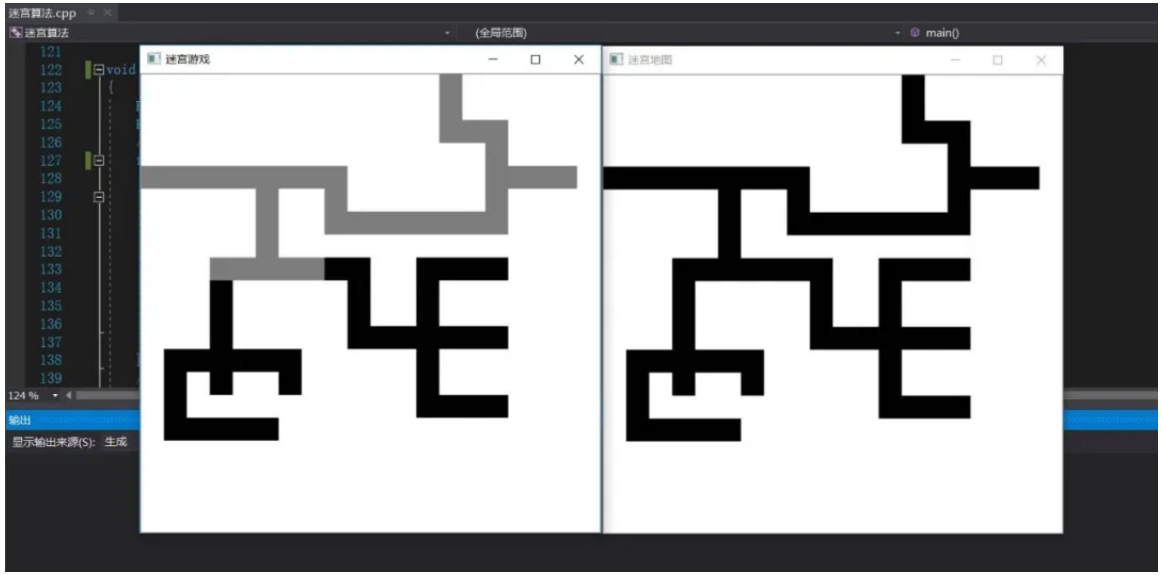
# 【手撕算法】opencv實現走迷宮算法

原創 是周旋哎 Opencv視覺實踐 今天

收錄於話題
#手撕算法
1個 >

點擊上方"**藍色小字**"關注我呀




好久沒更新了！我又帶著乾貨回來了。

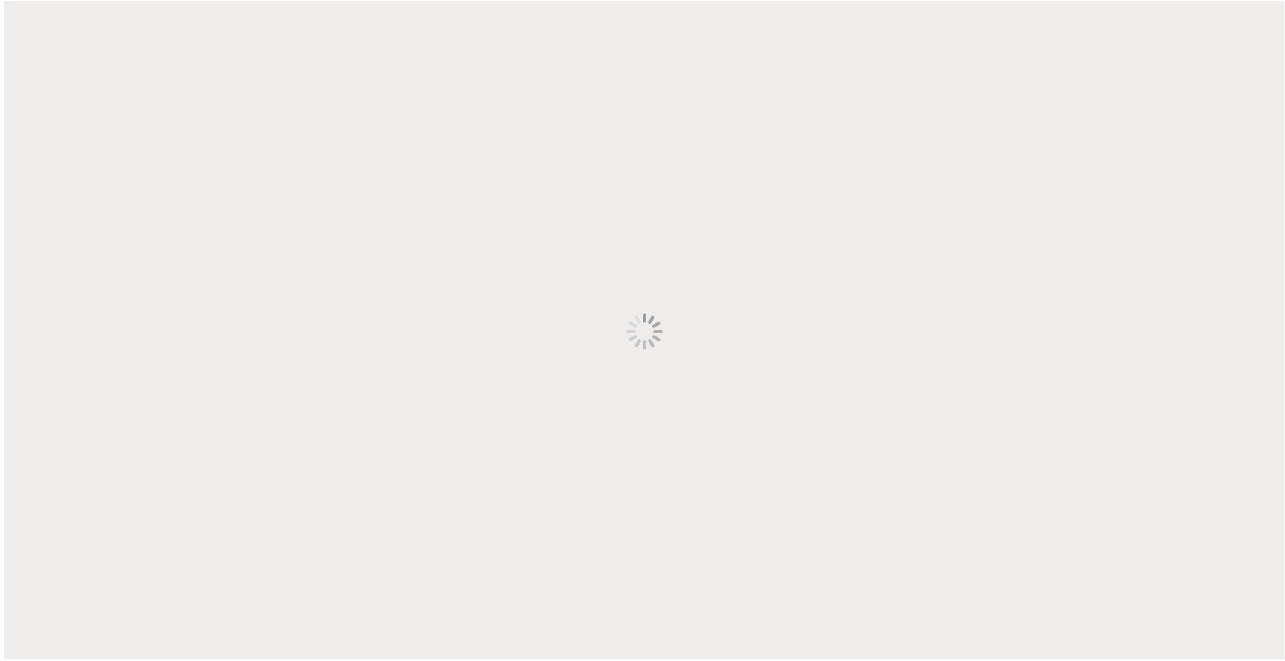
此外建了一個qq群：222954293，既方便大家一起交流學習，還可以傳一些程序文件，歡迎大家加入交流。


本文利用opencv實現了深度優先搜索DFS和廣度優先搜索BFS兩個算法來走迷宮，迷宮也是用opencv+鼠標畫的。

## 繪製迷宮

首先是繪製一個迷宮了，直接網上找一個迷宮圖然後opencv二值化處理一下也可以。

我是利用鼠標回調函數自己畫的，更簡潔明了一些。在畫迷宮時，我們鼠標點擊左鍵，則在點擊位置放置一塊牆（白色），點擊右鍵，則放一塊路（黑色），點擊中鍵，則放置一塊灰色的路，代表已經走過。具體效果如下動圖：



需要理解的是，迷宮（大小500*500）是由一塊一塊的磚（25*25）構建的，每一塊磚都由其中心點來表示，算法搜索也是一塊一塊的搜索，而不是一個像素一個像素的搜索（因為以像素為基本單位太小了，不直觀）。

**具體代碼：**

```
#define WINDOW_1   "迷宫地图" //显示绘制的迷宫地图
#define WINDOW_2   "迷宫游戏" //显示走迷宫的过程
#define show_speed  45 //显示速度（每走一步间歇时长）
#define wall_color 255 //迷宫墙的颜色 白色
#define back_color  1  //迷宫背景色，也就是路的颜色 黑色
#define step_color 125 //走迷宫的路径显示色 灰色
#define map_width 500 //迷宫大小
```

首先是一些宏定義，採用宏定義不僅更改方便，而且方便大家對後面的程序進行理解。宏定義的內容見註釋，包括迷宮各組成部分的顏色還有大小。

```
//绘制迷宫部分
bool g_bDrawingBox = false;  //绘制标识符
int step = 25; //走迷宫的步长（25*25为基本单位，一块一块的走）
Mat wallImage = Mat(step, step, CV_8UC1, wall_color);
Mat backImage = Mat(step, step, CV_8UC1, back_color);
Mat stepImage = Mat(step, step, CV_8UC1, step_color);
Mat Maze_map;
void on_MouseHandle(int event, int x, int y, int flags, void* param);
void fill_map(Mat roiImage_, Mat wallImage_);
void clear_map(Mat roiImage_, Mat backImage_);
```

**然後是繪製迷宮部分的變量聲明以及函數聲明。**

其中

```
int step = 25; //走迷宫的步长（25*25为基本单位，一块一块的走）
```

代表步長，迷宮長寬均500，每一個搭建迷宮的磚是25*25大小的，在走迷宮時也是按25*25的步長進行分析的。

```
Mat wallImage = Mat(step, step, CV_8UC1, wall_color);
Mat backImage = Mat(step, step, CV_8UC1, back_color);
Mat stepImage = Mat(step, step, CV_8UC1, step_color);
```

這三個分別代表“磚”，均是25*25大小，wallImage是牆磚，為白色，backImage是路磚，為黑色，stepImage代表走過的路，是灰色。

**绘制迷宫的具体主程序：**

```
    //【1】绘制迷宫部分
```

```
  Mat srcImage, dstImage;
  srcImage = Mat(map_width, map_width, CV_8UC1,255);//绘制画布
  imshow("原底色", srcImage);

  namedWindow(WINDOW_1);   //定义一个窗口
  setMouseCallback(WINDOW_1, on_MouseHandle, (void*)&srcImage);//对该窗口进行鼠标检测

  while (1) {
    srcImage.copyTo(dstImage);  //不断的用读取的图片更新临时图片tempImage
    imshow(WINDOW_1, dstImage);  //展示tempImage
    if (waitKey(10) == 27) break;//当按下Esc时程序结束
  }
  imwrite("迷宫图.jpg", dstImage);
  waitKey();
```

主要就是利用了鼠标回调函数，再看一下鼠标回调函数：

```
void on_MouseHandle(int event, int x, int y, int flags, void* param)
{
  Mat& image = *(Mat*)param; //得到要处理的图像
  switch (event) {          //检查鼠标事件
    case EVENT_LBUTTONDOWN: {  //检测到鼠标左键按下
      int x_index = (x / step)*step;
      int y_index = (y / step)*step;
      fill_map(image(Rect(x_index, y_index, step, step)), wallImage);//绘制墙
    }
    break;
    case EVENT_RBUTTONDOWN: {  //检测到鼠标右键按下
```

```
        int x_index = (x / step)*step;
        int y_index = (y / step)*step;
        clear_map(image(Rect(x_index, y_index, step, step)), backImage);//绘制背景
      }
      break;
      case EVENT_MBUTTONDOWN: {  //检测到鼠标左键按下
        int x_index = (x / step)*step;
        int y_index = (y / step)*step;
        fill_map(image(Rect(x_index, y_index, step, step)), stepImage);//绘制路

      }
      break;
    }
}
```

鼠标回调函数分别检测鼠标左键，右键以及中键按下三个事件，并绘制相应的“砖”。绘制砖用到了

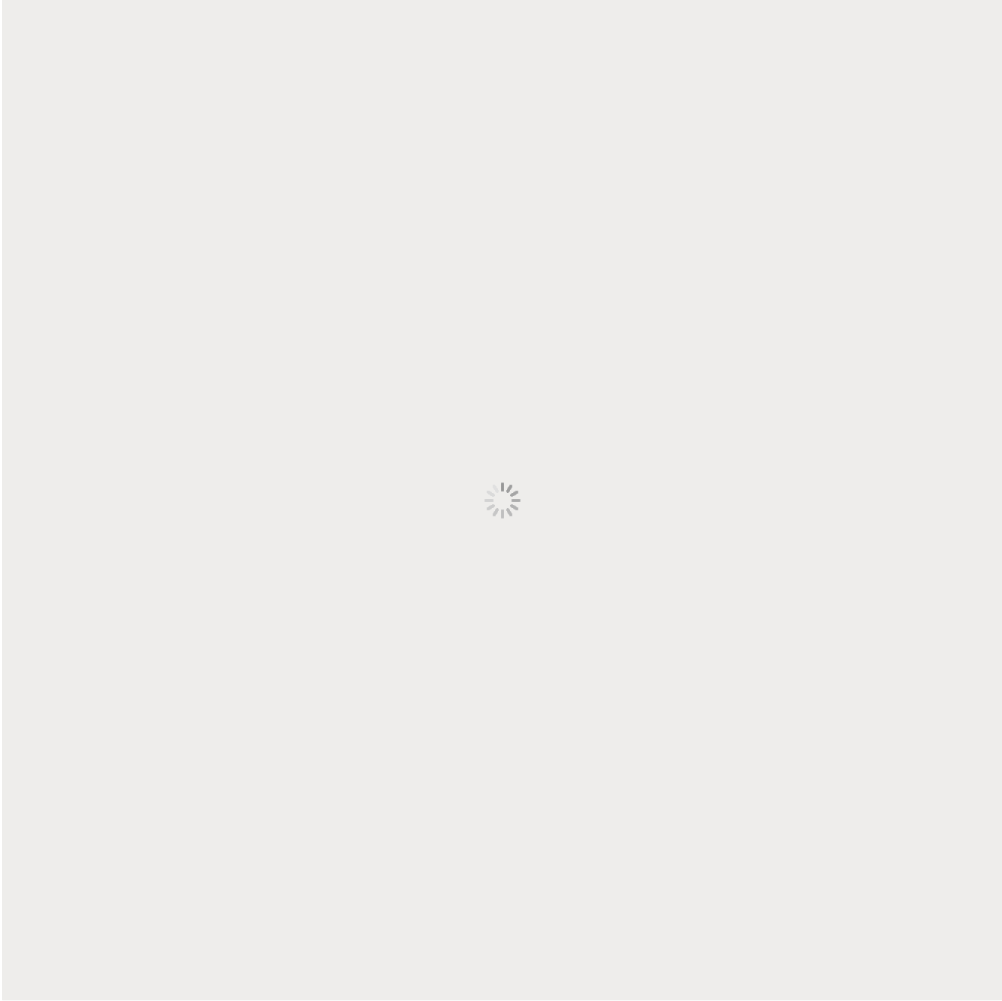
```
void fill_map(Mat roiImage_, Mat wallImage_);
void clear_map(Mat roiImage_, Mat backImage_);
```

这两个函数，其实本质就是将砖的样本图复制到迷宫地图的相应位置：

```
//填充（仅是方便理解使用，两个函数并无区别）
void fill_map(Mat roiImage_,Mat wallImage_)
{
  wallImage_.copyTo(roiImage_);
}
//清除（仅是方便理解使用，两个函数并无区别）
```

```
7  void clear_map(Mat roiImage_, Mat backImage_)
8  {
9    backImage_.copyTo(roiImage_);
10 }
```

据此，迷宫地图就可以随便绘制啦。下图为绘制好的迷宫图，上边为入口，左边为出口：



## 深度优先搜索DFS算法

算法原理仅简单介绍：

深度优先搜索，重点是深度，以迷宫为例，当一个小人一步步往前走，走到岔路口A时，可以向下或者向右，他会按照顺时针（随便定）选择，先向右走，向右走到深处，会有两种情况：

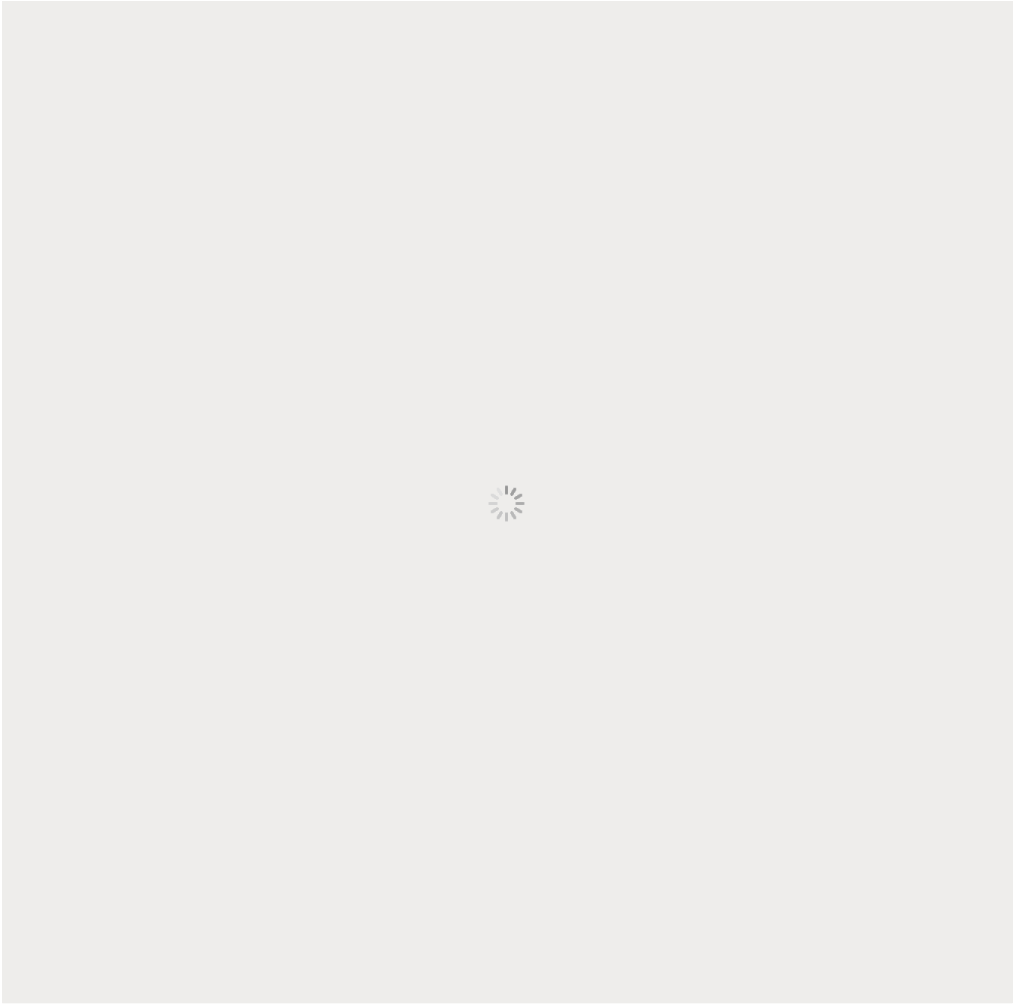
1. 深处是死胡同，这时他会退回到岔路口A，并向下走（也就是另一条未走的路）。
2. 深处又有岔路口，这时他同样按照顺时针选择一条路，继续向深处走。

以上两步是迭代的，我们可以以这两步为准则不停的迭代走下去，直到走到出口。所以程序实现深度优先搜索，可以利用迭代来做。

**首先是一些变量声明：**

```
//深度优先搜索DFS
bool Locate_Exit = false;//是否找到出口标识符
int X[4] = { 0,step,0,-step }; //增量数组，方便检查迷宫每一块上下左右四个方向的状态
int Y[4] = { -step,0,step,0 };
void Get_Maze_star();
void Maze_DFS(Point2i step_point_);
```

其中增量数组是用来方便检查每一步的上下左右的状况的，比如下图:



当走到红色的地方时，小人在走下一步时需要根据上下左右四个蓝点位置的像素值来判断能走的方向，上下均为白色的墙，不可走，右边是走过的路（为灰色）同样不可走，因此只能走前面。

**主程序：**

```
//【2】深度优先搜索DFS
Mat srcImage;
srcImage = imread("迷宫图.jpg",0);//读取灰度图
srcImage.copyTo(Maze_map);
```

```
  namedWindow(WINDOW_1);   //定义一个窗口
  namedWindow(WINDOW_2);   //定义一个窗口
  imshow(WINDOW_1, srcImage);
  imshow(WINDOW_2, Maze_map);
  Get_Maze_star();//开启DFS算法

  waitKey();
```

主程序读取迷宫图，然后开启**DFS算法。**

```
void Get_Maze_star()
{
  Point2i star_point;
  Point2i step_point;
  //【1】获取迷宫的起点
  for (int x = 0; x < map_width; x++)
  {
    if (Maze_map.at<uchar>(0, x) == back_color)
    {
      star_point = Point2i(x+12, 12);
      fill_map(Maze_map(Rect(x, 0, step, step)), stepImage);
      imshow(WINDOW_2, Maze_map);
      step_point = Point2i(star_point.x, star_point.y +step);
      break;
    }

  }
  //【2】深度优先进行搜索
```

```
    Maze_DFS(step_point);
}
```

开启DFS算法会首先对地图第一行元素进行遍历，查找第一个出现黑色像素（路为黑色，墙为白色）的坐标，就是入口。
然后将入口坐标传给

```
//【2】深度优先进行搜索
Maze_DFS(step_point);
```

函数，该函数会进行迭代搜索：

```
void Maze_DFS(Point2i step_point_)
{
  Point2i step_point;
  if (Locate_Exit == false)//还未找到出口
  {
    //【1】迭代终止条件
    if (step_point_.x < step || step_point_.y < step || map_width - step_point_.x < step || map_width - st
    {
      fill_map(Maze_map(Rect(step_point_.x - 12, step_point_.y - 12, step, step)), stepImage);
      imshow(WINDOW_2, Maze_map);
      Locate_Exit = true;
      return;
    }
    else if (Maze_map.at<uchar>(step_point_.y - step, step_point_.x) != back_color &&
      Maze_map.at<uchar>( step_point_.y, step_point_.x + step) != back_color &&
      Maze_map.at<uchar>(step_point_.y + step, step_point_.x) != back_color &&
      Maze_map.at<uchar>( step_point_.y, step_point_.x - step) != back_color)   //四面皆不可走（死胡同），返回
```

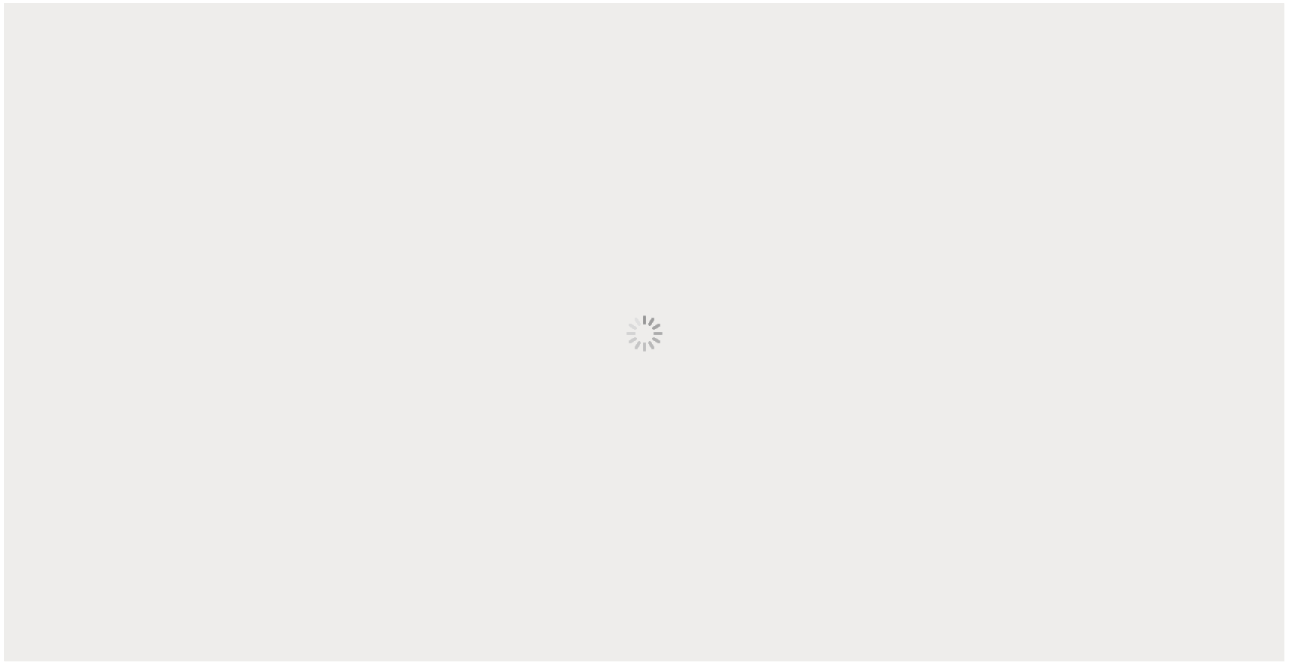
```
    {
      fill_map(Maze_map(Rect(step_point_.x - 12, step_point_.y - 12, step, step)), stepImage);
      imshow(WINDOW_2, Maze_map);
      return;
    }

    //【2】如果有路可走，顺时针判断可走的分叉并进行迭代
    for (int i = 0; i < 4; i++) {
      step_point.x = step_point_.x + X[i];
      step_point.y = step_point_.y + Y[i];
      if (Maze_map.at<uchar>(step_point.y, step_point.x) == back_color)
      {
        fill_map(Maze_map(Rect(step_point.x - 12, step_point.y - 12, step, step)), stepImage);
        imshow(WINDOW_2, Maze_map);
        waitKey(show_speed);
        Maze_DFS(step_point);
      }
    }
  }
  else
    return;
}
```

该函数有两个重要得点，首先是要确定迭代终止得条件，走迷宫得迭代终止条件是走到了终点，则直接将 Locate_Exit标识符置为true，永久退出迭代；或者走到了死胡同，则退出当前迭代，返回上一层迭代（也就是返回交叉路口，继续其他路径得探索）。

第二个点是如果没有走到死胡同或者终点，则要继续迭代，迭代就是将下一步的位置传给Maze_DFS（）函数。

**算法效果：**



## 广度优先搜索

而广度优先搜索，则重点是广度，以迷宫为例，当一个小人走到了岔路口A时，他同样可以向下或者向右走，他会将这两个选择放入到队列中，并将他们各自都走一遍，而每一条路走到新的岔路口，同样将所有岔路口加到队列中并将所有岔路口都走一遍。

当走到死胡同时，则没有可以往队列中添加的岔路口，并且也没有路可走，则当前路径探索完毕。

同时小人走过的岔路口都会从队列中删除。直到队列中没有岔路口可走或者走到了出口，则广度优先搜索算法结束。

首先是广度优先搜算算法的一些声明：

```
//广度优先算法BFS
bool inq[map_width][map_width] = { false }; //记录位置x,y是否入过队列
bool test(Point2i P);//判断当前点是否可以走
void Maze_BFS();//BFS算法
```

**主函数：**

```
//【2】广度优先算法BFS
Mat srcImage;
srcImage = imread("迷宫图.jpg", 0);
srcImage.copyTo(Maze_map);
namedWindow(WINDOW_1);   //定义一个窗口
namedWindow(WINDOW_2);   //定义一个窗口
imshow(WINDOW_1, srcImage);
imshow(WINDOW_2, Maze_map);
Maze_BFS();//开启BFS算法

waitKey();
```

**其中BFS算法：**

```
void Maze_BFS()
{
  queue<Point2i> Q;
  Point2i star_point;
  Point2i step_point;
  //【1】获取迷宫的起点
  for (int x = 0; x < 500; x++)
  {
    if (Maze_map.at<uchar>(0, x) == back_color)
    {
      star_point = Point2i(x + 12, 12);
      fill_map(Maze_map(Rect(x, 0, step, step)), stepImage);
```

```
      imshow(WINDOW_2, Maze_map);
      step_point = Point2i(star_point.x, star_point.y + step);
      break;
    }

  }
  Q.push(step_point);//将当前起点加入队列；
  //【2】进行广度搜索
  while (!Q.empty()) {
    Point2i top = Q.front();
    Q.pop();
    if (top.x < step || top.y < step || map_width - top.x < step || map_width - top.y < step)//终点的条件
    {
      fill_map(Maze_map(Rect(top.x - 12, top.y - 12, step, step)), stepImage);
      imshow(WINDOW_2, Maze_map);
      waitKey(show_speed);
      break;
    }
    for (int i = 0; i < 4; i++) {
      Point2i new_step;
      new_step.x = top.x + X[i];
      new_step.y = top.y + Y[i];
      if (test(new_step)) {
        fill_map(Maze_map(Rect(new_step.x - 12, new_step.y - 12, step, step)), stepImage);
        imshow(WINDOW_2, Maze_map);
        waitKey(show_speed);
        Q.push(new_step);
        inq[new_step.x][new_step.y] = true;
```

```
        }
      }
    }

}
```

首先还是获得迷宫入口，然后就可以开始BFS算法了。

BFS算法首先定义了一个点队列：

```
queue<Point2i> Q;
```

然后获取迷宫入口，并将入口点坐标加入到队列中。

```
//【1】获取迷宫的起点
  for (int x = 0; x < 500; x++)
  {
    if (Maze_map.at<uchar>(0, x) == back_color)
    {
      star_point = Point2i(x + 12, 12);
      fill_map(Maze_map(Rect(x, 0, step, step)), stepImage);
      imshow(WINDOW_2, Maze_map);
      step_point = Point2i(star_point.x, star_point.y + step);
      break;
    }

  }
  Q.push(step_point);//将当前起点加入队列；
```

当队列不为空时，则一直循环进行搜索：

```
//【2】进行广度搜索
while (!Q.empty()) {
  Point2i top = Q.front();
  Q.pop();
```

首先将队列首结点取出，然后判断首结点（也就是当前小人的位置）是不是满足出口条件

```
  if (top.x < step || top.y < step || map_width - top.x < step || map_width - top.y < step)//终点的条件
  {
    fill_map(Maze_map(Rect(top.x - 12, top.y - 12, step, step)), stepImage);
    imshow(WINDOW_2, Maze_map);
    waitKey(show_speed);
    break;
  }
```

如果满足出口，则跳出while循环，BFS算法结束。

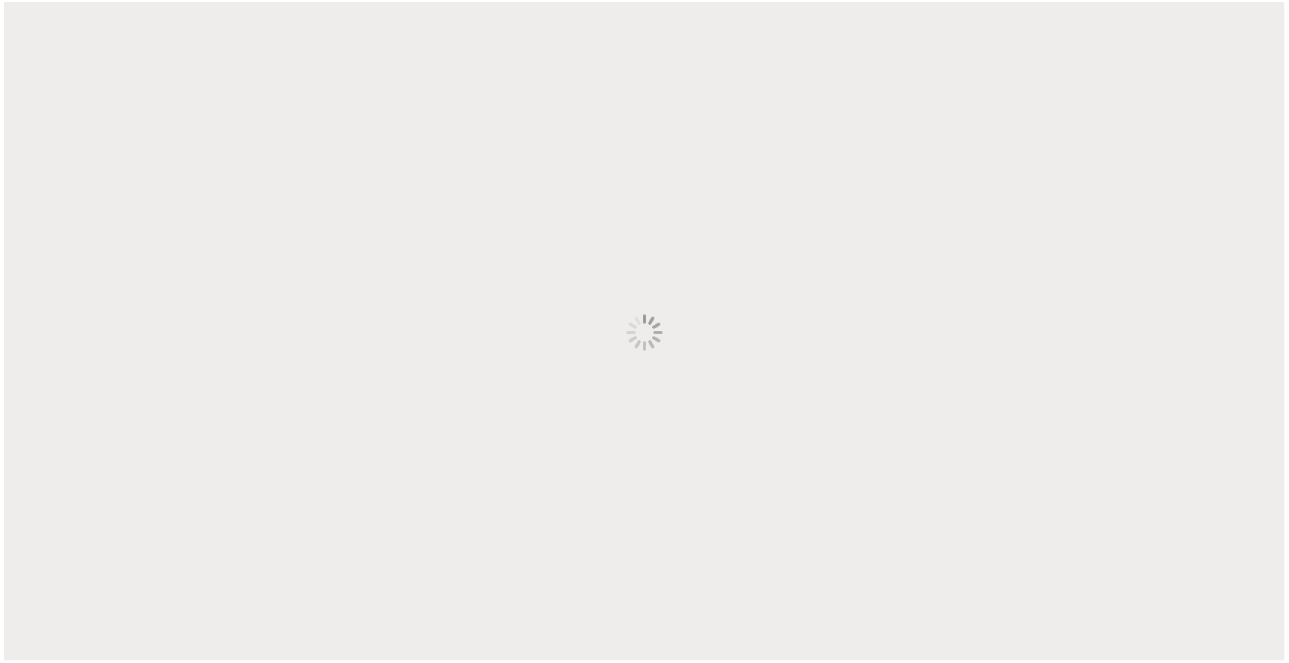
如果不满足，则继续走下一步：

```
   for (int i = 0; i < 4; i++) {
     Point2i new_step;
     new_step.x = top.x + X[i];
     new_step.y = top.y + Y[i];
     if (test(new_step)) {
       fill_map(Maze_map(Rect(new_step.x - 12, new_step.y - 12, step, step)), stepImage);
       imshow(WINDOW_2, Maze_map);
       waitKey(show_speed);
```

```
        Q.push(new_step);
        inq[new_step.x][new_step.y] = true;
      }
    }
```

利用增量数组，for循环访问当前小人位置的四个方向，如果是路可以走，则将可以走的路加入到队列。

**BFS算法运行结果**，可以与DFS算法运行结果做比较：



THE END

本文程序我就直接放到qq群里啦，算法代码这种实操文字实在是有些难以讲清楚，如果有不懂的地方，就来群里一起交流吧。