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

CÁC THUẬT TOÁN TÌM KIẾM TRÊN ĐỒ THỊ

MÔN HỌC: CƠ SỞ TRÍ TUỆ NHÂN TẠO

20120541 – Phan Thị Yến Nhi

Thành phố Hồ Chí Minh - 2022

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

CÁC THUẬT TOÁN TÌM KIẾM TRÊN ĐỒ THỊ

MÔN HỌC: CƠ SỞ TRÍ TUỆ NHÂN TẠO

| Giảng viên |

Thầy Nguyễn Ngọc Đức

Thầy Nguyễn Bảo Long

Thành phố Hồ Chí Minh - 2022

MỤC LỤC

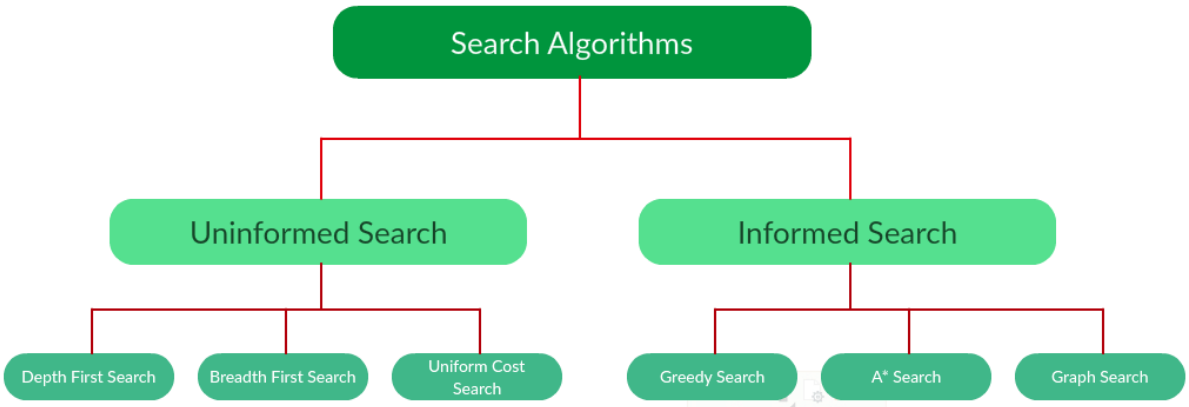
1) Tìm hiểu và trình bày thuật toán.

1.1 Nêu các thành phần của 1 bài toán tìm kiếm và cách giải một bài toán tìm kiếm nói chung. Phân loại bài toán tìm kiếm (Uninformed Search và Informed Search).

- Các thành phần của một bài toán tìm kiếm:
  Một bài toán tìm kiếm có thể được định nghĩa bằng năm 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(state space) hình thành nên một đồ thị có hướng với đỉnh là các thuật toán và cạnh là các hành động.
    Một đường đi (**path**) 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')*
- Cách giải một bài toán tìm kiếm:
  Một lời giải (solution) là một chuỗi hành động di chuyển từ trạng thái bắt đầu cho đến trạng thái đích.
  Các bước chính:
  Xác định mục tiêu cần đạt đến(goal formulation): là một tập hợp của các trạng thái (đích). Dựa trên trạng thái hiện tại (của môi trường) và đánh giá hiệu quả hành động (các tác tử).
  Phát biểu bài toán(problem formulation): với một mục tiêu, xác định các hành động và trạng thái cần xem xét.
  Quá trình tìm kiếm(search process): xem xét các chuỗi hành động có thể. Chọn chuỗi hành động tốt nhất.
  Giải thuật tìm kiếm:
  Đầu vào: một bài toán cần giải quyết.
  Đầu ra: một giải pháp, dưới dạng một chuỗi các hành động cần thực hiện.

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  static: seq, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation
  state ← UPDATE-STATE(state, percept)
  if seq is empty then do
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

- Phân loại bài toán tìm kiếm (Uninformed Search và Informed Search).



- **Uninformed Search Algorithms:**

  Các thuật toán tìm kiếm trong phần này không có thông tin bổ sung nào về nút mục tiêu ngoài thông tin được cung cấp trong định nghĩa vấn đề.

  Các kế hoạch để đạt được trạng thái mục tiêu từ trạng thái bắt đầu chỉ khác nhau bởi thứ tự và / hoặc độ dài của các hành động.

  Chỉ có khả năng sinh successor và phân biệt trạng thái đích.

  Tìm kiếm không được thông tin còn được gọi là Tìm kiếm mù.

  Các thuật toán này chỉ có thể tạo ra các trình kế thừa và phân biệt giữa trạng thái mục tiêu và trạng thái không mục tiêu.

  Mỗi chiến lược tìm kiếm là một thể hiện (đồ thị/ cây) của bài toán tìm kiếm tổng quát.

  Gồm: Depth First Search, Breadth First Search, Uniform Cost Search…

  Mỗi thuật toán này sẽ có:

  - A problem graph: chứa nút bắt đầu S và nút mục tiêu G.

- A strategy: mô tả cách thức mà biểu đồ sẽ được duyệt qua để đến G.
- A Fringe: là một cấu trúc dữ liệu được sử dụng để lưu trữ tất cả các trạng thái có thể có (các nút) mà bạn có thể đi từ các trạng thái hiện tại.
- A tree: kết quả trong khi đi qua nút mục tiêu.
- A solution plan: trong đó trình tự các nút từ S đến G.

- **Informed Search:**

Các thuật toán có thông tin về trạng thái mục tiêu, giúp tìm kiếm hiệu quả hơn. Bên cạnh đing nghĩa còn sử dụng tri thức cụ thể về bài toán. Có khả năng tìm lời giải cụ thể hơn so với Uninformed Search Algorithms.

Thông tin này được thu thập bằng phương pháp heuristic. Một cách đánh giá heuristic tốt sẽ làm cho quá trình tìm kiếm có thông tin hoạt động hiệu quả hơn hẳn một phương pháp tìm kiếm không có thông tin bất kỳ.

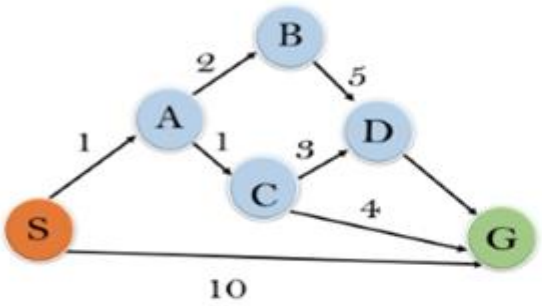
Heuristic: là các kỹ thuật dựa trên kinh nghiệm để giải quyết vấn đề, nhằm đưa ra một giải pháp mà không được đảm bảo là tối ưu Là một hàm ước lượng mức độ gần của một trạng thái so với trạng thái đích và được thiết kế cho từng bài toán tìm kiếm cụ thể.

Heuristic function: Hàm đánh giá dựa trên kinh nghiệm, dựa vào đó để xếp hạng thứ tự tìm kiếm, cách chọn hàm đánh giá quyết định nhiều đến kết quả tìm kiếm.

Gồm: A* Tree Search, A* Graph Search, Greendy Search…

1.2 Trình bày về bốn thuật toán DFS, BFS, UCS, A*:

Ví dụ:



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

a) Thuật toán DFS:

- Depth First Search (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 biểu đồ) và khám phá càng xa càng tốt dọc theo mỗi nhánh trước khi bẻ khóa ngược. Nó sử dụng chiến lược nhập trước - xuất trước cuối cùng và do đó nó được thực hiện bằng cách sử dụng một ngăn xếp.
- Ý tưởng chung:

  Từ một đỉnh, ta đi qua sâu vào từng nhánh. Khi đã duyệt hết nhánh của đỉnh, lùi lại để duyệt đỉnh tiếp theo. Thuật toán dừng khi đi qua hết tất cả các đỉnh có thể đi qua.

  Bắt đầu từ s, mọi đỉnh u kề với s tất nhiên sẽ đến được từ s. Với mỗi đỉnh u đó, những đỉnh v kề với u cũng đến được từ s... Ý tưởng đó gợi ý cho ta viết một thủ tục đệ quy DFS(u) mô tả việc duyệt từ đỉnh u bằng cách thông báo thăm đỉnh u và tiếp tục quá trình duyệt DFS(v) với v là một đỉnh chưa thăm kề với u. Để quá trình duyệt không lặp lại bất kì đỉnh nào, ta dùng kỹ thuật đánh dấu, khi thăm một đỉnh, ta sẽ đánh dấu đỉnh đó lại để các bước duyệt đệ quy kế tiếp không thăm lại đỉnh đó nữa
- Mã giả:

  Dùng đệ qui:

Algorithm 1: Recursive DFS

Data: G: The graph stored in an adjacency list
root: The starting node
Result: Prints all nodes inside the graph in the DFS order
$visited \leftarrow \{false\}$;
$DFS(root)$;

Function $DFS(u)$**:**
  if $visited[u] = true$ **then**
    return;
  end
  $print(u)$;
  $visited[u] \leftarrow true$;
  for $v \in G[u].neighbors()$ **do**
    $DFS(v)$;
  end
end

Khử đệ qui:

Algorithm 2: Iterative DFS

```
Data: G: The graph stored in an adjacency list
      root: The starting node
Result: Prints all nodes inside the graph in the DFS order
visited ← {false};
stack ← {};
stack.push(root);
while ¬stack.empty() do
    u ← stack.top();
    stack.pop();
    if visited[u] = true then
        continue;
    end
    print(u);
    visited[u] ← true;
    for v ∈ G[u] do
        if visited[v] = false then
            DFS(v);
        end
    end
end
```

- Đánh giá:

Tính đầy đủ: DFS hoàn chỉnh nếu cây tìm kiếm là hữu hạn, nghĩa là đối với cây tìm kiếm hữu hạn cho trước, DFS sẽ đưa ra giải pháp nếu nó tồn tại.

Tính tối ưu: DFS không phải là tối ưu, có nghĩa là số bước để đạt được giải pháp hoặc chi phí bỏ ra để đạt được nó là cao.

Độ phức tạp về thời gian: Tương đương với số lượng nút được duyệt trong DFS. $T(n) = 1 + n^2 + \cdots + n^d = O(n^d)$

Độ phức tạp thuật toán DFS phụ thuộc vào phương pháp biểu diễn đồ thị: độ phức tạp là $O(n^2)$ trong trường hợp đồ thị biểu diễn dưới dạng danh sách cạnh, với n là số đỉnh của đồ thị. Độ phức tạp là $O(n.m)$ trong trường hợp đồ thị biểu diễn dưới dạng ma trận kề, với n là số đỉnh, m là số cạnh của đồ thị. Độ phức tạp là $O(\max(n, m))$ trong trường hợp đồ thị biểu diễn dưới dạng danh sách kề, với n là số đỉnh, m là số cạnh của đồ thị.

Độ phức tạp về không gian: Tương đương với mức độ lớn mà phần rìa có thể nhận được . $S(n) = O(n * d)$

Với : d = độ sâu của cây tìm kiếm = số cấp độ của cây tìm kiếm.

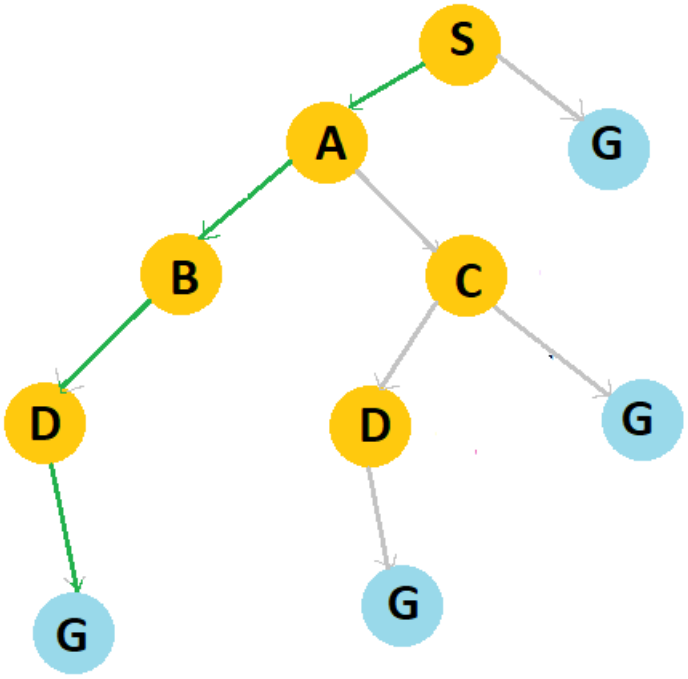
n^i = số lượng nút trong cấp độ i

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.

- Ví dụ:



- Initialization: { [S , 1] }
  Iteration1: { [S->A , 2] , [S->G , 2] }
  Iteration2: { [S->A->B , 3] , [S->A->C , 3] , [S->G , 2] }
  Iteration3: { [S->A->B->D , 4] , [S->A->C , 3] , [S->G , 2] }
  Iteration4: { [S->A->B->D->G , 5] , [S->A->C , 3] , [S->G , 2 } }
  Iteration5 đưa ra đầu ra cuối cùng là S->A->B->D->G

b) Thuật toán BFS:

- Breadth-first search (BFS): là một thuật toán để duyệt qua hoặc tìm kiếm cấu trúc dữ liệu dạng cây hoặc đồ thị. Nó bắt đầu ở gốc cây (hoặc một số nút tùy ý của biểu đồ, đôi khi được gọi là 'khóa tìm kiếm') và khám phá tất cả các nút lân cận ở độ sâu hiện tại trước khi chuyển sang các nút ở cấp độ sâu tiếp theo. Nó được thực hiện bằng cách sử dụng một hàng đợi.
- Ý tưởng chung:

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

Với đồ thị không trọng số và đỉnh nguồn ss. Đồ thị này có thể là đồ thị có hướng hoặc vô hướng, điều đó **không quan trọng** đối với thuật toán.

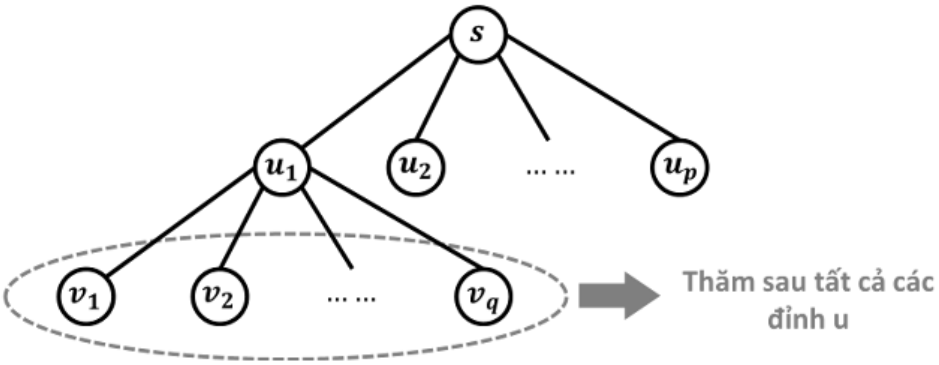
Có thể hiểu thuật toán như một ngọn lửa lan rộng trên đồ thị:

- Ở bước thứ 0, chỉ có đỉnh nguồn s đang cháy.
- Ở mỗi bước tiếp theo, ngọn lửa đang cháy ở mỗi đỉnh lại lan sang tất cả các đỉnh kề với nó.

Trong mỗi lần lặp của thuật toán, "vòng lửa" lại lan rộng ra theo chiều rộng. Những đỉnh nào gần s hơn sẽ bùng cháy trước.

Chính xác hơn, thuật toán có thể được mô tả như sau:

- Đầu tiên ta thăm đỉnh nguồn s.
- Việc thăm đỉnh s sẽ phát sinh thứ tự thăm các đỉnh (u1,u2,…up)(u1,u2,…up) kề với s (những đỉnh gần s nhất). Tiếp theo, ta thăm đỉnh u1, khi thăm đỉnh u1 sẽ lại phát sinh yêu cầu thăm những đỉnh (v1,v2,…,vq) kề với u1. Nhưng rõ ràng những đỉnh v này "xa" s hơn những đỉnh u nên chúng chỉ được thăm khi tất cả những đỉnh uu đều đã được thăm. Tức là thứ tự thăm các đỉnh sẽ là: s,u1,u2,…,up,v1,v2,…,vq,…



- Mã giả:

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  if problem's initial state is a goal then return empty path to initial state
  frontier ← a FIFO queue initially containing one path, for the problem's initial state
  reached ← a set of states; initially empty
  solution ← failure
  while frontier is not empty do
      parent ← the first node in frontier
      for child in successors(parent) do
          s ← child.state
          if s is a goal then
              return child
          if s is not in reached then
              add s to reached
              add child to the end of frontier
  return solution
```

- Đánh giá:

  Tính đầy đủ: BFS đã hoàn thành, nghĩa là đối với một cây tìm kiếm nhất định, BFS sẽ đưa ra giải pháp nếu nó tồn tại..

  Tính tối ưu: BFS là tối ưu miễn là chi phí của tất cả các cạnh bằng nhau.

  Độ phức tạp về thời gian: Tương đương với số lượng nút được duyệt trong BFS cho đến khi có giải pháp nông nhất. $T(n) = 1 + n^2 + \cdots + n^s = O(n^s)$

  Độ phức tạp thuật toán BFS phụ thuộc vào phương pháp biểu diễn đồ thị: độ phức tạp là $O(n^2)$ trong trường hợp đồ thị biểu diễn dưới dạng danh sách cạnh, với n là số đỉnh của đồ thị. Độ phức tạp là $O(n.m)$ trong trường hợp đồ thị biểu diễn dưới dạng ma trận kề, với n là số đỉnh, m là số cạnh của đồ thị. Độ phức tạp là $O(\max(n, m))$ trong trường hợp đồ thị biểu diễn dưới dạng danh sách kề, với n là số đỉnh, m là số cạnh của đồ thị.

  Độ phức tạp về không gian: Tương đương với mức độ lớn mà phần rìa có thể nhận được . $S(n) = O(n^s)$

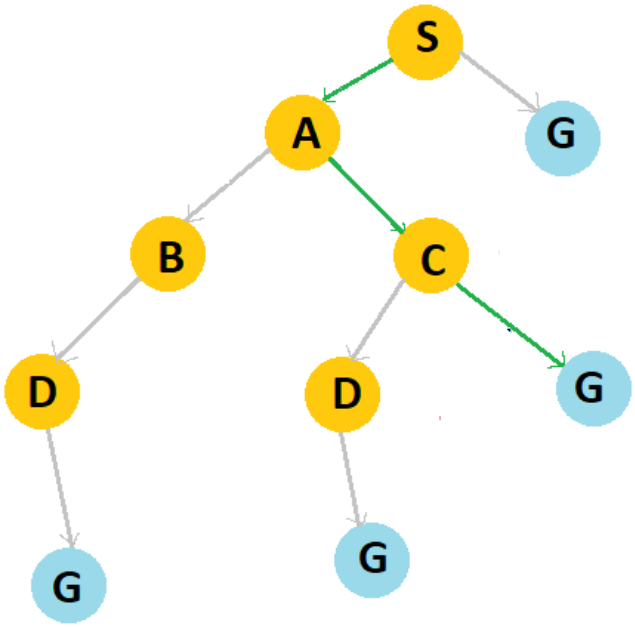
  Với : s= độ sâu của solution

  n^i = số lượng nút trong cấp độ i

  Thuận lợi: BFS sẽ cung cấp giải pháp nếu có bất kỳ giải pháp nào. 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.

- Ví dụ:



Initialization: { [S] }
Iteration1: { [S->A] }
Iteration2: { [S->A->B], [S->A->C] }
Iteration3: { [S->A->B->D], [S->A->C->D], [S->A->C->G] }
Iteration4 đưa ra đầu ra cuối cùng là S->A->C->G.

c) Thuật toán UCS:

Tìm kiếm theo chi phí thống nhất (UCS) là một thuật toán tìm kiếm được sử dụng để duyệt qua một cây hoặc đồ thị có trọng số. Thuật toán này phát huy tác dụng khi có một chi phí khác nhau cho mỗi cạnh. Mục tiêu chính của tìm kiếm chi phí thống nhất là tìm đường dẫn đến nút mục tiêu có chi phí tích lũy thấp nhất. Tìm kiếm chi phí thống nhất 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 tìm kiếm chi phí thống nhất được thực hiện bởi hàng đợi ưu tiên. Nó ưu tiên tối đa cho chi phí tích lũy thấp nhất. Tìm kiếm chi phí đồng nhất tương đương với thuật toán BFS nếu chi phí đường dẫn của tất cả các cạnh là như nhau.

- Ý tưởng chung:

Áp dụng các khái niệm tương tự từ cách tiếp cận trước đó, nhưng thay vì lưu trữ tất cả các nút nằm trên đường dẫn từ nút nguồn đến nút hiện tại, sẽ chỉ lưu trữ nút đến trước nút trong đường dẫn. Trong cách tiếp cận trước, mỗi nút tương tự như nút cha của nó sau khi thêm nút hiện tại. Vì vậy, chỉ có thể lưu trữ nút thay vì sao chép danh sách hoàn chỉnh vào nút.

Sau khi kết thúc thuật toán tìm kiếm chi phí thống nhất, sẽ nối nút đích vào đường dẫn và di chuyển đến nút và thêm nó vào đầu đường dẫn. Tiếp tục theo dõi các nút cho đến khi đến được nút nguồn. Tại thời điểm đó, sẽ có đường dẫn đi từ nút này sang nút khác.

- Mã giả:

Algorithm 1: Uniform-Cost Search

Data: s: the start node, *goal*: a function that can check if a node is a goal node, *successors*: a function that returns the nodes whose states we obtain by applying an action to the input node's state, c: a function that returns the cost of an edge between two nodes.

Result: The optimal path to a goal or *failure* if no node passes the goal test.

```
g(s) ← 0
// g(node) is the cost of the path from s to the node.
frontier ← a min-priority queue ordered by g, containing only s
expanded ← an empty set
while true do
    if frontier is empty then
        return failure
    end
    v ← pop the lowest-cost node from frontier
    if goal(v) = true then
        return v
    end
    expanded ← expanded ∪ {v.state}
    for w ∈ successors(v) do
        if w.state ∉ expanded and no frontier node represents
         w.state then
            g(w) ← g(v) + c(v, w)
            Insert w in frontier
        else if there's a frontier node u such that w.state = u.state
         and g(v) + c(v, w) < g(u) then
            Remove u from frontier
            Insert w in frontier
    end
end
```

- Đánh giá: với chi phí thấp nhất là ϵ, C^* là chi phí lời giải tối ưu

Tính đầy đủ: có nếu $\epsilon > 0$ và không gian trạng thái hữu hạn.

Nếu đồ thị tìm kiếm là hữu hạn, thì tìm kiếm đồ thị của UCS đã đầy đủ. Nếu đồ thị là vô hạn, nhưng không có nút nào có vô số lân cận và tất cả các cạnh có chi phí dương, thì UCS tìm kiếm đồ thị cũng sẽ đầy đủ. Điều kiện rằng chi phí cạnh phải hoàn toàn dương đảm bảo rằng sẽ không có một con đường vô hạn với các cạnh chi phí bằng không mà các nút UCS sẽ tiếp tục mở rộng mãi mãi.

UCS dạng cây có thể bị mắc kẹt trong một vòng lặp ngay cả khi đồ thị là hữu hạn.

Tính tối ưu: luôn tối ưu vì nó chỉ chọn một đường dẫn có chi phí đường dẫn thấp nhất.

Độ phức tạp về thời gian: gọi C* là chi phí của giải pháp tối ưu, và ϵ là mỗi bước để tiến gần hơn đến node đích. Khi đó số bước là

$C^*/\varepsilon + 1$. Do đó độ phức tạp về thời gian trong trường hợp xấu nhất là $O(b^{1+C^*/\varepsilon})$

Độ phức tạp về không gian: gọi C* là chi phí của giải pháp tối ưu, và ϵ là mỗi bước để tiến gần hơn đến node đích. Khi đó số bước là $C^*/\varepsilon + 1$. Do đó độ phức tạp về không gian trong trường hợp xấu nhất là $O(b^{1+C^*/\varepsilon})$

Thuận lợi:

UCS chỉ hoàn thành nếu các trạng thái là hữu hạn và không được có vòng lặp với trọng số bằng không.

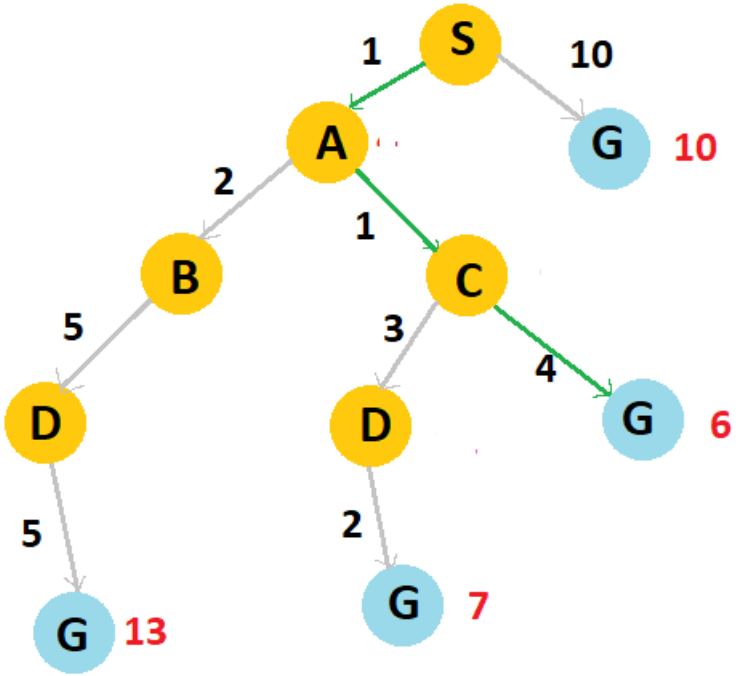
UCS chỉ là tối ưu nếu không có chi phí âm.

Nhược điểm:

Khám phá các tùy chọn theo mọi "hướng".

Không có thông tin về vị trí mục tiêu.

- Ví dụ:



Initialization: { [S , 0] }
Iteration1: { [S->A , 1] , [S->G , 10] }
Iteration2: { [S->A->C , 2] , [S->A->B , 3] , [S->G , 10] }
Iteration3: { [S->A->B , 3] , [S->A->C->D , 5] , [S->A->C->G , 6] , [S->G , 10] }
Iteration4: { [S->A->B , 3] , [S->A->C->G , 6] , [S->A->C->D->G , 7] , [S->G , 10] }
Iteration5: { [S->A->C->G , 6] , [S->A->C->D->G , 7] , [S->A-

>B->D , 8] , [S->G , 10] }
Iteration6 đưa ra đầu ra cuối cùng là S->A->C->G.

d) Thuật toán A*:

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)*.

Thuật toán tìm kiếm A * tìm đường đi ngắn nhất qua không gian tìm kiếm bằng cách sử dụng hàm heuristic, 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)*.

- Ý tưởng chung:

Kết hợp UCS và tìm kiếm tham lam (Greedy search). Tích hợp Heuristic vào quá trình tìm kiếm.

Tránh các đường có chi phí lớn.

Sử dụng hàm đánh giá f(n) = g(n) + h(n) với g(n) là chi phí ước lượng từ node gốc đến node hiện tại, h(n) là chi phí ước lượng từ node hiện tại n tới đích, f(n) là chi phí tổng thể ước lượng của đường đi qua node hiện tại n đến đích.

Một heuristic h(n) được xem là chấp nhận được nếu đối với mọi node n: 0 <= h(n) <= h*(n), trong đó h*(n) là chi phí thật(thực tế) đi từ node n đến đích, ước lượng chấp nhận được có xu hướng đánh giá "lạc quan".

Thông thường, admissible heuristic là các lời giải cho các bài toán được nới lỏng ràng buộc, ở đó có nhiều hành động được cho phép.

Nếu hàm đánh giá h(n) chấm nhận được thì thuật toán A* là tối ưu.

Có hai phương pháp để tính giá trị heuristic:

Exact heuristic: tìm chính xác giá trị của heuristic, nhưng tốn thời gian. Một số phương pháp: Tính toán trước khoảng cách giữa mỗi cặp ô trước khi chạy thuật toán A*, Nếu không có ô / chướng ngại vật nào bị chặn thì chúng ta chỉ có thể tìm giá trị chính xác của heuristic mà không cần tính toán trước bằng cách sử dụng công thức khoảng cách / Khoảng cách Euclide

Approximation heuristic: là một kỹ thuật được thiết kế để giải một vấn đề nhanh hơn khi các phương pháp cổ điển quá chậm hoặc để tìm một giải pháp gần đúng khi các phương pháp cổ điển không tìm được bất kỳ giải pháp chính xác nào.

Đề xuất: dùng hàm heuristic của Approximation heuristic, các bài toán nới lỏng:

- Khoảng cách Euclid (L^2): $h_2(n) = \sqrt{(x_n - x_{Goal})^2 + (y_n - y_{Goal})^2}$
- Khoảng cách Manhattan (L^1): $h_1(n) = |x_n - x_{Goal}| + |y_n - y_{Goal}|$
- Khoảng cách Chebyshev (L^∞): $h_\infty(n) = \max(|x_n - x_{Goal}|, |y_n - y_{Goal}|)$

Vì: sử dụng heuristic của các bài toán nới lỏng dễ dàng hơn, không phải xây dựng thuật toán, giúp làm giảm chi phí, loại bỏ các ràng buộc, tạo lời giải gần đúng, tìm kiếm dễ dàng hơn, tiết kiệm thời gian và các bài con độc lập với nhau . Tuy nhiên, các bài toán nới lỏng được sử dụng để lấy giá trị cho hàm heuristic nhằm định hướng tìm kiếm. Các lời giải của bài toán nới lỏng không phải lời giải của bài toán gốc.

- Mã giả:

```
1. Khởi tạo tập OPEN = {start}, CLOSED = {}, g(start) = 0, f(start) = h(start)
2. If OPEN == {}  then return failure
3. Chọn trạng thái có chi phí nhỏ nhất từ OPEN, n. Đưa n vào CLOSED.
4. If n là trạng thái đích then return solution
5. Mở rộng node n. Với mỗi m là node con của n, ta có:
        If m không thuộc OPEN hay CLOSED then:
                Gán g(m) = g(n) + C(n, m)
                Gán f(m) = g(m) + h(m)
                Thêm m vào OPEN
        If m thuộc OPEN hoặc CLOSED then:
                Gán g(m) = min {g(m), g(n) + cost(n, m)}
                Gán f(m) = g(m) + h(m)
        If f(m) giảm và m thuộc CLOSED then:
                Đưa m trở lại OPEN
End
Quay lại bước 2.
```

- **Đánh giá**: với chi phí thấp nhất là ϵ, C^* là chi phí lời giải tối ưu

Tính đầy đủ: có nếu $\epsilon > 0$ và không gian trạng thái hữu hạn.

Thuật toán A* là đầy đủ nếu: Hệ số phân nhánh là hữu hạn; Chi phí cho mọi hành động là cố định.

Tính tối ưu: Thuật toán A* là tối ưu nếu: Heuristic hợp lý và nhất quán.

Có thể chấp nhận điều kiện: điều kiện đầu tiên yêu cầu để tối ưu là *h(n)* phải là một hàm Heuristic có thể chấp nhận được cho tìm kiếm cây A*.

Độ phức tạp về thời gian: cấp số mũ. Khi xem xét biểu đồ, chúng ta có thể phải di chuyển hết các cạnh để đến ô đích từ ô. Vì

vậy, độ phức tạp thời gian trong trường hợp xấu hơn là O(E), trong đó E là số cạnh trong đồ thị

Độ phức tạp về không gian: cấp số mũ. Trong trường hợp xấu hơn, chúng ta có thể có tất cả các cạnh bên trong danh sách mở, vì vậy không gian bổ trợ cần thiết trong trường hợp xấu nhất là O(V), trong đó V là tổng số đỉnh.

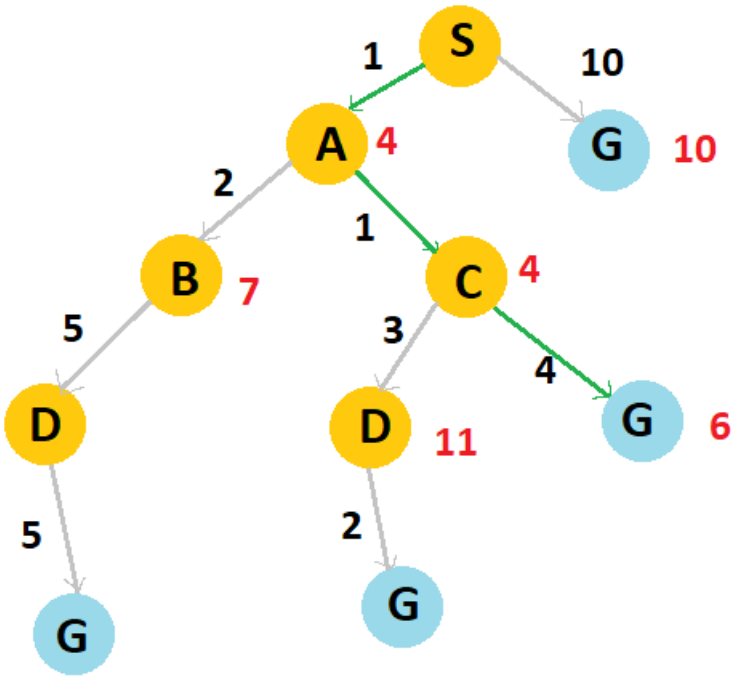
Thuận lợi:

- Thuật toán tìm kiếm A* là thuật toán tốt nhất so với các thuật toán tìm kiếm khác.
- Thuật toán tìm kiếm A* là tối ưu và hoàn chỉnh.
- Thuật toán này 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:



Initialization: {(S, 5)}

Iteration1: {(S--> A, 4), (S-->G, 10)}

Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}

Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}

Iteration 4 sẽ cho kết quả cuối cùng, vì S ---> A ---> C ---> G nó cung cấp đường dẫn tối ưu với chi phí 6.

2) So sánh:

2.1 So sánh sự khác biệt giữa UCS, Greedy và A*:

	UCS	Greedy	A*
Chiến lược/ Phân loại	Tìm kiếm mù, không sử dụng hàm hàm đánh giá	Tìm kiếm kinh nghiệm, có sử dụng hàm đánh giá	Tìm kiếm kinh nghiệm, có sử dụng hàm đánh giá
Chi phí	Tính từ node bắt đầu đến node hiện tại. Mở rộng nút có chi phí thấp nhất	Tính từ node trước đến node hiện tại. Mở rộng node có khoảng cách ước tính đến mục tiêu là nhỏ nhất.	Đi từ một node khởi đầu tới một nút đích cho trước (hoặc tới một nút thỏa mãn một điều kiện đích). Sử dụng một "đánh giá heuristic" để xếp loại từng node theo ước lượng về tuyến đường tốt nhất đi qua nút đó
Tìm đường dẫn	Tìm đường đi có chi phí thấp nhất, mở n node.	Hữu ích trong việc tìm đường đi ngắn nhất, nhưng không mở rộng tất cả các đỉnh	Hữu ích trong việc tìm đường đi ngắn nhất bằng cách mở rộng mở rộng tất cả các đỉnh
Chức năng đánh giá	Không sử dụng	Hàm đánh giá cho tìm kiếm đầu tiên tốt nhất là f (n) = h (n).	Hàm đánh giá cho tìm kiếm A * là f (n) = h (n) + g (n).
Past Knowledge	Không sử dụng đến kiến thức trong quá khứ.	Thuật toán tìm kiếm này không liên quan đến kiến thức trong quá khứ.	Thuật toán tìm kiếm này liên quan đến kiến thức trong quá khứ.
Tính đầy đủ	Có	chưa hoàn thành.	Có
Tính tối ưu	Tối ưu	không phải là tối ưu vì đường dẫn được tìm thấy có thể không tối ưu.	tối ưu vì đường dẫn được tìm thấy luôn là tối ưu.

Độ phức tạp thời gian	$O(b^{1+C^*/\varepsilon})$	$O(b^m)$ trong đó b là phân nhánh và m là độ sâu tối đa của cây tìm kiếm	$O(b^m)$ trong đó b là phân nhánh và m là độ sâu tối đa của cây tìm kiếm
Độ phức tạp không gian	$O(b^{1+C^*/\varepsilon})$	có thể là đa thức.	$O(b^m)$ trong đó b là phân nhánh và m là độ sâu tối đa của cây tìm kiếm
Loại nút được lưu giữ	Chỉ lưu các đỉnh cần thiết	giữ tất cả các nút rìa hoặc đường viền trong bộ nhớ trong khi tìm kiếm.	giữ tất cả các nút trong bộ nhớ trong khi tìm kiếm.
Hướng	Phát triển theo mọi hướng.	Hướng đến đích	Hướng đến đích
Phương thức hoạt động	Sử dụng phương pháp đo chiều dài của đường (path length), đảm bảo tìm ra đường gắn nhất. f(n) = g(n) trong đó g(n) là chi phí đường đi thấp nhất đến n	Sử dụng phương pháp đo lường (heuristic) cụ hể phần tốt nhất của một trạng thái để đạt ến đích nhanh nhất hoặc tìm thấy trạng thái ích mong đợi. f(n) = h(n) trong đó h(n) là h(n) là hàm heuristic	Sử dụng phương pháp đo chiều dài đường và hai thác thông tin heuristic đảm bảo tìm ra ường ngắn nhất nhưng nhanh hơn so với hương pháp uninformed. f(n) = g(n) + h(n) với g(n) là chi phí đường đi thấp nhất đến n, h(n) là hàm heuristic
Sử dụng thông tin về trạng thái đích	Không sử dụng	Có sử dụng	Có sử dụng

2.2 So sánh sự khác biệt giữa UCS và Dijkstra:

Cả thuật toán Dijkstra và thuật toán UCS đều có thể giải quyết vấn đề về đường đi ngắn nhất với cùng độ phức tạp về thời gian. Chúng có cấu trúc mã tương tự nhau. Ngoài ra, chúng tôi sử dụng cùng một công thức $dist[v] = \min(dist, dist[u] + w(u, v))$ để cập nhật giá trị khoảng cách của mỗi đỉnh.

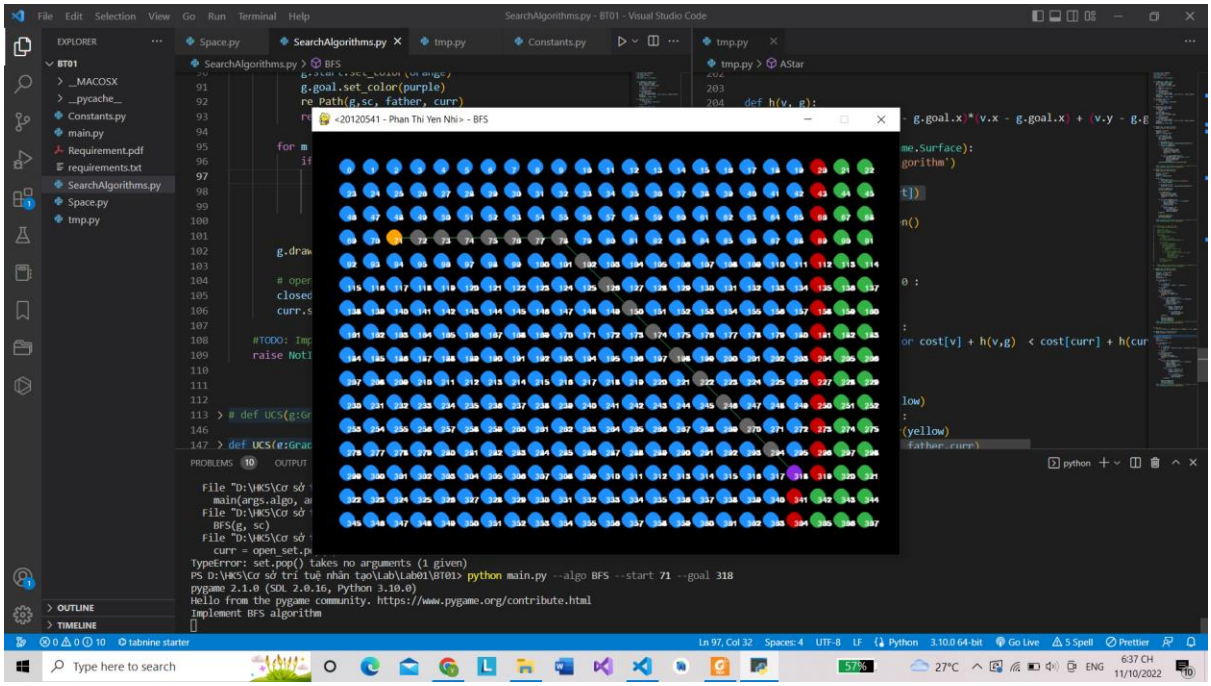
	UCS	**Dijkstra**
Khởi tạo	Một tập chỉ có đỉnh nguồn	Tập hợp tất cả các đỉnh trong đồ thị

Đường đi	Tìm kiếm đường đi ngắn nhất về mặt chi phí đến node đích	Tìm kiếm đường dẫn ngắn nhất từ gốc đến node khác trong đồ thị
Yêu cầu bộ nhớ	Chỉ lưu các đỉnh cần thiết	Lưu toàn bộ đồ thị
Thời gian chạy	Nhanh hơn do yêu cầu bộ nhớ ít hơn	Chậm hơn do diều cầu bộ nhớ lớn hơn
Application	Áp dụng cho cả đồ thị tường minh và đồ thị không tường minh.	Chỉ có thể áp dụng cho các đồ thị rõ ràng mà chúng ta biết tất cả các đỉnh và cạnh
Khoảng cách khi không tìm thấy đường đi	Không tồn tại	Vô cùng
Xây dựng	Xây dựng trên cây	Xây dựng trên đồ thị chung

3) Cài đặt:

a) Thuật toán BFS:

- Kết quả chạy chương trình:



- Quá trình tìm kiếm: từ node bắt đầu (node 71) đến node đích (node 318).

  + Khởi tạo: khởi tạo các list open_set, closed_set, father. Các đỉnh đều chưa được add và open_set, ngoại trừ node bắt đầu s (g.start – node 71).

  + Lặp lại các bước cho đến khi open_set rỗng:

    Lấy node curr ra khỏi open_set.(node ở đỉnh queue).

    Nếu curr là node đích => Truy vết tìm đường đi.

Xét tất cả những node child kề với curr mà chưa được đánh dấu: Đánh dấu child là đã thăm, lưu lại node cha của child, add child và open_set.

Cập nhật father cho mỗi node đã thăm. Xóa curr khỏi open_set và add curr vào closed_set.

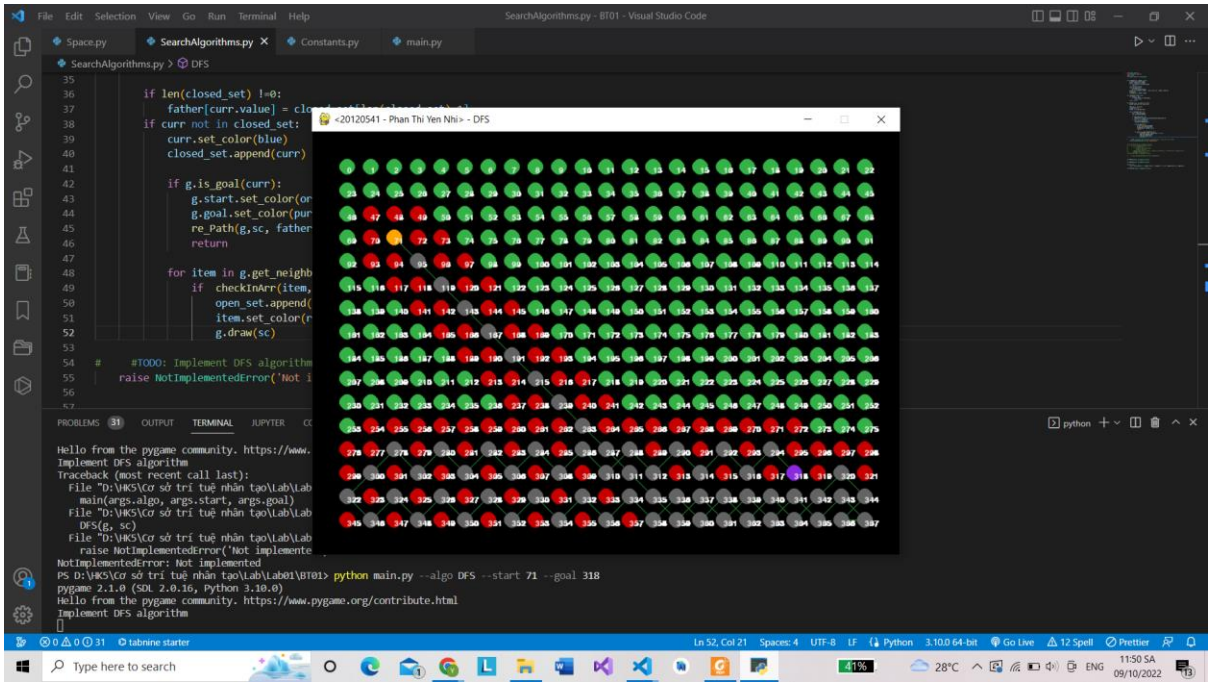
+ Truy vết tìm đường đi.

- Nhận xét:

BFS cũng là thuật toán tìm kiếm mù, mang tính chất vét cạn.
Có thể có nhiều cách di chuyển từ start tới goal, nhưng giải thuật BFS luôn trả về đường đi ngắn nhất (đi qua ít cạnh nhất).

b) Thuật toán DFS:

- Kết quả chạy chương trình:



- Quá trình tìm kiếm: từ node bắt đầu (node 71) đến node đích (node 318).

+ Khởi tạo : khởi tạo các list open_set, closed_set, father. Các đỉnh đều chưa được add và open_set, ngoại trừ node bắt đầu s (g.start – node 71).

+ Lặp lại các bước cho đến khi open_set rỗng:

Lấy node curr ra khỏi open_set.(node ở đỉnh stack)

Nếu curr là node đích => Truy vết tìm đường đi.

Xét tất cả những node child kề với curr mà chưa được đánh dấu: Đánh dấu child là đã thăm, lưu lại node cha của child, add child và open_set.

Cập nhật father cho mỗi node đã thăm. Xóa curr khỏi open_set và add curr vào closed_set.

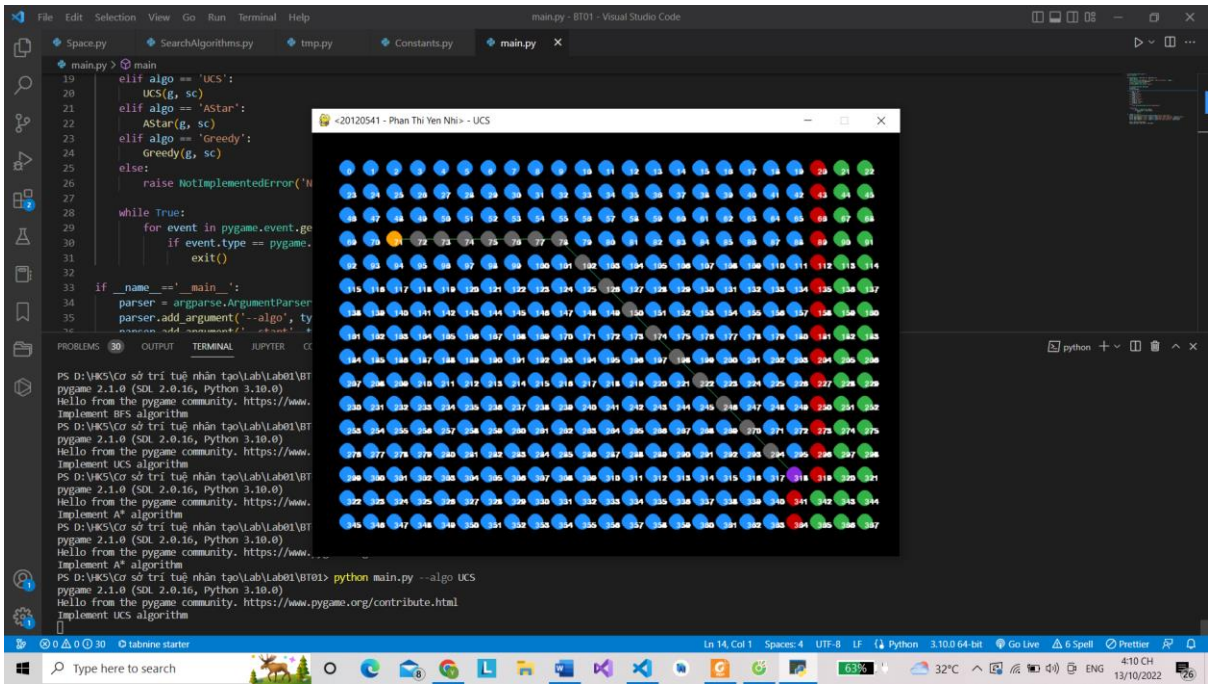
+ Truy vết tìm đường đi.

- Nhận xét:

DFS là thuật toán tìm kiếm mù, mang tính chất vét cạn và sẽ kém hiệu quả nếu tập đỉnh quá lớn. Tốc độ tương đối chậm và chưa tối ưu.
Hàm DFS(u) chỉ được gọi không quá n lần.
Có thể tồn tại nhiều đường đi từ start tới goal, nhưng giải thuật DFS luôn luôn trả về đường đi có thứ tự từ điển nhỏ nhất.

c) Thuật toán UCS:

- Kết quả chạy chương trình:



- Quá trình tìm kiếm: từ node bắt đầu (node 71) đến node đích (node 318).
+ Khởi tạo: khởi tạo các list open_set, closed_set, father, cost.
+ Lặp lại các bước cho đến khi open_set rỗng:
+ Bắt đầu lấy curr là node có chi phí thấp nhất theo cost(n) trong open_set và kiểm tra xem đã đến được node đích chưa, nếu curr là node đích => kết thúc, nếu không => tiếp tục.
+ Mở các node child của node curr. Vì node curr đã được truy cập do đó được thêm vào Danh sách đã truy cập(closed_set).
+ Duyệt qua từng node con của curr. Nếu chưa thăm => đánh dấu là đã thăm và tính lai chi phí cho từng node con. Sau đó, nó so sánh chi phí của đường dẫn rồi chọn đường dẫn chi phí tối thiểu và tiếp tục truyền. Nếu node con đã thăm và có chi phí cost(n) > cost(curr) + chi phí từ curr đến n thì cập nhật lại cost(n). Trong đó cost(n) là chi phí từ node gốc đến n.
+ Cập nhật father cho mỗi node đã thăm. Xóa curr khỏi open_set và add curr vào closed_set.
+ Truy vết tìm đường đi.
- Nhận xét:

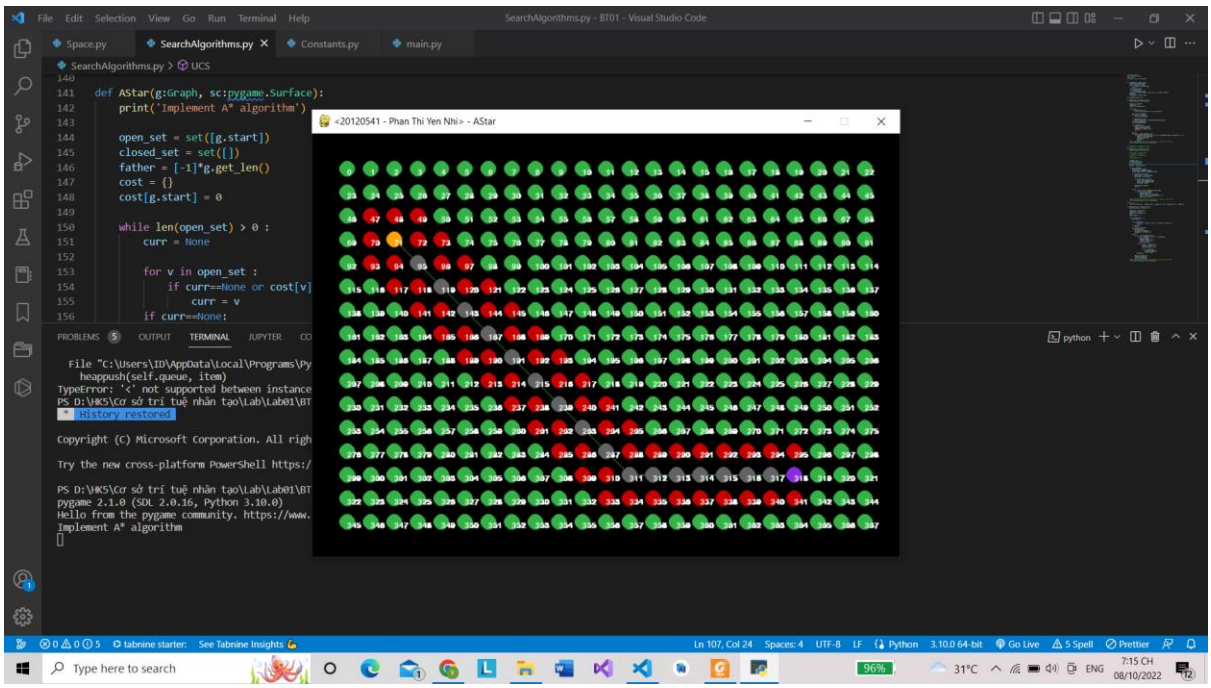
UCS chỉ dùng g(n) nên chậm.

Tìm kiếm chi phí thống nhất là một loại thuật toán tìm kiếm không được thông tin và là giải pháp tối ưu để tìm đường dẫn từ nút gốc đến nút đích với chi phí tích lũy thấp nhất trong không gian tìm kiếm có trọng số nơi mỗi nút có chi phí truyền tải khác nhau. Nó tương tự như Heuristic Search, nhưng không có thông tin Heuristic nào được lưu trữ, có nghĩa là h = 0.

Khi tất cả các cạnh có chi phí bằng nhau, UCS giống với Breadth-First Search.

d) Thuật toán A*:

- Kết quả chạy chương trình:



- Quá trình tìm kiếm: từ node bắt đầu (node 71) đến node đích (node 318).

+ Khởi tạo: khởi tạo các list open_set, closed_set, father, cost.

+ Lặp lại các bước cho đến khi open_set rỗng

Lấy node có chi phí thấp nhất theo cost(n) + h(n) trong open_set. Trong đó cost(n) là chi phí từ node gốc đến n, h(n) là heuristic.

Kiểm tra nếu node hiện tại là node đích => truy vết tìm đường đi.

Duyệt qua từng node con của curr. Nếu chưa thăm => đánh dấu là đã thăm và tính lai chi phí cho từng node con. Nếu node con đã thăm và có chi phí cost(n) > cost(curr) + chi phí từ curr đến n thì cập nhật lại cost(n).

Cập nhật father cho mỗi node đã thăm. Xóa curr khỏi open_set và add curr vào closed_set.

+ Truy vết tìm đường đi.

- Nhận xét:

  Tìm đường đi ngắn nhất một cách hiệu quả. Tốc độ và độ chính xác nhanh và hiệu quả.

  A* sử dụng một số kiến thức cụ thể về vấn đề, do đó hiệu quả hơn để tìm kiếm mục tiêu.

  Kết quả thu được vừa tối ưu vừa hiệu quả.

4) Một số thuật toán tìm kiếm khác:

a) Greedy search(Best first search):

- Ý tưởng chung:

  Trong Greedy search, mở rộng nút gần nhất với nút mục tiêu. "Độ gần" được ước tính bằng heuristic h (x).

  Chiến lược: Mở rộng nút gần nhất với trạng thái mục tiêu, tức là mở rộng nút có giá trị h thấp hơn.

- Mã giả:

Algorithm 2: Best-First Search

Data: s: the start node, *goal*: a function that can check if a node is a goal node, *successors*: a function that returns the nodes whose states we obtain by applying an action to the input node's state, c: the function that returns the cost of an edge between two nodes, f: the function to order the frontier by.

Result: The optimal path to a goal or *failure* if no node passes the goal test.

$g(s) \leftarrow 0$

// $g(node)$ is the cost of the path from s to the node.

frontier $\leftarrow$ a min-priority queue ordered by $\boldsymbol{f}$, containing only s

reached $\leftarrow$ a dictionary of the form {state: node}, containing only $s.state{:}s$

while *frontier isn't empty* **do**

  $v \leftarrow$ pop the the min-f node from *frontier*

  if $goal(v) = true$ **then**

    return v

  end

  $reached \leftarrow expanded \cup \{v.state\}$

  for $w \in EXPAND(v)$ **do**

    if $w.state \notin reached$ *or* $g(w) < g\,(reached[w.state])$ **then**

      $reached[w.state] \leftarrow w$

      Insert w into *frontier*

    end

  end

end

return *failure*

Function EXPAND(v)**:**

  for $w \in successors(v)$ **do**

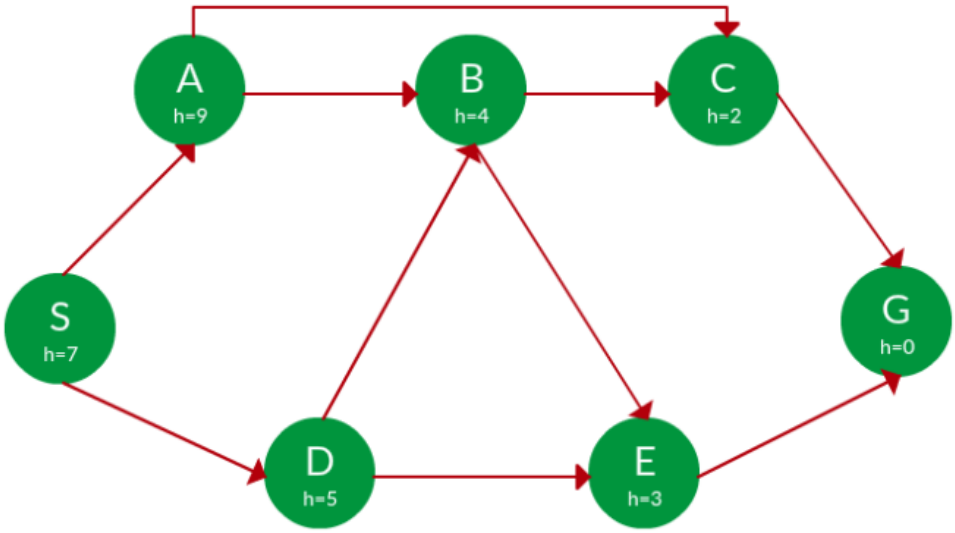
    $g(w) \leftarrow g(v) + c(v, w)$

    yield w

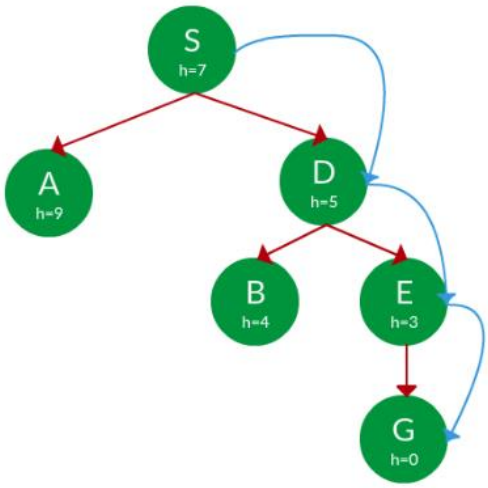
  end

- Đánh giá:

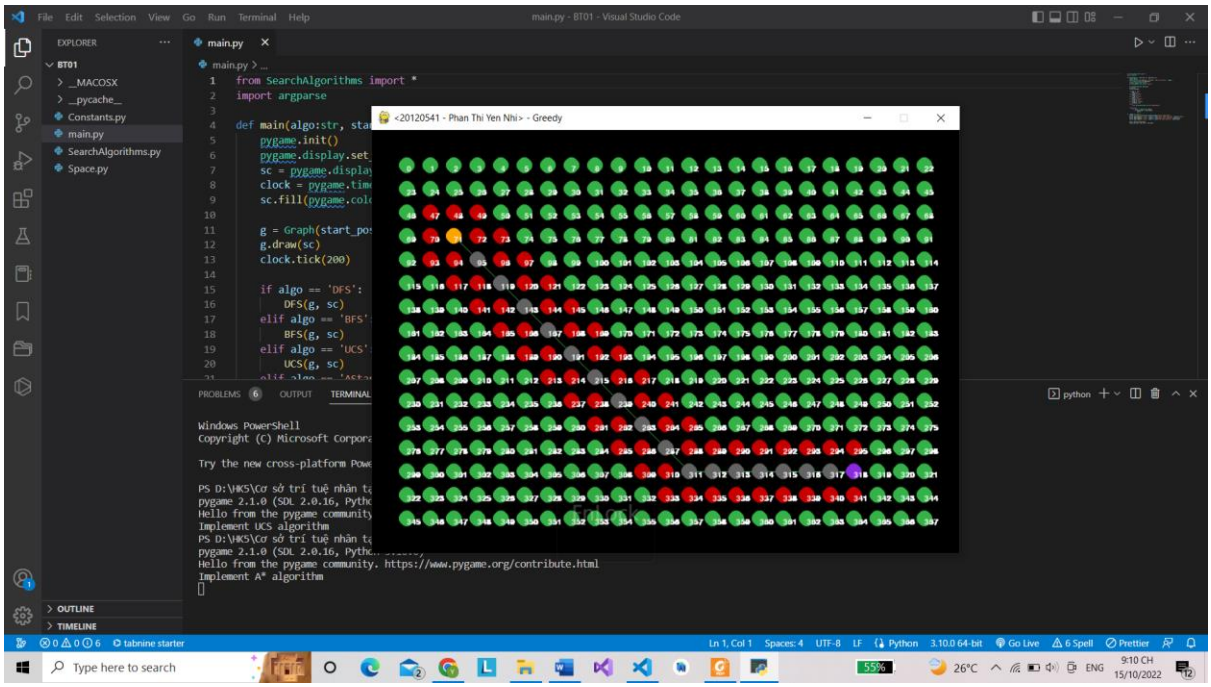
Tính đầy đủ: không đầy đủ. Vì có thể vướng trong các vòng lặp.
Tính tối ưu: Greedy không phải là tối ưu,
Độ phức tạp về thời gian: $O(b^m)$. Một hàm heuristic ốt có thể mang lại cải thiện lớn.
Độ phức tạp về không gian: $O(b^m)$. Lưu tất cả các node trong bộ nhớ.
Ví dụ: Tìm đường đi từ S đến G bằng cách sử dụng Greedy search. Các giá trị heuristic h của mỗi nút bên dưới tên của node.



Bắt đầu từ S, chúng ta có thể đi đến A (h = 9) hoặc D (h = 5). Chúng tôi chọn D, vì nó có chi phí heuristic thấp hơn. Bây giờ từ D, chúng ta có thể di chuyển đến B (h = 4) hoặc E (h = 3). Chúng tôi chọn E với chi phí heuristic thấp hơn. Cuối cùng, từ E, chúng ta đi đến G (h = 0). Toàn bộ truyền tải này được hiển thị trong cây tìm kiếm bên dưới, có màu xanh lam => đường đi: S -> D -> E -> G



- Kết quả chạy thuật toán:



b) Depth limited Search:

- Ý tưởng chung:

Thuật toán tìm kiếm giới hạn theo chiều sâu tương tự như tìm kiếm theo chiều sâu với giới hạn xác định trước. Tìm kiếm giới hạn độ sâu có thể giải quyết nhược điểm của đường dẫn vô hạn trong tìm kiếm theo độ sâu trước tiên. Trong thuật toán này, nút ở giới hạn độ sâu sẽ coi như nó không có nút kế thừa nào nữa.

Tìm kiếm giới hạn độ sâu có thể bị chấm dứt với hai Điều kiện thất bại:

Giá trị lỗi tiêu chuẩn: Nó chỉ ra rằng vấn đề không có bất kỳ giải pháp nào.

Giá trị lỗi cắt: Nó xác định không có giải pháp cho vấn đề trong một giới hạn độ sâu nhất định.

- Mã giả:

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

- Đánh giá:

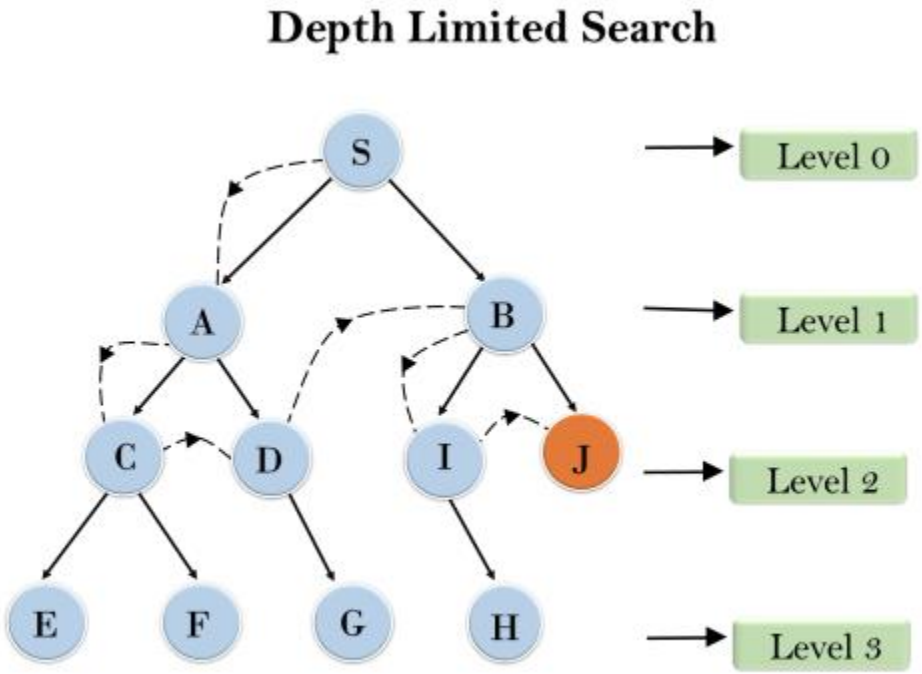
  Tính đầy đủ: Thuật toán tìm kiếm DLS hoàn tất nếu giải pháp nằm trên giới hạn độ sâu.

  Tối ưu: Tìm kiếm giới hạn độ sâu có thể được xem như một trường hợp đặc biệt của DFS và nó cũng không phải là tối ưu ngay cả khi $\ell > d$.

  Độ phức tạp về thời gian: Độ phức tạp về thời gian của thuật toán DLS là $O(b^m)$.

  Độ phức tạp không gian: Độ phức tạp không gian của thuật toán DLS là $O(b * m)$.

- Ví dụ:



c) Interative deepening depth-first Search:

- Ý tưởng chung:

Thuật toán đào sâu lặp đi lặp lại là sự kết hợp của thuật toán DFS và BFS. Thuật toán tìm kiếm này tìm ra giới hạn độ sâu tốt nhất và thực hiện nó bằng cách tăng dần giới hạn cho đến khi tìm thấy mục tiêu.

Thuật toán này thực hiện tìm kiếm theo chiều sâu đến một "giới hạn độ sâu" nhất định và nó tiếp tục tăng giới hạn độ sâu sau mỗi lần lặp cho đến khi tìm thấy nút mục tiêu.

Thuật toán tìm kiếm này kết hợp các lợi ích của tìm kiếm nhanh trên Breadth-first search và hiệu quả bộ nhớ của tìm kiếm theo chiều sâu.

Thuật toán tìm kiếm lặp lại là tìm kiếm không có thông tin hữu ích khi không gian tìm kiếm lớn và độ sâu của nút mục tiêu là không xác định.

- Mã giả:

Algorithm 3: Iterative Deepening

Data: s: the start node, *target*: the function that identifies a target node, *children*: the function that returns the children of a node.
Result: The shortest path between u and a target node
for $\ell = 0, 1, \ldots, \infty$ **do**
  $result \leftarrow$ apply DLDFS with limit depth ℓ
  if $result \neq CUTOFF$ **then**
    return *result*
  end
end

- Đánh giá:

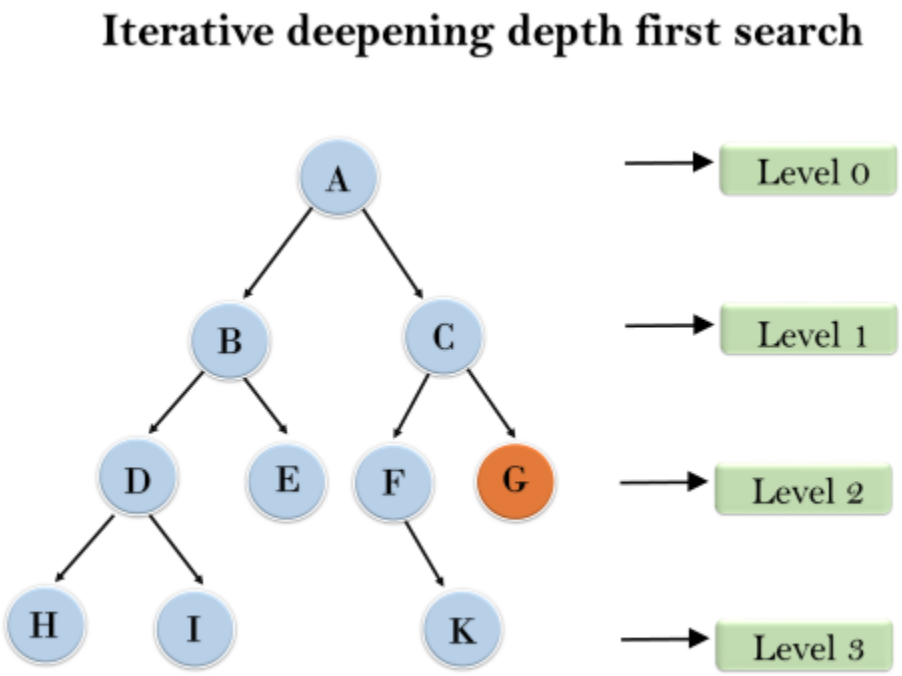
Tính hoàn chỉnh: Thuật toán này hoàn thành nếu hệ số phân nhánh là hữu hạn.

Độ phức tạp về thời gian: Giả sử b là hệ số phân nhánh và độ sâu là d thì độ phức tạp thời gian trong trường hợp xấu nhất là $O(b^m)$.

Độ phức tạp về không gian: Độ phức tạp không gian của IDDFS sẽ là $O(b * d)$

Tối ưu: Thuật toán IDDFS là tối ưu nếu chi phí đường dẫn là một hàm không giảm của độ sâu của nút.

- Ví dụ:



d) Bidirectional Search:

- Ý tưởng chung:

Thuật toán tìm kiếm hai chiều chạy hai tìm kiếm đồng thời, một ở trạng thái ban đầu của biểu mẫu được gọi là tìm kiếm chuyển tiếp và từ nút mục tiêu khác được gọi là tìm kiếm ngược, để tìm nút mục tiêu. Tìm kiếm hai chiều thay thế một biểu đồ tìm kiếm đơn lẻ bằng hai đồ thị con nhỏ, trong đó một đồ thị bắt đầu tìm kiếm từ đỉnh ban đầu và đồ thị khác bắt đầu từ đỉnh mục tiêu. Việc tìm kiếm dừng lại khi hai đồ thị này cắt nhau.

Tìm kiếm hai chiều có thể sử dụng các kỹ thuật tìm kiếm như BFS, DFS, DLS, v.v.

- Mã giả:

Algorithm 1: Bidirectional UCS

Data: s - the start node, t - the goal node, $succ_F(\cdot), succ_R(\cdot)$ - the functions returning the forward and reverse successors of a node, $c(\cdot)$ - the cost function

Result: The optimal path between s and t if it exists, $failure$ otherwise.

$Q_F, Q_R \leftarrow$ make min-priority queues for the frontiers initially containing only s and t

// Q_F is ordered by g_F, Q_R is ordered by g_R

$expanded_F, expanded_R \leftarrow$ make forward and reverse lookup tables

$search_F \leftarrow (Q_F, expanded_F, succ_F)$

$search_R \leftarrow (Q_R, expanded_R, succ_R)$

$solution \leftarrow failure$

$\mu \leftarrow \infty$

while $top_F + top_R < \mu$ **do**

  if *at least one queue is non-empty* **then**

    Choose the search to advance.

  else

    return *solution*

  end

  if *forward search was chosen* **then**

    $(solution, \mu) \leftarrow$ STEP$(search_F, search_R, solution, \mu)$

  else

    $(solution, \mu) \leftarrow$ STEP$(search_R, search_F, solution, \mu)$

  end

end

return *solution*

```
function STEP(search_1, search_2, solution, μ) :
    // 1 denotes the chosen and 2 the other search
       direction
    u ← pop the min-g_1 node from Q_1
    for v ∈ succ_1(u) do
        if v ∉ expanded_1 ∪ Q_1 or g_1(u) + c(u, v) < g_1(v) then
            g_1(v) ← g_1(u) + c(u, v)
            Add v to Q_1
            if v ∈ expanded_2 and g_1(v) + g_2(v) < μ then
                solution ← reconstruct the path through u and v
                μ ← g_1(v) + g_2(v)
            end
        end
    end
    return (solution, μ)
end
```

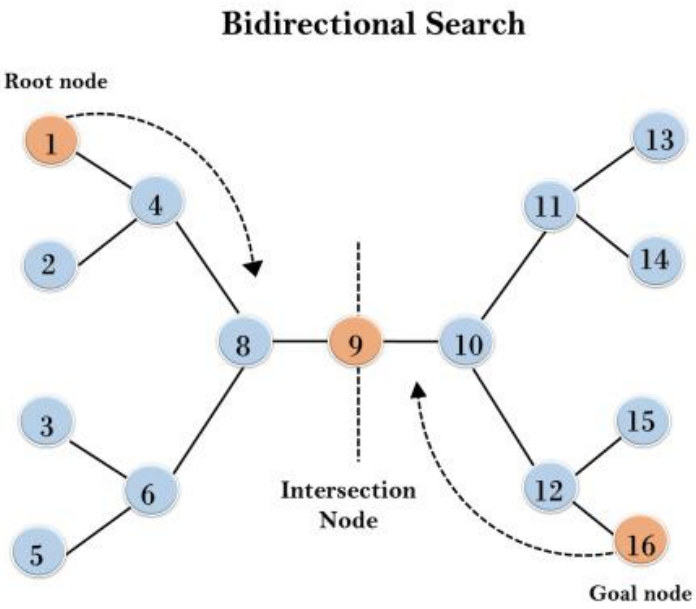
- Đánh giá:

  Tính đầy đủ: Tìm kiếm hai chiều sẽ hoàn tất nếu chúng tôi sử dụng BFS trong cả hai tìm kiếm.

  Độ phức tạp về thời gian: Độ phức tạp về thời gian của tìm kiếm hai chiều sử dụng BFS là $O(b^m)$.

  Độ phức tạp về không gian: Độ phức tạp về không gian của tìm kiếm hai chiều là $O(b * d)$.

  Tối ưu: Tìm kiếm hai chiều là Tối ưu.
- Ví dụ:



Đánh giá:

	Mức độ hoàn thành	Điểm
Tìm hiểu và trình bày được các thuật toán tìm kiếm trên đồ thị	Hoàn thành	9
So sánh các thuật toán.	Hoàn thành	9
Cài đặt được các thuật toán	Hoàn thành	9.5
Tìm hiểu thêm các thuật toán tìm kiếm.	Hoàn thành	9

- Trung bình: $\frac{9*4+9*2+9.5*3+9*1}{10} = 9.15$

Nguồn:

http://www4.hcmut.edu.vn/~huynhqlinh/TinhocDC/THDC13/Bai03_6.htm
https://www.geeksforgeeks.org/search-algorithms-in-ai/
https://vi.wikipedia.org/wiki/T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u
https://websitehcm.com/cac-thuat-toan-informed-search-algorithms/
https://www.geeksforgeeks.org/difference-between-best-first-search-and-a-search/#:~:text=Best%2Dfirst%20search%20is%20not,A*%20search%20is%20complete.
https://www.baeldung.com/cs/uniform-cost-search-vs-dijkstras
https://www.javatpoint.com/ai-uninformed-search-algorithms
https://www.baeldung.com/cs/find-path-uniform-cost-search
https://www.javatpoint.com/ai-uninformed-search-algorithms
https://huuvinhfit.files.wordpress.com/2015/01/chuong-3-tim-kiem-co-ban.pdf
https://viblo.asia/p/cac-giai-thuat-tim-kiem-tren-do-thi-1Je5EBRGKnL
https://vnoi.info/wiki/algo/graph-theory/breadth-first-search.md#%C4%91%C3%A1nh-gi%C3%A1-7
https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/a-star-algorithm
https://websitehcm.com/thuat-toan-tim-kiem-uninformed-search-algorithms/
Slide Cơ sở dữ liệu của thầy Nguyễn Ngọc Đức (GVLT)