Thuật Toán BIT*

1 Giới thiệu thuật toán BIT*

Batch Informed Trees (BIT*) là một trong những đóng góp nổi bật trong lĩnh vực lập hoạch chuyển động robot tối ưu, đánh dấu bước ngoặt trong việc kết hợp hiệu quả hai hướng tiếp cận truyền thống: tìm kiếm có định hướng theo đồ thị (heuristic graph search) và lập hoạch chuyển động dựa trên mẫu ngẫu nhiên (sampling-based motion planning). Thông qua mô hình hóa không gian cấu hình dưới dạng đồ thị hình học ngẫu nhiên ngầm (implicit Random Geometric Graph – RGG) và áp dụng chiến lược tìm kiếm theo lô định hướng heuristic (batch-based heuristic search), BIT* vừa đạt được tính mở rộng thời gian thực (anytime performance), vừa bảo toàn tính tối ưu tiệm cận (asymptotic optimality). Đây là hai đặc tính tưởng chừng mâu thuẫn nhưng lại được thống nhất khéo léo trong BIT*, điều chưa từng đạt được trọn vẹn trong các thuật toán trước đây như RRT, PRM, RRT* hay FMT*.

Bối cảnh và động lực

Lập hoạch chuyển động (motion planning) là một bài toán cơ bản nhưng đầy thách thức trong robot học, có tác động sâu rộng đến khả năng hoạt động tự chủ của các hệ thống như robot di động, robot thao tác, phương tiện bay không người lái (UAV), xe tự hành (autonomous vehicles), và thiết bị trợ giúp thông minh trong môi trường không cấu trúc. Bài toán yêu cầu tìm ra một chuỗi các trạng thái khả thi (tránh va chạm, tôn trọng ràng buộc vật lý) từ điểm bắt đầu đến mục tiêu trong không gian cấu hình (C-space) - không gian có thể có hàng chục hoặc hàng trăm bậc tự do (degrees of freedom - DOF).

Về mặt lý thuyết, bài toán có thể được giải bằng cách tìm đường đi tối ưu trong đồ thị biểu diễn không gian trạng thái. Tuy nhiên, hai trở ngại lớn xuất hiện:

- Không gian nhiều chiều (high-dimensionality) khiến việc rời rạc hóa toàn cục trở nên bất khả thi do số lượng trạng thái tăng theo cấp số mũ – hiện tượng được gọi là "lời nguyền chiều không gian".
- Tính phi tuyến và phức tạp hình học của ràng buộc va chạm, động học và điều khiển khiến cấu trúc không gian trở nên bất quy tắc, gây khó khăn cho các phương pháp dựa trên lưới hoặc hàm chi phí truyền thống.

Do đó, hai hướng tiếp cận chính đã được hình thành:

- **Tìm kiếm theo đồ thị (Graph-based search):** Đại diện là các thuật toán như Dijkstra và A*, hoạt động trên không gian trạng thái rời rạc. Những thuật toán này đảm bảo tối ưu cục bộ trên đồ thị và sử dụng heuristic để dẫn hướng tìm kiếm hiệu quả. Tuy nhiên, hiệu suất của chúng suy giảm nghiêm trọng trong không gian có nhiều bậc tự do.

- **Lập hoạch dựa trên mẫu (Sampling-based planning):** Ra đời nhằm tránh rời rạc hóa toàn cục, cho phép làm việc trực tiếp trong không gian liên tục thông qua việc lấy mẫu ngẫu nhiên. Các thuật toán như PRM và RRT đã mở ra hướng đi khả thi cho các bài toán phức tạp. Tuy nhiên, bản chất ngẫu nhiên và không có định hướng khiến quá trình tìm kiếm kém hiệu quả và không tối ưu trong thực tế.

Khoảng trống học thuật và nhu cầu tích hợp

Tính đến hiện tại, vẫn chưa có một thuật toán nào thực sự kết hợp được tất cả các yếu tố sau một cách hệ thống:

1. Khả năng tìm kiếm có định hướng dựa trên heuristic như A*;
2. Mô hình hóa không gian cấu hình thông qua mẫu ngẫu nhiên như RRT;
3. Tối ưu tiệm cận như RRT*;
4. Tính bất cứ lúc nào (anytime) như RRT-Connect.

Các cải tiến như Informed RRT* đã đưa heuristic vào quá trình lấy mẫu, và FMT* đã dùng lý thuyết đồ thị hình học ngầm để tăng tốc tìm kiếm. Tuy nhiên, cả hai đều còn hạn chế: hoặc là thiếu tính mở rộng theo thời gian, hoặc là thiếu cơ chế tái sử dụng thông tin từ các lần tìm kiếm trước đó.

Chính vì vậy, một câu hỏi nghiên cứu quan trọng được đặt ra: Liệu có thể thiết kế một thuật toán duy nhất vừa có thể mở rộng hiệu quả, vừa đảm bảo tìm kiếm có thứ tự và tối ưu tiệm cận không?

BIT*: Một khuôn khổ thống nhất

Batch Informed Trees (BIT*) ra đời như một câu trả lời khẳng định cho bài toán nêu trên. Cốt lõi của BIT* là sự kết hợp hài hòa giữa mô hình đồ thị ngầm và heuristic định hướng theo phong cách A*. Cụ thể:

- BIT* xem mỗi tập mẫu như một phần của một đồ thị hình học ngẫu nhiên ngầm (RGG) và tìm kiếm trong cấu trúc đó mà không cần xây dựng đồ thị tường minh.
- Thay vì mở rộng từng mẫu đơn lẻ, BIT* xử lý các lô mẫu (batches), cho phép tái sử dụng các thông tin tìm kiếm và đánh giá lại các miền có tiềm năng cải thiện lời giải.
- BIT* dùng hàng đợi ưu tiên để sắp xếp các cạnh ứng viên dựa trên ước lượng chi phí ($\hat{f} = \hat{g} + \hat{h}$), tương tự như trong A*, giúp tập trung vào các vùng "hứa hẹn" hơn trong không gian cấu hình.
- BIT* kế thừa triết lý tìm kiếm tăng dần (incremental search) từ Lifelong Planning A*, giúp tận dụng tối đa các kết quả tìm kiếm trước đó, hạn chế việc tính toán lại và kiểm tra va chạm dư thừa.

Đóng góp lý thuyết và thực nghiệm

BIT* là một bước tiến đáng kể trong lý thuyết lập hoạch robot. Nó được chứng minh là hoàn chỉnh theo xác suất (probabilistically complete) và tối ưu tiệm cận (asymptotically optimal). Trên thực nghiệm, BIT* thể hiện sự vượt trội rõ rệt:

- Trong các không gian mô phỏng $\mathbb{R}^2$ và $\mathbb{R}^8$, BIT* đạt tốc độ hội tụ và chất lượng lời giải vượt trội so với RRT, RRT*, Informed RRT* và FMT*.
- Trên robot HERB với không gian cấu hình 14 bậc tự do, BIT* không chỉ tìm được lời giải thành công trong các cấu hình hẹp (narrow passage), mà còn cải thiện đáng kể chi phí quỹ đạo và thời gian tính toán.

Tổng thể, BIT* không chỉ là một thuật toán cụ thể mà còn là một khuôn khổ tổng quát có khả năng mở rộng, có thể thích nghi với các dạng ràng buộc động học, môi trường thay đổi và nhiều loại robot khác nhau. Nó mở ra hướng tiếp cận mạnh mẽ cho các bài toán tối ưu toàn cục, đặc biệt trong lĩnh vực robot học thế hệ mới và trí tuệ nhân tạo hiện đại.

2 Trình bày thuật toán BIT*

1. Ý tưởng chính của BIT*

- **Mô hình hóa không gian trạng thái:** BIT* xem tập hợp các điểm lấy mẫu ngẫu nhiên trong không gian trạng thái như một đồ thị ngẫu nhiên hình học ẩn (Implicit Random Geometric Graph - RGG). Các đỉnh là các điểm mẫu, các cạnh được xác định dựa trên khoảng cách địa lý (ví dụ: k-láng giềng gần nhất hoặc trong bán kính r).
- **Tìm kiếm có hướng dẫn heuristic:** Thuật toán sử dụng heuristic để ưu tiên mở rộng các nút có khả năng dẫn đến đường đi tối ưu, tương tự như A*. Heuristic này giúp tìm kiếm có thứ tự ưu tiên, giảm số lượng nút cần mở rộng.
- **Tìm kiếm theo từng lô (batch):** Thuật toán xử lý các điểm mẫu theo từng lô, mỗi lô bổ sung thêm các điểm mẫu mới để làm tăng độ chính xác của đồ thị ẩn. Mỗi lô được tìm kiếm theo heuristic, và kết quả được cải thiện dần theo thời gian (anytime).
- **Tái sử dụng thông tin:** BIT* áp dụng kỹ thuật tìm kiếm tăng dần (incremental search) như trong Lifelong Planning A* (LPA*) để tái sử dụng thông tin từ các lô trước, giúp tăng hiệu quả tính toán.

2. Định nghĩa bài toán và các biến

- **Không gian trạng thái:** $X \subseteq \mathbb{R}^n$
- **Không gian tự do:** $X_{\text{free}} = X \setminus X_{\text{obs}}$
- **Điểm bắt đầu:** $x_{\text{start}} \in X_{\text{free}}$
- **Tập điểm đích:** $X_{\text{goal}} \subset X_{\text{free}}$

- **Đường đi:** $\sigma : [0, 1] \to X$
- **Hàm chi phí:** $s(\sigma)$
- **Mục tiêu:**

$$\sigma^* = \arg\min_{\sigma \in \Sigma} \{s(\sigma) \mid \sigma(0) = x_{\text{start}},\ \sigma(1) \in X_{\text{goal}},\ \forall t,\ \sigma(t) \in X_{\text{free}}\}$$

- **Tập điểm mẫu:** X_{samples} (lấy ngẫu nhiên trong X_{free})
- **Cây tìm kiếm:** $T = (V, E)$
  - V: tập đỉnh (các trạng thái đã mở rộng)
  - E: tập cạnh (các kết nối hợp lệ)

3. Mô tả chi tiết thuật toán

1. **Khởi tạo:**
   - Lấy mẫu ngẫu nhiên một tập điểm X_{samples} trong không gian trạng thái tự do X_{free}, bao gồm điểm bắt đầu x_{start} và điểm đích X_{goal}.
   - Xác định tham số đồ thị RGG (bán kính r hoặc số lượng láng giềng k) dựa trên số lượng mẫu để đảm bảo tính tối ưu xác suất.
2. **Xây dựng cây tìm kiếm:**
   - Tạo cây $T = (V, E)$ bắt đầu từ x_{start}.
   - Mở rộng cây theo heuristic dựa trên ước lượng chi phí:

$$\hat{f}(x) = \hat{g}(x) + \hat{h}(x)$$

     Trong đó:
     - $\hat{g}(x)$: ước lượng chi phí từ x_{start} đến x (cost-to-come).
     - $\hat{h}(x)$: ước lượng chi phí từ x đến X_{goal} (cost-to-go).
   - Mở rộng các cạnh không va chạm và ưu tiên theo giá trị $\hat{f}(x)$.
3. **Kết thúc batch:**
   - Khi tìm được đường đi hoặc không thể mở rộng thêm, kết thúc batch.
   - Lưu lại chi phí đường đi tốt nhất c_{best}.
4. **Thêm batch mới:**
   - Thêm một lô mẫu mới vào tập điểm, tập trung vào vùng có thể cải thiện đường đi hiện tại (ví dụ vùng elip giới hạn bởi c_{best}).
   - Cập nhật lại cây tìm kiếm bằng cách tái sử dụng thông tin từ batch trước theo phương pháp incremental search.
5. **Lặp lại:**
   - Quay lại bước 2 với lô mẫu mới, tiếp tục cải thiện đường đi.

4. Công thức tính toán chính

- **Heuristic (ước lượng chi phí qua x):**

$$\hat{f}(x) = \hat{g}(x) + \hat{h}(x)$$

- **Tập trạng thái tiềm năng cải thiện lời giải:**

$$X_{\hat{f}} := \{x \in X \mid \hat{f}(x) \leq c_{\text{best}}\}$$

- **Cây tìm kiếm:**

$$T = (V, E), \quad V \subseteq X_{\text{free}}, \quad E \subseteq V \times V$$

- **Chi phí thực tế trên cây:**

$$g_T(x) : \text{ chi phí thực tế từ } x_{\text{start}} \text{ đến } x \text{ trên cây } T$$

5. Ưu điểm của BIT*

- Kết hợp ưu điểm của A* và RRT*: Tìm kiếm có hướng dẫn heuristic như A* giúp mở rộng có thứ tự ưu tiên, đồng thời sử dụng lấy mẫu ngẫu nhiên như RRT* để mở rộng không gian tìm kiếm liên tục.
- Anytime và asymptotically optimal: Thuật toán nhanh chóng tìm được lời giải ban đầu và cải thiện dần theo thời gian, hội tụ về lời giải tối ưu khi số mẫu tăng vô hạn.
- Hiệu quả trong không gian nhiều chiều: Thích hợp cho các bài toán robot phức tạp như điều khiển cánh tay robot 14 bậc tự do.

3 Code thuật toán BIT* (Python)

Listing 1: Mã nguồn Python mô phỏng BIT*

```
import pygame
import random
import heapq
import math
import time

infinity = 10**10
fps = 60
sleep_time = 0.2
clock = pygame.time.Clock()

class Node:
  def __init__(self, state, start, goal):
    self.state = state
      self.g = self.distance(state, start)
      self.h = self.distance(state, goal)
```

```
      self.f = self.g + self.h
      self.g_real = infinity
      self.parent = self

  def__lt__(self, other):
    self_cost = self.cost()
    other_cost = other.cost()
    if self_cost == other_cost:
      return self.g_real < other.g_real
    else:
      return self_cost < other_cost

  def cost(self):
    return min(self.g_real + self.h, infinity)
  def distance(self, node1, node2):
    x1, y1 = node1
    x2, y2 = node2
    return ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** 0.5

class Edge:
  def __init__(self, node1, node2):
    self.node1 = node1
    self.node2 = node2
    node2.parent = node1
    self.temp = self.distance(node1, node2) + self.node2.h

  def __lt__(self, other):
    self_cost = self.cost()
    other_cost = other.cost()
    if self_cost == other_cost:
      return self.node1.g_real < other.node1.g_real
    else:
      return self_cost < other_cost

  def cost(self):
    return min(self.node1.g_real + self.distance(self.node1, self.
       node2) + self.node2.h, infinity)

  def distance(self, node1, node2):
    x1, y1 = node1.state
    x2, y2 = node2.state
    return ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** 0.5

class BIT_star_algorithm:
  def__init__(self, map_dimensions):
    self.map_w, self.map_h = map_dimensions
    self.white = (255, 255, 255)
    self.red = (255, 0, 0)
    self.green = (0, 255, 0)
    self.blue = (0, 0, 255)
    self.black = (0, 0, 0)
```

```
    self.yellow = (255, 255, 0)
    self.gray = (128, 128, 128)
    self.navy = (0, 0, 128)
    self.random_node_numbers = 50
    self.node_radius = 5
    self.edge_thickness = 2
    self.nodes = []
    self.edges = []
    self.random_nodes = []
    self.samples =[]
    self.obstacles = []
    self.r = infinity
    self.qe = []
    self.qv = []
    self.v = []
    self.vold = []
    self.new_path = []
    self.old_path = []
    self.cbest = infinity
    self.check_flag = True
    self.start_state = None
    self.goal_state = None
    self.start_node = None
    self.goal_node = None
    self.map = pygame.display.set_mode(map_dimensions)
    self.map.fill(self.white)
    self.map_name = "BIT_star_algorithm"
    pygame.display.set_caption(self.map_name)

  def update_start_goal(self):
    self.start_node = Node(self.start_state, self.start_state,
       self.goal_state)
    self.start_node.g_real = 0
    self.goal_node = Node(self.goal_state, self.start_state, self.
       goal_state)
    self.goal_node.g_real = infinity
    self.nodes.append(self.start_node)
    self.samples.append(self.goal_node)

  def draw_map(self):
    self.map.fill(self.white)
    pygame.draw.circle(self.map, self.green, self.start_state,
       self.node_radius + 5)
    pygame.draw.circle(self.map, self.yellow, self.goal_state,
       self.node_radius + 10)
    self.draw_obstacles()

  def draw_obstacles(self):
    for obs in self.obstacles:
      pygame.draw.rect(self.map, self.gray, obs)
```

```
def draw_edges(self):
  for node1, node2 in self.edges:
    pygame.draw.line(self.map, self.blue, node1.state, node2.
       state, self.edge_thickness)

def draw_nodes(self):
  for node in self.nodes:
    pygame.draw.circle(self.map, self.blue, node.state, self.
       node_radius)

def draw_samples(self):
  for node in self.samples:
    pygame.draw.circle(self.map, self.navy, node.state, self.
       node_radius)

def draw_node(self, node, color):
  pygame.draw.circle(self.map, color, node.state, self.
     node_radius)
  self.wait_time()

def draw_edge(self, node1, node2, color):
  pygame.draw.line(self.map, color, node1.state, node2.state,
     self.edge_thickness)
  self.wait_time()

def draw_old_path(self):
  previous_node = self.goal_node
  for node in self.old_path:
    pygame.draw.line(self.map, self.blue, previous_node.state,
       node.state, self.edge_thickness + 2)
    self.wait_time()
    previous_node = node
    pygame.draw.circle(self.map, self.blue, node.state, self.
       node_radius + 2)
    self.wait_time()

def draw_new_path(self):
  previous_node = self.goal_node
  for node in self.new_path:
    pygame.draw.line(self.map, self.red, previous_node.state,
       node.state, self.edge_thickness + 2)
    self.wait_time()
    previous_node = node
    pygame.draw.circle(self.map, self.red, node.state, self.
       node_radius + 2)
    self.wait_time()

def draw_ellipse(self):
  if self.cbest == infinity:
    return
  x1, y1 = self.start_state
```

```
    x2, y2 = self.goal_state
    center_x = (x1 + x2) / 2
    center_y = (y1 + y2) / 2
    c = self.distance(self.start_node, self.goal_node) / 2
    a = self.cbest / 2
    b = math.sqrt(a ** 2 - c ** 2)
    angle = math.degrees(math.atan2(y2 - y1, x2 - x1))
    ellipse_surface = pygame.Surface((2 * a, 2 * b), pygame.
        SRCALPHA)
    ellipse_rect = ellipse_surface.get_rect(center=(a, b))
    pygame.draw.ellipse(ellipse_surface, self.black, ellipse_rect,
        width = 2)
    ellipse_surface_rotate = pygame.transform.rotate(
        ellipse_surface, -angle)
    ellipse_rect_rotate = ellipse_surface_rotate.get_rect(center=(
        center_x, center_y))
    self.map.blit(ellipse_surface_rotate, ellipse_rect_rotate)

def wait_time(self):
    #pygame.event.clear()
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            exit()
    clock.tick(fps)
    pygame.display.update()
    time.sleep(sleep_time)

def make_random_state(self):
    x = random.randint(0, self.map_w)
    y = random.randint(0, self.map_h)
    return x, y

def make_random_nodes(self):
    node_list = []
    for i in range(0, self.random_node_numbers):
        node_flag = True
        node = None
        while node_flag:
            node_state = self.make_random_state()
            node = Node(node_state, self.start_state, self.goal_state)
            if self.is_free_state(node_state):
                node_flag = False
            else:
                node_flag = True
        node_list.append(node)
    self.random_nodes = node_list.copy()

def on_obstacle(self, obs, state):
    x_check, y_check = state
    x1, y1 = obs[0], obs[1]
    x2, y2 = x1 + obs[2], y1 + obs[3]
```

```
    if x1 <= x_check and x_check <= x2 and y1 <= y_check and
       y_check <= y2:
      return True
    else:
      return False

  def is_free_state(self, state):
    for obs in self.obstacles:
      if self.on_obstacle(obs, state):
        return False
    return True

  def collision_obs(self, node1, node2):
    x1, y1 = node1.state
    x2, y2 = node2.state
    for i in range(0, 101):
      u = i / 100
      x = u * x2 + (1 - u) * x1
      y = u * y2 + (1 - u) * y1
    if not self.is_free_state((x, y)):
      return True
    return False

  def distance(self, node1, node2):
    x1, y1 = node1.state
    x2, y2 = node2.state
    return ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** 0.5

  def distance_real(self, node1, node2):
    if self.collision_obs(node1, node2):
      return infinity
    else:
      return self.distance(node1, node2)

  def prune(self, c):
    self.samples = [x for x in self.samples if x.f < c]
    self.v = [v for v in self.v if v.f <= c]
    temp = self.edges.copy()
    self.edges = []
    for (v, w) in temp:
      if v.f > c or w.f > c:
        w.parent = w
        w.g_real = infinity
        self.draw_edge(v, w, self.white)
        self.draw_node(w, self.white)
      else:
        self.edges.append((v, w))
    self.samples.extend([v for v in self.v if v.g_real == infinity
       ])
    self.v = [v for v in self.v if v.g_real < infinity]
```

```
def add_edge_e(self, node1, node2):
  node2.g_real = node1.g_real + self.distance(node1, node2)
  node2.parent = node1
  self.edges.append((node1, node2))
  self.draw_edge(node1, node2, self.blue)

def add_edge_qe(self, edge):
  heapq.heappush(self.qe, edge)

def add_node_qv(self, node):
  heapq.heappush(self.qv, node)

def expand_vertex(self):
  heapq.heapify(self.qv)
  v = heapq.heappop(self.qv)
  xnear = [x for x in self.samples if self.distance(v, x) <=
     self.r]
  for x in xnear:
    if v.g + self.distance(v, x) + x.h < self.goal_node.g_real:
      self.add_edge_qe(Edge(v, x))
  if not v in self.vold:
    vnear = [x for x in self.v if self.distance(v, x) <= self.r]
      for w in vnear:
        if not (v, w) in self.edges:
          if v.g + self.distance(v, w) + w.h < self.goal_node.
             g_real:
            if v.g_real + self.distance_real(v, w) < w.g_real:
              self.add_edge_qe(Edge(v, w))

def update_radius(self):
  r = 2 * 1.1
  r = r * (((1 + 1 / 2) * (self.map_w * self.map_h / math.pi))
     ** 0.5)
  n = len(self.v) + len(self.samples)
  r = r * ((math.log(n) / n) ** 0.5)
  print("␣neiborhood_radius␣:␣", r)
  return r

def get_best_value(self, pq):
  heapq.heapify(pq)
  if len(pq) == 0:
    return infinity
  else:
    return pq[0].cost()

def bit_one_batch(self):
  self.v = self.nodes.copy()
  self.qe = []
  self.qv = []
  j = 0
  while True:
```

```
if len(self.qe) == 0 and len(self.qv) == 0:
  j = j + 1
  if j == 10:
    self.check_flag = False
    break
  if self.goal_node.g_real < self.cbest:
    break
  self.make_random_nodes()
  self.samples.extend(self.random_nodes.copy())
  self.prune(self.cbest)
  self.draw_samples()
  print(j, "␣count_samples␣:␣", len(self.samples))
  self.draw_samples()
  self.vold = self.v.copy()
  self.qv = self.v.copy()
  self.r = self.update_radius()
  self.wait_time()

while len(self.qv) > 0 and self.get_best_value(self.qv) <=
   self.get_best_value(self.qe):
  self.expand_vertex()
if len(self.qe) == 0:
  continue

heapq.heapify(self.qe)
edge = heapq.heappop(self.qe)
vm, xm = edge.node1, edge.node2

if vm.g_real + self.distance(vm, xm) + xm.h < self.goal_node
   .g_real:
  if vm.g + self.distance_real(vm, xm) + xm.h < self.
     goal_node.g_real:
    if vm.g_real + self.distance_real(vm, xm) < xm.g_real:
      if xm in self.v:
        for (x, y) in self.edges:
          if y == xm:
            xm.parent = xm
            xm.g_real = infinity
            self.draw_edge(x, y, self.white)
            self.edges.remove((x, y))

          else:
            self.samples = [x for x in self.samples if x !=
               xm]
            self.v.append(xm)
            self.add_node_qv(xm)
          self.add_edge_e(vm, xm)

          temp = self.qe.copy()
          self.qe = []
          for edge in temp:
```

```
                    x, y = edge.node1, edge.node2
                    if y == xm and x.g_real + self.distance(x, xm)
                       >= xm.g_real:
                      continue
                    else:
                      self.add_edge_qe(Edge(x, y))

        else:
          self.qe = []
          self.qv = []
      print("-"*30)
      self.nodes = self.v.copy()
      if self.cbest < infinity:
        self.old_path = self.new_path.copy()
      self.new_path = self.path_to_goal().copy()
      self.cbest = self.goal_node.g_real

    def path_to_goal(self):
      path = []
      node = self.goal_node
      while node != self.start_node:
        path.append(node)
        node = node.parent
      path.append(self.start_node)
      return path

def get_mouse_state():
    while True:
      for event in pygame.event.get():
        if event.type == pygame.QUIT:
          exit()
        if event.type == pygame.MOUSEBUTTONDOWN:
          if event.button == 1:
            return pygame.mouse.get_pos()

def get_mouse_obstacles(mapp):
    mapp.map.fill(mapp.white)
    mapp.wait_time()
    running = True
    temp = True
    while running:
      for event in pygame.event.get():
        if event.type == pygame.QUIT:
          exit()
        if event.type == pygame.MOUSEBUTTONDOWN:
          if event.button == 3:
            running = False
          if event.button == 1:
            if temp:
              first_state = pygame.mouse.get_pos()
              temp = False
```

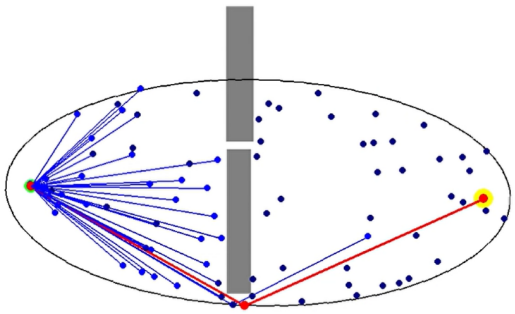
```
            else:
              second_state = pygame.mouse.get_pos()
              temp = True
              rect = pygame.Rect(first_state,(second_state[0]-
                 first_state[0],second_state[1]-first_state[1]))
              mapp.obstacles.append(rect)
              mapp.draw_obstacles()
              mapp.wait_time()

def get_mouse_start_goal(mapp):
  mapp.start_state = get_mouse_state()
  print("Toa␣do␣den␣diem␣xuat␣phat␣la␣:␣", mapp.start_state[0],
     mapp.start_state[1])
  pygame.draw.circle(mapp.map, mapp.green, mapp.start_state, mapp.
     node_radius + 5)
  mapp.wait_time()
  mapp.goal_state = get_mouse_state()
  print("Toa␣do␣diem␣dich␣muon␣den␣la␣:␣", mapp.goal_state[0],
     mapp.goal_state[1])
  pygame.draw.circle(mapp.map, mapp.yellow, mapp.goal_state, mapp.
     node_radius + 10)
  mapp.wait_time()
  mapp.update_start_goal()


def main():
  map_dimensions = (1300, 800)
  sleep_time = 2
  clock = pygame.time.Clock()
  fps = 60

  pygame.init()
  mapp = BIT_star_algorithm(map_dimensions)

  get_mouse_obstacles(mapp)
  get_mouse_start_goal(mapp)

  while mapp.check_flag:
    mapp.draw_map()
    mapp.draw_ellipse()
    clock.tick(fps)
    pygame.display.update()
    time.sleep(sleep_time)
    mapp.draw_nodes()
    mapp.draw_edges()
    mapp.draw_new_path()
    mapp.bit_one_batch()
    mapp.draw_old_path()
    mapp.draw_new_path()
    print("␣cbest_to_goal␣:␣", mapp.cbest)
    print("-"*40)
```

```
            clock.tick(fps)
            pygame.display.update()
            time.sleep(sleep_time)

        pygame.event.clear()
        pygame.event.wait(0)
        pygame.quit()

if __name__ == '__main__':
    main()
```

4 Demo thuật toán

4.1 Test 1:

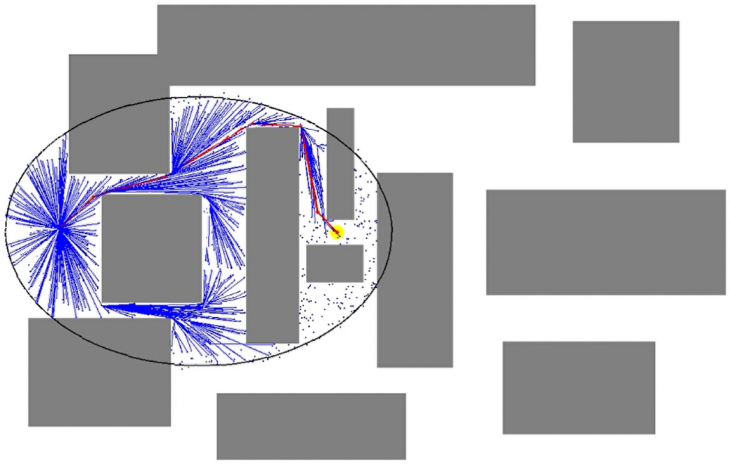
- Tọa độ điểm đầu xuất phát là: 257 407
- Tọa độ điểm đích muốn đến là: 950 426
- Đường dẫn VIDEO DEMO: `https://shorturl.at/ytbfx`



Hình 1: Chú thích cho ảnh

4.2 Test 2:

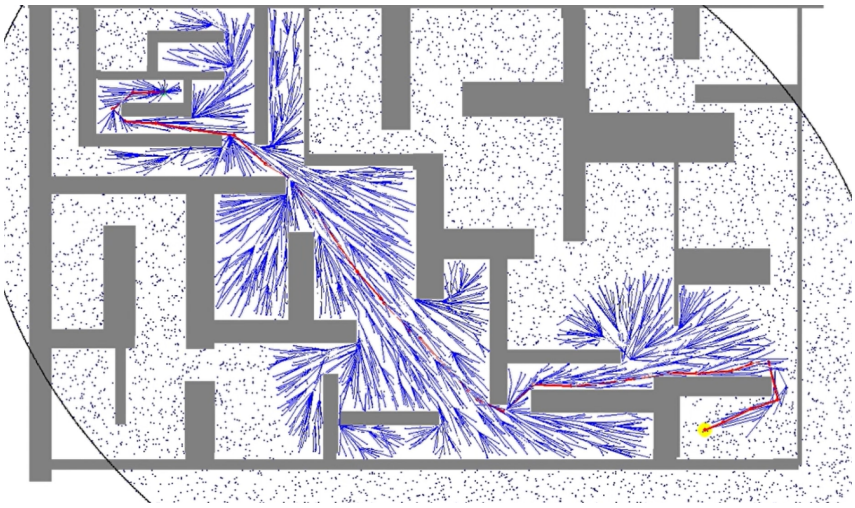
- Tọa độ điểm đầu xuất phát là: 110 413
- Tọa độ điểm đích muốn đến là: 535 419
- Độ dài đường đi ngắn nhất là: 592.6938482233808
- Đường dẫn VIDEO DEMO: `https://shorturl.at/Tvnza`



Hình 2: Chú thích cho ảnh

4.3 Test 3:

- Đường đi ngắn nhất là: 1482.3113809097642
- Đường dẫn VIDEO DEMO: `https://shorturl.at/X1MvP`



Hình 3: Chú thích cho ảnh