ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

ĐỒ ÁN

MÔN: CƠ SỞ TRÍ TUỆ NHÂN TẠO

CÁC THUẬT TOÁN TÌM KIẾM TRÊN ĐỒ THỊ

GIÁO VIÊN HƯỚNG DẪN: NGUYỄN BẢO LONG

SINH VIÊN THỰC HIỆN: 20120482 - NGUYỄN TẠ HUY HOÀNG

LỚP: 20CTT3

THÀNH PHỐ HỒ CHÍ MINH – 8/10/2022

Mục Lục

I. Tìm hiểu & trình bày thuật toán

1. Giới thiệu bài toán tìm kiếm trên đồ thị

Bài toán tìm kiếm trên đồ thị được phát biểu chung cho cả đồ thị có hướng hay vô hướng với ý nghĩa một cạnh vô hướng xem như đi theo chiều nào cũng được. Có thể nói, trong việc giải quyết nhiều bài toán ứng dụng trên đồ thị, thao tác tìm kiếm được dùng như là một trong những thao tác cơ bản, đến mức vai trò của nó trong ứng dụng đồ thị cũng giống như vai trò của các phép cộng, trừ, ... trong tính toán số học.

2. Thành phần của một bài toán tìm kiếm

Một bài toán tìm kiếm có thể được định nghĩa bằng 5 thành phần:

- Trạng thái bắt đầu (Initial state)
- Mô tả các hành động (action) có thể thực hiện
- Mô hình di chuyển (transition model): mô tả kết quả của các hành động
  - Thuật ngữ successor tương ứng với các trạng thái có thể di chuyển được với một hành động duy nhất
  - Trạng thái bắt đầu, hành động và mô hình di chuyển định nghĩa không gian trạng thái (state space) của bài toán
  - Không gian trạng thái hình thành nên một đồ thị có hướng với đỉnh là các trạng thái và cạnh là các hành động
  - Một đường đi trong không gian trạng thái là một chuỗi các trạng thái được kết nối bằng một chuỗi các hành động
- Kiểm tra đích (goal test): xác định một trạng thái có là trạng thái đích
- Một hàm chi phí đường đi (path cost) gán chi phí với giá trị số cho mỗi đường đi
  - Chi phí đường đi khi thực hiện hành động a từ trạng thái s để đến trạng thái s′ ký hiệu c(s, a, s′)

3. Cách giải một bài toán tìm kiếm nói chung

- Một lời giải (solution) là một chuỗi hành động di chuyển từ trạng thái bắt đầu cho đến trạng thái đích.
- Để giải một bài toán tìm kiếm ta thực hiện qua các bước sau:
  - Bước 1: Xác định trạng thái bắt đầu **(Initial state)** của bài toán và yêu cầu kết thúc cuả bài toán.
  - Bước 2: Xác định thuật toán tìm kiếm càn áp dụng cho bài toán để tìm lời giải.
  - Bước 3: Thực hiện lần lượt các hành động **(action)** để tạo nên mô hình di chuyển **(transition model)**, trong quá trình này cần chú đến chi phí đường **đi (path cost)**

- Bước 4: Kiểm tra đích **(goal test)** nếu sai thì tiếp tục thực hiện hành động, nếu đúng thì trả về kết quả => tạo ra một lời giải.

- Một lời giải tối ưu có chi phí đường đi thấp nhất trong số tất cả lời giải.

4. Phân loại

- Hầu hết các thuật toán được nghiên cứu bởi các nhà khoa học máy tính để giải quyết các bài toán đều là các thuật toán tìm kiếm. Tập hợp tất cả các lời giải có thể đối với một bài toán được gọi là không gian tìm kiếm. Thuật toán thử sai (**brute-force search**) hay các thuật toán tìm kiếm "sơ đẳng" không có thông tin sử dụng phương pháp đơn giản nhất và trực quan nhất. Trong khi đó, các thuật toán tìm kiếm có thông tin sử dụng **heuristics** để áp dụng các tri thức về cấu trúc của không gian tìm kiếm nhằm giảm thời gian cần thiết cho việc tìm kiếm.

- Ta có thể chia thành hai thuật toán tìm kiếm cơ bản:
  - Tìm kiếm không có thông tin (Uninformed Search)
  - Tìm kiếm có thông tin (Informed Search)

4.1 Tìm kiếm không có thông tin (Uninformed search Algorithm)

- **Uninformed search Algorithm** là một loại thuật toán tìm kiếm có mục đích chung hoạt động theo cách brute force. Các thuật toán tìm kiếm không thông tin không có thêm thông tin về trạng thái hoặc không gian tìm kiếm ngoài cách đi qua cây, vì vậy nó còn được gọi là tìm kiếm mù
- Nhược điểm của các giải thuật này là phần lớn các không gian tìm kiếm kích thước cực kì lớn, và một quá trình tìm kiếm (đặc biệt tìm kiếm theo cây) sẽ cần một khoảng thời gian đáng kể cho các ví dụ nhỏ. Do đó, để tăng tốc độ quá trình tìm kiếm, đôi khi chỉ có thể dùng giải thuật tìm kiếm có thông tin
- Một số thuật toán tìm kiếm mù: BFS, DFS, UCS

4.2 Tìm kiếm có thông tin (Informed search Algorithm)

- **Informed search Algorithm**, ngắn gọn, chứa một loạt kiến thức, như khoảng cách đến mục tiêu, chi phí đường dẫn, các bước để đến nút mục tiêu, trong số những bước khác, giúp nó tiếp cận giải pháp hiệu quả hơn mà không ảnh hưởng đến thời gian và chi phí tìm kiếm. Tuy nhiên, để đạt được điều này, nó sử dụng chức năng heuristic, tiếp tục dựa trên đánh giá được áp dụng cho các trạng thái trong cây tìm kiếm. Các đặc điểm khác của Thuật toán tìm kiếm được thông báo là:
  - Nó còn được gọi là Tìm kiếm có hướng dẫn hoặc Tìm kiếm heuristic.
  - Cố gắng giảm số lượng tìm kiếm.
  - Phụ thuộc vào chức năng đánh giá để xác định thứ tự các hoạt động.
  - Số lượng các nút dẫn đến heuristic tốt được đặt ở phía trước.
  - Đánh giá thường dựa trên các quan sát thực nghiệm.
  - Nó hữu ích hơn cho không gian tìm kiếm lớn.

- Hơn nữa, thành phần cụ thể theo vấn đề này của tìm kiếm được thông báo có thể được gọi là heuristic có thể chấp nhận được trong trường hợp khi chi phí ước tính luôn thấp hơn hoặc bằng chi phí thực tế để đạt được trạng thái mục tiêu. Chính kiến thức về vấn đề cụ thể này được sử dụng bởi các chiến lược tìm kiếm sáng suốt để tìm ra giải pháp tối ưu nhanh hơn.
- Sự chấp nhận của hàm heuristic được đưa ra là: **h (n) < = h * (n)** Trong đó **h (n)** là chi phí heuristic và **h * (n)** chi phí ước tính.

5. Một số thuật toán tìm kiếm cơ bản

5.1 Breadth-First Search (BFS)

- **Ý tưởng chính:** thuật toán BFS bắt đầu tìm kiếm từ nút gốc của cây và mở rộng tất cả các nút kế thừa ở cấp hiện tại trước khi chuyển sang các nút của cấp tiếp theo. BFS được triển khai bằng cách sử dụng cấu trúc dữ liệu hàng đợi FIFO.
- **Thuận lợi:** Nếu có nhiều hơn một giải pháp cho một vấn đề nhất định, thì BFS sẽ cung cấp giải pháp tối thiểu yêu cầu số bước ít nhất.
- **Nhược điểm:**
  + Nó yêu cầu nhiều bộ nhớ vì mỗi cấp độ của cây phải được lưu vào bộ nhớ để mở rộng cấp độ tiếp theo.
  + BFS cần nhiều thời gian nếu giải pháp ở xa nút gốc.

- **Time Complexity**: Độ phức tạp về thời gian của thuật toán BFS có thể được tính bằng số lượng nút được duyệt trong BFS cho đến nút nông nhất. Trong đó d = độ sâu của nghiệm nông nhất và b là một nút ở mọi trạng thái.

  T (b) = 1 + b2 + b3 + ……. + bd = O (bd)

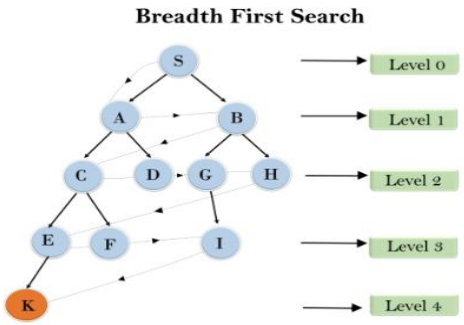
- **Space Complexity**: Độ phức tạp không gian của thuật toán BFS được cung cấp bởi kích thước bộ nhớ của biên giới là **O (bd).**

- **Completeness**: BFS đã hoàn thành, có nghĩa là nếu nút mục tiêu nông nhất nằm ở độ sâu hữu hạn nào đó, thì BFS sẽ tìm ra giải pháp.

- **Optimality**: BFS là tối ưu nếu chi phí đường dẫn là một hàm không giảm của độ sâu của nút.

- **Ví dụ**: Thuật toán sử dụng thuật toán BFS từ nút gốc S đến nút mục tiêu K. Thuật toán tìm kiếm BFS đi ngang trong các lớp, vì vậy nó sẽ đi theo đường dẫn được hiển thị bằng mũi tên chấm, và đường đi ngang sẽ là:

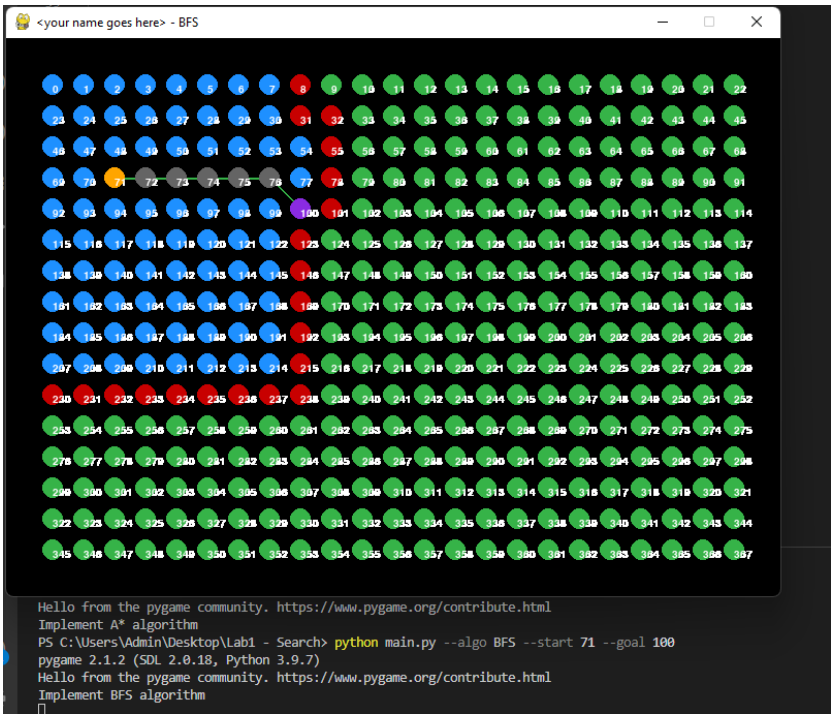
S → A → B → C → D → G → H → E → F → I → K



- **Mã giả**

```
open_set: list[Node] = [g.start], closed_set: list[Node] = []
i → (0,len(open_set)
    while len(open_set) != 0:
        N_current = open_set[i]
        if N_current = g.start: return
        open_set.remove(N_current)
        closed_set.append(N_current)
        neighbors = g.get_neighbors(N_current)
        while len(neighbors) != 0:
            if neighbors[0] not in closed_set and open_set:
                open_set.append(neighbors[0])
            neighbors.remove(neighbors[0])
```

- **Kết quả khi chạy chương trình:**



5.2 Depth-First Search (DFS)

- **Ý tưởng chính:** Ta sẽ bắt đầu tìm kiếm từ một đỉnh v_0 nào đó của đồ thị. Sau đó chọn u là một đỉnh tuỳ ý kề với v_0 và lặp lại quá trình đối với u. Ở bước tổng quát, giả sử ta đang xét đỉnh v. Nếu như trong số các đỉnh kề với v tìm được đỉnh w là chưa được xét thì ta sẽ xét đỉnh này (nó sẽ trở thành đã xét) và bắt đầu từ nó ta sẽ bắt đầu quá trình tìm kiếm còn nếu như không còn đỉnh nào kề với v là chưa xét thì ta nói rằng đỉnh này đã duyệt xong và quay trở lại tiếp tục tìm kiếm từ đỉnh mà trước đó ta đến được đỉnh v (nếu $v=v_0$, thì kết thúc tìm kiếm). Có thể nói tìm kiếm theo chiều sâu bắt đầu từ đỉnh v được thực hiện trên cơ sở tìm kiếm theo chiều sâu từ tất cả các đỉnh chưa xét kề với v.
- DFS sử dụng cấu trúc dữ liệu ngăn xếp để thực hiện nó.
- **Thuận lợi:**
  + DFS yêu cầu rất ít bộ nhớ vì nó chỉ cần lưu trữ một chồng các nút trên đường dẫn từ nút gốc đến nút hiện tại.
  + Mất ít thời gian hơn để đến được nút mục tiêu so với thuật toán BFS (nếu nó đi đúng đường)
- **Nhược điểm:** Có khả năng nhiều trạng thái tiếp tục tái diễn và không có gì đảm bảo cho việc tìm ra giải pháp. Thuật toán DFS dùng để tìm kiếm sâu hơn và đôi khi nó có thể đi đến vòng lặp vô hạn.
- **Completeness**: Thuật toán tìm kiếm DFS hoàn chỉnh trong không gian trạng thái hữu hạn vì nó sẽ mở rộng mọi nút trong cây tìm kiếm giới hạn

- **Time Complexity**: Độ phức tạp thời gian của DFS sẽ tương đương với nút được thuật toán duyệt qua. Nó được đưa ra bởi:

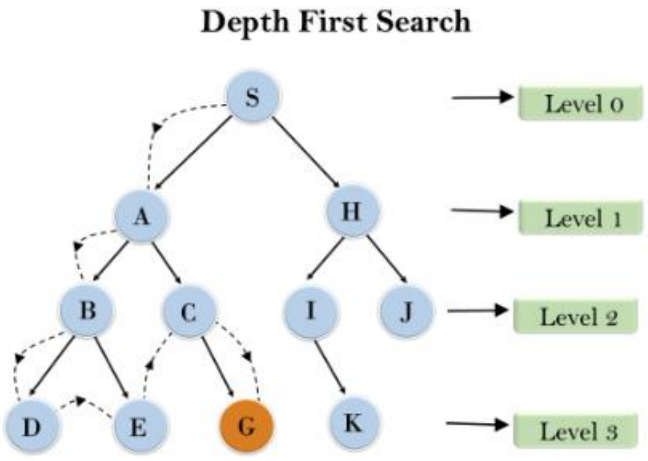
  T(n) = 1 + n2 + n3 +… + nm = O(nm)

  Trong đó, m = độ sâu tối đa của bất kỳ nút nào và giá trị này có thể lớn hơn nhiều so với d (nghiệm nông nhất)

- **Space Complexity**: Thuật toán DFS chỉ cần lưu trữ một đường dẫn duy nhất từ nút gốc, do đó độ phức tạp không gian của DFS tương đương với kích thước của tập rìa, là **O(bm)**

- **Optimal**: Thuật toán tìm kiếm DFS không tối ưu, vì nó có thể tạo ra một số lượng lớn các bước hoặc chi phí cao để đạt đến nút mục tiêu.

- **Ví dụ:** Nó sẽ bắt đầu tìm kiếm từ nút gốc S, và đi ngang qua A, rồi đến B, rồi D và E, sau khi đi qua E, nó sẽ dò ngược cây vì E không có nút kế thừa nào khác và vẫn không tìm thấy nút mục tiêu. Sau khi bẻ khóa ngược, nó sẽ đi ngang qua nút C và sau đó là G, và tại đây nó sẽ kết thúc khi tìm thấy nút mục tiêu.

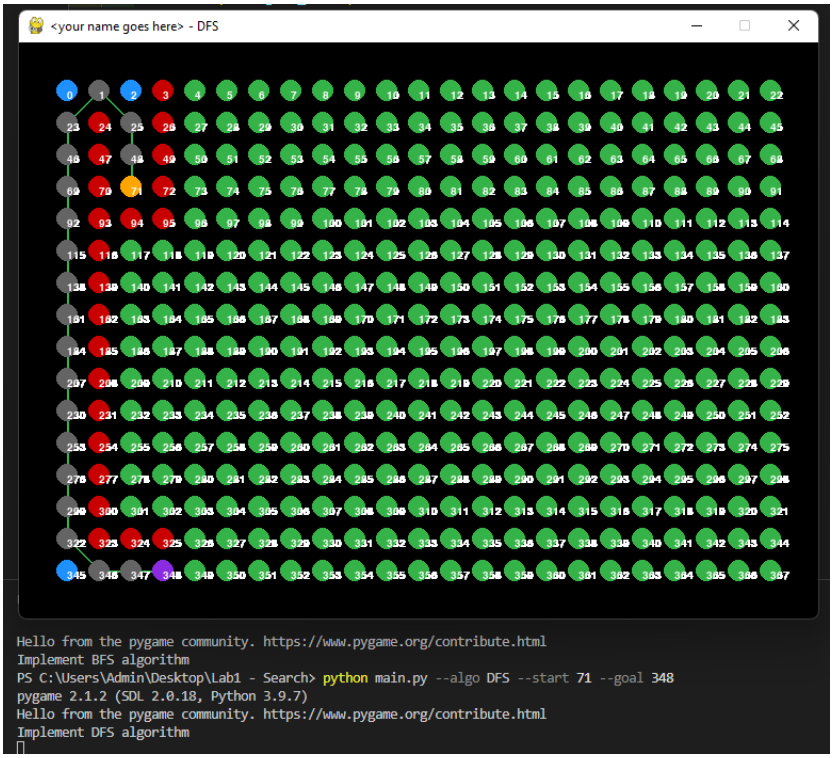


- **Mã giả**

```
open_set: list[Node] = [g.start]
closed_set: list[Node] = []
i → (0,len(open_set)
   while len(open_set) != 0:
      N_current = open_set[i]
      if N_current = g.start: return
      open_set.remove(N_current)
      closed_set.append(N_current)
```

```
neighbors = g.get_neighbors(N_current)
temp: list[Node] = []
while len(neighbors) != 0:
    if neighbors[0] not in closed_set and open_set:
        temp.append(neighbors[0])
    neighbors.remove(neighbors[0])
open_set[0:0] = temp
```

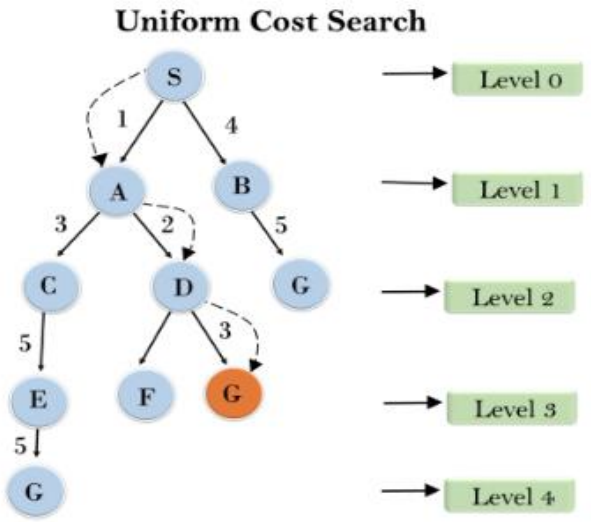
- **Kết quả khi chạy chương trình:**



5.3 Uniform cost search (UCS)

- **Ý tưởng chính:** Thuật toán này phát huy tác dụng khi có sẵn một chi phí khác nhau cho mỗi cạnh. Mục tiêu chính của UCS là tìm đường dẫn đến nút mục tiêu có chi phí tích lũy thấp nhất. UCS mở rộng các nút theo chi phí đường dẫn của chúng tạo thành nút gốc. Nó có thể được sử dụng để giải quyết bất kỳ biểu đồ / cây nào có nhu cầu về chi phí tối ưu. Thuật toán UCS được thực hiện bởi hàng đợi ưu tiên. Nó ưu tiên tối đa cho chi phí tích lũy thấp nhất. UCS tương đương với thuật toán BFS nếu chi phí đường dẫn của tất cả các cạnh là như nhau.
- **Thuận lợi:** tối ưu ở mọi trạng thái, con đường có chi phí thấp nhất được chọn.
- **Nhược điểm:** Nó không quan tâm đến số bước liên quan đến việc tìm kiếm và chỉ quan tâm đến chi phí đường dẫn. Do đó thuật toán này có thể bị mắc kẹt trong một vòng lặp vô hạn.

- **Ví dụ:**



- **Completeness:** UCS đã hoàn tất, chẳng hạn như nếu có giải pháp, UCS sẽ tìm ra giải pháp đó
- **Time Complexity:** Gọi C* là chi phí của giải pháp tối ưu và ε là mỗi bước để tiến gần hơn đến nút mục tiêu. Khi đó số bước là C* / ε + 1. Ở đây chúng tôi đã lấy +1, khi chúng tôi bắt đầu từ trạng thái 0 và kết thúc đến C* / ε
  Do đó, độ phức tạp thời gian trong trường hợp xấu nhất của UCS là **O(b1+[C* / ε])**
- **Space Complexity:** Logic tương tự là đối với độ phức tạp không gian, vì vậy, độ phức tạp không gian trong trường hợp xấu nhất của UCS là **O (b1 + [C* / ε]).**
- **Optimal:** Tìm kiếm theo chi phí thống nhất luôn tối ưu vì nó chỉ chọn một đường dẫn có chi phí đường dẫn thấp nhất.
  Mã giả: Được cài đặt tương tự như BFS nhưng bổ sung chi phí thấp nhất giữa node đang xét và nút được xét đến

```
Cost_two_Node(self, a: Node, b: Node)
        return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2))
Min_Cost(g: Graph, open_set: list[Node], cost):
    min = g.get_len()
    temp = open_set[0]
    for i in open_set:
        if min > cost[i.value]:
            min = cost[i.value]
            temp = i
    return temp
N_current = Min_Cost(g, open_set, cost)
while len(neighbors) != 0:
    if neighbors[0] not in closed_set and open_set:
```

```
                cost[neighbors[0].value] = cost[N_current.value] + \
                    Cost_two_Node(N_current, neighbors[0])
                open_set.append(neighbors[0])
            neighbors.remove(neighbors[0])
```

- **Kết quả khi chạy chương trình:**

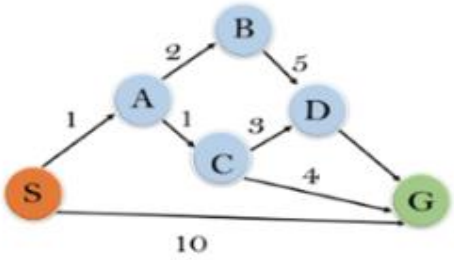
5.4 A* Search Algorithm (A*)

- **Ý tưởng chung:** A* search Algorithm là hình thức tìm kiếm ưu tiên nhất được biết đến nhiều nhất. Nó sử dụng hàm heuristic h(n) và chi phí để đạt được nút n từ trạng thái bắt đầu g(n). Nó đã kết hợp các tính năng của UCS và tìm kiếm ưu tiên tốt nhất tham lam, nhờ đó nó giải quyết vấn đề một cách hiệu quả. Thuật toán tìm kiếm A* tìm đường đi ngắn nhất qua không gian tìm kiếm bằng cách sử dụng hàm heuristic. Thuật toán tìm kiếm này mở rộng ít cây tìm kiếm hơn và cung cấp kết quả tối ưu nhanh hơn. Thuật toán A* tương tự như UCS ngoại trừ việc nó sử dụng g(n) + h (n) thay vì g (n).
- **Thuận lợi:** Thuật toán tìm kiếm A* là thuật toán tối ưu và hoàn chỉnh. Tốt nhất so với các thuật toán tìm kiếm khác và có thể giải quyết các vấn đề rất phức tạp
- **Nhược điểm:**

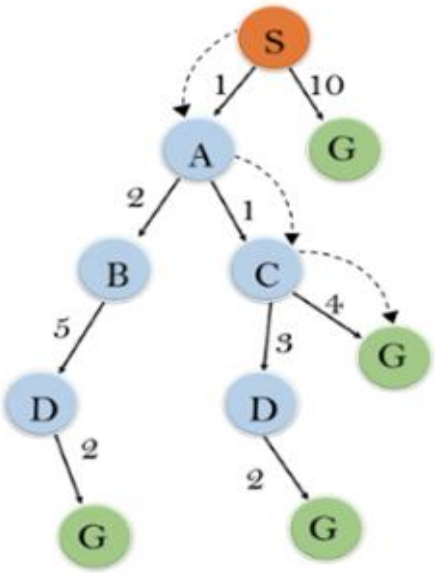
  + Nó không phải lúc nào cũng tạo ra đường đi ngắn nhất vì nó chủ yếu dựa trên heuristics và tính gần đúng.
  + Thuật toán tìm kiếm A* có một số vấn đề phức tạp

+ Hạn chế chính của A* là yêu cầu bộ nhớ vì nó giữ tất cả các nút được tạo trong bộ nhớ, vì vậy nó không thực tế cho các vấn đề quy mô lớn khác nhau.

- **Ví dụ**



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0



Khởi tạo: {(S, 5)}

F1: {(S → A, 4), (S → G, 10)}

F2: {(S → A → C, 4), (S → A → B, 7), (S → G, 10)}

F3: {(S → A → C → G, 6), (S → A → C → D, 11), (S → A → B, 7), (S → G, 10)}

F4: sẽ cho kết quả cuối cùng, vì S → A → C → G nó cho đường dẫn tối ưu với chi phí 6.

- **Time Complexity**: Độ phức tạp về thời gian của thuật toán tìm kiếm A* phụ thuộc vào hàm heuristic và số lượng nút được mở rộng theo cấp số nhân với độ sâu của nghiệm d. Vì vậy, độ phức tạp thời gian là O (b^d), trong đó b là hệ số phân nhánh.
- **Space Complexity**: Độ phức tạp không gian của thuật toán tìm kiếm A* là O(b^d)
- **Completeness:** Thuật toán A* hoàn thành miễn là:
  - Hệ số phân nhánh là hữu hạn.
  - Chi phí cho mọi hành động là cố định.
- **Optimal:** Thuật toán tìm kiếm A* là tối ưu nếu nó tuân theo hai điều kiện sau:
  - **Có thể chấp nhận:** điều kiện đầu tiên yêu cầu để tối ưu là h(n) phải là một heuristic có thể chấp nhận được cho tìm kiếm cây A*. Một heuristic có thể chấp nhận được là bản chất lạc quan
  - h(n) là một heuristic hợp lý khi: Không ước lượng quá cao chi phí đến đích và $h(n) \leq h^*(n)$ chi phí thấp nhất từ n đến đích
  - **Tính nhất quán:** Điều kiện bắt buộc thứ hai là tính nhất quán chỉ dành cho tìm kiếm đồ thị A*

❖ Lưu ý:
- Thuật toán A* trả về đường dẫn xuất hiện đầu tiên và nó không tìm kiếm tất cả các đường dẫn còn lại.
- Hiệu quả của thuật toán A* phụ thuộc vào chất lượng của heuristic.
- Thuật toán A* mở rộng tất cả các nút thỏa mãn điều kiện f(n).

Nếu hàm heuristic được chấp nhận, thì tìm kiếm cây A* sẽ luôn tìm ra đường dẫn có chi phí thấp nhất

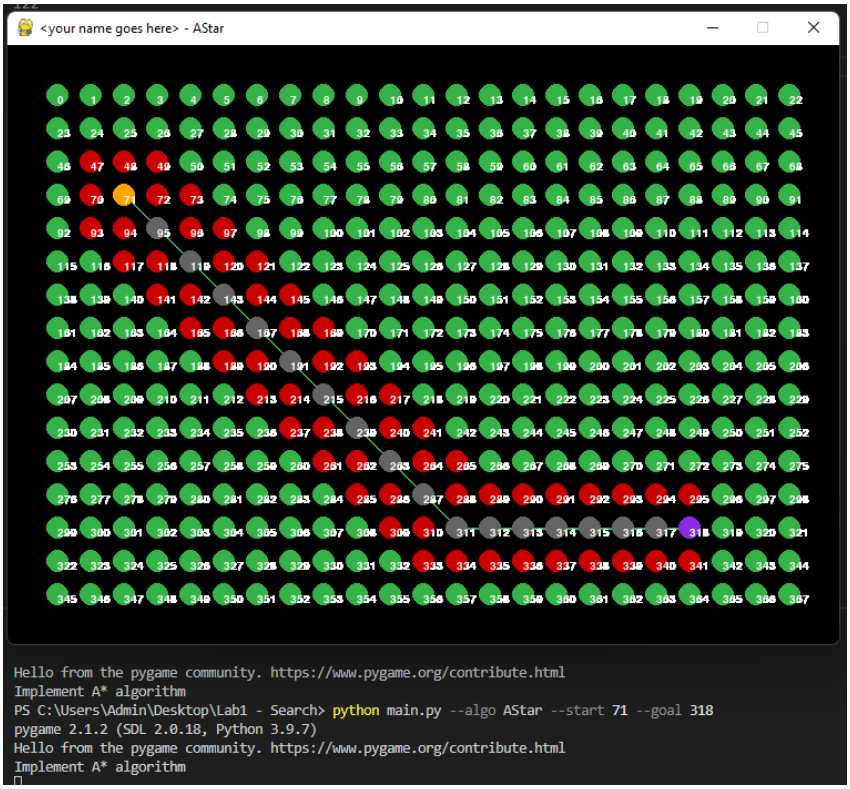
- Một heuristic là **chấp nhận được** nếu nó *không bao giờ đánh giá quá cao* chi phí thực sự để đạt được nút mục tiêu từ n. Nếu một heuristic là **nhất quán** , thì giá trị heuristic của n không bao giờ lớn hơn chi phí của người kế nhiệm nó n′, cộng với giá trị heuristic của người kế nhiệm.

- **Mã giả:**

```
Cost_two_Node(self, a: Node, b: Node)
        return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2))

def heuristic(self, node: Node):
   return abs(self.goal.x - node.x) + abs(self.goal.y - node.y)

def Min_Cost_Heuristic(g: Graph, open_set: list[Node], cost):
   min = g.get_len()
   temp = open_set[0]
   for i in open_set:
      if min > cost[i.value] + g.heuristic(i):
```

```
                min = cost[i.value] + g.heuristic(i)
                temp = i
        return temp

N_current = Min_Cost(g, open_set, cost)
while len(neighbors) != 0:
        if neighbors[0] not in closed_set and open_set:
            cost[neighbors[0].value] = cost[N_current.value] + \
                Cost_two_Node(N_current, neighbors[0])
            open_set.append(neighbors[0])
        neighbors.remove(neighbors[0])
```

- **Kết quả khi chạy chương trình**



II. So sánh

1. So sánh sự khác biệt giữa UCS , Greedy và A* .

Thuật toán	UCS	Greedy	A*
Phân loại	Uninformed Search	Informed search	Informed search
Chiến lược	Sẽ chọn tổng chi phí thấp nhất từ toàn bộ cây.	Tìm kiếm theo hàm heruristic Cố gắng mở các nút được ước lượng gần với đích nhất	Tìm kiếm theo hàm heruristic + chi phí từ node hiện tại tới node n
Độ phức tạp thời gian	$O(b^{1+[C^*/€]})$	$O(b^m)$	$O(b^d)$
Độ phức tạp không gian	$O(b^{1+[C^*/€]})$	$O(b^m)$ giữ hết các nút trong bộ nhớ	$O(b^d)$
Chi phí đường đi	Tính từ node bắt đầu đến node hiện tại	Chí tính từ node trước đến node hiện tại	
Tính tối ưu	Chỉ dùng g(n) nên chậm	Chỉ dựa h(n) nên không tối ưu	Sử dụng cả g(n) và h(n) Có nếu heuristic hợp lý và nhất quán
Tính hoàn thành	Luôn tìm thấy		Có nếu € > 0 và không gian trạng thái hữu hạn
Danh sách các node chờ duyệt	Hàng đợi ưu tiên	Hàng đợi ưu tiên	
Nhận xét chung	Khi đồ thi có chi phí ở mỗi bước là như nhau thì thuật toán trở thành phương pháp tìm kiếm theo chiều rộng.	Thuật toán GBFS lại quá phụ thuộc vào hàm heuristics. Mà khi hàm heuristics không tốt (ước tính không chuẩn) sẽ dẫn đến những đường đi không tối ưu	Thuật toán A* là 1 thuật toán tối ưu về tìm kiếm đường đi, rất linh động nhưng vẫn gặp một khuyết điểm cơ bản - đó là tốn khá nhiều bộ nhớ để lưu lại những trạng thái đã đi qua. A* là một thuật toán rất thông minh, được ***Google Maps*** sử dụng trong bài toán tìm đường đi ngắn nhất thời gian thực!

Ghi chú:

- Hàm đánh giá: f(n)=g(n) + h(n)
  g(n) – Chi phí từ node hiện tại tới node n
  h(n) – ước lượng khoảng cách từ node n -> đích
- d là độ sâu của cây cần duyệt, m là độ sâu tối đa của cây và mỗi trạng thái khi được phát triển sẽ sinh ra b trạng thái kề (b được gọi là nhân tố nhánh).
- Với chi phí di chuyển thấp nhất là ε, $C*$ là chi phí lời giải tối ưu

2. So sánh sự khác biệt giữa UCS và Dijkstra

Thuật toán của Dijkstra tìm đường dẫn ngắn nhất từ nút gốc đến mọi nút khác. tìm kiếm chi phí thống nhất cho các đường dẫn ngắn nhất về chi phí từ nút gốc đến nút mục tiêu. Tìm kiếm chi phí thống nhất là Thuật toán của Dijkstra tập trung vào việc tìm ra một con đường ngắn nhất đến một điểm kết thúc duy nhất thay vì con đường ngắn nhất đến mọi điểm.

UCS thực hiện điều này bằng cách dừng lại ngay khi tìm thấy điểm kết thúc. Đối với Dijkstra, không có trạng thái mục tiêu và quá trình xử lý tiếp tục cho đến khi tất cả các nút đã bị xóa khỏi hàng đợi ưu tiên, tức là cho đến khi các đường dẫn ngắn nhất đến tất cả các nút (không chỉ là một nút mục tiêu) đã được xác định.

UCS có ít yêu cầu về không gian hơn, trong đó hàng đợi ưu tiên được lấp đầy dần dần trái ngược với Dijkstra, bổ sung tất cả các nút vào hàng đợi khi bắt đầu với chi phí vô hạn.

Do những điểm trên, Dijkstra tốn nhiều thời gian hơn UCS

UCS thường được xây dựng trên cây trong khi Dijkstra được sử dụng trên đồ thị chung

Djikstra chỉ có thể áp dụng trong các biểu đồ rõ ràng trong đó toàn bộ biểu đồ được đưa ra làm đầu vào. UCS bắt đầu với đỉnh nguồn và dần dần đi qua các phần cần thiết của biểu đồ. Do đó, nó có thể áp dụng cho cả đồ thị rõ ràng và đồ thị ngầm (nơi các trạng thái / nút được tạo ra).

III. Tài liệu tham khảo

- websitehcm.com
- https://stackoverflow.com/
- https://cloud.ducnn.com/s/RZSiwZbLeMTD32W
- Các giải thuật tìm kiếm trên đồ thị (viblo.asia)