```
'''
File name      : gurobi_tsp.py
Author         : Jared Hansen
Date created   : 10/23/2019
Python version : 3.7.3

DESCRIPTION: The purpose of this script is to solve the TSP (Traveling
             Salesman Problem) using integer programming.
             For sake of disclosure, I should note that I began this code by
             using the script provided by Gurobi here
             https://www.gurobi.com/documentation/8.1/examples/tsp_py.html
             I have elected to use the Gurobi interface with Python to do the
             60-city problem since the pulp package was only computationally
             efficient enough to do the 6-, 20-, and 40-city problems.

'''





#==============================================================================
#==============================================================================
#==== IMPORT STATEMENTS
#==============================================================================
#==============================================================================
import datetime as dt
import itertools
import math
import matplotlib.pyplot as plt
import numpy as np
import random
import sys
from datetime import datetime
from gurobipy import *





#==============================================================================
#==============================================================================
#==== FUNCTION IMPLEMENTATIONS
#==============================================================================
#==============================================================================

def subtourelim(model, where):
    ''' This function is used to eliminate sub-tours from a given route.

    Parameters
    ----------
    model : gurobipy.Model
        Stored model formulation (e.g. constraints, objective, dec. vars.)
    where : int

    Returns
    -------
    n/a : updates the gurobipy.Model by removing subtours (adding constraints)
    '''
    if where == GRB.Callback.MIPSOL:
        # make a list of edges selected in the solution
        vals = model.cbGetSolution(model._vars)
        selected = tuplelist((i,j) for i,j in model._vars.keys() if vals[i,j] > 0.5)
        # find the shortest cycle in the selected edge list
        tour = subtour(selected)
        if len(tour) < n:
            # add subtour elimination constraint for every pair of cities in tour
```

```
            model.cbLazy(quicksum(model._vars[i,j]
                                  for i,j in itertools.combinations(tour, 2))
                          <= len(tour)-1)

def subtour(edges):
    ''' This function accepts a tuplelist (data type that is specific to
    GurobiPy) and finds the shortest sub-tour for the given cities.

    Parameters
    ----------
    edges : gurobipy.tuplelist
        Collection of cities (nodes) in the problem.

    Returns
    -------
    cycle : list
        List of the cities (nodes) that results in the shortest subtour.
    '''
    unvisited = list(range(n))
    cycle = range(n+1) # initial length has 1 more city
    while unvisited: # true if list is non-empty
        thiscycle = []
        neighbors = unvisited
        while neighbors:
            current = neighbors[0]
            thiscycle.append(current)
            unvisited.remove(current)
            neighbors = [j for i,j in edges.select(current,'*') if j in unvisited]
        if len(cycle) > len(thiscycle):
            cycle = thiscycle
    return cycle





#=========================================================================
#=========================================================================
#==== PROCEDURAL CODE
#=========================================================================
#=========================================================================

# How many cities are we dealing with?
n = 400
# Create labeled cities
cities = []
for i in range(n):
    # Append chars '0', '1', '2', ..., 'n' to the cities list
    cities.append(str(i))

# Create coordinates for each city
random.seed(1)
points = [(random.randint(0,100),random.randint(0,100)) for i in range(n)]
cities_x = np.array(points)[:,0]
cities_y = np.array(points)[:,1]

# Plot the points that we've generated, with cities labeled by number
fig, tsp_img = plt.subplots()
tsp_img.scatter(cities_x, cities_y)
for i, cityNum in enumerate(cities):
    tsp_img.annotate(cityNum, xy=(cities_x[i], cities_y[i]), xytext=(1,1),
                     textcoords='offset points',
                     fontsize=9)

# Dictionary of Euclidean distance between each pair of points
dist = {(i,j) :
    math.sqrt(sum((points[i][k]-points[j][k])**2 for k in range(2)))
```

```
    for i in range(n) for j in range(i)}
# Instantiate an instance of a model
m = Model()
# Create variables
vars = m.addVars(dist.keys(), obj=dist, vtype=GRB.BINARY, name='e')
for i,j in vars.keys():
    vars[j,i] = vars[i,j] # edge in opposite direction
# Add "assignment constraints" (each city gets arrived at once, departed from
# once).
m.addConstrs(vars.sum(i,'*') == 2 for i in range(n))

# Optimize model, time how long it takes to run
start_time = dt.datetime.now()
m._vars = vars
m.Params.lazyConstraints = 1
m.optimize(subtourelim)   # this eliminates sub-tours from our route
vals = m.getAttr('x', vars)
selected = tuplelist((i,j) for i,j in vals.keys() if vals[i,j] > 0.5)
timed = str(dt.datetime.now() - start_time)
print("The solver took " + timed + " to run for " + str(n) + " cities.")
print("(time is in hours : minutes : seconds.frac_of_second)")

# Determine the order in which we visit the cities
tour = subtour(selected)
assert len(tour) == n
tour.append(0)
# Output the results of the problem
print('')
print('Optimal tour: %s' % str(tour))
print('Optimal cost: %g' % m.objVal)
print('')

# Plot the solution over top of the points
fig, tsp_img = plt.subplots()
tsp_img.scatter(cities_x, cities_y)
for i, cityNum in enumerate(cities):
    tsp_img.annotate(cityNum, xy=(cities_x[i], cities_y[i]), xytext=(1,1),
                    textcoords='offset points',
                    fontsize=7)
for i in range(len(tour)-1):
    plt.plot([cities_x[cities.index(str(tour[i]))],cities_x[cities.index(str(tour[i+1]))]],
             [cities_y[cities.index(str(tour[i]))],cities_y[cities.index(str(tour[i+1]))]], 'c')
plt.title('Solution for ' + str(n) + ' Randomly Created Cities. Total distance: ' + str(round(m.objVal, 3)) + ' units')
plt.xlabel('x coordinate of the city')
plt.ylabel('y coordinate of the city')
plt.show()
```