1. 问题模型

假设待处理的任务数为j，一共m件;能够处理任务的机器数为i，一共n;每一台机器处理任务时的资源花费为SCij ; 每一台机器处理任务时花费的成本为CMij ;机器自身拥有的资源量为Sj；指派问题的模型如下：

Objection:

Constrain:

目标函数（1）中，寻求任务与机器间花费金钱成本最少的指派方案；约束条件(2)中，在生成的解方案中，每一个机器处理任务的资源消耗值不超过该机器自身拥有的最大资源值；约束（3）中，每个任务只能指派给一个机器处理；约束（4）机器与待处理任务之间存在一个二元决策变量Xij，取值范围为0或1。

二 . 算法逻辑（我这个程序的代码解释）

果蝇算法由main()、fitness\_func()两个函数组成。其中，通过main函数生成指派方案的初始集，并通过调用fitness\_func()从初始集中选出最优解（果蝇算法中的局部搜索进程），随后通过数次的迭代对最优解不断进行随机更新，寻找更优解，直到达到设定的最大迭代数。

三．果蝇算法步骤

1.算法数据的初始化

//设定需要处理的任务数、可以处理任务的机器数、最大迭代次数、种群规模

#本代码中，设置的任务数为20；可以处理任务的机器数为10；最大迭代数为1000；种群规模为100

//随机生成成本资源矩阵、资源花费矩阵、资源拥有量矩阵

#成本资源矩阵以机器数为行数，以任务数为列数，代表每一个机器处理任务的花费，比如：

依照上文设置的机器数为10，任务数为20，则成本矩阵是一个10\*20的矩阵。其中，矩阵内元素在[1,11]整数。

{ CostMoney = np.random.randint(1, 11, size=[NumMac, NumTask])}

#资源花费矩阵同成本矩阵一样，以机器数为行数，以任务数为列数，代表每台机器处理每个任务会消耗掉自身多少资源，比如：

同样依照上文的机器数与任务数，则资源花费矩阵是一个10\*20的矩阵，元素是在[10,100),不包括100

{CostSource = np.random.randint(10, 100, size=[NumMac, NumTask])}

如果CostSource[0] ==[10,20,30,40,50,60,70,80,90,99,99,90,80,70,60,50,40,30,20,10]

则0号机器处理0号任务会消耗自身资源10点，处理10号任务会消耗99点。

需要注意的是，和上面的成本矩阵不同，成本矩阵用于判断方案的成本情况，旨在对指派模型的目标函数的求得。而资源矩阵，则是对每台机器的约束。如果有一台机器，20个任务每个任务它都能用最快速度处理完，那么如果不加约束，则最优解就是将20个任务都分配给它。但现在加入约束，就是机器每处理一个任务就会消耗一定的资源，而每台机器自身拥有的资源有限，用完就不能再处理别的任务。

#资源拥有量矩阵是以机器数为列数的一维矩阵，和上面资源消耗情况的矩阵，共同形成对10个机器的任务分配约束，比如：

Source是一个1\*10的矩阵，一共有10个元素，取值范围在[100,1000)，不包括1000.

{Source = np.random.randint(100, 1000, size=[1, NumMac])}

若Source = [100,200,300,400,500,600,700,800,900,999]

则说明0号机器拥有资源100点，9号机器拥有资源999点。

//在初始的数据变量生成后，为变量设置取值边界。在生成的解方案中，元素不得超过机器号数的最大值，同时也不能小于机器号数的最小值。比如：

#该问题中的一组解是,考虑到总共有十个机器可以任务，因此将机器按0-9标号。因此，整个取值的下边界是一个1\*20，且元素都是0的矩阵；上边界同样是一个1\*20的矩阵，元素都是9。通过比较该解集中的每一个元素不大于9且不小于0，该方案在合理范围内。

{ bound = np.random.randint(1, 2, size=[2, NumTask])；

bound[0] = bound[0] \* (NumMac - 1) ;

bound[1] = bound[1] \* 0}

//生成初始代

#location是一个100\*20的矩阵，其元素范围取[0,10)，不包括10.

这是一个比较重要的矩阵，行数100代表随机生成的100种方案，这些方案不考虑资源问题，给20个任务随机分配给10个机器。

例如，location[0]=[1,9,8,6,0,3,4,5,7,8,9,6,5,4,2,3,6,5,1,9] 则说明第0中初始方案中，第0号任务随机分配给了1号机器，1号任务随机分配给9号机器。而本代码中放置在location中的随机分配方案的数量为100（方案数量可随机设置）。

2.fitness\_func()函数

//fitness\_func()几个存放数据的数组初始化

//计算上一步骤中初始代location的形状，用fall[]存放生成初始代中方案种群的总成本，detail定义分配方案的详细信息。

#location作为初始代方案种群（即果蝇种群）是一个行数为100，列数为20的矩阵，因此传入一个buf值来描述location的形状buf=(100,20)。

#fall是一个用来存放总成本矩阵，因初始代中生成100个指派方案，对应存入fall中的总成本为100。

#detial用来储存分配信息的详细方案，是一个3\*20的矩阵如：每一个指派方案中，20个任务指派给指定机器的方案、以及对应指派生成的成本花费表、资源花费表，比如：

detail[0]代表detail矩阵的第0行，存储指派方案；

detail[1]代表detail矩阵的第1行，存储每个任务指派机器的成本花费信息；

detail[2]代表detail矩阵的第2行，存储每个任务指派机器的资源花费信息；

//对每一个方案进行循环，初始共100次循环，因为有100种初始方案。

{ for i in range(buf[0]):}

//方案可行性检验和信息存储

//将当前方案下，遍历每一个任务对应指派机器的所需花费的成本，共20台机器，形成一个1\*20的矩阵。并将当前成本表中的成本按照正向排序，提取出对应的任务号，分别放置在buf1、buf2中。

#比如：若遍历20次后的成本花费矩阵为[5,3,1,2,4,5,7,6,9,3,8,3,2,1,5,4,3,6,8,7],按照顺序排列后是[1,1,2,2,3,3,3,3,4,4,5,5,5,6,6,7,7,8,8,9,],提取后任务号序列为[2,13,3,12,1,9,11,16,4,15,0,5,14,7,17,6,19,10,18,8]

{ buf1 = [CostMoney[gene[i, j], j] for j in range(buf[1])]；

buf2 = np.argsort(buf1)；

buf3 = copy.deepcopy(Source[0][:])}

//对每一个机器进行循环，一共进行20次，用来决定是否按照该方案指派。

//如果每一个任务消耗机器资源不大于机器自身的资源拥有量，则该任务的指派成立。将该任务指派的方案，及对应的资源消耗和成本花费的信息，存储到detail[]中。同时，更新每一个机器在指派后的资源拥有量，存储在buf3中。

{for j in range(buf[1]):#buf[1]为20

if buf3[gene[i, buf2[j]]]-CostSource[gene[i, buf2[j]], buf2[j]] >= 0:

detail[0, buf2[j]] = gene[i, buf2[j]]；

detail[1, buf2[j]] = CostSource[gene[i, buf2[j]], buf2[j]];

detail[2, buf2[j]] = CostMoney[gene[i, buf2[j]], buf2[j]];

buf3[gene[i, buf2[j]]] = buf3[gene[i, buf2[j]]] - CostSource[gene[i, buf2[j]], buf2[j]]}

//若该方案中，20个任务节点存在该节点的消耗机器资源小于机器自身的资源拥有量，则将该种节点保留，并等待二次分配。

//未分配任务的二次分配

//将未经过一次检测的任务节点存储在一个buf4的列表中。若buf4列表中，元素个数大于1，则遍历每一个未分配的任务节点。

//在挑选出未分配的任务节点数后，在所有的机器中，寻找该任务节点中自身资源还很充足的机器，存入buf5的列表中，并将符合条件的机器处理任务是产生的成本花费存储在buf6的列表中。

//若存在可接受分配的机器，选择存储在buf6中，成本资源最少的机器，并将方案为缺失的任务交给该机器，补齐detail[]存储节点中的机器序号，成本消耗表，以及资源消耗表。

#例，经过第一次任务循环后，未分配的任务节点列表为buf4 =[3,4,5,8]，则表明需要二次分配的任务为3，4，5，8这四个任务。依次循环四个任务节点，若满足任务节点3的机器号数依次为3，6，8，且每一个机器与任务3搭配方案所消耗的成本为8，10，5。

择选出搭配任务3的最佳机器为任务8，消耗成本为5，其他依次按照此种方式二次分配。

{for j in buf4:

buf5 = list([idx for (idx, val) in enumerate(buf3) if val >= CostSource[gene[i, j], j]])

buf6 = list([CostMoney[idx, j] for (idx, val) in enumerate(buf3) if val >= CostSource[gene[i, j], j]])

if len(buf5) >= 1:

buf7 = np.where(buf6 == min(buf6))[0]

detail[0, j] = buf5[buf7[0]]

detail[1, j] = CostSource[detail[0, j], j]

detail[2, j] = CostMoney[detail[0, j], j]

buf3[detail[0, j]] = buf3[detail[0, j]] - CostSource[detail[0, j], j]}

//若未能找出指派的节点，则输出无解，返回fall[ ]、detailall[ ]（总成本矩阵和分配信息矩阵）

//统计成本信息和方案数据

#统计一共消耗了多少钱、把本次情况消耗的钱情况存入fall[ ] ，一共100种情况；本次机器分配信息存入detailall中，共100种情况。

3.嗅觉搜索

//在初始100种情况中，寻找花费钱数最少的，作为最佳成本值f\_best,，记录下最小成本值出现的下表，在location中找出对应的指派方案。将该最低方案经过fitness\_func处理后，找到初始代中最佳方案(防止分配方案中存在分配超量的情况)

// location中初始100种方案通过fitness的处理后，已经选出当前最优方案，放在solution中，方案信息放在detail中，接下来就是1000（MaxIter）次迭代，对上面的最优解方案进行1000次重新优化

{ f\_best = min(buf);

buf1 = np.where(buf == f\_best)[0];

buf1 = buf1[0];

solution = copy.deepcopy(location[buf1, :]);

detail = copy.deepcopy((rdetail[buf1]))}

4. 进入重新迭代选取更优方案

//在迭代前，由于初始代中的最优解已经生成，此时果蝇均向最优的果蝇飞去。

{ for itera in range(MaxIter):

print(itera)}

//对最优方案解进行迭代，即对当前最优指派方案中被任务指派的机器进行移动，实现对最优解方案的更新，即让所有果蝇想比如：

第一代的最优解为[3,3,4,6,9,7,2,3,5,8,9,0,5,6,7,1,2,3,4,8]，将该方案矩阵与随机生成的1\*20矩阵，且元素在[-2,3)的矩阵相加。若该矩阵为[1,1,2,-1,2,2,-1,2,1,2,1,-2,1,1,2,1,-1,2,-2,1],则

0号任务： 3 + 1=4 任务0重新指派给4号机器

1号任务： 3 + 1 =4 任务1重新指派给4号机器

2号任务： 4 + 2 =6 任务2重新指派给6号机器

3号任务： 6 – 1 =5 任务3重新指派给5号机器

4号任务： 9 + 2 =11 任务4重新指派给11号机器

5号任务： 7 +2 =9 任务5重新指派给9号机器

6号任务： 2 – 1 =1 任务6重新指派给1号机器

7号任务： 3 + 2 =5 任务7重新指派给5号机器

8号任务： 5 + 1 =6 任务8重新指派给6号机器

9号任务： 8 + 2 =1 任务9重新指派给1号机器

10号任务： 9 + 1 =10 任务10重新指派给10号机器

11号任务： 0 – 2 =-2 任务11重新指派给-2号机器

12号任务： 5 + 1 =6 任务12重新指派给6号机器

13号任务： 6 + 1 =7 任务13重新指派给7号机器

14号任务： 7 + 2 =9 任务14重新指派给9号机器

15号任务： 1 + 1 =2 任务15重新指派给2号机器

16号任务： 2 – 1 =1 任务16重新指派给1号机器

17号任务： 3 + 2 =5 任务17重新指派给5号机器

18号任务： 4 – 2 =2 任务18重新指派给2号机器

19号任务： 8 + 1 =9 任务19重新指派给9号机器

综上最优解从[3,3,4,6,9,7,2,3,5,8,9,0,5,6,7,1,2,3,4,8]→[4,4,6,5,11,9,1,5,6,1,10,-2,6,7,9,2,1,5,2,9]

实现了任务指派机器序号的转移，将任务从原来指定的机器转移到另一个机器。

{ newss = np.sum([solution, np.random.randint(-2, 3, size=[1, NumTask])], axis = 0)}

5.更新后的评估机制

对待移动后生成的最优解应从取值边界和资源消耗与资源拥有量约束下而定义的fitness\_func（）函数方面，进行双重判断。

//更新的最优方案在边界限制中判断

//在算法数据的初始化中，定义了方案中元素取值的下边界不能低于bound[1]=[0，0，0…0](1\*20的矩阵)，同时也不能超过元素取值的上边界bound[0]=[9,9,9…9](1\*20的矩阵)

//若方案中的元素值超过上边界值bound[0]，则解集中超过的元素等于9；若方案中的元素低于下边界bound[1]中的元素值0时。则解集中低于0的元素等于0。

{for j in range(NumTask):

if newss[0, j]> bound[0, j]:

newss[0, j] = bound[0, j]

elif newss[0, j]< bound[1, j]:

newss[0, j] = bound[1, j]}

#例，在上面例子中，通过移动后，最优解集发生变化由[3,3,4,6,9,7,2,3,5,8,9,0,5,6,7,1,2,3,4,8]变为了[4,4,6,5,11,9,1,5,6,1,10,-2,6,7,9,2,1,5,2,9]。在比较变化后的每一个元素时，发现第10任务指派给的第10号机器超过上限；第11号任务指派给的第-2号机器地狱下边界范围。因此，通过上述边界取值，应将任务10→机器9、任务11→机器0。综上最优解由[4,4,6,5,11,9,1,5,6,1,10,-2,6,7,9,2,1,5,2,9]变成了[4,4,6,5,11,9,1,5,6,1,9,0,6,7,9,2,1,5,2,9]

5. 返回成本和方案信息，并更新最优解

//经过上一步对方案的判断及改进后，将完善后的方案存储在新建的列表中，依此循环100次后，形成新的果蝇种群

//在形成新的果蝇种群后，应对继续对每一个新方案进行fitness\_func（）函数处理

//若指派后的机器资源消耗量未超过自资源限制量，则更新后的方案是一个可行方案。

//若指派后的机器资源消耗量超过自身资源限制量，应对方案中未达到标准的任务点进行二次分配；

//若二次分配也无法寻找到符合约束条件的机器节点，则方案无法改变。

{ reala = fitness\_func(NumTask, NumMac, CostMoney, CostSource, Source, newss)}

//最后，与上一代进行成本比较，并保留最小成本值及分配信息，让果蝇继续向果蝇群中最优果蝇飞去，即想着成本最小的果蝇方向飞去。

#在比较过程中，若新生成的方案集合中的最小成本值小于上一代的最小成本值，则保留最小的成本值，即分配信息。（所有的果蝇继续想着新的最优果蝇，方向飞去，进入下一次寻优）

#若在比较的过程中，若新生成的方案集合中的最小成本值大于上一代的最小成本值，则保留上一代的最小的成本值，即分配信息。（所有的果蝇继续向着上一代的最优果蝇，方向飞去，组成新的种群进入下一次寻优）

//进行下一次迭代循环，直到迭代结束。

在完善后的初始代指派方案种群中，寻找最小成本的方案，记录成为最优解。

设置果蝇算法中的最大迭代次数 、种群数、任务数、机器数、成本花费表、资源花费表、资源拥有表、边界取值范围等数据初始化

生成指派方案的初始代

进入迭代阶段，对每一次更新后的最优方案群进行上述判断可行性

将生成的初始代经过fitness\_func()函数进行评价并完善

将完善的最优方案成本值与上一代最小成本值比较，若小于上一代，则更新指派任务中最小成本值记录；反之，则保留上一代成本值，继续迭代

结束

是否达到最大迭代值

G=G+1

No

yes