồ thị rồi trả ra đường đi từ start đến tất cả các node .nên ta thấy được thời gian của Dijsktra sẽ lâu hớn so với UCS.

### 5. A STAR




**A Star Search Algorithm**

Và ta thấy rằng AStar dường như là nhanh nhất và tối ưu nhất khi ta thực hiện thuật toán này với các thuật toán tìm kiếm dường đi khác. Các node được duyệt luôn đảm bảo hướng đến đích và có chi phí ngắn nhất. Qua đó có thể thấy, thuật toán này biểu hiện một cách "thông minh" trong việc tìm đường với chi phí nhỏ nhất.

**Tổng kết**

Qua những thông tin đã được tìm hiểu về các thuật toán tìm kiếm cũng có thể thấy được mức độ thích hợp của mỗi thuật toán khác nhau cho từng mục đích sử dụng khác nhau. Tuy nhiên không thể kết luận rằng thuật toán nào tốt là sử sụng hoàn toàn. Mà còn phải tùy từng hoàn cảnh trường hợp đánh giá khác nhau để áp dụng chúng cho phù hợp.

# PHẦN 4
# KẾT LUẬN

Đồ án này giúp chúng em tìm hiểu thêm về các thuật toán tìm đường và cài đặt các thuật toán này. Để đánh giá các thuật toán sắp xếp cũng như hiểu thêm về cách hoạt động của thuật toán. Tuy có nhiều khó khăn trong việc tìm hiểu đồ án nhưng chúng em đã cố gắng hoàn thành tốt và chỉn chu về nội dung của đồ án cũng như về phần trình bày.

Qua thời gian thực hiện đồ án. Chúng em đã nắm bắt được phần nào về thuật toán tìm kiếm. Tuy mỗi thuật toán có những ưu điểm và khuyết điểm khác nhau tùy vào các điều kiện để áp dụng vào các trường hợp để thuật toán chạy tốt nhất có thể.

Mặc dù đã rất cố gắng cho việc tìm hiểu và nghiên cứu đồ án, nhưng do thời gian và hiểu biết của chúng em vẫn còn hạn chế nên có thể vẫn sẽ xảy ra sai sót gì đó. Nên chúng em mong nhận được ý kiến đóng góp từ thầy.

*Chúng em xin chân thành cảm ơn!*

# Tài liệu tham khảo


[1] Andreas Soularidis. Uniform cost search (ucs) algorithm in python. 2022.

[2] Wikipedia. A* search algorithm. 2022.

[3] Wikipedia. Dijkstra's algorithm. 2022.

[4] Wikipedia. Handshaking lemma. 2022.

[5] Wikipedia. Tìm kiếm theo chiều sâu. 2022.

[6] Dương Anh Đức. *Nhập môn Cấu Trúc Dữ Liệu và Giải Thuật*. Đại Học Khoa Học Tự Nhiên TP.HCM, 2022.