

智能控制课设

Arena Challenge 实验报告

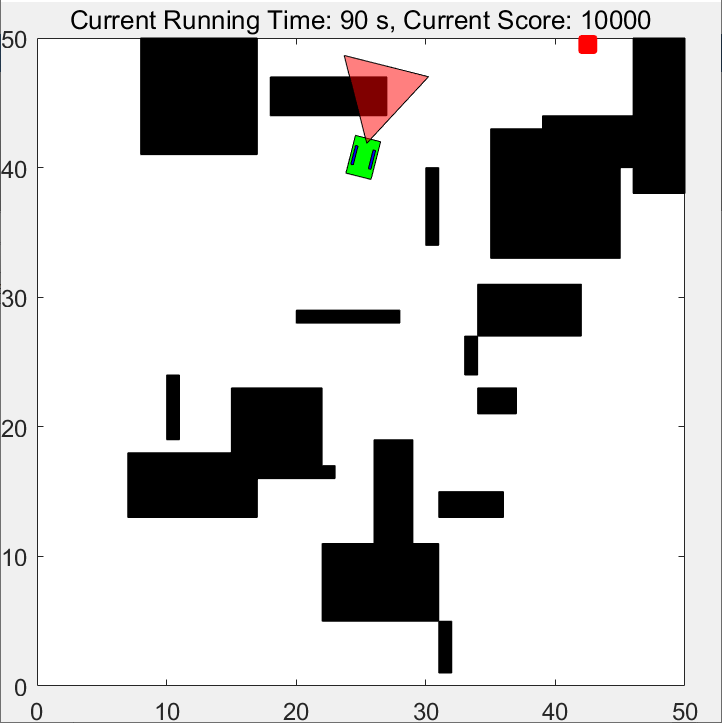
|  |  |
| --- | --- |
| 院系 |  |
| 专业班级 |  |
| 姓名 |  |
| 学号 |  |
| 指导老师 |  |

目录

# 实验环境

## Arena Challenge

控制小车在一个地图环境中从起点运行至终点，地图中随机放置着障碍物。小车是一个unicycle的动力系统，并装有“雷达”探测器，可探测前方障碍物情况。最终的成绩由小车到达终点的时间以及小车是否撞上障碍物等情况综合评估而得。



实验地图实例

## 小车动力学模型

小车的动力学模型，如下公式所示：

其中，分别是小车当前的空间位置和朝向。是小车的控制量，u相当于小车的“油门”踏板，控制小车的前车速度；v是小车的“方向盘”，控制小车的转动速度。这是挑战者在编写代码时，唯一可以控制小车的两个物理量。

设置了“饱和”机制，传递给仿真器的小车两个控制量，将会被限制在一定范围内。

## 雷达探测器

小车前方有一个前向的雷达传感器，可探测前方是否有障碍物，如有障碍物将，将会把障碍物信息告知挑战者。小车前方红色三角形所表示的雷达探测范围。

## 通用参数设置

[StartPos]  
x = 2   
y = 3  
heading = 0.835  
  
[EndPos]  
x = 42  
y = 49  
  
[Map]  
w=50  
h=50  
  
[RangeFinder]  
scanRange=8  
scanAngle=0.58  
  
[Agent]  
usat=1  
vsat=0.5  
  
[System]  
tend=150  
score=10000  
scoreTimes=1.5 % used for calculate rewardTime  
collideDeducts=3 % score deduction if collision occurs  
collidewithAgentDeducts=10 % score deduction if collision with smartAgents occurs  
obDeducts=2 % score deduction if agent move outside of screen  
timeoutDeducts=2 % score deduction if current time >rewardTime  
st=0.3 % simulator step  
showViewer=1 % set if show the viewer   
render\_st=0.3  
  
globalview=0 % set if global view is provided

# 控制算法

小车控制算法由两部分组成：路径规划和速度规划。首先使用路径规划，搜索或采样出起点到终点的可用路线，然后进行速度规划，在跟踪上述可用路线的同时，避开障碍物，给出合理的速度，使小车又快又安全地到达目标点。在本实验中，我使用A\*算法进行路径规划，动态DWA-P控制器规划速度。

## 地图处理

实验中采用非全局地图。地图采用bool形式存储，其中1表示为障碍物，0表示正常区域。

静态障碍物位置不会改变，可以存储到全局地图中；而动态小车会随时移动，存储在全局地图中会导致地图障碍物错误累计。故在实验中对两类障碍物分开处理，即存储静态障碍物，实时更新动态障碍物。实现代码为：

% 存储固定障碍物  
self.obs\_map = self.obs\_map | observation.scanMap == 1;  
% 添加移动障碍物  
map = self.obs\_map | observation.scanMap>1;

由于小车宽度为2，而Observation中获取的小车位置并不包含宽度信息。若直接使用得到的原始地图，易导致后续路径规划算法规划出错误的路径，例如：

表 -

| **单像素宽斜向** | **单像素宽直行** |
| --- | --- |
| s1 | s2 |

看似可以通过，实则由于小车宽度限制无法通过。

故在进行路径规划之前，首先要对检测到的障碍物地图进行膨胀处理。考虑到小车宽度为2，故使用八邻域膨胀，去除单/双像素宽的路径（之所以去掉双像素宽的路径，是因为，小车的宽度和通道宽度相同时，可供小车移动的容错率极低，难以在不撞到障碍物的条件下进行移动）。实现代码为：

map\_tmp = bwmorph(origin,'dilate',1);

由于目标点离上边界较近，直接使用原始地图容易导致小车走出地图范围，同时小车大概率不会走出其他三个边界，故为了平衡算法的速度和有效性，只将地图上边界设为障碍物（不保存在实际障碍物中，避免在后续地图更新中反复膨胀）。实现代码为：

map\_tmp(:,50) = 1;

由于膨胀后地图可能导致目标点及其附近被视为障碍物，从而导致后续规划算法无法进行，故需要对目标点附近进行单独处理，将其本身及其八邻域设为正常区域。实现代码为：

map\_tmp(self.dest(1)-1:self.dest(1)+1,self.dest(2)-1:self.dest(2)+1) = 0;

## 角度运算问题

由于角度是以为周期的周期变量，直接对角度进行加减运算会导致错误判断。

首先，需要限制Observation中获取的小车当前角度到。实现代码：

function observation = clip(~,observation)  
 % 限制角度范围  
 h = observation.agent.h;  
 observation.agent.h = wrapToPi(h);  
end

同时，角度相减的结果也需要限制范围到。实现代码：

function d = diff(~,a,b)  
 % 角度相减  
 d = wrapToPi(a-b);  
end

## 路径规划算法

随着数学等领域的发展，现在已有多种路径规划算法，包括Dijkstra算法、RRT算法、人工势场法和A\*算法等传统算法，也有遗传算法、粒子群算法和强化学习等智能算法。其中，智能算法虽然具有算法简洁、 鲁棒性好等优点，但其往往计算量大且难以实现。故在实验中，我选择了收敛速度比较快、稳定性较高的A\*算法作为路径规划算法。

### A\*算法实现

A\*算法通过启发式搜索找到路径。为充分利用MATLAB的矩阵运算能力，其实现可以表示为：

1. 以地图大小初始化open set、close set、G得分（实际与起点之间的距离）、F得分（F=G+H，H为启发式得分）、上一个点存储向量。其中open set、close set、上一个点存储向量均初始化为0；G得分、F得分初始化为inf。
2. 将起点open set置为1，G得分为0，F得分为H。
3. 循环下列过程
   1. 找到open set中F 值最小的点，作为当前要处理的点；
   2. 停止循环，若：
      * 该点为终点；
      * open set 中没有点为1。
   3. 将该点移到close set中；
   4. 获得该点对应八邻域
      * 去掉超出地图范围的领域点；
      * 去掉不可达/在close set中的领域点；
   5. 将有效邻域点open set置为1；
   6. 计算有效邻域点G得分
      * 若G得分相比之前的G得分变大或不变，则不改变其G得分、F得分、上一个节点；
      * 若G得分相比之前的G得分变小，则表示该路径更好，更新该邻域点G得分、F得分，并将该邻域点的上一个点设为当前点；
4. 从终点开始，每个方格沿着上一个点移动直至起点，即为搜索出的轨迹信息。若无法搜索到有效轨迹，则返回空。

其中启发式得分使用欧几里得距离，可以表示为：

实现代码为：

%% A Star  
  
function path = aStar(map, start, goal)  
 % 搜索失败 => 输出空  
 path = [];  
 % 地图大小  
 map\_size = size(map);  
 map\_lenth = numel(map);  
 % 转为线性索引 => 方便直接使用  
 start = sub2ind(map\_size,start(1),start(2));  
 goal = sub2ind(map\_size,goal(1),goal(2));  
  
 % 初始化 open set  
 open\_set = false(map\_size);  
 open\_set(start) = true;  
 % 初始化 close set  
 close\_set = false(map\_size);  
 % 记录上一个节点  
 parents = zeros(1, map\_lenth);   
 % 初始化G得分 => 起点到指定方格的移动代价  
 g\_score = inf(map\_size);  
 g\_score(start) = 0;  
 % 初始化F得分 => 预估总代价  
 f\_score = inf(map\_size);  
 f\_score(start) = getH(map\_size,start,goal);  
   
 sqrt2 = sqrt(2);  
  
 % open set 非空  
 while any(open\_set(:))  
  
 % 最小F得分  
 open\_f\_score = f\_score.\*open\_set;  
 [~, current] = min(open\_f\_score(:));  
  
 % 若到达终点  
 if current == goal  
 % 获得轨迹  
 exists = find(parents);  
 while any(current == exists)  
 current = parents(current);  
 path = [current path];   
 end  
 return  
 end   
   
 % current => close  
 open\_set(current) = false;  
 close\_set(current) = true;  
 f\_score(current) = inf;  
   
 % 当前G得分  
 current\_g\_score = g\_score(current);  
   
 % 获取8邻域  
 [xc,yc] = ind2sub(map\_size,current);  
 ns = [  
 xc + 1 yc + 1 sqrt2   
 xc + 1 yc 1   
 xc + 1 yc - 1 sqrt2   
 xc yc - 1 1   
 xc - 1 yc - 1 sqrt2   
 xc - 1 yc 1   
 xc - 1 yc + 1 sqrt2   
 xc yc + 1 1   
 ];  
 % 去除非法邻域点  
 ns = ns(ns(:,1)>0&ns(:,1)<=map\_size(1)&ns(:,2)>0&ns(:,2)<=map\_size(2),:);  
   
 % 去除不可达/close set中邻域点  
 neighbors = sub2ind(map\_size,ns(:,1),ns(:,2));  
 valid\_ns = ~map(neighbors) & ~close\_set(neighbors);  
 neighbors = neighbors(valid\_ns);  
 distances = ns(valid\_ns,3);  
  
 % 添加邻域点  
 open\_set(neighbors) = true;  
  
 % 邻域点当前路径G得分  
 tmp\_g\_scores = current\_g\_score + distances;  
   
 % 更新G变小的邻域点  
 index\_better = tmp\_g\_scores < g\_score(neighbors);  
 better\_neighbors = neighbors(index\_better);   
 parents(better\_neighbors) = current;  
 g\_score(better\_neighbors) = tmp\_g\_scores(index\_better);  
 f\_score(better\_neighbors) = g\_score(better\_neighbors) + getH(map\_size,better\_neighbors,goal);  
  
 end  
end  
  
function cost = getH(map\_size,from,to)  
 % Heuristic cost  
 [x\_from,y\_from] = ind2sub(map\_size,from);  
 [x\_to,y\_to] = ind2sub(map\_size,to);  
 cost = sqrt((x\_from - x\_to).^2 + (y\_from - y\_to).^2);  
end

由于该地图为局部地图，大多数情况下使用A\*算法搜索出来的路径并不是真实可走的路径，需要随着地图的更新，重新进行路径规划。为能够适应动态地图和简化后续速度规划跟踪轨迹，在实验中，每次触发action方法都会进行一次A\*搜索更新路径。

## 速度规划算法

速度规划算法需要跟踪前面搜索出的路径，在避开障碍物的同时，实现更快地到达目标点。我首先探索了最简单的P控制器，然后尝试了DWA算法，最后结合了DWA算法和P控制器的优点，实现了更好的速度规划。

### 普通P控制器

由于前面A\*算法动态更新路径，故可以直接使用P控制器让小车跟踪下一步/下下一步路径点。为简化计算，选择固定线速度v，控制角速度w的方式进行控制。

首先将当前地图，当前位置，目标点位置输入A\*算法得到规划路径：

route = aStar(map,self.now,self.dest);  
[x, y] = ind2sub(size(map),route);

若规划出路径长度较小，说明此时小车已到达目标点附近，直接使用目标点作为小车跟踪路径点；否则，为了让小车更好地跟踪路径，使用下下一步路径点作为跟踪点。实现代码为：

if length(x) > 4  
 x = x(3);  
 y = y(3);  
else  
 x = self.dest(1);  
 y = self.dest(2);  
end

计算当前位置与跟踪点连线方向角：

theta = atan2d(y-observation.agent.y,x-observation.agent.x)/180\*pi;

为了让小车更快地到达终点，将线速度设置为1。使用P控制器跟踪角度，其中为1：

beta = theta - observation.agent.h;  
action=[1,beta];

由于A\*算法规划出路径大概率会沿着障碍物移动，只使用P控制器极易撞上障碍物，尤其是在转弯的时候。故需要将障碍物再膨胀一圈，给P控制器下小车运动更大的容错空间（在上述地图处理部分膨胀次数改为2）。

经过实验，只使用P控制器可以通过较为简单的地图，例如“a1map”、“amap”等，但无法处理更加复杂的、道路狭窄（只有三格宽）的地图。这是因为地图在二次膨胀后，会导致相隔为3的两块障碍物被连接成一块，无法搜索出可用路径。同时，即使膨胀两次后，哪怕降低线速度，也存在产生碰撞的情况。因此，需要使用更好地、能够考虑障碍物的速度规划方法。

### DWA算法

DWA（动态窗口）算法是一种用于移动机器人局部路径规划的经典算法。它的目标是在不发生碰撞的前提下，找到机器人能够通过的最优路径，同时也能得到最优速度。主要通过遍历速度空间，以最大化评估函数来选择最佳的移动策略。

具体实现可以表示为：

1. 初始化速度空间，即当前状态下可达到的线速度和角速度 由于该实验环境没有加速度限制，故速度范围均可达到，角速度。由于小车雷达只检测前方的障碍物，贸然后退可能碰到未知障碍物，故限制线速度的负值大小，线速度；
2. 离散化速度空间。 由于后续需要对可能的速度进行遍历，故需要将速度空间离散化。同时，线速度最大值为1，角速度最大值为0.5，故设置线速度分辨率为0.05，角速度分辨率为0.05；
3. 遍历速度空间，对每一对速度进行下列运算：
   * 计算时间范围内，在保持该速度条件下，小车的轨迹。其中，表示时间分辨率，表示预测长度。为减少计算量并与实验仿真时间相对应，设置为0.3，设置为5。每段时间小车运动公式可以表示为：
   * 跟踪方向奖励。小车轨迹最后一点方向角越接近小车轨迹最后一点位置与跟踪点连线夹角，奖励越大。由于夹角大小绝对值，且角度制下差异更明显，奖励可以表示为：
   * 速度大小奖励。考虑小车轨迹最后一点，规划出线速度越大，小车移动越快，到达跟踪点的时间就越短，奖励越大，可以表示为：
   * 障碍物距离奖励。考虑整个轨迹，找到与障碍物相距最近的轨迹点与障碍物之间的距离。距离越大，小车运动越安全，奖励越大，可以表示为：
   * 当距离过小时（），小车轨迹距离障碍物过近，不考虑该速度情况，从速度空间中去除。当距离过大时（），为避免障碍物距离奖励对总奖励影响过大，将奖励限制到。
4. 若所有速度空间都不满足障碍物距离条件，小车已处于“障碍物”中，返回规划线速度和角速度均为0。
5. 分别归一化三类奖励，将每类奖励之和限制为1，可以表示为：

* 其中，表示某一类奖励，表示符合条件的速度项；

1. 计算总奖励，可以表示为：

* 其中；

1. 选择总奖励最大的速度条件为输出速度。

DWA实现代码为：

function action = dwa(self,map,observation,dest)  
 % 速度空间  
 v\_space = -0.3:0.05:self.sat(1);  
 w\_space = -self.sat(2):0.05:self.sat(2);  
 % x y theta  
 x0 = [observation.agent.x observation.agent.y observation.agent.h];  
 % 轨迹预测分辨率/时长  
 dt = 0.3;  
 k = 5;  
 % 初始化奖励  
 reward = [];  
 % 获取障碍物  
 [row,col] = find(map==1);  
 for v=v\_space  
 for w=w\_space  
 % 初始化轨迹  
 x = x0;  
 trajectory = x;  
 % 计算后续轨迹  
 for t=dt:dt:dt\*(k-1)  
 x = [x(1)+v\*cos(x(3))\*dt x(2)+v\*sin(x(3))\*dt x(3)+w\*dt];  
 trajectory = [trajectory;x];  
 end  
 % 跟踪方向奖励  
 theta = atan2d(dest(2)-x(2),dest(1)-x(1))/180\*pi;  
 angle\_reward = 180-abs(self.diff(theta,x(3)))/pi\*180;  
 % 速度大小奖励  
 v\_reward = abs(v);  
  
 % 据最近障碍物距离奖励  
 ob\_distance = inf;  
 % 找到最近障碍物距离  
 if ~isempty(row)  
 for t=1:k  
 x = trajectory(t,:);  
 ob\_tmp = min(vecnorm([row.';col.']-x(1:2).'));  
 ob\_distance = min(ob\_distance,ob\_tmp);  
 end  
 end  
 % 距离障碍物过近 => 不考虑该速度情况  
 if ob\_distance < 1.5  
 break;  
 end  
 % 距离障碍物过远 => 限制大小，避免影响总奖励  
 if ob\_distance > 6  
 ob\_distance = 6;  
 end  
 ob\_reward = ob\_distance;  
  
 % 保存奖励  
 reward = [reward;[v w angle\_reward ob\_reward v\_reward]];   
 end  
 end  
 % 若所有速度空间都不满足障碍物距离条件 => 静止  
 if isempty(reward)  
 action = [0 0];  
 return  
 end  
 % 分别归一化各类奖励  
 if sum(reward(:,3))~=0  
 reward(:,3) = reward(:,3)/sum(reward(:,3));  
 end  
 if sum(reward(:,4))~=0  
 reward(:,4) = reward(:,4)/sum(reward(:,4));  
 end  
 if sum(reward(:,5))~=0  
 reward(:,5) = reward(:,5)/sum(reward(:,5));  
 end  
 % 总奖励  
 reward\_ = reward(:,3:5)\*[0.08;0.5;0.1];  
 % 随机选取奖励值最大的速度  
 index = find(reward\_ == max(reward\_));  
 index = index(randi(length(index)));  
  
 action = [reward(index,1) reward(index,2)];  
end

由于DWA较好的避障能力，故其局部跟踪点可以设置为A\*算法规划出路径上较远的路径点。当距离目标点较近时，设置目标点为跟踪点。实现代码为：

if length(x\_) > 5  
 x = x\_(6);  
 y = y\_(6);  
else  
 x = self.dest(1);  
 y = self.dest(2);  
end

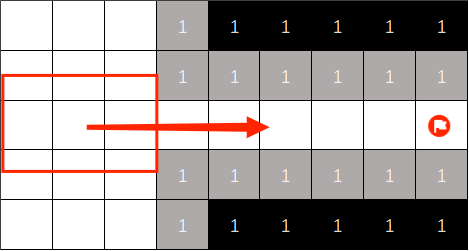
将当前地图、小车当前状态和跟踪点输入DWA中，得到线速度和角速度。

action\_dwa = self.dwa(map,observation,[x y]);

只使用DWA算法的进行速度规划时，相比于只使用P控制器，可以在只对地图膨胀一圈的条件下，很好地避开障碍物，同时保持较高的运动速度。

### 动态DWA-P控制器

DWA算法也存在一些问题。由于DWA算法为了避免与障碍物碰撞，判断了整个预测轨迹与障碍物间的距离差，而预测轨迹采用相同的速度组合，并不符合实际的情况，容易导致误判与障碍物产生碰撞，可以通过减少预测时间降低这一影响，但减少预测时间后虽然对当前轨迹的判断更加精确，但会增大后面的轨迹避开障碍物的难度。这一问题的存在会导致DWA算法对某些情况可能无法规划出有效速度，小车几乎停止移动，难以到达目标点。例如，障碍物间隔极小的情况，如下图所示：



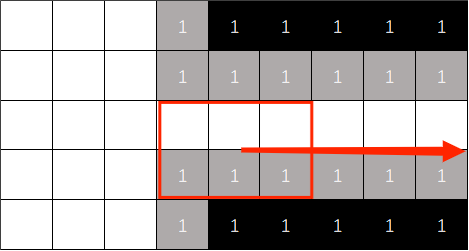
障碍物间隔极小

小车需要通过这条狭窄的道路，若只使用DWA算法，预测轨迹大概率会碰到障碍物（与障碍物相隔较近），因此规划出速度大概率会极低，导致小车在入口附近抖动，无法正常跟踪目标点。

而对于上述情况，P控制器就可以很好地处理。对于这样只有一条通道的轨迹，小车只需很好地跟随该路径，即可通过。P控制器就具有很好的跟随能力。为了避免小车运动太快，在转弯时容易与障碍物相撞的问题，将小车的线速度设为较低值，同时使用P控制器控制角速度。实现代码为：

% 计算角度差  
theta = atan2d(y-observation.agent.y,x-observation.agent.x)/180\*pi;  
beta = self.diff(theta,observation.agent.h)\*0.45;  
% 角度P控制器  
action = [0.45,beta];

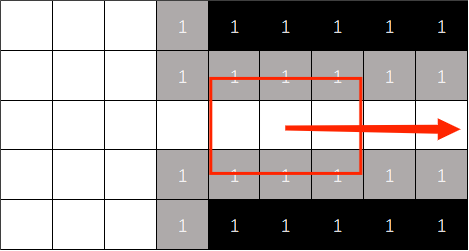
然而，由于地图是离散化的，若直接使用A\*算法规划的点作为跟踪点会导致小车与障碍物相距较近，如下图所示：



直接使用A\*算法规划的点

这会导致小车在后续转弯时与障碍物产生碰撞。由于前面A\*算法在搜索路径时首先考虑右、下方的轨迹，故产生的轨迹会偏向右、下方，则通过判断其跟踪点右、下方点是否为障碍物，将跟踪点对应向左、上方平移0.5个单位，即可解决这个问题。可以表示为：

最终规划路径如图所示：



平移处理后A\*算法规划的点

为了能够充分利用DWA算法和P控制器，我提出了动态DWA-P控制器算法。其核心思想是主要通过DWA算法来跟踪A\*轨迹，然后通过P控制器辅助处理DWA算法无法处理的部分。重点是如何判断什么时候应该使用DWA算法，什么时候应该使用P控制器。

我使用了较为简单的方式进行判断。记录最近四次DWA算法规划出的线速度，若最近四次DWA算法规划出的线速度之和小于阈值则判定DWA算法陷入了困境，使用P控制器替代进行速度规划。当小车成功离开此区域时，DWA规划出的线速度又会变大，线速度之和重新大于阈值，恢复使用DWA控制器。可以表示为：

总的实现过程及其代码如下：

（1）处理地图，获取相关信息

% 限制角度范围  
observation = self.clip(observation);  
% 获取目标点位置/当前位置/速度限制  
self.dest = [observation.endPos.x observation.endPos.y];  
self.now = round([observation.agent.x observation.agent.y]);  
self.sat = observation.agent.satLevel;  
% 更新障碍物地图  
map = self.updateMap(observation);

（2）处理走出地图的情况

对于走出地图的小车，直接使用P控制器，控制其向地图方向运动。

% 处理走出地图的情况  
if self.now(1) < 1  
 action = [0.5 self.diff(0,observation.agent.h)];  
 return  
elseif self.now(1) > 50  
 action = [0.5 self.diff(pi,observation.agent.h)];  
 return  
end  
if self.now(2) < 1  
 action = [0.5 self.diff(pi/2,observation.agent.h)];  
 return  
elseif self.now(2) > 50  
 action = [0.5 self.diff(-pi/2,observation.agent.h)];  
 return  
end

（3）使用A\*算法规划路径

若无法规划出路径，则暂停小车。

% A Star 规划路径  
route = aStar(map,self.now,self.dest);  
% 若路径为空 => 停止规划  
if isempty(route)  
 action = [0 0];  
 delete(self.last\_plot\_line);  
 return  
end  
% 显示规划路径  
[x\_, y\_] = ind2sub(size(map),route);  
self.showLine(x\_,y\_);

（4）使用DWA算法规划速度

% 设置DWA目标点  
if length(x\_) > 5  
 x = x\_(6);  
 y = y\_(6);  
else  
 x = self.dest(1);  
 y = self.dest(2);  
end  
% DWA速度规划  
action\_dwa = self.dwa(map,observation,[x y]);  
% 存储DWA规划历史速度  
self.last\_v = [self.last\_v(2:end) action\_dwa(1)];

（5）判断DWA算法是否失效，输出控制量

* 若DWA算法失效，则使用P控制器；
* 若DWA算法有效，则继续使用DWA算法。

if sum(self.last\_v) < 0.4  
 % 设置P控制器目标点  
 if length(x\_) > 3  
 x0 = x\_(4);  
 y = y\_(4);  
 if map(x0+1,y) == 1  
 x = x0 - 0.5;  
 elseif map(x0-1,y) == 1  
 x = x0 + 0.5;  
 else  
 x = x0 - 0.5;  
 end   
 if map(x0,y+1) == 1  
 y = y - 0.5;  
 elseif map(x0,y-1) == 1   
 y = y + 0.5;  
 else  
 y = y - 0.5;  
 end   
 else  
 x = self.dest(1);  
 y = self.dest(2);  
 if map(x+1,y) == 1  
 x = x - 0.5;  
 else   
 x = x + 0.5;  
 end   
 end  
 % 计算角度差  
 theta = atan2d(y-observation.agent.y,x-observation.agent.x)/180\*pi;  
 beta = self.diff(theta,observation.agent.h)\*0.45;  
 % 角度P控制器  
 action = [0.45,beta];  
else  
 action = action\_dwa;  
end

# 实验结果

所有实验均使用**非全局地图**，基于修改occupyMap中bug后进行测试。图中小点为视为障碍物的部分，蓝色的线为小车运动轨迹。

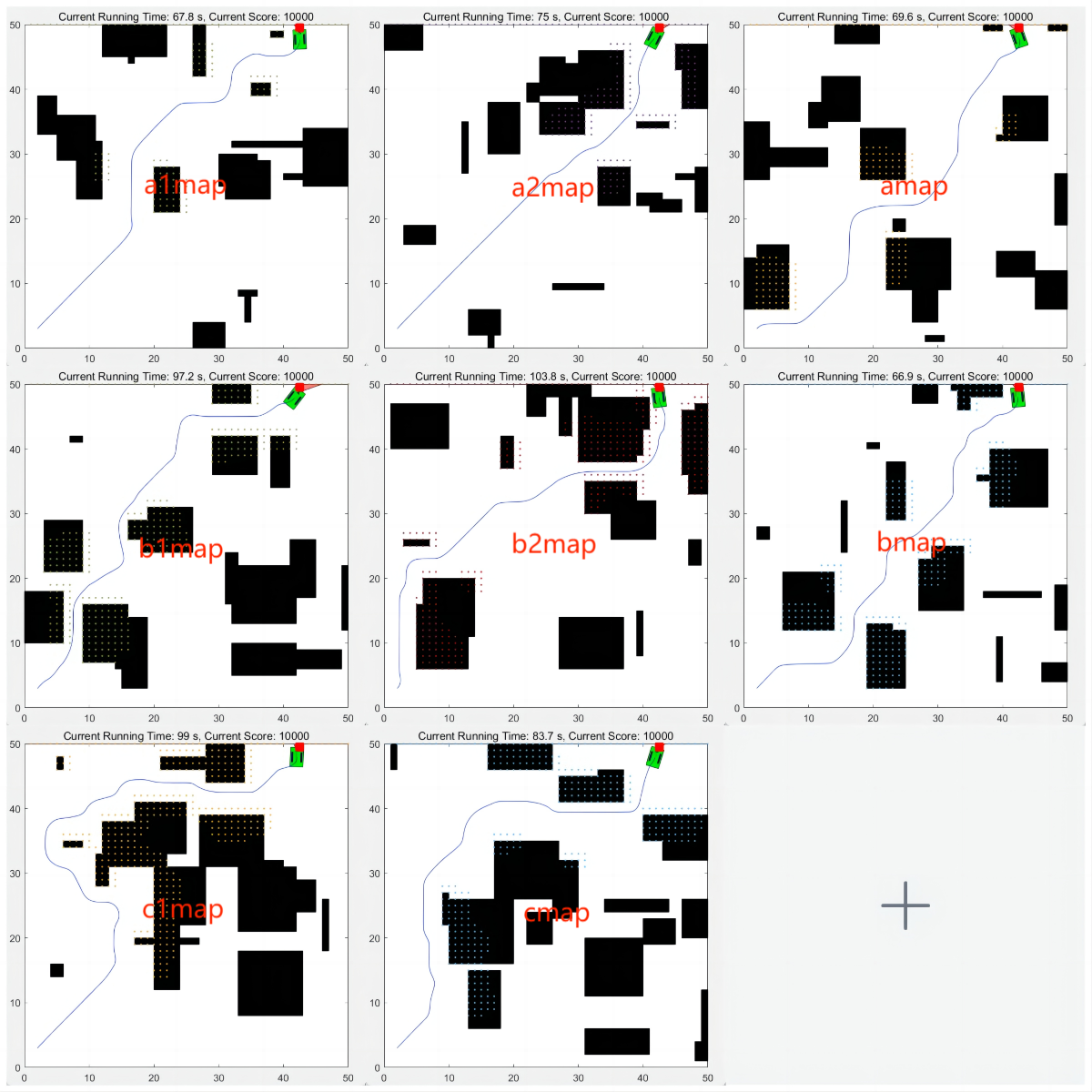
Bug为RangeFinder.m文件中occupyMap函数：

修改前：coords=rm\*coords+self.offset+[x;y];

修改后：coords=rm\*(coords+self.offset)+[x;y];

## 设定地图

在设定的8个地图上均取得了10000分。实验结果如下图所示，录制视频存储在movies文件夹中。

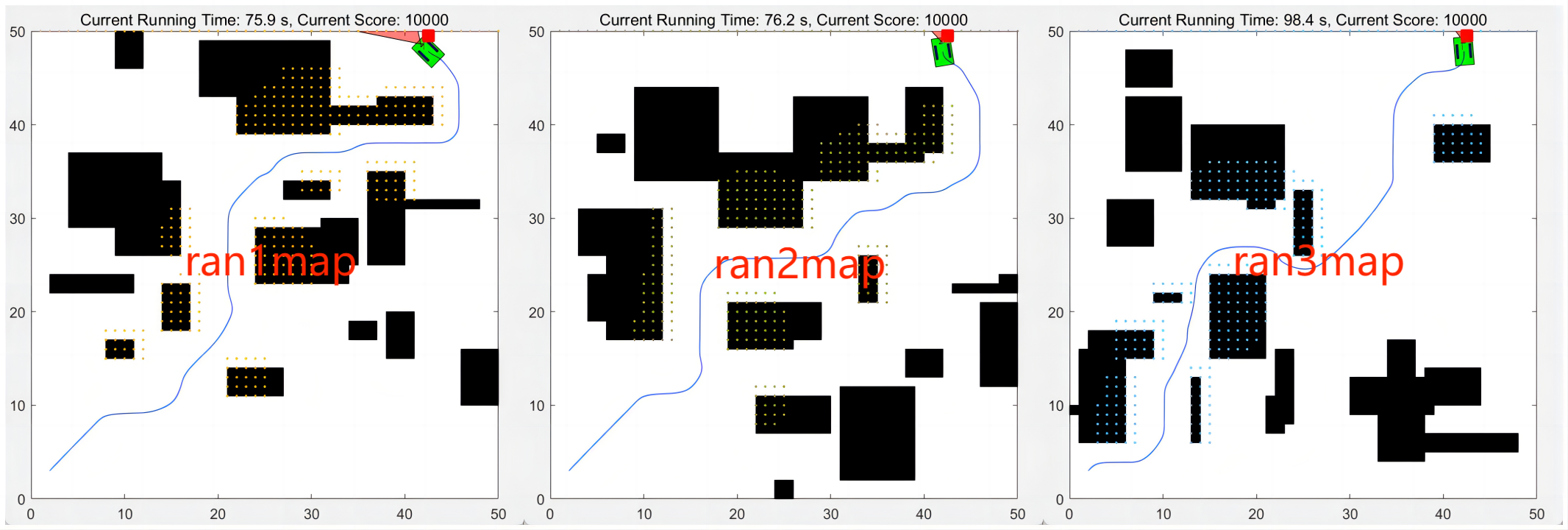


设定地图实验结果

成功在限定时间内，在不与障碍物碰撞的条件下，到达目标点。

## 随机地图

对于能够到达目标点的地图，小车在测试中均能到达目标点，并获得10000分。实验视频存储在movies文件夹中，三个随机生成的地图保存在maps文件夹中。实验结果如下图所示：



随机地图实验结果

综合上述实验结果可以发现，小车对地图的鲁棒性较强，对大多数地图都能得到满分。

## 智能小车障碍

动态更新A\*算法与动态DWA-P控制器算法具有一定的躲避移动障碍物的能力。

由于小车只能监测到前方的障碍物，无法感知左右两边和后方的智能小车，故小车目前只能避开前方的智能小车。且为了避免反向移动无法感知障碍物的问题，DWA设置中反向速度较低，无法避开距离较近且直接向小车冲来的智能小车。

在a1map上进行实验，设置smartAgent个数为5，实验结果为：

表 -

| 原始规划路径 | 前方遇到非直撞小车 | 新规划路径 |
| --- | --- | --- |
| a1 | smart1_013 | smart1 |

可以发现小车主动向左旋转以避开前方移动智能小车。

但对于主动撞向小车的智能小车，小车无法避开，尤其从雷达探测不到的地方。躲避失败的例子（smartAgent个数为10）如下表所示：

表 -

| 右侧智能小车撞击 | 起始点智能小车 | 复杂地段前方智能小车撞击 |
| --- | --- | --- |
| smart2 | smart3 | smart4 |

对于下列例子，理论上也无法避开：

* 智能小车不在雷达探测器视野范围内
* 小车起点处有智能小车
* 多小车夹击

由于算法本身限制（反向线速度低），对于前方障碍物，若智能小车与小车运动方向相同，小车会降低速度等待。但当智能小车与小车运动方向相反时，若横向重叠区域较小，则小车可以倾斜避开，但若横向重叠区域较大，由于速度限制，小车无法避开。示意图如下：

表 -

| 横向重叠区域大 | 横向重叠区域小 |
| --- | --- |
| ss | sss |

# 总结

实验中，提出了动态更新A\*算法作为路径规划，动态DWA-P控制器作为速度规划的规划算法。基于此算法，在非全局地图条件下，小车在设定的8个地图上都取得了满分，同时对随机地图的鲁棒性也比较高，在三个随机生成的地图上都取得了满分。也具有一定的避开动态障碍物的能力。

但目前仍然存在一些限制：

* 计算开销大。 每一次都要重新搜索地图，并在速度空间上遍历，导致了大量的重复计算。可以将A\*算法修改为D\* Lite算法。D\* Lite算法由于每次从终点开始搜索，在更新地图后，可以利用之前的搜素结果，加速搜索。同时对DWA算法可以将双重for循环改为矩阵运算以加速计算。
* 最终结果可能会受到路径规划结果的影响。 在非全局地图条件下，更新障碍物后，A\*算法搜索结果可能产生很大的变化，比如搜索路径在障碍块两边来回变化，这就对速度规划算法的鲁棒性要求较大，可能在左右旋转的过程中检测出更多的障碍物从而规划出更好的轨迹，但在调试过程中也出现过超参数不那么合适时，来回变化的搜索结果导致小车撞上障碍物的情况。
* 对动态障碍物的避障能力有限。 由于DWA速度空间对反向速度的处理效果不佳，同时P-控制器不单独具有避障的能力，难以避开撞向小车的动态障碍物。要想更好地避开动态障碍物，需要估计障碍物的速度，从而利用物理模型进行判断。

# 实现代码

classdef Policy < handle  
 properties  
 now; % 当前位置  
 dest; % 目标点位置  
 sat; % 速度限制  
 last\_v; % 历史速度  
 obs\_map; % 静态障碍物地图  
 last\_plot\_map; % 地图绘制句柄  
 last\_plot\_line; % 路径绘制句柄  
 traj; % 总轨迹  
 last\_plot\_traj; % 轨迹绘制句柄  
 end  
  
 methods  
   
 function self = Policy()  
 self.obs\_map = zeros([50 50],"logical");  
 % 记录4个历史速度  
 self.last\_v = ones([1 4]);  
 self.traj = [];  
 end  
   
 function action=action(self,observation)  
 % 记录轨迹  
 self.traj = [self.traj;[observation.agent.x observation.agent.y]];  
 self.showTraj();  
 % 限制角度范围  
 observation = self.clip(observation);  
 % 获取目标点位置/当前位置/速度限制  
 self.dest = [observation.endPos.x observation.endPos.y];  
 self.now = round([observation.agent.x observation.agent.y]);  
 self.sat = observation.agent.satLevel;  
 % 更新障碍物地图  
 map = self.updateMap(observation);  
   
 % 处理走出地图的情况  
 if self.now(1) < 1  
 action = [0.5 self.diff(0,observation.agent.h)];  
 return  
 elseif self.now(1) > 50  
 action = [0.5 self.diff(pi,observation.agent.h)];  
 return  
 end  
 if self.now(2) < 1  
 action = [0.5 self.diff(pi/2,observation.agent.h)];  
 return  
 elseif self.now(2) > 50  
 action = [0.5 self.diff(-pi/2,observation.agent.h)];  
 return  
 end  
   
 % A Star 规划路径  
 route = aStar(map,self.now,self.dest);  
 % 若路径为空 => 停止规划  
 if isempty(route)  
 action = [0 0];  
 delete(self.last\_plot\_line);  
 return  
 end  
 % 显示规划路径  
 [x\_, y\_] = ind2sub(size(map),route);  
 self.showLine(x\_,y\_);  
  
 % 设置DWA目标点  
 if length(x\_) > 5  
 x = x\_(6);  
 y = y\_(6);  
 else  
 x = self.dest(1);  
 y = self.dest(2);  
 end  
 % DWA速度规划  
 action\_dwa = self.dwa(map,observation,[x y]);  
 % 存储DWA规划历史速度  
 self.last\_v = [self.last\_v(2:end) action\_dwa(1)];  
 % sum(self.last\_v) < 0.4 => DWA无法正常规划，启用P控制器  
 % sum(self.last\_v) >= 0.4 => 使用DWA规划  
 if sum(self.last\_v) < 0.4  
 % 设置P控制器目标点  
 if length(x\_) > 3  
 x0 = x\_(4);  
 y = y\_(4);  
 if map(x0+1,y) == 1  
 x = x0 - 0.5;  
 elseif map(x0-1,y) == 1  
 x = x0 + 0.5;  
 else  
 x = x0 - 0.5;  
 end   
 if map(x0,y+1) == 1  
 y = y - 0.5;  
 elseif map(x0,y-1) == 1   
 y = y + 0.5;  
 else  
 y = y - 0.5;  
 end   
 else  
 x = self.dest(1);  
 y = self.dest(2);  
 if map(x+1,y) == 1  
 x = x - 0.5;  
 else   
 x = x + 0.5;  
 end   
 end  
 % 计算角度差  
 theta = atan2d(y-observation.agent.y,x-observation.agent.x)/180\*pi;  
 beta = self.diff(theta,observation.agent.h)\*0.45;  
 % 角度P控制器  
 action = [0.45,beta];  
 else  
 action = action\_dwa;  
 end   
 end  
  
 function action = dwa(self,map,observation,dest)  
 % 速度空间  
 v\_space = -0.3:0.05:self.sat(1);  
 w\_space = -self.sat(2):0.05:self.sat(2);  
 % x y theta  
 x0 = [observation.agent.x observation.agent.y observation.agent.h];  
 % 轨迹预测分辨率/时长  
 dt = 0.3;  
 k = 5;  
 % 初始化奖励  
 reward = [];  
 % 获取障碍物  
 [row,col] = find(map==1);  
 for v=v\_space  
 for w=w\_space  
 % 初始化轨迹  
 x = x0;  
 trajectory = x;  
 % 计算后续轨迹  
 for t=dt:dt:dt\*(k-1)  
 x = [x(1)+v\*cos(x(3))\*dt x(2)+v\*sin(x(3))\*dt x(3)+w\*dt];  
 trajectory = [trajectory;x];  
 end  
 % 跟踪方向奖励  
 theta = atan2d(dest(2)-x(2),dest(1)-x(1))/180\*pi;  
 angle\_reward = 180-abs(self.diff(theta,x(3)))/pi\*180;  
 % 速度大小奖励  
 v\_reward = abs(v);  
  
 % 据最近障碍物距离奖励  
 ob\_distance = inf;  
 % 找到最近障碍物距离  
 if ~isempty(row)  
 for t=1:k  
 x = trajectory(t,:);  
 ob\_tmp = min(vecnorm([row.';col.']-x(1:2).'));  
 ob\_distance = min(ob\_distance,ob\_tmp);  
 end  
 end  
 % 距离障碍物过近 => 不考虑该速度情况  
 if ob\_distance < 1.5  
 break;  
 end  
 % 距离障碍物过远 => 限制大小，避免影响总奖励  
 if ob\_distance > 6  
 ob\_distance = 6;  
 end  
 ob\_reward = ob\_distance;  
  
 % 保存奖励  
 reward = [reward;[v w angle\_reward ob\_reward v\_reward]];   
 end  
 end  
 % 若所有速度空间都不满足障碍物距离条件 => 静止  
 if isempty(reward)  
 action = [0 0];  
 return  
 end  
 % 分别归一化各类奖励  
 if sum(reward(:,3))~=0  
 reward(:,3) = reward(:,3)/sum(reward(:,3));  
 end  
 if sum(reward(:,4))~=0  
 reward(:,4) = reward(:,4)/sum(reward(:,4));  
 end  
 if sum(reward(:,5))~=0  
 reward(:,5) = reward(:,5)/sum(reward(:,5));  
 end  
 % 总奖励  
 reward\_ = reward(:,3:5)\*[0.08;0.5;0.1];  
 % 随机选取奖励值最大的速度  
 index = find(reward\_ == max(reward\_));  
 index = index(randi(length(index)));  
   
 action = [reward(index,1) reward(index,2)];  
 end  
   
 % 纯P控制尝试  
 function action = pMove(self,map,observation)  
 route = aStar(map,self.now,self.dest);  
 [x, y] = ind2sub(size(map),route);  
 self.showLine(x,y);  
 if length(x) > 4  
 x = x(3);  
 y = y(3);  
 else  
 x = self.dest(1);  
 y = self.dest(2);  
 end  
 theta = atan2d(y-observation.agent.y,x-observation.agent.x)/180\*pi;  
 beta = theta - observation.agent.h;  
 action=[1,beta];  
 end  
  
 function map = updateMap(self,observation)  
 % 更新障碍物地图  
 % 存储固定障碍物  
 self.obs\_map = self.obs\_map | observation.scanMap == 1;  
 % 添加移动障碍物  
 map = self.obs\_map | observation.scanMap>1;  
 % 膨胀地图  
 map = self.swellMap(map);  
 self.showMap(map);  
 end  
  
 function map = swellMap(self,origin)  
 % 膨胀一圈障碍物  
 map\_tmp = bwmorph(origin,'dilate',1);  
 % 设置顶上为障碍物 => 避免小车在目标点附近跑出地图  
 map\_tmp(:,50) = 1;  
 % 设置目标点附近不含障碍物 => 避免膨胀过程中将目标点设为障碍物  
 map\_tmp(self.dest(1)-1:self.dest(1)+1,self.dest(2)-1:self.dest(2)+1) = 0;  
 map = map\_tmp;  
 end  
   
 function showMap(self,map)  
 % 显示障碍物地图  
 hold on;  
 [x,y] = find(map==1);  
 delete(self.last\_plot\_map);  
 self.last\_plot\_map = plot(x,y,'.');  
 end  
  
 function showLine(self,x,y)  
 % 显示A Star规划得到的路径  
 hold on;  
 delete(self.last\_plot\_line);  
 self.last\_plot\_line = plot(x,y);  
 end  
  
 function showTraj(self)  
 % 显示小车走过的路径  
 hold on;  
 delete(self.last\_plot\_traj);  
 self.last\_plot\_traj = plot(self.traj(:,1),self.traj(:,2),'b');  
 end  
  
 function d = diff(~,a,b)  
 % 角度相减  
 d = wrapToPi(a-b);  
 end  
  
 function observation = clip(~,observation)  
 % 限制角度范围  
 h = observation.agent.h;  
 observation.agent.h = wrapToPi(h);  
 end  
 end  
end  
  
%% A Star  
  
function path = aStar(map, start, goal)  
 % 搜索失败 => 输出空  
 path = [];  
 % 地图大小  
 map\_size = size(map);  
 map\_lenth = numel(map);  
 % 转为线性索引 => 方便直接使用  
 start = sub2ind(map\_size,start(1),start(2));  
 goal = sub2ind(map\_size,goal(1),goal(2));  
  
 % 初始化 open set  
 open\_set = false(map\_size);  
 open\_set(start) = true;  
 % 初始化 close set  
 close\_set = false(map\_size);  
 % 记录上一个节点  
 parents = zeros(1, map\_lenth);   
 % 初始化G得分 => 起点到指定方格的移动代价  
 g\_score = inf(map\_size);  
 g\_score(start) = 0;  
 % 初始化F得分 => 预估总代价  
 f\_score = inf(map\_size);  
 f\_score(start) = getH(map\_size,start,goal);  
   
 sqrt2 = sqrt(2);  
  
 % open set 非空  
 while any(open\_set(:))  
  
 % 最小F得分  
 open\_f\_score = f\_score.\*open\_set;  
 [~, current] = min(open\_f\_score(:));  
  
 % 若到达终点  
 if current == goal  
 % 获得轨迹  
 path = current;  
 exists = find(parents);  
 while any(current == exists)  
 current = parents(current);  
 path = [current path];   
 end  
 return  
 end   
   
 % current => close  
 open\_set(current) = false;  
 close\_set(current) = true;  
 f\_score(current) = inf;  
   
 % 当前G得分  
 current\_g\_score = g\_score(current);  
   
 % 获取8邻域  
 [xc,yc] = ind2sub(map\_size,current);  
 ns = [  
 xc + 1 yc + 1 sqrt2   
 xc + 1 yc 1   
 xc + 1 yc - 1 sqrt2   
 xc yc - 1 1   
 xc - 1 yc - 1 sqrt2   
 xc - 1 yc 1   
 xc - 1 yc + 1 sqrt2   
 xc yc + 1 1   
 ];  
 % 去除非法邻域点  
 ns = ns(ns(:,1)>0&ns(:,1)<=map\_size(1)&ns(:,2)>0&ns(:,2)<=map\_size(2),:);  
   
 % 去除不可达/close set中邻域点  
 neighbors = sub2ind(map\_size,ns(:,1),ns(:,2));  
 valid\_ns = ~map(neighbors) & ~close\_set(neighbors);  
 neighbors = neighbors(valid\_ns);  
 distances = ns(valid\_ns,3);  
  
 % 添加邻域点  
 open\_set(neighbors) = true;  
  
 % 邻域点当前路径G得分  
 tmp\_g\_scores = current\_g\_score + distances;  
   
 % 更新G变小的邻域点  
 index\_better = tmp\_g\_scores < g\_score(neighbors);  
 better\_neighbors = neighbors(index\_better);   
 parents(better\_neighbors) = current;  
 g\_score(better\_neighbors) = tmp\_g\_scores(index\_better);  
 f\_score(better\_neighbors) = g\_score(better\_neighbors) + getH(map\_size,better\_neighbors,goal);  
  
 end  
end  
  
function cost = getH(map\_size,from,to)  
 % Heuristic cost  
 [x\_from,y\_from] = ind2sub(map\_size,from);  
 [x\_to,y\_to] = ind2sub(map\_size,to);  
 cost = sqrt((x\_from - x\_to).^2 + (y\_from - y\_to).^2);  
end