ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

Lab1

Môn : Cơ sở trí tuệ nhân tạo

Họ và tên: Nguyễn Vũ Hiếu

MSSV:20120478

Thành phố Hồ Chí Minh-2023

1. Study & Report

Các thành phần của một bài toán tìm kiếm bao gồm:

- Tập dữ liệu: Là tập hợp các đối tượng cần được tìm kiếm.
- Không gian trạng thái: Là tập hợp các trạng thái khả thi trong quá trình tìm kiếm, bao gồm trạng thái ban đầu, trạng thái kết thúc và các trạng thái trung gian.
- Hàm chi phí: Là hàm đo lường chi phí để đi từ một trạng thái này đến một trạng thái khác.
- Hàm mục tiêu: Là hàm đánh giá trạng thái hiện tại và đưa ra quyết định tìm kiếm tiếp theo.

Tìm kiếm có thông tin và tìm kiếm không rõ ràng là hai loại tìm kiếm khác nhau.

Tìm kiếm có thông tin là loại tìm kiếm trong đó các thông tin về mục tiêu được cung cấp trước. Ví dụ, tìm kiếm trên một bản đồ để tìm đường đi từ A đến B, ta biết A và B nằm ở đâu trên bản đồ và có thể sử dụng thông tin địa lý để tìm đường đi. Trong tìm kiếm có thông tin, người dùng có thể sử dụng các thông tin có sẵn để tối ưu hóa quá trình tìm kiếm và giảm thiểu thời gian tìm kiếm

Tìm kiếm không rõ ràng là loại tìm kiếm trong đó thông tin về mục tiêu không được cung cấp trước. Ví dụ, tìm kiếm trên một trang web để tìm các thông tin liên quan đến một chủ đề cụ thể, nhưng không biết chính xác những thông tin đó là gì. Trong trường hợp này, các thuật toán tìm kiếm sẽ dựa trên các chỉ số và độ tương đồng để tìm các kết quả phù hợp. Trong tìm kiếm không rõ ràng, người dùng phải dựa vào các thuật toán tìm kiếm để tìm kiếm các kết quả phù hợp, do đó, thời gian tìm kiếm có thể tốn nhiều hơn và kết quả có thể không chính xác hoàn toàn.

Mã giả chung của bài toán tìm kiếm:

```
function search(problem):
    Initialize the frontier using the initial state of the problem
    Initialize the explored set to be empty

    while frontier is not empty do
        state = select_next_state(frontier)
        if state is the goal state then
            return solution

        add state to explored set
        for each action in problem.actions(state) do
            child = problem.result(state, action)
```

if child is not in explored set and child is not in frontier then

add child to frontier

else if child is in frontier and path_cost(child) < path_cost(frontier[child]) then

replace frontier[child] with child

return failure

❖ DFS

- Ý tưởng:
  -Ý tưởng chung của thuật toán Depth-First Search (DFS) là duyệt một đồ thị (hoặc cây) bằng cách khám phá tất cả các đỉnh của một nhánh trước khi đi sang nhánh khác. Thuật toán này bắt đầu từ một đỉnh bất kỳ, duyệt qua tất cả các đỉnh kề với đỉnh đó theo một thứ tự nào đó, và tiếp tục duyệt từng đỉnh đó tương tự cho đến khi không còn đỉnh nào để duyệt hoặc đã tìm thấy đỉnh đích cần tìm kiếm.
- Mã giả

```
DFSTraversal(start_node):
    visited := a set to store references to all visited nodes

    stack := a stack to store references to nodes we should visit later
    stack.push(start_node)
    visited.add(start_node)

    while stack is not empty:
        current_node := stack.pop()
        process current_node

        for neighbor in current_node.neighbors:
            if neighbor is not in visited:
                stack.push(neighbor)
                visited.add(neighbor)
```

*Đệ quy

```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)
```

- Các thuộc tính
  -Tính đầy đủ: Thuật toán DFS sẽ duyệt qua tất cả các đỉnh trong đồ thị nếu có thể đi từ đỉnh bắt đầu tới tất cả các đỉnh còn lại. Tuy nhiên, nếu đồ thị không liên thông, chỉ các đỉnh được kết nối với đỉnh bắt đầu mới được duyệt qua.

-Tối ưu: Thuật toán DFS không đảm bảo tìm được đường đi ngắn nhất (shortest path) giữa hai đỉnh trong đồ thị. Nếu đồ thị có trọng số âm hoặc có chu trình âm, thuật toán DFS có thể không tìm được đường đi ngắn nhất hoặc không tìm được đường đi đó.
-Độ phức tạp thời gian: Độ phức tạp thời gian của thuật toán DFS là O(V+E), trong đó V là số đỉnh và E là số cạnh trong đồ thị. Thuật toán sẽ truy cập mỗi đỉnh và cạnh đúng một lần.
-Độ phức tạp không gian: Độ phức tạp không gian của thuật toán DFS là O(V), trong đó V là số đỉnh trong đồ thị. Việc lưu trữ stack để duyệt đồ thị có thể sử dụng đến O(V) không gian lưu trữ.

❖ BFS

- Ý tưởng:
  Thuật toán BFS (Breadth-First Search) là một thuật toán tìm kiếm trên đồ thị, bắt đầu từ một đỉnh bất kỳ trong đồ thị và duyệt qua tất cả các đỉnh khác của đồ thị theo thứ tự tăng dần của khoảng cách từ đỉnh bắt đầu. Ý tưởng chính của thuật toán BFS là sử dụng hàng đợi (queue) để lưu trữ các đỉnh sẽ được duyệt tiếp theo.
- Mã giả

```
BFSTraversal(start_node):
visited := a set to store references to all visited nodes

    queue := a queue to store references to nodes we should visit later
    queue.enqueue(start_node)
    visited.add(start_node)

    while queue is not empty:
        current_node := queue.dequeue()
        process current_node

        for neighbor in current_node.neighbors:
            if neighbor is not in visited:
                queue.enqueue(neighbor)
                visited.add(neighbor)
```

- Các thuộc tính
  -Tính đầy đủ: BFS đảm bảo sẽ tìm được tất cả các đỉnh kết nối với đỉnh bắt đầu nếu đồ thị là liên thông.
  -Tối ưu: BFS tìm đường đi ngắn nhất giữa đỉnh bắt đầu và các đỉnh kết nối nếu đồ thị có trọng số không âm.
  -Độ phức tạp thời gian: Độ phức tạp thời gian (time complexity): Độ phức tạp thời gian của BFS là O(V + E), trong đó V là số đỉnh của đồ thị, E là số cạnh của đồ thị. Tức là thuật toán phải duyệt qua tất cả các đỉnh và cạnh của đồ thị.

-Độ phức tạp không gian: Độ phức tạp không gian của BFS cũng là O(V + E), do việc lưu trữ các đỉnh trong hàng đợi.

❖ UCS

- Ý tưởng:
  Thuật toán UCS (Uniform Cost Search) là một thuật toán tìm kiếm theo chi phí đồng đều, tức là tìm kiếm đường đi ngắn nhất từ một đỉnh đến tất cả các đỉnh kết nối. Ý tưởng chính của thuật toán UCS là duyệt đỉnh theo thứ tự tăng dần của chi phí từ đỉnh bắt đầu đến đỉnh đó.
- Mã giả

```
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.dequeue
        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
```

- Các thuộc tính
  -Tính đầy đủ: Nếu tồn tại đường đi từ đỉnh bắt đầu đến đỉnh đích, thuật toán UCS sẽ tìm ra đường đi đó vì nó duyệt tất cả các đỉnh và cập nhật đường đi ngắn nhất tới mỗi đỉnh khi tìm thấy một đường đi mới có chi phí thấp hơn.
  -Tối ưu: Thuật toán UCS đảm bảo tìm được đường đi ngắn nhất giữa hai đỉnh với chi phí không âm. Nếu có nhiều đường đi cùng chi phí, thuật toán UCS sẽ chọn một đường đi có thứ tự từ điển nhỏ hơn (theo định nghĩa của các đỉnh).

  -Độ phức tạp thời gian: Độ phức tạp thời gian của thuật toán UCS là O($b^{1+\frac{C^*}{\epsilon}}$), trong đó b là số lượng các đỉnh kết nối với đỉnh hiện tại, C* là chi phí đường đi ngắn nhất từ đỉnh bắt đầu đến đỉnh đích và ϵ là số lượng các cạnh có chi phí nhỏ hơn một số rất nhỏ, ví dụ như 0.01. Tuy nhiên, nếu mỗi đỉnh có một số lượng đỉnh kết nối lớn và

giá trị chi phí không được giới hạn thì độ phức tạp thời gian của thuật toán UCS có thể rất cao.
-Độ phức tạp không gian: Độ phức tạp không gian của thuật toán UCS phụ thuộc vào số lượng đỉnh và cạnh của đồ thị. Trong trường hợp tệ nhất, độ phức tạp không gian của thuật toán là O($b^{1+\frac{C^*}{\epsilon}}$), trong đó b là số lượng các đỉnh kết nối với đỉnh hiện tại, C* là chi phí đường đi ngắn nhất từ đỉnh bắt đầu đến đỉnh đích và ϵ là số lượng các cạnh có chi phí nhỏ hơn một số rất nhỏ, ví dụ như 0.01.

❖ A*

- Ý tưởng

Ý tưởng của thuật toán A* (A Star) là sử dụng thông tin về khoảng cách dự kiến từ một đỉnh đến đích để xác định đỉnh tiếp theo để duyệt. Thuật toán A* cố gắng tối ưu hóa việc duyệt đồ thị bằng cách kết hợp chi phí thực tế từ đỉnh bắt đầu đến đỉnh hiện tại với khoảng cách dự kiến từ đỉnh hiện tại đến đỉnh đích.

- Mã giả

```
Put node_start in the OPEN list with f(node_start) = h(node_start) (initialization)
while the OPEN list is not empty {
   Take from the open list the node node_current with the lowest
         f(node_current) = g(node_current) + h(node_current)
   if node_current is node_goal we have found the solution; break
   Generate each state node_successor that come after node_current
   for each node_successor of node_current {
         Set successor_current_cost = g(node_current) + w(node_current,
      node_successor)
         if node_successor is in the OPEN list {
               if g(node_successor) ≤ successor_current_cost continue
         } else if node_successor is in the CLOSED list {
               if g(node_successor) ≤ successor_current_cost continue
               Move node_successor from the CLOSED list to the OPEN list
         } else {
               Add node_successor to the OPEN list
               Set h(node_successor) to be the heuristic distance to node_goal
      }
         Set g(node_successor) = successor_current_cost
         Set the parent of node_successor to node_current
   }
   Add node_current to the CLOSED list
}
if(node_current != node_goal) exit with error (the OPEN list is empty)
```

- Các thuộc tính

-Tính đầy đủ: Thuật toán A* đảm bảo tìm được đường đi từ đỉnh bắt đầu đến đỉnh đích nếu tồn tại đường đi đó.

-Tối ưu: Nếu hàm khoảng cách dự kiến (heuristic) được sử dụng làm hàm h(n) là chính xác và đồ thị không có trọng số âm, thuật toán A* đảm bảo tìm được đường đi ngắn nhất từ đỉnh bắt đầu đến đỉnh đích.
-Độ phức tạp thời gian: Độ phức tạp thời gian của thuật toán A* phụ thuộc vào hàm khoảng cách dự kiến (heuristic) được sử dụng. Trong trường hợp tốt nhất, độ phức tạp thời gian là O(b^d), trong đó b là số lượng đỉnh kề của một đỉnh bất kỳ trong đồ thị và d là độ sâu của đỉnh đích trong cây tìm kiếm. Trong trường hợp xấu nhất, độ phức tạp thời gian là O(b^d), nhưng trong thực tế, thuật toán A* thường hoạt động nhanh hơn vì các hàm khoảng cách dự kiến được sử dụng để giảm số lượng đỉnh được duyệt.
-Độ phức tạp không gian: Độ phức tạp không gian của thuật toán A* phụ thuộc vào số lượng đỉnh và cạnh của đồ thị. Thuật toán A* sử dụng một hàng đợi ưu tiên (priority queue) để lưu trữ các đỉnh chưa được duyệt, vì vậy độ phức tạp không gian của thuật toán là O(V), trong đó V là số lượng đỉnh trong đồ thị.

- Định nghĩa heuristics trong A Star
  Trong thuật toán A*, heuristic được sử dụng để ước lượng chi phí còn lại để đi từ đỉnh hiện tại đến đỉnh đích.
  Một số phương án:
  - Euclidean distance heuristic: Ước lượng khoảng cách giữa hai điểm A và B được tính bằng cách sử dụng độ dài của đoạn thẳng nối hai điểm đó. Công thức được sử dụng để tính toán ước lượng này là: h(n) $=\sqrt{((x_{goal} - x_{current})^2 + (y_{goal} - y_{current})^2)}$, trong đó (x_current, y_current) là tọa độ của đỉnh hiện tại, và (x_goal, y_goal) là tọa độ của đỉnh đích.
  - Manhattan distance heuristic: Ước lượng khoảng cách giữa hai điểm A và B được tính bằng tổng khoảng cách giữa các thành phần tọa độ của hai điểm đó. Công thức được sử dụng để tính toán ước lượng này là: h(n) = $|x\_goal - x\_current| + |y\_goal - y\_current|$.
  - Diagonal distance heuristic: Ước lượng khoảng cách giữa hai điểm A và B được tính bằng độ dài của đường chéo của hình chữ nhật giới hạn bởi hai điểm đó. Công thức được sử dụng để tính toán ước lượng này là: h(n) = max($|x\_goal - x\_current|, |y\_goal - y\_current|$).
  - Minimum spanning tree heuristic: Ước lượng khoảng cách từ đỉnh hiện tại đến đỉnh đích được tính bằng độ dài của cây khung nhỏ nhất bao phủ toàn bộ đồ thị.
  - Landmark heuristic: Ước lượng khoảng cách từ đỉnh hiện tại đến đỉnh đích được tính bằng khoảng cách từ đỉnh hiện tại đến một hoặc nhiều "điểm mốc" (landmarks) được chọn trước trong đồ thị.

2. Comparison

- UCS, Greedy và A Star

-UCS, Greedy và A* đều là các thuật toán tìm kiếm đường đi trong không gian trạng thái. Tuy nhiên, chúng có những khác biệt trong cách thức hoạt động và đặc tính.

	UCS	Greedy	A Star
Cách hoạt động	Đánh giá chi phí của một nút bằng tổng chi phí từ gốc đến nút đó, thông qua việc tính toán chi phí để đi từ một nút đến một nút con bất kỳ.	Chỉ đánh giá chi phí đến đích và chọn nút con gần đích nhất để tiếp tục tìm kiếm.	Kết hợp giữa UCS và Greedy, tìm đường đi dựa trên cả hàm chi phí và heuristic function
Kết quả	Tìm kiếm tối ưu, tức là đảm bảo tìm được đường đi ngắn nhất từ điểm bắt đầu đến điểm kết thúc.	Tìm kiếm không tối ưu, tức là không đảm bảo tìm được đường đi ngắn nhất. Nó chỉ tìm kiếm đến đích gần nhất.	Tìm kiếm tối ưu, tức là đảm bảo tìm được đường đi ngắn nhất từ điểm bắt đầu đến điểm kết thúc.
Độ phức tạp	Độ phức tạp thời gian của UCS phụ thuộc vào số lượng trạng thái trong không gian trạng thái và giá trị của chi phí. Độ phức tạp không gian của UCS phụ thuộc vào số lượng trạng thái trong không gian trạng thái.	Độ phức tạp thời gian của Greedy Search phụ thuộc vào số lượng trạng thái trong không gian trạng thái và hàm heuristic. Độ phức tạp không gian của Greedy Search phụ thuộc vào số lượng trạng thái trong không gian trạng thái.	A* có độ phức tạp thời gian và không gian cao hơn Greedy do phải duyệt toàn bộ không gian trạng thái. A* có độ phức tạp thời gian và không gian gần bằng nhau, nhưng độ phức tạp của A* có thể cao hơn do cần tính toán heuristic.

- UCS và Dijkstra
  UCS (Uniform Cost Search) và Dijkstra là hai thuật toán tìm kiếm đường đi ngắn nhất phổ biến trong trường hợp trọng số trên các cạnh của đồ thị là không âm. Mặc dù cả hai thuật toán đều có mục đích tìm đường đi ngắn nhất, tuy nhiên, chúng có sự khác biệt về chiến lược tìm kiếm.

	UCS	Dijkstra
Ý tưởng	UCS thường triển khai thông qua một hàng đợi đơn giản, sắp xếp các nút theo chi phí của chúng.	Thuật toán Dijkstra thường triển khai bằng cách sử dụng một hàng đợi ưu tiên (min-heap hoặc max-heap) để chọn nút có trọng số thấp nhất tại mỗi bước.
Kết quả	Trong một đồ thị có trọng số âm, UCS không thể đảm bảo	Dijkstra yêu cầu tất cả các trọng số phải không âm, và việc sử

	tìm kiếm đường đi ngắn nhất và có thể trả về kết quả không chính xác.	dụng nó trong đồ thị có trọng số âm có thể dẫn đến kết quả không chính xác hoặc lặp vô hạn.
Độ phức tạp	UCS: Độ phức tạp thời gian trung bình của UCS là O(b^(d+1)), trong đó "b" là hệ số nhánh trung bình và "d" là độ sâu của nút đích trong đồ thị. UCS thường cần lưu trữ một danh sách nút trong hàng đợi ưu tiên.	Độ phức tạp thời gian của thuật toán Dijkstra là O(V^2) hoặc O(E + V log V), tùy thuộc vào cách thuật toán được triển khai. "V" là số lượng nút, và "E" là số lượng cạnh trong đồ thị. Dijkstra có thể nhanh hơn UCS trong trường hợp số lượng cạnh là O(V^2). Dijkstra cần lưu trữ các giá trị chi phí tới từng nút do đó cần nhiều không gian lưu trữ hơn UCS

3. Code

-DFS:

```
def DFS(g: SearchSpace, sc: pygame.Surface):
    open_set = [g.start.id]
    closed_set = []
    father = [-1] * g.get_length()

    g.start.set_color(ORANGE, sc)  # Set the start node to orange
    g.goal.set_color(PURPLE, sc)  # Set the goal node to purple
    while open_set:
        current_node_id = open_set.pop()
        current_node = g.get_node_by_id(current_node_id)

        if(current_node_id!=g.start.id and current_node_id!=g.goal.id):
            # Set the current node to yellow
            current_node.set_color(YELLOW, sc)
            pygame.time.delay(20)
        if g.is_goal(current_node):
            break

        if current_node_id in closed_set:
            continue

        for neighbor in g.get_neighbors(current_node):
            neighbor_id = neighbor.id
            if neighbor_id not in closed_set and neighbor_id not in open_set:
```

```
                open_set.append(neighbor_id)
                father[neighbor_id] = current_node.id
                # Color the neighbor red (Node in open_set)
                if(neighbor_id!=g.start.id and neighbor_id!=g.goal.id):
                    g.get_node_by_id(neighbor_id).set_color(RED, sc)

        closed_set.append(current_node_id)

        if(current_node_id!=g.start.id and current_node_id!=g.goal.id):
            current_node.set_color(BLUE, sc)

    # Here, you can trace back the path and color it white
    if g.is_goal(current_node):
        path = [current_node.id]
        while current_node.id != g.start.id:
            current_node = g.get_node_by_id(father[current_node.id])
            path.append(current_node.id)

        # Reverse the path
        path = path[::-1]

        # Color the path white (The path you found)
        for i in range(1, len(path)):
            node1 = g.get_node_by_id(path[i - 1])
            node2 = g.get_node_by_id(path[i])
            x1, y1 = node1.rect.center
            x2, y2 = node2.rect.center
            pygame.draw.line(sc, WHITE, (x1, y1), (x2, y2), 5)  # Adjust line
width as needed
            pygame.time.delay(50)
            pygame.display.update()
      raise NotImplementedError('not implemented')
```

*Giải thích các bước:

- Khởi tạo tập open_set với một phần tử duy nhất, đó là nút bắt đầu, và tập closed_set trống. Đồng thời khởi tạo mảng father để theo dõi cha của mỗi nút.
- Đánh dấu nút bắt đầu (màu cam) và nút đích (màu tím) theo yêu cầu. Bắt đầu vòng lặp chính: While open_set còn các phần tử. Lấy một nút từ open_set để xem xét (bước này tương ứng với việc pop một nút từ ngăn xếp trong DFS).

- Kiểm tra xem nút hiện tại có phải nút đích không. Nếu đúng, dừng quá trình tìm kiếm.
- Kiểm tra xem nút hiện tại đã nằm trong tập closed_set chưa. Nếu nó nằm trong closed_set, bỏ qua nút này. Nếu nút hiện tại không phải là nút đích hoặc nút bắt đầu, đánh dấu nút này màu vàng (YELLOW).
- Lặp qua các nút lân cận của nút hiện tại, kiểm tra xem chúng hợp lệ (không nằm trong closed_set hoặc open_set). Nếu nút lân cận hợp lệ, thêm nó vào open_set, gán nút hiện tại là cha của nó, và đánh dấu nó màu đỏ (RED) để thể hiện rằng nó thuộc tập open_set.
- Sau khi xem xét và đánh dấu màu nút hiện tại, thêm nó vào closed_set. Quá trình này tiếp tục cho đến khi tìm thấy nút đích hoặc open_set trở thành rỗng (không còn nút nào để xem xét).
- Nếu tìm thấy nút đích, quá trình dừng lại, và sau đó bạn có thể sử dụng mảng father để theo dõi ngược lại từ nút đích đến nút bắt đầu, xây dựng đường đi, nếu không tìm được đường đi thì trả về một ngoại lệ.

*Kết quả: Tìm được đường đi tới nút đích khá nhanh.

BFS:

```
def BFS(g: SearchSpace, sc: pygame.Surface):
    open_set = [g.start.id]
    closed_set = []
    father = [-1] * g.get_length()
    g.start.set_color(ORANGE, sc)  # Set the start node to orange
    g.goal.set_color(PURPLE, sc)  # Set the goal node to purple

    while open_set:
        current_node_id = open_set.pop(0)  # Pop the first node in the queue
        current_node = g.get_node_by_id(current_node_id)

        if(current_node_id!=g.start.id and current_node_id!=g.goal.id):
            # Set the current node to yellow
            current_node.set_color(YELLOW, sc)
            pygame.time.delay(20)

        if g.is_goal(current_node):
            break

        if current_node_id in closed_set:
            continue

        for neighbor in g.get_neighbors(current_node):
```

```
            neighbor_id = neighbor.id
            if neighbor_id not in closed_set and neighbor_id not in open_set:
                open_set.append(neighbor_id)
                father[neighbor_id] = current_node.id
                # Color the neighbor red (Node in open_set)
                if(neighbor_id!=g.start.id and neighbor_id!=g.goal.id):
                    g.get_node_by_id(neighbor_id).set_color(RED, sc)

        closed_set.append(current_node_id)
        if(current_node_id!=g.start.id and current_node_id!=g.goal.id):
            current_node.set_color(BLUE, sc)
        pygame.display.update()
            # Here, you can trace back the path and color it white
    if g.is_goal(current_node):
        path = [current_node.id]
        while current_node.id != g.start.id:
            current_node = g.get_node_by_id(father[current_node.id])
            path.append(current_node.id)

        # Reverse the path
        path = path[::-1]

        # Color the path white (The path you found)
        for i in range(1, len(path)):
            node1 = g.get_node_by_id(path[i - 1])
            node2 = g.get_node_by_id(path[i])
            x1, y1 = node1.rect.center
            x2, y2 = node2.rect.center
            pygame.draw.line(sc, WHITE, (x1, y1), (x2, y2), 5)  # Adjust line
width as needed
            pygame.time.delay(20)
            pygame.display.update()
    raise NotImplementedError('not implemented')
```

*Giải thích các bước

- Khởi tạo tập open_set với một phần tử duy nhất, đó là nút bắt đầu, và tập closed_set trống. Đồng thời khởi tạo mảng father để theo dõi cha của mỗi nút.
- Đánh dấu nút bắt đầu (màu cam) và nút đích (màu tím) theo yêu cầu. Bắt đầu vòng lặp chính: While open_set còn các phần tử. Lấy một nút từ open_set để xét .

- Trong trường hợp này, bạn luôn lấy nút đầu tiên của hàng đợi. Kiểm tra xem nút hiện tại có phải là nút đích không. Nếu đúng, dừng quá trình tìm kiếm.
- Kiểm tra xem nút hiện tại đã nằm trong tập closed_set chưa. Nếu nó nằm trong closed_set, bỏ qua nút này. Nếu nút hiện tại không phải là nút đích hoặc nút bắt đầu, đánh dấu nó màu vàng (YELLOW) để thể hiện rằng đang xem xét nó.
- Lặp qua các nút lân cận của nút hiện tại, kiểm tra xem chúng hợp lệ (không nằm trong closed_set hoặc open_set). Nếu nút lân cận hợp lệ, thêm nó vào open_set, gán nút hiện tại là cha của nó, và đánh dấu nó màu đỏ (RED) để thể hiện rằng nó thuộc tập open_set.
- Sau khi xem xét và đánh dấu màu nút hiện tại, thêm nó vào closed_set và đánh dấu màu nút hiện tại là màu xanh (BLUE) để thể hiện rằng nó đã được xem xét và thuộc tập closed_set. Quá trình này tiếp tục cho đến khi tìm thấy nút đích hoặc open_set trở thành rỗng (không còn nút nào để xem xét).
- Nếu tìm thấy nút đích, quá trình dừng lại, và sau đó bạn có thể sử dụng mảng father để theo dõi ngược lại từ nút đích đến nút bắt đầu, xây dựng đường đi, nếu không tìm được đường đi thì trả về một ngoại lệ.

*Kết quả: Tìm được đường đi tới nút đích khá chậm.

UCS:

```
def UCS(g: SearchSpace, sc: pygame.Surface):
    open_set = [(0, g.start.id)]
    closed_set = []
    father = [-1] * g.get_length()
    cost = [100_000] * g.get_length()
    cost[g.start.id] = 0

    g.start.set_color(ORANGE, sc)  # Set the start node to orange
    g.goal.set_color(PURPLE, sc)  # Set the goal node to purple
    while open_set:
        current_cost, current_node_id = heapq.heappop(open_set)
        current_node = g.get_node_by_id(current_node_id)

        if(current_node_id!=g.start.id and current_node_id!=g.goal.id):
            # Set the current node to yellow
            current_node.set_color(YELLOW, sc)
            pygame.time.delay(20)

        if g.is_goal(current_node):
            break
```

```
        if current_node_id in closed_set:
            continue

        for neighbor in g.get_neighbors(current_node):
            neighbor_id = neighbor.id
            new_cost = cost[current_node_id] + 1  # Uniform cost is assumed to be
1 here

            if neighbor_id not in closed_set and (neighbor_id not in [node[1] for
node in open_set] or new_cost < cost[neighbor_id]):
                cost[neighbor_id] = new_cost
                father[neighbor_id] = current_node.id
                # Add the neighbor to the open set with its cost
                heapq.heappush(open_set, (new_cost, neighbor_id))
                # Color the neighbor red (Node in open_set)
                if(neighbor_id!=g.start.id and neighbor_id!=g.goal.id):
                    g.get_node_by_id(neighbor_id).set_color(RED, sc)

        closed_set.append(current_node_id)

        if(current_node_id!=g.start.id and current_node_id!=g.goal.id):
            current_node.set_color(BLUE, sc)

    # Here, you can trace back the path and color it white
    if g.is_goal(current_node):
        path = [current_node.id]
        while current_node.id != g.start.id:
            current_node = g.get_node_by_id(father[current_node.id])
            path.append(current_node.id)

        # Reverse the path
        path = path[::-1]

        # Color the path white (The path you found)
        for i in range(1, len(path)):
            node1 = g.get_node_by_id(path[i - 1])
            node2 = g.get_node_by_id(path[i])
            x1, y1 = node1.rect.center
            x2, y2 = node2.rect.center
            pygame.draw.line(sc, WHITE, (x1, y1), (x2, y2), 5)  # Adjust line
width as needed
            pygame.time.delay(20)
            pygame.display.update()
    raise NotImplementedError('not implemented')
```

*Giải thích các bước:

- Khởi tạo tất cả các biến cần thiết, bao gồm open_set, closed_set, father (để theo dõi đường đi), và cost (để theo dõi chi phí đến từng nút).
- Đánh dấu nút bắt đầu (g.start) thành màu cam và nút đích (g.goal) thành màu tím. Bắt đầu vòng lặp chính cho tìm kiếm.
- Lấy nút có chi phí thấp nhất từ open_set bằng cách sử dụng heap (đã được sắp xếp theo chi phí).
- Kiểm tra xem nút hiện tại có phải là nút đích hay không. Nếu đúng, thoát khỏi vòng lặp vì đã tìm thấy đường đi đến đích. Nếu nút hiện tại không phải là nút đích, đánh dấu nó thành màu vàng (YELLOW) .
- Kiểm tra xem nút hiện tại có trong closed_set (đã được duyệt) hay chưa. Nếu có, bỏ qua nút này và tiếp tục vòng lặp.
- Duyệt qua tất cả các nút lá của nút hiện tại (các nút kề).
    - Tính toán chi phí mới để đến từ nút bắt đầu thông qua nút hiện tại và cập nhật nó trong cost.
    - Nếu nút lá không nằm trong closed_set hoặc có chi phí thấp hơn chi phí hiện tại, thì cập nhật thông tin của nút lá và thêm nó vào open_set.
    - Đánh dấu nút lá đó thành màu đỏ (RED) để thể hiện rằng nó trong tập open_set.
- Thêm nút hiện tại vào closed_set để đánh dấu rằng nó đã được xem xét.
- Đánh dấu nút hiện tại thành màu xanh (BLUE) để thể hiện rằng nó đã được đóng và không xem xét lại nữa. Quá trình này tiếp tục cho đến khi tìm thấy nút đích hoặc open_set trở thành rỗng (không còn nút nào để xem xét).
- Nếu tìm thấy nút đích, quá trình dừng lại, và sau đó bạn có thể sử dụng mảng father để theo dõi ngược lại từ nút đích đến nút bắt đầu, xây dựng đường đi, nếu không tìm được đường đi thì trả về một ngoại lệ.

*Kết quả: Thời gian tìm kiếm đường đi khá chậm vì phải so sánh chi phí các node với nhau

AStar:

```
def AStar(g: SearchSpace, sc: pygame.Surface):
    def heuristic(node, goal):
        # A simple Euclidean distance heuristic
        x1, y1 = node.rect.center
        x2, y2 = goal.rect.center
        return ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** 0.5

    open_set = [(0, g.start.id)]
    closed_set = []
    father = [-1] * g.get_length()
    cost = [float('inf')] * g.get_length()
```

```
    cost[g.start.id] = 0

    g.start.set_color(ORANGE, sc)  # Set the start node to orange
    g.goal.set_color(PURPLE, sc)  # Set the goal node to purple

    while open_set:
        open_set.sort()  # Sort the open set by cost
        _, current_node_id = open_set.pop(0)  # Pop the node with the lowest cost
        current_node = g.get_node_by_id(current_node_id)

        if(current_node_id!=g.start.id and current_node_id!=g.goal.id):
            # Set the current node to yellow
            current_node.set_color(YELLOW, sc)
            pygame.time.delay(20)

        if g.is_goal(current_node):
            break

        if current_node_id in closed_set:
            continue

        for neighbor in g.get_neighbors(current_node):
            neighbor_id = neighbor.id
            neighbor_cost = cost[current_node_id] + 1  # Assuming uniform cost
            if neighbor_cost < cost[neighbor_id]:
                father[neighbor_id] = current_node.id
                cost[neighbor_id] = neighbor_cost
                priority = neighbor_cost + heuristic(neighbor, g.goal)  # A* cost
function
                open_set.append((priority, neighbor_id))
                # Color the neighbor red (Node in open_set)
                if(neighbor_id!=g.start.id and neighbor_id!=g.goal.id):
                    g.get_node_by_id(neighbor_id).set_color(RED, sc)

        closed_set.append(current_node_id)
        if(current_node_id!=g.start.id and current_node_id!=g.goal.id):
            current_node.set_color(BLUE, sc)
        pygame.display.update()

    if g.is_goal(current_node):
        path = [current_node.id]
        while current_node.id != g.start.id:
            current_node = g.get_node_by_id(father[current_node.id])
            path.append(current_node.id)
```

```
        # Reverse the path
        path = path[::-1]

        # Color the path white (The path you found)
        for i in range(1, len(path)):
            node1 = g.get_node_by_id(path[i - 1])
            node2 = g.get_node_by_id(path[i])
            x1, y1 = node1.rect.center
            x2, y2 = node2.rect.center
            pygame.draw.line(sc, WHITE, (x1, y1), (x2, y2), 5)  # Adjust line
width as needed
            pygame.time.delay(20)
            pygame.display.update()
    raise NotImplementedError('not implemented')
```

*Giải thích các bước:

- Tạo hàm heuristic sử dụng khoảng cách Euclidean giữa hai tọa độ trung tâm của nút hiện tại và nút đích. Hàm này giúp thuật toán ước tính đường đi một cách hiệu quả và tối ưu hóa quá trình tìm kiếm.
- Khởi tạo một hàng đợi ưu tiên open_set và mảng closed_set, đánh dấu nút đã duyệt. Khởi tạo mảng father để theo dõi quá trình tìm đường đi và mảng cost để theo dõi chi phí tới mỗi nút. Khởi tạo giá trị cost tới nút bắt đầu là 0. Đặt màu của nút bắt đầu thành màu cam và của nút đích thành màu tím.
- Bắt đầu vòng lặp chính cho đến khi open_set không còn nút nào. Trong mỗi lần lặp, lấy nút có giá trị ưu tiên thấp nhất từ open_set dựa trên hàm cost + heuristic (dựa vào hàm heuristic để ước tính chi phí còn lại tới nút đích).
- Đặt màu của nút hiện tại thành màu vàng (nếu không phải là nút đầu và nút đích). Kiểm tra xem nút hiện tại có phải là nút đích chưa. Nếu có, dừng thuật toán. Nếu nút hiện tại đã được xem xét (trong closed_set), bỏ qua và quay lại bước 5.
- Lặp qua tất cả các hàng xóm của nút hiện tại:
    - Tính toán giá trị chi phí mới từ nút bắt đầu tới hàng xóm này.
    - So sánh giá trị chi phí mới với giá trị chi phí đã ghi nhận trước đó của hàng xóm. Nếu giá trị chi phí mới nhỏ hơn, cập nhật father để ghi nhận quá trình tìm đường và cập nhật giá trị cost cho hàng xóm này.
    - Tính toán giá trị ưu tiên cho hàng xóm dựa trên chi phí mới và hàm heuristic và thêm hàng xóm này vào open_set.
    - Đặt màu của hàng xóm thành màu đỏ (nếu không phải là nút đầu và nút đích) để chỉ ra rằng nó đang trong open_set.
- Đánh dấu nút hiện tại đã được xem xét bằng cách thêm nó vào closed_set và đặt màu nó thành màu xanh. Lặp lại vòng lặp với nút có giá trị ưu tiên thấp nhất trong open_set.

- Nếu tìm thấy nút đích, quá trình dừng lại, và sau đó bạn có thể sử dụng mảng father để theo dõi ngược lại từ nút đích đến nút bắt đầu, xây dựng đường đi, nếu không tìm được đường đi thì trả về một ngoại lệ.

*Kết quả: Thời gian tìm đường đi khá nhanh vì heuristic này hiệu quả đối với bài toán.

❖ Tài liệu tham khảo

https://www.liquisearch.com/uniform-cost_search/pseudocode
https://mat.uab.cat/~alseda/MasterOpt/AStar-Algorithm.pdf
https://courses.engr.illinois.edu/cs225/sp2023/resources/bfs-dfs/
https://www.verywellmind.com/what-is-a-heuristic-2795235
https://stackoverflow.com/questions/2082534/what-is-the-difference-between-greedy-search-and-uniform-cost-search
https://stackoverflow.com/questions/12806452/whats-the-difference-between-uniform-cost-search-and-dijkstras-algorithm#:~:text=Dijkstra's%20algorithm%20searches%20for%20shortest,cost%20to%20a%20goal%20node.
https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/
https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/
https://www.programiz.com/dsa/graph-dfs