

```

1  #!/usr/bin/python3
2
3  from which_pyqt import PYQT_VER
4  if PYQT_VER == 'PYQT5':
5      from PyQt5.QtCore import QLineF, QPointF
6  elif PYQT_VER == 'PYQT4':
7      from PyQt4.QtCore import QLineF, QPointF
8  else:
9      raise Exception('Unsupported Version of PyQt: {}'.format(PYQT_VER))
10
11
12
13  import copy
14  import time
15  import numpy as np
16  from TSPClasses import *
17  import heapq
18
19
20
21  class TSPSolver:
22      def __init__( self, gui_view ):
23          self._scenario = None
24
25      def setupWithScenario( self, scenario ):
26          self._scenario = scenario
27
28
29      ''' <summary>
30          This is the entry point for the default solver
31          which just finds a valid random tour
32      </summary>
33      <returns>results array for GUI that contains three ints: cost of solution, time
34      not counting initial BSSF estimate)</returns> '''
35      def defaultRandomTour( self, start_time, time_allowance=60.0 ):
36
37          results = {}
38
39
40          start_time = time.time()
41
42          cities = self._scenario.getCities()
43          ncities = len(cities)
44          foundTour = False
45          count = 0
46          while not foundTour:
47              # create a random permutation
48              perm = np.random.permutation( ncities )
49
50              #for i in range( ncities ):
51                  #swap = i
52                  #while swap == i:
53                      #swap = np.random.randint(ncities)
54                  #temp = perm[i]
55                  #perm[i] = perm[swap]
56                  #perm[swap] = temp
57
58              route = []
59
60              # Now build the route using the random permutation
61              for i in range( ncities ):
62                  route.append( cities[ perm[i] ] )
63
64              bssf = TSPSolution(route)
65              #bssf_cost = bssf.cost()
66              #count++;

```

```

67         count += 1
68
69         #if costOfBssf() < float('inf'):
70         if bssf.costOfRoute() < np.inf:
71             # Found a valid route
72             foundTour = True
73     #} while (costOfBssf() == double.PositiveInfinity);           // until a
    valid route is found
74     #timer.Stop();
75
76     results['cost'] = bssf.costOfRoute()
    #costOfBssf().ToString();           // load results array
77     results['time'] = time.time() - start_time
78     results['count'] = count
79     results['soln'] = bssf
80
81     # return results;
82     return results
83
84     def defaultRandomTourBSSF( self ):
85         cities = self._scenario.getCities()
86         ncities = len(cities)
87         foundTour = False
88         while not foundTour:
89             # create a random permutation
90             perm = np.random.permutation( ncities )
91             route = []
92
93             # Now build the route using the random permutation
94             for i in range( ncities ):
95                 route.append( cities[ perm[i] ] )
96
97             bssf = TSPSolution(route)
98
99             if bssf.costOfRoute() < np.inf:
100                 # Found a valid route
101                 foundTour = True
102         return bssf
103
104
105
106     def greedyBSSF( self ):
107         results = {}
108
109         cities = self._scenario.getCities()
110         visited = []
111         ncities = len(cities)
112         foundTour = False
113         currCity = cities[0]
114         visited.append(currCity)
115         while len(visited) < ncities:
116             nextCity = self.getNextCity_Greedy(currCity, cities, visited)
117             visited.append(nextCity)
118             currCity = nextCity
119         greedySolution = TSPSolution(visited)
120         return greedySolution
121
122     def greedy( self, start_time, time_allowance=60.0 ):
123         start_time = time.time()
124         results = {}
125         count = 0
126         cities = self._scenario.getCities()
127         visited = []
128         ncities = len(cities)
129         foundTour = False
130         currCity = cities[0]
131         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
165         #or the random tour happens to be better use it.
166         greedyBSSF = self.greedyBSSF()
167         randomTourBSSF = self.defaultRandomTourBSSF()
168         if greedyBSSF.costOfRoute() < randomTourBSSF.costOfRoute():
169             bssf = greedyBSSF
170         else:
171             bssf = randomTourBSSF
172
173         #Generate initial adjMatrix
174         initAdjMatrix = self.generateAdjMatrix(self.cities)
175         initPC = PriorityCount(0, 1, 0)
176         # Heap tuples order: lowerBound, currCity, unvisitedCities, adjMatrix, path
177         rootProblem = (initPC, initCity, unvisitedCities, initAdjMatrix, [initCity])
178         subProblems = []
179         heapq.heappush(subProblems, rootProblem)
180
181         totalHeldStates+=1
182         while len(subProblems) > 0:
183             #check if maxMunStoredStates needs to be updated
184             if len(subProblems) > maxNumStoredStates:
185                 maxNumStoredStates = len(subProblems)
186             #check if the time allowance has been exceeded
187             if (time.time() - start_time) > time_allowance:
188                 optimalSolutionFound = False
189                 break
190
191             currProblem = heapq.heappop(subProblems)
192             if currProblem[0].cost < bssf.costOfRoute():
193                 # If ALL cities have been visited, e.g. if ALL values of list are false
194                 if not True in currProblem[2]:
195                     bssf = TSPSolution(currProblem[4])
196                     count +=1
197
198                 for toCityIndex, toCity in enumerate(self.cities):

```

```

198         # Get the intersect of the cities to which the current city can
199         reach and the cities that are still unvisited
200         reachableCitiesBools =
201         toCity._scenario._edge_exists[currProblem[1]._index, :]
202         validRowIndices = [a and b for a, b in zip(reachableCitiesBools,
203         currProblem[2])]
204         if validRowIndices[toCityIndex]:
205
206             individualPathCost =
207             currProblem[3][currProblem[1]._index, toCityIndex]
208             nextAdjMatrix =
209             self.applyAdjMask(copy.deepcopy(currProblem[3]),
210             currProblem[1], self.cities[toCityIndex])
211             rowStepCost, nextAdjMatrix = self.reduceRows(nextAdjMatrix)
212             colStepCost, nextAdjMatrix = self.reduceCols(nextAdjMatrix)
213
214             nextPathCost = rowStepCost + colStepCost + currProblem[0].cost
215             + individualPathCost
216             if nextPathCost < bssf.costOfRoute():
217                 nextUnvisitedCities = copy.deepcopy(currProblem[2])
218                 nextUnvisitedCities[toCityIndex] = False
219
220                 path = copy.deepcopy(currProblem[4])
221                 path.append(self.cities[toCityIndex])
222                 nextPathPC = PriorityCount(nextPathCost, len(path),
223                 currProblem[0].order + 1)
224                 # Heap tuples order: lowerBound, currCity,
225                 unvisitedCities, adjMatrix, path
226                 nextProblem = (nextPathPC, self.cities[toCityIndex],
227                 nextUnvisitedCities, nextAdjMatrix, path)
228                 heapq.heappush(subProblems, nextProblem)
229                 totalHeldStates+=1
230
231         #Not including the sub-states that are implicitly pruned, just ones that
232         have been pushed onto the heap.
233         else:
234             totalPrunedStates+= 1
235
236         results['cost'] = bssf.costOfRoute()
237         results['time'] = time.time() - start_time
238         results['count'] = count
239         results['soln'] = bssf
240
241         #Logging the results needed for the chart so I don't have to mess with the GUI
242         print("NON-GUI RESULTS")
243         print("Total # of held states: ", totalHeldStates)
244         print("Max # of stored states at a given time: ", maxNumStoredStates)
245         print("Total # of pruned states: ", totalPrunedStates)
246         print("Optimal solution found?", optimalSolutionFound)
247
248         return results
249
250 def fancy( self, start_time, time_allowance=60.0 ):
251     pass
252
253 def getNextCity_Greedy(self, currCity, cities, visited):
254     currShortestPath = np.inf
255     closestCity = None
256     currCityCoords = np.array((currCity._x, currCity._y))
257     for city in cities:
258         if city not in visited:
259             toCityCoords = np.array((city._x, city._y))
260             # Gets the Euclidean distance
261             pathCost = np.linalg.norm(currCityCoords - toCityCoords)
262             if pathCost < currShortestPath:
263                 currShortestPath = pathCost

```

```

254         closestCity = city
255     return closestCity
256
257     def generateAdjMatrix(self, cities):
258         array = np.full((len(cities), len(cities)), np.inf)
259         for x in range(0, len(cities)):
260             for y in range(0, len(cities)):
261                 array[x,y] = cities[x].costTo(cities[y])
262         return array
263
264
265     def reduceRows(self, adjMatrix):
266         rowReductionCost = 0
267         for rowIndex, rowCity in enumerate(self.cities):
268             #minVal = minimum of the given row
269             minVal = np.amin(adjMatrix[rowIndex,:])
270             if minVal < np.inf:
271                 rowReductionCost += minVal
272                 adjMatrix[rowIndex,:] -= minVal
273         return rowReductionCost, adjMatrix
274
275     def reduceCols(self, adjMatrix):
276         colReductionCost = 0
277         for colIndex, colCity in enumerate(self.cities):
278             #minVal = minimum of the given col
279             minVal = np.amin(adjMatrix[:,colIndex])
280             if minVal < np.inf:
281                 colReductionCost += minVal
282                 adjMatrix[:,colIndex] -= minVal
283         return colReductionCost, adjMatrix
284
285     def applyAdjMask(self, currAdjMatrix, fromCity, toCity):
286         rowToMask = fromCity._index
287         colToMask = toCity._index
288         for colIndex, colVal in enumerate(currAdjMatrix[rowToMask,:]):
289             currAdjMatrix[rowToMask, colIndex] = np.inf
290         for rowIndex, rowVal in enumerate(currAdjMatrix[:,colToMask]):
291             currAdjMatrix[rowIndex, colToMask] = np.inf
292
293         currAdjMatrix[rowToMask, colToMask] = np.inf
294         currAdjMatrix[colToMask, rowToMask] = np.inf
295         return currAdjMatrix
296
297     # Unique class used to server as a key for the heap. First it will consider the cost of
298     # the path,
299     # If those happen to be identical it will take the path that was found first according
300     # to it's order.
301     class PriorityCount:
302         def __init__(self, cost, length, order):
303             self.cost = cost
304             self.length = length
305             self.order = order
306
307         def __lt__(self, other):
308             if (self.cost/self.length) < (other.cost/other.length):
309                 return self
310             elif self.cost > other.cost:
311                 return other
312             else:
313                 if self.order > other.order:
314                     return other
315                 else:
316                     return self

```

	# Cities	Seed	Runtime	Cost of best tour found	Max # of stored states	# of BSSF Updates	Total # of states created	Total # of states pruned
1	15	20	60.08332	12713	3641	168	15718	8374
2	16	902	59.71596	7954*	1525	47	12582	1988
3	10	20	0.427141	7499*	96	6	323	92
4	10	50	0.489094	8165*	97	10	307	90
5	11	50	1.025985	10058*	157	16	684	164
6	11	51	1.095618	8573*	102	13	697	125
7	13	62	12.1412	8234*	923	94	5419	1026
8	18	2	60.006	17341	4037	53	9495	213
9	20	40	60.04478	19268	2988	41	7353	308
10	20	2	60.00462	18622	6680	67	8728	469

## TSP Project 5 Report

2) Time Complexity: Each iteration the algorithm must do the following procedures:

- Insert/Delete/Pop from Priority Queue:  $O(\log n)$ ,  $O(\log n)$ ,  $O(1)$
- Logical Checks (exceeded time/is new BSSF):  $O(1)$
- Generate list of valid paths to check:  $O(n)$
- Update Adjacency Matrix:  $O(n^2)$  Since possibly near all of the values can be updated in the  $n \times n$  matrix on a given pass.
- Generate other info for new subproblem node:  $O(1)$

Therefore the overall time complexity of a given iteration is  **$O(n^2)$**

There are up to  $(n-1)!$  Potential worst-case iterations when no branches are pruned. Therefore, the overall worst-case time complexity is  **$O(n!)$**  yet the Branch-and-Bound technique reduces this to something far more manageable.

Space Complexity: There are multiple elements that need to be stored for each node in the problem tree. I have included those calculations in Question 3. Its overall complexity is of  $O(n^2)$  for one iteration. At any given time it can hold up to, in the absolute worst cast  $(n-1)!$  iterations. Therefore, the overall worst-case space complexity is  $O(n!)$ . Again, the branch and bound technique reduces this real average space complexity significantly.

3) I represented each state as a tuple containing the following information:

- A PriorityCount object (Currently contains two values, the current lower bound and the order in which it was added. Since this is the first element Python's heapq library interprets it as a the 'key' of the tuple and uses it to balance the heap. The `__lt__` comparison method has been overloaded to return the object with the lowest lower bound multiplied by the number of cities visited and, in the case of a tie, return the one added most recently as it is more likely to have explored more of the tree. I've structured it in this way to make different preferences easy to plug in later)  **$O(1)$**
- A City object which represents the current root of the subtree to analyze.  **$O(1)$**
- A Boolean list of the unvisited cities. This represents all of the direct ancestor nodes of the current root of the given subtree and is the same length as the total number of cities. (While this could be extrapolated out of the last element of the tuple containing a list of visited cities including it saves computation at the expense of a little space)  **$O(n)$**
- A city adjacency matrix with all of the direct ancestor's changes. This is inherently of size  $n \times n$  where  $n$  is the number of cities.  **$O(n^2)$**
- A growing list of the path taken, when it's length is equal to the length of total cities it is a potential candidate for the new BSSF.  **$O(n)$**

4) Python's heapq data structure is a priority queue built as a heap. This means that insertions, deletions run in  $O(\log n)$  time while time to find the minimum value runs in  $O(1)$  time. In a priority queue some sort of property is assigned (either implicitly by the data structure or explicitly by declaring it on insertion) and is placed in the queue according to the value of that property in comparison to all others. As a queue, the only interactions a client has with it is pushing and popping. Popping will return the node of most priority as defined and pushing will insert a new value according to that given property.

5) To get my initial BSSF I have implemented the greedy algorithm approach. This gets the cost of all immediate unvisited cities for which a path exists and takes the one of lowest cost successively until all nodes have been explored. In a direct accuracy vs. time comparison it is an effective way to get the initial BSSF as it will disregard many inefficient routes near instantaneously.

7) Regarding the table I found it interesting how I was able to come up with a solution for the 16-city route in the nick of time but not for the 15 city route. While they could be more efficient I feel like the times are generally pretty good. As it is now I am weighing breadth more significant than depth which for me generally ended up with faster results as apposed to a more depth leaning approach. (In other words, the “weight” is calculated by the path cost multiplied by the number of cities in the path. This makes the impact of immediate good paths more evident but updates of the BSSF more infrequent). With occasional rare exceptions, like the first two, the run time grew as a steady function of the number of input cities. While it certainly isn’t a factorial growth it is fairly rapid. It is important to note that the other features, such as the number of observed BSSF updates, the max number of stored states and the total # of created states all grow similarly, not factorial but very quickly.