2023 光电设计大赛小车组视觉方案思路

目录

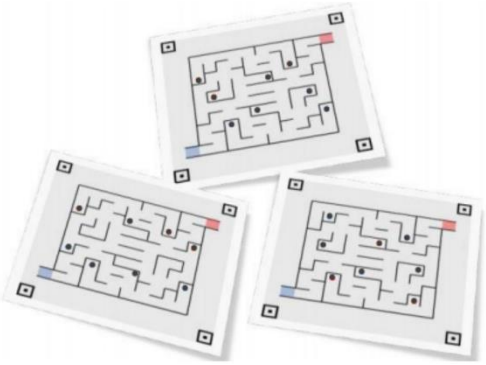
1. 方案设计

1.1 赛题分析

视觉任务部分主要分为**宝藏图识别**、**宝藏类型判断**和**路径规划**三大部分。考虑到图像处理对于单片机的**性能要求较高**，而本次比赛需处理的图像**相对简单**，在**较小分辨率**下（640*480 左右），**树莓派 4B** 即可胜任。

由于路径规划过程中，小车处于开始寻宝或者刚结束宝藏类型判断后，不需要进行宝藏图的识别和宝藏类型的判断，故图像部分的计算和路径规划设置**在时间上并不冲突**，而后续暂定的动态规划算法属于**全局搜索算法（大概就是要把各种路线“遍历一遍”）**，计算量不是很小，这里我们仍然采用**树莓派 4B** 进行路径规划算法的设计。

1.2 宝藏图识别



可以看到，官方所提供的宝藏地图有以下特点：

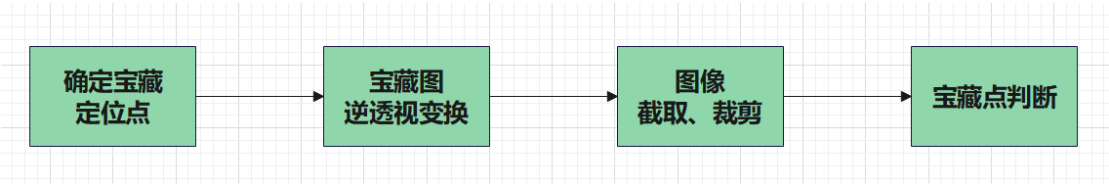
① 各宝藏图均具有四个**定位点**，定位点的形状与日常所使用的二维码的定位点的设计完全相同。

② 地图的形状，即其赛道的连通情况是固定的，这说明我们只需要预先将地图**“写入”算法**中，而不需要根据每次判断进行自动生成，进一步降低了难度。

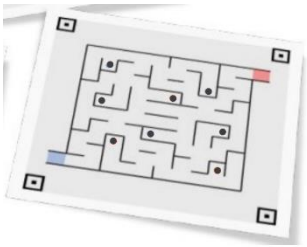
③ 宝藏（即黑点）的位置是相对于地图中心对称的，这说明我们不需要判断地图的上下颠倒情况，并且，我们可以利用这一特点**初步进行宝藏点判断的正确情况**。

④ 地图有效部分（即迷宫那部分）相对于宝藏图的大小是固定的，说明我们只需要在确定定位点坐标的情况下，可以使用固定的相对位置阈值进行图像的分割，即直接进行相对大小的图像截取，得到有效图像。（就是比如图像大小为 640*480，我们在确定定位点，并进行图像逆透视后，假设转为 500*500 像素大小，我们只需要固定截取（40,40，460,460）这部分即可得到需要得到图像部分）。

算法思路流程图



1.2.1 确定宝藏定位点代码



观察宝藏定位点形状可以发现，宝藏定位点形状为——黑白黑的重叠正方向区域，这里暂且提供两种判断思路，一种是寻找**白色连通域**作为定位点的判断特征，另一种是寻找正方形**轮廓的重叠关系**进行特征判断（**推荐这个方法**）。

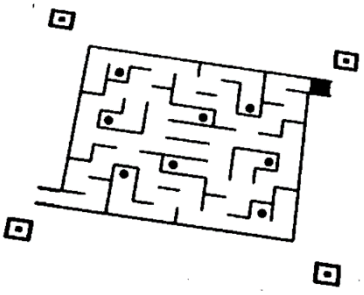
1.2.2 白色连通域方法：

主体思路为寻找到所有图像中的白色连通域，并判断**连通域面积大小**，判断面积大小**最接近的四个区域**则认为是四个定位点区域。

由于藏宝图的颜色比较单一，而红色和蓝色区域的判断对于我们的影响并不大，我们先使用 opencv 的二值化函数对图像进行**二值化处理**。

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)  #其中 img 为原宝藏图像。
ret, gray_bin = cv2.threshold(gray,0,255,cv2.THRESH_BINARY|cv2.THRESH_OTSU)
```

效果如下：



接着，我们利用 cv2.connectedComponentsWithStats 函数求得各连通域的面积，具体代码如下：

```
num_labels, labels, stats, centroids = cv2.connectedComponentsWithStats(gray2_bin, connectivity=8)
```

其中，**stats** 结果如下：

```
array([[     0,     36,    582,    419,  16743],
       [     0,      0,    718,    455, 308555],
       [   483,     41,     26,     19,    303],
       [    94,    106,     24,     19,    302],
       [   548,    327,     27,     22,    394],
```

```
       [   124,    398,     26,     23,    393]], dtype=int32)
```

其前四列分别代表的是连通区域的几何中心位置，第四列代表的是连通区域像素点数。通过一个简单的**遍历搜索**，我们找到最接近面积的四块连通区域，得到各定位点的位置：

```
distance = stats[:,-1]
record = [0,0]
result = []
i = 0
j = 0
for k in range(4):
    deta = len(gray2_bin) * len(gray2_bin[0])
    min_dis = len(gray2_bin) * len(gray2_bin[0])
    for i in range(num_labels):
        for j in range(i + 1 , num_labels):
            if i == j or distance[i] == -1 or distance[j] == -1:
                continue
            deta = abs(distance[i] - distance[j])
            if deta < min_dis:
                min_dis = deta
                record = [i,j]
    print(record)
    result.append(record[1])
    print(distance)
    distance[record[0]] = (distance[record[0]] + distance[record[1]]) / 2
    distance[record[1]] = -1
```

最后我们得到 result 的结果如下所示：

```
[3, 5, 4, 2]
```

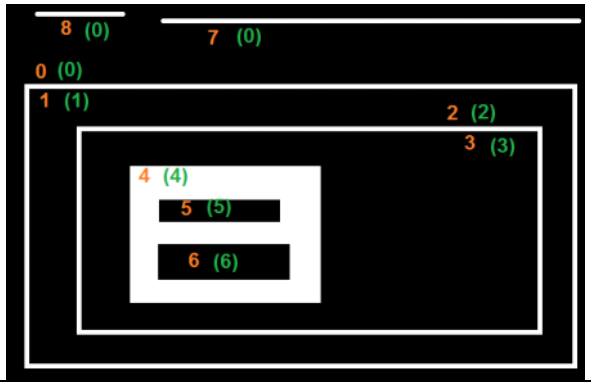
其结果表示定位点位置分别为 stats 中的第 3,5,4,2 行的数据。至此，定位点搜索完成。（**该方法在后期测试时，发现如果图像错误，会出现找到过多连通域，从而计算卡死的情况，所以建议使用下一方法。**）

1.2.3 轮廓搜索特征查找方法

Opencv 中通过使用 findContours 函数，简单几个的步骤就可以检测出物体的轮廓，轮廓获取效果如下所示：

我们可以认为，定位点的轮廓由三个共中心的方形轮廓组合而成，而当我们使用 findContours 函数，并设置输出结构为等级树结构 cv2.RETR_TREE 时（即得到结果中包含了**轮廓父子集的关系**信息），我们只要判断最具特殊性的从外向内数的**第二层轮廓**，即是否存在这么一个**存在父轮廓，且存在子轮廓，但是不存在同级的轮廓**。

PS：设置 cv2.RETR_TREE，得到的层次结构表 hierarchy 数据为四列多行结果，例如：




```
# >>> hierarchy
# array([[[ 7, -1,  1, -1],
#         [-1, -1,  2,  0],
#         [-1, -1,  3,  1],
#         [-1, -1,  4,  2],
#         [-1, -1,  5,  3],
#         [ 6, -1, -1,  4],
#         [-1,  5, -1,  4],
#         [ 8,  0, -1, -1],
#         [-1,  7, -1, -1]]])
```

第 0 行数据的结果第 1 列表示同级的下一个轮廓编号为 7，第 2 列表示，没有同级的上一个轮廓，第 3 列表示其子轮廓（即其完全包含的轮廓）编号为 1，第 4 列表示不存在父轮廓（即其完全被包含的轮廓）。从结果中，我们可以看出轮廓 0,7,8 为同一层级，编号 7,8 不存在父子轮廓，轮廓 0 存在子轮廓不存在父轮廓……

概括而言：
hierarchy[i][0]：第 i 条轮廓下一条轮廓
hierarchy[i][1]：第 i 条轮廓上一条轮廓
hierarchy[i][2]：第 i 条轮第一条子轮廓
hierarchy[i][3]：第 i 条轮第一条父轮廓

按照以上思路，我们构写代码如下：

```
# 读入图像
gray_pic = cv2.cvtColor(pic, cv2.COLOR_BGR2GRAY)  # 灰度处理图像
ret, monochrome = cv2.threshold(gray_pic, 0, 255, cv2.THRESH_BINARY | cv2.THRESH_OTSU)  # 二值化
monochrome = cv2.medianBlur(monochrome, 3) #中值滤波
_, contours, hierarchy = cv2.findContours(monochrome, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
center_index = []
center = []
image = pic.copy()
for i in range(len(contours)):
    if hierarchy[0][i][0] == -1 and hierarchy[0][i][1] == -1 and hierarchy[0][i][2] != -1 and
hierarchy[0][i][3] != -1: #独生子，无兄弟，但是有儿子，有爹
        if hierarchy[0][hierarchy[0][i][2]][2] == -1 and hierarchy[0][hierarchy[0][i][2]][3] == i
and hierarchy[0][hierarchy[0][i][3]][2] == i: #儿子是自己的，爹也是自己的
```

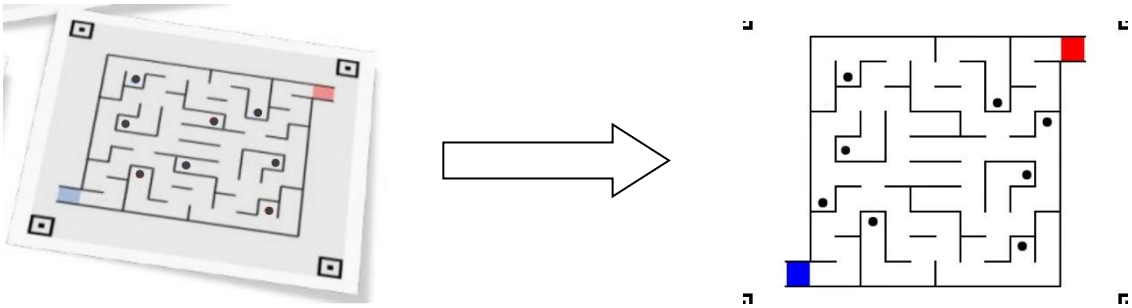
```
            epsilon = 0.05 * cv2.arcLength((contours[hierarchy[0][i][3]]), True)  # epsilon 为逼近精度，这里使用轮廓周长的 4%作为精度
            approx = cv2.approxPolyDP(contours[hierarchy[0][i][3]], epsilon, True)
            if len(approx) == 4:
                center_index.append(hierarchy[0][i][3])
```

其中 cv2.approxPolyDP 是多边形逼近函数，为了进一步确定得到的符合分层条件的轮廓是否为正方向，即多边形逼近的边数是否为 4。

函数形式为：**approxCourve= cv2.approxPolyDP(curve,epsilon,closed)** 。其中，curve 表示要逼近的轮廓点的集合，此输入常为 findContours 函数得到的轮廓 contours；epsilon 为手动设定的逼近精度大小，为指定**近似精度**的参数， 这是原始曲线和它的近似之间最大距离，需要手动调参； closed：如果为 true，则闭合近似曲线（其第一个和最后一个顶点为连接的）；否则，不闭合。

1.2.4 宝藏图逆透视

逆透视变换其意义就是通过上述所找到的四个定位点，将其线性拉伸收缩图像，使得图像变为方正的矩形图像。



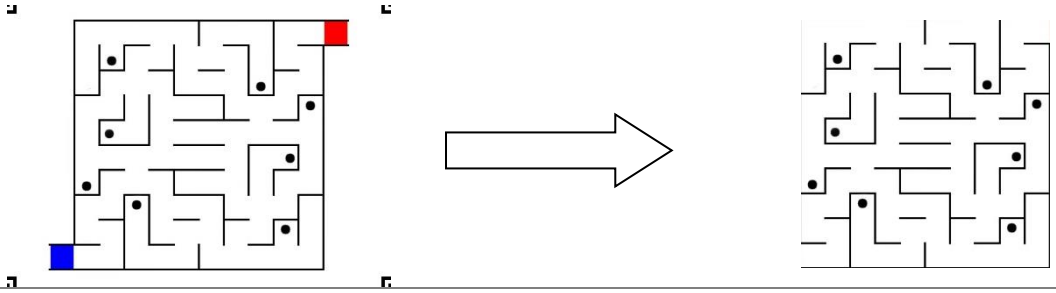
其分为两步构成：利用 trans_matrix = cv2.getPerspectiveTransform(pts1, pts2)函数生成**透视变换矩阵**，再利用 dst = cv2.warpPerspective(pic, trans_matrix, (w, h))函数得到逆透视变换后结果。

其中，pts1 参数为定位点列表（数据类型要求为 numpy 数组，float 类型）；pts2 为自定义的目标图像的四个位置点，这里可以人为定义，例如 pts2 = np.float32([[0, 0], [500, 0], [0, 500], [500, 500]])，表示我们要逆透视得到的结果图像长宽均为 500。trans_matrix 则为函数 cv2.getPerspectiveTransform 得到的变换矩阵；pic 为输入的要变换的图像数据；dst 为最终得到结果。

1.2.5 图像剪裁

由于图像定位点到我们关心的有效图像区域位置是固定的，这里我们只需要利用最简单的图像常数剪裁方式就行。

对应代码如下：



```
feature_min_w = 88
feature_max_w = 413
feature_min_h = 30
feature_max_h = 473
feature_min = 57  # 57
feature_max = 442  # 442
feature1 = feature[feature_min_h:feature_max_h, feature_min_w:feature_max_w]
```

PS：图像的本质其实就是数组，这里相当于是做了一个数组的截取。

1.2.6 宝藏点判断

将以上的图像截取为 10*10 的小图像：

```
ww_w = (feature_max_w - feature_min_w) // 10
ww_float_w = (feature_max_w - feature_min_w) / 10
ww_h = (feature_max_h - feature_min_h) // 10
ww_float_h = (feature_max_h - feature_min_h) / 10
img_divided = np.empty((100, ww_h, ww_w), dtype=float)  # 创建一个空的三维 numpy 数组

if img_divided[0].ndim != 2:
    return 0, []

for i in range(10):
    for j in range(10):
        # 先获取到 feature1 的一个切片
        feature1_slice = feature1[int(ww_float_h * i):int(ww_float_h * i) + ww_h,
int(ww_float_w * j):int(ww_float_w * j) + ww_w]
        # 然后将这个切片的大小调整为 (ww_h, ww_w)，并赋值给 img_divided[i * 10 + j]
        img_divided[i * 10 + j] = cv2.resize(feature1_slice, (ww_w, ww_h))  # 注意这里的
ww_w 和 ww_h 的顺序
```

这里就是一个简单的图像截取，与上述原理相似，只是多了一层 i、j 循环，不再赘述。

通过观察，我们可以看出，要判断该点是否存在宝藏，我们只需要看图像中心是否有“黑点”即可，即判断以下区域：

同样进行图像截取，我们找到黑点面积最多的 8 个区域，认为其存在宝藏点。

```
list_blackArea = []
ROI = 10
treasure = []
for i in range(10):
    for j in range(10):
        temp = img_divided[i * 10 + j][ROI: ww_h - ROI, ROI: ww_w - ROI]
        list_blackArea.append(np.sum(temp == 0))  # 记录 100 区域内，各个区域中像素点是黑色的个数
for i in range(8):
    temp_tre = list_blackArea.index(max(list_blackArea))
    treasure.append(temp_tre)
    list_blackArea[temp_tre] = 0
treasure = sorted(treasure)
```

这里的 ROI 即我们手动调节的关注区域度参数，这里表示我们要舍弃的周围一圈的长度，即去除地图“墙壁”信息的黑点对于我们宝藏判断的干扰。

temp_tre = list_blackArea.index(max(list_blackArea))则是找到黑点最多的 8 个区域。

1.2.7 结果验证

```
true_flag = 1
for i in range(4):
    if ((treasure[i] // 10) + (treasure[7 - i] // 10) == 9) &
((treasure[i] % 10) + (treasure[7 - i] % 10) == 9) & (
            true_flag == 1):
        true_flag = 1
    else:
        true_flag = 0
```

这里就是一个简单的验证得到的宝藏点是否中心对称的代码，结果输出在 true_flag 中，若 true_flag == 1，则表示宝藏对称，反之不对称。

1.3 宝藏类型判断

题目要求对上述“宝藏”进行拍照，很明显，其“宝藏”的区分特点主要为颜色和形状。一

般对颜色空间的图像进行有效处理都是在 HSV 空间进行的，然后对于基本色中对应的 HSV 分量需要给定一个严格的范围。那我们的总体思路就是，**将图像默认的 BGR 通道**（opencv 中的图像默认为 BGR 而不是 RGB），**转换为 HSV 通道，后根据阈值设定进行分类**。

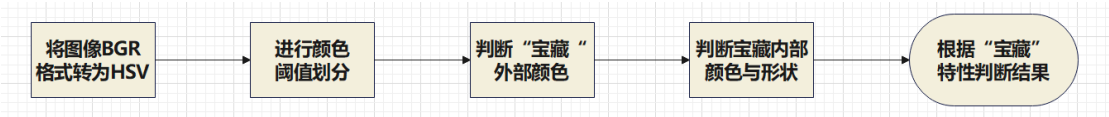
各颜色的 HSV 值可以通过参考以下表格：（CSDN 讲解链接）

	黑	灰	白	红		橙	黄	绿	青	蓝	紫
hmin	0	0	0	0	156	11	26	35	78	100	125
hmax	180	180	180	10	180	25	34；	77	99	124	155
smin	0	0	0	43		43	43	43	43	43	43
smax	255	43	30	255		255	255	255	255	255	255
vmin	0	46	221	46		46	46	46	46	46	46
vmax	46	220	255	255		255	255	255	255	255	255

可以看到，颜色主要是由参量 H 进行区分的，红色、蓝色、绿色的 H 值相差较远，而黄色的 H 值与红色、绿色相距较近。

首先要对 HSV 三个阈值有这样的概念：**区分各种颜色的区别**（除了黑色和白色），**我们利用参量 H，区分背景白色和底线黑色与各颜色的区分，我们用 S 和 V。**对应到实际，其实就是我们在实验室调好的颜色阈值，如果到实际比赛场地不适用，其实是很正常的，因为场地的光线强度各种条件不同，而需要进行调节的，其实是后两个参数 S 和 V，**千万不能病急乱投医，把参数 H 调整太多!!**（当然，前提是相机的参数，曝光什么的不存在问题）。

这里的代码和思路其实就很明白了，用以下流程图表示。



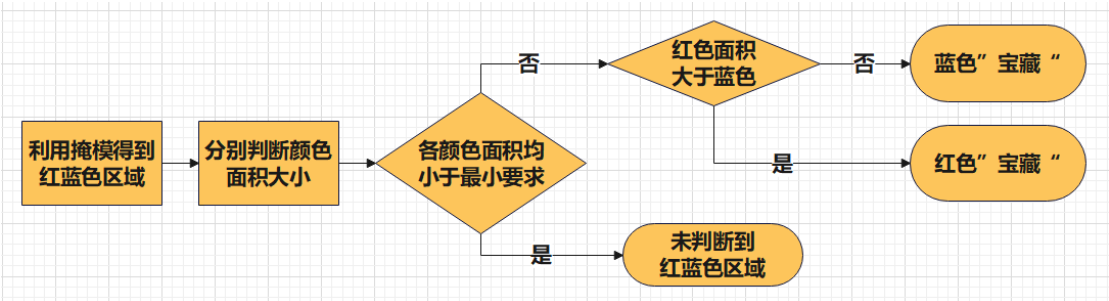
1.3.1 图像格式转换具体思路与代码讲解

图像格式转换的工作，在 OpenCV 中只需要一行代码即可搞定：

```
hsv = cv2.cvtColor(dst, cv2.COLOR_BGR2HSV)
```

将需要转换的 BGR 图像 dst 作为 `cv2.cvtColor` 函数的参数（该函数就是用来进行图像格式转换的），选择转换格式为 `cv2.COLOR_BGR2HSV`，即将 BGR 转为 HSV，我们用 `hsv` 来接收最终转换后的结果图像。

1.3.2 颜色阈值划分和外部颜色判断具体思路与代码讲解



这里我们用到了 OpenCV 图像掩模 Mask 的概念，可以理解为就是我们通过透过一张留有孔洞的“遮挡物”来观察我们感兴趣的图像部分（ROI region of interest 区域），也就是观察剔除了图像背景的图像宝藏区域。另外我们还用到了 cv2.contourArea 函数，其作用是利用 findContours 函数，在我们框选出我们的 ROI 区域后，计算该区域的内面积。

具体代码如下所示：

```
#红色识别
lower_red1 = np.array([0, 43, 80])
upper_red1 = np.array([10, 255, 255])
lower_red2 = np.array([156, 46, 80])
upper_red2 = np.array([180, 255, 255])
red_mask1 = cv2.inRange(hsv, lower_red1, upper_red1)
red_mask2 = cv2.inRange(hsv, lower_red2, upper_red2)
red_mask = cv2.bitwise_or(red_mask1, red_mask2)

contours_red, _ = cv2.findContours(red_mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
#识别红色最大轮廓
if len(contours_red) > 0:
    max_contour_red = max(contours_red, key=cv2.contourArea)
    # 计算红色轮廓包围面积
    area_red = cv2.contourArea(max_contour_red)
else:
    area_red = 0
#蓝色识别
lower_blue = np.array([90, 30, 80])
upper_blue = np.array([155, 255, 255])
blue_mask = cv2.inRange(hsv, lower_blue, upper_blue)
contours_blue, _ = cv2.findContours(blue_mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
#识别蓝色最大轮廓
if len(contours_blue) > 0:
    max_contour_blue = max(contours_blue, key=cv2.contourArea)
    #计算蓝色轮廓包围面积
    area_blue = cv2.contourArea(max_contour_blue)
else:
    area_blue = 0
# print("Red_Area = " + str(area_red))
# print("Blue_Area = " + str(area_blue))
#判断面积是否大于最小要求面积（滤波）红＞蓝则为红，红＜蓝则为蓝
if(max(area_red, area_blue) >= min_area_TH):
    if area_red > area_blue:
        Red_find = True
    elif area_red < area_blue:
        Blue_find = True
else:
    pass
    # print("Red not find")
    #面积大小不符合要求
```

1.3.3 内部颜色和形状判断具体思路与代码讲解

内部颜色判断的思路与上述相同，没有什么区别，只是颜色阈值划分，从红色和蓝色变成了黄色和绿色，其本质原理还是相同的。

内部形状判断主要就是为了判断“圆”和“三角”两种形状。由于我们不能保证小车每次拍摄宝藏的位置与宝藏之间的距离相同，我们先利用 cv2.arcLength 函数进行内部形状的周长估计，从而自适应调整拟合度，即 cv2.approxPolyDP 函数中的 epsilon 参数，以绿色区域判断为例，这部分代码如下所示：

```
perimeter = cv2.arcLength(max_contour_green, True)
epsilon = Triangle_approch * perimeter
approx = cv2.approxPolyDP(max_contour_green, epsilon, True)
```

其中，Triangle_approch 是我们需要人为调整的基本拟合度的参数。

其余的思路与上述类似，这里不多赘述，就是在颜色判断基础上加入了形状判断，必须同时满足颜色和形状要求，我们才能确定内部的元素特征。具体代码如下：

```
if Red_find or Blue_find:
    #判断绿色
    lower_green = np.array([35, 43, 46])
    upper_green = np.array([77, 255, 255])
    green_mask = cv2.inRange(hsv, lower_green, upper_green)
    contours_green, _ = cv2.findContours(green_mask,
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    #找绿色最大轮廓
    if len(contours_green) > 0:
        max_contour_green = max(contours_green, key=cv2.contourArea)
        area_green = cv2.contourArea(max_contour_green)
    else:
        area_green = 0
    #判断黄色
    lower_yellow = np.array([11, 30, 50])
    upper_yellow = np.array([34, 255, 255])
    yellow_mask = cv2.inRange(hsv, lower_yellow, upper_yellow)
    contours_yellow, _ = cv2.findContours(yellow_mask,
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    #找黄色最大轮廓
    if len(contours_yellow) > 0:
        max_contour_yellow = max(contours_yellow, key=cv2.contourArea)
        area_yellow = cv2.contourArea(max_contour_yellow)
    else:
        area_yellow = 0
    #判断内部颜色是否满足最小面积要求  （若效果不佳，这部分后面黄色圆和绿色三角可以分开判断）

    # print("Green_Area = " + str(area_green))
    # print("Yellow_Area = " + str(area_yellow))

    if(max(area_green, area_yellow) >= min_inside_area_TH):
        if area_green > area_yellow:
```

```
                Green_inside = True
            else:
                Yellow_inside = True
        if Green_inside:
            perimeter = cv2.arcLength(max_contour_green, True)
            epsilon = Triangle_approch * perimeter
            approx = cv2.approxPolyDP(max_contour_green, epsilon, True)
            if len(approx) == 3:
                Triangle_find = True
        #若内部为黄色，判断是否为圆形
        elif Yellow_inside:
            perimeter = cv2.arcLength(max_contour_yellow, True)
            epsilon = Circle_approch * perimeter
            approx = cv2.approxPolyDP(max_contour_yellow, epsilon, True)
            if len(approx) > 8:
                Circle_find = True
        if Red_find and Triangle_find:#红色宝藏且内部为三角形（红队真宝藏）
            return 3
        elif Red_find and Circle_find:#红色宝藏且内部为圆形（红队假宝藏）
            return 4
        elif Blue_find and Triangle_find:#蓝色宝藏且内部为三角形（蓝队假宝
藏）
            return 1
        elif Blue_find and Circle_find:#蓝色宝藏且内部为圆形（蓝队真宝藏）
            return 2
        else:#其他类型（判断出错，重新进行图像采集）
            return 5
```

1.4 路径规划

由于地图可以分为 10*10 的小块部分，那么这其实就是一个很简单的图论问题，每一个小块即一个节点，存在通路与否即节点间距离为 1 或者∞。

由于地图本身较小，我们可以采用动态规划 DP 的思想进行算法的编写。动态规划的主要作用，本质上是相对于暴力求解，即将每一种可能进行尝试，动态规划在此基础上具有**“记忆功能”**，即它可以对之前错误的集合进行一个**记录**，并尽早进行错误道路的排除。其会导致计算内存的增加，但是会极大地减小

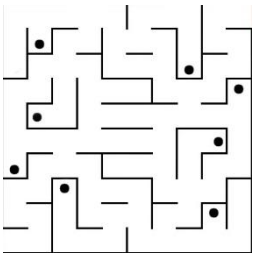
Ps:这里的话需要对**递归思想**有基础，不然很容易绕晕，但强行理解应该也不是不行，递归思想大概就是**多次调用同一个函数，直到触发退出机制，再逐级退出。**

1.4.1 图像通路信息存储

首先，我们将图像通路情况转换为代码，这里可以根据自己的习惯和需要进行转换。这里以我的代码为例。

```
# initialize
# 1.迷宫
infor = np.empty((10, 10, 4))   # 上下左右 通 1 阻 0
```

我们建立一个 10*10*4 的三维空数组，若该节点（即小块区域）在某一方向可以通行，即其对应位置记为 1，否则记为 0。



根据地图，我们对代码中地图信息进行以下初始化：

```
infor[0][0][0:4] = [0, 1, 0, 1]
infor[0][1][0:4] = [0, 1, 1, 1]
infor[0][2][0:4] = [0, 0, 1, 1]
infor[0][3][0:4] = [0, 1, 1, 1]
infor[0][4][0:4] = [0, 1, 1, 0]
infor[0][5][0:4] = [0, 1, 0, 1]
infor[0][6][0:4] = [0, 0, 1, 1]
infor[0][7][0:4] = [0, 1, 1, 0]
infor[0][8][0:4] = [0, 1, 0, 1]
infor[0][9][0:4] = [0, 0, 1, 1]

infor[1][0][0:4] = [1, 1, 0, 0]
infor[1][1][0:4] = [1, 0, 0, 0]
infor[1][2][0:4] = [0, 1, 0, 1]
infor[1][3][0:4] = [1, 0, 1, 0]
infor[1][4][0:4] = [1, 1, 0, 1]
infor[1][5][0:4] = [1, 0, 1, 1]
infor[1][6][0:4] = [0, 1, 1, 0]
infor[1][7][0:4] = [1, 1, 0, 0]
infor[1][8][0:4] = [1, 0, 0, 1]
infor[1][9][0:4] = [0, 1, 1, 0]

infor[2][0][0:4] = [1, 0, 0, 0]
infor[2][1][0:4] = [0, 1, 0, 1]
infor[2][2][0:4] = [1, 1, 1, 1]
infor[2][3][0:4] = [0, 1, 1, 0]
infor[2][4][0:4] = [1, 0, 0, 1]
infor[2][5][0:4] = [0, 0, 1, 1]
infor[2][6][0:4] = [1, 1, 1, 0]
infor[2][7][0:4] = [1, 0, 0, 0]
```

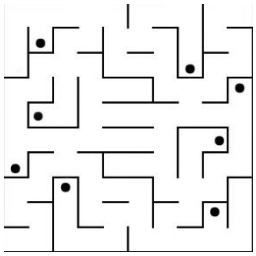
```
infor[2][8][0:4] = [0, 1, 0, 1]
infor[2][9][0:4] = [1, 0, 1, 0]

infor[3][0][0:4] = [0, 1, 0, 1]
infor[3][1][0:4] = [1, 0, 1, 0]
infor[3][2][0:4] = [1, 1, 0, 0]
infor[3][3][0:4] = [1, 1, 0, 1]
infor[3][4][0:4] = [0, 0, 1, 1]
infor[3][5][0:4] = [0, 0, 1, 0]
infor[3][6][0:4] = [1, 0, 0, 1]
infor[3][7][0:4] = [0, 1, 1, 1]
infor[3][8][0:4] = [1, 0, 1, 0]
infor[3][9][0:4] = [0, 1, 0, 0]

infor[4][0][0:4] = [1, 1, 0, 0]
infor[4][1][0:4] = [0, 0, 0, 1]
infor[4][2][0:4] = [1, 0, 1, 0]
infor[4][3][0:4] = [1, 1, 0, 1]
infor[4][4][0:4] = [0, 0, 1, 1]
infor[4][5][0:4] = [0, 0, 1, 1]
infor[4][6][0:4] = [0, 1, 1, 1]
infor[4][7][0:4] = [1, 0, 1, 1]
infor[4][8][0:4] = [0, 0, 1, 1]
infor[4][9][0:4] = [1, 1, 1, 0]

infor[5][0][0:4] = [1, 1, 0, 1]
infor[5][1][0:4] = [0, 0, 1, 1]
infor[5][2][0:4] = [0, 1, 1, 1]
infor[5][3][0:4] = [1, 0, 1, 1]
infor[5][4][0:4] = [0, 0, 1, 1]
infor[5][5][0:4] = [0, 0, 1, 1]
infor[5][6][0:4] = [1, 1, 1, 0]
infor[5][7][0:4] = [0, 1, 0, 1]
infor[5][8][0:4] = [0, 0, 1, 0]
infor[5][9][0:4] = [1, 1, 0, 0]

infor[6][0][0:4] = [1, 0, 0, 0]
infor[6][1][0:4] = [0, 1, 0, 1]
infor[6][2][0:4] = [1, 0, 1, 1]
infor[6][3][0:4] = [0, 1, 1, 0]
infor[6][4][0:4] = [0, 0, 0, 1]
infor[6][5][0:4] = [0, 0, 1, 1]
infor[6][6][0:4] = [1, 1, 1, 0]
infor[6][7][0:4] = [1, 1, 0, 0]
infor[6][8][0:4] = [0, 1, 0, 1]
infor[6][9][0:4] = [1, 0, 1, 0]

infor[7][0][0:4] = [0, 1, 0, 1]
```

```
infor[7][1][0:4] = [1, 0, 1, 0]
infor[7][2][0:4] = [0, 1, 0, 0]
infor[7][3][0:4] = [1, 1, 0, 1]
infor[7][4][0:4] = [0, 0, 1, 1]
infor[7][5][0:4] = [0, 1, 1, 0]
infor[7][6][0:4] = [1, 0, 0, 1]
infor[7][7][0:4] = [1, 1, 1, 1]
infor[7][8][0:4] = [1, 0, 1, 0]
infor[7][9][0:4] = [0, 1, 0, 0]

infor[8][0][0:4] = [1, 0, 0, 1]
infor[8][1][0:4] = [0, 1, 1, 0]
infor[8][2][0:4] = [1, 1, 0, 0]
infor[8][3][0:4] = [1, 0, 0, 1]
infor[8][4][0:4] = [0, 1, 1, 1]
infor[8][5][0:4] = [1, 1, 1, 0]
infor[8][6][0:4] = [0, 1, 0, 1]
infor[8][7][0:4] = [1, 0, 1, 0]
infor[8][8][0:4] = [0, 1, 0, 0]
infor[8][9][0:4] = [1, 1, 0, 0]

infor[9][0][0:4] = [0, 0, 1, 1]
infor[9][1][0:4] = [1, 0, 1, 0]
infor[9][2][0:4] = [1, 0, 0, 1]
infor[9][3][0:4] = [0, 0, 1, 1]
infor[9][4][0:4] = [1, 0, 1, 0]
infor[9][5][0:4] = [1, 0, 0, 1]
infor[9][6][0:4] = [1, 0, 1, 1]
infor[9][7][0:4] = [0, 0, 1, 1]
infor[9][8][0:4] = [1, 0, 1, 1]
infor[9][9][0:4] = [1, 0, 1, 0]

infor[9][0][2] = 0  # 将起点和终点封死
infor[0][9][3] = 0
```

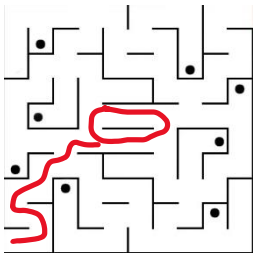
以 infor[0][0][0:4] = [0, 1, 0, 1]为例，其表示地图的第 0 行第 0 列（从 0 开始数），其不可向上通行、可向下通行、不可向左通行、可向右通行。

1.4.2 动态规划规则设置

动态规划的规则设置，主要在于明确：我当前需要暂存的信息、我要传递给下一级调用所需传递的信息、我需要全局记录的信息和我的终止判断信息。



动态规划其终究是个思想，那么就需要我们根据实际问题进行具体建模。很明显，这是一个“迷宫寻迹”的问题。我们先假设是自己在地图中进行寻迹，若我的出发点是（9,0），首先我要知道我的**当前位置**，并且需要知道**我当前可以走的路**或者说理论上可以行进的方向且未进行“完全探寻”的道路选择有哪些。当然，还有记录我**上一时刻的位置**（为了防止重复来回走，进入死循环），以及我需要一直牢记我的**目标位置点**。另外，为了防止出现在一个环形路段中反复绕弯，即出现以下情况，我们还需要对**当前已走过的所有路径**进行记



录。

在回退条件设置上，若出现以下情况，我们则退出此次路径寻找：

① 该路径已抵达终点

② 该路径下不存在未尝试过的走向选择

③ 该路径走到了已走过路线，即形成回环了

最后，我们还要设置最终“完成路径查找”条件，即**回退回起点位置，且起点的可行进方向数为 0。**

1.4.2 动态规划代码讲解

接下来我们从代码层面来解释动态规划过程：

```
def plan(map, pos_x, pos_y, goal_x, goal_y, direction, record, success):  # 动态规划
    if record == []:
        return pos_x, pos_y
    if len(set(record)) != len(record): #若路径重复，即出现回环，则退出该层（往后退）
        record.pop(-1)
        return pos_x, pos_y
    if pos_x == goal_x and pos_y == goal_y: #若到达终点，则后退，并记录该路径
        success.append(record.copy()) #记录成功走到终点的路径
        record.pop(-1) #往回退的同时删除当前路径 record 最后一个数据
        pos_x = record[-1] // 10 #运行到这里其实说明已经往后回退，则重新更新退回后的位置点
        pos_y = record[-1] % 10
        return pos_x, pos_y
    if map[pos_x][pos_y][0] == 1 and direction != 1: #若可以向上走而上一步不是往下走（避免重复行进）
        pos_x = pos_x - 1 #更新当前位置
        record.append(pos_x * 10 + pos_y) #记录当前位置
        pos_x, pos_y = plan(map, pos_x, pos_y, goal_x, goal_y, 0, record, success) #继续前进
        pos_x = record[-1] // 10 #运行到这里其实说明已经往后回退，则重新更新退回后的位置点
        pos_y = record[-1] % 10
    if map[pos_x][pos_y][1] == 1 and direction != 0: #若可以向下走而上一步不是往上走（避免重复行进）
        pos_x = pos_x + 1 #更新当前位置
        record.append(pos_x * 10 + pos_y) #记录当前位置
        pos_x, pos_y = plan(map, pos_x, pos_y, goal_x, goal_y, 1, record, success)
        pos_x = record[-1] // 10 #运行到这里其实说明已经往后回退，则重新更新退回后的位置点
```

```
    pos_y = record[-1] % 10
if map[pos_x][pos_y][2] == 1 and direction != 3: #若可以向左走而上一步不是往右走（避免重复行进）
    pos_y = pos_y - 1 #更新当前位置
    record.append(pos_x * 10 + pos_y)
    pos_x, pos_y = plan(map, pos_x, pos_y, goal_x, goal_y, 2, record, success)
    pos_x = record[-1] // 10 #运行到这里其实说明已经往后回退，则重新更新退回后的位置点
    pos_y = record[-1] % 10 #记录当前位置
if map[pos_x][pos_y][3] == 1 and direction != 2: #若可以向右走而上一步不是往左走（避免重复行进）
    pos_y = pos_y + 1 #更新当前位置
    record.append(pos_x * 10 + pos_y)
    pos_x, pos_y = plan(map, pos_x, pos_y, goal_x, goal_y, 3, record, success)
    pos_x = record[-1] // 10 #运行到这里其实说明已经往后回退，则重新更新退回后的位置点
    pos_y = record[-1] % 10 #记录当前位置
record.pop(-1) #当前已无路可走，即上下左右都走过了或者不可走，则后退，并删除最后一个记录
return pos_x, pos_y
```

具体为什么每个 plan 后的代码只有在退回后才能运行，这个和代码运行时编译器的解读有关系。这其实和单片机控制里面的中断程序很相似，或者说其实递归的原理可以理解为软件上的多级中断。

PS：其实 python 相对于 C 语言、C++不适合做递归，这个和 python 的数据结构有关系，我不太会……

以上代码可以将所有可以到达终点的路径进行全部存储，并保存在 success 参数中，接下来我们只需要找到“最优”的一条路径就行，具体代码如下：

```
def DF_FUN(map1, c_l, g_l, dir_car):  # dir_truenow 是小车当前运动的方向,绝对方向
    record = []
    success = []
    frequency = []
    corner = 0  # 转弯的总次数
    record.append(c_l[0] * 10 + c_l[1])
    plan(map1, c_l[0], c_l[1], g_l[0], g_l[1], -1, record, success) #调用动态规划
    # ----------------------将 plan 中枚举出来的所有情况根据转弯数目进行打分，取最小得分的情况
    if len(success) == 0: #若不存在到达终点的路径（被别人的车堵住了或者其他情况）
        return [], []  # 若车被堵死在死胡同，停止或者选择撞开？
    len_succ = []
    order_dir = []
    for i in range(len(success)):  # n 种走法
        score = 0 #计算每条路径的得分（惩罚分数，分数越低说明路径越优）
        dir_now = dir_car  # 车头现在的朝向
        for j in range(1, len(success[i])):  # 每种走法的每一步,上帝视角,绝对朝向
            dir_bef = dir_now
            if success[i][j] - success[i][j - 1] == 1:
                dir_now = 3
            elif success[i][j] - success[i][j - 1] == -1:
                dir_now = 2
            elif success[i][j] - success[i][j - 1] == 10:
                dir_now = 1
            else:
                dir_now = 0
```

```
            if dir_bef == dir_now:  #小车运动方向不变 （即若小车相对于上一位置直行）
                score = score + 1
            else: #小车运动方向改变 （即若小车相对于上一位置转弯）
                score = score + 2
        len_succ.append(score)  # len_succ 中添加一种走法的总惩罚得分
    len_min = len_succ.index(min(len_succ))
    #一开始需要对车头方向进行校准，即确定上一位置朝向，用于判断下一位置得分
    if success[len_min][1] - success[len_min][0] == 1:
        dir_record = 3  # 右
    elif success[len_min][1] - success[len_min][0] == -1:
        dir_record = 2  # 左
    elif success[len_min][1] - success[len_min][0] == 10:
        dir_record = 1  # 下
    else:
        dir_record = 0  # 上
```

2.环境配置

2.1 树莓派环境配置及常用软件

树莓派环境搭建（不过这个也有开始买来就装好的，可以直接用人家安装好的系统）
树莓派安装 OpenCV 教程
其他辅助软件推荐:
MobaXterm（下载免费的就行，这个是用来远程连接树莓派，方便上传/下载文件，不过需要开启 SSH 协议，这个树莓派环境搭建里面有，或者 CSDN 搜索一下就行）
VNC Viewer（这个是用来远程连接树莓派界面，这样不需要一直用屏幕看树莓派信息，远程用电脑看就行）
需要了解的 sudo 指令，超级权限者知识
PS：常用sudo命令： sudo su（开启root权限）sudo nano（利用root权限修改部分文件，因为普通的打开权限文件是只读的，权限不够）

2.2 视觉 python 代码撰写软件

可以使用 pycharm 或者 VS code，其各有优势，建议都使用:
pycharm 环境配置方便，但是打开速度慢（因为每次开启都会自动配置一堆文件），可以利用 pycharm 的 debug 模式进行逐行代码输出检查，适合用于查错。
VS code，主要使用到了其 Jupyter 环境，其优势在于可以分行分段进行代码运行，可以自动输出当前行最后一个变量的值，所有参数当前值都是全局记录的，运行顺序只与选择代码框，即 cell 的顺序有关。
综合来说，建议用 VS code 写代码，用 pycharm 做 debug，即代码查错。

2.3 摄像头选择

这次使用的摄像头是（全局 720p+180 帧），相机利用 USB 连上树莓派即可使用，不需要另外安装插件（当时是花了 678 元，不知道会不会涨价）。

【淘宝】https://m.tb.cn/h.5iOVulZ?tk=jafOW221tLv CZ0001 「1080P 全局快门 USB 摄像头拍摄运动物体高速电脑台式识别树莓派免驱」
点击链接直接打开 或者 淘宝搜索直接打开

其优势在于，全局快门相对于卷帘快门，其不会出现飞影现象，这有利于高速移动的小车进行图像获取。此外，该相机可以利用以下软件进行参数调节，包括白平衡、清晰度、曝光度等，并且可以对所更改参数进行记录。

参数调节建议：

若是进行颜色区分（本次任务），则

自动曝光——打开

自动白平衡——打开

色调——0

其余参数——适当调节

（适当调节即调节相机使得所获得的图像最接近我们人眼所观察的图像即可，**必须先调节相机使得图像正常，再更改所有代码中的颜色阈值!!!!!!**）

帧测速软件.exe

（该软件直接复制到桌面就能用，很小）

2.4 树莓派和单片机的连接

本身购买的单片机（轮趣科技 ROS 机器人控制板 STM32F407 主控小车四驱巡线雷达驱控一体）就支持树莓派的连接，并预留了树莓派固定的位置，这个应该在官方文件就有，或者找客服要一下就行，这块不是我做的所以也不太清楚。

3.基础知识学习建议

OpenCV 基础知识

Python 等编程语言的学习找 B 站播放量多的最新的就行，这个无所谓，或者甚至如果有其他语言基础，稍微涉猎一下，知道基础语法，即用即查就行。