Analysis of Greedily Fed 2-opt Heuristic Solution Finding Method to the Traveling Salesperson Problem

Adam Welker

Paul Wertenbaker

Jacob West

14 April 2021

Professor Tony Martinez

CS 312 Winter 2021

Abstract

The traveling salesman problem is a famous NP-Hard problem in the realm of computing. Due to its complex nature, heuristic solutions are often used to find useful solutions to the traveling salesman problem in reasonable time. Our team has created an adapted version of the greedy solution finding algorithm that incorporates 2 opt heuristic solution finding techniques in order to quickly arrive at a locally optimal route. The algorithm showed promising performance on problems of a small size.

Introduction

The traveling salesman problem is a well explored problem in the field of computing and computer science. The goal of this problem is to find the fastest route within a network of nodes that passes each node only once. Additionally, this route through the network must begin and end at the same node. Though seemingly simple, when analyzed, the problem quickly reveals itself to be complex and tedious to solve completely.

Viewing the problem through a mathematical lens, we can begin to solve for the complexity of finding an exact solution to the traveling salesman problem. We begin by defining a set V which contains n number of nodes in a graph. Thereafter, we also introduce set E, which contains m weighted edges that connect the nodes of set V. In a worst-case scenario, we say that the edges of E are directional and that $|E|$ is equal to $|V|*(|V| - 1)$, making each node connected to every other node. Finally, we define our ideal solution as the subset of E that connects every vertex in V with the lowest total summation of edge weights.

Analyzing our worst case, we see that the number of possible solutions to this problem is n!. This is because after exploring a node from set V, we may explore $|V| - b$ paths, where b is the number of nodes we have already visited. This classifies the traveling salesman problem as an NP-Complete problem, and thus shows the difficulty in computing the perfect solution with a brute force method.

Such difficulty as lead computer scientists to instead search for heuristic or imperfect solutions to the traveling salesman problem. Such solutions include greedy solutions, local optimums, and others. While not as ideal as the perfect solution, these heuristic approaches offer solutions that can be near optimal and extremely useful for practical computing, given their significantly better time complexities.

This report explores our team's approach to find a heuristic solution to the traveling salesman problem by adapting a greedy solving method to a 2opt solving method.

The Greedy Algorithm

A simple heuristic solution to the traveling salesman problem is the greedy approach. Greedy algorithms are common in computing; they involve solving a problem by taking the most rewarding immediate path rather than planning far in advance. While the solutions that these algorithms output are often less than optimal, they have the advantage of a reduced time complexity.

In the case of the Traveling Salesman problem, a greedy algorithm would begin with a staring node in the set *V* previously mentioned. It would then find the shortest edge in set *E* that connects to *V* and add it to a solution set *S*. It would then repeat the process at the opposite vertex in the previously selected edge. Finally, this process is repeated until the edges in *S* completely cover the set *V*. The set *S* is then returned as the solution to the problem.

As we can see, there are several disadvantages to this method of solution finding. As previously mentioned, the greedy approach only finds one solution and makes no attempt to compare it to others, making a locally optimal solution at best and at worst a randomly guessed solution. Additionally, if the algorithm arrives at a dead end, or a node that only leads to previously visited vertices, it will not find any solution.

However, the greedy method is extremely advantageous in terms of time and memory complexity. Its simple nature enables solutions to large graphs to be found in extremely fast and reasonable time. For this reason, our team began finding a solution to the traveling salesman problem using the following greedy algorithm:

```
function greedy(V,E):
        S = {V[0]}
        iter = V[0]4
        while(|S| < |V|)
                cost = []
                queue = heapq
        for i in 0:|V|:
                if S contains V[i]:
                        cost[i] = inf
                else:
                        cost[i] =
                        iter.costTo(V[i])

        queue.push(cost, V)
        iter = queue.delete_min
        S.add(iter)

        Return s
End
```

We can see from this algorithm that the time complexity of the greedy approach works out to be $O(n^2)$, because it must do n evaluations of length for all n vertices in set *V*. This phenomenal time complexity makes the greedy algorithm useful, because it out paces many intelligent search approaches such as the Branch and Bound method ($O(n^2n!)$). Despite in losses accuracy of solutions, the greedy method remains an elegant and fast solution to the Traveling Salesman Problem.

The 2opt Correction Approach

Our team's primary goal in this project was to encounter a way to better the solutions found by the greedy algorithm while maintaining its competitive speed. One of the principle purposes in finding heuristic solutions is expedite the solution finding process. While intelligent searches like branch and bound are very useful in finding accurate solutions, their worst-case time complexity remains equal to the factorial based efficiency of a brute force solution finding method. We believed that a better

approach to finding accurate solutions to the Traveling Salesman problem in reasonable time would be to build off the greedy algorithm in attempt to find a more accurate solution within the same Big-O time complexity.

The approach that we chose to use to reach this goal was to adapt the our greed algorithm to use a 2 opt optimization process, hoping to improve our greedy solutions after finding them.

The 2 opt algorithm is a well-known way to improve a known solution to the traveling salesman problem. It involves switching adjacent edges in the solution set *S* and evaluating the new solution set. If the solution set has a lower cost, then the new set becomes *S*, and the process is repeated[i]. This can be done until a set is found in which any combination of adjacent swaps does not produce a more efficient solution, at which point a local minimum has been found. The algorithm uses the following pseudocode:

```
function 2opt (S)
       solutionFound = true
       while solutionFound:
              solutionFound = false
              for i in |S|:
                 S' = S
                 swap(S'[i, S'[i+1])
                 if cost(S') > cost(S):
                     S = S'
                     solutionFound= true
                     break
       return S
end
```

As we can observe from the pseudocode, this algorithm for the 2 opt solution has a factorial time complexity and memory complexity, given in a worst case, the 2 opt process will calculate every possible route. However, this rarely happens. More likely, a maximum of *n* or less combinations will be found and tried Thus, it is safe to assume that the 2 opt algorithm is able to be preformed in quadratic time comparable to the greedy algorithm. This is because *n* solutions must be tried *n* times within the main while loop.

Given the similar time complexity to the greedy algorithm, we chose to use the following approach, feeding the results of our greedy algorithm into a 2 opt optimizer:

```
function GreedyFed2opt (E,V)
       S = Greedy(E,V)
       S = 2opt(S)
       return S
end
```

Our team implemented the 2opt solution finding technique in python along with the greedy algorithm, and then preformed experiments where results were compared between the original greedy solution and the solution found by feeding the greedy route into the 2opt function. These were also compared to the accuracy and speed of a traditional branch and bound solution and a randomly found route. We did this attempting to prove that the greedy fed 2opt function could significantly improve the results of a greedy algorithm without sacrificing time complexity.

Results

For empirical testing, a control (random) solution was tested against the three previously mentioned approaches for 5

different problems of equal size. This was repeated for 6 different sizes of problem. Averages were taken for the route cost and time of computation for each set of trials, and the data was tabulated. The next pages contain the tabulated results as well as screen shots of each algorithm functioning within the GUI environment where it was tested.

Tabulated Results

	Random		Greedy			Branch and Bound			Greedy fed 2opt		
# Cities	Time (sec)	Path Length	Time (sec)	Path Length	% of Random	Time (sec)	Path Length	% of Greedy	Time (sec)	Path Length	% of Greedy
15	0.000744	20642	0.000687	9673	0.468608	22.18676	4662	0.48196	0.02458	8828	0.912643
30	0.01356	39545	0.000917	19016	0.48087	600.0395	7491	0.393931	0.495755	16231	0.853544
40	0.435658	53522	0.001564	21131	0.39481	600.1651	10616	0.50239	0.427557	21161	1.00142
60	35.51058	82114	0.003269	27418	0.333902	600.608	13691	0.499343	3.602504	26243	0.957145
100	TB	inf	0.009162	38687	0	602.6763	21815	0.563885	35.83741	37555	0.97074

Images of Testing

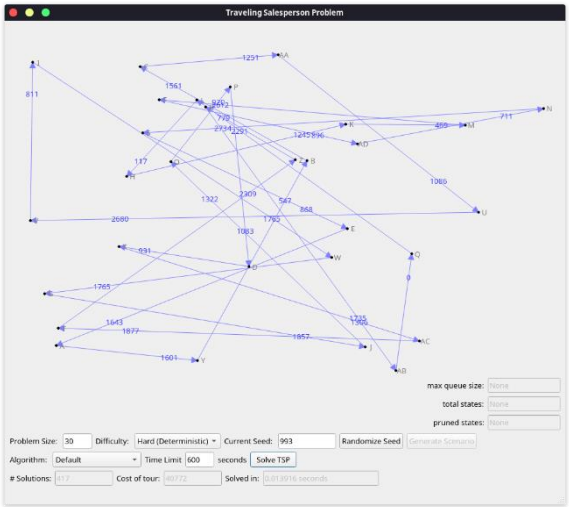

Figure 1: Random solution to a problem of 30 cities

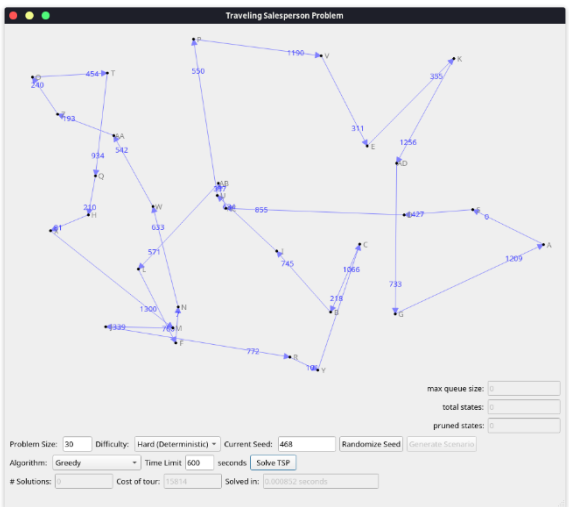

Figure 2: Greedy solution to a problem of 30 cities

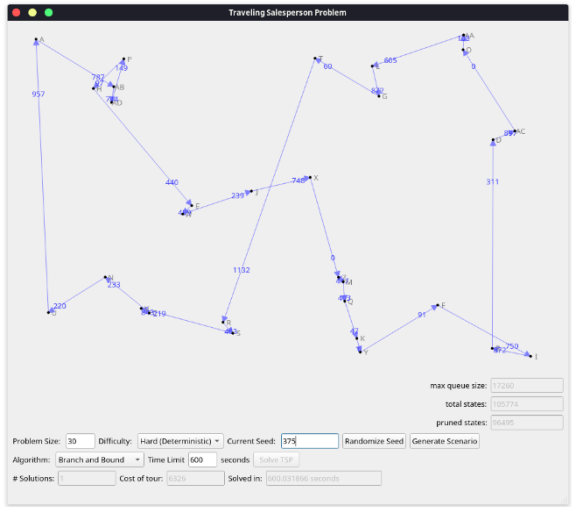

Figure 3: Branch and Bound solution to problem size of 30 cities

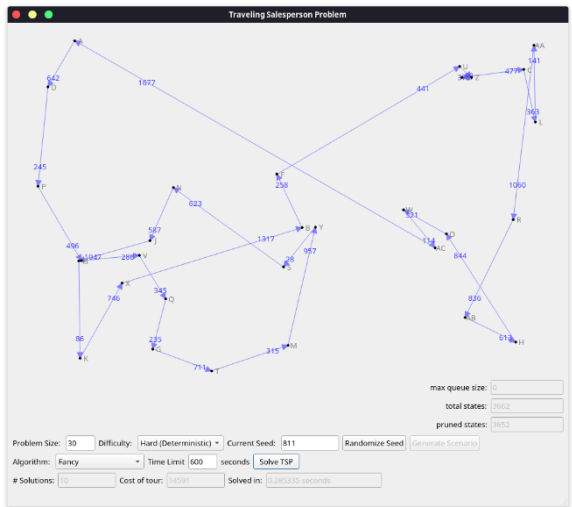

Figure 4: 2opt solution to a problem size of 30 cities

Analysis

The results we received after empirical testing were mixed. As was predicted before, we did see that the 2 opt solution consistently found better routes than the greedy solution. In all but one trial, our two 2opt solution was able to find routes that were close to 5% less costly than its greedy counterpart. This however is overshadowed by the performance of the intelligent search, which found results typically upwards of 50% less costly than the greedy approach. Fortunately, all algorithms produced significantly better results than the random route selection process.

Unfortunately, the 2 opt solution did not perform as expected in terms of time complexity. The assumption that the 2 opt function would quickly settle on to a local minimum was simply not true as the number of cities increased. Hence, as the problem size increases, we see an exponential growth of the time taken to find the 2 opt heuristic solution.

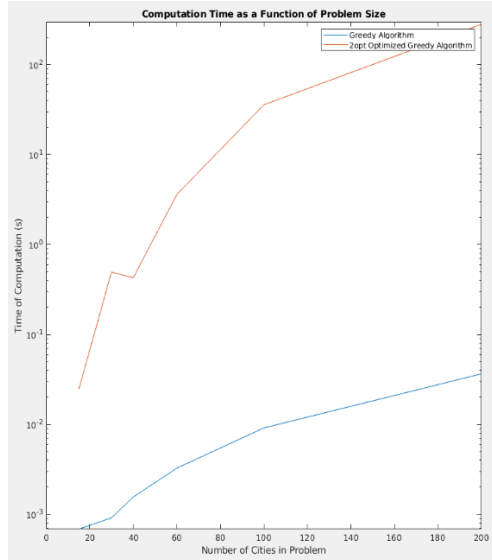

Figure 5: Comparison of Computation times between the greedy and 2 opt algorithms

As we clearly notice from the graph, the 2 opt time complexity did not closely follow the time complexity of the greedy algorithm. Our team ran a regression of the empirical data in MATLAB to find that the time complexity of the greedy fed 2opt could be estimated to be $\Theta(0.0068 * n^2)$ with an r^2 value of about 0.978. While this is still

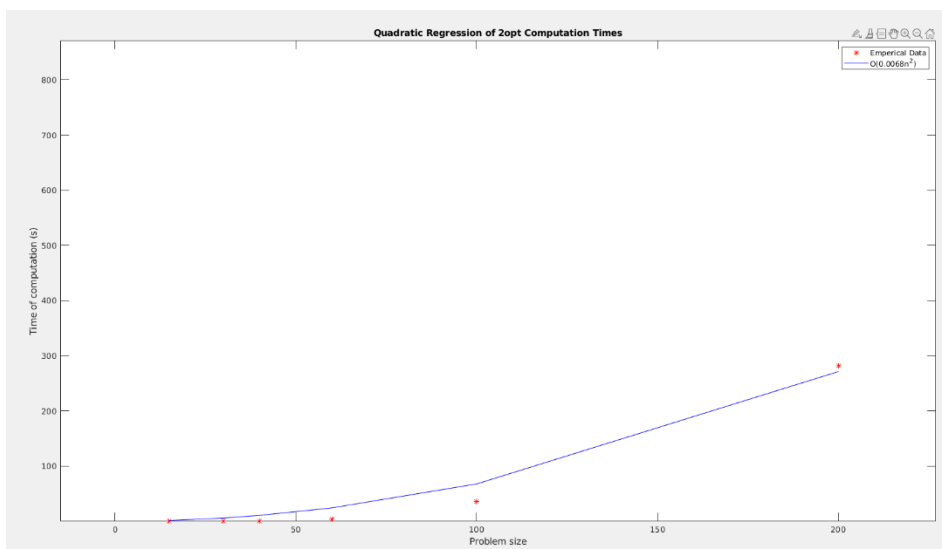

Figure 6: Quadratic Regression of Empirical 2 opt Time Data

within $O(n^2)$, it is grossly larger than the time complexity of the greedy solution.

We can also identify the optimal range of problem size for the greedy 2opt algorithm, which seems to be between 15-30 cities, due to best percent increases in route efficiency over the purely greedy solution found in this range. After 30 cities, our heuristic process appears to have diminished returns.

Conclusion and Future Work

We see from our analysis that the greedy fed 2 opt solution was in fact, consistently able to find better solutions in reasonably fast time. This, however, was done at the great cost of less efficient routes than an intelligent search could produce. We may simplify these findings down to the following pros and cons list:

Pros:

- Solutions are found much quicker than a branch and bound approach
- Consistently finds better results that a purely greedy algorithm

Cons:

- Finds significantly less efficient routes than intelligent search
- Takes significantly longer than a greedy algorithm
- Experiences diminishing returns of route quality to time of computation as number of nodes increases beyond ~30.

Additionally, these results open the way for future work. Given the 2 opt algorithm's effectiveness when paired with a greedy algorithm, we might consider the following directions for future research:

- Feeding the 2 opt algorithm solution sets found by the branch and bound algorithm
- Increasing the depth of search of the greedy algorithm to include 2-3 steps deeper and pairing these results with the 2 opt algorithm
- Coming up with new stopping criterion for 2 opts' main loop to improve run time performance

Overall, we consider the development of this algorithm a success, and have found that the greedy fed 2 opt heuristic solution is a viable way to find locally optimal solutions to the traveling salesman problem.

---

[i] Venhuis, Mikko. "The basics of search algorithms explained with intuitive visualizations" *Towards data science.* https://towardsdatascience.com/around-the-world-in-90-414-kilometers-ce84c03b8552. Accessed March 2021.

Appendix

The pages which follow contain all the source code needed to replicate this project.

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

import math
import random
import signal
import sys
import time


from which_pyqt import PYQT_VER
if PYQT_VER == 'PYQT5':
    from PyQt5.QtWidgets import *
    from PyQt5.QtGui import *
    from PyQt5.QtCore import *
# elif PYQT_VER == 'PYQT4':
#   from PyQt4.QtGui import *
#   from PyQt4.QtCore import *
else:
    raise Exception('Unsupported Version of PyQt: {}'.format(PYQT_VER))


#TODO: Error checking on txt boxes
#TODO: Color strings


# Import in the code with the actual implementation
from TSPSolver import *
#from TSPSolver_complete import *
from TSPClasses import *


class PointLineView( QWidget ):
    def __init__( self, status_bar, data_range ):
        super(QWidget,self).__init__()
        self.setMinimumSize(950,600)

        self.pointList  = {}
        self.edgeList   = {}
        self.labelList   = {}
        self.status_bar = status_bar
```

```
        self.data_range = data_range
        self.start_pt = None
        self.end_pt = None

    def displayStatusText(self, text):
        self.status_bar.showMessage(text)

    def clearPoints(self):
        self.pointList = {}

    def clearEdges(self,removeColors = None):
        self.edgeList = {}
        if removeColors:
# allows removal of edge labels without removing node labels, for example
            for color in removeColors:
                if color in self.labelList:
                    del self.labelList[color]
        else:
            self.labelList = {}
        self.repaint()

    def addPoints( self, point_list, color ):
        if color in self.pointList:
            self.pointList[color].extend( point_list )
        else:
            self.pointList[color] = point_list

#   def setStartLoc( self, point ):
#       self.start_pt = point
#       self.repaint()
#
#   def setEndLoc( self, point ):
#       self.end_pt = point
#       self.repaint()


    def addEdge( self, startPt, endPt, label, edgeColor, labelColor=None, xoffset=0.0 ):
```

```
        if not labelColor:
            labelColor = edgeColor

        assert( type(startPt) == QPointF )
        assert( type(endPt)   == QPointF )
        assert( type(label)   == str )

        edge = QLineF(startPt, endPt)
        if edgeColor in self.edgeList.keys():
            self.edgeList[edgeColor].append( edge )
        else:
            self.edgeList[edgeColor] = [edge]

        midp = QPointF( (edge.x1()*0.2 + edge.x2()*0.8),
                        (edge.y1()*0.2 + edge.y2()*0.8) )
        self.addLabel( midp, label, labelColor, xoffset=xoffset )

    def addLabel( self, point, label, labelColor, xoffset=0.0 ):
        if labelColor in self.labelList.keys():
            self.labelList[labelColor].append( (point,label,xoffset) )
        else:
            self.labelList[labelColor] = [(point, label,xoffset)]




    def paintEvent(self, event):
        painter = QPainter(self)
        painter.setRenderHint(QPainter.Antialiasing,True)

        xr = self.data_range['x']
        yr = self.data_range['y']
        w = self.width()
        h = self.height()
```

```
        w2h_desired_ratio = (xr[1]-xr[0])/(yr[1]-yr[0])
        if w / h < w2h_desired_ratio:
             scale = w / (xr[1]-xr[0])
        else:
             scale = h / (yr[1]-yr[0])

        tform = QTransform()
        tform.translate(self.width()/2.0,self.height()/2.0)
        tform.scale(1.0,-1.0)
        painter.setTransform(tform)

        for color in self.edgeList:
            c = QColor(color[0],color[1],color[2])
            painter.setPen( c )
            for edge in self.edgeList[color]:
                ln = QLineF( scale*edge.x1(), scale*edge.y1(), scale*edge.x2(), scale*edge.y2() )
                painter.drawLine( ln )

        for color in self.edgeList:
            c = QColor(color[0],color[1],color[2])
            painter.setPen( c )
            for edge in self.edgeList[color]:
                #arrow_scale = .015
                arrow_scale = 5.0
                unit_edge = ( edge.x2() - edge.x1(), edge.y2() - edge.y1() )
                unit_edge_mag = math.sqrt( ( edge.x2() - edge.x1())**2 + ( edge.y2() - edge.y1() )**2 )
                unit_edge = (unit_edge[0] / unit_edge_mag, unit_edge[1] / unit_edge_mag )
                unit_edge_perp = (-unit_edge[1], unit_edge[0])

                temp_tform = QTransform()
                temp_tform.translate(self.width()/2.0,self.height()/2.0)
                temp_tform.scale(1.0,-1.0)
```

```
                temp_tform.translate(scale*edge.x2(),scale*edge.y2())
                temp_tform.scale(1.0,-1.0)
                painter.setTransform(temp_tform)
                #painter.drawText( RECT, label[1], align )

                tri_pts = []
                tri_pts.append( QPointF(0,0) )
                tri_pts.append( QPointF(-arrow_scale*(2*unit_edge[0] + unit_edge_perp[0]),
                                                arrow_scale*(2*unit_edge[1] + unit_edge_perp[1])) )
                tri_pts.append( QPointF(-arrow_scale*(2*unit_edge[0] - unit_edge_perp[0]),
                                                arrow_scale*(2*unit_edge[1] - unit_edge_perp[1])) )
                tri = QPolygonF( tri_pts )
                b = painter.brush()
                painter.setBrush( c )
                painter.drawPolygon( tri )
                painter.setBrush( b )

        painter.setTransform(tform)
        font = QFont("Monospace")
        font.setStyleHint(QFont.TypeWriter)

        R = 1.0E3
        CITY_SIZE = 2.0 # DIAMETER
        RECT = QRectF(-R,-R,2.0*R,2.0*R)
        align = QTextOption( Qt.Alignment(Qt.AlignHCenter | Qt.AlignVCenter) )
        for color in self.labelList:
            c = QColor(color[0],color[1],color[2])
            painter.setPen( c )
            for label in self.labelList[color]:
                temp_tform = QTransform()
                temp_tform.translate(self.width()/2.0,self.height()/2.0)
                temp_tform.scale(1.0,-1.0)
                pt = label[0]
```

```
                xoff = label[2]
                temp_tform.translate(scale*pt.x()+xoff,scale*pt.y())
                temp_tform.scale(1.0,-1.0)
                painter.setTransform(temp_tform)
                painter.drawText( RECT, label[1], align )

        painter.setTransform(tform)
        for color in self.pointList:
            c = QColor(color[0],color[1],color[2])
            painter.setPen( c )
            b = painter.brush()
            painter.setBrush(c)
            for point in self.pointList[color]:
                pt = QPointF(scale*point.x(), scale*point.y())
                painter.drawEllipse( pt, CITY_SIZE, CITY_SIZE)
            painter.setBrush(b)



class Proj5GUI( QMainWindow ):

    def __init__( self ):
        super(Proj5GUI,self).__init__()

        self.RED_STYLE   = "background-color: rgb(255, 220, 220)"
        self.PLAIN_STYLE = "background-color: rgb(255, 255, 255)"
        self._MAX_SEED = 1000

        self._scenario = None
        self.initUI()
        self.solver = TSPSolver( self.view )
        self.genParams = {'size':None,'seed':None,'diff':None}


```

```python

    def newPoints(self):
        # TODO - ERROR CHECKING!!!!
        seed = int(self.curSeed.text())
        random.seed( seed )

        ptlist = []
        RANGE = self.data_range
        xr = self.data_range['x']
        yr = self.data_range['y']
        npoints = int(self.size.text())
        while len(ptlist) < npoints:
            x = random.uniform(0.0,1.0)
            y = random.uniform(0.0,1.0)
            if True:
                xval = xr[0] + (xr[1]-xr[0])*x
                yval = yr[0] + (yr[1]-yr[0])*y
                ptlist.append( QPointF(xval,yval) )
        return ptlist

    def generateNetwork(self):
        points = self.newPoints() # uses current rand seed
        diff = self.diffDropDown.currentText()
        rand_seed = int(self.curSeed.text())
        self._scenario = Scenario( city_locations=points, difficulty=diff, rand_seed=rand_seed )

        self.genParams = {'size':self.size.text(),'seed':self.curSeed.text(),'diff':diff}
        self.view.clearEdges()
        self.view.clearPoints()

        self.addCities()



    def addCities( self ):
        cities = self._scenario.getCities()
        self.view.clearEdges()
        for city in cities:
```

```
            self.view.addLabel( QPointF(city._x, city._y), city._name, \
                                labelColor=(128,128,128), xoffset=10.0 )

    def generateClicked(self):
        self.generateNetwork()
        self.view.addPoints( [QPointF(c._x,c._y) for c in self._scenario.getCities()], (0,0,0) )
        self.solveButton.setEnabled(True)
        self.graphReady = True
        self.checkGenInputs()
        self.numSolutions.setText( '--' )
        self.tourCost.setText( '--' )
        self.solvedIn.setText( '--' )
        self.maxQSize.setText( '--' )
        self.totalStates.setText( '--' )
        self.prunedStates.setText( '--' )
        self.statusBar.showMessage('')
        self.view.repaint()


    def displaySolution( self ) :                        # what about calling this somehow every time a new bssf is found?
        self.view.clearEdges([(64,64,255)])             # get rid of edge labels but not point labels
        if self._solution:
            self.addCities()
            edges = self._solution.enumerateEdges()
            if edges:
                edgeColor  = (128,128,255)
                labelColor = (64,64,255)
                for edge in edges:
                    pt1,pt2,label = edge
                    self.view.addEdge( QPointF(pt1._x,pt1._y), \
                                       QPointF(pt2._x,pt2._y), \
                                       '{}'.format(
```

```
label), edgeColor, labelColor )
        else:
            self.statusBar.showMessage('No Solution Found.')
        self.view.repaint()


    def randSeedClicked(self):
        new_seed = random.randint(0, self._MAX_SEED-1)
        self.curSeed.setText( '{}'.format(new_seed) )
        self.view.repaint()

    def solveClicked(self):                             # need to reset display??? and say "processing..." at bottom???
        self.solver.setupWithScenario(self._scenario)

        max_time = float( self.timeLimit.text() )
        # TODO - start on a separate thread
        self.view.clearEdges([(64,64,255)])             # get rid of edge labels but not point labels
        self.numSolutions.setText( '--' )
        self.tourCost.setText( '--' )
        self.solvedIn.setText( '--' )
        self.maxQSize.setText( '--' )
        self.totalStates.setText( '--' )
        self.prunedStates.setText( '--' )
        self.statusBar.showMessage('Processing...')
        #self.view.repaint()
        #app.processEvents()
        solve_func = 'self.solver.'+self.ALGORITHMS[self.algDropDown.currentIndex()][1]
        results = eval(solve_func)(time_allowance=max_time )
        if results:
            self.statusBar.showMessage('')
            self.numSolutions.setText( '{}'.format(
```

```
results['count']) )
            self.tourCost.setText( '{}'.format(results['cost']) )
            self.solvedIn.setText( '{:6.6f} seconds'.format(results['time']) )
            self._solution = results['soln']
            if 'max' in results.keys():
                self.maxQSize.setText( '{}'.format(results['max']))
            if 'total' in results.keys():
                self.totalStates.setText( '{}'.format(results['total']))
            if 'pruned' in results.keys():
                self.prunedStates.setText( '{}'.format(results['pruned']))
            #if self._solution:
            self.displaySolution()
        else:
            print( 'GOT NULL SOLUTION BACK!!' )      #probably shouldn't ever use this...
        self.view.repaint()
#       app.processEvents()

    def checkGenInputs(self):
        seed  = self.curSeed.text()
        size = self.size.text()
        diff = self.diffDropDown.currentText()

        if self._scenario:
            if self.genParams['seed'] == seed and \
               self.genParams['size'] == size and \
               self.genParams['diff'] == diff:
                self.generateButton.setEnabled(False)
                self.solveButton.setEnabled(True)
            elif (seed == '') or (size == ''):
                self.generateButton.setEnabled(False)
                self.solveButton.setEnabled(True)
            else:
                self.generateButton.setEnabled(True
```

```
)
                self.solveButton.setEnabled(False)


    def checkInputValue(self, widget, validrange):
        assert( type(widget) == QLineEdit )
        retval = None
        valid  = False
        try:
            sval = widget.text()
            if sval == '':
                valid = True
            else:
                ival = int(sval)
                if validrange:
                    if ival >= validrange[0] and ival <= validrange[1]:
                        retval = ival
                        valid = True
        except:
            pass

        if not valid:
            widget.setStyleSheet( self.RED_STYLE )
        else:
            widget.setStyleSheet( '' )

        return '' if retval==None else retval

    ALGORITHMS = [ \
        ('Default                              ','defaultRandomTour'), \
        ('Greedy','greedy'), \
        ('Branch and Bound','branchAndBound'), \
        ('Fancy','fancy') \
    ]
                # whitespace hack to get longest to display correctly

    def initUI( self ):
        self.setWindowTitle('Traveling Salesperson
```

```
Problem')
        self.setWindowIcon( QIcon('icon312.png') )

        self.statusBar = QStatusBar()
        self.setStatusBar( self.statusBar )

        vbox = QVBoxLayout()
        boxwidget = QWidget()
        boxwidget.setLayout(vbox)
        self.setCentralWidget( boxwidget )


        SCALE = 1.0
        self.data_range     = { 'x':[-1.5*SCALE,1.5*SCALE], \
                                'y':[-SCALE,SCALE] }
        self.view           = PointLineView( self.statusBar, \
                                             self.data_range )
        self.randSeedButton = QPushButton('Randomize Seed')
        self.generateButton = QPushButton('Generate Scenario')
        self.solveButton    = QPushButton('Solve TSP')

        self.curSeed    = QLineEdit('20')
        self.curSeed.setFixedWidth(100)
        self.size           = QLineEdit('15')
        self.size.setFixedWidth(50)
        self.timeLimit      = QLineEdit('60')
        self.timeLimit.setFixedWidth(50)
        self.numSolutions   = QLineEdit('--')
        self.numSolutions.setFixedWidth(100)
        self.tourCost       = QLineEdit('--')
        self.tourCost.setFixedWidth(100)
        self.solvedIn       = QLineEdit('--')
        self.solvedIn.setFixedWidth(200)
        self.maxQSize       = QLineEdit('--')
```

```
        #self.maxQSize.setFixedWidth(100)
        self.totalStates    = QLineEdit('--')
        #self.totalStates.setFixedWidth(200)
        self.prunedStates   = QLineEdit('--')
        #self.prunedStates.setFixedWidth(200)

        self.diffDropDown   = QComboBox(self)
        self.algDropDown    = QComboBox(self)

        h = QHBoxLayout()
        h.addWidget( self.view )
        vbox.addLayout(h)

        h = QHBoxLayout()
        h.addStretch(1)
        h.addWidget( QLabel( 'max queue size:' ) )
        h.addWidget( self.maxQSize )
        self.maxQSize.setEnabled(False)
        vbox.addLayout(h)

        h = QHBoxLayout()
        h.addStretch(1)
        h.addWidget( QLabel( 'total states:' ) )
        h.addWidget( self.totalStates )
        self.totalStates.setEnabled(False)
        vbox.addLayout(h)

        h = QHBoxLayout()
        h.addStretch(1)
        h.addWidget( QLabel( 'pruned states:' ) )
        h.addWidget( self.prunedStates )
        self.prunedStates.setEnabled(False)
        vbox.addLayout(h)


        h = QHBoxLayout()
        h.addWidget( QLabel('Problem Size: ') )
        h.addWidget( self.size )
        h.addWidget( QLabel('Difficulty: ') )
        h.addWidget( self.diffDropDown )
        h.addWidget( QLabel('Current Seed: ') )
```

```
        h.addWidget( self.curSeed )
        h.addWidget( self.randSeedButton )
        h.addWidget( self.generateButton )
        h.addStretch(1)
        vbox.addLayout(h)

        h = QHBoxLayout()
        h.addWidget( QLabel('Algorithm: ') )
        h.addWidget( self.algDropDown )
        h.addWidget( QLabel( 'Time Limit' ) )
        h.addWidget( self.timeLimit )
        h.addWidget( QLabel( 'seconds' ) )
        h.addWidget( self.solveButton )
        h.addStretch(1)
        vbox.addLayout(h)

        h = QHBoxLayout()
        h.addWidget( QLabel( '# Solutions:' ) )
        h.addWidget( self.numSolutions )
        h.addWidget( QLabel( 'Cost of tour:' ) )
        h.addWidget( self.tourCost )
        h.addWidget( QLabel( 'Solved in:' ) )
        h.addWidget( self.solvedIn )
        self.numSolutions.setEnabled(False)
        self.tourCost.setEnabled(False)
        self.solvedIn.setEnabled(False)
        h.addStretch(1)
        vbox.addLayout(h)


        self.lastPath = (None,None)
        self.solveButton.setEnabled(False)

        self.curSeed.textChanged.connect(self.checkGenInputs)
        self.size.textChanged.connect(self.checkGenInputs)

        self.randSeedButton.clicked.connect(self.randSeedClicked)
        self.generateButton.clicked.connect(self.
```

```
generateClicked)
        self.solveButton.clicked.connect(self.solveClicked)

        self.diffDropDown.addItem('Easy                                '
)                   # Weird hack to make box wide enough to show all of last item
        self.diffDropDown.addItem('Normal')
        self.diffDropDown.addItem('Hard')
        self.diffDropDown.addItem('Hard (Deterministic)')
        self.diffDropDown.activated.connect(self.diffChanged)
        self.diffDropDown.setCurrentIndex(3)
        self.diffChanged(3) # to handle start state

        for alg in self.ALGORITHMS:
            self.algDropDown.addItem( alg[0] )
        self.algDropDown.activated.connect(self.algChanged)
        self.algDropDown.setCurrentIndex(2)
        self.algChanged(2) # to handle start state

        self.graphReady = False

        self.show()


    def diffChanged(self, text):
        self.checkGenInputs()

    def algChanged(self, text):
        pass



if __name__ == '__main__':
    # This line allows CNTL-C in the terminal to kill the program
    signal.signal(signal.SIGINT, signal.SIG_DFL)
```

```

    app = QApplication(sys.argv)
    w = Proj5GUI()
    sys.exit(app.exec())

```

```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 time
import numpy as np
from TSPClasses import *
from CityWrapper import *
import heapq
import itertools
INFINITY = math.inf


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.  Note this could be used to find your
        initial BSSF.
        </summary>
        <returns>results dictionary for GUI that contains three ints: cost of solution,
        time spent to find solution, number of permutations tried during search, the
        solution found, and three null values for
```

```
fields not used for this
        algorithm</returns>
    '''

    def defaultRandomTour( self, time_allowance=60.0 ):
        results = {}
        cities = self._scenario.getCities()
        ncities = len(cities)
        foundTour = False
        count = 0
        bssf = None
        start_time = time.time()
        while not foundTour and time.time()-start_time < time_allowance:
            # 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)
            count += 1
            if bssf.cost < np.inf:
                # Found a valid route
                foundTour = True
        end_time = time.time()
        results['cost'] = bssf.cost if foundTour else math.inf
        results['time'] = end_time - start_time
        results['count'] = count # Number of solutions discovered.
        results['soln'] = bssf # Object containing the route.
        results['max'] = None # Max size of the queue.
        results['total'] = None # Total states generated.
        results['pruned'] = None # Number of states pruned.
```

```
        return results


    ''' <summary>
        This is the entry point for the greedy solver, which you must implement for
        the group project (but it is probably a good idea to just do it for the branch-and
        bound project as a way to get your feet wet).  Note this could be used to find your
        initial BSSF.
        </summary>
        <returns>results dictionary for GUI that contains three ints: cost of best solution,
        time spent to find best solution, total number of solutions found, the best
        solution found, and three null values for fields not used for this
        algorithm</returns>
    '''

    # Time complexity: O(n^2) because we compare every city to every other city when
    #                  we are trying to find the min cost.
    # Space complexity: O(n). Because of a map of size n called visitedCities,
    #                   a heapQueue that is of size n (worst case),
    #                   and a route list of size n.
    def greedy(self, time_allowance=60.0):

        results = {}
        self._cities = self._scenario.getCities()
        visitedCities = {}

        self._startTime = time.time()
        self._time_allowance = time_allowance
        route = []
        route.append(self._cities[0])  # Add starting city to the route
```

```
        visitedCities[self._cities[0]._name] = True # Prevent us from visiting our starting city

        runningCost = 0

        # For every city, get the minimal cost to another city and add it to our route
        i = 0
        counter = 0
        # This is O(n^2) because of an n size loop inside of an n size loop
        while (counter < len(self._cities)):
            currCity = self._cities[i]
            heapFromCurrCity = []

            # Find costs from current city to every other city
            for j in range(len(self._cities)):
                tempCity = self._cities[j]
                try:
                    isVisited = visitedCities[tempCity._name]  # Will not throw exception if city has
                    # been visited. Therefore, skip it.
                except:
                    cost = currCity.costTo(tempCity)
                    wrappedCity = CityWrapper(cost, tempCity, j)
                    heapq.heappush(heapFromCurrCity, wrappedCity)

            # Obtain the closest city, make it impossible to visit in the future, and add it to the route.
            if len(heapFromCurrCity) == 0:
                # currCity is the last city. We are done.
                break
            wrappedCity = heapq.heappop(
```

```
heapFromCurrCity)
            closestCityToCurrentCity = wrappedCity._city
            cost = wrappedCity._cost

            i = wrappedCity._indexInCities
            visitedCities[closestCityToCurrentCity._name] = True  # Prevent us from visiting the closest city
            # in future calculations
            route.append(closestCityToCurrentCity)
            runningCost += cost
            counter += 1

        endTime = time.time()
        bssf = TSPSolution(route)
        bssf.cost = INFINITY
        if len(visitedCities.keys()) == len(self._cities):
            bssf.cost = runningCost

        timePassed = endTime - self._startTime

        results['time'] = timePassed
        results['cost'] = bssf.cost
        results['count'] = 0  # Number of solutions discovered. Will always be 1 for the greedy solution
        results['soln'] = bssf  # Object containing the route.
        results['max'] = 0  # Max size of the queue. Will always be 0 for the greedy solution
        results['total'] = 0  # Total states generated. Will always be 0 for the greedy solution
        results['pruned'] = 0  # Number of states pruned. Will always be 0 for the greedy solution

        return results


    ''' <summary>
```

```
        This is the entry point for the branch-and-bound algorithm that you will implement
        </summary>
        <returns>results dictionary for GUI that contains three ints: cost of best solution,
        time spent to find best solution, total number solutions found during search (does
        not include the initial BSSF), the best solution found, and three more ints:
        max queue size, total number of states created, and number of pruned states.</returns>
    '''

    def branchAndBound( self, time_allowance=60.0 ):
        pass



    ''' <summary>
        This is the entry point for the algorithm you'll write for your group project.
        </summary>
        <returns>results dictionary for GUI that contains three ints: cost of best solution,
        time spent to find best solution, total number of solutions found during search, the
        best solution found.  You may use the other three field however you like.
        algorithm</returns>
    '''
    # method that executes the 2 opt solve
    # PSEUDOCODE
    # repeat until no improvement is made or time runs out {
    #    start_again:
    #    best_distance = calculateTotalDistance( existing_route)
    #    for (i = 1; i < number of nodes eligible to be swapped; i++) {
    #       for (k = i + 1; k ≤ number of nodes
```

```
eligible to be swapped; k++) {
    #           new_route = 2optSwap(existing_route, i, k)
    #           new_distance = calculateTotalDistance(new_route)
    #           if (new_distance < best_distance) {
    #               existing_route = new_route
    #               best_distance = new_distance
    #               goto start_again
    #           }
    #       }
    #   }
    # }
    # Time Complexity: O(n^4) because we have two nested for loops that iterate just under n times.
    #                  Inside of the double nested for loop, we call twoOptSwap() which is an O(n)
    #                  operation function. Surrounding all of this is a while loop that runs for as many
    #                  times as we find an improved solution (worst case is O(n)). Therefore, the total
    #                  time complexity is O(n^4).
    # Space Complexity: O(n^2) because we have a list called self._results['soln'].route object that
    #                   holds a unique routes that we generate from the twoOptSwap() function.
    #                   The worst case scenario is such that we generate n unique solutions for n cities,
    #                   thus, bringing the resulting worst case space complexity up to O(n^2).
    def fancy(self, time_allowance=60.0):
        self.initResults(time_allowance) # sets self_results to be equal to the greedy results
        self._generatedRoutes = {}  # A map used to track unique routes our algorithm generates
        self._generatedRoutes[str(self._results['soln'].route)] = True  # Add the first unique route

```

```
                        # (found from the greedy solution)

                        # to self._generatedRoutes
        self._startTime = time.time() # Initializing a global variable we will reference in a time checker function

        improvementMade = True

        # This while loop increases our time complexity to O(n^4) because it wraps an O(n^3) time complexity
        # operation. This while loop will iterate for as many times as we find a cheaper route (worst case is
        # n cheaper routes)
        while improvementMade and not self.isTimeUp():
            improvementMade = False
            existing_route = self._results['soln'].route
            best_distance = self._results['soln'].cost
            numNodesEligibleToBeSwapped = len(existing_route) - 1 # All nodes are eligible, except for the pivot node.

                # Therefore, elibible nodes = all cities - 1
            # This for loop makes the time complexity be O(n^3) because it performs O(n^2) operations
            # just under n times.
            for i in range(1, numNodesEligibleToBeSwapped):
                # This for loop makes the time complexity to be O(n^2)
                for k in range(1, numNodesEligibleToBeSwapped + 1):

```

```
                    if self.isTimeUp():
                        return self.wrapThingsUp()

                    # O(n) function call.
                    new_route, new_distance, isUniqueRoute = self.twoOptSwap(existing_route, i, k)
                    if new_distance < best_distance:
                        self._results['total'] += 1 # Because we have generated a unique new state
                        self._results['count'] += 1 # Update the count of solutions discovered.
                        self._results['soln'].route = new_route
                        self._results['soln'].cost = new_distance
                        improvementMade = True
                        i = numNodesEligibleToBeSwapped # Just to force us to leave the i loop
                        break # Just to force us to leave the k loop
                    elif isUniqueRoute:
                        self._results['pruned'] += 1 # A route is 'pruned' only if it is unique
                        self._results['total'] += 1 # Because we have generated a unique new state

        return self.wrapThingsUp()

    def wrapThingsUp(self):
        self._results['cost'] = self._results['soln'].cost
        return self._results

    # Time Complexity: O(1) because the City.costTo(OtherCity) function is O(1).
    # Space Complexity: O(1) because we are storing only an integer :)
    def updateRunningCost(self, new_route,
```

```
runningCost):
        childCityIndex = len(new_route) - 1
        parentCityIndex = len(new_route) - 2
        runningCost += new_route[parentCityIndex].costTo(new_route[childCityIndex])
        return runningCost

    # Checks self._generatedRoutes to see if our new_route is actually a route that we have
    # created before, and have happened to re-create. This function is useful for updating
    # our total states created and total states pruned counts.
    #
    # Time Complexity: O(1) because we are performing only an get and an insertion into a map
    # Space Complexity: O(n) because in the end we will have a map of size n cities
    def checkIsUniqueRoute(self, new_route):
        isUniqueRoute = False

        try:
            existsInGeneratedRoutes = self._generatedRoutes[str(new_route)]
        except:
            # It threw an error because the key does not exist in self._generatedRoutes
            # therefore, it is a unique route
            # Add the unique route to self._generatedRoutes and then return True
            self._generatedRoutes[str(new_route)] = True
            isUniqueRoute = True

        return isUniqueRoute


    # Builds a swapped route, calculates its cost, and checks to see if this newly
    # created swapped route is unique or if it has been created before.
```

```
    #
    # Time Complexity: O(n) because we are adding every item from route to new_route, only
    #                   in a newly arranged manner.
    # Space Complexity: O(n) because we are building a new list (new_route) of size n cities
    def twoOptSwap(self, route, i, k):
        new_route = []
        runningCost = 0

        # Take route[0] to route[i - 1] and add them in order to new_route
        for index in range(i):
            new_route.append(route[index])
            if len(new_route) > 1: # Don't add the cost for the first city
                runningCost = self.updateRunningCost(new_route, runningCost)


        # Take route[i] to route[k] and add them in reverse order to new_route
        temp_index = k
        while temp_index ≥ i:
            new_route.append(route[temp_index])
            runningCost = self.updateRunningCost(new_route, runningCost)
            temp_index -= 1

        # Take route[k+1] to end and add them in order to new_route
        for index in range(k + 1, len(route)):
            new_route.append(route[index])
            runningCost = self.updateRunningCost(new_route, runningCost)

        # Makes sure we are returning a unique route because sometimes this
        # two opt function flips the new_route to something that we've already
        # generated
```

```
        isUniqueRoute = self.checkIsUniqueRoute(new_route)

        return new_route, runningCost, isUniqueRoute

    # Returns true if we have ran out of time
    #
    # Time Complexity: O(1) because we are performing constant time operations
    #                  with the time library
    # Space Complexity: O(1) because we are only updating global variables
    def isTimeUp(self):
        currTime = time.time()
        tempTimePassed = currTime - self._startTime
        totalTimePassed = tempTimePassed + self._results['time']
        if totalTimePassed ≥ self._time_allowance:
            return True # Time is up
        return False # Time is not up

    # Builds an initial BSSF from the greedy algorithm
    #
    # Time Complexity: Same as the greedy algorithm
    # Space Complexity: Same as the greedy algorithm
    def initResults(self, time_allowance):
        self._results = self.greedy(time_allowance)
```

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


import math
import numpy as np
import random
import time


class TSPSolution:
    def __init__(self, listOfCities):
        self.route = listOfCities
        self.cost = self._costOfRoute()

    # print( [c._index for c in listOfCities] )

    def _costOfRoute(self):
        cost = 0
        last = self.route[0]
        for city in self.route[1:]:
            cost += last.costTo(city)
            last = city
        cost += self.route[-1].costTo(self.route[0])
        return cost

    def enumerateEdges(self):
        elist = []
        c1 = self.route[0]
        for c2 in self.route[1:]:
            dist = c1.costTo(c2)
            if dist == np.inf:
                return None
            elist.append((c1, c2, int(math.ceil(dist))))
            c1 = c2
        dist = self.route[-1].costTo(self.route[0])
        if dist == np.inf:
            return None
        elist.append((self.route[-1], self.route[0], int(math.ceil(dist))))
        return elist
```

```python


def nameForInt(num):
    if num == 0:
        return ''
    elif num <= 26:
        return chr(ord('A') + num - 1)
    else:
        return nameForInt((num - 1) // 26) + nameForInt((num - 1) % 26 + 1)


class Scenario:
    HARD_MODE_FRACTION_TO_REMOVE = 0.20  # Remove 20 % of the edges

    def __init__(self, city_locations, difficulty, rand_seed):
        self._difficulty = difficulty

        if difficulty == "Normal" or difficulty == "Hard":
            self._cities = [City(pt.x(), pt.y(), \
                                 random.uniform(0.0, 1.0) \
                                 ) for pt in city_locations]
        elif difficulty == "Hard (Deterministic)":
            random.seed(rand_seed)
            self._cities = [City(pt.x(), pt.y(), \
                                 random.uniform(0.0, 1.0) \
                                 ) for pt in city_locations]
        else:
            self._cities = [City(pt.x(), pt.y()) for pt in city_locations]

        num = 0
        for city in self._cities:
            # if difficulty == "Hard":
```

```
            city.setScenario(self)
            city.setIndexAndName(num, nameForInt(num + 1))
            num += 1

        # Assume all edges exists except self-edges
        ncities = len(self._cities)
        self._edge_exists = (np.ones((ncities, ncities)) - np.diag(np.ones((ncities)))) > 0

        if difficulty == "Hard":
            self.thinEdges()
        elif difficulty == "Hard (Deterministic)":
            self.thinEdges(deterministic=True)

    def getCities(self):
        return self._cities

    def randperm(self, n):  # isn't there a numpy function that does this and even gets called in Solver?
        perm = np.arange(n)
        for i in range(n):
            randind = random.randint(i, n - 1)
            save = perm[i]
            perm[i] = perm[randind]
            perm[randind] = save
        return perm

    def thinEdges(self, deterministic=False):
        ncities = len(self._cities)
        edge_count = ncities * (ncities - 1)  # can't have self-edge
        num_to_remove = np.floor(self.HARD_MODE_FRACTION_TO_REMOVE * edge_count)

        can_delete = self._edge_exists.copy()

        # Set aside a route to ensure at least one tour exists
        route_keep = np.random.permutation(ncities)
```

```
        if deterministic:
            route_keep = self.randperm(ncities)
        for i in range(ncities):
            can_delete[route_keep[i], route_keep[(i + 1) % ncities]] = False

        # Now remove edges until
        while num_to_remove > 0:
            if deterministic:
                src = random.randint(0, ncities - 1)
                dst = random.randint(0, ncities - 1)
            else:
                src = np.random.randint(ncities)
                dst = np.random.randint(ncities)
            if self._edge_exists[src, dst] and can_delete[src, dst]:
                self._edge_exists[src, dst] = False
                num_to_remove -= 1


class City:
    def __init__(self, x, y, elevation=0.0):
        self._x = x
        self._y = y
        self._elevation = elevation
        self._scenario = None
        self._index = -1
        self._name = None

    def __str__(self):
        return self._name

    # Overwritten comparator function
    def __eq__(self, other):
        if self._name == other._name:
            return True
        return False

    def setIndexAndName(self, index, name):
```

```
        self._index = index
        self._name = name

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

    ''' <summary>
        How much does it cost to get from this city to the destination?
        Note that this is an asymmetric cost function.

        In advanced mode, it returns infinity when there is no connection.
        </summary> '''
    MAP_SCALE = 1000.0

    def costTo(self, other_city):

        assert (type(other_city) == City)

        # In hard mode, remove edges; this slows down the calculation...
        # Use this in all difficulties, it ensures INF for self-edge
        if not self._scenario._edge_exists[self._index, other_city._index]:
            return np.inf

        # Euclidean Distance
        cost = math.sqrt((other_city._x - self._x) ** 2 +
                         (other_city._y - self._y) ** 2)

        # For Medium and Hard modes, add in an asymmetric cost (in easy mode it is zero).
        if not self._scenario._difficulty == 'Easy':
            cost += (other_city._elevation - self._elevation)
```

```
            if cost < 0.0:
                cost = 0.0  # Shouldn't it cost something to go downhill, no matter how steep??????

        return int(math.ceil(cost * self.MAP_SCALE))


```

```python
from TSPClasses import *

class CityWrapper:

    def __init__(self, cost, city, indexInCities):
        self._cost = cost
        self._city = city
        self._indexInCities = indexInCities

    def __lt__(self, other):
        return self._cost < other._cost
```