**论文编号：**

**基于迷宫寻路的A\*算法优化**

**摘要**: A算法是解决迷宫寻路问题中最常用且高效的路径规划算法之一。然而，传统A算法在复杂环境或大规模迷宫中容易面临计算速度较慢的问题。本文围绕A算法的核心机制，从启发式函数的选择与优化、双向搜索策略、开放列表数据结构优化以及并行计算五个方面进行深入研究与实际实现。针对不同场景下A\*算法的瓶颈，提出了改进的启发式函数以及高效的双向搜索策略。通过将开放列表采用更高效的数据结构管理，并引入并行计算以加速路径搜索过程，整体提升了算法的运行效率与扩展能力。实验结果表明，优化后的A\*算法在多种复杂迷宫环境中均能显著减少路径搜索时间，并生成更优路径。本文的研究为迷宫寻路及路径规划问题提供了系统的优化思路和实践方案，具有较高的应用价值。

**关键词:** 迷宫寻路，A\*算法优化，启发式搜索，双向搜索，并行计算，路径规划，数据结构优化

**Optimization of A Star Algorithm for Maze Pathfinding**

**Abstract**: The A star algorithm is one of the most commonly used and efficient pathfinding algorithms for maze navigation. However, the traditional A star algorithm often faces the challenge of slow computation in complex environments or large-scale mazes. This paper focuses on the core mechanisms of the A star algorithm, conducting in-depth research and practical implementation from five perspectives: heuristic function selection and optimization, bidirectional search strategies, open list data structure optimization, and parallel computing. To address the bottlenecks of the A star algorithm in different scenarios, this study proposes an improved heuristic function and efficient bidirectional search strategies. By managing the open list with more efficient data structures and introducing parallel computing to accelerate the pathfinding process, the algorithm’s operational efficiency and scalability are significantly enhanced. Experimental results show that the optimized A star algorithm reduces path search time and generates better paths in various complex maze environments. This study provides systematic optimization ideas and practical solutions for path planning problems, offering significant application value.

**Key Words**: Maze Pathfinding, A star Algorithm Optimization, Heuristic Search, Bidirectional Search, Parallel Computing, Path Planning, Data Structure Optimization

—————————————

**1** **引 言**

**1.1 研究背景与意义**

迷宫寻路问题在计算机科学与人工智能领域具有重要的研究价值和广泛的应用场景。无论是在游戏开发中为角色规划移动路径，还是在机器人导航与自动驾驶中为车辆或机器人提供实时路径指引，都需要高效、稳健的寻路算法来完成从起点到目标点的路径搜索。A\*（A-Star）算法凭借其结合了启发式信息与实际代价的搜索特点，一直是解决迷宫寻路与通用路径规划问题的首选算法之一。然而，随着应用需求的不断升级和运行环境的日益复杂，传统 A\* 算法在大规模或障碍物分布不均的迷宫中往往暴露出若干不足之处：当搜索空间非常庞大时，A\* 的计算速度会显著下降，而开放列表和关闭列表中的节点数也会呈指数级增加，导致算法的内存占用量成倍上涨。这些问题在实时性要求较高或计算资源有限的场景下尤为突出。因此，对 A\* 算法进行针对性的优化，对于各种需要路径规划的场景（如游戏、机器人、自主车辆等）都具有重要的理论与实践意义。

**1.2 当前迷宫寻路算法的现状**

迷宫寻路问题本质上是一种在离散或连续空间内寻找最短（或近似最短）路径的图搜索问题，因此衍生出多种搜索算法。早期常见的方法包括 BFS（宽度优先搜索）和 DFS（深度优先搜索）。其中，BFS 能够保证在无权图上找到最短路径，但随着迷宫规模的增大，其在无效区域也会进行大量搜索，从而耗费大量时间与空间；DFS 虽然编程实现简单，但更适合快速找到一条可行路径，无法保证最短路径，且在复杂迷宫中容易陷入死胡同。在加权图环境中，Dijkstra 算法在保证最优解的前提下可以寻找最短路径，但由于没有启发式信息的辅助，它在大量无效区域上也会进行冗余探索，导致效率难以令人满意。A\* 算法综合了 Dijkstra 的最优性和启发式搜索的高效性，通过在搜索过程中引入启发式函数h(n)，使得对目标的距离或代价估计更加精准，从而在大多数情形下显著减少了无效搜索区域。

**1.3 传统 A\* 算法的优缺点分析**

A\* 算法在寻路效率和最优解保证上具有明显的优势。通过不同的启发式函数设计（如曼哈顿距离、欧几里得距离、切比雪夫距离等），A\* 算法可以适应多种类型的网格地图和不同的约束条件。同时，相比不使用启发式信息的 Dijkstra 算法，A\* 能够更快地集中在从起点到目标点的“更可能”区域上进行搜索，减少了大范围无效搜索。只要所选用的启发式函数满足一致性或可接受性的要求，A\* 算法可保证最终找到的路径是全局最优解。

然而，随着迷宫规模或环境复杂度的提升，A\* 算法也面临一些制约因素：

* 大量节点处理：在搜索过程中，需要维护两个核心数据结构——开放列表（Open List）和关闭列表（Closed List），当迷宫规模较大时，开放列表中的节点会迅速膨胀，造成较大的内存消耗和查询更新开销。
* 搜索速度降低：在障碍物密度较高或分布不均的环境中，A\* 需要频繁地更新许多不必要的节点信息，导致路径搜索速度明显下降。
* 对启发式函数的依赖：启发式函数过于保守会降低搜索效率，过于激进又可能导致解的不准确或搜索不稳定，且很难做到在各种环境下都具备较好的自适应性。

**1.4 本文的研究目标**

针对 A\* 算法在迷宫寻路中遇到的时间上瓶颈与挑战，本文从以下四个方面开展系统的研究与改进，力图在保证寻路准确性的同时，显著提升算法的执行效率与可扩展性：

* 启发式函数的选择与优化：结合迷宫环境中常见的障碍分布特征，提出改进的启发式函数设计，使其在不同障碍密度或形状下保持良好的估计能力。
* 双向搜索策略：在起点和终点同时进行搜索，或在中间区域进行会合，从而大幅缩小搜索空间，加快寻路速度。
* 开放列表数据结构优化：针对开放列表、关闭列表等关键数据结构进行改进，使插入、检索与更新操作更加高效。
* 并行计算的应用：利用多线程或多核并行手段，将搜索过程进行任务分解和同步，从而降低单线程环境下的时间消耗。

**2 相关工作介绍**

迷宫寻路问题最早可以追溯到图搜索和运筹学领域，其核心在于在离散或连续的空间中寻找一条从起点到目标点的最优或次优路径。随着应用需求的不断扩大，研究者们逐渐从单纯的遍历式搜索算法向具有启发式信息的搜索算法过渡，并对后者进行了多种形式的改进与优化。

**2.1 传统图搜索算法**

* BFS（广度优先搜索）

BFS 能够在无权图上保证最短路径，但其特点是会在所有可达节点上“平铺式”展开搜索。这意味着在规模较大的迷宫环境中，BFS 经常会拓展大量与最优路径无关的区域，不仅导致计算量大幅增加，而且在高障碍密度的迷宫中也难以应对复杂的路径分支。尽管 BFS 的实现简单且容易与其他算法组合，但其效率不足往往难以满足实时性要求。

* DFS（深度优先搜索）

与 BFS 相比，DFS 更偏向纵深搜索，能较快找到一条可行路径，但并不保证路径最优性。在复杂迷宫中，DFS 可能会在不必要的支路上进行过深的探索，从而浪费算力。此外，DFS 对最短路径的搜索不具备天然优势，通常只能作为初步搜集可行路径的方式。

* Dijkstra 算法

Dijkstra 算法是一种适用于加权图的单源最短路径搜索算法，能在保证最优解的前提下扩展搜索范围。然而，由于缺乏对目标点的启发式信息，Dijkstra 在大规模或障碍物分布复杂的环境中，会进行大量无谓的节点扩展，效率依然有限。在未使用任何启发式的情况下，Dijkstra 往往需要遍历几乎所有可达区域，导致搜索成本显著增加

**2.2 A\*算法与启发式搜索**

A\*算法在Dijkstra算法的基础上引入了启发式函数h(n)，结合了从起点到当前节点的实际代价g(n)与到目标节点的估计代价 h(n)，使得搜索更具有针对性。

A\* 算法在路径规划领域广受欢迎，主要原因在于：

* 在满足可接受性或一致性的条件下，A\* 能够保证所找到路径的最优性；
* 启发式的引入大幅降低了无谓的搜索节点，通常比 Dijkstra 算法显著提高效率；
* A\* 算法具有很强的可扩展性，可以根据实际需求灵活地设计或替换启发式函数。

随着应用需求的多样化，研究者们在 A\* 算法的基础上进行了进一步的优化和改进。常见的思路包括：

* 基于分块/抽象寻路的 A\*（如 HPA\*）：将大规模地图分为若干子区域，每个子区域内部使用 A\* 搜索并在更高层次将子区域连通，以减少整体的搜索空间。
* 迭代式加深 A\*（IDA\*）：将深度限制与启发式估计相结合，逐层加深搜索过程，对特定场景下能起到减少搜索开销的作用。
* 双向 A\*（Bi-directional A\*）：同时从起点和目标点两端进行搜索，在中间区域会合，从而显著缩短单向搜索需要遍历的范围。

**3 算法基础**

A\*算法核心在于通过启发式函数结合代价函数，动态地在图或网格中找到从起点到终点的最优路径。

**3.1 算法基本原理**

A\* 算法是对 Dijkstra算法的一种改进，将路径的“实际已走代价”与“尚未走完的估计代价”结合起来进行搜索，从而在保证最优解或近似最优解的同时，有效缩小搜索范围。其核心思想可以表述为：在图搜索中，每一个节点n都具有以下两个代价函数：

* g(n)：从起点到当前节点n的实际累积代价。
* h(n)：从当前节点n到目标节点的启发式估计代价。

A\* 算法在搜索时，会计算综合评价函数：



并在每一步从“开放列表”中选取拥有最小f(n)值的节点进行扩展。具体过程一般包括以下关键步骤：

* 初始化

将起点节点start加入开放列表（Open List）中，并设置 g(start)=0；

将闭合列表（Closed List）置空。

* 从开放列表中选取节点

在开放列表中寻找f(n)值最小的节点 n，将其移出开放列表并加入闭合列表；若n为目标节点，算法结束并可回溯得到最优路径；若开放列表为空，则说明搜索无解。

* 邻居节点拓展

对节点n的每个可行邻居节点（在迷宫中通常为上下左右四个格子，代表四个行进方向）进行检查；若邻居节点在闭合列表中，则跳过；若邻居节点不在开放列表中，则将其加入开放列表，并设置相应的g(neighbor)与h(neighbor)值；若邻居节点已在开放列表中，则根据新的g(neighbor)值判断是否需要更新其父节点或代价。

* 重复搜索与路径回溯

重复上述选取与扩展过程，直到找到目标节点或无法找到任何可行路径；当目标节点被取出开放列表时，即可追溯父节点指针（Parent Pointer）还原整条最优路径。

在经典的 A\* 算法中，算法会持续通过比较节点的f(n)值来决定搜索方向，因此启发式函数在引导搜索朝向目标方向方面发挥了关键作用。只有当 h(n)满足一定条件（如可接受性和一致性）时，A\*才能够在保持最优解的前提下显著减少搜索范围。

**3.2 启发式函数的设计原则**

1.可接受原则

若启发式函数h(n)对任何节点n都不大于从n到目标节点的真实最短距离（或真实最小代价），则称 h(n)为可接受的，即



其中表示从节点n到目标节点的实际最短路径代价。

满足可接受原则的启发式函数不会对目标距离进行高估。这样，A\* 在选择扩展节点时不会过早“跳过”潜在的最优路径，因而可以保证在搜索完成后找到最优解或者近似最优解。

2.一致性原则：

一致性（又称单调性，Monotonicity）要求对每一条从节点n指向其相邻节点n′的有向边，都满足如下条件：



其中是从n到的实际代价。

一致性保证了沿着搜索路径向前推进时，启发式函数不会跳跃性下降，从而使得 f(n)=g(n)+h(n)在扩展时呈现出一个非减的趋势。若启发式函数满足一致性，则当一个节点被放入闭合列表后，无需再次对其重新打开（re-open），极大地简化了 A\* 算法的实现与提升了运行效率。同时，一致性原则通常也隐含着可接受性：如果h(n)在所有边上都遵循上述不等式，则不太可能在起始节点处就超过到目标的真实距离。

**3.3 常见启发式函数的设计**

通过启发式函数h(n)获取的启发式估计代价选择对 A\* 算法的性能与最优性影响重大。一般来说，一个良好的启发式函数应当尽可能地逼近从当前节点到目标节点的实际最短距离或最小代价，同时要满足可接受性，否则可能破坏 A\* 的最优性。常见的启发式函数类型包括曼哈顿距离、欧几里得距离、切比雪夫距离。

1.曼哈顿距离(Manhattan Distance)：

在正交网格上，如果只能进行上下左右 4 个方向的移动，曼哈顿距离是最常用、最简单的估计方式。曼哈顿距离定义为：



其中，表示目标节点的坐标,表示节点n的坐标。它假定从节点n到目标只需在水平方向或垂直方向各移动若干步。

2.欧几里得距离 (Euclidean Distance)：

在可以进行任意方向移动网格中，欧几里得距离能够更准确地估计两点之间的最短距离。欧几里得距离定义为：



3.切比雪夫距离（Diagonal Distance）

切比雪夫距离主要用于棋盘格或允许对角移动的网格环境中。它表示两个点之间在横向、纵向或对角方向上移动所需的最少步数。切比雪夫距离定义为：



**4 A\*算法优化方法**

本节围绕A\*算法在迷宫寻路中的时间效率瓶颈，提出并详细阐述四个主要的优化方向：启发式函数的选择与优化、开放列表数据结构优化、双向搜索策略、并行计算。通过这些手段的有机结合，可在保证算法最优性或近似最优性的前提下，大幅提升搜索效率。下文将依次介绍每一项优化方法的原理、实现方式及其对算法性能的影响。以下小节将按照“启发式函数—数据结构优化—双向搜索—并行计算”的顺序，逐一阐述对 A\* 算法进行改进的具体方式。

**4.1 启发式函数的选择与优化**

上文已经谈到了三种基本常用的预估函数距离计算方式，它们都有各自的试用范围。

对于大多数迷宫环境而言，随着其复杂度和通道曲折程度的增加，大面积空旷区域出现的概率通常较低。换言之，迷宫中的移动往往局限于上下左右四个方向，而并不具备大范围的斜向或任意方向移动空间。在此情况下，曼哈顿距离就成为启发式函数的最优选择之一。它以水平和垂直步数为度量，既能准确反映走迷宫时的实际步进方式，又兼具计算简单、开销低等优势，能够很好地满足迷宫寻路的需求。

关于预估函数另一个方面的优化是改进启发函数的权重系数。在经典A\*算法中，实际代价 g(n)与启发式代价h(n)往往被简单地视作等比相加以兼顾较好的搜索效率和寻路质量。实际的迷宫环境中，若对速度或精度有更高要求，单一的 1:1 权重可能不足以应对多变的场景。通过调整g(n)与h(n)的权重系数，我们可以使算法在更偏向实际代价或启发式代价之间灵活切换：当实际代价权重占主导时，算法更接近 Dijkstra 的风格，注重当前已付出的代价；反之，如果启发式权重远超实际代价，则意味着搜索会急于朝着目标方向推进，但得到的路径可能不是全局最优。

基于此思路，可以实现的做法是在启发式函数前加上一个可调系数，形成



使得当w较大时算法更偏向启发式，搜索速度更快；当w较小时则更强调当前实际代价，接近最优路径。

通常情况下，我们在迷宫寻路时不太关注最优路径，仅仅只是想要寻找到从迷宫的入口抵达出口的路径，或者在迷宫中往往只有一条路径可走，这时，可以将w系数调高，使h(n)占比更大，提高搜索速度。

**4.2 对开放列表进行数据结构优化**

在A\*算法中，开放列表作为待处理节点的核心容器，需要频繁执行“选择并删除最优节点”“检查节点是否已在集合中”“插入新节点”三类关键操作。若能在数据结构层面进行针对性优化，便可有效降低这些操作的时间开销，从而提升算法整体效率。下文将从几种常用的数据结构入手，探讨它们在 OpenList 管理中的优劣与适用场景。

1.基础结构：数组或链表

将开放列表直接存储在一个未排序数组或普通链表中是最为直观的做法，但其在查找最优节点时需要遍历所有元素，渐进时间复杂度为O(n)；而插入新节点只需将元素追加到末尾（数组）或添加到链表表头/表尾，渐进时间复杂度平均为O(1)。这两种数据结构易插入、难查找，对规模较小或对插入操作频率要求极高的场景，这种简单结构可以为编程与维护提供便利。但当节点数量较多时，其在查找和删除最优节点方面的性能瓶颈就会比较明显。

2.有序数组与有序链表

为了加速查找最值，可以考虑将元素按照优先级进行排序，使得找到最优节点的过程得以简化。

对于有序数组，可在查找最优节点时直接访问首元素，时间复杂度为O(1)；然而，为维持有序性，每当删除最优节点或插入新节点时，有序数组可能需要移动大量元素，导致插入成本飙升至O(n)。

对于有序链表，在删除或定位最优节点时可在表头或表尾实现，时间复杂度为O(1)，但插入新元素时需要逐个遍历找到合适位置，导致时间复杂度变为O(n)，且元素的检查也通常只能通过顺序扫描完成。

因此，排序策略的优点是“获取最优节点”的时间复杂度降低，但插入操作往往会变得更昂贵，对于节点更新或插入频繁的情形可能并不划算。

5.二叉堆

在 A\* 算法的优先级选择里，最常见的符合“仅需快速获取最值”需求的结构即为二叉堆。与上面多数基于“有序集合”或“遍历集合”的做法不同，堆只需保证堆顶元素（最小或最大）符合优先级要求，而无需对所有元素进行全局排序。

将节点加到堆末尾，再上浮到合适位置，平均开销 O(logn)，弹出堆顶并用末尾元素替换，再下沉到恰当层级，平均开销同样是O(logn)，可在堆顶节点实现 O(1)。

二叉堆无疑是A\* 算法中最优的选择之一，既兼顾维护成本又具备足够的插入与删除效率。在工程实践中，如果对“集合中是否包含某节点”的检查需求不算太苛刻（可用辅助数据结构或直接遍历小规模情形），采用二叉堆通常能带来显著性能提升。

**4.3 双向搜索策略**

双向搜索（Bi-directional Search）通过同时从起点和终点发起搜索，并在中间某个区域交汇，大幅缩小了搜索空间。与单向搜索相比，理论上可将搜索时间缩减到原来的。

具体实现分为正向搜索和反向搜索两部分。其中，正向搜索以传统A\*的方式，从起点向外进行扩展；反向搜索同样以A\*进行搜索但目标交换，将终点作为“起点”，进行反向扩展。当某一方的节点在另一方的已访问节点中出现时，说明搜索路径在此节点已成功对接，可通过回溯或组合路径得到完整路径。搜索时，需要维护两套开放列表与关闭列表，分别对应正向和反向搜索，同时，需要搜索过程中定期检查两端搜索的拓展情况，平衡搜索深度以防止“偏向”其中一端。

双向搜索通常在较大的地图中呈现明显优点，尤其当起点与终点距离较远时，搜索时间可显著缩短。

**4.4 并行计算**

A\*算法的瓶颈通常在节点扩展及开放列表更新。当地图规模大、障碍物分布复杂、实时性要求高时，单线程 A\* 往往难以充分利用现代多核 CPU 或 GPU 资源。通过并行计算手段将节点扩展、启发式计算等过程拆分给多个线程或处理单元执行，可以在相同时间内处理更多节点，从而加快整体搜索进度。

在每一步取出待扩展节点之后，可将其相邻节点（邻居集合）分配给不同线程并行处理，包括启发式函数的计算、判断节点是否已经访问、更新开放列表或关闭列表等，加速数据处理速度。

同时，并行计算可以与A\*双向搜索策略相结合，进一步降低搜索消耗的时间。

在使用多线程，需要注意以下问题的妥善处理：

1.多线程需要频繁访问并修改开放列表、关闭列表等结构，若没有妥善的同步机制可能导致数据竞争或死锁。

2.若某些线程所负责的区域障碍物极多而另一些区域相对简单，就会出现局部线程繁忙、其他线程闲置的情况。需根据实时负载状况动态调配线程任务，以尽量平衡负载。

**5 实验设计与分析**

**5.1 实验环境与参数设置**

1.硬件环境

CPU：13th Gen Intel(R) Core(TM) i9-13900HX

内存：32GB

2.软件环境

操作系统：Windows 11

集成开发环境：Clion 2024

编程语言：C++ 20

依赖库：标准模板库

3.算法版本

* ABase\*：原始 A\* 算法（单向搜索+曼哈顿距离+二叉堆），作为对照组；
* AOpt1\*：在ABase\*的基础上加入改进启发式函数（调整权重系数）；
* AOpt2\*：在ABase\*的基础上加入双向搜索；
* AOpt3\*：在ABase\*的基础上加入并行计算；
* AOpt4\*：在AOpt1\*的基础上加入双向搜索；
* AOpt5\*：在AOpt2\*的基础上加入并行计算；
* AOpt6\*：在AOpt1\*的基础上加入并行计算；
* AFinal\*：综合所有改进。

**5.2 迷宫设计**

为模拟不同场景下的复杂度与规模差异，本文设计了三套迷宫（Maze1、Maze2、Maze3）。

Maze1（小规模）

大小：19\*19(不计外围边框)

起点：(0,0)

终点：(18,18)

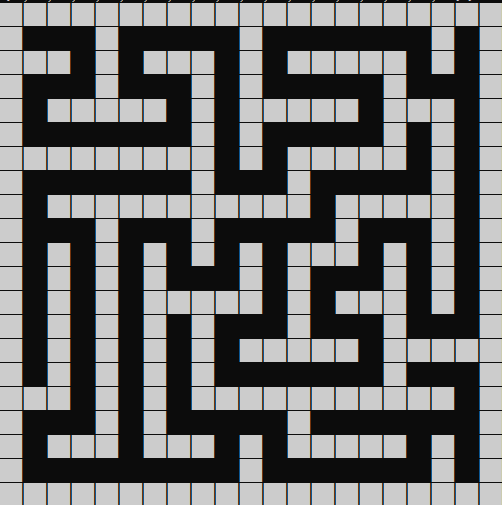


图1 Maze1迷宫地图

Maze2（中等规模）

大小：51\*51(不计外围边框)

起点：(0,0)

终点：(50,50)

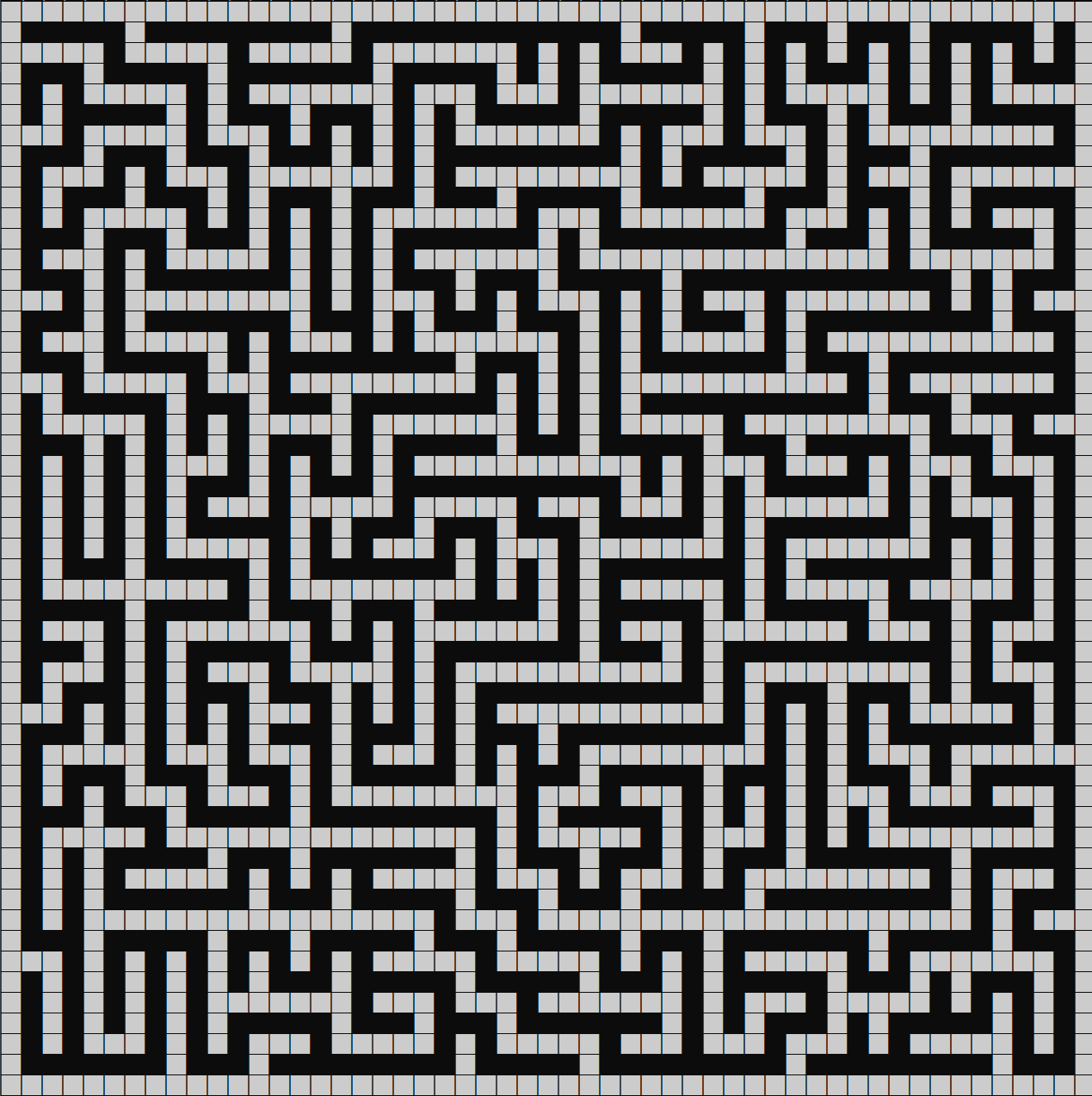


图2 Maze2迷宫地图

Maze2（大等规模）

大小：101\*101(不计外围边框)

起点：(0,0)

终点：(100,100)

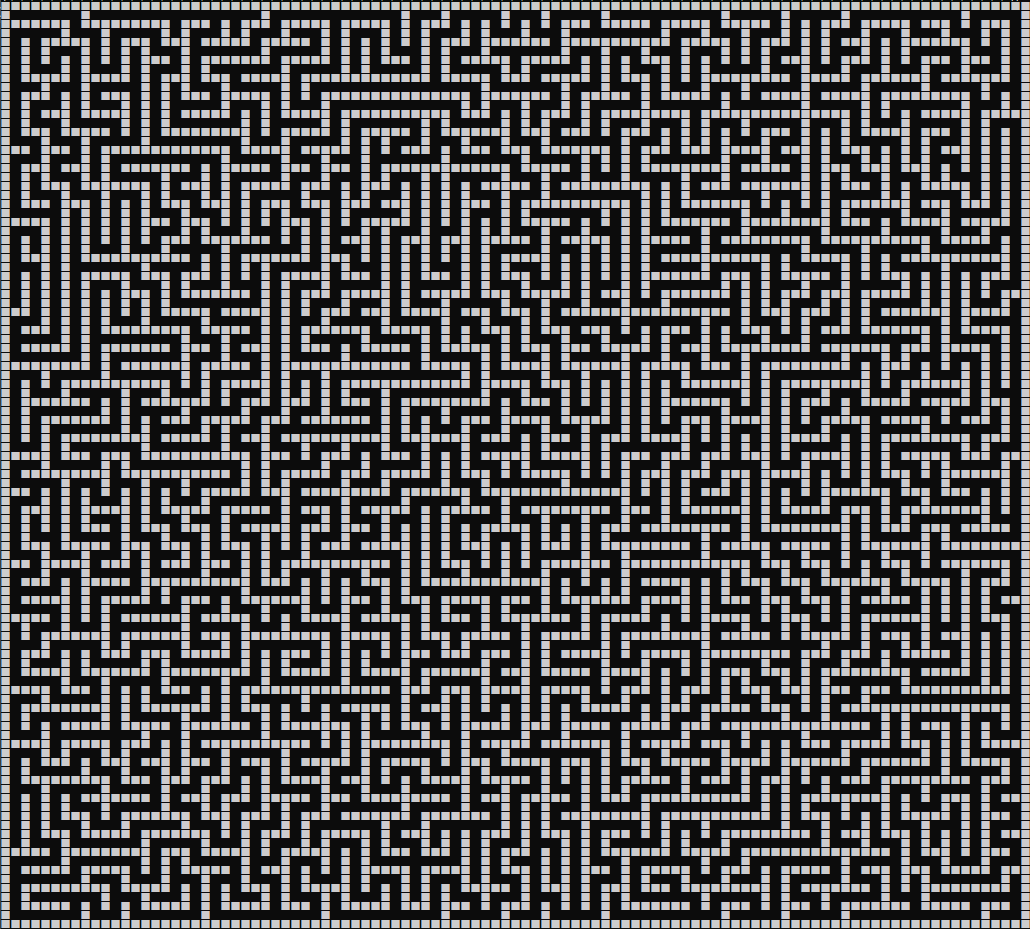


图3 Maze3迷宫地图

**5.3 相关说明**

为了简化程序编写过程，减少繁杂多余的逻辑，本程序在优化时，采用类封装的方式，将关键函数定义为虚函数，具体修改时，通过类的派生与函数重写，使每个类实现特定的优化功能，减少重复代码。

由于核心代码较多，将代码以附录的形式呈现。

**5.4 实验结果记录与分析**

1. 不同权重系数对比

表1 显示了在四种不同权重系数w（0.5、1、1.5、2）下，A\* 算法在三种规模迷宫 (Maze1、Maze2、Maze3) 中的运行时间（单位：纳秒 ns）。其中w=1被视为对照组（即传统 A\* 中 f(n)=g(n)+h(n的情况）。

|  |  |  |  |
| --- | --- | --- | --- |
| 权重系数 | Maze1 | Maze2 | Maze3 |
| W=1(对照) | 405200 | 405200 | 5796100 |
| W=0.5 | 190600 | 307200 | 4065800 |
| W=1.5 | 69500 | 116300 | 1775400 |
| W=2 | 73500 | 118700 | 2091600 |

表1 不同权重系数实验结果对比表(单位：ns)

从数据可以看出，w=0.5 和 w=1.5 相对于 w=1 都在三种迷宫中显著缩短了搜索时间，而 w=1.5 整体效果最为突出，例如 Maze3 从 5796100 ns 直接降到 1775400 ns，说明在大规模且障碍分布复杂的环境中，适当加大启发式比重能使搜索更快朝目标方向推进，从而减少冗余探索。不过，当 w 进一步增大到 2 时，Maze3 的运行时间反而比 w=1.5 略有上升 (2091600 ns)，这意味着过度依赖启发式会在某些情况下带来“误导”，必须再次回溯或修正，进而增加整体开销。对于小规模迷宫 (Maze1、Maze2)，不管权重往大或往小调整，都可能比 w=1 更好一点，但整体差异不会像 Maze3 那样显著。综合来看，w=1.5 是本实验中较为理想的单一权重选择，但实际应用仍需根据迷宫形态和障碍分布加以微调。

2.不同算法版本对比

表2展示了多个 A\* 优化版本在三种规模迷宫上的运行时间（单位：ns）。每个版本代表单个优化策略或不同优化策略的组合或变形。当存在权重系数优化时，w取1.5。

|  |  |  |  |
| --- | --- | --- | --- |
| 算法版本 | Maze1 | Maze2 | Maze3 |
| ABase\* | 590300 | 703600 | 10234200 |
| AOpt1\* | 389600 | 623300 | 7783800 |
| AOpt2\* | 356600 | 625700 | 4917200 |
| AOpt3\* | 104200 | 174800 | 3263400 |
| AOpt4\* | 147500 | 297700 | 2483800 |
| AOpt5\* | 385600 | 671000 | 2228000 |
| AOpt6\* | 112600 | 198800 | 3138700 |
| AFinal\* | 626900 | 831100 | 2744300 |

表2 不同算法实验结果记录表(单位：ns)

ABase\*作为对比基准，它的时间远高于大部分带有额外优化的版本，特别是在Maze3上体现明显。

AOpt1\* 体现了初步的优化策略，运行时间显著优于 ABase\*。

AOpt2\* 进一步缩短 Maze3 的时间到 4917200 ns，表明在大规模场景下，适当叠加“双向搜索”等方案起到关键作用。

AOpt3\* 在三种规模上都显示极高的效率，体现了并行加速的可行性。

AOpt4\* ~ AOpt6\*为任意两种策略的组合，虽然它们对于路径搜索效率都有了一定提升，但这些版本与其他版本相比性能表现并非单调提升。例如AOpt5\*在 Maze3 方面比 AOpt3\* 更快，但 在Maze1、Maze2 并未一定胜出。

最终版本在Maze1、Maze2迷宫路径搜索上表现甚至较一些中间版本更差，在Maze3上的表现也不及部分中间版本。

AOpt4\* ~ AOpt6\*与AFinal\*的表现说明，在进行算法优化时，一次结合多种优化策略可能会互相产生冲突，导致结果可能并不是最优，甚至产生负优化。

AOpt2\*、AOpt3\*分别在 Maze3 上取得了比较好的成绩，说明双向搜索、高效数据结构、改进启发式等方案对大规模复杂迷宫最具“破冰”作用。同时，在多个优化策略叠加在一起时，需要考虑算法整体的耦合度；若引入并行和启发式修正后，对优先队列操作频度变高，再加上双向搜索同时在两端进行节点管理，可能带来一系列同步或冗余开销，导致结果并不一定最佳。

**5.5 实验偏差与误差来源**

由于个人实验环境并不完美，每次运行结果可能产生不小偏差，以下是一些常见的误差来源及其简要说明：

1.硬件环境的瞬时变化

CPU 频率因温度或系统省电策略发生波动，带来执行速度的不同；同时，内存带宽或缓存争用可能因并发进程或后台任务而变动。

2. 操作系统调度

不同时间段，系统调度器会将本程序分配到不同核心或与其他任务抢占 CPU，导致性能结果不稳定，以及若电脑同时在运行其他占用较多资源的软件，对测量影响更显著。

3. 编译器选择

实验若在不同编译器或不同版本之间进行对比，也会产生编译优化差异。

为减小这类误差，通常应多次运行并取平均值，同时在同一环境下保持相似的负载条件，尽量排除外界对测量的干扰，并结合对算法逻辑本身的客观评估做出综合判断**。**

**5.6 结语**

在实际实验中，每一次运行都存在不小的偏差，有时，优化后的算法效率可能更低，不过在大多数情况下，优化可以保持其高效性。只是在多线程并行策略中，算法往往会出现优化失败甚至负优化话地情况，但是在每一次程序修改完后初次构建运行时，其有可以保持高效性，这或许是未来地一个探究方向之一。

在真实项目中，开发者应结合具体的地图规模、障碍分布、以及对实时性的要求来选择合适的启发式权重与优化策略，并通过重复测试评估加以验证。只有在“算法设计、实现细节、实验环境”三者都达成较好平衡时，才能在实践中获得最优或近似最优的寻路性能。

**6 结 论**

本研究在传统A\*算法的基础上，系统梳理了其在迷宫寻路中的适用性与潜在瓶颈，并从多维度给出了改进策略与对应的实验评估。通过在小规模、中等规模和大规模的三种迷宫环境下进行实验，可以明确看出，合理地调节启发式函数的权重系数能够有效平衡实际代价与预估代价的比重，从而在搜索过程中实现对目标方向的精准引导。在大规模迷宫中，由于路径更复杂且障碍物的分布相对密集，适度加大启发式比重往往可以带来更显著的性能提升，让算法在更短时间内收敛；然而，当权重过高时，搜索有时会被过于激进的启发式带往错误方向而产生回溯，导致一定程度的性能回退。对比不同优化策略的若干版本 A\* 算法也可发现，单项优化在特定场景可能收效良好，但多重优化手段的简单叠加并不必然带来性能的线性提升，甚至在小规模或障碍分布不均的迷宫内还会因为数据结构、锁冲突或实现复杂度而出现性能倒退。由此可以推断，优化策略的适配度与迷宫本身的特征关联极大，需要对目标场景的规模、障碍布局以及实时性需求做充分评估后，再将双向搜索、多线程并行、剪枝、改进启发式等模块灵活组合，以取得最好效果。

在更广泛的应用视角下，由于实际项目中可能会遇到动态障碍、多维度权重、甚至是实时性极高的要求，A\* 算法及其改进版本都依赖于更高水平的工程实现和更精细的调参。若考虑将机器学习方法融入寻路过程，使算法在执行过程中对启发式函数进行自适应或分段加权的优化，则有可能进一步降低搜索过程中的冗余节点扩展量，从而在大规模、复杂度较高的迷宫环境中获得更加显著的效率提升。

同时，实验结果也显示本人电脑实验环境尚不具备完全理想的硬件与软件条件，导致单次运行结果存在一定程度的波动。针对这些问题，除了采取多次测试并取平均值的方式，还应通过更深入的剖析对锁竞争或数据结构在不同阶段的访问频度进行量化研究，并根据地图本身的特征和硬件条件来选择最合适的并发深度、启发式设计和数据结构管理方式。

未来的研究方向或许可以集中在三个层面。第一，面向动态或非确定性迷宫环境的改进，充分考虑迷宫在障碍分布和地图形变上的变化规律，让算法能够在路径发生突变时迅速作出局部调整而不必重新全局搜索。第二，加强对迷宫大数据特征的分析与利用，通过对历史搜索记录进行收集或对地形特点做预处理，设计自适应或统计型启发式函数，从而让算法在不同区域采用差异化策略。第三，结合更专业的性能分析工具，针对不同算法版本可能出现的偏差进行系统排查，详细记录虚函数调用、内存分配与回收、多线程调度、编译器优化等关键要素的开销，并利用更精细的度量手段评估剪枝、双向搜索、并行化带来的实际收益，力求在大规模测试中减少系统性干扰，得到更稳定、具备可推广价值的结论。

A\*依然是迷宫寻路乃至通用路径规划的重要基石，将其与多种优化策略、机器学习方法和高效工程实现相结合，仍有很大的研究与应用空间。

**参考文献**

1. 冉东可, 彭富伦, 李红光. 基于A\*算法的路径规划研究综述. 电子技术与软件工程, 2020(24): 11-12.
2. 秦锋,吴健,张学锋,赵晶丽.基于A\*的双向预处理改进搜索算法.计算机系统应用,2019,28(5):95-101.

**附录 核心代码**

#include <queue>  
#include <vector>  
#include <iostream>  
#include <cmath>  
#include <algorithm>  
#include <limits>  
struct Node {  
 int x, y;   
 double g, h;   
 Node \*parent;   
 Node(int x\_, int y\_, double g\_ = 0.0, double h\_ = 0.0, Node \*parent\_ = nullptr)  
 : x(x\_), y(y\_), g(g\_), h(h\_), parent(parent\_) {}  
 double f() const { return g + h; }  
};  
struct NodeCompare {  
 bool operator()(const Node \*a, const Node \*b) const  
 {  
 return a->f() > b->f();  
 }  
};  
**class ABaseStar**{  
 public:  
 ABaseStar(){};  
 ~ABaseStar(){}  
 virtual double heuristic(int x1, int y1, int x2, int y2)  
 {  
 return std::abs(x1 - x2) + std::abs(y1 - y2);  
 }  
 virtual std::vector<std::pair<int, int> > getNeighborCoords(int x, int y,const std::vector<std::vector<int> > &grid)  
 {  
 static const std::vector<std::pair<int, int> > directions = {  
 {0, 1}, {1, 0}, {0, -1}, {-1, 0}  
 };  
 std::vector<std::pair<int, int> > neighbors;  
 for (auto &dir: directions) {  
 int nx = x + dir.first;  
 int ny = y + dir.second;  
 if (nx >= 0 && nx < (int) grid.size() &&  
 ny >= 0 && ny < (int) grid[0].size() &&  
 grid[nx][ny] == 1) {  
 neighbors.emplace\_back(nx, ny);  
 }  
 }  
 return neighbors;  
 }  
 virtual std::vector<Node \*> reconstructPath(Node \*goal)  
 {  
 std::vector<Node \*> path;  
 Node \*current = goal;  
 while (current) {  
 path.push\_back(current);  
 current = current->parent;  
 }  
 std::reverse(path.begin(), path.end());  
 return path;  
 }  
 virtual std::vector<Node \*> aStarSearch(const Node &start,  
 const Node &goal,  
 const std::vector<std::vector<int> > &grid)  
 {  
 int rows = grid.size();  
 int cols = grid[0].size();  
 std::vector<std::vector<double> > gScore(  
 rows, std::vector<double>(cols, std::numeric\_limits<double>::*max*()));  
 std::vector<std::vector<bool> > closedSet(rows, std::vector<bool>(cols, false));  
 std::priority\_queue<Node \*, std::vector<Node \*>, NodeCompare> openSet;  
 Node \*startNode = new Node(start.x, start.y, 0.0,  
 heuristic(start.x, start.y, goal.x, goal.y));  
 gScore[start.x][start.y] = 0.0;  
 openSet.push(startNode);  
  
 while (!openSet.empty()) {  
 Node \*current = openSet.top();  
 openSet.pop();  
 if (closedSet[current->x][current->y]) {  
 delete current;  
 continue;  
 }  
 closedSet[current->x][current->y] = true;  
 if (current->x == goal.x && current->y == goal.y) {  
 auto path = reconstructPath(current);  
 while (!openSet.empty()) {  
 Node \*n = openSet.top();  
 openSet.pop();  
 delete n;  
 }  
 return path;  
 }  
 auto neighborCoords = getNeighborCoords(current->x, current->y, grid);  
 for (auto [nx, ny]: neighborCoords) {  
 if (closedSet[nx][ny]) {  
 continue;  
 }  
 double tentative\_g = gScore[current->x][current->y] + 1.0;   
 if (tentative\_g < gScore[nx][ny]) {  
 gScore[nx][ny] = tentative\_g;  
 double hVal = heuristic(nx, ny, goal.x, goal.y);  
 Node \*neighborNode = new Node(nx, ny, tentative\_g, hVal, current);  
 openSet.push(neighborNode);  
 }  
 }  
 }  
 return {};  
 }  
  
};

**class AOpt1Star : public ABaseStar** {  
public:  
 virtual double heuristic(int x1, int y1, int x2, int y2) override  
 {  
 return 1.5\* (std::abs(x1 - x2) + std::abs(y1 - y2));  
 }  
};

**class AOpt2Star : public ABaseStar**  
{  
 public:  
 virtual std::vector<Node\*> reconstructPathBidirectional(Node\* intersectionFwd, Node\* intersectionBwd) {  
 std::vector<Node\*> pathFwd;  
 {  
 Node\* curr = intersectionFwd;  
 while (curr) {  
 pathFwd.push\_back(curr);  
 curr = curr->parent;  
 }  
 std::reverse(pathFwd.begin(), pathFwd.end());  
 }  
 std::vector<Node\*> pathBwd;  
 {  
 Node\* curr = intersectionBwd;  
 while (curr) {  
 pathBwd.push\_back(curr);  
 curr = curr->parent;  
 }  
 }  
 if (!pathBwd.empty()) {  
 pathBwd.erase(pathBwd.begin());  
 }  
 std::reverse(pathBwd.begin(), pathBwd.end());  
 pathFwd.insert(pathFwd.end(), pathBwd.begin(), pathBwd.end());  
 return pathFwd;  
 }  
 virtual std::vector<Node\*> aStarSearch (  
 const Node& start,  
 const Node& goal,  
 const std::vector<std::vector<int>>& grid) override  
{  
 int rows = (int)grid.size();  
 int cols = (int)grid[0].size();  
 std::vector<std::vector<int>> gScoreFwd(rows, std::vector<int>(cols, std::numeric\_limits<int>::*max*()));  
 std::vector<std::vector<int>> gScoreBwd(rows, std::vector<int>(cols, std::numeric\_limits<int>::*max*()));  
 std::vector<std::vector<bool>> closedSetFwd(rows, std::vector<bool>(cols, false));  
 std::vector<std::vector<bool>> closedSetBwd(rows, std::vector<bool>(cols, false));t  
 std::priority\_queue<Node\*, std::vector<Node\*>, NodeCompare> openSetFwd;  
 std::priority\_queue<Node\*, std::vector<Node\*>, NodeCompare> openSetBwd;  
 Node\* startNode = new Node(start.x, start.y, 0, heuristic(start.x, start.y, goal.x, goal.y));  
 Node\* goalNode = new Node(goal.x, goal.y, 0, heuristic(goal.x, goal.y, start.x, start.y));  
 gScoreFwd[start.x][start.y] = 0;  
 gScoreBwd[goal.x][goal.y] = 0;  
 openSetFwd.push(startNode);  
 openSetBwd.push(goalNode);  
 std::vector<std::vector<Node\*>> cameFromFwd(rows, std::vector<Node\*>(cols, nullptr));  
 std::vector<std::vector<Node\*>> cameFromBwd(rows, std::vector<Node\*>(cols, nullptr));  
 cameFromFwd[start.x][start.y] = startNode;  
 cameFromBwd[goal.x][goal.y] = goalNode;  
 while (!openSetFwd.empty() && !openSetBwd.empty()) {  
 Node\* currentFwd = openSetFwd.top();  
 openSetFwd.pop();  
 if (closedSetFwd[currentFwd->x][currentFwd->y]) {  
 delete currentFwd;  
 } else {  
 closedSetFwd[currentFwd->x][currentFwd->y] = true;  
 if (closedSetBwd[currentFwd->x][currentFwd->y]) {  
 Node\* meetBwd = cameFromBwd[currentFwd->x][currentFwd->y];  
 auto fullPath = reconstructPathBidirectional(currentFwd, meetBwd);  
 return fullPath;  
 }  
 for (auto [nx, ny] : getNeighborCoords(currentFwd->x, currentFwd->y, grid)) {  
 if (closedSetFwd[nx][ny])  
 continue;  
 int tentative\_g = gScoreFwd[currentFwd->x][currentFwd->y] + 1;  
 if (tentative\_g < gScoreFwd[nx][ny]) {  
 gScoreFwd[nx][ny] = tentative\_g;  
 double hVal = heuristic(nx, ny, goal.x, goal.y);  
 Node\* neighborNode = new Node(nx, ny, tentative\_g, hVal, currentFwd);  
 cameFromFwd[nx][ny] = neighborNode;  
 openSetFwd.push(neighborNode);  
 }  
 }  
 }  
 Node\* currentBwd = openSetBwd.top();  
 openSetBwd.pop();  
 if (closedSetBwd[currentBwd->x][currentBwd->y]) {  
 delete currentBwd;  
 } else {  
 closedSetBwd[currentBwd->x][currentBwd->y] = true;  
 if (closedSetFwd[currentBwd->x][currentBwd->y]) {  
 Node\* meetFwd = cameFromFwd[currentBwd->x][currentBwd->y];  
 auto fullPath = reconstructPathBidirectional(meetFwd, currentBwd);  
 return fullPath;  
 }  
 for (auto [nx, ny] : getNeighborCoords(currentBwd->x, currentBwd->y, grid)) {  
 if (closedSetBwd[nx][ny]) continue;  
 int tentative\_g = gScoreBwd[currentBwd->x][currentBwd->y] + 1;  
 if (tentative\_g < gScoreBwd[nx][ny]) {  
 gScoreBwd[nx][ny] = tentative\_g;  
 double hVal = heuristic(nx, ny, start.x, start.y);  
 Node\* neighborNode = new Node(nx, ny, tentative\_g, hVal, currentBwd);  
 cameFromBwd[nx][ny] = neighborNode;  
 openSetBwd.push(neighborNode);  
 }  
 }  
 }  
 }  
 return {};  
}  
};

*#include <omp.h>*

**class AOpt3Star : public ABaseStar**{  
public:  
 virtual std::vector<Node\*> aStarSearch(const Node& start,  
 const Node& goal,  
 const std::vector<std::vector<int>>& grid) override  
{  
 int rows = grid.size();  
 int cols = grid[0].size();  
 std::vector<std::vector<double>> gScore(rows, std::vector<double>(cols, std::numeric\_limits<double>::*max*()));  
 std::vector<std::vector<bool>> closedSet(rows, std::vector<bool>(cols, false));  
 std::priority\_queue<Node\*, std::vector<Node\*>, NodeCompare> openSet;  
 Node\* startNode = new Node(start.x, start.y, 0.0, heuristic(start.x, start.y, goal.x, goal.y), nullptr);  
 gScore[start.x][start.y] = 0.0;  
 openSet.push(startNode);  
 while (!openSet.empty()) {  
 Node\* current = openSet.top();  
 openSet.pop();  
 if (closedSet[current->x][current->y]) {  
 delete current; // 释放这块内存  
 continue;  
 }  
 closedSet[current->x][current->y] = true;  
 if (current->x == goal.x && current->y == goal.y) {  
 std::vector<Node\*> path = reconstructPath(current);  
 return path;  
 }  
 auto neighborCoords = getNeighborCoords(current->x, current->y, grid);  
 std::vector<Node\*> newNodes;  
 newNodes.reserve(neighborCoords.size());  
 #pragma omp parallel  
 {  
 std::vector<Node\*> localNodes;  
 localNodes.reserve(neighborCoords.size());  
 #pragma omp for nowait  
 for (int i = 0; i < (int)neighborCoords.size(); i++) {  
 int nx = neighborCoords[i].first;  
 int ny = neighborCoords[i].second;  
 if (closedSet[nx][ny]) {  
 continue;  
 }  
 double tentative\_g = gScore[current->x][current->y] + 1.0;  
 if (tentative\_g < gScore[nx][ny]) {  
 Node\* neighborNode = new Node(nx, ny, tentative\_g,  
 heuristic(nx, ny, goal.x, goal.y),  
 current);  
 localNodes.push\_back(neighborNode);  
 }  
 }  
 #pragma omp critical  
 {  
 newNodes.insert(newNodes.end(), localNodes.begin(), localNodes.end());  
 }  
 }  
 for (auto n : newNodes) {  
 int nx = n->x;  
 int ny = n->y;  
 double tentative\_g = n->g;  
 if (tentative\_g < gScore[nx][ny]) {  
 gScore[nx][ny] = tentative\_g;  
 openSet.push(n);  
 } else {  
 delete n;   
 }  
 }  
 }  
 return {};  
}  
 };

**class AOpt4Star : public AOpt2Star** {  
 virtual double heuristic(int x1, int y1, int x2, int y2) override  
 {  
 return 1.5 \* (std::abs(x1 - x2) + std::abs(y1 - y2));  
 }  
};

**class AOpt5Star : public AOpt2Star** {  
public:  
 void forwardSearchThreadFunc(  
 const std::vector<std::vector<int> > &grid,  
 const Node &start,  
 const Node &goal,  
 std::vector<std::vector<double> > &gScoreFwd,  
 std::vector<std::vector<bool> > &closedFwd,  
 std::vector<std::vector<Node \*> > &cameFromFwd,  
 std::priority\_queue<Node \*, std::vector<Node \*>, NodeCompare> &openSetFwd,  
 std::mutex &mtxFwd,  
 std::atomic<bool> &found,  
 Node \*&meetFwd,  
 Node \*&meetBwd,  
 std::vector<std::vector<bool> > &closedBwd,  
 std::vector<std::vector<Node \*> > &cameFromBwd,  
 std::mutex &mtxBwd  
 )  
 {  
 Node \*startNode = new Node(start.x, start.y, 0.0,  
 heuristic(start.x, start.y, goal.x, goal.y)); {  
 std::lock\_guard<std::mutex> lock(mtxFwd);  
 gScoreFwd[start.x][start.y] = 0.0;  
 cameFromFwd[start.x][start.y] = startNode;  
 openSetFwd.push(startNode);  
 }  
 while (!found) {  
 Node \*current = nullptr;  
 {  
 std::lock\_guard<std::mutex> lock(mtxFwd);  
 if (openSetFwd.empty()) {  
 break;   
 }  
 current = openSetFwd.top();  
 openSetFwd.pop();  
 }  
 if (closedFwd[current->x][current->y]) {  
 delete current;  
 continue;  
 }  
 closedFwd[current->x][current->y] = true;  
 {  
 std::lock\_guard<std::mutex> lockBwd(mtxBwd);  
 if (closedBwd[current->x][current->y]) {  
 meetFwd = current;  
 meetBwd = cameFromBwd[current->x][current->y];  
 found = true;  
 break;  
 }  
 }  
 auto neighbors = getNeighborCoords(current->x, current->y, grid);  
 for (auto &nb: neighbors) {  
 int nx = nb.first;  
 int ny = nb.second;  
 if (closedFwd[nx][ny]) {  
 continue;  
 }  
 double tentative\_g = gScoreFwd[current->x][current->y] + 1.0;  
 double old\_g = gScoreFwd[nx][ny];  
 if (tentative\_g < old\_g) {  
 Node \*neighborNode = new Node(nx, ny, tentative\_g,  
 heuristic(nx, ny, goal.x, goal.y), current); {  
 std::lock\_guard<std::mutex> lock(mtxFwd);  
 gScoreFwd[nx][ny] = tentative\_g;  
 cameFromFwd[nx][ny] = neighborNode;  
 openSetFwd.push(neighborNode);  
 }  
 }  
 }  
 }  
 }  
 void backwardSearchThreadFunc(  
 const std::vector<std::vector<int> > &grid,  
 const Node &start,   
 const Node &goal,   
 std::vector<std::vector<double> > &gScoreBwd,  
 std::vector<std::vector<bool> > &closedBwd,  
 std::vector<std::vector<Node \*> > &cameFromBwd,  
 std::priority\_queue<Node \*, std::vector<Node \*>, NodeCompare> &openSetBwd,  
 std::mutex &mtxBwd,  
 std::atomic<bool> &found,  
 Node \*&meetFwd,  
 Node \*&meetBwd,  
 std::vector<std::vector<bool> > &closedFwd,  
 std::vector<std::vector<Node \*> > &cameFromFwd,  
 std::mutex &mtxFwd  
 )  
 {  
 Node \*goalNode = new Node(start.x, start.y, 0.0,  
 heuristic(start.x, start.y, goal.x, goal.y)); {  
 std::lock\_guard<std::mutex> lock(mtxBwd);  
 gScoreBwd[start.x][start.y] = 0.0;  
 cameFromBwd[start.x][start.y] = goalNode;  
 openSetBwd.push(goalNode);  
 }  
 while (!found) {  
 Node \*current = nullptr; {  
 std::lock\_guard<std::mutex> lock(mtxBwd);  
 if (openSetBwd.empty()) {  
 break;  
 }  
 current = openSetBwd.top();  
 openSetBwd.pop();  
 }  
 if (closedBwd[current->x][current->y]) {  
 delete current;  
 continue;  
 }  
 closedBwd[current->x][current->y] = true;  
 {  
 std::lock\_guard<std::mutex> lockFwd(mtxFwd);  
 if (closedFwd[current->x][current->y]) {  
 meetBwd = current;  
 meetFwd = cameFromFwd[current->x][current->y];  
 found = true;  
 break;  
 }  
 }  
 auto neighbors = getNeighborCoords(current->x, current->y, grid);  
 for (auto &nb: neighbors) {  
 int nx = nb.first;  
 int ny = nb.second;  
 if (closedBwd[nx][ny]) {  
 continue;  
 }  
 double tentative\_g = gScoreBwd[current->x][current->y] + 1.0;  
 double old\_g = gScoreBwd[nx][ny];  
 if (tentative\_g < old\_g) {  
 Node \*neighborNode = new Node(nx, ny, tentative\_g,  
 heuristic(nx, ny, goal.x, goal.y), current); {  
 std::lock\_guard<std::mutex> lock(mtxBwd);  
 gScoreBwd[nx][ny] = tentative\_g;  
 cameFromBwd[nx][ny] = neighborNode;  
 openSetBwd.push(neighborNode);  
 }  
 }  
 }  
 }  
 }  
 virtual std::vector<Node \*> aStarSearch(  
 const Node &start,  
 const Node &goal,  
 const std::vector<std::vector<int> > &grid) override  
 {  
 int rows = (int) grid.size();  
 int cols = (int) grid[0].size();  
 std::vector<std::vector<double> > gScoreFwd(  
 rows, std::vector<double>(cols, std::numeric\_limits<double>::infinity()));  
 std::vector<std::vector<bool> > closedFwd(rows, std::vector<bool>(cols, false));  
 std::vector<std::vector<Node \*> > cameFromFwd(rows, std::vector<Node \*>(cols, nullptr));  
 std::priority\_queue<Node \*, std::vector<Node \*>, NodeCompare> openSetFwd;  
 std::mutex mtxFwd;  
 std::vector<std::vector<double> > gScoreBwd(  
 rows, std::vector<double>(cols, std::numeric\_limits<double>::infinity()));  
 std::vector<std::vector<bool> > closedBwd(rows, std::vector<bool>(cols, false));  
 std::vector<std::vector<Node \*> > cameFromBwd(rows, std::vector<Node \*>(cols, nullptr));  
 std::priority\_queue<Node \*, std::vector<Node \*>, NodeCompare> openSetBwd;  
 std::mutex mtxBwd;  
 std::atomic<bool> found(false);  
 Node \*meetFwd = nullptr;  
 Node \*meetBwd = nullptr;  
 std::thread tFwd([this, &grid, &start, &goal, &gScoreFwd, &closedFwd, &cameFromFwd,  
 &openSetFwd, &mtxFwd, &found, &meetFwd, &meetBwd, &closedBwd, &cameFromBwd, &mtxBwd] {  
 this->forwardSearchThreadFunc(  
 grid,  
 start,  
 goal,  
 gScoreFwd,  
 closedFwd,  
 cameFromFwd,  
 openSetFwd,  
 mtxFwd,  
 found,  
 meetFwd,  
 meetBwd,  
 closedBwd,  
 cameFromBwd,  
 mtxBwd  
 );  
 });  
 std::thread tBwd([this, &grid, &start, &goal, &gScoreBwd, &closedBwd, &cameFromBwd,  
 &openSetBwd, &mtxBwd, &found, &meetFwd, &meetBwd, &closedFwd, &cameFromFwd, &mtxFwd] {  
 this->backwardSearchThreadFunc(  
 grid,  
 goal,   
 start,   
 gScoreBwd,  
 closedBwd,  
 cameFromBwd,  
 openSetBwd,  
 mtxBwd,  
 found,  
 meetFwd,  
 meetBwd,  
 closedFwd,  
 cameFromFwd,  
 mtxFwd);  
 });  
 tFwd.join();  
 tBwd.join();  
 if (found && meetFwd && meetBwd) {  
 return reconstructPathBidirectional(meetFwd, meetBwd);  
 }  
 return {};  
 }  
};

**class AOpt6Star : public AOpt3Star**{  
 public:  
 virtual double heuristic(int x1, int y1, int x2, int y2) override  
 {  
 return 1.5 \* (std::abs(x1 - x2) + std::abs(y1 - y2));  
 }  
 };

**class AFinalStar : public AOpt5Star**{  
 public:  
 virtual double heuristic(int x1, int y1, int x2, int y2) override  
 {  
 return 1.5 \* (std::abs(x1 - x2) + std::abs(y1 - y2));  
 }  
};