

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN



BÁO CÁO MÔN CƠ SỞ TRÍ TUỆ NHÂN TẠO

NỘI DUNG: SEARCH ALGORITHMS

Thực hiện: Lương Văn Triều

Giáo viên hướng dẫn: Thầy Nguyễn Bảo Long

TP Hồ Chí Minh, ngày 16 tháng 10 năm 2022

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN



BÁO CÁO MÔN CƠ SỞ TRÍ TUỆ NHÂN TẠO
NỘI DUNG: SEARCH ALGORITHMS

TP Hồ Chí Minh, ngày 16 tháng 10 năm 2022

MỤC LỤC

PHẦN NỘI DUNG	4
I. Tìm hiểu và trình bày thuật toán	4
1. Các thành phần của một bài toán tìm kiếm.	4
2. Cách giải một bài toán tìm kiếm nói chung.	5
3. Phân loại bài toán tìm kiếm (Informed Search và Uninformed Search)	5
II. Trình bày về 4 thuật toán DFS, BFS, UCS, A*	7
1. DFS	7
2. BFS	9
3. UCS	11
4. A*	14
5. Bellman – Ford	17
6. Dijkstra	19
III. So sánh các thuật toán.	22
1. So sánh sự khác biệt giữa UCS, Greedy và A*	22
2. So sánh sự khác biệt giữa UCS và Dijkstra.	23
IV. Cài Đặt Thuật Toán.	24
1. Thuật toán DFS.	24
2. Thuật toán BFS.	25
3. Thuật toán UCS	27
4. Thuật toán A*	28
V. Đánh giá thang điểm.	29
1. Tìm hiểu và trình bày các thuật toán	29
2. So sánh các thuật toán với nhau.	29
3. Cài đặt thuật toán.	29
4. Tìm hiểu các thuật toán tìm kiếm bên ngoài yêu cầu.	29
TÀI LIỆU THAM KHẢO	31

PHẦN NỘI DUNG

I. Tìm hiểu và trình bày thuật toán

1. Các thành phần của một bài toán tìm kiếm.

- Trạng thái ban đầu (Initial state)
- Mô tả các hoạt động (action) có thể thực hiện
- Mô hình di chuyển (transition model): mô tả kết quả của các hành động
 - + thuật ngữ successor tương ứng với các trạng thái có thể di chuyển được với một hành động duy nhất
 - + Trạng thái bắt đầu, hành động và mô hình di chuyển định nghĩa không gian trạng thái (state space) của bài toán
 - + Không gian trạng thái hình thành nên một đồ thị có hướng với đỉnh là các trạng thái và cạnh là các hành động
 - + Một đường đi trong không gian trạng thái là một chuỗi các trạng thái được kết nối bằng một chuỗi các hành động
- 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.

2. Cách giải một bài toán tìm kiếm nói chung.

- Tìm kiếm là quy trình tìm chuỗi hành động để tới được trạng thái đích.
- Một thuật toán tìm kiếm nhận một bài toán là input và trả về kết quả là lời giải dưới dạng chuỗi hành động.
- Thực thi: Khi đã tìm thấy lời giải, thực hiện các hành động, khi thực hiện lời giải, không nhận tín hiệu trả về từ các hành động ta cần hệ thống open – loop.
- Mã giả:

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
persistent: 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
    goal ← FORMULATE-GOAL(state)
    problem ← FORMULATE-PROBLEM(state, goal)
    seq ← SEARCH(problem)
    if seq = failure then return a null action
action ← FIRST(seq)
seq ← REST(seq)
return action
```

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

Thông số	Informed Search	Uninformed Search
Định nghĩa	Được gọi là tìm kiếm theo phương pháp Heuristic	Được gọi là tìm kiếm mù

Sử dụng kinh nghiệm	Tìm kiếm dựa trên kinh nghiệm	Không sử dụng kinh nghiệm trong quá trình tìm kiếm
Mức độ tìm thấy giải pháp	Tìm thấy giải pháp nhanh chóng hơn	Tìm thấy giải pháp chậm hơn so với Informed Search
Hoàn thành	Nó có thể đầy đủ hoặc không	Nó luôn luôn hoàn thành
Chi phí	Chi phí thấp	Chi phí cao
Thời gian thực hiện	Nó tiêu tốn ít thời gian hơn vì nó tìm kiếm nhanh chóng hơn	Nó tiêu tốn thời gian vừa phải vì nó tìm kiếm chậm
Hướng đi	Có hướng đi được đưa ra về giải pháp	Không có đề xuất nào được đưa ra liên quan đến giải pháp trong đó
Thực hiện	Ít dài dòng hơn trong khi thực hiện	Dài dòng hơn trong khi thực hiện
Mức độ hiệu quả	Nó hiệu quả hơn vì hiệu quả có tính đến chi phí và hiệu suất. Chi phí ít hơn và giải pháp tìm ra nhanh chóng hơn	Nó tương đối kém hiệu quả hơn vì chi phí nhiều hơn và độ độ tìm kiếm chậm hơn
Yêu cầu về tính toán	Các yêu cầu tính toán được giảm bớt	Yêu cầu tính toán tương đối cao hơn
Quy mô	Có phạm vi rộng trong việc xử lý các vấn đề tìm kiếm lớn	Giải quyết một nhiệm vụ tìm kiếm lớn là một thách thức

Các ví dụ về thuật toán	+ Greedy Search + A* Search + Graph Search	+ Depth First Search + Breadth First Search + Uniform Cost Search
--------------------------------	--	---

II. Trình bày về 4 thuật toán DFS, BFS, UCS, A*.

1. DFS

- Ý tưởng chung.

Từ đỉnh (nút) gốc ban đầu, thuật toán duyệt đi xa nhất theo từng nhánh, khi nhánh đã duyệt hết, lùi về từng đỉnh để tìm và duyệt những nhánh tiếp theo. Quá trình duyệt chỉ dừng lại khi tìm thấy đỉnh cần tìm hoặc tất cả đỉnh đều đã được duyệt qua.

- Mã giả.

```

DFS(G,v)  ( v is the vertex where the search starts )
  Stack S := {};  ( start with an empty stack )
  for each vertex u, set visited[u] := false;
  push S, v;
  while (S is not empty) do
    u := pop S;
    if (not visited[u]) then
      visited[u] := true;
      for each unvisited neighbour w of u
        push S, w;
      end if
    end while
  END DFS()

```

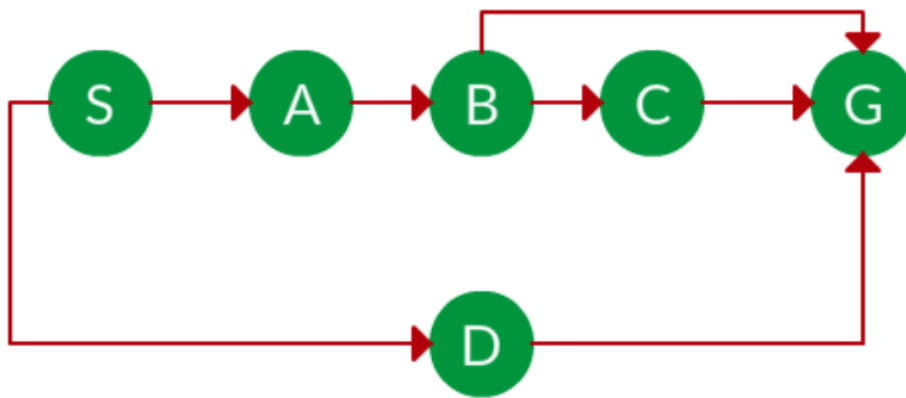
- Đánh giá về thuật toán.

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

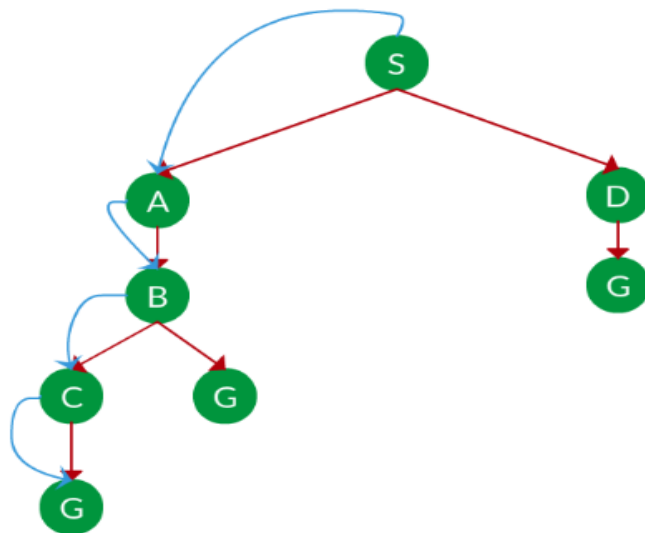
+ Tối ưu: DFS không phải là tối ưu, có nghĩa là số bước để đạt được giải pháp hoặc chi phí bỏ ra để đạt được nó là cao.

+ Độ phức tạp: Về thời gian là $O(n^d)$ với d là độ sâu của cây tìm kiếm và n là số trong cấp độ, về không gian là $O(n \cdot d)$.

- **Ví dụ:** Dùng DFS tìm đường đi từ đỉnh S đến đỉnh G trong đồ thị dưới đây.



Khi DFS đi ngang qua cây "nút sâu nhất trước", nó sẽ luôn chọn nhánh sâu hơn cho đến khi đạt được giải pháp (hoặc nó hết nút và chuyển sang nhánh tiếp theo). Đường đi được hiển thị bằng các mũi tên màu xanh dương.



Kết quả: {S, A, B, C, G}.

2. BFS

- Ý tưởng chung.

Với đồ thị không trọng số và đỉnh nguồn s. 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 đầu, 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.

Thuật toán

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

Bước 1: Khởi tạo

Các đỉnh đều ở trạng thái chưa được đánh dấu. Ngoại trừ đỉnh nguồn s đã được đánh dấu. Một hàng đợi ban đầu chỉ chứa 1 phần tử là s.

Bước 2: Lặp lại các bước sau cho đến khi hàng đợi rỗng:

Lấy đỉnh u ra khỏi hàng đợi.

Xét tất cả những đỉnh v kề với u mà chưa được đánh dấu, với mỗi đỉnh v đó:

Đánh dấu v đã thăm.

Lưu lại vết đường đi từ u đến v.

Đẩy v vào trong hàng đợi (đỉnh v sẽ chờ được duyệt tại những bước sau).

Bước 3: Truy vết tìm đường đi.

- **Mã giả.**

```
BFS (G, s)                                //Where G is the graph and s is the source node
    let Q be queue.
    Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.

    mark s as visited.
    while ( Q is not empty)
        //Removing that vertex from queue,whose neighbour will be visited now
        v = Q.dequeue( )

        //processing all the neighbours of v
        for all neighbours w of v in Graph G
            if w is not visited
                Q.enqueue( w )           //Stores w in Q to further visit its neighbour
                mark w as visited.
```

- **Đánh giá về thuật toán.**

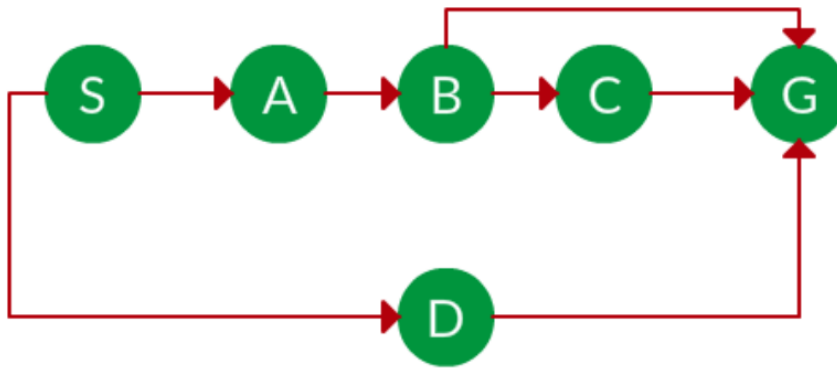
+ Tính đầy đủ: Có nếu cây tìm kiếm là hữu hạn.

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

+ Độ phức tạp: Về thời gian là $O(n^s)$ với s là độ sâu cây, và n là số nút trong cấp độ, Về không gian thì nó tương đương với độ lớn của rìa có thể nhận được là $O(n^s)$.

- **Ví dụ.**

Dùng BFS tìm đường đi từ S đến G trong đồ thị dưới đây



Khi BFS đi ngang qua cây "nút nông nhất trước", nó sẽ luôn chọn nhánh nông hơn cho đến khi nó đạt được giải pháp (hoặc nó hết nút và chuyển sang nhánh tiếp theo). Đường ngang được hiển thị bằng các mũi tên màu xanh dương.

Kết quả: {S, D, G}.

3. UCS

- Ý tưởng chung.

Tìm kiếm chi phí thống nhất là một biến thể của thuật toán Dijkstra. Ở đây, thay vì chèn tất cả các đỉnh vào hàng đợi ưu tiên, ta chỉ chèn nguồn, sau đó chèn từng đỉnh khi cần thiết.

Trong mỗi bước, ta kiểm tra xem mục đã ở trong hàng đợi ưu tiên chưa (sử dụng mảng đã truy cập). Nếu có, ta thực hiện giảm, nếu không, ta chèn nó.

Biến thể này của Dijkstra hữu ích cho các đồ thị vô hạn và những đồ thị quá lớn để biểu diễn trong bộ nhớ. Tìm kiếm thống nhất-chi phí chủ yếu được sử dụng trong Trí tuệ nhân tạo.

Thuật toán:

Trong thuật toán này từ trạng thái bắt đầu, ta sẽ truy cập các vùng lân cận và sẽ chọn trạng thái ít tốn kém nhất, sau đó ta sẽ chọn trạng thái ít tốn kém nhất tiếp theo từ tất cả các trạng thái chưa được truy cập và lân cận của các vùng đã truy cập, theo cách

này, ta sẽ cố gắng đạt được trạng thái mục tiêu (lưu ý rằng ta sẽ không tiếp tục con đường qua trạng thái mục tiêu), ngay cả khi ta đạt được trạng thái mục tiêu, ta sẽ tiếp tục tìm kiếm các con đường khả thi khác (nếu có nhiều mục tiêu). Ta sẽ giữ một hàng đợi ưu tiên sẽ cung cấp trạng thái tiếp theo ít tốn kém nhất từ tất cả các trạng thái lân cận của các vùng đã truy cập.

- Mã giả.

```
begin
procedure UniformCostSearch(Graph, root, goal)
  node:= root, cost = 0
  frontier:= priority queue containing node only
  explored:= empty set
do
  if frontier is empty
    return failure
  node:= frontier.pop()
  if node is goal
    return solution
  explored.add(node)
  for each of node's neighbors n
    if n is not in explored
      if n is not in frontier
        frontier.add(n)
      else if n is in frontier with higher cost
        replace existing node with n
```

- Đánh giá về thuật toán.

+ Tính đầy đủ: Đầy đủ, chẳng hạn như nếu có giải pháp thì UCS sẽ tìm ra giải pháp đó.

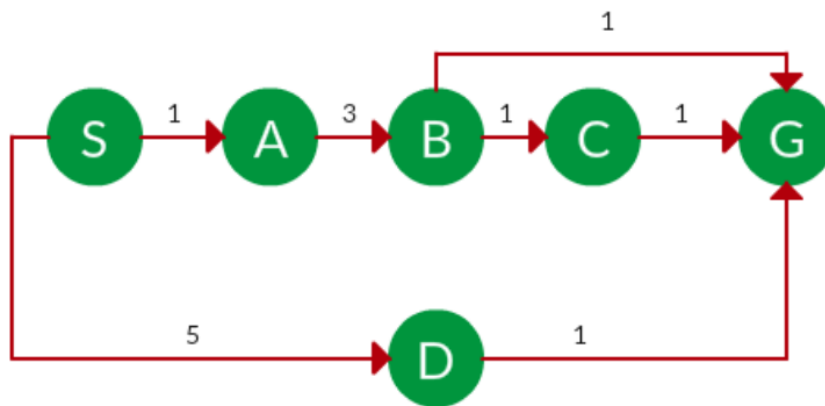
+ Tính tối ưu: Tìm kiếm theo chi phí tổng quát luôn tối ưu vì nó chỉ chọn một đường dẫn có chi phí đường dẫn thấp nhất.

+ 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 nút mục tiêu. Khi đó số bước là $\lceil C^* / \epsilon \rceil + 1$. Ở đây đã lấy +1, khi bắt đầu từ trạng thái 0 và kết thúc đến C^* / ϵ . Do đó, độ phức tạp thời gian trong trường hợp xấu nhất của tìm kiếm theo chi phí tổng quát là $O(b^{1 + \lceil C^* / \epsilon \rceil})$, Về

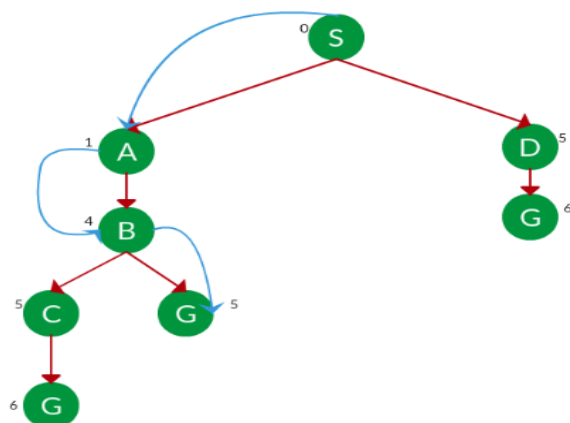
không gian Logic tương tự là đối với độ phức tạp không gian, vì vậy, độ phức tạp không gian trong trường hợp xấu nhất của Tìm kiếm theo chi phí thống nhất là $O(b^l + [C * \epsilon])$.

- **Ví dụ.**

Cho đồ thị như bên dưới, hãy tìm đường đi từ S đến G bằng thuật toán UCS.



Dựa trên chiến lược UCS, con đường có chi phí tích lũy ít nhất được chọn. Lưu ý rằng do có nhiều tùy chọn ở phần rìa, thuật toán khám phá hầu hết chúng miễn là chi phí của chúng thấp và loại bỏ chúng khi tìm thấy đường dẫn chi phí thấp hơn; những đường dẫn bị loại bỏ này không được hiển thị bên dưới. Đường đi thực tế được hiển thị bằng đường màu xanh dương.



Kết quả: {S, A, B, G}.

Chi phí: 5.

4. A*

- Ý tưởng chung.

Xét bài toán tìm đường – bài toán mà A* thường được dùng để giải. A* xây dựng tăng dần tất cả các tuyến đường từ điểm 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 (informed tìm kiếm thuật toán), 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 điểm bất kỳ cho trước tới đích. Trong trường hợp tìm đường đi, đánh giá này có thể là khoảng cách đường chim bay – một đánh giá xấp xỉ thường dùng cho khoảng cách của đường giao thông.

Điểm khác biệt của A* đối với tìm kiếm theo lựa chọn tốt nhất là nó còn tính đến khoảng cách đã đi qua. Điều đó làm cho A* “đầy đủ” và “tối ưu”, nghĩa là, A* sẽ luôn luôn tìm thấy đường đi ngắn nhất nếu tồn tại một đường đi như vậy. A* không đảm bảo sẽ chạy nhanh hơn các thuật toán tìm kiếm đơn giản hơn. Trong một môi trường dạng mê cung, cách duy nhất để đến đích có thể là trước hết phải đi về phía xa đích và cuối cùng mới quay lại. Trong trường hợp đó, việc thử các nút theo thứ tự “gần đích hơn thì được thử trước” có thể gây tốn thời gian.

- Mã giả.

```

function A*(điểm_xuất_phát,đích)
    var đóng:= tập_rỗng
    var q:= tạo_hàng_đợi(tạo_đường_đi(điểm_xuất_phát))
    while q không_phải_tập_rỗng
        var p:= lấy_phần_tử_đầu_tiên(q)
        var x:= nút_cuối_cùng_của_p
        if x in đóng
            continue
        if x = đích
            return p
        bổ_sung_x_vào_tập_đóng
        foreach y in các_đường_đi_tiếp_theo(p)
            đưa_vào_hàng_đợi(q, y)
    return failure

```

- Đánh giá về thuật toán.

Với e là chi phí di chuyển thấp nhất và C^* là chi phí lời giải tối ưu.

+ Tính đầy đủ: Có nếu $e > 0$ và không gian trạng thái hữu hạn.

+ Tính tối ưu: Có nếu heuristic hợp lý (không ước lượng chi phí quá cao đến đích, $h(n) \leq h^*(n)$ chi phí thấp nhất từ n đến đích) và nhất quán ($h(n)$ là một heuristic nhất quán nếu với mỗi succesor n' của n , khoảng cách ước tính đến đích từ n không lớn hơn khoảng cách đến đích ước tính từ n' cộng với chi phí di chuyển từ n đến n' : $h(n) \leq c(n, n') + h(n')$)

Một heuristic nhất quán cũng là một heuristic hợp lý.

+ Độ phức tạp: Thời gian là cấp số mũ và không gian cũng là cấp số mũ.

- Đề xuất heuristic và giải thích tại sao chọn nó. Như thế nào là một heuristic chấp nhận được.

+ Đề xuất heuristic và giải thích lý do: Tìm kiếm đồ thị chỉ tối ưu khi chi phí chuyển tiếp giữa hai nút liên tiếp A và B , được cho bởi $h(A) - h(B)$, nhỏ hơn hoặc bằng chi

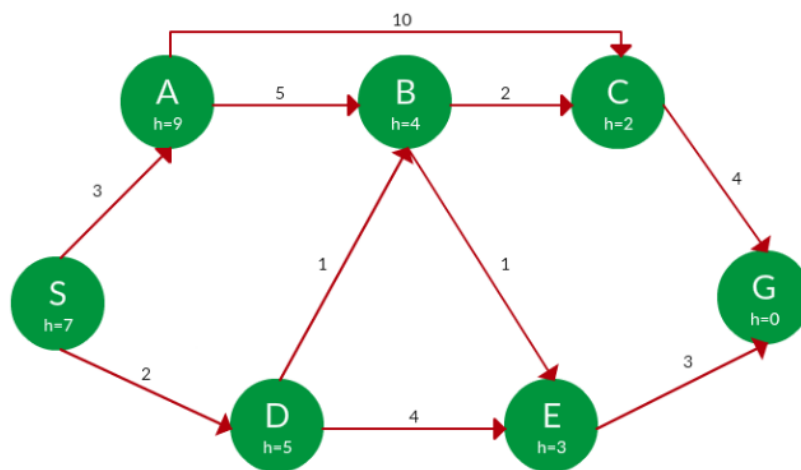
phí lùi giữa hai nút g (A đến B). Tính chất này của phương pháp heuristic tìm kiếm đồ thị được gọi là tính nhất quán.

+ Một ước lượng (heuristic) $h(n)$ được xem là chấp nhận được nếu đối với mọi nút n: $0 \leq h(n) \leq h^*$ được nếu đối với mọi nút n: $0 \leq h(n) \leq h(n)$, trong đó $h^*(n)$ là chi phí thật (thực tế) để đi từ nút n đến đích.

Một ước lượng chấp nhận được không bao giờ đánh giá quá cao (overestimate) đối với chi phí để đi tới đích.

- Ví dụ.

Cho đồ thị bên dưới dùng thuật toán A* để tìm đường đi từ S đến G.



Bắt đầu từ S, thuật toán tính $g(x) + h(x)$ cho tất cả các nút ở rìa ở mỗi bước, chọn nút có tổng thấp nhất. Toàn bộ quy trình được hiển thị trong bảng dưới đây. Lưu ý rằng trong tập lặp thứ tư, chúng ta nhận được hai đường dẫn có tổng chi phí $f(x)$ bằng nhau, vì vậy chúng ta mở rộng chúng trong tập tiếp theo. Con đường có chi phí thấp hơn khi mở rộng hơn nữa là con đường đã chọn.

Path		$h(x)$	$g(x)$	$f(x)$
S		7	0	7
S -> A		9	3	12
S -> D	✓	5	2	7
S -> D -> B	✓	4	$2 + 1 = 3$	7
S -> D -> E		3	$2 + 4 = 6$	9
S -> D -> B -> C	✓	2	$3 + 2 = 5$	7
S -> D -> B -> E	✓	3	$3 + 1 = 4$	7
S -> D -> B -> C -> G		0	$5 + 4 = 9$	9
S -> D -> B -> E -> G	✓	0	$4 + 3 = 7$	7

Kết quả: {S, D, B, E, G}.

Chi phí: 7.

5. Bellman – Ford

- Ý tưởng chung:

+ Ta thực hiện duyệt n lần, với n là số đỉnh của đồ thị.

+ Với mỗi lần duyệt, ta tìm tất cả các cạnh mà đường đi qua cạnh đó sẽ rút ngắn đường đi ngắn nhất từ đỉnh gốc tới đỉnh khác.

+ Ở lần duyệt thứ n, nếu còn bất kỳ cạnh nào có thể rút ngắn đường đi, điều đó chứng tỏ đồ thị có chu trình âm và ta kết thúc thuật toán.

- Mã giả:

```

function bellmanFord(G, S)
  for each vertex V in G
    distance[V] <- infinite
    previous[V] <- NULL
  distance[S] <- 0

  for each vertex V in G
    for each edge (U,V) in G
      tempDistance <- distance[U] + edge_weight(U, V)
      if tempDistance < distance[V]
        distance[V] <- tempDistance
        previous[V] <- U

  for each edge (U,V) in G
    If distance[U] + edge_weight(U, V) < distance[V]
      Error: Negative Cycle Exists

  return distance[], previous[]

```

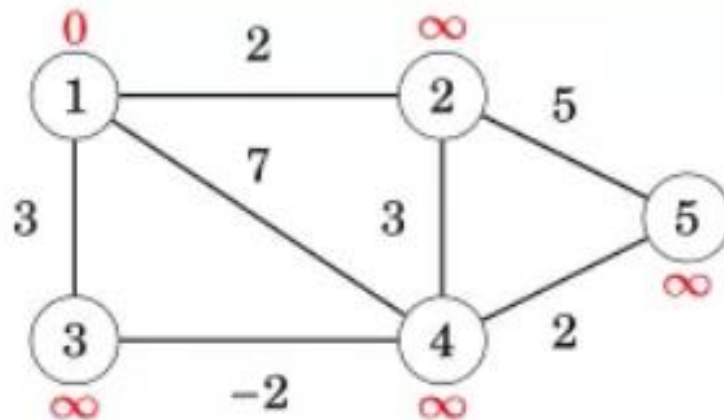
- **Đánh giá thuật toán:**

+ **Tính tối ưu:** Có tối ưu.

+ **Tính đầy đủ:** Có đầy đủ.

+ **Độ phức tạp:** Về thời gian thì trường hợp tốt nhất là $O(E)$ và xấu nhất là $O(VE)$, với E là số cạnh và V là số đỉnh. Về không gian là $O(V)$.

- **Ví dụ:** Tìm đường đi ngắn nhất từ node 1 đến các nốt còn lại.



Đầu tiên ta khởi tạo khoảng cách từ node 1 đến các node còn lại là vô cùng và từ node 1 đến chính nó là 0.

Ta thực hiện 4 vòng lặp.

Vòng lặp đầu tiên, ta cập nhật đường đi ngắn nhất thông qua các cạnh (1,2); (1,3); (1,4).

Vòng lặp thứ 2, cạnh (2,5) và (3,4) là các cạnh tối ưu.

Vòng lặp thứ 3, ta chỉ thấy cạnh (4,5) cải thiện đường đi từ 1 tới 5.

Ở vòng lặp thứ 4, ta nhận thấy không còn cạnh nào có thể tối ưu được nữa. Suy ra ta kết thúc thuật toán.

Kết quả thu được:

+ Từ node 1 đến 2: $1 \Rightarrow 2$.

+ Từ node 1 đến 3: $1 \Rightarrow 3$.

+ Từ node 1 đến 4: $1 \Rightarrow 3 \Rightarrow 4$.

+ Từ node 1 đến 5: $1 \Rightarrow 3 \Rightarrow 4 \Rightarrow 5$.

6. Dijkstra

- Ý tưởng chung.

Giống như thuật toán Bellman-Ford, thuật toán Dijkstra cũng tối ưu hóa đường đi bằng cách xét các cạnh (u,v), so sánh hai đường đi $S \rightarrow v$ sẵn có với đường đi $S \rightarrow u \rightarrow v$.

Thuật toán hoạt động bằng cách duy trì một tập hợp các đỉnh trong đó ta đã biết chắc chắn đường đi ngắn nhất. Mỗi bước, thuật toán sẽ chọn ra một đỉnh u mà chắc chắn sẽ không thể tối ưu hơn nữa, sau đó tiến hành tối ưu các đỉnh v khác dựa trên các cạnh (u,v) đi ra từ đỉnh u . Sau N bước, tất cả các đỉnh đều sẽ được chọn, và mọi đường đi tìm được sẽ là ngắn nhất.

Cụ thể hơn, thuật toán sẽ duy trì đường đi ngắn nhất đến tất cả các đỉnh. Ở mỗi bước, chọn đường đi $S \rightarrow u$ có tổng trọng số nhỏ nhất trong tất cả các đường đi đang được duy trì. Sau đó tiến hành tối ưu các đường đi $S \rightarrow v$ bằng cách thử kéo dài thành $S \rightarrow u \rightarrow v$ như đã mô tả trên.

- Mã giả.

```
function Dijkstra(Graph, source):
    for each vertex v in Graph:                // Initialization
        dist[v] := infinity                    // initial distance from source to vertex v is set to infinite
        previous[v] := undefined               // Previous node in optimal path from source
    dist[source] := 0                          // Distance from source to source
    Q := the set of all nodes in Graph          // all nodes in the graph are unoptimized - thus are in Q
    while Q is not empty:                      // main loop
        u := node in Q with smallest dist[ ]
        remove u from Q
        for each neighbor v of u:              // where v has not yet been removed from Q.
            alt := dist[u] + dist_between(u, v)
            if alt < dist[v]                    // Relax (u,v)
                dist[v] := alt
                previous[v] := u
    return previous[ ]
```

- Đánh giá thuật toán.

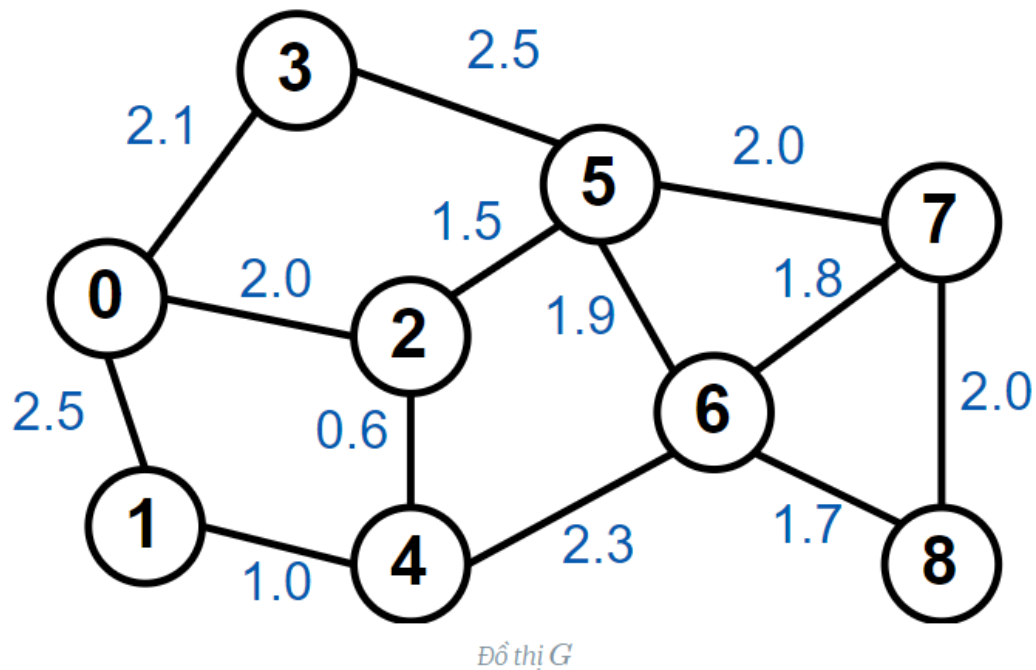
+ Tính đầy đủ: Đầy đủ

+ Tính tối ưu: Có

+ Độ phức tạp: Về thời gian là $O(E \log V)$ với E là số cạnh và V là số đỉnh. Về không gian là $O(V)$.

- Ví dụ.

Cho đồ thị vô hướng G , tìm khoảng cách từ đỉnh 0 đến tất cả các đỉnh còn lại trong đồ thị.



Đầu tiên khởi tạo khoảng cách nhỏ nhất ban đầu tới các đỉnh khác là vô cùng và tới cách tới vị trí gốc là 0. Ta được danh sách sau:

0	1	2	3	4	5	6	7	8
0	$(\infty, -)$	$(\infty, -)$	$(\infty, -)$	$(\infty, -)$	$(\infty, -)$	$(\infty, -)$	$(\infty, -)$	$(\infty, -)$

Chọn đỉnh 0 có giá trị nhỏ nhất, xét các đỉnh kề của đỉnh 0: Xét đỉnh 1, khoảng cách từ gốc đến đỉnh 1 là $2.5 < \infty$ nên ghi nhận giá trị mới là $(2.5, 0)$ (nghĩa là khoảng cách đến đỉnh gốc hiện tại ghi nhận là 2.5, đỉnh kề liền trước là đỉnh 0). Xét tương tự cho đỉnh 2 và 3, ta được dòng thứ 2 trong bảng.

Làm các bước tương tự như vậy cuối cùng ta được.

0	1	2	3	4	5	6	7	8
0	$(\infty, -)$	$(\infty, -)$	$(\infty, -)$	$(\infty, -)$	$(\infty, -)$	$(\infty, -)$	$(\infty, -)$	$(\infty, -)$
–	$(2.5, 0)$	$(2.0, 0)$	$(2.1, 0)$	$(\infty, -)$	$(\infty, -)$	$(\infty, -)$	$(\infty, -)$	$(\infty, -)$
–	$(2.5, 0)$	–	$(2.1, 0)$	$(2.6, 2)$	$(3.5, 2)$	$(\infty, -)$	$(\infty, -)$	$(\infty, -)$
–	$(2.5, 0)$	–	–	$(2.6, 2)$	$(3.5, 2)$	$(\infty, -)$	$(\infty, -)$	$(\infty, -)$
–	–	–	–	$(2.6, 2)$	$(3.5, 2)$	$(\infty, -)$	$(\infty, -)$	$(\infty, -)$
–	–	–	–	–	$(3.5, 2)$	$(4.9, 4)$	$(\infty, -)$	$(\infty, -)$
–	–	–	–	–	–	$(4.9, 4)$	$(5.5, 5)$	$(\infty, -)$
–	–	–	–	–	–	–	$(5.5, 5)$	$(6.6, 6)$
–	–	–	–	–	–	–	–	$(6.6, 6)$

Thuật toán kết thúc khi chọn được khoảng cách nhỏ nhất cho tất cả các đỉnh.

III. So sánh các thuật toán.

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

Điểm khác biệt	UCS	Greedy	A*
Loại	Tìm kiếm mù	Tìm kiếm có kinh nghiệm	Tìm kiếm có kinh nghiệm
Tính tối ưu	Không tối ưu bằng Greedy và A*	Tối ưu nhưng không bằng A*	Tối ưu nhất
Không gian bộ nhớ	Sử dụng ít nhất không gian bộ nhớ	Sử dụng bộ nhớ ít hơn A*	Sử dụng nhiều bộ nhớ nhất
Thời gian thực hiện	Trường hợp xấu nhất là $O(b^1 + \lceil C^*/\epsilon \rceil)$	$O(b^m)$ thực thi nhanh hay chậm còn tùy thuộc vào	Cấp số mũ, phụ thuộc vào heuristic

		heuristic có đủ tốt hay là không	
Tính đầy đủ	Có	Có	Phụ thuộc vào heuristic

2. So sánh sự khác biệt giữa UCS và Dijkstra.

- + Sự khác biệt chính giữa hai thuật toán này là cách lưu trữ các đỉnh Q. Trong thuật toán Dijkstra, ta khởi tạo Q với tất cả các đỉnh trong G. Do đó, thuật toán Dijkstra chỉ có thể áp dụng cho các đồ thị rõ ràng mà chúng ta biết tất cả các đỉnh và cạnh.
- + Tuy nhiên, thuật toán tìm kiếm chi phí đồng nhất bắt đầu với đỉnh nguồn và dần dần đi qua các phần đồ thị cần thiết. Do đó, nó có thể áp dụng cho cả đồ thị tường minh và đồ thị không tường minh.
- + Đối với bài toán đường đi ngắn nhất một cặp, thuật toán Dijkstra có nhiều yêu cầu về bộ nhớ hơn khi chúng ta lưu toàn bộ đồ thị trong bộ nhớ. Ngược lại, thuật toán tìm kiếm chi phí đồng nhất chỉ lưu trữ đỉnh nguồn ở đầu và ngừng mở rộng khi chúng ta đến đỉnh đích. Do đó, thuật toán tìm kiếm chi phí đồng nhất cuối cùng có thể chỉ lưu trữ một phần đồ thị.
- + Mặc dù cả hai thuật toán đều có cùng độ phức tạp về thời gian đối với bài toán đường đi ngắn nhất một cặp, thuật toán của Dijkstra có thể tốn nhiều thời gian hơn do yêu cầu về bộ nhớ.
- + Khi chúng ta triển khai Q với hàng đợi ưu tiên min-heap, mỗi thao tác hàng đợi sẽ mất $O(\log n)$ thời gian, N số đỉnh ở Q đầu. Thuật toán Dijkstra đặt tất cả các đỉnh vào Q đầu. Đối với một đồ thị lớn, các đỉnh của nó sẽ tạo ra chi phí lớn khi thực hiện các thao tác trên Q.

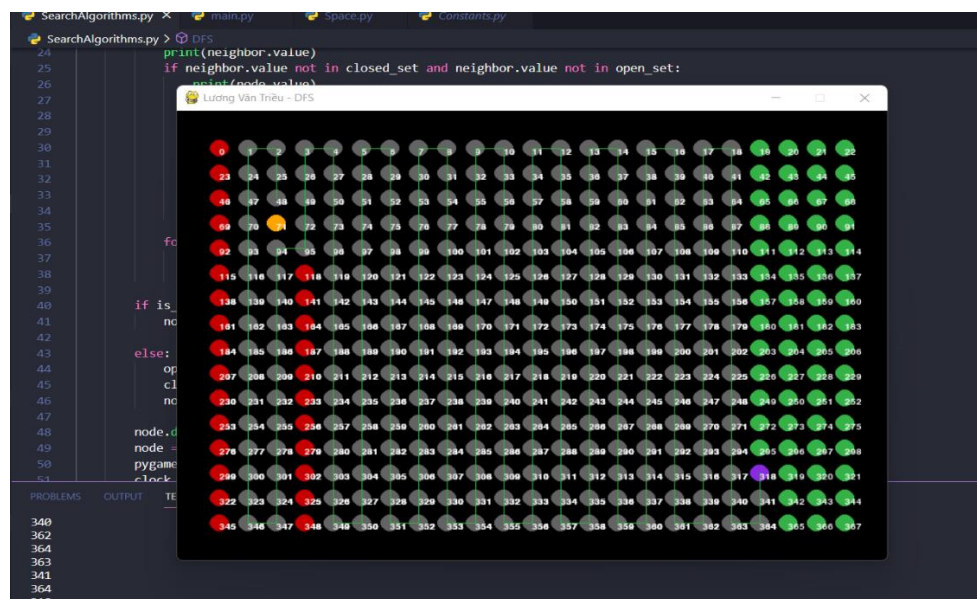
+ Tuy nhiên, thuật toán tìm kiếm chi phí đồng nhất bắt đầu với một đỉnh duy nhất và dần dần bao gồm các đỉnh khác trong quá trình xây dựng đường dẫn. Do đó, chúng ta sẽ làm việc trên một số lượng đỉnh nhỏ hơn nhiều khi chúng ta thực hiện các hoạt động hàng đợi ưu tiên.

+ Một sự khác biệt nhỏ khác giữa hai thuật toán này là các giá trị khoảng cách cuối cùng trên các đỉnh không thể truy cập được từ đỉnh nguồn. Trong thuật toán Dijkstra, nếu không có đường đi giữa đỉnh nguồn S và đỉnh v thì giá trị khoảng cách của nó ($dist[v]$) là dương vô cùng. Tuy nhiên, trong thuật toán tìm kiếm chi phí thống nhất, chúng ta không thể tìm thấy giá trị đó trong bản đồ khoảng cách cuối cùng, tức là $dist[v]$ không tồn tại.

IV. Cài Đặt Thuật Toán.

1. Thuật toán DFS.

- Kết quả sau khi thực hiện.



- Mô tả quá trình tìm kiếm.

Bước 1: Tập Open chứa đỉnh gốc s chờ được xét.

Bước 2: Kiểm tra tập Open có rỗng không.

Nếu tập Open không rỗng, lấy một đỉnh ra khỏi tập Open làm đỉnh đang xét p . Nếu p là đỉnh g cần tìm, kết thúc tìm kiếm. Nếu tập Open rỗng, tiến đến bước 4.

Bước 3: Đưa đỉnh p vào tập Close, sau đó xác định các đỉnh kề với đỉnh p vừa xét. Nếu các đỉnh kề không thuộc tập Close, đưa chúng vào đầu tập Open. Quay lại bước 2.

Bước 4: Kết luận không tìm ra đỉnh đích cần tìm.

Quy ước:

+ Open: là tập hợp các đỉnh chờ được xét ở bước tiếp theo theo ngăn xếp (ngăn xếp: dãy các phần tử mà khi thêm phần tử vào sẽ thêm vào đầu dãy, còn khi lấy phần tử ra sẽ lấy ở phần tử đứng đầu dãy).

+ Close: là tập hợp các đỉnh đã xét, đã duyệt qua.

+ s : là đỉnh xuất phát, đỉnh gốc ban đầu trong quá trình tìm kiếm.

+ g : đỉnh đích cần tìm.

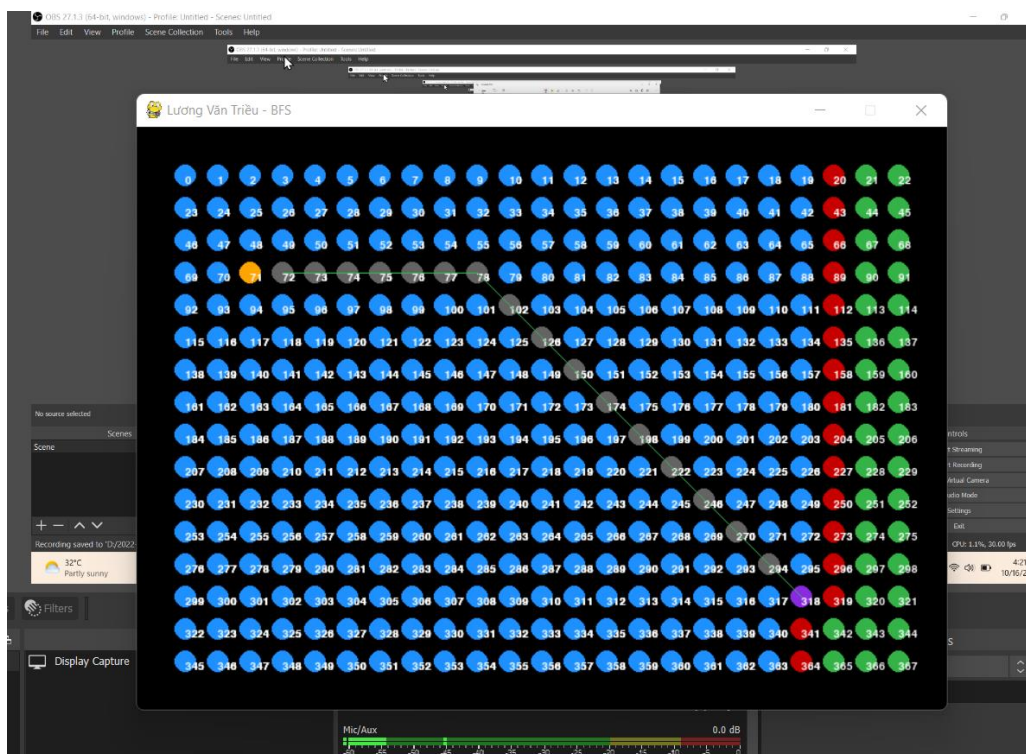
+ p : đỉnh đang xét, đang duyệt.

- **Nhận xét và kết quả thu được.**

Thuật toán sẽ duyệt qua tất cả các đỉnh nếu liên thông. Nó là thuật toán tìm kiếm mù mang tính chất vét cạn và sẽ kém hiệu quả nếu đỉnh quá lớn. Như ta thấy ở ví dụ trên thì nó duyệt qua gần như toàn bộ tập đỉnh, chi phí thời gian là tương đối lớn.

2. Thuật toán BFS.

- **Kết quả sau khi thực hiện.**



- Mô tả quá trình tìm kiếm.

Thuật toán

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

Bước 1: Khởi tạo

Các đỉnh đều ở trạng thái chưa được đánh dấu. Ngoại trừ đỉnh nguồn s đã được đánh dấu. Một hàng đợi ban đầu chỉ chứa 1 phần tử là s .

Bước 2: Lặp lại các bước sau cho đến khi hàng đợi rỗng:

Lấy đỉnh u ra khỏi hàng đợi.

Xét tất cả những đỉnh v kề với u mà chưa được đánh dấu, với mỗi đỉnh v đó:

Đánh dấu v đã thăm.

Lưu lại vết đường đi từ u đến v.

Đẩy v vào trong hàng đợi (đỉnh v sẽ chờ được duyệt tại những bước sau).

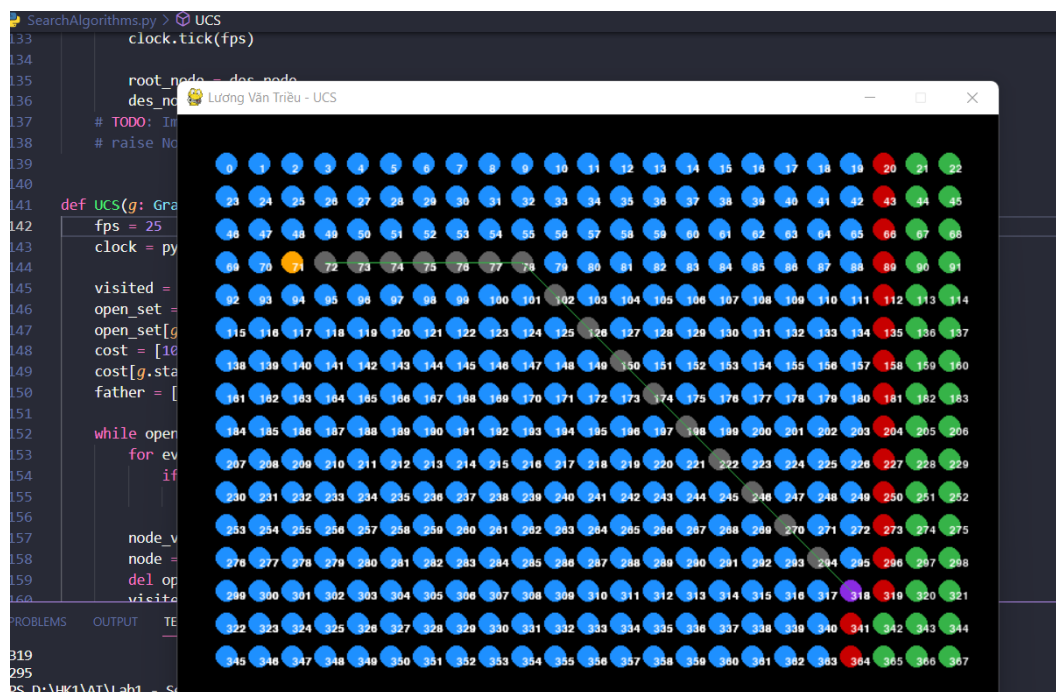
Bước 3: Truy vết tìm đường đi.

- Nhận xét và kết quả thu được.

Cũng như DFS thì BFS cũng là thuật toán tìm kiếm mù, nó mang tính chất vét cạn. Thuật toán sẽ duyệt qua tất cả các đỉnh nếu đồ thị liên thông. Đối với những đồ thị có số lượng lớn đỉnh thì thuật toán sẽ tốn rất nhiều chi phí về thời gian.

3. Thuật toán UCS.

- Kết quả sau khi thực hiện.



- Mô tả quá trình tìm kiếm.

Trong thuật toán này từ trạng thái bắt đầu, ta sẽ truy cập các vùng lân cận và sẽ chọn trạng thái ít tốn kém nhất, sau đó ta sẽ chọn trạng thái ít tốn kém nhất tiếp theo từ

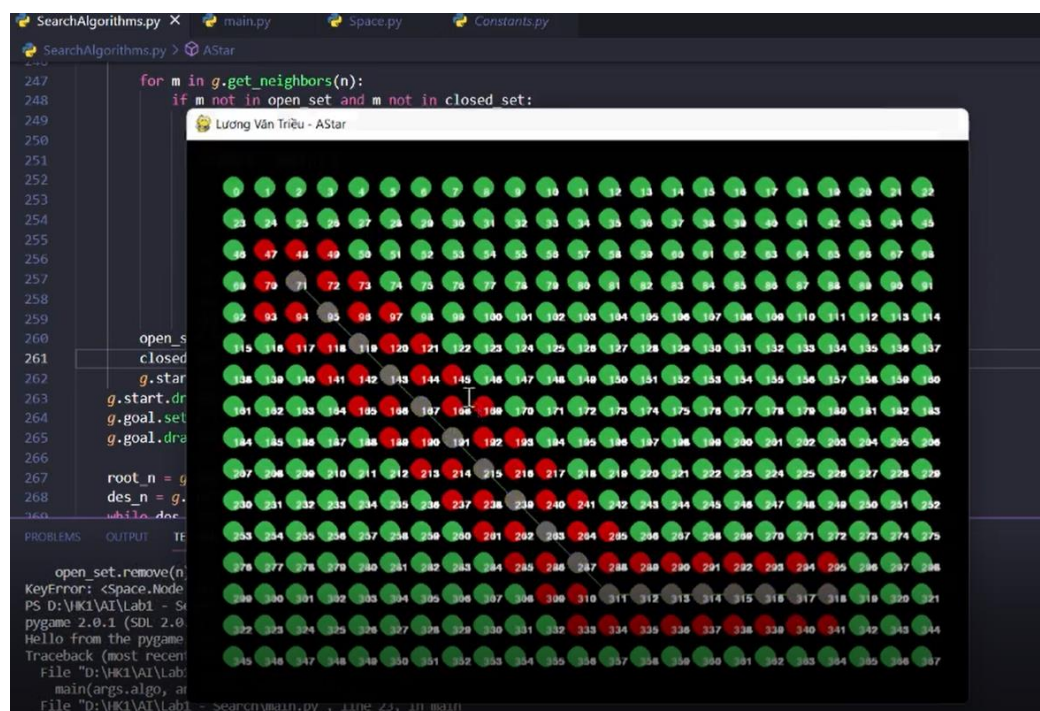
tất cả các trạng thái chưa được truy cập và lân cận của các vùng đã truy cập, theo cách này, ta sẽ cố gắng đạt được trạng thái mục tiêu (lưu ý rằng ta sẽ không tiếp tục con đường qua trạng thái mục tiêu), ngay cả khi ta đạt được trạng thái mục tiêu, ta sẽ tiếp tục tìm kiếm các con đường khả thi khác (nếu có nhiều mục tiêu). Ta sẽ giữ một hàng đợi ưu tiên sẽ cung cấp trạng thái tiếp theo ít tốn kém nhất từ tất cả các trạng thái lân cận của các vùng đã truy cập.

- Nhận xét và kết quả thu được.

Thuật toán UCS cũng là thuật toán tìm kiếm mù, theo như ví dụ trên nó cũng dò qua gần như hết các node trong không gian từ nút gốc đến nút đích. Nếu số lượng đỉnh lớn thì nó cũng sẽ tiêu tốn nhiều chi phí về thời gian. Khả năng tìm kiếm của nó luôn tìm thấy không như BFS và DFS thì nó có thể bị kẹt, tối ưu thì nó cũng hơn BFS và DFS nếu như $cost > 0$. Ta cũng thấy rằng nếu như đồ thị có chi phí ở mỗi bước là như nhau thì thuật toán UCS sẽ biến thành thuật toán tìm kiếm theo chiều rộng.

4. Thuật toán A*.

- Kết quả sau khi thực hiện.



The screenshot displays a Python IDE with a file named `SearchAlgorithms.py` open. The code implements the A* search algorithm, showing the `get_neighbors` method and the main search loop. The grid visualization shows a 20x20 grid of nodes, with the start node (0) at (0,0) and the goal node (1) at (19,19). The grid is color-coded: green for nodes in the open set, red for nodes in the closed set, and grey for nodes that are neither. The path from the start to the goal is highlighted in white. The IDE also shows a traceback for a `KeyError` exception, indicating an issue with the `open_set` dictionary.

```
SearchAlgorithms.py > AStar
247 for m in g.get_neighbors(n):
248     if m not in open_set and m not in closed_set:
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```

```
open_set.remove(n)
KeyError: <Space.Node
PS D:\HK1\AI\Lab1 - S
pygame 2.0.1 (SDL 2.0
Hello from the pygame
Traceback (most recent
File "D:\HK1\AI\Lab1
main(args.algo, at
File "D:\HK1\AI\Lab1 - Search\main.py", line 23, in main
```

- Mô tả quá trình tìm kiếm.

A* xây dựng tăng dần tất cả các tuyến đường từ điểm 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 (informed tìm kiếm thuật toán), 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 điểm bất kỳ cho trước tới đích. Trong trường hợp tìm đường đi, đánh giá này có thể là khoảng cách đường chim bay – một đánh giá xấp xỉ thường dùng cho khoảng cách của đường giao thông.

- Nhận xét và kết quả thu được.

Ta thấy rằng việc chạy thuật toán nhanh hơn hẳn các thuật toán BFS, DFS, UCS, và vùng tìm kiếm nhỏ hơn rất nhiều so với những thuật toán trên đó là do nó có heuristic tốt và nó đã có kinh nghiệm khi dò đường cho node. Và nó có tính tối ưu nhất trong 4 thuật toán mà ta cài đặt.

V. Đánh giá thang điểm.

1. Tìm hiểu và trình bày các thuật toán

Phần này em tự đánh giá 4 điểm.

2. So sánh các thuật toán với nhau.

Phần này em tự đánh giá 1.5 điểm.

3. Cài đặt thuật toán.

Phần này em tự đánh giá 3 điểm.

4. Tìm hiểu các thuật toán tìm kiếm bên ngoài yêu cầu.

Phần này em có tìm hiểu thêm 2 thuật toán là Bellman – Ford và Dijkstra nên em tự đánh giá 1 điểm.

Tổng : 9.5 điểm.

TÀI LIỆU THAM KHẢO

1. <https://websitehcm.com/thuat-toan-tim-kiem-uninformed-search-algorithms/#:~:text=Uninformed%20search%20l%C3%A0%20m%E1%BB%99t%20l%E1%BA%A1i,g%E1%BB%8Di%20l%C3%A0%20t%C3%ACm%20ki%E1%BA%BFm%20m%C3%B9>.
2. <https://www.geeksforgeeks.org/search-algorithms-in-ai/>
3. <https://www.geeksforgeeks.org/difference-between-informed-and-uninformed-search-in-ai/>
4. <https://labs.flinters.vn/wp-content/uploads/2020/10/AI-algos-1-e1547043543151.png/>
5. <https://developer945330148.wordpress.com/2018/04/30/thuat-toan-tim-kiem-sau-depth-first-search/>
6. <http://www.cs.toronto.edu/~heap/270F02/node36.html>
7. <https://vnoi.info/wiki/algo/graph-theory/breadth-first-search.md>
8. <https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/>
9. <https://websitehcm.com/thuat-toan-tim-kiem-uninformed-search-algorithms/>
10. <https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/>
11. <https://www.baeldung.com/cs/uniform-cost-search-vs-dijkstras>
12. <https://python.plainenglish.io/a-algorithm-in-python-79475244b06f>
13. <https://tek4.vn/khoa-hoc/cau-truc-du-lieu-va-giai-thuat/thuat-toan-dfs-tim-kiem-theo-chieu-sau-tren-do-thi>
14. <https://tek4.vn/khoa-hoc/cau-truc-du-lieu-va-giai-thuat/thuat-toan-bfs-tim-kiem-theo-chieu-rong-tren-do-thi>
15. [https://vi.wikipedia.org/wiki/Gi%E1%BA%A3i_thu%E1%BA%ADt_t%C3%ACm_ki%E1%BA%BFm_A%](https://vi.wikipedia.org/wiki/Gi%E1%BA%A3i_thu%E1%BA%ADt_t%C3%ACm_ki%E1%BA%BFm_A%20)
16. <https://vnoi.info/wiki/algo/graph-theory/shortest-path.md>

17.<https://chidokun.github.io/2021/09/dijkstra-algorithm/>