05 Travelling Salesman Problem

Gideon Hale

1. Code (see Appendix)

2. Time and Space Complexity

Closer inspection is provided in the appendix for each portion of the code. I will provide here a brief summary of the complexities of the main parts.

I. defaultRandomTour()
- This could take at the very most O(n) space and O(kn) time, where k is the number of attempts it takes to pick a random path that is viable. So k could be anywhere from 1 to n!, but will likely be closer to the former than the latter, assuming an even probability distribution.

II. greedy()
- This is $O(n^3)$ time based upon an even probability distribution of selecting the starting point. Essentially, this algorithm goes from the starting point, picks the closest city, then repeats from that city until it gets all the way back to the start. If ever it hits a point where the only unvisited cities are unreachable, it starts over with a different randomly generated starting point. We assume that such a solution exists.
- It will also take O(n) space, since the most of anything it stores at one time will be a complete tour of the cities.

III. branchAndBound()
- This will take $O(kn^3)$ time, where k represents the absolute total number of generated states. It is valued somewhere from n to n!, though again, hopefully closer to the former than the latter. The only reason it would be n! is if every possible state is generated and examined. This is unlikely, since it is the point of the Branch and Bound concept.
- Expanding a State into its children is takes n * State.__init__(), which becomes n^3, since that initialization automatically includes the cost matrix reduction algorithm, which takes $O(n^2)$ time. That is where the n^3 part comes from in the above.
- There will be $O(pn^2)$ space where p is the largest size that the priority queue ever gets. The n^2 part comes from the size of each maintained State object.
- In the code, k == childrenCount and p == maxPQSize.

IV. Priority Queue

- In my code, I use the built-in heapq. Let p be the largest size it ever gets.
- According to the documentation, this takes O(log p) time for the heappush() and heappop() operations
- Additionally, it requires O(p) space for the entire Branch and Bound algorithm.

3. State class

This class holds a reduced cost matrix, a lower bound, a running path, and an ExemptCities object. The ExemptCities is a namedtuple object containing two arrays listing which cities are not to be reduced either as sources or destinations, respectively.

The time complexity of the State class upon initialization is $O(n^2)$ due to updating the reduced cost matrix. Any other time costs are trivial for this object.

The space requirement is $O(n^2)$ as well since each State must hold a separate reduced cost matrix of size n x n.

4. Priority Queue

The priority queue utilized in branchAndBound() is the built-in heapq. It operates like a standard heap. I have overridden the “less than” operator in the State class to make the priority ordering incorporate both depth and lower bound together to determine proper priority.

The time and space complexity for the priority queue are discussed above.

5. Initializing the BSSF

This algorithm uses greedy() to initialize the Best Solution So Far (BSSF). See 2.II. above. The greedy algorithm process is described in detail under that section.

6. Empirical Data

Num cities	Seed	Greedy		Brand and Bound		Max queue size	BSSF update count	Total states created	Total states pruned
		Running time (sec)	Cost of tour	Running time (sec)	Cost of best tour found				
11	3	0.00062	9858	0.13901	8011	755	1	1426	754
12	22	0.00089	11824	0.32915	10092	1709	1	3284	1706
13	16	0.00081	8418	1.12244	8132	949	1	10794	582
14	40	0.00078	13589	6.46675	10508	20416	1	53805	18502
15	20	0.00075	11081	3.68635	10534	7806	1	30155	7787
16	902	0.00078	12421	6.19883	7954	21682	1	31365	21638
17	1	0.00075	12165	24.3733	10109	75311	2	140k	75189
18	10	0.00381	15061	56.1829	1197	103k	2	307k	100k
19	101	0.00132	16425	60.0008	16425	223k	0	317k	0
20	8	0.00160	13631	60.2014	13631	197k	0	282k	0
21	7	0.00107	14536	60.0009	11717	161k	1	258k	0
22	6	0.00208	14935	60.0011	14935	159k	0	219k	0

optimal tour

7. Results

All data above was collected by running TSP in hard mode.

It appears that my algorithm is pretty incapable of solving the optimal tour of a set of cities of size larger than 18 in under 60 seconds. I think the way the algorithm works, there is a node building phase where a lot of nodes are created, and then a pruning phase, where all the extra nodes get dropped. This is why in the larger samples, no states get pruned.

The space complexity expressed by Max Queue Size seems to get bigger with each larger completed TSP search, but on the ones we capped, the max queue size

stagnates. I think this is because only so many states can be generated in 60 seconds, so if a scenario of cities and edges requires more than that, it will get cut off.

The time complexity increases fairly regularly as the number of cities increases. This is simply because a larger set of cities requires more states and therefore more time. Obviously as the number of cities gets large enough (19, in my particular case), the time caps out at 60 seconds. We could increase this cap to undergo a more robust study, but this suffices for the scope of this class.

8. Mechanisms

My comparison operator __lt__() in the State object at the bottom determined how the priority was figured. Therefore, it was in that method that I had to locate the depth vs. breadth balance. The calculation is simply lowerBound - depth, causing for a large depth to take priority over another node with shallower depth and where all else is equal. Unfortunately, I do not think it helped much, because when I remove the depth out of the equation completely, the time decreases in each case by a few seconds.

However, depth could have been further prioritized by adding a multiplier to the depth calculation and then setting it to a large positive integer. Further experimentation could influence what scalar would improve the process for different numbers of cities.

Additionally, one could use two priority queues and alternate which is pulled from each time.

Appendix

```
#!/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
elif PYQT_VER == 'PYQT6':
    from PyQt6.QtCore import QLineF, QPointF
else:
    raise Exception('Unsupported Version of PyQt: {}'.format(PYQT_VER))


import time
import numpy as np
```

```python
from TSPClasses import *
import heapq
import itertools


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>
    '''

    # O(nk) time, O(n) space
    def defaultRandomTour(self, time_allowance = 60.0):

        results = {}
        cities = self._scenario.getCities() # this takes up O(n) space
        ncities = len(cities)
        foundTour = False
        count = 0
        bssf = None
        start_time = time.time()

        # Assuming a viable tour exists, this loops k times, where k can be
anywhere
        # from 1 to n!. It will in practice be much closer to 1 than n!.
        while not foundTour and time.time()-start_time < time_allowance:
            perm = np.random.permutation(ncities) # random permutation cause
O(n!) time
```

```
            route = [] # this ends up being O(n) space, and is reset at each
iteration
            # loops n times
            for i in range( ncities ): route.append( cities[ perm[i] ] )

            bssf = TSPSolution(route)
            count += 1
            if bssf.cost < np.inf: foundTour = True

        end_time = time.time()

        results['cost'] = bssf.cost if foundTour else math.inf
        results['time'] = end_time - start_time
        results['count'] = count
        results['soln'] = bssf
        results['max'] = None
        results['total'] = None
        results['pruned'] = None

        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>
    '''

    # O(n^3) time, O(n) space
    def greedy( self, time_allowance = 60.0 ):

        results = {}
        cities = self._scenario.getCities() # O(n) space
        ncities = len(cities)
```

```
        foundValidTour = False
        count = 0
        bssf = None
        start_time = time.time()

        # Assuming a viable tour exists and we have regular probability, this
loops n times maximum
        while not foundValidTour and time.time()-start_time < time_allowance:
            visitedCities = [False for _ in range(ncities)] # O(n) space
            # This random permutation will in practice find a viable starting
point in O(n) time.
            currIndex = np.random.randint(ncities)
            startCity = cities[currIndex]
            route = [] # O(n) space

            # loops n times
            for _ in cities:
                currCity = cities[currIndex]
                route.append(currCity)
                visitedCities[currIndex] = True
                nextIndex = None
                # loops n times
                for i in range(ncities):
                    if visitedCities[i]: continue
                    if nextIndex == None or currCity.costTo(cities[i]) <
currCity.costTo(cities[nextIndex]):
                        nextIndex = i
                currIndex = nextIndex

            count += 1
            bssf = TSPSolution(route)
            if bssf.cost < np.inf: foundValidTour = True

        end_time = time.time()

        results['cost'] = bssf.cost if foundValidTour else math.inf
        results['time'] = end_time - start_time
        results['count'] = count
        results['soln'] = bssf
        results['max'] = None
        results['total'] = None
        results['pruned'] = None

        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>
    '''

    # O(kn^3) time where k is the total number of generated states between n and
n!
    # (see while loop)
    # O(pn^2) space where m is the max size of the priority queue
    def branchAndBound(self, time_allowance = 60.0):

        results = {}
        cities = self._scenario.getCities() # this is O(n) space
        ncities = len(cities)
        count = 0
        start_time = time.time()
        bssf = self.greedy(time_allowance)['soln'] # this is O(n^3) time (see
above)

        pq = [] # say this only ever has a max length of p (see also maxPQSize 5
lines later)
        startCity = 0
        state1 = State(cities, startCity) # each State object is O(n^2) space
        heapq.heappush(pq, (None, state1)) # O(log p) time

        maxPQSize = 0
        childrenCount = 0
        prunedCount = 0

        # This will loop from n to n! times, since that is the absolute maximum
number of
        # possible states. It will be closer to n, in part due to the time
constraint.
        # Let's just say that it loops k times.
        while len(pq) > 0 and time.time() - start_time < time_allowance:
```

```
            parent = heapq.heappop(pq)[1] # O(log p) time
            if parent.lowerBound > bssf.cost:
                prunedCount += 1
                continue

            # loops n times
            for nextDest in range(len(cities)):
                if nextDest in parent.path: continue

                # in creating a state, it is automatically reduced, which takes
O(n^2) time
                child = State(parent, nextDest)
                childrenCount += 1

                if child.lowerBound > bssf.cost: continue

                 # O(log p) time for heappush
                if child.depth() < ncities: heapq.heappush(pq, (None, child))
                elif child.edgeMat[child.lastCity()][child.path[0]] != np.inf:
                    # _buildRout() is O(n) time
                    newBSSF = TSPSolution(self._buildRoute(child.path))
                    if newBSSF < bssf:
                        bssf = newBSSF
                        count += 1

            if len(pq) > maxPQSize: maxPQSize = len(pq)

        end_time = time.time()


        results['cost'] = bssf.cost
        results['time'] = end_time - start_time
        results['count'] = count
        results['soln'] = bssf
        results['max'] = maxPQSize
        results['total'] = childrenCount
        results['pruned'] = prunedCount

        return results

    # O(n) time
    def _buildRoute(self, path):
        cities = self._scenario.getCities()
```

```
        return [cities[cityIndex] for cityIndex in path]


    ''' <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>
    '''

    def fancy( self, time_allowance=60.0 ): pass


class State:
    from collections import namedtuple
    ExemptCities = namedtuple('ExemptCities', ['srcs', 'dests'])

    # each State object takes O(n^2) time and O(n^2) space to create and maintain
    def __init__(self, param1, param2):
        if type(param1) == list: # the first instance
            cities = param1
            start = param2
            # a matrix of edge costs
            self.edgeMat = [[src.costTo(dest) for dest in cities] for src in
cities]
            self.lowerBound = 0
            self.path = [start] # an array of city indices
            self.exemptCities = self.ExemptCities([],[])
        else: # all following instances
            parent = param1
            nextCityIndex = param2
            self.edgeMat = [row.copy() for row in parent.edgeMat]
            self.lowerBound = parent.lowerBound.copy()
            self.path = parent.path.copy()
            self.exemptCities =
self.ExemptCities(parent.exemptCities.srcs.copy(),
                                        parent.exemptCities.dests.copy())

            self._addToPath(nextCityIndex)
```

```
        self._reduce()

    def __str__(self):
        output = "(" + str(self.lowerBound) + ")\n"
        output += str(self.edgeMat)
        return output

    def __lt__(self, other):
        return self.lowerBound - self.depth() < other.lowerBound - other.depth()

    # O(n) time
    def _addToPath(self, next):
        last = self.lastCity()
        self.path.append(next)

        # infinitize all rows and columns next (which is actually just an index
of the next city)
        self.lowerBound += self.edgeMat[last][next]
        for destIndex in self.edgeMat[last]: destIndex = np.inf
        for srcIndex in self.edgeMat: srcIndex[next] = np.inf

        # mark the last and next as exempt as src and dest, respectively
        self.exemptCities.srcs.append(last)
        self.exemptCities.dests.append(next)

    # O(n^2) time
    def _reduce(self):
        addToLowBnd = 0

        # loops n times
        for src in range(len(self.edgeMat)):
            if src in self.exemptCities.srcs: continue

            shortestEdgeCost = min(self.edgeMat[src]) # O(n) time

            if shortestEdgeCost != np.inf:
                # loops n times
                for dest in range(len(self.edgeMat[src])):
                    self.edgeMat[src][dest] -= shortestEdgeCost

            addToLowBnd += shortestEdgeCost

        edgeMatTrans = np.transpose(self.edgeMat) # O(n^2) time
        # loops n times
        for dest in range(len(edgeMatTrans)):
```

```
            if dest in self.exemptCities.dests: continue

            shortestEdgeCost = min(edgeMatTrans[dest])

            if shortestEdgeCost != np.inf:
                # loops n times
                for src in range(len(edgeMatTrans[dest])):
                    edgeMatTrans[dest][src] -= shortestEdgeCost

            addToLowBnd += shortestEdgeCost

        self.edgeMat = np.transpose(edgeMatTrans) # O(n^2) time

        self.lowerBound += addToLowBnd

    def lastCity(self): return self.path[-1]

    def depth(self): return len(self.path)
```