HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

KHOA AN TOÀN THÔNG TIN

BÁO CÁO BÀI TẬP LỚN

A* Algorithm

Batch-Informed RRT* (BIT*)

Giảng viên hướng dẫn	TS. Nguyễn Kiều Linh
Nhóm	3
Lớp	D23-111

Hà Nội – 2025

I. A* Algorithm

1. Giới thiệu

- Thuật toán A* (A-star) là một trong những thuật toán tìm kiếm đường đi ngắn nhất phổ biến và hiệu quả nhất. Nó được sử dụng rộng rãi trong lập trình game, định tuyến GPS và nhiều ứng dụng khác trong lĩnh vực trí tuệ nhân tạo và robot.
- A* là một thuật toán tìm kiếm theo cơ chế heuristic, tức là nó sử dụng thông tin dự đoán để hướng dẫn quá trình tìm kiếm. Thuật toán này kết hợp giữa tìm kiếm theo chi phí thấp nhất (like Dijkstra) và tìm kiếm theo hướng dẫn (Greedy Best-First Search), giúp tìm ra đường đi ngắn nhất một cách nhanh chóng.
- Thuật toán này đặc biệt hiệu quả trong các bài toán lập kế hoạch đơn truy vấn (single-query path planning), nơi cần tìm đường đi tối ưu giữa hai điểm cụ thể mà không cần khảo sát toàn bộ không gian.

2. A*: Thuật toán lai giữa Dijkstra và GBFS

a. Tổng quát

- A* hoạt động bằng cách đánh giá mỗi điểm (nút) trong không gian bằng một hàm chi phí ước tính:

  f(n) = g(n) + h(n)

  g(n): Chi phí thực từ điểm bắt đầu đến điểm n.

  h(n): Chi phí ước lượng còn lại từ n đến đích (heuristic).

  f(n): Tổng chi phí dự đoán khi đi qua n.

b. Dijkstra

- Sử dụng hàm: f(n) = g(n) với g(n) là chi phí thực từ điểm xuất phát đến đến điểm n.
- Cơ chế: Luôn mở rộng (thêm vào priority_queue) điểm có g(n) nhỏ nhất trước, đảm bảo luôn tìm được đường đi ngắn nhất với mọi trọng số cạnh không âm.

c. Greedy Best First Search

- Sử dụng hàm: f(n) = h(n) với h(n) là giá trị heuristic ước lượng chi phí còn lại từ n đến đích.
- Cơ chế: "Tham lam" chọn luôn điểm có h(n) nhỏ nhất trước, giúp đi nhanh về phía đích nhưng không đảm bảo được đường đi sẽ là ngắn nhất.

3. Cách hoạt động và triển khai

- Thuật toán A* kết hợp những điểm mạnh nhất của hai thuật toán sau:
    - *Thuật toán Dijkstra:* Tìm đường đi ngắn nhất từ một nút nguồn đến tất cả các nút khác.
    - *Tìm kiếm ưu tiên theo hướng đích (Greedy Best-First Search):* Luôn ưu tiên khám phá nút có vẻ gần đích nhất, dựa trên một hàm ước lượng (heuristic).
- Hãy tưởng tượng bạn đang cố tìm đường ngắn nhất giữa hai thành phố trên bản đồ. Trong khi thuật toán Dijkstra sẽ tìm kiếm theo mọi hướng và thuật toán GBFS có thể đi thẳng đến đích (nhưng có thể bỏ lỡ lối tắt), thì A* làm điều thông minh hơn: nó xét đến cả hai yếu tố sau:
    - Khoảng cách đã đi từ điểm xuất phát.
    - Ước lượng thông minh về khoảng cách còn lại đến đích.
- Sự kết hợp này giúp A* đưa ra quyết định hợp lý hơn khi chọn đường đi tiếp theo, khiến nó vừa hiệu quả lại vừa chính xác.
- A* hoạt động bằng cách duyệt mở rộng các đường đi có chi phí thấp nhất từ điểm bắt đầu đến đích, thông qua một hàm chi phí f(n) = g(n) + h(n).
- Từng bước:
    - Bước 1: Tạo 1 open set(hàng đợi ưu tiên), chứa nút bắt đầu start
      g(start) = 0, f(start) = g(start) + h(start)
      Tạo 1 closed set rỗng để lưu các nút đã xét
    - Bước 2: Lặp
      Chọn nút n trong open set có f(n) min
      Nếu n là đích -> kết thúc và truy vết đường đi
    - Bước 3: Di chuyển n khỏi open set, thêm vào closed set
    - Bước 4: Duyệt các đỉnh kề v với n
      Nếu v trong closed set -> bỏ qua.
      g(v) = g(n) + k(n, v) với k(n, v) là chi phí quãng đường từ n tới v.
      Nếu v chưa có trong open set hoặc g(v) có giá trị mới nhỏ hơn thì cập nhật trong open set.
      Ghi nhớ n là cha của v.
      Nếu open set hết mà chưa tới đích: Không có đường đi từ start đến finish.

4. Code

- Pseudo code:

```
1. Put node_start in the OPEN list with f(node_start) = h(node_start)
   (initialization)
2. while the OPEN list is not empty {
3.   Take from the open list the node node_current with the lowest
4.     f(node_current) = g(node_current) + h(node_current)
5.   if node_current if node_goal we have found the solution; break
6.   Generate each state node_successor that come after node_current
7.   for each node_successor of node_current {
8.     Set successor_current_cost = g(node_current) + w(node_current,
   node_successor)
9.     if node_successor is in the OPEN list {
10.      if g(node_successor) ≤ successor_current_cost continue (to line 20)
11.    } else if node_successor is in the CLOSED list {
12.      if g(node_successor) ≤ successor_current_cost continue (to line 20)
13.      Move node_successor from the CLOSED list to the OPEN list
14.    } else {
15.      Add node_successor to the OPEN list
16.      Set h(node_successor) to be the heuristic distance to node_goal
17.    }
18.    Set g(node_successor) = successor_current_cost
19.    Set the parent of node_successor to node_current
20.  }
21.  Add node_current to the CLOSED list
22.}
23.if(node_current != node_goal) exit with error (the OPEN list is empty)
```

Code:

```
# PathfindingAlgorithm.py

from typing import List, Optional
from node import Node  # Giả sử bạn có module node.py định nghĩa class
Node

class PathfindingAlgorithm:
    """
    Base class cho các thuật toán tìm đường.
    """

    def __init__(self) -> None:
        self.finished: bool = False
        self.start_node: Optional[Node] = None
        self.end_node: Optional[Node] = None

    def start(self, start_node: Node, end_node: Node) -> None:
        """
        Reset trạng thái và khởi tạo thuật toán.

        :param start_node: node bắt đầu
        :param end_node: node đích
        """
        self.finished = False
```

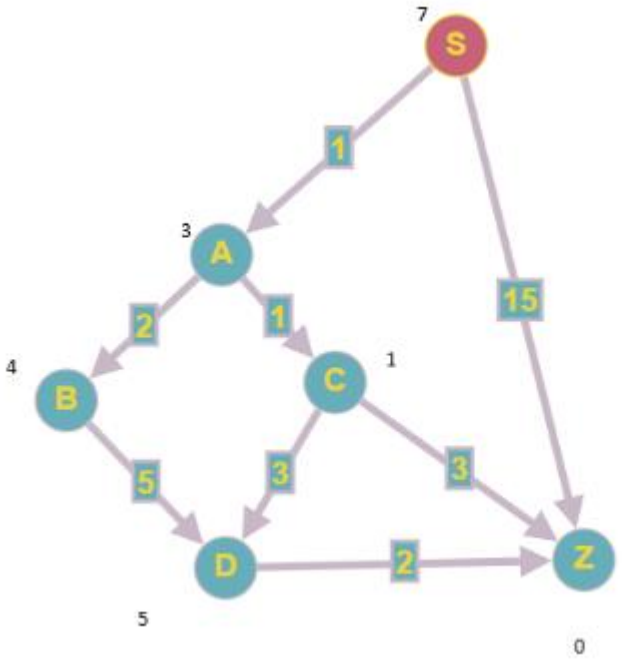
```
        self.start_node = start_node
        self.end_node = end_node

    def next_step(self) -> List[Node]:
        """
        Thực hiện một bước/luồng của thuật toán.

        :return: danh sách các Node đã được cập nhật trong bước này
        """
        return []

# AStar.py

import math
from pathfinding_algorithm import PathfindingAlgorithm
from node import Node  # Giả sử bạn có module node.py định nghĩa class
Node

class AStar(PathfindingAlgorithm):
    """
    Lớp triển khai thuật toán A* kế thừa từ PathfindingAlgorithm.
    """

    def __init__(self) -> None:
        super().__init__()
        self.open_list: list[Node] = []
        self.closed_list: list[Node] = []

    def start(self, start_node: Node, end_node: Node) -> None:
        """
        Reset trạng thái và khởi tạo cho lần tìm đường mới.

        :param start_node: node bắt đầu
        :param end_node: node đích
        """
        super().start(start_node, end_node)
        self.open_list = [self.start_node]
        self.closed_list = []
        # Khởi tạo chi phí
        self.start_node.distance_from_start = 0.0
        self.start_node.distance_to_end = 0.0

    def next_step(self) -> list[Node]:
        """
        Thực hiện một bước của thuật toán A*. Trả về list các node đã
được cập nhật.

        :return: list các Node cập nhật trong bước này
        """
        if not self.open_list:
            self.finished = True
            return []

        updated_nodes: list[Node] = []
        # Chọn node có tổng chi phí (g + h) nhỏ nhất
        current_node = min(
            self.open_list,
            key=lambda n: n.distance_from_start + n.distance_to_end
```

```
        )
        self.open_list.remove(current_node)
        current_node.visited = True

        # Đánh dấu cạnh dẫn từ referer nếu có
        if current_node.referer is not None:
            for edge in current_node.edges:
                if edge.get_other_node(current_node) ==
current_node.referer:
                    edge.visited = True
                    break

        # Nếu đã đến đích
        if current_node.id == self.end_node.id:
            self.open_list = []
            self.finished = True
            return [current_node]

        # Khám phá các neighbor
        for neighbor_info in current_node.neighbors:
            neighbor = neighbor_info.node
            edge = neighbor_info.edge
            # Tính chi phí tạm thời từ start
            tentative_g = (
                current_node.distance_from_start
                + math.hypot(
                    neighbor.longitude - current_node.longitude,
                    neighbor.latitude - current_node.latitude
                )
            )

            # Đánh dấu các cạnh chưa được đánh dấu trên bản đồ
            if neighbor.visited and not edge.visited:
                edge.visited = True
                neighbor.referer = current_node
                updated_nodes.append(neighbor)

            # Nếu neighbor đã trong open_list
            if neighbor in self.open_list:
                if neighbor.distance_from_start <= tentative_g:
                    continue
            # Nếu neighbor đã qua closed_list
            elif neighbor in self.closed_list:
                if neighbor.distance_from_start <= tentative_g:
                    continue
                self.closed_list.remove(neighbor)
                self.open_list.append(neighbor)
            else:
                # Lần đầu gặp neighbor
                self.open_list.append(neighbor)
                neighbor.distance_to_end = math.hypot(
                    neighbor.longitude - self.end_node.longitude,
                    neighbor.latitude - self.end_node.latitude
                )

            # Cập nhật chi phí và đường dẫn
            neighbor.distance_from_start = tentative_g
            neighbor.referer = current_node
```

```
            neighbor.parent = current_node

        # Đưa current_node vào closed_list
        self.closed_list.append(current_node)
        # Trả về các node được cập nhật và node hiện tại
        return [*updated_nodes, current_node]
```

5. Minh họa

- Giả sử ta có đồ thị như sau:



- Minh hoạ thuật toán A*:

Bước	U	Kề (U)	g(v) = g(u) + k(u, v)	f(v) = g(v) + h(v)	Open	Closed
0					S(7)	
1	S(7)	A(3), Z(0)	g(A) = 0 + 1 = 1 g(Z) = 0 + 15 = 15	f(A) = 1 + 3 = 4 f(Z) = 15 + 0 = 15	A(4), Z(15)	S(7)
2	A(4)	B(4), C(1)	g(B) = 1 + 2 = 3 g(C) = 1 + 1 = 2	f(B) = 3 + 4 = 7 f(C) = 2 + 1 = 3	C(3), B(7), Z(15)	A(4)

3	C(3)	D(5), Z(0)	g(D) = 2 + 3 = 5 g(Z) = 2 + 3 = 5	f(D) = 5 + 5 = 10 f(Z) = 5 + 0 = 5	Z(5), B(7), D(10)	C(3)
4	Z(5)	Rỗng			B(7), D(10)	Z(5)

- Ta có: Z là con của C, D, S

  C là con của A

  A là con của S

  → Vậy đường đi ngắn nhất từ S đến Z là: S → A → C → Z

6. Nhận xét

a. Tổng quát

- Tối ưu hóa và hoàn chỉnh (Optimal & Complete)
  - Nếu hàm heuristic h(n) là đánh giá tốt và chấp nhận được (admissible – không đánh giá quá cao), A* sẽ luôn tìm ra đường đi ngắn nhất (nếu có).
  - Đồng thời, nếu có lời giải, A* chắc chắn sẽ tìm ra (complete).
- Kết hợp giữa tìm kiếm theo chi phí và dự đoán f(n) = g(n) + h(n)
  → Do đó, thuật toán này vừa chính xác vừa có tốc độ tốt hơn tìm kiếm theo chiều rộng (BFS) hoặc tìm kiếm theo chi phí nhỏ nhất (Dijkstra).
- Linh hoạt: Có thể tùy chỉnh dễ dàng bằng cách thay đổi heuristic để phù hợp với từng bài toán cụ thể (như game, bản đồ, AI, robot...).
- Có thể sử dụng trong nhiều không gian trạng thái: Không chỉ dùng cho đồ thị, mà còn cho lưới, mê cung, bản đồ đường đi, hoặc các bài toán như giải Rubik, 8 puzzle...
- Ưu điểm là thế nhưng A* vẫn tồn tại một vài nhược điểm từ Dijkstra và GBFS:
- Tốn bộ nhớ: A* lưu trữ tất cả các trạng thái đã duyệt (trong open set và closed set), nên ngốn nhiều RAM, đặc biệt với các bài toán có không gian trạng thái lớn.
- Hiệu suất phụ thuộc vào hàm heuristic: Nếu hàm h(n) không chính xác hoặc đánh giá quá thấp/quá cao, thuật toán có thể mất hiệu quả (chạy chậm hoặc không tối ưu) hoặc chỉ có thế chạy như Dijkstra nhưng lâu hơn.

b. Ưu điểm

- So với Dijkstra:
  - Hiệu suất: Hàm heuristic dùng để ước lượng chi phí (khoảng cách) từ nút hiện tại đích → nó giúp cho thuật toán sẽ tập trung tìm kiếm theo hướng gần đến đích trong khi Dijkstra sẽ có xu hướng tìm kiếm lan tỏa đồng đều tính từ nút hiện tại theo hình tròn. Tìm kiếm cho đến khi tìm được đường đi ngắn nhất tới đích → A* đạt hiệu suất tối ưu hơn do số lượng nút cần duyệt ít hơn.
  - Bộ nhớ: A* chỉ mở rộng các nút có tiềm năng hướng đến đích,và nếu hàm h(n) (heuristic) có thể chấp nhận được thì nó sẽ loại đi các nút thừa,giảm số nút cần lưu trữ trong khi Dijkstra phải duyệt toàn bộ các nút có thể có khoảng cách ngắn hơn khoảng cách hiện tại tới đích,tính cả các nút có hướng ngược lại tới đích. Gây tốn bộ nhớ để lưu trữ → A* chỉ lưu trữ các nút cần thiết còn Dijkstra lưu trữ tất cả các nút mà nó duyệt.
  - Cân bằng giữa thời gian và độ chính xác: Nếu hàm heuristic hoạt động tốt thì nó có thể tìm được đường đi tối ưu mà không cần duyệt toàn bộ đồ thị. Nó có thể điều chỉnh để đánh đổi giữa tốc độ và độ chính xác theo tỉ lệ nghịch trong khi Dijkstra luôn phải duyệt theo quy tắc tìm nút gần nhất trước mà không quan tâm đến hướng của đích, nó không thể tối ưu hóa bằng cách hi sinh tốc độ để có độ chính xác → A* luôn linh hoạt theo từng bài toán cụ thể.
- So với GBFS:
  - Luôn tìm được đường đi ngắn nhất: Do sử dụng cả chi phí tích lũy g(n) thay vì chỉ sử dụng mỗi h(n) nên A* sẽ luôn đảm bảo tìm được đường đi ngắn nhất.
  - Hiệu quả: A* không mở rộng đến các nút không cần thiết, tránh được bẫy nhánh do h(n) sai lệch vì xét cả g(n) trong khi GBFS dễ mắc kẹt trong các nhánh không dẫn đến đích.
  - Bộ nhớ: A* không lưu trữ những nút không cần thiết trong khi GBFS chỉ dựa vào h(n) nên có thể mở rộng ra những nút không cần thiết nên sẽ gây mất mát bộ nhớ.

c. Nhược điểm

- Khó thiết kế được hàm heuristic tối ưu: Nếu h(n) không chính xác, không admissible (vượt quá chi phí thực) hoặc không consistent (không đảm bảo tính ổn định) thì A* có thể không tìm được đường tối ưu, bị lạc hướng và

mất nhiều thời gian hơn, trong trường hợp xấu A* có thể kém hiệu quả hơn Dijkstra.

- Tốn bộ nhớ: Chính vì hàm h(n) không tốt sẽ dẫn đến việc A* phải lưu nhiều nút trung gian hơn mà những nút này có thể không dẫn đến đích hoặc không phải đường đi tối ưu nhất.
- Tốn thời gian: Nếu hàm h(n) quá phức tạp sẽ dẫn đến việc tính hàm f(n) rất lâu trong khi Dijkstra hay GBFS chỉ cần tính hàm g(n)/h(n) đơn lẻ thì A* phải tính hàm f(n) = g(n) + h(n), dẫn đến việc tại mỗi bước sẽ mất nhiều công tính toán hơn.

7. Đọc thêm

Thuật Toán A*: Tìm Đường Ngắn Nhất Trong Đồ Thị

The A* Algorithm: A Complete Guide | DataCamp

Thuật Giải A*

A* algorithm EASY explained (example) - YouTube

A* Pathfinding

Algorithm Visualization

Github A*

II. Batch-Informed RRT* (BIT*)

1. Giới thiệu

- BIT* (Batch-Informed Trees) là một thuật toán lập kế hoạch đường đi kết hợp giữa phương pháp lấy mẫu (Sampling-based Planning) và phương pháp tìm kiếm trên đồ thị (Graph-based Search). Nó được thiết kế để cải thiện hiệu suất của RRT* bằng cách tích hợp các ý tưởng từ A*, giúp tìm kiếm đường đi hiệu quả hơn trong không gian tĩnh có chướng ngại vật phức tạp.
- BIT* xây dựng một cây hoặc đồ thị bằng cách lấy mẫu ngẫu nhiên trong không gian trạng thái và kết nối các mẫu này lại để tìm đường đi tối ưu. Quá trình lấy mẫu trong BIT* có thể được thực hiện theo hai cách chính:
    - Lấy mẫu không thiên vị (Uniform Sampling): Chọn điểm một cách ngẫu nhiên và đồng đều trong toàn bộ không gian cấu hình.
    - Lấy mẫu có thiên vị (Biased Sampling): Hướng việc lấy mẫu đến các vùng quan trọng, như khu vực gần đích hoặc dọc theo đường đi tiềm năng.
- Không giống như RRT* truyền thống, BIT* sử dụng tìm kiếm theo tập hợp (batch processing) và thông tin heuristic để sắp xếp lại ưu tiên trong quá trình mở rộng cây. Điều này giúp thuật toán hội tụ nhanh hơn và giảm số lượng mẫu cần thiết để tìm ra đường đi ngắn nhất.
- BIT* chủ yếu được sử dụng trong bài toán lập kế hoạch đường đi đơn truy vấn (Single-query Path Planning) và lập kế hoạch đường đi trong không gian tĩnh, tức là tìm kiếm đường đi tối ưu giữa hai điểm cụ thể mà không cần xây dựng một cây phủ toàn bộ không gian như RRT*.

2. BIT*: Thuật toán lai giữa RRT* và A*

a. Sơ qua về RRT* và A*

RRT:*

- RRT* là phiên bản cải tiến của RRT, dùng để tìm đường đi tối ưu trong không gian có chướng ngại vật.
- Cách hoạt động:
    - Tạo cây bằng cách chọn các điểm ngẫu nhiên trong không gian.
    - Kết nối điểm mới vào cây nếu đường đi hợp lệ.
    - Sau khi tìm được đường đi đầu tiên, tiếp tục tối ưu hóa bằng cách tái cấu trúc cây.

- Ưu điểm:
    - Phù hợp với không gian có chướng ngại vật phức tạp.
    - Phù hợp với không gian động, nơi chướng ngại vật có thể thay đổi theo thời gian.
    - Hội tụ dần đến lời giải tối ưu theo thời gian.
- Nhược điểm:
    - Tốn nhiều thời gian để hội tụ về đường đi tối ưu.
    - Việc mở rộng cây ngẫu nhiên có thể làm chậm quá trình tối ưu.

A:*

- A* là thuật toán tìm kiếm đường đi tối ưu dựa trên đồ thị.
- Cách hoạt động:
    - Duy trì một hàng đợi ưu tiên các nút dựa trên chi phí f(n) = g(n) + h(n), trong đó:
        - g(n): chi phí từ điểm xuất phát đến n.
        - h(n): ước lượng khoảng cách từ n đến đích (hàm heuristic).
    - Luôn mở rộng nút có giá trị f(n) nhỏ nhất trước.
- Ưu điểm:
    - Tìm đường đi tối ưu rất nhanh nếu có một hàm heuristic tốt.
- Nhược điểm:
    - Không mở rộng cây, chỉ hoạt động trên tập hợp các điểm đã biết (không gian tĩnh).
    - Khi có nhiều chướng ngại vật hoặc không gian rộng, việc tính toán chi phí có thể trở nên chậm.

b. BIT*

- Batch-Informed RRT* (BIT*) là một thuật toán lập kế hoạch đường đi được phát triển nhằm tận dụng những điểm mạnh của hai thuật toán nổi tiếng trong lĩnh vực này: *RRT** (Rapidly-exploring Random Tree Star) và *A** (A-star).
- BIT* được thiết kế để cải thiện hiệu suất tìm kiếm đường đi tối ưu, đặc biệt trong các không gian có chướng ngại vật phức tạp.
- Cả RRT* và A* đều có ưu và nhược điểm riêng:
    - *RRT** mạnh trong việc mở rộng cây tìm kiếm một cách ngẫu nhiên, giúp tìm ra đường đi khả thi nhanh chóng trong môi trường phức tạp. Tuy nhiên, nó hội tụ về nghiệm tối ưu khá chậm.

- *A** đảm bảo tìm được đường đi tối ưu với chi phí thấp nhất nếu có đủ thời gian và bộ nhớ, nhưng khi không gian tìm kiếm lớn, nó có thể trở nên chậm và không hiệu quả.

- BIT* được phát triển nhằm kết hợp ưu điểm của cả hai thuật toán này:
  - Tận dụng *khả năng mở rộng không gian nhanh chóng của RRT**.
  - Kế thừa *cấu trúc ưu tiên tìm kiếm của A** để hướng dẫn quá trình mở rộng hiệu quả hơn.
- Kết quả là một thuật toán có khả năng tìm kiếm đường đi tối ưu nhanh chóng, thích nghi tốt với những không gian lớn và phức tạp. BIT* hoạt động theo cách lặp, dần cải thiện nghiệm theo thời gian mà vẫn đảm bảo tính tối ưu như A*, nhưng có thể cập nhật và điều chỉnh linh hoạt hơn giống như RRT*.

Đặc điểm	RRT*	A*	BIT* (Kế thừa từ)
Tạo mẫu ngẫu nhiên	✓ Có	✗ Không	✓ Từ RRT*
Tối ưu hóa đường đi	✓ Có (nhưng chậm)	✓ Có	✓ Nhanh hơn RRT*
Chọn điểm mở rộng tốt nhất	✗ Ngẫu nhiên	✓ Theo chi phí `f(n)`	✓ Từ A* (hàng đợi ưu tiên)
Duy trì cấu trúc cây	✓ Có	✗ Không	✓ Từ RRT*
Hội tụ về tối ưu	Chậm	Nhanh nhưng giới hạn không gian	Nhanh hơn do chọn lọc điểm tốt nhất

3. Cách triển khai

- Ý tưởng chính:
  - Xử lý các mẫu theo lô (batch) để thực hiện tìm kiếm.
  - Sử dụng hàm heuristic để hướng dẫn hiệu quả việc tìm kiếm trong 1 loạt các RGG ẩn ngày càng dày đặc, đồng thời tái sử dụng thông tin từ các lần trước.

➔ Có nghĩa là mỗi bước sẽ tạo ra 1 tập điểm (lô) ngẫu nhiên (RRG ẩn) nhưng chưa thêm vào cây, rồi dựa vào hàm heuristic để xem xét đường đi từ đầu đến cạnh được chọn xem cạnh nào tốt nhất (chi phí ít nhất và không bị chặn) sau đó mới thêm vào node

- Tóm tắt thuật toán:
  - B1: Khởi tạo:

- Tập hợp đỉnh V ban đầu chứa trạng thái ban đầu.
- Tập hợp cạnh E ban đầu rỗng.
- Tập hợp mẫu ban đầu chứa trạng thái mục tiêu.
- Hai hàng đợi ưu tiên được sử dụng: QE (hàng đợi cạnh) và QV (hàng đợi đỉnh), ban đầu rỗng.
- Bán kính r của RGG ẩn ban đầu được đặt thành vô cùng.

o B2: Lặp:
- Sinh ra 1 tập điểm (lô) ngẫu nhiên nếu lô rỗng hoặc đã duyệt hết.
- Tiếp tục duyệt để tìm đường đi.

o B3: Dựa vào heuristic để tỉa cây:
- Nếu như g(x) + h(x) >= c_best (giá trị tốt nhất hiện tại) → loại bỏ đỉnh
- Với:
  - g(x) là đường đi ngắn nhất từ x_start đến node x.
  - h(x) là ước lượng khoảng cách từ x đến x_goal.

o B4: Tạo lô mẫu mới và đưa vào RGG:
- Nếu chưa có nghiệm (chưa tìm được đường đi nào từ start đến end) thì tiếp tục sinh theo toàn bộ không gian
- Nếu đã có 1 nghiệm thì sẽ tạo ra 1 vùng "elip thông minh" giữa start và end.

o B5: Gán nhãn và đưa vào hàng đợi:
- Với mỗi node trong cây V, nếu node có thể kết nối đến các điểm mới sinh trong batch (nằm trong bán kính r) → *được gán nhãn là "có thể mở rộng"*.
- Các node này được thêm vào hàng đợi QV.

o B6: Cập nhật lại bán kính của đồ thị RGG:
- Tính bán kính dựa vào số điểm đang có:

$$r(q) = \gamma \left(\frac{log(q)}{q}\right)^{\frac{1}{d}}$$

q: Số điểm hiện tại (Kể cả mẫu)

d: Số chiều không gian

γ: Hằng số

- Càng nhiều điểm → bán kính nhỏ hơn

o B7: Tìm kiếm heuristic:

f(u, v) = g(u) + cost(u ->v) + h(v)

- Luôn mở rộng cạnh có chi phí ước lượng nhỏ nhất.

- Ghi nhận đường đi ngắn nhất đến v (nếu có).
- Cập nhật cạnh E.
- Quá trình tiếp tục cho đến khi:
  - Tìm được đường đến x_goal mới tốt hơn.
  - Hoặc không còn cạnh nào để mở rộng → kết thúc batch.

o B8: Kết thúc batch:
- Nếu tìm được nghiệm tốt hơn → cập nhật c_best.
- Tất cả các node/mẫu không thể cải thiện thêm sẽ bị loại.
- Bắt đầu batch mới → sinh mẫu mới, cập nhật cây.

4. Code

- Pseudo code:

1. V←{x_{start}}; E←∅; T = (V, E);
2. $X_{unconn} \leftarrow X_{goal}$;
3. $Q_V \leftarrow V$; $Q_E \leftarrow \emptyset$;
4. $V_{soln} \leftarrow V \cap X_{goal}$; $V_{unexpnd} \leftarrow V$; $X_{new} \leftarrow X_{unconn}$;
5. $c_i \leftarrow min_{v_{goal} \in v_{soln}}\{gT(V_{goal})\}$;
6. repeat
7. if $Q_E \equiv \emptyset \; and \; Q_V \equiv \emptyset \; then$
8. $X_{reuse} \leftarrow Prune(T, X_{unconn}, c_i)$;
9. $X_{sampling} \leftarrow Sample(m, X_{start}, X_{goal}, c_i)$;
10. $X_{new} \leftarrow X_{sampling} \cup X_{reuse}$;
11. $X_{unconn} \overset{+}{\leftarrow} X_{new}$;
12. $Q_V \leftarrow V$;
13. while BestQueueValue(Q_V)≤ BestQueueValue(Q_E) do ExpandNextVertex(Q_V, Q_E, c_i);
14. (v_{min}, x_{min})←PopBestInQueue(Q_E);
15. if $gT(v_{min})+\hat{c}(v_{min}, x_{min})+\hat{h}(x_{min})<c_i$ then
16. if $gT(v_{min})+\hat{c}(v_{min}, x_{min})< gT(x_{min})$ then
17. c_{edge}←c(v_{min}, x_{min});
18. if $gT(v_{min})+c_{edge}+\hat{h}(x_{min})<c_i$ then
19. if $gT(v_{min})+c_{edge}<gT(x_{min})$ then
20. if $x_{min} \in V$ then
21. v_{parent}←Parent(x_{min});
22. E←{(v_{parent}, x_{min})};

23. else
24. $X_{unconn} \overset{-}{\leftarrow} \{x_{min}\}$;
25. $V \overset{+}{\leftarrow} \{x_{min}\}$;
26. $Q_V \overset{+}{\leftarrow} \{x_{min}\}$;
27. $V_{unexpnd} \overset{+}{\leftarrow} \{x_{min}\}$;
28. if $x_{min} \in X_{goal}$ then
29. $V_{sol'n} \overset{+}{\leftarrow} \{x_{min}\}$;
30. $E \overset{+}{\leftarrow} \{(v_{min}, x_{min})\}$;
31. $c_i \leftarrow min_{v_{goal} \in v_{sol'n}} \{gT(V_{goal})\}$;
32. else
33. $Q_E \leftarrow \emptyset$, $Q_V \leftarrow \emptyset$;
34. until STOP;
35. return T;

- Code:

```
#! /usr/bin/env python3
#! -*- coding: utf-8 -*-

from queue import PriorityQueue
from typing import List, Tuple
import numpy as np
import time
import json
from Node import Node
from Map import Map
from PrintColours import *


class bitstar:
    """BIT* algorithm class. This class implements the BIT* algorithm
for path planning."""

    def __init__(
        self,
        start: Node,
        goal: Node,
        occ_map: Map,
        no_samples: int = 20,
        rbit: float = 100,
        dim: int = 2,
        log_dir: str = None,
        stop_time: int = 60,
    ) -> None:
        """Initialize BIT* algorithm.

        Args:
```

```
            start (Node): Node object representing the start position.
            goal (Node): Node object representing the goal position.
            occ_map (Map): Map object representing the occupancy grid.
            no_samples (int, optional): Number of samples to be
generated in each iteration. Defaults to 20.
            rbit (float, optional): Radius of the ball to be considered
for rewire. Defaults to 100.
            dim (int, optional): Dimension of the search space.
Defaults to 2.
            log_dir (str, optional): Directory to save the log files.
Defaults to None (no logging).
            stop_time (int, optional): Time limit for the algorithm to
run in seconds. Defaults to 60s.
        """
        # Set the start node.
        self.start = start
        # Set the goal node.
        self.goal = goal
        # Set the occupancy grid.
        self.map = occ_map
        # Set the dimension of the search space.
        self.dim = dim
        # Set the radius of the ball within which the nodes are
considered to be near each other for making connections.
        self.rbit = rbit
        # Number of samples to be generated in each step in the
algorithm.
        self.m = no_samples
        # The current cost-to-come of the goal node.
        self.ci = np.inf
        # The old cost-to-come of the goal node.
        self.old_ci = np.inf
        # The minimum cost-to-come of the goal node.
        self.cmin = np.linalg.norm(self.goal.np_arr -
self.start.np_arr)
        # Used to get the length of the map.
        self.flat_map = self.map.map.flatten()
        # Set the time limit for the algorithm to run. Default is 60s.
This is used to stop the algorithm as it can only be asymptotically
optimal.
        self.stop_time = stop_time

        # Set of all the vertices in the tree.
        self.V = set()
        # Set of all the edges in the tree.
        self.E = set()
        # Set of all the vertices used for visualization.
        self.E_vis = set()
        # Set of all new vertices.
        self.x_new = set()
        # Set of all vertices that are to be reused.
        self.x_reuse = set()
        # Set of all vertices that are not expanded.
        self.unexpanded = set()

        # Set of all vertices that are not connected to the tree.
        self.unconnected = set()
        # Set of all vertices that are in the goal set.
```

```
        self.vsol = set()

        # Priority queue for the vertices.
        self.qv = PriorityQueue()
        # Priority queue for the edges.
        self.qe = PriorityQueue()
        # This is a workaround when the gt + c + h_hat values for two
edges and the order of the edges in the queue is used to break the tie.
        self.qe_order = 0
        # This is a workaround when the gt + h_hat values are the same
for two nodes and the order of the nodes in the queue is used to break
the tie.
        self.qv_order = 0

        # Add the start node to the tree.
        self.V.add(start)
        # Add the goal node to the unconnected set.
        self.unconnected.add(goal)
        # Add the start node to the unexpanded set.
        self.unexpanded = self.V.copy()
        # Add the start node to the x_new set.
        self.x_new = self.unconnected.copy()

        # Add the start node to the priority queue.
        self.qv.put((start.gt + start.h_hat, self.qv_order, start))
        # Increment the order of the priority queue.
        self.qv_order += 1
        # Get the current Prolate HyperSpheroid (PHS) for the current
cost.
        self.get_PHS()
        # Set the flag to save the log files.
        self.save = False
        # If the log directory is not None, then save the log files.
        if log_dir is not None:
            # Reset the save flag.
            self.save = True
            # Get the path to the log directory.
            self.log_dir = log_dir
            # Template Dictionary to store the contents as a JSON in
the log file.
            self.json_contents = {
                "new_edges": [],
                "rem_edges": [],
                "final_path": [],
                "ci": [],
                "final_edge_list": [],
            }

    def gt(self, node: Node) -> float:
        """Get the cost of the path from the start to the node by
traversing through the Tree.

        Args:
            node (Node): The node for which the cost is to be
calculated.

        Returns:
```

```
            g_t (float): The cost of the path from the start to the
node by traversing through the Tree.
        """
        # If the node is the start node, then the cost is 0.
        if node == self.start:
            return 0
        # If the node is not in the tree, then the cost is infinity.
        elif node not in self.V:
            return np.inf
        # Return the cost of the path from the start to the node by
traversing through the Tree. This is the sum of the cost of the edge
from the parent to the node and the cost of the path from the start to
the parent.
        return node.par_cost + node.parent.gt

    def c_hat(self, node1: Node, node2: Node) -> float:
        """Estimated cost of the edge from node1 to node2 using L2
norm.

        Args:
            node1 (Node): The first node.
            node2 (Node): The second node.

        Returns:
            c_hat (float): The estimated L2 norm cost of the straight
line path from node1 to node2.
        """
        # Return the L2 norm of the difference between the two nodes.
        return np.linalg.norm(node1.np_arr - node2.np_arr)

    def a_hat(self, node1: Node, node2: Node) -> float:
        """This is the sum of the estimated cost of the path from start
to node1 (L2 Norm), the estimated cost of the path from node1 to node2
(L2 norm), and the heuristic cost (L2 Norm) of node2 from goal.

        Args:
            node1 (Node): The first node.
            node2 (Node): The second node.

        Returns:
            g_hat(node1) + c_hat(node1, node2) + h_hat(node2) (float):
The total estimated cost.
        """
        # Return the sum of the estimated cost of the path from start
to node1 (L2 Norm), the estimated cost of the path from node1 to node2
(L2 norm), and the heuristic cost (L2 Norm) of node2 from goal.
        return node1.g_hat + self.c_hat(node1, node2) + node2.h_hat

    def c(self, node1: Node, node2: Node, scale: int = 10) -> float:
        """True cost of the edge between node1 and node2. This is the
L2 norm cost of the straight line path from node1 to node2. If the path
is obstructed, the cost is set to infinity.

        Args:
            node1 (Node): The first node.
            node2 (Node): The second node.
            scale (int, optional): The number of divisions to be made
in the straight line path to check for obstacles. Defaults to 10.
```

```
        Returns:
            c(float): The true cost of the edge between node1 and
node2.
        """
        # Get the coordinates of the two nodes.
        x1, y1 = node1.tup
        x2, y2 = node2.tup

        # Get the number of divisions to be made in the straight line
path to check for obstacles.
        n_divs = int(scale * np.linalg.norm(node1.np_arr -
node2.np_arr))

        # For each division, check if the node is occupied. If it is,
then return infinity for the whole edge.
        for lam in np.linspace(0, 1, n_divs):
            # Using the parametric equation of the line, get the
coordinates of the node.
            x = int(x1 + lam * (x2 - x1))
            y = int(y1 + lam * (y2 - y1))
            # If the node is occupied, then return infinity.
            if (x, y) in self.map.occupied:
                return np.inf
        # Return the L2 norm of the difference between the two nodes if
the edge is not obstructed.
        return self.c_hat(node1, node2)

    def near(self, search_set: set, node: Node) -> set:
        """Returns the set of nodes in the search_set which are within
the radius of the ball centered at node (rbit). The node itself is not
included in the returned set.

        Args:
            search_set (set): The set of nodes to be searched for near
nodes.
            node (Node): The node about which the ball is centered.

        Returns:
            Near Nodes (set): The set of nodes in the search_set which
are within the radius of the ball centered at node (rbit).
        """
        # Set to store the near nodes.
        near = set()
        # For each node in the search_set, check if it is within the
radius of the ball centered at node (rbit). If it is, then add it to
the set of near nodes.
        for n in search_set:
            if (self.c_hat(n, node) <= self.rbit) and (n != node):
                near.add(n)
        # Return the set of near nodes.
        return near

    def expand_next_vertex(self) -> None:
        """Expands the next vertex in the queue of vertices to be
expanded (qv). This function is called by the main loop of the
algorithm."""
```

```
        # Get the next vertex to be expanded from the queue of vertices
to be expanded (qv).
        vmin = self.qv.get(False)[2]
        # Set of nodes in the Tree which are within the radius of the
ball centered at vmin (rbit).
        x_near = None
        # If vmin is in the set of unexpanded nodes, then the set of
near nodes is the set of unconnected nodes which are within the radius
of the ball centered at vmin (rbit).
        if vmin in self.unexpanded:
            x_near = self.near(self.unconnected, vmin)
        # Else, the set of near nodes is the intersection of the set of
unconnected nodes and the set of new nodes which are within the radius
of the ball centered at vmin (rbit).
        else:
            intersect = self.unconnected & self.x_new
            x_near = self.near(intersect, vmin)

        for x in x_near:
            # Edge is added to the queue of edges if the edge is
estimated cost less than the current cost (ci).
            if self.a_hat(vmin, x) < self.ci:
                # Actual cost of the edge is calculated.
                cost = vmin.gt + self.c(vmin, x) + x.h_hat
                # Edge is added to the queue of edges.
                self.qe.put((cost, self.qe_order, (vmin, x)))
                self.qe_order += 1

        if vmin in self.unexpanded:
            # Gets the set of nodes near vmin that is already in the
Tree.
            v_near = self.near(self.V, vmin)
            for v in v_near:
                # For all nodes in the near list. If the edge is not in
the all edges set, and the estimated cost of the edge is less than the
current cost (ci), and the estimated cost of the path from start to v
and
                if (
                    (not (vmin, v) in self.E)
                    and (self.a_hat(vmin, v) < self.ci)
                    and (vmin.g_hat + self.c_hat(vmin, v) < v.gt)
                ):
                    # Cost of the edge is calculated.
                    cost = vmin.gt + self.c(vmin, v) + v.h_hat
                    # Edge is added to the queue of edges.
                    self.qe.put((cost, self.qe_order, (vmin, v)))
                    self.qe_order += 1
            # Vertex is removed from the set of unexpanded nodes.
            self.unexpanded.remove(vmin)

    def sample_unit_ball(self) -> np.array:
        """Samples a point uniformly from the unit ball. This is used
to sample points from the Prolate HyperSpheroid (PHS).

        Returns:
            Sampled Point (np.array): The sampled point from the unit
ball.
        """
```

```
        u = np.random.uniform(-1, 1, self.dim)
        norm = np.linalg.norm(u)
        r = np.random.random() ** (1.0 / self.dim)
        return r * u / norm

    def samplePHS(self) -> np.array:
        """Samples a point from the Prolate HyperSpheroid (PHS) defined
by the start and goal nodes.

        Returns:
            Node: The sampled node from the PHS.
        """
        # Calculate the center of the PHS.
        center = (self.start.np_arr + self.goal.np_arr) / 2
        # The transverse axis in the world frame.
        a1 = (self.goal.np_arr - self.start.np_arr) / self.cmin
        # The first column of the identity matrix.
        one_1 = np.eye(a1.shape[0])[:, 0]
        U, S, Vt = np.linalg.svd(np.outer(a1, one_1.T))
        Sigma = np.diag(S)
        lam = np.eye(Sigma.shape[0])
        lam[-1, -1] = np.linalg.det(U) * np.linalg.det(Vt.T)
        # Calculate the rotation matrix.
        cwe = np.matmul(U, np.matmul(lam, Vt))
        # Get the radius of the first axis of the PHS.
        r1 = self.ci / 2
        # Get the radius of the other axes of the PHS.
        rn = [np.sqrt(self.ci**2 - self.cmin**2) / 2] * (self.dim - 1)
        # Create a vector of the radii of the PHS.
        r = np.array([r1] + rn)

        # Sample a point from the PHS.
        while True:
            try:
                # Sample a point from the unit ball.
                x_ball = self.sample_unit_ball()
                # Transform the point from the unit ball to the PHS.
                op = np.matmul(np.matmul(cwe, r), x_ball) + center
                # Round the point to 7 decimal places.
                op = np.around(op, 7)
                # Check if the point is in the PHS.
                if (int(op[0]), int(op[1])) in self.intersection:
                    break
            except:
                print(CBOLD, CRED2, op, x_ball, r, self.cmin, self.ci,
cwe, CEND)
                exit()

        return op

    def get_PHS(self) -> None:
        """Generates the latest Prolate HyperSpheroid (PHS) for a new
cost threshold (ci)."""
        # Get the set of points in the PHS.
        self.xphs = set([tuple(x) for x in
np.argwhere(self.map.f_hat_map < self.ci)])
        # Get the set of points that are in the PHS and free space.
        self.intersection = self.xphs & self.map.free
```

```
    def sample(self) -> Node:
        """Samples a node from the Prolate HyperSpheroid (PHS) or the
free space of the map depending on the current cost (ci).

        Returns:
            Node: The sampled node from the PHS or the free space of
the map.
        """
        # A random point is sampled from the PHS.
        xrand = None
        # Do not generate a new PHS if the cost threshold (ci) has not
changed.
        if self.old_ci != self.ci:
            self.get_PHS()

        # If the cardinality of the PHS is less than the cardinality of
the free space, sample from the PHS.
        if len(self.xphs) < len(self.flat_map):
            xrand = self.samplePHS()
        # Else sample from the free space.
        else:
            xrand = self.map.new_sample()
        # Return the sampled node as a Node object.
        return Node(xrand)

    def prune(self) -> None:
        """Prunes the search tree based on the current cost threshold
(ci). It removes all nodes from the search tree which have a f_hat
value greater than the current cost threshold (ci). It also removes all
edges which connect to a node which has been removed from the search
tree."""
        # Set of removed nodes from the search tree but can be reused.
        self.x_reuse = set()
        # Remove all nodes from the search tree which have a f_hat
value greater than the current cost threshold (ci).
        new_unconnected = set()
        for n in self.unconnected:
            if n.f_hat < self.ci:
                new_unconnected.add(n)
        self.unconnected = new_unconnected

        # A list of removed edges from the search tree. This is used to
update the visualization.
        rem_edge = []
        # Sort the nodes in the search tree by their g_t value.
        sorted_nodes = sorted(self.V, key=lambda x: x.gt, reverse=True)
        # Remove all nodes from the search tree which have a f_hat
value greater than the current cost threshold (ci). Also remove all
nodes which have a g_t + h_hat value greater than the current cost
threshold (ci).
        for v in sorted_nodes:
            # Do not remove the start or goal nodes.
            if v != self.start and v != self.goal:
                if (v.f_hat > self.ci) or (v.gt + v.h_hat > self.ci):
                    self.V.discard(v)
                    self.vsol.discard(v)
                    self.unexpanded.discard(v)
```

```
                    self.E.discard((v.parent, v))
                    self.E_vis.discard((v.parent.tup, v.tup))
                    # If the save flag is set to True, add the removed
edge to the list of removed edges.
                    if self.save:
                        rem_edge.append((v.parent.tup, v.tup))
                    v.parent.children.remove(v)
                    # Add the removed node to the set of nodes which
can be reused if the node's f_hat < ci.
                    if v.f_hat < self.ci:
                        self.x_reuse.add(v)
                    else:
                        # If the node's f_hat > ci we delete the node.
                        del v
        # If the save flag is set to True, save the removed edges.
        if self.save:
            self.save_data(None, rem_edge)
        # Add the  goal node back to the set of unexpanded nodes.
        self.unconnected.add(self.goal)

    def final_solution(self) -> Tuple[List[Tuple[float, float]],
float]:
        """Returns the final solution path and the path length.

        Returns:
            Tuple[List[Tuple[float, float]], float]: The final solution
path and the path length.
        """
        # If the goal node has an infinite g_t value, then there is no
solution.
        if self.goal.gt == np.inf:
            return None, None
        # Empty list to store the solution path.
        path = []
        # Path length is initialized to 0.
        path_length = 0
        # Start from the goal node and traverse the parent nodes until
the start node is reached.
        node = self.goal
        while node != self.start:
            path.append(node.tup)
            path_length += node.par_cost
            node = node.parent
        # Add the start node to the path.
        path.append(self.start.tup)
        # Reverse the path and return the path and the path length.
        return path[::-1], path_length

    def update_children_gt(self, node: Node) -> None:
        """Updates the true cost of a node from start (gt) of all the
children of a node in the search tree. This is used when an edge is
added/removed from the search tree.

        Args:
            node (Node): The node whose children's true cost needs to
be updated.
        """
        # Update the true cost of the children of the node.
```

```
        for c in node.children:
            # The true cost of the child is the true cost of the parent
+ the cost of the edge connecting the parent and the child.
            c.gt = c.par_cost + node.gt
            # Recursively update the true cost of the children of the
child.
            self.update_children_gt(c)

    def save_data(
        self, new_edge: tuple, rem_edge: list, new_final: bool = False
    ) -> None:
        """Saves the data as a JSON file for the current iteration of
the algorithm. It is used to generate the plots and animations.

        Args:
            new_edge (tuple): The new edge added to the search tree.
            rem_edge (list): The list of edges removed from the search
tree.
            new_final (bool, optional): Whether the final solution path
has changed. Defaults to False.
        """
        # Update the current cost (ci).
        self.json_contents["ci"].append(self.ci)
        # New edges added to the search tree.
        self.json_contents["new_edges"].append(new_edge)
        # Removed edges from the search tree.
        self.json_contents["rem_edges"].append(rem_edge)

        # If the final solution path has changed, update the final
solution path.
        if new_final:
            # Get the final solution path and the path length.
            current_solution, _ = self.final_solution()
            # Add the final solution path to the JSON file.
            self.json_contents["final_path"].append(current_solution)
        else:
            # If the final solution path has not changed, add None to
the JSON file.
            self.json_contents["final_path"].append(None)

    def dump_data(self, goal_num: int) -> None:
        """Dumps the data as a JSON file for the current simulation
run.

        Args:
            goal_num (int): The Simulation run number. This is used to
name the JSON file. The JSON file is saved in the log directory.
        """
        print(f"{CGREENBG}Data saved.{CEND}")
        # Add the final edge list to the JSON file.
        self.json_contents["final_edge_list"] = list(self.E_vis)

        # Converting json_contents to json object.
        json_object = json.dumps(self.json_contents, indent=4)

        # Open a file and dump the JSON object.
        with open(
            f"{self.log_dir}/path{goal_num:02d}.json",
```

```
            "w",
        ) as outfile:
            # Write the JSON object to the file.
            outfile.write(json_object)

        # Reset the JSON contents.
        self.json_contents = {
            "new_edges": [],
            "rem_edges": [],
            "final_path": [],
            "ci": [],
            "final_edge_list": [],
        }

    def make_plan(self) -> Tuple[List[Tuple[float, float]], float,
List[float]]:
        """The main BIT* algorithm. It runs the algorithm until the
time limit is reached. It also saves the data for the current
simulation run if the save flag is set to True.

        Returns:
            Tuple[List[Tuple[int, int]], float, List[float]]: The final
solution path, the path length and the list of time taken for each
iteration.
        """
        # If the start or goal is not in the free space, return None.
        if self.start.tup not in self.map.free or self.goal.tup not in
self.map.free:
            print(f"{CYELLOW2}Start or Goal not in free space.{CEND}")
            return None, None, None

        # If the start and goal are the same, return the start node,
path length 0 and None for the time taken.
        if self.start.tup == self.goal.tup:
            print(f"{CGREEN2}Start and Goal are the same.{CEND}")
            self.vsol.add(self.start)
            self.ci = 0
            return [self.start.tup], 0, None

        # Initialize the iteration counter.
        it = 0
        # Initialize the number of times the goal has been reached.
        goal_num = 0
        # Start the timer for the algorithm.
        plan_time = time.time()
        # Start the timer for the current simulation run.
        start = time.time()

        # List to store the time taken for each simulation run.
        time_taken = []
        try:
            # Start the main loop of the algorithm.
            while True:
                # If the time limit is reached, return the final
solution path.
                if time.time() - plan_time >= self.stop_time:
                    print(
```

```
f"\n\n{CITALIC}{CYELLOW2}==========================================
Stopping due to time limit
==========================================={CEND}"
                    )
                    path, path_length = self.final_solution()
                    return path, path_length, time_taken

                # Increment the iteration counter.
                it += 1
                # If the Edge queue and the Vertex queue are empty.
                if self.qe.empty() and self.qv.empty():
                    # Prune the search tree.
                    self.prune()
                    # Set of Sampled nodes.
                    x_sample = set()
                    # Sample m nodes.
                    while len(x_sample) < self.m:
                        x_sample.add(self.sample())
                    # Add the sampled nodes and reuse nodes to the new
nodes set.
                    self.x_new = self.x_reuse | x_sample
                    # Add the new nodes to the unconnected set.
                    self.unconnected = self.unconnected | self.x_new
                    for n in self.V:
                        # Add all the nodes in the search tree to the
Vertex queue.
                        self.qv.put((n.gt + n.h_hat, self.qv_order, n))
                        self.qv_order += 1

                while True:
                    # Run until the vertex queue is empty.
                    if self.qv.empty():
                        break
                    # Expand the next vertex.
                    self.expand_next_vertex()

                    # If the Edge queue is empty, continue.
                    if self.qe.empty():
                        continue
                    if self.qv.empty() or self.qv.queue[0][0] <=
self.qe.queue[0][0]:
                        break

                # If the Edge queue is empty, continue.
                if not (self.qe.empty()):
                    # Pop the next edge from the Edge queue.
                    (vmin, xmin) = self.qe.get(False)[2]
                    # The Four conditions for adding an edge to the
search tree given in the paper.
                    if vmin.gt + self.c_hat(vmin, xmin) + xmin.h_hat <
self.ci:
                        if vmin.gt + self.c_hat(vmin, xmin) < xmin.gt:
                            # Calculate the cost of the edge.
                            cedge = self.c(vmin, xmin)
                            if vmin.gt + cedge + xmin.h_hat < self.ci:
                                if vmin.gt + cedge < xmin.gt:
```

```
                                        # Remove the edge from the search
tree.
                                        rem_edge = []
                                        # If the node is in the search tree
remove the edge.
                                        if xmin in self.V:
                                            # Remove the edge from the edge
set.
                                            self.E.remove((xmin.parent,
xmin))
                                            # Remove the edge from the edge
set for the JSON file. Done in a funny way.

self.E_vis.remove((xmin.parent.tup, xmin.tup))
                                            # A funny way to remove node
xmin from the children of its parent.

xmin.parent.children.remove(xmin)
                                            # Add the edge to the list of
removed edges for the JSON file.

rem_edge.append((xmin.parent.tup, xmin.tup))
                                            # Update the parent of the
node.
                                            xmin.parent = vmin
                                            # Update the cost of the edge.
                                            xmin.par_cost = cedge
                                            # Get the new gt value of the
node.
                                            xmin.gt = self.gt(xmin)
                                            # Add the edge to search
tree/edge set.
                                            self.E.add((xmin.parent, xmin))
                                            # Add the edge to the search
tree/edge set for the JSON file. Done in a funny way.

self.E_vis.add((xmin.parent.tup, xmin.tup))
                                            # A funny way to add node xmin
to the children of its new parent.
                                            xmin.parent.children.add(xmin)
                                            # Update the gt values of the
children of the node.
                                            self.update_children_gt(xmin)
                                        else:
                                            # Add the node to the search
tree/vertex set.
                                            self.V.add(xmin)
                                            # Update the parent of the
node.
                                            xmin.parent = vmin
                                            # Update the cost of the edge.
                                            xmin.par_cost = cedge
                                            # Get the new gt value of the
node.
                                            xmin.gt = self.gt(xmin)
                                            # Add the edge to search
tree/edge set. Done in a funny way.
                                            self.E.add((xmin.parent, xmin))
```

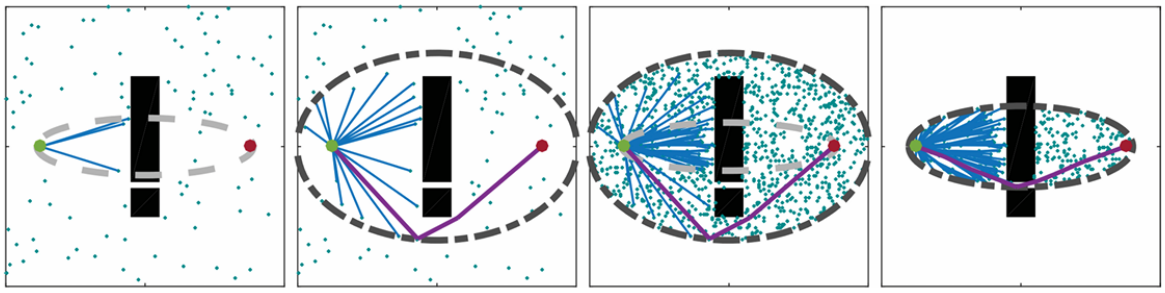
```
                                        # Add the edge to the search
tree/edge set for the JSON file. Done in a funny way.

self.E_vis.add((xmin.parent.tup, xmin.tup))
                                        self.qv_order += 1  # Why is
this here?
                                        # Add the node to the
Unexpanded set.
                                        self.unexpanded.add(xmin)
                                        # If the node is the goal, add
it to the solution set.
                                        if xmin == self.goal:
                                            # Solution set.
                                            self.vsol.add(xmin)
                                        # A funny way to add node xmin
to the children set of its parent.
                                        xmin.parent.children.add(xmin)
                                        # Remove the node from the
unconnected set.
                                        self.unconnected.remove(xmin)

                                    # Create a new edge for the JSON
file.
                                    new_edge = (xmin.parent.tup,
xmin.tup)
                                    # Set the ci to max of the goal gt
and the cmin. This is done so that in weird cases where the goal gt is
very close to cmin and due to the float inaccuracy the goal gt is less
than cmin, causing the algorithm to crash.
                                    self.ci = max(self.goal.gt,
self.cmin)

                                    # if the save flag is set, save the
data.
                                    if self.save:
                                        self.save_data(
                                            new_edge, rem_edge, self.ci
!= self.old_ci
                                        )
                                    # If there is a change in the ci
value. The algorithm has found a new solution.
                                    if self.ci != self.old_ci:
                                        # If the time limit is reached,
return the current solution.
                                        if time.time() - plan_time >=
self.stop_time:
                                            print(

f"\n\n{CITALIC}{CYELLOW2}==========================================
Stopping due to time limit
=========================================={CEND}"
                                            )
                                            path, path_length =
self.final_solution()
                                            return path, path_length,
time_taken
                                        # Print the solution.
                                        print(
```

```
f"\n\n{CBOLD}{CGREEN2}==============================================
GOAL FOUND {goal_num:02d} times
=============================================={CEND}"
                                        )
                                        # The time taken to find the
solution.
                                        print(
                                            f"{CBLUE2}Time Taken:{CEND}
{time.time() - start}",
                                            end="\t\t",
                                        )
                                        # Append the time taken to the
list of time taken.
                                        time_taken.append(time.time() -
start)

                                        # Reset the start time for the
next solution.
                                        start = time.time()
                                        # Get the solution path and the
length of the path.
                                        solution, length =
self.final_solution()
                                        # Print the solution length.
                                        print(
                                            f"{CBLUE2}Path
Length:{CEND} {length}{CEND}"
                                        )
                                        # Print the old_ci, new_ci, ci
- cmin, and the difference in the ci values.
                                        print(
                                            f"{CBLUE2}Old CI:{CEND}
{self.old_ci}\t{CBLUE2}New CI:{CEND} {self.ci}\t{CBLUE2}ci -
cmin:{CEND} {round(self.ci - self.cmin, 5)}\t {CBLUE2}Difference in
CI:{CEND} {round(self.old_ci - self.ci, 5)}"
                                        )
                                        # Print the solution path.
                                        print(f"{CBLUE2}Path:{CEND}
{solution}")
                                        # Set the old_ci to the new_ci.
                                        self.old_ci = self.ci
                                        # If the save flag is set, Dump
the data.
                                        if self.save:
                                            self.dump_data(goal_num)
                                        # Increment the goal number.
                                        goal_num += 1

                    else:
                        # Reset the Edge queue and the Vertex queue.
                        self.qe = PriorityQueue()
                        self.qv = PriorityQueue()

                else:
                    # Reset the Edge queue and the Vertex queue.
                    self.qe = PriorityQueue()
                    self.qv = PriorityQueue()
```

```
        except KeyboardInterrupt:
            # If the user presses Ctrl+C, return the current solution
path, the time taken to find the solution, and the path length.
            print(time.time() - start)
            print(self.final_solution())
            path, path_length = self.final_solution()
            return path, path_length, time_taken
```

5. Hoạt động

- So với RRT* (Mở rộng từng node một của một lô - 1 sample/batch) thì BIT* sẽ hoạt động theo hướng tạo ra 1 lô (batch) các điểm ngẫu nhiên phân bố đều trong không gian trạng thái trước khi tìm kiếm.
- Cách tạo lô ngẫu nhiên:
    - Ban đầu, thuật toán sinh ra một tập hợp các điểm ngẫu nhiên phân bố đều trong toàn bộ không gian tìm kiếm.
    - Khi đã tìm thấy một đường đi ban đầu, *BIT thu hẹp không gian lại thành một tập hợp nhỏ hơn* (gọi là *informed set* – tập hợp có thể chứa đường đi tốt hơn).
    - Trong các lần lặp tiếp theo, thuật *toán chỉ lấy các điểm ngẫu nhiên trong informed set*, thay vì toàn bộ không gian.
- Cách BIT tìm đường đi*
    - BIT* tìm kiếm theo *chất lượng giải pháp*, có nghĩa là:
        - *Sắp xếp các cạnh tiềm năng*: Thuật toán sắp xếp các cạnh trong đồ thị hiện tại dựa trên giá trị *heuristic* (đánh giá tiềm năng của một cạnh giúp cải thiện đường đi).
        - *Mở rộng theo thứ tự ưu tiên*: Các cạnh gần với đường đi tốt hơn sẽ được mở rộng trước.
        - *Cập nhật và tối ưu hóa*: Nếu tìm thấy một đường đi tốt hơn, thuật toán sẽ *loại bỏ các điểm không còn hữu ích*, thu hẹp không gian tìm kiếm và tăng độ chính xác bằng cách thêm nhiều điểm hơn.
- Ví dụ 1:



The search of a batch expands outwards from the minimum solution propsed by a heuristic. (a)

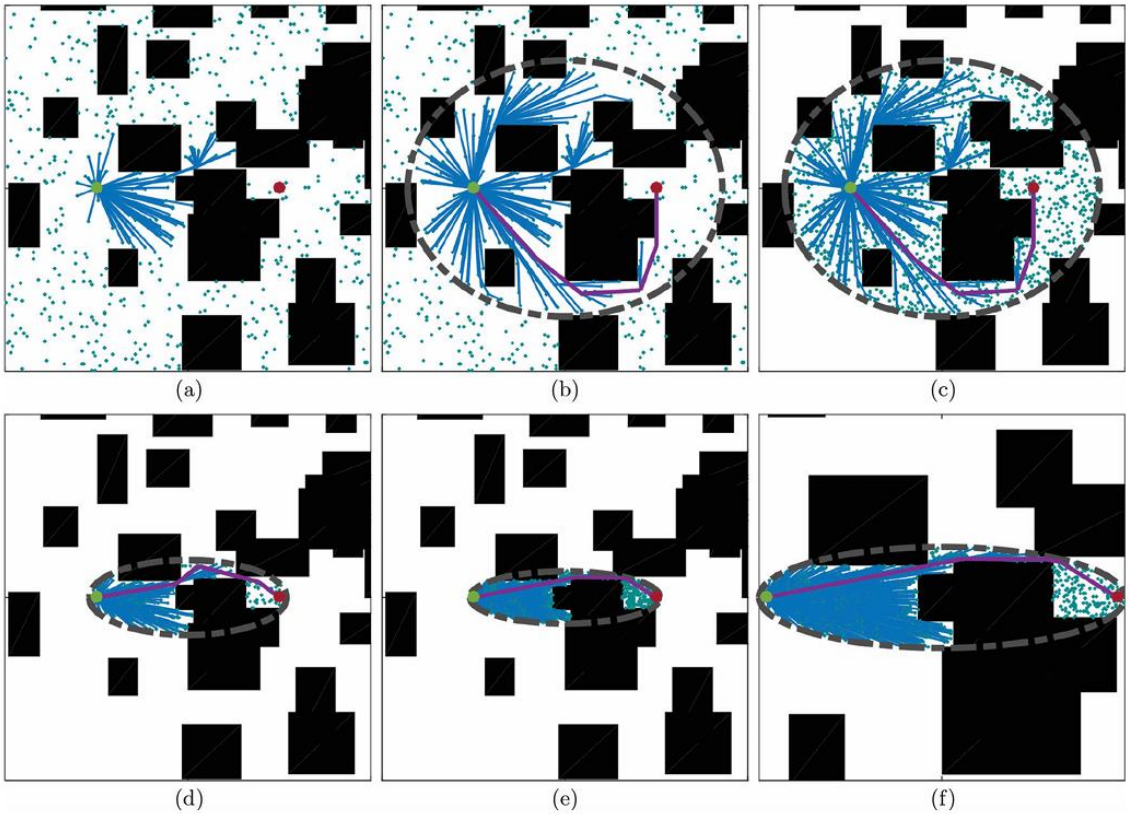
The search stops when a solution is found or the batch of samples has been completely searched. (b)

A batch of new samples is then added and the search resumes from the minimum solution. (c)

This process repeats indefinitely, focusing both the approximation of the domain and its search. (d)

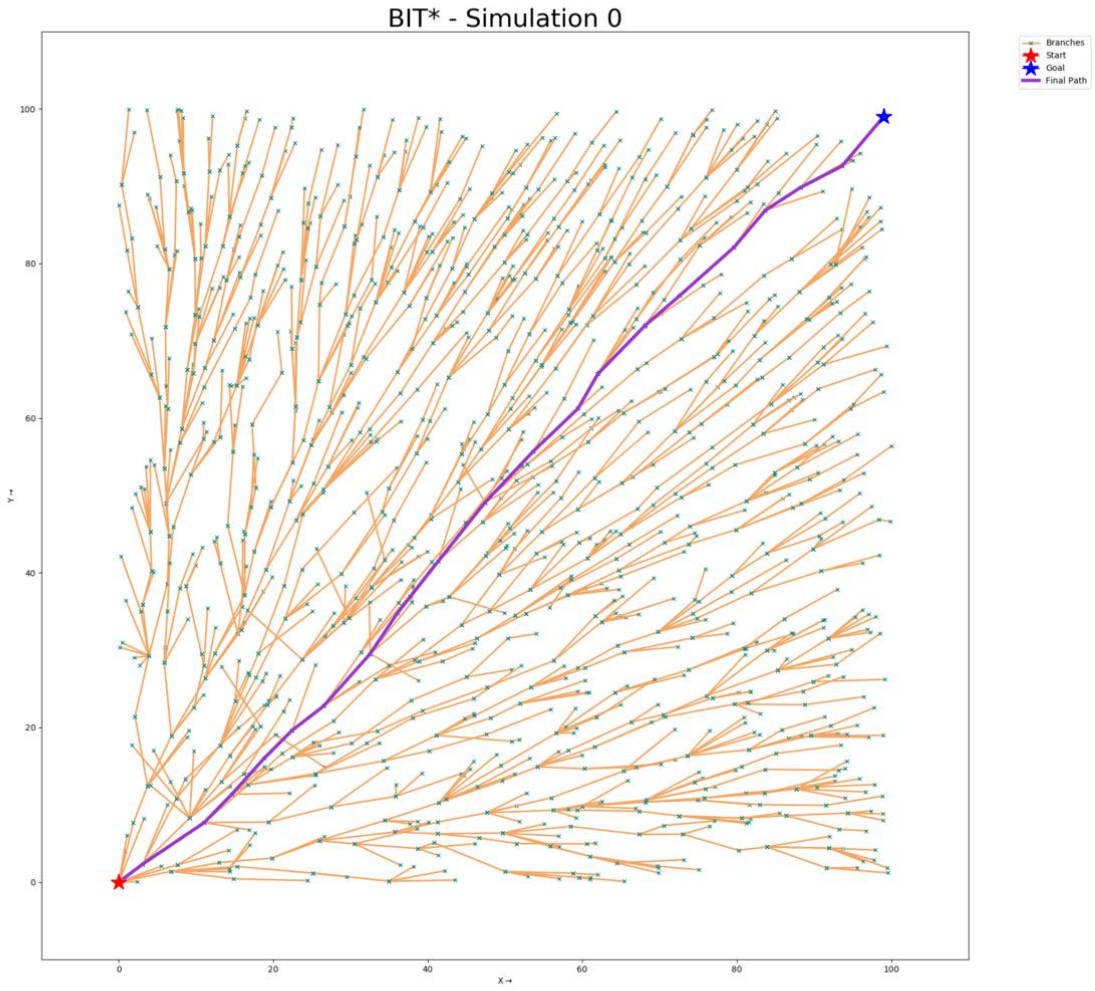
- Một ví dụ đơn giản về cách tìm kiếm dựa trên lấy mẫu có thứ tự của BIT*. Trạng thái bắt đầu và mục tiêu được hiển thị lần lượt bằng màu xanh lá cây và màu đỏ, trong khi đường đi hiện tại được tô màu tím. Tập hợp các trạng thái có thể cung cấp một giải pháp tốt hơn (tập hợp được thông tin) được biểu diễn bằng đường nét đứt màu xám đậm, trong khi tiến trình của lô mẫu hiện tại được minh họa bằng đường nét đứt màu xám nhạt, cho thấy tập hợp được thông tin mà cạnh hiện tại xác định.
    - (a): Tìm kiếm của lô mẫu đầu tiên mở rộng ra ngoài theo hướng dẫn của một heuristic.
    - (b): Quá trình này kết thúc khi tìm thấy một lời giải. Lưu ý rằng việc sắp xếp thứ tự tìm kiếm giúp tìm ra lời giải mà không cần xét đến bất kỳ trạng thái nào ngoài tập hợp được thông tin mà nó xác định.
    - (c): Tìm kiếm tiếp tục sau khi cắt bớt các trạng thái không cần thiết và tăng độ chính xác bằng cách lấy thêm mẫu.
    - (d): Quá trình tìm kiếm thứ hai dừng lại khi tìm thấy một giải pháp được cải thiện hơn.

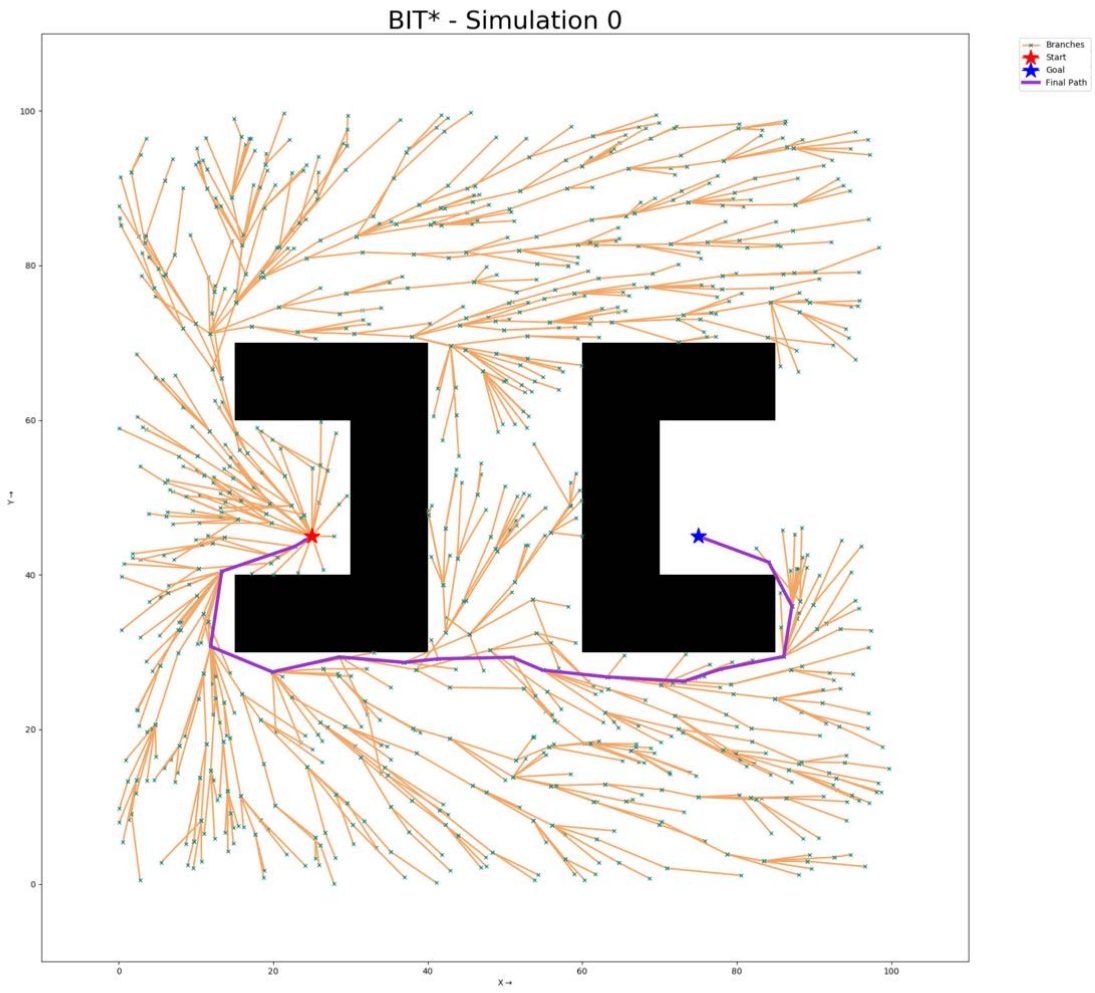
- Ví dụ 2:

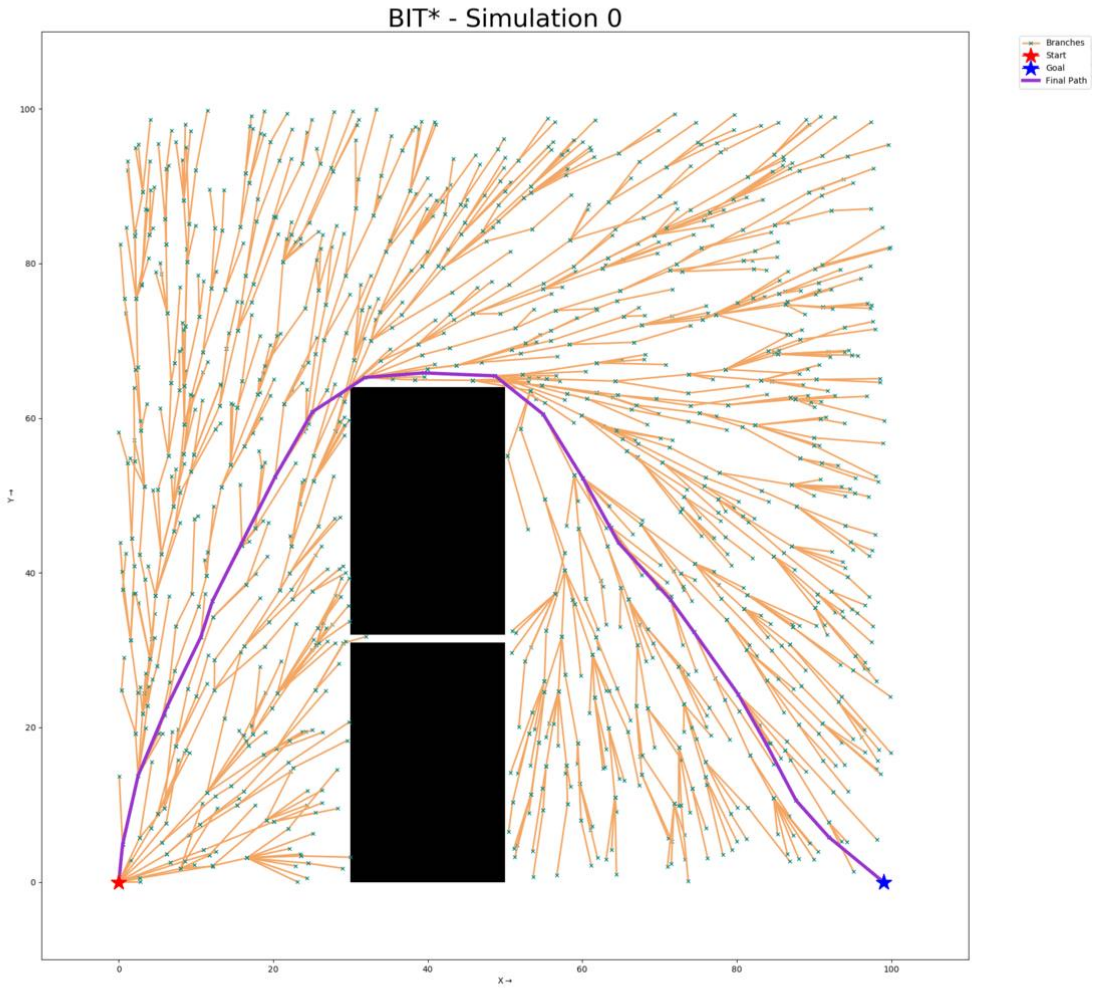


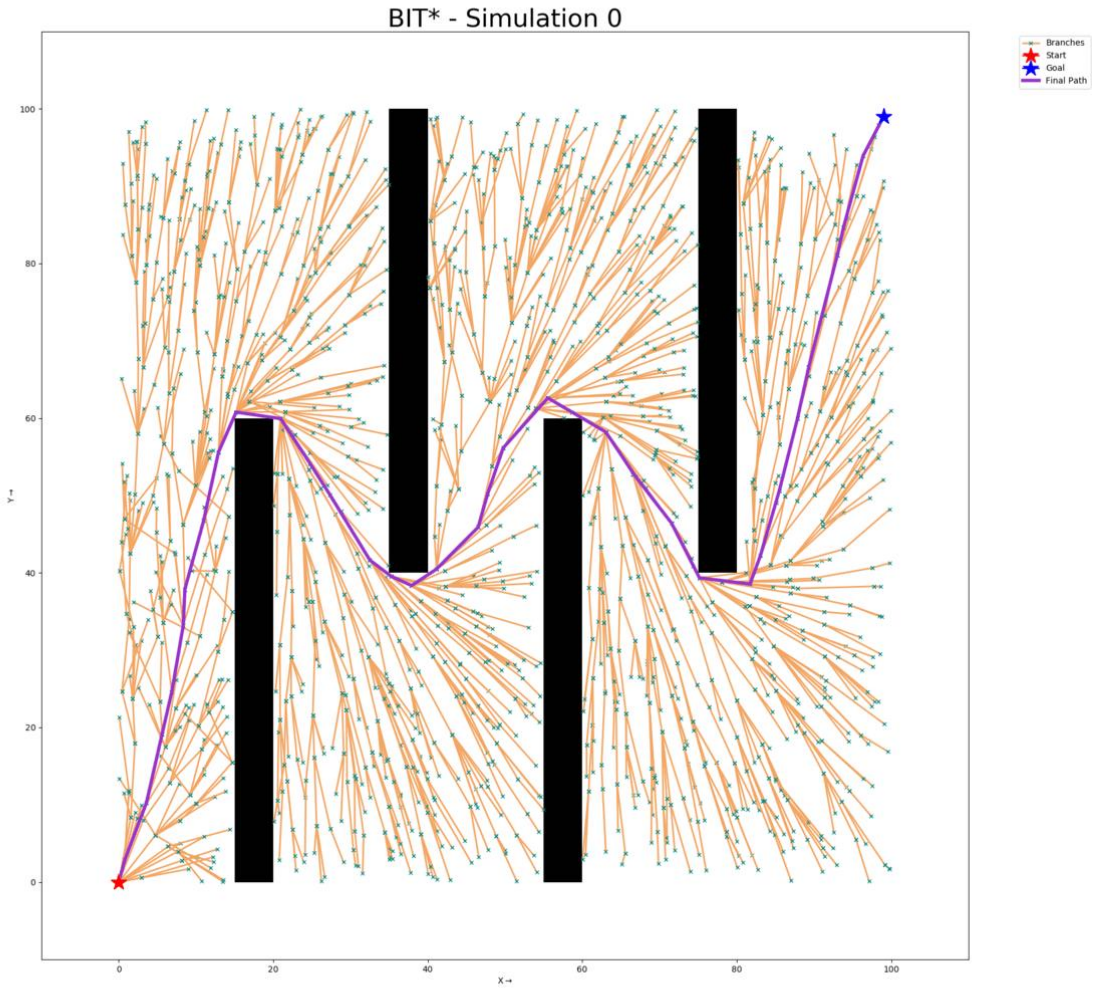
- Một ví dụ về cách thuật toán BIT* sử dụng các kỹ thuật gia tăng để tìm kiếm hiệu quả theo thứ tự chất lượng giải pháp tiềm năng từ các lô mẫu. Quá trình tìm kiếm bắt đầu với một lô mẫu được phân bố đều trong không gian lập kế hoạch. Lô này được tìm kiếm theo hướng từ điểm xuất phát theo thứ tự chất lượng giải pháp tiềm năng (hình a). Tìm kiếm tiếp tục cho đến khi lô mẫu được sử dụng hết hoặc tìm được một giải pháp không thể cải thiện với các mẫu hiện tại (hình b). Sau đó, các mẫu mới được thêm vào tập hợp được thông tin và các kỹ thuật gia tăng được sử dụng để tiếp tục tìm kiếm (hình c – e). Điều này tạo ra một thuật toán thực hiện tìm kiếm có thứ tự bất cứ lúc nào và gần như chắc chắn hội tụ tiệm cận đến giải pháp tối ưu, được hiển thị phóng to trong (hình f). Lưu ý rằng BIT* sắp xếp tất cả các khía cạnh của tìm kiếm và không bao giờ xem xét các trạng thái trong một lô mà không thể cung cấp giải pháp tốt nhất (tức là chỉ tìm kiếm trong tập hợp được thông tin xác định bởi giải pháp cuối cùng trong đồ thị hiện tại).

6. Minh họa


BIT* - Simulation 0
Branches
Start
Goal
Final Path
100
80
60
40
20
0
Y→
0
20
40
60
80
100
X→

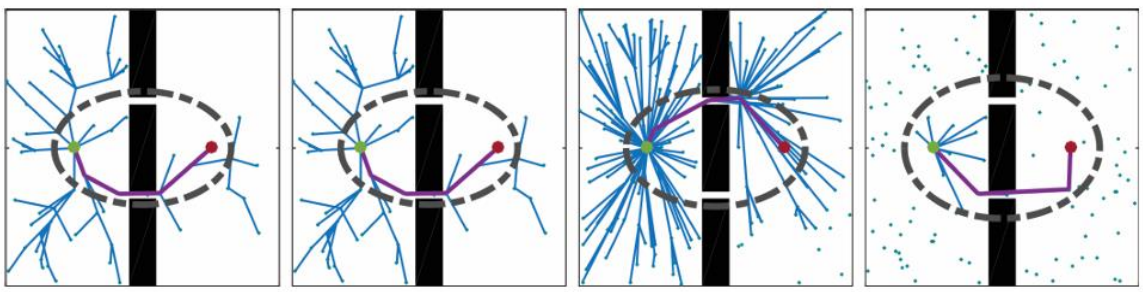

BIT* - Simulation 0
Branches
Start
Goal
Final Path
Y→
X→
0
20
40
60
80
100


BIT* - Simulation 0
Branches
Start
Goal
Final Path
Y→
X→
0
20
40
60
80
100



7. Nhận xét

a. Nhìn chung



(a) RRT* (b) RRT # (c) FMT* (d) BIT*

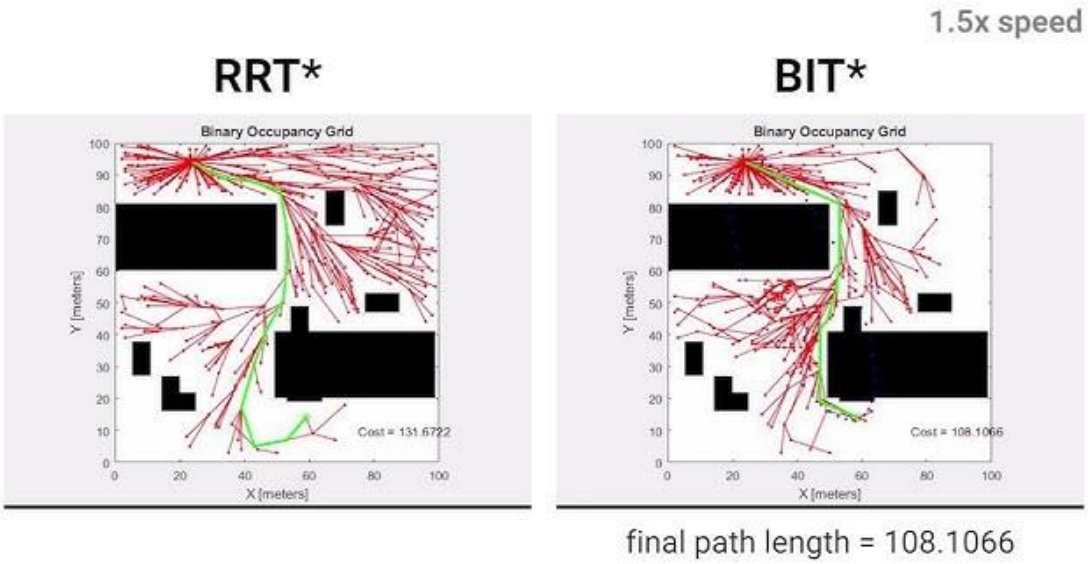
- Hiệu suất tương đối của các thuật toán tìm kiếm RRT* (a), BIT* (d) được minh họa thông qua lời giải ban đầu và tập thông tin thu được. Mỗi thuật toán được chạy cho đến khi tìm thấy một lời giải ban đầu, sau đó dừng lại. Tập hợp các trạng thái có thể mang lại lời giải tốt hơn được hiển thị bằng đường nét đứt màu xám đậm.
- Việc tìm kiếm bên ngoài tập hợp này được chứng minh là không cần thiết đối với lời giải được tìm thấy bởi từng thuật toán, minh họa sự kém hiệu quả của quá trình tìm kiếm ban đầu trong RRT*.
- Ngoài ra, bằng cách sắp xếp thứ tự tìm kiếm dựa trên chất lượng lời giải tiềm năng, BIT* không xem xét bất kỳ mẫu nào không thể cung cấp lời giải tốt nhất trong xấp xỉ hiện tại của nó (tức là trong lô mẫu hiện tại).
- Trạng thái bắt đầu và trạng thái đích của bài toán được hiển thị lần lượt bằng màu xanh lá cây và màu đỏ, trong khi đồ thị được xây dựng bởi thuật toán được hiển thị bằng màu xanh lam và lời giải ban đầu được đánh dấu bằng màu tím.

b. Ưu điểm

- BIT* (Batch Informed Trees) kết hợp hai ý tưởng chính:
  - *Lô mẫu ngẫu nhiên* của *RRT** để mở rộng không gian tìm kiếm. Nhưng không sinh điểm ngẫu nhiên từng cái như RRT*, mà tạo từng lô điểm rồi xử lý có hệ thống.
  - *Tìm đường tối ưu* theo *A** (Sử dụng *heuristic*) để ưu tiên mở rộng theo hướng có tiềm năng tốt nhất.
  - Khi thời gian chạy tiến đến vô hạn, kết quả hội tụ đến lời giải tối ưu.
- Tối ưu dần
  - BIT* bảo đảm rằng nếu chạy đủ lâu, đường đi nó tìm được sẽ hội tụ đến lời giải *tối ưu toàn cục* (Tính tối ưu đảm bảo chính xác cao).
- Tận dụng thông tin định hướng
  - BIT* không mở rộng mù quáng mà sử dụng *hàm heuristic* để *ưu tiên* những đoạn có tiềm năng tốt, giúp tăng tốc quá trình tìm đường.
- Xử lý theo lô
  - Thay vì mở rộng từng điểm, BIT* tạo một *lô điểm ngẫu nhiên,* rồi tổ chức mở rộng một cách *có hệ thống và chiến lược,* giúp tránh lãng phí tài nguyên tính toán.
- Cấu trúc dữ liệu hiệu quả

- Sử dụng hàng đợi ưu tiên và cây để mở rộng và kết nối đỉnh hiệu quả, đảm bảo chỉ xử lý những phần thực sự có tiềm năng tốt.

- Hiệu quả trong không gian cấu hình cao
  - Do tính chất heuristic + batch, BIT* hoạt động tốt hơn khi số chiều tăng, tránh bị "nổ tổ hợp" như nhiều thuật toán khác.
- Tính mở rộng
  - Cấu trúc của BIT* cho phép mở rộng dễ dàng theo thời gian (ví dụ: thêm ràng buộc, cập nhật chi phí, mở rộng bản đồ động…).
- Dừng sớm được
  - BIT* có thể trả ra một đường đi hợp lệ *ngay cả khi chưa chạy đến cuối*, và cải thiện dần theo thời gian. Điều này rất hữu ích trong các hệ thống thời gian thực.
- So với RRT*:



  - Tính chính xác:
    - BIT* sẽ chính xác hơn so với RRT* nếu như xét ở một không gian lớn, có nhiều vật cản nhờ cắt tỉa cây và tái sử dụng node cũ
  - Khả năng định hướng
    - BIT* sử dụng hàm heuristic để định hướng trong khi RRT* mở rộng node 1 cách ngẫu nhiên
    - So với RRT*, BIT* có khả năng tận dụng thông tin định hướng từ hàm heuristic, kết hợp với cơ chế pruning thông minh để cải thiện tốc độ hội tụ. Điều này đặc biệt hiệu quả

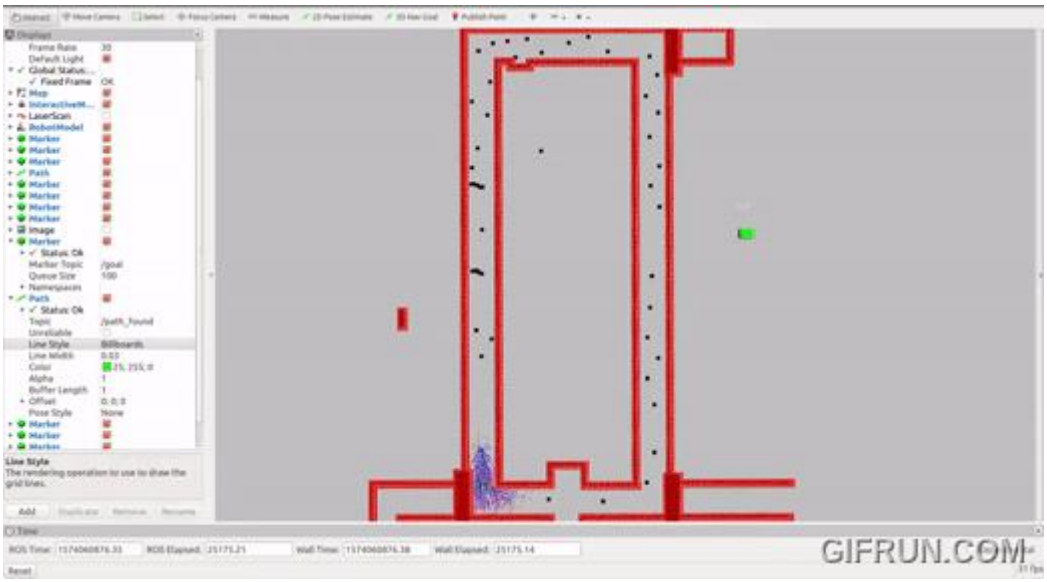
trong các môi trường có cấu trúc phức tạp, nhiều vật cản, nơi RRT* có thể mất thời gian lang thang ngẫu nhiên trước khi tìm ra nghiệm đầu tiên.

- Vì BIT* có tính chính xác cao nhưng tốc độ không bằng RRT* nên phù hợp sử dụng trong không gian tĩnh.

c. Nhược điểm

- Tuy nhiên, ngay trong lần chạy đầu tiên, thuật toán không thể ngay lập tức tìm ra đường đi ngắn nhất vì những lý do sau:
  - Lô mẫu ban đầu không đầy đủ để tìm ra đường tốt nhất:
    - Khi bắt đầu, BIT* chỉ sử dụng một tập hợp *mẫu ngẫu nhiên ban đầu* trong không gian.
    - Nếu lô này *không chứa đủ điểm quan trọng* (ví dụ: một nút gần đích nhưng kết nối tốt), thuật toán chưa thể tìm ra đường tối ưu.
    - Điều này tương tự như RRT*, nơi kết quả phụ thuộc vào vị trí của các nút mẫu.
  - Heuristic của A* chưa tối ưu ngay từ đầu:
    - Mặc dù BIT* ưu tiên mở rộng theo *hàm heuristic của A**, nhưng điều này không đảm bảo tìm ngay được đường tốt nhất.
    - Lý do: Các nút mở rộng ban đầu *chưa chắc đã tạo ra một đường hợp lý* để đến đích.
    - Nếu một đường đi ban đầu tồn tại, nó có thể không phải đường ngắn nhất, mà chỉ là một đường *tạm thời tốt*.
  - Cần thêm mẫu để cải thiện chất lượng đường đi:
    - Sau khi sử dụng hết các điểm mẫu ban đầu, BIT* thêm mẫu mới và tiếp tục tìm kiếm.
    - Các mẫu mới thu hẹp không gian tìm kiếm dần vào vùng có chất lượng đường tốt hơn.
    - Điều này giúp thuật toán tiệm cận dần đến đường đi tối ưu.
  - Giới hạn của việc chỉ dùng heuristic để mở rộng:
    - *Nếu chỉ mở rộng theo heuristic tối ưu nhất*, thuật toán sẽ giống như Greedy Best-First Search, có thể bị mắc kẹt trong đường cụt.
    - BIT* phải đảm bảo rằng *tất cả các lộ trình tiềm năng đều được xem xét*, nên cần lô mẫu và kiểm tra nhiều hướng mở rộng.
- So với RRT*:

- Bộ nhớ:
  - BIT* lưu trữ tất cả các mẫu đã sinh ra trong quá trình mở rộng cây, trong khi RRT* chỉ giữ lại những điểm quan trọng. Điều này khiến BIT* tiêu tốn bộ nhớ nhiều hơn so với RRT*.
- Tốc độ và môi trường:



  - BIT* có khả năng mở rộng kém hơn RRT*, RRT* có thể dễ dàng cập nhật khi có chướng ngại vật mới bằng cách tạo lại node gần nhất trong khi BIT* xử lý theo hướng thu hẹp khoảng cách nên khó thích nghi khi môi trường thay động,
  - RRT* nhanh hơn trong không gian lớn, ít vật cản và có sự thay đổi do mở rộng cây theo hướng ngẫu nhiên, giúp nhanh chóng bao phủ không gian, BIT* kém hiệu quả hơn trong không gian động do nó tạo ra quá nhiều điểm mẫu nên sẽ mất thời gian tính toán và đánh giá hàm heuristic, trong khi RRT* cứ tiếp tục mở rộng mà không bị tắc ở việc tối ưu những phần nhỏ.

8. Đọc thêm

Github BIT*

AIT*, RRT*, FMT*, BIT*

Robot Path Planning: RRT* v/s BIT*