```
from operator import itemgetter, attrgetter, mul
import random
import sys
import os
import math
import re
import csv
import time
import datetime
import numpy as np
from scipy.special import comb
"""Developped by Jing Deng, amended by Eric Delmelle (contact: delmelle@gmail.com)"""
"""(1)try to add a constraint of area for the selection of patches"""
"""(2)for the new generationselection, we always keep the best two chromosomes """
"""(3)output all the solutions and selected etlism solutions"""
"""(4)Record the time information"""
"""(6)This version is to test with real dataset, id actually starts from 0"""
"""(7)Read the solution, and output the list of the selected patches"""
"""(8)Change the first generation to select less patches"""
"""(9)While reading data the first time, index the area of the patches,"""
"""and change the area objective as a percentage of the total area """
"""(10)Improve the method of refining the chromosome, """
"""(11)Output the number of patches that are selected and total area """
"""(12)Add another input as the mark for whether the parcels are selected,
and read in another datatable should be included."""
"""(13)This version is to test new dataset, adding more types of distances"""
"""(14)Considering cluster (also called reserve group) before we evaluate the solution"
""
"""(15)Now the distance is updated to: min, max, avg and centroid distance"""
"""(16)Slightly change the method of identifying cluster, the minimum distance between
two patches should be larger than threshold"""
"""(17)This version do two-step optimization/randomization"""
"""(18)Change the random selection of cluster to use some algorithm to improve the sele
ction of clusters, tabu idea"""
"""(19)Change the area constraint to each cluster, divided by the number of clusters"""

"""(20)Modify the method of fixing area and distance constraint"""
"""(21)Slightly change the function for generate intial chrom and mutate"""
"""(22)While fixing the constraint issue, remove the smaller one first"""
"""(23)Add the prune area process"""
"""(24)In this version, we relax the probability surface more, and add prune area proce
ss before and after the checkChro function"""
"""(25)Change some fix area steps based on probability not just distances"""
"""(26)This version is based on ga05112016weare, change it back to 100 polygon example"
""

# GLOBAL VARIABLE
solution_found = False


"""GA parameters"""
popN = 75 # n number of chromos per each generation
genesPerCh = 409 # length of chromos
crossover_rate = 0.8
mutation_rate = 0.3
max_iterations2 =75#75  # the number of iterations for patch selection process


"""Application parameters"""
alpha = .8
```

```
distThreshold = 20000.0 # distance threshold for clustering, this will be used as interval for generating centroids
intervalFactor = 1 # when generate the cluster centroid points, calculate the interval by mutiply distance threshold and this factor
max_iterations = 100 #M: number of outer loop iterations
target = 1000000.0
TabuLength = 10

solutionPoolIterations = 0
while solutionPoolIterations < 1:   #solutionPoolItereations is =1. Increase if you want to run the entire GA process multiple times

    #patchN = 2 # select a number of patches for land conservation
    totalRecords = 0  # how many pairs, records
    matrix = list() # to store all the data
    matrix2 = list() # to store the mark table
    distMatrixMin = np.zeros((genesPerCh,genesPerCh))
    distMatrixMax = np.zeros((genesPerCh,genesPerCh))
    distMatrixAvg = np.zeros((genesPerCh,genesPerCh))
    bigDistMatrix = np.zeros((genesPerCh,genesPerCh))
    bestN = 2 # output only the best two solutions
    areaCon = 0.0
    areaConP = 0.16
    areaTConP = 0.31
    areaTCon = 0.0
    areaList = [0.0]*genesPerCh
    indexList = [0]*genesPerCh # to mark whether it is available to change, preserved
    # We may not use the centroid points below
    cXList = [0.0]*genesPerCh # to store the x values for the patches' centroid points

    cYList = [0.0]*genesPerCh # to store the y values for the patches' centroid points

    markList = list() # to store the patches that are already preserved
    areaTotal = 0


    #These are all outFiles
    outputFile = "C:\\GIS\\generations.txt"
    bestOutputFile = "C:\\GIS\\bestChroms.txt"
    timeFile = "C:\\GIS\\time.txt"
    clusterFile = "C:\\GIS\\cluster.txt"
    outputCentroidFile = "C:\\GIS\\centroid.txt"
    outFile = "C:\\GIS\\out.txt"
    fOut = open(outFile, 'w')

    #For clusters:
    enableOutput = False #whether output the centroid points
    interval = 20000 # the interval distance between two cluster centroids
    cInter = 5 #what is the interval between two clusters
    clusterNum = 0 #how many clusters in total
     cList = [] #store the cluster ids
     #cCList = [] #store the number of patches for each cluster
     noClusSet = list() #to mark out those bad cluster combinations
     clusterList = [0]*genesPerCh #the final cluster result
     cluXList = list() #to store the centroid points of the clusters
     cluYList = list()
     clusMark = [0]*genesPerCh #to mark the cluster number for the patches
     distMask = [0.0]*genesPerCh #to store the minimum distance from patch to its closest cluster
     distMask2 = [0.0]*genesPerCh #to recalculate the distance, standardized and consider biodiversity weight
```

```
    distMask3 = [0.0]*genesPerCh #to store the distance biodiversity ration.
    weiList = [0.0]*genesPerCh #to store the biodiversity weight for each patchset
    #probList = [0.0]*genesPerCh #to store the probability for patch selection for each gene
    Ccombs = 0 #given the number of total clusters and clusters to select, the number of all the possible combinations
    #The extent of the data
    xMin = 0.0
    yMin = 0.0
    xMax = 0.0
    yMax = 0.0
    #To store the previous best selection of cluster
    cpList=[]
    #The best objective value for previous solution
    opVal = -99999999999999999999999.0

    """Based on the centroids of all the patches, get the extent"""
    def getExtent(cXList, cYList):
      xMin = 10000000000000000
      yMin = 10000000000000000

      xMax = -1
      yMax = -1
      extentLst = []
      for i in cXList:
        if xMin > float(i):
          xMin = float(i)
        if xMax < float(i):
          xMax = float(i)

      for i in cYList:
        if yMin > float(i):
          yMin = float(i)
        if yMax < float(i):
          yMax = float(i)
      extentLst = list()
      extentLst.append(xMin)
      extentLst.append(yMin)
      extentLst.append(xMax)
      extentLst.append(yMax)
      return extentLst

    """Read in the data from a csv file"""
    """Also generate a number of cluster centroids"""
    def init ():
      global totalRecords # need to modify the global copy of this variable
      global matrix
      global matrix2
      global areaTotal
      global areaList
      global indexList
      global cXList
      global cYList
      global markList
      global distMatrixMin
      global distMatrixMax
      global distMatrixAvg
      global interval
      global weiList

      parcelsDist = "C:\\GIS\\landconservation\\work\\weare\\data\\distance.csv"
```

```
168.      parcelsMark = "C:\\GIS\\landconservation\\work\\weare\\data\\info.csv"
169.      with open(parcelsDist,'rb') as file:
170.        contents = csv.DictReader(file)
171.        for row in contents:
172.            matrix.append(row)
173.            totalRecords = totalRecords + 1
174.    # Store the information about whether the parcel is already conserved in matrix2
175.      with open(parcelsMark,'rb') as file:
176.        contents2 = csv.DictReader(file)
177.        for row in contents2:
178.            matrix2.append(row)
179.      #store the size of patches in a list and calculate the total area
180.      for i in range(genesPerCh):
181.        if int(matrix2[i]['CON']) == 1:
182.          indexList[i] = 1
183.          markList.append(i)
184.        else:
185.          indexList[i] = 0
186.        if areaList[i] == 0:
187.          areaList[i] = float(matrix2[i]['ACRES'])
188.        if cXList[i] == 0.0:
189.          cXList[i] = matrix2[i]['X']
190.        if cYList[i] == 0.0:
191.          cYList[i] = matrix2[i]['Y']
192.        if weiList[i] == 0.0:
193.          weiList[i] = float(matrix2[i]['W'])
194.      #standardize the weilist
195.      weiMax = max(weiList)
196.      weiMin = min(weiList)
197.      for i in range(genesPerCh):
198.        weiList[i] = float((weiList[i]-weiMin)/(weiMax-weiMin)+0.0001)
199.      #print weiList
200.      for i in range(totalRecords):
201.        index = int(matrix[i]['PAID'])
202.        index2 = int(matrix[i]['PBID'])
203.        # Read the distance and store in the matrix
204.        distMatrixMin[index][index2]=float(matrix[i]['DISTMIN'])
205.        distMatrixMin[index2][index]=distMatrixMin[index][index2] # for later searching
    purpose
206.        distMatrixMax[index][index2]=float(matrix[i]['DISTMAX'])
207.        distMatrixMax[index2][index]=float(matrix[i]['DISTMAX'])
208.        distMatrixAvg[index][index2]=float(matrix[i]['DISTAVG'])
209.        distMatrixAvg[index2][index]=float(matrix[i]['DISTAVG'])
210.      #We print out the data to check whether we read in the data correctly
211.      #print matrix2[0]['ACRES']
212.      #print matrix2[0]['X']
213.      #print matrix2[0]['Y']
214.      #print distMatrixMin[21][184]
215.      global areaCon
216.      global areaTCon
217.      for j in range(len(areaList)):
218.        areaTotal = areaTotal + float(areaList[j])
219.      areaCon = float(areaConP * areaTotal)
220.      areaTCon = areaTotal * areaTConP
221.      #print 'Total area of the patches is:%s' %(areaTotal)
222.      #print 'Constraint area for each cluster is:%s' %(areaCon)
223.      #print 'Constraint area for all the site selection is: %s' %(areaTCon)
224.
225.      #Generate the clusters
226.      #below is using the predefined reserve centroid points
227.      lst = [947994,187782,991752, 237660]
```

```
228.       interval = 20000
229.       generateCluster(lst, interval)
230.       #lst = getExtent(cXList,cYList)
231.       #interval = int(lst[2] - lst[0])/5
232.       #generateCluster(lst,interval) #For the real data extent
233.
234.
235.
236.     """This function is to generate the list of clusters based on the extent of the rea
    l data, and calculate the centroid points"""
237.     """Parameter is the extent of """
238.     def generateCluster(extentLst,interval):
239.       # Get the extent and generate the list to store all the centroid points
240.       global xMin
241.       global xMax
242.       global yMin
243.       global yMax
244.       global Ccombs
245.       xMin = extentLst[0]
246.       yMin = extentLst[1]
247.       xMax = extentLst[2]
248.       yMax = extentLst[3]
249.
250.       global clusterNum
251.       global cInter
252.       global cluXList
253.       global cluYList
254.
255.       for i in range(int(xMin),int(xMax),interval):
256.         x = i
257.         for j in range(int(yMin),int(yMax),interval):
258.           y = j
259.           cluXList.append(x)
260.           cluYList.append(y)
261.           clusterNum = clusterNum + 1
262.       cInter = int(clusterNum / 10)
263.       #print "Generate "+str(clusterNum)+" cluster centroid points."
264.       #Ccombs = numPair(clusterNum - 1) #this can actually only handle two clusters sel
    ection, and the first one is not considered
265.       Ccombs = comb(clusterNum,p,exact=True) #this is not working correctly
266.       #Ccombs = 2925 #I give it a value directly C25,3
267.       #Ccombs = 300
268.       #print "There are "+str(Ccombs)+" combinations in total."
269.       if enableOutput == True:
270.         cenFile = open(outputCentroidFile,'w')
271.         for i in range(len(cluXList)):
272.           cenFile.write(str(cluXList[i])+"\t"+str(cluYList[i])+"\n")
273.         cenFile.close()
274.       global bigDistMatrix
275.       for i in range(genesPerCh):
276.         for j in range(len(cluXList)):
277.           bigDistMatrix[i][j] = distPatch(float(cXList[i]),float(cYList[i]),float(cluXL
    ist[j]),float(cluYList[j]))
278.
279.     """c:the number of clusters in total, p: how many clusters to select"""
280.     """Keep working until the clusters fit the requirments, the area balance"""
281.     def selCluster(c,p):
282.       cList2 = randomClu(c,p)
283.       #cList2 = [8,16,18]
284.       cList3 = tuple(cList2)
285.       #print cList2
```

```
286.      while cList3 in noClusSet or balanceClus(cList2) == False:
287.        #if it's not balanced, have to select again
288.        #print "not balance"
289.        cList2 = randomClu(c,p)
290.        cList3 = tuple(cList2)
291.      else:
292.        return cList2
293.
294.    """This function is to do random selection of clusters"""
295.    def randomClu(c,p):
296.      cl = set() #the id of selected cluster, can't have repeat id
297.      while len(cl) < p:
298.        # if p > 1:
299.        #   #calculate the interval
300.        #   inter1 = len(cl)*int(c/p)
301.        #   inter2 =(len(cl)+1)*int(c/p)
302.        #   if inter1 == 0:
303.        #     inter1 = 1
304.        #   if inter2 > c-1:
305.        #     inter2 = c-1
306.        #   cl.add(random.randint(inter1,inter2))
307.        # else:
308.        #   cl.add(random.randint(1,c-1))
309.
310.        #cl.add(random.randint(1,c-1))
311.        cl.add(random.randint(0,c-1))
312.      cl = list(cl)
313.      return cl
314.
315.    """This function is to swap the cluster selection, random select one cluster number
    to swap"""
316.    """The change is based on the existing list cList"""
317.    def swapCluster(cl,c,p):
318.      cList2 = randomClu2(cl,c,p)
319.      cList3 = tuple(cList2)
320.      count = 0
321.      while cList3 in noClusSet or balanceClus(cList2) == False:
322.        #if it's not balanced, have to select again
323.        #print "swap clusters again"
324.        #print len(noClusSet)
325.        if count < p:
326.          cList2 = randomClu2(cl,c,p)
327.          count += 1
328.        else:
329.          cList2 = randomClu(c,p)
330.        cList3 = tuple(cList2)
331.      else:
332.        return cList2
333.
334.    """This function is to implement the swap, it should be applied together with the f
   unction swapCluster"""
335.    """c:the number of clusters in total, p: how many clusters to select"""
336.    def randomClu2(cl,c,p):
337.      #get a random position
338.      loc = random.randint(0,p-1)
339.      #print "the random location"
340.      #print loc
341.      #print cList
342.      list2 = list(cl) #don't want to change cl
343.      cID = list2[loc]
344.      del list2[loc] #remove the element at that position
```

```
345.      cList2 = set(list2)
346.      a = random.randint(0,1)
347.      while len(cList2)<p:
348.        # if a==0:
349.        #   cID += cInter
350.        # else:
351.        #   cID -= cInter
352.        # if cID > c-1:
353.        #   cID -= int(c/2)
354.        # if cID < 1:
355.        #   cID += int(c/2)
356.        # cList2.add(cID)
357.        cList2.add(random.randint(1,c-1)) #random change one cluster number
358.      cl = list(cList2)
359.      return cl
360.
361.    """Random select clusters and identify cluster number for each patch"""
362.    """The input parameter is the list of cluster id"""
363.    """Generate a weight mask based on distance mask and weight"""
364.    def defineClusterPatch(cl):
365.      #print cl
366.      global distMask
367.      global clusMask
368.      for i in range(genesPerCh):
369.        # For each patch get the minDist
370.        distL = {}
371.        if len(cl)==0:
372.          print "wrong!!!"
373.          #print cl
374.
375.        for j in cl:
376.          x1 = float(cXList[i]) #patch centroid point
377.          y1 = float(cYList[i])
378.          x2 = float(cluXList[j]) #cluster centroid
379.          y2 = float(cluYList[j])
380.          dist = distPatch(x1,y1,x2,y2)
381.          key = str(j)
382.          distL[key] = dist
383.        minDist = sorted(distL.items(),key=itemgetter(1))
384.        clusMark[i] = int(minDist[0][0])
385.        distMask[i] = float(minDist[0][1])
386.
387.      reCalDist(alpha,bigDistMatrix)
388.
389.    """This function is to recalculate the distmask"""
390.    """It is the normalized distance"""
391.    def reCalDist(alpha,bigDistMatrix):
392.      global distMask2
393.      global distMask3
394.      #distMin = float(min(distMax))
395.      #distMax = float(max(distMax))
396.      distMax = float(np.amax(bigDistMatrix))
397.      distMin = float(np.amin(bigDistMatrix))
398.      distRg = float(distMax - distMin)
399.      distMax = 61140.7880702
400.      distMin = 200.3293703
401.      distRg = float(distMax - distMin)
402.      # a = distPatch(float(cXList[11]),float(cYList[11]),float(cluXList[8]),float(cluY
    List[8]))
403.      # di = float((a-distMin)/distRg)+0.001
404.      # print di
```

```
405.      #print sum(weiList)
406.      for i in range(len(distMask)):
407.        distMask2[i] = (float(distMask[i])-
    float(distMin)+0.0001)/(float(distRg)+0.001)
408.        distMask3[i] = float(math.pow(math.pow(distMask2[i],1),alpha)/math.pow(weiList[
    i],(1-alpha)))
409.
410.
411.    """This function is to check whether the selection of cluster make sense, the patch
    es that belong to a cluster have enough sum area"""
412.    def balanceClus(cl):
413.      #todo: for each cluster, calulate whether the patches area can be larger than the
     constraints
414.      defineClusterPatch(cl)
415.      global noClusSet
416.      #Check each cluster, if not working, return false
417.      #areaPerClu = float( areaTCon / p)
418.      areaPerClu = areaCon
419.      for eachClus in cl:
420.        # Each cluster
421.        areaTotal = 0.0
422.        for j in range(genesPerCh):
423.          #print eachClus
424.          if clusMark[j] == int(eachClus):
425.            areaTotal += float(areaList[j])
426.        if areaTotal < areaPerClu:
427.          #If it doesn't fit the solution, then this combination should be marked out
428.          #print "The solution doesn't balance, remove the solution."
429.          if len(noClusSet)<TabuLength:
430.            noClusSet.append(tuple(cl))
431.          else:
432.            del noClusSet[0]
433.            noClusSet.append(tuple(cl))
434.          #print areaTotal
435.          return False
436.      return True #in the end can return true
437.
438.
439.    """To set the probability"""
440.    """To set the probability"""
441.    def setPatchesProb(clusMark,distMask,cluList):
442.      probaPatches = [0.0]*genesPerCh
443.      listAll={}
444.      # For each cluster, sort the distance list and select the closest one
445.      for eachCluster in range(p):
446.        areaP = 0.0 #total area of the patch
447.        listC = {}
448.        listC2 = {} #To store all the preserved patches, they need to be add into clust
    er first
449.        for bit in range(genesPerCh):
450.          # add the distance and gene id into the dictionary, then sort the dictionary

451.          if clusMark[bit] == cluList[eachCluster]:
452.            key = str(bit)
453.            if int(indexList[bit]) == 1: #already preserved
454.              listC2[key] = distMask[bit]
455.              probaPatches[bit] = 1
456.            else:
457.              listC[key] = distMask[bit]
458.
459.        # The first round, add all the preserved patches
```

```
460.         for item in listC2:
461.           pIdx = int(item[0])
462.           areaP += float(areaList[pIdx])
463.           probaPatches[pIdx] = 1 #They have to be selected
464.
465.         # Then if we still have room, find the patches by order, smaller distance has m
    ore priority
466.         sorted_l = sorted(listC.items(),key=itemgetter(1))
467.         prob = 0.999 # the initial high probability
468.         while areaP < float(0.6*areaCon) and len(sorted_l) > 0:
469.         #while areaP < float(0.7*areaTCon/p) and len(sorted_l) > 0:
470.           pIdx = int(sorted_l[0][0])
471.           if probaPatches[pIdx] != 1:
472.             probaPatches[pIdx] = prob#These patches all have high probability to be sel
    ected
473.             prob = prob - 0.01
474.           areaP += float(areaList[pIdx])
475.           sorted_l.pop(0) #after sorting, it is a list, and remove the first
476.
477.         # The second round
478.         probA = 0.5 # the initial high probability
479.         while areaP < float(0.9*areaCon) and len(sorted_l) > 0:
480.         #while areaP < float(areaTCon/p) and len(sorted_l) > 0:
481.           pIdx = int(sorted_l[0][0])
482.           if probaPatches[pIdx] != 1:
483.             probaPatches[pIdx] = probA#These patches all have high probability to be se
    lected
484.           areaP += float(areaList[pIdx])
485.           sorted_l.pop(0) #after sorting, it is a list, and remove the first
486.           probA = probA - 0.01
487.
488.         # The last round, the patches left in the list will have very low probability t
    o be selected
489.         probB = 0.001
490.         for item in sorted_l:
491.           pIdx = int(item[0])
492.           if probaPatches[pIdx] != 1:
493.             probaPatches[pIdx] = probB
494.
495.       #print probaPatches
496.       #print it out to a file, this is for testing purpose
497.       # outputProbFile = "C:\\GIS\\landconservation\\work\\simulateData100poly\\prob"+s
    tr(cluList)+".txt"
498.       # outputProbFile = "C:\\GIS\\landconservation\\work\\simulateData100poly\\prob.tx
    t"
499.       # if cluList == [8,16,18]:
500.       #   probFile = open(outputProbFile,'w')
501.       #   print >> probFile, probaPatches
502.       #   probFile.close
503.       return probaPatches
504.
505.
506.     """This function is to generate a number of patches selection based on the probaPat
    ches"""
507.     """Input is the probability list, and output is the chromosomes"""
508.     """Apply the two constraints in the chromosome selection, totalArea and distance th
    reshold"""
509.     """Make sure for each cluster, at least certain area is selected"""
510.     """Also to ensure the performance, make sure one chromsome strictly follow the prob
    abilities"""
511.
```

```
512.    def generatePop2(probaPatches, popN):
513.      chromos, chromo = [], []
514.      #global cCList
515.      #add other chromos
516.      while len(chromos) < popN:
517.        chromo = []
518.        for bit in range(genesPerCh):
519.          if probaPatches[bit] > 0.9:
520.            chromo.append(1) #The reserved ones
521.          else:
522.            if random.random() < probaPatches[bit]:
523.              chromo.append(1)
524.            else:
525.              chromo.append(0)
526.
527.        c1 = checkChro(chromo,probaPatches)
528.        if sum(c1)==0:
529.          print "check12"
530.        if sum(c1)>genesPerCh:
531.          print "check chromo here2"
532.        chromos.append(c1)
533.      return chromos
534.
535.
536.    """This method will completely select all the patches"""
537.    def relaxChro(ch):
538.      for i in range(genesPerCh):
539.        ch[i] = 1
540.      #return ch
541.
542.    """This method is to check the area constraints and the fix the distance conflict""
    "
543.    """It seems like here is the problem, we should repeatedly do checking distance and
    ara constraint until we reach a certain criteria, say 10 iterations"""
544.    def checkChro(ch,probaPatches):
545.      fixed = False
546.      cd = checkDist(ch)
547.      if cd == False:
548.        #to fix the distance first if these is any problem
549.        ch = fixDist(ch,probaPatches)
550.        if sum(ch) == genesPerCh:
551.          return ch
552.      if checkDist(ch) == False:
553.        print "Distance problem is not fixed"
554.
555.      ca = checkArea(ch)
556.      if ca == "C100":
557.        #no need to fix anything
558.        ch1 = pruneArea(ch,0,probaPatches)
559.        if checkArea(ch1) !="C100":
560.          print "Area problem 1!"
561.        return ch1
562.      elif ca == "C99":
563.        #only need to fix the total area
564.        fixAreaTotal(ch, probaPatches)
565.        if sum(ch) == genesPerCh:
566.          return ch
567.        if checkArea(ch) != "C100" :
568.          #print ch
569.          print "Area problem 2!"
570.      else:
```

```
571.         #need to fix the cluster too
572.         fixArea(ch,probaPatches)
573.         if sum(ch) == genesPerCh:
574.           return ch
575.         if checkArea(ch)!="C100":
576.           print "Area problem 3!"
577.
578.       if checkDist(ch) == True and checkArea(ch) == "C100":
579.         fixed = True
580.       else:
581.         print "here has problem again"
582.       return ch
583.
584.     """This function is to check whether it has distance conflict"""
585.     def checkDist(ch):
586.       if sum(ch)==genesPerCh:
587.         return True
588.       for bit in range(genesPerCh-1):
589.         if ch[bit] == 1:
590.           for bit2 in range(bit+1,genesPerCh):
591.             if ch[bit2] == 1:
592.               if clusMark[bit]!=clusMark[bit2] and distMatrixMin[bit][bit2] < distThres
    hold:
593.                 return False
594.       return True
595.
596.     """This method is to check the distance conflict and fix it"""
597.     """All the patches in one cluster have to be far away from all the patches in anoth
    er cluster, using the threshold distance"""
598.     """While detecting conflict, identify the two conflicting patches and drop one of t
    hem"""
599.     def fixDist(ch,probaPatches):
600.       global noClusSet
601.       #First, check the distance conflicts, in this step, may remove several patches
602.       for bit in range(genesPerCh-1):
603.         if ch[bit] == 1:
604.           patch1 = bit
605.           for bit2 in range(bit+1,genesPerCh):
606.             if ch[bit2] == 1:
607.               patch2 = bit2
608.               if clusMark[patch1]!=clusMark[patch2] and distMatrixMin[patch1][patch2] <
     distThreshold:
609.                 #They are too close, need to fix it
610.                 if probaPatches[patch1]==1 and probaPatches[patch2]==1:
611.                   #print "There is distance conflict in the reserved patches: "+str(pat
    ch1) + " and "+str(patch2)+", please readjust the threshold distance!"
612.                   #Can't use this cluster group, so next time it won't show up
613.                   #noClusSet.add(tuple(cList))
614.                   relaxChro(ch)
615.                   return ch # directly finish this
616.                 elif probaPatches[patch1] == 1:
617.                   # can't change the patch1
618.                   ch[patch2] = 0
619.                   probaPatches[patch2] = 0 #can't add this patch, cause it will cause c
    onflict with a reserved patch
620.                   # clus = int(clusMark[patch2])
621.                   # for a in range(len(cList)):
622.                   #   if clus == cList[a]:
623.                   #     cCList[a] = cCList[a] - 1
624.                   #     break
625.                 elif probaPatches[patch2] == 1:
```

```
626.                   # can't change patch2
627.                   ch[patch1] = 0
628.                   probaPatches[patch1] = 0
629.                   # clus = int(clusMark[patch1])
630.                   # for a in range(len(cList)):
631.                   #   if clus == cList[a]:
632.                   #     cCList[a] = cCList[a] - 1
633.                   #     break
634.                 else:
635.                   #if probaPatches[patch1] > probaPatches[patch2]:
636.                   #instead of using the probability, using the distance
637.                   if distMask3[patch1] < distMask3[patch2]:
638.                     ch[patch2] = 0
639.                     #instead of chaing the proability to 0, decrease it, lower the prob
    ability of choosing this again
640.                     # this penalty is higher because it's not random
641.                     probaPatches[patch2] = probaPatches[patch2] - 0.5
642.                     # clus = int(clusMark[patch2])
643.                     # for a in range(len(cList)):
644.                     #   if clus == cList[a]:
645.                     #     cCList[a] = cCList[a] - 1
646.                     #     break
647.                   #elif probaPatches[patch1] < probaPatches[patch2]:
648.                   elif distMask3[patch1] > distMask3[patch2]:
649.                     ch[patch1] = 0
650.                     probaPatches[patch1] = probaPatches[patch1] - 0.5
651.
652.                   else:
653.                     #03112016 can slightly improve this by choosing the one with bigger
     area
654.                     if float(areaList[patch1]) < float(areaList[patch2]):
655.                       ch[patch1] = 0
656.                       probaPatches[patch1] = probaPatches[patch1] - 0.4
657.
658.                     else:
659.                       ch[patch2] = 0
660.                       probaPatches[patch2] = probaPatches[patch2] - 0.4
661.
662.       return ch
663.
664.
665.     """This method is to refine the chromos applying the area constraints"""
666.     """Add a new feature to keep area balance between different clusters"""
667.     """The input probablity of the patches have already been changed"""
668.     """A random patch will be selected to add the total area, but also based on the clu
    ster requirments"""
669.     """The previous name is refineChrom"""
670.     """In stead of this random fix, maybe should do deterministic way, based on the pro
    bability"""
671.     """In this version, I will rank the probability and add the top ones into the list,
     if all the probability is less than 0, then return relaxed chrom"""
672.     """Now since i added two fix area method, this one is the back up"""
673.     def fixArea_bak(ch,probaPatches):
674.       # calculate the total area and sort the area of all the chromos
675.       area = 0.0
676.       #cAreas = {} # to store the total area for different clusters
677.       cArea = [0.0]*p
678.       #update the area list
679.       for i in range(p):
680.         cIndx = cList[i]
681.         for bit in range(genesPerCh):
```

```
682.           if ch[bit]==1 and clusMark[bit]==cIndx:
683.             cArea[i] += float(areaList[bit])
684.
685.       listArea={} #sort the probability
686.       listA_sort={}
687.       for bit in range(genesPerCh):
688.         if probaPatches[bit] > 0.1 and ch[bit]==0:
689.           # put the available patches into the list and sort by probabilities
690.           key = str(bit)
691.           #listArea[key] = probaPatches[bit]
692.           #listArea[key] = float(areaList[bit])*probaPatches[bit]
693.           listArea[key] = float(distMask3[bit])
694.           #should i use the probability or the area?
695.       listA_sort = sorted(listArea.items(),key=itemgetter(1))
696.
697.       if len(listA_sort) == 0:
698.         #nothing to add to the list
699.         relaxChro(ch)
700.         return ch
701.       #else:
702.         #print listA_sort[0]
703.       #check the cluster list
704.       for i in range(p):
705.         if cArea[i] < areaCon:
706.           #only do these when cluster doesn't not meet the minimum area requirement
707.           cIndx = cList[i]
708.           l = len(listA_sort)
709.           a = 0
710.           while a < l-1:
711.             if a == len(listA_sort) - 1:
712.               break
713.             #keep adding a patch into the list
714.             if cArea[i] < areaCon:
715.               idx = int(listA_sort[a][0]) #patch id
716.               if clusMark[idx] == cIndx:
717.                 ch[idx] = 1
718.                 cArea[i] += float(areaList[idx])
719.                 listA_sort.pop(a) # i don't know whether should keep this
720.               else:
721.                 a = a + 1
722.             else:
723.               break
724.         if cArea[i] < areaCon:
725.           #still don't reach the cluster requirement
726.           relaxChro(ch)
727.           return ch
728.
729.       #check the complete probability list
730.       while sum(cArea) < areaTCon and len(listA_sort)>0:
731.         #If all the clusters meet the minimum requirement, but the total area dose not
    meet the requirement
732.         idx2 = int(listA_sort[0][0]) #directly add the highest patch
733.         ch[idx2] = 1
734.         for i in range(p):
735.           if int(clusMark[idx2]) == cList[i]:
736.             cArea[i] += float(areaList[idx2])
737.         listA_sort.pop(0)
738.
739.       if sum(cArea) < areaCon:
740.         #still didn't meet the requirements after adding all the patches
741.         relaxChro(ch)
```

```
742.       return ch
743.
744.     """New version after 04182016"""
745.     """The function is to apply another two are fix process"""
746.     def fixArea2(ch,probaPatches):
747.       #cAreas = {} # to store the total area for different clusters
748.       cArea = [0.0]*p
749.       #update the area list
750.       for i in range(p):
751.         for bit in range(genesPerCh):
752.           if ch[bit]==1 and int(clusMark[bit])==int(cList[i]):
753.             cArea[i] += float(areaList[bit])
754.
755.       #for each cluster, fix the cluster area issue
756.       for i in range(p):
757.         if cArea[i] < areaCon:
758.           a = cList[0]
759.           ch = fixAreaClus(ch,a,probaPatches)
760.           if sum(ch)==genesPerCh:
761.             return ch
762.
763.       check = checkArea(ch)
764.       if check != "C99" and check!="C100":
765.         print "something wrong, please check here"
766.         relaxChro(ch)
767.       return ch
768.
769.     """When fixing area, need to make sure that not introducing new distance conflicts,
    so each new added one will be compared with others"""
770.     """If fail to fix the total area, return relaxed chromosome"""
771.     def fixAreaTotal(ch,probaPatches):
772.       areaT = 0.0
773.       for bit in range(genesPerCh):
774.         if ch[bit]==1:
775.           areaT += float(areaList[bit])
776.       if areaT > areaTCon:
777.         return
778.
779.       listArea={}
780.       listA_sort={}
781.       for bit in range(genesPerCh):
782.         if probaPatches[bit] > 0 and ch[bit]==0:
783.           # put the available patches into the list and sort by probability
784.           key = str(bit)
785.           #listArea[key] = float(probaPatches[bit])
786.           listArea[key] = float(-distMask3[bit])
787.
788.       if len(listArea) == 0:
789.         relaxChro(ch)
790.         return
791.       # print "the length"
792.       # print len(listArea)
793.       #Use the probability, decending order
794.       listA_sort = sorted(listArea.items(),key=itemgetter(1),reverse=True)
795.       while len(listA_sort)>0:
796.         if areaT>areaTCon:
797.           break
798.         pId = int(listA_sort[0][0])
799.         #check all the genes in the chromosome, whether they have conflict with this pa
    tch
800.         addpId = True
```

```
801.        for bit in range(genesPerCh):
802.          if bit!=pId and ch[bit] == 1 and clusMark[bit]!=clusMark[pId]:
803.            if distMatrixMin[bit][pId] < distThreshold:
804.              #not able to add
805.              addpId = False
806.              break
807.        if addpId == True:
808.          ch[pId] = 1
809.          areaT += areaList[pId]
810.          #print "add" + str(pId)
811.        listA_sort.pop(0)
812.
813.      #print ch
814.      #After the fix process, see whether we can finally reach the total area constrain
    t
815.      if areaT < areaTCon:
816.        #it still not enough area
817.        relaxChro(ch)
818.      return
819.      #return ch
820.
821.    """The fix process for this function is to fix the single clusters first, then the
    total area"""
822.    def fixArea(ch,probaPatches):
823.      cSList = [] # To store the cluster
824.      cArea = [0.0]*p
825.      #calculate the total area for the specific cluster
826.      for i in range(p):
827.        for bit in range(genesPerCh):
828.          if ch[bit]==1 and int(clusMark[bit])==cList[i]:
829.            cArea[i] += float(areaList[bit])
830.        if cArea[i] < areaCon:
831.          cSList.append(int(cList[i]))
832.
833.      while len(cSList)>0:
834.        cs = cSList[0]
835.        if sum(ch) != genesPerCh:
836.          fixAreaClus(ch,probaPatches,cs)
837.          del cSList[0]
838.        else:
839.          return
840.      fixAreaTotal(ch,probaPatches)
841.
842.    def fixAreaClus(ch,probaPatches,cn):
843.      listArea={}
844.      listA_sort={}
845.      cArea = 0.0
846.      for bit in range(genesPerCh):
847.        if probaPatches[bit] > 0 and ch[bit]==0 and clusMark[bit]==int(cn):
848.          key = str(bit)
849.          #listArea[key] = float(probaPatches[bit])
850.          listArea[key] = float(-distMask3[bit])
851.        if ch[bit]==1 and clusMark[bit]==int(cn):
852.          cArea += areaList[bit]
853.      if len(listArea) == 0:
854.        relaxChro(ch)
855.        return ch
856.
857.      listA_sort = sorted(listArea.items(),key=itemgetter(1),reverse=True)
858.
859.      while len(listA_sort)>0 and cArea<areaCon:
```

```
860.         pId = int(listA_sort[0][0])
861.         addpId = True
862.         for bit in range(genesPerCh):
863.           if bit!=pId and ch[bit] == 1 and clusMark[bit]!=clusMark[pId] and distMatrixM
    in[bit][pId] < distThreshold:
864.             addpId = False
865.             break
866.         if addpId == True:
867.           ch[pId] = 1
868.           cArea += float(areaList[pId])
869.         listA_sort.pop(0)
870.       if cArea < areaCon:
871.         #can't fix this chromosome without violating distance threshold
872.         relaxChro(ch)
873.
874.
875.     """Add this function to prune the area a little bit after i get final solution"""
876.     """Try to identify which patches are removable, then based on their dstance, remove
     from the furthest one"""
877.     """Add some randomness in this function"""
878.     """Before running this function, we need to make sure the area fits the constraint
    first"""
879.     def pruneArea(ch,a, probList):
880.       if sum(ch) == genesPerCh:
881.         return ch
882.       a = random.randint(0,2)
883.       # calculate the total area and sort the area of all the chromos
884.       area = 0.0
885.       areaT = 0.0
886.       cArea = [0.0]*p
887.       minP = [0.0]*p # to store the patch id of the smallest patch in each cluster, cou
    ld be potentially removed
888.       listDrop={} #store the potential removable patches
889.       listDrop_sort={} #store the patches by distance
890.       #for each cluster, calculate area first
891.       for i in range(p):
892.         for bit in range(genesPerCh):
893.           if ch[bit]==1 and clusMark[bit]==cList[i]:
894.             cArea[i] += float(areaList[bit])
895.       areaT = sum(cArea)
896.
897.       # Then in each cluster, identify the potential removable patches
898.       for i in range(p):
899.         listArea={} #store the area
900.         for bit in range(genesPerCh):
901.           if ch[bit]==1 and clusMark[bit]==cList[i] and probList[bit]<0.9:
902.             key = str(bit)
903.             listArea[key] = float(areaList[bit])
904.         # sort the area
905.         listA_sort = sorted(listArea.items(),key=itemgetter(1),reverse= True)
906.         #if len(listArea)==0:
907.           #print "No removable patches for cluster "+str(i)
908.         #print listA_sort
909.         for items in listA_sort:
910.           pArea = float(items[1])
911.           if areaT - pArea >= float(areaTCon) and cArea[i] - pArea >= float(areaCon):
912.             pIdx = int(items[0])
913.             if a==0:
914.             #   #using area
915.               listDrop[str(pIdx)]=float(-distMask2[pIdx])
916.               #listDrop[str(pIdx)]=float(areaList[pIdx])
```

```
917.              #listDrop[str(pIdx)] = float(areaList[pIdx]/distMask3[pIdx])
918.            else:
919.              listDrop[str(pIdx)]=float(areaList[pIdx])
920.              #listDrop[str(pIdx)]=float(-distMask3[pIdx])
921.              #listDrop[str(pIdx)]=float(probList[pIdx])
922.          else:
923.            # Don't need to check further, no patch can be removed
924.            break
925.      if len(listDrop) == 0:
926.        #print "no need to prune"
927.        return ch
928.
929.      #by area or probability, ascending
930.      #by distance, descending
931.      listDrop_sort = sorted(listDrop.items(),key=itemgetter(1))
932.
933.      # Check the whole list of removable patches
934.      while len(listDrop_sort) > 0:
935.        pId = int(listDrop_sort[0][0])
936.        pArea = areaList[pId]
937.        clusId = clusMark[pId]
938.        delpId = True #whether we can remove this patch
939.        if areaT - pArea < float(areaTCon):
940.          delpId = False
941.        else:
942.          for i in range(p):
943.            if cList[i] == clusId:
944.              if cArea[i] - pArea < float(areaCon):
945.                delpId = False
946.              break
947.        if delpId == True:
948.          ch[pId] = 0
949.          areaT = areaT - pArea
950.          cArea[i] = cArea[i] - pArea
951.        # Drop the option after we check it
952.        listDrop_sort.pop(0)
953.
954.      #print ch
955.      if sum(ch)==0:
956.        print "check11"
957.        #print areaT
958.      return ch
959.
960.    """This function is to check whether solution reach the area requirements"""
961.    """In this version, we add also add check for the cluster, if all clusters and the
    total area meet, return 100, either, if only total area doesn't meet, return99"""
962.    """Else, return the clusters, 0:the first one; 1:the second one.... each time use t
    his function can only return one cluster id"""
963.    def checkArea(ch):
964.      totalArea = 0.0
965.      cArea=[0.0]*p
966.      for i in range(p):
967.        for bit in range(genesPerCh):
968.          if ch[bit]==1 and clusMark[bit]==cList[i]:
969.            cArea[i] += float(areaList[bit])
970.      totalArea = sum(cArea)
971.
972.      for i in range(p):
973.        if cArea[i] < areaCon:
974.          return cList[i]
975.
```

```
      if totalArea < areaTCon:
        return "C99"
      else:
        return "C100"

    """This function is to calculate the distance between two points"""
    def distPatch(x1,y1,x2,y2):
      return math.pow(math.sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2)),1)

    """Define a cluster class"""
    class Clus:
      def __init__(self):
        self.cNum = 0
        self.XList = list()
        self.YList = list()
        self.cenX = 0.0
        self.cenY = 0.0
        self.idList = list()


    """This function evaluates the sum distance from every patch centroid to cluster centroid"""
    """The lower value, the better; which means more compact and less number of patches """
    """Instead of using the predefined cluster centroid, i recalculate the centroid points"""
    def evaluate(chromo):
            clusterList = [0]*genesPerCh
            distList = [0.0]*genesPerCh #to get the update of the distance to centroid

            for i in range(genesPerCh):
              clusterList[i] = clusMark[i]*chromo[i] #it has to be selected first
            #f4.write(str(clusterList)+"\n")

            output = 10000.0
            clusObjList = {} # to store all the cluster objects
            #identify the clusters
            for i in range(genesPerCh):
              k = clusterList[i]
              if k != 0:
                key = str(k)
                if key not in clusObjList:
                  clusObj = Clus()
                  clusObj.cNum = k
                  clusObj.XList.append(float(cXList[i]))
                  clusObj.YList.append(float(cYList[i]))
                  clusObj.idList.append(i)
                  clusObjList[key] = clusObj
                else:
                  #only need to update the x, y list
                  clusObj = clusObjList[key]
                  clusObj.XList.append(float(cXList[i]))
                  clusObj.YList.append(float(cYList[i]))
                  clusObj.idList.append(i)
            #Update the cluster centroid and calculate the distances of the other points to the centroid points
            for a,b in clusObjList.items():
              b.cenX = sum(b.XList)/float(len(b.XList))
              b.cenY = sum(b.YList)/float(len(b.YList))
              for i in b.idList:
                #update the distance
```

```
1032.                 distList[i] = distPatch(float(cXList[i]),float(cYList[i]),b.cenX,b.cen
    Y)
1033.             for i in range(genesPerCh):
1034.               output -
    = chromo[i]*distList[i] #Here it doesn't matter which cluster the patch belongs to, add
     them all together
1035.             return output
1036.
1037.           """This function evaluates the sum distance from every patch centroid to clu
    ster centroid"""
1038.           """The lower value, the better; which means more compact and less number of
    patches"""
1039.           """Use the predefined cluster centroid"""
1040.           """Include two objective function"""
1041.           def evaluate2(chromo):
1042.             F1=0.0 #distance
1043.             F2=0.0 #weights
1044.             #output = 10000.0
1045.             #identify the clusters
1046.             for i in range(genesPerCh):
1047.               #cN = int(clusMark[i])
1048.               #dist = distPatch(float(cluXList[cN]),float(cluYList[cN]),float(cXList[i
    ]),float(cYList[i]))
1049.
1050.               if (distMask2[i],1) <0.000001:
1051.                   F1 += math.pow(0.0001,1) * chromo[i]
1052.               else:
1053.                   F1 += math.pow(distMask2[i],1) * chromo[i]
1054.
1055.               F2 += weiList[i]*chromo[i]
1056.               #output -
    = chromo[i]*dist #Here it doesn't matter which cluster the patch belongs to, add them a
    ll together
1057.               #output += chromo[i]*distMask[i]
1058.
1059.
1060.
1061.
1062.
1063.             out = (1-alpha)*F2 - alpha*F1 #maximize this
1064.             output = [F1,F2,out]
1065.             return output
1066.
1067.           """Based on the objective function, select the best chromosome"""
1068.           """Not using this function"""
1069.           def topChrom (chromos,iteration):
1070.             outputs, F1, F2, errors = [], [], [], []
1071.             i = 1
1072.             for chromo in chromos:
1073.               #f4.write(str(iteration)+"\t")
1074.               output = evaluate2(chromo)
1075.               F1.append(output[0])
1076.               F2.append(output[1])
1077.               outputs.append(output[2])
1078.               error = target-output[2]
1079.               #except ZeroDivisionError:
1080.               if error <= 0:
1081.                 global solution_found
1082.                 solution_found = True
1083.                 error = 0
1084.                 #print '\nSOLUTION FOUND'
```

```
1085.                 #print '%s: \t%s=%s' %(str(i).rjust(5), chromo, str(output).rjust(10))

1086.                 break
1087.               else:
1088.                 #error = 1/math.fabs(target-output)
1089.                 errors.append(error)
1090.
1091.               i+=1
1092.             fitnessScores = calcFitness (errors) # calc fitness scores from the errors
     calculated
1093.             pairedPop = zip ( chromos, outputs, F1, F2, fitnessScores) # pair each chr
    omo with its ouput and fitness score
1094.             rankedPop = sorted (pairedPop,key = itemgetter(1),reverse=True) # sort the
     paired pop by descending fitness score
1095.             topChrom = rankedPop[0]
1096.             #print "top"+str(rankedPop[0][1])+str(rankedPop[0][2])
1097.             return topChrom
1098.
1099.           """Calulates fitness as a fraction of the total fitness"""
1100.           """Since there are negative values, I need to take this into consideration""
    "
1101.           def calcFitness (errors):
1102.             fitnessScores = []
1103.             totalError = sum(errors)
1104.             i = 0
1105.             # fitness scores are a fraction of the total error
1106.             for error in errors:
1107.               fitnessScores.append (float(errors[i])/float(totalError))
1108.               i += 1
1109.             return fitnessScores
1110.
1111.           """Calulates fitness as a fraction of the total fitness"""
1112.           """Use model performance as the input"""
1113.           def calcFitness2 (outputs):
1114.             fitnessScores = []
1115.             totalError = sum(outputs)
1116.             i = 0
1117.             # fitness scores are a fraction of the total error
1118.             for output in outputs:
1119.               fitnessScores.append (float(output)/float(totalError))
1120.               i += 1
1121.             return fitnessScores
1122.
1123.           def displayFit (error):
1124.             bestFitDisplay = 100
1125.             dashesN = int(error * bestFitDisplay)
1126.             dashes = ''
1127.             for j in range(bestFitDisplay-dashesN):
1128.               dashes+=' '
1129.             for i in range(dashesN):
1130.               dashes+='+'
1131.             return dashes
1132.
1133.           """Takes a population of chromosomes and returns a list of tuples where each
     chromo is paired to its fitness scores and ranked accroding to its fitness"""
1134.           def rankPop (chromos,iteration):
1135.             outputs, F1, F2 = [], [], []
1136.             i = 1
1137.             for chromo in chromos:
1138.               if sum(chromo)==genesPerCh:
1139.                 #not a feasible solution
```

```
1140.                 output = -9999999999999999999
1141.                 F1.append(-999)
1142.                 F2.append(-999)
1143.                 outputs.append(output)
1144.               else:
1145.                 out = evaluate2(chromo)
1146.                 F1.append(out[0])
1147.                 F2.append(out[1])
1148.                 outputs.append(out[2])
1149.
1150.             fitnessScores = calcFitness2 (outputs) # calc fitness scores from the erro
    s calculated
1151.             #print "fitnessScore = "
1152.             #print fitnessScores
1153.             pairedPop = zip ( chromos, outputs, F1, F2, fitnessScores) # pair each chr
    omo with its ouput and fitness score
1154.             #print "pairedPop="
1155.             #for k in range(0,len(pairedPop),1):
1156.             #  print pairedPop[k][1:]
1157.
1158.           #  print pairedPop
1159.             rankedPop = sorted ( pairedPop,key = itemgetter(1),reverse=True ) # sort t
    he paired pop by descending fitness score
1160.           #  print "rankedPop="
1161.           #  for k in range(0,len(rankedPop),1):
1162.           #    print rankedPop[k][1:]
1163.
1164.             #print rankedPop
1165.             return rankedPop
1166.
1167.           """ taking a ranked population selects two of the fittest members using roul
    ette method"""
1168.           """In this way, the better solution has higher possibility to be chosen"""
1169.           def selectFittest (fitnessScores, rankedChromos):
1170.             #print fitnessScores
1171.             count = 0
1172.             while count < 5: # ensure that the chromosomes selected for breeding are h
    ave different indexes in the population
1173.               index1 = roulette (fitnessScores)
1174.               index2 = roulette (fitnessScores)
1175.               if index1 == index2:
1176.                 count +=1
1177.                 continue
1178.               else:
1179.                 break
1180.             if index1 == index2 and index1-1>-1:
1181.               index2 = index1 - 1
1182.             elif index1 == index2 and index1+1<len(fitnessScores):
1183.               index2 = index1+1
1184.
1185.             ch1 = rankedChromos[index1] # select and return chromosomes for breeding
1186.             ch2 = rankedChromos[index2]
1187.             if len(ch1)>genesPerCh:
1188.               print len(ch1)
1189.               #print index1
1190.             return ch1, ch2
1191.
1192.           """Fitness scores are fractions, their sum = 1. Fitter chromosomes have a la
    rger fraction.  """
1193.           def roulette (fitnessScores):
1194.             index = 0
```

```
1195.             cumalativeFitness = 0.0
1196.             r = random.random()
1197.
1198.             for i in range(len(fitnessScores)): # for each chromosome's fitness score

1199.               cumalativeFitness += fitnessScores[i] # add each chromosome's fitness sc
    ore to cumalative fitness
1200.               if cumalativeFitness > r: # in the event of cumalative fitness becoming
    greater than r, return index of that chromo
1201.                 #print cumalativeFitness
1202.                 #print r
1203.                 return i
1204.
1205.           """Modify this function to enable the crossover for specific cluster"""
1206.           def crossover (ch1, ch2):
1207.             # at a random chiasma
1208.             r = random.randint(1,genesPerCh-1)
1209.             ch3 = ch1[:r]+ch2[r:]
1210.             ch4 = ch2[:r]+ch1[r:]
1211.             #print len(ch2)
1212.             #print len(ch4)
1213.             return ch3, ch4
1214.
1215.           """In this version, the mutate function is to randomly change """
1216.           """Modify this function to give the higher probability patch less likely to
    be changed"""
1217.           def mutate (ch,probList):
1218.             # if len(ch)>genesPerCh:
1219.             #   print "mutate ch problem!"
1220.             global mutation_rate
1221.             mutatedCh = []
1222.             for i in range(genesPerCh):
1223.               #print i
1224.               if probList[i] > 0.8 or probList[i] < 0.2:
1225.                 mutation_rate = mutation_rate * 0.5
1226.
1227.               if random.random() < mutation_rate:
1228.                 mutatedCh.append(1-ch[i])
1229.               else:
1230.                 mutatedCh.append(ch[i])
1231.
1232.             #assert mutatedCh != ch
1233.             #after mutate, we need to make sure the total area is still under control

1234.             #also the distance thing is under control too
1235.             mutatedCh2 = checkChro(mutatedCh,probList)
1236.             # if sum(mutatedCh2) == 0:
1237.             #   print "check2"
1238.             #   relaxChro(mutatedCh2)
1239.             return mutatedCh2
1240.
1241.           """Using breed and mutate it generates two new chromos from the selected pai
    r"""
1242.           def breed (ch1, ch2,probList):
1243.             newCh1, newCh2 = [], []
1244.             if random.random() < crossover_rate: # rate dependent crossover of selecte
    d chromosomes
1245.               newCh1, newCh2 = crossover(ch1, ch2)
1246.             else:
1247.               newCh1, newCh2 = ch1, ch2
1248.             #random select one to mutate
```

```
1249.             if random.randint(0,1) == 0:
1250.               newnewCh1 = mutate (newCh1,probList) # mutate crossovered chromos
1251.               # if sum(newnewCh1)==0:
1252.               #   print "check7"
1253.               return newnewCh1
1254.             else:
1255.               newnewCh2 = mutate (newCh2,probList)
1256.               # if sum(newnewCh2)==0:
1257.               #   print "check7"
1258.               return newnewCh2
1259.
1260.           """ Taking a ranked population return a new population by breeding the ranke
    d one"""
1261.           def iteratePop (rankedPop,probList):
1262.             fitnessScores = [ item[-
    1] for item in rankedPop ] # extract fitness scores from ranked population
1263.             rankedChromos = [ item[0] for item in rankedPop ] # extract chromosomes fr
    om ranked population
1264.             #print '%s'%(rankedChromos)
1265.             newpop = []
1266.             #newpop.extend(rankedChromos[:popN/15]) # known as elitism, conserve the b
    est solutions to new population
1267.             #Reserve the best two chromosomes
1268.             for aa in range(bestN):
1269.               newpop.append(rankedChromos[aa])
1270.
1271.             a = bestN
1272.             # Add one chromosome with random change
1273.             if len(newpop) != popN:
1274.               ch1 = rankedChromos[a]
1275.               ch2 = mutate(ch1,probList)
1276.               newpop.append(ch2)
1277.             while len(newpop) != popN:
1278.               ch1, ch2 = [], []
1279.               ch1, ch2 = selectFittest (fitnessScores, rankedChromos) # select two of
    the fittest chromos
1280.               ch3 = breed (ch1, ch2,probList) # breed them to create one new chromosom
    e
1281.               if sum(ch3)==0:
1282.                 print "check6"
1283.               if sum(ch3)>genesPerCh:
1284.                 print "problem for breed"
1285.               newpop.append(ch3) # and append to new population
1286.             return newpop
1287.
1288.           def configureSettings ():
1289.             configure = raw_input ('T - Enter Target Number \tD - Default settings: ')

1290.             match1 = re.search( 't',configure, re.IGNORECASE )
1291.             if match1:
1292.               global target
1293.               target = input('Target int: ' )
1294.
1295.           def main():
1296.             klt = 0
1297.             K = 3
1298.             A = 0.5
1299.             al = .8
1300.       #      al = float(sys.argv[1])
1301.             S = 10000
1302.             print "alpha = " + str(al)
```

```
1303.             print "Area = " + str(A)
1304.             print "Separation distance = " + str(S)
1305.             F1 = 0
1306.             F2 = 0
1307.             global areaConP
1308.             areaConP = float(A)/float(K)
1309.       #      solutionPool = open("S:\\CLAS\\GEES\\LABS\\MedicalGeography\\ericstuff\\CnewsolutionPoolNew_S_" + str(S) +"_A_" + str(int(100*A))+"_alpha_" + str(al) + "_.txt", "a")
1310.             solutionPool = open("C:\\GIS\\EnewsolutionPoolNew_S_" + str(S) +"_A_" + str(int(100*A))+"_alpha_" + str(al) + "_.txt", "a")
1311.
1312.
1313.             #Just modify these three parameters
1314.             global distThreshold
1315.             distThreshold = int(S)
1316.             global p
1317.             p = int(K)
1318.             global areaTConP
1319.             areaTConP = float(A)
1320.             global alpha
1321.             alpha = float(al)
1322.             global bestOutputFile
1323.             bestOutputFile = "C:\\GIS\\landconservation\\work\\simulateData100poly\\bestChroms"+str(alpha)+".txt"
1324.             beginTime = datetime.datetime.now()
1325.             init ()
1326.             f2 = open(bestOutputFile,'w')
1327.             print >> f2, 'Iteration\tChromosome\tPerformance\tF1\tF2\tNumber of new\tTotalArea of new\tNumber of all\tTotalArea of all'
1328.
1329.             iterations = 0
1330.             outStr = "" #the output result
1331.             out = 0.0 #the performance
1332.             global noClusSet
1333.       #      while iterations != max_iterations and solution_found != True and len(noClusSet)<Ccombs:
1334.             while iterations < max_iterations:
1335.               #print "outer loop iterations = "+str(iterations)
1336.
1337.               #print "aa" + str(iterations)
1338.               iteration2 = 0 # the loop for local improvement
1339.               global cList
1340.               global cpList
1341.               global opVal
1342.
1343.           #    print '\nCurrent iterations:', iterations+1
1344.               if iterations == 0:
1345.                 cList = selCluster(clusterNum,p)
1346.
1347.               #global probList
1348.               probList = setPatchesProb(clusMark,distMask3,cList)
1349.           #    print "cList="
1350.           #    print cList
1351.               #print "clusMark="
1352.               #print clusMark
1353.               #print "probList="
1354.               #print probList
1355.               chromos = generatePop2(probList,popN)
1356.               #print "chromos="
1357.               #print chromos
```

```
1358.
1359.               while iteration2 < max_iterations2:
1360.       #          print "inner loop iterations = "+str(iteration2)
1361.                 rankedPop = rankPop(chromos,iteration2)
1362.                 topPop = rankedPop[0]
1363.                 # we need to consider the situation that all the solutions are the sam
    e, that means they all don't fit the constraints
1364.                 if sum(topPop[0]) == 0:
1365.                   print "something is wrong"
1366.                 elif sum(topPop[0]) == genesPerCh:
1367.                   #No need to iterate
1368.                   iteration2 = max_iterations2
1369.                 elif sum(topPop[0]) > genesPerCh:
1370.                   print "problem rankedpop"
1371.                 else:
1372.                   chromos = iteratePop(rankedPop,probList)
1373.                   iteration2 += 1
1374.                 if topPop[1] > -99:
1375.                   print str(klt) + "\t" + str(iterations) +"_"+ str(iteration2) + "\t"
     + str(-
    float(topPop[1])) + "\t" + str(float(topPop[2])) + "\t" + str(float(topPop[3]))
1376.                   fOut.write(str(klt) + "\t" + str(-float(topPop[1])) +"\n")
1377.                   klt +=1
1378.
1379.               # After certain number of iterations, get the best one
1380.               rankedPop = rankPop(chromos,iterations+1)
1381.               topPop = rankedPop[0]
1382.               #print "Below is the best one: solution, output, fitness"
1383.               #print out the best solution
1384.               solution=[] #the final cluster result
1385.               areaT = 0 #The total area of selected patches
1386.               areaTN = 0 # The total area of newly selected patches
1387.               countP = 0 # The number of patches that are selected
1388.               countPN = 0 # The total number of newly selected patches
1389.               for aa in range(genesPerCh):
1390.                 solution.append(topPop[0][aa]*clusMark[aa])
1391.                 if topPop[0][aa]==1:
1392.                   areaT += float(areaList[aa])
1393.                   countP += 1
1394.                   if int(indexList[aa]) != 1:
1395.                     areaTN += float(areaList[aa])
1396.                     countPN += 1
1397.               #Print to screen
1398.               if topPop[1] > -999:
1399.           #        print "here we go"
1400.       #            print str(iterations) + "\t" + str(-
    float(topPop[0])) + "\t" + str(-float(topPop[1])) + "\t" + str(-float(topPop[2]))
1401.                   fOut.write(str(iterations) + "\t" + str(-float(topPop[1])) +"\n")
1402.
1403.           #    print str(solution)+"\t"+str(topPop[1])+"\t"+str(topPop[2])
1404.               if iterations!=0:
1405.                 if float(topPop[1]) > opVal:
1406.                   #print "Cluster selection is better than last one!"
1407.                   #found a better one, swap the cluster solution based on this one
1408.                   opVal = float(topPop[1])
1409.                   F1 = float(topPop[2])
1410.                   F2 = float(topPop[3])
1411.                   cpList = list(cList)
1412.                   #noClusSet.add(tuple(cList))
1413.                   if len(noClusSet)<TabuLength:
1414.                     noClusSet.append(tuple(cList))
```

```
                    else:
                      del noClusSet[0]
                      noClusSet.append(tuple(cList))
                    cList = swapCluster(cList,clusterNum,p)

                    # only print it out when it's improved
                    print >> f2, '%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s' %(iterations+1,sol
    ution,-topPop[1],topPop[2],topPop[3],countPN,areaTN,countP,areaT)
                    #print solution
                    #print distMask2
                    #iterations = iterations+1
                    #out = 10000 - opVal
                    out = -opVal
                  else:
                    #print "Cluster selection is worse than last one!"
                    #If it's not better, go back to the old list, and swap based on that
     one
                    cList = swapCluster(cpList,clusterNum,p)
                    #Put this one into the tabu one
                    #noClusSet.add(tuple(cList))
                    if len(noClusSet)<10:
                      noClusSet.append(tuple(cList))
                    else:
                      del noClusSet[0]
                      noClusSet.append(tuple(cList))

            #         print "The length of the no cluster set is: "+ str(len(noClusSet))

                else:
                  opVal = float(topPop[1])
                  F1 = float(topPop[2])
                  F2 = float(topPop[3])
                  cpList = list(cList)
                  print >> f2, '%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s' %(iterations+1,solut
    ion,-topPop[1],topPop[2],topPop[3],countPN,areaTN,countP,areaT)
                  iterations = iterations+1
                iterations += 1
              if out > -999:
                solutionSet = set(solution)
                counter = 0
                for x in solutionSet:
                  if int(x) !=0:
                      solutionPool.write(str(x))
                      if counter < K:
                          solutionPool.write(",")
                          counter+=1

                solutionPool.write(str(al) + "\t" + str(out)+"\t"+str(F1)+"\t"+str(F2))


                solutionPool.write("\n")
                solutionPool.close()
              f2.close()

              currentTime = datetime.datetime.now()
              duration = (currentTime - beginTime).total_seconds()
              outStr = str(S)+"\t"+str(K)+"\t"+str(A)+"\t"+str(out)+"\t"+str(F1)+"\t"+st
    r(F2)+"\t"+str(duration)
              print outStr
            if __name__ == "__main__":
                main()
```

```
1470.
1471.
1472.           fOut.close()
1473.           solutionPoolIterations+=1
```