《人工智能与智能驾驶基础》课程作业

路径规划

学院：汽车学院

姓名：贾林轩

学号：1853688

软件版本：MATLAB 2021b

一． 建立栅格地图

采用网络占用法表现地图。这种方法会将整个地图区域分割为若干个单位网络栅格，每个栅格的最基本状态有非占用和占用。

非占用栅格，即机器人可以自由通过的栅格，比如下图中的白色栅格。占用栅格也就是机器人无法通过的栅格，即障碍物，比如下图中的黑色栅格。单位网络栅格的大小决定了地图的分辨率，单元格的大小往往根据应用场景的不同设置为不同的数值。

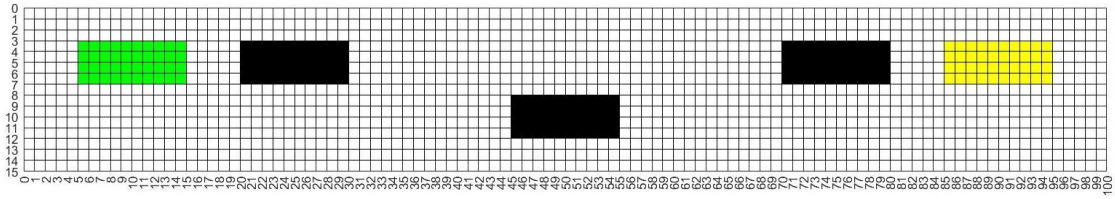
Shangeditu1.m 文件是建立栅格地图的相关代码。

1. 建立原始栅格地图

首先，设置颜色列表 color_list，然后用 colormap 命令读取颜色列表。

然后设置场地大小，设置起始位置、终点位置及障碍物位置。

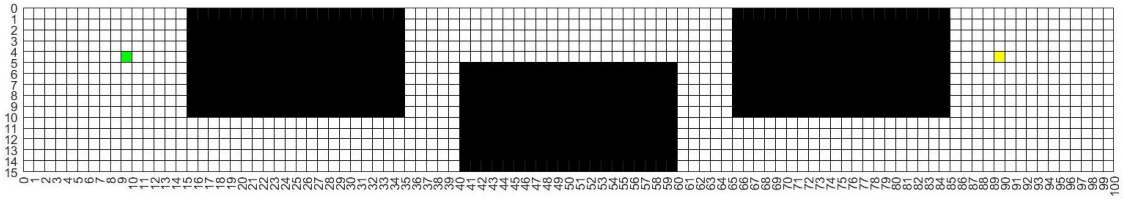
结果如下：



2. 进行防碰撞化处理

利用包围圆半径，对障碍物进行膨胀处理，同时将起始位置和终点位置抽象为点，同时路径规划可以以点为基准进行。

结果如下：



二． 进行路径规划

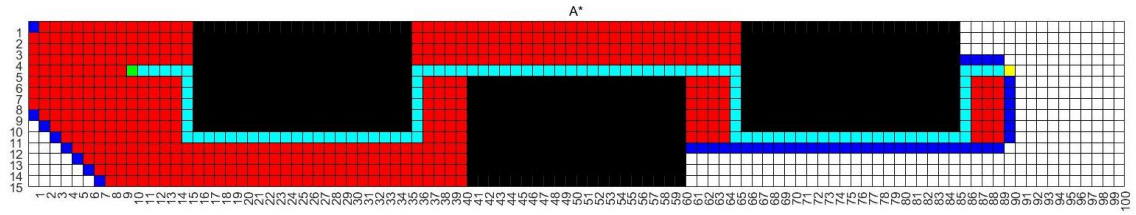
Astar.m 中为使用A*搜索算法进行路径规划的程序，Dijkstra.m 中为使用Dijkstra搜索算法进行路径规划的程序。

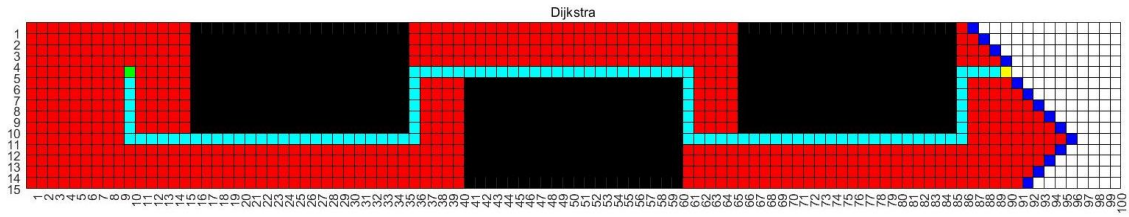
Dijkstra 是一个迭代的过程，每个节点具备两个属性值，一个是 node_cost，这个 node_cost 记录了当前节点到起点的已知的最小总代价值，这个 node_cost 会在算法迭代过程中由于规划出来的路径不相同而被更新成不同的值。另一个属性值是 parent，parent 表明了根据当下规划出来的路径，该节点对应的父节点。在每轮迭代的开始，我们会找出未完成遍历列表中的所有节点中 node_cost 最小的那个节点并设置为 current_node，并将这个被

选中的 current_node 移除未完成列表，设置为已完成遍历。每轮迭代中，我们都会对 current_node 的所有未完成遍历的相邻节点进行单独的判定，判定相邻节点当前的 node_cost 是否大于 current_node 的 node_cost 与连接边的代价值之和。如果大于的话，将相邻节点的 node_cost 更新为 current_node 的 node_cost 加上连接边的代价值，并且将相邻节点的 parent 更新为 current_node，如果这个相邻节点尚未被加入未完成列表，则将其添加进列表中。如果小于的话，则保持原样，不执行任何操作。

A*可以看作 Dijkstar 和贪婪算法的结合，Dijkstar 在规划中注重的是当前节点到起点的总代价值，也称后退代价。而贪婪算法注重的是当前节点到终点的总代价值，也称前进代价。那么所谓的结合实际上就是说，A*同时考虑到了后退代价和前进代价。从表达式上看就是：后退代价+前进代价；评价函数将会计算当前节点的后退、前进代价的总和，这个数值表明这个节点在规划中的重要程度，或者说是优先考虑程度。在 A*中节点的评价函数返回值越小就表明该节点的重要性越高，或者说更应该被优先考虑。A*算法在每轮迭代中都会选择未完成列表中评价函数返回值最小的节点。那么有了评价函数的介入，A*算法看待节点的态度就不再是“一视同仁”了，根据评价函数的结果，A*会将节点分“三六九等”，那么 A*就获得了目的性。前面说过 A*是 Dijkstar 和贪婪算法的组合，那么 g(n)和 h(n)占 f(n)的比重就决定了 A*的行为，假如 h(n)占更高的比重，那么 A*就表现得更像贪婪算法，假如 g(n)占更高的比例，那么 A*就表现得更像 Dijkstar。举个极端例子，当我们设 h(n) = 0 的时候，A*便会退化回 Dijkstar。当我们设 g(n) = 0 时，A*就变成了贪婪算法。所以权衡好 g(n)和 h(n)的比例是至关重要的。在只允许上下左右移动的栅格地图中，我们一般使用曼哈顿距离作为启发函数，用于估算两点之间的距离。因为曼哈顿距离即满足一致性和“h(goal) = 0“的特性，而且算法简单，计算成本低。

运行主程序 run.m，其结果如下：





命令行窗口

```
A* plan succeeded! iteration times: 575 path length: 104
Dijkstra plan succeeded! iteration times: 766 path length: 104
>>
```

可以看出，尽管两种搜索算法都找到的最短路径的距离相同，但是A*算法迭代了 575 次，而Dijkstra算法迭代了 766 次。可知A*算法的性能相对更为优异。

三． 路径还原

将 Astar 和 Dijkstra 函数中的路径信息返回到主程序中，并在原栅格地图中对上述路径信息中的坐标点进行赋值，即可将路径还原到未膨胀前的原始栅格地图中，检查路径的合理性。

结果如下: