## 算法分析与设计期中大作业

——经典与现代寻路算法的调查报告

1. **研究背景**

在机器人技术、自动驾驶汽车等领域，寻路算法扮演着基础且核心的角色。随着现实世界环境的日益复杂化，经典的的寻路策略已变得不适应，亟需更先进的算法解决方案。即便是像A\*这样的经典搜索算法，也面临着在某些场景下无法提供最优化解决方案的挑战。近年来随着计算机技术的飞速发展，寻路算法也不断更新迭代，随着寻路中使用当行网格的普及，A\*算法在寻路实现中所占比例非常小。

本研究报告旨在通过迷宫问题这一经典场景，设计并实现一系列相关算法，以深入探讨和分析传统寻路算法的精髓，并梳理近年来涌现的创新性改进算法。在本报告中，我将融合并应用在课程中学习的动态规划、回溯等算法概念，对各类算法的时间复杂度和可行性进行细致的分析和评估。期望能够在这一过程中理解并掌握算法设计的基础知识，洞察不同寻路算法的优势与局限。此外，我还将尝试提出自己的独到见解，提出自己的理解与思考。

1. **寻路算法原理**

一般来说寻路可能包含了两个概念，一个是寻路，一个是轨迹规划。

寻路单指path planning，主要是找到一条可行路径，不考虑具体对象是以何种速度或者状态通过的，多数时候有路径即可，轨迹一般使用的都比较简单。

轨迹规划指trajectory planning，主要描述的是如何通过一提哦啊路径，其输出的是每个时刻该对象的各种状态，对于2维小车寻路来说有他的水平xy的速度，加速度，方向；对于3维无人机，输出的除了方向速度以外可能还有无人机本身的姿态要求。

这里主要介绍Path Planning的相关内容。而寻路算法又可分为启发式和盲目式，盲目式为不知道终点位置，只能慢慢尝试，启发式为知道终点位置，朝着终点位置不断尝试。

* 1. **经典寻路算法**
     1. **穷举法寻路**

穷举法又称为枚举法或者蛮力法，是一种简单直接解决问题的方法，常常是基于问题的直接描述去编写程序，基本思路就是暴力破解，比如说求可行路径，那么就在当前位置使用for循环依次遍历四个方向，直到可以找到可行解。

**缺点：**操作比较费时，时间复杂度高，往往是指数级，也往往是盲目式的。

**优点：**算法相对简洁，易于理解。可以解决许多计算领域的问题（只要机器性能足够或时间开销可承受），并且应用十分广泛。

**优化：**对于穷举法自身的优化，一般只能减少其执行的系数，但是数量级不会改变。或可搭配剪枝及回溯算法思想进行优化。

穷举思想在深度优先搜索（DFS）和广度优先搜索（BFS）中均有体现，可以利用二者来分析穷举思想，这两者都是盲目式寻路方式。

1. **DFS**

DFS简单理解就是一条路走到黑，也就是在深度方向的穷举。DFS维护的是一个栈，满足“LIFO”规则，总是访问容器中最后进入的元素。

* 1. 先判断是否到达目标位置，如果到达目标位置，再试探有无其他更短的路径。
  2. 如果没有到达目标位置，则找到下一步可以到达的位置，直到找到目标位置。

1. **BFS**

BFS简单理解就是层次遍历，也就是在广度方向的穷举。BFS维护的是一个队列，满足“FIFO”规则，总是访问容器中最先进入的元素。广度优先搜索的优点是找出的第一条路径就是最短路径，所以经常用来搜索最短路径。

关于BFS的优化：BFS通常是一种盲目搜索，也就是非启发式的，不一定是最优结果，在下面的贪心求解中，可以为BFS引入贪心策略，让其每一步都选择当前最优选择。

* 1. 从入口元素开始，判断上下左右的临边元素是否满足条件，如果满足条件就入队列
  2. 取队首元素并出队列。寻找其相邻未被访问的元素，将其如队列并标记元素的前驱节点为队首元素。
  3. 重复步骤②，直到队列为空（没有找到可行路径）或者找到了终点。最后从终点开始，根据节点的前驱节点找出一条最短的可行路径。
     1. **分治思想寻路**

分治法的思想即为分而治之，将大问题分解为许多可解决的小问题。分治法寻路的基本流程是将大迷宫分割成若干个不相交的小迷宫；在每个小迷宫中独立地应用其他寻路算法（如BFS、A\*、Dijkstra等）以找到局部最短路径；合并局部解，通过在子图边界处的路径选择和合并，形成整个大迷宫的最短路径。

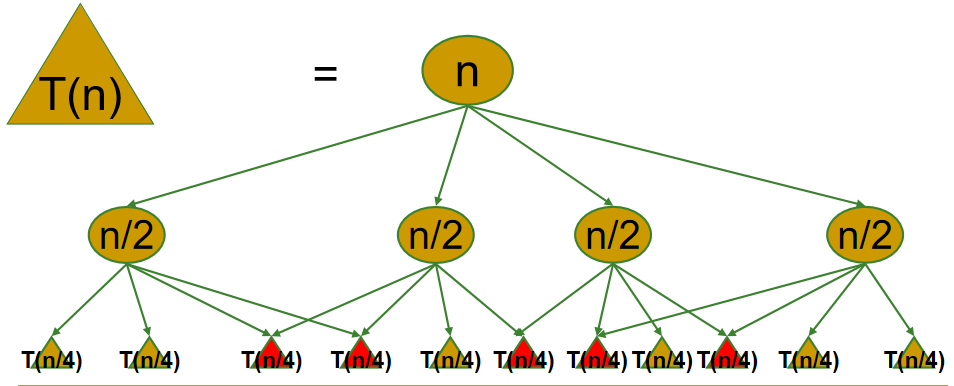
**缺点：**迷宫的合理划分比较复杂，尤其是当迷宫结构不规则时；递归地应用寻路算法可能导致较大的计算开销；局部最短路径的合并可能复杂，需要精心设计以确保全局最优解；额外的空间开销也大。最大难点在于迷宫问题如何合理选择每个小块的终点和起点。

**优点：**通过局部寻路减少了计算量，尤其是在迷宫非常大时；各个小迷宫的寻路可以并行进行，提高了算法的效率。

**优化：**采用启发式方法进行迷宫的智能划分，以减少不必要的计算；限制递归的深度，避免过多的递归调用；根据子迷宫的特点选择最合适的寻路算法，如在小区域使用BFS，在较大区域使用A\*或Dijkstra；或者也可以结合回溯等思想，作为流程的一部分。

* + 1. **动态规划寻路**

动态规划算法与分治法类似，其基本思想也是将待求问题分解成若干个子问题，但是不同于分治的将地图分割，各个子问题之间有着联系，经分解的子问题往往不是相互独立的，不同子问题的数目常常只有多项式量级，如果使用分治法求解，有些子问题被重复计算了很多次。如果能够保存已解决的子问题的答案，而在需要时再找出已经求得的答案，就可以避免大量重复计算，从而得到多项式时间的算法。



动态规划算法的基本要素：

1. 最优子结构性质
2. 重叠子问题性质

动态规划解决问题的关键点：

1. 状态定义：定义子问题，如何表示目标规模的问题和更小规模的问题。例如常见的方法：定义状态 dp[n]，表示规模为 n 的问题的解，dp[n - 1] 就表示规模为 n−1 的子问题的解。在实战中 dp[n] 的具体含义需要首先整理清楚再往下做。
2. 状态的转移：子问题之间的关系，例如定义好状态 dp[n]，此时子问题是 dp[n-1] 等，并且大规模的问题的解依赖小规模问题的解，此时需要知道怎样通过小规模问题的解推出大规模问题的解。这一步就是列状态转移方程的过程。一般的状态转移方程可以写成如下形式
3. 初始化
4. 边界条件

# 初始化 base case

dp[0][0][...] = base

# 进行状态转移

for 状态1 in 状态1的所有取值：

for 状态2 in 状态2的所有取值：

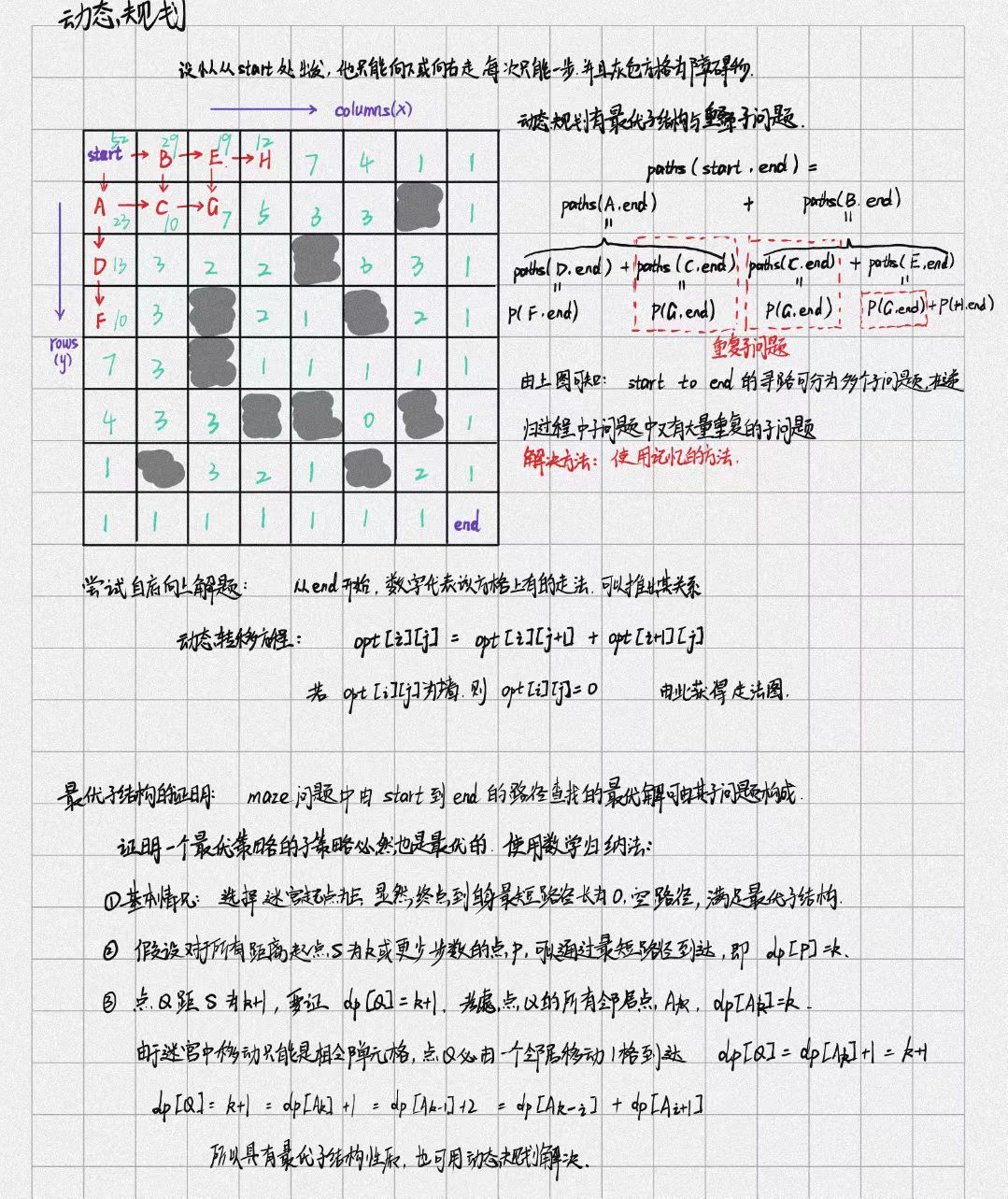
for ...

dp[状态1][状态2][...] = 求最值(选择1，选择2...)

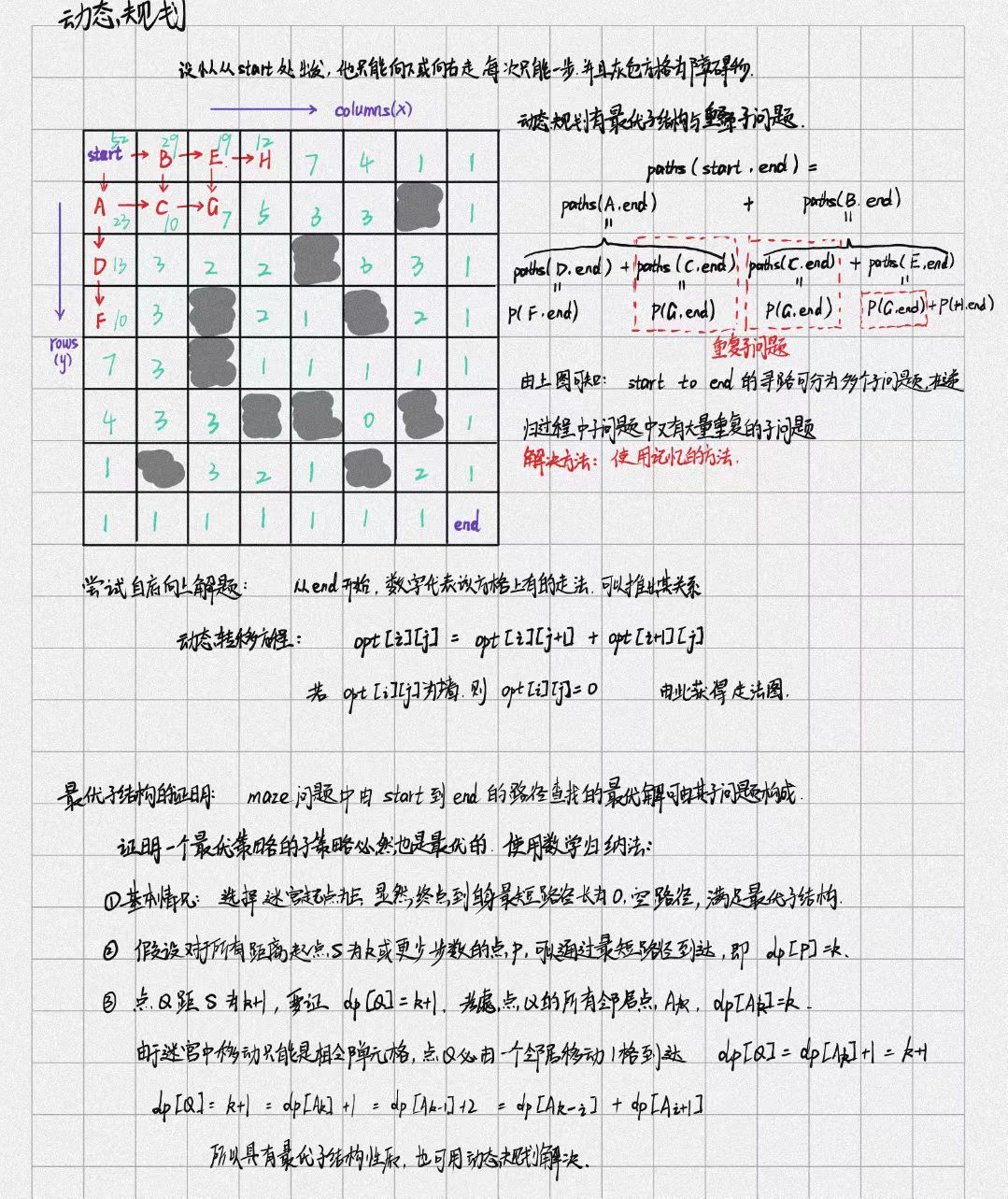
针对有权图，可以使用Floyd算法，又称为插点法，是一种利用动态规划的思想寻找给定的加权图中多源点之间最短路径的算法，与Dijkstra算法类似。算法的主要思想是动态规划（DP），而求最短路径需要不断松弛。

针对无权图，可以使用动态规划思想，具体实现依靠DFS或BFS深度优先搜索。

**动态规划分析：**

****

**最优子结构证明：**

****

* + 1. **贪心法寻路**

贪心算法和动态规划都常用于解决优化问题。它们之间存在一些相似之处，比如都依赖最优子结构性质，但工作原理不同。贪心算法总是作出在当前看来最好的选择，并不从整体最优考虑，所作出的选择只是在某种意义上的局部最优选择。虽然贪心算法不能对所有问题都得到整体最优解，但对许

多问题它能产生整体最优解。如单源最短路经问题，最小生成树问题等。在一些情况下，即使贪心算法不能得到整体最优解，其最终结果却是最优解的很好近似。

贪心算法的求解过程是多步判断的过程，最终的判断序列对应问题的最优解，贪心算法需要正确性证明，可以使用举反例的方法。

**缺点：**依据某种短视的贪心选择性质判断，性质的好坏决定算法的成败；

**优点：**贪心算法适用于组合优化问题，算法简单，时空复杂度低

**优化：**贪心算法非常经典，如果要保证贪心算法可以找到最优解，需要判断问题是否具有贪心选择性质，只有当局部最优选择始终可以导致全局最优解时，贪心算法才能保证得到最优解；是否满足最优子结构，原问题的最优解要包含子问题的最优解。

贪心思想在多种寻路算法中都有所体现，比方可以和BFS结合的Greedy BFS，以及Dijkstra，甚至A\*，以及基于A\*进行优化的多种算法。

**贪心算法解题步骤：**

1. 问题分析：梳理与理解问题特性，包括状态定义、优化目标和约束条件等。这一步在回溯和动态规划中都有涉及。
2. 确定贪心策略：确定如何在每一步中做出贪心选择。这个策略能够在每一步减小问题的规模，并最终解决整个问题。
3. 正确性证明：通常需要证明问题具有贪心选择性质和最优子结构。这个步骤可能需要用到数学证明，例如归纳法或反证法等。

**贪心算法在寻路中的应用：**

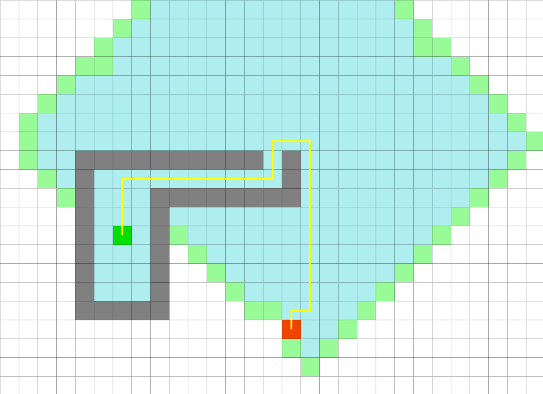
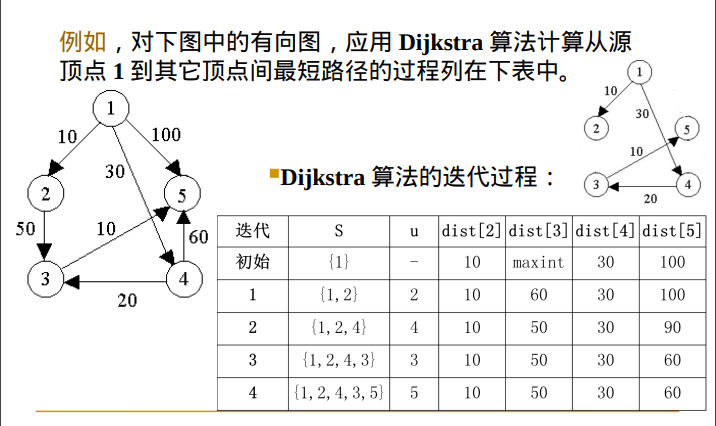
实际上，贪心算法在寻路中有着很广泛的应用，经典的Dijkstra算法和A\*算法都运用了贪心策略。在这里我也将以Dijkstra算法为例，给出贪心策略的正确性证明。

1. **Dijkstra算法**

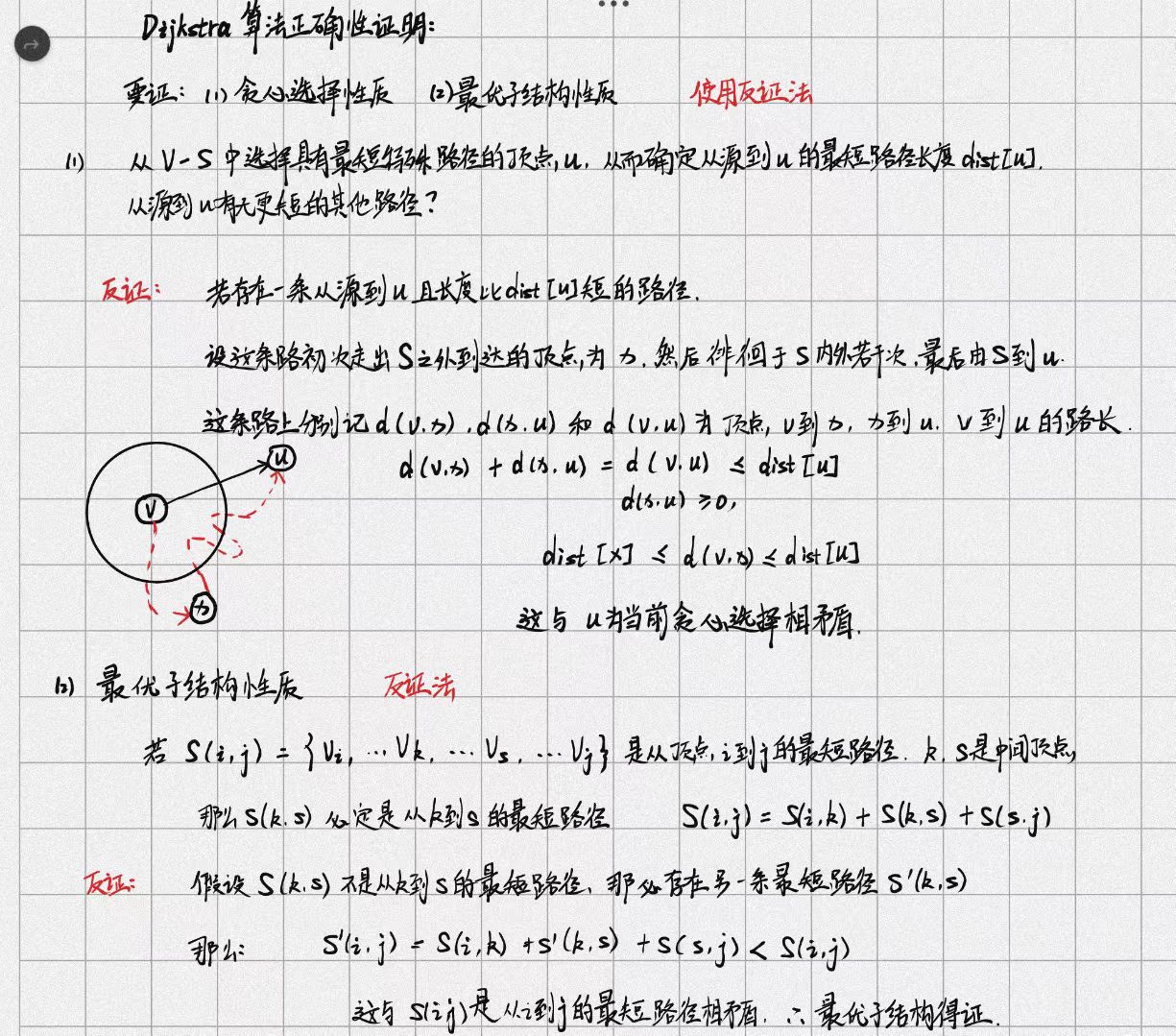
Dijkstra 算法可以求出由初始点到任意其他点的最短距离以及最短路径。Dijkstra 算法采用贪心策略，即每次都查找与该点代价最小的点。它是广度优先搜索的一种。它要求图中不存在负权边，或者，对于网格路径查找而言，不存在负的代价值（即每往外探测一个节点，都会增加这个路径的 “代价”）。其时间复杂度为 。

**算法思路：**

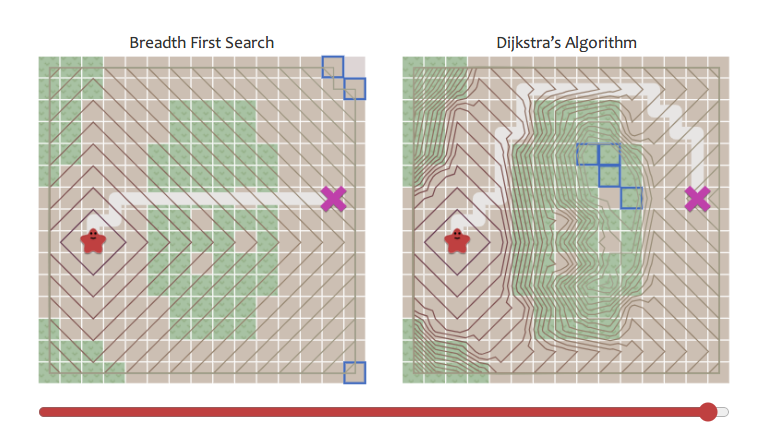
课上有讲，简单概括。设置顶点集合 S 并不断地作贪心选择来扩充这个集合。 一个顶点属于集合 S 当且仅当从源到该顶点的最短路径长度已知。 设 u 是图的任一个顶点，把从源到 u 且中间只经过 S 中顶点的路称 为从源到 u 的特殊路径。



**Dijkstra算法的贪心策略正确性证明：**



**计算复杂性:** 对于具有n个顶点和e条边的带权有向图，如果用带权邻接矩阵表示这个图，那么Dijkstra算法的主循环体需要时间。这个循环需要执行次，所以完成循环需要 时间。算法的其余部分所需要时间不超过。



1. **A\*算法寻路**

在A∗算法出现之前，人们都在用DFS和BFS进行搜索，然而，这两种算法在展开子节点时都属于盲目型搜索，也就是说，它不会选择在下一次搜索中更优的那个节点，继而借此跳转到该节点进行下一步的搜索，而是盲目全局搜索。如果运气不好，在此情形中，均需要试探完整个解集空间，显然，DFS和BFS只适用于问题规模不大的搜索问题中。

A\*算法属于启发式搜索算法，在寻路方面，它可以是在图形平面上，有多条路径，求出最低成本的算法，是非常流行的该类搜索算法中的一个，它一般被用于路径优化领域，例如导航、游戏里面的人物移动等。它的特别之处是在检查最短路径中时，检查每个可能符合需要的节点时都引入了全局的信息，然后对当前节点距终点的距离做出估计，并作为评价该节点处于最短路线上的可能性的量度。

**算法思路：**

A\*算法是把启发式方法如BFS（完全使用贪心策略），和常规方法如Dijsktra算法结合在一起的算法。它与 Dijkstra 算法很大的一个不同是，处理过程中，选取的节点会受到与终点的位置关系（距离等）的影响，即评估。这一个特性可以大大减少不需要的搜索，提高寻路效率。

A\* 算法使用如下来计算代价或优先级。代价越小，即优先级越高。f(n)是是节点 n 的综合优先级。当我们选择下一个要遍历的节点时，我们总会选取综合优先级最高（值最小）的节点；g(n) 是节点 n 距离起点的代价；h(n) 是节点 n 距离终点的预计代价，这也就是 A \* 算法的启发函数。

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

**核心在启发函数h(n)：**

1. 倘若 h(n) 恒为 0，即不启用启发函数来影响选取节点，此时的 A\* 算法实际上就是上文的 Dijkstra 算法（因为其没有考虑终点位置）。
2. 倘若 h(n) 为节点到终点确定代价，那么就可以最快的速度来找到确切路径。但是这是比较难的，因为 h(n) 是预计，路上可能会有障碍物等等。
3. 倘若 h(n) 恒小于等于实际的代价，那么 A\* 算法就一定能找到最短的路径，但是 h(n) 与实际的误差越大，其就会访问更多的节点，耗时也就约多。
4. 倘若 h(n) 恒大于实际的代价，那么 A\* 算法就不一定能找到最短的路径了，不过速度会较快。

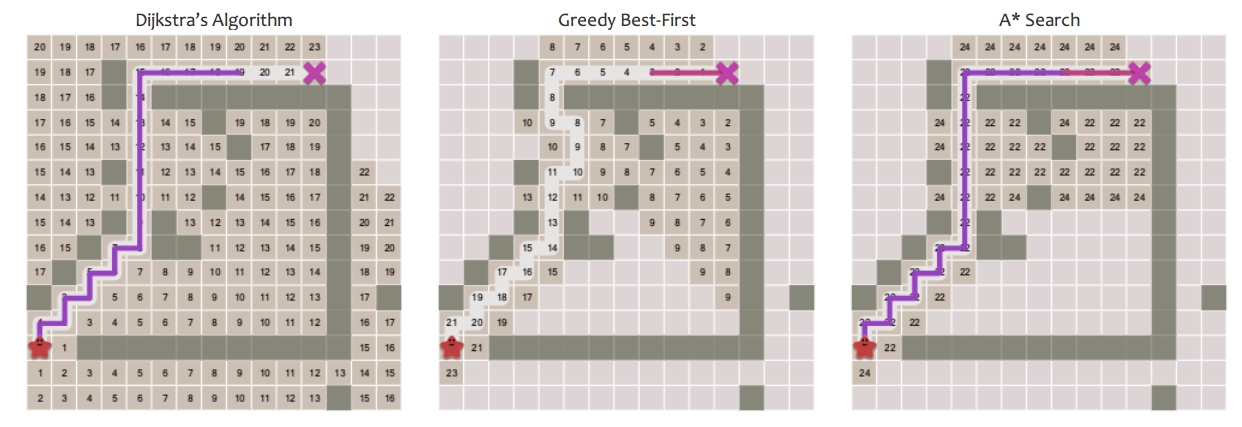
**通常采用的启发函数为各种距离：**

1. 曼哈顿距离：当前点到终点的横、纵坐标的差的绝对值的和。通常运用于上下左右移动时。
2. 欧式距离：即两点直线距离。通常用于可任意方向移动时（例如直升机）。
3. 对角距离：即在曼哈顿距离的基础上再增加可斜对角移动的条件，并响应更改距离。

本报告设计的迷宫问题为上下左右移动，所以采用曼哈顿距离：

**实现逻辑：**

1. 如上文所示，A\* 算法在处理时的核心在于 openlist（还需要关注的，随时可能再被抽出来访问的）、closelist（无需再关注的）。首先将起点设为 cur 并加入 open，然后再向四周（例如：上下左右）访问安全的节点（不是障碍或者膨胀障碍），并且在访问的时候将访问到的节点的父节点设为起点（即这些节点是由父节点所请求访问的），并将这些节点和其对应的 f、g、h 信息加入 open 里面。
2. 接下来将 cur 从 open 内移除并转入 close，然后在 open 中选 f 最小，即代价最小的节点，再按上文所示向四周访问（不在 close（倘若已在 close 则略过） 且不是障碍或者膨胀障碍）并记录父节点。请注意，倘若已在 open 里则看看由 cur 到那个节点的 f 能否更小，能则刷新，否则略过。
3. 倘若在访问周边节点时访问到了终点则寻路成功。如果出现了 open 为空的情况则寻路结束且没有路径。
4. 之后再按照和 Dijkstra 算法相同的方式，通过父节点信息，从终点反推即可。



**A\*算法的贪心性质正确性证明：**

A\*算法可以看作是在Dijkstra算法的基础上添加上启发函数，上述已经证明了Dijkstra的贪心策略，A\*的类似，但是需要考虑启发函数，会影响算法的速度和精确度。

frontier = PriorityQueue()

frontier.put(start, 0)

came\_from = dict()

cost\_so\_far = dict()

came\_from[start] = None

cost\_so\_far[start] = 0

while not frontier.empty():

current = frontier.get()

if current == goal:

break

for next in graph.neighbors(current):

new\_cost = cost\_so\_far[current] + graph.cost(current, next)

if next not in cost\_so\_far or new\_cost < cost\_so\_far[next]:

cost\_so\_far[next] = new\_cost

priority = new\_cost + heuristic(goal, next)#添加启发函数

frontier.put(next, priority)

came\_from[next] = current

**优点：**A\* 算法优点在于搜索路径直接，是一种直接的搜索算法，因此被广泛应用于路径规划问题

**缺点：**当 8 个邻居的代价中存在多个最小值时 A\* 算法不能保证搜索到最优路径（参考优化思路）；A\* 算法并没有完全遍历所有可行解，所得到的结果不一定是最优（调整 h(n)）

**优化：**当存在多个相同 f(n) 的节点时，AStar 不再具有目标倾向性，会拓展没必要的节点。打破对称性，让 AStar 具有更强的目标倾向性，可以减少扩展没必要的节点，主要修改 h(n)

* + 1. **回溯法寻路**

实际上和穷举法一样，不算一个完整的寻路算法，也是有许多寻路算法使用到了这种思想。思路远比具体实现的算法更重要，具体算法的实现可以体现多种思想。比方BFS可以基于穷举和回溯进行优化。

 回溯算法实际上是一个类似枚举的搜索尝试过程，主要是在搜素尝试过程中寻找问题的解，当发现已满足求解条件时，就“回溯”返回，尝试别的路径。回溯法是一种选优搜索法，按选优条件向前搜索，以达到目标，但当搜索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种不通就回退再走的技术称为回溯法，而满足回溯条件的某个状态的点称为“回溯”点，许多复杂的，规模较大的问题都可以使用回溯法，有“通用解题方法”的美称。从一条路往前走，能进则进，不能进则退回来，换一条路再试。

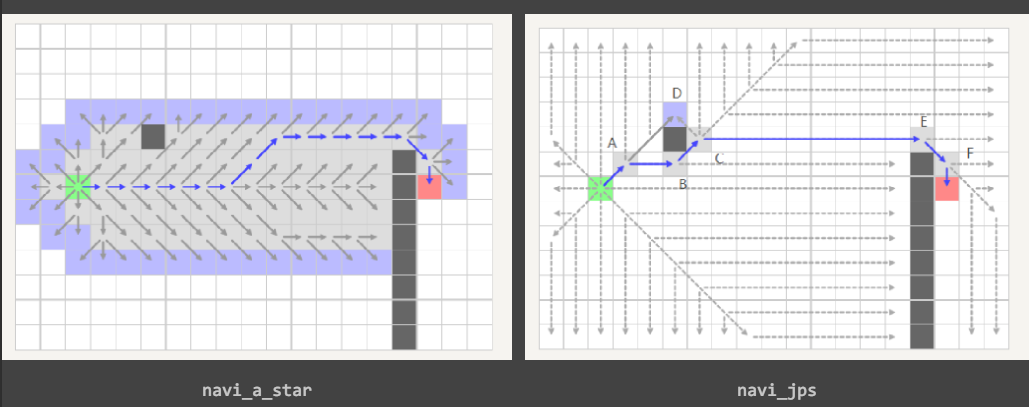
具体实现需要依赖栈或队列的数据结构。

* 1. **新型寻路算法**
     1. **JPS算法寻路**

JPS（Jump Point Search）算法，有时候也翻译成跳点算法。它是 A\* 的一种改进方法，实现了效率的大幅度提升。相较于它的老前辈们，JPS 算法的历史非常短，提出至今不过十年左右的时间，但是因为它出色的效率，现在已经基本上成为了 A\* 改进型的标杆算法了。

　　但是因为算法特性问题，它的使用场景比较受限，它只支持规则格子地图上的可用和阻挡两种情况，无法处理更复杂的地形，比如地图上某些地块有更高的优先级这种情况，而且它需要两个点之间可以对角线行走。

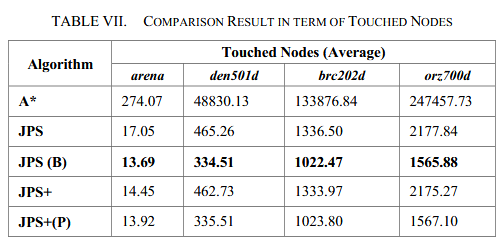
JPS 进行的优化在于，它根据当前地图上的状况，把一些点给排除了，不需要加入到开放集合中。开放集合中的点少了，需要计算的次数自然就少了。而可以加入到开放集合中的点被称为跳点，这也就是跳点算法名字的由来。



**JPS 算法的步骤要复杂一些，大概如下：**

1. 首先将起点设为当前点，并且将它加入封闭集合中。
2. 从当前点开始，检查当前点是否有前置点，如果没有前置点的，说明是起始点，起始点的扩散方向为周围全部的 8 个方向。如果不是起始点的，则会首先判断自己有没有强制邻居点，如果有的话，将强制邻居点加入到开放列表中，然后根据前置点和自己的位置判断自己所在的方向，如果自己是在平直方向上，则沿着该方向搜索，如果是在斜角方向上，则先向斜角两个向量方向搜索，然后再沿着斜角搜索。
3. 在延展搜索的过程中，如果碰到了地图边界或者是阻挡，则停止这个方向上的搜索。对搜索到的每一个点检查它是否有强制邻居点，如果有，则它就是跳点，停止当前的搜索，并且将该点加入到开放列表中。
4. 检查开放集合，如果为空，则地图搜索已经结束了，跳至步骤 6。
5. 如果开放集合不为空，则取出其中总开销最小的一个点设为当前点，比较它的位置是否与目标终点一致，如果一致，则搜索结束，否则将它加入到封闭集合中，然后跳回步骤 2。
6. 当搜索结束时，如果当前点的位置不是目标终点，那么寻路失败。否则从当前点也就是终点开始一直找前置点，这条线路的逆序就是寻路的结果。

JPS提出数十年，期间也经历了许多优化：



* + 1. **Dynamic A\*算法**

D\* 是 Dynamic A\* 即动态 A\* 的简写，其算法和 A \* 类似。其主要区别则在于 D\* 算法会在运算时动态地调整代价的计算方式，在应对动态地图、增量地图等条件下表现优秀。

D\* 算法的核心优势在于，它可能会遇到地图改变的问题（或者实际地图与假设地图不同的情况），而立即局部修正路线的效率较上者两个更高。

因此，D 算法核心在于假设，假设没有探测到的位置为某一环境（例如畅通），并基于假设以及每次观察到的信息后进行的对变化地图区块的新路线修正来找到尽可能短的路径。即 Real-Time Replanning in Dynamic and Unknown Environments 为对 D 算法最好的诠释。

A \* 是正向搜索，而 D\* 特点是反向搜索，即从终点开始搜索过程。在初次遍历时候，与 Dijkstra 算法一致，它将每个节点的信息都保存下来。

**D\* 包含了下面三种增量搜索算法：**

1. 原始的 D \* 由 Anthony Stentz 发表。

② Focussed D 由 Anthony Stentz 发表，是一个增量启发式搜索算法，结合了 A 和原始 D \* 的思想。

③ D Lite 是由 Sven Koenig 和 Maxim Likhachev 基于 LPA 构建的算法。

* + 1. **蚁群算法寻路**

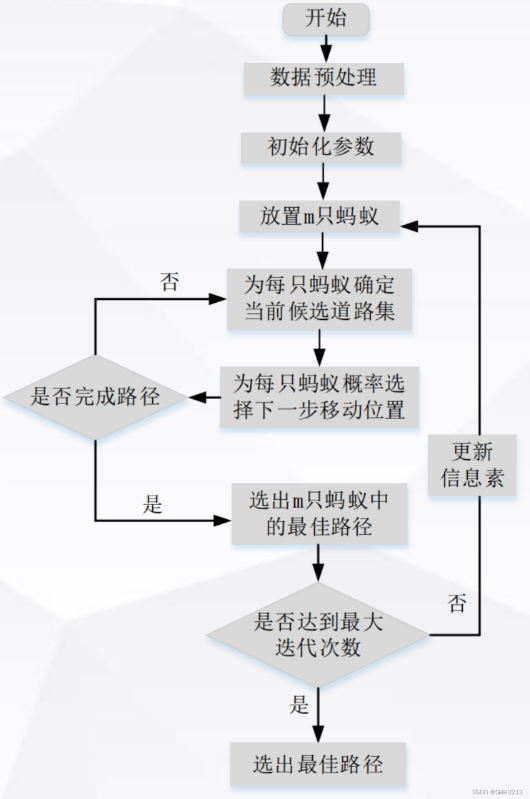
蚁群算法的基本原理来源于自然界中蚂蚁觅食的最短路径问题。

**基本思想：**

①用蚂蚁的行走路径表示待优化问题的可行解， 整个蚂蚁群体的所有路径构成待优化问题的解空间。

②路径较短的蚂蚁释放的信息素量较多， 随着时间的推进， 较短的路径上累积的信息素浓度逐渐增高， 选择该路径的蚂蚁个数也愈来愈多。

③最终， 整个蚂蚁会在正反馈的作用下集中到最佳的路径上， 此时对应的便是待优化问题的最优解。



* + 1. **基于人工智能的寻路算法**

人工智能的发展为寻路带来了新的思路，许多学者将人工智能技术应用于寻路中，很好地提高了寻路算法的智能性。近年来在这方面的找到的文章有：

1. **Global Path Planning Method by Fusion of A-star Algorithm and Sparrow Search Algorithm（2022 IEEE 11th Data Driven Control and Learning Systems Conference）（2022）**

本文提出了一种结合Astar和麻雀搜索算法（SSA）的全局路径规划算法（A-SSA）。

第一阶段：算法利用麻雀搜索算法在栅格地图中识别出若干关键节点。麻雀算法的灵感来源于麻雀群体的捕食行为，通过模拟这一自然过程，算法能够高效地探索环境并识别出重要的节点。

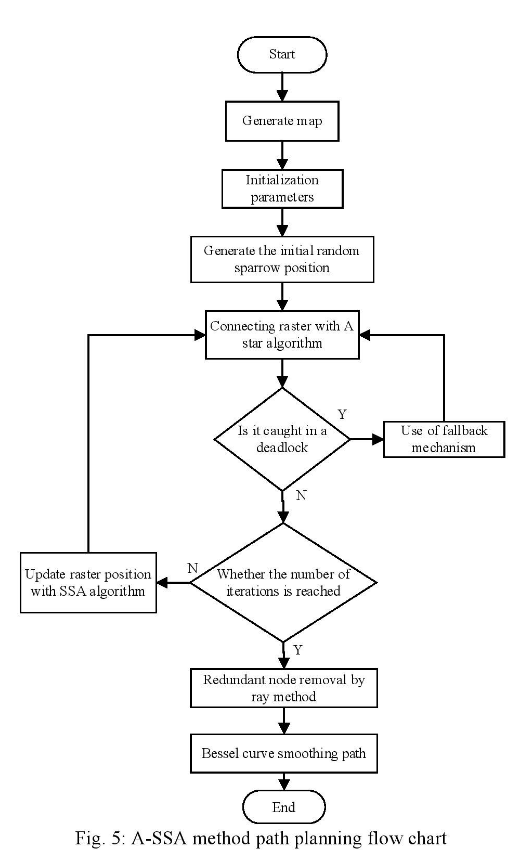
第二阶段：一旦关键节点被确定，A\*算法随后被应用于这些节点之间的连接，以形成一条初步的安全路径。这一阶段的目的是利用A\*算法的高效性来快速找到连接关键节点的最短路径。

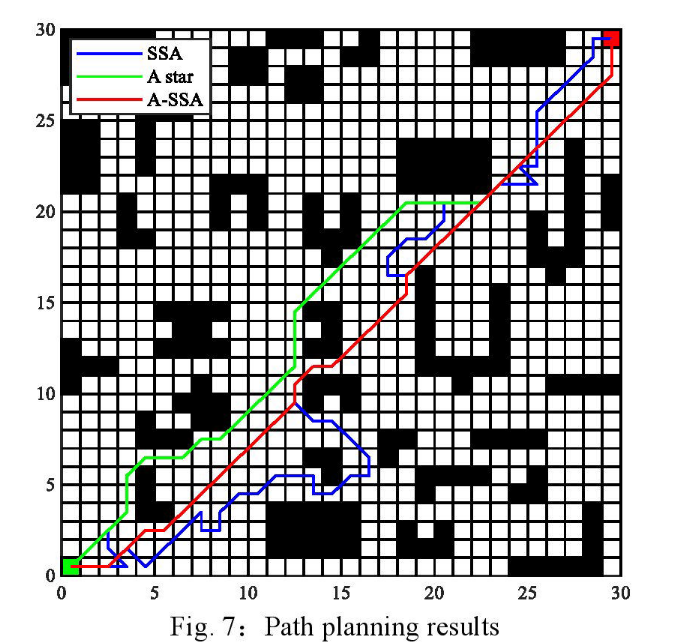
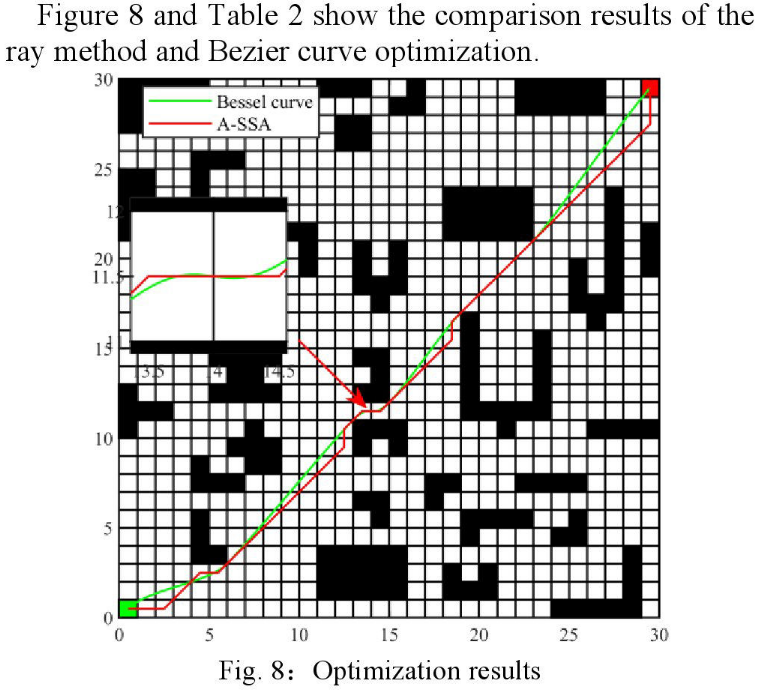
路径优化：在获得初步路径后，算法采用射线法对路径进行简化，去除不必要的冗余节点。这一步骤是为了减少路径长度并提高规划效率。随后，利用贝塞尔曲线对简化后的关键节点进行平滑处理，生成一条连续、无碰撞且平滑的最短路径。

死锁问题解决：针对路径规划过程中可能出现的死锁问题，本研究提出了一种后退机制。当AGV在规划路径时陷入困境，该机制允许AGV回退至先前的节点，并重新搜索可能的路径，从而有效避免了死锁的发生。

仿真实验：为了验证A-SSA算法的有效性，本文在不同障碍物覆盖率（20%、25%、30%、35%、40%）的30x30栅格地图内进行了仿真实验。实验结果表明，与其他两种算法相比，A-SSA算法规划出的路径长度最短，且在多次运行中均能稳定地收敛至最优解。

结论：A-SSA算法在AGV的最短路径规划问题上显示出了显著的优势，不仅路径长度最短，而且在面对不同障碍物覆盖率时均能有效避免死锁问题。这一研究成果为AGV在复杂环境中的路径规划提供了有价值的参考，并为未来的算法优化和应用提供了新的方向。

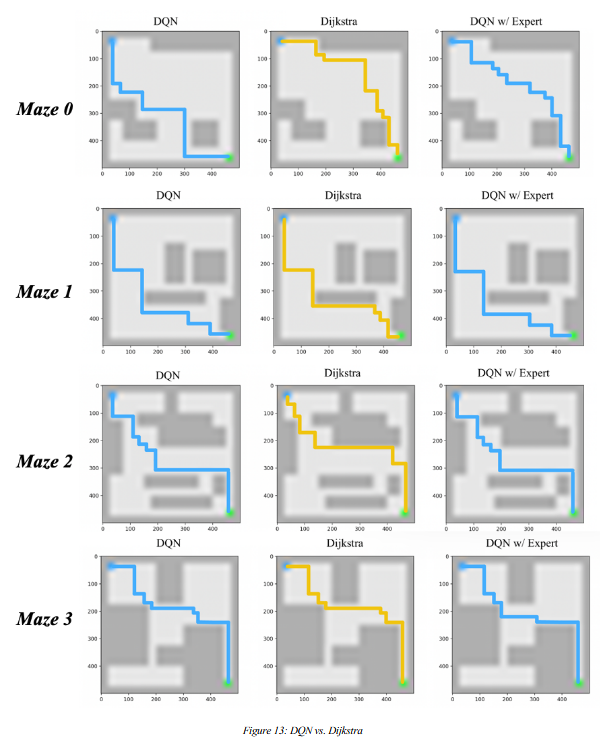


1. **An Application of Reinforcement Learning Techniques in Traditional Pathfinding (UCLA Masters of Applied Statistics Thesis)（2022）**

现代导航依靠寻路算法来确定两个地点之间的最短距离。这些算法以稳健的方式搜索图，从初始节点开始，分析连接到目的地的相邻位置。尽管这种技术能持续找到最优路线，但寻路仍依赖于特定环境的先验知识。强化学习是机器学习的一个分支，能够通过有效的探索、数据收集和利用来实现类似的结果。作为人工智能的一种形式，强化学习侧重于通过激励和惩罚来了解环境，从而做出最优决策，最终实现理想的目标收敛。

这个研究在定制迷宫环境中训练了三种无模型强化学习技术，即优势行为批判（A2C）、近端策略优化（PPO）和深度 Q 网络（DQN）。与标准寻路方法 Dijkstra 算法相比，结果表明 DQN 可以找到类似的路线，尤其是在专家指导行为的预先训练下，能以省时高效的方式达到这些最优解。



强化学习是人工智能领域真正的计算机想象力的最佳体现。通过试错学习，RL 不仅能执行决策，还能了解哪些决策能更好地解决手头的问题，同时注意到错误的行为。与视频游戏类似，RL 代理通过完成支线任务获得奖励，从而获得完成主线任务所需的工具集。这种稳健的方法展示了 RL 与生俱来的能力，即通过动态适应性做出明智决策并建立行为模式。传统的机器学习技术（如监督学习）通过一系列正确的操作进行训练，而 RL 则不同，它利用环境的后果作为信号，不断改进积极的行为。因此，这项研究展示了 RL 在极少引导的情况下捕捉复杂结构的能力。

1. **融合JPS和改进A\*算法的移动机器人路径规划（计算机科学与探索 北大中文核心 国家重点研发计划国家自然科学基金）（2022）**

针对传统 A\*算法在场景较大的栅格地图路径规划时，很多冗余节点的遍历导致寻路算法内存消耗大、 计算速度慢等问题，本文提出了一种对 A\*算法的改进策略。本文提出了一种改进的A算法，旨在解决传统A算法在大规模栅格地图路径规划中存在的内存消耗大和计算速度慢的问题。文章的主要创新点包括：

* 1. 启发函数的改进：文章改进了启发函数的具体计算方式，采用切比雪夫距离替代了传统的欧氏距离。这一改进使得启发式函数能够更精确地反映实际最佳路径，从而减少了算法在搜索过程中需要扩展的节点数量。
  2. 跳点搜索（JPS）策略的应用：文章引入了JPS策略，通过筛选出关键的跳点，并将这些跳点添加到OpenList和ClosedList中，代替了A\*算法中大量不必要的邻节点搜索。这种方法通过实现长距离的跳跃，显著减少了内存占用和节点评估的次数。
  3. 仿真测试与实验验证：为了验证改进后的A\*算法的效果，作者在五种不同尺寸的二维栅格地图上进行了仿真测试。测试结果表明，改进后的算法在减少寻路过程中评估的节点数量和提高寻路速度方面具有显著优势。此外，文章还将改进后的算法应用于移动机器人路径规划器的实验中，证明了其在大场景下进行快速路径规划的能力。

1. 效率与性能的提升：在实验中，与传统A\*算法相比，改进后的算法在路径搜索耗费时间上减少了92.2%，在扩展的节点数量上减少了97.37%，满足了移动机器人在复杂环境中快速路径规划的需求。
2. 改进的A\*算法不仅提高了路径规划的速度，同时也显著降低了内存消耗，提高了寻路效率。尽管如此，同时改进算法也具有局限性，即生成的路径可能不够平滑，这可能影响到机器人的运动效率。



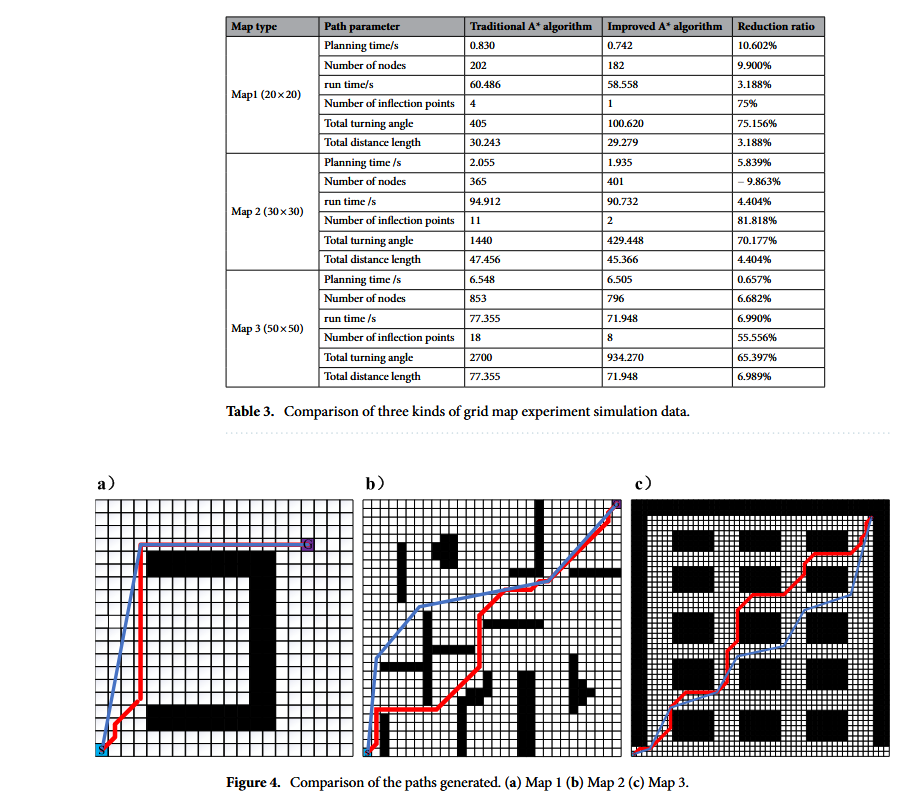
1. **Combined improved A\* and greedy algorithm for path planning of multi‑objective mobile robot（Nature子刊 Scientific Reports）（2022）**

本文提出的寻路算法在多目标移动机器人（AMR）的路径规划问题上的创新点主要包括：

* 1. 评价函数的改进：对A\*算法的评价函数进行了改进，以加快算法的收敛速度。
  2. 节点优化：移除了A\*算法中不必要的节点，同时只保留路径规划所需的关键拐点，这有助于减少算法的计算量并提高效率。
  3. **结合贪婪算法：**将改进的A\*算法与贪婪算法相结合，用于多目标点规划，这样可以在单一算法执行中同时考虑多个目标点。
  4. 路径平滑优化：设计了基于Floyd算法思想的路径平滑优化算法，通过去除冗余拐点来优化路径，使路径更加平滑，减少了路径长度和拐点数量。
  5. 多目标规划效率提升：改进的A\*算法不仅适用于单目标点搜索，还提高了多目标节点规划的效率。

1. 环境建模：提出了一种环境建模方法，通过网格化地图来表示环境，这有助于简化复杂环境的抽象过程。路径规划目标函数：考虑了AMR在电量不足时返回充电站的需求，通过目标函数确保AMR在最复杂情况下也能找到返回充电点的路径。

通过在不同尺寸的网格地图上进行模拟实验，验证了改进算法在减少路径长度、减少拐点、降低规划时间方面的有效性。与其他算法的比较：与Dijkstra算法、Best-First Search、BFS、RRT和双向A算法等其他算法相比，改进的A算法在路径长度和转弯角度上都有显著减少，平滑度有显著提升。



1. **算法设计与实现**
   1. **问题定义**

为了更直观地分析各种寻路算法的思想及效率，我选择了经典的迷宫问题，并设计了可视化的程序界面，用于显示寻路的流程与效率，便于分析。

**程序代码片段定义如下：**

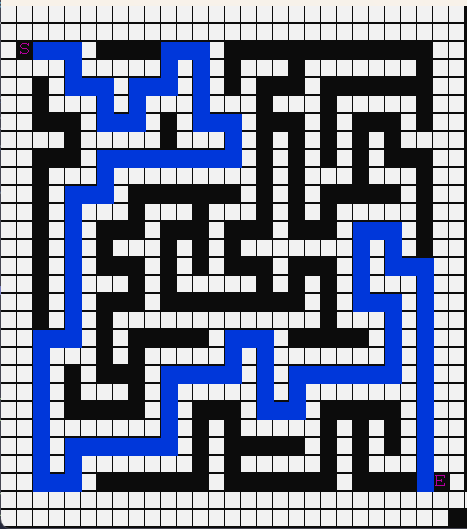
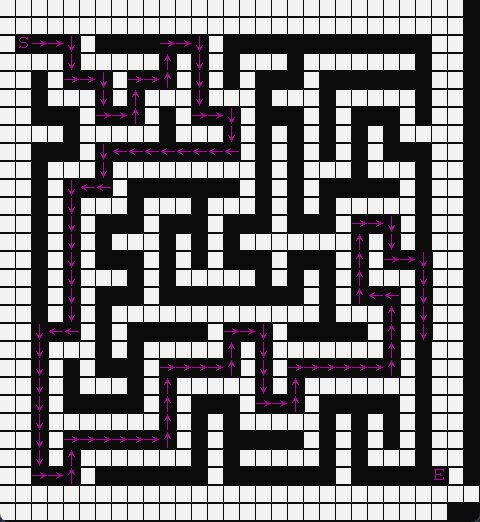
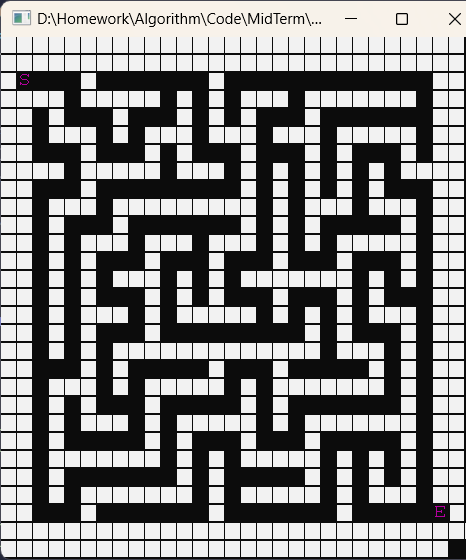
Maze.h用于迷宫求解问题相关函数的声明和变量常量的定义

Tools.cpp用于迷宫求解问题相关函数的实现，包括迷宫显示，交互以及寻路算法

Maze.cpp用于迷宫求解问题的逻辑交互，根据用户输入选择不同算法或功能

初始时，会在控制台调用地图生成函数，生成一个随机迷宫地图，随后等待用户按键，选择不同的寻路算法。比如1代表DFS，2代表BFS，依此类推。迷宫界面如下，墙用白色色块表示，起始位置为S，代表start，终点为E，代表end。在搜索过程中，选择的方向用箭头表示，方格中的小箭头代表位于该方格的选择方向，当找到路径时，使用蓝色色块填充路径。寻路由于地图和算法的不同可能不仅一条，但是只是输出最先找到的那条路径。

程序系统功能包括地图绘制，地图清空与重绘。



* 1. **经典寻路算法**

地图数据常常可以用图这类数据结构表示，在图结构中常用的搜索算法也可以应用到寻路中。经过上部分寻路算法原理的分析，可以发现一些常见的寻路算法就体现了算法分析与设计课程上学习的几种思想。

比方基本的深度优先搜索（DFS）和广度优先搜索（BFS）算法就是穷举思想的直接体现。它们通过探索所有可能的路径（DFS）或层级（BFS）来寻找解决问题的方法；Dijkstra算法和A\*搜索算法则展示了贪心法的应用，这些算法在每一步选择当前看起来最优的路径，希望这样的局部最优决策能够导致全局最优解；值得注意的是，这些算法设计思想并不是相互排斥的。实际上，它们可以结合使用，以提高算法的效率，例如深度优先搜索在路径寻找过程中也涉及了回溯操作，这不仅体现了穷举的思想，也体现了回溯的策略。

此外，动态规划也是一种高效的算法设计技巧，它通过将问题分解为子问题，并将子问题的解存储起来（记忆化），避免了重复计算，从而提高了算法的效率。在某些情况下，动态规划可以视为贪心法和分治法的结合，它既考虑了子问题的最优解，也考虑了这些子问题解的组合方式。

在本文中，我将采用BFS作为穷举法的示例，DFS作为结合了回溯和穷举的示例，Dijkstra和A-star算法作为贪心法的实例，同时，这两种算法还能展示启发式算法与非启发式算法之间的差异。至于动态规划思想，我将结合广度优先遍历的策略，解决迷宫问题，这一问题具有明显的重叠子问题和最优子结构特性。通过运用记忆化手段处理重叠子问题，以期望找到迷宫问题的高效解决方案。

* + 1. **DFS**

DFS维护一个栈，，后进先出，总是访问容器中最后一个进入的元素。DFS会沿着一条路走到死，如果没有路了就进行回溯。DFS不一定会找到最短路径。

**数据结构设计：**

栈遵从后进先出的特点，所以使用一个固定大小的数组用于实现栈的结构，该数组用于保存搜索过程中的节点信息，作为临时存储空间。节点信息在DFS中包括横纵坐标，在后面A-star中还包括启发值等，所以为了方便起见，我定义同一个数据结构POSITION存储点的相关信息。

struct POSITION {

int x, y; //坐标

int G, H, F; //G当前开销，H预估开销，F总开销

POSITION\* Parent;//父节点

//构造函数

POSITION(int x = 0, int y = 0) : x(x), y(y), G(INT\_MAX), H(0), F(0), Parent(NULL) {}

};

POSITION Stack[2500] = { {0, 0} }; // 自定义数据结构-栈

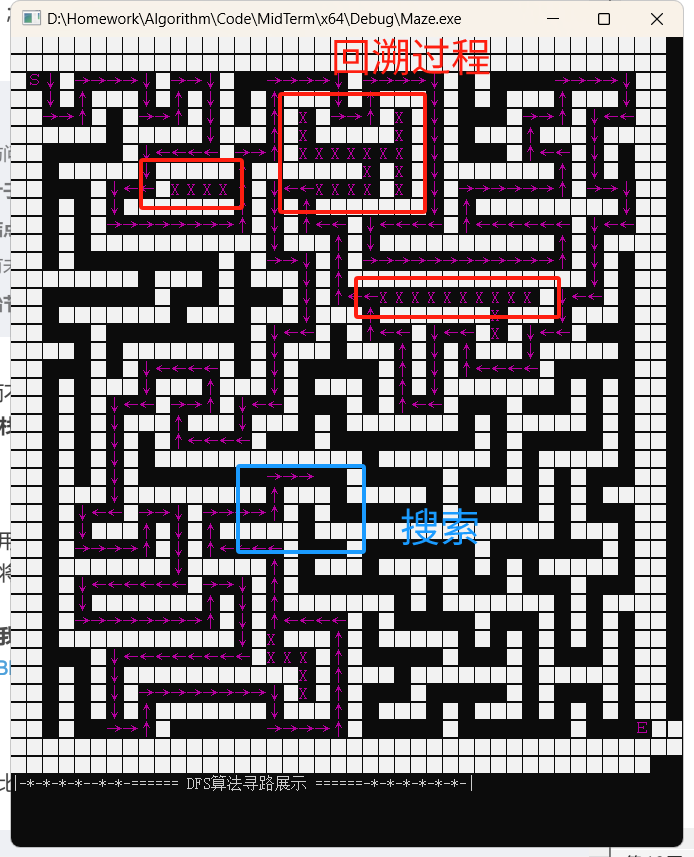
**实现思路：**

1. **初始化：**设置一个栈Stack来存储搜索过程中经过的节点；并初始化栈顶指针top为-1，表示栈为空；以起点为起始位置，设置初始方向。
2. **循环搜索：**使用一个while循环，条件是栈顶指针大于-2（表示栈不为空）并且当前位置不是终点位置；在每一步中，按照一定的顺序依次尝试移动（右、下、左、上），直到找到可移动的方向或者所有方向都不可行。也就是迭代实现DFS算法
3. **移动过程：**如果某个方向是通路（ROAD），就将当前位置移动到该方向，并将该位置入栈，更新栈顶指针；如果某个方向不可行（墙壁或者已经访问过的位置），则返回上一个节点并尝试其他方向。
4. **路径记录：**在移动的过程中，记录每个位置的移动方向（RIGHT、DOWN、LEFT、UP），以便后续回溯路径。
5. **回溯路径和结果打印：**如果搜索到了终点或者栈为空，表示搜索结束；清空地图，然后根据移动记录回溯路径，并将路径上的节点高亮输出。

**DFS 的特点：**

是优先探索深度而不是广度。它会尽可能深地搜索每一个路径，直到找到目标节点或者遍历完所有节点。通过使用栈来保存遍历过程中的节点。

**结果展示：**可以看到在选择DFS寻路时，存在回溯操作（搜索过程中的“X”），很直观地看出来DFS深度遍历的特点。也可以看到其较盲目的特点。



**核心代码：**

// DFS（深度优先搜索）主要循环，不断尝试向不同方向移动，直到达到终点或无法继续移动

while (top > -2 && (x != MaxLine || y != MaxColume + 1))

step++; // 记录扩展节点数top++; // 栈顶指针加一

Stack[top].x = x;Stack[top].y = y;

// 打印向右移动的路径标记

SetPosition(x, y);

cout << RIGHTSTR;

// 如果右侧是通路，则向右移动

if (map[x][y + 1] == ROAD){

map[x][y] = RIGHT;

y++;

}

else{

// 否则，尝试向下移动（同理）

SetPosition(x, y);

cout << DOWNSTR;

if (map[x + 1][y] == ROAD){

map[x][y] = DOWN;

x++;

}

else{// 尝试向左移动（同理）

else{// 尝试向上移动（同理）

else{ // 如果四个方向都无法移动，则标记为死路

map[x][y] = DEAD;

SetPosition(x, y);

cout << DEADSTR;

top -= 2; // 回溯到前一个节点

x = Stack[top + 1].x;

y = Stack[top + 1].y;

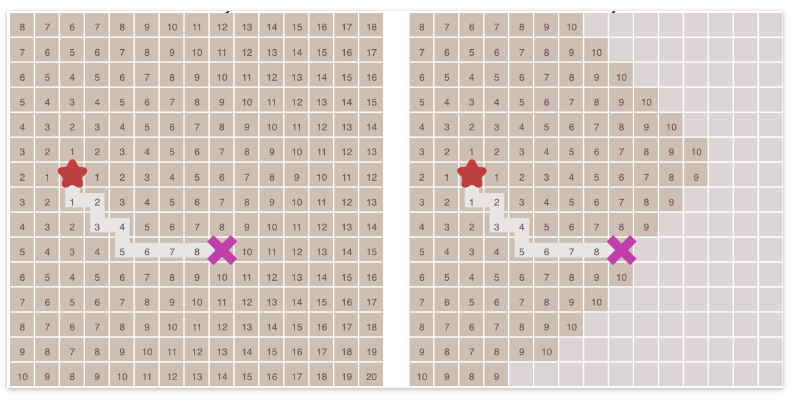
**复杂度分析：**

**时间复杂度：**DFS的时间复杂度取决于搜索过程中访问的节点数量，即扩展节点数。在最坏情况下，DFS可能会探索整个迷宫，扩展所有可能的路径直到找到目标节点或者确定不存在路径。假设迷宫的尺寸为N×M，最坏情况下，DFS会访问所有的N×M=V个节点，E条边。DFS的时间复杂度为O(V+E)。

**空间复杂度：**DFS的空间复杂度取决于栈的大小，即存储搜索路径的空间。DFS的空间复杂度主要取决于栈的大小，即存储搜索路径所需的空间。在最坏情况下，DFS会将所有可能的路径都压入栈中，直到找到目标节点或者确定不存在路径，DFS的空间复杂度为O(N×M)。

* + 1. **BFS**

BFS 从起点开始，首先遍历起点周围邻居点，然后再遍历邻居的邻居，这样逐步的向外一层一层扩散（类似地震波），直到找到终点。在左图中，BFS 算法遍历了图中所有的点，这通常没有必要，因为对于有明确终点的问题来说，一旦到达终点便可以提前终止算法。



**数据结构设计：**

BFS的实现依赖队列，同样可以使用结构体数组构造自定义队列，这样的好处在于结构体可以自定义，存放相关信息。

struct QUEUE

{

int x, y;//坐标

int PreSub;//前一个节点的下标

};

QUEUE Queue[2500] = { {0, 0, 0} }; // 自定义数据结构-队列

int head = -1, tail = -1; // 队列头尾指针

**算法设计：**

队列的概念直观，操作也简单，同时具有高效性，向队列的两端添加/移除元素的操作的时间复杂度为O(1)。BFS的基本思想是按照层级逐层遍历，即从起始节点开始，首先访问其所有的相邻节点，然后再访问相邻节点的相邻节点，以此类推。这样可以保证先访问的节点距离起始节点更近。

对于BFS算法，需要关心以下五个内容：

* 1. 遍历范围（迷宫的大小）；
  2. 初始点和目标点（即迷宫的起点和终点）；
  3. 搜索方向（在本例中，我们搜索上下左右四个方向，通常情况下也是如此）；
  4. 访问标记（即是否已经访问过需要访问的位置，BFS不存在二次访问）；
  5. 父结点 （即我是因为访问了哪个节点，才需要访问这个结点）。

实现思路：

初始化：

int step = 0;

int x, y, PreSub;

tail = head = 0;

memset(Queue, 0, sizeof(Queue));

Queue[head].PreSub = -1;

Queue[head].x = 2;

Queue[head].y = 1;

广度优先遍历：

// 广度优先搜索，不断扩展队列中的节点，直到找到终点或队列为空

while (tail <= head){

// 获取当前队列头部节点的坐标和前驱节点索引

x = Queue[tail].x;

y = Queue[tail].y;

PreSub = Queue[tail].PreSub;

step++; // 记录扩展节点数

// 尝试向上移动

if (map[x - 1][y] == ROAD){

map[x - 1][y] = UP;

SetPosition(x - 1, y);

cout << UPSTR;//可视化

// 将新节点加入队列，并设置前驱节点索引

head++;

Queue[head].x = x - 1;

Queue[head].y = y;

Queue[head].PreSub = tail;

// 如果新节点是终点，则退出循环

if (Queue[head].x == MaxLine && Queue[head].y == MaxColume + 1)

break;

}

// 尝试向下移动（同理）

if (map[x + 1][y] == ROAD){}

// 尝试向右移动（同理）

if (map[x][y + 1] == ROAD){}

// 尝试向左移动（同理）

if (map[x][y - 1] == ROAD){}

Sleep(DELAY / 2); // 延迟，可观察搜索过程

tail++; // 出队

if (kbhit())// 如果按下键盘3键，则返回

if (getch() == '3')

return;

}

填充最短路径：

tail = Queue[head].PreSub;

SetConsoleTextAttribute(hout, 28); // 路径颜色

if (Queue[head].x == MaxLine && Queue[head].y == MaxColume + 1)

while (tail != -1){

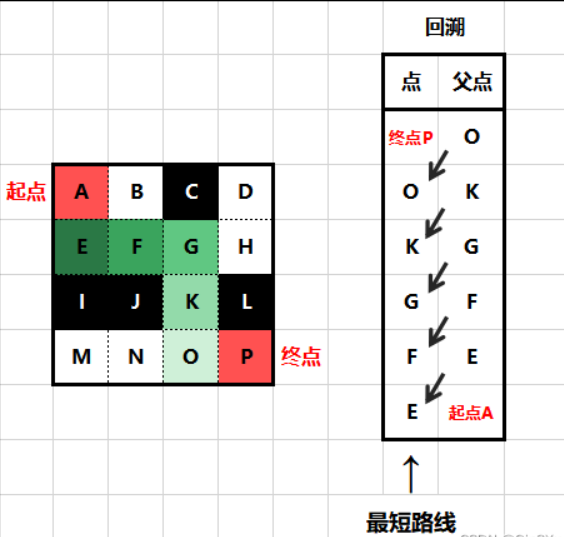
SetPosition(Queue[tail].x, Queue[tail].y);

cout << BLANKSTR; //输出色块

tail = Queue[tail].PreSub;//依次出队

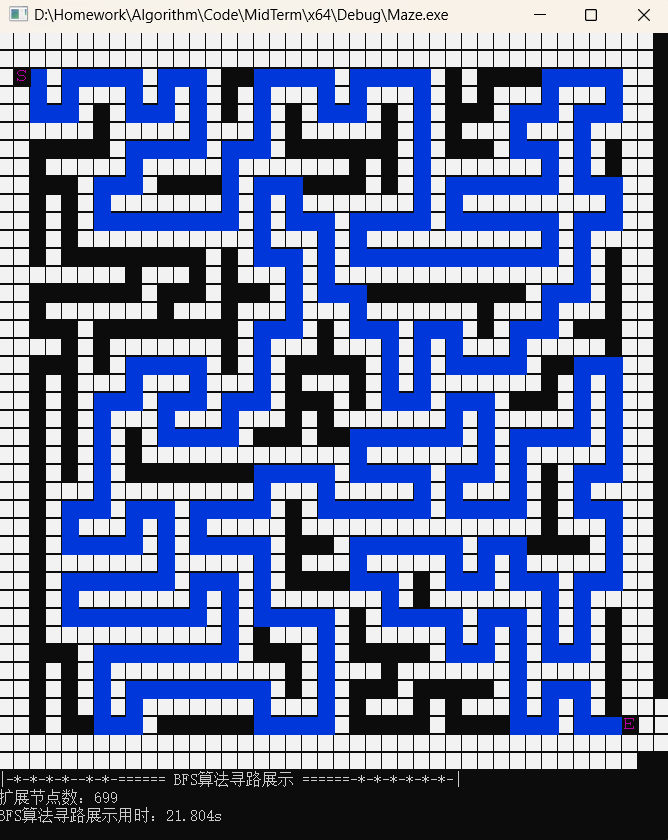
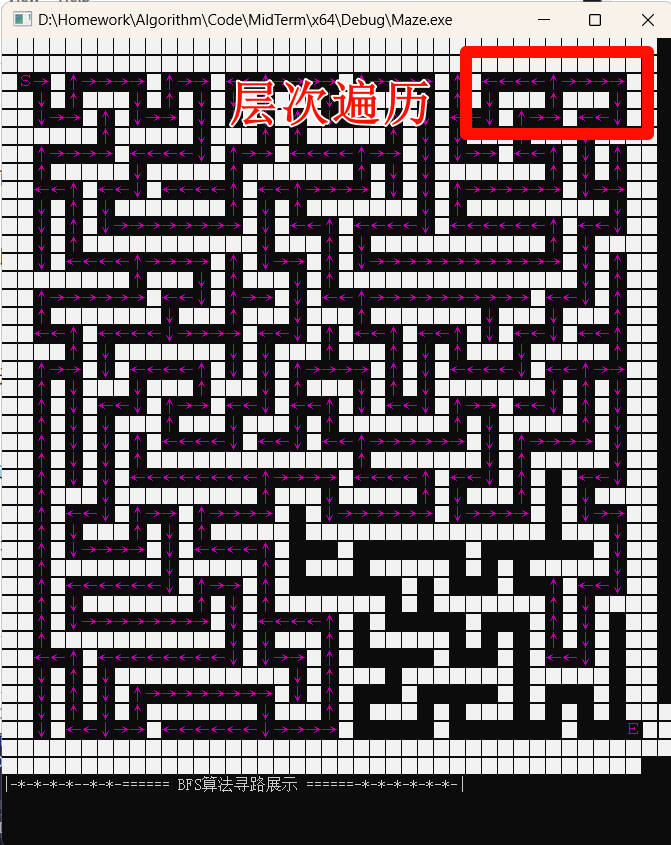
}

终点处理（ 路径回溯 ）：在搜索到终点后，可以通过回溯的方法，由终点根据结点的父结点倒推出路径。根据BFS算法不走回头路的特点，每个结点只有唯一的一个父结点，我们可以利用这个特性，从终点逆推出走向起点的路径。



**结果展示：**

可以看到BFS的搜索过程类似于层次遍历，有多条路径同时搜索，，可以找到最短路径。



**复杂度分析：**

**时间复杂度：**假设一个图有 N 个节点和 M 条边，BFS 会走遍所有节点，时间是 O(N)，然后由于每个节点会检查所有的出边，最终所有的边都会被检查过，时间是 O(M) ，所以 BFS 的时间复杂度是 O(N+M)

**空间复杂度：**队列里面最多可能存放所有节点，空间复杂度为 O(N)

**对比分析：**

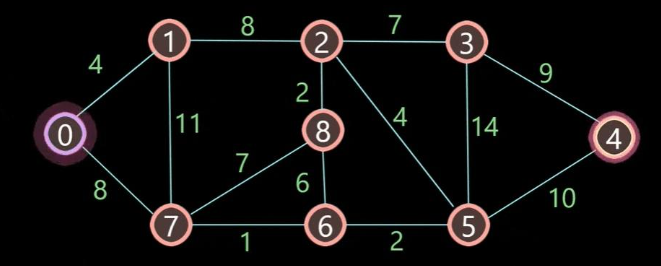
DFS和BFS相比较，DFS实现简单，容易理解和实现，对于深度优先搜索问题，DFS 往往能够更快地找到可行解。在搜索树比较深且解比较分散时，DFS 的内存消耗最少（从扩展节点数量可以看出）。

DFS不一定能够找到最短路径，因为它不考虑距离和路径长度，但BFS则可以。如果图中有环状路径，DFS将陷入无限循环，因此需要保证节点的独特性，BFS则不会。

* + 1. **Dijkstra**

Dijsktra和BFS的本质区别在于BFS是按照人为预先定义好的顺序访问容器中的节点，而Dijsktra是访问当前容器中累计成本最低的节点，累计成本为g(n)。g(n)是从起始节点到节点n的累计成本的当前最佳估计。

在算法中需要计算每一个节点距离起点的总移动代价（到起点的总权重），并把待遍历的节点加入优先队列中（按照代价排序），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。当图形为网格图，并且每个节点之间的移动代价相等，那么 Dijkstra 算法将和 BFS 一样。算法不知道目标节点的位置，因此它只能向所有可能的方向扩展节点直到发现目标节点为止



**算法设计：**

1. **初始化：**算法使用了一个二维数组 dist 来记录从起点到每个节点的最短距离，并将所有距离初始化为 ，表示“无穷大”。起点 到自身的距离被初始化为0。
2. **前驱节点记录：**prev 数组用于记录最短路径上的前驱节点，这对于后续路径的回溯是必要的。
3. **已确定节点集：**settled 数组用于标记已经找到最短路径的节点，以避免重复处理。
4. **未确定节点集：**使用 std::vector<POSITION> 类型的 unsettled 队列存储尚未确定最短路径的节点，这是Dijkstra算法的核心。
5. **节点扩展：**算法通过四个方向扩展节点，对于每个方向上的相邻节点，如果它是一个有效的移动（即在迷宫范围内且不是墙壁），并且没有被标记为已确定，算法将计算通过当前节点到达该相邻节点的距离，并更新 dist 和 prev 数组。
6. **终点检查：**当扩展到终点 (MaxLine, MaxColume + 1) 时，变量 reachEnd 被设置为 true，若设置为false，则代表寻找最短路径。
7. **路径回溯：**一旦找到最短路径，算法将使用 prev 数组从终点回溯到起点，以构建完整的最短路径。
8. **可视化：**使用 SetConsoleTextAttribute 和 SetPosition 来在控制台上显示路径。

**复杂度分析：**

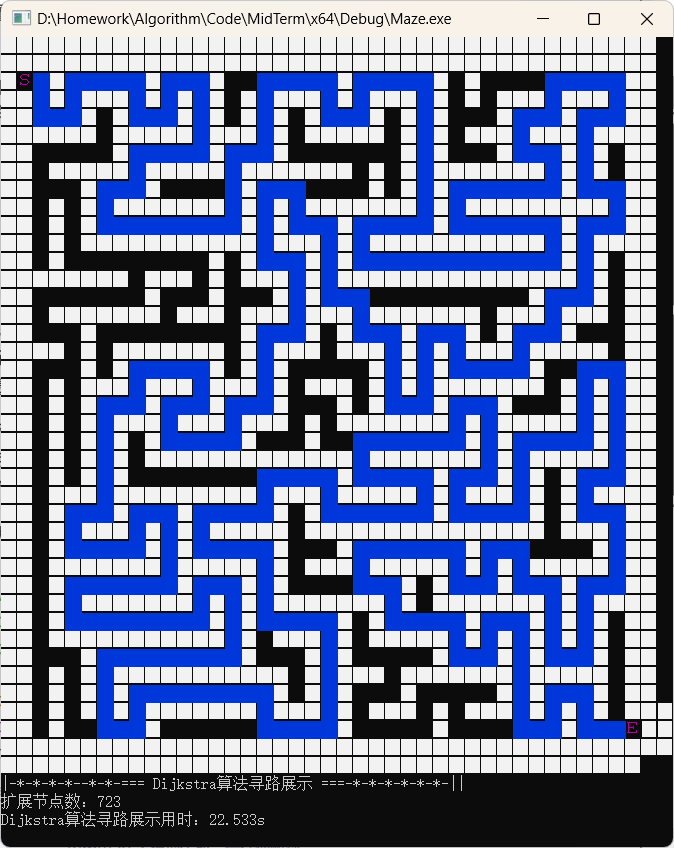
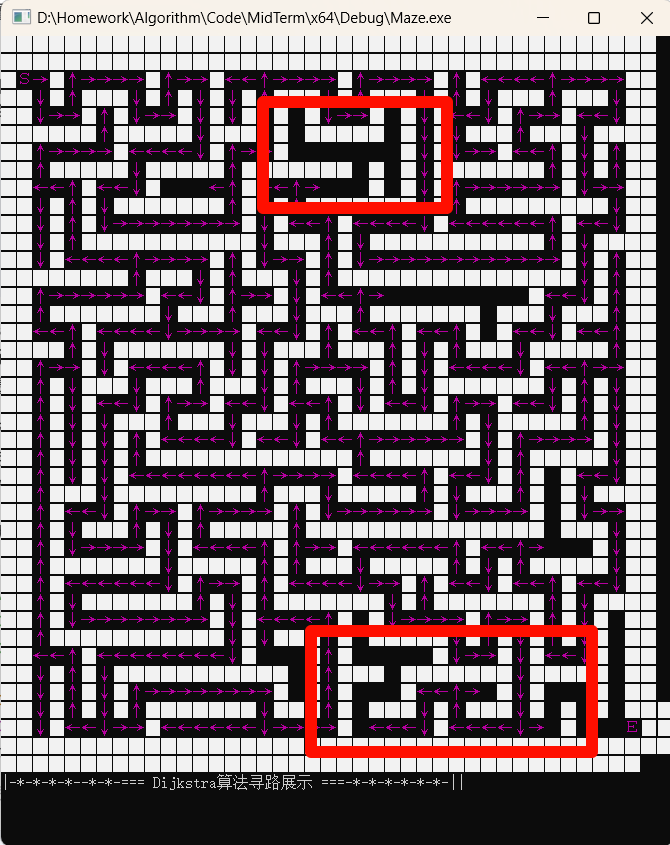
**时间复杂度：**Dijkstra算法的时间复杂度主要取决于未确定节点的扩展次数和每个节点的相邻节点检查次数,对于每个未确定节点，算法最多执行4次相邻节点的检查（上、下、左、右）。不用最小优先队列为 ,用最小优先队列 ,我使用了普通队列.

**空间复杂度：**算法使用了几个二维数组来存储节点的距离和前驱节点信息，以及一个队列来存储未确定节点。因此，空间复杂度是 。

**结果展示：**

从可视化过程中可以很好地看出Dijkstra的步骤，先初始化，再访问未访问集合，存储所有未确定最短路径的顶点，从未访问集合中选择一个具有当前已知最短路径的顶点，将其标记为已访问，并更新其相邻顶点的距离。对于当前顶点的每个邻居，如果通过当前顶点到达邻居的距离比已知的更短，则更新邻居的最短路径距离，然后重复，直到未访问集合为空，或者所有顶点的最短路径都已经找到。

而且Dijkstra可以找到最短路径，如果把reachEnd设为false，那么则会搜索所有点。会出现在搜索过程中先找到了终点，然后继续搜索剩余的点。



* + 1. **A-star**

A\*算法在机器人、导航、游戏等领域有广泛的应用。A\*可以说是Dijkstra的改进版，目的在于解决Dijkstra效率低的问题，上面说到Dijkstra不知道目标节点的位置，所以只能向所有方向扩展直到发现目标节点。之前提到的GBFS算法的缺点在于，过分关注目标点的位置，即GBFS每次访问的节点具有到目标节点的最小代价，但是在有障碍物的情况下路径不是最优解。

两者取长补短，诞生了A\*算法，A\*同时借鉴了两种算法，在Dijkstra基础上引入了启发式函数h(n)， h(n)表示了当前节点到目标节点的成本。保证了最优性的同时，加入了目标节点的信息，提升了效率。

g(n): 对从初始节点到节点n的累计成本的当前最佳估计。

h(n): 节点n到目标节点的代价估计（即目标代价）。

f(n) = g(n) + h(n) ： 从初始节点，通过当前节点n，再到目标节点的代价估计。

1. **初始化**：定义起点 start 和终点 end，初始化从起点到所有节点的距离 distToStart 为无穷大，初始化前驱节点 prev，以及标记节点是否已确定 settled。
2. **优先队列**：使用最小优先队列（基于堆的数据结构）来存储节点，队列中的每个元素都是一个三元组 (currentF, x, y)，其中 currentF 是节点的启发式估计值（从当前节点到终点的估计成本），F由G+H得到，其中G为从起点到现在点的代价，H为从该点到终点的代价，因为是二维栅格，所以使用了哈密顿距离。将起点的 F 值（实际成本加启发式成本）和坐标推入优先队列。
3. **主循环**：当优先队列不为空时，执行循环。从队列中取出具有最小 F 值的节点。
4. **确定节点**：如果节点已经被确定（即已经被访问过），则跳过；否则，将其标记为已确定。
5. **终点检查**：如果当前节点是终点，设置 reachEnd 为 true 并退出循环。
6. **扩展节点**：对于当前节点的四个方向，计算到达相邻节点的实际成本，并结合启发式函数计算总成本。如果新计算的成本比已知的更小，更新相邻节点的 distToStart 和 prev，并将相邻节点加入优先队列。
7. **回溯**：一旦找到终点，算法将使用 prev 数组从终点回溯到起点，构建最短路径。打印出从起点到终点的最短路径。

A\*算法最为核心的过程，就在每次选择下一个当前搜索点时，是从所有已探知的但未搜索过点中(可能是不同层，亦可不在同一条支路上)，选取f值最小的结点进行展开。而所有“已探知的但未搜索过点”可以通过一个按f值升序的队列(即优先队列)进行排列。 这样，在整体的搜索过程中，只要按照类似广度优先的算法框架，从优先队列中弹出队首元素（f值），对其可能子结点计算g、h和f值，直到优先队列为空(无解)或找到终止点为止。

**数据结构设计：**

bool reachEnd = false; //终点标志

POSITION start = { 2, 2 }; // 起点

POSITION end = { MaxLine, MaxColume + 1 }; // 终点

// 用于记录从起点到每个节点的距离

std::vector<std::vector<int>> distToStart(MaxLine + 3, std::vector<int>(MaxColume + 4, INT\_MAX));

// 用于记录每个节点的前驱节点

std::vector<std::vector<POSITION>> prev(MaxLine + 3, std::vector<POSITION>(MaxColume + 4, { -1, -1 }));

// 用于记录每个节点的 F 值

std::vector<std::vector<bool>> settled(MaxLine + 3, std::vector<bool>(MaxColume + 4, false));

// 优先队列，按 F 值从小到大排序

std::priority\_queue<std::tuple<int, int, int>, std::vector<std::tuple<int, int, int>>, std::greater<std::tuple<int, int, int>>> pq;

//用于G值计算

distToStart[start.x][start.y] = 0;

**核心部分：**

while (!pq.empty())//当未确定集合不为空时

{

int currentF, x, y;//当前节点的 F 值和坐标

// 取出 F 值最小的节点

std::tie(currentF, x, y) = pq.top();

pq.pop();

if (settled[x][y])

continue;

settled[x][y] = true;//标记当前节点为已确定

if (x == end.x && y == end.y)//如果到达终点

{

reachEnd = true;

break;

}

// 检查四个方向的相邻节点

for (int dir = 0; dir < 4; ++dir)

{

// 定义方向数组，方便遍历四个方向

int dx[4] = { 0, 1, 0, -1 };

int dy[4] = { 1, 0, -1, 0 };

int newX = x + dx[dir];

int newY = y + dy[dir];

// 确保新的坐标在迷宫范围内且不是墙壁

if (newX >= 0 && newX < MaxLine + 1 && newY >= 1 && newY < MaxColume + 2 && map[newX][newY] != WALL)

{

// 计算新的距离和启发值

int newDistToStart = distToStart[x][y] + 1;

int newHeuristic = Heuristic({ newX, newY }, end);

int newTotalCost = newDistToStart + newHeuristic;

// 如果找到了更短的路径，则更新距离和前驱节点

if (newTotalCost < distToStart[newX][newY]){

distToStart[newX][newY] = newTotalCost;

prev[newX][newY] = { x, y };

pq.push({ newTotalCost, newX, newY });

step++;

// 打印当前节点的探索方向

SetPosition(newX, newY);

根据方向输出箭头

Sleep(DELAY / 2); // 如果需要观察过程

if (newX == MaxLine && newY == MaxColume + 1)

reachEnd = true;

}

}

}

if (reachEnd)

break;

}

**复杂度分析：**

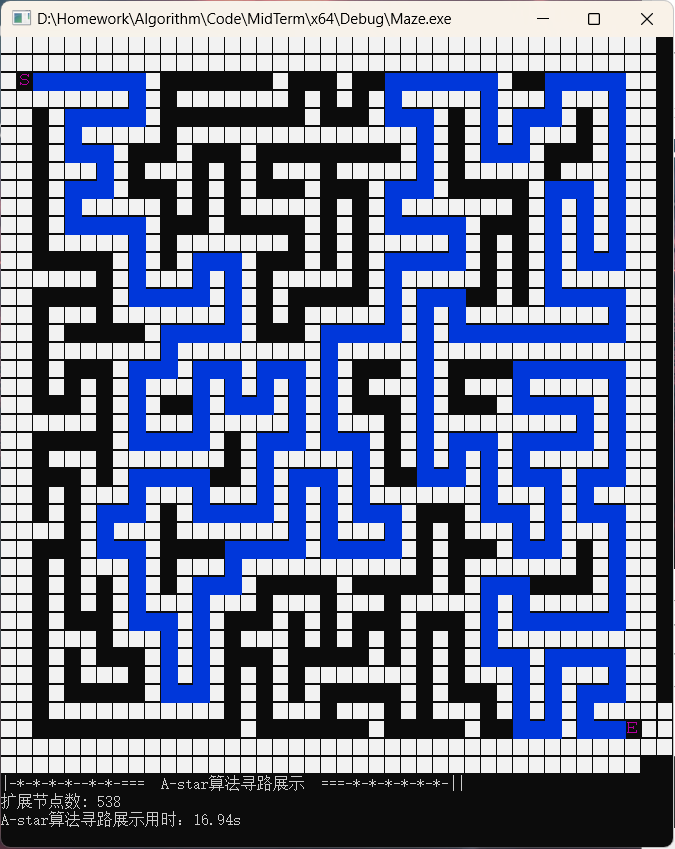
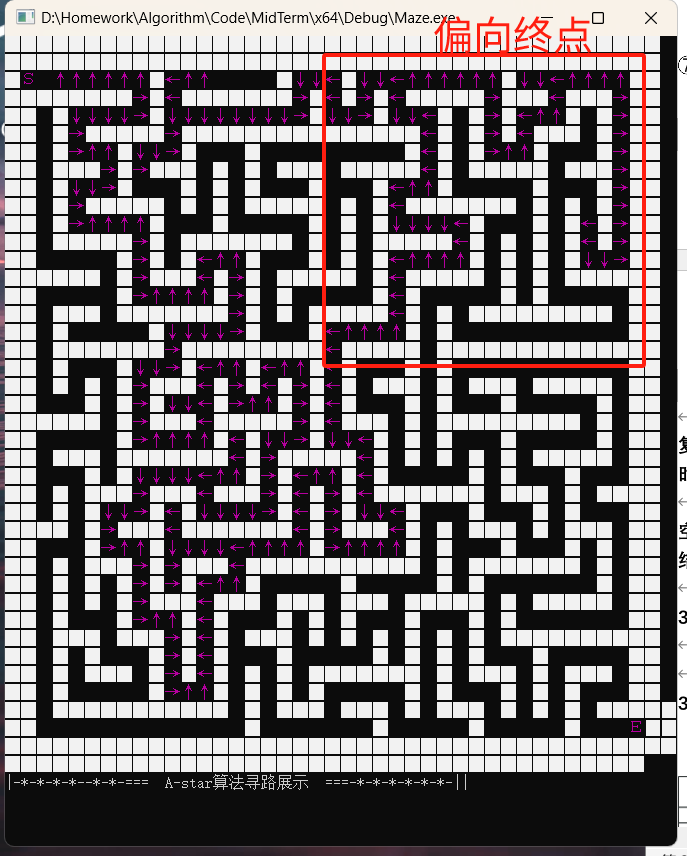
**时间复杂度：**A算法的时间复杂度取决于启发式函数的质量。理想情况下，如果启发式函数是完美且一致的（即对于任何起点和终点，启发式估算与实际成本相等），A算法的时间复杂度是 O(b^d)，其中 b 是分支因子（每个节点的平均邻居数），d 是起点到终点的实际距离。

然而，在实际应用中，启发式函数通常不是完美的，因此时间复杂度可能会更高。在最坏的情况下，如果启发式函数总是低估实际成本，A算法的时间复杂度可能退化到Dijkstra算法的水平，即，其中 V 是顶点数，E 是边数。但是，由于启发式函数的引导，A通常在实际应用中表现更好。

**空间复杂度：**A\*算法需要存储所有节点的 distToStart、prev 和 settled 信息，以及一个可能包含所有节点的优先队列。因此，空间复杂度为 O(V)。

**结果展示：**

可以看出在加入了启发式函数后，A-star的搜索不再那么盲目，效率要比Dijkstra提升不少。我们可以看出，在最短路径搜寻效率上，一般有A\*>Dijkstra、双向BFS，其中Dijkstra、双向BFS到底哪个算法更优，还得看具体情况。



* + 1. **使用动态规划思想的BFS算法**

动态规划问题一般形式就是求最值（最长递增子序列、最小编辑距离）其本质就是穷举，但不是暴力穷举，其思想源于暴力穷举，但使用了“备忘录”或DP Table进行优化。动态规划三要素：重叠子问题、最优子结构、状态转移方程。在上一部分我已经证明了迷宫问题的最优子结构和重叠子问题性质，以及他的状态转移方程。在这里我结合动态规划的备忘录应对重叠字结构以及自底向上的回溯思想。

数据结构设计和BFS类似。

**寻路过程：**

1. **初始化**：算法初始化一个距离数组 dist 和一个前驱节点数组 prev。dist 用于记录从起点到每个节点的最短距离，而 prev 用于在找到路径后回溯。起点 (2, 1) 被加入到队列 q 中，并标记为已访问（距离设置为0）。
2. **广度优先搜索**：算法使用一个循环来处理队列中的节点。对于队列中的每个节点，算法检查其所有四个方向的邻居。
3. **邻居检查：**对于每个邻居节点，算法检查其是否在迷宫范围内、不是墙壁，并且之前没有被访问过。如果邻居节点未被访问，算法更新该节点的距离，并将其前驱节点设置为当前节点。将邻居节点加入队列中，以便后续处理。
4. **终点检查：**如果在搜索过程中到达终点，算法将停止搜索。
5. **路径回溯：**一旦找到终点，算法使用 prev 数组从终点回溯到起点，构建最短路径。

**动态规划思想：**

1. **子问题分解：**将问题分解为更小的子问题。
2. **最优子结构：**问题的解决方案包含子问题的解决方案。
3. **记忆化：**存储子问题的解，避免重复计算。

**数据结构设计：**

// 用于记录从起点到每个节点的最短距离

int dist[MaxLine + 3][MaxColume + 4] = { 0 };

// 用于记录最短路径上每个节点的前驱节点

POSITION prev[MaxLine + 3][MaxColume + 4];

// 初始化最短距离数组，除了起点外，其他都设为 INT\_MAX

dist[2][1] = 0; // 起点到自身的距离设为0

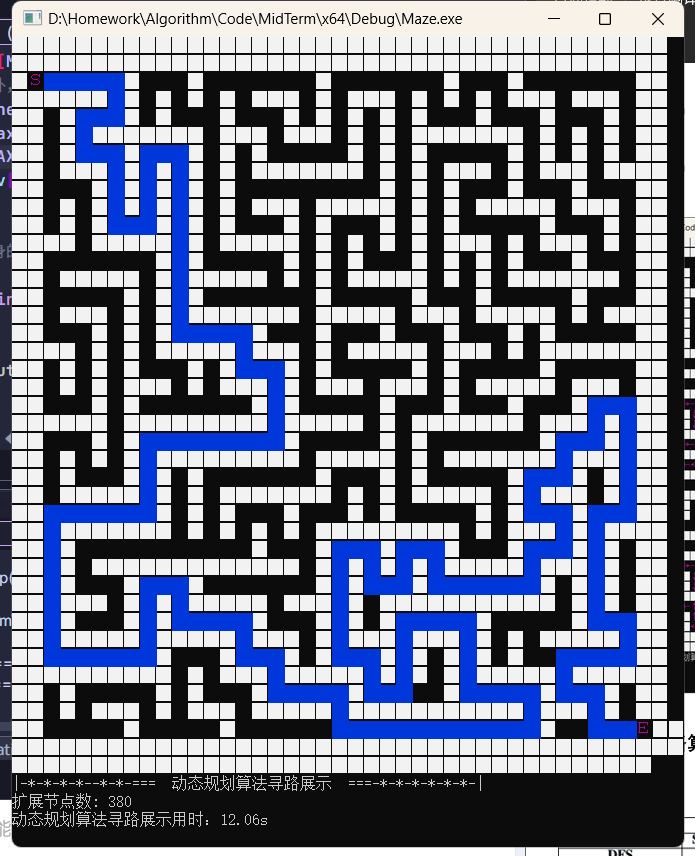
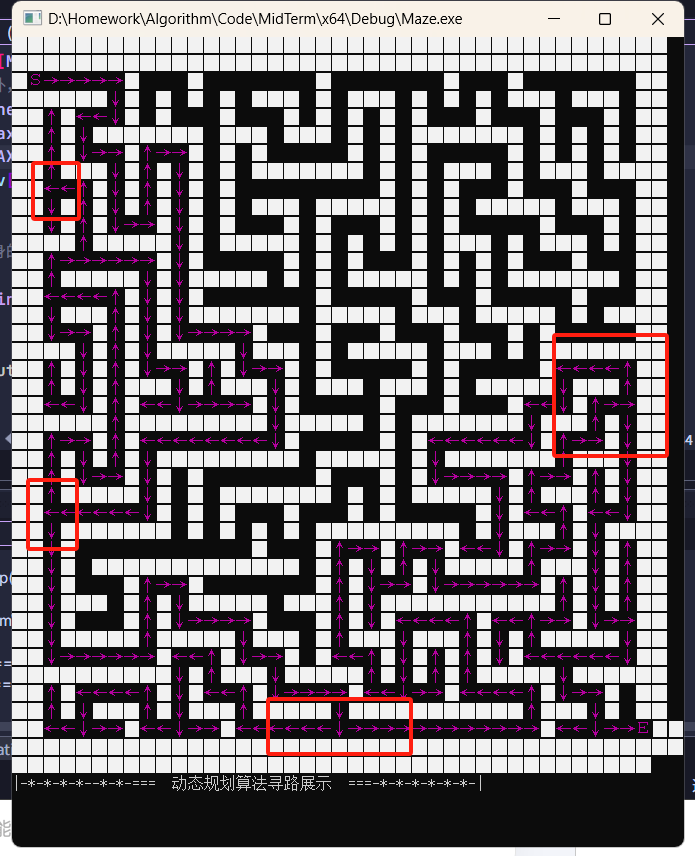
std::queue<std::pair<int, int>> q;

**复杂度分析：  
时间复杂度：**搜索过程中对于队列中的每个节点都会检查四个方向的邻居，最坏的情况下每个节点可能被访问一次，以及四个邻居，但是邻居属于是子问题可能已经解决可以直接调用，所以时间复杂度大概是O(V).路径回溯的时间复杂度为O(P),P是最短路径上节点的数量，在最坏情况下可能接近于V。

**空间复杂度：**用于存储子问题解的dist数组和用于回溯的prev数组，大小为O(V),存储了迷宫中每个节点的信息。广度优先搜索：对于队列中的每个节点，算法都会检查四个方向的邻居。在最坏的情况下，每个节点都可能被访问一次，并且每个节点有四个邻居。因此，对于每个节点，算法都会执行四次检查和可能的更新操作。这意味着 BFS 的时间复杂度是 O(4 \* V)，简化为 O(V)。其他一些路径向量，变量基本固定或小于O(V).

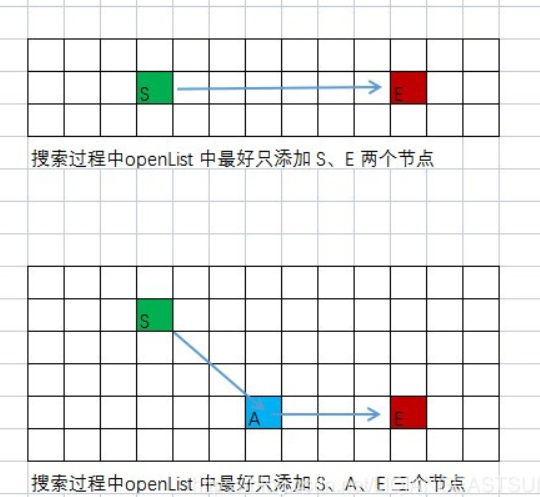
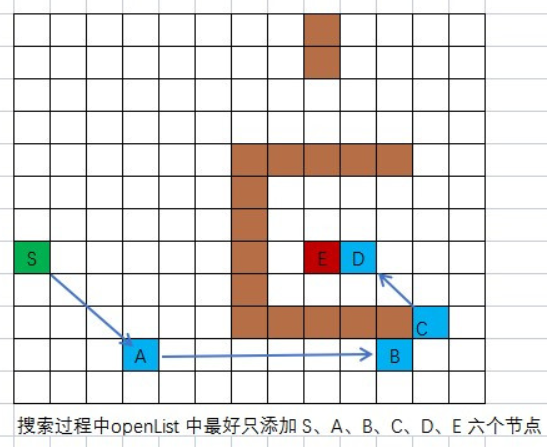
**结果展示：**

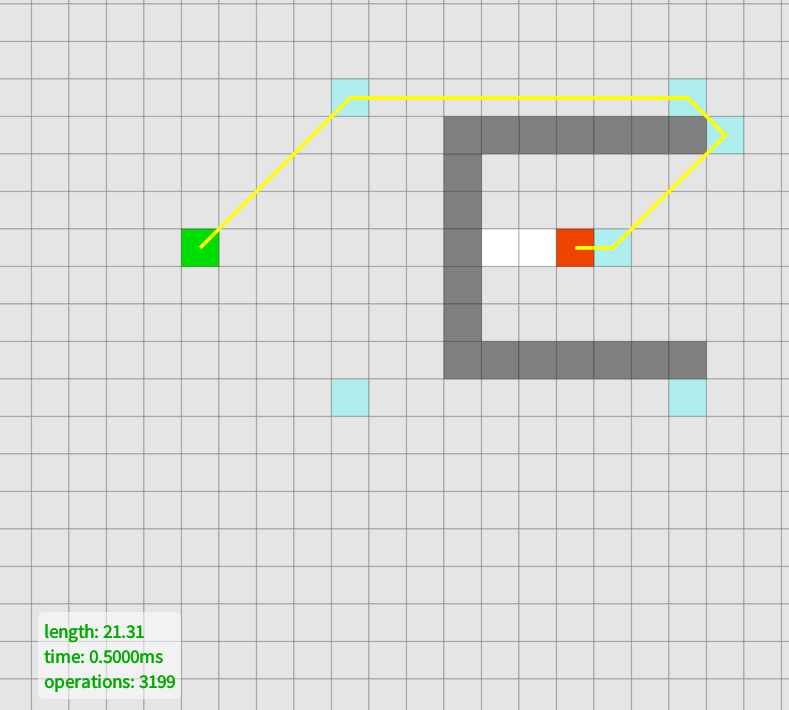
可以看出使用了动态规划的BFS算法效率要更加高效，因为动态规划的记忆化策略使得已经解决的子问题的解记录下来，方便后续调用不需要再重复计算了。



* 1. **新型寻路算法**
     1. **JPS寻路算法**

AStar 算法在扩展节点时会把所有相邻的节点考虑进去，当地图比较大时，openList 中的节点数量会很多，搜索效率较低。





JPS的实现还是比较复杂的，由于时间关系没有完全实现，在这里给出伪代码

横向纵向的格子的单位消耗为10，对角单位消耗为14。

定义一个OpenList，用于存储和搜索当前最小值的格子。

定义一个CloseList，用于标记已经处理过的格子，以防止重复搜索。

def 获取邻居点

if 当前点是起点

返回当前点九宫格内的非障碍点

elseif 当前点与父节点是对角向

判断并添加相对位置右方的邻居点

判断并添加相对位置下方的邻居点

判断并添加相对位置对角的邻居点

判断并添加相对位置左下角的强迫邻居

判断并添加相对位置左上角的强迫邻居

elseif 当前点与其父节点是横向

判断并添加相对位置右方的邻居点

判断并添加相对位置上方的强迫邻居

判断并添加相对位置下方的强迫邻居

elseif 当前点与父节点是纵向

同横向逻辑，判断并处理下方，左右向强迫邻居

def 递归寻找跳跃点

if 传入点是终点

返回终点

if 传入朝向是对角向

if 传入点存在强迫邻居

返回此传入点

if （递归寻找跳跃点 传入点：横向+1 朝向：横向）结果不为空

返回此传入点

if （递归寻找跳跃点 传入点：纵向+1 朝向：纵向）结果不为空

返回此传入点

elseif 横向

if 上下方有强迫邻居

返回此传入点

elseif 纵向

if 左右方有强迫邻居

返回此传入点

返回 递归寻找跳跃点 传入点：横向+1,纵向+1 朝向 对角

def Main

起点加进OpenList中

While（OpenList.Count > 0）:

从OpenList中取出F值最小的点并设置为当前点

把当前点加进CloseList

邻居点s = 获取邻居点（当前点）

for 邻居点s

跳跃点 = 递归寻找跳跃点（邻居点）

if 跳跃点不再CloseList中

计算并设置当前点与跳跃点的G值

计算并设置当前点与跳跃点的H值

计算并设置跳跃点的F值

将当前点设置为跳跃点的父节点

如果邻居点在OpenList中

计算当前值的G与该邻居点的G值

如果G值比该邻居点的G值小

将当前点设置为该邻居点的父节点

更新该邻居点的GF值

若不在

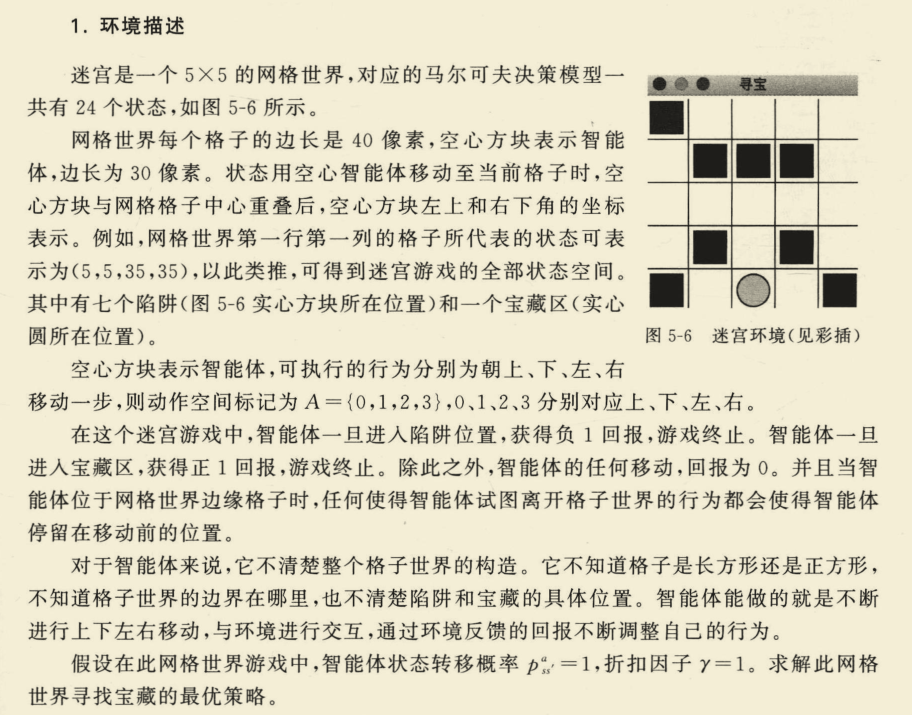
计算并设置当前点与该邻居点的G值

计算并设置当前点与该邻居点的H值

计算并设置该邻居点的F值

将当前点设置为该邻居点的父节

* + 1. **基于强化学习的寻路算法**

网上找到的相关论文和博客，论文没来得及复现，给出自己的理解。

Sarsa和Q-Learning两种方法的流程几乎是一样的，主要区别在于Q值的更新公式不一样。程序流程：

1. 初始化环境

env = Maze()

1. 初始化Q表格，

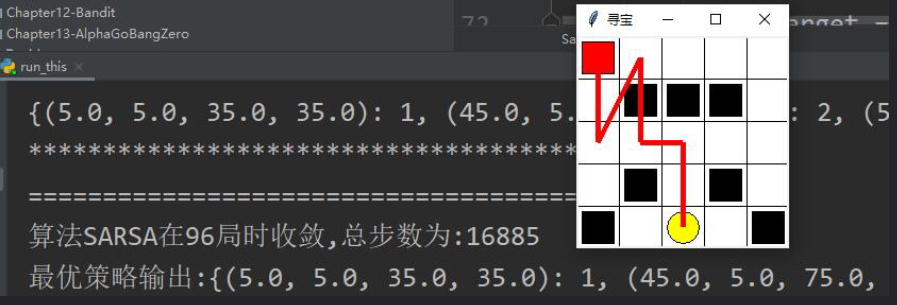
RL = SarsaTable(actions=list(range(env.n\_actions)))

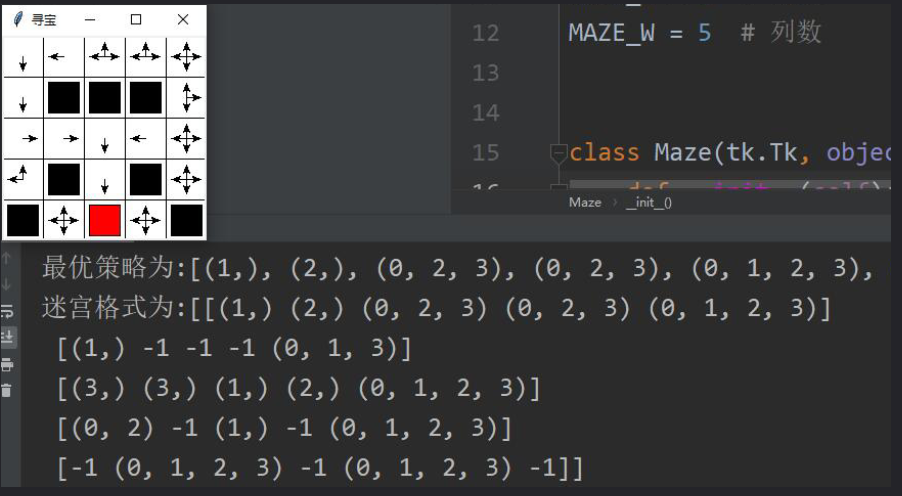
1. 设定100幕迭代，每次迭代首先初始化状态，即将初始位置放在左上角。

observation = env.reset()

1. 基于当前状态选择动作，采用的是epsilon-贪心选择，epsilon取值为0.9，即每次有90%的概率选择当前状态的最优动作，10%的概率进行随机选择，即探索。选择前，先检查该状态是否在Q表格中存在，不存在就添加。
2. 保存临时策略，策略即当前状态下的选择的动作，在程序中可以理解为一个字典，键就是当前状态，键值就是动作。
3. 采取动作并获得下一个状态和回报以及是否终止信息
4. (这一步只有Sarsa有，Q-Learning没有)：再次获取下一个动作，由于Sarsa需要五个值，因此还需要根据下一个状态来再次选择一次动作而Q-Learning不需要再次进行动作选择（体现了离轨策略的思想）
5. 更新Q表格，这一步是两者区别的关键，两者的更新公式不一样。 两者的区别就在于下一时刻的动作a‘如何选择。Sarsa和第一次选择动作一样，再次进行动作选择；而Q-Learning直接基于下一个状态S’，在Q表格中选择最大价值的动作。Q-Learning的效果应该会比Sarsa要好。
6. 先判断是否到达终止状态，若到达，结束这一幕，并再次判断是否收敛；这里收敛的条件设为三次策略policy不变化，如果不收敛，将临时的策略进行保存；如果收敛，跳出循环，结束操作。

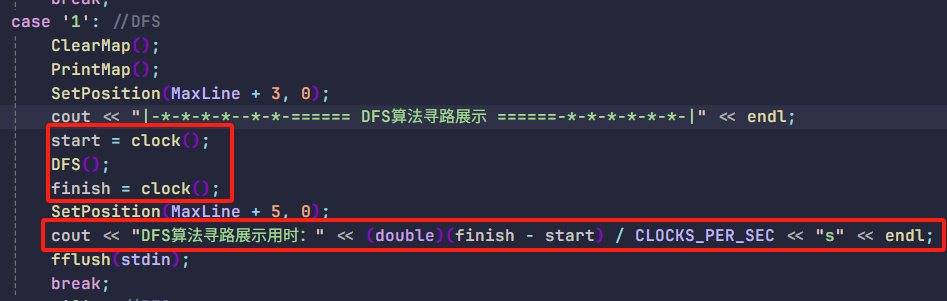
结果展示：





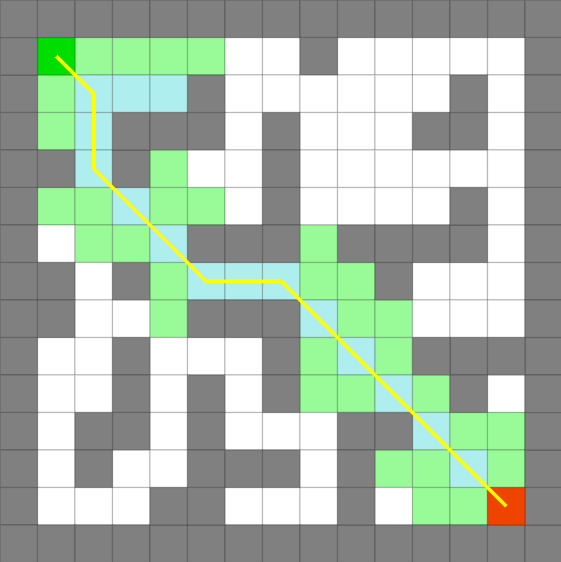
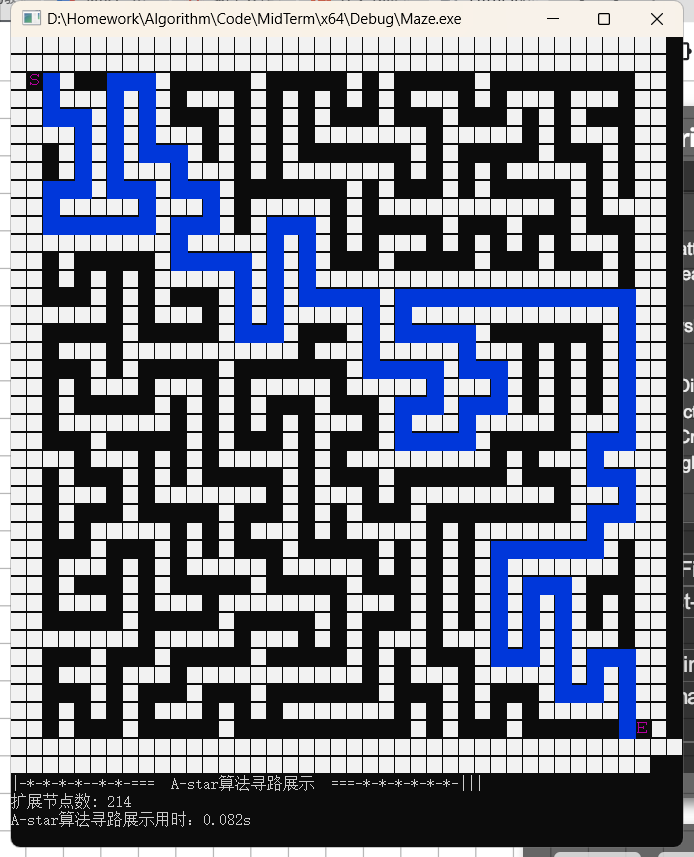
1. **算法比较与性能评估**
   1. **启发式寻路VS非启发式寻路**

在性能评估这里，为了排除其他无关因素干扰，我将输出箭头，延时等全部注释掉，只保留寻路的逻辑。并引入ctime库，用于记录寻路算法所用的时间。



在资源开销上，我使用扩展节点的个数或者子问题个数来表示，可能有一定误差，但是大体可以反映不同算法的效率差异。

DFS,BFS,Dijkstra均为非启发式寻路，他们的方向选择更像是固定了顺序，而A\*，JPS则为启发式函数，它们引入启发值，根据估值函数来评价子节点的好坏，也就是进行排序。简单的来说A\*就是将估值函数分成两个部分，一个部分是路径价值，另一个部分是一般性启发价值，合在一起算估整个结点的价值。

|  |  |  |  |
| --- | --- | --- | --- |
| **算法名称** | **用时（mine）** | **扩展节点（mine）** | **PathFinding.js** |
| **DFS** | 0.119s | 177 | / |
| **BFS** | 0.095s | 258 | 0.9ms |
| **Dijkstra(**最短路径**)** | 0.122s | 723 | 0.3ms |
| **A\*** | 0.082s | 214 | 0.2-0.4ms |
| **JPS** | / | / | 0.1-0.2ms |

可以看出启发式寻路算法A\*和JPS效率要比非启发式高，这也说明启发式搜索在迷宫这种部分已知环境中效率更高。

* 1. **A-star VS JPS/JPS+**

JPS/JPS+ 算法里只有跳点才会被加入openlist里，排除了大量不必要的点，最后找出来的最短路径也是由跳点组成。这也是 JPS/JPS+ 高效的主要原因。

JPS ：

* 绝大部分地图，使用 JPS 算法都会比 A\* 算法更快，内存占用也更小（openlist里节点少了很多）。
* JPS 在跳点判断上，要尽可能避免递归的深度过大（或者期待一下以后出现避免递归的算法），否则在超大型的地图里递归判断跳点可能会造成灾难。
* JPS 也可以用于动态变化的地图，只是每次地图改变都需要再进行一次 JPS 搜索。
* JPS 天生拥有合并节点（亦或者说是在一条直线里移除中间不必要节点）的功能，但是仍存在一些可以继续合并的地方。
* JPS 只适用于 网格（grid）节点类型，不支持 Navmesh 或者路径点(Way Point)。

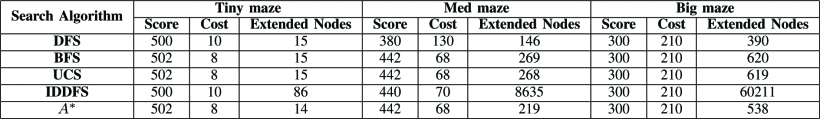
JPS+ ：

* JPS+ 相比 JPS 算法又是更快上一个档次（特别是避免了过多层递归判断跳点），内存占用则是每个格子需要额外记录8个方向的距离数据。
* JPS+ 算法由于包含预处理过程，这让它面对动态变化的地图有天生的劣势（几乎是不可以接受动态地图的），因此更适合用于静态地图。
* JPS+ 预处理的复杂度为 O(n) ，n 代表地图格子数。

JPS/JPS+ 是A\*算法的优秀替代者，绝大部分情况下更快和更小的内存占用已经足够诱人。在GDC 2015 关于 JPS+ 算法的演讲中，Steve Rabin 给出的数据甚至是比A\* 算法快70~350倍。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **算法** | **性能** | **内存占用** | **支持动态地图** | **预处理** | **支持节点类型** |
| A\* | 中等 | 大 | 支持 | 无 | 网格、Navmesh、路径点 |
| JPS | 快 | 小 | 支持 | 无 | 网格 |
| JPS+ | 非常快 | 大 | 不支持 | 有，O(n) | 网格 |

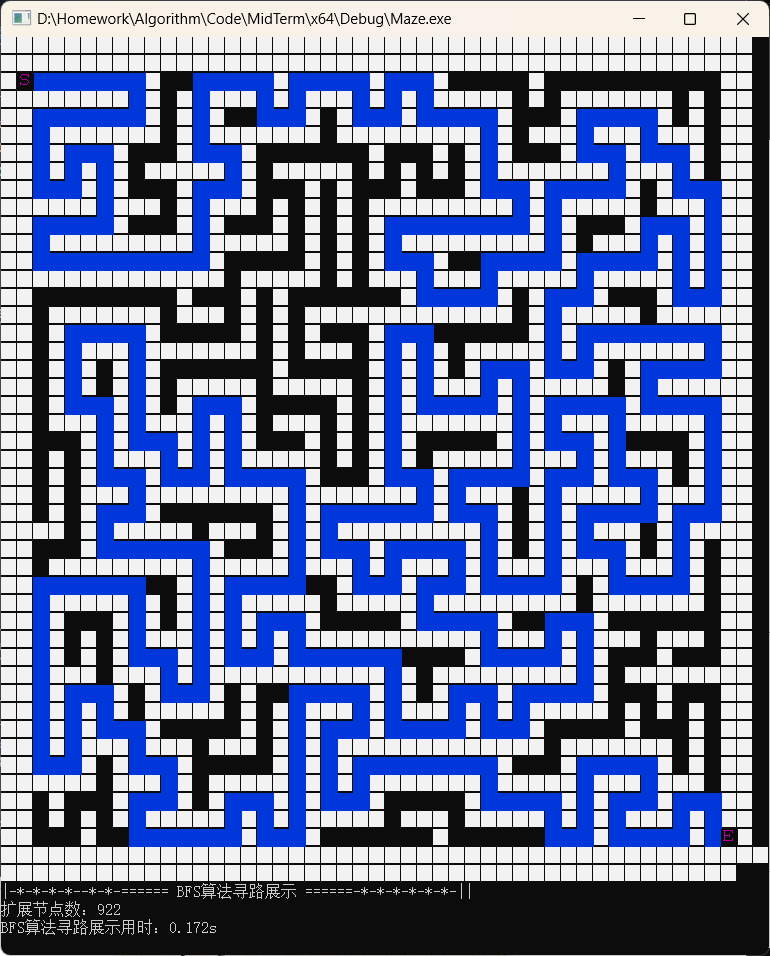
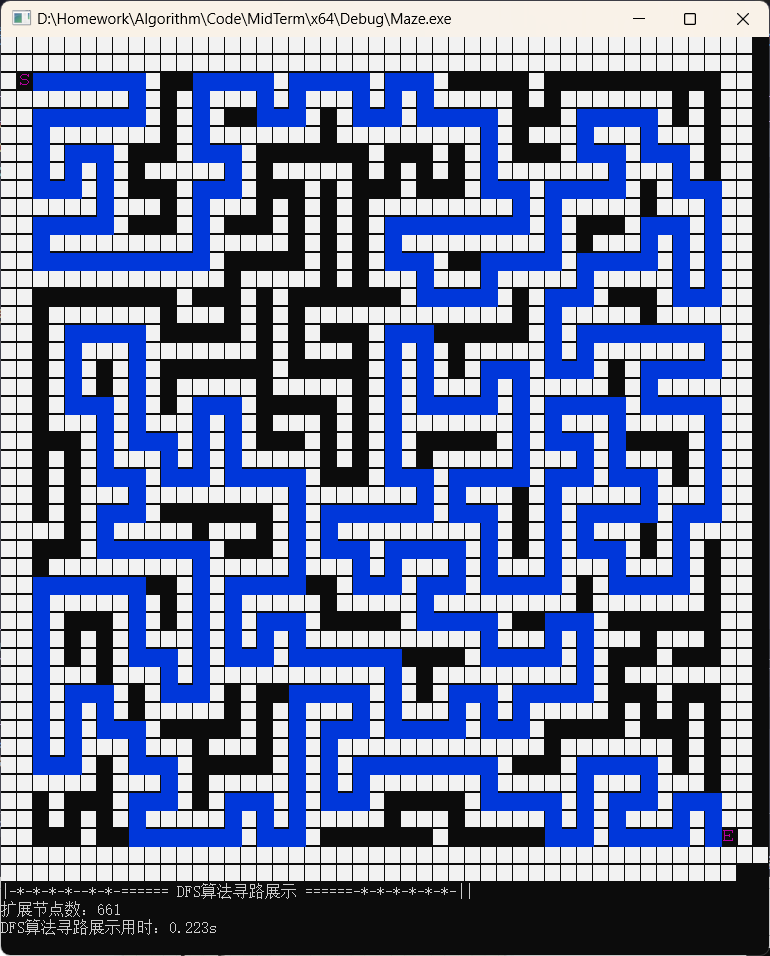
* 1. **用吃豆人游戏比较DFS/BFS/UCS/A\***

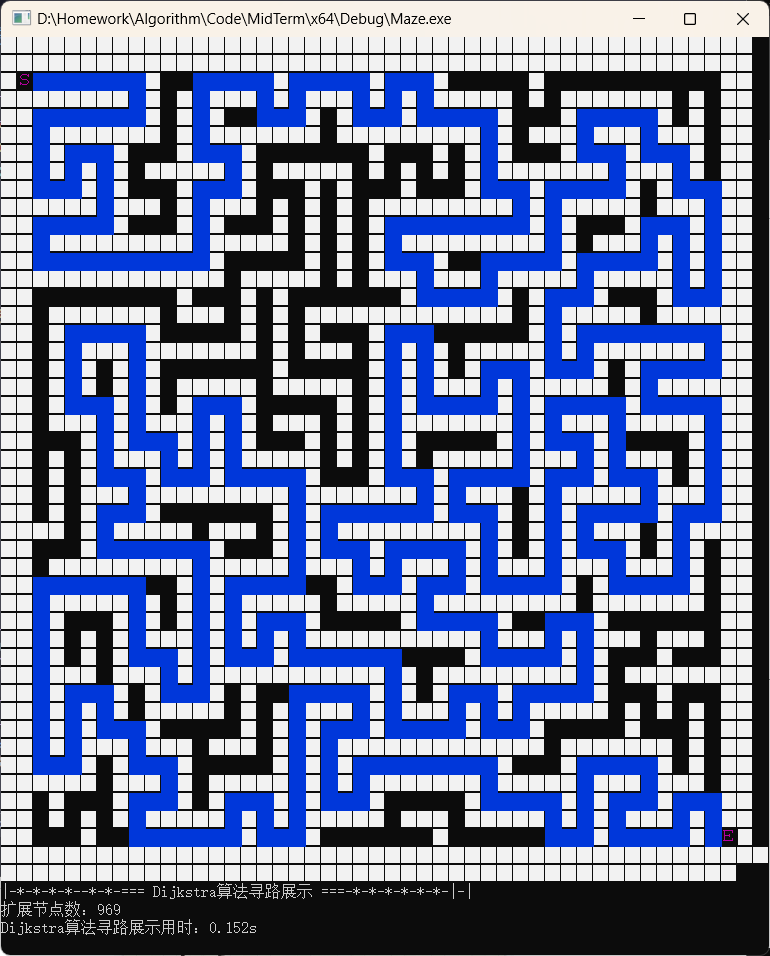
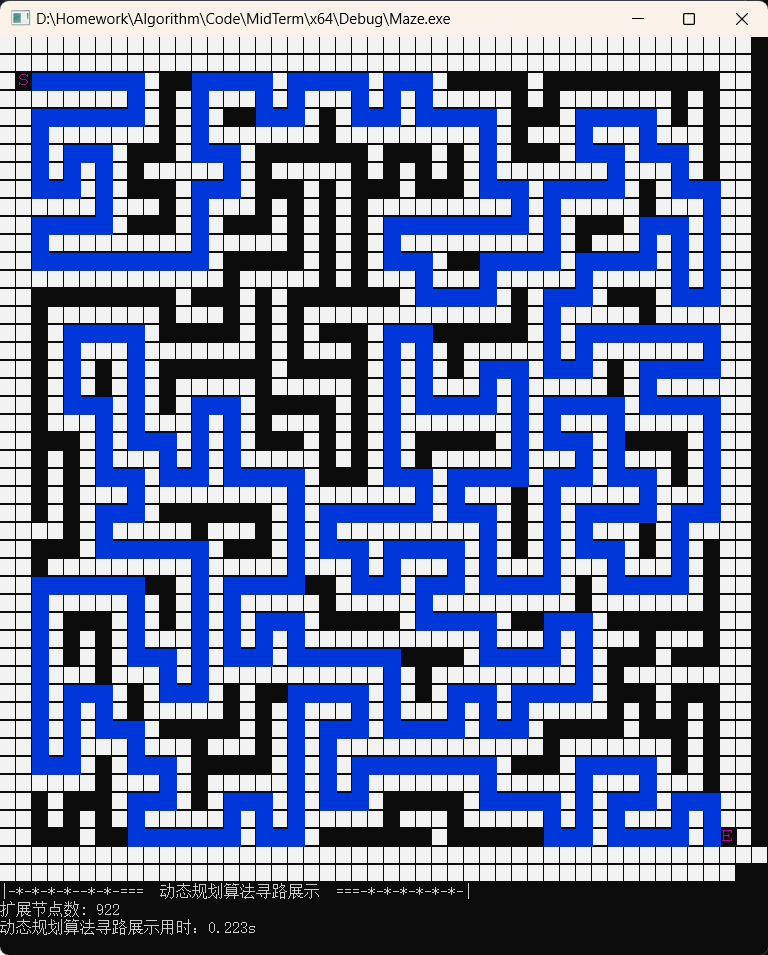
****

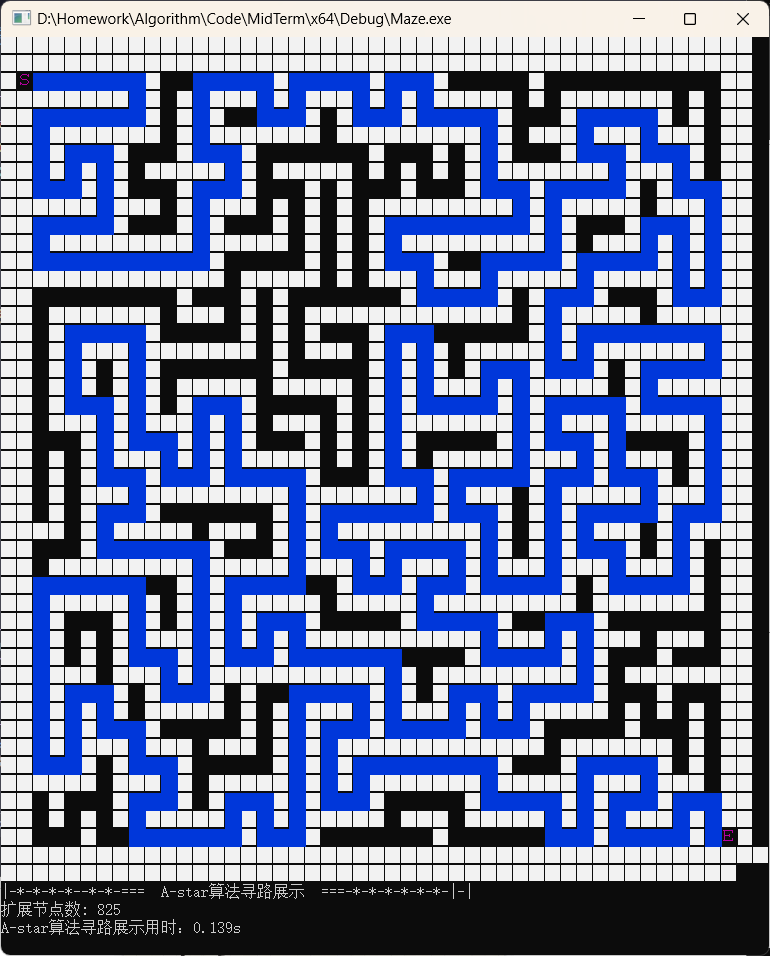
在人工智能课程中框架较完善且可信度较高的吃豆人游戏中，在多种地图下测试几种寻路算法，发现启发式寻路函数的搜索效率和准确度都要高一些。另外可以看出扩展节点数，BFS因为广度优先所以最多，DFS深度优先最少，A\*在两者之间，这与我的程序结果一致。

* 1. **个人实现的算法效率比较**

使用44\*44的二维方格地图测试，因为计时会有波动所以取10次的平均值.pathFinding.js网站地图不一样，仅供参考比较。







|  |  |  |  |
| --- | --- | --- | --- |
| **算法名称** | **用时（mine）** | **扩展节点（mine）** | **PathFinding.js** |
| **DFS** | 0.223s | 661 | / |
| **BFS** | 0.172s | 922 | 0.9ms |
| **DPSearch** | 0.223s | 922 |  |
| **Dijkstra(**最短路径**)** | 0.152s | 969 | 0.3ms |
| **A\*** | 0.139s | 825 | 0.2-0.4ms |
| **JPS** | / | / | 0.1-0.2ms |

可以看出,引入启发函数的A-star算法总比Dijkstra效率高，而随着地图规模增大，Dijkstra效率接近于甚至超过了BFS，但是Dijkstra由于遍历所有节点，扩展的结点数总是最大的。DFS的扩展节点数最少，但是用时最大A\*效率最高，扩展节点数在BFS和DFS之间。另外我的动态规划思想的广度优先遍历按理应该是要比BFS要快，因为子问题可以直接调用，但是可能因为引入了更多的中间数组，所以反而导致速度变慢，后续可以进行优化。

|  |  |  |  |
| --- | --- | --- | --- |
| 算法 | 效率 | 扩展的节点个数 | 适用场景 |
| Dijkstra | 高 | 最多 | 短路径，无需考虑搜索速度 |
| A\* | 高 | 较少 | 启发式函数可用，能够提供较好的估计 |
| BFS | 中等 | 较多 | 简单、无权重图，搜索全图 |
| DFS | 低 | 较少 | 搜索路径长度较深的迷宫，可能会陷入局部最优解 |
| Greedy BFS | 低 | 较少 | 启发式函数可用，但不保证找到最优解 |
| Jump Point Search | 高 | 较少 | 网格地图，需要高效的路径搜索算法 |
| Bidirectional Search | 高 | 较少 | 需要同时从起点和终点进行搜索 |

1. **个人思考与总结**
   1. **个人思考**

在经过调查后发现，A\*算法是一种静态路网中求解最短路径最有效的直接搜索方法，也是解决许多搜索问题的有效算法，算法中的距离估算值与实际值接近，最终搜索速度越快，是最常用的启发式算法。

经典的非启发式寻路算法都有着盲目性，效率不如启发式算法：

1. 对于BFS，它的优点在于可以找到最优的一条路径，缺点是需要遍历整个地图。
2. 对于DFS，它的优点在于不需要遍历整个地图，缺点在于不一定是最优路径。
3. 对于Dijkstra，它的优点在于无差别的遍历当前最短路径，对于查找起始点到任意点的最短路径该算法很有效，缺点是：对于点对点的路径查找很浪费。

对于A\*，它能很快的找到一条相对最优的路径，而且搜索的节点比前三个算法都要少。可以理解为A\*喜寿了DFS和BFS的优点，寻找到的路径优劣程度介于BFS和DFS之间。而现实的工程项目，地图矩阵都较大，使用BFS效率过低，DFS找到的路径又不可靠，所以A\*是最好的选择。

但是A\*虽然具有启发函数，但是也存在效率上与最优化上的不足，本人通过学习经典的寻路算法，以及近年来新提出的优化的寻路算法，提出对于A\*优化的思考。

优化A\*一般从以下四个方面着手：

1. **Openlist开放集合优化**

对于经典的A\*算法，以下针对openlist的5个操作必不可少：

* 1. 添加操作：将节点添加到openlist中
  2. 删除操作：将节点从openlist中删除
  3. 获取长度：获取openlist中节点的个数
  4. 判断是否存在：判断某个节点是否已经保存在openlist中
  5. 排序：对openlist根据每个节点F的值从小到大排序

要提升A\*的速度，可以想办法把这五个操作的时间复杂度降低下来，这也是A\*算法的各种优化算法的核心思想。个人理解，可以使用优先队列等更高效的数据结构来进行优化，这样可以优化A\*的效率，远比自己实现的数据结构效率要高，这可能是因为官方实现有性能优化。

优先队列各种操作的时间复杂度如下：

|  |  |
| --- | --- |
| 优先队列的操作 | 时间复杂度 |
| 添加 |  |
| 删除 |  |
| 排序 |  |
| 获取长度 |  |
| 判断存在与否 |  |

另外，还可以通过优化openlist改变A\*的路径，类似于曼哈顿距离，如果要求只能上下左右移动，且地图是二维栅格，那么从一点到另一点可能存在长度相同的多条路径。但是我们根据实际业务的不同，会想要不同的结果。这两种走势其实和openlist有一定的关系。其中涉及到openlist的排序问题。

为此，我们可以进行优化：如果openlist的排序是稳定的，第一次使用A\*走出某路径1，那么下一次使用A\*也会走出路径1,；如果openlist的排序不稳定，那么从S到G的路径，多次使用A\*的结果是不一样的，可能第一次是路径1，第二次事路径2。当openlist的排序是稳定的时候，那么先后才进入openlist优先队列的节点P1=RIGHT,P2=DOWN,一定P1会先被取出来，这样每次使用A\*获得到的路径也是稳定的；当排序不稳定时，可能会是P2先被取出来。

1. **getNeighbour获取邻居节点优化**

A\*相比于其他非启发式算法的优势在于其可以洞察终点的位置，也就是相当于A\*的眼睛，getNeighbour即为眼睛的实现。

在这里，我们可以增加路径的随机性来让眼界更宽些，通常获取邻居时会把顺序写死，这样无意中就导致了某个方向是优先级最高的策略，参照（1），虽然有多重路径但是可能已经把顺序写死了，寻路就没那么灵活了。我们可以根据周围环境进行概率的分配，这样可以消除邻居的优先次序，让A\*更加智能。

|  |  |  |
| --- | --- | --- |
| **5** | **1** | **6** |
| **2** | **0** | **3** |
| **7** | **4** | **8** |

另一方面，可以寻找更优路径，最基本的A\*获取周围4个邻居，然后向这四个方向走，稍微优化下可以获取周围8个邻居，向8个方向走。A\*的眼界最多也就8个视野，如果想让A\*看的更远，可以在扩大A\*眼界范围，也可以是周围两圈，三圈等等。当然，眼界越大，就意味着A\*要计算的节点也就越多，A\*的速度一定会变慢，优点是可以让A\*找到更优的路径。

许多学者在这方面进行了探索，并取得了很好的成功，比方2011年提出的JPS算法就是针对A\*的getNeighbour函数进行优化的算法。A\*方向为上下左右，JPS则只允许走斜线的情况，在寻路速度上有着大幅度提升，它主要是通过减少邻居节点来提升速度，对邻居节点进行剪枝，删去不必要的邻居。

1. **启发式函数优化**

个人感觉是很重要的部分，因为A\*相对于传统无启发算法的区别就是加了启发函数。没有启发函数它和Dijkstra没有太大区别。启发函数是A\*算法最核心的部分，可以说是A\*的大脑，指导着A\*的下一步走向。A\*的启发式函数如下，我们可以调整代价，构造适合问题的启发函数：

F=G+H+C

G：该点到起点的实际路径长度

H：该点到终点估算的估计长度

C：从当前节点走到该点的格外代价

另外，可以在可穿障碍物上进行优化，有些障碍物的地方是可以走的，但是尽量不要从障碍物上走，也就是可穿障碍物概念。优化策略可以是: 在C函数在判断邻居节点是否是障碍物，如果是障碍物，就增加C的值，从而达到增加F值的目的。对于F函数来说，如果你对A\*要求太高，既要它少拐点，又要它不穿越障碍物，它可能会实现，但是有时候这两个需求是互相冲突的，就导致F函数值的混乱，使得A\*走出非常奇怪的路线,所以需要合理构造启发函数。

1. **Map地图优化**

map可以看成A\*的人生地图，在map上，A\*诞生（起点），随着启发式大脑的计算，预测着对未来的期望，一步一步走向终点，它的轨迹就是A\*在map上路径的体现。

针对map地图的优化，我们可以通过控制维度来提升速度，矩阵越大，意味着数据越多，A\*要搜索的范围就可以越大，A\*搜索的时间也就越长。所以矩阵的大小是降低A\*搜索时间最直接的因素。比方城镇地图，假设城镇100km\*100km，如果一个像素点为1m\*1m这样就要比1mm\*1mm减小几个数量级，搜索速度也快了不少。这里涉及到自动驾驶里高精地图的概念。有的时候不一定精度越高越好，需要看是在什么领域，不同的问题情形应该构造不同的算法。

另一方面，A\*的经典优化策略是分层优化，其思想通俗说就是对map做一个预处理，每个块可以理解为一个map，下面有3\*7个map，每个map在边界上有多个出口和入口，蓝色即表示从一个口到另一个口有通路。



预处理的意思就是：在每个map中，我们使用A\*计算出从起点到终点的路径，把路径提前记录下来。然后将3\*7个小map拼凑成一个大map，这样每个块的路径已经存在，就不需要把map的维度降的很低了。这个方法适用于map很大的情况。

1. **引入人工智能策略**

近年来人工智能飞速发展，为许多领域注入了新的活力。A\*算法中的启发函数很好地体现了智能体的思想，在这学期的另外一门课人工智能原理与技术中，我们使用CS188的配套练习，前面有智能体设计，其中也涉及了相关的寻路算法，比方BFS,DFS,UCS以及A\*等，c188要求使用吃豆人吃掉目标豆子，图中可能存在鬼魂等，但是得益于完善的框架，只需要设计相关算法即可，在这个项目中可以直观体会不同算法的优劣，特别是在加上评估函数，即启发函数后的寻路效率更加高效。

A\*除了传统的改进措施，除了上面之外，还有双向A\*等。然而这些智能性在某些情况下可能不够。在实际问题中，有学者将寻路算法和计算机视觉等方法结合起来，通过计算机视觉技术作为寻路算法的眼睛，取得了很好的效果。

* 1. 神经网络算法是人工智能领域中的一种非常 优秀的算法，它主要模拟动物神经网络行为，进行分布式并行信息处理。但它在路径规划中的应用却并不成功，因为路径规划中复杂多变的环境很难用数学公式进行描述， 如果用神经网络去预测学习样本分布空间以外的点，其效果必然是非常差。尽管神经网络具有优秀的学习能力，但 是泛化能力差是其致命缺点。但因其学习能力强鲁棒性好，它与其他算法的结合应用已经成为路径规划领域研究 的热点。
  2. 遗传算法 ( Genetic Algorithms，简称 GA) 是当代人工智能科学的一个重要研究分支，是一种模拟达尔文 遗传选择和自然淘汰的生物进化过程中的计算模型。它的思想源于生物遗传学和适者生存的自然规律，是按照基 因遗传学原理而实现的一种迭代过程的搜索算法。最大 的优点是易于与其他算法相结合，并充分发挥自身迭代的优势，缺点是运算效率不高，不如蚁群算法有先天优势，但其改进算法也是目前研究的热点。
  3. 粒子群算法 ( POS) 也是一种迭代算法，它模拟鸟群飞行捕食行为，和遗传算法相似，它也是从随机解出发，通过迭代寻找最优解，也是通过适应度来评价解的 品质，但它比遗传算法规则更为简单，它没有遗传算法的 “交叉”和“变异”操作，有记忆功能，它通过追随当前搜索到的最优值来寻找全局最优。具有算法简洁、易于实现、 鲁棒性好、算法对种群大小不十分敏感、收敛速度快等优点，但易陷入局部最优解。
  4. **总结**

在路径规划的研究中，无论是点到点的路径规划还是全覆盖路径规划，已知环境下无障碍物的路径规划问题都比较成熟。而在未知环境下有障碍物的路径规划也取得了重大进展。然而，每个具体规划算法都存在一些不足之处，因此路径规划研究的重点依然是新的高效的改进路径规划算法。另外，混合路径规划也会成为未来路径规划研究发展的方向。具体表现如下几个方面：

结合遗传算法和神经网络利用神经网络控制机器人的运动规则，利用神经元的传感器获取未知环境的信息，利用遗传算法实现神经网络的权值设置，从而实现在未知环境下机器人的路径规划。

将蚁群算法与人工神经网络相结合，可以减少空间复杂度，同时提高路径规划准确率。

将多传感器应用于机器人中是未知环境信息路径规划的趋势。该方法能获取更准确的环境信息，更准确地障碍物和机器人定位，实现更好的路径规划。

多机器人合作进行路径规划，成为人们研究的一个新的热点之一。如何划分未知环境，如何对机器人进行分工，如何设置机器人的体系结构及机器人间的通信方式等成为新的研究问题。该问题的解决能更好地减少机器人间的冲突问题，以便进行更好的路径规划。

寻路算法是一个发展了很久很久的方向，有许多经典的算法，以及其优化，通过本次实验我学习了很多，对于算法的分析和设计更加熟练，编程能力也得到了锻炼，但是由于时间原因，一些文章没有深入调研，代码也存在bug，后续我会继续学习，继续完善。

1. **参考文献**
2. An application of reinforcement learning techniques in traditional pathfinding[EB/OL]. [2024-05-13].
3. XIANG D, LIN H, OUYANG J, 等. Combined improved a\* and greedy algorithm for path planning of multi-objective mobile robot[J/OL]. Scientific Reports, 2022, 12(1): 13273. DOI:10.1038/s41598-022-17684-0.
4. SANG Y, CHEN X, CHEN Q, 等. A route planning for oil sample transportation based on improved a\* algorithm[J/OL]. Scientific Reports, 2023, 13(1): 22041. DOI:10.1038/s41598-023-49266-z.
5. SAGMING M N, HEYMANN R, HURWITZ E. Visualising and solving a maze using an artificial intelligence technique[C/OL]//2019 IEEE AFRICON. Accra, Ghana: IEEE, 2019: 1-7[2024-05-14]. DOI:10.1109/AFRICON46755.2019.9134044.
6. LIU Z, LIU H, LU Z, 等. A dynamic fusion pathfinding algorithm using delaunay triangulation and improved a-star for mobile robots[J/OL]. IEEE ACCESS, 2021, 9: 20602-20621. DOI:10.1109/ACCESS.2021.3055231.
7. LIU L, WANG B, XU H. Research on path-planning algorithm integrating optimization a-star algorithm and artificial potential field method[J/OL]. Electronics, 2022, 11(22): 3660. DOI:10.3390/electronics11223660.
8. KONG R, TONG X. Anytime dynamic heuristic search for suboptimal solution on path search[C/OL]//2020 13th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI). Chengdu, China: IEEE, 2020: 1070-1074[2024-05-14]. https://ieeexplore.ieee.org/document/9263589/. DOI:10.1109/CISP-BMEI51763.2020.9263589.
9. KONG R, TONG X. Anytime dynamic heuristic search for suboptimal solution on path search[C/OL]//2020 13th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI). Chengdu, China: IEEE, 2020: 1070-1074[2024-05-14]. https://ieeexplore.ieee.org/document/9263589/. DOI:10.1109/CISP-BMEI51763.2020.9263589.
10. KAPI A Y, SUNAR M S, ALGFOOR Z A. A comparison of pathfinding algorithm for code optimization on grid maps[J/OL]. International Journal of Advanced Computer Science and Applications, 2022, 13(12)[2024-05-13]. http://thesai.org/Publications/ViewPaper?Volume=13&Issue=12&Code=IJACSA&SerialNo=47. DOI:10.14569/IJACSA.2022.0131247.
11. FOEAD D, GHIFARI A, KUSUMA M B, 等. A systematic literature review of a\* pathfinding[C/OL]//BUDIHARTO W, KURNIAWAN A, SUHARTONO D, 等. 5TH INTERNATIONAL CONFERENCE ON COMPUTER SCIENCE AND COMPUTATIONAL INTELLIGENCE 2020: 卷 179. Amsterdam: Elsevier Science Bv, 2021: 507-514[2024-05-13]. https://www.webofscience.com/wos/alldb/full-record/WOS:000654256300064. DOI:10.1016/j.procs.2021.01.034.
12. ERKE S, BIN D, YIMING N, 等. An improved a-star based path planning algorithm for autonomous land vehicles[J/OL]. International Journal of Advanced Robotic Systems, 2020, 17: 172988142096226. DOI:10.1177/1729881420962263.
13. CHEN Y, WANG P, LIN Z, 等. Global path planning method by fusion of a-star algorithm and sparrow search algorithm[C/OL]//2022 IEEE 11th Data Driven Control and Learning Systems Conference (DDCLS). Chengdu, China: IEEE, 2022: 205-209[2024-05-14]. https://ieeexplore.ieee.org/document/9858435/. DOI:10.1109/DDCLS55054.2022.9858435.
14. CHEN R. Research on the development path of higher education model innovation based on quadratic planning algorithm[J/OL]. Applied Mathematics and Nonlinear Sciences, 2024, 9(1).
15. BROWN B S. A thesis submitted in partial satisfaction of the requirements for the degree master of applied statistics[J].
16. 张广林, 胡小梅, 柴剑飞, 等. 路径规划算法及其应用综述[J/OL]. 现代机械, 2011(5): 85-90. DOI:10.13667/j.cnki.52-1046/th.2011.05.014.
17. 杨科选. 人工智能寻路算法及其在游戏中的应用研究[D/OL]. 中南大学, 2010[2024-05-12].
18. <https://www.redblobgames.com/pathfinding/a-star/introduction.html>
19. <http://qiao.github.io/PathFinding.js/visual/>
20. <https://blog.csdn.net/qq_34446253/article/details/51427423>
21. **附录**

代码文件在压缩包中