**论文编号：**

**A\*算法的起源分析与优化改进**

王蔚达

(同济大学软件学院，上海 201800)

**摘 要:** A\*算法是基于Dijkstra算法与最佳优先搜索算法的改进算法，也是是当下常用的路径规划算法。本文对于其起源与其定义进行了介绍。同时，为了解决经典A\*算法搜索时间较长的问题，即平衡A\*算法的最短路径和搜索速度之间的关系，论文从引入权重系数、减少搜索邻域、采用双向搜索策略和JPS策略、路径平滑处理等四个方面对传统A\*算法进行了改进。并通过设计实验进行测试，证明了改进的有效性。

**关键词:** A\*算法；路径规划；启发函数；双向搜索；跳点搜索算法

**Origin analysis and optimization improvement of the A\* algorithm**

Wang Weida

*(School of Software Engineering, Tongji University, Sahnghai 201800,China)*

**Abstract**: The A\* algorithm is an improved algorithm based on Dijkstra's algorithm and the best-first search algorithm, and is also a common path planning algorithm nowadays. The paper introduces the origin and definition of the classical A\* algorithm.Meanwhile, in order to solve the problem of the long search time of the classical A\* algorithm, which means balancing the relationship between the shortest path of the A\* algorithm and the search speed, the thesis improves the traditional A\* algorithm in four aspects, including the introduction of weight coefficients, the reduction of searching neighbourhoods, the application of two-way search strategy and JPS strategy, and the path smoothing process.It was also tested by designing experiments to prove the effectiveness of the improvement.

**Key Words**: A\* algorithm;path planning; heuristic function; two-way search; jump point search

—————————————

1. **引 言**

**1.1 背景**

目前路径规划算法基本分为基于图搜索算法的传统算法和智能算法两种类型。其中，图方法以Dijkstra算法、A\*[1]算法等经典方法最为知名，还包括群体智能算法（如蚁群算法等）以及智能算法（如遗传算法等）。在如智能仓储这样的静态环境中。A\*算法以其卓越的优越性、完备性和高效性，在路径规划中得到了广泛的应用。

**1.2 问题描述**

在传统图搜索算法中，有Dijkstra与BF两种算法，二者的优缺点也相对明显——Dijkstra算法可以找出最短路径，但搜索时间较长；BF算法可以较快地找出路径，但路径不一定是最短路径。两者各有所长，但却不能取长补短。A\*算法可以说是集大成者，集合了Dijkstra与BF算法的特性。

但经典的A\*算法有搜索时间长、较难平衡搜索准确性与搜索时间的问题。基于此，本文经过算法分析与设计了四种方式对算法进行改进。并将相应改进后的算法应用于虚拟模型上，以检测其优化效果。

**2 基本算法**

**2.1 BFS算法**

这里的BFS算法是指常规意义下的BFS算法。从物体所在初始点开始，访问图中的结点。迭代检查待检查结点集中的结点，并把和该结点最靠近的尚未检查的结点加入待检查结点集。接着，该结点集从初始结点向外扩展，直到到达目标结点。

BFS算法能够保证在所有的边都为非负的代价值时，找到一条从初始点到目标点的最短路径。但其缺点也相对较为明显，它需要菱形的形式，几乎遍历图上所有的点，这是非常不划算的。

**2.2 Dijkstra算法**

Dijkstra算法是BFS算法的拓展，BFS算法仅应用在图上各条路径花费都相同的情况下，而Dijkstra可以用于路径有权重的情况。在Dijkstra算法中，需要计算每一个节点距离起点的总移动代价，并不断更新，以确保每个节点的最短路径。这需要需要一个优先队列结构，对于所有待遍历的节点，放入优先队列中会按照代价进行排序。在算法运行的过程中，每次都从优先队列中选出代价最小的作为下一个遍历的节点。直到到达终点为止。

Dijkstra算法也可以保证能够找到一条权重最小的路径，但它的缺点也与BFS的类似，它也是需要遍历一遍整张地图才可以找到终点。

**2.3 BF算法**

BF（Best-First Search）算法被称为最佳优先搜索算法，也在一些文章中其也被称为Greedy Best First Search[2]。它则是Dijkstra算法的优化，其原理也与Dijkstra算法类似。也使用一个优先队列，以每个节点到达终点的距离作为优先级，每次始终选取到终点移动代价最小（离终点最近）的节点作为下一个遍历的节点。

这样做可以大大加快路径的搜索速度，但也值得注意的是，如果起点和终点之间存在障碍物，则最佳优先算法找到的很可能不是最短路径[3]。

**3 传统A\*算法**

A\* 算法是静态环境中用于求解最优路径的最有效的直接搜索算法，它结合了最佳优先搜索算法和Dijkstra 算法的思想，在保证得到最优路径的基础上，采用启发式搜索。

**3.1 估价函数**

A\*算法通过一个估价函数来确定搜索方向，从起点开始向周围扩展，通过估价函数计算得到周围每个节点的代价值，选择最小代价节点作为下一个扩展节点，重复这一过程直到到达目标点，生成最终路径．在搜索过程中，由于路径上的每个节点都是具有最小代价的节点，因此得到的路径代价是最小的。A\*算法的估价函数*f (n)*表示为



式中的*f(n)*表示从起始点经由任意节点*n*到达目标点的估价函数，*g(n)*表示起始点到节点*n*的实际代价，*h(n)*又被称为启发函数（heuristic function）,表示节点*n*到目标点的估计代价。

**3.2 启发函数**

启发函数*h(n)*应该小于实际情况*h\*(n)*，否则将会出现估计不准确的情况。对于不同的运动方式有不同的启发函数，用于估计点与终点之间的距离，下面对于常见的三种启发函数进行介绍。

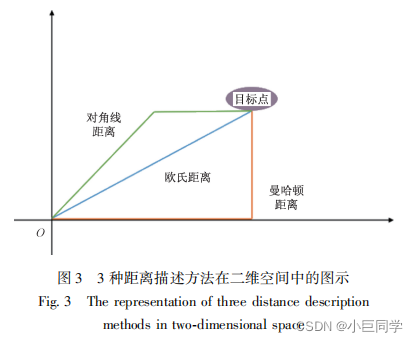


图1 三种距离描述方法的二维图示

**3.2.1 曼哈顿距离（Manhattan distance）**

如果个体只能沿着上下左右四个方向移动，则常用曼哈顿距离充当启发函数*h(n)*，其表达式如下：



**3.2.2 对角线距离（Diagonal distance）**

如果个体只能沿着上下左右以及对角线八个方向移动，则常用对角线距离充当启发函数*h(n)*，其表达式如下：



其中，*D*1表示个体向上下左右方向走时的花费，*D*2表示个体沿对角线走时的花费。当*D*1=*D*2时，称之为切比雪夫距离（Chebyshev Distance），当*D*1*D*2时，称之为八进制距离（Octile Distance）。

**3.2.3 欧几里得距离（Euclidean distance）**

如果个体可以沿着任意方向移动，则可以使用欧几里得距离充当启发函数*h(n)*，其表达式如下：



值得注意的是，常有示例将其写作如下形式：



这是不合理的，*h(n)*应该与*g(n)*阶数相匹配才便于之后对于算法的调整。

综上，不同的移动方式对应于不同的启发函数，且可知当点与终点之间无障碍时，*h(n)=h\*(n)*,此时达到最佳估计，估计代价与实际代价相同，此时函数将仅仅寻找最佳路径而不扩展其他任何节点[4]。

**3.3 算法核心**

A\*算法朝着趋近目标的方向进行搜索，搜索过程中对搜索方向上的每个节点进行考察．当到达某一节点时，该节点的周围节点会被添加到OpenList[5]，A\*算法会选择 OpenList 中具有最小估价值的节点作为下一个扩展节点，同时将该节点加入到ClosedList，重复执行这一过程，直到目标节点添加到 OpenList，寻路成功。其中OpenList采用优先队列来实现，优先队列是一种二叉堆结构，能够使得每次弹出的都是队列中的最小值，且用时较小。

其伪代码如下：

**Algorithm** A\*

**Input:** start, goal, h(n)

**Output:** Path

01: OpenList.clear();

02: ClosedList.clear();

03: OpenList.push(start)

04: **while** OpenList.empty()≠0 **do**

05: current=OpenList.pop()

06: **if** current = goal **then**

07: **return** true

08: **else**

09: **for** m in unvisited current’s neighbors **do**

10: **if** g(m) =infinite **then**

11: g(m)←g(current)+cost(current,m)

12: OpenList.push(m)

13: **else** g(m)>g(current)+cost(current,m) **then**

14: g(m)←g(current)+cost(current,m)

15: **end if**

16:  **end for**

17:  **endif**

18: ClosedList.push(current)

19: **end while**

**3.4 算法比较**

均衡比较Dijkstra、BF、A\*三个算法（如图1）不难发现，A\*算法继承了Dijkstra算法能够找到最短路径的特点，同时也继承了BF能够迅速访问周围节点的特点，这与其估价函数有很大关系。

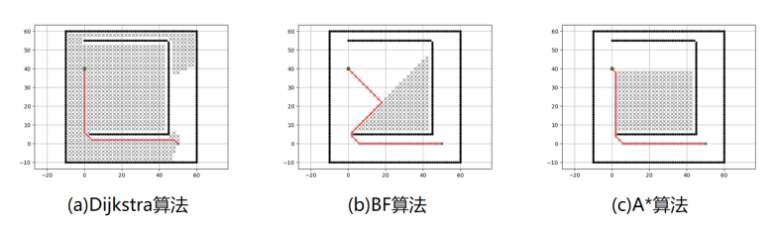


图2 Dijkstra、BF、A\*算法比较

**4 改进A\*算法**

本文中从以下四个方式对A\*算法进行改进：（1）引入权重系数，实现动态加权；（2）减少搜索邻域，基于8邻域搜索进行改进；（3）改变搜索策略，进行双向搜索、破坏对称性；（4）将路径平滑处理，采用贝塞尔曲线进行处理。

**4.1 引入权重系数**

如上所述，可以将A\*算法看作是对于Dijkstra与BF算法的调和，等价于是对于寻路过程的准确性与速度的调和。基于此，在A\*算法的原估价函数*f (n)*的基础上加上权重系数*w(n)*用于调整路线，以满足不同需求，改进后的表达式如下：



其中*w(n)*对于*f(n)*的影响总结如下：

1. 当*w(n)\*h(n)=*0时，*f(n)=g(n)，*A\*算法演变为Dijkstra算法，此时准确性达到最佳，速度达到最慢，能够保证找到最短路径；
2. 当*w(n)\*h(n)>>g(n)*时，A\*算法演变成为BF算法，此时，速度达到最大，能够以较短的速度找到一条路径，但路径不一定最短；
3. 当*w(n)\*h(n)*介于其之间时，可以综合速度和准确性，获得一条还不错的路径。

基于上述分析，可以提出两种改进方案：

1. **静态权重法：**将w设置为定值，用于调整选择方式。在一些文献中，将其称为决胜因子（tie breaking factor），引入此值可以减少A\*算法探索相同估价函数值*f(n)*路径的条数。还有公式来确定固定权值*w*[6]：



1. **动态权重法：**考虑到实际运行情况，路径搜索算法在开始搜索时，最重要的是迅速移动到任意位置；而在搜索接近结束时，最重要的是移动到目标点。所以在启发式函数中设计权重 *w( w≥*1*)* ，当接近目标时，降低这个权值，从而降低启发函数的重要性，同时增加了路径真实代价的相对重要性。如此，可以实现根据路径实时更新权重。

由此，采用静态加权与动态加权两种方式对于例子进行调整，获得搜索路径结果如下：

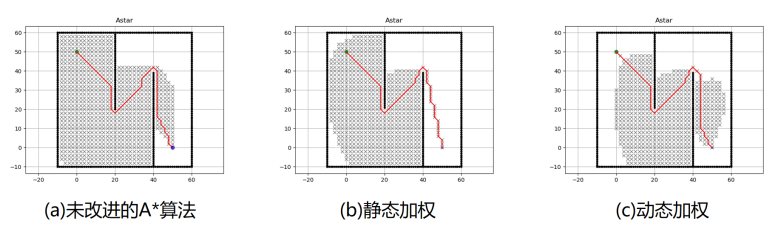


图3引入权重系数调整A\*算法

**4.2 减少搜索邻域**

传统的A\*算法，是在当前节点即父节点的周围进行8个方向的搜索，在不存在障碍物的前提条件下，移动机器人可以向周围8个方向的子节点移动，如下图所示。

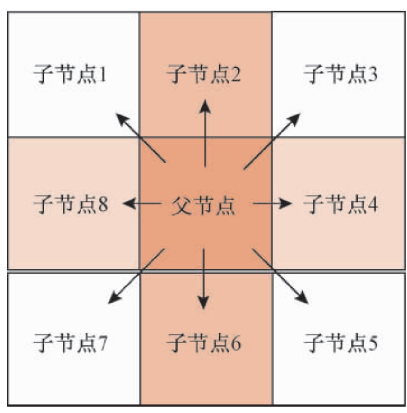
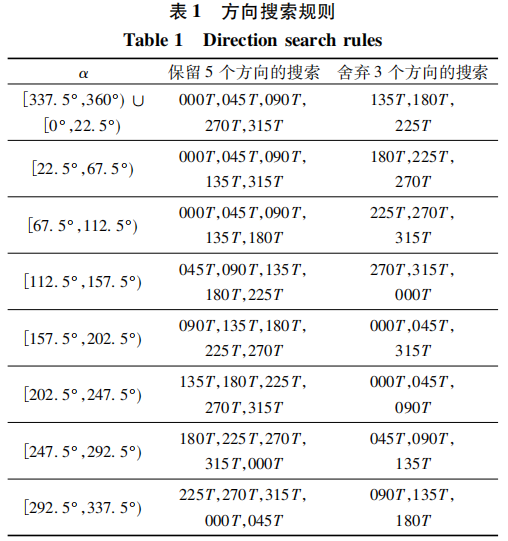


图4 子节点图[7]

在实际操作中，起点与终点位置相对确定时，可以通过减少搜索邻域的方式来提升搜索效率[7]。对于实际操作中，可以细分为以下两个步骤：

**第一步** 将传统的8个搜索节点方向改为5个，主要判断依据是起始点与目标点的连线与正北方向的夹角，规则如表1所示。

表1 方向搜索规则[7]



**第二步** 根据父节点与8个子节点的关系，将障碍物与当前结点的相对位置为参照分类：设定子节点2和字节点6为Ⅰ类，子节点8和子节点4为Ⅱ类。如果障碍物为Ⅰ类，则删除子节点2和子节点6; 如果障碍物为Ⅱ类，则删除子节点4和子节点8;如果障碍物不是Ⅰ类和Ⅱ类，则不做处理。

基于上述分析，可以得到结果如下：

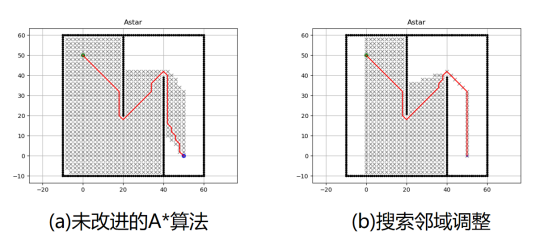


图5 调整邻域的A\*优化算法

**4.3 改变搜索策略**

在此处，有两种搜索策略上的改进——双向搜索和JPS。

**4.3.1 双向搜索**

当给出了起点状态与终点状态时，使用普通的搜索从起点向下搜索，则效率会很低，搜索树会非常庞大。这时可以考虑使用双向搜索，及从起点与终点同时向中间搜索，搜索到同一个状态时，将从起点与终点搜索的值相加得到最终值的搜索。

对于A\*算法而言，需要两个OpenList，分别从起点与终点开始搜索，直到出现两个OpenList中有点重复，则此时便可以路径相连接。

但是，传统的双向收缩也会出现搜索顺序的先后，如何促进两者相遇的问题，对此解决方案如下：

1. 针对搜索顺序的先后问题，采用目标重定（retargeting）的方式：搜索一段恒定的次数，获得一个正向候选者（best forward candidate），之后进行反向搜索；反向搜索以正向候选者作为目标，继续收缩相同次数，获得反向候选者（best backward candidate），如此循环，直至获得最终的交点。
2. 针对于两路相交的问题，引入动态窗口的概念，在正向反向路径同时搜索时，判断两条路径搜索节点的距离，以两个搜索节点的横坐标和纵坐标的距离差为基础，建立搜索动态窗口，当两个搜索节点横纵坐标差同时小于某一定值时停止搜索。然后获得这两个节点坐标值，再次使用 A\*算法，搜索出正向搜索节点至反向搜索节点的路径，这样就得到了一条由正向搜索路径，中间搜索路径和反向搜索路径组成的最终路径[1]。
3. 斯坦福大学的团队提出了front-to-front的解法可以有效地解决上述两个问题：将OpenList中估价函数非无穷大的部分的集合称为front，目标点称为end。在该算法中，正反搜索都是从自己的front向对方的front中搜索[8]。其将估价函数的g(n)的部分拆解成两个部分：



其中*f(x,y)*是点对(x,y)的估价函数，其越小，则该点对优先级越高。*g(start,x)*是从起点到节点*x*的正向搜索的已付代价，*h(x,y)*是从节点*x*到节点*y*的代价的启发式值，*g(y,goal)*是从目标点到节点*y*的反向搜索的已付代价。

如此，正向和反向搜索点均放入同一个OpenList中，解决了搜索先后顺序的问题。此外，再加入了*h(x,y)*起到了正向引导作用。

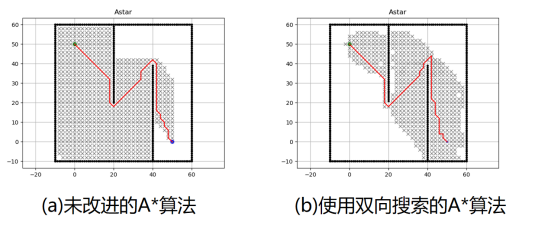


图6 基于双向搜索对A\*的改进

**4.3.2 JPS优化**

JPS算法全称为Jump Point Search，又称为跳点搜索算法，它保留了A\*算法的主体框架，与其区别在于A\*每次只能移动到周围的某一位置，而JPS可以一次跨越多格。

**4.3.2.1 概念描述**

这里对邻居节点进行分类，分类之后进行不同的处理：（1）隐性邻居（inferior neighbors）：从父节点直接到达此节点的开销比父节点经由子节点再到达此节点的开销更加低，不需要向此类节点扩展，如图8中灰色节点；（2）显性邻居（natural neighbors）：从父节点经由子节点再到达此节点的开销更低，此类节点需要扩展，如图8中白色节点；（3）强迫邻居（forced neighbors）：父节点直接到达此节点的开销更加低，但路径被障碍物挡住，只能经由子节点到达，此类节点最终被设置为跳点，如图8中红色节点。

跳点：满足以下条件之一的点：①如果点C是起点或目标点，则C是跳点；②如果C有强迫邻居，则C是跳点；③如果 P(C)（parent(C)）到C是水平或者垂直移动，并且C经过水平或垂直方向移动可以到达跳点，则C是跳点，例如图8(c)中，C是跳点；④如果P(C)到C是对角线移动，并且C经过水平或垂直方向移动可以到达跳点，则C是跳点，如图8(d)中，C是跳点。

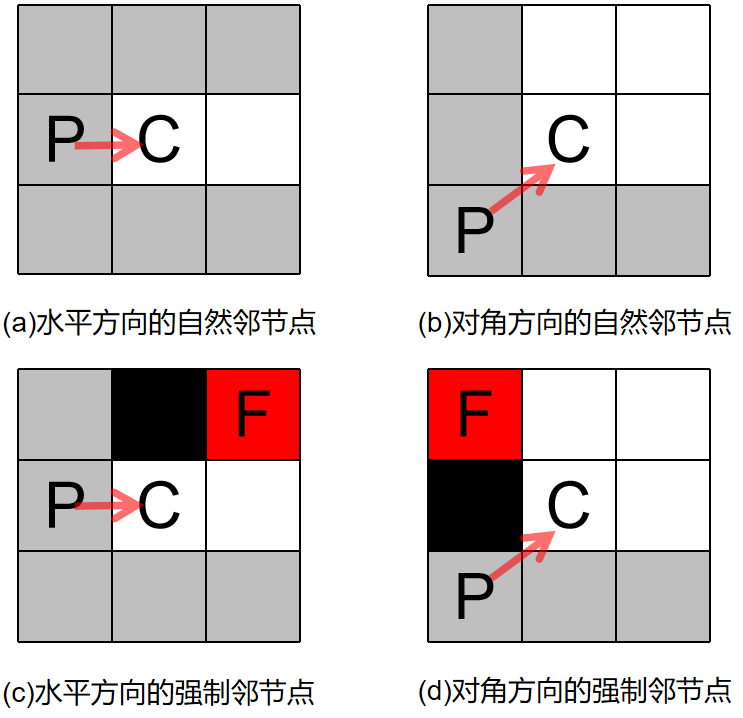


图7 强制邻节点搜索策略

**4.3.2.2 跳跃规则**

（1）JPS 搜索跳点的过程中，如果直线方向、对角线方向都可以移动，则首先在直线方向搜索跳点，然后再在对角线方向搜索跳点。

（2）只有跳点才会加入OpenList，因为跳点会改变行走方向，而非跳点不会改变行走方向，最后寻找出来的路径点也都是跳点。[9]

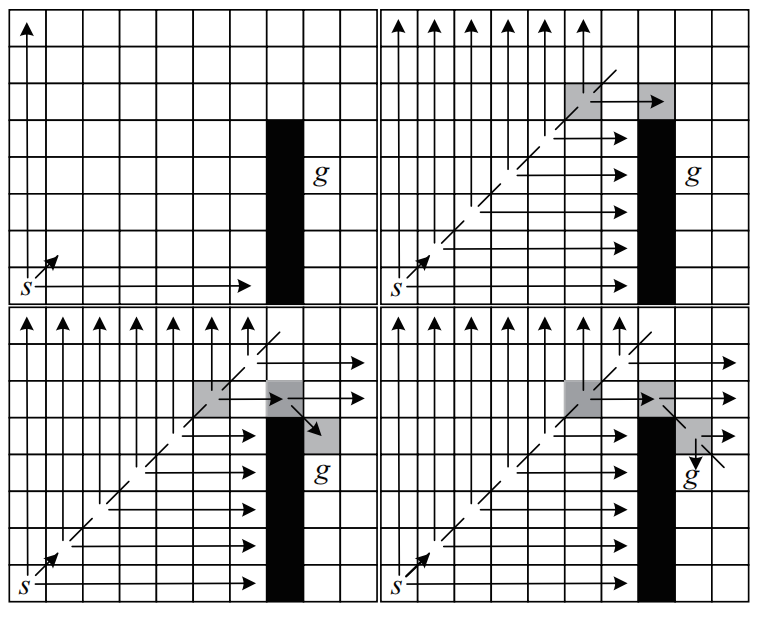


图8 JPS算法寻路过程[10]

在地图栅格图8中，s表示起始节点，g表示目标节点，首先将起始节点s添加到OpenList，并沿着水平和竖直方向进行扩展（扩展方向取决于父节点到当前节点的方向，当父节点不存在时，先向水平和竖直方向扩展），直到遇到障碍物或到达地图边缘，这时候开始沿着对角线扩展．当沿着对角线扩展到一个新的节点时，继续沿水平和竖直方向扩展，重复此过程，直到竖直方向的扩展到达地图的边缘，水平方向的扩展发现一个强制邻节点，这时候将强制邻节点添加到OpenList，强制邻节点作为一个跳点继续向右扩展，直到到达地图边缘，此时沿着对角线继续扩展，在扩展过程中发现下一个强制邻节点，将其加入OpenList并作为新的跳点继续沿水平和竖直方向扩展，直到发现目标点．这时停止迭代，将目标点加入OpenList，得到最终路径。

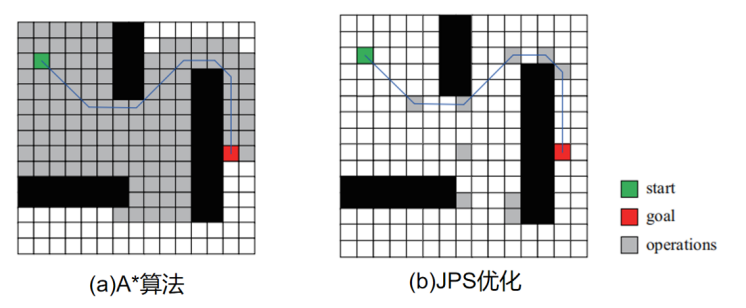


图9 JPS优化效果图[10]

**4.3.2.3 优化分析**

大多数情况下，尤其是在复杂环境情况下，JPS相较于A算法更佳，它可以通过跳跃来减少节点的访问过程（即减少了将点放入OpeanList的过程），对于迷宫问题中非常实用。对于较大空间内，且终点离起点较近时，JPS算法的效果明显不如A算法。JPS算法只是减少了OpenList中元素的个数，但是它增加了障碍物搜索（status query）所需的时间。

值得注意的是，JPS策略只可以在规范均匀网络（uniform grid map）上使用和实现，对于广义图搜索中并不适用[11][12][13]。

**4.4 路径平滑处理**

在大多数情况下，寻路算法是应用于实体上，如果路径不够平滑，则会有一定危险性上的问题。加之A\*算法规划路径中也确实存在折线多、转折次数多的问题，由此需要设计算法对路线进行改良优化。

王红卫等人在A\*算法规划路径的基础上，遍历路径中的所有节点，当某一节点前后连线节点连线上无障碍物时，将延长线路的这一中间节点删除，建立平滑A\*模型，可以达到累计转折次数降低约50%、累计转折角度减少30%~60%的显著成果[14]。

本文在曲线平滑处理上也提出了一种基于贝塞尔曲线（Bézier Splines）的路径平滑处理方法。由*P1、P2、*...、*Pn*构成的多阶参数形式贝塞尔曲线公式如下：



实际选取的阶数可以由路径所需的精确程度而调整。基于此对于路径改进的效果如下：

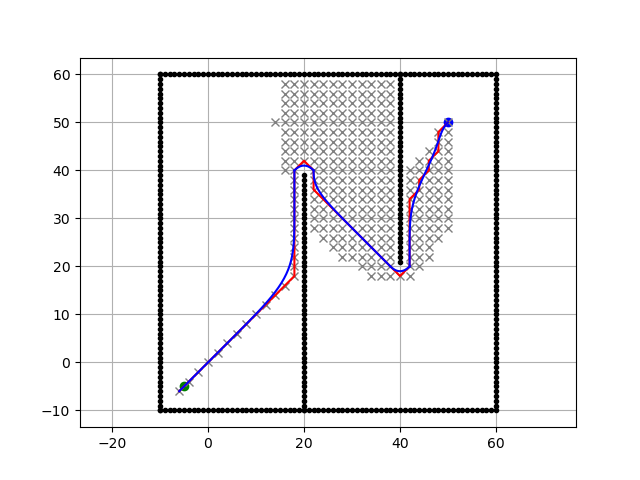


图10 贝塞尔曲线平滑处理

1. **实验建立与结果分析**

本实验基于Python 3.9环境，采用matplotlib库，虚拟了一个地图，并采用不同改进后算法分别从起点出发寻找其终点。同时将路径搜索过程以图像的形式直观的展示出来，并比较从不同角度进行优化对于传统A\*算法的优化效果。

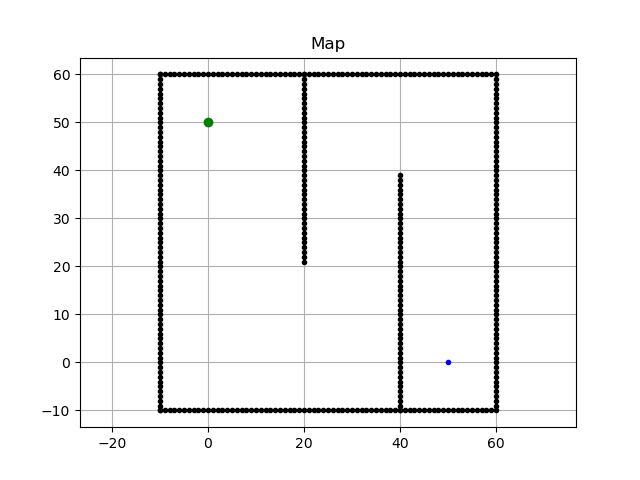


图11 虚拟地图

在虚拟实验过程中，以访问节点个数以及路径节点数（是否最短）作为指标，其结果如下：

表2 算法效果比较

|  |  |  |
| --- | --- | --- |
| 方法 | 访问节点个数 | 是否最短 |
| Dijktra算法 | 1071 | 是 |
| 最佳优先搜索 | 349 | 否 |
| 传统A\* | 809 | 是 |
| 静态权值 | 691 | 是 |
| 动态权值 | 616 | 是 |
| 减少搜索邻域 | 674 | 是 |
| 双向搜索 | 566 | 是 |
| JPS | 6（仅计算跳点） | 是 |

可见上述改进方案在此案例下都有一定程度的优化效果，除JPS优化外，其他优化策略都可以使得访问节点个数在BF算法与传统A\*搜索算法之间，可以在一定程度上提升了A\*算法的搜索速度。

**6 结 论**

A\*算法是起源于Dijkstra算法与最佳优先搜索算法的一种算法，其一定程度上继承了Dijkstra能够找到最短路径的准确性，以及最佳优先搜索算法的寻路速度。

基于A\*算法的定义，可以从以下三个方面进行优化：（1）引入权重系数，实现动态或静态加权。（2）减少搜索邻域，将8邻域转化成5邻域进行简化处理；（3）改变搜索策略，可以使用双向搜索、JPS搜索的方式进行算法优化。

基于实验成果来看，以上改进方法优化效果良好。

**参考文献**

1. 陈德童, 刘贤达, and 刘生伟. "基于双向搜索改进A\*算法的自动导引车路径规划." 计算机应用 041-0z2(2021).J. M. Miranda, "A BIST and boundary-scan economics framework", IEEE Design & Test of Computers, 1997, pp. 17-23.
2. Heusner, M., T. Keller , and M. Helmert . " Understanding the Search Behaviour of Greedy Best-First Search." 2017.
3. Heusner, M. , T. Keller , and M. Helmert . "Search Progress and Potentially Expanded States in Greedy Best-First Search." Twenty-Seventh International Joint Conference on Artificial Intelligence {IJCAI-18 2018.
4. Rong, Z. , and E. A. Hansen . "Sweep A\*: Space-efficient heuristic search in partially ordered graphs." IEEE International Conference on Tools with Artificial Intelligence IEEE, 2003:427-434.
5. Bundy, A. , and L. Wallen . "A\* Algorithm." Springer Berlin Heidelberg (1984).
6. Liang, Q. , et al. "An Adaptive Weight A\* Algorithm Applied to UAV Path Planning." The International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (2022).
7. 吴飞龙, and 郭世永. "融合改进A和动态窗口法的AGV动态路径规划." 科学技术与工程 030(2020):020.
8. Barker, J. K. . "Front-To-End Bidirectional Heuristic Search." Dissertations & Theses - Gradworks (2015).
9. 魏博闻, and 严华. "一种面向非结构化环境的改进跳点搜索路径规划算法." 科学技术与工程 021.006(2021):2363-2370.
10. 赵晓等. "基于改进A算法的移动机器人路径规划." 机器人 6(2018):8.
11. Qiu, L. . "Pathfinding on Grid Maps Based on Online Graph Pruning." Journal of Guizhou University(Natural Sciences) (2014).
12. 周熙栋, 张辉, and 陈波. "非结构化场景下基于改进JPS算法的移动机器人路径规划.".
13. Liu, S. , et al. "Planning Dynamically Feasible Trajectories for Quadrotors Using Safe Flight Corridors in 3-D Complex Environments." IEEE Robotics & Automation Letters (2017):1-1.
14. 王红卫等. "基于平滑A\*算法的移动机器人路径规划." 同济大学学报：自然科学版 38.11(2010):5.