HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

MÔN HỌC TOÁN RỜI RẠC 2

o0o

BÀI TẬP LỚN
NHÓM 6

Đề tài: **"THUẬT TOÁN BATCH-INFORMED RRT* (BIT*)"**

Giảng Viên: **TS. Nguyễn Kiều Linh**

Thành viên nhóm:
Dương Minh Thái - B23DCCN742(NT)
Hoàng Minh Quân - B23DCCN672
Nguyễn Thành Đạt - B23DCCN138
Đặng Xuân Quang - B23DCCN686
Phạm Đức Của - B23DCCN104
Bùi Văn Đạt - B23DCCN126
Nguyễn Doãn Hợp - B23DCCN350
Lê Vũ Minh - B23DCCN552

Nhóm Lớp: **12**
Hệ đào tạo: **Đại học chính quy**

Hà Nội, 05/2025

NHẬN XÉT CỦA GIẢNG VIÊN

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

Điểm: **(Bằng chữ:**)

Hà Nội, ngày tháng năm 20...

Giảng viên

Mục lục

I. Giới thiệu thuật toán

Thuật toán BIT* là một thuật toán lập kế hoạch đường đi dựa trên việc kết hợp các ưu điểm của cả phương pháp tìm kiếm trên đồ thị và phương pháp dựa trên lấy mẫu. BIT* hoạt động bằng cách nhận ra rằng một tập hợp các mẫu mô tả một đồ thị hình học ngẫu nhiên (RGG) ẩn. Điều này cho phép BIT* kết hợp hiệu quả tìm kiếm có thứ tự của các kỹ thuật dựa trên đồ thị, như A*, với khả năng mở rộng theo thời gian thực của các thuật toán dựa trên lấy mẫu, như RRT.

1.1 Ý tưởng chính của thuật toán

– Các thuật toán dựa trên lấy mẫu như RRT* có tính tối ưu bất tiệm cận nhưng có thể chậm trong việc tìm ra nghiệm tốt, đặc biệt là trong không gian trạng thái phức tạp hoặc chiều cao.

– Các phương pháp tìm kiếm trên đồ thị như A* hiệu quả khi không gian rời rạc hoặc được rời rạc hóa tốt, nhưng khó áp dụng trực tiếp cho không gian liên tục.

– BIT* tận dụng lợi thế của việc xử lý các mẫu theo lô (batch) để thực hiện tìm kiếm có thứ tự trên không gian lập kế hoạch liên tục, đồng thời duy trì hiệu suất theo thời gian thực.

– BIT* sử dụng hàm heuristic để hướng dẫn hiệu quả việc tìm kiếm trong một loạt các RGG ẩn ngày càng dày đặc, đồng thời tái sử dụng thông tin từ các lần tìm kiếm trước.

1.2 Đặc điểm của BIT*

– Tối ưu bất tiệm cận: BIT* được chứng minh là tối ưu bất tiệm cận, nghĩa là nó sẽ hội tụ đến nghiệm tối ưu khi số lượng mẫu tăng lên vô hạn.

– Hoàn chỉnh xác suất: BIT* cũng là hoàn chỉnh xác suất, nghĩa là nếu có một nghiệm tồn tại, BIT* sẽ tìm thấy nó với xác suất tiến đến 1 khi số lượng mẫu tăng lên vô hạn.

– Hiệu suất theo thời gian thực (Anytime Performance): BIT* có khả năng tìm ra các nghiệm khả thi nhanh chóng và tiếp tục cải thiện chất lượng nghiệm theo thời gian.

– Tìm kiếm có thông tin: Việc sử dụng hàm heuristic và tập trung lấy mẫu vào vùng có khả năng chứa nghiệm tốt hơn giúp BIT* tìm kiếm hiệu quả hơn so với RRT*.

– Hiệu quả trong không gian chiều cao: Các thử nghiệm cho thấy BIT* hoạt động tốt hơn các thuật toán khác, bao gồm RRT*, Informed

RRT* và FMT*, đặc biệt là trong không gian trạng thái có chiều cao.

– Tái sử dụng thông tin: BIT* tái sử dụng thông tin từ các lô tìm kiếm trước, giúp tăng tốc độ hội tụ. – Cân bằng giữa thăm dò và khai thác: Bằng cách sử dụng tìm kiếm heuristic theo lô và lấy mẫu có thông tin, BIT* cân bằng giữa việc thăm dò không gian trạng thái và khai thác các vùng hứa hẹn.

1.3 Mối quan hệ với các thuật toán khác

– RRT*: BIT* có thể được xem như một sự tổng quát hóa của RRT*.

– Informed RRT*: Với kích thước lô là một mẫu (m=1), BIT* tương tự như Informed RRT*, tập trung tìm kiếm trong vùng elip có thông tin.

– Fast Marching Trees: Với một lô duy nhất và heuristic bằng không,BIT* tương tự như FMT*.

– Lifelong Planning A (LPA*): Hàng đợi cạnh QE của BIT* là một mở rộng của hàng đợi đỉnh của LPA* cho các bài toán trong không gian liên tục, bao gồm cả ước tính heuristic về chi phí cạnh.

– Hàm heuristic: Được sử dụng để ước tính chi phí từ một trạng thái đến mục tiêu.

– Hằng số RGG (r hoặc k): Xác định cách các cạnh được hình thành trong RGG ẩn.

– Số lượng mẫu trên mỗi lô (m): Xác định số lượng mẫu mới được thêm vào trong mỗi lần lặp.

II. Trình bày thuật toán

2.1 Thuật toán BIT*

Algorithm 1 BIT*: Batch Informed Trees

$V \leftarrow \{x_{\text{start}}\}$; $E \leftarrow \emptyset$; $T \leftarrow (V, E)$
$X_{\text{unconn}} \leftarrow \mathcal{X}_{\text{goal}}$
$Q_V \leftarrow \emptyset$; $Q_E \leftarrow \emptyset$
$V_{\text{sol'n}} \leftarrow V \cap \mathcal{X}_{\text{goal}}$
$V_{\text{unexpand}} \leftarrow V$; $X_{\text{new}} \leftarrow X_{\text{unconn}}$
$c_i \leftarrow \min_{v_{\text{goal}} \in V_{\text{sol'n}}} \{g_T(v_{\text{goal}})\}$
while not STOP **do**
if $Q_E = \emptyset$ and $Q_V = \emptyset$ **then**
$X_{\text{reuse}} \leftarrow \text{Prune}(T, X_{\text{unconn}}, c_i)$
$X_{\text{sampling}} \leftarrow \text{Sample}(r, \mathcal{X}_{\text{start}}, \mathcal{X}_{\text{goal}}, c_i)$
$X_{\text{new}} \leftarrow X_{\text{reuse}} \cup X_{\text{sampling}}$
$X_{\text{unconn}} \leftarrow X_{\text{new}}$
$Q_V \leftarrow V$
end if
while $\text{BestQueueValue}(Q_V) \leq \text{BestQueueValue}(Q_E)$ **do**
$\textsc{ExpandNextVertex}(Q_V, Q_E, c_i)$
$(v_{\min}, x_{\min}) \leftarrow \textsc{PopBestInQueue}(Q_E)$
if $g_T(v_{\min}) + \hat{c}(v_{\min}, x_{\min}) + \hat{h}(x_{\min}) < c_i$ **then**
if $g_T(v_{\min}) + \hat{c}(v_{\min}, x_{\min}) < g_T(x_{\min})$ **then**
$c_{\text{edge}} \leftarrow c(v_{\min}, x_{\min})$
if $g_T(v_{\min}) + c_{\text{edge}} + \hat{h}(x_{\min}) < c_i$ **then**
if $g_T(v_{\min}) + c_{\text{edge}} < g_T(x_{\min})$ **then**
if $x_{\min} \in V$ **then**
$v_{\text{parent}} \leftarrow \text{Parent}(x_{\min})$
$E \leftarrow E \cup \{(v_{\text{parent}}, x_{\min})\}$
else
$X_{\text{unconn}} \leftarrow X_{\text{unconn}} \setminus \{x_{\min}\}$
$V \leftarrow V \cup \{x_{\min}\}$
$Q_V \leftarrow Q_V \cup \{x_{\min}\}$
$V_{\text{unexpand}} \leftarrow V_{\text{unexpand}} \cup \{x_{\min}\}$
if $x_{\min} \in \mathcal{X}_{\text{goal}}$ **then**
$V_{\text{sol'n}} \leftarrow V_{\text{sol'n}} \cup \{x_{\min}\}$
end if
$E \leftarrow E \cup \{(v_{\min}, x_{\min})\}$
$c_i \leftarrow \min_{v_{\text{goal}} \in V_{\text{sol'n}}} \{g_T(v_{\text{goal}})\}$
end if
end if
end if
end if
else
$Q_E \leftarrow \emptyset$; $Q_V \leftarrow \emptyset$
end if
end while
end while
return T

Bước 1: Khởi tạo

$$
\begin{array}{ll}
V \leftarrow \{x_{\text{start}}\}; & E \leftarrow \emptyset; \quad \mathcal{T} = (V, E); \\
X_{\text{unconn}} \leftarrow X_{\text{goal}}; & \\
Q_V \leftarrow V; & Q_E \leftarrow \emptyset; \\
V_{\text{sol'n}} \leftarrow V \cap X_{\text{goal}}; & V_{\text{unexpnd}} \leftarrow V; \quad X_{\text{new}} \leftarrow X_{\text{unconn}}; \\
c_i \leftarrow \min\limits_{v_{\text{goal}} \in V_{\text{sol'n}}} \{g_{\mathcal{T}}(v_{\text{goal}})\}; &
\end{array}
$$

Khởi tạo cây tìm kiếm $\mathcal{T} = (V, E)$ là đồ thị vô hướng với:

- Tập đỉnh ban đầu chứa trạng thái bắt đầu: $V \leftarrow \{x_{\text{start}}\}$
- Tập cạnh ban đầu rỗng: $E \leftarrow \emptyset$

Theo dõi các điểm đích chưa được kết nối:

- Tập tất cả các điểm đích khả thi: X_{goal}
- Tập con các đích chưa được kết nối với cây: $X_{\text{unconn}} \leftarrow X_{\text{goal}}$

Sử dụng hai hàng đợi ưu tiên:

- Q_V: hàng đợi đỉnh (ban đầu chứa x_{start})
- Q_E: hàng đợi cạnh (ban đầu rỗng)

Cập nhật thông tin về đích và trạng thái mở rộng:

- Đích đã kết nối: $V_{\text{sol'n}} \leftarrow V \cap X_{\text{goal}}$
- Các đỉnh chưa mở rộng: $V_{\text{unexpnd}} \leftarrow V$
- Chuẩn bị bước tạo nhánh mới: $X_{\text{new}} \leftarrow X_{\text{unconn}}$

Tính chi phí nhỏ nhất đến một đích đã kết nối:

$$c_i \leftarrow \min_{v_{\text{goal}} \in V_{\text{sol'n}}} \{g_{\mathcal{T}}(v_{\text{goal}})\}$$

- Duyệt qua tất cả $v_{\text{goal}} \in V_{\text{sol'n}}$ để chọn đích có chi phí nhỏ nhất
- $g_{\mathcal{T}}(v_{\text{goal}})$: tổng chi phí từ x_{start} đến v_{goal} trong cây $\mathcal{T}$

Bước 2: Bổ sung

- **if $Q_E = \emptyset$ and $Q_V \neq \emptyset$ then**
  - $X_{\text{reuse}} \leftarrow \text{Prune}(T, X_{\text{known}}, c_I)$
  - $X_{\text{sampling}} \leftarrow \text{Sample}(m, X_{\text{unif}}, X_{\text{pool}}, c_I)$
  - $X_{\text{new}} \leftarrow X_{\text{reuse}} \cup X_{\text{sampling}}$
  - $X_{\text{known}} \leftarrow X_{\text{new}}$
  - $Q_V \leftarrow \emptyset$
- Kiểm tra xem cả hai hàng đợi (hoặc tập hợp): if $Q_E = \emptyset$ and $Q_V = \emptyset$ then
  - Nếu cả hai điều đúng:
    * Loại bỏ các nút (node) không còn hữu ích trong cây tìm kiếm, giữ lại các nút có khả năng dẫn đến đích với chi phí thấp hơn c_I:
      $$X_{\text{reuse}} \leftarrow \text{Prune}(T, X_{\text{known}}, c_I)$$
      +Prune(T, X_{new}, c_I) cũng để tái xét tất cả và loại bỏ các điểm không cần thiết trong tập X_{known}, dựa theo chi phí hiện tại c_I
      + X_{reuse}: Tập con của X_{new} chứa các node có giá trị, tái sử dụng cho vòng lặp tiếp theo
    * Sinh m mẫu điểm mới từ không gian cấu hình, nhằm khám phá thêm các hướng đi mới:
      $$X_{\text{sampling}} \leftarrow \text{Sample}(m, X_{\text{start}}, X_{\text{goal}}, c_I)$$
      · X_{sampling}: tập không gian mẫu chứa m điểm trong không gian
      · X_{start}: điểm bắt đầu
      · c_I: chi phí hiện tại
    * Gộp X_{reuse} và X_{sampling} lại thành tập X_{new} để chuẩn bị cho bước tiếp theo:
      $$X_{\text{new}} \leftarrow X_{\text{reuse}} \cup X_{\text{sampling}}$$
    * Cập nhật lại tập các node chưa kết nối:
      $$X_{\text{known}} \leftarrow X_{\text{new}}$$
    * Đưa các node hiện có (V) trở lại hàng đợi Q_V để tiếp tục thuật toán:
      $$Q_V \leftarrow V$$

Bước 3: Tối ưu mở rộng tìm kiếm

Tối ưu mở rộng tìm kiếm làm cho hành trình ngắn hơn nhanh hơn và hiệu quả hơn trong không gian tìm đường:

- **while** $\text{BestQueueValue}(Q_V) \leq \text{BestQueueValue}(Q_E)$ **do**
  $\text{ExpandNextVertex}(Q_V, Q_E, c_I)$;

  $(v_{\min}, x_{\min}) \leftarrow \text{PopBestInQueue}(Q_E)$;

Lặp mở rộng đỉnh (vertex) từ hàng đợi Q_V chừng nào giá trị tốt nhất của Q_V nhỏ hơn hoặc bằng giá trị tốt nhất trong Q_E:

$$\text{while BestQueueValue}(Q_V) \leq \text{BestQueueValue}(Q_E) \text{ do}$$

- Từng bước mở rộng vùng không gian khả thi bằng cách xây dựng đồ thị kết nối dần dần, theo hướng có chi phí tốt:

$$\text{ExpandNextVertex}(Q_V, Q_E, c_I);$$

- Chọn bước mở rộng tốt nhất hiện tại để tiến hành kết nối trong cây:

$$(v_{\min}, x_{\min}) \leftarrow \text{PopBestInQueue}(Q_E)$$

Bước 4: Tìm đường đi tối ưu

Tìm đường đi tối ưu từ điểm bắt đầu đến đích trong không gian có chướng ngại vật (sử dụng heuristic để tăng hiệu quả).

if $g_T(v_{\min}) + \hat{c}(v_{\min}, x_{\min}) + \hat{h}(x_{\min}) < c_i$ **then**
  if $g_T(v_{\min}) + \hat{c}(v_{\min}, x_{\min}) < g_T(x_{\min})$ **then**
    $c_{\text{edge}} \leftarrow c(v_{\min}, x_{\min})$
    if $g_T(v_{\min}) + c_{\text{edge}} + \hat{h}(x_{\min}) < c_i$ **then**
      if $g_T(v_{\min}) + c_{\text{edge}} < g_T(x_{\min})$ **then**
        if $x_{\min} \in V$ **then**
          $v_{\text{parent}} \leftarrow \text{Parent}(x_{\min})$
          $E \leftarrow E \cup \{(v_{\text{parent}}, x_{\min})\}$
        else
          $X_{\text{unconn}} \leftarrow \{x_{\min}\}$
          $V \leftarrow V \cup \{x_{\min}\}$
          $Q_V \leftarrow Q_V \cup \{x_{\min}\}$
          $V_{\text{unexpand}} \leftarrow V_{\text{unexpand}} \cup \{x_{\min}\}$
          if $x_{\min} \in X_{\text{goal}}$ **then**
            $V_{\text{sol'n}} \leftarrow V_{\text{sol'n}} \cup \{x_{\min}\}$
          $E \leftarrow E \cup \{(v_{\min}, x_{\min})\}$
          $c_i \leftarrow \min_{v_{\text{goal}} \in V_{\text{sol'n}}} g_T(v_{\text{goal}})$
else
  $Q_E \leftarrow \emptyset; \quad Q_V \leftarrow \emptyset$

until STOP;
return T;

Các biến sử dụng trong thuật toán:

- $g_T(v)$: Chi phí từ gốc đến đỉnh v
- $\hat{c}(v_{\min}, x_{\min})$: chi phí ước lượng từ $v_{\min}$ đến $x_{\min}$
- $\hat{h}(x_{\min})$: hàm heuristic
- $c(v_{\min}, x_{\min})$: chi phí thật
- $x_{\min}$: đỉnh gần nhất để mở rộng

- c_I: Chi phí hiện tại tốt nhất để đến đích
- $\text{Parent}(x)$: hàm trả về đỉnh cha của đỉnh x
- c_{edge}: chi phí thật của cạnh được chọn

Xét điều kiện chọn đỉnh mở rộng, nếu tổng chi phí đến $x_{\min}$ nhỏ hơn chi phí tốt nhất đến hiện tại, thì tiếp tục mở rộng:

$$\text{if } g_T(v_{\min}) + \hat{c}(v_{\min}, x_{\min}) + \hat{h}(x_{\min}) < c_i \text{ then}$$

Nếu đường đi mới đến $x_{\min}$ tốt hơn, cập nhật chi phí thật của cạnh:

$$\text{if } g_T(v_{\min}) + \hat{c}(v_{\min}, x_{\min}) < g_T(x_{\min}) \text{ then}$$
$$c_{\text{edge}} \leftarrow c(v_{\min}, x_{\min});$$

Kiểm tra lại điều kiện chi phí thật với heuristic, nếu tổng chi phí thật cộng với heuristic vẫn tốt hơn đường đi hiện tại, tiếp tục:

$$\text{if } g_T(v_{\min}) + c_{\text{edge}} + \hat{h}(x_{\min}) < c_i \text{ then}$$

So sánh và cập nhật nếu đường đi tốt hơn:

$$\text{if } g_T(v_{\min}) + c_{\text{edge}} < g_T(x_{\min}) \text{ then}$$

Nếu $x_{\min}$ đã thuộc cây tìm kiếm, cập nhật cạnh từ đỉnh cha hiện tại đến $x_{\min}$:

$$\text{if } x_{\min} \in V \text{ then}$$
$$v_{\text{parent}} \leftarrow \text{Parent}(x_{\min});$$
$$E \leftarrow E \cup \{(v_{\text{parent}}, x_{\min})\};$$

Nếu $x_{\min}$ là đỉnh mới, thêm vào cây, các tập mở rộng và kiểm tra xem nó có phải đích không:

$$\text{else} \quad X_{\text{unconn}} \leftarrow \{x_{\min}\};$$
$$V \leftarrow V \cup \{x_{\min}\};$$
$$Q_V \leftarrow Q_V \cup \{x_{\min}\};$$
$$V_{\text{unexpand}} \leftarrow V_{\text{unexpand}} \cup \{x_{\min}\};$$
$$\text{if } x_{\min} \in X_{\text{goal}} \text{ then}$$
$$V_{\text{sol'n}} \leftarrow V_{\text{sol'n}} \cup \{x_{\min}\};$$

Cập nhật cạnh và chi phí tốt nhất:

$$E \leftarrow E \cup \{(v_{\min}, x_{\min})\};$$
$$c_i \leftarrow \min_{v_{\text{goal}} \in V_{\text{sol'n}}} \{g_T(v_{\text{goal}})\};$$

Nếu không thỏa mãn điều kiện ban đầu:

$$Q_E \leftarrow \emptyset; \quad Q_V \leftarrow \emptyset;$$

Lặp lại thuật toán cho đến khi hội đủ điều kiện dừng:

$$\text{until STOP;}$$

2.2 Thuật toán Prune

Mã giả

1: $X_{\text{reuse}} \leftarrow \emptyset$
2: $X_{\text{unconn}} \overset{-}{\leftarrow} \{x \in X_{\text{unconn}} \mid \hat{f}(x) \geq c_i\}$
3: **for all** $v \in V$ in order of increasing $g_\tau(v)$ **do**
4:     **if** $\hat{f}(v) \geq c_i$ or $g_\tau(v) + \hat{h}(v) > c_i$ **then**
5:         $V \overset{-}{\leftarrow} \{v\}$
6:         $V_{\text{sol'n}} \overset{-}{\leftarrow} \{v\}$
7:         $V_{\text{unexpnd}} \overset{-}{\leftarrow} \{v\}$
8:         $v_{\text{parent}} \overset{-}{\leftarrow} \text{Parent}(v)$
9:         $E \overset{-}{\leftarrow} \{(v_{\text{parent}}, v)\}$
10:         **if** $\hat{f}(v) < c_i$ **then**
11:             $X_{\text{reuse}} \overset{+}{\leftarrow} X_{\text{reuse}} \cup \{v\}$
12:         **end if**
13:     **end if**
14: **end for**
15: **return** X_{reuse}

Giải thích thuật toán

Thuật toán cắt tỉa (Prune) lặp qua các đỉnh trong đồ thị và kiểm tra xem đỉnh nào đủ điều kiện để giữ lại hoặc loại bỏ dựa trên các giá trị hàm $\hat{f}(v)$, $g_\tau(v)$, và $\hat{h}(v)$. Nếu một đỉnh thỏa mãn điều kiện, nó sẽ được cô lập và lưu vào các tập hợp ứng viên hoặc kết quả cuối cùng. Mục tiêu là giảm bớt các đỉnh không cần thiết, làm giảm chi phí tính toán và tăng hiệu quả tìm kiếm.

Cách chạy

1. Khởi tạo các tập hợp (Dòng 1 và 2):

- $X_{\text{reuse}} \overset{-}{\leftarrow} \emptyset$: Tạo một tập rỗng để lưu các đỉnh tái sử dụng.
- $X_{\text{unconn}} \overset{-}{\leftarrow} \{x \in X_{\text{unconn}} \mid \hat{f}(x) \geq c_i\}$: Lọc tập X_{unconn} giữ lại các phần tử có $\hat{f}(x) \geq c_i$.

2. Lặp qua các đỉnh (Dòng 3): Lặp theo thứ tự tăng dần của $g_\tau(v)$ để ưu tiên xử lý các đỉnh có chi phí nhỏ hơn trước.

3. Kiểm tra điều kiện cắt tỉa (Dòng 4):

$$\text{Nếu } \hat{f}(v) \geq c_i \quad \text{hoặc} \quad g_\tau(v) + \hat{h}(v) > c_i$$

thì đỉnh v sẽ được xét để cắt tỉa.

4. Cập nhật các tập hợp (Dòng 5–9):

- $V \overset{-}{\leftarrow} \{v\}$: Cô lập đỉnh v.
- $V_{\text{sol'n}} \overset{-}{\leftarrow} \{v\}$: Thêm v vào tập các đỉnh quan trọng.
- $V_{\text{unexpnd}} \overset{-}{\leftarrow} \{v\}$: Đánh dấu không mở rộng thêm đỉnh này.
- $v_{\text{parent}} \overset{-}{\leftarrow} \text{Parent}(v)$: Lấy đỉnh cha.
- $E \overset{-}{\leftarrow} \{(v_{\text{parent}}, v)\}$: Cập nhật cạnh liên kết.
- Nếu $\hat{f}(v) < c_i$ thì thêm v vào X_{reuse}.

5. Trả về kết quả (Dòng 10): Trả về tập X_{reuse} chứa các đỉnh có thể tái sử dụng.

Giải thích cách chạy

1. **Bước 1:** Thuật toán sẽ bắt đầu lọc các đỉnh không kết nối dựa trên giá trị hàm $\hat{f}(x)$.
2. **Bước 2:** Sau đó, thuật toán sẽ tiếp tục xử lý các đỉnh trong đồ thị theo thứ tự của $g_\tau(v)$.
3. **Bước 3:** Mỗi khi gặp một đỉnh thỏa mãn điều kiện cắt tỉa, thuật toán sẽ cô lập đỉnh đó, lưu nó vào các tập hợp như $V_{\text{sol'n}}$, V_{unexpnd}, và cập nhật mối quan hệ cha-con.
4. **Bước 4:** Sau khi cắt tỉa xong, thuật toán sẽ trả về các đỉnh cần giữ lại hoặc tái sử dụng.

2.3 Thuật toán mở rộng đỉnh (Thuật toán 1, dòng 14)

Mã giả

Algorithm 2 ExpandNextVertexExpandNextVertex($Q_V \subseteq V, Q_E \subseteq V \times (V \cup X), c_i \in \mathbb{R}_{\geq 0}$)

1: $\mathbf{v}_{\min} \leftarrow$ PopBestInQueue(Q_V);
2: **if** $\mathbf{v}_{\min} \in V_{\text{unexpnd}}$ **then**
3: $\quad X_{\text{near}} \leftarrow \text{Near}(X_{\text{unconn}}, \mathbf{v}_{\min}, r_{\text{BIT}^*})$;
4: **else**
5: $\quad X_{\text{near}} \leftarrow \text{Near}(X_{\text{new}} \cap X_{\text{unconn}}, \mathbf{v}_{\min}, r_{\text{BIT}^*})$;
6: **end if**
7: $Q_E \xleftarrow{+} Q_E \cup \left\{ (\mathbf{v}_{\min}, \mathbf{x}) \in V \times X_{\text{near}} \;\middle|\; \hat{g}(\mathbf{v}_{\min}) + \hat{c}(\mathbf{v}_{\min}, \mathbf{x}) + \hat{h}(\mathbf{x}) < c_i \right\}$;
8: **if** $\mathbf{v}_{\min} \in V_{\text{unexpnd}}$ **then**
9: $\quad V_{\text{near}} \leftarrow \text{Near}(V, \mathbf{v}_{\min}, r_{\text{BIT}^*})$;
10: $\quad Q_E \leftarrow Q_E \cup \{(\mathbf{v}_{\min}, \mathbf{w}) \in V \times V_{\text{near}} \mid$
$\quad (\mathbf{v}_{\min}, \mathbf{w}) \notin E,\ \hat{g}(\mathbf{v}_{\min}) + \hat{c}(\mathbf{v}_{\min}, \mathbf{w}) + \hat{h}(\mathbf{w}) < c_i \ \vee\ \hat{g}(\mathbf{v}_{\min}) + \hat{c}(\mathbf{v}_{\min}, \mathbf{w}) < g_T(\mathbf{w}) \Big\}$;
11: $\quad V_{\text{unexpnd}} \xleftarrow{-} V_{\text{unexpnd}} \setminus \{\mathbf{v}_{\min}\}$;
12: **end if**

Giải thích thuật toán

Mỗi đỉnh trong cây được mở rộng hoặc cát tỉa trong mỗi đợt. Xử lý tất cả các cạnh đi ra từ các đỉnh sẽ dẫn đến BIT* liên tục xem xét các cạnh bị từ chối trước đó. Có thể tránh được điều này bằng cách sử dụng tập hợp các đỉnh chưa được mở rộng, V_{unexpnd}, và các mẫu mới, X_{new}, để chỉ thêm vào hàng đợi các cạnh chưa từng được xét đến trước đó.

Tóm tắt cách chạy:

1. Lấy đỉnh có giá trị ước lượng tốt nhất từ hàng đợi đỉnh (vertex queun) (Dòng 1)

2. Xem đỉnh nguồn đã được mở rộng hay chưa (dòng 2).

   - Nếu đỉnh đó *chưa được mở rộng*, thì tất cả các mẫu (đỉnh) đi ra từ nó trong bán kính r_{BIT^*} của $v_{\min}$ là các đỉnh tiềm năng (dòng 3).
   - Ngược lại, nếu đỉnh *đã được mở rộng* thì mọi kết nối đến các mẫu "cũ" chưa kết nối đều đã được xét và loại bỏ, và chỉ các mẫu "mới" sẽ được xem là đỉnh kết nối tiềm năng (dòng 5).

3. (dòng 6) Tập con các cạnh tiềm năng, sẽ được thêm vào hàng đợi trong cả hai trường hợp, nếu có khả năng cải thiện nghiệm hiện tại tức là thỏa mãn hàm

$\hat{g}(\mathbf{v}_{\min}) + \hat{c}(\mathbf{v}_{\min}, \mathbf{x}) + \hat{h}(\mathbf{x}) < c_i$;.

4. Việc các cạnh nối đến các mẫu *đã kết nối* (tức là các kết nối lại – *rewiring*) có được xem là "mới" hay không cũng phụ thuộc vào việc đỉnh nguồn đã từng được mở rộng chưa (dòng 7).
   - Nếu đỉnh *chưa được mở rộng*, thì tất cả các đỉnh đã kết nối gần đó đều được xem là đỉnh kết nối tiềm năng (dòng 8).
   - Tập con các cạnh tiềm năng này, nếu có khả năng cải thiện nghiệm hiện tại và cây hiện tại, sẽ được thêm vào hàng đợi (dòng 9),
   - Đỉnh sẽ được đánh dấu là đã mở rộng (dòng 10).

Nếu một đỉnh đã từng được mở rộng, thì không thực hiện kết nối lại (rewiring) nào nữa. Mặc dù có thể có các cải thiện trong cây cho phép các cạnh đã từng được xét đến trước đây cải thiện các đỉnh đã kết nối, nhưng việc xét lại các kết nối này sẽ dẫn đến việc phải lặp lại quá trình xét các cạnh không khả thi. Tương tự như trong RRT*, việc không thực hiện kết nối lại này không ảnh hưởng đến tính tối ưu tiệm cận với xác suất gần như chắc chắn (almost-sure asymptotic optimality).

Cải thiện thuật toán

Vấn đề: Nhiều hệ thống có giới hạn thời gian tính toán để giải quyết các bài toán lập kế hoạch. Trong trường hợp này việc kết nối lại cây (rewiring) trước khi tìm được nghiệm ban đầu sẽ làm giảm khả năng của BIT* giải được bài toán đã cho,dưới đây là mã giả cho thuật toán.

Algorithm 3 ExpandNextVertex$(Q_V \subseteq V,\ Q_E \subseteq V \times (V \cup \mathcal{X}),\ c_i \in \mathbb{R}_{\geq 0})$

1: $\mathbf{v}_{\min} \leftarrow \text{PopBestInQueue}(Q_V)$;
2: **if** $\mathbf{v}_{\min} \in V_{\text{unexpnd}}$ **then**
3: $\quad X_{\text{near}} \leftarrow \text{Near}(X_{\text{unconn}}, \mathbf{v}_{\min}, r_{\text{BIT}^*})$;
4: $\quad V_{\text{unexpnd}} \xleftarrow{-} \{\mathbf{v}_{\min}\}$;
5: $\quad V_{\text{delayed}} \xleftarrow{+} \{\mathbf{v}_{\min}\}$;
6: **else**
7: $\quad X_{\text{near}} \leftarrow \text{Near}(X_{\text{new}} \cap X_{\text{unconn}}, \mathbf{v}_{\min}, r_{\text{BIT}^*})$;
8: **end if**
9: $Q_E \xleftarrow{+} \left\{ (\mathbf{v}_{\min}, \mathbf{x}) \in V \times X_{\text{near}} \;\middle|\; \hat{g}(\mathbf{v}_{\min}) + \hat{c}(\mathbf{v}_{\min}, \mathbf{x}) + \hat{h}(\mathbf{x}) < c_i \right\}$;
10: **if** $\mathbf{v}_{\min} \in V_{\text{delayed}}$ **and** $c_i < \infty$ **then**
11: $\quad V_{\text{near}} \leftarrow \text{Near}(V, \mathbf{v}_{\min}, r_{\text{BIT}^*})$;
12: $\quad Q_E \leftarrow Q_E \cup \{(\mathbf{v}_{\min}, \mathbf{w}) \in V \times V_{\text{near}} \mid$
$\quad (\mathbf{v}_{\min}, \mathbf{w}) \notin E,\ \hat{g}(\mathbf{v}_{\min}) + \hat{c}(\mathbf{v}_{\min}, \mathbf{w}) + \hat{h}(\mathbf{w}) < c_i \ \vee\ \hat{g}(\mathbf{v}_{\min}) + \hat{c}(\mathbf{v}_{\min}, \mathbf{w}) < g_T(\mathbf{w}) \}$;
13: $\quad V_{\text{delayed}} \xleftarrow{-} \{\mathbf{v}_{\min}\}$;
14: **end if**

Phương pháp trong thuật toán này sẽ trì hoãn việc kết nối lại cây (rewiring) cho đến khi tìm được nghiệm ban đầu. Nó ưu tiên việc khám phá dồ thị ngẫu nhiên hình học (RRG) để tìm ra nghiệm => tối ưu hơn trong các bài toán bị giới hạn bởi thời gian. Việc kết nối lại cây (rewiring) vẫn được thực hiện sau khi tìm được nghiệm, phương pháp này không ảnh hưởng tới tính tối ưu tiệm cận với xác suất gần như chắc chắn. (Các thay đổi so với thuật toán trước được bôi đỏ)

Giải thích thuật toán: Việc kết nối lại (rewiring) được trì hoãn (delayed) bằng cách theo dõi riêng biệt các đỉnh chưa được mở rộng đối với các mẫu chưa kết nối gần đó, Vunexpnd và các đỉnh chưa được mở rộng đối với các đỉnh đã kết nối gần đó, Vdelayed. Điều này cho phép BIT* ưu tiên tìm nghiệm bằng cách chỉ xét các cạnh nối với các mẫu mới cho đến khi nghiệm được tìm thấy, sau đó cải thiện bằng cách xét các kết nối lại từ các đỉnh hiện có.

Giải thích cách chạy:

- Một đỉnh sẽ được chuyển từ tập đỉnh chưa từng mở rộng sang tập trì hoãn khi các cạnh nối đến các đỉnh con chưa kết nối tiềm năng của nó được thêm vào hàng đợi cạnh (dòng 4–5).
- Các đỉnh trong tập trì hoãn sẽ được mở rộng như các kết nối lại tiềm năng từ các đỉnh đã kết nối sau khi nghiệm được tìm thấy (dòng 9), và nhãn "trì hoãn" sẽ được xóa bỏ (dòng 12).
- Phần mở rộng này yêu cầu khởi tạo và đặt lại tập Vdelayed cùng với các tập nhãn khác

III. Code thuật toán

3.1 Run

```python
import argparse
from BIT_Star import *
import BIT_Star as BIT_Star
import Node as node
from Node import *
from Map import *
from Visualize import *
import sys
import shutil
from datetime import datetime
from PIL import Image
from PrintColours import *


def main(
    map_name: str,
    vis: bool,
    start: list,
    goal: list,
    rbit: float,
    samples: int,
    dim: int,
    seed: int,
    stop_time: int,
    fast: bool,
) -> None:
    """The main function for the BIT* algorithm this function will run the algorithm after parsing the command line arguments. Call the bitstar class/function to run the algorithm and if the vis or fast flag is set then call the visualizer to show the path. If the fast flag is set then the fast visualizer is called else the normal visualizer is called. This also creates a text file that contains the path length and time taken for each seed. All the output including plots is saved in the Output folder. The logs are saved in the Logs folder and deleted after the algorithm is run.

    Args:
        map_name (str): The name of the map to run the algorithm on.
        vis (bool): If set to true then the visualizer will be called to show the path after the algorithm is run.
        start (list): The start coordinates in the form ['x', 'y'].
        goal (list): The goal coordinates in the form ['x', 'y'].
        rbit (float): The radius of the ball (rbit*) to be used in the algorithm.
        samples (int): The number of samples to be used in the algorithm.
        dim (int): The dimension of the search space.
        seed (int): The number of seeds to be used in the algorithm.
        stop_time (int): The time in seconds after which the algorithm will stop.
        fast (bool): If set to true then the fast visualizer will be called to show the path after the algorithm is run.
    """
    # Get the working directory of this file.
    pwd = os.path.abspath(os.path.dirname(__file__))

    # Empty list to store the time taken and path length for each seed.
    time_taken_all = []
    path_lengths = []

    # The text file to store the path length and time taken for each seed.
    text_path = f"{pwd}/../Output/path_lengths_and_times_{map_name}.txt"
    # Make the directory if it does not exist.
    os.makedirs(os.path.dirname(text_path), exist_ok=True)

    # Run the algorithm for each seed.
    for seed in range(seed):
        # Set the seed for the random number generator.
        random.seed(seed)
        np.random.seed(seed)

        # Get the start and goal coordinates.
```

```python
        start = []
        goal = []
        for i in range(opt.dim):
            # Convert the strings to floats.
            start.append(float(opt.start[i]))
            goal.append(float(opt.goal[i]))

        # Convert the lists to numpy arrays.
        start = np.array(start)
        goal = np.array(goal)

        # Create the log directory.
        log_dir = f"{pwd}/../Logs/{map_name}"
        os.makedirs(log_dir, exist_ok=True)

        # Get the occupancy grid map.
        map_path = f"{pwd}/../gridmaps/{map_name}.png"
        # Open the image and convert it to a numpy array.
        occ_map = np.array(Image.open(map_path))

        # Set the start and goal coordinates in the node class. This is done so that the node class can access the start and goal coordinates.
        node.start_arr = start
        node.goal_arr = goal

        # Create the start and goal nodes. The start node has a cost of 0.
        start_node = Node(tuple(start), gt=0)
        goal_node = Node(tuple(goal))

        # Create the map object and pass the occupancy grid map, start node, and goal node.
        map_obj = Map(start=start_node, goal=goal_node, occ_grid=occ_map)
        planner = None

        # Decide which planner to use based on the vis and fast flags. If the vis or fast flag is set then a log directory must be passed to the planner.
        if vis or fast:
            planner = bitstar(
                start=start_node,
                goal=goal_node,
                occ_map=map_obj,
                no_samples=samples,
                rbit=rbit,
                dim=dim,
                log_dir=log_dir,
                stop_time=stop_time,
            )
        # Else the planner is called without the log directory so that the logs are not saved.
        else:
            planner = bitstar(
                start=start_node,
                goal=goal_node,
                occ_map=map_obj,
                no_samples=samples,
                rbit=rbit,
                dim=dim,
                stop_time=stop_time,
            )
        # Make the plan.
        path, path_length, time_taken = planner.make_plan()

        print(
            f"{CGREEN2}Seed: {seed}\t\tFinal CI: {planner.ci}\tOld CI: {planner.old_ci}\tFinal Path Length: {path_length}\nPath:{CEND} {path}\n{CGREEN2}Time Taken per iteration:{CEND} {time_taken}\n{CEND}"
        )
        # Append the time taken and path length to the lists.
        time_taken_all.append(time_taken)
        path_lengths.append(path_length)
        # Convert the lists to strings and write them to the text file.
        time_taken_str = ", ".join([str(t) for t in time_taken])
        with open(text_path, "a") as f:
            f.write(
                f"Seed: {seed}\nPath Length: {path_length}\nTime Taken: {time_taken_str}\n"
            )
```

```python

        if vis or fast:
            # Create the output directory. The directory name is the map name and the current date and time.
            output_dir = f"{pwd}/../Output/{map_name} - {str(datetime.now().strftime('%Y-%m-%d %H-%M-%S'))}/"
            os.makedirs(output_dir, exist_ok=True)

            # Invert the occupancy grid map so that the free space is white and the occupied space is black. Weird bug in matplotlib which requires us to do this or use cv2.CvtColor function.
            inv_map = np.where((occ_map == 0) | (occ_map == 1), occ_map ^ 1, occ_map)

            # Create the visualizer object and pass the start and goal coordinates, the occupancy grid map, and the output directory.
            visualizer = Visualizer(start, goal, inv_map, output_dir)

            print(
                f"{CGREEN2}{CBOLD}{len(os.listdir(log_dir))} files in Log Directory:{CEND} {log_dir}/{sorted(os.listdir(log_dir))}\n"
            )
            # Read the log files.
            visualizer.read_json(log_dir, max_iter=np.inf)

            for i in range(len(os.listdir(log_dir))):
                # For each simulation draw with the fast visualizer if the fast flag is set.
                visualizer.draw(i, fast)
            # After all the simulations are drawn, set the title of the plot and show it is done drawing.
            visualizer.ax.set_title("BIT* - Final Path", fontsize=30)
            # Wait for the user to close the plot.
            plt.show()

        # Delete the log directory. This is done so the results of this experiment does not affect the results of the next experiment.
        print(
            f"{CRED2}================== Deleting log directory: {log_dir} =================={CEND}"
        )
        # Remove the log directory.
        shutil.rmtree(log_dir)


def parse_opt() -> argparse.Namespace:
    """Parse the command line arguments and return the arguments.

    Returns:
        argparse.Namespace: The arguments passed from the command line.
    """
    # Create the parser.
    parser = argparse.ArgumentParser()
    # Adding the other arguments.
    parser.add_argument(
        "--map_name",
        type=str,
        default="Default",
        help="Name of the map file. The map must be in the gridmaps folder. The map must be a png file. The map must be a black and white image where black is the obstacle space and white is the free space. Name of the map must be in the form 'map_name.png'. Do not enter the file extension in the map_name argument. The included maps are Default (or empty), Enclosure, Maze, Random, Symmetry, and Wall_gap. If none, will Default 100x100 empty grid will be used. Eg: --map_name Default",
    )
    parser.add_argument(
        "--vis",
        action="store_true",
        help="Whether or not to save and visualize outputs",
    )
    parser.add_argument("--start", nargs="+", help="Start coordinates. Eg: --start 0 0")
    parser.add_argument("--goal", nargs="+", help="Goal coordinates. Eg: --goal 99 99")
    parser.add_argument(
        "--rbit", type=float, default=10, help="Maximum Edge length. Eg: --rbit 10"
    )
    parser.add_argument(
        "--samples",
```

```
        type=int,
        default=50,
        help="Number of new samples per iteration. Eg: --samples 50",
    )
    parser.add_argument(
        "--dim", type=int, default=2, help="Dimensions of working space. Eg: --dim 2"
    )
    parser.add_argument(
        "--seed",
        type=int,
        default=1,
        help="Random seed for reproducibility. Eg: --seed 1",
    )
    parser.add_argument(
        "--stop_time",
        type=int,
        default=60,
        help="When to stop the algorithm. Eg: --stop_time 60",
    )
    parser.add_argument(
        "--fast",
        action="store_true",
        help="Whether or not to only plot the final edge list of each iteration. Note: when this flag is set the vis is also considered set. Eg: --fast",
    )
    # Parse the arguments.
    opt = parser.parse_args()
    # Return the arguments.
    if opt.fast:
        opt.vis = True
    return opt


if __name__ == "__main__":
    # If no arguments are passed, then print the help message.
    if len(sys.argv) == 1:
        sys.argv.append("--help")

    # Parse the arguments.
    opt = parse_opt()
    print(f"{CYELLOW2}{opt}{CEND}")

    # Make sure the start and goal coordinates are of the correct dimension.
    assert len(opt.start) == opt.dim
    assert len(opt.goal) == opt.dim

    # Start the experiment.
    main(**vars(opt))
```

3.2 Bit-Star

```
#! /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,
```

```python
        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.
```

```python
        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.

```

```python
        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.
```

```python
        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
```

```python
    ) -> 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:
```

```python
            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.
```

```python
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
```

3.3 Map

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

import random
import numpy as np
from Node import Node


class Map:
    """Map class for BIT*. This class is used to represent the map. It contains the start and goal coordinates, the obstacles, the map, the free and occupied sets, and the f_hat map."""

    def __init__(self, start: Node, goal: Node, occ_grid: np.array) -> None:
        """Initialize the Map class with the start, goal and the occupancy grid.

        Args:
            start (np.array): Start coordinates of the robot in the form np.array([x, y]).
            goal (np.array): Goal coordinates of the robot in the form np.array([x, y]).
            occ_grid (np.array, optional): Occupancy grid of the map which is a 2D numpy array of 0s (occupied) and 1s (free).
        """
        # Set start and goal.
        self.start = start
        self.goal = goal

        self.start_arr = self.start.np_arr
        self.goal_arr = self.goal.np_arr

        # Obstacles set
        self.obstacles = set()
        # Dimensions of the search space.
        self.dim = 2

        # 2D occupancy grid of the map with 0s (occupied) and 1s (free). This is converted from an image.
        self.map = occ_grid
        # Get all the indices of the free cells.
        ind = np.argwhere(self.map > 0)
        # Create a set of tuples of the free cells from the indices for faster lookup.
        self.free = set(list(map(lambda x: tuple(x), ind)))
        # Get all the indices of the occupied cells.
        ind = np.argwhere(self.map == 0)
        # Create a set of tuples of the occupied cells from the indices for faster lookup.
        self.occupied = set(list(map(lambda x: tuple(x), ind)))
        # Get the f_hat map.
        self.get_f_hat_map()

    def sample(self) -> tuple:
        """Sample a random point from the free set. This is used to generate a new node in the tree. We don’t use this in the current implementation as it can theoretically be slower than the new_sample function if the obstacle set is large.

        Returns:
            tuple: Random point sampled from the free set.
        """
        # Sample until a point is found in the free set.
        while True:
            # Sample a random point uniformly from the map and in the continuous space.
            x, y = np.random.uniform(0, self.map.shape[0]), np.random.uniform(
                0, self.map.shape[1]
            )
            # Convert the point to an integer tuple and check if it is in the free set.
            if (int(x), int(y)) in self.free:
                # Return the point.
                return (x, y)

    def new_sample(self) -> tuple:
        """Sample a random point from the free set. This is used to generate a new node in the tree. This is a modified version of the sample function which first samples a random point from the free set and then adds a random noise to it to generate
```

```
        a new point and return it if it is in the free set. This could theoretically be faster than the sample function especially if the obstacle set is large.

        Returns:
            tuple: Random point sampled from the free set.
        """
        # Sample until a point is found in the free set.
        while True:
            # Sample from the free set.
            free_node = random.sample(list(self.free), 1)[0]
            # Add a random noise to the sampled point.
            noise = np.random.uniform(0, 1, self.dim)
            # Add the noise to the sampled point.
            new_node = free_node + noise
            # If the integer tuple of the new node is in the free set, return it.
            if (int(new_node[0]), int(new_node[1])) in self.free:
                return new_node

    def get_f_hat_map(self) -> None:
        """Get the f_hat map which is the heuristic map used by the BIT* algorithm. This is the sum of the L2 norm of the distance from the goal and the start to the current point. This is precomputed for all nodes in the map and stored in a 2D numpy array as a lookup table"""
        # Get the dimensions of the map.
        map_x, map_y = self.map.shape
        # Initialize the f_hat map with zeros of the same dimensions as the map.
        self.f_hat_map = np.zeros((map_x, map_y))
        # For each point in the map, calculate the f_hat value and store it in the f_hat map.
        for x in range(map_x):
            for y in range(map_y):
                # f_hat(x) = g_hat(x) + h_hat(x) for each point x in the map.
                f_hat = np.linalg.norm(
                    np.array([x, y]) - self.goal_arr
                ) + np.linalg.norm(np.array([x, y]) - self.start_arr)
                # Store the f_hat value in the f_hat map.
                self.f_hat_map[x, y] = f_hat
```

3.4 Note

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

import numpy as np
from typing import Generic, TypeVar

# Set global variables to avoid circular definitions.
global start_arr
global goal_arr

# Define the generic type T.
T = TypeVar("T")


class Node(Generic[T]):
    """Node class for BIT*. This class is used to represent a node in the tree. It contains the coordinates of the node, the parent node, the edge cost, gt, children set, start, goal, g_hat, h_hat, and f_hat.

    Args:
        Generic (Node): Generic type for the parent node.
    """

    def __init__(
        self,
        coords: tuple,
        parent: T = None,
        gt: float = np.inf,
        par_cost: float = None,
    ) -> None:
        """Initialize the Node class with the coordinates, parent, gt, par_cost, children, start, goal, g_hat, h_hat, and f_hat.
```

```python

        Args:
            coords (tuple): Coordinates of the node in the form (x, y).
            parent (T, optional): Parent node of the current node. This is an instance of the Node class as the parent node is also a node. Defaults to None.
            gt (float, optional): The cost through the tree to get from the start to the current node. Defaults to np.inf.
            par_cost (float, optional): The cost of the edge from the parent node to the current node. Defaults to None.
        """
        # Extract coordinates from tuple.
        self.x = coords[0]
        self.y = coords[1]
        # Create tuple and numpy array for easy access.
        self.tup = (self.x, self.y)
        self.np_arr = np.array([self.x, self.y])

        # Initialize parent, edge cost (par_cost), and g_t.
        self.parent = parent
        self.par_cost = par_cost
        self.gt = gt
        # Initialize the children set.
        self.children = set()

        # Initialize start and goal.
        global start_arr
        self.start = start_arr
        global goal_arr
        self.goal = goal_arr

        # Generate g_hat and h_hat.
        self.g_hat = self.gen_g_hat()
        self.h_hat = self.gen_h_hat()
        # f_hat is the sum of g_hat and h_hat.
        self.f_hat = self.g_hat + self.h_hat

    def gen_g_hat(self) -> float:
        """Generate the g_hat value for the current node. This is the L2 norm between the current node and the start node.

        Returns:
            g_hat (float): The g_hat value for the current node.
        """
        # Return the L2 norm between the current node and the start node.
        return np.linalg.norm(self.np_arr - self.start)

    def gen_h_hat(self) -> float:
        """Generate the h_hat value for the current node. This is the L2 norm between the current node and the goal node.

        Returns:
            h_hat (float): The h_hat value for the current node.
        """
        # Return the L2 norm between the current node and the goal node.
        return np.linalg.norm(self.np_arr - self.goal)

    def __str__(self) -> str:
        """String representation of the Node class.

        Returns:
            str: String representation of the Node class.
        """
        # Return the string representation of the tuple.
        return str(self.tup)

    def __repr__(self) -> str:
        """String representation of the Node class.

        Returns:
            str: String representation of the Node class.
        """
        # Return the string representation of the tuple.
        return str(self.tup)
```

3.5 Printcolours

```
CEND = "\33[0m"

CBOLD = "\33[1m"
CLIGHTEN = "\33[2m"
CITALIC = "\33[3m"
CURL = "\33[4m"
CBLINK = "\33[5m"

CBLINK2 = "\33[6m"
CSELECTED = "\33[7m"
CHIDE = "\33[8m"
CSCORE = "\33[9m"
CDURL = "\33[21m"

CBLACK = "\33[30m"
CRED = "\33[31m"
CGREEN = "\33[32m"
CYELLOW = "\33[33m"
CBLUE = "\33[34m"
CVIOLET = "\33[35m"
CBEIGE = "\33[36m"
CWHITE = "\33[37m"

CBLACKBG = "\33[40m"
CREDBG = "\33[41m"
CGREENBG = "\33[42m"
CYELLOWBG = "\33[43m"
CBLUEBG = "\33[44m"
CVIOLETBG = "\33[45m"
CBEIGEBG = "\33[46m"
CWHITEBG = "\33[47m"

CGREY = "\33[90m"
CRED2 = "\33[91m"
CGREEN2 = "\33[92m"
CYELLOW2 = "\33[93m"
CBLUE2 = "\33[94m"
CVIOLET2 = "\33[95m"
CBEIGE2 = "\33[96m"
CWHITE2 = "\33[97m"

CGREYBG = "\33[100m"
CREDBG2 = "\33[101m"
CGREENBG2 = "\33[102m"
CYELLOWBG2 = "\33[103m"
CBLUEBG2 = "\33[104m"
CVIOLETBG2 = "\33[105m"
CBEIGEBG2 = "\33[106m"
CWHITEBG2 = "\33[107m"
```

3.6 Visualize

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

import numpy as np
import json
from matplotlib.patches import Ellipse
import matplotlib.pyplot as plt
import os
import cv2
from typing import List, Tuple
import tqdm


class Visualizer:
    """Visualizer class for visualizing the BIT* algorithm."""

    def __init__(
```

```
        self, start: np.array, goal: np.array, occ_map: np.array, output_dir: str
    ) -> None:
        """Initialize the visualizer.

        Args:
            start (np.array): Start point.
            goal (np.array): Goal point.
            occ_map (np.array): Occupancy map of the environment.
            output_dir (str): Output directory to save the plots.
        """
        print(output_dir)
        # Set of all edges in the tree.
        self.edges = set()
        # Final path from start to goal.
        self.final_path = None
        # Initial cost to go from start to goal.
        self.ci = np.inf

        # Simulation number.
        self.sim = 0
        # Map of the environment converted to RGB.
        self.occ_map = cv2.cvtColor(occ_map, cv2.COLOR_BGR2RGB)
        # Start and goal points as numpy arrays.
        self.start = start
        self.goal = goal

        # All the new edges, removed edges, final paths, costs, and final edges over all the simulations.
        (
            self.all_new_edges,
            self.all_rem_edges,
            self.all_final_paths,
            self.all_cis,
            self.all_final_edge_list,
        ) = (
            [],
            [],
            [],
            [],
            [],
        )
        # A dictionary of all the lines in the plot.
        self.lines = {}

        # Initialize the plot.
        self.fig, self.ax = plt.subplots(figsize=(20, 20))
        # The output directory to save the plots.
        self.output_dir = output_dir

    def read_json(self, folder: str, max_iter: int = np.inf) -> None:
        """Reads the json files from the log directory.

        Args:
            folder (str): Name of the log directory.
            max_iter (int, optional): If nothing is given it reads all the simulations logs if a number is given 0 <= max_iter <= len(simulations) then that amount of simulations are read. Defaults to np.inf meaning all files are read.
        """
        # Sort the files in the folder.
        files = sorted(os.listdir(folder))
        # If max_iter is not given then read all the files.
        max_iter = min(max_iter, len(files))
        for i in tqdm.tqdm(range(max_iter), desc="Reading JSON files", total=max_iter):
            # Open one file at a time and read the data.
            with open(os.path.join(folder, files[i]), "r") as f:
                # Load the data from the json file.
                data = json.load(f)
                # Get all new edges and append it to the list.
                self.all_new_edges.append(data["new_edges"])
                # Get all remove edges and append it to the list.
                self.all_rem_edges.append(data["rem_edges"])
                # Get all final paths and append it to the list.
                self.all_final_paths.append(data["final_path"])
                # Get all the costs and append it to the list.
                self.all_cis.append(np.array(data["ci"]))
```

```python
                # Get all the final edges and append it to the list.
                self.all_final_edge_list.append(data["final_edge_list"])

    def draw_final_path(self, path: List[Tuple[float, float]]) -> None:
        """Draw the final path from start to goal.

        Args:
            path (list): Path from start to goal.
        """
        # If the path is empty then return.
        if len(path) == 0:
            return
        # Convert to a numpy array for easy plotting.
        path = np.array(path)
        # Split the path into x and y coordinates.
        x, y = path[:, 0], path[:, 1]
        # Plot the path.
        self.ax.plot(
            y,
            x,
            color="darkorchid",
            lw=4,
            label="Final Path",
        )

    def draw_ellipse(self, ci: float, colour: str = "dimgrey") -> None:
        """Draws the Prolate Hyperspheroid (PHS) for the given ci.

        Args:
            ci (float): Cost to go from start to goal.
            colour (str, optional): The color of the ellipse while plotting. Defaults to "dimgrey".
        """
        # Return if ci is infinity.
        if ci == np.inf:
            return
        # Calculate the minimum ci which is the L2 norm from the start to the goal.
        cmin = np.linalg.norm(self.goal - self.start)
        # Get the center of the ellipse.
        center = (self.start + self.goal) / 2.0
        # Get the first radius of the Prolate Hyperspheroid (PHS).
        r1 = ci
        # Get the second radius of the Prolate Hyperspheroid (PHS).
        r2 = np.sqrt(ci**2 - cmin**2)
        # Get the angle of the Prolate Hyperspheroid (PHS) with respect to the x-axis.
        theta = np.arctan2(self.goal[0] - self.start[0], self.goal[1] - self.start[1])
        # Convert the angle from radians to degrees.
        theta = np.degrees(theta)
        # Use matplotlib patches to draw the ellipse.
        patch = Ellipse(
            (center[1], center[0]),
            r1,
            r2,
            angle=theta,
            color=colour,
            fill=False,
            lw=5,
            ls="--",
            label="Prolate Hyperspheroid (PHS)",
        )
        # Add the ellipse to the plot.
        self.ax.add_patch(patch)

    def draw_edge(self, edge: List[List[List[float]]]) -> None:
        """Draws the edge between two points.

        Args:
            edge (List[List[List[float]]]): List of tuples of the form [[[x1, y1], [x2, y2]], [[x2, y2], [x3, y3]], ... ].
        """
        # Convert the edge to a tuple for easy indexing.
        edge_tup = tuple(map(tuple, edge))

        # Plot the edge between the two points.
        l = self.ax.plot(
```

```
            [edge[0][1], edge[1][1]],
            [edge[0][0], edge[1][0]],
            color="sandybrown",
            lw=2,
            marker="x",
            markersize=4,
            markerfacecolor="darkcyan",
            markeredgecolor="darkcyan",
            label="Branches",
        )
        # Add the line to the dictionary.
        self.lines[edge_tup] = 1

    def draw_tree(self, sim: int) -> None:
        """This is a slow drawing function. It draws the tree from the start to the goal as the algorithm progresses through a simulation.

        Args:
            sim (int): Simulation number.
        """
        # Get the start and goal.
        start = self.start
        goal = self.goal
        # Calls redraw_map to redraw the map.
        self.redraw_map(sim)

        # Get the figure and axes.
        fig = self.fig
        ax = self.ax

        # Loop through all the edges and draw them.
        for i in range(len(self.all_new_edges[sim])):
            # Get the new edge, removed edge, final path and cost to go with each step in the simulation.
            new_edge = self.all_new_edges[sim][i]
            rem_edge = self.all_rem_edges[sim][i]
            path = self.all_final_paths[sim][i]
            ci = self.all_cis[sim][i]

            # Remove the edges that are no longer in the tree.
            if rem_edge:
                for rem_e in rem_edge:
                    rem_e_tup = tuple(map(tuple, rem_e))
                    try:
                        ax.lines.remove(self.lines[rem_e_tup][0])
                        self.edges.remove(rem_e_tup)
                    except:
                        continue

            # Add the new edges to the tree.
            if new_edge is not None:
                new_e_tup = tuple(map(tuple, new_edge))
                self.edges.add(new_e_tup)
                self.draw_edge(new_edge)

            # Plot the final path if it exists.
            if path is None:
                if self.final_path is not None:
                    self.draw_final_path(self.final_path)
            else:
                self.final_path = path
                self.draw_final_path(path)

            # Plot the PHS for the current cost to go.
            self.draw_ellipse(ci, colour="dimgrey")
            # Plot the start and goal.
            ax.plot(
                start[1],
                start[0],
                color="red",
                marker="*",
                markersize=20,
            )
            ax.plot(
                goal[1],
```

```python
                goal[0],
                color="blue",
                marker="*",
                markersize=20,
            )
            # Remove the legend to avoid duplicates and add the legend again.
            handles, labels = self.ax.get_legend_handles_labels()
            by_label = dict(zip(labels, handles))
            plt.legend(
                by_label.values(),
                by_label.keys(),
                bbox_to_anchor=(1.05, 1.0),
                loc="upper left",
            )

            # Show the plot and pause for a short time without blocking.
            plt.show(block=False)
            plt.pause(0.0001)

    def draw_fast(self, sim: int) -> None:
        """This is a fast drawing function. It draws the final tree, final path and PHS for a given simulation.

        Args:
            sim (int): Simulation number.
        """
        # Get the final path, edges, and cost to go for the given simulation.
        self.edges = self.all_final_edge_list[sim]
        path = self.all_final_paths[sim][-1]
        ci = self.all_cis[sim][0]

        # Redraw the map.
        self.redraw_map(sim)

        # If the final path exists, draw it.
        if path is not None:
            self.final_path = path
            self.draw_final_path(path)

        # Draw the PHS.
        self.draw_ellipse(ci, colour="dimgrey")
        # Draw the start and goal.
        self.ax.plot(
            self.start[1],
            self.start[0],
            color="red",
            marker="*",
            markersize=20,
        )
        self.ax.plot(
            self.goal[1],
            self.goal[0],
            color="blue",
            marker="*",
            markersize=20,
        )
        # Remove the legend to avoid duplicates and add the legend again.
        handles, labels = self.ax.get_legend_handles_labels()
        by_label = dict(zip(labels, handles))
        plt.legend(
            by_label.values(),
            by_label.keys(),
            bbox_to_anchor=(1.05, 1.0),
            loc="upper left",
        )
        # Save the plot at the given output directory and show the plot.
        plt.savefig(f"{self.output_dir}/Bitstar_Simulation_{sim:02d}.png")
        # Show the plot and pause for a short time without blocking.
        plt.show(block=False)
        plt.pause(1)

    def draw(self, sim: int, fast: bool = False) -> None:
        """This function calls either the fast or slow drawing function depending on the value of fast flag.

```

```python
        Args:
            sim (int): Simulation number.
            fast (bool, optional): If True, the fast drawing function is called. Defaults to False.
        """
        # If the fast flag is True, call the fast drawing function.
        if fast:
            self.draw_fast(sim)
        # Else, call the slow drawing function.
        else:
            self.draw_tree(sim)

    def redraw_map(self, sim: int) -> None:
        # Clear the current plot.
        plt.close()
        self.fig, self.ax = plt.subplots(figsize=(20, 20))
        # Get the occupancy map for this simulation.
        im = self.ax.imshow(self.occ_map, cmap=plt.cm.gray, extent=[0, 100, 100, 0])

        # Loop through all the edges and draw them.
        for e in self.edges:
            self.draw_edge(e)

        # Plot the start and goal.
        self.ax.plot(
            self.start[1],
            self.start[0],
            color="red",
            marker="*",
            markersize=20,
            label="Start",
        )
        self.ax.plot(
            self.goal[1],
            self.goal[0],
            color="blue",
            marker="*",
            markersize=20,
            label="Goal",
        )
        # Set the title and axis labels.
        self.ax.set_title(f"BIT* - Simulation {sim}", fontsize=30)
        self.ax.set_xlabel(r"X$\rightarrow$", fontsize=10)
        self.ax.set_ylabel(r"Y$\rightarrow$", fontsize=10)

        # Remove the legend to avoid duplicates and add the legend again.
        handles, labels = self.ax.get_legend_handles_labels()
        by_label = dict(zip(labels, handles))
        plt.legend(
            by_label.values(),
            by_label.keys(),
            bbox_to_anchor=(1.05, 1.0),
            loc="upper left",
        )
        # Set the axis limits.
        self.ax.set_xlim(-10, 110)
        self.ax.set_ylim(-10, 110)
```

IV. Demo thuật toán

4.1 Các bước chi tiết (đã điều chỉnh cho lưới ô vuông)

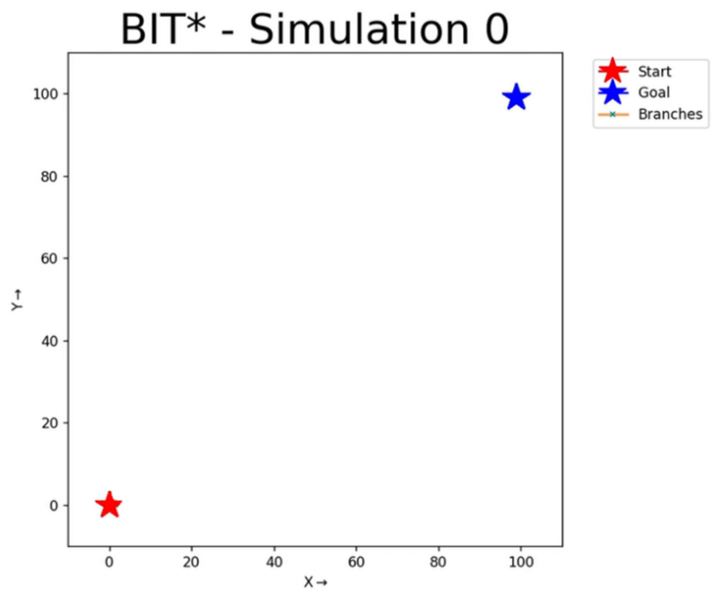
1. **Khởi tạo:**
   - Điểm bắt đầu (Start): $(1, 1)$
   - Điểm kết thúc (Goal): $(9, 9)$
   - Chướng ngại vật: Ô $(5, 5)$.
   - Cây tìm kiếm (Tree): Ban đầu chỉ chứa nút gốc là ô $(1, 1)$.
   - Tập hợp các nút đã mở (Open Set): Ban đầu chứa nút gốc $(1, 1)$ với chi phí $g = 0$ và chi phí heuristic h (ví dụ: khoảng cách Manhattan đến mục tiêu).
   - Tập hợp các nút đã đóng (Closed Set): Ban đầu rỗng.
   - Hàm heuristic (h): Sử dụng khoảng cách Manhattan:
     $h(n) = |n_x - goal_x| + |n_y - goal_y|$. Ví dụ:
     $h(1, 1) = |1 - 9| + |1 - 9| = 8 + 8 = 16$.
2. **Lặp** (trong mỗi lần lặp, chúng ta mô phỏng việc xử lý một "batch" các mẫu tiềm năng):
   - Chọn nút tốt nhất để mở rộng: Thay vì lấy mẫu ngẫu nhiên, trong mỗi bước của một phiên bản đơn giản hóa cho lưới, chúng ta có thể chọn nút có chi phí $f = g + h$ thấp nhất từ Open Set để mở rộng.
   - Mở rộng nút: Lấy nút hiện tại (gọi là current_node) từ Open Set có chi phí f thấp nhất và chuyển nó sang Closed Set.
   - Tìm các ô lân cận hợp lệ: Xác định các ô lân cận của current_node (ví dụ: 4 ô xung quanh - Bắc, Nam, Đông, Tây). Một ô lân cận là hợp lệ nếu nó nằm trong lưới và không phải là chướng ngại vật và chưa nằm trong Closed Set.
   - Đánh giá và thêm lân cận vào Open Set: Đối với mỗi ô lân cận hợp lệ (neighbor):
     * Tính chi phí mới để đến neighbor từ điểm bắt đầu thông qua current_node:
       $g_{new} = g(current\_node) + cost(current\_node, neighbor)$.
       Trong lưới ô vuông, chi phí di chuyển giữa các ô lân cận thường là 1.
     * Tính chi phí heuristic cho neighbor:
       $h(neighbor) = |neighbor_x - 9| + |neighbor_y - 9|$.

    * Tính chi phí f cho neighbor: $f(neighbor) = g_{new} + h(neighbor)$.
    * Nếu neighbor chưa có trong Open Set hoặc nếu chúng ta tìm thấy một đường đi đến neighbor với chi phí g_{new} thấp hơn chi phí g hiện tại của nó trong Open Set, cập nhật thông tin của neighbor (chi phí g, chi phí f, và nút cha là current_node) và thêm nó vào Open Set (hoặc cập nhật nó nếu đã có).
  - Kiểm tra mục tiêu: Nếu một trong các ô lân cận là ô mục tiêu $(9, 9)$, chúng ta đã tìm thấy đường đi. Dừng thuật toán và truy vết ngược lại từ ô mục tiêu theo các nút cha để tái tạo đường đi.
  - Mô phỏng "Batch" và "Pruning": Thay vì xử lý một lô lớn các mẫu ngẫu nhiên cùng lúc và sau đó tỉa bớt, chúng ta có thể coi mỗi lần lặp mở rộng một nút tốt nhất như một phần của việc "xem xét" một "batch" các khả năng lân cận.

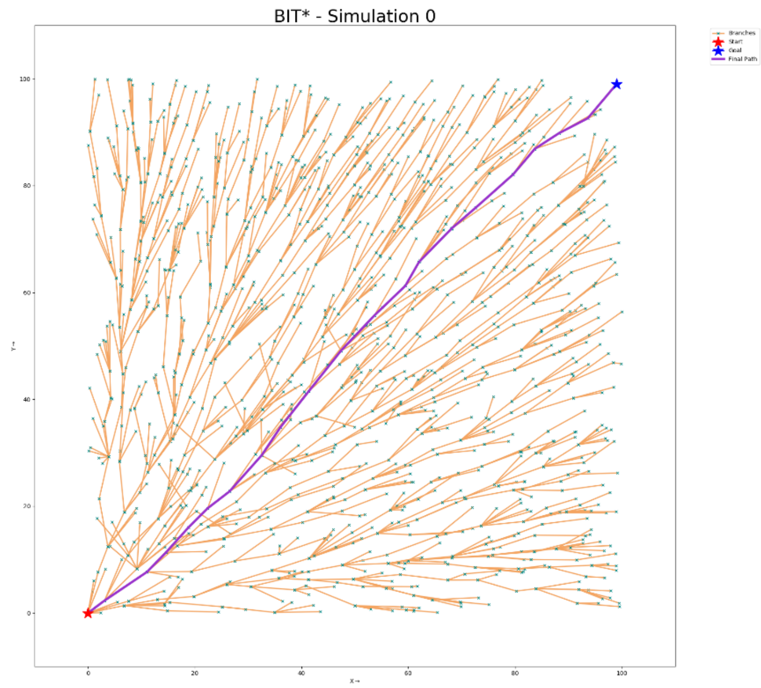
3. **Truy vết đường đi:** Khi ô mục tiêu được thêm vào Closed Set, chúng ta có thể tái tạo đường đi bằng cách bắt đầu từ ô mục tiêu và theo dõi các nút cha cho đến khi chúng ta đến ô bắt đầu.

4. **Hình ảnh minh họa:**

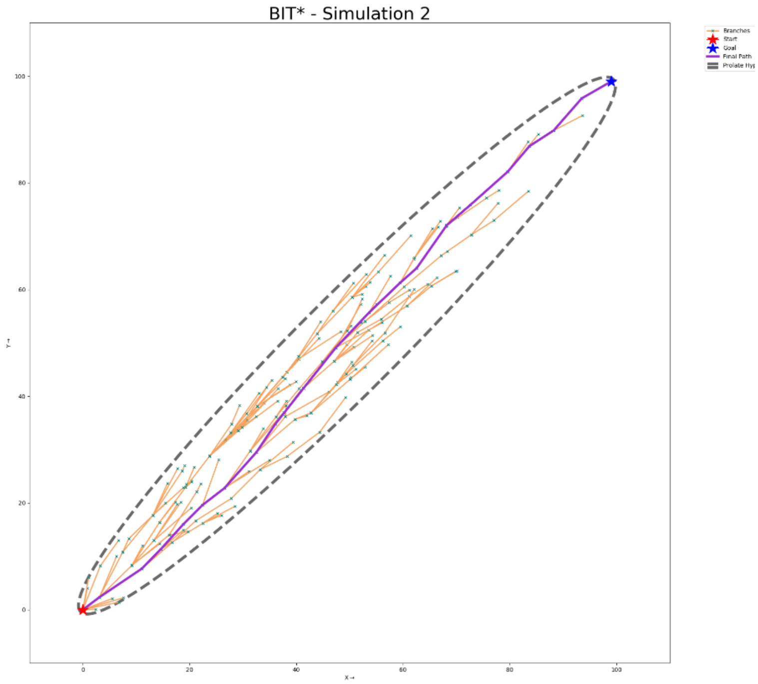
   – Khởi tạo Start and Goal



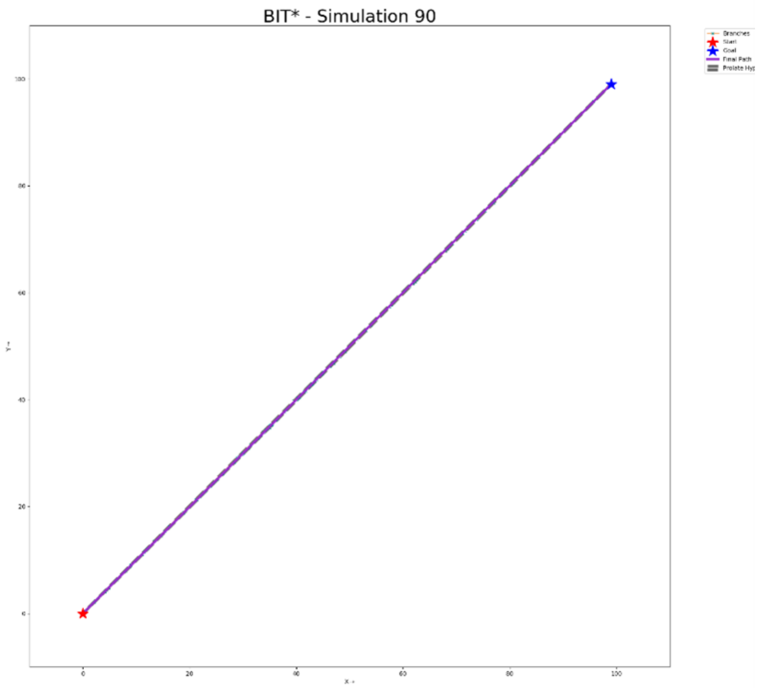
   – Tìm được đường đi từ Start đến Goal



– Truy vết lại các đường đi để tìm đường đi ngắn nhất



– Tìm được đường đi ngắn nhất từ Start đến Goal



4.2 Minh họa một vài bước đầu tiên

– **Bắt đầu:**

  * Open Set: $\{(1,1) : f = 16, g = 0, parent = None\}$
  * Closed Set: $\{\}$

– **Lặp 1:**

  * Chọn $(1,1)$ từ Open Set (có f thấp nhất). Chuyển $(1,1)$ sang Closed Set.
  * Các ô lân cận hợp lệ của $(1,1)$ là $(2,1)$ và $(1,2)$ (giả sử chỉ di chuyển 4 hướng).
  * Đánh giá $(2,1)$: $g = 1$, $h = |2-9| + |1-9| = 7+8 = 15$, $f = 1+15 = 16$, $parent = (1,1)$. Thêm $(2,1)$ vào Open Set.
  * Đánh giá $(1,2)$: $g = 1$, $h = |1-9| + |2-9| = 8+7 = 15$, $f = 1+15 = 16$, $parent = (1,1)$. Thêm $(1,2)$ vào Open Set.
  * Open Set: $\{(2,1) : f = 16, g = 1, parent = (1,1)\}$, $\{(1,2) : f = 16, g = 1, parent = (1,1)\}$
  * Closed Set: $\{(1,1)\}$

  [Tiếp tục các bước lặp khác tương tự...]

– **Kết thúc - Truy vết đường đi:**

  * Khi node $(9,9)$ được mở rộng, dừng thuật toán.
  * Dùng parent của từng node để truy ngược đường đi từ đích về gốc.
  * Ví dụ: $(9,9) \leftarrow (8,9) \leftarrow (7,9) \leftarrow \ldots \leftarrow (1,1)$

4.3 Kết luận

– Tuy demo này không sinh mẫu ngẫu nhiên, nó mô phỏng tư tưởng của BIT* bằng cách ưu tiên mở rộng các node tốt nhất (f nhỏ nhất) - chính là triết lý "informed search".

– Mỗi bước lặp đóng vai trò như xử lý một "batch nhỏ".

– Chiến lược chỉ mở rộng những khả năng hứa hẹn giúp tiết kiệm chi phí tính toán và hướng đến nghiệm tối ưu.

V. Nhận xét.

Batch-Informed RRT* (BIT*) là một thuật toán lập kế hoạch chuyển động hiện đại, mang lại sự cải tiến đáng kể so với các thuật toán dựa trên lấy mẫu truyền thống như RRT* và Informed RRT*. Một trong những điểm nổi bật nhất của BIT* là khả năng kết hợp giữa phương pháp lấy mẫu ngẫu nhiên và kỹ thuật tìm kiếm heuristic theo thứ tự, tận dụng được cả khả năng mở rộng trong không gian lớn và khả năng tập trung tìm kiếm vào các vùng hứa hẹn.

BIT* hoạt động dựa trên ý tưởng rằng một tập hợp các mẫu có thể tạo thành một đồ thị hình học ngẫu nhiên (Random Geometric Graph — RGG) ẩn. Nhờ nhận thức này, BIT* tổ chức việc khám phá không gian như một quá trình tìm kiếm có hướng dẫn dựa trên heuristic, nhưng trong bối cảnh của một tập mẫu rời rạc, được sinh ra theo từng lô (batch). Điều này cho phép thuật toán vừa đảm bảo tính khả thi theo thời gian thực — nghĩa là có thể nhanh chóng tìm được nghiệm sơ bộ, vừa không ngừng cải thiện chất lượng nghiệm khi có thêm thời gian và tài nguyên tính toán.

Về mặt lý thuyết, BIT* giữ được hai tính chất rất quan trọng trong lập kế hoạch chuyển động, đó là tối ưu bất tiệm cận và hoàn chỉnh xác suất. Điều này đồng nghĩa rằng, với số lượng mẫu tăng lên vô hạn, BIT* sẽ chắc chắn tìm được nghiệm tối ưu nếu tồn tại. Đây là một ưu điểm lớn, đặt BIT* ngang hàng với RRT* và Informed RRT*, nhưng với tốc độ hội tụ nghiệm tốt hơn trong thực tế.

Một lợi thế quan trọng khác của BIT* là khả năng tái sử dụng thông tin từ các lô mẫu trước. Thay vì loại bỏ hoàn toàn thông tin cũ khi sinh thêm mẫu mới, BIT* tận dụng các cạnh và các trạng thái đã được khám phá để tiết kiệm chi phí tính toán, đồng thời đảm bảo rằng tiến trình tìm kiếm không bị lặp lại một cách lãng phí. Cách thiết kế này giúp BIT* đạt được sự cân bằng rất tốt giữa thăm dò (exploration) và khai thác (exploitation).

Mặt khác, nhờ sử dụng một hàm heuristic để hướng dẫn việc mở rộng tìm kiếm, BIT* có khả năng tập trung nhanh vào các khu vực có tiềm năng chứa nghiệm tốt, thay vì phải thăm dò toàn bộ không gian một cách ngẫu nhiên như RRT*. Điều này đặc biệt có ích trong các bài toán lập kế hoạch có không gian trạng thái lớn, phức tạp hoặc có nhiều chướng ngại vật.

Tuy nhiên, việc sử dụng batch cũng mang đến những thách thức nhất định. Kích thước batch (số lượng mẫu mỗi lô) cần được lựa chọn cẩn thận để đảm bảo hiệu suất: batch quá nhỏ có thể dẫn đến việc tìm kiếm kém hiệu quả, trong khi batch quá lớn có thể làm tăng chi phí tính toán mỗi vòng lặp. Ngoài ra, BIT* yêu cầu phải quản lý các hàng đợi ưu tiên (priority queue) cho các đỉnh và các cạnh, điều này đòi hỏi việc triển khai phải được tối ưu tốt nếu muốn thuật toán hoạt động hiệu quả trong các tình huống thực tế.

Nhìn chung, BIT* đại diện cho một sự kết hợp tinh tế giữa lý thuyết và thực tiễn: nó tận dụng các thuộc tính hình học ngẫu nhiên để đảm bảo khả năng khám phá, đồng thời ứng dụng các chiến lược tìm kiếm heuristic để cải thiện tốc độ hội tụ. Nhờ đó, BIT* đã chứng minh được tính ưu việt của mình trong nhiều thử nghiệm, đặc biệt là ở các môi trường có không gian trạng thái cao chiều và các ràng buộc động phức tạp.