ĐẠI HỌC QUỐC GIA TPHCM

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

BỘ MÔN CÔNG NGHỆ TRI THỨC

BÁO CÁO LAB 1

Đề tài: SEARCH IN GRAPH

Môn học: CƠ SỞ TRÍ TUỆ NHÂN TẠO

Sinh viên thực hiện:
Trần Đình Nhật Trí - 21120576

Giáo viên hướng dẫn:
Nguyễn Bảo Long

Ngày 22 tháng 10 năm 2023

Mục lục

1 Nghiên cứu và diễn giải thuật toán

1.1 Bài toán tìm kiếm

Trong toán học về lý thuyết độ phức tạp tính toán, lý thuyết tính toán và lý thuyết quyết định, bài toán tìm kiếm là một loại bài toán tính toán được biểu diễn bằng một mối quan hệ nhị phân. Cụ thể, gọi V là một thuật toán nhận hai đầu vào và luôn kết thúc. Khi đó dưới đây là một bài toán tìm kiếm:

Đầu vào: x
Mục tiêu: Tìm y sao cho $V(x, y) = 1$, hoặc báo y không tồn tại.

Tất cả bài toán tìm kiếm có thể được viết dưới dạng này. Cụ thể, ta có thể lấy dạng trên như là một định nghĩa của bài toán tìm kiếm: một bài toán được xem là bài toán tìm kiếm nếu có tồn tại một thuật toán luôn kết thúc V thuộc dạng trên.

1.1.1 Các yếu tố của một bài toán tìm kiếm

Một bài toán tìm kiếm thường có các yếu tố đặc trưng sau:

- Trạng thái về không gian (State Space)
- Trạng thái bắt đầu (Start Space)
- Trạng thái đích (Goal State)

1.1.2 Mã giả cho bài toán tìm kiếm chung

Algorithm 1 Graph Search

```
1:  procedure GraphSearch(start)
2:      initialize the open set using the initial state of problem
3:      initialize the closed set to be empty
4:      while True do
5:          if the frontier is empty then
6:              return failure
7:          end if
8:          choose a leaf node and remove it from the open set
9:          if the node contains a goal state then
10:             return the corresponding solution
11:         end if
12:         add the node to the closed set
13:         if node not in the open set or closed set then
14:             expand the chosen node, adding the resulting nodes to the open set
15:         end if
16:     end while
17: end procedure
```

1.2 Các loại bài toán tìm kiếm

Có rất nhiều thuật toán tìm kiếm vô cùng mạnh mẽ phù hợp cho những trường hợp cụ thể, Tuy nhiên, ta sẽ chỉ thảo luận sáu thuật toán tìm kiếm cơ bản, được chia thành hai loại dưới đây:

1.2.1 Thuật toán không hiểu biết (Uninformed Search)

Còn được gọi là thuật toán tìm kiếm mù, hoạt động theo cách brute force. Những thuật toán này không có bất kì thông tin gì về không gian hay trạng thái tìm kiếm ngoài cách duyệt cây. Những thuật toán điển hình cho loại này:

- Depth First Search (DFS)
- Breath First Search (BFS)
- Uniform Cost Search (UCS)

1.2.2 Thuật toán hiểu biết (Informed Search)

Thuật toán loại này được sinh ra giúp giải quyết vấn đề duyệt trâu của những thuật toán uninformed search. Thuật toán này được cung cấp những thông tin hữu ích cho việc tìm kiếm như chi phí đường đi, cách tiếp cận mục tiêu, khoảng cách mục tiêu còn bao xa,... những kiến thức này giúp cho thuật toán hạn chế tìm kiếm các không gian không cần thiết và thấy các đường đi hiệu quả hơn. Những thuật toán ta thường thấy thuộc loại này đó là:

- Greedy Search
- A* Search
- Graph Search

1.3 Nghiên cứu thuật toán

Trong phần này ta sẽ đi sâu vào bốn thuật toán tìm kiếm rất nổi tiếng: `DFS`, `BFS`, `UCS` và `AStar`.

1.3.1 Depth First Search (DFS)

DFS là một thuật toán để duyệt qua hoặc tìm kiếm cấu trúc dữ liệu dạng cây hoặc đồ thị. Thuật toán bắt đầu tại nút gốc (chọn một số nút tùy ý làm nút gốc trong trường hợp đồ thị) và kiểm tra từng nhánh càng xa càng tốt trước khi quay lui.

Kết quả của một *DFS* là một cây bao trùm (spanning tree). Cây khung (spanning tree) là một đồ thị không có vòng lặp.

Ý tưởng

Ta có ý tưởng cơ bản như sau:

- Từ nút bắt đầu, ta lấy một nút bất kì kề với nút đó.
- Tiếp tục lấy một nút kề của nút vừa xét rồi duyệt tiếp đến khi không thể đi tiếp được nữa thì ta sẽ quay lui và xét một nút kề khác.
- Cứ lặp lại như vậy cho đến khi quay lại nút bắt đầu và nút bắt đầu cũng không còn nút kề để duyệt nữa thì thuật toán dừng.

Mã giả

Để thực hiện duyệt theo DFS, chúng ta cần sử dụng cấu trúc dữ liệu ngăn xếp có kích thước tối đa bằng tổng số nút trong biểu đồ:

Algorithm 2 Depth-First Search

```
1:  procedure DFS(start)
2:      s ← stack
3:      mark start as visited and push start into s
4:      while s is not empty do
5:          pop the top element u of s
6:          if u is not visited then
7:              mark u as visited
8:              for each neighbor of u do
9:                  push neighbor into s
10:             end for
11:         end if
12:     end while
13: end procedure
```

Phân tích thuật toán

- Tính hoàn tất (Completeness): DFS hoàn tất nếu cây tìm kiếm là hữu hạn, nghĩa là đối với một cây tìm kiếm hữu hạn nhất định, DFS sẽ đưa ra giải pháp nếu nó tồn tại.
- Tính tối ưu (Optimality): DFS không tối ưu, có nghĩa là số bước để đạt được giải pháp hoặc chi phí bỏ ra để đạt được giải pháp đó cao.
- Tính phức tạp (Complexity):
  1. Độ phức tạp thời gian (Time complexity): Độ phức tạp về thời gian của DFS sẽ tương đương với nút được thuật toán duyệt qua. Nó được cho bởi:

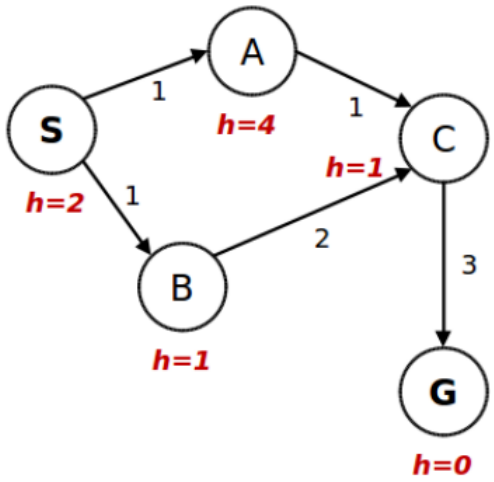
     $$T(n) = 1 + n^2 + n^3 + ... + n^m = O(n^m)$$

     Trong đó, m là độ sâu tối đa của bất kỳ nút nào và giá trị này có thể lớn hơn nhiều so với d (Nông nhất độ sâu giải pháp).

2. Độ phức tạp không gian (Space complexity): Thuật toán DFS chỉ cần lưu trữ một đường dẫn duy nhất từ nút gốc, do đó độ phức tạp về không gian của DFS tương đương với kích thước của tập rìa, là $O(b^m)$.

Minh họa mã giả

Ta sẽ sử dụng đồ thị dưới đây để minh họa cách hoạt động của cả 4 thuật toán `DFS, BFS, UCS` và `A*`.



Hình 1: Đồ thị biểu diễn các khả năng đi từ S đến G

Dựa vào mã giả trên, ta duyệt đồ thị như sau:

B1: Visited: []
Stack: [S]

B2: Visited: [S]
Stack: [A, B]

B3: Visited: [S, A]
Stack: [C, B]

B4: Visited: [S, A, C]
Stack: [G, B]

B5: Visited: [S, A, C, G] -> Tới đích: dừng
Stack: [B]

Kết quả cuối cùng là: $S->A->C->G$

1.3.2 Breadth First Search (BFS)

BFS là thuật toán tìm kiếm phổ biến nhất để duyệt qua một cây hoặc biểu đồ bằng cách duyệt theo chiều rộng (những nút nào gần nút xuất phát hơn sẽ được duyệt trước).

Ứng dụng của *BFS* có thể giúp ta giải quyết tốt một số bài toán trong thời gian và không gian tối thiểu. Đặc biệt là bài toán tìm kiếm đường đi ngắn nhất từ một nút gốc tới tất cả các nút khác. Trong đồ thị không có trọng số hoặc tất cả trọng số bằng nhau, thuật toán sẽ luôn trả ra đường đi ngắn nhất có thể. Ngoài ra, thuật toán này còn được dùng để tìm các thành phần liên thông của đồ thị, hoặc kiểm tra đồ thị hai phía, ...

Ý tưởng

Ta có ý tưởng cơ bản như sau:

- Từ một nút, ta tìm các nút kề rồi duyệt qua các nút này.
- Tiếp tục tìm các nút kề của nút vừa xét rồi duyệt tiếp đến khi đi qua hết tất cả các nút có thể đi.

Mã giả

Thuật toán sử dụng một cấu trúc dữ liệu hàng đợi (queue) để chứa các nút sẽ được duyệt theo thứ tự ưu tiên chiều rộng:

Algorithm 3 Breadth-First Search

```
1: procedure BFS(start)
2:     q ← queue
3:     mark start as visited and enqueue start to q
4:     while q is not empty do
5:         remove head u of q
6:         mark and enqueue all unvisited neighbors of u
7:     end while
8: end procedure
```

Phân tích thuật toán

- Tính đầy đủ (Completeness): *BFS* luôn là đầy đủ nếu đồ thị là hữu hạn. Nếu đồ thị là vô hạn thì BFS là đầy đủ khi đạt điều kiện sau:
  nút đích có thể đi đến ngay từ đầu và không có nút nào có số lượng nút kề vô hạn.
- Tính tối ưu (Optimality): *BFS* là tối ưu nếu chi phí đường đi là hàm không giảm theo độ sâu của nút
- Tính phức tạp (Complexity):

1. Độ phức tạp thời gian (Time complexity): Độ phức tạp thời gian của thuật toán BFS có thể đạt được bằng số lượng nút đi qua trong BFS cho đến khi nút nông nhất. Trong đó d = độ sâu của nghiệm nông nhất và b là một nút ở mọi trạng thái.

$$T(b) = 1 + b^2 + b^3 + ... + b^d = O(b^d)$$

2. Độ phức tạp không gian (Space complexity): Độ phức tạp về không gian của thuật toán BFS được tính bằng kích thước bộ nhớ của danh sách nút là $O(b^d)$.

Minh họa mã giả
Từ Hình 1, ta có thể biểu diễn mã giả như sau:

B1: Visited: []
Queue: [S]

B2: Visited: [S]
Queue: [A, B]

B3: Visited: [S, B]
Queue: [A, C]

B4: Visited: [S, B, A]
Queue: [C, G]

B5: Visited: [S, B, A, C]
Queue: [G]

B6: Visited: [S, B, A, C, G] -> Tới đích: dừng
Queue: []

Kết quả cuối cùng là: $S- > B- > A- > C- > G$

1.3.3 Uniform Cost Search (UCS)

UCS là một loại tìm kiếm không xác định thực hiện tìm kiếm dựa trên chi phí đường dẫn thấp nhất. UCS giúp chúng ta tìm đường đi từ nút bắt đầu đến nút mục tiêu với chi phí đường đi tối thiểu.

Kết quả của một DFS là một cây bao trùm (spanning tree). Cây khung (spanning tree) là một đồ thị không có vòng lặp.

Ý tưởng

Giả sử ta cần di chuyển từ nút A đến nút B, ta sẽ chọn con đường nào? $A- > C- > B$ hoặc $A- > B$?

Chi phí đường đi từ đường A đến đường B là 5 và chi phí đường đi từ A đến C đến B là 4 (2+2). Vì UCS sẽ xem xét chi phí đường đi nhỏ nhất, nghĩa là 4. Do đó, A đến C đến B sẽ được chọn.

Mã giả

Algorithm 4 Uniform Cost Search

```
1: procedure UCS(start)
2:     g(start) ← 0                    ▷ g(node) is the cost of the path from the start to the node
3:     open <- min-priority queue ordered by g, containing only s
4:     closed <- an empty set
5:     while true do
6:         if open is empty then
7:             return failure
8:         end if
9:         v <- pop the lowest-cost node from open
10:        for each neighbor of neighbors(v) do
11:            if neighbor not in closed and neighbor not in open then
12:                g(neighbor) ← g(v) + d(v, neighbor)    ▷ d(v, neighbor) is the distance between v and neighbor
13:                insert neighbor to open
14:            else if u in open such that neighbor = u and g(v) + c(v, neighbor) < g(u) then
15:                remove u from open
16:                insert neighbor to open
17:            end if
18:        end for
19:    end while
20: end procedure
```

Phân tích thuật toán

- Tính đầy đủ (Completeness): Nếu đồ thị tìm kiếm là hữu hạn thì phiên bản tìm kiếm đồ thị của UCS là đầy đủ. Nếu đồ thị là vô hạn nhưng không có nút nào có số lượng lân cận vô hạn và tất cả các cạnh đều có chi phí dương thì UCS tìm kiếm đồ thị cũng sẽ là đầy đủ. Điều kiện chi phí biên hoàn toàn dương đảm bảo rằng sẽ không có một đường đi vô hạn với các cạnh có chi phí bằng 0 mà các nút UCS sẽ tiếp tục mở rộng mãi mãi

- Tính tối ưu (Optimality): UCS luôn mở rộng nút có tổng chi phí đường đi thấp nhất từ nút ban đầu. Vì vậy, chúng luôn tối ưu (vì mọi giải pháp rẻ hơn đều đã được tìm thấy).

- Tính phức tạp (Complexity):

    1. Độ phức tạp thời gian (Time complexity): Hãy gọi chi phí tối thiểu của một cạnh trong biểu đồ tìm kiếm là ϵ. Ngoài ra, cho $C*$ là chi phí của đường đi tối ưu tới bất kỳ nút mục tiêu nào. Để UCS hoàn thiện, giả định rằng $\epsilon > 0$.
    Khi đó, độ sâu trong cây tìm kiếm mà tại đó UCS tìm thấy nút mục tiêu gần nhất tối đa là $1 + [\frac{C^*}{\epsilon}]$. Nếu là giới hạn trên của hệ số phân nhánh thì độ phức tạp về thời gian của UCS dạng cây là $O(b^{1+[\frac{C^*}{\epsilon}]})$.

2. Độ phức tạp không gian (Space complexity): Vì *open* có thể chứa tất cả các nút ở độ sâu của mục tiêu gần nhất nên độ phức tạp của không gian cũng là $O(b^{1+[\frac{C^*}{\epsilon}]})$

Minh họa mã giả

Dựa vào số chi phí tích lũy trên mỗi nút đi qua ở Hình 1, ta có thể tìm được những nút có chi phí bé hơn giúp cho việc tìm kiếm đường đi tốt hơn.

open set: (0, S)
closed set:

open set: (1, S-A), (1, S-B)
closed set: S

open set: (1, S-B), (2, S-A-C)
closed set: S, A

open set: (2, S-A-C), (3, S-B-C)
closed set: S, A, B

open set: ~~(3, S-B-C)~~, (5, S-A-C-G) -> loại S-B-C vì các nút đã được duyệt.
closed set: S, A, B, C

open set: (5, S-A-C-G) -> Tìm được đường đi: dừng
closed set: S, A, B, C, G

Kết quả cuối cùng là : $S->A->C->G$ với chi phí là 5.

1.3.4 A Star (A^*)

A^* là thuật toán tìm kiếm trong đồ thị. Thuật toán này tìm một đường đi từ một nút khởi đầu tới một nút đích cho trước (hoặc tới một nút thỏa mãn một điều kiện đích). Thuật toán này sử dụng một "đánh giá heuristic" để xếp loại từng nút theo ước lượng về tuyến đường tốt nhất đi qua nút đó. Thuật toán này duyệt các nút theo thứ tự của đánh giá heuristic này. Do đó, thuật toán A* là một ví dụ của tìm kiếm theo lựa chọn tốt nhất (best-first search).

Ý tưởng

A^* xây dựng tăng dần tất cả các tuyến đường từ nút xuất phát cho tới khi nó tìm thấy một đường đi chạm tới đích. Tuy nhiên, cũng như tất cả các thuật toán tìm kiếm có thông tin, nó chỉ xây dựng các tuyến đường "có vẻ" dẫn về phía đích.

Để biết những tuyến đường nào có khả năng sẽ dẫn tới đích, A^* sử dụng một **đánh giá heuristic** về khoảng cách từ nút bất kỳ cho trước tới đích, và dựa trên đánh giá này, thuật toán có thể tìm được những đường đi có chi phí thấp và giúp cho quãng đường đi đến đích là ngắn nhất.

Mã giả

Algorithm 5 A Star

```
1: procedure ASTAR(start)
2:     g(start) ← 0                          ▷ g(start) is the cost of the path from the start to the node
3:     f(start) ← g(start) + h(start)    ▷ f(start) is the total cost of g(start) and heuristic value h(start)
4:     open <- min-priority queue ordered by f, containing only s
5:     closed <- an empty set
6:     while true do
7:         if open is empty then
8:             return failure
9:         end if
10:        v <- pop the lowest-cost node from open
11:        for each neighbor of neighbors(v) do
12:            if neighbor not in closed and neighbor not in open then
13:                g(neighbor) ← g(v) + d(v, neighbor)    ▷ d(v, neighbor) is the distance between v and neighbor
14:                f(neighbor) ← g(neighbor) + h(neighbor)
15:                insert neighbor to open
16:            else if u in open such that neighbor = u then
17:                if g(v) + c(v, neighbor) + h(neighbor) < g(u) + h(u) then
18:                    remove u from open
19:                    insert neighbor to open
20:                end if
21:            end if
22:        end for
23:    end while
24: end procedure
```

Phân tích thuật toán

- Tính đầy đủ (Completeness): Nếu đồ thị có hệ số phân nhánh hữu hạn và tất cả các trọng số đều lớn hơn 0 thì A^* là đầy đủ.
- Tính tối ưu (Optimality): Khi A^* được đưa ra một *heuristic* có thể chấp nhận được, nó sẽ luôn tìm được một con đường ngắn nhất vì phương pháp *heuristic* tối ưu sẽ không bao giờ cho phép nó bỏ qua các con đường có thể ngắn hơn.
- Tính phức tạp (Complexity):
    1. Độ phức tạp thời gian (Time complexity): Độ phức tạp về thời gian của A^* phụ thuộc vào heuristic. Trong trường hợp xấu nhất là không gian tìm kiếm không giới hạn, số lượng nút được mở rộng sẽ theo cấp số nhân theo độ sâu của lời giải (đường đi ngắn nhất) d: $O(b^d)$, trong đó b là hệ số phân nhánh (số lượng nút kế thừa trung bình trên mỗi trạng thái) .

2. Độ phức tạp không gian (Space complexity): Độ phức tạp về không gian của A^* luôn là $O(b^d)$, vì chúng ta cần theo dõi mọi nút trong đồ thị mọi lúc, ngay cả những nút mà chúng ta chưa từng truy cập và sẽ không bao giờ truy cập.

- Hàm Heuristic: Heuristic trong A^* chính là khoảng cách từ nút đang xét tới nút đích (*goal*) của đồ thị:
$$h(n) = \sqrt{(n.x - goal.x)^2 + (n.y - goal.y)^2}$$
- Heuristic có thể chấp nhận (Admissible Heuristic): Phương pháp heuristic được coi là có thể chấp nhận được nếu nó không bao giờ trả về chi phí lớn hơn chi phí tối ưu cho cùng một tuyến đường. Ví dụ: nếu tuyến đường hiệu quả nhất giữa hai nút có chi phí là n thì một phương pháp phỏng đoán có thể chấp nhận được sẽ không bao giờ trả lại chi phí lớn hơn chi phí này.

Minh họa mã giả

Với mỗi nút được xét, ngoài chi phí đường đi thì ta sẽ cộng thêm hệ số `heuristic` để tìm đường đi tốt từ Hình 1:

open set: (0 + 2, S)
closed set:

open set: (1 + 1, S-B), (1 + 4, S-A)
closed set: S

open set: (3 + 1, S-B-C), (1 + 4, S-A)
closed set: S, B

open set: (1 + 4, S-A), (6 + 0, S-B-C-G)
closed set: S, B, C

open set: (2 + 1, S-A-C), (6 + 0, S-B-C-G)
closed set: S, B, C, A

open set: (5 + 0, S-A-C-G), (6 + 0, S-B-C-G)
closed set: S, A, B, C

open set: (5 + 0, S-A-C-G), ~~(6 + 0, S-B-C-G)~~ -> Loại S-B-C-G vì có chi phí tới đích lớn hơn
closed set: S, A, B, C, G

Kết quả cuối cùng là : $S->A->C->G$ với chi phí là 5.

1.3.5 Greedy

Greedy (Thuật toán tham lam) là một thuật toán bất kì tuân theo phương pháp phỏng đoán giải quyết vấn đề nhằm đưa ra lựa chọn tối ưu cục bộ ở mỗi giai đoạn. Trong nhiều vấn đề, chiến lược tham lam không tạo ra giải pháp tối ưu, nhưng phương pháp *heuristic* tham lam có thể mang

lại giải pháp tối ưu cục bộ gần đúng với giải pháp tối ưu toàn cục trong một khoảng thời gian hợp lý.

Ý tưởng

Ý tưởng hoạt động của *Greedy* rất đơn giản: tại mỗi bước, nó luôn lựa chọn tối ưu nhất trong số các tùy chọn có sẵn, mà không quan tâm đến tác động của quyết định hiện tại đối với tương lai. Điều này có nghĩa rằng thuật toán tham lam luôn đi theo lựa chọn tốt nhất ngay tại thời nút hiện tại, mà không xem xét các lựa chọn ở các bước sau.

Mã giả

Algorithm 6 Greedy

```
1: procedure GREEDY(start)
2:     f(start) ← h(start)        ▷ f(start) is the total cost of g(start) and heuristic value h(start)
3:     open <- min-priority queue ordered by f, containing only s
4:     closed <- an empty set
5:     while true do
6:         if open is empty then
7:             return failure
8:         end if
9:         v <- pop the lowest-cost node from open
10:        for each neighbor of neighbors(v) do
11:            if neighbor not in closed and neighbor not in open then
12:                f(neighbor) ← h(neighbor)
13:                insert neighbor to open
14:            else if u in open such that neighbor = u then
15:                if h(neighbor) < h(u) then
16:                    remove u from open
17:                    insert neighbor to open
18:                end if
19:            end if
20:        end for
21:    end while
22: end procedure
```

Phân tích thuật toán

- Tính đầy đủ (Completeness): *Greedy* vẫn không đầy đủ ngay cả trong không gian trạng thái hữu hạn, giống như tìm kiếm theo chiều sâu.
- Tính tối ưu (Optimality): *Greedy* là không tối ưu vì nó không quan tâm tới quãng đường tương lai như thế nào, nó chỉ quan tâm đường nào nó đi tiếp theo là có chi phí ngắn nhất.
- Tính phức tạp (Complexity):

1. Độ phức tạp thời gian (Time complexity): Độ phức tạp về thời gian trong trường hợp xấu nhất của tìm kiếm đầu tiên tốt nhất của *Greedy* là $O(b^m)$.
2. Độ phức tạp không gian (Space complexity): Độ phức tạp về không gian và thời gian trong trường hợp xấu nhất đối với phiên bản cây là $O(b^m)$, trong đó m là độ sâu tối đa của không gian tìm kiếm.

Minh họa mã giả

Với mỗi nút được xét, ngoài chi phí đường đi thì ta sẽ cộng thêm hệ số `heuristic` để tìm đường đi tốt từ Hình 1:

open set: (2, S)
closed set:

open set: (1, S-B), (4, S-A)
closed set: S

open set: (1, S-B-C), (4, S-A)
closed set: S, B

open set: (0, S-B-C-G), (4, S-A)
closed set: S, B, C

open set: (0, S-B-C-G), (4, S-A)
closed set: S, B, C

open set: (0, S-B-C-G) -> Tìm được đường đi: dừng
closed set: S, B, C, G

Kết quả cuối cùng là : $S->B->C->G$

1.4 So sánh

1.4.1 UCS, Greedy, AStar

`UCS, Greedy và A*` là ba thuật toán tìm kiếm phổ biến được sử dụng trong lĩnh vực trí tuệ nhân tạo và khoa học máy tính để giải quyết các vấn đề tìm kiếm và tìm đường. Mỗi thuật toán này có đặc điểm và ứng dụng riêng:

UCS

UCS là một biến thể của thuật toán Dijkstra và được sử dụng để tìm đường đi ngắn nhất trong đồ thị có trọng số. Nó khám phá không gian tìm kiếm theo cách tối thiểu hóa tổng chi phí của đường đi từ nút ban đầu đến nút đích. *UCS* xem xét cả chi phí tích luỹ cho đến nay và chi phí để đến các nút láng giềng trong quá trình ra quyết định. Nó là thuật toán đầy đủ và tối ưu, có nghĩa là nó đảm bảo tìm ra đường đi tối ưu nếu có.

Greedy

Greedy là một thuật toán tìm kiếm không thông thạo, nó đưa ra quyết định khám phá nút xuất hiện dường như gần nhất đến mục tiêu dựa trên hệ số *heuristic*. Khác với UCS và A^*, nó không xem xét tổng chi phí để đến một nút mà chỉ ước tính chi phí từ nút hiện tại đến mục tiêu. *Greedy* có thể rất hiệu quả nhưng không đảm bảo tìm ra giải pháp tối ưu. Nó có thể bị kẹt ở điểm cực tiểu địa phương.

A*

A^* là một thuật toán tìm kiếm học tham số kết hợp lợi ích của cả UCS và Tìm kiếm Tham lam. Nó sử dụng hàm tham số để ước tính chi phí từ nút hiện tại đến mục tiêu và chi phí thực tế từ nút ban đầu đến nút hiện tại.

A^* xem xét cả tổng chi phí đã tích luỹ đến nay và chi phí ước tính đến mục tiêu, làm cho nó là thuật toán đầy đủ và tối ưu khi sử dụng hàm tham số thỏa mãn (không đánh giá cao chi phí thực tế). A^* được rộng rãi sử dụng cho các vấn đề tìm đường và duyệt đồ thị, thường tìm ra giải pháp tối ưu trong khi hiệu quả hơn UCS.

Tóm lại, sự lựa chọn giữa `UCS, Greedy` và `A*` phụ thuộc vào vấn đề cụ thể và yêu cầu:

Sử dụng UCS khi bạn cần tìm đường đi tối ưu trong đồ thị có trọng số và hiệu suất không phải là ưu tiên hàng đầu. Sử dụng *Greedy* khi bạn muốn ưu tiên việc khám phá hướng tới mục tiêu mà không quan tâm đến tính tối ưu. Sử dụng A^* khi bạn muốn cân đối giữa việc tìm đường đi tối ưu và khám phá không gian tìm kiếm một cách hiệu quả, với điều kiện bạn có hàm tham số phù hợp.

1.4.2 UCS, Dijkstra

`UCS` và `Dijkstra` có nhiều điểm tương đồng, nhưng cũng có một số điểm khác biệt quan trọng. Dưới đây là sự so sánh giữa `UCS` và thuật toán `Dijkstra`:

Điểm tương đồng

- Cả `UCS` và thuật toán `Dijkstra` được sử dụng để tìm đường đi ngắn nhất trong đồ thị có trọng số (weighted graph).
- Cả hai thuật toán đều dựa trên nguyên tắc tối thiểu hóa chi phí tổng cộng để đến đích.
- `UCS` và `Dijkstra` đều sử dụng hàng đợi ưu tiên (priority queue) để lựa chọn nút tiếp theo để mở rộng dựa trên chi phí tích luỹ đến nút đó.

Điểm khác biệt

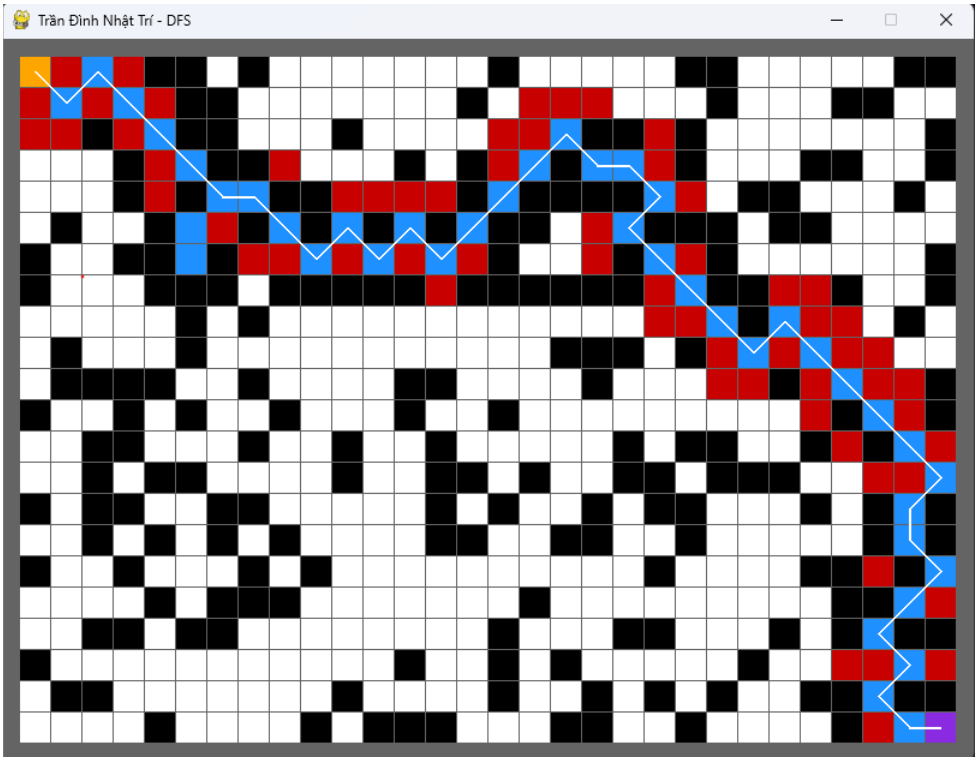
- `Dijkstra` tìm đường đi ngắn nhất từ nút gốc đến mọi nút khác. `UCS` cho các đường đi ngắn nhất về mặt chi phí từ nút gốc đến nút đích.
- `UCS`, dừng lại ngay khi tìm thấy nút đích. Đối với `Dijkstra`, không có trạng thái đích và quá trình xử lý tìm kiếm tiếp tục cho đến khi tất cả các nút bị xóa khỏi hàng đợi ưu tiên, tức là cho đến khi xác định được đường đi ngắn nhất đến tất cả các nút (không chỉ nút đích).
- `UCS` có ít yêu cầu về không gian hơn, trong đó hàng đợi ưu tiên được lấp đầy dần dần trái ngược với của `Dijkstra`, hàng đợi này sẽ thêm tất cả các nút vào hàng đợi ngay từ đầu với chi phí vô hạn.

Tóm lại, `UCS` và thuật toán `Dijkstra` đều có mục tiêu tìm đường đi ngắn nhất trong đồ thị, tuy nhiên `Dijkstra` chỉ có thể áp dụng trong các biểu đồ rõ ràng, trong đó toàn bộ nút trong đồ thị được đưa ra làm đầu vào. `UCS` bắt đầu từ đỉnh nguồn và dần dần đi qua các nút cần thiết của đồ thị. Do đó, nó có thể áp dụng cho cả biểu đồ rõ ràng và biểu đồ ẩn (nơi tạo ra trạng thái/nút).

2 Thực hiện thuật toán

Trong mục này, ta sẽ áp dụng những thuật toán trên để giải mã mê cung (maze solving) giúp tìm ra được đường đi tốt trong mê cung.

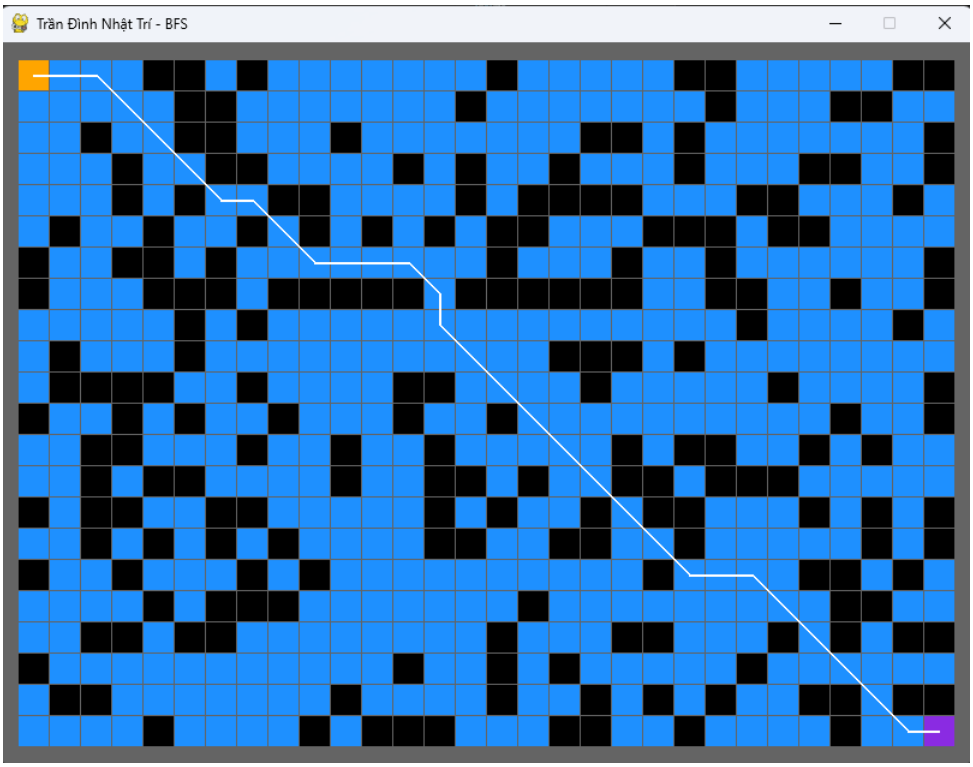
2.1 Depth-First Search (DFS)



Hình 2: Giải mê cung sử dụng thuật toán `DFS`

Ở hình trên ta thấy *DFS* đi khá nhanh (do số điểm nó duyệt trong mê cung không nhiều).

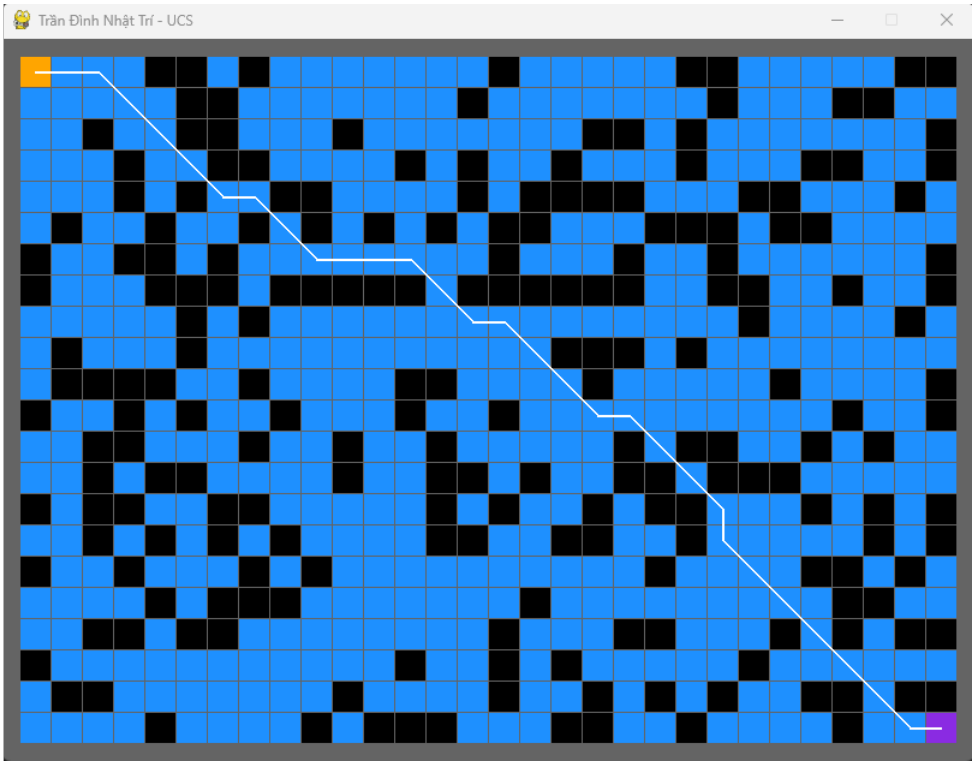
2.2 Breadth-First Search (BFS)



Hình 3: Giải mê cung sử dụng thuật toán `BFS`

Với BFS, ta thấy nó duyệt hết tất cả các điểm trên mê cung tới khi đìm thấy điểm đích.

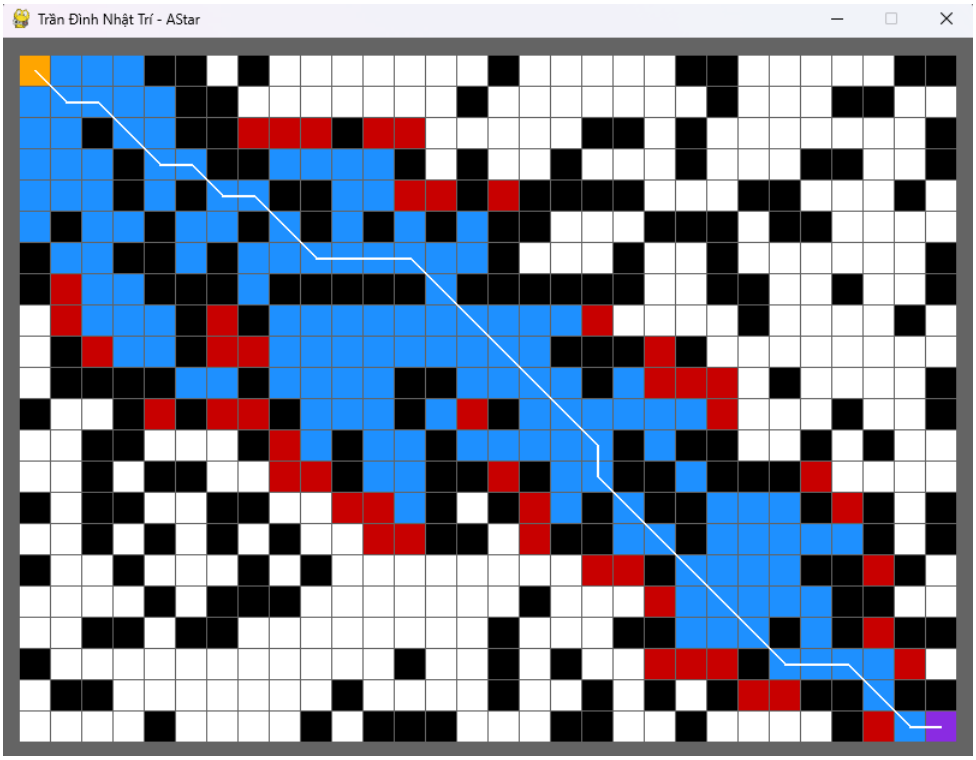
2.3 Uniform Cost Search (UCS)



Hình 4: Giải mê cung sử dụng thuật toán `UCS`

Tương tự như *BFS*, *UCS* cũng sẽ duyệt tất cả các điểm cho tới khi tìm thấy điểm đích, tuy nhiên với khả năng tính chi phí đường đi, *UCS* **có thể** tìm ra con đường tốt hơn BFS một chút.

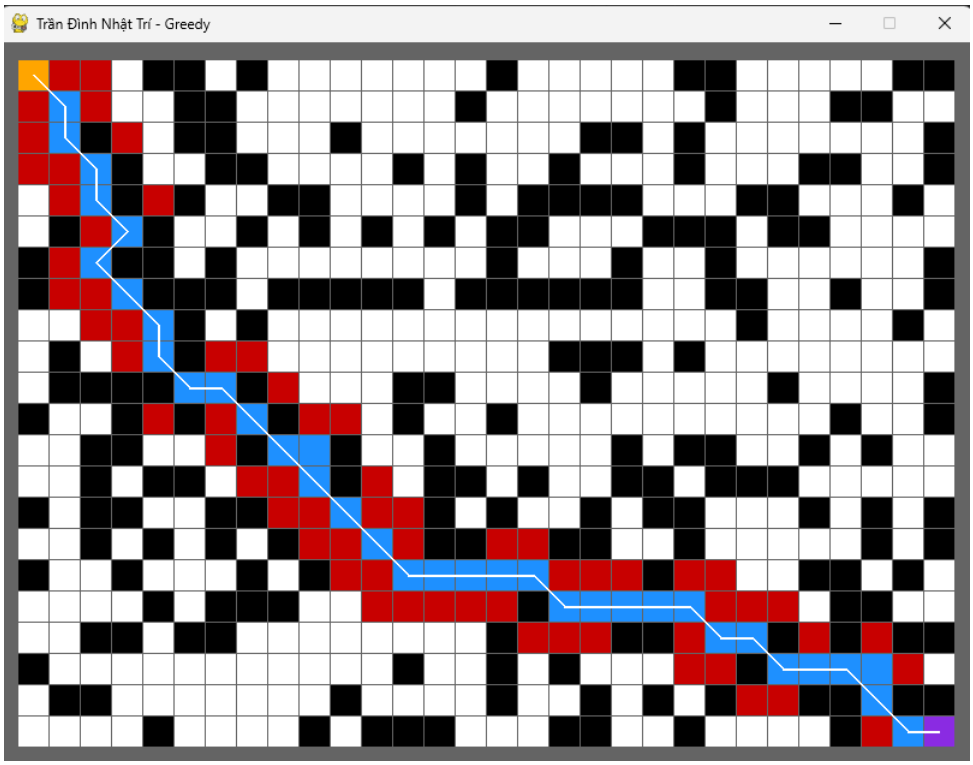
2.4 A Star Search (A^*)



Hình 5: Giải mê cung sử dụng thuật toán `A*`

Với thuật toán A^*, quá trình tìm đường đi của nó rất nhanh, nó có thể nhận ra được những quãng đường nào thuận tiện để đến đích dựa vào giá trị *heuristic*, giúp cho việc duyệt các điểm đi trở nên gọn hơn và thuận tiện hơn.

2.5 Greedy



Hình 6: Giải mê cung sử dụng *Thuật toán tham lam*

Greedy cho ra kết quả nhanh và không duyệt nhiều điểm vì nó chỉ lựa những điểm có giá trị *heuristic* bé nhất ở mỗi điểm nó đi qua.

3 Tài liệu trích dẫn

Search Problem

- wikipedia
- cs.stackexchange

Pseudo code

- aimacode

DFS

- wikipedia
- viblo
- geeksforgeeks
- javatpoint

BFS

- chidokun
- vnoi
- geeksforgeeks

UCS

- educative
- cs.standford

A Star

- wikipedia
- baeldung

Greedy

- javatpoint

Comparison

- stackoverflow