華中科技大學

课 程 实 验 报 告

课程名称： 人工智能导论

选题 基于 A* 算法的最优路径规划系统

学号 U202215345

姓名 贾竟一

指导教师 冯琪

报告日期 2023 年 12 月 19 日

计算机科学与技术学院

目　　录

1 项目摘要

本项目旨在设计和实现一个基于 A* 算法的最优路径规划系统，以真实地图为基础，通过对 opencv 图像学操作，得到位图背景加栅格坐标数据的二维平面地图，地图规模适中。

系统允许用户设定起点和终点，并具有一定鲁棒性进行起始点合法性判断。在此基础上，通过可视化界面，动态展示 OPEN 表和 CLOSED 表的变化过程，最终呈现最优路径所需的理论地图距离，并输出栅格地图中在该启发式函数下，所产生的最优路径。用户可选择单步执行或连续执行，观察扩展过和待扩展的所有路径具有良好的可视性和直观性。

在算法的改进方面，系统考虑引入路况信息，以优化启发式函数，采用不同的距离作为估价函数，使路径规划更加实用和适应实际情况，从而提高路径规划的准确性和效率。

关键词：A* 算法，Opencv, 启发式函数，栅格地图。

2 问题描述

2.1 题目背景

随着城市交通和路径规划的复杂性不断增加，如何选择最优路径成为了一个至关重要的问题。A* 算法作为最优路径规划问题的经典算法，可以在机器人自动导航，城市路径规划等领域有着重要的实际意义。该项目旨在开发一个基于 A* 算法，利用真实地图实现，并通过可视化界面展示算法过程的最优路径规划系统。

2.2 A* 算法描述

2.2.1 A* 算法原理

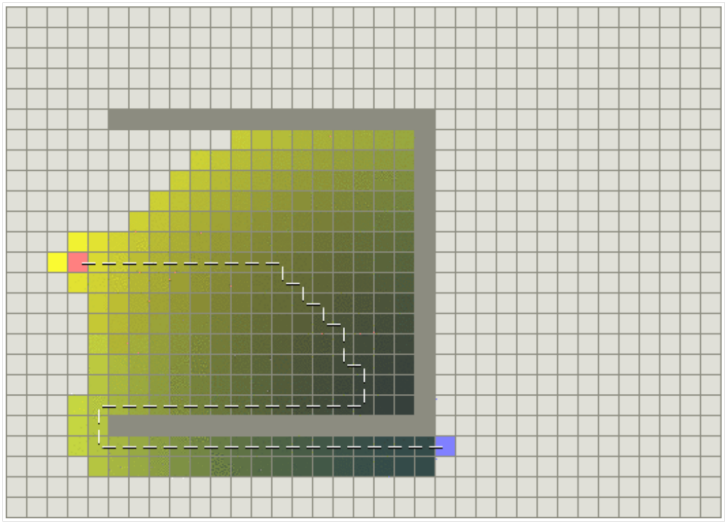

图 2-1

1968 年发明的 A star 算法就是把启发式方法（heuristic approaches）如 BFS，和常规方法如 Dijsktra 算法结合在一起的算法。有点不同的是，类似 BFS 的启发式方法经常给出一个近似解而不是保证最佳解。然而，尽管 A star 基于无法保证最佳解的启发式方法，A star 却能保证找到一条最短路径。

在简单的情况中，它和 BFS 一样快, 在凹型障碍物的例子中，A* 找到一条和 Dijkstra 算法一样好的路径.

A star 把 Dijkstra 算法（靠近初始点的结点）和 BFS 算法（靠近目标点的

结点）的信息块结合起来。在讨论 A star 的标准术语中，g(n) 表示从初始结点到任意结点 n 的代价，h(n) 表示从结点 n 到目标点的启发式评估代价（heuristic estimated cost）。在上图中，yellow(h) 表示远离目标的结点而 teal(g) 表示远离初始点的结点。当从初始点向目标点移动时，A star 权衡这两者。每次进行主循环时，它检查 f(n) 最小的结点 n，其中 f(n) = g(n) + h(n)。

2.2.2 启发函数

启发式函数 h(n) 告诉 A* 从任意结点 n 到目标结点的最小代价评估值。选择一个好的启发式函数是重要的。

1) 一种极端情况，如果 h(n) 是 0，则只有 g(n) 起作用，此时 A 演变成 Dijkstra 算法，这保证能找到最短路径。
2) 如果 h(n) 经常都比从 n 移动到目标的实际代价小（或者相等），则 A 保证能找到一条最短路径。h(n) 越小，A 扩展的结点越多，运行就得越慢。
3) 如果 h(n) 精确地等于从 n 移动到目标的代价，则 A star 将会仅仅寻找最佳路径而不扩展别的任何结点，这会运行得非常快。尽管这不可能在所有情况下发生，你仍可以在一些特殊情况下让它们精确地相等（译者：指让 h(n) 精确地等于实际值）。只要提供完美的信息，A star 会运行得很完美，认识这一点很好。
4) 如果 h(n) 有时比从 n 移动到目标的实际代价高，则 A star 不能保证找到一条最短路径，但它运行得更快。
5) 另一种极端情况，如果 h(n) 比 g(n) 大很多，则只有 h(n) 起作用，A star 演变成 BFS 算法。

2.2.3 伪代码

```
OPEN = priority queue containing START
CLOSED = empty set
while lowest rank in OPEN is not the GOAL:
  current = remove lowest rank item from OPEN
  add current to CLOSED
  for neighbors of current:
    cost = g(current) + movementcost(current, neighbor)
```

```
    if neighbor in OPEN and cost less than g(neighbor):
      remove neighbor from OPEN, because new path is better
    if neighbor in CLOSED and cost less than g(neighbor):  [2]
      remove neighbor from CLOSED
    if neighbor not in OPEN and neighbor not in CLOSED:
      set g(neighbor) to cost
      add neighbor to OPEN
      set priority queue rank to g(neighbor) + h(neighbor)
      set neighbor's parent to current

reconstruct reverse path from goal to start
by following parent pointers
```

2.3 地图选择

在本项目中，我选取了北京中心四环的主要道路作为导航地图 1，武汉汤逊湖作为导航地图 2，对应城市道路规划和湖面避障问题。分别基于 Opencv 图像学处理得到二值化图像，然后利用对像素点的操作，得到了大小规模适中的栅格地图，在此栅格地图上进行最优路径规划。

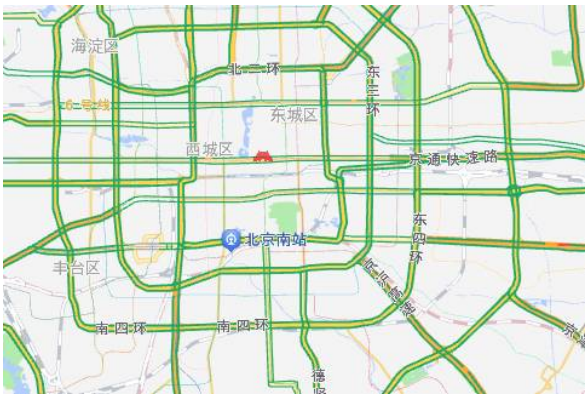

图 2-2　导航地图 1 - 北京

图 2-3　导航地图 2 - 汤逊湖

3 项目实现

3.1 A* 算法

A* 算法在之前以及有过介绍，这里不再赘述，下面主要介绍项目中 A* 算法的实现思路。

本项目中的 A* 算法如下实现:

1) 定义 Node 结构体表示单个节点, 包含坐标和累积代价值等属性。初始时起点和终点作为开头的两个 Node 对象。
2) 根据真实地图构建栅格计算障碍地图, 以 vector 形式存储坐标。
3) 定义估价函数，默认距离使用欧几里得距离作为 h(n) 函数
4) 定义 motion 函数，每个点可以想周围 8 个方向进行扩展。
5) 采用优先队列来保存待搜素的点，如果该点已经被搜素过，则用 visit 数组标记
6) 每次弹出队头结点，即根据估计函数得到的距离目标点最近的点，并将该点加入 CLOSED 表。判断该点是否到达目标点，若未到达，则扩展其八个连接方向的下一个结点。
7) 若新的点合法且未被访问过，则加入 OPEN 表，标记为待扩展结点，加入优先队列。
8) 最后利用回溯函数计算得到最优路径，并画出最终答案

总体来说, 该实现利用 A* 算法有效解决了给定地图环境下 robot 从起点到目标点的最短路径问题。

下面对其中重点部分的代码实现以注释的形式做一定说明。

3.1.1 motion

```
//定义了机器人在规划过程中可以移动的方向和代价
std::vector<Node> get_motion_model(){
  return {Node(1, 0, 1),
          Node(0, 1, 1),
          Node(-1, 0, 1),
          Node(0, -1, 1),
```

```
        Node(-1, -1, std::sqrt(2)),
        Node(-1, 1, std::sqrt(2)),
        Node(1, -1, std::sqrt(2)),
        Node(1,  1, std::sqrt(2))};
}
```

3.1.2 Heuristic function

```
//定义了机器人在规划过程中可以移动的方向和代价
/**
 * 计算启发式函数（Heuristic
    function）的值，用于估计两个节点之间的代价（距离）。
 *
 * @param n1 第一个节点
 * @param n2 第二个节点
 * @param w 权重参数，默认为1.0
 * @return 两个节点之间的启发式函数值（代价）
 */
float calc_heuristic(Node* n1, Node* n2, float w = 1.0) {
  // 根据节点的坐标计算欧几里得距离，并乘以权重参数w
  return w * std::sqrt(std::pow(n1->x - n2->x, 2) + std::pow(n1->y
      - n2->y, 2));
}
```

3.1.3 最优路径回溯函数

```
std::vector<std::vector<float> > calc_final_path(Node * goal,
    float reso, cv::Mat& img, float img_reso){
  std::vector<float> rx;
  std::vector<float> ry;// 存储路径的x ，y坐标
  Node* node = goal;
  int total_dis = 0;
  // 从目标节点开始回溯路径，直到达到起始节点（起始节点的父节点为NULL）
  while (node->p_node != NULL){
```

```
    node = node->p_node;
    // 将离散化的路径坐标（乘以分辨率reso）添加到rx和ry向量中
    rx.push_back(node->x * reso);
    ry.push_back(node->y * reso);
    // 在图像img上标记路径，将路径对应的图像网格标记为蓝色
    cv::rectangle(img,
        cv::Point(node->x*img_reso+1, node->y*img_reso+1),
        cv::Point((node->x+1)*img_reso, (node->y+1)*img_reso),
        cv::Scalar(255, 0, 0), -1);
        total_dis ++;
  }
  // 返回包含路径离散坐标的二维浮点数向量
  cout << "the total dis under this heuristic function is ";
  cout << total_dis << endl;
  return {rx, ry};
}
```

A* 算法是本项目的核心，在一般的路径规划问题中，我们通常会选取欧式距离来构建估价函数，但不同路况下实际情况并不相同。

本算法代码实现部分的另一个精华是数据结构的选取，我构建了一个结点类，定义了一个比较函数，从而采用了优先队列的形式，不同的数据结构对算法时间性能的影响较大。

3.2 地图与物体建模

地图的建模在相当大的程度上决定了本项目的性能上限，能否准确高效地建模是项目的一大核心任务。

在导航问题中，基本的研究对象便是地图与物体。项目开始前，我自定义了网格大小以及物体大小。然后自定义图片与实际地图的比例 reso 参数，作为根据实际图片进行建模时的缩放比例。

此外，由于实际地图像素过大，导致程序处理时间过长，因此在项目中，我先将地图进行了压缩处理，压缩得到的地图像素大致在 ae2 * be2 左右。最后，在导航地图 1 中，由于道路占地图区域的面积很小，为了便于定义起始位置，我将地图左上角和右下角特殊定义为了可行区域。

建模有以下几个关键环节。

3.2.1 地图图片预处理

不同的图片有不同的视觉性质，处理方法略有不同。此处以导航地图 1 为例进行说明。在经过二值化处理和膨胀操作后，原图中的道路被处理为白色，不可走的街区被处理为黑色。之后将原图中的图像数据结构转化为易于处理的二维数组，障碍物标记为 1(0)，可选道路标记为 0(1).

同时，项目还需具有自定义起始和目标点的功能，如不采用默认点，还需要对输入的点进行简单的合法性判断，显然，若选取的点处于障碍物之内，这个点是不合法的。同时，为保证系统鲁棒性，需要在地图的四周边界处添加障碍物。

3.2.2 障碍物地图构建

在得到二值化的图像后，我们对图像中的每一个像素点进行遍历，若为黑色，则可以确定该像素点表示障碍物。所有的障碍物像素用两个向量来表示，在障碍地图构建过程中，先将地图初始化为原图尺寸 *reso 大小，之后将每个像素点初始化为 0。然后遍历障碍物向量组，将每一个向量组对应的坐标下的二维地图向量标记为 1，表示障碍物。

在此基础上，还有两个注意事项：一是结合 robot 自身的大小，确定是否为的可行点，二是根据最终的栅格地图利用 Opencv 进行可视化显示。

首先确定是否为可行点比较简单，只需要判断物体中心点到障碍物的距离

与物体的半径的大小关系即可。在可视化显示时，本项目添加了另一个缩放比例，即图像缩放比例 imgreso，可以根据该参数调控从栅格地图到可视化界面的地图比率。避免因为图像尺寸太小而导致可视化程度不强。

```
float d = std::sqrt(std::pow((ox[k] - x), 2) + std::pow((oy[k] -
    y), 2));
        if (d <= vr / reso) {
          // 如果距离小于等于虚拟半径vr/reso，则将该网格标记为障碍物
          obmap[i][j] = 1;
          // 在图像img上标记障碍物，将对应的图像网格填充为黑色
          cv::rectangle(img,
                        cv::Point(i * img_reso + 1, j * img_reso + 1),
                        cv::Point((i + 1) * img_reso, (j + 1) *
                            img_reso),
                        cv::Scalar(0, 0, 0), -1);
          break;
        }
```

3.3 可视化显示

本项目要求用可视化界面演示算法执行过程，画出扩展过的所有路径，画出最优路径，能展示 OPEN 表和 CLOSED 表的动态变化过程。动态展示可以通过控制图像 waitKey 的时间参数来调控循环更新速率来实现，逐帧显示 A* 算法更新过程。

出于直观考虑，整个图像在二值化后的地图上进行展示，黑色表示障碍物，白色表示可行区域，蓝色表示出发点，红色表示目标点，红色表示当前搜索过程中已经扩展过的结点，即 CLOSED 表中的结点，绿色表示当前搜素过程中待扩展的结点，即 OPEN 表中的结点。最后的最优路径用蓝色线画出，所需的最短搜索路径在输出框中展示。

可视化主要采用了 opencv 库，窗口用 namedWindow 创建，每个点用小方格表示，用 rectangle 函数画出，通过循环中的 waitKey 实现了动态变化过程的展示。同时，为了保证更加直观的体验 A* 算法的执行过程，项目也支持单步运行，通过修改项目源码顶端的常量“SINGLE STEP MODE”即可实现单步调试。

图 3-1　可视化样例

3.4　启发式函数改进

在导航地图 1 中，我们选取了北京的主要道路交通图，是典型的城市道路规划问题，由于实际在城市中，往往欧式几何距离最近的点不一定是最佳路径，因为倘若涉及到更加繁琐的红绿灯和转弯，会影响车辆的平均速率，进而导致所得最佳路径并非实际最优路径。

而司机们通常更喜欢转弯更少的道路，不仅可以减少因转弯而减速的次数，而且简明的道路规划还可以更加容易记忆和行驶。

为此，我改进了原本按照欧几里得距离定义下的启发函数，采用曼哈顿距离，即两点间的距离等于水平和竖直距离的变化量之和。这种改进策略在一定程度上避免了上述问题。

```
/**
 * 计算曼哈顿距离作为启发式函数的值，用于估计两个节点之间的代价。
 *
 * @param n1 第一个节点
 * @param n2 第二个节点
 * @param w 权重参数，默认为1.0
 * @return 两个节点之间的曼哈顿距离启发式函数值（代价）
 */
```

```
float calc_heuristic(Node* n1, Node* n2, float w = 1.0) {
  // 计算节点之间的水平和垂直距离
  float dx = std::abs(n1->x - n2->x);
  float dy = std::abs(n1->y - n2->y);
  // 乘以权重参数w并返回启发式函数值（代价）
  return w * (dx + dy);
}
```

在导航地图 2 中，我们只考虑 robot 不能涉入海中，对于具体的行驶路线没有特殊要求，在这种前提下，显然欧式距离最优即为实际结果最优，不需要对结果进行优化。

此外，若有类似国际象棋棋盘类型的路径规划问题，即斜方向行走的代价和水平竖直方向相当，则可以考虑用切比雪夫距离作为启发式函数进行优化，效果显然更好。

```
/**
/**
 * 计算切比雪夫距离作为启发式函数的值，用于估计两个节点之间的代价。
 *
 * @param n1 第一个节点
 * @param n2 第二个节点
 * @param w 权重参数，默认为1.0
 * @return 两个节点之间的切比雪夫距离启发式函数值（代价）
 */
float calc_heuristic(Node* n1, Node* n2, float w = 1.0) {
  // 计算节点之间的水平和垂直距离
  float dx = std::abs(n1->x - n2->x);
  float dy = std::abs(n1->y - n2->y);
  // 取水平和垂直距离的最大值，乘以权重参数w并返回启发式函数值（代价）
  return w * std::max(dx, dy);
}
```

4 结果分析

对地图进行二值化操作是首要任务，下面为地图二值化结果

图 4-1 二值化操作

膨胀操作结果

图 4-2 dilated

可见，在对图像进行二值化和膨胀操作后得到的结果可以基本翻译地图中的道路情况，通过调整膨胀 kernel 也可以对膨胀程度进行调节。

4.1 导航地图 1

下面分别展示在欧式距离和曼哈顿距离下的单步和连续运行结果以及 CLOSE 表和 OPEN 表，最终结果的动态展示。

4.1.1 Euclidean distance

欧式距离也称欧几里得距离，是最常见的距离度量，衡量的是多维空间中两个点之间的绝对距离。

运行过程中：

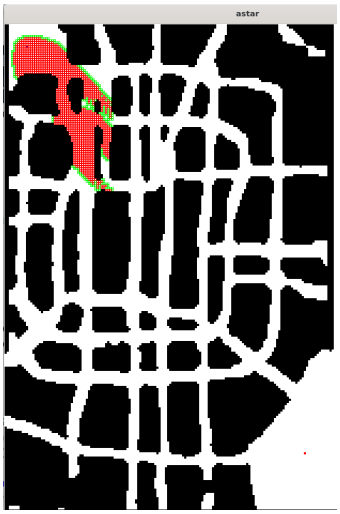

图 4-3　Euclidean distance - 1

运行结果：

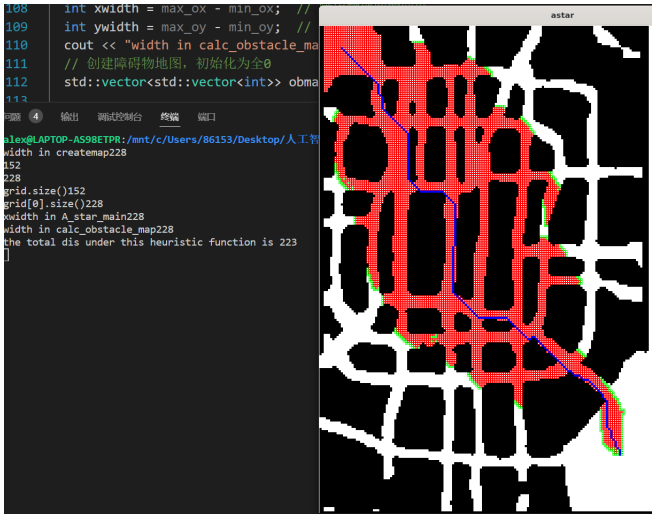

图 4-4　Euclidean distance - 2

最终总代价为 223

4.1.2 Manhattan Distance

曼哈顿距离——两点在南北方向上的距离加上在东西方向上的距离，即 d（i，j）=|xi-xj|+|yi-yj|。对于一个具有正南正北、正东正西方向规则布局的城镇街道，从一点到达另一点的距离正是在南北方向上旅行的距离加上在东西方向上旅行的距离，因此，曼哈顿距离又称为出租车距离。

运行结果：

图 4-5 Manhattan Distance

最终总代价为 288，不过需要注意的是，左上角和右下角自定义的可行区域也会影响最终代价。

虽然欧式距离在结果上要短于曼哈顿距离，但是在实际情况中，曼哈顿距离下的估价函数往往更符合实际最优。

地图规模较大，单步运行效果不明显，报告中之展示运行刚刚开始后某的单步运行结果。

通过上述样例可以看出，程序可以动态地展示 OPEN 表和 CLOSED 表的变化过程，对于最终结果求解也有明晰的可视化显示，可以良好完成既定目标。

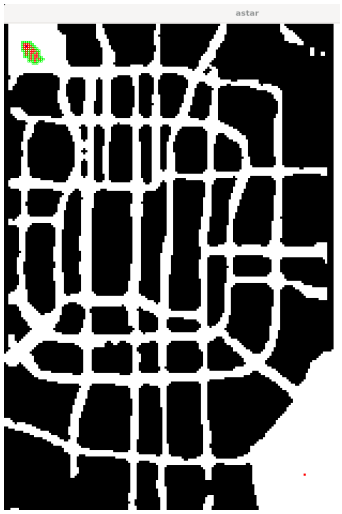

图 4-6　单步运行

4.2　导航地图 2

二值化结果

图 4-7　mask

得到的二值化图像可以基本反映地图实际的水陆情况，其中白色的表示湖面，也就是导航中需要注意躲避的障碍物。

在欧式距离下的运行结果，最终代价为 166

同样，地图规模较大，报告中只展示其中一步的单步运行结果。

图 4-8　导航地图 2

图 4-9　单步运行

5 小结与展望

整个项目做完之后收获不小，这个项目从地图的构建到 A* 算法选取最优路径，到最后启发函数的优化，综合性较强，涉及到了 opencv 的图像处理，包括图像读取，显示，到在窗口上动态表示 OPEN 表和 CLOSE 表，以及图像的二值化和膨胀等操作。定义了 Node 类，来表示每一个像素点，除了坐标信息之外还包含到达该点的代价和父亲结点。项目的核心是 A* 算法，如何利用 A* 算法对建模后的地图进行求解是项目完成性能的关键，通过合理选取估价函数，利用优先队列的数据结构，通过对到达该点的代价进行排序实现了对最优路径的迭代。最后利用父亲结点对整个地图进行了回溯。

整个项目按照要求完成了预定的所有功能，但是仍存在一些不可忽视的不足。譬如整个地图的建模过分依赖于图像的性质。无法对任意形式出现的地图进行建模，在图像分辨率达到较大规模时候往往出现超负荷现象。对于复杂道路系统倘若借助图像进行建模是肯定有很大局限性的。此外，限于期末周时间限制，我预想中想要制作一个可以利用鼠标控制的 UI 界面未能实现，指定起始点和目标点需要利用坐标。可交互性不强，鲁棒性不足，泛化性有待提高，是本项目的不足之处。

通过本次项目，我学到了很多人工智能邻域实操方面的知识，对于 A* 算法及其启发式函数有了更深的理解，在项目中遇到的很多意想不到的问题也锻炼了我处理突发情况的能力。对于地图建模这个问题，我之前完全没有了解，在充分思考后选择了目前这种方法，也是对我组织架构能力的一次锻炼。

感谢有这样一次机会，让我能够对人工智能领域的经典问题有一次深入的实践。本次项目也激发了我对导航和机器人问题的思考，希望以后可以对该类问题能有更进一步的认识。争取能够自己做出可以在实际中应用的程序。路漫漫其修远兮，吾将上下而求索。

6 附录

```
/**
 * 地图1 ： 228 * 152
 * 地图2 : 118 * 96
 * 默认模式为自动 + 地图1,
    切换注释可切换梯度, SINGLE_STEP_MODE置为1可切换单步运行
*/
#include<iostream>
#include<cmath>
#include<limits>
#include<queue>
#include<vector>
#include<opencv2/opencv.hpp>
#include<opencv2/core/core.hpp>
#include<opencv2/highgui/highgui.hpp>
#define SINGLE_STEP_MODE 1 //一个字符迭代一次

using namespace std;



class Node{
public:
  int x;
  int y;
  float sum_cost;
  Node* p_node;

  Node(int x_, int y_, float sum_cost_=0, Node*
      p_node_=NULL):x(x_), y(y_), sum_cost(sum_cost_),
      p_node(p_node_){};
};

// 定义栅格大小和分辨率
const int gridWidth = 200; // 栅格宽度
```

```
const int gridHeight = 200; // 栅格高度

// 定义栅格数据结构
typedef std::vector<std::vector<int>> GridData;

// 将图像转换为栅格数据结构
GridData convertImageToGrid(const cv::Mat& image) {
    GridData grid(image.rows, std::vector<int>(image.cols));

    for (int i = 0; i < image.rows; ++i) {
        for (int j = 0; j < image.cols; ++j) {
            // 获取像素值
            int pixelValue = image.at<uchar>(i, j);
            // 判断像素是否为黑色（障碍物）
            if(pixelValue >= 127) grid[i][j] = 0;
            else grid[i][j] = 1;
            // if(pixelValue >= 127) grid[i][j] = 1;
            // else grid[i][j] = 0;
        }
    }

    return grid;
}
std::vector<std::vector<int>> create_map_from_grid(const
    std::vector<std::vector<int>>& grid, float reso, cv::Mat& img,
    int img_reso) {
  // 获取栅格地图的宽度和高度
  int width = grid[0].size();
  int height = grid.size();
  //cout << "width in createmap" << width<<endl;
  // 创建地图，初始化为全0
  std::vector<std::vector<int>> map(height, std::vector<int>(width,
      0));

  // 遍历栅格地图的每个网格
```

```
  for (int i = 0; i < height; i++) {
    for (int j = 0; j < width; j++) {
      // 如果栅格值为1，则表示该网格为障碍物
      if (grid[i][j] == 1) {
        map[i][j] = 1;
        // 在图像img上标记障碍物，将对应的图像网格填充为黑色
        cv::rectangle(img,
                      cv::Point(j * img_reso + 1, i * img_reso + 1),
                      cv::Point((j + 1) * img_reso, (i + 1) *
                          img_reso),
                      cv::Scalar(0, 0, 0), -1);
      }
    }
  }

  // 返回创建的地图
  return map;
}
std::vector<std::vector<float> > calc_final_path(Node * goal,
    float reso, cv::Mat& img, float img_reso){
  std::vector<float> rx;
  std::vector<float> ry;// 存储路径的x ，y坐标
  Node* node = goal;
  int total_dis = 0;
  // 从目标节点开始回溯路径，直到达到起始节点（起始节点的父节点为NULL）
  while (node->p_node != NULL){
    node = node->p_node;
    // 将离散化的路径坐标（乘以分辨率reso）添加到rx和ry向量中
    rx.push_back(node->x * reso);
    ry.push_back(node->y * reso);
    // 在图像img上标记路径，将路径对应的图像网格标记为蓝色
    cv::rectangle(img,
        cv::Point(node->x*img_reso+1, node->y*img_reso+1),
        cv::Point((node->x+1)*img_reso, (node->y+1)*img_reso),
        cv::Scalar(255, 0, 0), -1);
```

```
        total_dis ++;
    }
    // 返回包含路径离散坐标的二维浮点数向量
    cout << "the total dis under this heuristic function is ";
    cout << total_dis << endl;
    return {rx, ry};

}



std::vector<std::vector<int>> calc_obstacle_map(
    std::vector<int> ox, std::vector<int> oy,
    const int min_ox, const int max_ox,
    const int min_oy, const int max_oy,
    float reso, float vr,
    cv::Mat& img, int img_reso) {

    int xwidth = max_ox - min_ox; // 障碍物地图的宽度
    int ywidth = max_oy - min_oy; // 障碍物地图的高度
    //cout << "width in calc_obstacle_map" << xwidth <<endl;
    // 创建障碍物地图，初始化为全0
    std::vector<std::vector<int>> obmap(ywidth,
        std::vector<int>(xwidth, 0));

    //遍历障碍物地图的每个网格
    for (int i = 0; i < xwidth; i++) {
        int x = i + min_ox; // 当前网格的x坐标
        for (int j = 0; j < ywidth; j++) {
            int y = j + min_oy; // 当前网格的y坐标
            for (int k = 0; k < ox.size(); k++) {
                // 计算当前网格与障碍物之间的距离
                float d = std::sqrt(std::pow((ox[k] - x), 2) +
                    std::pow((oy[k] - y), 2));
                if (d <= vr / reso) {
                    // 如果距离小于等于虚拟半径vr/reso，则将该网格标记为障碍物
```

```
            obmap[i][j] = 1;
            // 在图像img上标记障碍物，将对应的图像网格填充为黑色
            cv::rectangle(img,
                          cv::Point(i * img_reso + 1, j * img_reso + 1),
                          cv::Point((i + 1) * img_reso, (j + 1) *
                              img_reso),
                          cv::Scalar(0, 0, 0), -1);
            break;
          }
        }
      }
    }

    // 返回障碍物地图
    return obmap;
}

/**
 *检验点是否合法
*/
bool verify_node(Node* node,
                 const vector<vector<int>>& obmap,
                 int min_ox, int max_ox,
                 int min_oy, int max_oy){
    if (node->x < min_ox || node->y < min_oy || node->x >= max_ox ||
        node->y >= max_oy){
      return false;
    }

    if (obmap[node->x-min_ox][node->y-min_oy]) return false;

    return true;
}
```

```
/**
 * 计算启发式函数（Heuristic
    function）的值，用于估计两个节点之间的代价（距离）。
 *
 * @param n1 第一个节点
 * @param n2 第二个节点
 * @param w 权重参数，默认为1.0
 * @return 两个节点之间的启发式函数值（代价）
 */

float calc_heuristic(Node* n1, Node* n2, float w = 1.0) {
  // 根据节点的坐标计算欧几里得距离，并乘以权重参数w
  return w * std::sqrt(std::pow(n1->x - n2->x, 2) + std::pow(n1->y
      - n2->y, 2));
}

// /**
// * 计算曼哈顿距离作为启发式函数的值，用于估计两个节点之间的代价。
// *
// * @param n1 第一个节点
// * @param n2 第二个节点
// * @param w 权重参数，默认为1.0
// * @return 两个节点之间的曼哈顿距离启发式函数值（代价）
// */
// float calc_heuristic(Node* n1, Node* n2, float w = 1.0) {
//   // 计算节点之间的水平和垂直距离
//   float dx = std::abs(n1->x - n2->x);
//   float dy = std::abs(n1->y - n2->y);
//   // 乘以权重参数w并返回启发式函数值（代价）
//   return w * (dx + dy);
// }

// /**
// * 计算切比雪夫距离作为启发式函数的值，用于估计两个节点之间的代价。
// *
```

```
// * @param n1 第一个节点
// * @param n2 第二个节点
// * @param w 权重参数，默认为1.0
// * @return 两个节点之间的切比雪夫距离启发式函数值（代价）
// */
// float calc_heuristic(Node* n1, Node* n2, float w = 1.0) {
//  // 计算节点之间的水平和垂直距离
//  float dx = std::abs(n1->x - n2->x);
//  float dy = std::abs(n1->y - n2->y);
//  //
    取水平和垂直距离的最大值，乘以权重参数w并返回启发式函数值（代价）
//  return w * std::max(dx, dy);
// }
//定义了机器人在规划过程中可以移动的方向和代价
std::vector<Node> get_motion_model(){
  return {Node(1, 0, 1),
          Node(0, 1, 1),
          Node(-1, 0, 1),
          Node(0, -1, 1),
          Node(-1, -1, std::sqrt(2)),
          Node(-1, 1, std::sqrt(2)),
          Node(1, -1, std::sqrt(2)),
          Node(1,  1, std::sqrt(2))};
}

void a_star_planning(float sx, float sy,
                     float gx, float gy,
                     vector<float> ox_, vector<float> oy_,
                     float reso, float rr)
{
    //创建起点和终点
  Node* nstart = new Node((int)std::round(sx/reso),
      (int)std::round(sy/reso), 0.0);
      //reso表示分辨率，即最小的方格大小；sx，sy表示出发点的横纵坐标
  Node* ngoal = new Node((int)std::round(gx/reso),
```

```
    (int)std::round(gy/reso), 0.0);

// 将障碍物坐标转换为整数，并计算最小和最大的 x、y 坐标值
vector<int> ox;
vector<int> oy;

int min_ox = std::numeric_limits<int>::max();
int max_ox = std::numeric_limits<int>::min();
int min_oy = std::numeric_limits<int>::max();
int max_oy = std::numeric_limits<int>::min();


for(float iox:ox_){
    int map_x = (int)std::round(iox*1.0/reso);
    ox.push_back(map_x);
    min_ox = std::min(map_x, min_ox);
    max_ox = std::max(map_x, max_ox);
}

for(float ioy:oy_){
    int map_y = (int)std::round(ioy*1.0/reso);
    oy.push_back(map_y);
    min_oy = std::min(map_y, min_oy);
    max_oy = std::max(map_y, max_oy);
}
// 计算地图的宽度和高度
int xwidth = max_ox-min_ox;
int ywidth = max_oy-min_oy;
//cout << "xwidth in A_star_main" << xwidth << endl;
//visualization 创建用于可视化的背景图像
cv::namedWindow("astar", cv::WINDOW_NORMAL);
int count = 0;
int img_reso = 4;
cv::Mat bg(img_reso*ywidth,
          img_reso*ywidth,
```

```
            CV_8UC3,
            cv::Scalar(255,255,255));
    // 在背景图上绘制起点和终点的矩形
    cv::rectangle(bg,
                cv::Point(nstart->x*img_reso+1,
                    nstart->y*img_reso+1),
                cv::Point((nstart->x+1)*img_reso,
                    (nstart->y+1)*img_reso),
                cv::Scalar(255, 0, 0), -1);
    cv::rectangle(bg,
                cv::Point(ngoal->x*img_reso+1, ngoal->y*img_reso+1),
                cv::Point((ngoal->x+1)*img_reso,
                    (ngoal->y+1)*img_reso),
                cv::Scalar(0, 0, 255), -1);
// 创建访问地图和路径代价地图
std::vector<std::vector<int> > visit_map(xwidth,
    vector<int>(ywidth, 0));

std::vector<std::vector<float> > path_cost(xwidth,
    vector<float>(ywidth, std::numeric_limits<float>::max()));

path_cost[nstart->x][nstart->y] = 0;
// 计算障碍物地图，并在背景图上绘制障碍物
std::vector<std::vector<int> > obmap = calc_obstacle_map(
                                        ox, oy,
                                        min_ox, max_ox,
                                        min_oy, max_oy,
                                        reso, rr,
                                        bg, img_reso);

auto cmp = [](const Node* left, const Node* right){return
    left->sum_cost > right->sum_cost;};
std::priority_queue<Node*, std::vector<Node*>, decltype(cmp)>
    pq(cmp);
// 将起点加入优先队列
```

```
pq.push(nstart);
std::vector<Node> motion = get_motion_model();
// A*算法的主循环
while (true)
{
  if(SINGLE_STEP_MODE)
  {
    char c = getchar();
    if(c == EOF) exit(0);
  }
  else {};
  Node * node = pq.top();
  // 如果节点已访问过，则忽略
  if (visit_map[node->x-min_ox][node->y-min_oy] == 1){
    pq.pop();
    delete node;
    continue;
  }
    else
  {
    pq.pop();
    visit_map[node->x-min_ox][node->y-min_oy] = 1;
  }
  // 如果当前节点是目标节点，则记录最终代价并结束循环
  if (node->x == ngoal->x && node->y==ngoal->y){
    ngoal->sum_cost = node->sum_cost;
    ngoal->p_node = node;
    break;
  }
  // 遍历所有可能的运动模式
  for(int i=0; i<motion.size(); i++){
    Node * new_node = new Node(
      node->x + motion[i].x,
      node->y + motion[i].y,
      path_cost[node->x][node->y] + motion[i].sum_cost +
```

```
            calc_heuristic(ngoal, node),
        node);
    // 检查新节点是否合法
    if (!verify_node(new_node, obmap, min_ox, max_ox, min_oy,
        max_oy)){
      delete new_node;
      continue;
    }
    // 检查新节点是否已访问过
    if (visit_map[new_node->x-min_ox][new_node->y-min_oy]){
      delete new_node;
      continue;
    }
    else{

    }
    // 在背景图上绘制新节点
    cv::rectangle(bg,
                  cv::Point(new_node->x*img_reso+1,
                      new_node->y*img_reso+1),
                  cv::Point((new_node->x+1)*img_reso,
                      (new_node->y+1)*img_reso),
                  cv::Scalar(0, 255, 0));
    count++;
    cv::imshow("astar", bg);
    cv::waitKey(1);
    // 更新路径代价并将新节点加入优先队列
    if (path_cost[node->x][node->y] + motion[i].sum_cost <
        path_cost[new_node->x][new_node->y]){
      path_cost[new_node->x][new_node->y] =
          path_cost[node->x][node->y] + motion[i].sum_cost;
      pq.push(new_node);
    }
  }
    cv::rectangle(bg,
```

```
                cv::Point(node->x*img_reso+1, node->y*img_reso+1),
                cv::Point((node->x+1)*img_reso,
                    (node->y+1)*img_reso),
                cv::Scalar(0, 0, 255));
                //cout << "CLOSE : " << node -> x << " " << node
                    -> y << endl;
  }
  // 计算最终路径并绘制到背景图上
  calc_final_path(ngoal, reso, bg, img_reso);
  delete ngoal;
  delete nstart;
  cv::imshow("astar", bg);
  cv::waitKey(0);
};

int main(){
  // float sx = 10.0;
  // float sy = 10.0;
  // float gx = 80.0;
  // float gy = 170.0;
  float sx = 10.0;
  float sy = 10.0;
  float gx = 140.0;
  float gy = 210.0;
    cv::Mat img = cv::imread("..//..//pic//bj.jpg");
    // cv::Mat img = cv::imread("/..//pic//map.jpg");

// 筛选色域范围
    cv::Scalar lower_blue = cv::Scalar(80, 160, 130);
    cv::Scalar upper_blue = cv::Scalar(120, 210, 150);
    // cv::Scalar lower_blue = cv::Scalar(220, 200, 130);
    // cv::Scalar upper_blue = cv::Scalar(255, 230, 180);
    // imshow("src_img", img);
    // cv::waitKey(0);
    // 蓝色掩罩
```

```
  cv::Mat blue_mask;
  cv::inRange(img, lower_blue, upper_blue, blue_mask);
  cv::imshow("mask", blue_mask);
  cv::waitKey(0);

  //膨胀
  cv::Mat kernel_dilate =
      cv::getStructuringElement(cv::MORPH_RECT, cv::Size(4, 4));
  cv::Mat dilated_mask;
  cv::dilate(blue_mask, dilated_mask, kernel_dilate);
  cv::imshow("dilated_mask",dilated_mask);
  cv::waitKey(0);
if (img.empty()) {
    std::cout << "Failed to read the image." << std::endl;
}
  // 将图像转换为栅格数据结构
// GridData grid = convertImageToGrid(blue_mask);
GridData grid = convertImageToGrid(dilated_mask);

float grid_size = 1.0;
float robot_size = 1.0;

vector<float> ox;
vector<float> oy;

// 假设grid为二维数组表示的栅格地图
float reso = 1.0, img_reso = 1.0;
std::vector<std::vector<int>> map = create_map_from_grid(grid,
    reso, img, img_reso);



while(grid[sx][sy])
{
  cout << "Illegal start point!" << "input again with space :" <<
```

```
      endl;
  cin >> sx >> sy;
}
while(grid[gx][gy])
{
  cout << "Illegal goal point!" << "input again with space :" <<
      endl;
  cin >> gx >> gy;
}

int map_x = map.size();
int map_y = map[0].size();
cout << map_x << endl;
  cout << map_y << endl;
for(float i=0; i<map_y; i++){
  ox.push_back(i);
  oy.push_back(1.0*map_y);
}
for(float i=0; i<map_x; i++){
  ox.push_back(1.0*map_x);
  oy.push_back(i);
}
for(float i=0; i<map_y + 1; i++){
  ox.push_back(i);
  oy.push_back(map_y*1.0);
}
for(float i=0; i<map_x+1; i++){
  ox.push_back(0.0);
  oy.push_back(i);
}
cout << "grid.size() : " << grid.size() << endl;
cout << "grid[0].size() : " << grid[0].size() << endl;
for (int i = 0; i < grid.size(); ++i) {
     for (int j = 0; j < grid[i].size(); ++j) {
         if(grid[i][j] == 1)
```

```
            {
              ox.push_back((float)i);
              oy.push_back((float)j);
            }
        }
    }
  a_star_planning(sx, sy, gx, gy, ox, oy, grid_size, robot_size);
  return 0;
}
```