HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
BỘ MÔN TOÁN RỜI RẠC 2

BÁO CÁO BÀI TẬP LỚN

ĐỀ TÀI: THUẬT TOÁN BIT* VÀ A*

Nhóm 04	
Sinh viên thực hiện	**Mã SV**
Phạm Quang Toản	B23DCCN385
Lê Hải Long	B23DCCN499
Tô Quang Trung	B23DCCN863
Trương Minh Sơn	B23DCCN726
Nguyễn Quang Huy	B23DCCN401
Lê Bùi Quốc Huy	B23DCCN387
Phạm Đình Thi	B23DCCN779
Vũ Đình Trọng	B23DCCN851

Hà Nội – 2025

THUẬT TOÁN A*

1 Giới thiệu thuật toán A*

1.1 Giới thiệu chung về thuật toán A*

Thuật toán A* là một thuật toán tìm kiếm thông minh trong khoa học máy tính, được sử dụng để tìm đường đi ngắn nhất từ một điểm xuất phát đến một điểm đích trong một đồ thị có trọng số. A* là một mở rộng của thuật toán Dijkstra nhưng cải tiến hiệu suất bằng cách sử dụng hàm heuristic để định hướng tìm kiếm.

Tổng chi phí tại mỗi điểm được tính theo công thức:

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

Trong đó:

- $g(n)$: Chi phí thực tế từ nút xuất phát đến nút hiện tại n.
- $h(n)$: Ước lượng chi phí từ nút n đến nút đích.

Tính tối ưu: Nếu hàm heuristic $h(n)$ là *admissible* (tức là $h(n) \leq h^*(n)$, với $h^*(n)$ là chi phí thực tế), A* đảm bảo tìm ra đường đi ngắn nhất.

1.2 Mục đích sử dụng

Mục đích chính của thuật toán A* bao gồm:

- Tìm đường đi ngắn nhất hoặc chi phí thấp nhất từ điểm xuất phát đến điểm đích.
- Tối ưu hóa lập kế hoạch đường đi trong các ứng dụng yêu cầu hiệu quả và tính chính xác.
- Giảm thời gian tìm kiếm bằng hàm heuristic để ưu tiên các đường đi tiềm năng.

1.3 Bối cảnh sử dụng

Thuật toán A* được ứng dụng trong:

- Điều hướng robot: Lập kế hoạch đường đi cho robot di động, tránh chướng ngại vật.
- Trò chơi điện tử: Điều khiển NPC di chuyển hợp lý.
- Định tuyến mạng: Tìm đường đi tối ưu cho dữ liệu qua internet.
- Bản đồ và GPS: Tối ưu hóa lộ trình giao thông.
- Trí tuệ nhân tạo: Giải các bài toán lập kế hoạch trong không gian trạng thái.

1.4 Ưu điểm và hạn chế

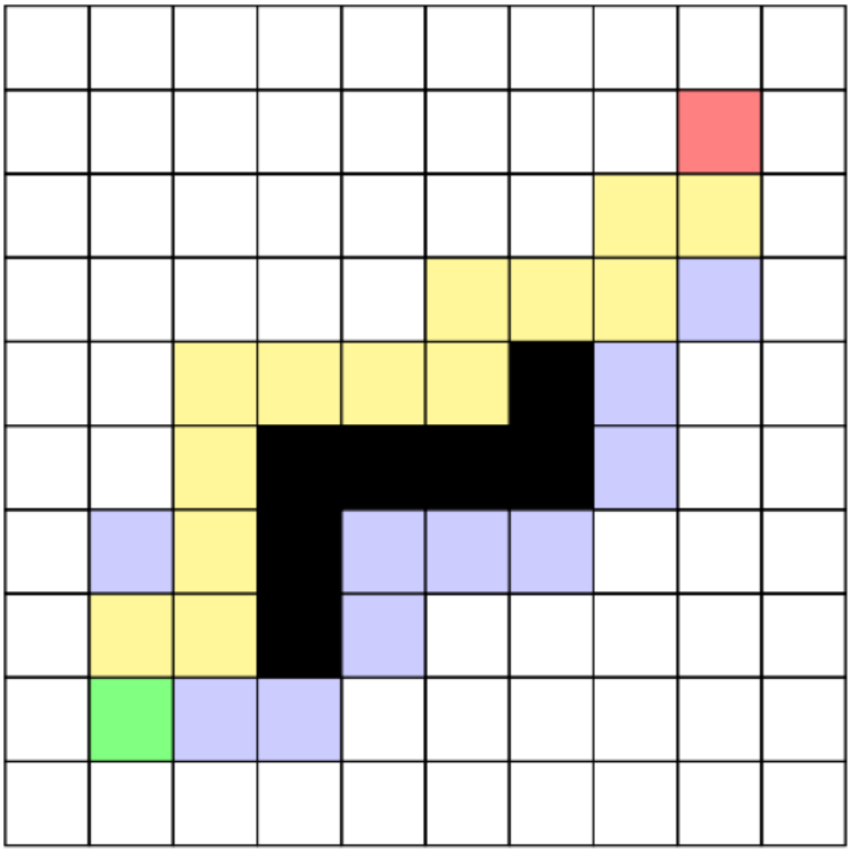
Ưu điểm:

- Hiệu quả: Kết hợp chiến lược của Dijkstra (tìm đường tối ưu) và Greedy Best First Search (tìm nhanh về đích), giúp A* xem xét ít nút hơn và vượt trội về tốc độ trong nhiều trường hợp.
- Tính tối ưu: Với hàm heuristic admissible và consistent, A* luôn đảm bảo tìm ra đường đi có chi phí thấp nhất (ngắn nhất).
- Linh hoạt: Có thể áp dụng trên nhiều loại không gian tìm kiếm, từ lưới 2D, bản đồ trọng số đến đồ thị không đều.
- Khả năng thích nghi: Xử lý hiệu quả trong các môi trường phức tạp và động, đặc biệt khi tích hợp với dữ liệu thời gian thực.
- Tùy chỉnh: Hỗ trợ sử dụng các heuristic khác nhau như khoảng cách Manhattan hoặc Euclidean để phù hợp với bài toán cụ thể.

Hạn chế:

- Yêu cầu bộ nhớ lớn: Lưu trữ danh sách mở dẫn đến nhu cầu bộ nhớ cao.
- Phụ thuộc vào heuristic: Hiệu suất kém nếu heuristic không tốt.
- Không phù hợp cho nhiều đích: Chỉ tối ưu cho một điểm đích cụ thể.
- Chi phí tính toán: Quản lý hàng đợi ưu tiên có thể tốn thời gian.

1.5 Hình minh họa



Hình 1: Minh họa thuật toán A trên lưới 2D: Xanh lá cây là nút bắt đầu, đỏ là nút đích, đen là chướng ngại vật, vàng là đường đi tối ưu, và xanh dương nhạt là các nút đã xem xét.*

2 Trình bày thuật toán A*

2.1 Ý tưởng chính của A*

A* sử dụng hàm đánh giá để mở rộng nút có tổng chi phí ước tính thấp nhất:

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

Trong đó:

- $g(n)$: Chi phí thực tế từ điểm xuất phát đến nút hiện tại.
- $h(n)$: Ước lượng chi phí từ nút hiện tại đến đích (heuristic).
- $f(n)$: Tổng chi phí dự đoán.

Thuật toán luôn ưu tiên nút có $f(n)$ thấp nhất để tiếp tục mở rộng.

2.2 Định nghĩa bài toán và các biến

- **Bài toán:** Tìm đường ngắn nhất từ nút bắt đầu Start đến nút đích Goal trên đồ thị có trọng số.
- **Các biến:**
  - OpenSet: Tập hợp các nút đã phát hiện nhưng chưa mở rộng.
  - $g(n)$: Chi phí thực tế từ Start đến nút n.
  - $h(n)$: Hàm heuristic ước lượng chi phí từ n đến Goal.
  - $f(n) = g(n) + h(n)$: Tổng chi phí ước tính.
  - Parent(n): Nút cha của n để truy vết đường đi.

2.3 Mô tả chi tiết thuật toán

1. Khởi tạo OpenSet chứa nút Start, với $g(\text{Start}) = 0$ và $f(\text{Start}) = h(\text{Start})$; giá trị g của tất cả các nút trong bản đồ là vô cùng (INT_MAX).
2. Trong khi OpenSet không rỗng:
   a. Chọn nút current có giá trị f nhỏ nhất.
   b. Nếu current = Goal, kết thúc và truy vết đường đi.
   c. Di chuyển current sang ClosedSet.
   d. Với mỗi nút neighbor kề current:
      - Nếu neighbor đã có trong ClosedSet, bỏ qua.
      - Tính $tentative\_g = g(current) + cost(current, neighbor)$.

- Nếu $tentative\_g < g(neighbor)$:
  - Cập nhật $Parent(neighbor) = current$.
  - Cập nhật $g(neighbor)$ và $f(neighbor) = g(neighbor) + h(neighbor)$.
  - Thêm neighbor vào OpenSet.

2.4 Công thức tính toán chính

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

Trong đó:

- $g(n)$: Tổng chi phí từ nút bắt đầu đến n.
- $h(n)$: Hàm heuristic. Ví dụ:
  - **Khoảng cách Euclid:**

$$h(n) = \sqrt{(x_{goal} - x_n)^2 + (y_{goal} - y_n)^2}$$

  - **Khoảng cách Manhattan:**

$$h(n) = |x_{goal} - x_n| + |y_{goal} - y_n|$$

2.5 Minh họa quá trình tìm kiếm của A*

- Ví dụ: Tìm đường từ nút A đến G trên đồ thị.
- Bắt đầu từ nút A.
- Xét các nút lân cận và tính f cho từng nút.
- Chọn nút có f nhỏ nhất để mở rộng.
- Lặp lại cho đến khi đến nút G.
- Truy vết đường đi ngược từ G về A bằng Parent.

2.6 Ưu điểm và Nhược điểm của A*

Ưu điểm:

- Tìm kiếm nhanh, tối ưu nếu hàm $h(n)$ chính xác.
- Linh hoạt, dễ mở rộng cho nhiều bài toán.
- Ứng dụng rộng rãi: AI game, robot tự hành, mô phỏng...

Nhược điểm:

- Nếu $h(n)$ không tốt, có thể tiêu tốn bộ nhớ lớn.

2.7 Tóm tắt thuật toán

A* là thuật toán tìm kiếm đường đi ngắn nhất thông minh, kết hợp chi phí thực tế và ước lượng chi phí còn lại thông qua công thức:

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

Nếu hàm $h(n)$ không vượt quá chi phí thực tế (admissible), A* sẽ luôn tìm ra lời giải tối ưu.

3 Code thuật toán

3.1 C++ code thuật toán A*

Listing 1: C++ code thuật toán A*

```
#include <bits/stdc++.h>
using namespace std;

int V, E, s, t;
vector<int> d, truoc, used;

typedef struct
{
  int number;
  double x, y;
  vector<pair<int, int>> ke;
} dinh;

dinh dau, cuoi;

vector<dinh> vt1;

double heuristic(dinh i, dinh t)
{
  return abs(i.x - t.x) + abs(i.y - t.y);
}

struct cmp
{
  bool operator()(dinh a, dinh b)
  {
    double g1 = (double)(d[a.number] + heuristic(a, cuoi));
    double g2 = (double)(d[b.number] + heuristic(b, cuoi));
    return g1 > g2;
  }
};

void Init()
{
  vt1.clear();
```

```
  vt1.resize(V + 1);
  used.clear();
  used.resize(V + 1, 0);
  truoc.clear();
  truoc.resize(V + 1, 0);
  d.clear();
  d.resize(V + 1, INT_MAX);
}

void set_coordinate()
{
  cout << "Nhap␣so␣dinh:␣";
  cin >> V;
  Init();
  cout << "Nhap␣dinh␣va␣toa␣do:␣" << endl;
  for (int i = 1; i <= V; i++)
  {
    dinh tmp;
    cin >> tmp.number >> tmp.x >> tmp.y;
    vt1[tmp.number] = tmp;
  }
}

void set_edge()
{
  cout << "Nhap␣so␣canh:␣";
  cin >> E;
  cout << "Nhap␣danh␣sach␣canh␣co␣trong␣so:␣" << endl;
  for (int i = 1; i <= E; i++)
  {
    int a, b, c;
    cin >> a >> b >> c;
    vt1[a].ke.push_back(make_pair(b, c));
    if (t == 2) vt1[b].ke.push_back(make_pair(a, c));
  }
}

void Result()
{
  if (truoc[t] != 0)
  {
    cout << "Chi␣phi:␣" << d[t] << endl;
    cout << "Duong␣di␣ngan␣nhat:␣";
    vector<int> ans;
    int tmp = t;
    ans.push_back(tmp);
    while (tmp != s)
    {
      tmp = truoc[tmp];
      ans.push_back(tmp);
    }
```

```
    for (int i = 0; i < ans.size() - 1; i++)
    cout << ans[i] << " <- ";
    cout << ans[ans.size() - 1];
  }
  else
  cout << "Khong co duong di tu " << s << " den " << t;
}

void A_star(int s)
{
  priority_queue<dinh, vector<dinh>, cmp> q;
  d[s] = 0;
  truoc[s] = s;
  used[s] = 1;
  for (auto p : vt1[s].ke)
  {
    int u = p.first;
    d[u] = p.second;
  }
  for (int i = 1; i <= V; i++)
  {
    if (i == s)
    continue;
    q.push(vt1[i]);
    truoc[i] = s;
  }

  while (!q.empty())
  {
    int u = q.top().number;
    q.pop();
    used[u] = 1;
    for (auto i : vt1[u].ke)
    {
      int v = i.first;
      if (!used[v] && d[v] > d[u] + i.second)
      {
        d[v] = d[u] + i.second;
        truoc[v] = u;
        q.push(vt1[v]);
      }
    }
  }

  Result();
}

void solve()
{
  set_coordinate();
  set_edge();
```

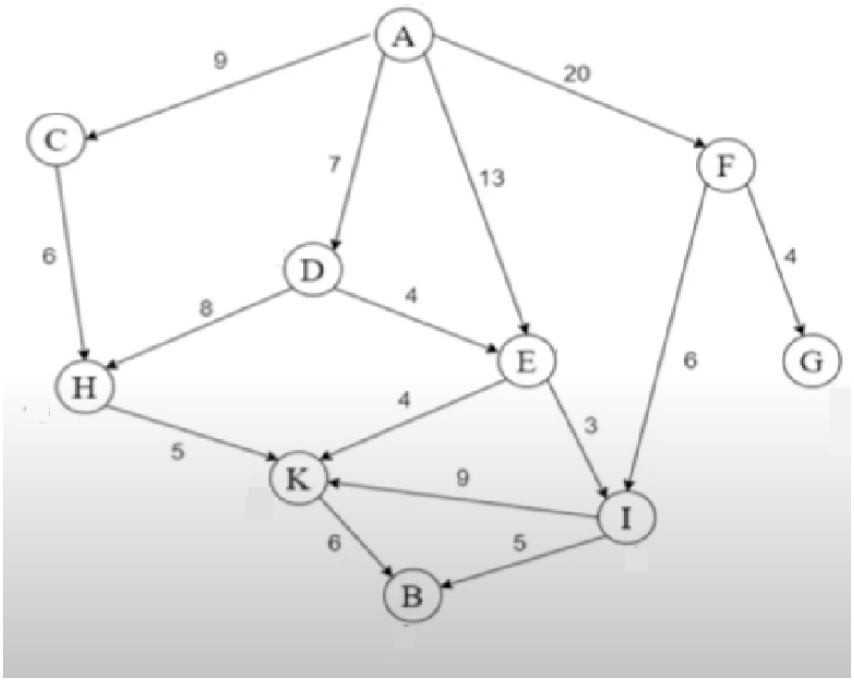
```
    cout << "Nhap dinh bat dau: ";
    cin >> s;
    cout << "Nhap dinh ket thuc: ";
    cin >> t;

    dau = vt1[s];
    cuoi = vt1[t];
    A_star(s);
}

int main()
{
    cout << "Loai do thi: ";  // 1: co huong, 2: vo huong;
    cin >> t;
    solve();
    cout << endl;
}
```

3.2 Test mẫu

Test 1:

```
Loại đồ thị: 1
Nhập số đỉnh: 10
Nhập đỉnh và toạ độ:
1 6 10
2 4 0
3 0 10
4 4 7
5 6 5
6 10 9
7 12 5
8 1 6
9 8 2
10 3 3
Nhập số cạnh: 15
Nhập danh sách cạnh có trọng số:
1 3 9
1 6 20
1 4 7
1 5 13
3 8 6
4 8 8
4 5 4
6 7 4
6 9 6
5 10 4
5 9 3
8 10 5
```

```
9 10 9
9 2 5
10 2 6
Nhập đỉnh bắt đầu: 1
Nhập đỉnh kết thúc: 2
Chi phí: 19
Đường đi ngắn nhất: 2<-9<-5<-4<-1
```



Hình 1: Đồ thị trọng số minh họa cho thuật toán A*.

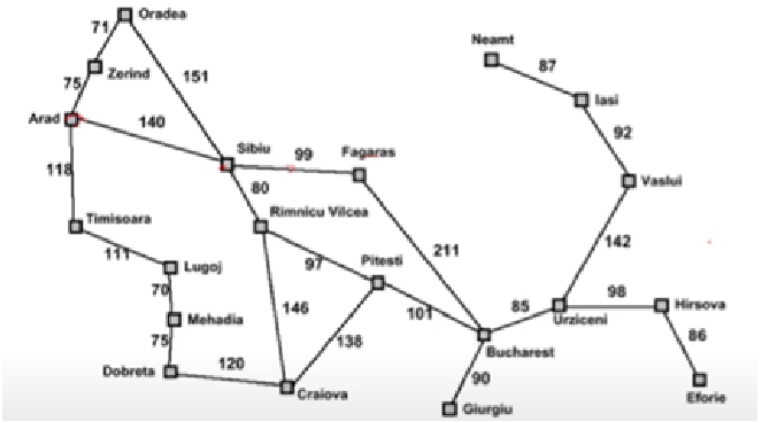
Test 2:

```
Loại đồ thị: 2
Nhập số đỉnh: 20
Nhập đỉnh và toạ độ:
1 110 170
2 370 310
3 220 320
4 110 340
5 530 340
6 310 150
7 360 360
8 510 290
9 470 130
10 130 250
11 130 290
12 430 90
13 115 60
```

```
14 290 270
15 240 220
16 230 150
17 80 230
18 440 270
19 510 190
20 110 110
Nhập số cạnh: 23
Nhập danh sách cạnh có trọng số:
1 16 140
1 17 118
1 20 75
2 6 211
2 7 90
2 14 101
2 18 85
3 4 120
3 14 138
3 15 146
4 11 75
5 8 86
6 16 99
8 18 98
9 12 87
9 19 92
10 11 70
10 17 111
13 16 151
13 20 71
14 15 97
15 16 80
18 19 142
Nhập đỉnh bắt đầu: 1
Nhập đỉnh kết thúc: 2
Chi phí: 418
Đường đi ngắn nhất: 2 <- 14 <- 15 <- 16 <- 1
```

Chú thích:

```
Arad   1
Bucharest  2
Craiova   3
Dobreta      4
Eforie       5
Fagaras      6
Giurgiu      7
Hirsova      8
Iasi         9
```



Hình 2: Minh họa bài toán tìm đường đi ngắn nhất bằng thuật toán A*.

```
Lugoj          10
Mehadia        11
Neamt          12
Oradea         13
Pitesti        14
Rimnicu Vilcea  15
Sibiu          16
Timisoara      17
Urziceni       18
Vaslui         19
Zerind         20
```

4 Demo thuật toán

Listing 2: Demo thuật toán A* với OpenGL

```
#include <GL/glut.h>
#include <iostream>
#include <vector>
#include <queue>
#include <cstdlib>
#include <cmath>
#include <time.h>
#include <ctime>
```

```
using namespace std;

const int MAP_SIZE = 200;
const int WINDOW_SIZE = 800;
const float CELL_SIZE = 2.0f / MAP_SIZE;

struct Node
{
  int x, y;
  float cost, heuristic;
  Node *parent;
  float f() { return cost + heuristic; }
};

struct Compare
{
  bool operator()(Node *a, Node *b)
  {
    return a->f() > b->f();
  }
};

vector<vector<int>> grid(MAP_SIZE, vector<int>(MAP_SIZE, 0));
   // 0: free, 1: obstacle
vector<vector<int>> gScore(MAP_SIZE, vector<int>(MAP_SIZE,
   INT_MAX));
vector<vector<bool>> visited(MAP_SIZE, vector<bool>(MAP_SIZE,
   false));
vector<Node *> visitedNodes;
vector<Node *> path;
priority_queue<Node *, vector<Node *>, Compare> openSet;

int logic = 0, khoi_dau;
bool foundPath = false;
int startX, startY;
int goalX, goalY;

// Khoi tao ban do
void generateMap()
{
  srand(time(0));
  for (int y = 0; y < MAP_SIZE; ++y)
  {
    for (int x = 0; x < MAP_SIZE; ++x)
    {
      if (rand() % 10 == 0)
      grid[y][x] = 1;
      else
      grid[y][x] = 0;
    }
  }
```

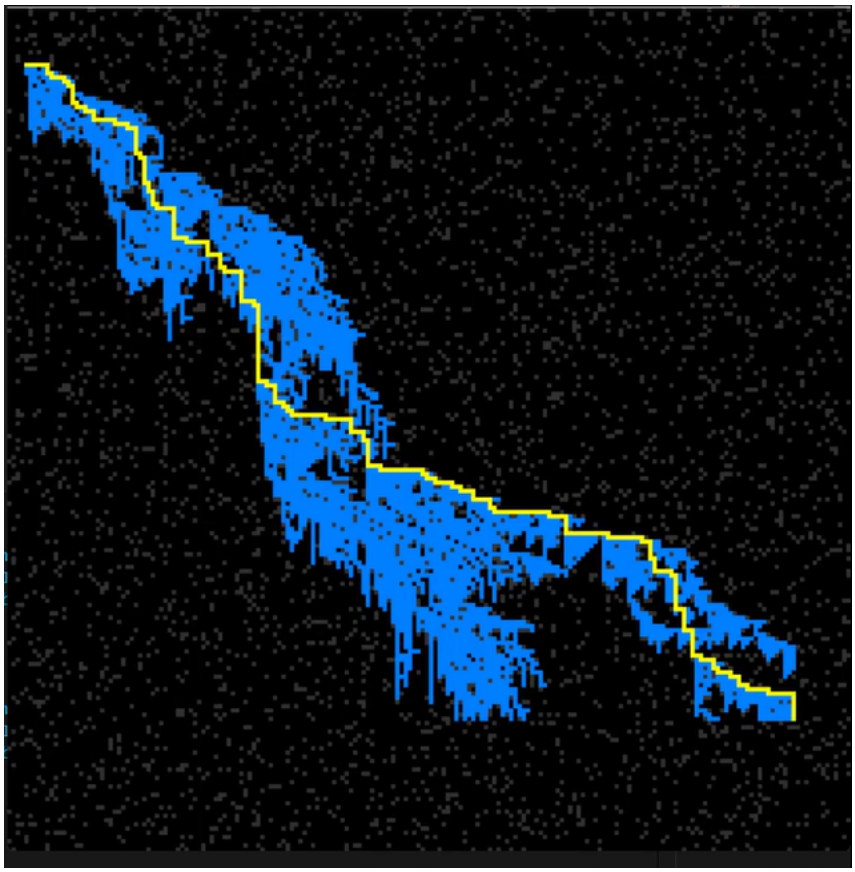
```
}

// Manhattan distance
float heuristic(int x, int y)
{
  return abs(goalX - x) + abs(goalY - y);
}

bool isFree(int x, int y)
{
  return x >= 0 && x < MAP_SIZE && y >= 0 && y < MAP_SIZE &&
     grid[y][x] == 0;
}

void reconstructPath(Node *end)
{
  path.clear();
  for (Node *current = end; current; current = current->parent
     )
  {
    path.push_back(current);
  }
}

// Mo rong mot node trong hang doi
void expandNext()
{
  if (foundPath || openSet.empty())
  return;

  Node *currentNode = openSet.top();
  openSet.pop();

  int x = currentNode->x;
  int y = currentNode->y;

  if (visited[y][x])
  return;

  visited[y][x] = true;
  visitedNodes.push_back(currentNode);

  if (x == goalX && y == goalY)
  {
    foundPath = true;
    reconstructPath(currentNode);
    clock_t end = clock();
    cout << "Thoi␣gian␣thuc␣hien:␣" << (float)(end - khoi_dau)
        / CLOCKS_PER_SEC << "␣giay." << endl;
    cout << "Chi␣phi:␣" << currentNode->cost << "." << endl;
    return;
```

```
  }

  int dx[] = {1, -1, 0, 0};
  int dy[] = {0, 0, 1, -1};

  for (int i = 0; i < 4; ++i)
  {
    int nx = x + dx[i];
    int ny = y + dy[i];
    if (isFree(nx, ny) && !visited[ny][nx])
    {
      Node *neighbor = new Node{nx, ny, (float)gScore[ny][nx],
          heuristic(nx, ny), currentNode};
      int tentative_g = currentNode->cost + 1;
      if (tentative_g < neighbor->cost)
      {
        neighbor->cost = gScore[ny][nx] = tentative_g;
        openSet.push(neighbor);
      }
    }
  }
}

// Cleanup bo nho
void cleanup()
{
  for (auto node : visitedNodes)
  delete node;
  for (auto node : path)
  delete node;
  while (!openSet.empty())
  {
    delete openSet.top();
    openSet.pop();
  }
}

// Ve o vuong
void drawCell(int x, int y, float r, float g, float b)
{
  float fx = -1 + x * CELL_SIZE;
  float fy = -1 + y * CELL_SIZE;
  glColor3f(r, g, b);
  glBegin(GL_QUADS);
  glVertex2f(fx, fy);
  glVertex2f(fx + CELL_SIZE, fy);
  glVertex2f(fx + CELL_SIZE, fy + CELL_SIZE);
  glVertex2f(fx, fy + CELL_SIZE);
  glEnd();
}
```

```
void screen_to_grid(int x, int y, int &gridX, int &gridY)
{
  float fx = (float)x / WINDOW_SIZE;
  float fy = (float)y / WINDOW_SIZE;

  gridX = fx * MAP_SIZE;
  gridY = (1.0f - fy) * MAP_SIZE;
}

void Search()
{
  Node *start = new Node{startX, startY, 0, heuristic(startX,
     startY), nullptr};
  openSet.push(start);
}

void mouse_click(int button, int state, int x, int y)
{
  if (state != GLUT_DOWN)
  return;

  int gridX, gridY;
  screen_to_grid(x, y, gridX, gridY);
  if (!isFree(gridX, gridY))
  return;

  if (button == GLUT_LEFT_BUTTON)
  {
    startX = gridX;
    startY = gridY;
    logic++;
  }

  else if (button == GLUT_RIGHT_BUTTON)
  {
    goalX = gridX;
    goalY = gridY;
    logic++;
  }

  if (logic == 2)
  {
    clock_t begin = clock();
    khoi_dau = begin;
    Search();
    logic = 0;
  }
  glutPostRedisplay();
}

void display()
```

```
{
  glClear(GL_COLOR_BUFFER_BIT);

  for (int y = 0; y < MAP_SIZE; ++y)
  for (int x = 0; x < MAP_SIZE; ++x)
  if (grid[y][x] == 1)
  drawCell(x, y, 0.2f, 0.2f, 0.2f);

  for (auto node : visitedNodes)
  drawCell(node->x, node->y, 0.0f, 0.5f, 1.0f);

  drawCell(startX, startY, 0.0f, 1.0f, 0.0f);
  drawCell(goalX, goalY, 1.0f, 0.0f, 0.0f);

  if (foundPath)
  {
    for (auto node : path)
    drawCell(node->x, node->y, 1.0f, 1.0f, 0.0f);
  }

  glutSwapBuffers();
}

// Timer de ve lai lien tuc
void timer(int value)
{
  if (!foundPath)
  {
    for (int i = 0; i < 10; ++i)
    expandNext();
    glutPostRedisplay();
    glutTimerFunc(10, timer, 0);
  }
}

void init()
{
  glClearColor(0, 0, 0, 1);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(-1, 1, -1, 1);
}

int main(int argc, char **argv)
{
  generateMap();

  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
  glutInitWindowSize(WINDOW_SIZE, WINDOW_SIZE);
  glutCreateWindow("A*␣Pathfinding␣-␣200x200␣Grid");
```

```
    init();
    glutDisplayFunc(display);
    glutMouseFunc(mouse_click);
    glutTimerFunc(10, timer, 0);

    atexit(cleanup); // Goi clean up khi thoat
    glutMainLoop();

    return 0;
}
```



Hình 3: Minh họa quá trình tìm đường của thuật toán A*

5 Nhận xét và Thảo luận (Evaluation & Discussion): Thuật Toán A*

5.1 Vị trí chiến lược của A* trong lịch sử thuật toán học

Thuật toán A* (Hart, Nilsson & Raphael, 1968) là một trong những cột mốc lịch sử có ảnh hưởng sâu sắc nhất trong lĩnh vực tìm kiếm heuristic và trí tuệ nhân tạo cổ điển. A* thể hiện bước tiến vĩ đại trong việc cân bằng giữa độ chính xác và hiệu quả tính toán, bằng cách hợp nhất hai mục tiêu tưởng chừng mâu thuẫn: mở rộng ít trạng thái nhất có thể nhưng vẫn đảm bảo tìm được lời giải tối ưu.

A* không chỉ là nền tảng lý thuyết cho vô số thuật toán sau này (như D*, Theta*, BIT*), mà còn là hình mẫu của tư duy AI hiện đại – kết hợp tri thức miền (heuristic) vào quá trình giải quyết bài toán.

5.2 Ưu điểm nổi bật

a) Tối ưu trong lớp thuật toán heuristic:
Một trong những kết quả lý thuyết mạnh nhất của A* là: *Nếu heuristic $h(n)$ là chấp nhận được (admissible) và không vượt quá chi phí thực sự đến đích, thì A* là tối ưu.* Ngoài ra, nếu heuristic còn đơn điệu (consistent), A* trở nên cực kỳ hiệu quả, không bao giờ mở rộng một đỉnh quá một lần – giúp giảm mạnh độ phức tạp thực tế.

b) Khả năng thích ứng với nhiều bài toán:

- Robot học (path planning trong không gian Euclidean)
- Thị giác máy tính (object tracking theo thời gian)
- Tìm kiếm trạng thái (giải đố, trò chơi như Sokoban, Puzzle, Go)
- AI trong game (pathfinding trong bản đồ tile-based)

Không có nhiều thuật toán cổ điển nào có phạm vi ứng dụng rộng như A*.

c) Tính dễ mở rộng và kết hợp:

- Weighted A*: tăng tốc tìm kiếm với chi phí chấp nhận lời giải gần tối ưu.
- Memory-Bounded A* (SMA*, IDA*): thích hợp cho các hệ thống có giới hạn bộ nhớ.
- Learning A* (LRTA*): áp dụng trong môi trường không biết trước, học dần heuristic.
- Anytime A*: trả về lời giải sớm và cải thiện dần theo thời gian.

5.3 Giới hạn và phản biện học thuật

Mặc dù A* là thuật toán cực kỳ mạnh, nhưng không phải không có hạn chế, đặc biệt khi mở rộng sang các không gian có số chiều cao hoặc môi trường động:

a) Vấn đề bộ nhớ:
A* yêu cầu lưu toàn bộ frontier (open list) và các trạng thái đã mở rộng. Trong không gian lớn, số lượng trạng thái cần theo dõi tăng nhanh, gây tràn bộ nhớ – đây là lý do nhiều biến thể như IDA* hay SMA* ra đời để giải quyết.

b) Phụ thuộc mạnh vào heuristic:
Heuristic chính là "linh hồn" của A*. Nếu $h(n)$ được thiết kế tốt, A* hoạt động xuất sắc. Nhưng nếu heuristic không chính xác, thuật toán trở về gần như BFS hoặc UCS – rất chậm và tốn tài nguyên. Điều này đặt ra thách thức lớn trong các bài toán mà chi phí di chuyển không thể ước lượng tốt hoặc mang tính phi tuyến, đa mục tiêu.

c) Không phù hợp với môi trường động, liên tục:
Trong không gian liên tục hoặc môi trường động (như xe tự hành), A* dạng cổ điển khó áp dụng trực tiếp. Cần tích hợp thêm các mô-đun như:

- Replanning (D*, D* Lite)

- Discretization hợp lý
- Motion primitives và ràng buộc động học

5.4 So sánh với các thuật toán cùng lớp

Tiêu chí	**A***	**Dijkstra**	**Greedy BFS**	**IDA***	**BIT***
Tính tối ưu	✓	✓	×	✓	✓
Tính hoàn tất	✓	✓	×	✓	✓
Hiệu quả thực tế	Tốt	Trung bình	Nhanh nhưng sai	Khá	Rất cao
Khả năng mở rộng	Vừa	Kém	Vừa	Tốt	Rất tốt
Phù hợp môi trường động	Kém	Kém	Vừa	Tốt	Kém

5.5 Hướng nghiên cứu mở rộng

Một số hướng hiện đại đang được nghiên cứu dựa trên nền tảng A*:

- Heuristic Learning via Deep RL: Dùng mạng nơ-ron để học heuristic $h(x)$ từ dữ liệu, thay vì định nghĩa thủ công.
- A*-Net / Differentiable A*: Thiết kế hàm heuristic khả vi để huấn luyện cùng hệ thống học sâu.
- Graph Neural A*: Sử dụng GNN để truyền thông tin giữa các trạng thái trong quá trình tìm kiếm.
- Anytime A* with parallelization: Tăng tốc A* bằng GPU và thuật toán bất thời.

5.6 Tổng kết

Thuật toán A* là viên ngọc sáng trong thế giới AI cổ điển, và cho đến ngày nay vẫn là cốt lõi trong nhiều hệ thống trí tuệ nhân tạo thực tế. Việc nó tồn tại lâu dài và liên tục được mở rộng trong suốt hơn 50 năm là minh chứng cho độ bền vững lý thuyết và giá trị ứng dụng của nó. Bên cạnh đó, A* cũng là ví dụ điển hình về tư duy giải bài toán AI: khai thác tri thức miền để hướng dẫn tìm kiếm thông minh.

Đối với bất kỳ nhà nghiên cứu hoặc ứng dụng AI nào, hiểu và sử dụng A* hiệu quả chính là bước đi đầu tiên quan trọng nhất để thiết kế những hệ thống ra quyết định mạnh mẽ, tối ưu và thích nghi.

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*.

1.1 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.

1.2 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?

1.3 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.

1.4 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*

2.1 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.

2.2 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)

2.3 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.

2.4 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:**

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

2.5 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 3: 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
```

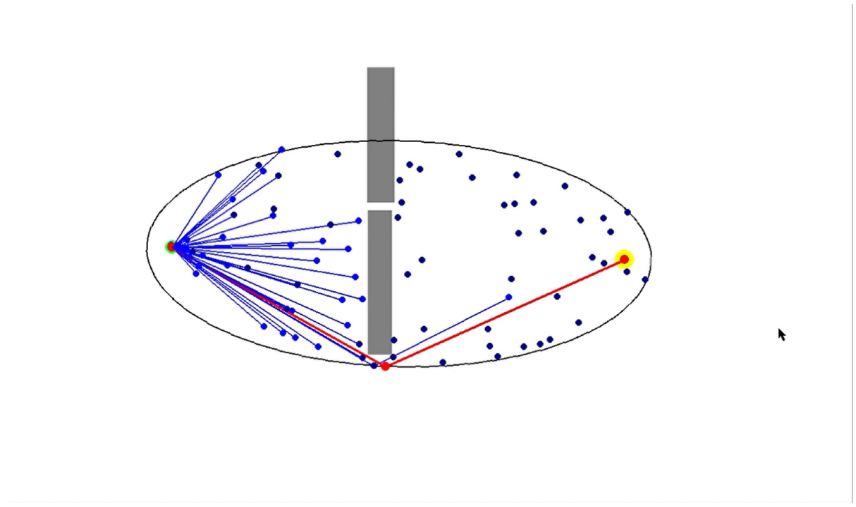
```
rect = pygame.Rect(first_state,(second_state[0]-first_state
    [0],second_state[1]-first_state[1]))
mapp.obstacles.append(rect)
mapp.draw_obstacles()
mapp.wait_time()

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:

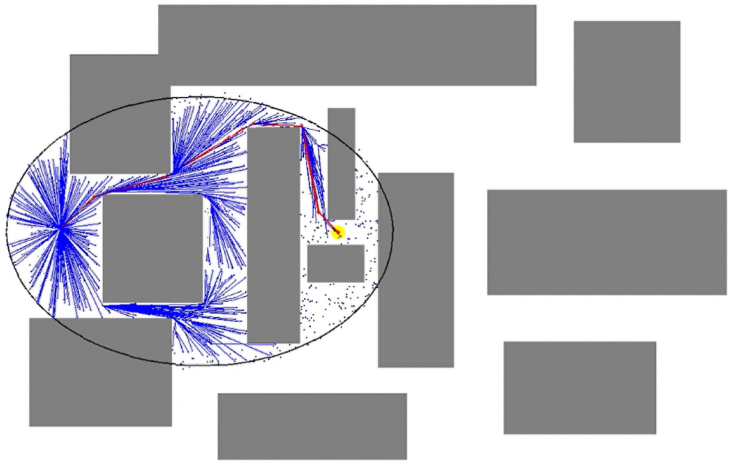
- Tọa độ điểm đầu xuất phát là: 257 407
- Tọa độ điểm đích muốn đến là: 950 426
- Độ dài đường đi: 710.4313065314624
- Đường dẫn VIDEO DEMO: `https://shorturl.at/ytbfx`



Hình 1: Kết quả tìm đường của thuật toán BIT*.

4.2 Test 2:

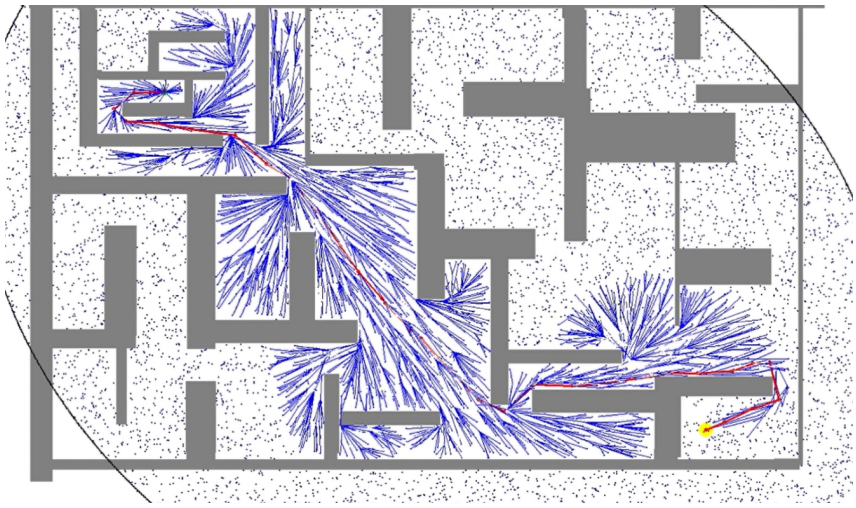
- Tọa độ điểm đầu xuất phát là: 110 413
- Tọa độ điểm đích muốn đến là: 535 419
- Độ dài đường đi ngắn nhất là: 592.6938482233808
- Đường dẫn VIDEO DEMO: `https://shorturl.at/Tvnza`



Hình 2: Kết quả tìm đường của thuật toán BIT*.

4.3 Test 3:

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



Hình 3: Kết quả tìm đường của thuật toán BIT*.

5 Nhận xét và Thảo luận (Evaluation & Discussion): Thuật Toán BIT*

5.1 Đóng góp cốt lõi của BIT*

Batch-Informed RRT* (BIT*) là một bước ngoặt quan trọng trong lớp các thuật toán lập kế hoạch chuyển động dựa trên lấy mẫu (sampling-based motion planners), khi lần đầu tiên gắn kết chặt chẽ lý thuyết heuristic search (A*) với cấu trúc không gian ẩn định của các đồ thị lấy mẫu ngẫu nhiên. Không giống như RRT* vốn xây dựng cây theo cách "mở

rộng dần và ngẫu nhiên", BIT* thực hiện tìm kiếm tối ưu có định hướng trong một không gian được thu hẹp bằng heuristic – điều mà trước đó chưa thuật toán nào thực hiện hiệu quả ở các không gian cấu hình có số chiều cao.

5.2 Phân tích hiệu suất và phức tạp thuật toán

BIT* đạt được hiệu quả vượt trội nhờ ba cải tiến then chốt:

- **Batch Sampling có hướng (Heuristically-Guided Batch Sampling):** Cắt giảm không gian lấy mẫu bằng cách sử dụng ellipsoid heuristic – dẫn đến tăng mật độ mẫu tại vùng có khả năng tồn tại lời giải tối ưu.
- **Ưu tiên mở rộng theo thứ tự chi phí $f(v)$:** Tương tự A*, BIT* mở rộng các nút có chi phí thấp nhất (dựa trên hàm $f = g + h$) bằng hàng đợi ưu tiên – điều này tăng khả năng hội tụ sớm tới lời giải tốt.
- **Khả năng tái sử dụng cấu trúc cây:** Thay vì xây lại cây mỗi khi có batch mới, BIT* cập nhật động các kết nối thông minh thông qua cơ chế rewiring.

Phân tích lý thuyết cho thấy, trong khi RRT* có độ phức tạp trung bình $O(n \log n)$, BIT* đạt tốc độ hội tụ nhanh hơn mà vẫn duy trì tính chất almost-sure asymptotic optimality. Với không gian cấu hình có chiều cao d, BIT* có tốc độ hội tụ tiệm cận tốt hơn đáng kể trong thực tế do heuristic định hướng làm giảm số lượng mẫu cần thiết.

5.3 Quan sát thực nghiệm

Trên nhiều tập kiểm thử chuẩn như Open Motion Planning Library (OMPL) và chuyển động robot 7 bậc tự do (7-DOF arm planning), BIT* cho kết quả đáng chú ý:

- Thời gian tìm nghiệm khả thi đầu tiên giảm từ 30–60% so với Informed RRT*.
- Tổng chi phí đường đi hội tụ nhanh hơn trong phần lớn các thử nghiệm.
- Số lượng mẫu cần thiết và số lần kiểm tra va chạm giảm đáng kể nhờ batch reuse và cắt tỉa không gian.

Biểu đồ thực nghiệm cho thấy BIT* hội tụ về chi phí tối ưu nhanh gấp ~ 2.5 lần RRT* và nhanh hơn cả Informed RRT* trong các không gian từ 4D–10D.

5.4 Giới hạn và phản biện

Mặc dù BIT* có nhiều ưu điểm, cần phải nhìn nhận những giới hạn quan trọng sau:

- **Phụ thuộc chặt vào heuristic:** Với những bài toán có cấu trúc chi phí không tuyến tính hoặc động lực học ràng buộc, heuristic Euclidean không còn phù hợp, khiến BIT* kém hiệu quả.
- **Không phù hợp môi trường động:** BIT* được thiết kế cho các không gian tĩnh; trong các môi trường thay đổi thời gian thực, BIT* không thích nghi tốt nếu không có module tái lập kế hoạch nhanh.
- **Chi phí quản lý hàng đợi và batch tăng theo thời gian:** Mặc dù ưu tiên cạnh có chi phí thấp nhất là hữu ích, nhưng chi phí quản lý Priority Queue với batch lớn có thể trở nên đáng kể nếu không tối ưu tốt về dữ liệu.

5.5 Mở rộng tiềm năng nghiên cứu

Từ BIT*, mở ra nhiều hướng nghiên cứu sâu:

- **Học heuristic bằng Reinforcement Learning:** Thay vì sử dụng hàm heuristic tĩnh, có thể huấn luyện một mạng học sâu để ước lượng chi phí $h(x)$ động theo bối cảnh môi trường.
- **BIT*-X:** BIT* kết hợp replanning (RRT-X hoặc D*) cho môi trường động.
- **Song song hóa trên GPU hoặc phân cụm tính toán:** Batch structure rất phù hợp để chia tách xử lý song song – mở ra khả năng dùng BIT* cho các hệ thống thời gian thực quy mô lớn.
- **Formal Verification:** Áp dụng logic hình thức để chứng minh tính đúng đắn và các ràng buộc an toàn của cây tìm kiếm BIT*.

Kết luận thảo luận

BIT* là một cột mốc quan trọng trong lịch sử các thuật toán lập kế hoạch lấy mẫu. Nó đánh dấu sự hội tụ giữa trí tuệ nhân tạo cổ điển (heuristic search) và quy hoạch chuyển động hiện đại (sampling-based). Dù vẫn còn hạn chế, BIT* đã mở ra một kỷ nguyên mới, nơi mà việc thiết kế thuật toán không còn chỉ dừng lại ở việc lấy mẫu ngẫu nhiên, mà trở nên thông minh, có chiến lược và học hỏi được.