Traveling Salesman Report

Niels Turley

1. Source code

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  
from TSPClasses import \*  
import heapq  
import itertools  
import queue  
from Node import Node  
  
class TSPSolver:  
 def \_\_init\_\_(self, gui\_view):  
 self.\_scenario = None  
  
 def setupWithScenario(self, scenario):  
 self.\_scenario = scenario

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  
 results['soln'] = bssf  
 results['max'] = None  
 results['total'] = None  
 results['pruned'] = None  
 return results  
  
   
 # Time complexity: O(S \* L), where S is the total number of states created and L is the complexity of computing   
 # the lower bound (which is O(n^2), iterating through the whole matrix). The worst case for S would be exploring   
 # the whole tree, which would be O(b^n) where b is the average branching factor and n is the depth. However,   
 # pruning allows it to not explore the whole tree, so depending on how well I've implemented it, it could be much   
 # less than that.  
   
 # Space complexity: O(T \* max\_queue\_size), where T is the size of the data structure for each state (which is O(  
 # n^2), for the matrix).  
 def branchAndBound(self, time\_allowance=60.0):  
 # initialize tracking variables  
 results = {}  
 count = 0  
 max\_queue\_size = 1  
 total\_states = 1  
 total\_pruned = 0  
 init\_results = self.defaultRandomTour(time\_allowance)  
 bssf = init\_results['soln']  
 cities = self.\_scenario.getCities()  
 ncities = len(cities)  
 start\_time = time.time()  
  
 # create a matrix of the distances between cities  
 matrix = np.zeros((ncities, ncities))  
 for i in range(ncities):  
 for j in range(ncities):  
 matrix[i][j] = cities[i].costTo(cities[j])  
  
 # initial reduction of the matrix  
 # for each row, subtract the smallest value from each element  
 cost = 0  
 for i in range(ncities):  
 row\_min = np.min(matrix[i])  
 cost += row\_min  
 matrix[i] = matrix[i] - row\_min  
 # for each column, subtract the smallest value from each element  
 for i in range(ncities):  
 col\_min = np.min(matrix[:, i])  
 cost += col\_min  
 matrix[:, i] = matrix[:, i] - col\_min  
  
 # create a priority queue  
 pq = []  
 remaining = list(range(ncities))  
 current = remaining.pop(0)  
 root = Node(matrix, cost, [current], remaining)  
 heapq.heappush(pq, (root.cost, root))  
  
 # begin the search  
 while len(pq) > 0 and time.time() - start\_time < time\_allowance:  
 # get the next node to expand  
 cost, node = heapq.heappop(pq)  
 # if the node is a leaf node  
 if len(node.remaining) == 0:  
 count += 1  
 # if the node is a better solution than the current best solution  
 if node.cost < bssf.cost:  
 # update the best solution  
 route = []  
 for city in node.path:  
 route.append(cities[city])  
 bssf = TSPSolution(route)  
 # prune the queue  
 for item in pq:  
 if item[1].cost >= bssf.cost:  
 pq.remove(item)  
 total\_pruned += 1  
 # if the time limit has been reached  
 if time.time() - start\_time > time\_allowance:  
 # return the best solution found so far  
 break  
 # if the node is not a leaf node  
 else:  
 # if the node is a better solution than the current best solution  
 children, num\_pruned, num\_created = self.expand\_node(node, bssf)  
 total\_states += num\_created  
 total\_pruned += num\_pruned  
 for child in children:  
 # add the child to the priority queue  
 heapq.heappush(pq, (child.cost \* len(child.remaining), child))  
 if len(pq) > max\_queue\_size:  
 max\_queue\_size = len(pq)  
  
 end\_time = time.time()  
 results['cost'] = bssf.cost  
 results['time'] = end\_time - start\_time  
 results['count'] = count  
 results['soln'] = bssf  
 results['max'] = max\_queue\_size  
 results['total'] = total\_states  
 results['pruned'] = total\_pruned  
 return results  
  
 # Time complexity: O(n^2), where n is the number of cities. We have to iterate through the whole matrix to reduce it.  
   
 # Space complexity: O(n^2), where n is the number of cities. We are only storing the reduced 2D matrix.  
 def reduce\_matrix(self, node):  
 # reduce the matrix by subtracting the minimum value in each row from each value in the row  
 # and subtracting the minimum value in each column from each value in the column  
 # return the reduced matrix and the total cost of the reductions  
 matrix = node.matrix  
 path = node.path  
  
 cost = 0  
 # subtract the minimum value in each row from each value in the row  
 for i in range(len(matrix)):  
 row\_min = np.min(matrix[i])  
 if row\_min == np.inf:  
 if i not in path:  
 return None, np.inf  
 else:  
 continue  
 matrix[i] -= row\_min  
 cost += row\_min  
  
 # subtract the minimum value in each column from each value in the column  
 for i in range(len(matrix)):  
 col\_min = np.min(matrix[:, i])  
 if col\_min == np.inf:  
 if i not in path:  
 return None, np.inf  
 else:  
 continue  
 matrix[:, i] -= col\_min  
 cost += col\_min  
 # return the reduced matrix and the total cost of the reductions  
 return matrix, cost  
  
 # Time complexity: O(n^3). The first for loop iterates through the remaining list, and then we must reduce the  
 # matrix for each child node created. The length of the remaining list will worst case be length n, and reducing  
 # takes O(n^2) time, giving us a total of O(n^3) time.  
  
 # Space complexity: O(n^3). The 2D matrix is created for each child node (n^2) and stored in the children list  
 # (length n).  
 def expand\_node(self, node, bssf):  
 # create a list of all the children of the node  
 children = []  
 num\_pruned = 0  
 num\_created = 0  
 # for each city that has not been visited  
 for city in node.remaining:  
 # create a child node  
 num\_created += 1  
 child\_path = node.path.copy()  
 former\_city = child\_path[-1]  
 child\_path.append(city)  
 child\_remaining = node.remaining.copy()  
 child\_remaining.remove(city)  
 child = Node(np.copy(node.matrix), 0, child\_path, child\_remaining)  
  
 # get initial cost of the parent node + traveling to the city  
 cost = node.cost  
 cost += node.matrix[former\_city][city]  
  
 # set the row and column of the child node's matrix to infinity  
 child.matrix[former\_city] = np.inf  
 child.matrix[:, city] = np.inf  
  
 # reduce the child node's matrix and add the cost of the reductions to the child node's cost  
 child.matrix, child.cost = self.reduce\_matrix(child)  
 child.cost += cost  
  
 if child.cost > bssf.cost:  
 num\_pruned += 1  
 continue  
 # add the child node to the list of children  
 children.append(child)  
 # return the list of children  
 return children, num\_pruned, num\_created

class Node:  
 def \_\_init\_\_(self, matrix, cost, path, remaining):  
 self.matrix = matrix  
 self.cost = cost  
 self.path = path  
 self.remaining = remaining

1. Time and space complexity discussion
   1. branch\_and\_bound()
      1. Time complexity: O(S \* L), where S is the total number of states created and L is the complexity of computing the lower bound (which is O(n^2), iterating through the whole matrix). The worst case for S would be exploring the whole tree, which would be O(b^n) where b is the average branching factor and n is the depth. However, pruning allows it to not explore the whole tree, so depending on how well I've implemented it, it could be much less than that.
      2. Space complexity: O(T \* max\_queue\_size), where T is the size of the data structure for each state (which is O(n^2), for the matrix). We are only storing these states in the priority queue, so the total complexity is O(n^2 \* max\_queue\_size) or just O(T \* max\_queue\_size).
   2. reduce\_matrix()
      1. Time complexity: O(n^2). We have to iterate through each row (n rows) and find the minimum (takes n time to find) to subtract from the row’s cells. This gives us O(n^2). We have to do the same for the columns, which is also O(n^2). Added the two time complexities together doesn’t affect the complexity class, so it remains O(n^2)
      2. Space complexity: O(n^2). We are only storing the 2D matrix that we are reducing. All other lists (path, remaining) are superseded by the O(n^2).
   3. expand\_node()
      1. Time complexity: O(n^3). The first for loop iterates through the remaining list, and each iteration we must reduce the matrix for each child node created. The length of the remaining list of the for loop will worst case be length n, and reducing takes O(n^2) time since we have to iterate through the entire matrix, giving us a total of O(n^3) time.
      2. Space complexity: O(n^3). We store each child 2D matrix (n^2) in a list of length n. All other lists (path, remaining) are superseded by the O(n^3).
2. Describe the data structures used

Each *matrix* is a 2D numpy array, which takes a total of O(n^2) space.

Each *state* is a Node class, which contains the variables matrix, cost, path (as a list), and remaining (as a list).

I’m also using a priority queue…

1. Describe the priority queue data structure

These states are stored in a *priority queue*, which I implemented using the heapq class. Using a priority queue allows me to easily pick the next best state to expand since it uses a min binary heap to store elements, though it isn’t as fast to insert into as a list. I added each state as a tuple (priority, Node). I determined the best state (the priority it was given in the priority queue) as a function of dividing the cost of the state by the number of remaining cities and picking the smallest.

1. Describe the approach for the initial BSSF

I ran out of time implementing the greedy so I just used the defaultRandomTour ¯\\_(ツ)\_/¯

1. Table

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| # Cities | Seed | Running time | Cost of best tour found (\*=optimal) | Max # of stored states | # of BSSF updates | Total # of states created | Total # of states pruned |
| 15 | 20 | 2.279 | 10534\* | 74 | 18 | 12218 | 10475 |
| 16 | 902 | 5.306 | 7954\* | 85 | 6 | 25706 | 22712 |
| 50 | 183 | 60 | 21271 | 1849 | 13 | 103643 | 87221 |
| 45 | 703 | 60 | 18693 | 1249 | 11 | 118276 | 94699 |
| 40 | 460 | 60 | 17147 | 805 | 7 | 129359 | 112569 |
| 35 | 750 | 60 | 15553 | 486 | 1 | 142089 | 130975 |
| 30 | 816 | 60 | 14774 | 410 | 1 | 168933 | 151272 |
| 18 | 112 | 20.045 | 10420\* | 108 | 11 | 87549 | 77051 |
| 10 | 399 | 0.206 | 8012\* | 33 | 5 | 1541 | 1168 |
| 20 | 245 | 19.896 | 10252\* | 137 | 6 | 72646 | 66179 |

1. Discuss the results of the table

The table shows a pretty dramatic exponential increase in the running time as the number of cities increases, which is to be expected. It looks like there is a cut off between 20 and 30 cities where the algorithm hits the time limit—though it should be noted that the chosen seed can really change how long the algorithm takes, with some bigger problem sizes taking a short amount of time.

Another thing to note is that the BSSF updates weren’t at all correlated with the problem size. This is probably because I was using the random permutation function supplied by default, which might get a good path or might not. This is also reflected in the number of states created, which roughly increased exponentially following the problem size, but also increased more so based on *lower* number of BSSF updates (i.e. size 30 had 1 update, but 168933 states created which was the maximum in this table). This seems to indicate that the BSSF I originally had was a good one, but not the best one, and so the algorithm spent the entirety of the time created and discarding new states.

1. Discuss mechanisms used to dig deeper

One of the solutions I tried was somehow incorporating the depth of the node as well as the cost of the state into the priority queue. By involving depth into the priority, the algorithm would be more inclined to reach solutions in the nodes deeper down the node children tree. My first thought was to rank all of the costs, then rank the depth of the state, then add the two ranks together and use that as the priority, but I worried that having to rank each cost as I was inserting it would prove tedious and time-consuming (something like make a priority queue for the costs, then make another priority queue for the depth + costs). Instead, a simpler approach I took was to multiply the cost by the length of the remaining cities list. This reduced my runtimes by about 25%, more so for smaller problem sizes.

Another solution I attempted but was unable to finish was implementing the greedy algorithm. If I had been able to implement it in time, finding a lower BSSF from the greedy algorithm would’ve enabled me to prune states quicker and cut swaths out from the branching tree, especially in the beginning.