Lab 2: Thực hành thuật toán A*

ThS. Lê Thị Thùy Trang

2024-12-26

Bài thực hành này hướng dẫn triển khai thuật toán tìm kiếm A* bằng ngôn ngữ Python để giải quyết một bài toán thực tế. Thuật toán A* là một thuật toán tìm kiếm có thông tin, nghĩa là nó tận dụng một hàm heuristic để hướng dẫn tìm kiếm của mình đến mục tiêu. Hàm heuristic này ước tính chi phí để đạt được mục tiêu từ một nút nhất định, cho phép thuật toán ưu tiên các đường dẫn đầy hứa hẹn và tránh khám phá các đường dẫn không cần thiết.

1. Cách thuật toán A* hoạt động

Hãy tưởng tượng bạn đang cố gắng tìm tuyến đường ngắn nhất giữa hai thành phố trên bản đồ. Trong khi thuật toán Dijkstra sẽ khám phá theo mọi hướng và Best-First Search có thể hướng thẳng đến đích (có khả năng bỏ qua các lối tắt), A* thực hiện một chiến lược thông minh hơn. Nó xem xét cả hai:

- Khoảng cách đã đi từ điểm bắt đầu
- Ước tính thông minh về khoảng cách còn lại đến đích

Sự kết hợp này giúp A* đưa ra quyết định sáng suốt về con đường nào để khám phá tiếp theo, giúp nó vừa hiệu quả vừa chính xác.

Các thành phần chính của A*:

- Các nút: Các điểm trong đồ thị (các giao điểm trên bản đồ)
- Edges: Các kết nối giữa các nút (con đường kết nối các giao điểm)
- Chi phí đường đi: Chi phí để di chuyển từ nút này đến nút khác
- Heuristic: Chi phí ước tính từ bất kỳ nút nào đến mục tiêu
- Không gian tìm kiếm: Tập hợp tất cả các đường đi có thể khám phá.

2. Hàm chi phí của A*

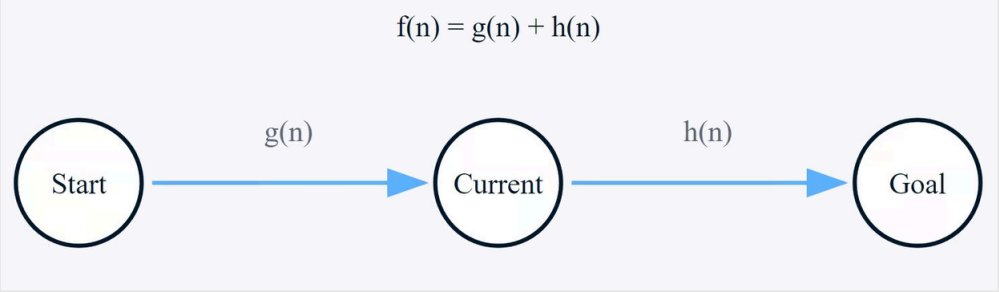

Figure 1: Hàm chi phí của thuật toán A*

Hiệu quả của thuật toán A* đến từ việc đánh giá thông minh các đường đi bằng ba thành phần chính: $g(n), h(n)$ và $f(n)$. Các thành phần này hoạt động cùng nhau để hướng dẫn quá trình tìm kiếm đến các đường đi hứa hẹn nhất.

2.1 Hàm chi phí $g(n)$

Hàm chi phí $g(n)$ biểu diễn khoảng cách chính xác, đã biết từ nút bắt đầu ban đầu đến vị trí hiện tại trong tìm kiếm của chúng ta. Không giống như các giá trị ước tính, chi phí này là chính xác và được tính bằng cách cộng tất cả các trọng số cạnh riêng lẻ đã đi qua dọc theo đường dẫn đã chọn.

Về mặt toán học, đối với một đường dẫn qua các nút n_O(nút bắt đầu) đến n_k (nút hiện tại), chúng ta có thể biểu diễn $g(n)$ như sau:

$$g(n_k) = \sum_{i=0}^{k-1} w(n_i, n_{i+1})$$

- Trong đó $w(n_i, n_{i+1})$ là trọng số của cạnh kết nối từ nút n_i đến nút n_{i+1}

Khi di chuyển qua đồ thị, giá trị này sẽ tích lũy, cung cấp thước đo rõ ràng về các nguồn lực thực tế (có thể là khoảng cách, thời gian hoặc bất kỳ số liệu nào khác) mà ta đã sử dụng để đạt được vị trí hiện tại.

2.2 Hàm Heuristic $h(n)$

Hàm heuristic $h(n)$ cung cấp chi phí ước tính từ nút hiện tại đến nút đích, đóng vai trò "phỏng đoán có thông tin" của thuật toán về đường đi còn lại.

Về mặt toán học, đối với bất kỳ nút n nào, ước tính heuristic phải thoả mãn điều kiện $h(n) <= h^*(n)$, trong đó $h^*(n)$ là chi phí hiện tại để đi đến đích, tức là chi phí ước tính không bao giờ được cao hơn chi phí hiện tại.

Trong các bài toán dạng lưới hoặc dạng đồ thị, hàm heuristic phổ biến là khoảng cách Manhattan và khoảng cách Euclid. Đối với nút hiện tại có toạ độ (x_1, y_1) và nút đích có toạ độ (x_2, y_2), các khoảng cách này được tính bằng công thức:

- Khoảng cách Manhattan

$$h(n) = |x_1 - x_2| + |y_1 - y_2|$$

- Khoảng cách Euclid

$$h(n) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

2.3 Tổng chi phí ước tính $f(n)$

Tổng chi phí ước tính $f(n)$ là nền tảng của quá trình ra quyết định của thuật toán A*, kết hợp cả chi phí đường đi thực tế và ước tính heuristic để đánh giá tiềm năng của từng nút, đối với bất kỳ nút n nào, chi phí này được tính như sau:

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

Trong đó:

- $g(n)$ là chi phí thực tế từ điểm bắt đầu đến nút hiện tại
- $h(n)$ là chi phí ước tính từ nút hiện tại đến nút đích

Thuật toán luôn chọn nút có giá trị $f(n)$ thấp nhất từ danh sách mở, do đó đảm bảo sự cân bằng tối ưu giữa chi phí đã biết và khoảng cách ước tính còn lại.

3. Quản lý danh sách nút

Thuật toán A* duy trì hai loại danh sách thiết yếu.

Danh sách mở:

- Chứa các nút cần được đánh giá
- Sắp xếp theo giá trị $f(n)$ (thấp nhất trước)
- Các nút mới được thêm vào khi chúng được phát hiện

Danh sách đóng:

- Chứa các nút đã được đánh giá
- Giúp tránh việc đánh giá lại các nút
- Được sử dụng để tái tạo lại đường đi cuối cùng

Thuật toán liên tục chọn nút có giá trị $f(n)$ thấp nhất từ danh sách mở, đánh giá nút đó và di chuyển nút đó đến danh sách đóng cho đến khi nút đó đạt đến mục tiêu hoặc xác định không có đường đi nào tồn tại.

4. Triển khai thuật toán A* bằng python

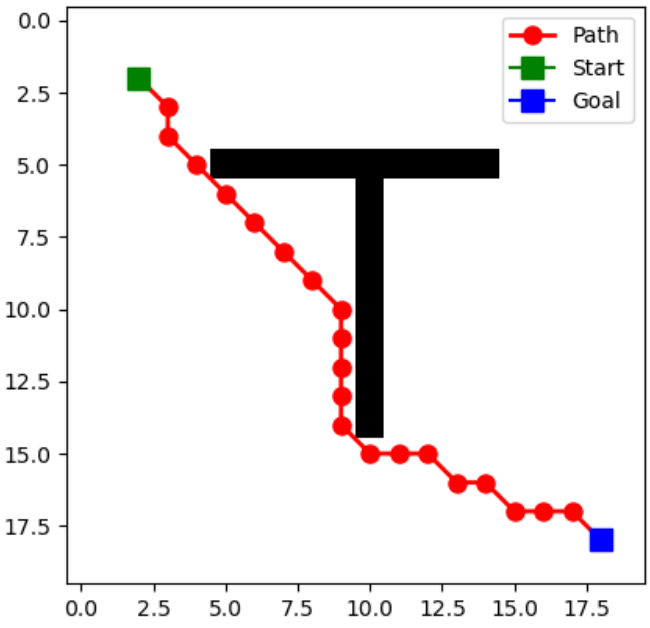

Figure 2: Đường đi ngắn nhất của robot

Dưới đây là một chương trình áp dụng thuật toán A* để điều khiển robot di chuyển trong một kho hàng tự động. Kho hàng được biểu diễn dưới dạng lưới $20 * 20$ với tất cả các ô ban đầu đều là ô trống và có 1 tường ngang và 1 tường dọc, được biểu diễn là:

- 0: nếu ô trống, robot có thể di chuyển qua
- 1: nếu ô chứa vật cản, robot không thể đi qua.

Robot cần di chuyển từ vị trí bắt đầu đến vị trí đích theo đường đi ngắn nhất có thể.

Chương trình đã thực hiện các yêu cầu của bài toán, theo các bước

4.1 Import the necessary libraries

```
from typing import List, Tuple, Dict, Set
import numpy as np
import heapq
from math import sqrt
import matplotlib.pyplot as plt
```

4.2 Helper functions

Đầu tiên, ta tạo một cấu trúc nút sẽ lưu trữ thông tin vị trí và tìm đường cho mỗi điểm trong không gian tìm kiếm.

Sau đó, ta triển khai một hàm để tính khoảng cách giữa các điểm bằng khoảng cách Euclidean.

Tiếp theo, ta cần một hàm để tìm các vị trí lân cận hợp lệ trong lưới, kiểm tra cẩn thận các ranh giới và chướng ngại vật.

Cuối cùng, ta sử dụng một hàm để tái tạo đường đi sau khi đã tìm thấy mục tiêu.

```
def create_node(position: Tuple[int, int], g: float = float('inf'),
                h: float = 0.0, parent: Dict = None) -> Dict:
    """
    Create a node for the A* algorithm.

    Args:
        position: (x, y) coordinates of the node
        g: Cost from start to this node (default: infinity)
        h: Estimated cost from this node to goal (default: 0)
        parent: Parent node (default: None)

    Returns:
        Dictionary containing node information
    """
    return {
        'position': position,
        'g': g,
        'h': h,
        'f': g + h,
        'parent': parent
    }
```

```
def calculate_heuristic(pos1: Tuple[int, int], pos2: Tuple[int, int]) -> float:
    """
    Calculate the estimated distance between two points using Euclidean distance.
```

```
    """
    x1, y1 = pos1
    x2, y2 = pos2
    return sqrt((x2 - x1)**2 + (y2 - y1)**2)
```

```
def get_valid_neighbors(grid: np.ndarray, position: Tuple[int, int]) -> List[Tuple[int, int]]:
    """
    Get all valid neighboring positions in the grid.

    Args:
        grid: 2D numpy array where 0 represents walkable cells and 1 represents obstacles
        position: Current position (x, y)

    Returns:
        List of valid neighboring positions
    """
    x, y = position
    rows, cols = grid.shape

    # All possible moves (including diagonals)
    possible_moves = [
        (x+1, y), (x-1, y),    # Right, Left
        (x, y+1), (x, y-1),    # Up, Down
        (x+1, y+1), (x-1, y-1),  # Diagonal moves
        (x+1, y-1), (x-1, y+1)
    ]

    return [
        (nx, ny) for nx, ny in possible_moves
        if 0 <= nx < rows and 0 <= ny < cols  # Within grid bounds
        and grid[nx, ny] == 0                # Not an obstacle
    ]
```

```
def reconstruct_path(goal_node: Dict) -> List[Tuple[int, int]]:
    """
    Reconstruct the path from goal to start by following parent pointers.
    """
    path = []
    current = goal_node

    while current is not None:
        path.append(current['position'])
        current = current['parent']

    return path[::-1]  # Reverse to get path from start to goal
```

4.3 Main A* algorithm implementation

Để triển khai thuật toán, ta sử dụng hàng đợi ưu tiên nhằm đảm bảo rằng chương trình luôn khám phá các đường đi có nhiều hứa hẹn trước tiên.

Chương trình sẽ duy trì hai tập hợp: một tập hợp mở cho các nút mà ta cần kiểm tra và một tập hợp đóng cho các nút đã được duyệt.

Chi phí đường đi được cập nhật bất cứ khi nào ta tìm thấy đường đi ngắn hơn cho đến khi đạt được mục tiêu.

```
def find_path(grid: np.ndarray, start: Tuple[int, int],
              goal: Tuple[int, int]) -> List[Tuple[int, int]]:
    """
    Find the optimal path using A* algorithm.

    Args:
        grid: 2D numpy array (0 = free space, 1 = obstacle)
        start: Starting position (x, y)
        goal: Goal position (x, y)

    Returns:
        List of positions representing the optimal path
    """
    # Initialize start node
    start_node = create_node(
        position=start,
        g=0,
        h=calculate_heuristic(start, goal)
    )

    # Initialize open and closed sets
    open_list = [(start_node['f'], start)]  # Priority queue
    open_dict = {start: start_node}         # For quick node lookup
    closed_set = set()                      # Explored nodes

    while open_list:
        # Get node with lowest f value
        _, current_pos = heapq.heappop(open_list)
        current_node = open_dict[current_pos]

        # Check if we've reached the goal
        if current_pos == goal:
            return reconstruct_path(current_node)

        closed_set.add(current_pos)

        # Explore neighbors
        for neighbor_pos in get_valid_neighbors(grid, current_pos):
            # Skip if already explored
            if neighbor_pos in closed_set:
                continue

            # Calculate new path cost
            tentative_g = current_node['g'] + calculate_heuristic(current_pos, neighbor_pos)

            # Create or update neighbor
            if neighbor_pos not in open_dict:
                neighbor = create_node(
                    position=neighbor_pos,
                    g=tentative_g,
                    h=calculate_heuristic(neighbor_pos, goal),
                    parent=current_node
```

```
                )
                heapq.heappush(open_list, (neighbor['f'], neighbor_pos))
                open_dict[neighbor_pos] = neighbor
            elif tentative_g < open_dict[neighbor_pos]['g']:
                # Found a better path to the neighbor
                neighbor = open_dict[neighbor_pos]
                neighbor['g'] = tentative_g
                neighbor['f'] = tentative_g + neighbor['h']
                neighbor['parent'] = current_node

    return []  # No path found
```

4.4 Visualization

Hàm này trực quan hoá bố cục lưới với các chướng ngại vật và vẽ đường đi tối ưu đã tính toán từ điểm xuất phát đến đích. Hình 2 chính là kết quả của chương trình với hàm này.

```
def visualize_path(grid: np.ndarray, path: List[Tuple[int, int]]) -> None:
    """
    Visualize the grid with the path marked by '*'.

    Args:
        grid: 2D numpy array representing the warehouse
        path: List of positions representing the path
    """
    # Create a copy of the grid to avoid modifying the original
    grid_copy = np.copy(grid)

    # Mark the path on the grid
    for (x, y) in path:
        grid_copy[x][y] = 8  # Use 8 to represent the path

    # Print the grid
    for row in grid_copy:
        print(' '.join(['*' if cell == 8 else str(cell) for cell in row]))

def plot_grid(grid: np.ndarray, path: List[Tuple[int, int]]) -> None:
    """
    Plot the grid and the path using matplotlib.

    Args:
        grid: 2D numpy array representing the warehouse
        path: List of positions representing the path
    """
    # Create a figure and axis
    fig, ax = plt.subplots()

    # Plot the grid
    ax.imshow(grid, cmap='Greys', interpolation='none')

    # Plot the path
    if path:
        path_x = [p[1] for p in path]  # Column (y)
```

```
        path_y = [p[0] for p in path]  # Row (x)
        ax.plot(path_x, path_y, color='red', marker='o', markersize=8, linewidth=2, label='Path')

    # Mark start and goal positions
    start = path[0] if path else None
    goal = path[-1] if path else None
    if start:
        ax.plot(start[1], start[0], color='green', marker='s', markersize=10, label='Start')
    if goal:
        ax.plot(goal[1], goal[0], color='blue', marker='s', markersize=10, label='Goal')

    # Add legend
    ax.legend()

    # Show the plot
    plt.show()
```

4.5 Main function

```
def main():
    grid = np.zeros((20, 20))  # 20x20 grid, all free space initially
    # Add some obstacles
    grid[5:15, 10] = 1  # Vertical wall
    grid[5, 5:15] = 1   # Horizontal wall
    # Define start and goal positions
    start_pos = (2, 2)
    goal_pos = (18, 18)

    # Find the path
    path = find_path(grid, start_pos, goal_pos)

    if path:
        print(f"Path found with {len(path)} steps!")
        visualize_path(grid, path)
        plot_grid(grid, path)
    else:
        print("No path found!")

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

5. Thực hành

5.1 Chuẩn bị môi trường thực hành

Để thực hiện bài thực hành này, ta sử dụng một số thư viện như `typing`, `numpy`, `heapq`, `math`, `matplolib`, trong đó `typing`, `math`, `heapq` là thư viện chuẩn của Python, không cần cài đặt. Thư viện `numpy` và `matplotlib` cần cài đặt qua `pip` để sử dụng.

Trước tiên, ta cần cài đặt `pip`. Đối với `Python 3.4++`, sử dụng `ensurepip` để cài đặt `pip`. Mở cmd rồi thực thi lệnh sau:

```
python -m ensurepip -upgrade
```

Sau khi hoàn tất, kiểm tra lại bằng lệnh:

```
python --version
```

Nếu thấy thông báo về phiên bản của `pip`, thì việc cài đặt đã thành công.

Sau đó, gõ lệnh sau để cài đặt thư viện thông qua `pip`:

```
pip install numpy matplotlib
```

5.2 Yêu cầu

Dựa trên code mẫu trên, phát triển một chương trình để điều khiển robot di chuyển trong một kho hàng tự động. Kho hàng được biểu diễn dưới dạng một lưới 2D, trong đó:

- 0: ô trống, robot có thể di chuyển qua với chi phí = 1.
- 1: ô chứa vật cản, robot không thể đi qua.
- 2: ô bùn lầy, robot có thể đi qua nhưng với chi phí = 3.
- 3: ô đá, robot có thể đi qua nhưng với chi phí = 5.

Lưu ý: Robot chỉ có thể di chuyển theo 4 hướng (lên, xuống, trái, phải), không di chuyển chéo.

Robot cần di chuyển từ vị trí bắt đầu (start) đến vị trí đích (goal) một cách tối ưu nhất, tức là đi theo đường có tổng chi phí thấp nhất có thể.

Yêu cầu: Sửa đổi code mẫu để tìm đường đi của robot trong trường hợp này