```python
#!/usr/bin/python3

from which_pyqt import PYQT_VER
if PYQT_VER == 'PYQT5':
    from PyQt5.QtCore import QLineF, QPointF
elif PYQT_VER == 'PYQT4':
    from PyQt4.QtCore import QLineF, QPointF
else:
    raise Exception('Unsupported Version of PyQt: {}'.format(PYQT_VER))



import copy
import time
import numpy as np
from TSPClasses import *
import heapq



class TSPSolver:
    def __init__( self, gui_view ):
        self._scenario = None

    def setupWithScenario( self, scenario ):
        self._scenario = scenario


    ''' <summary>
        This is the entry point for the default solver
        which just finds a valid random tour
        </summary>
        <returns>results array for GUI that contains three ints: cost of solution, time
        spent to find solution, number of solutions found during search (
not counting initial BSSF estimate)</returns> '''
    def defaultRandomTour( self, start_time, time_allowance=60.0 ):

        results = {}


        start_time = time.time()

        cities = self._scenario.getCities()
        ncities = len(cities)
        foundTour = False
        count = 0
        while not foundTour:
            # create a random permutation
            perm = np.random.permutation( ncities )

            #for i in range( ncities ):
                #swap = i
                #while swap == i:
                    #swap = np.random.randint(ncities)
                #temp = perm[i]
                #perm[i] = perm[swap]
                #perm[swap] = temp

            route = []

            # Now build the route using the random permutation
            for i in range( ncities ):
                route.append( cities[ perm[i] ] )

            bssf = TSPSolution(route)
            #bssf_cost = bssf.cost()
            #count++;
```

```
            count += 1

            #if costOfBssf() < float('inf'):
            if bssf.costOfRoute() < np.inf:
                # Found a valid route
                foundTour = True
        #} while (costOfBssf() == double.PositiveInfinity);                // until a
        valid route is found
        #timer.Stop();

        results['cost'] = bssf.costOfRoute()
        #costOfBssf().ToString();                          // load results array
        results['time'] = time.time() - start_time
        results['count'] = count
        results['soln'] = bssf

       # return results;
        return results

    def defaultRandomTourBSSF( self ):
        cities = self._scenario.getCities()
        ncities = len(cities)
        foundTour = False
        while not foundTour:
            # create a random permutation
            perm = np.random.permutation( ncities )
            route = []

            # Now build the route using the random permutation
            for i in range( ncities ):
                route.append( cities[ perm[i] ] )

            bssf = TSPSolution(route)

            if bssf.costOfRoute() < np.inf:
                # Found a valid route
                foundTour = True
        return bssf



    def greedyBSSF( self ):
        results = {}

        cities = self._scenario.getCities()
        visited = []
        ncities = len(cities)
        foundTour = False
        currCity = cities[0]
        visited.append(currCity)
        while len(visited) < ncities:
            nextCity = self.getNextCity_Greedy(currCity, cities, visited)
            visited.append(nextCity)
            currCity = nextCity
        greedySolution = TSPSolution(visited)
        return greedySolution

    def greedy( self, start_time, time_allowance=60.0 ):
        start_time = time.time()
        results = {}
        count = 0
        cities = self._scenario.getCities()
        visited = []
        ncities = len(cities)
        foundTour = False
        currCity = cities[0]
        visited.append(currCity)
```

```
132             while len(visited) < ncities:
133                 nextCity = self.getNextCity_Greedy(currCity, cities, visited)
134                 visited.append(nextCity)
135                 currCity = nextCity
136             greedySolution = TSPSolution(visited)
137 
138             results['cost'] = greedySolution.costOfRoute()
139             results['time'] = time.time() - start_time
140             results['count'] = count
141             results['soln'] = greedySolution
142 
143             return results
144 
145 
146         def branchAndBound( self, start_time, time_allowance=60.0 ):
147 
148             #Stats for report
149             maxNumStoredStates = 0
150             totalHeldStates = 0
151             totalPrunedStates = 0
152             optimalSolutionFound = True
153 
154             #Initialize values to begin branch & bound algorithm
155             results = {}
156             start_time = time.time()
157             foundTour = False
158             count = 0
159             self.cities = self._scenario.getCities()
160             initCity = self.cities[0]
161             unvisitedCities = [True] * len(self.cities)
162             unvisitedCities[initCity._index] = False
163 
164             #Get initial bssf by the greedy simple tour, if that does not have a valid path
                or the random tour happens to be better use it.
165             greedyBSSF = self.greedyBSSF()
166             randomTourBSSF = self.defaultRandomTourBSSF()
167             if greedyBSSF.costOfRoute() < randomTourBSSF.costOfRoute():
168                 bssf = greedyBSSF
169             else:
170                 bssf = randomTourBSSF
171 
172             #Generate initial adjMatrix
173             initAdjMatrix = self.generateAdjMatrix(self.cities)
174             initPC = PriorityCount(0, 1, 0)
175             # Heap tuples order: lowerBound, currCity, unvisitedCities, adjMatrix, path
176             rootProblem = (initPC, initCity, unvisitedCities, initAdjMatrix, [initCity])
177             subProblems = []
178             heapq.heappush(subProblems, rootProblem)
179 
180             totalHeldStates+=1
181             while len(subProblems) > 0:
182                 #check if maxMunStoredStates needs to be updated
183                 if len(subProblems) > maxNumStoredStates:
184                     maxNumStoredStates = len(subProblems)
185                 #check if the time allowance has been exceeded
186                 if (time.time() - start_time) > time_allowance:
187                     optimalSolutionFound = False
188                     break
189 
190                 currProblem = heapq.heappop(subProblems)
191                 if currProblem[0].cost < bssf.costOfRoute():
192                     # If ALL cities have been visited, e.g. if ALL values of list are false
193                     if not True in currProblem[2]:
194                         bssf = TSPSolution(currProblem[4])
195                         count +=1
196 
197                     for toCityIndex, toCity in enumerate(self.cities):
```

```
                    # Get the intersect of the cities to which the current city can reach and the cities that are still unvisited
                    reachableCitiesBools = toCity._scenario._edge_exists[currProblem[1]._index, :]
                    validRowIndices = [a and b for a, b in zip(reachableCitiesBools, currProblem[2])]
                    if validRowIndices[toCityIndex]:

                        individualPathCost = currProblem[3][currProblem[1]._index,toCityIndex]
                        nextAdjMatrix = self.applyAdjMask(copy.deepcopy(currProblem[3]), currProblem[1], self.cities[toCityIndex])
                        rowStepCost, nextAdjMatrix = self.reduceRows(nextAdjMatrix)
                        colStepCost, nextAdjMatrix = self.reduceCols(nextAdjMatrix)

                        nextPathCost = rowStepCost + colStepCost + currProblem[0].cost + individualPathCost
                        if nextPathCost < bssf.costOfRoute():
                            nextUnvisitedCities = copy.deepcopy(currProblem[2])
                            nextUnvisitedCities[toCityIndex] = False

                            path = copy.deepcopy(currProblem[4])
                            path.append(self.cities[toCityIndex])
                            nextPathPC = PriorityCount(nextPathCost, len(path), currProblem[0].order + 1)
                            # Heap tuples order: lowerBound, currCity, unvisitedCities,  adjMatrix, path
                            nextProblem = (nextPathPC, self.cities[toCityIndex], nextUnvisitedCities, nextAdjMatrix, path)
                            heapq.heappush(subProblems, nextProblem)
                            totalHeldStates+=1

            #Not including the sub-states that are implicitly pruned, just ones that have been pushed onto the heap.
            else:
                totalPrunedStates+= 1
        results['cost'] = bssf.costOfRoute()
        results['time'] = time.time() - start_time
        results['count'] = count
        results['soln'] = bssf

        #Logging the results needed for the chart so I don't have to mess with the GUI
        print("NON-GUI RESULTS")
        print("Total # of held states: ", totalHeldStates)
        print("Max # of stored states at a given time: ", maxNumStoredStates)
        print("Total # of pruned states: ", totalPrunedStates)
        print("Optimal solution found?", optimalSolutionFound)

        return results

    def fancy( self, start_time, time_allowance=60.0 ):
        pass



    def getNextCity_Greedy(self, currCity, cities, visited):
        currShortestPath = np.inf
        closestCity = None
        currCityCoords = np.array((currCity._x, currCity._y))
        for city in cities:
            if city not in visited:
                toCityCoords = np.array((city._x, city._y))
                # Gets the Euclidean distance
                pathCost = np.linalg.norm(currCityCoords - toCityCoords)
                if pathCost < currShortestPath:
                    currShortestPath = pathCost
```

```
                    closestCity = city
        return closestCity

    def generateAdjMatrix(self, cities):
        array = np.full((len(cities), len(cities)), np.inf)
        for x in range(0, len(cities)):
            for y in range(0, len(cities)):
                array[x,y] = cities[x].costTo(cities[y])
        return array


    def reduceRows(self, adjMatrix):
        rowReductionCost = 0
        for rowIndex, rowCity in enumerate(self.cities):
            #minVal = minimum of the given row
            minVal = np.amin(adjMatrix[rowIndex,:])
            if minVal < np.inf:
                rowReductionCost += minVal
                adjMatrix[rowIndex,:] -= minVal
        return rowReductionCost, adjMatrix

    def reduceCols(self, adjMatrix):
        colReductionCost = 0
        for colIndex, colCity in enumerate(self.cities):
            #minVal = minimum of the given col
            minVal = np.amin(adjMatrix[:,colIndex])
            if minVal < np.inf:
                colReductionCost += minVal
                adjMatrix[:,colIndex] -= minVal
        return colReductionCost, adjMatrix

    def applyAdjMask(self, currAdjMatrix, fromCity, toCity):
        rowToMask = fromCity._index
        colToMask= toCity._index
        for colIndex, colVal in enumerate(currAdjMatrix[rowToMask,:]):
            currAdjMatrix[rowToMask, colIndex] = np.inf
        for rowIndex, rowVal in enumerate(currAdjMatrix[:,colToMask]):
            currAdjMatrix[rowIndex, colToMask] = np.inf

        currAdjMatrix[rowToMask, colToMask] = np.inf
        currAdjMatrix[colToMask, rowToMask] = np.inf
        return currAdjMatrix

# Unique class used to server as a key for the heap. First it will consider the cost of the path,
# If those happen to be identical it will take the path that was found first according to it's order.
class PriorityCount:
    def __init__(self, cost, length, order):
        self.cost = cost
        self.length = length
        self.order = order

    def __lt__(self, other):
        if (self.cost/self.length) < (other.cost/other.length):
            return self
        elif self.cost > other.cost:
            return other
        else:
            if self.order > other.order:
                return other
            else:
                return self

```