A*算法研究与实践（C++实现）

潘韬睿 22180304

1 A*算法基本原理解析

1.1 A*算法介绍

1.1.1 算法原理

A*算法是一种常用的启发式路径搜索算法，它结合了广度优先搜索的完整性和深度优先搜索的效率，通过在搜索过程中引入启发函数来提高效率。A*使用估价函数 f(n)=g(n)+h(n)，其中：g(n)表示从起点到节点 n 的实际代价，被称作代价函数；h(n)表示从节点 n 到终点的启发式估计代价，被成为启发函数。

A*算法的目标是找到从起点到终点的最短路径，它在每次扩展节点时选择具有最小 f(n)值的节点来探索，以保证路径是最优的。

1.1.2 算法实现

A* 算法使用优先队列来保持待探索的节点。其主要步骤如下：

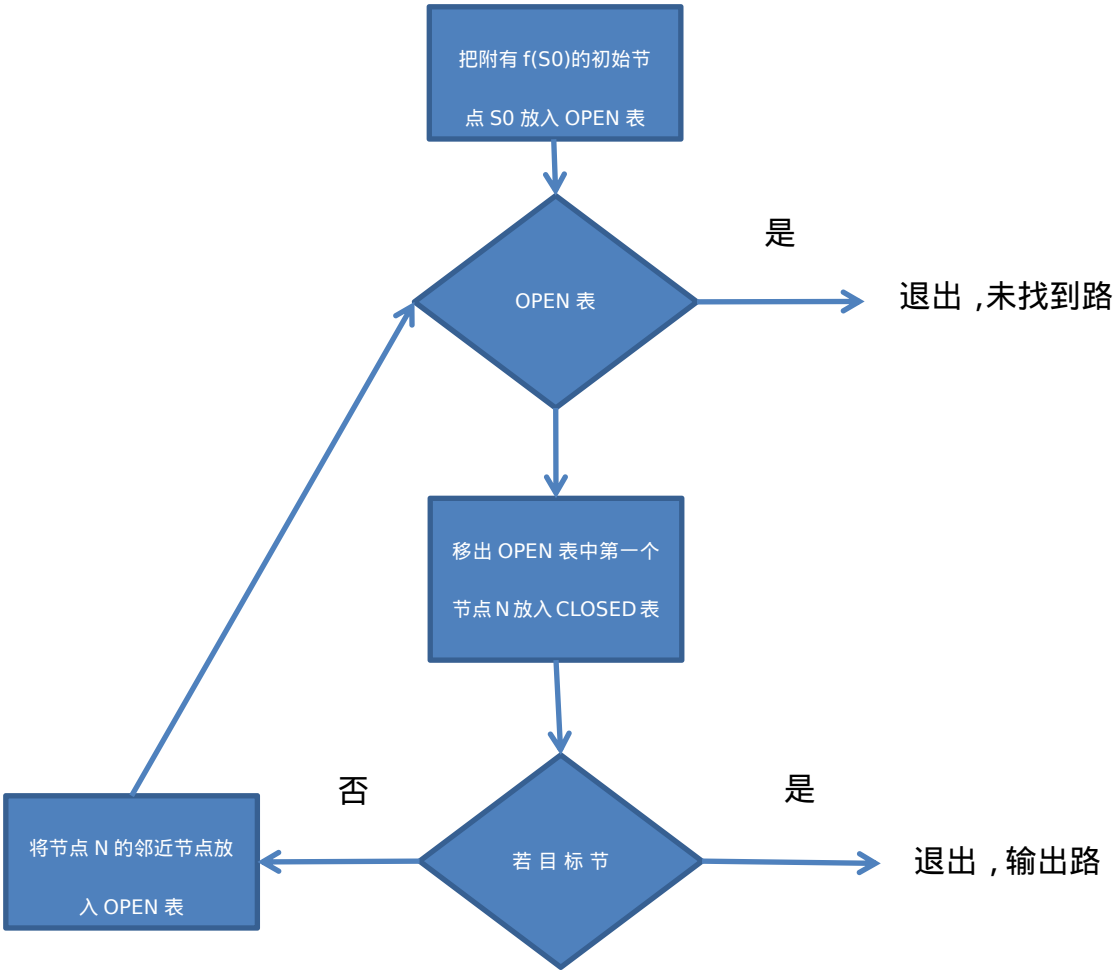
1.初始化：将起点添加到开放列表（优先队列），并设置初始估价 f(n)。

2.节点扩展：从开放列表中选取估价最小的节点进行扩展，并将其邻居加入开放列表 openlist。如果某个邻居节点的 g(n)值小于当前值，则更新其 g(n)值及父节点。

目标判断：当当前节点为终点时，停止搜索并回溯路径。

3.重复搜索：不断从开放列表中选取代价最小的节点，直到找到终点或无路径可达。

该算法的流程图如下：



1.2 启发函数和代价函数的作用

1.2.1 启发函数 h(n)

启发函数 h(n)是从当前节点 n 到目标节点的预估代价。常见的启发函数包括：

曼哈顿距离：用于网格中只允许水平和垂直移动的情况。

欧氏距离：用于网格中允许对角线移动且实际代价为路程的情况。

切比雪夫距离：用于网格中允许八个方向移动且实际代价为步数的情况。

启发函数决定了算法的效率和性能。如果启发函数低估了实际代价(即是“乐观的”) ，则 A* 算法的搜索路径接近最优，扩展节点较少。如果启发函数过于保守或不精确，搜索会变得接近于无启发式的广度优先搜索，扩展更多的节点。

可以说，启发函数的设计就是 A*算法的灵魂，只有启发函数的设计接近实际的需求，A*算法才可以给出最优的路线。1.2.3 会对该算法的最优性进行深入

讨论。

1.2.2 代价函数 g(n)

代价函数 g(n)是从起点到节点 n 的实际代价，即已经走过的路径长度。它对路径的准确性有决定性影响，较小的 g(n)值意味着当前路径的代价较低，因此节点会优先被扩展的该概率更大（还需要看启发函数的设计）。

在 A* 算法中，启发函数和代价函数共同作用，既保证了路径的准确性，又提高了搜索效率。

1.2.3 A*算法的最优性条件

在课上第一次了解 A*算法的时候，听说这是一种启发式的搜索策略，理所当然地认为该算法就像遗传算法和模拟退火算法无法保证一定找到全局最优解。但是随着学习实验的深入，我总结了 A*算法在最优性方面的相关特性。

在实际运用 A*的时候，通常需要将二维地图进行网格化，并且规定路线需要经过网格格点（或者网格中心），在这种限制下必然会出现精度的损失，导致规划出来的路线必然不是实际问题的最优解，网格越精细，精度越高。

而到 A*算法负责的路径规划阶段，A*算法是能够找到全局最优解，但有一个前提条件：启发式函数必须是可接受的（admissible）且一致的（consistent）。

可接受性意味着启发式函数从任意节点到目标节点的估计值不能超过实际最优路径的代价。这确保了 A*不会被错误的低估误导，从而确保找到最优解。

一致性（或称单调性）是指启发式函数满足三角不等式，即从一个节点直接到达目标的估计代价不应大于通过另一个中间节点再到达目标的代价总和。即对于任何两个相邻节点 n 和 m，启发函数应该满足：

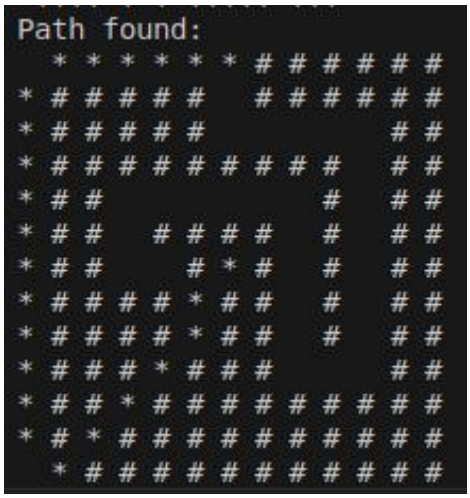
$$h(n) \leq c(n,m)+h(m)$$

其中 c(n,m)是从 n 到 m 的实际代价（允许斜着走，可以是对角线距离）。如果启发式函数满足这两个条件，A*算法可以保证找到从起点到目标的全局最优解。但如果启发式函数不满足这些条件，A*算法可能会找到次优解或更长的路径。

2 A*算法的 C++实现与代码解析

2.1 实验环境与背景介绍

完成在一个网格图上起点和终点之间的寻路操作 ,假设网格的边长为 1,起点和终点在网格的中心点，可以在相邻节点的中心点之间移动，并且支持斜着走，可以从一个节点向 8 个方向移动，启发函数用欧式距离来实现。



2.2 节点类 Node 的实现

2.2.1 节点坐标（x，y）

x 和 y 代表节点在网格中的坐标。A* 算法在二维网格上搜索路径，因此每个节点都有对应的 (x, y) 坐标。

2.2.2 gCost（从起点到该节点的实际代价）

gCost 是从起点到当前节点的实际代价。在 A* 中，gCost 代表了从起点出发经过的路径长度（包括可能的障碍物绕行）。

2.2.3 hCost（从该节点到终点的启发代价）

hCost 是启发函数估算的从当前节点到目标节点的代价。启发函数通常根据节点间的距离计算，例如欧氏距离、曼哈顿距离或切比雪夫距离。

2.2.4 fCost（总代价）

fCost 是节点的总代价，即 gCost 与 hCost 之和，用于评估节点优先级。A* 算法总是优先选择 fCost 最小的节点进行扩展，以保证找到最优路径。

2.2.5 父节点（parent）

parent 是指向当前节点的父节点的指针，记录了路径的来源。在 A* 完成搜索后，通过追溯父节点可以重建从起点到终点的路径。

2.2.6 构造函数

构造函数初始化了节点的坐标和父节点，并默认将 gCost, hCost, 和 fCost 初始化为 0。

2.2.7 calculateFCost()函数

calculateFCost() 用于计算节点的 fCost，即总代价 f(n)=g(n)+h(n)f(n) = g(n) + h(n)f(n)=g(n)+h(n)，这是 A* 算法中决定扩展顺序的核心值。每次更新 gCost 和 hCost 后，都需要调用此函数来更新 fCost。

```cpp
// 节点模块
struct Node
{
    int x, y;       // 横纵坐标
    double f, g, h; // g是代价函数，指的是已经走过的路程；h是启发函数，指该节点到终点的欧氏距离；f=g+h，决定优先队列的排序
    Node *parent;   // 父节点代表到达该节点的节点

    Node(int x, int y, double g, double h, Node *parent = nullptr) // 构造函数
        : x(x), y(y), g(g), h(h), f(g + h), parent(parent)
    {
    }
    // 记住只重载了<
    bool operator<(const Node &other) const
    {
        return f > other.f; // 优先队列中较小的f值具有更高的优先级
    }
};
```

2.3 AStar 函数的实现

2.3.1 函数输入

地图储存在二维数组中，用 vector 实现，并且需要输入起点和终点的坐标。

```
// A*算法实现，输入地图以及起点终点信息
void aStar(const vector<vector<int>> &grid, pair<int, int> start, pair<int, int> goal)
```

2.3.2 函数初始化

可以向四周共八个方向移动，并且定义一个存储节点的优先队列，按照估价函数 f 来排序。

```
// 定义四个方向：上、下、左、右，目前不能斜着走
vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}, {-1, -1}, {1, -1}, {1, 1}, {-1, 1}};

priority_queue<Node> openList;       // 存放节点的优先队列，按照f排序
unordered_map<int, Node *> allNodes; // 运用哈希表更好地管理节点

// 起点
Node *startNode = new Node(start.first, start.second, 0, euclideanDistance(start.first, start.second, goal.first, goal.second));
openList.push(*startNode);
allNodes[start.first * grid[0].size() + start.second] = startNode;

// 使用unordered_map来跟踪访问过的节点
unordered_map<int, bool> closedList;
```

2.3.3 当 OPEN 表不为空时的关键循环

定义 current 为 OPEN 表中 f 值最小的节点。

```
while (!openList.empty())
{
    Node current = openList.top();
    openList.pop();
```

找到的节点为目标节点则输出线路，退出循环。

```
// 如果找到目标节点，输出路径
if (current.x == goal.first && current.y == goal.second)
{
    cout << "*代表路线，#代表障碍物" << endl
         << "Path found: " << endl;
    printResult(allNodes[current.x * grid[0].size() + current.y], grid);
    cout << endl
         << "起点为(" << start.first << "," << start.second << ")" << "  终点为( " << goal.first << "," << goal.second << ")" << endl
    cout << "代价为: " << current.g << endl;
    return;
}
```

如果不为目标节点，就将邻居节点遍历，符合条件的放入 OPEN 表。

```
for (const auto &dir : directions)
{
    int newX = current.x + dir.first;
    int newY = current.y + dir.second;

    if (isInBounds(newX, newY, grid) && isWalkable(newX, newY, grid) &&
        !closedList[newX * grid[0].size() + newY])
    {

        double gNew = current.g + euclideanDistance(current.x, current.y, newX, newY);
        double hNew = euclideanDistance(newX, newY, goal.first, goal.second);

        int nodeKey = newX * grid[0].size() + newY;

        if (allNodes.find(nodeKey) == allNodes.end() || gNew < allNodes[nodeKey]->g)
        {
            Node *neighbor = new Node(newX, newY, gNew, hNew, allNodes[current.x * grid[0].size() + current.y]);
            openList.push(*neighbor);
            allNodes[nodeKey] = neighbor;
        }
    }
}
```

3 实验结果分析

3.1 测试用例设计

为了体现 A*算法可以在网格图中找到全局最优解的特性，我设计的几个测试用例尽量保证岔路的路程相近，甚至在路口处远路更像是最段路（满足启发函数更小的特点），以下是我的测试用例：

```
// 0表示通行，1表示障碍物
vector<vector<int>> grid0 = {
    {0, 0, 0, 0, 1},
    {0, 1, 0, 0, 1},
    {0, 1, 0, 0, 0},
    {0, 0, 0, 1, 0},
    {1, 1, 0, 0, 0}};
// 默认起点和终点
pair<int, int> start0 = {0, 0};
pair<int, int> goal0 = {4, 4};
```

```
vector<vector<int>> grid1 = {
    {0, 0, 0, 1, 0, 0},
    {0, 1, 0, 1, 0, 0},
    {0, 0, 0, 0, 1, 0},
    {1, 1, 1, 0, 1, 0},
    {0, 0, 0, 0, 0, 0},
    {0, 1, 1, 1, 1, 0}};

// 起点 (0, 0)，终点 (5, 5)
pair<int, int> start1 = {0, 0};
pair<int, int> goal1 = {5, 5};
```

```cpp
vector<vector<int>> grid2 = {
    {0, 0, 0, 0, 1, 0, 0, 0},
    {1, 1, 0, 1, 1, 1, 0, 0},
    {0, 0, 0, 0, 0, 1, 0, 1},
    {0, 1, 1, 1, 0, 0, 0, 0}};

// 起点 (3, 0)，终点 (0, 7)
pair<int, int> start2 = {3, 0};
pair<int, int> goal2 = {0, 7};
```

```cpp
vector<vector<int>> grid3 = {
    {0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0},
    {0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0},
    {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0},
    {1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0},
    {0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
    {0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1},
    {0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}};

// 起点 (0, 0)，终点 (12, 12)
pair<int, int> start3 = {0, 0};
pair<int, int> goal3 = {12, 12};
```

```cpp
vector<vector<int>> grid4 = {
    {0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1},
    {0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1},
    {0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1},
    {0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1},
    {0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1},
    {0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1},
    {0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1},
    {0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1},
    {0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1},
    {0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1},
    {0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1},
    {0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
    {0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}};

// 起点 (0, 0)，终点 (12, 12)
pair<int, int> start4 = {0, 6};
pair<int, int> goal4 = {6, 6};
```

3.2 结果分析

通过实验验证，A*算法在找到最优路径时的表现较好。实验使用不同尺寸和布局的网格进行测试。网格的障碍物布局对搜索效率的影响：障碍物较多时，A*需要扩展更多节点，路径查找时间增加。启发函数的选择：启发函数对搜索路径的效率影响显著，采用欧氏距离作为启发函数时，能更好估计节点间的实际距离，从而减少搜索节点数，提高效率。

下图为样例 4 的输出：

4 A*算法和相近算法比较分析

4.1 A*算法与 dijkstra 的比较研究

A*算法和 Dijkstra 算法都是经典的路径搜索算法，但它们在处理方式、启发性和效率上存在显著区别。虽然两者的应用场景有所不同，但是 A*算法在策略上很难不让人联想到 dijkstra 算法，在我看来，网格图中进行最段路规划 A*算法可以看作是对 dijkstra 算法的改进版本。

4.1.1 算法的目的

Dijkstra 算法是一种通用的单源最短路径算法，用于从起点到所有其他节点找到最短路径，主要应用于加权图中。Dijkstra 适用于任何图，不论是否有目标节点。

A*算法是专门用于目标导向的路径搜索算法，从起点找到到达特定目标的最短路径。它在搜索过程中引入了启发函数，使得算法更加高效，主要用于启发式搜索。

4.1.2 启发函数

Dijkstra 算法没有启发式函数，它只依赖从起点出发的实际代价(g(n)来决定节点的优先级。算法扩展时只考虑从起点到当前节点的代价，不考虑目标节点的方向或距离。

A*算法引入了启发函数(h(n)，它估计从当前节点到目标节点的代价。这种启发式搜索使得 A* 算法在寻找目标节点时更加智能，能够优先扩展更有可能到达目标的节点。

4.1.3 搜索效率

Dijkstra 算法算法扩展所有可能的路径，最终保证找到从起点到每个节点的最短路径。这意味着它的效率较低，尤其在目标节点距离起点较远时，因为它无法“猜测”目标的方向。

A*算法由于引入了启发函数 h(n) ,A* 算法可以优先扩展更有希望到达目标的路径，而不必像 Dijkstra 那样扩展每一条路径。启发函数 h(n)引导算法朝目标方向前进，减少不必要的节点扩展，从而提高效率。

4.1.4 最优性和完整性

Dijkstra 算法保证最优性，即找到从起点到所有其他节点的最短路径。它会遍历所有可能路径，确保路径代价最低。

A*算法在启发函数 h(n)可接受（不会高估代价）且一致（满足三角不等式）的情况下，同样可以保证找到全局最优解。

4.1.5 时间复杂度

Dijkstra 算法的时间复杂度取决于使用的数据结构。如果采用二叉堆实现优先队列，时间复杂度为 O((V + E) log V) ，其中 V 是节点数，E 是边数。

A*算法的时间复杂度在最坏情况下接近 Dijkstra 的复杂度 O((V + E) log V)。但由于启发函数的引导作用，在实际应用中，A* 通常扩展的节点数会比 Dijkstra 少，因此表现更加高效。

4.2 A*算法与 D*算法的比较研究

D*算法（Dynamic A*）是 A*算法的动态版本，旨在处理动态环境中的路径规划问题。与 A*算法不同，D*可以应对路径上出现的变化，比如障碍物的变化或被阻挡的路径。以下是 D*算法和 A*算法之间的主要区别 D*的改进之处：

4.2.1 算法背景与应用场景

A*算法是一种静态路径规划算法，适用于已知且不变的环境。它假设起点、终点和所有障碍物在搜索过程中是固定的，一旦计算出路径，便假设该路径可以顺利执行。

D*算法设计用于动态和未知环境，在路径规划过程中，环境可能会变化（例如，新的障碍物出现或消失）。D*算法能在机器人或智能体发现障碍物阻挡其路径时，自适应地重新规划路径，常用于机器人导航和自动驾驶等领域。

4.2.2 动态性

A*算法假设整个地图在算法执行时已经完全已知且不会变化，一旦生成了从起点到终点的最优路径，算法便完成工作。在遇到障碍物或地图变化时，A*算法需要完全重新计算。

D*算法是为动态环境设计的，能够在路径上的环境发生变化时高效地更新原有路径。它不需要从头开始计算，而是通过增量更新已存在的路径，节省了大量计算资源。当机器人在前进过程中遇到新的障碍物时，D*可以调整之前的路径，而不必重新规划整个路径。

4.2.3 工作原理

A*算法通过使用代价函数 $f(n) = g(n) + h(n)$，从起点向目标扩展最优路径。它一次性完成路径规划，并假设路径上的环境不会改变。

D*算法从目标节点反向开始，逐步推进到起点。它首先计算一个初步路径，但在实际执行过程中，当发现路径中不可预见的障碍时，会动态调整。D*使用一个类似 A*的代价函数，但在遇到变化时，能有效更新路径，而无需从头搜索。

4.2.4 增量更新

A*算法一旦障碍物发生变化，A* 算法没有机制来部分更新路径，必须重新从头开始运行整个算法。

D*算法通过增量更新来处理变化。当路径被阻断时，D*只需重新计算受影响的部分，不需要完全重新规划。这个增量更新的过程使得 D* 在动态环境下更高效。

4.2.5 时间和空间效率

A*算法的时间和空间复杂度依赖于整个搜索空间的大小。在大规模搜索空间中，尤其是在动态变化的环境中，它的效率会变得较低。

D*算法通过局部更新机制提高了效率，减少了重新计算的范围。在动态变化的环境中，D*算法的时间复杂度相较于 A*更为可控，因为它只重新计算路径变化的部分。

4.2.6 路径的最优性

A*算法在静态、已知的环境下，可以保证找到全局最优路径，但在动态环境下没有处理变化的机制，因此可能会因障碍物变化而失效。

D*算法可以在动态环境中找到接近全局最优的路径，虽然最优性在某些情况下可能稍有损失，但它的主要目标是在动态环境中有效工作。通过增量调整，它能保持接近最优的路径规划。