BÁO CÁO CUỐI KỲ

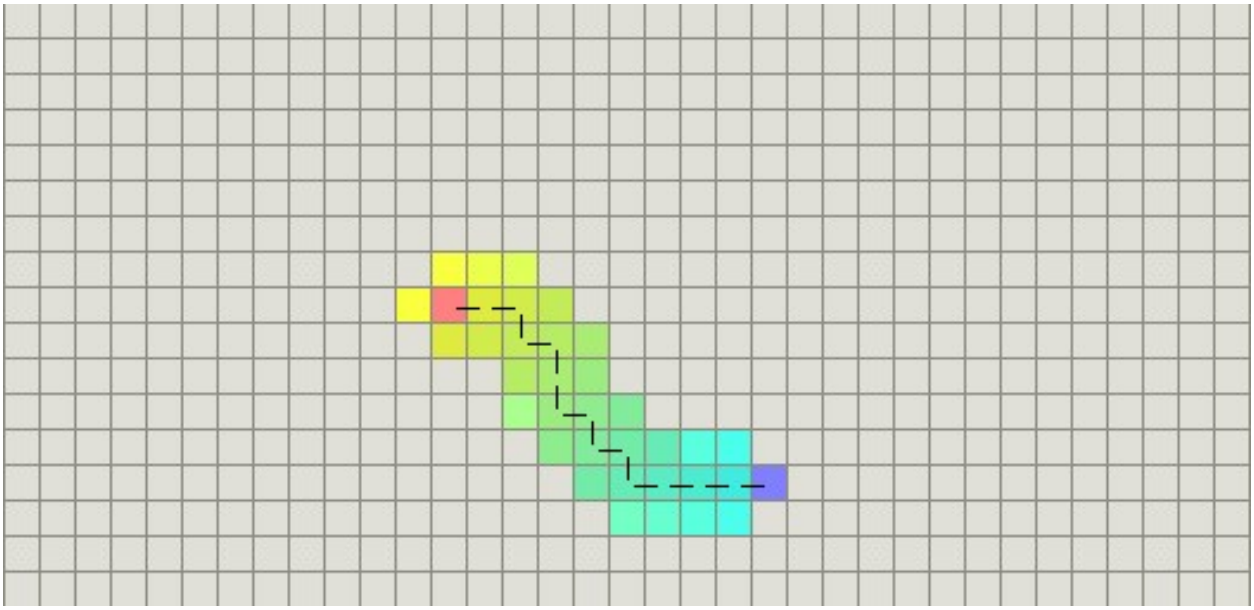
Thuật toán tìm đường A*

và ứng dụng trên mạng lưới tàu điện ngầm London

—

Thành viên - Mã học viên:

- Lê Thanh Tùng - 20007905
- Phạm Tuấn Cường - 20007928
- Phùng Lệ Diễm – 20007927



[Go to the App]

MỤC LỤC

Một số khái niệm đã học

1. Các khái niệm cơ bản về thuật toán và các phương pháp thiết kế thuật toán

"Thuật toán + Cấu trúc dữ liệu = Chương trình"

(Algorithms + Data Structures = Programs)

Lý thuyết

Bài toán được cấu tạo bởi hai thành phần cơ bản:

- ➢ Thông tin vào (input): Cung cấp cho ta các dữ liệu đã có
- ➢ Thông tin ra (output): Những yếu tố cần xác định.

Thuật toán là một dãy hữu hạn các thao tác đơn giản được sắp xếp theo một trình tự xác định dùng để giải một bài toán.

Một số phương pháp diễn đạt thuật toán

- ➢ Liệt kê từng bước
- ➢ Sơ đồ khối
- ➢ Giả mã (pseudo-code)

Phân tích để đánh giá thuật toán:

- ➢ Có đúng đắn không?
- ➢ Có hiệu quả không?

Các thuật toán

- ➢ Kỹ thuật đệ quy
- ➢ Phương pháp chia để trị
- ➢ Phương pháp quay lui
- ➢ Phương pháp nhánh cận
- ➢ Phương pháp quy hoạch động
- ➢ Phương pháp tham lam (Greedy)
- ➢ Phương pháp trực tuyến (Online)

- ➢ Phương pháp ngẫu nhiên (Randomized)
- ➢ Phương pháp xấp xỉ (Approximation)

Một số ứng dụng

Bài toán gióng hàng hai trình tự toàn cục (global pairwise sequence alignment) trong tin sinh học và giải thuật Needleman-Wunsch theo phương pháp quy hoạch động.

Giải bài toán cho một số nguyên (N) và một mảng (arr) gồm các số nguyên gồm các số nhỏ hơn N, tìm trong mảng arr một mảng con (sub_arr) ngắn nhất có tổng bằng N bằng phương pháp quy hoạch động.

2. Đường đi, chu trình trong đồ thị và ứng dụng

Lý thuyết

Đồ thị vô hướng G: là một cặp không có thứ tự (unordered pair) G:=(V, E), trong đó:

- ➢ **V:** tập các đỉnh hoặc nút
- ➢ **E:** tập các cặp không thứ tự chứa các đỉnh phân biệt, được gọi là cạnh. Hai đỉnh thuộc một cạnh được gọi là các đỉnh đầu cuối của cạnh đó.

Đồ thị có hướng G: là một cặp có thứ tự G:=(V, A), trong đó

- ➢ **V**: tập các đỉnh hoặc nút,
- ➢ **A**: tập các cặp có thứ tự chứa các đỉnh, được gọi là các cạnh có hướng hoặc cung. Một cạnh e = (x, y) được coi là có hướng từ x tới y; x được gọi là điểm đầu/gốc và y được gọi là điểm cuối/ngọn của cạnh.

Đồ thị hỗn hợp G là một bộ ba có thứ tự G:= (V,E,A) với V, E và A được định nghĩa như trên.

Một khuyên (loop) là một cạnh (vô hướng hoặc có hướng) nối từ một đỉnh về chính nó.

Đơn đồ thị là đồ thị mà không có khuyên và không có cạnh bội

Đa đồ thị là đồ thị mà không thỏa mãn đơn đồ thị

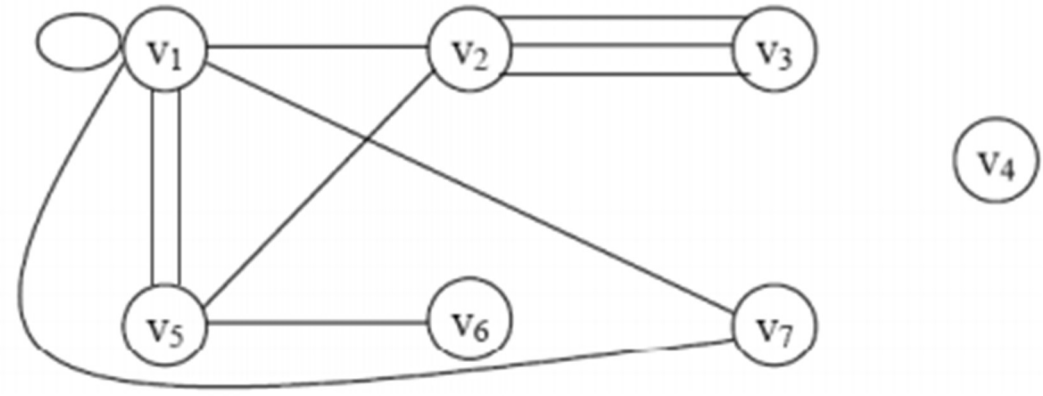
Đa đồ thị có hướng là một đồ thị có hướng, trong đó, nếu x và y là hai đỉnh thì đồ thị được phép có cả hai cung (x, y) và (y, x).

Đơn đồ thị có hướng là một đồ thị có hướng, trong đó, nếu x và y là hai đỉnh thì đồ thị chỉ được phép có tối đa một trong hai cung (x, y) hoặc (y, x).

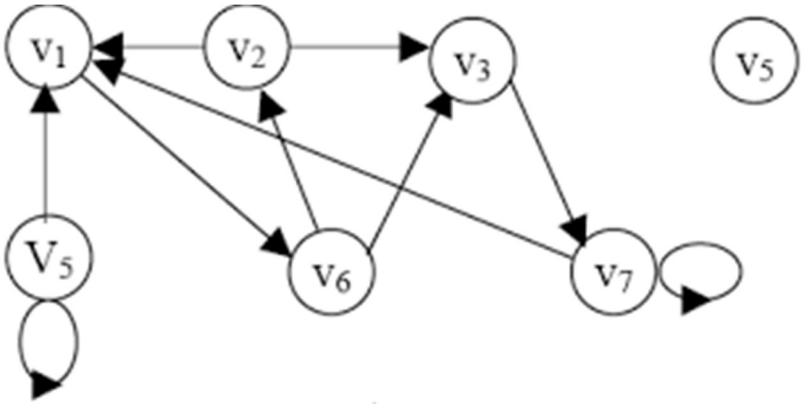
Đồ thị có trọng số là đồ thị mà mỗi cạnh được gắn với một giá trị nào đó, được gọi là trọng số, độ dài, chi phí, hoặc các tên khác tùy theo ứng dụng.

Tính chất:

1. Hai cạnh của một đồ thị được coi là kề nhau nếu chúng có chung một đỉnh.
2. Hai đỉnh được coi là kề nhau nếu chúng được nối với nhau bởi một cạnh.
3. Một cạnh và đỉnh nằm trên cạnh đó được coi là liên thuộc với nhau.
4. Bậc của đỉnh trong đồ thị vô hướng G = (V,E), ký hiệu deg(v), là số các cạnh liên thuộc với nó, riêng khuyên tại một đỉnh được tính hai lần cho bậc của nó.
5. Bậc của đỉnh trong đồ thị có hướng: bậc ra (bậc vào) của đỉnh v trong đồ thị có hướng là số cung của đồ thị đi ra khỏi nó (đi vào nó) và ký hiệu là deg+(v) (deg- (v)).



Hình 1. Đa đồ thị vô hướng. deg(v1) = 7



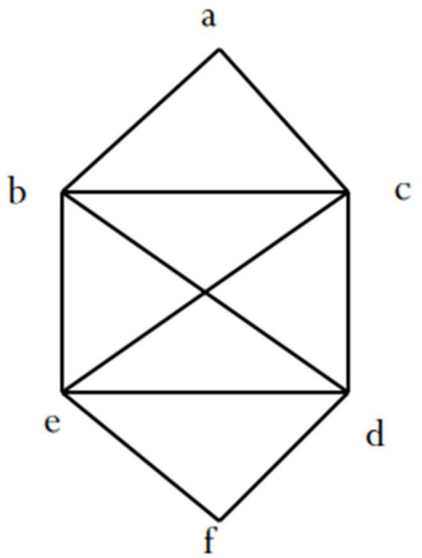
Hình 2. Đa đồ thị có hướng. deg+(v1) = 1, deg-(v1) = 3

Đường đi Euler: là đường đi mà đi qua tất cả các cạnh của đồ thị, mỗi cạnh đi qua đúng một lần.

Chu trình Euler: là đường đi Euler mà có đỉnh đầu và đỉnh cuối trùng nhau.

Đường đi Hamilton: là đường đi mà đi qua tất cả các đỉnh của đồ thị, mỗi đỉnh đúng một lần.

Chu trình Hamilton: là chu trình xuất phát từ một đỉnh, đi thăm tất cả những đỉnh còn lại mỗi đỉnh đúng 1 lần, cuối cùng quay trở lại đỉnh xuất phát.



Hình 1. Đường đi Euler là: e - b - a - c - b - d - c - e - d

Chu trình Euler là: a - b - c - e - b - d - e - f - d - c - a

Đường đi Hamilton là: a - b - c - f - d - c

Chu trình Hamilton là: a - b - c - f - d - c - a

Một số thuật toán về đường đi, chu trình

- ➢ Thuật toán Dijkstra
- ➢ Thuật toán Bellman–Ford (Richard Bellman, Lester Ford Jr - 1958).
- ➢ Thuật toán A* search (Peter Hart, Nils Nilsson, Bertram Raphael – 1968).
- ➢ Thuật toán Floyd–Warshall (Robert Floyd, Stephen Warshall – 1962).
- ➢ Thuật toán Johnson (Donald B. Johnson – 1977).
- ➢ Thuật toán Viterbi (Andrew Viterbi – 1967).

Ứng dụng

Có rất nhiều ứng dụng của đồ thị Euler trong cuộc sống của chúng ta, cụ thể là một số bài toán sau:

- ➢ Bài toán Domino
- ➢ Bài toán người phát thư Trung Hoa
- ➢ Lắp ráp đoạn DNA

3. Cây bao trùm tối thiểu của đồ thị và ứng dụng

Lý thuyết

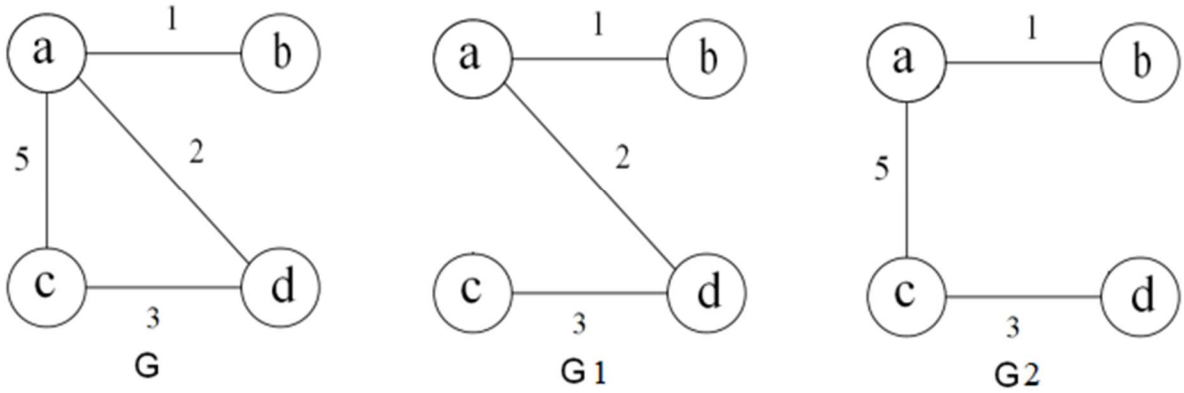
Cây là một đồ thị vô hướng liên thông và không có bất kỳ chu trình nào.

Không có chu trình: không có đường đi khép kín qua các đỉnh

Chỉ có 1 đường đi duy nhất giữa hai đỉnh.

Cây bao trùm của một đồ thị là cây (gồm các đỉnh, cạnh của G) có chứa tất cả các đỉnh của G.

Cây bao trùm tối thiểu của một đồ thị liên thông có trọng số G là cây bao trùm có tổng trọng số các cạnh là nhỏ nhất.



Hình 3: Đồ thị G là một cây với cây bao trùm G1, G2 và cây bao trùm tối thiểu là G1 có tổng trọng số là 6

Tính chất

1. Có thể có một vài cây bao trùm nhỏ nhất có cùng trọng số và có số cạnh nhỏ nhất; cụ thể hơn, nếu tất cả các cạnh của một đồ thị đều có trọng số bằng nhau, thì tất cả các cây bao trùm của đồ thị đó đều là nhỏ nhất.
2. Tính duy nhất: Nếu mỗi cạnh có trọng số riêng biệt thì sẽ chỉ có một, và chỉ một cây bao trùm nhỏ nhất
3. Tính chất vòng: Với một chu trình C bất kỳ trong đồ thị, nếu trọng số của cạnh e nào đó của C lớn hơn trọng số của các cạnh còn lại của C, thì cạnh đó không thể thuộc về cây bao trùm nhỏ nhất.
4. Tính chất cắt: Với nhát cắt C bất kỳ trong đồ thị, nếu trọng số của một cạnh e của C nhỏ hơn trọng số của các cạnh còn lại của C, thì cạnh này thuộc về tất cả các cây bao trùm nhỏ nhất của đồ thị.
5. Cạnh có trọng số nhỏ nhất: Nếu một cạnh của đồ thị với trọng số nhỏ nhất e là duy nhất, thì cạnh này sẽ thuộc về bất kỳ một cây bao trùm nhỏ nhất nào

Một số thuật toán về cây bao trùm tối thiểu

Thuật toán Prim

1. Bắt đầu với cây chỉ có 1 đỉnh T0.
2. Phát triển cây theo các bước, mỗi bước thêm một đỉnh vào cây đã có bằng một cạnh. Dãy các cây được phát triển T1, T2, . . . , Tn-1.

3. Chiến lược tham lam: Tại mỗi bước dựng cây Ti+1 từ cây Ti với việc thêm vào đỉnh "gần nhất".
4. Đỉnh gần nhất với Ti: Đỉnh không thuộc Ti và được nối với Ti bằng cạnh có trọng số nhỏ nhất.
5. Thuật toán dừng lại khi tất cả các đỉnh đã được thêm vào.

Thuật toán Kruskal

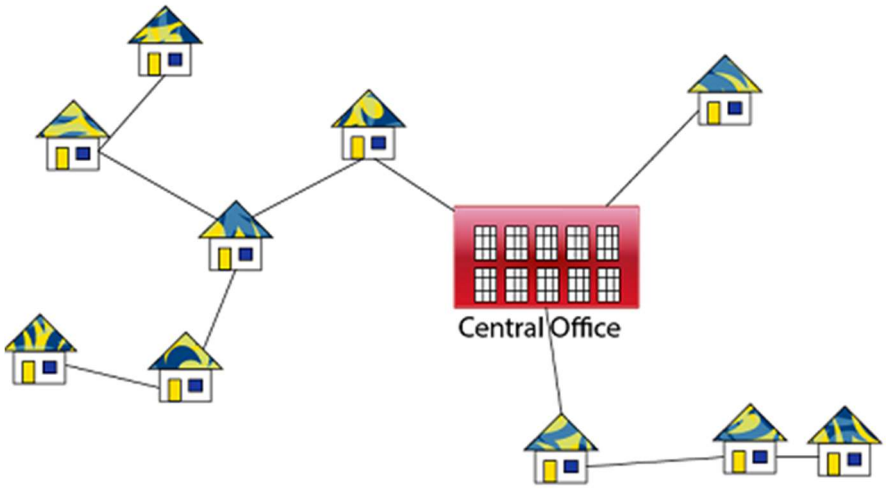
1. Các cạnh được sắp xếp theo thứ tự tăng dần của trọng số.
2. Bắt đầu bằng một rừng (forest) rỗng.
3. Xây dựng MST theo các bước, mỗi bước thêm một cạnh
    a. Trong quá trình dựng MST luôn có một "rừng": các cây không liên thông.
    b. Thêm vào cạnh có trọng số nhỏ nhất trong các cạnh chưa thêm vào cây và không tạo thành chu trình.
    c. Như vậy tại mỗi bước một cạnh có thể:
        i. Mở rộng một cây đã có
        ii. Nối hai cây thành một cây mới
        iii. Tạo cây mới
4. Thuật toán dừng lại khi tất cả các đỉnh đã được thêm vào.

Thuật toán Borůvka

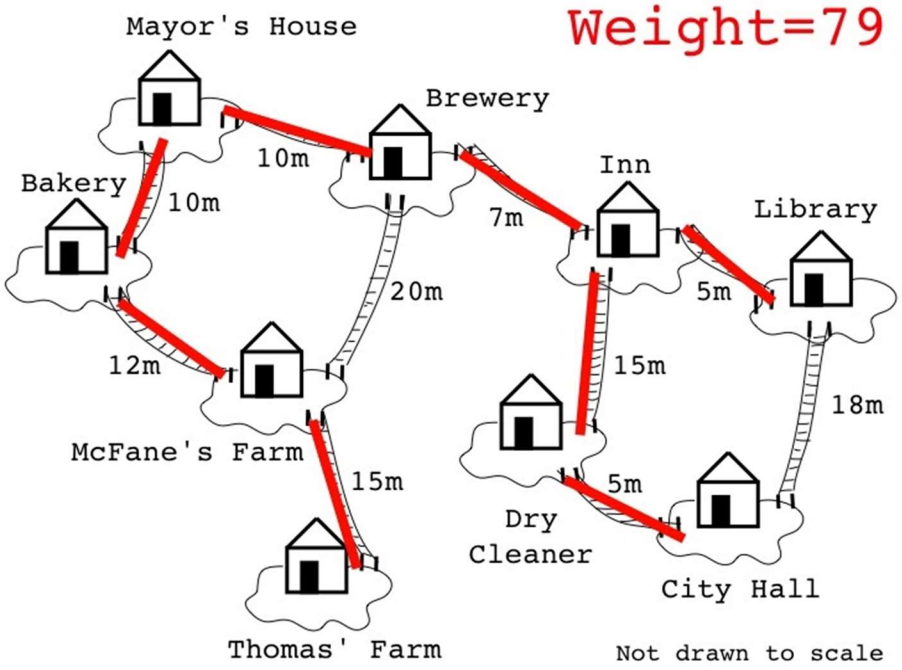
1. Khởi tạo một tập rỗng T
2. Chừng nào các đỉnh của G vẫn chưa liên thông bởi tập cạnh trong T:
3. Khởi tạo một tập rỗng E
4. Với mỗi thành phần:
5. Thêm cạnh nhỏ nhất nối S với một đỉnh không thuộc S vào E
6. Thêm tập hợp E vào T.
7. Tập hợp T là một cây bao trùm nhỏ nhất của G.

Ứng dụng

- ➢ Thiết kế mạng (Network Design): máy tính, viễn thông, giao thông, điện, nước...
- ➢ Thiết kế các thuật toán xấp xỉ cho bài toán NP-khó: ví dụ bài toán TSP.
- ➢ Bài toán phân cụm dữ liệu (Clustering).
- ➢ Dùng cho quy hoạch dân sự
- ➢ Dùng tìm đường đi trong bản đồ
- ➢ Theo dõi và xác minh khuôn mặt theo thời gian thực (tức là định vị khuôn mặt người trong luồng video).
- ➢ Đặt đường ống nối các bãi khoan ngoài khơi, nhà máy lọc dầu và thị trường tiêu thụ
- ➢ Xây dựng đường cao tốc hoặc đường sắt kéo dài qua một số thành phố với chi phí tối thiểu



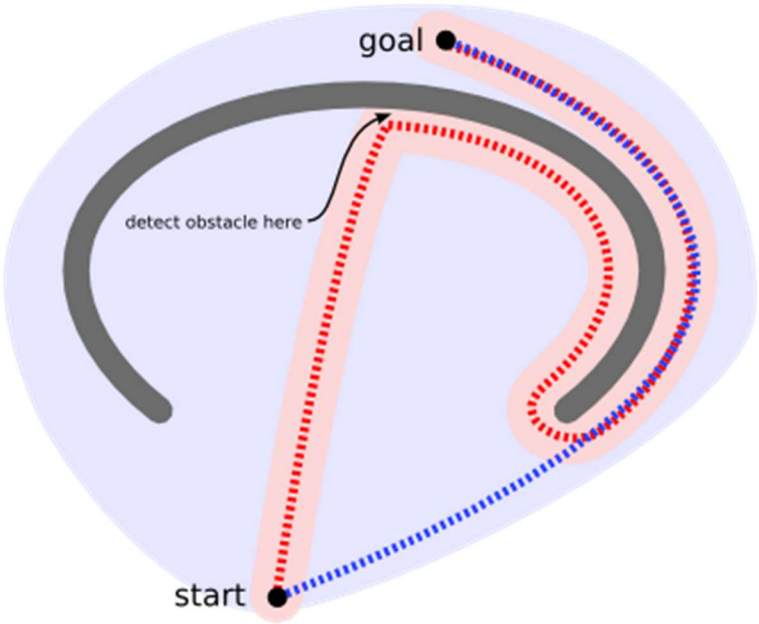
Hình 7. Lắp đặt dây mạng



Hình 8. Xây dựng các cây cầu đi qua tất cả các hòn đảo với chi phí tối thiểu (chi phí tỉ lệ thuận với chiều dài của cây cầu)

Bài toán tìm đường đi ngắn nhất

Giới thiệu bài toán



Ta bắt đầu với một điểm (start) và mục tiêu là tìm đường đi (ngắn nhất) từ đó đến đích (goal). Nếu trên đoạn đường từ điểm bắt đầu đến đích không có bất cứ chướng ngại vật nào thì đường đi ngắn nhất chính là 1 đường thẳng nối điểm start và goal.

Tuy nhiên, vấn đề là trên đường đi có những chướng ngại vật (obstacle) và ta không thể đi xuyên qua những chướng ngại vật đó. Thay vào đó, ta cần phải đi vòng hoặc tìm cách né tránh các chướng ngại vật.

Như với hình vẽ trên, chướng ngại vật khiến cho việc đi bằng đường thẳng từ start đến goal (đường màu đỏ) là không khả thi, ta sẽ phải đi vòng nếu lúc đầu ta quyết định đi theo đường thẳng. Nhưng đường ngắn nhất rõ ràng là đường màu xanh, khi mà ta đã tính toán ra được cách né tránh chướng ngại vật.

Một số thuật toán tìm đường đi ngắn nhất

Breadth First Search (BFS)

Mục tiêu của thuật toán BFS (tìm kiếm theo chiều rộng) là duyệt qua tất cả các đỉnh của đồ thị và tránh tình trạng bị vòng (đi qua 1 đỉnh nhiều lần). Nhờ đó ta sẽ tìm được đường đi từ 1 điểm đến 1 điểm bất kỳ trong đồ thị.

Thuật toán:

1. Khởi tạo danh sách các đỉnh đã duyệt (visited) - cũng là output của thuật toán.
2. Khởi tạo một hàng đợi (queue)
3. Chọn 1 đỉnh bất kỳ **j**, xác định các đỉnh kề.
4. Nếu đỉnh **j** chưa nằm trong visited thì thêm **j** vào visited, các đỉnh kề được cho vào hàng đợi.
5. Lấy ra đỉnh đầu tiên trong hàng đợi và lặp lại bước 3 (đỉnh đầu tiên lúc này đóng vai trò như **j**). Lặp cho đến khi không còn đỉnh nào trong hàng đợi.

Độ phức tạp của thuật toán

1. Độ phức tạp về thời gian của thuật toán BFS được biểu diễn dưới dạng O(V + E), trong đó V là số nút và E là số cạnh.
2. Độ phức tạp không gian của thuật toán là O(V).

Thuật toán Dijkstra

Thuật toán tìm kiếm theo chiều rộng (BFS) giúp tìm kiếm một cách đồng đều theo tất cả các hướng xuất phát từ một đỉnh của đồ thị. Tuy nhiên, trong trường hợp việc đi từ từng đỉnh đến đỉnh khác có chi phí/khoảng cách khác nhau thì thuật toán BFS lại không tối ưu đối với việc tìm đường đi ngắn nhất (chi phí nhỏ nhất) (vì BFS sẽ tìm kiếm đồng bộ theo tất cả các hướng).

Để cải thiện việc này, ta sử dụng thuật toán **Dijkstra** để ưu tiên tìm kiếm theo hướng có chi phí nhỏ nhất.

Về ý tưởng, ở thuật toán **Dijkstra** ta tìm quãng đường ngắn nhất (chi phí nhỏ nhất) tại mỗi điểm **j** trong đồ thị khi đi từ điểm xuất phát.

Thuật toán:

1. Quãng đường từ điểm bắt đầu đến 1 điểm bất kỳ **J** được ký hiệu là **d(J)**. Bắt đầu tại một điểm (start), dễ thấy ***d(start) = 0.*** Đối với các điểm khác trong đồ thị $d(j) = +\infty$
2. Xác định các đỉnh kề (K) (hiện đang có giá trị d(K)) của đỉnh hiện tại (gọi là C). Với mỗi đỉnh kề, cộng giá trị d(C) với khoảng cách từ C-K ta sẽ được d(K)', nếu nhỏ hơn giá trị d(K) đang có thì lấy giá trị nhỏ hơn (d(K)') và đặt d(K) = d(K)'.
3. Điểm C được đánh dấu là đã đi qua.
4. Nếu vẫn còn đỉnh chưa đi qua thì quay trở lại bước 3.

Độ phức tạp của thuật toán

3. Độ phức tạp về thời gian của thuật toán BFS được biểu diễn dưới dạng O(ElogV), trong đó V là số nút và E là số cạnh.
4. Độ phức tạp không gian của thuật toán là O(V).

Thuật toán A*

Thuật toán **Dijkstra** giúp cải thiện tốc độ tìm kiếm đường ngắn nhất giữa 2 điểm đối với đồ thị có trọng số so với thuật toán BFS, thông qua việc tính toán quãng đường ngắn nhất đến từng đỉnh và sau đó tiếp tục suy rộng, nhờ đó chỉ phải xét những quãng đường ngắn nhất một cách tối ưu hơn.

Tuy nhiên, thuật toán này vẫn có thể được cải thiện hơn nữa, nếu bên cạnh việc sử dụng quãng đường ngắn nhất thì hướng tìm kiếm dựa theo khoảng cách ước lượng đến đích trên đồ thị sẽ giúp tiết kiệm thời gian tìm kiếm.

Lời giải bài toán

A* thực tế là sử dụng thuật toán Dijkstra để tính khoảng cách từ điểm bắt đầu (S) đến 1 đỉnh bất kỳ (C) rồi sau đó lại ước lượng khoảng cách từ (C) đến điểm kết thúc (E). Kết hợp 2 thông tin này để tìm ra đường đi tốt nhất.

Thuật toán:

Gọi **g(p)** là khoảng cách từ điểm p đến điểm bắt đầu (S), **h(p)** là khoảng cách ước lượng từ điểm p đến điểm kết thúc (E). Ta có **f(p) = g(p) + h(p)** là khoảng cách từ điểm bắt đầu đến điểm kết thúc. Ta cần xác định một chu trình để f(p) có thể nhỏ nhất.

Các bước của thuật toán:

1. Open: tập các trạng thái đã được sinh ra nhưng chưa được xét đến.

2. Close: tập các trạng thái đã được xét đến.

3. Cost(p, q): là khoảng cách giữa 2 điểm p, q.

Với các định nghĩa trên:

B1. Open = {A} (Khởi tạo là đỉnh bắt đầu)

Close: = {}

B2. while (Open !={})

Chọn đỉnh tốt nhất p trong Open (xóa p khỏi Open).

Nếu p là trạng thái kết thúc thì thoát.

Chuyển p vào Close và tìm các đỉnh kế tiếp (q) sau p.

- Nếu q đã có trong Open
    - Nếu g(q) > g(p) + Cost(p, q)
        - g(q) = g(p) + Cost(p, q)
        - f(q) = g(q) + h(q)
        - prev(q) = p (đỉnh cha của q là p)

- Nếu q chưa có trong Open
    - g(q) = g(p) + cost(p, q)
    - f(q) = g(q) + h(q)
    - prev(q) = p
    - Thêm q vào Open
- Nếu q có trong Close
    - Nếu g(q) > g(p) + Cost(p, q)
        - Bỏ q khỏi Close
        - Thêm q vào Open

Như vậy tập Close là output của thuật toán

Ứng dụng

Mô tả ứng dụng

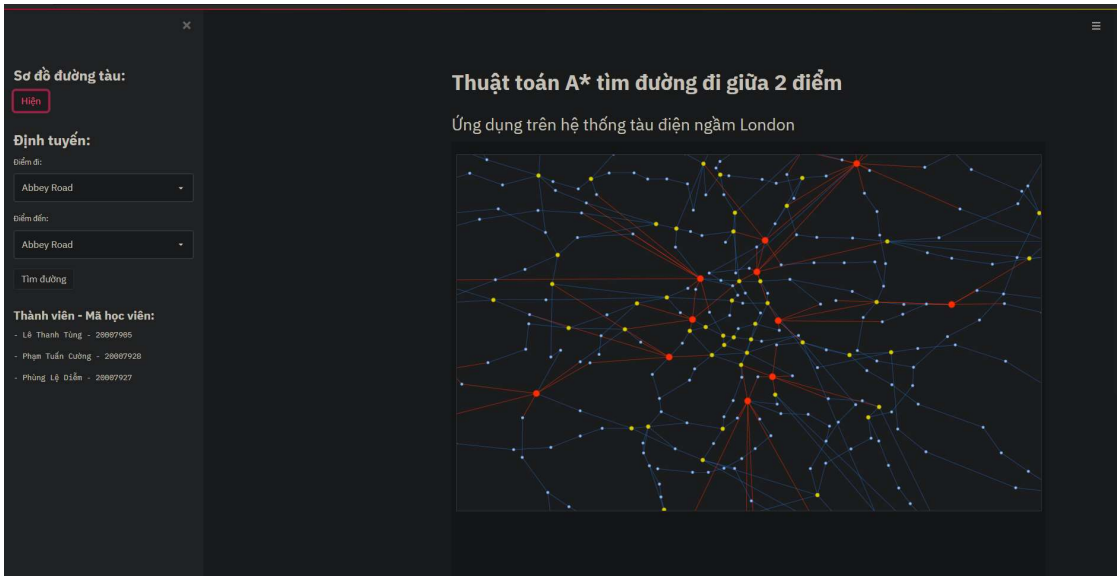
Tên ứng dụng:

Thuật toán A* tìm đường đi giữa 2 điểm

ứng dụng trên hệ thống tàu điện ngầm London

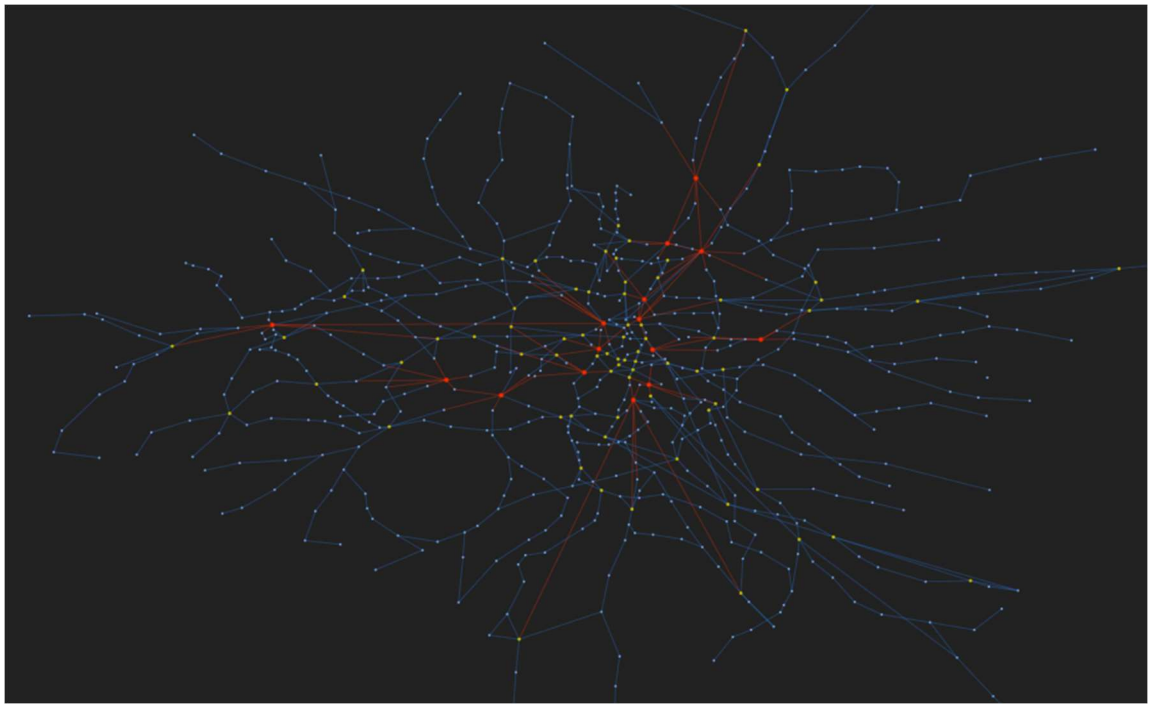
https://london-underground-route.herokuapp.com/

Mô tả giao diện:



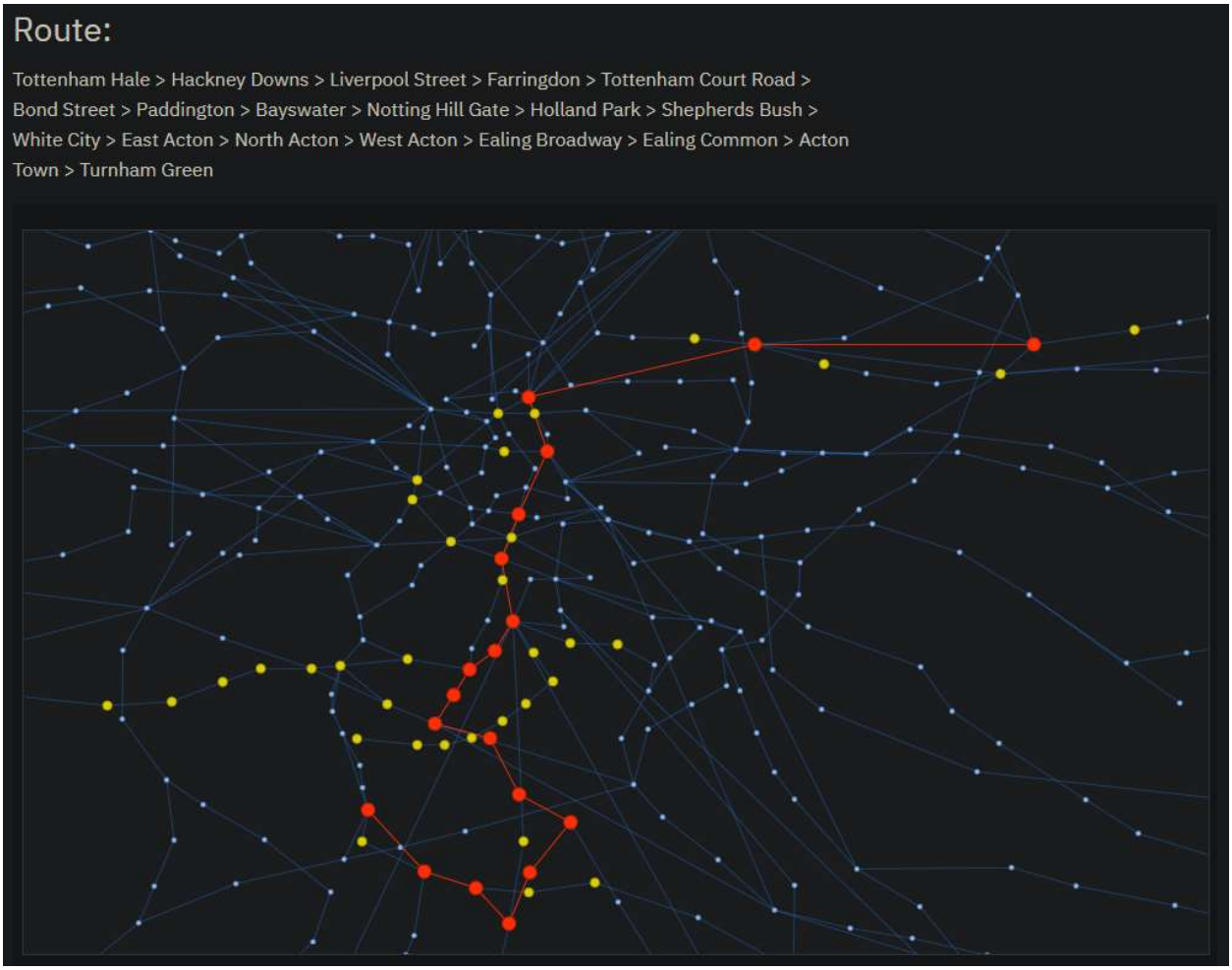
Mô tả tính năng:

- Hiển thị đồ thị các ga tàu điện ngầm và các ga có kết nối.



- Cho phép lựa chọn ga đi, ga đến và tìm đường đi ngắn nhất theo thuật toán A*.

- Hiển thị đồ thị mô tả đường đi ngắn nhất, trong đó hiển thị rõ đường đi ngắn nhất (màu đỏ) và các ga mà thuật toán A* đã kiểm tra.



Các thư viện, phần mềm liên quan:

- Python: ngôn ngữ lập trình được sử dụng để cài đặt thuật toán A* và các thư viện hỗ trợ liên quan.
- Pyvis: thư viện python, vẽ đồ thị và hiển thị với định dạng html
- StreamLib: thư viện python, xây dựng UI cho ứng dụng.
- Heroku: dịch vụ cung cấp máy chủ miễn phí

Dữ liệu sử dụng:

Đồ thị mạng lưới tàu điện ngầm London: https://gitlab.com/edent/force-directed-london-tube-map/-/blob/master/TfL%20Graph.json?fbclid=IwAR1MaukaI_misi_lh34ea5g6XDWH9rOn9D9Ga7WyrX0Tc58mytUngd0oE4w

Tọa độ các ga tàu điện:

https://www.doogal.co.uk/london_stations.php

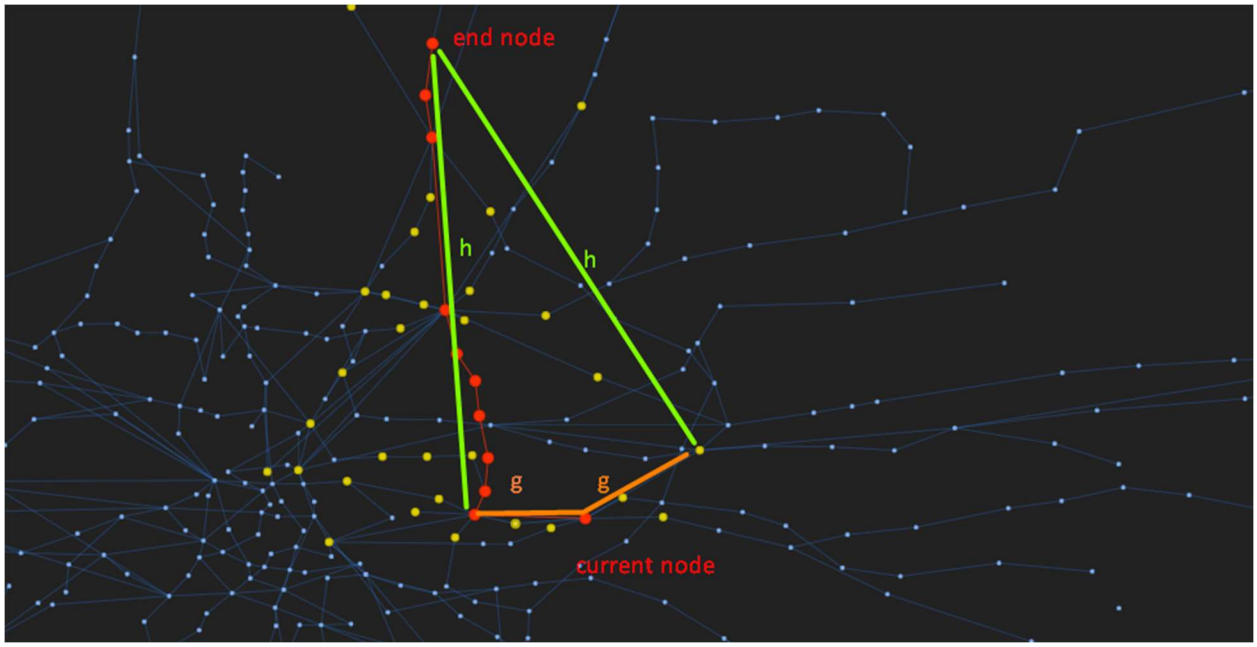
Lời giải

Tại đỉnh đang xét hiện tại:

Bước 1:

- Kiểm tra tất cả các đỉnh kề
- Tính toán tổng điểm g + h trên từng điểm, trong đó:
    - G là khoảng cách theo chuẩn 1 giữa đỉnh hiện tại và đỉnh kề
    - H là khoảng cách theo chuẩn 1 giữa đỉnh kề và đỉnh kết thúc
    - Khoảng cách chuẩn 1 được tính bằng công thức:

```
def norm1(start, end, coor):
    x1 = coor[start][0]
    x2 = coor[end][0]
    y1 = coor[start][1]
    y2 = coor[end][1]
    return abs(x1 - x2) + abs(y1 - y2)
```



- Tìm ra 2 đỉnh có tổng điểm nhỏ nhất

Bước 2:

- Kiểm tra tất cả các đỉnh kề với 2 đỉnh này
- Tiếp tục tính tổng điểm g + h cho tất cả các đỉnh này và tìm ra 2 đỉnh có điểm nhỏ nhất.
- Thuật toán đệ quy sẽ tiếp tục tìm 2 đỉnh mới gần nhất với đỉnh kết thúc.
- Thuật toán xác định tất cả các đỉnh liên quan sẽ kết thúc khi đỉnh gần nhất xét duyệt chính là đỉnh kết thúc

```
visited = set()
def AStarSearch(start_node, end_node, coor, link, max_search_node = max_search_node):
    cost = []
    candidate = []
    nonlocal visited
    for node in start_node:
        for sub_node in link[node]:
            visited.add(sub_node)
            cost.append([node, sub_node, norm1(node, sub_node, coor) + norm1(sub_node, end_node, coor)]) # g + h
    temp = sorted(cost, key = lambda x: x[2])
    temp = temp[:min(max_search_node,len(temp))]

    nearest_nodes = [x[1] for x in temp]
    if end_node not in nearest_nodes:
        candidate.extend(temp)
        candidate.extend(AStarSearch(nearest_nodes, end_node, coor,     link))
    else:
        candidate.extend(temp)
    return candidate
```

- Kết thúc thuật toán A* search, ta thu được chuỗi các điểm có tiềm năng.

Bước 3:

Xác định tuyến đường ngắn nhất bằng cách truy ngược kết quả thuật toán A* từ điểm kết thúc.

```
def find_path(start_node, end_node, candidate):
    path = []
    temp = [x for x in candidate if x[1] == end_node]
    for elem in temp:
        if elem[0] not in potential:
            potential.append(elem[0])
            if elem[0] != start_node[0]:
                path.append(elem)
                path.extend(find_path(start_node, elem[0],  candidate))
            else:
                path.append(elem)
    return path
candidate = AStarSearch(start_node, end_node, coor, link)
path = find_path(start_node, end_node,  candidate)[::-1]
path = list(list(zip(*path))[0]) +  [end_node]
return path, candidate, visited
```