ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO LAB 01
Graph Search

Môn: Cơ sở Trí tuệ nhân tạo - CSC14003

Lớp: 21_22

Tên: Lê Nguyễn

Mã số sinh viên: 21120511

Thành phố Hồ Chí Minh - 2023

Mục lục

Chương 1

Bài toán tìm kiếm

1.1 Mở đầu

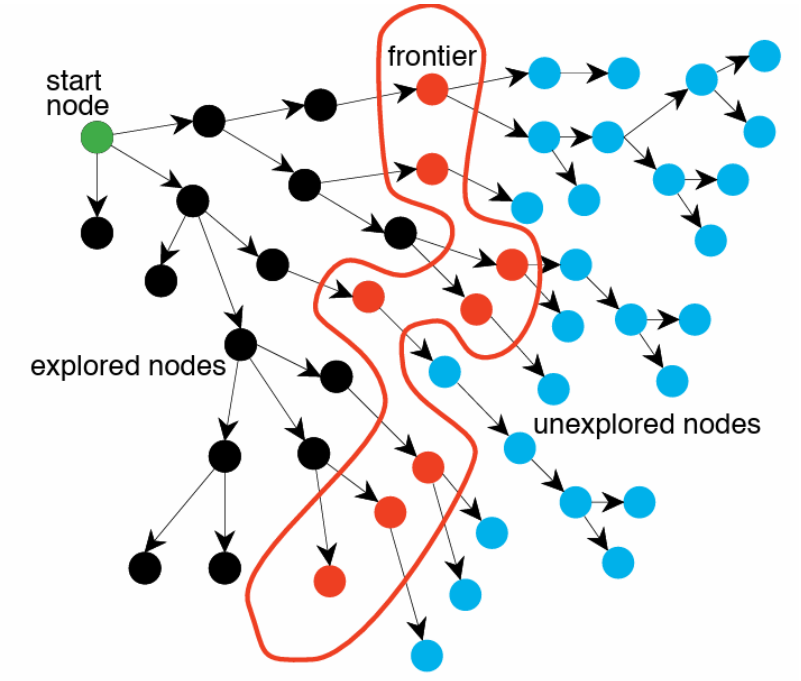
Thông thường, một vài vấn đề thực tế có điểm bắt đầu và điểm kết thúc, hay còn gọi là đáp án của vấn đề đó. Thế nhưng ta không có một thuật toán hay gì mà ta có thể thực hiện theo và tìm được đáp án, do đó ta cần **tìm kiếm** đáp án cho vấn đề đó [1]. Thế nhưng điều kiện tiên quyết để giải quyết bài toán tìm kiếm đó, ta cần biểu diễn được nó [2].

Ta có thể biểu diễn một **bài toán tìm kiếm** thông qua năm thành phần chính, để giải quyết tốt hơn, ta đưa bài toán tìm kiếm thành 1 đồ thị:

- **Không gian trạng thái**: Bao gồm tất cả các trạng thái có thể có của bài toán. Trạng thái trong bài toán tìm kiếm là thể hiện của một giai đoạn trong quá trình giải bài toán tìm kiếm, một trạng thái sẽ bao gồm những thông tin về môi trường xung quanh bài toán tìm kiếm [2]. Ta xem 1 trạng thái như là 1 nút của đồ thị, do đó cả không gian trạng thái là cả 1 đồ thị.
- **Trạng thái bắt đầu**: Là nơi bắt đầu của bài toán hay là nơi bắt đầu việc tìm kiếm của ta.
- **Trạng thái kết thúc**: Là nơi mà bài toán kết thúc, hay ta có thể nói là đích đến của ta khi thực hiện việc tìm kiếm, lúc này ta sẽ tìm được "đáp án" cho bài toán. Giống như thực tế, đôi khi đích đến không chỉ một mà ta còn có thể nhiều đích đến, do đó trạng thái kết thúc có thể là một trạng thái hoặc là một tập hợp các trạng thái. Ngoài ra trạng thái kết thúc có thể được biểu diễn bằng một điều kiện nào đó chứ không còn là một trạng thái xác định nữa [2].
- **Hàm chuyển trạng thái con**: Là một hàm nhận trạng thái hiện tại và cho ra tất cả các trạng thái mà trạng thái hiện tại có thể di chuyển đến được. Ta có thể xem hàm chuyển trạng thái con này như là một cạnh giữa hai nút trong đồ thị với mỗi nút tương ứng với một trạng thái.
- **Hàm chi phí**: Hàm này cho ta biết việc chuyển giữa một trạng thái này sang trạng thái khác sẽ tốn chi phí như thế nào. Ta có thể xem hàm chi phí như là một trọng số của cạnh của đồ thị.

Giờ ta đã có thể biểu diễn được một bài toán tìm kiếm rồi, tiếp theo ta chỉ cần *tìm kiếm* được đáp án. Thế nhưng ta sẽ tìm kiếm như thế nào ? Ta xét một đồ thị là không gian các trạng thái, bắt đầu tại một nút trên đồ thị, xem đó là trạng thái bắt đầu. Ta tạo hai danh sách gọi là *frontier* và *expanded*. Trong đó *frontier* sẽ chứa danh sách những nút mà ta đã tìm thấy và "chuẩn bị" đi đến, ta gọi những nút đó là *discovered node* (hay *nút đã khám phá*), còn *expanded* sẽ chứa những nút mà ta đã đi đến và tìm tất cả những nút tiếp theo mà ta có

thể đi đến được, ta gọi những nút đó là *expanded node*. Còn những nút còn lại, ta sẽ gọi là *unexplored node*. Nguyên lý là ta đi đến nút nào, gọi nút đó là *current node*, ta sẽ mở rộng nút đó, thêm *current* vào *expanded* và thêm các nút được mở rộng vào $frontier$ nếu nó chưa nằm trong $frontier$ và *expanded*, sau đó ta lấy các nút trong $frontier$ làm *current*, mở rộng tiếp tục cho đến khi *current node* của ta là trạng thái đích hoặc nếu $frontier$ đã hết nút, ta sẽ trả về không có đáp án.



Hình 1.1: Minh hoạ cho nguyên lý của cách giải một bài toán tìm kiếm. Nguồn [1].

Dưới đây là mã giả cho cách giải tổng quát một bài toán tìm kiếm thông qua đồ thị:

Thuật toán 1: Tìm kiếm đồ thị

```
1  start ← trạng thái bắt đầu;
2  frontier ← {start};
3  expanded ← ∅;
4  goal ← hàm xem nút hiện tại có là trạng thái đích hay chưa;
5  T ← hàm chuyển trạng thái con;
6  while frontier ≠ ∅ do
7      current ← nút được lấy ra từ frontier;
8      Thêm current vào expanded;
9      if goal(current) then
10         return đã tìm thấy đáp án;
11     else
12         for neighbor ∈ T(current) do
13             if neighbor ∉ frontier và neighbor ∉ expanded then
14                 Thêm neigbor vào frontier;
15 return không tìm thấy đáp án;
```

1.2 Các định nghĩa cơ bản

Định nghĩa 1. Hệ số phân nhánh tiến (*forward branching factor*) của một nút là số lượng cạnh đi ra từ nút đó.

Định nghĩa 2. Hệ số phân nhánh lùi (*backward branching factor*) của một nút là số lượng cạnh đi vào (đi đến) nút đó.

Định nghĩa 3. Một thuật toán tìm kiếm được nói là có **tính đầy đủ** (*completeness*) nếu tồn tại ít nhất một đáp án thì thuật toán tìm kiếm của ta đảm bảo sẽ tìm được đáp án đó. Để có tính đầy đủ, một thuật toán tìm kiếm phải **có tính hệ thống** (*systematic*), nghĩa là nó có thể đi đến được mọi trạng thái mà đi được từ trạng thái ban đầu.

Định nghĩa 4. Một thuật toán tìm kiếm được nói là có **tính tối ưu** (*optimality*) nếu nó có thể tìm được đáp án có chi phí thấp nhất trong tất cả các đáp án.

Định nghĩa 5. Độ phức tạp thời gian (*time complexity*) của một thuật toán tìm kiếm là thời gian của thuật toán chạy và được biểu diễn bằng độ dài đường đi dài nhất m và hệ số phân nhánh lớn nhất b.

Định nghĩa 6. Độ phức tạp không gian (*space complexity*) của một thuật toán tìm kiếm là bộ nhớ mà thuật toán dùng và được biểu diễn bằng m và b.

1.3 Tìm kiếm không thông tin và tìm kiếm có thông tin

Tìm kiếm không thông tin (*Uninformed Search*) là tìm kiếm mà ngoài việc biết được trạng thái đích là gì thì thuật toán không còn thông tin gì khác, ví dụ như trạng thái hiện tại có gần trạng thái đích hay không ? Ngoài ra tìm kiếm không thông tin còn được gọi là *tìm kiếm mù* do ta không biết thông tin gì khác về trạng thái đích ngoài việc trạng thái đích trong như thế nào.

Khác với bên trên **tìm kiếm có thông tin** (*Informed Search*) hay còn gọi là *tìm kiếm Heuristic*, sẽ có thêm thông tin để biết được mình đã gần trạng thái đích hay chưa, thông tin này ở dạng hàm, được gọi là **hàm heuristic**, kí hiệu là $h(n)$:

$$h(n) = \text{ước lượng chi phí của đường đi ngắn nhất từ } n \text{ đến trạng thái đích}$$

thông thường, các chi phí, ta gọi là $g(n)$, là tổng các trọng số từ trạng thái bắt đầu đến nút n. Thế nhưng nếu chi phí là bằng nhau, thì ta sẽ chọn đi đến nút kế tiếp như nào, lúc đấy $h(n)$ sẽ giúp ta chọn nút đi đến kế tiếp, $h(n)$ có thể được xem như là *knowledge* của thuật toán tìm kiếm.

Định nghĩa 7. Ta gọi một hàm heuristic $h(n)$ là **chấp nhận được** (*admissible*) nếu nó không *ước lượng quá cao* chi phí từ n đến trạng thái đích, nghĩa là:

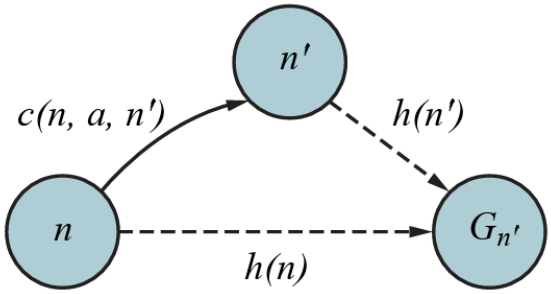
$$0 \leq h(n) \leq h^*(n)$$

với $h^*(n)$ là chi phí tối ưu thật sự từ n đến trạng thái đích.

Định nghĩa 8. Ta gọi một hàm heuristic $h(n)$ là **nhất quán** (*consistent*) nếu với mọi nút n và mọi nút con n' của nó, ta có:

$$h(n) \leq cost(n, n') + h(n')$$

trong đó $cost(n, n')$ là chi phí để đi từ n đến n'. Ngoài ra, mọi hàm heuristic $h(n)$ nếu nhất quán thì chấp nhận được, ngược lại thì không. Do đó nhất quán là một điều kiện chặn hơn của chấp nhận được.



Hình 1.2: Ta có thể thấy định nghĩa 8 giống như **bất đẳng thức tam giác**, hình 1.2 được lấy từ [3], trong đó $c(n, a, n')$ là chi phí đi từ n đến n' thông qua hành động a.

Ta có thể thấy điểm khác nhau đầu tiên giữa tìm kiếm có thông và không có thông tin là tìm kiếm có thông tin có thêm hàm heuristic $h(n)$ để định hướng cho nó, việc này giúp nó tìm được đích nhanh hơn tìm kiếm không thông tin do không đi những đường đi có chi phí cao, hoặc đi những đường đi có khả năng không dẫn tới đích, ngoài ra nó cũng không phải đi hết tất cả các trạng thái để tìm được trạng thái đích. Do đó, tìm kiếm có thông tin sẽ có thể tìm được các đường đi có chi phí thấp nhất, vì vậy khả năng tối ưu của nó cao hơn tìm kiếm không có thông tin.

Thế nhưng, tìm kiếm không thông tin sẽ dễ dàng cài đặt hơn do ta không cần quan tâm đến các chi phí hay việc chọn hàm heuristic của tìm kiếm có thông tin. Tiếp theo tìm kiếm không thông tin là có hệ thống khi nó luôn đi đến mọi trạng thái từ trạng thái ban đầu, nhờ vậy, các thuật toán tìm kiếm không tin dễ dàng có tính đầy đủ hơn.

Và điều quan trọng nhất, ta cần biết bài toán nào nên áp dụng phương pháp tìm kiếm nào, trong các bài toán có không gian trạng thái quá lớn, tìm kiếm không thông tin sẽ mất thời gian rất lâu do phải đi hết (hoặc gần hết) các trạng thái, trong các bài toán không quá phức tạp, tìm kiếm không thông tin lại tốt hơn do dễ dàng cài đặt.

Các thuật toán thuộc tìm kiếm không có thông tin có thể kể đến như Breadth-First-Search, Depth-First-Search, Uniform-Cost-Search và tìm kiếm có thông tin như Greedy Best-First-Search, A* Search.

Chương 2

Các thuật toán tìm kiếm

2.1 Breadth-first Search (BFS)

2.1.1 Ý tưởng chính

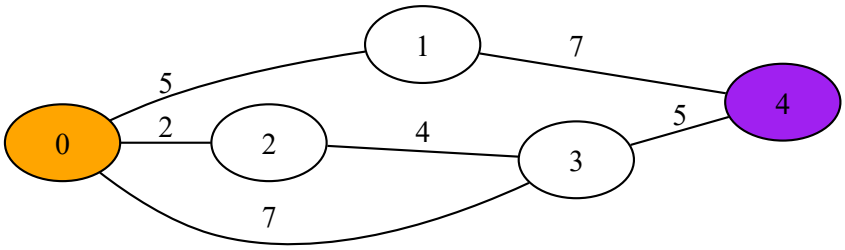
BFS hay *tìm kiếm theo chiều rộng* sẽ tìm kiếm bằng cách tạo một *cây tìm kiếm* thông qua không gian trạng thái (tức là biểu diễn bằng cây thay vì bằng đồ thị như ta nói phía trên). Thực hiện như nguyên lý tìm kiếm đồ thị ở hình 1.1 và thuật toán 1, nhưng lúc này ta xem *frontier* như một hàng đợi, những nút nào vào trước sẽ được lấy ra trước. Do đó tại nút ở độ sâu d của cây tìm kiếm, ta sẽ mở rộng tất cả các nút con của nó ở độ sâu $d+1$. Với cách này ta có thể đi đến mọi trạng thái của bài toán bằng cách mở rộng cây tìm kiếm, khiến cây tìm kiếm ngày càng "rộng" ra cho đến khi nào ta tìm được đích.

2.1.2 Mã giả

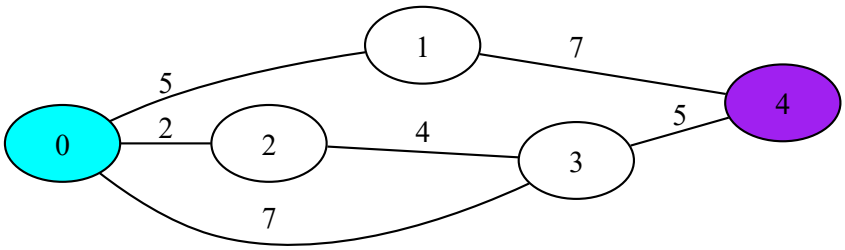
Thuật toán 2: Breadth-First-Search

```
1  start ← trạng thái bắt đầu;
   /* ta xem frontier như là một hàng đợi                    */
2  frontier ← {start};
3  expanded ← ∅;
4  goal ← hàm xem nút hiện tại có là trạng thái đích hay chưa;
5  T ← hàm chuyển trạng thái con;
6  while frontier ≠ ∅ do
7      current ← nút được lấy ra từ frontier;
8      Thêm current vào expanded;
9      if goal(current) then
10         return đã tìm thấy đáp án;
11     else
12         for neighbor ∈ T(current) do
13             if neighbor ∉ frontier và neighbor ∉ expanded then
14                 Thêm neigbor vào frontier;
15 return không tìm thấy đáp án;
```

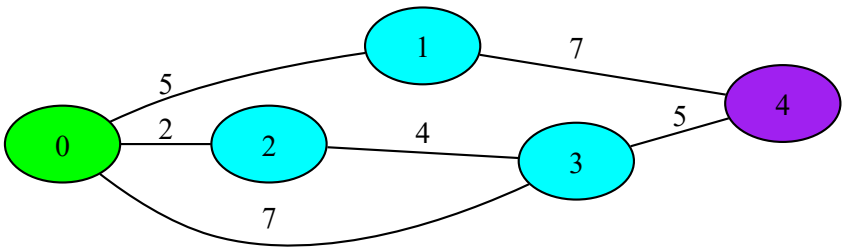
2.1.3 Ví dụ



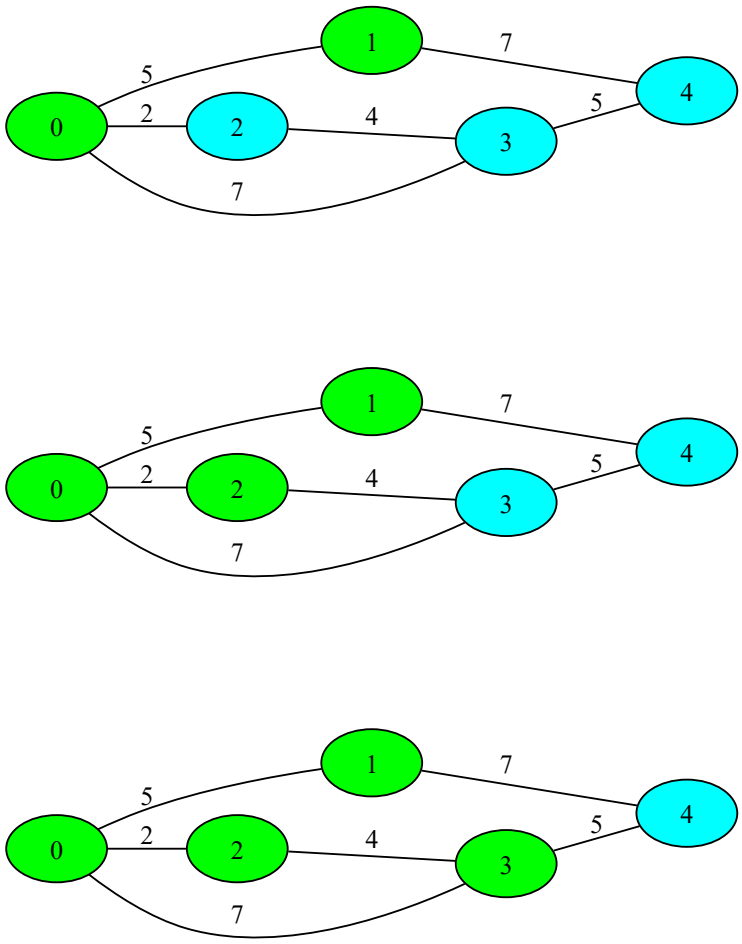
Hình 2.1: Ta sẽ dùng đồ thị như hình để thực hiện tìm kiếm, trong đó trạng thái bắt đầu có màu cam và trạng thái kết thúc có màu tím.



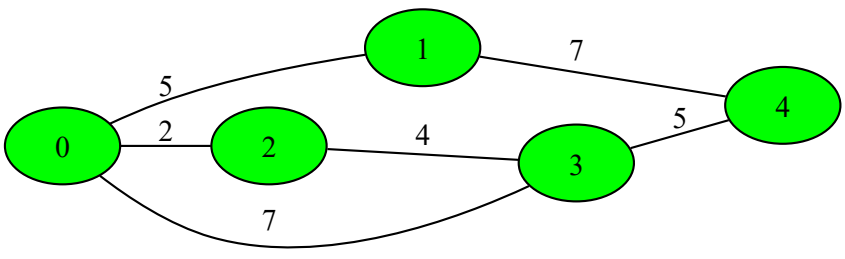
Hình 2.2: Đầu tiên, BFS xuất phát ở trạng thái bắt đầu và đưa nút đầu tiên vào *frontier* (ta tô màu xanh cho các nút trong *frontier*).



Hình 2.3: Tiếp theo BFS đưa nút hiện tại (là nút ban đầu) vào *expanded* (ta tô màu xanh lá cây) và mở rộng các nút con, đưa các nút đó vào *frontier*.



Hình 2.4: Do *frontier* là một hàng đợi nên tại BFS đã mở rộng hết tất cả các nút cùng chiều sâu (là 1, 2, 3).



Hình 2.5: Và cuối cùng BFS dừng khi nó đã đến (ta xem các nút ở *frontier* như các nút ta đã thấy nhưng chưa được mở rộng và ta sẽ kiểm tra trạng thái đích ở các nút đã được mở rộng) được trạng thái đích (là 4).

2.1.4 Đánh giá

- **Tính đầy đủ**: dễ thấy, do BFS sẽ đi đến tất cả các trạng thái mà có thể đi được từ trạng thái ban đầu, do đó BFS có tính hệ thống, vậy BFS có tính đầy đủ. Nhưng lưu ý rằng, chỉ có tính đầy đủ khi và chỉ khi hệ số phân nhánh tiến của một nút bất kì là hữu hạn, bởi vì nếu vô hạn thì ở độ sâu $d+1$ của nút có độ sâu d sẽ có vô hạn nút, do đó BFS sẽ mất vô hạn thời gian để mở rộng các nút đó.
- **Tính tối ưu**: Trong trường hợp đồ thị có chi phí là bằng nhau từ trạng thái bắt đầu đến một độ sâu d bất kì, khi đó BFS sẽ có tính tối ưu do nó tìm được đáp án có "chi phí" ít nhất, chi phí ở đây là số nút mà nó phải đi qua. Thế nhưng trong các đồ thị có chi phí ví dụ như trọng số, thì số nút ngắn nhất không có nghĩa là chi phí ít nhất, do đó không thoả mãn tính tối ưu.
- **Độ phức tạp thời gian**: Giả sử rằng ở một độ sâu bất kì, nút có hệ số phân nhánh tiến là b và giả sử đường đi dài nhất mà BFS tìm được có độ dài là d. Khi đó ở độ sâu đầu tiên, BFS phải mở rộng b nút, ở độ sâu thứ hai, phải mở rộng thêm b nút, tức tổng là $b \cdot b = b^2$ nút và ở độ sâu thứ d phải mở rộng b^d nút. Do đó độ phức tạp thời gian là $O(b^d)$.
- **Độ phức tạp không gian**: Các nút được sinh ra đều được lưu trên bộ nhớ, thế nên độ phức tạp không gian giống như độ phức tạp thời gian là $O(b^d)$ [2].

2.2 Uniform-Cost-Search (UCS)

2.2.1 Ý tưởng chính

UCS hay *tìm kiếm chi phí đồng nhất* như là một "bản" tổng quát của BFS khi nó trở nên hiệu quả hơn (đảm bảo tính tối ưu) khi thực hiện trên đồ thị có các chi phí như trọng số, nếu một đồ thị có các chi phí bằng nhau hay không có chi phí, lúc này UCS sẽ thành BFS. Thay vì mở rộng nút gần nhất như BFS, UCS sẽ chọn nút có *chi phí đường đi thấp nhất* để mở rộng, chi phí sẽ được tính theo công thức:

$$path\_cost(n) = path\_cost(n') + cost(n', n)$$

trong đó $path\_cost(n)$ là chi phí đường đi từ nút đầu tiên đến n, n' nút cha của n và $cost(n', n)$ là chi phí để từ n' đến n.

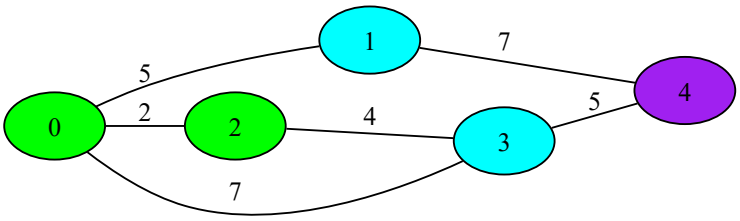
Trong UCS, *frontier* được xem là một *hàng đợi ưu tiên*, trong đó khi thêm một nút mới vào, ta sẽ sắp xếp lại toàn bộ *frontier*, đưa các nút có $path\_cost$ thấp lên đầu, ta xem $path\_cost$ như độ ưu tiên, $path\_cost$ càng thấp ưu tiên càng cao. Ngoài ra nếu một nút n đã có trong *frontier* nhưng có $path\_cost$ cao hơn $path\_cost$ hiện tại mà ta tính được (do đường đi khác đi, dẫn đến tổng chi phí khác) thì ta sẽ thay thế $path\_cost$ cũ thành $path\_cost$ mới.

2.2.2 Mã giả

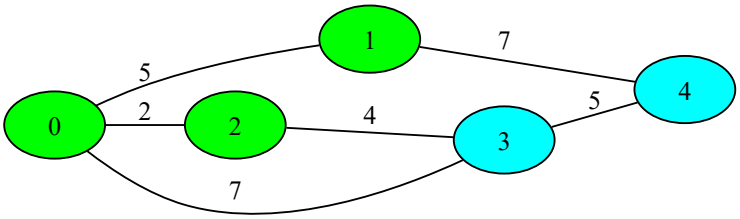
Thuật toán 3: Uniform-Cost-Search

```
start ← trạng thái bắt đầu;
/* ta xem frontier như là một hàng đợi ưu tiên                                  */
/* ta sẽ thêm nút n với path_cost của nó vào frontier                           */
/* sau đó frontier sẽ sắp xếp lại theo độ ưu tiên, path_cost bé nhất sẽ lên đầu */
/* do start là nút đầu tiên nên path_cost(start) = 0                            */
frontier ← {{0, start}};
expanded ← ∅;
goal ← hàm xem nút hiện tại có là trạng thái đích hay chưa;
T ← hàm chuyển trạng thái con;
/* w(n', n) sẽ trả về trọng số của cạnh n'n                                     */
w ← hàm lấy trọng số của một cạnh;
while frontier ≠ ∅ do
    cost, current ← path_cost và nút tương ứng được lấy ra từ frontier;
    Thêm current vào expanded;
    if goal(current) then
        return đã tìm thấy đáp án;
    else
        for neighbor ∈ T(current) do
            neighbor_cost ← cost + w(current, neighbor);
            if neighbor ∉ frontier và neighbor ∉ expanded then
                Thêm {neighbor_cost, neighbor} vào frontier;
            /* nếu neighbor có trong frontier nhưng path_cost cao hơn cost hiện tại, ta
               thay thế path_cost cũ thành cost hiện tại                        */
            else if neighbor ∈ frontier và path_cost(neighbor) > neighbor_cost then
                path_cost(neighbor) ← neighbor_cost
return không tìm thấy đáp án;
```

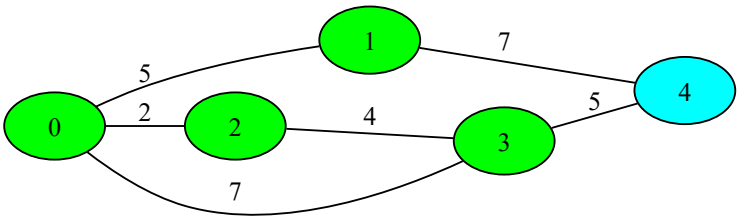
2.2.3 Ví dụ



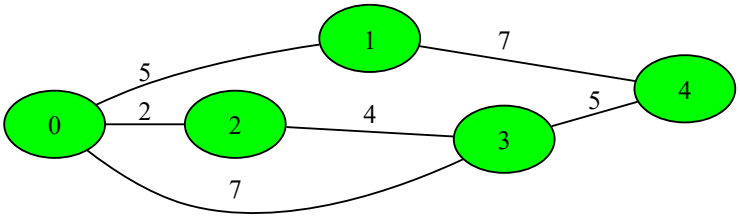
Hình 2.6: Ta xét cùng đồ thị hình 2.1. Ở các bước đầu, UCS giống như BFS ở hình 2.2, 2.3, thế nhưng khi chọn các nút để đi tiếp sau khi mở rộng nút 0, UCS sẽ chọn nút 2 do $path\_cost(2) = path\_cost(0) + w(0,2) = 2$ là nhỏ nhất trong đó $path\_cost(1) = 5$ và $path\_cost(3) = 7$.



Hình 2.7: Tiếp theo, ta thấy $path\_cost(3) = path\_cost(2) + w(2,3) = 2 + 4 = 6$ nhỏ hơn $path\_cost(3) = 7$ cũ nên ta thay thế $path\_cost(3)$ và chọn đi 1 do $path\_cost(1) = 5$ là nhỏ nhất.



Hình 2.8: Tiếp theo, ta thấy $path\_cost(4) = path\_cost(1) + w(1,4) = 5 + 7 = 12$ lớn hơn $path\_cost(3) = 6$, do đó ta sẽ đi tiếp đến 3.



Hình 2.9: Cuối cùng ta sẽ có $path\_cost(4) = path\_cost(3) + w(3,4) = 6 + 5 = 11$ bé hơn $path\_cost(4)$ cũ nên thay thế, đồng thời *frontier* cũng còn lại duy nhất 4, ta đi tới 4 và đến được đích.

2.2.4 Đánh giá

- **Tính đầy đủ**: UCS chỉ đầy đủ khi tại mọi bước trong quá trình tìm kiếm, chi phí của mỗi lần chuyển sẽ luôn lớn hơn một số dương ε nào đó. Giả sử lần chuyển đầu tiên có chi phí là 1/2, lần sau là 1/4, lần sau là 1/8, và cứ thế, do đó không tồn tại $\varepsilon > 0$ sao cho chi phí mỗi lần chuyển đều lớn hơn ε, khi đó chi phí đường đi sẽ là 1 cho dù phải đi qua vô hạn nút. Thế nhưng nếu chi phí đường đi tối ưu có giá trị lớn hơn 1 thì UCS sẽ không bao giờ tìm được đáp án.

- **Tính tối ưu**: Như ta đã biết, nếu đồ thị không có chi phí hoặc chi phí như nhau thì UCS là BFS, do đó có tính tối ưu. Nếu đồ thị có chi phí, thì UCS sẽ tìm ra đáp án có chi phí nhỏ nhất, do đó cũng có tính tối ưu.

- **Độ phức tạp thời gian**: Đặt C^* là chi phí đường đi tối ưu nhất, khi đó với mỗi lần chạy, chi phí đường đi sẽ gần với chi phí đương đi tối ưu thêm ít nhất một khoảng ε (ta dùng ε ở phía trên tính đầy đủ), ở khởi đầu chi phí là 0, lần tiếp theo là ε, tiếp theo nữa là 2ε, cho đến khi là $(C^*/\varepsilon)\varepsilon = C^*$, do đó độ sâu mà ta phải đi là $C^*/\varepsilon + 1$. Vậy như ở trên BFS, độ phức tạp thời gian là $O(b^{C^*/\varepsilon+1})$ với b là hệ số phân nhánh tiến tối đa.

- **Độ phức tạp không gian**: Ta biết mỗi nút đều được lưu trên bộ nhớ, do đó độ phức tạp không gian cũng giống với độ phức tạp thời gian là $O(b^{C^*/\varepsilon+1})$.

2.3 Depth-First-Search (DFS)

2.3.1 Ý tưởng chính

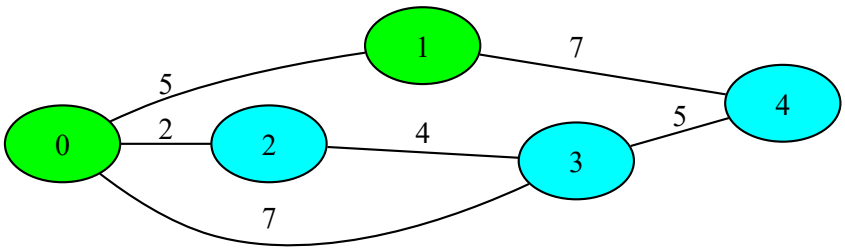
DFS hay *tìm kiếm theo chiều sâu* sẽ xem *frontier* như một ngăn xếp, những nút được thêm vào sau sẽ được lấy ra trước, do đó DFS luôn chọn những nút sâu nhất để mở rộng (thêm càng về sau, thì nút càng sâu), ngoài ra ở DFS ta có thể bỏ tập *expanded* đi nếu đồ thị dạng cây, lúc này bộ nhớ được sử dụng sẽ nhẹ đi. Ý tưởng của DFS là khi ta chọn 1 nút để đi, ta sẽ tiến hành đi sâu dần đường đi từ nút đó cho đến khi ta không thể đi được nữa, sau đó ta sẽ lấy nút sâu kế tiếp từ *frontier* (nút phía trước nút mà ta lấy ra, nút càng về sau càng sâu như ta đã nói phía trên) để đi và cuối cùng dừng nếu gặp được trạng thái đích.

2.3.2 Mã giả

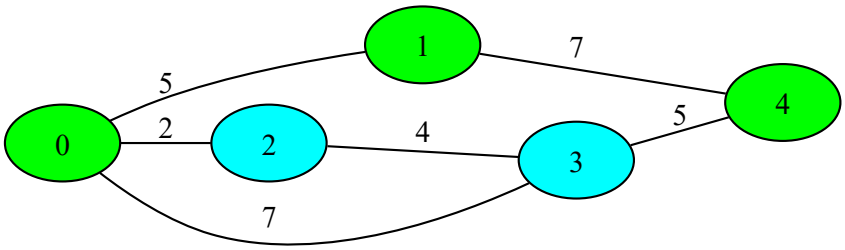
Thuật toán 4: Depth-First-Search

```
1  start ← trạng thái bắt đầu;
   /* ta xem frontier như là một ngăn xếp                    */
2  frontier ← {start};
3  expanded ← ∅;
4  goal ← hàm xem nút hiện tại có là trạng thái đích hay chưa;
5  T ← hàm chuyển trạng thái con;
6  while frontier ≠ ∅ do
7      current ← nút được lấy ra từ frontier;
8      Thêm current vào expanded;
9      if goal(current) then
10         return đã tìm thấy đáp án;
11     else
12         for neighbor ∈ T(current) do
13             if neighbor ∉ frontier và neighbor ∉ expanded then
14                 Thêm neigbor vào frontier;
15 return không tìm thấy đáp án;
```

2.3.3 Ví dụ



Hình 2.10: Ta dùng đồ thị hình 2.1 làm ví dụ. Ở giai đoạn đầu, DFS cũng giống như BFS ở hình 2.2 và 2.3. Thế nhưng sau khi đã mở rộng được các nút $1, 2, 3$ ta sẽ chọn nút tiếp theo bằng xem nút nào được thêm cuối cùng vào $frontier$, ta thêm lần lượt là $3, 2, 1$ (do ta ưu tiên những nút có số nhỏ hơn) và do 1 thêm vào cuối nên ta lấy nút 1 đầu tiên và đi tiếp.



Hình 2.11: Sau khi đã chọn mở rộng nút 1, ta tìm được nút 4 và đưa nút 4 và *frontier*, do 4 được đưa vào cuối cùng nên ta lấy 4 ra đầu tiên và đi tới 4, đồng thời ta cũng tìm được trạng thái đích.

2.3.4 Đánh giá

- **Tính đầy đủ**: Trên một đồ thị dạng cây hữu hạn thì DFS luôn đảm bảo tìm được đáp án, thế nhưng ở đồ thị vô hạn, nó sẽ đi liên tục trên một đường đi vô hạn. Trên các đồ thị có hướng không chu trình (*acyclic directed graph*) nếu ta không có tập *expanded*, DFS có thể mở rộng một nút nhiều lần thông qua các đường đi khác nhau, nếu may mắn nút có hệ số phân nhánh lùi thấp, ta vẫn có thể đảm bảo được việc tìm ra đáp án trong hữu hạn thời gian. Trong các đồ thị có chu trình (*cyclic graph*), nếu ta không có tập *expanded* thì DFS sẽ đi vòng lại, dẫn đến bị kẹt trong một vòng lặp vô hạn [3]. Do đó DFS không đảm bảo được tính đầy đủ.

- **Tính tối ưu**: DFS không đảm bảo được tính tối ưu, do đáp án mà nó trả về là đáp án đầu tiên mà nó tìm thấy, tức là DFS sẽ chọn một đường đi khả thi và đi cho đến khi tìm được đáp án rồi trả về đáp án. Đường đi mà DFS chọn có thể không tối ưu hoặc có thể là tồi nhất.

- **Độ phức tạp thời gian**: Xét trên một đồ thị hữu hạn và ta có tập *expanded*. Giả sử trạng thái đích ở độ sâu m và mỗi nút có hệ số phân nhánh tiến tối đa là b. Trong trường hợp xấu nhất DFS phải đi hết toàn bộ đồ thị, khi nó không chọn đúng đường đi và phải đi lại, do đó mất $O(b^m)$, riêng trường hợp tốt nhất, DFS chỉ cần đi đúng một đường đi dẫn đến kết quả và mất $O(m)$.

- **Độ phức tạp không gian**: DFS chỉ lưu những nút mà trên đường đi nó chọn, giả sử trạng thái đích nằm ở độ sâu m và hệ số phân nhánh tiến tối đa là b, khi đó DFS chỉ cần lưu $b \cdot m + 1$ nút. Nếu ta thêm tập *expanded* thì phải lưu thêm tối đa m nút, do đó sẽ lưu $b \cdot m + 1 + m = (b + 1) \cdot m + 1$ nút, vậy độ phức tạp không gian vẫn sẽ là $O(b \cdot m)$.

2.4 A* Search

2.4.1 Ý tưởng chính

Tìm kiếm A* là một trong những thuật toán tìm kiếm nổi bật của tìm kiếm có thông tin. Giống như UCS, A* cũng xem $frontier$ như một *hàng đợi ưu tiên* nhưng trong đó ta sẽ ưu tiên theo hàm $f(n)$ được tính như sau:

$$f(n) = path\_cost(n) + h(n)$$

với $path\_cost(n)$ là chi phí đường đi từ nút đầu tiên đến n và $h(n)$ là hàm heuristic tại nút n. Trong đó A* sẽ chọn những nút có $f(n)$ nhỏ nhất, ta biết $f(n)$ nhỏ nhất khi $g(n)$ và $h(n)$ nhỏ nhất, do đó việc này đảm bảo tìm được chi phí đường đi tối ưu đồng thời cũng gần trạng thái đích do càng gần trạng thái đích thì $h(n)$ càng nhỏ. Vì vậy việc quan trọng nhất là ta phải chọn được $h(n)$ sao cho A* là tốt nhất, nếu ta chọn $h(n)$ tồi thì đôi khi A* còn tệ hơn UCS.

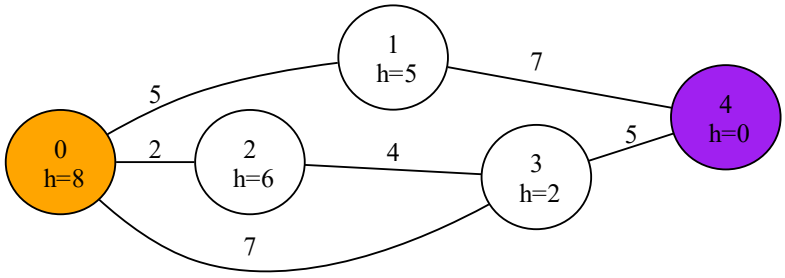
Ngoài ra, có những bài toán có "dạng" cụ thể để ta có thể chọn được hàm $h(n)$ tốt nhất, ví dụ bài toán tìm đường đi giữa hai thành phố thì ta có thể đặt $h(n)$ là khoảng cách nối giữa hai thành phố đó (độ dài đường thẳng nối hai thành phố), nếu bài toán chúng ta được thực hiện trên một mặt hai chiều chia thành từng ô vuông nhỏ (ví dụ pixel màn hình máy tính) thì ta có thể chọn $h(n)$ là *khoảng cách Manhattan* hoặc *khoảng cách Chebyshev*.

2.4.2 Mã giả

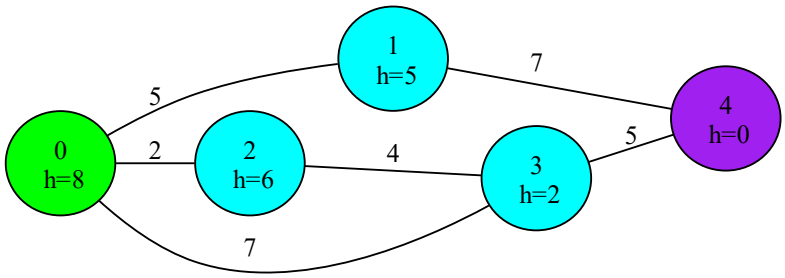
Thuật toán 5: A* Search

```
start ← trạng thái bắt đầu;
/* ta xem frontier như là một hàng đợi ưu tiên                                  */
/* ta sẽ thêm nút n cùng với f(n) vào frontier                                  */
/* ta sẽ ưu tiên những nút có f(n) nhỏ lên phía trước                           */
/* do start là nút đầu tiên nên f(start) = path_cost(start) + h(start) = 0 + h(start) = h(start)
   */
frontier ← {{h(start), start}};
expanded ← ∅;
goal ← hàm xem nút hiện tại có là trạng thái đích hay chưa;
T ← hàm chuyển trạng thái con;
/* path_cost(n) sẽ trả về chi phí đường đi từ trạng thái ban đầu đến nút n     */
path_cost ← hàm trả về chi phí đường đi tối ưu;
/* w(n', n) sẽ trả về trọng số của cạnh n'n                                     */
w ← hàm lấy trọng số của một cạnh;
while frontier ≠ ∅ do
    current ← nút được lấy ra từ frontier;
    Thêm current vào expanded;
    if goal(current) then
        return đã tìm thấy đáp án;
    else
        for neighbor ∈ T(current) do
            neighbor_cost ← path_cost(current) + w(current, neighbor);
            if neighbor ∉ frontier và neighbor ∉ expanded then
                /* Khác với UCS, ta sẽ ưu tiên lấy nút từ frontier dựa vào
                   f = path_cost + h                                            */
                f ← neighbor_cost + h(neighbor);
                Thêm {f, neighbor} vào frontier;
            else if neighbor ∈ frontier và neighbor_cost < path_cost(neighbor) then
                /* Cũng giống như UCS, nếu ta tìm được đường đi có path_cost tốt hơn, ta
                   sẽ thay thế path_cost cũ                                     */
                path_cost(neighbor) ← neighbor_cost;
return không tìm thấy đáp án;
```

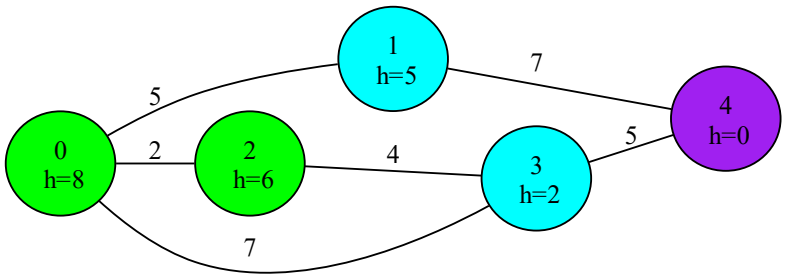
2.4.3 Ví dụ



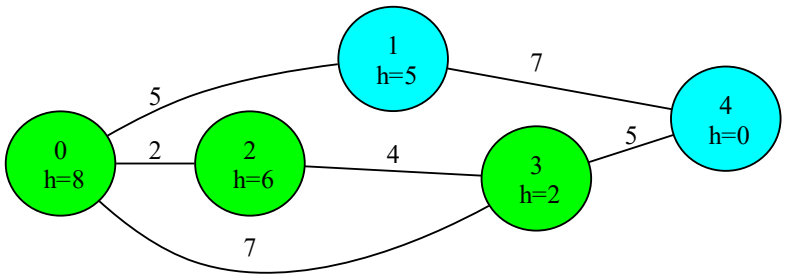
Hình 2.12: Ta sẽ dùng lại đồ thị ở hình 2.1 thế nhưng ta sẽ thêm *heuristic* cho từng nút (được kí hiệu là $h =$).



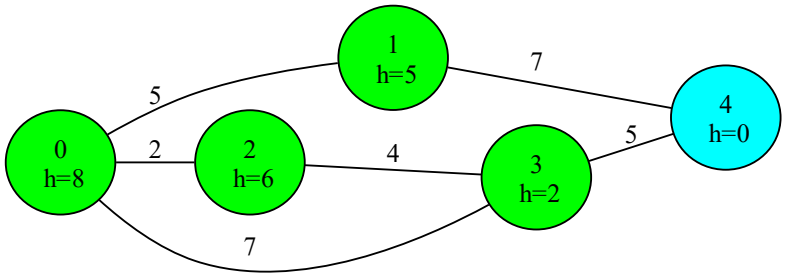
Hình 2.13: Ở giai đoạn đầu, A* sẽ chọn mở rộng nút 0 do chỉ có mỗi nút 0 trong $frontier$, sau mở rộng được nút $1, 2, 3$. Về cơ bản thì giai đoạn này cũng gần giống với các thuật toán trên.



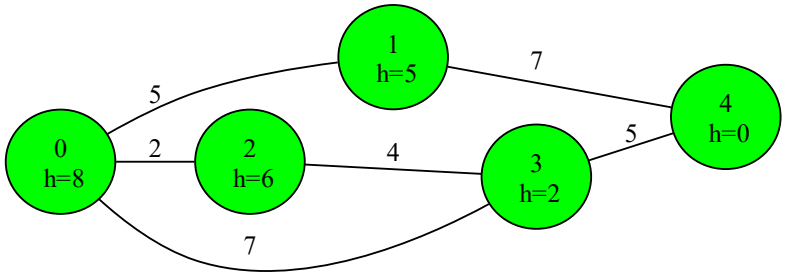
Hình 2.14: Ở giai đoạn tiếp theo, ta có $f(2) = path\_cost(2)+h(2) = 2+6 = 8$, $f(1) = 5+5 = 10$ và $f(3) = 7+2 = 9$. Do nút 2 có giá trị f nhỏ nhất nên ta sẽ ưu tiên đi đến 2 và mở rộng được 3.



Hình 2.15: Tiếp theo, $path\_cost(3)$ thông qua đường đi 2 sẽ là 6 so với 7 nếu đi qua 0, do đó ta thay thế $path\_cost(3) = 6$. Tiếp theo $f(3) = path\_cost(3) + h(3) = 6 + 2 = 8$ vẫn bé hơn $f(1) = 10$ nên ta sẽ chọn đi tiếp đến 3 và mở rộng được nút 4.



Hình 2.16: Tiếp theo, 3 đã mở rộng được nút 4, ta có $path\_cost(4) = path\_cost(3) + 5 = 11$ do đó $f(4) = 11 + h(4) = 11$ lớn hơn so với $f(1)$ nên ta sẽ chọn đi đến 1 thay vì 4.



Hình 2.17: Cuối cùng, đi đến 1 ta mở rộng được 4 và $path\_cost(4)$ mới sẽ là $path\_cost(1)+7 = 12$ do đó lớn hơn $path\_cost(4)$ cũ là 11, nên ta sẽ chọn đi tới 4 thông qua 3. Và như vậy ta đã đến được trạng thái đích.

2.4.4 Đánh giá

- **Tính đầy đủ**: Như UCS, nếu chi phí của mỗi lần chuyển luôn lớn hơn một số dương ε nào đó, khi đó A^* có tính đầy đủ.
- **Tính tối ưu**: A* có tính tối ưu hay không sẽ phụ thuộc vào hàm heuristic ta chọn có chấp nhận được hay không. Vì vậy ta chỉ cần chọn được hàm heuristic chấp nhận được, A* sẽ tối ưu.

  > **Chứng minh.** Ta dùng chứng minh phản chứng. Giả sử hàm heuristic chấp nhận được nhưng A* chưa tối ưu, khi đó chi phí đường đi mà đáp án A* trả về (gọi là p) sẽ có chi phí cao hơn đường đi có chi phí tối ưu thật sự (gọi là p'), đặt chi phí tối ưu thật sự là C^*. Đường đi p sẽ gồm $start \to goal$ và đường đi p' sẽ gồm $start \to s$ (bởi vì khi có được p, A* sẽ dừng, do đó p' sẽ dừng tại nút s nào đó mà chưa đi được đến $goal$). Vì vậy, tại bước cuối cùng, $frontier$ sẽ chọn $goal$ thay vì nút s, do đó:
  >
  > $$\begin{aligned} & f(goal) \leq f(s) \\ \Rightarrow\ & path\_cost(goal) + h(goal) \leq path\_cost(s) + h(s) \\ \Rightarrow\ & path\_cost(goal) \leq path\_cost(s) + h(s) \\ \Rightarrow\ & path\_cost(goal) \leq path\_cost(s) + h^*(s)\ \ (h \text{ chấp nhận được}) \\ \Rightarrow\ & path\_cost(goal) \leq C^* \end{aligned}$$
  >
  > Điều trên sai với giả sử ban đầu (nghĩa là $path\_cost(goal) > C^*$), thế nên A^* sẽ tối ưu khi mà hàm heuristic h chấp nhận được. □

  Thế nhưng ngược lại, A* vẫn có thể tối ưu nếu hàm heuristic không chấp nhận được. Ngoài ra nếu ta chọn được hàm heuristic nhất quán thì A* luôn có tính tối ưu.
- **Độ phức tạp thời gian**: Trong trường hợp xấu nhất, A* phải tìm kiếm toàn bộ không gian trạng thái, do đó nếu hệ số phân nhánh tiến tối đa là b, độ sâu tối đa là m thì độ phức tạp sẽ là $O(b^m)$.
- **Độ phức tạp không gian**: Cũng giống như các thuật toán trên, mỗi nút sinh ra đều được lưu trên bộ nhớ, do đó độ phức tạp không gian trong trường hợp xấu nhất cũng giống với độ phức tạp thời gian là $O(b^m)$.

2.5 Greedy Best-First-Search

2.5.1 Ý tưởng chính

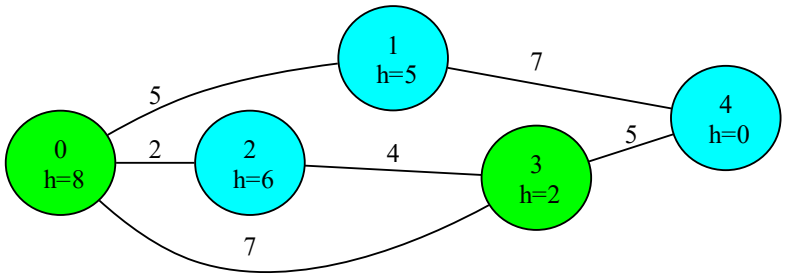
Khác đi một chút so với A* và UCS, Greedy cũng xem $frontier$ như hàng đợi ưu tiên nhưng sẽ ưu tiên những node có heuristic (thay vì chi phí đường đi như UCS hay hàm f như A*) tốt hơn (tức là thấp hơn). Do đó ta có thể nói Greedy là *tối ưu cục bộ* do tại một bước bất kì nào đó, Greedy luôn tối ưu, tìm được nút có giá trị heuristic tốt nhất.

2.5.2 Mã giả

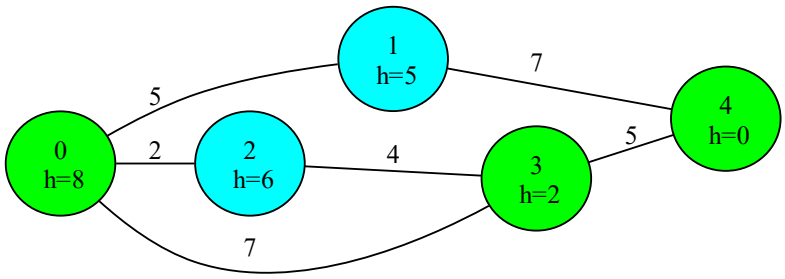
Thuật toán 6: Greedy Best-First-Search

```
1  start ← trạng thái bắt đầu;
   /* ta xem frontier như là một hàng đợi ưu tiên                    */
   /* những nút có heuristic càng tốt (thấp hơn) sẽ được ưu tiên lên đầu */
2  frontier ← {{h(start), start}};
3  expanded ← ∅;
4  goal ← hàm xem nút hiện tại có là trạng thái đích hay chưa;
5  T ← hàm chuyển trạng thái con;
6  while frontier ≠ ∅ do
7      current ← nút được lấy ra từ frontier;
8      Thêm current vào expanded;
9      if goal(current) then
10         return đã tìm thấy đáp án;
11     else
12         for neighbor ∈ T(current) do
13             if neighbor ∉ frontier và neighbor ∉ expanded then
14                 Thêm {h(neighbor), neigbor} vào frontier;
15 return không tìm thấy đáp án;
```

2.5.3 Ví dụ



Hình 2.18: Ta sẽ dùng lại hình 2.12 làm ví dụ. Ở các bước ban đầu, Greedy cũng tương tự như các thuật toán trên, để chọn giữa các nút được mở rộng $1, 2, 3$, Greedy sẽ chọn nút có heuristic thấp nhất, do đó nút 3 với $h = 2$ sẽ được chọn để đi tiếp.



Hình 2.19: Ở bước cuối cùng, sau khi chọn mở rộng 3, ta tìm được nút 4 với $h = 0$, đồng thời cũng là tốt nhất nên ta chọn đi đến 4, ngoài ra 4 cũng là trạng thái đích nên Greedy dừng.

2.5.4 Đánh giá

- **Tính đầy đủ**: Greedy luôn có tính đầy đủ trên một đồ thị hữu hạn. Giả sử nếu các nút có heuristic là như nhau, thì Greedy tương tự như BFS.
- **Tính tối ưu**: Ta thấy Greedy chỉ chọn những nút có heuristic tốt nhất nên sẽ phụ thuộc nhiều vào heuristic, Greedy không quan tâm đến các chi phí đường đi, do đó đường đi mà Greedy tìm ra sẽ có thể không có chi phí tối ưu. Thế nên Greedy không đảm bảo tính tối ưu.
- **Độ phức tạp thời gian**: Trong trường hợp xấu nhất, Greedy phải tìm kiếm hết không gian trạng thái, nếu mỗi nút có hệ số phân nhánh tiến là b và độ sâu tối đa của không gian trạng thái là m thì độ phức tạp thời gian là $O(b^m)$.
- **Độ phức tạp không gian**: Giống như DFS, khi hết đường Greedy phải quay lui về để tìm các đường đi khác. Thế nhưng, ở DFS khi ta chỉ quay lui về các nút con của những nút cha đã được mở rộng, thì Greedy lại quay lui về ngẫu nhiên về các nút khác trong đồ thị và mở rộng các nút đó, lúc mở rộng ta phải lưu lại các nút đó (cùng heuristic của nó) để dùng sau này, nên trường hợp xấu nhất là ta phải lưu hết tất cả các nút, do đó độ phức tạp là $O(b^m)$.

Chương 3

So sánh các thuật toán tìm kiếm

3.1 Giữa UCS, Greedy và A*

Ta cứ xem rằng, cả 3 thuật toán UCS, Greedy và A* đều đánh giá nút kế tiếp mà nó sẽ mở rộng dựa trên một hàm đánh giá $f(n)$ nào đó với n là nút nó chọn để đánh giá. Trong đó:

- UCS chọn cho mình hàm $f(n) = path\_cost(n)$ với $path\_cost(n)$ là chi phí đường đi từ trạng thái ban đầu cho đến nút n.
- Greedy chọn cho mình hàm $f(n) = h(n)$ với $h(n)$ là giá trị heuristic của nút n.
- A* chọn cho mình hàm $f(n) = path\_cost(n) + h(n)$, ta có thể thấy A* đã tổng hợp và "hoàn thiện" hơn so với hai thuật toán trên.

Như phân tích về tìm kiếm có thông tin và không có thông tin ở phía trên, UCS thuộc tìm kiếm không có thông tin (do không có hàm heuristic để xác định trạng thái đích) và Greedy, A* thuộc tìm kiếm có thông tin.

Đầu tiên, ta xem $path\_cost$ như là "tri thức của quá khứ" còn h như là "tri thức của tương lai". Lúc đó UCS sẽ có tri thức quá khứ để nó có thể tìm được đường đi nào có chi phí tối ưu nhất từ trạng thái ban đầu tới đích, thế nhưng nó thiếu đi tri thức tương lai, do đó nó không biết mình đã đến được gần trạng thái đích hay chưa, nên vẫn chưa quá tối ưu. Tiếp theo Greedy có tri thức tương lai, nhưng lại thiếu tri thức quá khứ, nên khi chọn những nút mà nó cho là có heuristic tốt nó sẽ bỏ qua hết các chi phí để đi đến nút nó, cho dù nó có tối ưu hay không. Lúc này ta xem Greedy như tối ưu cục bộ, những nút nó chọn sẽ tối ưu trong một thời điểm nhưng không tối ưu ở tất cả thời điểm. Còn A^* sử dụng cả tri thức tương lai và quá khứ thế nên ta có thể xem nó là tối ưu nhất.

Thứ hai, trong trường hợp đơn giản đồ thị hữu hạn, Greedy luôn có tính đầy đủ, còn A* với UCS ta cần đảm bảo điều kiện các chi phí chuyển từ trạng thái này sang trạng thái khác (hay trọng số) phải lớn hơn một số dương ε nào đó, thế nên Greedy đảm bảo tính đầy đủ trong nhiều trường hợp hơn. Tiếp theo, về tính tối ưu, UCS luôn đảm bảo được tính tối ưu trong mọi trường hợp (việc trọng số âm khá khó để xảy ra nên ta sẽ không quan tâm đến nó), còn A* lại phụ thuộc vào việc ta có chọn được hàm heuristic chấp nhận được hay không, còn Greedy như ta đã nói phía trên, nó không có tính tối ưu do nó chỉ quan tâm đến các nút có heuristic tốt mà không quan tâm đến chi phí thật sự để đi đến nút đó.

Cuối cùng, về độ phức tạp thời gian cả Greedy và A* đều là $O(b^m)$ với b là hệ số phân nhánh tiến tối đa của mỗi nút và m là độ sâu tối đa của không gian trạng thái, trong khi đó UCS có độ phức tạp thời gian là $O(b^{C^*/\varepsilon+1})$ với C^* là chi phí đường đi tối ưu và ε là số dương nào đó (ta có thể chọn ε để UCS thoả luôn tính đầy đủ) thế nên trong một vài trường hợp ε và C^* đủ tốt thì UCS có thể nhanh hơn A* và cả Greedy. Cả 3 đều có độ phức tạp không gian giống với thời gian nên cũng tuỳ vào ε với C^*.

3.2 Giữa UCS và Dijkstra

Sự khác biệt đầu tiên chính là mục đích của UCS và Dijkstra, trong khi UCS muốn tìm được đường đi có chi phí tối ưu (đường đi ngắn nhất) từ trạng thái bắt đầu (nút đầu tiên) cho đến trạng thái đích (nút cuối cùng) thì Dijkstra sẽ tìm đường đi ngắn nhất từ một đỉnh (cách gọi khác của nút) s mà ta chọn đến các đỉnh còn lại trong đồ thị (chứ không phải duy nhất một đỉnh như UCS).

Trước tiên, ta cần biết Dijkstra hoạt động như nào. Dijkstra dùng hai tập đỉnh là S và Q trong đó S là chứa tất cả các đỉnh mà đường đi ngắn nhất từ s tới nó được tìm thấy, ta có thể xem nó giống như *expanded* trong UCS và Q là một hàng đợi ưu tiên, cũng giống như $frontier$ của UCS, thế nhưng ta sẽ thêm tất cả các nút vào Q ở ngay khi bắt đầu thay vì thêm lần lượt trong quá trình chạy như $frontier$ của UCS, điều này cũng là bất lợi của Dijkstra khi ta phải biết trước tất cả các nút để thêm vào, do đó không dùng được với những *đồ thị không tường minh*[1]

Thứ hai, đó là vấn đề sử dụng bộ nhớ, Dijkstra phải thêm tất cả các node vào Q từ ban đầu, giả sử mỗi nút có hệ số phân nhánh tiến là b và độ sâu của đồ thị là m, nên nó phải lưu b^m trong mọi trường hợp, trong khi đó, chỉ trường hợp xấu nhất UCS mới lưu b^m. Thế nên trong đa số trường hợp, UCS sử dụng bộ nhớ thấp hơn Dijkstra.

Thứ ba, đó là về thời gian chạy, từ mục đích của cả 2 thuật toán, ta có thể đoán rằng Dijkstra sẽ chạy lâu hơn UCS trong đa số trường hợp do Dijkstra phải đi đến tất cả các đỉnh để tìm được đường đi ngắn nhất, trong khi đó UCS chỉ cần tìm được đường đi ngắn nhất từ nút bắt đầu cho đến nút đích. Giả sử cả Q và $frontier$ đều được dùng trên một cấu trúc dữ liệu giống nhau, trong đó có 2 hàm mà ta cần để ý là *insert()* dùng để thêm nút vào hàng đợi và *change_priority()* dùng để thay đổi giá trị ưu tiên của nút trong hàng đợi. Do Q phải thêm tất cả các nút vào từ đầu (dùng hàm *insert()*) với giá trị ưu tiên là ∞ (trừ nút s có giá trị ưu tiên là 0) thế nên khi thay đổi giá trị ưu tiên một nút trong một lần chạy, ta lại dùng thêm *change_priority()* thay vì chỉ dùng duy nhất hàm *insert()* trong cả quá trình chạy lẫn lúc bắt đầu như UCS.

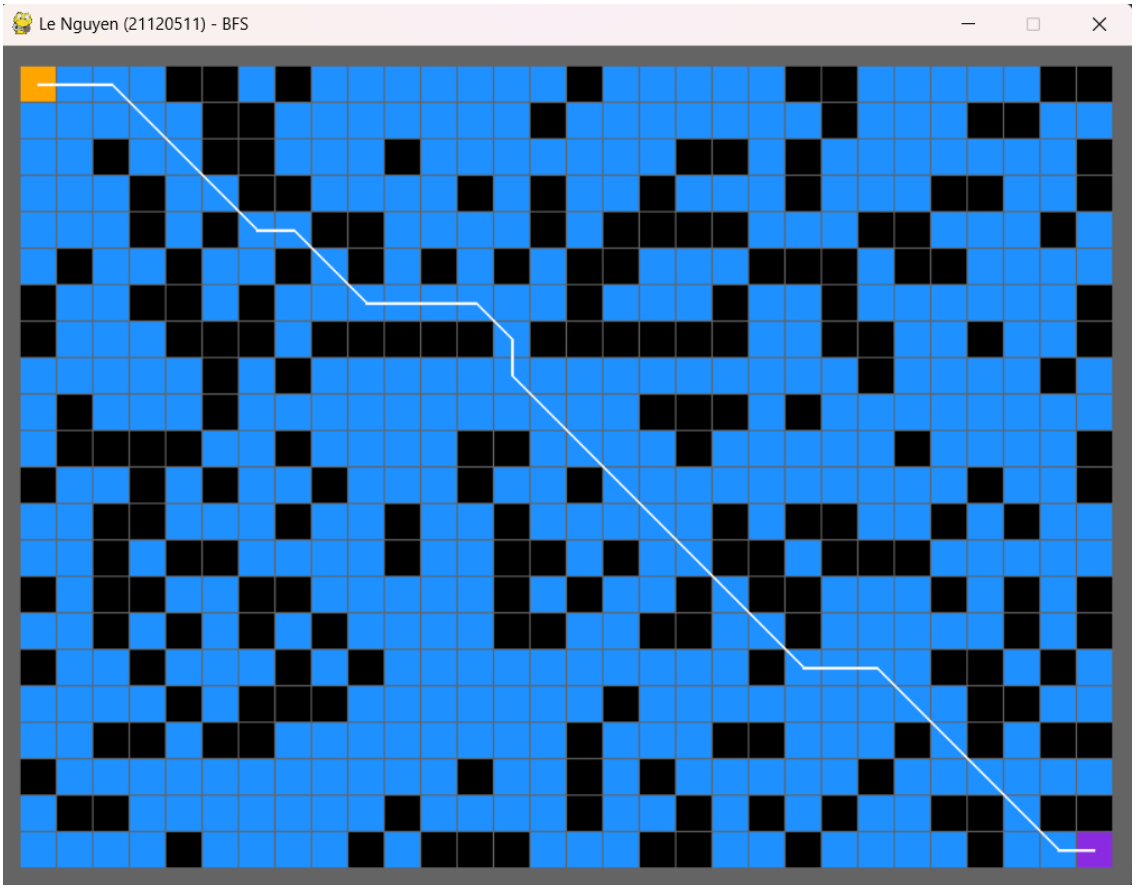
Cuối cùng, kết luận lại, UCS sẽ tốt hơn Dijkstra trong đa số trường hợp và trong thực tế, UCS được sử dụng nhiều hơn và hiệu quả hơn.

[1] implicit graph: là những đồ thị mà đỉnh của nó không thể lưu dễ dàng trong bộ nhớ của máy tính được.

Chương 4

Thực hiện các thuật toán tìm kiếm

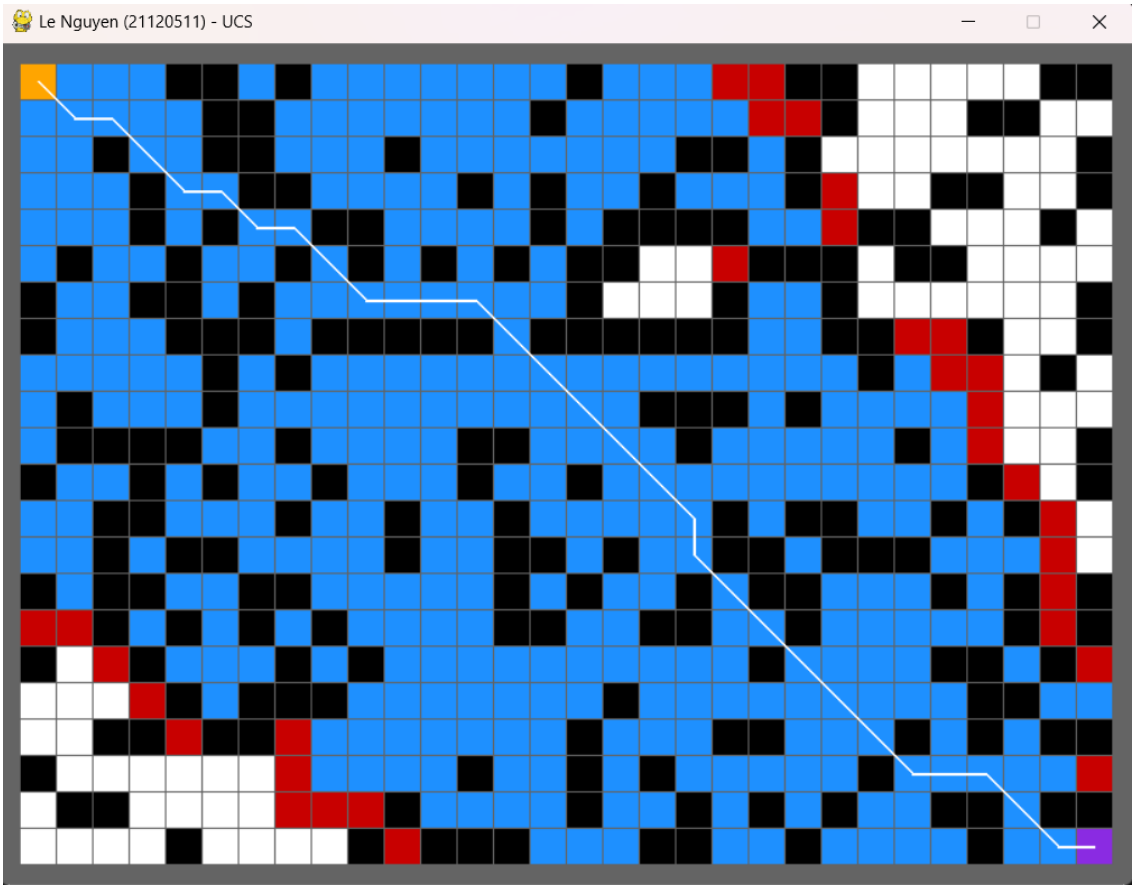
4.1 BFS



Hình 4.1: BFS được thực hiện trên một mê cung.

Ta có thể thấy BFS đã tìm kiếm hết toàn bộ các ô trong mê cung, (các ô nằm trong *expanded* sẽ được tô màu xanh). Trong quá trình tìm kiếm BFS sẽ cố gắng mở rộng theo chiều rộng các nút với thứ tự [up, down, left, right, left_up, left_down, right_up, right_down].

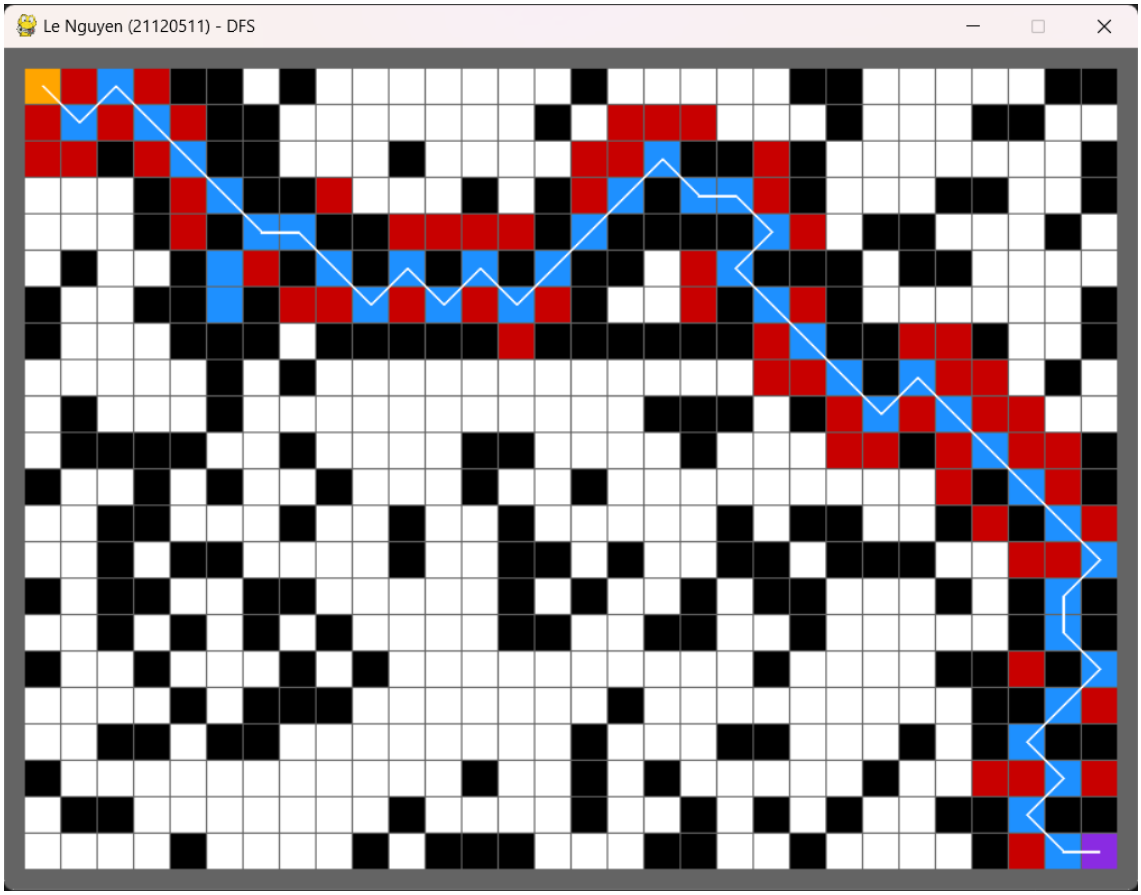
4.2 UCS



Hình 4.2: UCS được thực hiện trên một mê cung

Ở UCS, ta sẽ xét trọng số mỗi node theo chính thứ tự của nó như sau: [up = 6, down = 3, left = 7, right = 2, left_up = 8, left_down = 5, right_up = 4, right_down = 1]. Khi đó UCS của ta sẽ ưu tiên đi hướng xuống nghiêng về phía bên phải, vì vậy như hình phía trên các nút được mở rộng (màu xanh) có xu hướng tập trung về phía xuống nghiêng bên phải của mê cung.

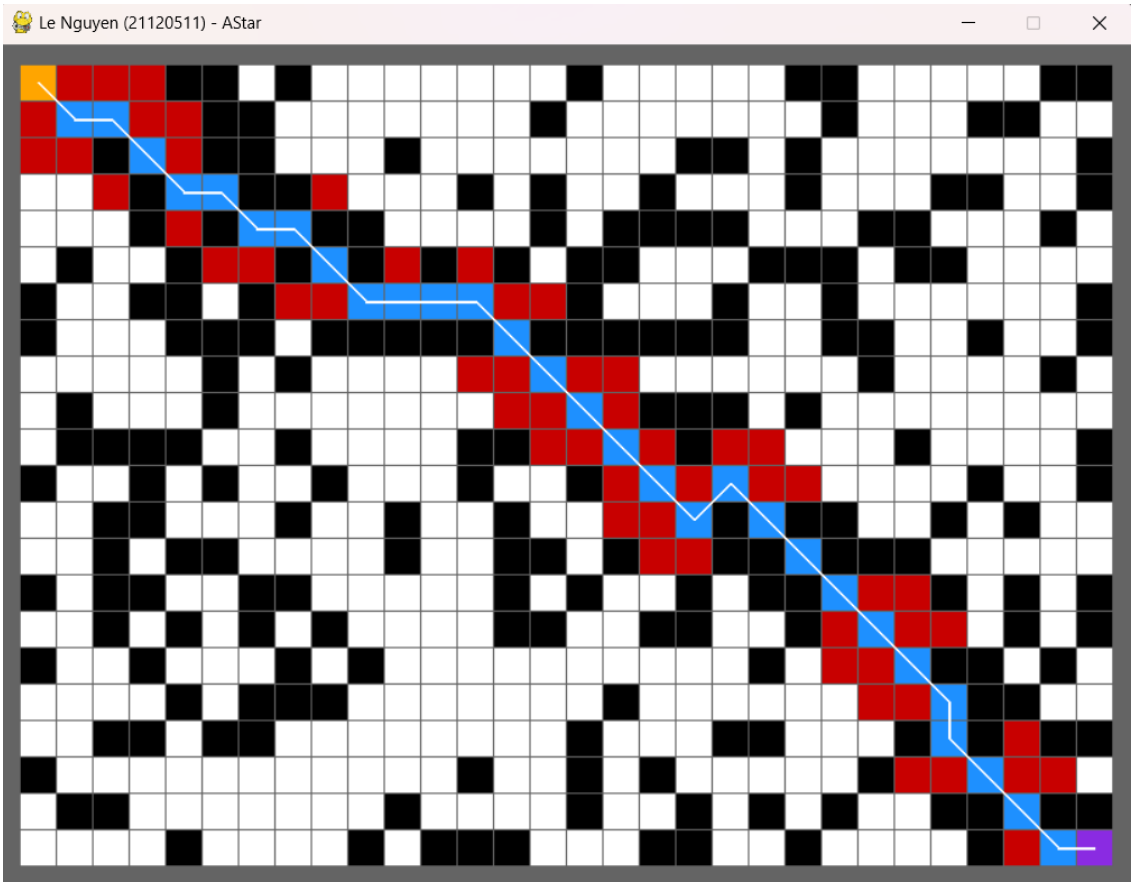
4.3 DFS



Hình 4.3: DFS được thực hiện trên một mê cung

So các nút được thêm vào theo thứ tự [up, down, left, right, left_up, left_down, right_up, right_down] nên theo cơ chế ngăn xếp của DFS, những nút right_down sẽ được lấy ra trước và ta có một kết quả như trên.

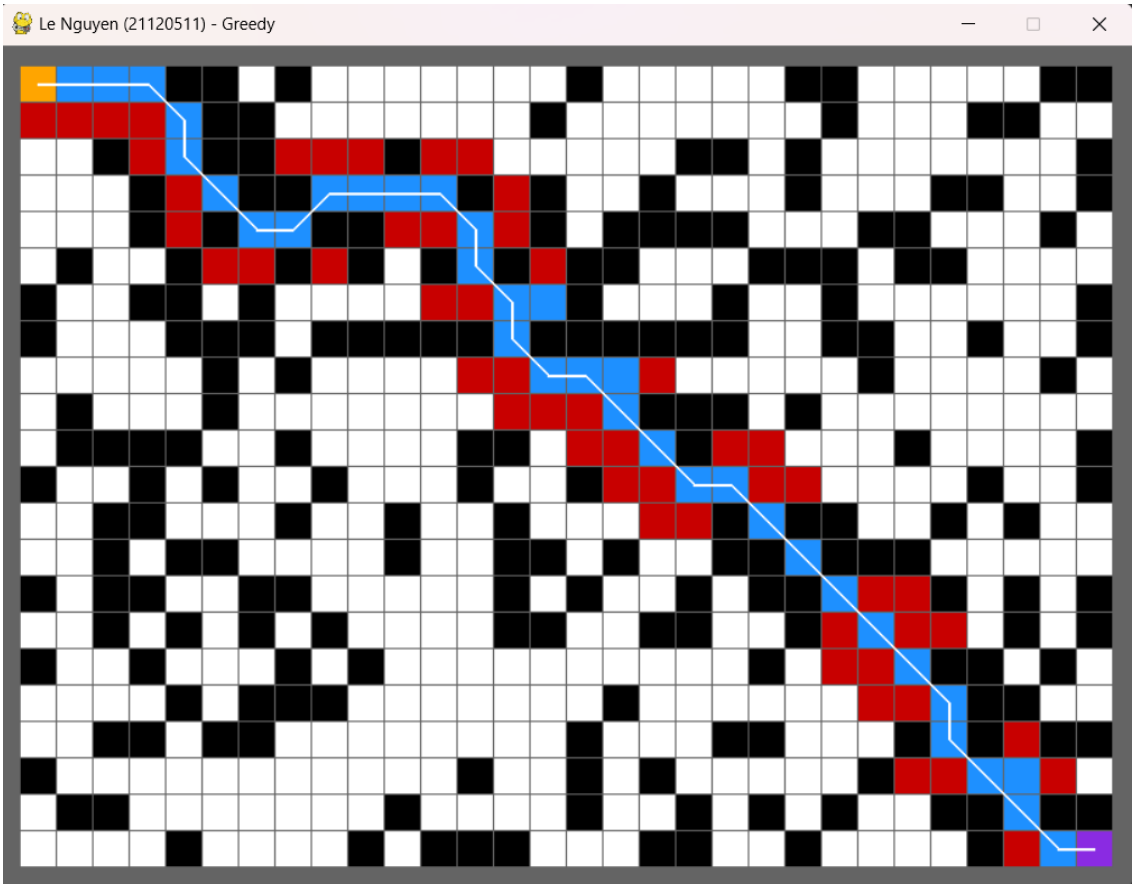
4.4 A*



Hình 4.4: A* được thực hiện trên một mê cung

Như ta đã nói phía trên, ta sẽ chọn hàm *heuristic* là *chebyshev distance* do bài toán được thực hiện trên một mặt phẳng hai chiều, chia thành các ô vuông. Ở *chebyshev distance* thuật toán của ta sẽ tối ưu hơn cho 8 hướng (thay vì 4 hướng như *manhattan distance*, *mahattan distance* không thể đi xéo được). Kết hợp thêm trọng số được đặt như UCS, A* của ta sẽ đi một mạch hướng xuống nghiêng bên phải để đi về đích, dựa trên hình, ta cũng có thể thấy A* nhanh và cũng tốn ít bộ nhớ nhất (số nút ô màu ít hơn).

4.5 Greedy



Hình 4.5: Greedy được thực hiện trên một mê cung

Ở Greedy ta cũng dùng hàm heuristic là *chebyshev distance* như A^* nên Greedy cũng có xu hướng đi xuống nghiêng về bên phải, ta thấy Greedy có đường đi dài hơn A* do nó chỉ quan tâm đến heuristic mà bỏ quên đi chi phí đường đi.

Tài liệu tham khảo

[1] URL: https://www.cs.ubc.ca/~kevinlb/teaching/cs322%20-%202009-10/Lectures/Search2.pdf.

[2] Tô Hoài Việt Lê Hoài Bắc. *Cơ sở Trí tuệ nhân tạo*. Nhà xuất bản khoa học và kỹ thuật, 2014.

[3] Stuart Jonathan Russell, Peter Norvig, and Ming-Wei Chang. Chapter 3: Solving problems by searching. In *Artificial Intelligence: A modern approach 4th Edition*, page 82–128. Pearson, 2022.

[4] URL: https://www.cs.ubc.ca/~kevinlb/teaching/cs322%20-%202008-9/Lectures/Search5.pdf.

[5] Ariel Felner. Position paper: Dijkstra's algorithm versus uniform cost search or a case against dijkstra's algorithm. 01 2011. URL: https://ojs.aaai.org/index.php/SOCS/article/view/18191.