1 作业要求

设计一个场景，包括起点，终点，障碍物，采用 2 种及以上的规划方法实现规划过程，采用 2 种及以上的横向控制方法实现路径跟踪过程，要求实现模拟移动机器人运动的过程，机器人本体的运动控制模型不限。

2 环境与定义介绍

首先我们完成作业的环境是基于 CoppeliaSim Edu 软件的拟真环境，用其中的模拟物件搭建了作业所需的地图以及机器人。在 CoppeliaSim Edu 软件的拟真环境的基础上，利用软件自带的 python 接口，用相关的 python 代码控制小车的运动。下面我们具体介绍一下每一部分。

2.1 CoppeliaSim Edu 设置

在 CoppeliaSim Edu 中，我们主要介绍地图设置和小车设置两部分，其中小车部分也包含了对小车的运动模型以及相关约束的介绍。

2.1.1 小车设置

对于小车，我们所使用的是一个平面二轮差速小车模型，其包含两个只有前后方向速度的运动轮和一个用于支持的辅助万向轮，具体如下图所示。

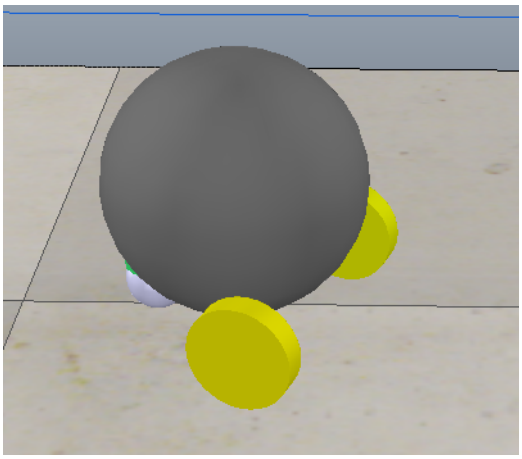


Figure 1: 二轮差速小车模型

对于小车的运动约束，由它的运动模型不难发现，这是一个有三个自由度但只有两个机动度的模型，它通过两个差速车轮能够控制自己的朝向和朝向方向上的速度。也就是说，它能直行、倒推，同时可以进行转弯，但不能左右平移。为了更严格地控制小车的行动，我们设置它不能原地转弯，不能倒退运动。

2.1.2 地图设置

在 CoppeliaSim Edu 中，我们用颜色区分不同物体。我们设置一个绿色圆盘，作为目的地；设置一些白色长方体作为障碍物；对于小车，我们分别在其前后设置一个红色圆盘和一个蓝色圆盘，用以区分其朝向，这两个圆盘与小车绑定，会与其一起进行运动。具体地图如下图所示 之所以要如此严格地区分颜色，是因为 CoppeliaSim Edu 软件到

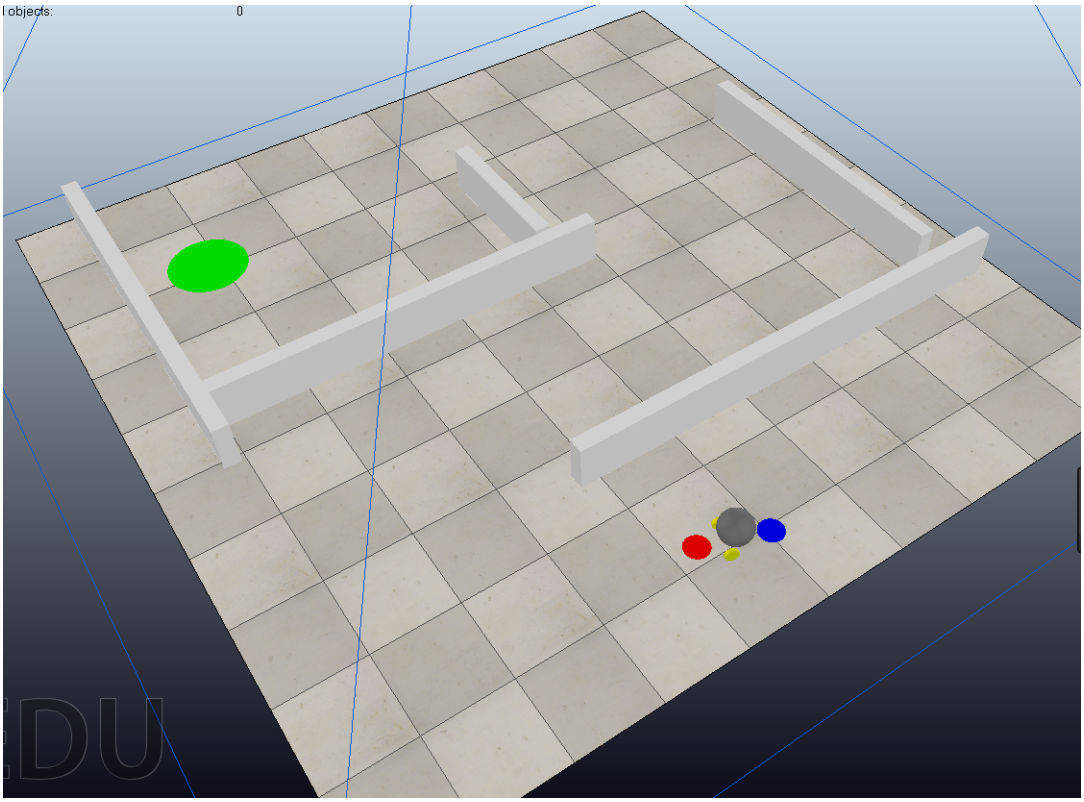


Figure 2: 地图布置俯视图

python 的信息交互由基于 opencv 的图像识别完成，观察图 2，可以看到地图上方的蓝色线条，这代表地图上方的摄像头的视野范围。我们所编写的 python 代码通过读取该摄像头的信息获取当前时刻的地图状态，进而控制小车行动。

2.2 Python 代码运行环境

我们在 Windows10 系统上通过 pycharm2024 编译器运行 python3.8 代码完成作业。

3 算法设置与代码实现

这一部分我们主要介绍所选择的算法并简要地说明实现的方式，最后大致地展示它们的成果，篇幅有限，对应方法的完整录屏视频已经打包在邮件中。同时声明，本文所使用的用于检验路径规划效果和路径跟踪效果的 reporter 类来自于导师，相关代码同样在附件中。

3.1 路径规划算法

这一部分我们介绍所选用的路径规划算法，分别是 A* 算法和 DIjkstra 算法。注意，为了给保证下面的路径平滑的操作更简单，我们这里设置不单止要避开障碍物，还要求所规划的路径要至少与障碍物保持一定的距离，这里我们设置为 30。

3.1.1 A* 算法

A*（A-star）算法是一种在图中寻找从初始节点到目标节点最短路径的启发式搜索算法。它结合了 Dijkstra 算法的确保性（保证找到一条最短路径）和贪心算法的高效性（快速找到目标）。A* 算法通过评估函数 $f(n) = g(n) + h(n)$ 来工作，其中 $g(n)$ 是从起始点到任何顶点 n 的实际成本，而 $h(n)$ 是从顶点 n 到目标的估计最低成本，通常用启发式函数来计算，这个函数需要事先设计来反映实际的地形或环境特征。理想情况下，$h(n)$ 应该不会高估实际的成本，这种情况下，A* 算法保证找到一条最低成本路径。

启发式是衡量地图中一个点到终点距离的算法，是 A* 算法的核心，对算法的性能和效率至关重要。常用的启发式有欧氏距离（欧几里得距离），曼哈顿距离以及切比雪夫距离等。这里我们实现两种启发式，分别是欧氏距离和曼哈顿距离。其中欧氏距离的计算方法如下

$$distance(OU) = \sqrt{(x_{cur} - x_{goal})^2 + (y_{cur} - y_{goal})^2} \tag{1}$$

而曼哈顿距离的计算方法为

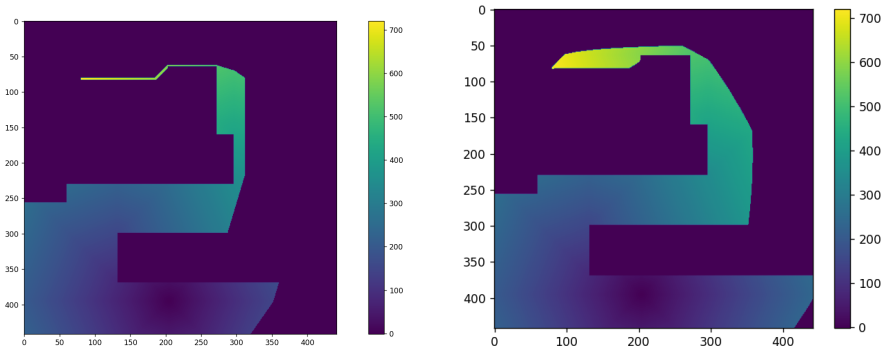
$$distance(Manhaten) = |x_{cur} - x_{goal}| + |y_{cur} - y_{goal}| \tag{2}$$

A* 算法的优点很明显，它既是完备的，也是最优的，但是它也有缺点——非常依赖启发式同时非常占用空间。

在具体的算法实现上，我们用一个 Astar 类实现 A* 寻路方法。在类中，首先用一个二维矩阵 obstacle_map 表示地图，其中元素为 0 表示有障碍物，1 表示没有障碍物；再用一个相同尺寸的二维矩阵 g 表示到达某一点经过的路径长度，且初始化时将 g 的元素全部设置为-1，表示未遍历过，即未进入过 openSet 的点；设置一个 f 矩阵，表示 A* 算法中某一点的 g 值和 h 值的和，初始时设置为无穷大，同时，定义 f=1000000 的点为在 closet 中的点；最后设置一个三维矩阵 last，记录地图上任意自由点到起点的最短路径的上一个顶点。

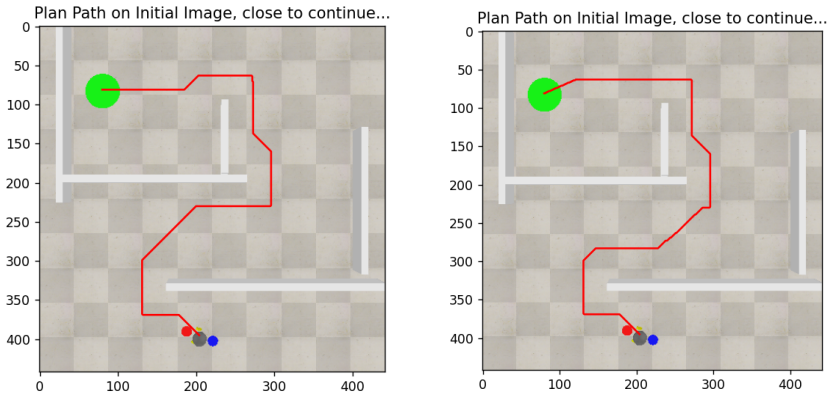
接下来为了迎合小车的运动学约束，我们作出如下修改在检查邻点的步骤中，我们设置小车能够朝向周围八个方向的任意一个，但是针对某一特定方向，小车只能向原方向和旁边两个方向进行运动，为此我们额外设置一个 oriMap 二维矩阵表示小车到达每一个点时的朝向。且设置上下左右的点的距离为一，斜对角的点距离为 1.4。具体设置大致如此，其余过程没有其他特殊设置因此不再赘述，详情请见代码及相关注释。

下面我们分别展示 A* 算法两个启发式的运行结果，首先通过观察已遍历顶点观察两个启发式的区别，接着观察两种方法生成的路径



(a) 曼哈顿距离　　(b) 欧几里得距离

Figure 3: 已遍历点结果对比



(a) 曼哈顿距离　　(b) 欧几里得距离

Figure 4: 所生成路径结果对比

从图 3 和图 4 可以看到，相比之下，欧几里得距离计算了更多的点。同时，通过比较我们也可以发现，曼哈顿距离更急于进行斜方向前进，每当避过（膨胀过的）障碍物之后，总是第一时间进行斜向行走，由其计算方式我们也可以理解这种现象——对于曼哈顿距离，一个斜向的正确前进能减去更多的误差。

接着，我们从耗费时间、路径长度、路径距离障碍物的最小距离和平均距离四个方面来比较两种启发式的结果（表格与下面 DIjkstra 方法一起给出）。

	cost_time	path_length	Min to Obs	Avg to Obs
A* OU	9.61	721.99	31	45.7
A* M	6.91	721.99	31	43.25
Dijkstra	13.84	721.99	31	47.29

观察表格，我们不难发现，欧氏距离所花费的时间比曼哈顿距离的启发式略多，与其检查了更多的点同时计算复杂度更高的现象吻合；此外，两种启发式得到的路径在栅格化抽象的条件下的长度都是一致的，哪怕实际上两条路径并不完全一致，这也体现了 A* 算法的最优性。最后，我们发现两条路径的距离障碍物最小距离都很好地保持了与

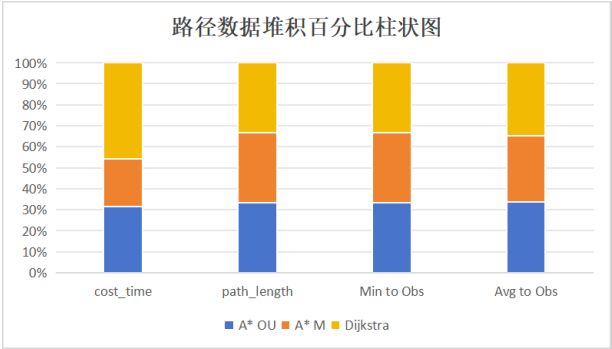


Figure 5: 路径数据百分比堆积柱状图

障碍物的最小距离，多出的 1 误差猜测是由于 opencv 库的识别精度原因。

3.1.2 Dijkstra 算法

DIjkstra 的基本思想是以起始点为中心向外层层扩展，直到扩展到终点为止。在扩展的过程中，始终保持从起点到已求出最短路径的终点之间的路径长度是最短的。它的优点在于实现简单，鲁棒性强，适用于所有权重为正的情况，对图的结构没有特殊要求；缺点在于耗时更长，内存占用也不小。

本文实现的 DIjkstra 方法的实现主要基于上文的 A* 算法的 Astar 类。不难发现，只要将启发式的计算改为恒零，A* 算法就会退化为 DIjkstra 算法。下面给出算法的已遍历点和最终规划路径结果。

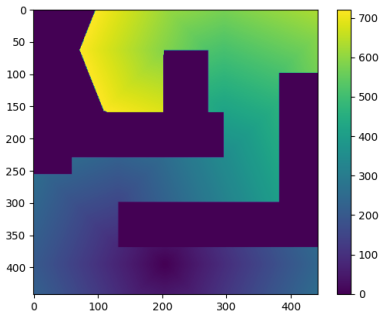


Figure 6: DIjkstra 方法的已遍历点图

由表格观察，相对而言，DIjkstra 结点的结果最差，不仅耗时最长，距离障碍物的平均距离也是最远的。

3.2 路径平滑算法

显然，对于上文的路径规划算法所得到的路径并不适合直接指导机器人移动，因为它们的转向都是突兀的，因此需要进行路径平滑操作这里的路径平滑算法参考的是

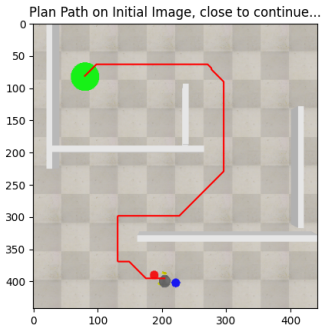
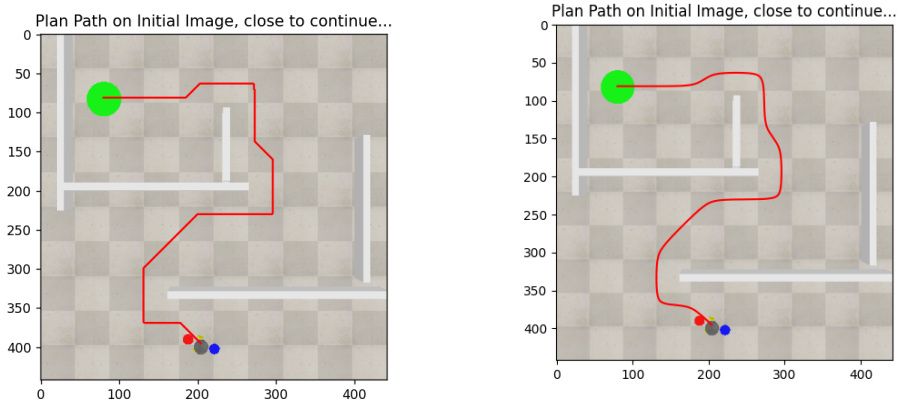


Figure 7: DIjkstra 方法的规划路径图

Understanding Robot Motion: Path Smoothing | by James Teow | Medium。下面以曼哈顿距离为例子，给出平滑后的结果和相关数据对比。



(a) 曼哈顿距离原路径　　(b) 平滑后距离

Figure 8: 路径平滑示例

	path_length	Min to Obs	Avg to Obs	max angle difference
A* M	721.99	31	43.25	45
smoothed	681.6	30.65	42.81	5.23

可见原路径经过平滑后，在转弯的地方变得更加平缓了，同时观察前后路径数据对比，不难发现路径总长度也缩短了，代价则是距离障碍物的距离减小。此外，最明显的区别在于路径中最大转角的变化，由原本的 45 度变为 5.23 度，更适合机器人实际使用。

3.3 路径跟踪算法

这里介绍我们所选用的指导小车跟踪所给路径前进的算法，这里我们选择的是 PID 控制法和 pure pursuit 方法。

3.3.1 PID 控制

PID（Proportional-Integral-Derivative）控制算法是一种经典的控制方法，通过计算误差的比例（P）、积分（I）和微分（D）来实现对被控对象的精确控制。在路径跟踪中，PID 控制器通过计算车辆当前位置与参考路径之间的偏差（即误差），根据比例、积分和微分参数来调整车辆的运动状态（如转向角度和速度），从而实现对参考路径的跟踪。

PID 方法的优点在于结构简单，稳定性强，且通过调整 PID 参数，可以实现较高的控制精度；缺点则是参数调整困难，且在处理非线性系统时可能会出现不稳定的情况。

在具体实现中，我们用了两个 PID 系统控制来控制小车的行动，两个 PID 系统分别控制小车的前进速度和转向。我们先在原路径上以一定间隔采样路径点，然后计算小车与离其最近的路径点的距离和方向误差，并以此输入 PID 系统计算小车左右轮的速度，最后的左右轮速度是两个 PID 系统结果的和。同时，为了防止小车接近路径点时速度趋于 0 的现象，我们给小车一个最低的基础速度。

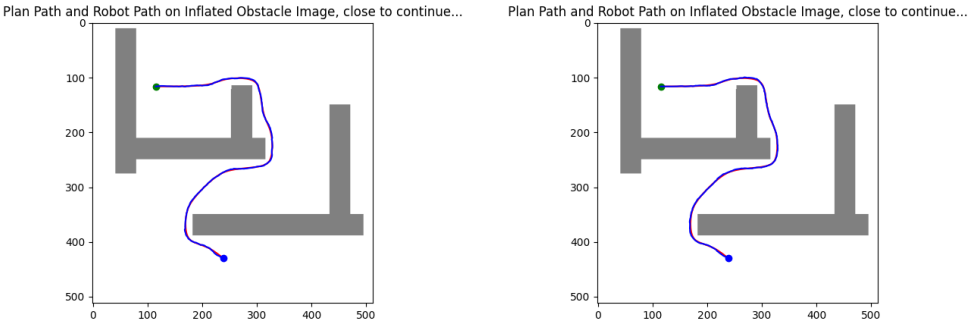
3.3.2 pure pursuit 控制

Pure Pursuit（纯追踪）算法是一种基于几何追踪的路径跟随算法。它通过计算车辆当前位置到预瞄点（goal point）的曲率，使车辆沿着经过预瞄点的圆弧行驶，从而实现轨迹跟踪。Pure Pursuit 算法的核心在于通过设计合理的预瞄距离，计算出轨迹跟踪的控制曲率。

该方法的优点有实现简单，响应快速；确定则是依赖于预瞄距离的选择且对复杂路径的跟踪效果较差。

3.3.3 两种方法结果对比

两种方法指导机器人的实际路径与计划路径的对比图如下图所示，图中红色曲线为计划路径，蓝色曲线为实际路径（然而大部分红色曲线已经被蓝色曲线掩盖）。



(a) PID 控制　　(b) pure pursuit 控制

Figure 9: 路径跟踪结果对比图

下面是对两种控制方法所生成的实际路径的一些数据，分别是实际路径相较于计划路径的路径总长度变化、距离障碍物最小和平均距离变化，以及路径偏差量和路径能量。

观察表中差距，我们发现 PID 方法的偏移更小但能量更大，不过，实际上能量偏大可能

	PID	pure pursuit
路径总长度变化	11.17	10.28
最小障碍物距离变化	0.04	0.87
平均障碍物距离变化	1.38	1.37
路径偏差	274.57	298.43
路径能量	304.21	226.67

是 opencv 库的识别精度导致的，因此这一项指标仅供参考。其余指标也都是 PID 控制较优，体现了 PID 控制方法的精度较高的特点。