A\*算法族

王宇涵 赵冉 牟新 李禹艺 邢泽斌

**摘要：**A\*算法是一种经典的路径搜索算法，在图论、计算机科学、机器人等领域具有广泛的应用。近年来，随着自动驾驶技术的发展，A\*算法在该领域得到了广泛的应用和研究。本文介绍了 A\*算法的基本思想和工作原理，并探讨了它在实际应用中的优点和局限性。同时，我们调研了 A\*算法的改进和优化，提高了它的效率和准确性。最后，我们通过对 A\*算法的仿真证明了它在实际应用中的效果和优势。

**关键字**：关键词:A\*算法;路径搜索;图论;计算机科学;自动寻路;

**应用背景：**自动驾驶技术是目前汽车领域的热点之一。自动驾驶车辆需要能够在复杂的城市道路和高速公路上自主导航，这需要车辆具备良好的路径搜索和规划能力。在自动驾驶领域，寻路问题是一个至关重要的问题，而 A\*算法是一种常见的路径搜索算法，已经被广泛应用于自动驾驶车辆的导航中。从业人员提出了不少基于 A\*算法的路径搜索方法，并通过结合深度学习技术和图神经网络，对复杂的城市道路和高速公路进行有效的路径规划和搜索。同时，针对 A\*算法的计算效率问题，提出了基于多智能体的协作搜索方法，使得自动驾驶车辆之间的路径搜索更加高效和智能化。目前，A\*算法仍然是自动驾驶领域的研究热点，通过研究人与的努力，他们希望为自动驾驶领域的寻路问题提供一种新的解决方案，为自动驾驶技术的发展做出贡献。

1. **A\*算法族介绍**

**这一部分将对A\*算法族中的几种重要算法进行总结和报告**

1. **A\*算法**

**简介：**A\*搜索算法是一种在图形平面上，有多个节点的路径，求出最低通过成本的算法。常用于游戏中的NPC的移动计算，或网络游戏中机器人的移动计算。该算法综合了best-first search 和Dijkstra算法的优点：在进行启发式搜索提高算法效率的同时，可以保证找到一条最优路径。

**过程**：定义起点 ，终点 ，从起点（初始状态）到任意点的实际距离为，从任意点到终点（最终状态）的最短距离估计函数 , 以及每个点的估价函数。为启发函数

A \* 算法每次从优先队列中取出一个 最小的元素，然后更新相邻的状态。具体的更新过程请看伪代码。这里的估价函数遵循以下特性：

* 如果为，即只计算任意顶点到目标的评估函数 ，而不计算从起点到该点的距离，则算法转化为使用贪心策略的best-first search 算法，速度最快，但可能得不出最优解
* 如果不大于到目标点的实际距离，则一定可以求出最优解，而且越小，需要计算的节点越多，算法效率越低。常见的评估函数——欧几里得距离、曼哈顿距离、切比雪夫距离。

如果

为0,即只需求出起点到任意顶点的最短路径,而不计算任何评估函数,则转化为最短路问题问题,即Dijkstra算法,此时需要计算最多的顶点

**伪代码**：

1. OPEN = priority queue containing START

2. CLOSED = empty set

3. while lowest rank in OPEN is not the GOAL:

4. current = remove lowest rank item from OPEN

5. add current to CLOSED

6. for neighbors of current:

7. cost = g(current) + movementcost(current, neighbor)

8. if neighbor in OPEN and cost less than g(neighbor):

9. remove neighbor from OPEN, because new path is better

10. if neighbor in CLOSED and cost less than g(neighbor):

11. remove neighbor from CLOSED

12. if neighbor not in OPEN and neighbor not in CLOSED:

13. set g(neighbor) to cost

14. add neighbor to OPEN

15. set priority queue rank to g(neighbor) + h(neighbor)

16. set neighbor's parent to current

17.

18. reconstruct reverse path from goal to start

19. by following parent pointers

20.

1. **迭代加深搜索**

**简介**：迭代加深是一种每次限制搜索深度的深度优先搜索。迭代加深搜索的本质还是深度优先搜索，只不过在搜索的同时带上了一个深度，当 达到设定的深度时就返回，一般用于找最优解。如果一次搜索没有找到合法的解，就让设定的深度加一，重新从根开始。既然是为了找最优解，为什么不用 BFS 呢？我们知道 BFS 的基础是一个队列，队列的空间复杂度很大，当状态比较多或者单个状态比较大时，使用队列的 BFS 就显出了劣势。事实上，迭代加深就类似于用 DFS 方式实现的 BFS，它的空间复杂度相对较小。当搜索树的分支比较多时，每增加一层的搜索复杂度会出现指数级爆炸式增长，这时前面重复进行的部分所带来的复杂度几乎可以忽略，这也就是为什么迭代加深是可以近似看成 BFS 的。

**过程**：首先考虑将问题表达成广度优先搜索模型，如果发现广度优先搜索算法在空间上不够优秀，且需要寻找最优解时，则考虑下面的过程。首先设定一个较小的深度作为全局变量，进行 DFS。每进入一次 DFS，将当前深度加一，当发现 大于设定的深度  就返回。如果在搜索的途中发现了答案就可以回溯，同时在回溯的过程中可以记录路径。如果没有发现答案，就返回到函数入口，增加设定深度，继续搜索。

**伪代码**：

1. // Returns true if target is reachable from

2. // src within max\_depth

3. bool IDDFS(src, target, max\_depth)

4. for limit from 0 to max\_depth

5. if DLS(src, target, limit) == true

6. return true

7. return false

8. bool DLS(src, target, limit)

9. if (src == target)

10. return true;

11. // If reached the maximum depth,

12. // stop recursing.

13. if (limit <= 0)

14. return false;

15. foreach adjacent i of src

16. if DLS(i, target, limit?1)

17. return true

18. return false

1. **IDA\*算法**

**简介：**IDA\*算法是一种图遍历和路径搜索算法，可以在加权图中找到指定起始节点和一组目标节点中的任意成员之间的最短路径。 它是迭代加深搜索的一种变体，借鉴了 A\* 搜索算法的思想，使用启发式函数来评估到达目标的估价函数。 由于是深度优先搜索算法，其内存使用率低于A\*，但不同于普通的迭代加深搜索，IDA\*专注于探索最有希望的节点，因此不会在搜索树中处处走到相同的深度 。IDA\* 是 A\* 的内存受限版本。 它完成了 A\* 所做的一切，它具有 A\* 的最佳特性以找到最短路径，但它使用的内存比 A\* 少。

**过程：IDA\***算法的基本过程包括初始化、遍历未访问的节点和遍历已访问的节点三个阶段。在初始化阶段，算法将起点设置为源节点，终点设置为目标节点，并计算起点到目标节点的最短距离。在遍历未访问的节点阶段，算法对于每个未访问的节点 n，计算从起点到该节点的最短距离。在遍历已访问的节点阶段，算法比较该节点到目标节点的距离和该节点到起点的距离，如果该节点到目标节点的距离更短，则将其标记为已访问，并更新到目标节点的最短距离。最后，算法返回起点到目标节点的最短距离。IDA\*算法的贪心策略能够保证得到全局最优解。

**伪代码：**

1. Procedure IDA\_STAR(StartState)

2. Begin

3. PathLimit := H(StartState) - 1;

4. Succes := False;

5. Repeat

6. inc(PathLimit);

7. StartState.g = 0;

8. Push(OpenStack, StartState);

9. Repeat

10. CurrentState := Pop(OpenStack);

11. If Solution(CurrentState) then

12. Success = True

13. Elseif PathLimit >= CurrentState.g + H(CurrentState) then

14. For each Child(CurrentState) do

15. Push(OpenStack, Child(CurrentState));

16. until Success or empty(OpenStack);

17. until Success or ResourceLimtsReached;

18. end;

1. **D\*算法**

A\* 在静态路网中非常有效（very efficient for static worlds），但不适于在动态路网，环境如权重等不断变化的动态环境下。 D\*是动态A\*（D-Star,Dynamic A Star） 卡内及梅隆机器人中心的Stentz在1994和1995年两篇文章提出，主要用于机器人探路。是火星探测器采用的寻路算法。

D\*算法的主要方法如下：

1.先用Dijstra算法从目标节点G向起始节点搜索。储存路网中目标点到各个节点的最短路和该位置到目标点的实际值h,k（k为所有变化h之中最小的值,当前为k=h。每个节点包含上一节点到目标点的最短路信息1(2),2(5),5(4)，4（7）。则1到4的最短路为1-2-5-4。

原OPEN和CLOSE中节点信息保存。

2.机器人沿最短路开始移动，在移动的下一节点没有变化时，无需计算，利用上一步Dijstra计算出的最短路信息从出发点向后追述即可，当在Y点探测到下一节点X状态发生改变，如堵塞。机器人首先调整自己在当前位置Y到目标点G的实际值h(Y)，h(Y)=X到Y的新权值c(X,Y)+X的原实际值h(X).X为下一节点(到目标点方向Y->X->G），Y是当前点。k值取h值变化前后的最小。

3.用A\*或其它算法计算，这里假设用A\*算法,遍历Y的子节点，点放入CLOSE,调整Y的子节点a的h值，h(a)=h(Y)+Y到子节点a的权重C(Y,a),比较a点是否存在于OPEN和CLOSE中，方法如下：

while ()

{

从OPEN表中取k值最小的节点Y;

遍历Y的子节点a, 计算a的h值 h(a) = h(Y) + Y到子节点a的权重C(Y, a)

{

if (a in OPEN) 比较两个a的h值

if (a的h值小于OPEN表a的h值)

{

更新OPEN表中a的h值; k值取最小的h值

有未受影响的最短路经存在

break;

}

if (a in CLOSE) 比较两个a的h值 //注意是同一个节点的两个不同路径的估价值

if (a的h值小于CLOSE表的h值)

{

更新CLOSE表中a的h值; k值取最小的h值; 将a节点放入OPEN表

有未受影响的最短路经存在

break;

}

if (a not in both)

将a插入OPEN表中;　//还没有排序

}

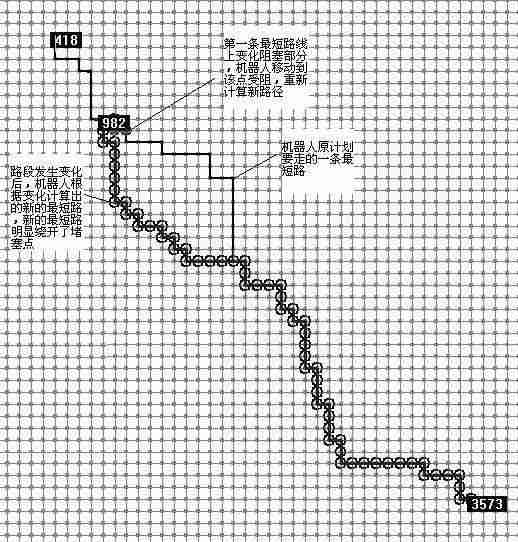
放Y到CLOSE表；

OPEN表比较k值大小进行排序；

}

机器人利用第一步Dijstra计算出的最短路信息从a点到目标点的最短路经进行。

D\*算法在动态环境中寻路非常有效，向目标点移动中，只检查最短路径上下一节点或临近节点的变化情况，如机器人寻路等情况。对于距离远的最短路径上发生的变化，则感觉不太适用。



上图是Drew在4000个节点的随机路网上做的分析演示，细黑线为第一次计算出的最短路，红点部分为路径上发生变化的堵塞点，当机器人位于982点时，检测到前面发生路段堵塞，在该点重新根据新的信息计算路径，可以看到圆圈点为重新计算遍历过的点，仅仅计算了很少得点就找到了最短路，说明计算非常有效，迅速。绿线为计算出的绕开堵塞部分的新的最短路径。

1. **Hybrid A\*算法**

**解决的问题：**

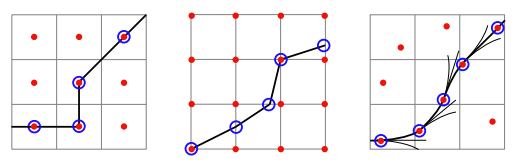
该算法解决的问题是：假设无人车有充分的感知和定位能力，则其能够在线重新规划，生成障碍物地图，并且能够在未知环境中行驶。

**算法的大致过程：**

在连续坐标系下进行启发式搜索，改进了A\*算法，第二个阶段用数值非线性优化的方法（即共轭梯度下降法）对生成路径的“质量”进行提升，使其至少是局部最优的。

**STAGE 1: 启发式搜索+损失函数设计**

第一个阶段其实是对传统的A\* 算法进行改进，与之不同的是，hybrid A\* 是在连续坐标系下进行启发式搜索，并且能够保证生成的轨迹满足车辆非完整性约束（下文对车辆的非完整性约束进行补充），但算法运行过程中该路径并不一定是全局最优的，尽管如此，这条路径是在全局最优解的“附近”（这里的全局最优解指的是由传统A\* 算法生成的）。下图（来自文献1）可以很好地解释传统A\* 与hybrid A\* 的异同。两种算法都是基于网格世界的（grid world），A\* 是赋予每个网格的中心点相应的损失并且算法只访问这些中心点，而hybrid A\* 是先在这些网格中挑选满足车辆3D连续状态的点，并将损失赋值给这些点。（下图中间的图是用其他A\* 变体算法实现的）



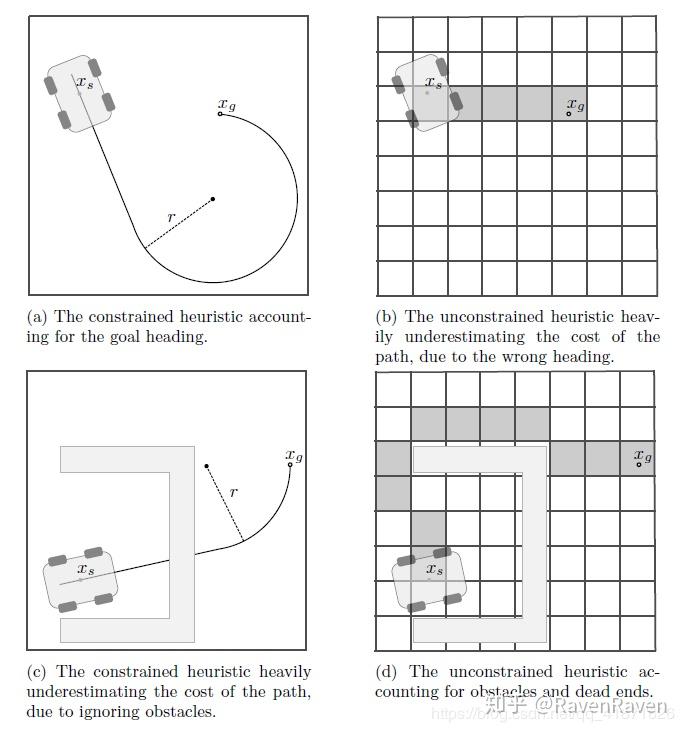
**搜索中所用的两种启发函数**

在传统的A\* 算法中，由于启发函数的选取不同，运行算法后节点扩张（expansion）的效率也就不同（目的是希望算法在遍历最少的节点的情况下找到最优路径）。传统的A\* 算法的启发函数一般是2D欧几里得距离，而hybrid A\* 算法构造了两个启发函数。

第一个启发函数是Constrained heuristics ，只考虑车辆的非完整性约束而不考虑障碍物（优点是相比直接用欧几里得距离损失要好一个数量级）。该启发函数忽略了环境中的障碍物等信息，只考虑车辆的运动学特性，从终止点,开始，计算从该点到其他点的最短路径。具体的返回值就是：

2. 第二个启发函数是第一个的对偶，称为Unconstrained heuristics，只考虑障碍物信息而不考虑车辆的非完整性约束条件（优点是引入该启发函数后能够发现2D空间中所有的U形障碍物和死胡同dead end）。随后使用2D动态规划的方法（其实就是传统的2D A\* 算法）计算每个节点到终点的最短路径。

算法使用的损失函数就是两种启发函数的最大值。下图很好地解释了两种启发函数：图a,c是第一种启发函数下生成的路径，可以看出是连续的；而图b,d则是在第二种启发函数下生成的离散路径（与传统A\* 算法得到的结果是类似的）



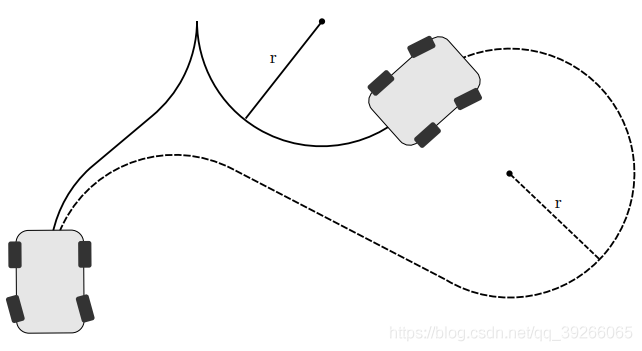
路径节点扩张的过程

论文中称节点扩张的过程为Analytic Expansions（解析扩张？）。首先状态的表示为：

，其中

是节点的位置，而

是节点的朝向（heading），在前文所述的搜索过程中，使用到的是离散的控制动作（control actions, or steering），因此之前每个网格中的连续状态点是不可达的。为了进一步改进搜索速度和提高准确度，这就可以利用Reed-Shepp曲线。A\* 的搜索过程都是用直线相连接，而hybrid A\* 则是在与网格精度一致的前提下（对应某一小段时间）使用三种控制动作：最大左转，最大右转，不转向来生成路径，因此该路径是一些受车辆转弯半径约束的圆弧和直线。

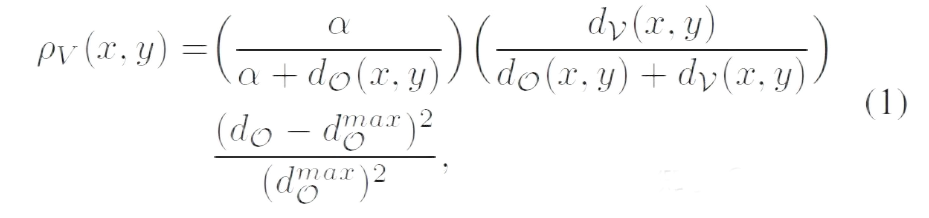


以上过程会生成一些子节点，在此基础上，假设不考虑环境（对应第一个启发函数），算法会通过计算从起点到终点的最优Reed-Shepp曲线的方式，再生成一个额外的子节点；之后算法基于现有的障碍物地图对该路径进行碰撞检测，无碰撞路径对应的点会加到扩张树中。可以看出与直线相比，Reed-Shepp曲线的计算量是很大的。所以论文中作者使用简单的selection rule，在每N个节点中选取一个计算Reed-Shepp曲线（这里的N随启发函数递减而减少，即越发靠近终点时，N越小）。下面这两张图也很好的解释了扩张过程。

**路径损失函数的设计**

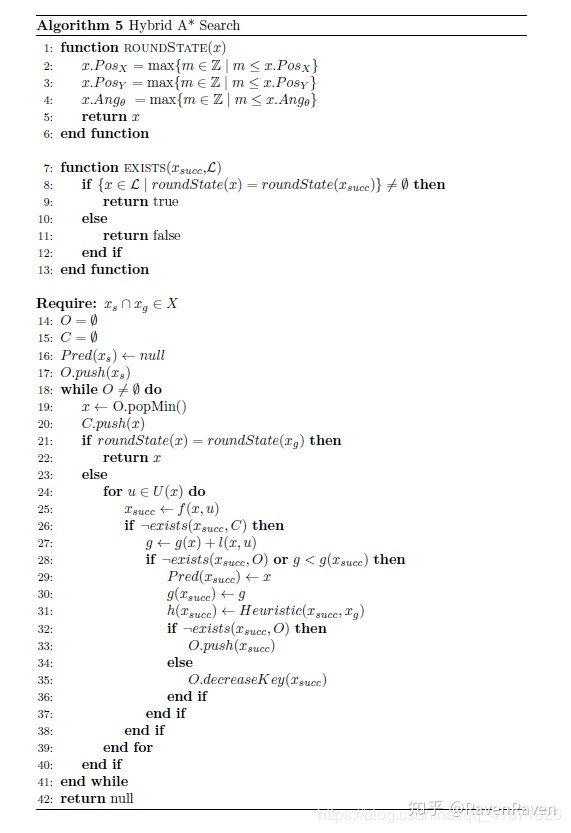
损失函数的设计困难在于所生成路径的要求太多：首先路径长度或损失应该是接近最优的；路径必须是光滑的；生成的路径必须与障碍物保持一定的距离（传统的人工势场法的缺点是在狭窄路段构造了高势场，使得机器人或车辆无法通过）

**解决方法：**构造Voronoi势场函数（具体公式是下列三项连乘，voronoi场的物理意义我还没有搞清楚，之后查下相关资料），voronoi场的值随着导航中所有可行空间的大小成比例缩放。

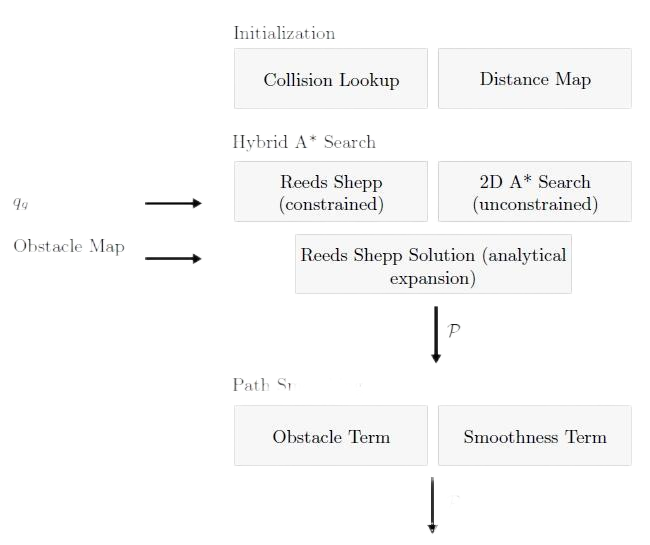


**hybrid A\* 算法伪代码及实现流程**

与A\*算法类似，算法也是维护两个列表，一个open list, 一个是closed list。算法的结束条件是：open list为空或者已经搜索到终点。Line 21: 算法不一定会准确搜索到终点，因此引入RoundState函数，在判断当前节点是否到达终点之前对此进行估算。如果没有达到终点，算法会通过执行动作空间中的所有动作对路径节点进行扩张。如果生成的节点不在C中（也就是没有被算法遍历过），则计算cost-so-far；如果生成的节点不在O中（已被遍历过）或者所得到的cost-so-far小于当前节点已有的cost，这时用当前得到的较小的cost更新cost-so-far



**算法实现的流程：** 这里是第二篇文献中的实现流程，算法的输入为事先定义好的障碍物地图，经过以上的两步：hybrid A\* 搜索和路径平滑之后在rviz中显示路径。



1. **ARA\*算法**

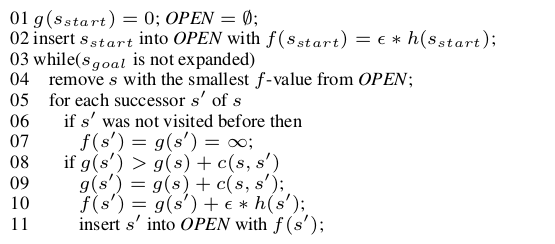
膨胀启发式A\*算法（A\* search with inflated heuristics, 以下称为“加权A\*”）是一种被证明在多数情况都更快但是不能保证找到最优解的算法，加权是指在传统A\*算法的启发式函数中乘上一个大于1的膨胀因子 ，同时该算法还可以通过 来表征对应次优解（可能是最优解，下文的次优解都有此含义不再赘述）的好坏。因此，为了构造一个可以评估次优解优劣的任意时刻规划算法，我们可以迭代运行加权A\*，同时不断减小 使其次优解不断向最优解靠近。这一简单想法就可以得到一系列带有次优性因子（即膨胀因子 ，用来衡量前文所说的次优解的优劣）的次优解。但是只有这样的话仍然无法解决问题，因为简单的重复运行加权A\*会有大量已经被计算过的节点被重复计算，这会浪费大量的计算时间。增量启发式搜索（incremental heuristic search）可以避免这样的问题，但是诸多解的次优性因子又无法保证。

为此提出了ARA\*（Anytime Repairing A\*）算法，这是一种高效的任意时刻规划算法，仍然采用迭代运行加权A\*的思想，但是在保留次优性因子的前提下重用了之前计算的结果。对于之前已经计算过的且结果正确的节点不用重复计算的好处就是算法效率的大大提升，这正是我们所期待的。我们也做了两种不同情况下的算法测试：一是六自由度的模拟运动机械臂，以及移动机器人的路径规划问题。

我们所知的唯一一个任意时刻启发式搜索算法是Anytime A\*，其也是通过先进行加权A\*再优化的方法。但是该算法的问题在于除了首次搜索时可以将给定的膨胀因子视为次优性因子外，不能表征其他次优解的好坏。我们的实验中也说明了ARA\*可以逐步降低解的范围（提高解的质量），而且贼几把快。ARA\*的另一个优点是可以证明它在第一次搜索中对于每一个节点至多计算一次，而不像上面的那个算法。这个特点可以使ARA\*在得到第一个解之前就知道计算量的极大值是多少，这很牛逼。不过，之后我们会谈到Anytime A\*也有一些很牛逼的地方，我们可以用在ARA\*上。

**加权A\*：**一致性是指启发式函数的值不大于到目标点的真实代价距离，这保证了启发式函数的可采纳性（admissible）。膨胀式启发函数（指用大于一的数与启发式函数

相乘）可以减少扩展节点的数量，使得搜索更快。但是这样就丧失了算法的可采纳性，得到的结果不再能被保证是最优解。为了方便和之后我们所给出的ARA\*相比较，我们在图1中给出了加权A\*算法的伪代码。



当 为1时加权A\*就变成了标准的A\*算法，就可以得到问题的最优解。当>1 时只能得到次优解，但是解的次优性是被所决定的：解的长度不大于最优解的倍。

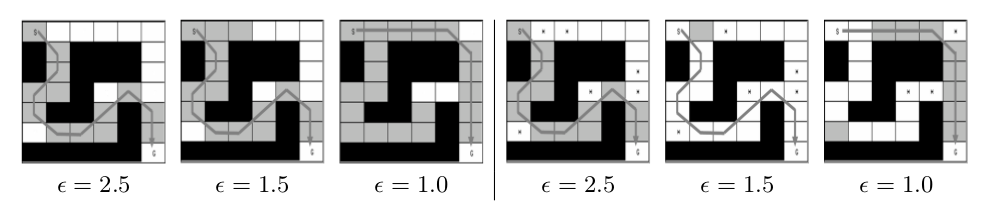
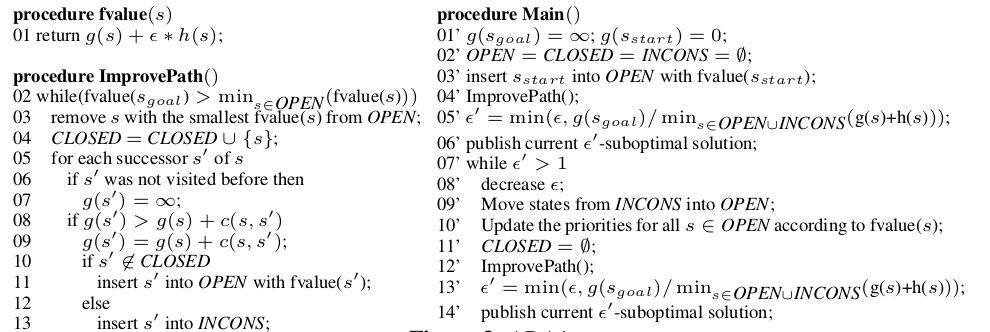


图2中的左边三幅显示了加权A\*在权重分别为2.5,1.5和1.0（无膨胀）时的搜索操作，在这个简单例子中我们使用八邻接的方式，黑色区域为障碍物。S代表起点，G代表目标点。相邻节点的代价距离为1。启发式函数为节点距目标点x和y值中的较大值（切比雪夫距离）。被扩展的节点用灰色表示（A\*算法在扩展到目标点之前一个节点就会停止，所以，在图中目标点并不是灰色）。得到的路径用灰色箭头表示。加权A\*扩展的节点数要比=1 的标准A\*少，但是得到的结果为次优解。

**ARA\*：搜索结果的重复使用:**

ARA\*多次执行A\*算法，首先选取一个较大的进行计算，之后在计算之前将逐渐减小直到=1。这样得到的每一个解都受到对应 的约束。每次减小 后都从头开始运行一遍A\*显然是不划算的，现在我们来介绍ARA\*如何能是之前的结果被重复利用，从而使计算量大大减少的。首先我们引入了一个名为 ImprovePath 的函数，用来对于给定的重新计算（recompute）出一条路径。之后我们介绍ARA\*的主函数，在主函数中，ImprovePath 函数随着的减小被反复调用。



有一致性启发式函数的A\*算法对于任何一个节点都不会扩展多余一次，但是当 被设置为大于一时，一致性可能会受到破坏，这就导致A\*可能对某些节点重复拓展。实际上，如果我们加入每个节点只扩展一次的限制，那么 仍然可以作为次优性的界限。为此，我们通过检测那些g值降低的节点，只有其未曾被扩展时才将其加入到OPEN列表中。已经扩展过的节点都保存在CLOSED列表中。

加入这个步骤后，对于任意节点都至多被扩展一次，但是这样OPEN列表就不能保证包含所有的非一致节点，只能保存那些未曾被扩展的局部不一致点。但是，我们必须将所有的非一致节点信息都保留下来，因为这是之后迭代搜索的基础。因此我们在算法中加入了 INCONS 列表来保存不在OPEN列表中的非一致节点（图12,13行）。这样， OPEN 和 INCONS 列表的结合就可以准确地将所有局部不一致点储存起来，并在每一次新的搜索中将它们作为非一致性传播的起点。

ImprovePath 函数和A\*算法仅有的不同就是终止条件。因为 ImprovePath 函数要用到过去搜索的结果，目标节点可能永远不会出现局部不一致，因此它可能永远不会被放入OPEN列表。结果就是A\*的终止条件此时变成无效的。不过，A\*搜索会在目标节点的f值等于OPEN列表中的最小f值时停止，即我们在 ImprovePath 函数中所使用的终止条件（图3,第2行）。这样的终止条件使得目标节点不会被扩展，同样与目标节点的f值相同的节点也可能不被扩展。（但是，在ARA\*算法中我们再将f值作为变量因为每一次调用 ImprovePath 函数后

都会发生改变，如果要将全部节点的f值重新计算会消耗大量的时间，所以我们设计一个函数来专门用来更新f值，其仅为更新OPEN列表和目标节点的f值。）

**ARA\*: 迭代搜索：**现在我们介绍算法的主循环，用来执行搜索迭代的部分。首先作初始化，之后循环调用不同 值的加权A\*算法 ImprovePath 函数。在每次调用 ImprovePath 函数之前，都会将 INCONS 表中的数据移入 OPEN 表中。由于 OPEN 的节点是按照某种距离测度排序的，所以在与 INCONS 表合并之后必须按照当前f值重新排序（图3，09'-10'行）。这样，在每次调用 ImprovePath 函数后，得到的次优解的代价都不大于最优解的倍。

1. **算法应用**

该部分将描述A\*算法族在现实生活中的应用

1. **A\*在游戏寻路中的应用**

在游戏开发中，寻路可以用来将物体从初始位置以最短路径移动到目的地。

比如说角色扮演游戏中，敌人需要攻击最近的玩家，此时游戏开发者可以使用A-Star算法用于确定敌人可以到达玩家的最短路径。

假设敌人在起始节点，玩家是目的节点，A-Star算法寻路伪代码如下：

|  |
| --- |
| function A\*(start, goal)  open\_list = set containing start  closed\_list = empty set  start.g = 0  start.f = start.g + heuristic(start, goal)  while current is not goal  current = open\_list element with lowest f cost  remove current from open\_list  add current to closed\_list  for each neighbor of current  if neighbor not in closed\_list  neighbor.f = current.g + heuristic(neighbor, goal)  if neighbor is not in open\_list  add neighbor to open\_list  else  openneighbor = neighbor in open\_list  if neighbor.g < openneighbor.g  openneighbor.g = neighbor.g  openneighbor.parent = neighbor.parent  if open\_list is empty  return false  return backtrack\_path(goal) |

在游戏中，玩家每向四周移动一格，敌人利用A\*算法计算离玩家的最近距离从而随之移动一格，而试图不断靠近玩家角色，在游戏过程中位置不断变化，如果敌人设法到达玩家角色所在的位置并以游戏结束为标志，则对角色的追击将停止。

1. **A\*在无线传感器网络节能路由中的应用**

## 背景

传感器网络（WSN）中的节点具有有限的计算能力、有限的能量资源和内存。这些限制限制了要使用的路由算法。

通常在路由算法中，选择最佳路径用于数据从源到目的地的传输。在一段时间内，如果为所有通信选择相同的路径以在快速传输时间方面获得更好的性能，那么位于该路径上的那些节点将更快地耗尽。许多路由算法的问题在于它们以网络中不均匀的能量消耗为代价最小化网络中的总能量消耗。这种方法会导致网络分区，因为作为有效路径一部分的一些节点会更快地耗尽电池能量。在许多情况下，一旦关键节点中的电池电量耗尽，传感器网络的生命周期就结束了。

A-Star算法在WSN中搜索最佳路径的准则不仅是要获得能量消耗最小的路径，而且要看到路径中选择的节点包含足够的剩余能量。这将确保延长传感器网络的整体寿命。

## 2.2. 网络模型

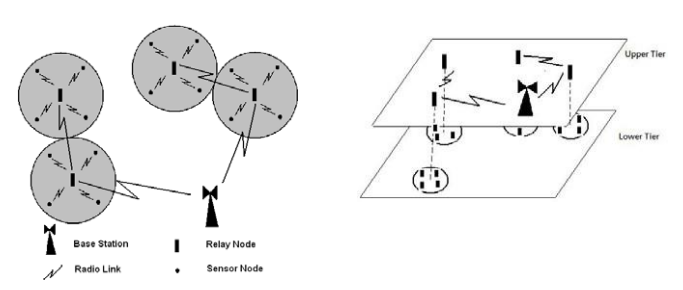


图 1 网络模型

使用如图1所示的传感器网络模型，该模型表明传感器网络被划分成簇，每个簇包含一个中继节点和许多传感器节点。其中，中继节点用于平衡数据收集以延长网络寿命。在分层传感器网络的情况下，中继节点可以用作簇头。还有一个 BS基站，其中信息由传感器节点通过中继节点转发。每个中继节点可以与其他中继节点或与 BS 通信。

## **基于A\*算法的路由**

给定 n 个中继节点的集合，编号从 1 到 n，和一个 BS，编号为 n+1 ，及其他们的位置，A-Star 算法的目标是找到传感器网络中数据收集的时间表，例如网络的生命周期被最大化。每个传感器节点在每一轮中只传输一个包含固定位数的数据包。每个数据收集周期称为一轮生命周期由第一个中继节点耗尽功率（N-of-N 度量）之前的轮数来衡量。

在模型中，路由调度是由一些称为基站的中心实体计算的。我们假设中继节点的数量和位置是预先确定的，因此各个中继节点之间的距离是已知的。随着已知距离，可以计算数据通信消耗的能量。 BS 可以使用这些事实并可以在每一轮之后更新每个中继节点的能量级别。

### 2.3.1. 路线安排

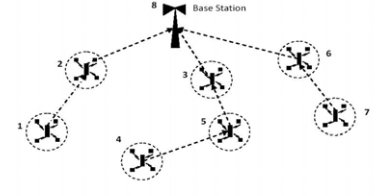


图 2 带标记的中继节点与基站组成的WSN



图 3 数组形式的路由表

A-Star 算法基本上用于使用 g(n) 和 h(n) 函数找到任何源到目的地之间的有效路径。在我们的模型中，BS 将准备路由表并将广播到所有中继节点。 A-Star 算法将应用于每个中继节点。应用该算法的中继节点将是源节点，BS 将是目的节点。将创建这样的 N 条不同的路线，并将合并这些信息。这个合并的路由信息放在一个数组中（如图2）。当前路由调度广播后，所有中 继节点将遵循它。

### 2.3.2. 改进的A\*算法：Proposed A-Star Algorithm

在g(n)和h(n)的基础上，引入预定义的阈值能量水平(Level 1)，可用作传感器节点的阈值水平。如果传感器节点的剩余能量低于这个水平，算法在搜索路径时应该避开这个节点，而是应该搜索替代路径，其中没有剩余能量低于 Level1 的节点。

使用新参数 l(n) 是能量较少 的弱节点的路径成本计数。这将保留当前路径中有多少节点低于能级 Level1 的计数器。此时估计成本函数f(n) = ( g(n) + h(n) , l(n) )。其中，level 1的值根据策略添加，被视为与传感器节点中剩余能量成比例的值，比如在剩余能量为初始能量的0.1%时较大，而在剩余能量为初始能量的30%时较小。

伪代码如下：

|  |
| --- |
| Input : Sensor Network  Output : Life of Sensor Network in terms of rounds  1. BEGIN  2. InitilizeNetwork()  3. EstimateDistance() // finds distance to the BS（计算到BS的距离）  4. WHILE NOT END\_ASTAR() // N-of-N metric（N-N指标）  5. InitializeSolArray() // initialize solution array to store routing schedule（初始化存储路由表的解决方案数组）  6. FOR each node i in the Network DO  7. CreateTree (i) //using A-Star algorithm（使用A\*算法）  8. PrepareSolArray() //prepare routing schedule（准备路由表）  9. END FOR  10. BroadcastSolution() // BS broadcasts routing schedule（BS广播路由表）  11. UpdateEnergy() //Energy update for relay nodes（中继节点能量更新）  12. CountRound = CountRound + 1 //count n/w life  13. END WHILE  14. PRINT CountRound //print n/w lifetime in  15. END |

1. **A\*在船舶气象航线中的应用**

## 路径规划空间建模

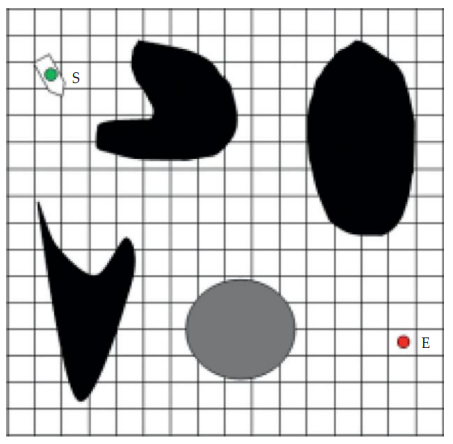


图 4 网格模型

采用网格模型，对航行海域环境进行建模。船的起始位置S用绿点表示，目标位置E用红点表示，黑色阴影区域为静态障碍物，灰色阴影区域为覆盖区，时气象信息形成的不可航行区，也可视为静态障碍物。

波高是影响船舶航行的主要气象因素之一。因此，航路规划仅考虑数值天气预报信息中的波高信息。为简单起见，将浪高大于6 m的海域识别为静态障碍物（浅水、岛屿、礁等），直接波高小于或等于6m的海域标记为船舶航行畅通无阻的通航区域。

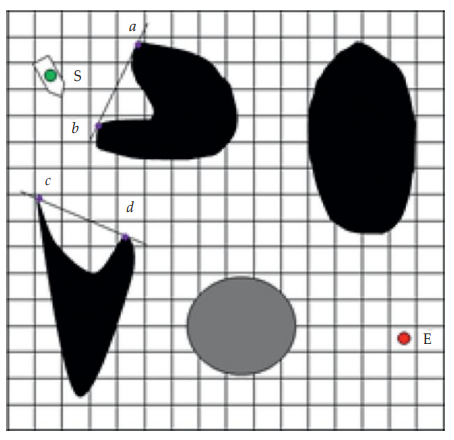


图 5 凹型障碍物的凸型

由于凹形障碍物的 凹形区域可能会导致一些路径规划算法陷入局部振荡。因此， 凹障碍物需要凸起，如图5所示，两条直线分别与 障碍物左上角和左下角的缺口相切，形成线段ab和cd。

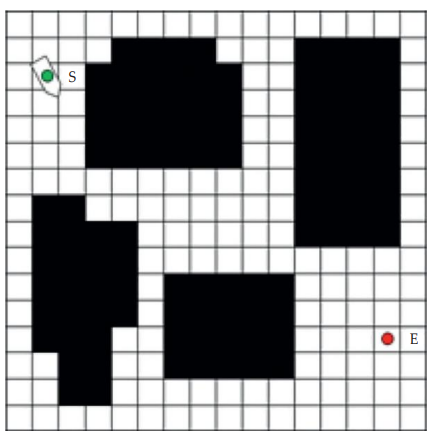


图 6障碍物的扩展

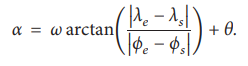
同时障碍物的轮廓需要延长一定距离，若凸障碍物时任意格大小的一部分，则填满网格作为不可通航区域，由于凸障碍物一般不会填满网格，因此扩容处理也会形成一定长度的安全距离。

总的来说，采用网格模型，对航行海域环境进行建模，并在建模时凹形障碍物凸起扩大。

## 改进的A\*算法

由于经典A\*计算量仍然较大，因此根据目标点相对于起点的位置，限制算法在每个节点的搜索方向，以8个搜索方向的方格地图为例，经过全局路径规划预处理后，A-star算法的具体改进过程如下：

1. 用直线连接起点和目标点，测量直线与真北方向的夹角α。 α的计算方法如下，其中，λe为目标点经度，λs为起点经度，φe为目标点纬度，φs为起点纬度, θ 和ω 是常数。





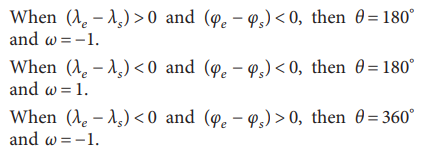


图 7 α角计算公式

1. 根据α与搜索度方向的关系，确定预留搜索度方向。
2. 将夹角α作为启发式信息引入A-star算法中寻找最优路径：

|  |
| --- |
| 1. 将起点添加到打开的表中：  2. 在打开的表中找到F值最低的节点，将当前节点设置为F值最低的节点。  3. 将当前节点从打开的表中移出，放入关闭的表中。  4. 依次确定保留的5个搜索方向指向的5个节点。  5. 如果该节点已经在关闭表中，则忽略该节点。  6. 如果该节点不在open表中，则将该节点添加到open表中，并以当前节点作为该节点的父节点，记录该节点的F、G、H值。  7. 如果该节点已经在open表中，当通过当前节点到达该节点时，如果G的值小于不经过当前节点到达该节点的G的值，则当前节点作为该节点的父节点，重新计算该节点的F、G、H值。否则，该节点的 F、G 和 H 值保持不变。  8. 停止搜索：如果目标点已经添加到关闭表中，保存路径；如果目标点没有添加到关闭表和打开表是空的，路径不存在。 |

1. **A\*在无人机路线规划中的应用**

为在与敌作战中做出更科学的决策，要求规划算法快速找到最佳解决方案，可在承担最小危险、消耗最少燃料的约束下，将A\*算法应用于无人机航线规划。

## 路径规划空间建模

在敌方防御区域内执行攻击任务时，无人机应该能够选择能够到达目标点的路线，并确保其雷达可探测概率小，航程短。将整个空间网格划分形成网络图后，可以将该优化问题的搜索空间简化为一组离散点，从而将问题转化为从网络中的所有节点中选择一些节点，形成从头到尾的路径。同时，此路径应该是所有可能路径中成本最低的路径。

考虑到无人机在中空或高空飞行不能利用地形因子来回避，并且在执行任务期间其飞行高度变化不大，因此可以将航线规划问题简化为二维航线规划问题。如果需要更高的精度，则有利于进一步细化网格并增加节点数量。网格的划分应考虑特定任务的精度要求和所提供的数字地形图的精度。

根据敌人的防御力量确定一个矩形区域，这样可以节省内存空间和计算时间。然后，将面积除以矩形大小以生成无人机可能通过的节点。

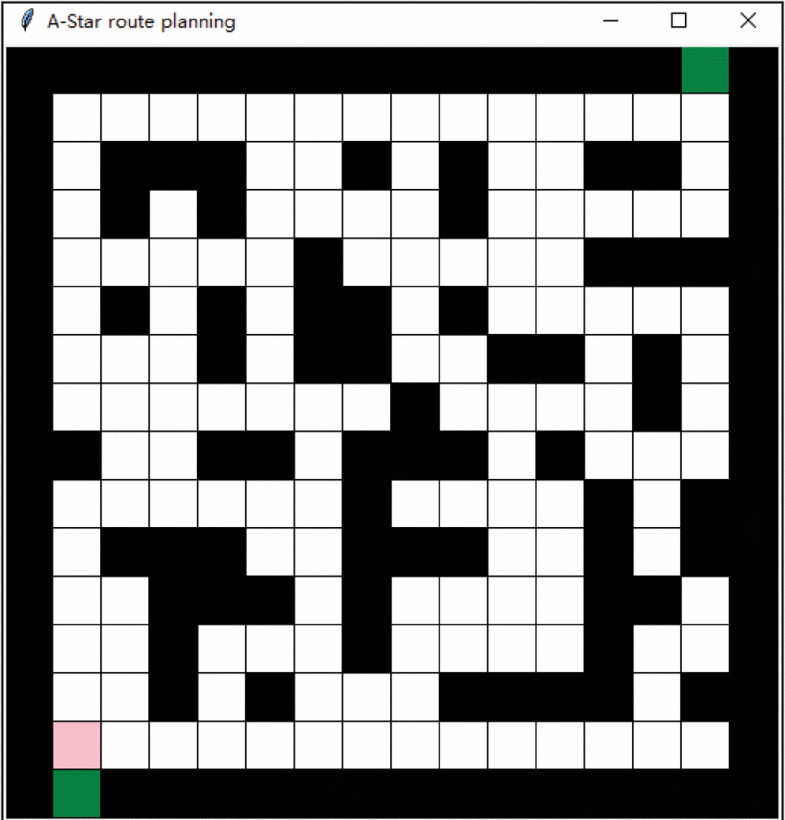
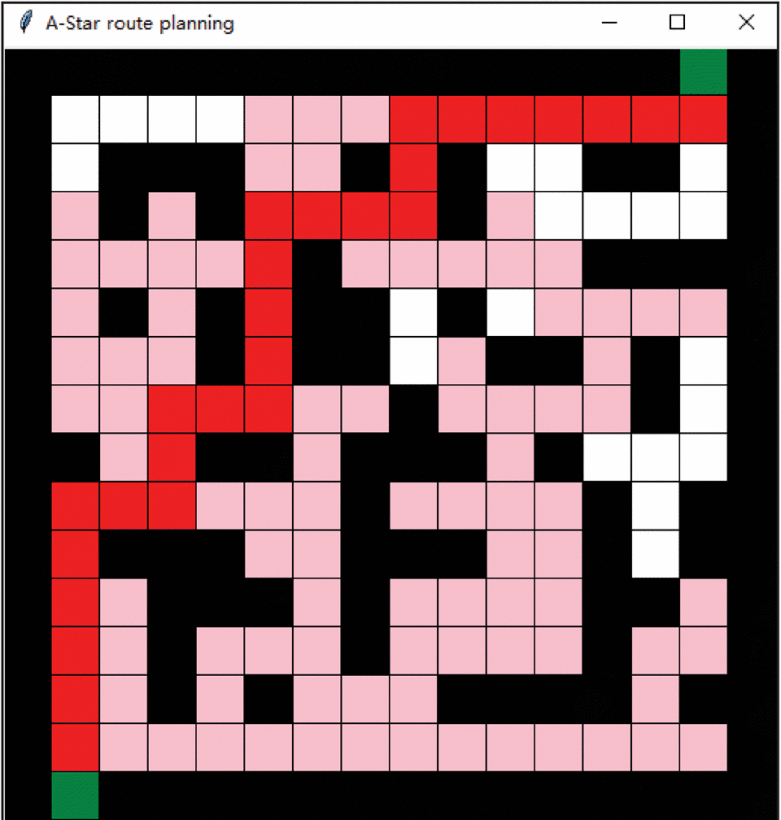
 

图 8 初始状态与A\*规划后仿真生成的轨迹

## 成本函数

f=g+hd+ht

该模型下使用成本函数如上，其中，g表示路线的燃料成本;hd代表曼哈顿距离，而ht演示节点是否为威胁点。如果节点是无人机的威胁单元，ht将升得非常高，以防止车辆飞入该区域。

此成本函数可选择性能和航程符合要求的某一条航线，以在计算最佳解决方案所花费的时间与选择最佳航程的额外成本之间取得平衡。

1. **算法仿真**

我们在两个环境上做了相应的仿真实验，来确认A\*算法的可行性。

1. **旅游所需最短时间计算**

**A\*算法**

1》算法简介

BFS以起点A为圆心，先搜索A周围的所有点，形成一个类似圆的搜索区域，再扩大搜索半径，进一步搜索其它没搜索到的区域，直到终点B进入搜索区域内被找到。DFS则是让搜索的区域离A尽量远，离B尽量近。BFS保证的是从起点到达路线上的任意点花费的代价最小（但是不考虑这个过程是否要搜索很多格子）；DFS保证的是通过不断矫正行走方向和终点的方向的关系，使发现终点要搜索的格子更少（但是不考虑这个过程是否绕远）。A\*算法的设计同时融合了BFS和DFS的优势，既考虑到了从起点通过当前路线的代价（保证了不会绕路），又不断的计算当前路线方向是否更趋近终点的方向（保证了不会搜索很多图块），是一种静态路网中最有效的直接搜索算法。而在这个场景中估价函数的值由SPFA给出。

**SPFA:**

1. 用数组dis记录每个结点的最短路径估计值，用邻接表或邻接矩阵来存储图G（这里用的是邻接表）。
2. 采用动态逼近法：设立一个先进先出的队列用来保存待优化的结点，优化时每次取出队首结点u，并且用u点当前的最短路径估计值对离开u点所指向的结点v进行松弛操作，如果v点的最短路径估计值有所调整，且v点不在当前的队列中，就将v点放入队尾。这样不断从队列中取出结点来进行松弛操作，直至队列空为止。
3. 在Bellman\_Ford算法中，我们需要依次遍历每一条边（a----->b，长为w）来更新我们的dis[b]。但是如果dis[a]在循环中如果一直没有出现变化，那么公式（dis[b]=min(dis[b],dis[a]+w)）就一直不会更新我们的dis[b]，做了很多的无用操作，对程序的运行速度造成了比较大的影响。因此，只有当dis[a]发生了变化时，我们才去更新所有以a为起点的点。而在时间复杂度上，虽然SPFA算法的时间复杂度有退化的可能性，但基本上优于Bellman\_Ford。

2》算法的思路

如果我们的搜索数量极其的大的时候，我们再使用bfs的时候，就很有可能会TLE。A\*算法其实是“BFS + 贪心”，在搜索过程中，我们用一个估价函数对当前的情况进行估价，得到最好的状态，然后从这个状态去进行搜索，需要注意的是如果我们用A\*算法计算最短距离的时候，我们这个点的到终点估计距离是要小于等于这个点到终点的真实距离的，且我们的A\*算法不可以存在负权边。

**查找过程：**

1. 1.在待遍历列表中（刚开始只有点A），我们在列表中查找一个估价（当前点到终点距离估价，使用的是SPFA算出的dist）最小的点（k），
2. 2.对点k进行一次BFS，也就是遍历以它为起点的所有出边，将能查找的点添加到队列中，并将点K从队列中移除。
3. 重复1、2步骤直到到底B点，或者队列已经为空说明没有路径可以到达点B。

**估价函数：**

F = G + H。

G=从起点A移动到当前节点x的移动代价

H=从当前节点x移动到终点B的估算成本。

首先明确一下f的性质并给出简略证明：

（定义实际值为ans）f<=ans,且f==ans时BFS最易搜得最优解。

证明：1.1.显然,估价函数正确的估到了正确答案，那么沿着这个路径一定是正解；（此时f一般为实时更新）

1.2.f<=ans : Astar 是以优先队列为基础实现的，则f越优越先被更新，若f>g，则当前是按错误路径搜索。

3》与Dijkstra算法的区别

1.Dijkstra算法计算源点到其他所有点的最短路径长度，A\*关注点到点的最短路径(包括具体路径)。

2.Dijkstra算法建立在较为抽象的图论层面，A\*算法可以更轻松地用在诸如游戏地图寻路中。

3.Dijkstra算法的实质是广度优先搜索，是一种发散式的搜索，所以空间复杂度和时间复杂度都比较高。对路径上的当前点，A\*算法不但记录其到源点的代价，还计算当前点到目标点的期望代价，是一种启发式算法，也可以认为是一种深度优先的算法。

4.由第一点，当目标点很多时，A\*算法会带入大量重复数据和复杂的估价函数，所以如果不要求获得具体路径而只比较路径长度时，Dijkstra算法会成为更好的选择。

4》具体编程步骤

估价函数，初始化，起点放入队列

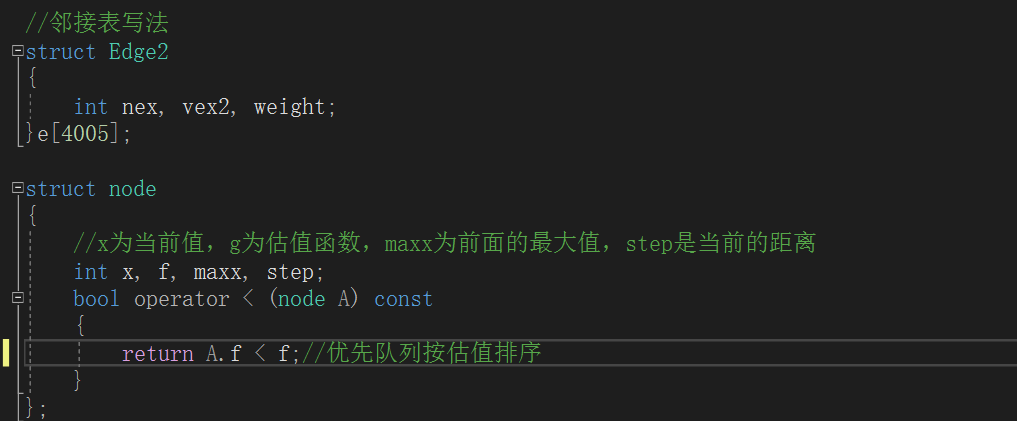
当队列非空时，（队列存储所有变小的点）

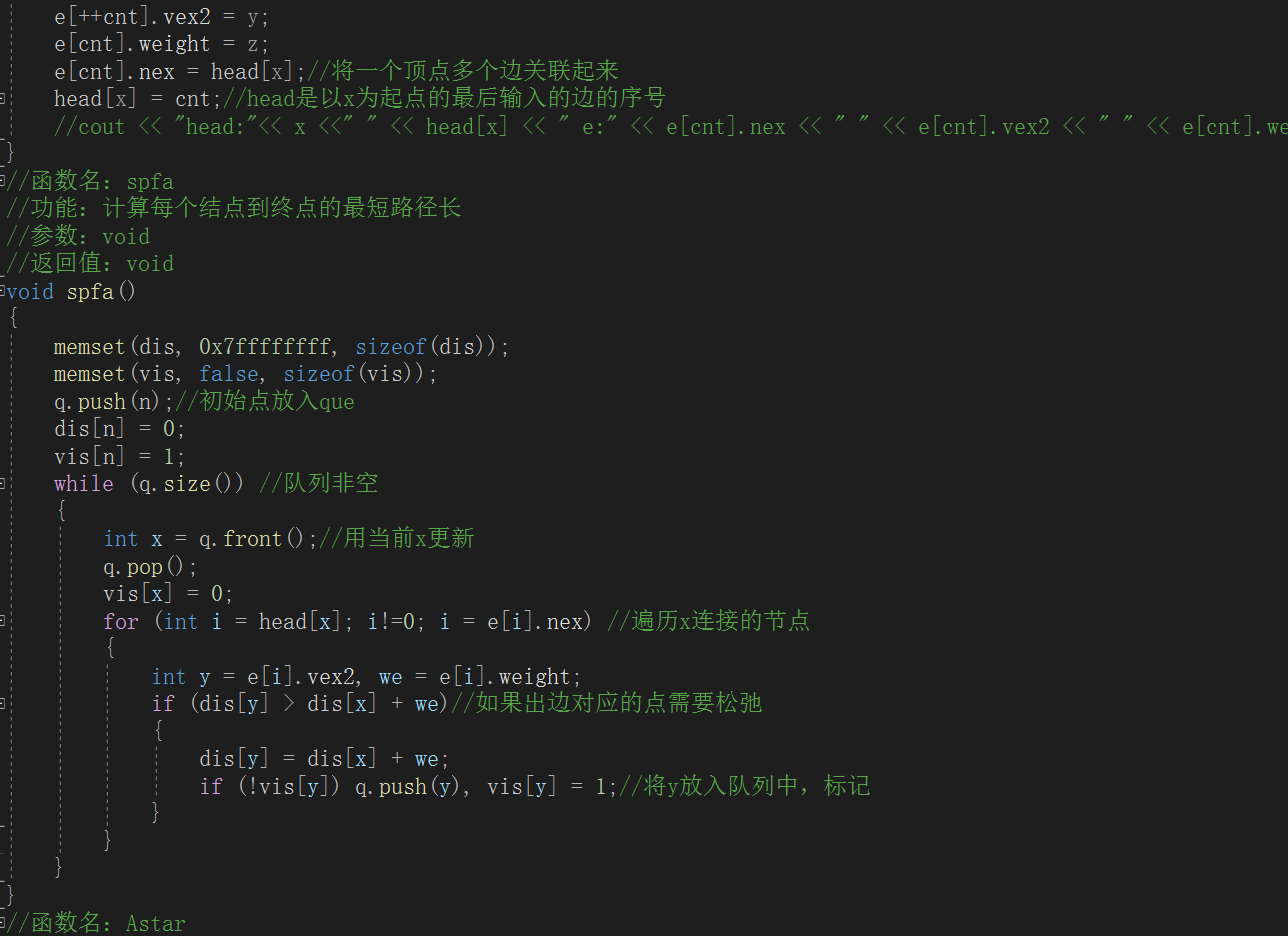
取出队列中的头个节点t

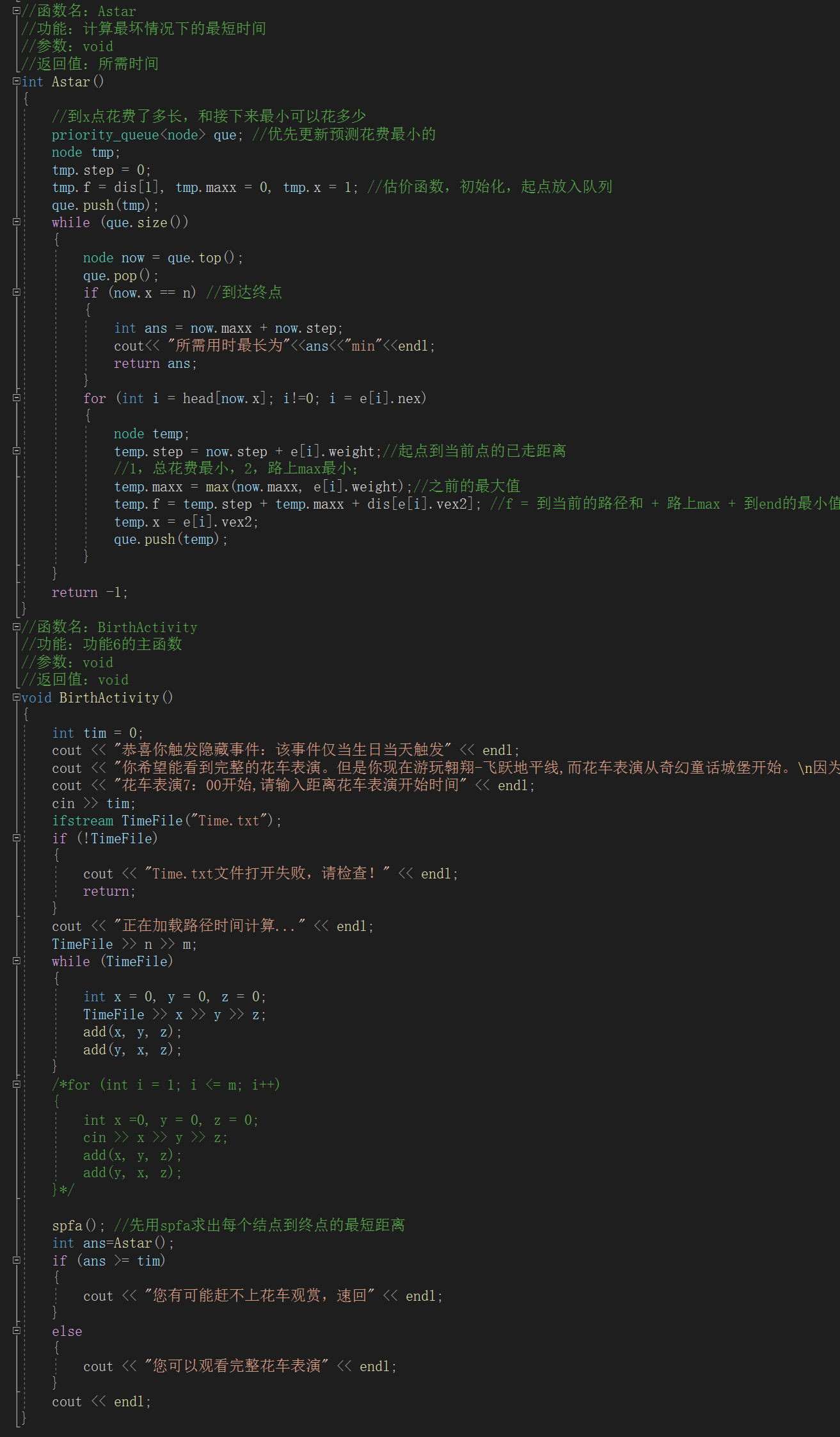
更新now的所有出边now----->tmp，计算该出边的估价函数,tmp入队

f = 到当前的路径和 + 路上max + 到end的最小值

如果当前更新结点为终点，则输出估计的时间







1. **八数码问题：**

A\*算法

做法：

引入一个估值函数，用来估计某个点到达终点的距离。

记f是估值函数，g是真实值，那么f(state) <= g(state)，越接近越好（当估值是0时，类似于Dijkstra算法）

记dist是从起点到state状态的步数；

利用的是优先队列，排序依据是dist[state] + f(state)

证明：

反证法：

假设终点第一次出堆时不是最小值，那么意味着dist[end] > dist优

那么说明堆中存在一个最优路径中的某个点（起码起点在路径上），记该点为u，

dist优 = dist[u] + g(u) >= dist[u] + f(u)

-> dist[end] > dist优 >= dist[u] + f(u),说明优先队列中存在一个比出堆元素更小的值，这就矛盾了。

所以说终点第一次出堆时就是最优的。

应用的环境：

1、有解（无解时，仍然会把所有空间搜索，会比一般的bfs慢，因为优先队列的操作是logn的）

2、边权非负，如果是负数，那么终点的估值有可能是负无穷，终点可能会直接出堆

性质：

除了终点以外的其他点无法在出堆或者如堆的时候确定距离，只能保证终点出堆时是最优的可以。

**核心代码：**

#include<cstdio>

#include<iostream>

#include<cstring>

#include<queue>

#include<unordered\_map>

#include"Digit.h"

using namespace std;

typedef pair<int, string> PIS;

unordered\_map <string, int> dist;

unordered\_map <string, pair<string, char>> pre;

priority\_queue<PIS, vector<PIS>, greater<PIS>> heap;

string ed = "12345678x";

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

char op[] = "urdl";

int f(string state)//求估值函数,这里是曼哈顿距离

{

int res = 0;

for (int i = 0; i < 9; i++)

{

if (state[i] != 'x')

{

int t = state[i] - '1';

res += abs(t / 3 - i / 3) + abs(t % 3 - i % 3);

}

}

return res;

}

string bfs(string start)

{

heap.push({ f(start) , start });

dist[start] = 0;

while (heap.size())

{

auto t = heap.top();

heap.pop();

cout << "当前节点：state " << t.second << " 估值：" << t.first << endl;

string state = t.second;

int step = dist[state];//记录到达state的实际距离

if (state == ed) break;//如果到达终点就break

int k = state.find('x');

int x = k / 3, y = k % 3;

string source = state;//因为在下面state会变，所以留一个备份

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

{

int a = x + dx[i], b = y + dy[i];

if (a >= 0 && a < 3 && b >= 0 && b < 3)

{

swap(state[x \* 3 + y], state[a \* 3 + b]);

if (!dist.count(state) || dist[state] > step + 1)

{

dist[state] = step + 1;

pre[state] = { source, op[i] };

heap.push({ dist[state] + f(state), state });

}

swap(state[x \* 3 + y], state[a \* 3 + b]);//因为要多次交换，所以要恢复现场

}

}

}

string res;

while (ed != start)

{

res += pre[ed].second;

ed = pre[ed].first;

}

reverse(res.begin(), res.end());

return res;

}

void getdigit()

{

string start, seq;

for (int i = 0; i < 9; i++)

{

char c;

cin >> c;

start += c;

if (c != 'x') seq += c;

}

int cnt = 0;

for (int i = 0; i < 8; i++)

for (int j = i + 1; j < 8; j++)

if (seq[i] > seq[j])

cnt++;

if (cnt % 2) puts("unsolvable");

else cout << bfs(start) << endl;

}

首先定义了一个估值函数f，这里使用的是曼哈顿距离，用于估算当前状态到达目标状态的距离。然后使用一个priority\_queue来存储状态，每次取出估值最小的状态进行扩展。同时使用一个unordered\_map来存储状态到达起点的实际距离和之前的状态及操作。

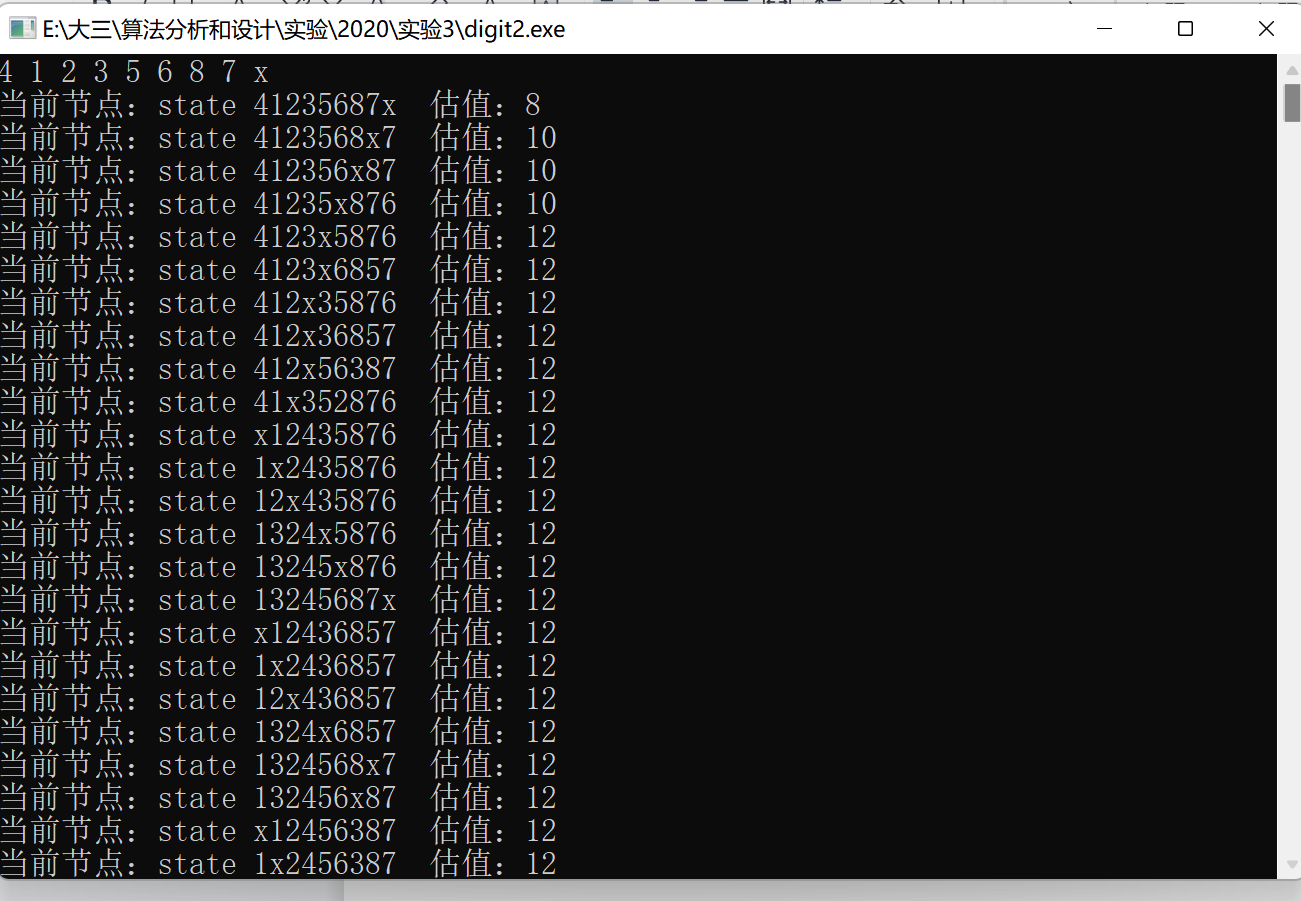
在bfs函数中，首先将起点状态加入队列，并将其距离设置为0。然后每次取出估值最小的状态进行扩展，如果扩展到的状态之前没有到达过，或者当前到达状态的距离更小，则更新状态距离和之前状态，并将其加入队列。当到达终点状态时，跳出循环。

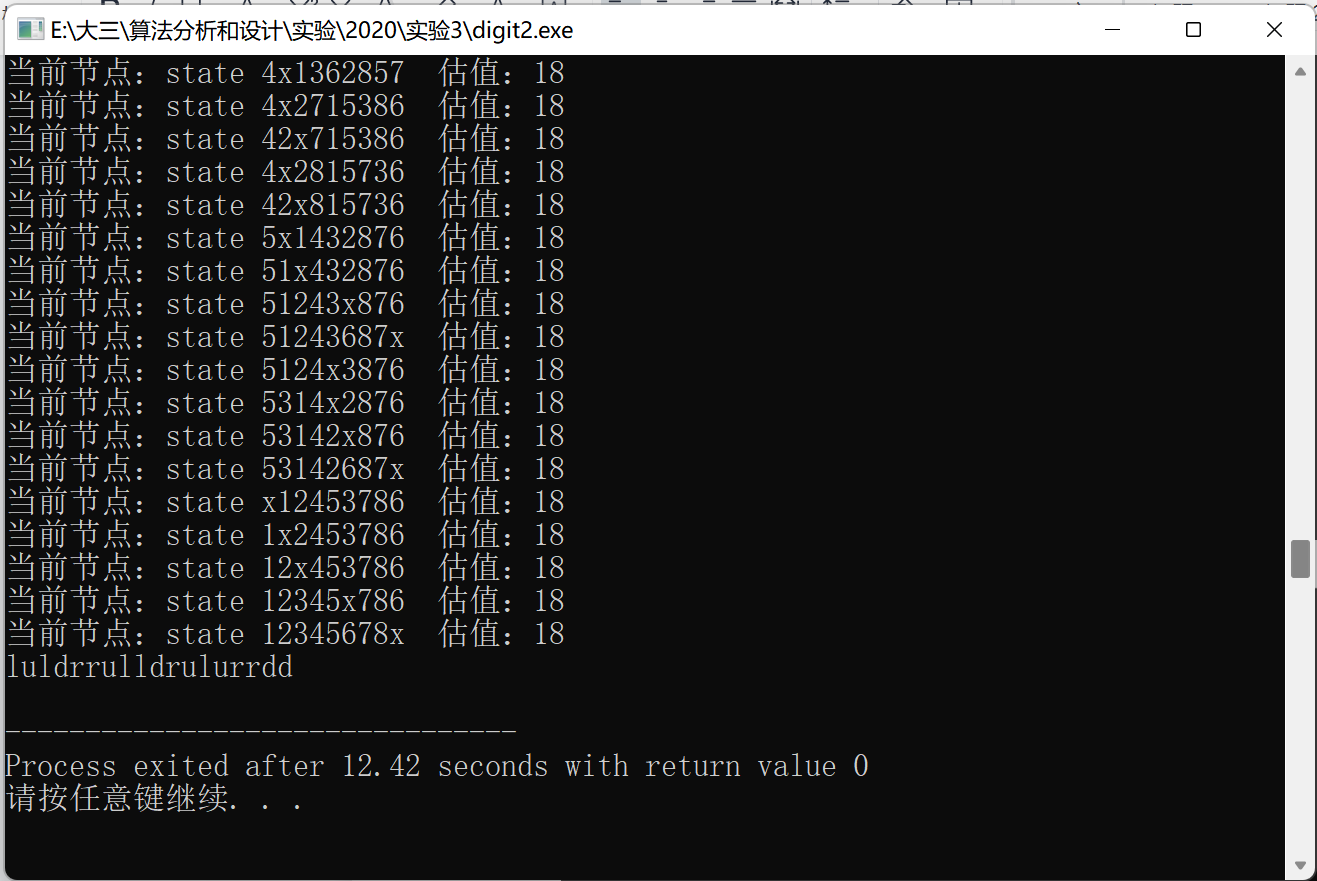
最后，从终点状态开始遍历之前的状态及其操作，得到最终的解法。

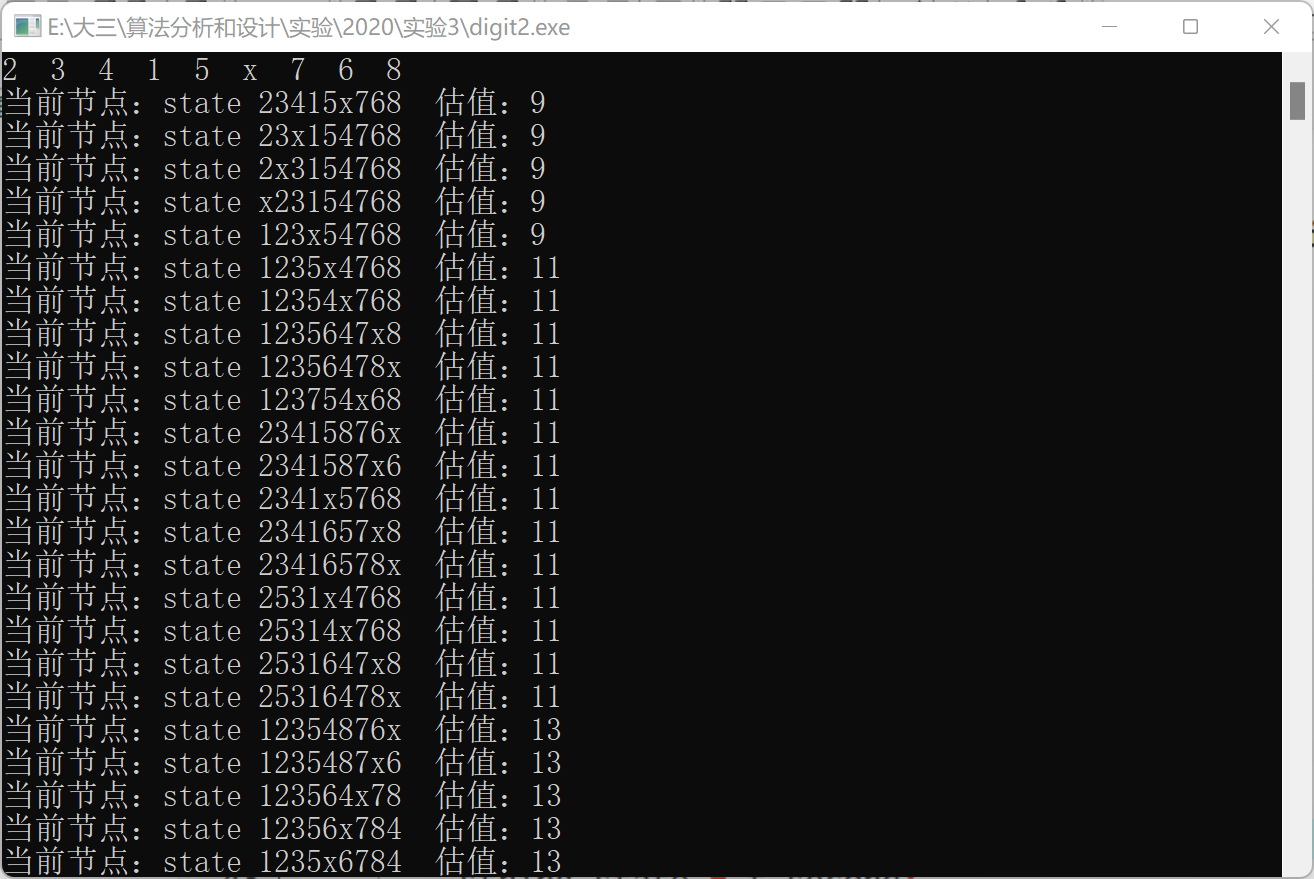
值得注意的是，由于交换状态时需要修改字符串，因此在交换后需要恢复现场，以便进行下一次交换。

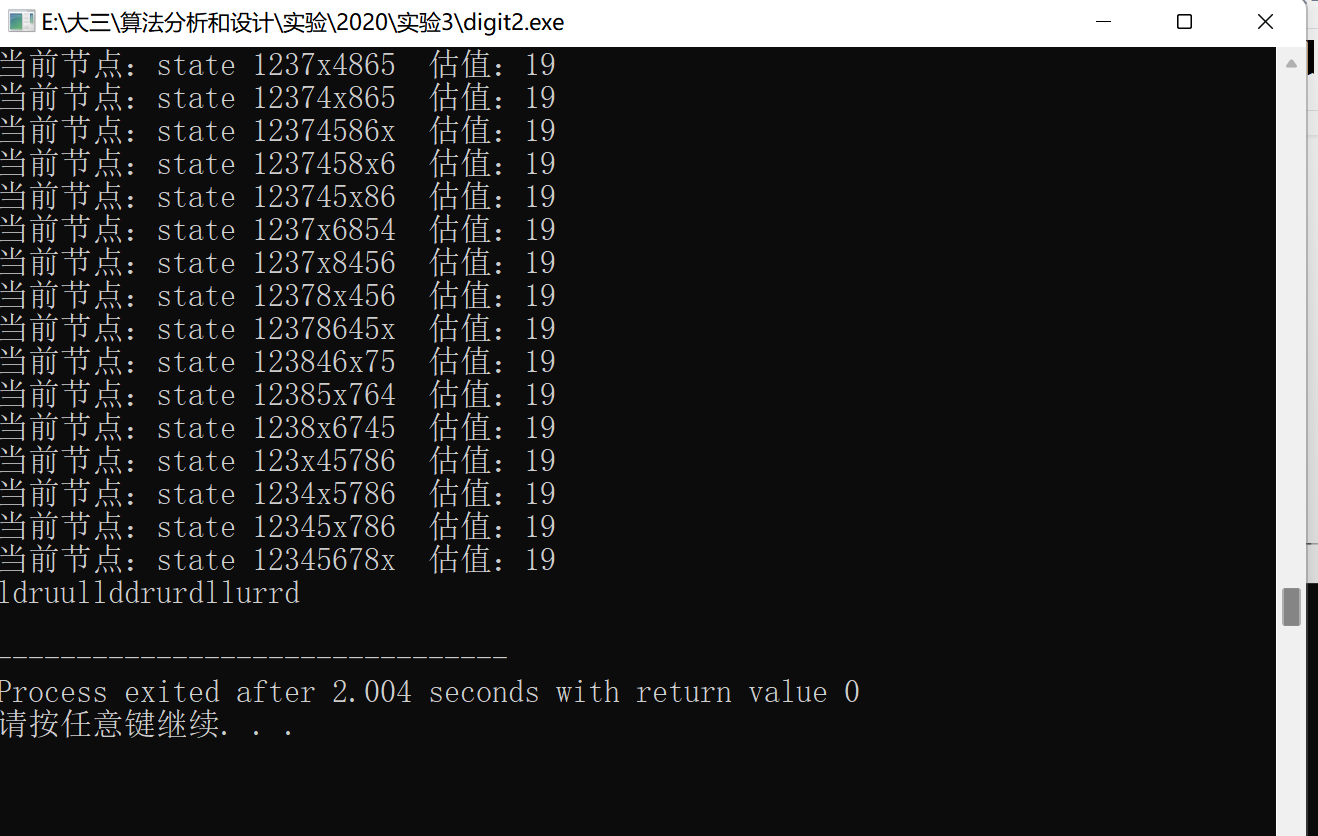
其他与剪枝一类似。

**结果：**









#include <bits/stdc++.h>
  
using namespace std;
  
  
typedef pair<int , string> PIS;
  
  
unordered\_map<string , int> dist;
  
unordered\_map<string , pair<string , char>> pre;
  
priority\_queue<PIS , vector<PIS> , greater<PIS>> heap;
  
string ed = "12345678x";
  
int dx[4] = {-1 , 0 , 1 , 0} , dy[4] = {0 , 1 , 0 , -1};
  
char op[] = "urdl";
  
  
int f(string state)//求估值函数,这里是曼哈顿距离
  
{
  
 int res = 0;
  
 for(int i = 0 ; i < 9 ; i++)
  
 {
  
 if(state[i] != 'x')
  
 {
  
 int t = state[i] - '1';
  
 res += abs(t / 3 - i / 3) + abs(t % 3 - i % 3);
  
 }
  
 }
  
 return res;
  
}
  
  
string bfs(string start)
  
{
  
 heap.push({f(start) , start});
  
 dist[start] = 0;
  
  
 while(heap.size())
  
 {
  
 auto t = heap.top();
  
 heap.pop();
  
 cout<<"当前节点：state "<<t.second<<" 估值：" <<t.first<<endl;
  
 string state = t.second;
  
 int step = dist[state];//记录到达state的实际距离
  
 if(state == ed) break;//如果到达终点就break
  
  
 int k = state.find('x');
  
 int x = k / 3 , y = k % 3;
  
  
 string source = state;//因为在下面state会变，所以留一个备份
  
 for (int i = 0; i < 4; i ++ )
  
 {
  
 int a = x + dx[i], b = y + dy[i];
  
 if (a >= 0 && a < 3 && b >= 0 && b < 3)
  
 {
  
 swap(state[x \* 3 + y], state[a \* 3 + b]);
  
 if (!dist.count(state) || dist[state] > step + 1)
  
 {
  
 dist[state] = step + 1;
  
 pre[state] = {source, op[i]};
  
 heap.push({dist[state] + f(state), state});
  
 }
  
 swap(state[x \* 3 + y], state[a \* 3 + b]);//因为要多次交换，所以要恢复现场
  
 }
  
 }
  
  
 }
  
  
 string res;
  
 while(ed != start)
  
 {
  
 res += pre[ed].second;
  
 ed = pre[ed].first;
  
 }
  
 reverse(res.begin() , res.end());
  
 return res;
  
}
  
  
int main()
  
{
  
 string start , seq;
  
 for(int i = 0 ; i < 9 ; i++)
  
 {
  
 char c;
  
 cin >> c;
  
 start += c;
  
 if(c != 'x') seq += c;
  
 }
  
  
 int cnt = 0;
  
 for(int i = 0 ; i < 8 ; i ++)
  
 for(int j = i + 1 ; j < 8 ; j++)
  
 if(seq[i] > seq[j])
  
 cnt++;
  
  
 if(cnt % 2) puts("unsolvable");
  
 else cout << bfs(start) << endl;
  
  
 return 0;
  
}