# 项目一：基本原理

装箱问题是典型的NP难问题，一般使用启发式算法——贪婪算法、蚁群算法、禁忌搜索、遗传算法NF(Next Fit)近似算法，FF(First Fit)近似算法、FFD（First Fit Decreasing）近似算法、BF(best Fit)、BFD(Best Fit Deceasing)等。但是本次比赛我们一开始想要采取Alpha Go建立神经网络并训练，在分析多重比赛条件之后，我们没有采取它。转而使用了以重力势能为参考标准的算法，因为我们在观察了多处的装箱之后发现，在不考虑其他因素的情况下，较规则物体总是先排放在下，且体积或者面积比较大的物体的排列顺序优先。

本算法是基于最小的重力势能原理，将图形排放得越往下，利用率就会越高。在每次排放图形时，试图通过平移和旋转，进而尽量地降低图形的重心和最高高度，即使其更稳定地排布，最后找到属于当前待排图形的最优排放姿势。一开始，优先建立一个至少为4050的二维笛卡尔坐标系，以便于后续的图形排放工作。对于待排数据，本算法先对其进行一个面积的计算处理，然后再按照面积顺序将其由大到小地排列（因面积大的图形优先排放，所以整体的势能会处于一个较低状态即排放较稳定）并编号。本算法优先排布矩形，先进行矩形的各边计算，如果有n个矩形的排列能使n个矩形的某边之长等于40，则将n个矩形优先按照序列排布；如果没有使n个矩形的某边之长为40，则取其最接近40的值的n个矩形的排列为最优解，进而将这n个矩形优先排布。剩余矩形找到与已排图形相等或最相近边者优先叠加在已排矩形的上方。其余三角形则按照势能最小原理排布，而后每进行一次迭代就会排放待排序列里的图形，并对每个未排放的坐标点进行图形排放的重叠检测（包括与已排图形的检测和笛卡尔坐标系外的检测，详细见下数学模型（4））和旋转（详细见下数学模型（2））并记录，最终比较每个排样姿势的重心最小值，找到属于此图形的最佳排样姿势（即找到此图形合法排放下拥有最小重力势能的姿势）。排布完成后记录利用率和所有图形的排布点。

但是仅仅的按照面积排序是有局限性的，这样排序只能够保证每次排样的局部最优解，并不能够保证全局有一个好的排列。所以在将所有图形排序完成第一遍之后，本算法结合了回溯算法的思想，将顶层排布较不理想的图形重新排布，每遍历第n次就将排布最高的n个图形取出且将其代排顺序随机打乱，后重新排布，排布的方法与之前的排布相同。记Y为所有图形的理论面积与长度40的比值（即Y=所有图形理论面积/40），当遍历到第n次时，此时取出n个图形后，已排图形的最大y值小于Y时，不再继续遍历。且最终比较每次遍历后的利用率，图形的最终排布取利用率最高的排布样式。

# 项目二：数学模型

1. **最小重力势能原理算法：**

满足物理公式：

此时所有的图形的势能都可以由此算出，其中m指图形的质量（此处指图形的面积），g为重力常数（此处可省略），h则指图形的形心y坐标的值。

当某姿势的G值最小时，此姿势就是当前待排图形的最优排放姿势。

1. **旋转原理：**

以(,)作为参考点进行旋转个角度，依照以下数学公式得：

式中,为图形顶点m的坐标（m=1...n，n为图形顶点个数）

1. **规则图形形心计算原理：**

规则图形的形心计算公式如下：

式中指规则图形的第个顶点的横坐标,指规则图形的第个顶点的纵坐标。

1. **重叠检测算法原理：**

对于图形的重叠检测问题，本算法不是以图形为单位去检测，而是深化一开始的笛卡尔坐标系思想，将待排图形以顶点与形心等点的坐标位置信息来代表图形，并将重叠检测简化为两个步骤，只要其中之一有重叠则排样姿势不可取。

步骤一：如果待排图形的顶点和形心均在已排图形之外则此姿势合格。判断点P与图形的关系可以按照如下数学等式判断：



如果C0，则说明此时的P点是在图形的外部；如果C=0，则说明此时P点在图形的边上；如果C0，那么此时的P点在图形内。通过这一判据就可以判断点与图形的关系，进而判断待排图形与已排图形是否有重叠。

步骤二：检测待排图形的线与已排图形的线是否相交。在检测时，两条线重叠并不算相交，只有一个端点在另一条线上也不算相交，只有穿过才算相交。如图检测线=P1P2与线=P3P4是否相交，则可以转化为以下数学等式：



如果最终算出来的且，那么这两条线就是相互穿过即相交，此时可判断两个图形重叠；否则两条线就是存在着其他的情况。

如果对图形的顶点和形心检测过了步骤一，并且对图形的线检测过了步骤二，那就说明图形并没有重叠，即排样姿势是合格的。

# 项目三：代码实现方式

近年来，Python语言不断为高校的计算机研究以及智能机器学习等作为编程语言。与Python相比，[MATLAB](https://baike.baidu.com/item/MATLAB/263035" \t "_blank)主要专注于工程和科学计算。然而即使在计算领域，也经常会遇到文件管理、[界面设计](https://baike.baidu.com/item/%E7%95%8C%E9%9D%A2%E8%AE%BE%E8%AE%A1" \t "_blank)、[网络通信](https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E9%80%9A%E4%BF%A1)等各种需求。而Python有着丰富的扩展库，可以轻易完成各种高级任务，开发者可以用Python实现完整应用程序所需的各种功能。所以本算法使用计算机编程语言Python编写GUI界面，且其内部算法运算也是通过Python编写，部分关键实现算法如下：

1. 矩形排布：将传入的矩形取出并按面积从大到小排序，在一段可排长度内按长度最长面积最大安放。如在排布最开始时，待排长度D=40，先取第一个矩形R0按长l安放，再取下一个矩形R1按长边安放，直至到Rn时超过待排长度D，此时向下递归，取矩形Rn+1；全部矩形取完后向上回溯一个矩形Rn-1按宽h安放，直至最终完成递归得到最大的利用长度和最大的面积。

for i in range(len(rects)): # [[s, num, gender, [[], [], [], []]], ..]  
 s = rects[i][0]  
 num = rects[i][1]  
 gender = rects[i][2]  
 graph = rects[i][3]  
 l = graph[2][0] - graph[0][0]  
 h = graph[2][1] - graph[0][1]  
 new\_rects = rects.copy()  
 for j in range(i+1):  
 new\_rects.pop(0)  
 self.rollAnswer[num] = l  
 self.rollArea += s  
 new\_blank = blank - l  
 if new\_blank >= 0:  
 self.\_go(new\_blank, new\_rects)  
 self.rollAnswer.pop(num)  
 self.rollArea -= s  
 self.rollAnswer[num] = h  
 self.rollArea += s  
 new\_blank = blank - h  
 if new\_blank >= 0:  
 self.\_go(new\_blank, new\_rects)  
 self.rollAnswer.pop(num)  
 self.rollArea -= s

1. 三角形排布：将传入的三角形取出并按面积从大到小排序。利用将排放区域40×50划分出来的栅格点做横向遍历：即在每个横坐标上从下到上遍历，直至找到第一个可以排放的点，并做180×2°次旋转取到该位置上的重心最小姿势安放；在每一个图形的遍历中会得到40（栅格点横向数量）个姿势，取其中最高点最小的位置便是此三角形最优的位置。

for graphs in self.tris: # [[s, num, gender, [[], [], [], []]], ..]  
 s = graphs[0]  
 num = graphs[1]  
 gender = graphs[2]  
 thisGrapf = graphs[3]  
 chosenOne = self.getBestPos(s, num, gender, thisGrapf, mode=1)  
 self.\_\_flag.wait() # 暂停  
 if self.stopFlag: # 终止循环  
 return False  
 print(**'保存图形'**, chosenOne) # [Yc, y\_max, Xc, gender, location, num, s]  
 self.refreshData(chosenOne, save=True)  
 y\_max = self.saveData(chosenOne)

1. 三角形回溯：在首次将所有图形排布完毕之后会进行回溯，目的是增加上方图形排布的利用率，具体做法是：依次取出一个递增数量的最高的图形进行最高点+最高点（见三角形排布方法）的遍历，并输出最高利用率的一次记录。
2. 重叠检测：根据重叠检测原理，对于规则的图形（三角形，矩形），按照主动方向被动方检测的方式，即：在已排布图形区域，待排图形依次检查主动方的边线+中线与被动方的边线是否相交、主动方的各个点+重心是否在被动方区域内

'''  
待排图形与已排图形的重叠检测  
根据已排图形来  
location:待检查图形, [[], [], []]  
Xc:形心x  
Yc:形心y  
:return: 重叠T/不重叠F  
'''  
# 判断是否出界  
for point in location:  
 x = point[0]  
 y = point[1]  
 if (x < 0) or (x > self.gridX):  
 return True # 出现出界  
 if (y < 0) or (y > self.gridY):  
 return True # 出现出界  
 …  
 # 检查形心  
 exist0 = self.judgePointInner(Xc, Yc, settledGraph)  
 if exist0 == -1 or exist0 == 0: # 形心不能在里面或边上  
 return True # 形心在里面  
  
 # 检查点在图形内  
 for i in range(len(location)):  
 x = location[i][0]  
 y = location[i][1]  
 exist1 = self.judgePointInner(x, y, settledGraph) # 图形的顶点  
 if exist1 == -1: # 顶点可以在边上但不能在里面  
 return True #形心在里面  
  
 # 检查边界线香蕉  
 …  
 for line0 in line\_already:  
 for line1 in line\_noready:  
 exist = self.judgeLineCross(line1, line0) # 检查线段  
 if exist:  
 return True # 出现香蕉  
return False # 检查中没有发现重叠的情况

1. 判断点在多边形内：判断一个点是否在图形内，即判断点是否在图形的各个角内

flag = -1 # -1在里面；0在边上；1在外面  
for i in range(len(location)):  
 point = location[i]  
 if i == 0:  
 point\_next = location[i + 1]  
 point\_bef = location[-1]  
 elif i == len(location) - 1:  
 point\_next = location[0]  
 point\_bef = location[i - 1]  
 else:  
 point\_next = location[i + 1]  
 point\_bef = location[i - 1]  
 v0 = [x - point[0], y - point[1]]  
 v1 = [point\_next[0] - point[0], point\_next[1] - point[1]]  
 v2 = [point\_bef[0] - point[0], point\_bef[1] - point[1]]  
  
 # 叉乘之积  
 answer = (v0[0]\*v1[1] - v1[0]\*v0[1]) \* (v0[0]\*v2[1] - v2[0]\*v0[1])  
 if answer > 0: # 在外面或在边上  
 flag = 1  
 return flag  
 if answer == 0:  
 flag = 0  
  
return flag # 在里面

1. 判断线相交：

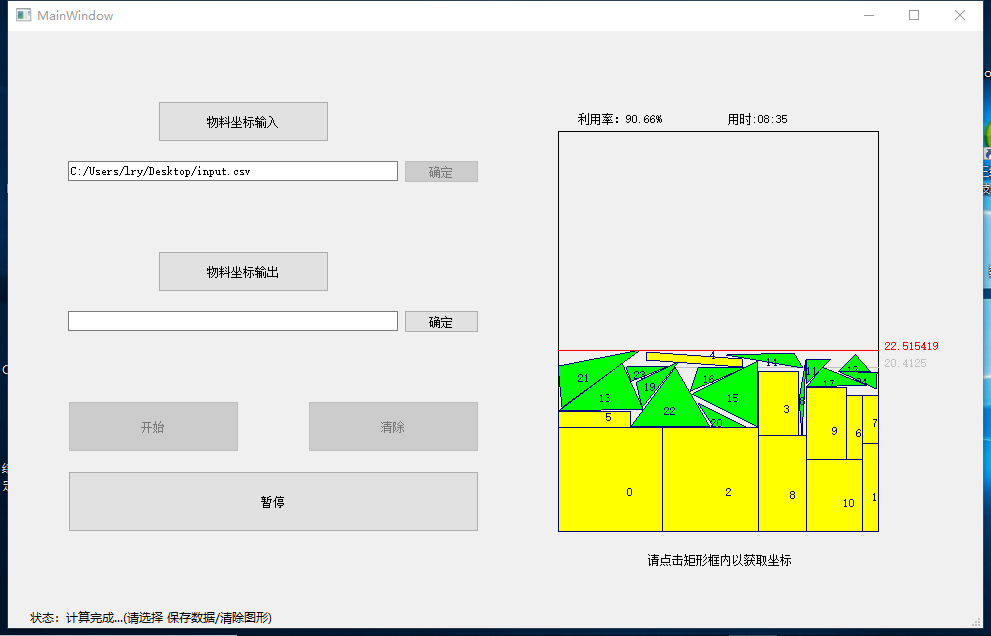
'''  
 两条线重叠不相交，只有一个端点在另一条上也不相交,只有穿过才香蕉  
 :line1:[[x0, y0], [x1, y1]]  
 :line2:[[x0, y0], [x1, y1]]  
 :return: 香蕉T/不相交F  
 '''  
 x0, y0 = line1[0][0], line1[0][1]  
 x1, y1 = line1[1][0], line1[1][1]  
 x2, y2 = line2[0][0], line2[0][1]  
 x3, y3 = line2[1][0], line2[1][1]  
 # a交b  
 vec0 = (x1-x0, y1-y0)  
 vec1 = (x2-x0, y2-y0)  
 vec2 = (x3-x0, y3-y0)  
 a = vec0[0]\*vec1[1] - vec0[1]\*vec1[0]  
 b = vec0[0]\*vec2[1] - vec0[1]\*vec2[0]  
 # b交a  
 vec0 = (x3 - x2, y3 - y2)  
 vec1 = (x0 - x2, y0 - y2)  
 vec2 = (x1 - x2, y1 - y2)  
 c = vec0[0] \* vec1[1] - vec0[1] \* vec1[0]  
 d = vec0[0] \* vec2[1] - vec0[1] \* vec2[0]  
 if a\*b<0 and c\*d<0:  
 return True  
 else:  
 return False

# 项目四：模拟测试数据及效果

1. **模拟测试数据：**

****

1. **效果：**



# 项目五：算法流程图

