深入理解并实现基本的 A*寻路算法

杨其臻

Aug 16, 2025

寻路算法在多个领域具有广泛应用，包括游戏开发、机器人导航和物流路径规划等。在这些场景中，算法需要高效地找到从起点到终点的最优路径。传统的算法如 Dijkstra 或广度优先搜索（BFS）虽能保证最优性，但效率较低，尤其在大型地图中；贪心算法虽快，却无法保证最优解。A* 算法应运而生，其核心思想是「启发式引导的代价优先搜索」，通过平衡效率与最优性，成为寻路问题的首选方案。这一平衡源于算法对实际移动成本和启发式预估的智能结合，确保在多数情况下快速找到最短路径。

1　A* 算法的核心概念

A* 算法依赖三种关键数据结构：开放列表用于存储待探索节点，通常实现为优先队列以高效提取最小代价节点；封闭列表记录已探索节点，避免重复计算；节点对象包含属性如 g 值（从起点到当前节点的实际代价）、h 值（启发式预估的当前节点到终点代价），以及 f 值（总代价，计算公式为 $f(n) = g(n) + h(n)$，其中 n 代表节点）。

代价函数是算法的核心。$g(n)$ 表示实际移动成本，例如在网格地图中，直线移动代价为 1，对角线移动为 $\sqrt{2}$。启发函数 $h(n)$ 是算法的「智能之源」，其设计需遵循两个原则：可接受性要求 $h(n)$ 永远不高估真实代价（即 $h(n) \leq h^*(n)$，其中 $h^*(n)$ 是实际代价，确保最优性）；一致性则需满足三角不等式（即 $h(n) \leq c(n, m) + h(m)$，其中 $c(n, m)$ 是从 n 到 m 的代价，提升效率）。常用启发函数包括曼哈顿距离（适用于 4 方向移动，公式为 $|dx| + |dy|$）、欧几里得距离（任意方向移动，公式为 $\sqrt{dx^2 + dy^2}$），以及对角线距离（8 方向移动，优化了对角线路径计算）。

2　A* 算法流程详解

A* 算法流程通过伪代码清晰展示。以下是关键步骤：

```
初始化 OpenList 和 ClosedList
将起点加入 OpenList
while OpenList 非空 :
    取出 f 值最小的节点 N
    将 N 加入 ClosedList
    if N 是终点 :
        回溯路径，算法结束
    遍历 N 的每个邻居 M:
        if M 在 ClosedList 中 或 不可通行 : 跳过
```

```
      计算 new_g = g(N) + cost(N → M)
      if M 不在 OpenList 或 new_g < 当前 g(M):
         更新 M 的 g, h, f 值
         记录 M 的父节点为 N
         if M 不在 OpenList: 加入 OpenList
路径不存在
```

这段伪代码定义了算法骨架。首先初始化数据结构并将起点加入开放列表。循环中，每次从开放列表提取 f 值最小节点（即当前最优候选），若该节点是终点，则通过回溯父节点生成路径。否则，遍历其邻居节点：跳过已关闭或障碍节点；计算新 g 值（实际代价），如果更优则更新节点属性并加入开放列表。分步图解虽有助于理解，但通过文字描述，算法核心在于「节点展开」阶段（评估邻居）、「代价更新」阶段（优化路径），以及「路径回溯」阶段（从终点反向追踪父节点）。例如，在网格图中，算法优先探索启发式引导的方向，避免无效搜索。

3 代码实现（Python 示例）

以下 Python 实现基于网格地图（二维数组），使用 heapq 模块优化开放列表管理。首先定义节点类：

```
# 定义节点类，存储位置、代价和父节点信息
class Node:
   def __init__(self, x, y):
      self.x, self.y = x, y # 节点坐标
      self.g = float('inf') # 起点到当前代价，初始无穷大
      self.h = 0 # 启发式预估代价
      self.f = float('inf') # 总代价 f = g + h
      self.parent = None # 父节点指针，用于回溯路径
```

节点类封装了关键属性：坐标 (x, y)、g 值（初始设为无穷大，表示未探索）、h 值（启发式预估）、f 值（总和），以及 parent（指向父节点，便于路径回溯）。初始化时 g 和 f 设为无穷大，确保算法能正确更新首次访问的节点。

启发函数使用曼哈顿距离，适合 4 方向移动：

```
# 曼哈顿距离启发函数，计算两点间预估代价
def heuristic(a, b):
   return abs(a.x - b.x) + abs(a.y - b.y)
```

该函数简单高效，输入两个节点对象，输出其 x 和 y 坐标差的绝对值之和。曼哈顿距离满足可接受性（不高估真实代价），且计算复杂度低，适合基础实现。

A* 主算法实现如下：

```
import heapq

def a_star(grid, start, end):
   open_list = [] # 优先队列存储 (f 值 , 节点 ID, 节点对象)
```

```
    closed_set = set() # 集合记录已关闭节点坐标
    start_node = Node(*start)
    end_node = Node(*end)
    start_node.g = 0 # 起点 g 值为 0
    start_node.h = heuristic(start_node, end_node)
    start_node.f = start_node.g + start_node.h
    heapq.heappush(open_list, (start_node.f, id(start_node), start_node))

    while open_list:
        current = heapq.heappop(open_list)[-1] # 提取 f 最小节点
        if (current.x, current.y) == (end_node.x, end_node.y):
            return reconstruct_path(current) # 到达终点，回溯路径
        closed_set.add((current.x, current.y))

        for dx, dy in [(0,1), (1,0), (0,-1), (-1,0)]: # 遍历 4 方向邻居
            nx, ny = current.x + dx, current.y + dy
            if not (0 <= nx < len(grid) and 0 <= ny < len(grid[0]) and grid[nx][ny] ==
                ↪ 0):
                continue # 跳过越界或障碍（grid 值非 0）
            if (nx, ny) in closed_set:
                continue # 跳过已关闭节点
            neighbor = Node(nx, ny)
            new_g = current.g + 1 # 假设移动代价为 1
            if new_g < neighbor.g: # 发现更优路径
                neighbor.g = new_g
                neighbor.h = heuristic(neighbor, end_node)
                neighbor.f = neighbor.g + neighbor.h
                neighbor.parent = current
                heapq.heappush(open_list, (neighbor.f, id(neighbor), neighbor))
    return None # 路径不存在
```

算法以网格地图（grid）、起点和终点坐标作为输入。初始化时，起点 g 值设为 0，计算 f 值后加入开放列表（使用 heapq 实现最小堆）。循环中，不断提取 f 最小节点；若为终点，则调用回溯函数；否则加入封闭集。遍历邻居时，检查边界和障碍（grid 值为 0 表示可行），若邻居未在开放列表或新 g 值更优，则更新属性并加入堆。id(neighbor) 用于唯一标识节点，避免堆比较错误。该实现高效处理了路径搜索的核心逻辑。

路径回溯函数如下：

```
def reconstruct_path(node):
    path = []
    while node:
        path.append((node.x, node.y)) # 添加当前节点坐标
```

```
        node = node.parent # 移至父节点
    return path[::-1] # 反转路径，从起点到终点
```

回溯函数从终点节点开始，沿 parent 指针向上遍历，存储每个节点坐标。最后反转列表，得到从起点到终点的有序路径。时间复杂度为 $O(k)$，其中 k 是路径长度，确保高效输出。

4 优化与变种

基础 A* 可通过优化提升性能。使用二叉堆（如 Python 的 heapq）管理开放列表，将节点插入和提取的时间复杂度降至 $O(\log n)$，远优于线性搜索。结合字典存储节点状态（如坐标到节点的映射），避免重复创建对象，减少内存开销。常见变种算法包括双向 A*，从起点和终点同时搜索，在中间相遇以加速；以及 Jump Point Search（JPS），通过跳过直线可达节点优化大尺度移动。路径平滑处理也很关键，例如剔除冗余拐点：用 Raycasting 检测直线可达性，移除不必要的中间节点，生成更自然的路径。这些优化在复杂场景中显著提升效率。

5 实战应用与注意事项

A* 在游戏开发中广泛应用，例如处理动态障碍物（通过定期重新规划路径）或不同地形代价（如沼泽移动代价高于道路）。实战中常见问题包括路径非最短（原因常为启发函数违反可接受性，需检查 $h(n)$ 是否高估）、算法卡死（确保终点可达并验证障碍物标记），或效率低下（可优化启发函数或采用 JPS）。调试时，记录节点扩展顺序和代价值，帮助定位逻辑错误。这些注意事项确保算法在真实环境中可靠运行。

A* 算法的核心优势在于灵活平衡效率与最优性。启发式引导减少了不必要的搜索，而可接受性保证了解的最优性，使其在静态地图中表现卓越。然而，其局限在于动态环境适应性弱，需结合 D* Lite 等算法处理实时变化。延伸学习方向包括进阶算法如 D* Lite（动态路径规划）或 Theta*（优化角度移动），以及三维空间寻路技术如 NavMesh。掌握 A* 为复杂寻路问题奠定了坚实基础。