# -\*- coding: utf-8 -\*-

"""

Created on Sun Jan 23 18:55:59 2022

@author: Blake Splitter with assistance from Amy Burton and Dr. Matthew Saltzman

"""

from decimal import DivisionByZero

from multiprocessing.sharedctypes import Value

import sys

import os

import arcpy

import datetime

import random

from inspect import signature

import GraphMeasures

import networkx as nx

import numpy as np

import math

class input\_vals:

    def \_\_init\_\_(self, in\_t, in\_p\_f, in\_n\_f, in\_c\_f, in\_vb\_f, in\_vr\_f, dc,

                   nr, t, fin\_t, tol, msc, pp, nf):

        self.in\_table = in\_t

        self.in\_pop\_field = in\_p\_f

        self.in\_name\_field = in\_n\_f

        self.in\_county\_field = in\_c\_f

        self.in\_voteblue\_field = in\_vb\_f

        self.in\_votered\_field = in\_vr\_f

        self.distcount = dc

        self.num\_recoms = nr

        self.temp = t

        self.final\_t = fin\_t

        self.tol = tol

        self.maxstopcounter = msc

        self.pop\_perc = pp

        self.num\_flips = nf

    def default\_user\_input(self):

        self.in\_table = PATH + "\\SC\_Precincts\_2021\_v7"

        self.in\_pop\_field = "POPULATION"

        self.in\_name\_field = "OBJECTID"

        self.in\_county\_field = "COUNTY"

        self.in\_voteblue\_field = "PresBlue"

        self.in\_votered\_field = "PresRed"

        self.distcount = 7

        self.num\_recoms = 10

        self.temp = 20

        self.final\_t = 0.1

        self.tol = 30

        self.max\_stop\_counter = 50

        self.pop\_perc = 15

        self.num\_flips = 5

    @property

    def coolingrate(self):

        """Calculates the necessary cooling rate to get from initial temperature to

        final temperature in the requested number of recom steps"""

        return (self.final\_t / self.temp) \*\* (1 / self.num\_recoms)

    @property

    def total\_obj\_vals\_entries(self):

        """Returns the number of objective value entries needed"""

        return (self.num\_recoms \* (self.num\_flips + 1) + 1)

class objective\_vals:

    def \_\_init\_\_(self, ip):

        self.dev\_vals = [0] \* ip.total\_obj\_vals\_entries

        self.avg\_comp\_vals = [0] \* ip.total\_obj\_vals\_entries

        self.fairscore\_vals = [0] \* ip.total\_obj\_vals\_entries

        self.CDI\_Count\_vals = [0] \* ip.total\_obj\_vals\_entries

        self.excess\_GU\_vals = [0] \* ip.total\_obj\_vals\_entries

        self.change\_type\_vals = [None] \* ip.total\_obj\_vals\_entries

    def fill\_obj\_vals(self, dev, comp, fair, cdi\_data, change, it):

        self.dev\_vals[it] = dev

        self.avg\_comp\_vals[it] = comp

        self.fairscore\_vals[it] = fair

        self.CDI\_Count\_vals[it] = cdi\_data.cdi\_count

        self.excess\_GU\_vals[it] = cdi\_data.excess\_GU

        self.change\_type\_vals[it] = change

    def \_\_repr\_\_(self):

        last\_it = np.max(np.nonzero(self.dev\_vals))

        dev = self.dev\_vals[last\_it]

        fair = self.fairscore\_vals[last\_it]

        cdi = self.CDI\_Count\_vals[last\_it]

        eGU = self.excess\_GU\_vals[last\_it]

        ct = self.change\_type\_vals[last\_it]

        return "{0} iterations filled. Last entry: deviation = {1}, fairness = {2}, cdi\_count = {3}, excess\_GU = {4}, change\_type = {5}".format(last\_it, dev, fair, cdi, eGU, ct)

class District:

    """A class that will hold the statistics associated with each district"""

    ideal\_pop = 0  #The ideal population that a district would like to be

    num\_dists = 0  #The number of districts

    def \_\_init\_\_(self, num):

        self.num = num  #District numbers will range from 1 to n

        self.Area = None  #Area in units of square kilometers

        self.Perimeter = None  #Perimeter in units of kilometers

        self.HypArea = None  #Hypothetical area after a proposed change

        self.HypPerimeter = None  #Hypothetical perimeter after a proposed change

        self.VoteCountRed = None  #Red votes in a district

        self.VoteCountBlue = None  #Blue votes in a district

        self.HypVoteCountRed = None  #Hypothetical red vote count after a proposed change

        self.HypVoteCountBlue = None  #Hypothetical blue vote count after a proposed change

        self.Population = None  #Population of the district

        self.Dist\_nbrs = []  #Set of district neighbors

    def \_\_repr\_\_(self):

        return "District Number {0}. Population: {1}. Area: {2} km^2. Perimeter: {3} km. PP Score: {4}. Ideal population target: {5}.".format(self.num, self.Population, self.Area, self.Perimeter, self.PPCompactScore, self.ideal\_pop)

    @property

    def PPCompactScore(self):

        '''Polsby-Popper Compactness Score (ranges from 0 to 1; 1 is best)'''

        try:

            return 4 \* math.pi \* self.Area / self.Perimeter \*\* 2 if self.Perimeter != 0 else None

        except TypeError:

            return None

    @property

    def invPPCompactScore(self):

        '''Returns the inverse of the Polsby-Popper Compactness Score and subtracts 1 so that the ideal score is 0'''

        return (1 / self.PPCompactScore) - 1 if self.PPCompactScore != 0 and self.PPCompactScore != None else None

    @property

    def HypPPCompactScore(self):

        '''Hypothetical Polsby-Popper Compactness after a proposed change'''

        try:

            return 4 \* math.pi \* self.HypArea / self.HypPerimeter \*\* 2 if self.HypPerimeter != 0 else None

        except TypeError:

            return None

    @property

    def TotalVotes(self):

        '''Sums red votes and blue votes for a district'''

        try:

            return self.VoteCountBlue + self.VoteCountRed

        except TypeError:

            return None

    @property

    def HypTotalVotes(self):

        '''Sums hypothetical red votes and hypothetical blue votes for a district'''

        try:

            return self.HypVoteCountBlue + self.HypVoteCountRed

        except TypeError:

            return None

    @property

    def BlueShare(self):

        '''Returns the share of blue votes as a proportion of total votes'''

        try:

            return self.VoteCountBlue / self.TotalVotes if self.TotalVotes != 0 else None

        except TypeError:

            return None

    @property

    def HypBlueShare(self):

        '''Returns the share of hypothetical blue votes as a proportion of the hypothetical total votes'''

        try:

            return self.HypVoteCountBlue / self.HypTotalVotes if self.HypTotalVotes != 0 else None

        except TypeError:

            return None

    @property

    def EfficiencyGap(self):

        '''Returns the Efficiency gap as calculated by (wastedRed votes - wastedBlue votes) / total votes'''

        try:

            return (self.WastedRed - self.WastedBlue) / self.TotalVotes if self.TotalVotes != 0 else None

        except TypeError:

            return None

    @property

    def AbsEfficiencyGap(self):

        '''Returns the absolute value of the efficiency gap'''

        try:

            return abs(self.EfficiencyGap)

        except TypeError:

            return None

    @property

    def HypEfficiencyGap(self):

        '''Returns the Hypothetical Efficiency gap as calculated by (wastedRed votes - wastedBlue votes) / total votes'''

        try:

            return (self.HypWastedRed - self.HypWastedBlue) / self.HypTotalVotes if self.HypTotalVotes != 0 else None

        except TypeError:

            return None

    @property

    def WinThreshold(self):

        '''Returns the number of votes needed to win an election in a district'''

        try:

            return math.ceil(self.TotalVotes / 2 + 0.5)  #The '+0.5' is needed to deal with cases where the total number of votes is even

        except TypeError:

            return None

    @property

    def HypWinThreshold(self):

        '''Returns the number of hypothetical votes needed to win an election in a district'''

        try:

            return math.ceil(self.HypTotalVotes / 2 + 0.5)  #The '+0.5' is needed to deal with cases where the total number of votes is even

        except TypeError:

            return None

    @property

    def WastedBlue(self):

        '''Returns the number of wasted blue votes. If the blue party wins, then this value will be

        the number of blue votes beyond the win threshold. If the blue party loses, then this will be the

        number of blue votes. For coding simplicity, all ties are won by the blue party.'''

        try:

            if self.VoteCountBlue >= self.VoteCountRed:

                return self.VoteCountBlue - self.WinThreshold

            else:

                return self.VoteCountBlue

        except TypeError:

            return None

    @property

    def WastedRed(self):

        '''Returns the number of wasted red votes. If the red party wins, then this value will be

        the number of red votes beyond the win threshold. If the red party loses, then this will be the

        number of red votes. For coding simplicity, all ties are won by the blue party.'''

        try:

            if self.VoteCountRed >= self.VoteCountBlue:

                return self.VoteCountRed - self.WinThreshold

            else:

                return self.VoteCountRed

        except TypeError:

            return None

    @property

    def HypWastedBlue(self):

        '''Returns the number of wasted blue votes after a proposed change. If the blue party wins, then this value will be

        the number of hypothetical blue votes beyond the win threshold. If the blue party loses, then this will be the

        number of hypothetical blue votes. For coding simplicity, all ties are won by the blue party.'''

        try:

            if self.HypVoteCountBlue >= self.HypVoteCountRed:

                return self.HypVoteCountBlue - self.HypWinThreshold

            else:

                return self.HypVoteCountBlue

        except TypeError:

            return None

    @property

    def HypWastedRed(self):

        '''Returns the number of wasted red votes after a proposed change. If the red party wins, then this value will be

        the number of hypothetical red votes beyond the win threshold. If the red party loses, then this will be the

        number of hypothetical red votes. For coding simplicity, all ties are won by the blue party.'''

        try:

            if self.HypVoteCountRed >= self.HypVoteCountBlue:

                return self.HypVoteCountRed - self.HypWinThreshold

            else:

                return self.HypVoteCountRed

        except TypeError:

            return None

    def UpdateStats(self, a, p, vcr, vcb):

        self.Area = a

        self.Perimeter = p

        self.VoteCountRed = vcr

        self.VoteCountBlue = vcb

    def UpdateCompStats(self, a, p):

        self.Area = a

        self.Perimeter = p

    def UpdateHypStats(self, a, p):

        self.HypArea = a

        self.HypPerimeter = p

    def ConfirmStats(self, status):

        if status == True:

            self.UpdateStats(self.HypArea, self.HypPerimeter, self.HypVoteCountRed, self.HypVoteCountBlue)

        self.HypArea = 0

        self.HypPerimeter = 0

        self.HypVoteCountRed = 0

        self.HypVoteCountBlue = 0

    @staticmethod

    def pop\_list(dist\_list):

        '''Returns a list of populations for each district'''

        pop\_list = [d.Population for d in dist\_list]

        while True:

            try:

                pop\_list.remove(None)  #Removes dummy district

            except ValueError:

                break

        return pop\_list

    @staticmethod

    def EG\_list(dist\_list, AV=False):

        '''Returns a list of efficiency gaps for each district. If AV is true, then we return the absolute values'''

        if AV == False:

            eg\_list = [d.EfficiencyGap for d in dist\_list]

        else:

            eg\_list = [d.AbsEfficiencyGap for d in dist\_list]

        while True:

            try:

                eg\_list.remove(None)  #Removes dummy district

            except ValueError:

                break

        return eg\_list

    def reset\_vals(self):

        '''Resets several values. Useful during recom'''

        self.Area = 0

        self.Perimeter = 0

        self.VoteCountRed = 0

        self.VoteCountBlue = 0

        self.Population = 0

        self.Dist\_nbrs = []

class CDI:

    """A class that will contain all information about the county-district-intersection matrix"""

    def \_\_init\_\_(self, stateG):

        distcount = len(set(dict(stateG.nodes("District Number")).values()))  #Finds number of unique districts

        num\_counties = len(set(dict(stateG.nodes("County Number")).values()))  #Finds number of unique counties

        units\_in\_CDI = np.zeros([distcount, num\_counties], dtype=int)

        # #Adds 1 to the matrix element A[i,j] if there is a precinct in the ith district and jth county

        for n in stateG:

            if stateG.nodes[n]["District Number"] == 0:

                continue

            else:

                units\_in\_CDI[stateG.nodes[n]["District Number"] - 1][stateG.nodes[n]["County Number"] - 1] += 1

        self.cdi\_mat = units\_in\_CDI

        self.distcount = distcount

        self.num\_counties = num\_counties

    @property

    def cdi\_count(self):

        '''Counts number of nonzero entries in the CDI matrix. Then subtracts either the distcount or number of counties, so that the ideal value will be zero.'''

        return np.count\_nonzero(self.cdi\_mat) - max(self.distcount, self.num\_counties)

    @property

    def excess\_GU(self):

        '''GU stands for Geographical Unit. In this loop, we count the number of GUs in each county that are not in the most prevalent district. The ideal number of excess GUs is zero.'''

        excess\_GU\_mat = [0] \* max(self.distcount, self.num\_counties)

        transpose = self.cdi\_mat.transpose()

        idx = 0

        for row in transpose:

            maxval = max(row)

            excess\_GU\_mat[idx] = sum(row) - maxval

            idx += 1

        return sum(excess\_GU\_mat)

    def upd\_cdi\_mat\_flip(self, stateG, GU, leaving\_dist, entering\_dist):

        '''Updates the cdi matrix after a flip'''

        self.cdi\_mat[leaving\_dist - 1][stateG.nodes[GU]["County Number"] - 1] -= 1

        self.cdi\_mat[entering\_dist - 1][stateG.nodes[GU]["County Number"] - 1] += 1

class counters:

    '''A class that will contain the counters used in the simulated annealing step'''

    def \_\_init\_\_(self):

        self.flipcount = 0  #The number of flips done in total in the code

        self.recomcount = 0  #The number of recombination steps done in total in the code

        self.stopcounter = 0  #The number of consecutive failed recombination steps in the current iteration

    @property

    def currentit(self):

        '''Returns the sum of the flip count and the recom count'''

        return self.flipcount + self.recomcount  #This returns the index that will be populated in obj\_vals

    def \_\_repr\_\_(self):

        return "flipcount = {0}. recomcount = {1}. stopcounter = {2}. currentit = {3}".format(self.flipcount, self.recomcount, self.stopcounter, self.currentit)

def arcprint(message, \*variables):

    '''

    Prints a message using arcpy.AddMessage() unless it can't; then it uses

    print.

    '''

    if RUNSPOT == "ArcGIS":

        arcpy.AddMessage(message.format(\*variables))

    elif RUNSPOT == "console":

        newmessage = message

        variables = list(variables)

        j = 0

        while j < len(variables):  #This while loop puts the variable(s) in the correct spot(s) in the string

            if isinstance(variables[j], float):

                variables[j] = round(variables[j], 3)

            newmessage = newmessage.replace("{"+str(j)+"}",str(variables[j]))  #Replaces {i} with the ith variable

            j=j+1

        print(newmessage)

    else:

        raise RuntimeError("No value for RUNSPOT has been assigned")

"""def arcerror(message, \*variables):

    '''

    Prints an error message using arcpy.AddError() unless it can't; then it

    uses print.

    '''

    if RUNSPOT == "ArcGIS":

        arcpy.AddError(message.format(\*variables))

    elif RUNSPOT == "console":

        newmessage = message

        variables = list(variables)

        j = 0

        while j < len(variables):  #This while loop puts the variable(s) in the correct spot(s) in the string

            if isinstance(variables[j], float):

                variables[j] = round(variables[j], 3)

            newmessage = newmessage.replace("{"+str(j)+"}", str(variables[j]))  #Replaces {i} with the ith variable

            j=j+1

        raise RuntimeError(newmessage)

    else:

        raise RuntimeError("No value for RUNSPOT has been assigned")"""

def build\_alpha(metric\_count, num\_maps):

    '''Builds the normalized weight vectors for use in simulated annealing'''

    alpha = [[0] \* metric\_count] \* num\_maps  #cols \* rows

    for i in range(num\_maps):

        for j in range(metric\_count):

            alpha[i][j] = random.randint(1, 1000)

        tot = sum(alpha[i])

        for j in range(metric\_count):

            alpha[i][j] = alpha[i][j]/tot

        eps = 0.000001

        if sum(alpha[i]) > 1 + eps or sum(alpha[i]) < 1 - eps:

            raise ValueError("The elements of alpha must sum to 1. alpha[{0}] = {1}".format(i, alpha[i]))

    return alpha

def initialize\_map(ip, timetxt):

    """Generates an initial map using Spatially Constrained Multivariate Clustering"""

    #Counts number of rows in in\_table

    row\_count = int(arcpy.GetCount\_management(ip.in\_table).getOutput(0))  #getOutput(0) returns the value at the first index position of a tool

    #Creates name for the output map

    out\_table = ip.in\_table + "\_SA" + "\_{0}".format(ip.distcount) + "dists" + timetxt

    #Using Spatially Constrained Multivariate Clustering to create a random starting district

    if not arcpy.ListFields(ip.in\_table, "Test\_val"): #if field does not exist

        arcpy.AddField\_management(ip.in\_table, "Test\_val","LONG",field\_alias="Test\_val")

        arcprint("Adding 'Test\_val' field to in\_table")

    with arcpy.da.UpdateCursor(ip.in\_table, 'Test\_val') as cursor:

        for row in cursor:

            row[0] = random.randint(1,100000)

            cursor.updateRow(row)

        del cursor, row

    arcprint("Running Spatially Constrained Multivariate Clustering to create the initial map...")

    mapflag = False

    failcount = 1

    while mapflag == False:  #We try SCMC to create an initial map. We have 20 attempts before the code gives up

        try:

            arcpy.stats.SpatiallyConstrainedMultivariateClustering(ip.in\_table, out\_table, "Test\_val", size\_constraints="NUM\_FEATURES", min\_constraint=0.65\*row\_count/ip.distcount, number\_of\_clusters=ip.distcount, spatial\_constraints="CONTIGUITY\_EDGES\_ONLY")

            mapflag = True

            new\_row\_count = int(arcpy.GetCount\_management(out\_table).getOutput(0))  #Returns the number of rows in out\_table

            if new\_row\_count != row\_count:

                mapflag = False

                arcprint("SCMC did not keep all GUs from the original map.")

                failcount = SCMC\_restart(failcount, ip)

        except arcpy.ExecuteError:  #Occurs if SCMC cannot create a map with the given constraints

            mapflag = False

            failcount = SCMC\_restart(failcount, ip)

    arcprint("Spatially Constrained Multivariate Clustering succeeded.")

    #Adds populations as a column in out\_table

    arcpy.management.JoinField(out\_table, "SOURCE\_ID", ip.in\_table, ip.in\_name\_field, ip.in\_pop\_field)

    #Adds vote totals as a column in out\_table

    arcpy.management.JoinField(out\_table, "SOURCE\_ID", ip.in\_table, ip.in\_name\_field, ip.in\_voteblue\_field)

    arcpy.management.JoinField(out\_table, "SOURCE\_ID", ip.in\_table, ip.in\_name\_field, ip.in\_votered\_field)

    #Adds county numbers to out\_table

    arcpy.management.JoinField(out\_table, "SOURCE\_ID", ip.in\_table, ip.in\_name\_field, ip.in\_county\_field)

    return out\_table

def SCMC\_restart(failcount, ip):

    """Protocol for restarting SCMC if necessary"""

    arcprint("Attempt number {0} at using Spatially Constrained Multivariate Clustering (SCMC) failed. Trying again.", failcount)

    failcount = failcount + 1

    if failcount >= 20:

        raise RuntimeError("{0} attempts failed to produce a starting map for SCMC.".format(failcount))

    with arcpy.da.UpdateCursor(ip.in\_table, 'Test\_val') as cursor:  #Resets the random values

        for row in cursor:

            row[0] = random.randint(1, 100000)

            cursor.updateRow(row)

        del cursor, row

    return failcount

"""def temp\_dist(out\_table):

    '''Creates a column named "temp\_dist" and zeros it out. This is used to simplify Zonal Geometry later'''

    if not arcpy.ListFields(out\_table["name"], "temp\_dist"):

        arcpy.AddField\_management(out\_table["name"], "temp\_dist", "SHORT", field\_alias="Temporary District")

    with arcpy.da.UpdateCursor(out\_table["name"], "temp\_dist") as cursor:

        for row in cursor:

            row[0] = 0

            cursor.updateRow(row)

        del cursor"""

def make\_county\_dict(ip, out\_table):

    '''Populates county\_dict with 1-x, based on the sorted county numbers'''

    county\_list = [row.getValue (ip.in\_county\_field) for row in arcpy.SearchCursor (out\_table["name"])]  #Gets original county values

    county\_list = list(map(int, county\_list))  #Converts strings to integers

    county\_list = sorted(list(set(county\_list)))  #Sorts the list and deletes duplicate values

    county\_dict = {}

    i = 1

    for county in county\_list:

        county\_dict[county] = i  #Populates a dictionary that associates each original county value with its sorted value

        i += 1

    return county\_dict

def create\_buffer\_dist(out\_table):

    '''Creates a dummy GU that surrounds the state Assigns this dummy GU a district value of 0.'''

    buffer\_fc = out\_table["name"] + "\_Buffer"

    arcpy.Buffer\_analysis(out\_table["name"], buffer\_fc, "1 Kilometers", line\_side="OUTSIDE\_ONLY", dissolve\_option="ALL")

    arcpy.Append\_management(buffer\_fc, out\_table["name"], schema\_type="NO\_TEST")  #Adds the newly created buffer to the out\_table

    row\_count = int(arcpy.GetCount\_management(out\_table["name"]).getOutput(0)) #getOutput(0) returns the value at the first index position of a tool.

    sql\_statement = """{} = {}""".format(arcpy.AddFieldDelimiters(out\_table["name"], "OBJECTID"), row\_count)

    with arcpy.da.UpdateCursor(out\_table["name"], out\_table["dist\_field"], sql\_statement) as cursor: #Selects only the last row of the table

        for row in cursor:

            row[0] = 0  #Dist\_Assign = 0

            cursor.updateRow(row)

        del cursor, row

    arcpy.management.Delete(buffer\_fc)  #Deletes the buffer feature class

def CreateNeighborList(out\_table):

    '''Creates a neighbor list for out\_table and deletes all single-point adjacencies'''

    neighbor\_list = out\_table["name"] + "\_nbr\_list"

    arcpy.PolygonNeighbors\_analysis(out\_table["name"], neighbor\_list, ["OBJECTID", "CLUSTER\_ID"], None, None, None, "KILOMETERS")

    arcprint("Deleting all rows from neighbor list with single-point adjacencies...")

    with arcpy.da.UpdateCursor(neighbor\_list, ["NODE\_COUNT", "OBJECTID"]) as cursor:

        for row in cursor:

            if row[0] > 0:

                cursor.deleteRow()  #Deletes all rows with that have single-point adjacency

        cursor.reset()  #Resets the cursor back to the first row

        i = 1

        for row in cursor:

            row[1] = i  #Relabels the OBJECTID with its row number (several rows were deleted above)

            i = i + 1

        del cursor

    return neighbor\_list

"""# def find\_dist\_pops(ip, out\_table, dist\_list):

#     '''Finds each district population by summing all geographical units'''

#     sumpop = []

#     sumpop = [0] \* ip.distcount

#     with arcpy.da.SearchCursor(out\_table["name"], [ip.in\_pop\_field, out\_table["dist\_field"]]) as cursor:

#         for row in cursor:

#             #i = district number minus 1, since district numbers range from 1 to distcount

#             i = int(row[1] - 1)

#             sumpop[i] = row[0] + sumpop[i]

#     idealpop = round(sum(sumpop) / ip.distcount)

#     District.ideal\_pop = idealpop

#     for i in range(ip.distcount):

#         dist\_list[i].Population = sumpop[i]

#     arcprint("The starting population of each district (i.e. sumpop) is {0}. Thus, the ideal population for a district is {1}.", sumpop, idealpop)"""

def update\_nbr\_list(neighbor\_list):

    '''Initializes neighbor\_list so that each entry in src\_dist and nbr\_dist is reset to match original districts'''

    if not arcpy.ListFields(neighbor\_list, "src\_dist"):  #Adds src\_dist and nbr\_dist to neighbor\_list if they don't already exist. These fields will be the ones that change mid-algorithm

        arcpy.AddField\_management(neighbor\_list, "src\_dist", "SHORT", field\_alias="Source District")

        arcpy.AddField\_management(neighbor\_list, "nbr\_dist", "SHORT", field\_alias="Neighbor District")

    orig\_dist\_names = []

    lstFields = arcpy.ListFields(neighbor\_list)

    for field in lstFields:

        if field.name in ["src\_CLUSTER\_ID", "src\_ZONE\_ID", "nbr\_CLUSTER\_ID", "nbr\_ZONE\_ID"]:

            orig\_dist\_names.append(field.name)

    odn = orig\_dist\_names  #An alias

    #Copies all original district numbers into src\_dist and nbr\_dist

    #Note: odn[0] = "src\_CLUSTER\_ID" and odn[1] = "nbr\_CLUSTER\_ID"

    with arcpy.da.UpdateCursor(neighbor\_list, [odn[0], odn[1], 'src\_dist', 'nbr\_dist']) as cursor:

        for row in cursor:

            row[2] = row[0]  #src\_dist = src\_CLUSTER\_ID

            row[3] = row[1]  #nbr\_dist = nrb\_CLUSTER\_ID

            cursor.updateRow(row)

        del cursor

def build\_adj\_graph(out\_table, nbr\_list, ip):

    '''Builds the adjacency graph for all GUs using adjacencies found from neighbor\_list'''

    stateG = nx.Graph()  #Creates an empty graph that will contain adjacencies for the entire state

    origdist = {}  #Initializes a dictionary that will contain the original district number for each GU

    distnum = {}  #Initializes a dictionary that will contain the district number for each GU

    popnum = {}  #Initializes a dictionary that will contain the population for each GU

    countynum = {}  #Initializes a dictionary that will contain the county number for each GU

    area = {}  #Initializes a dictionary that will contain the area value for each GU (in square km)

    length = {}  #Initializes a dictionary that will contain each edge length (in km)

    boundary = {}  #Initializes a dictionary that will describe each edge as being a boundary edge or not

    blue\_votes = {}  #Initializes a dictionary that will contain the number of blue votes

    red\_votes = {}  #Initializes a dictionary that will contain the number of red votes

    sf\_name\_field = ip.in\_name\_field

    sf\_pop\_field = ip.in\_pop\_field

    arcprint("Creating the adjacency graph for all Geographical Units...")

    arcprint("Adding nodes to stateG...")

    with arcpy.da.SearchCursor(out\_table["name"], [sf\_name\_field, sf\_pop\_field, out\_table["county\_field"], out\_table["dist\_field"], "Area\_sq\_km", ip.in\_voteblue\_field, ip.in\_votered\_field]) as cursor:

        for row in cursor:

            popnum[row[0]] = row[1]  #Finds population of each GU

            countynum[row[0]] = row[2]  #Finds county number for each GU

            origdist[row[0]] = row[3]  #Finds original district number for each GU

            distnum[row[0]] = row[3]  #Finds district number for each GU.

            area[row[0]] = row[4]  #Finds the area for each GU

            blue\_votes[row[0]] = row[5]  #Finds number of blue votes for each GU

            red\_votes[row[0]] = row[6]  #Finds number of red votes for each GU

            stateG.add\_node(row[0])  #Adds each GU to the node list for stateG

            del row

        del cursor

    arcprint("Adding edges to stateG...")

    with arcpy.da.SearchCursor(nbr\_list, ["src\_OBJECTID", "nbr\_OBJECTID", "src\_dist", "nbr\_dist", "OBJECTID", "LENGTH"]) as cursor:

        for row in cursor:

            stateG.add\_edge(row[0], row[1])

            length[(row[0], row[1])] = float(row[5])

            if row[2] != row[3]:

                if row[2] == 0 or row[3] == 0:

                    boundary[(row[0], row[1])] = 2  #If either GU is the dummy district

                else:

                    boundary[(row[0], row[1])] = 1  #If edge represents a district boundary

            else:

                boundary[(row[0], row[1])] = 0  #If edge is not a district boundary

        del cursor

    arcprint("Adding Population attribute")

    nx.set\_node\_attributes(stateG, popnum, "Population")

    arcprint("Adding District Number attribute")

    nx.set\_node\_attributes(stateG, distnum, "District Number")

    arcprint("Adding Original District Number attribute")

    nx.set\_node\_attributes(stateG, origdist, "Original District Number")

    arcprint("Adding County Number attribute")

    nx.set\_node\_attributes(stateG, countynum, "County Number")

    arcprint("Adding Area attribute")

    nx.set\_node\_attributes(stateG, area, "Area")

    arcprint("Adding Blue Vote attribute")

    nx.set\_node\_attributes(stateG, blue\_votes, "Blue Votes")

    arcprint("Adding Red Vote attribute")

    nx.set\_node\_attributes(stateG, red\_votes, "Red Votes")

    arcprint("Adding Length attribute for edges")

    nx.set\_edge\_attributes(stateG, length, "Length")

    arcprint("Adding Boundary status attribute for edges")

    nx.set\_edge\_attributes(stateG, boundary, "Boundary")

    return stateG

def populate\_dist\_list(stateG, dist\_list):

    '''Populates dist\_list with all statistics based on the stateG graph'''

    distcount = len(set(dict(stateG.nodes("District Number")).values()))  #Finds number of unique districts (includes dummy district)

    for i in range(distcount):

        dist\_list[i] = District(i)  #Reinitializes (and therefore resets) the dist\_list

    for n in stateG.nodes:

        if stateG.nodes[n]["District Number"] == 0:  #Skips dummy district

            continue

        else:

            dist\_num = stateG.nodes[n]["District Number"]

            try:

                dist\_list[dist\_num].Area += stateG.nodes[n]["Area"]

                dist\_list[dist\_num].Population += stateG.nodes[n]["Population"]

                dist\_list[dist\_num].VoteCountBlue += stateG.nodes[n]["Blue Votes"]

                dist\_list[dist\_num].VoteCountRed += stateG.nodes[n]["Red Votes"]

            except TypeError:  #Handles initialization case where these values started as 'None'

                dist\_list[dist\_num].Area = 0

                dist\_list[dist\_num].Population = 0

                dist\_list[dist\_num].VoteCountBlue = 0

                dist\_list[dist\_num].VoteCountRed = 0

                dist\_list[dist\_num].Area += stateG.nodes[n]["Area"]

                dist\_list[dist\_num].Population += stateG.nodes[n]["Population"]

                dist\_list[dist\_num].VoteCountBlue += stateG.nodes[n]["Blue Votes"]

                dist\_list[dist\_num].VoteCountRed += stateG.nodes[n]["Red Votes"]

    dist\_list[0].Population = None

    ideal\_pop = round(sum(District.pop\_list(dist\_list)) / (distcount - 1))  #The -1 excludes the dummy district

    District.ideal\_pop = ideal\_pop

    District.num\_dists = distcount - 1

    arcprint("The starting population of each district is {0}. Thus, the ideal population for a district is {1}.", District.pop\_list(dist\_list), ideal\_pop)

    for d in dist\_list:

        if d.Perimeter == None:

            d.Perimeter = 0

    for e in list(stateG.edges):

        if stateG[e[0]][e[1]]["Boundary"] == 1 or stateG[e[0]][e[1]]["Boundary"] == 2:

            dist\_num0 = stateG.nodes[e[0]]["District Number"]

            dist\_num1 = stateG.nodes[e[1]]["District Number"]

            dist\_list[dist\_num0].Perimeter += stateG[e[0]][e[1]]["Length"]

            dist\_list[dist\_num1].Perimeter += stateG[e[0]][e[1]]["Length"]

            dist\_list[dist\_num0].Dist\_nbrs.append(stateG.nodes[e[1]]["District Number"])

            dist\_list[dist\_num1].Dist\_nbrs.append(stateG.nodes[e[0]]["District Number"])

    for i in range(distcount):

        dist\_list[i].Dist\_nbrs = list(set(dist\_list[i].Dist\_nbrs))  #Extracts unique values

"""def County\_Intersections(stateG):

    '''Finds the number of GUs in each county-district-intersection (CDIs)'''

    distcount = len(set(dict(stateG.nodes("District Number")).values())) - 1 #Finds number of unique districts (The -1 excludes the dummy district)

    num\_counties = len(set(dict(stateG.nodes("County Number")).values()))  #Finds number of unique counties

    units\_in\_CDI = np.zeros([distcount, num\_counties], dtype=int)

    # #Adds 1 to the matrix element A[i,j] if there is a precinct in the ith district and jth county

    for n in stateG:

        units\_in\_CDI[stateG.nodes[n]["District Number"] - 1][stateG.nodes[n]["County Number"] - 1] += 1

    #Counts the number of nonzero entries and subtracts either the number of districts or the number of counties---whichever is larger.

    #Then, the ideal value for CDI\_Count would be zero.

    CDI\_Count = np.count\_nonzero(units\_in\_CDI) - max(distcount, num\_counties)

    #GU stands for Geographical Unit. In this loop, we count the number of GUs in each county that are not in the most prevalent district

    excess\_GU\_mat = [0] \* max(distcount, num\_counties)

    transpose = units\_in\_CDI.transpose()

    idx = 0

    for row in transpose:

        maxval = max(row)

        excess\_GU\_mat[idx] = sum(row) - maxval

        idx += 1

    excess\_GU = sum(excess\_GU\_mat)

    return(units\_in\_CDI, CDI\_Count, excess\_GU)"""

def pop\_deviation(dist\_list):

    '''Returns a single positive integer that sums each district's deviation from the ideal population.

    Lower numbers for 'deviation' are better. A value of zero would indicate that every district has an equal number of people'''

    distcount = District.num\_dists

    absdev = [0 for i in range(distcount - 1)]  #The -1 skips the dummy district

    for i in range(distcount - 1):

        if dist\_list[i].Population != None:

            absdev[i] = abs(dist\_list[i].Population - District.ideal\_pop)

    deviation = round(sum(absdev))

    return deviation

def comp\_score(dist\_list, inverse=False):

    '''Computes the average Polsby-Popper score for a list of districts'''

    if inverse == False:

        comp\_list = [d.PPCompactScore for d in dist\_list]

    else:

        comp\_list = [d.invPPCompactScore for d in dist\_list]

    while True:

        try:

            comp\_list.remove(None)  #Should remove the dummy district

        except ValueError:

            break

    if len(comp\_list) != District.num\_dists:

        raise ValueError("comp\_score did not return the proper number of districts")

    PP\_Comp\_Score = sum(comp\_list) / len(comp\_list)  #Averages the compactness scores

    return PP\_Comp\_Score

def fair\_score(dist\_list):

    '''Computes the average fairness score for each district'''

    fairness\_list = District.EG\_list(dist\_list)

    while True:

        try:

            fairness\_list.remove(None)  #Should remove the dummy district

        except ValueError:

            break

    if len(fairness\_list) != District.num\_dists:

        raise ValueError("fair\_score did not return the proper number of districts")

    fairness\_score = sum(fairness\_list) / len(fairness\_list)

    return fairness\_score

"""def find\_dist\_nbrs(out\_table, dist\_list):

    '''Finds the starting list of district neighbors. This list will be edited throughout the script,

    but it will only be a superset of the district neighbors. Some entries will not be true district neighbors,

    but all viable district neighbors will be present.'''

    DistNbrList = out\_table["name"] + "\_dist\_nbr\_list"

    DistNbrPairs = []

    DNP = DistNbrPairs  #An alias

    arcpy.analysis.PolygonNeighbors(out\_table["name"], DistNbrList, out\_table["dist\_field"], both\_sides="NO\_BOTH\_SIDES")

    with arcpy.da.SearchCursor(DistNbrList, ["src\_Dist\_Assign", "nbr\_Dist\_Assign"], '''{}<{}'''.format("src\_Dist\_Assign", "nbr\_Dist\_Assign")) as cursor:

        for row in cursor:

            DNP.append(tuple(sorted((row[0], row[1]))))  #Appends a district neighbor pair

        del cursor, row

    for dnp in DNP:

        dist\_list[dnp[0] - 1].Dist\_nbrs.append(dnp[1] - 1)

        dist\_list[dnp[1] - 1].Dist\_nbrs.append(dnp[0] - 1)"""

def flip(stateG, dist\_list, cdi\_data):

    '''This is the flip algorithm. We move one GU across district lines'''

    adj\_flag = False

    edge\_count = stateG.number\_of\_edges()

    while adj\_flag == False:

        boundaryflag = False

        while boundaryflag == False:

            rand\_edge = random.randint(0, edge\_count - 1)  #Selects a random edge from stateG.edges

            pair = list(stateG.edges)[rand\_edge]

            n1 = pair[0]

            n2 = pair[1]

            if stateG.nodes[n1]["District Number"] == 0 or stateG.nodes[n2]["District Number"] == 0:  #If the node selected is the dummy node (which represents the dummy district)

                continue  #Resets the loop if either node represents the dummy node

            if stateG[n1][n2]["Boundary"] == 1:  #If the edge represents a district boundary

                boundaryflag = True

            else:

                continue  #Resets the loop if the edge does not represent a district boundary

        r = random.randint(0,1)

        GU = pair[r]  #This is the GU that will move districts

        other\_GU = pair[r - 1]  #If r = 0, then r - 1 = -1, which is the index for the last entry of the vector

        leaving\_dist = stateG.nodes[GU]["District Number"]

        entering\_dist = stateG.nodes[other\_GU]["District Number"]

        if stateG.nodes[GU]["District Number"] == 0:

            raise RuntimeError("GU should not be selected as the dummy district")

        DistrictNodes = [n for n in stateG.nodes() if stateG.nodes[n]["District Number"] == leaving\_dist]  #Finds all nodes in leaving\_dist

        DistrictNodes.remove(GU)

        sg\_for\_leaving\_dist = stateG.subgraph(DistrictNodes)  #Creates a subgraph containing all nodes from DistrictNodes

        if nx.is\_connected(sg\_for\_leaving\_dist):

            adj\_flag = True  #True if the leaving\_dist is still contiguous

        else:

            adj\_flag = False  #Keeps the adj\_flag at False if the move would create a discontiguity

    stateG.nodes[GU]["District Number"] = entering\_dist  #Changes the district number in stateG

    '''#Checks that the flip won't cause a discontiguity

    leaving\_dist\_nodes = []

    for n in stateG.nodes():

        if stateG.nodes[n]["District Number"] == leaving\_dist:

            leaving\_dist\_nodes.append(n)

    leaving\_dist\_graph = stateG.subgraph(leaving\_dist\_nodes)  #Creates a subgraph for leaving\_dist

    if nx.is\_connected(leaving\_dist\_graph) == False:

        arcprint("Flipping GU {0} from district {1} to district {2} would create a discontiguity. Retrying Flip.", GU, leaving\_dist, entering\_dist)

        return False  #Indicates that the Flip was unsuccessful'''

    for nbr in list(stateG.neighbors(GU)):  #Cycles through neighboring nodes to adjust boundary status, perimeter, and district neighbors

        nbr\_dist = stateG.nodes[nbr]["District Number"]

        if nbr\_dist == entering\_dist:

            stateG[GU][nbr]["Boundary"] = 0

            dist\_list[entering\_dist].Perimeter -= stateG[GU][nbr]["Length"]  #This is now 'interior' perimeter

            dist\_list[leaving\_dist].Perimeter -= stateG[GU][nbr]["Length"]  #Neither GU nor nbr is in leaving\_dist

        elif nbr\_dist == 0:  #If nbr is the dummy district

            stateG[GU][nbr]["Boundary"] = 2

            dist\_list[entering\_dist].Perimeter += stateG[GU][nbr]["Length"]

            dist\_list[leaving\_dist].Perimeter -= stateG[GU][nbr]["Length"]

            dist\_list[entering\_dist].Dist\_nbrs.append(nbr\_dist)  #Add neighboring district to neighbor list. We delete duplicates later

        elif nbr\_dist == leaving\_dist:  #If nbr is part of the leaving\_dist

            stateG[GU][nbr]["Boundary"] = 1

            dist\_list[entering\_dist].Perimeter += stateG[GU][nbr]["Length"]

            dist\_list[leaving\_dist].Perimeter += stateG[GU][nbr]["Length"]

            dist\_list[entering\_dist].Dist\_nbrs.append(nbr\_dist)  #Add neighboring district to neighbor list. We delete duplicates later

        else:  #Neighboring district is not entering\_dist, leaving\_dist, or the dummy district

            stateG[GU][nbr]["Boundary"] = 1

            dist\_list[entering\_dist].Perimeter += stateG[GU][nbr]["Length"]

            dist\_list[leaving\_dist].Perimeter -= stateG[GU][nbr]["Length"]

            dist\_list[entering\_dist].Dist\_nbrs.append(nbr\_dist)  #Add neighboring district to neighbor list. We delete duplicates later

    dist\_list[entering\_dist].Population += stateG.nodes[GU]["Population"]  #Adds population to the dist\_list population entry corresponding to entering\_dist

    dist\_list[leaving\_dist].Population -= stateG.nodes[GU]["Population"]  #Subtracts population from the dist\_list population entry corresponding to leaving\_dist

    dist\_list[entering\_dist].Area += stateG.nodes[GU]["Area"]  #Adds area to the dist\_list area entry corresponding to entering\_dist

    dist\_list[leaving\_dist].Area -= stateG.nodes[GU]["Area"]  #Subtracts area from the dist\_list area entry corresponding to leaving\_dist

    dist\_list[entering\_dist].VoteCountRed += stateG.nodes[GU]["Red Votes"]  #Adds red votes to the dist\_list red votes entry corresponding to entering\_dist

    dist\_list[leaving\_dist].VoteCountRed -= stateG.nodes[GU]["Red Votes"]  #Subtracts red votes from the dist\_list red votes entry corresponding to leaving\_dist

    dist\_list[entering\_dist].VoteCountBlue += stateG.nodes[GU]["Blue Votes"]  #Adds blue votes to the dist\_list blue votes entry corresponding to entering\_dist

    dist\_list[leaving\_dist].VoteCountBlue -= stateG.nodes[GU]["Blue Votes"]  #Subtracts blue votes from the dist\_list blue votes entry corresponding to leaving\_dist

    dist\_list[entering\_dist].Dist\_nbrs = list(set(dist\_list[entering\_dist].Dist\_nbrs))  #Extracts unique values

    cdi\_data.upd\_cdi\_mat\_flip(stateG, GU, leaving\_dist, entering\_dist)  #Updates the CDI matrix

    arcprint("Completed Flip algorithm. Flipped GU {0} from district {1} to district {2}.", GU, leaving\_dist, entering\_dist)

    return True  #Indicates that the flip was successful

    """# deviation[count] = pop\_deviation(sumpop,idealpop,distcount)

    # #arcprint("absolute deviation is {0}",deviation[count])

    # DeltaE\_dev = deviation[count] - deviation[count-1]

    # arcprint("DeltaE\_dev = {0}. T = {1}",DeltaE\_dev,T)

    # if DeltaE\_dev <0:

    #     ip.temp = ip.temp\*.997

    #     continue

    # else :

    #     rand = random.uniform(0,1)

    #     try:

    #         p = 1/math.exp(DeltaE\_dev/T)

    #     except OverflowError:

    #         p = 0

    #     arcprint("p = {0}. rand = {1}", p, rand)

    #     if rand <= p:

    #         T = T \* .997

    #         continue

    #     else: #undoes the district changes previously made.

    #         count = count-1

    #         sumpop[currdist - 1] = sumpop[currdist-1] + distpop

    #         sumpop[newdist - 1] = sumpop[newdist-1] - distpop

    #         with arcpy.da.UpdateCursor(out\_table, [NameField, DistField, PopField],"{0} = '{1}'".format(NameField,randshape)) as cursor:

    #             for row in cursor:

    #                 row[1] = currdist

    #                 cursor.updateRow(row)

    #             del cursor"""

def recom(stateG, dist\_list, tol, count, dist1=-1, dist2=-1):

    '''Does a Recombination step for the graph stateG'''

    #1. Determine if two districts are adjacent

    #2. Grab all GUs from those two districts and create a subgraph.

    #3. Error check: Verify that the subgraph is connected

    #4. Wilson's Algorithm

    #5. Make sure that the resulting tree acquires all attributes from stateG

    #6. FindEdgeCut

    #7. Reassign the new subgraphs to their proper districts

    distcount = len(set(dict(stateG.nodes("District Number")).values())) - 1  #Finds number of unique districts (-1 excludes dummy district)

    dist\_adj\_flag = False

    orig\_dist1 = dist1

    orig\_dist2 = dist2

    if orig\_dist1 > distcount:

        arcprint("dist\_1 was too large. We will randomly reselect this district.")

    if orig\_dist2 > distcount:

        arcprint("dist\_2 was too large. We will randomly reselect this district.")

    while dist\_adj\_flag == False:

        if orig\_dist1 <= -1 or orig\_dist1 > distcount:

            dist1 = random.randint(1, distcount)  #Randomly selects dist1 if it wasn't provided as input or if it is out of range

        if orig\_dist2 <= -1 or orig\_dist2 > distcount:

            dist2 = random.randint(1, distcount)  #Randomly selects dist2 if it wasn't provided as input or if it is out of range

        if dist2 not in dist\_list[dist1].Dist\_nbrs or dist1 not in dist\_list[dist2].Dist\_nbrs:

            continue  #Restarts the loop if this district neighbor pair can't be located in dist\_list

        dist1\_and\_dist2\_nodes = []

        for n in stateG.nodes():

            if stateG.nodes[n]["District Number"] == dist1 or stateG.nodes[n]["District Number"] == dist2:

                dist1\_and\_dist2\_nodes.append(n)

        two\_dist\_graph = stateG.subgraph(dist1\_and\_dist2\_nodes)  #Creates a subgraph of the two districts

        if nx.is\_connected(two\_dist\_graph) == False:

            arcprint("District {0} and District {1} are not adjacent. Reselecting districts", dist1, dist2)

            try:

                dist\_list[dist1].Dist\_nbrs.remove(dist2)

            except ValueError:

                pass

            try:

                dist\_list[dist2].Dist\_nbrs.remove(dist1)

            except ValueError:

                pass

            dist\_adj\_flag = False  #Keeps the flag at False

        else:  #If the two districts are indeed adjacent

            dist\_adj\_flag = True

    tree = wilson(two\_dist\_graph, random)  #Creates a uniform random spanning tree for two\_dist\_graph using Wilson's algorithm

    subgraphs = find\_edge\_cut(tree, tol)  #Finds an edge to remove from the tree to create two districts

    #This next section of code decides which subgraph should become district 1 and which should become district 2

    if subgraphs:  #If subgraphs is not empty

        s0d1count = 0

        s0d2count = 0

        s1d1count = 0

        s1d2count = 0

        for i in subgraphs[0]:

            if stateG.nodes[i]["District Number"] == dist1:

                s0d1count += 1

            elif stateG.nodes[i]["District Number"] == dist2:

                s0d2count += 1

        for i in subgraphs[1]:

            if stateG.nodes[i]["District Number"] == dist1:

                s1d1count += 1

            elif stateG.nodes[i]["District Number"] == dist2:

                s1d2count += 1

        #Assigns either dist1 or dist2 to the moved GUs

        if s0d1count + s1d2count >= s0d2count + s1d1count:

            for i in subgraphs[0]:

                stateG.nodes[i]["District Number"] = dist1

            for i in subgraphs[1]:

                stateG.nodes[i]["District Number"] = dist2

        else:

            for i in subgraphs[0]:

                stateG.nodes[i]["District Number"] = dist2

            for i in subgraphs[1]:

                stateG.nodes[i]["District Number"] = dist1

        #Resets dist\_list entries

        dist\_list[dist1].reset\_vals()

        dist\_list[dist2].reset\_vals()

        #Cycles through nodes to update boundary list

        for GU in two\_dist\_graph.nodes:

            GU\_dist = stateG.nodes[GU]["District Number"]

            for nbr in list(stateG.neighbors(GU)):  #Cycles through neighboring nodes to update boundary status, perimeter, and Dist\_nbrs

                nbr\_dist = stateG.nodes[nbr]["District Number"]

                if nbr\_dist == GU\_dist:

                    stateG[GU][nbr]["Boundary"] = 0

                    dist\_list[GU\_dist].Perimeter += 0

                elif nbr\_dist == 0:  #If nbr is the dummy node

                    stateG[GU][nbr]["Boundary"] = 2

                    dist\_list[GU\_dist].Perimeter += stateG[GU][nbr]["Length"]

                    dist\_list[GU\_dist].Dist\_nbrs.append(0)

                else:  #If the GU and nbr are in different districts

                    stateG[GU][nbr]["Boundary"] = 1

                    dist\_list[GU\_dist].Dist\_nbrs.append(nbr\_dist)

                    dist\_list[GU\_dist].Perimeter += stateG[GU][nbr]["Length"]

            #Updates the dist\_list instance

            dist\_list[GU\_dist].Area += two\_dist\_graph.nodes[GU]["Area"]

            dist\_list[GU\_dist].Population += two\_dist\_graph.nodes[GU]["Population"]

            dist\_list[GU\_dist].VoteCountRed += two\_dist\_graph.nodes[GU]["Red Votes"]

            dist\_list[GU\_dist].VoteCountBlue += two\_dist\_graph.nodes[GU]["Blue Votes"]

        dist\_list[dist1].Dist\_nbrs = list(set(dist\_list[dist1].Dist\_nbrs))  #Isolates unique values

        dist\_list[dist2].Dist\_nbrs = list(set(dist\_list[dist2].Dist\_nbrs))  #Isolates unique values

        arcprint("Recom number {0} succeeded. Reorganized districts {1} and {2}", count.recomcount, dist1, dist2)

        print("\n")

        return True  #Indicates that the recom\_success\_flag is True

    else:  #If subgraphs were empty (i.e. we couldn't find an edge to cut that split population within tolerance)

        return False  #Indicates that the recom\_success\_flag is False

def wilson(graph, rng):

    '''Returns a uniform spanning tree on G'''

    walk = loopErasedWalk(graph, rng)

    currentNodes = [n for n in walk]

    uniformTree = nx.Graph()

    for i in range(len(walk) - 1):

        uniformTree.add\_edge(walk[i], walk[i + 1])

    treeNodes = set(uniformTree.nodes)

    neededNodes = set(graph.nodes) - treeNodes

    while neededNodes:

        v = rng.choice(sorted(list(neededNodes))) # sort for code repeatability

        walk = loopErasedWalk(graph, rng, v1 = [v], v2 = currentNodes)

        currentNodes += walk

        for i in range(len(walk) - 1):

            uniformTree.add\_edge(walk[i], walk[i + 1])

        treeNodes = set(uniformTree.nodes)

        neededNodes = set(graph.nodes) - treeNodes

    pass

    nx.set\_node\_attributes(uniformTree, dict(graph.nodes("Population")), "Population")

    nx.set\_node\_attributes(uniformTree, dict(graph.nodes("District Number")), "District Number")

    nx.set\_node\_attributes(uniformTree, dict(graph.nodes("Original District Number")), "Original District Number")

    nx.set\_node\_attributes(uniformTree, dict(graph.nodes("County Number")), "County Number")

    nx.set\_node\_attributes(uniformTree, dict(graph.nodes("Area")), "Area")

    nx.set\_node\_attributes(uniformTree, dict(graph.nodes("Blue Votes")), "Blue Votes")

    nx.set\_node\_attributes(uniformTree, dict(graph.nodes("Red Votes")), "Red Votes")

    nx.set\_edge\_attributes(uniformTree, dict(graph.edges("Length")), "Length")

    nx.set\_edge\_attributes(uniformTree, dict(graph.edges("Boundary")), "Boundary")

    return uniformTree

def loopErasedWalk(graph, rng, v1 = None, v2 = None):

    '''Returns a loop-erased random walk between components v1 & v2'''

    if v1 is None:

        v1 = [rng.choice(sorted(list(graph.nodes)))]

    if v2 is None:

        v2 = [rng.choice(sorted(list(graph.nodes)))]

    v = rng.choice(sorted(v1))

    walk = [v]

    while v not in v2:

        v = rng.choice(sorted(list(graph.neighbors(v))))

        if v in walk:

            walk = walk[0:walk.index(v)]

        walk.append(v)

    return walk

def find\_edge\_cut(tree, tol):

    '''Input a tree graph and a percent tolerance. The function will remove a

    random edge that splits the tree into two pieces such that each piece

    has population within that percent tolerance. The variable 'tol' should be a positive real

    number in (0,100].'''

    if tol > 100 or tol <= 0 or (isinstance(tol, float) == False and isinstance(tol, int) == False):

        raise ValueError("tol must be a float or integer variable in the range (0,100].")

    if nx.is\_tree(tree) == False:

        raise ValueError("The input graph must be a tree.")

    tree\_edge\_list = list(tree.edges)

    random.shuffle(tree\_edge\_list)  #Randomly shuffles the edges of T

    e = None

    num\_edges = len(tree\_edge\_list)

    for i in range(num\_edges):

        e = tree\_edge\_list[i]  #Edge to delete

        tree.remove\_edge(\*e)

        subgraphs = nx.connected\_components(tree)

        subgraphs\_lst = list(subgraphs)

        subgraphs\_lst[0] = sorted(subgraphs\_lst[0])

        subgraphs\_lst[1] = sorted(subgraphs\_lst[1])

        dist\_pop1 = sum(value for key, value in nx.get\_node\_attributes(tree, "Population").items() if key in subgraphs\_lst[0])  #Finds population sum for first district

        dist\_pop2 = sum(value for key, value in nx.get\_node\_attributes(tree, "Population").items() if key in subgraphs\_lst[1])  #Finds population sum for second district

        total\_pop = dist\_pop1 + dist\_pop2

        avg\_pop = total\_pop / 2

        if abs(dist\_pop1 - avg\_pop) > 0.01 \* tol \* avg\_pop or abs(dist\_pop2 - avg\_pop) > 0.01 \* tol \* avg\_pop:  #If both proposed districts are outside the prescribed tolerance

            tree.add\_edge(\*e)  #Adds the edge back to the tree if it didn't meet the tolerance

        else:  #This is what we want: both proposed districts within the prescribed tolerance

            if i == 0:

                pass

                #arcprint("Population requirement was met. Removing edge {0}. Required {1} iteration.", e, i+1)

            else:

                pass

                #arcprint("Population requirement was met. Removing edge {0}. Required {1} iterations.", e, i+1)

            return subgraphs\_lst

        if i == num\_edges - 1:

            #arcprint("No subgraphs with appropriate criteria requirements were found. Required {0} iterations.\n", i+1)

            return []  #Returns empty subgraphs list if no appropriate subgraphs were found.

def main(\*args):

    """Runs the primary instance of the algorithm."""

    global RUNSPOT  #Allows RUNSPOT to be changed inside a function

    if sys.executable == r"C:\Program Files\ArcGIS\Pro\bin\ArcGISPro.exe":  #Change this line if ArcGIS is located elsewhere

        RUNSPOT = "ArcGIS"

    else:

        RUNSPOT = "console"

    # Set environment settings

    global CURRENTDIR

    global PATH

    CURRENTDIR = os.getcwd()

    PATH = CURRENTDIR + "\\SC\_Redistricting\_Updated.gdb"

    arcprint("Current PATH is {0}", PATH)

    arcpy.env.workspace = PATH

    arcpy.env.overwriteOutput = True

    #Get user input

    sig = signature(input\_vals.\_\_init\_\_)

    if len(sys.argv) == len(sig.parameters):

        arcprint("Using sys.argv")

        ip = input\_vals(sys.argv)  #First attempts to take input from system arguments (Works for ArcGIS parameters, for instance)

    elif len(args) == len(sig.parameters):

        arcprint("Using args")

        ip = input\_vals(args)  #Second, tries to take input from explicit input into main()

    else:

        arcprint("Using default variable choices")

        ip = input\_vals(None, None, None, None, None, None, None, None, None, None, None, None, None, None)  #Dummy values to be overwritten in next line

        ip.default\_user\_input()  #Finally, manually assigns input values if they aren't provided

    #Marking the start time of the run.

    now = datetime.datetime.now()

    arcprint("Starting date and time : {0}", now.strftime("%m-%d-%y %H:%M:%S"))

    timetxt = now.strftime("\_%m%d%y\_%H%M")

    #This builds alpha, which is the normalized unit vector that details how much we care about any given metric.

    metric\_count = 5

    alpha = build\_alpha(metric\_count, 10) ###10 SHOULD BE USER INPUT

    tol = ip.tol  #tol will be modified later

    temp = ip.temp  #temp will be modified later

    #Creates an initial map using Spatially Constrained Multivariate Clustering

    out\_table = {}  #Creates a dictionary that will hold all information about the out\_table

    out\_table["name"] = initialize\_map(ip, timetxt)

    #Adds Area and Perimeter values to the out\_table

    arcpy.CalculateGeometryAttributes\_management(out\_table["name"], [["Area\_sq\_km", "AREA\_GEODESIC"], ["Perimeter\_km", "PERIMETER\_LENGTH\_GEODESIC"]], "KILOMETERS", "SQUARE\_KILOMETERS")

    #Adds temp\_dist as a column in out\_table. Will simplify Zonal Geometry later

    #temp\_dist(out\_table)

    #Assigns DistField as "Dist\_Assign" and creates the field if it's not already there

    if not arcpy.ListFields(out\_table["name"], "Dist\_Assign"):

        arcpy.AddField\_management(out\_table["name"], "Dist\_Assign", "SHORT", field\_alias="DIST\_ASSIGNMENT")

    out\_table["dist\_field"] = "Dist\_Assign"

    #Adds a field named County\_Num to out\_table

    arcpy.AddField\_management(out\_table["name"], "County\_Num", "SHORT", field\_alias="County\_Num")

    out\_table["county\_field"] = "County\_Num"

    #Populates county\_dict with 0-x, based on the sorted county numbers

    county\_dict = make\_county\_dict(ip, out\_table)

    #Initializes a list of lists that will categorize each geographic unit (GU) into its district

    #GU\_list = [[ ] for d in range(ip.distcount)]

    #Copies all CLUSTER\_ID's into Dist\_Assign and adds updated county numbers to out\_table

    with arcpy.da.UpdateCursor(out\_table["name"], [out\_table["dist\_field"], "CLUSTER\_ID", "SOURCE\_ID", "County", out\_table["county\_field"]]) as cursor:

        for row in cursor:

            row[0] = row[1]  #Dist\_Assign = CLUSTER\_ID

            row[4] = county\_dict[int(row[3])]  #County\_Num = sorted county value (if there are n counties, then this value will be between 0 and n-1)

            #GU\_list[row[0] - 1].append(row[2])

            cursor.updateRow(row)

        del cursor, row

    #Creates dummy GU that surrounds the state

    create\_buffer\_dist(out\_table)

    #Runs CreateNeighborList and returns the name of the neighbor\_list

    neighbor\_list = CreateNeighborList(out\_table)

    #Creates an instance of District for each District and the dummy district (the GU surrounding the state)

    dist\_list = [None] \* (ip.distcount + 1)

    for i in range(ip.distcount):

        dist\_list[i] = District(i)  #Initializes District variables for each district

    #Adds src\_dist and nbr\_dist to neighbor\_list

    update\_nbr\_list(neighbor\_list)

    stateG = build\_adj\_graph(out\_table, neighbor\_list, ip)

    #Finds district populations and adds them to the dist\_list instances

    #find\_dist\_pops(ip, out\_table, dist\_list)

    populate\_dist\_list(stateG, dist\_list)

    #Finds District Neighbor Pairs (dict) and returns the District Neighbor List (string).

    #find\_dist\_nbrs(out\_table, dist\_list)

    #Finds compactness information

    #[old\_var, MapStats] = GraphMeasures.main(out\_table["name"], out\_table["dist\_field"], ip.in\_voteblue\_field, ip.in\_votered\_field)  #Populates DistrictStats and MapStats using GraphMeasures

    # comp\_list = [o.PPCompactScore for o in DistrictStats]  #comp is a list of compactness scores

    # for i in range(len(comp\_list)): comp\_list[i] = (1 / comp\_list[i]) - 1  #Inverts the PP compactness score and subtracts 1 to make ideal value = 0

    #mm\_value = MapStats.MedianMean  #mm\_value is the Median Mean Score

    #Populates County-District-Intersection (CDI) values

    cdi\_data = CDI(stateG)

    #arcprint("The fairness scores for this map are: Median\_Mean = {0}", fair)

    arcprint("CDI\_Count = {0}", cdi\_data.cdi\_count)

    arcprint("Total number of precincts (calculated by np.sum(cdi\_data.cdi\_mat)) = {0}", np.sum(cdi\_data.cdi\_mat))

    #Creates vectors of zeros that will hold values for population deviation, average compactness, etc.

    obj\_vals = objective\_vals(ip)

    ov = obj\_vals  #An alias

    #Populates the zeroth entry for all vectors

    ov.fill\_obj\_vals(pop\_deviation(dist\_list), comp\_score(dist\_list), fair\_score(dist\_list), cdi\_data, "initialization", 0)

    #Initializing the main line of the Simulated Annealing Algorithm

    count = counters()  #Keeps track of the various counters needed

    #Starting the main line of the Simulated Annealing Algorithm

    while temp > 0.1 and count.recomcount < ip.num\_recoms and count.stopcounter < ip.max\_stop\_counter:

        for i in range(ip.num\_flips):

            flip\_success\_flag = False

            while flip\_success\_flag == False:

                flip\_success\_flag = flip(stateG, dist\_list, cdi\_data)  #Does the Flip algorithm and returns "True" if flip succeeded

            count.flipcount += 1

            ov.fill\_obj\_vals(pop\_deviation(dist\_list), comp\_score(dist\_list), fair\_score(dist\_list), cdi\_data, "flip", count.currentit)

        count.stopcounter = 0  #Resets the stopcounter

        recom\_success\_flag = False

        while recom\_success\_flag == False:

            recom\_success\_flag = recom(stateG, dist\_list, tol, count)  #Does recombination algorithm and returns "True" if recom succeeded

            count.stopcounter += 1

            if count.stopcounter >= ip.max\_stop\_counter:

                arcprint("We failed in {0} consecutive recom attempts. Skipping this recom step.", count.stopcounter)

                break

        count.recomcount += 1

        if recom\_success\_flag == True:

            ov.fill\_obj\_vals(pop\_deviation(dist\_list), comp\_score(dist\_list), fair\_score(dist\_list), cdi\_data, "recom", count.currentit)

        else:

            ov.fill\_obj\_vals(pop\_deviation(dist\_list), comp\_score(dist\_list), fair\_score(dist\_list), cdi\_data, "failed\_recom", count.currentit)

        temp = temp \* ip.coolingrate

    #Adds Dist\_Assign values back to GUs in map

    with arcpy.da.UpdateCursor(out\_table["name"], [out\_table["dist\_field"], "OBJECTID"]) as cursor:

        for row in cursor:

            n = row[1]

            row[0] = stateG.nodes[n]["District Number"]

            cursor.updateRow(row)

        del cursor, row

    arcprint("We finished!")

if \_\_name\_\_ == "\_\_main\_\_":

    main()