Project 5 Report

# 1. Time and Space complexity

Branch and Bound PsuedoCode

**Initialize Values**  - time is constant, space is linear

**Get initial bssf -** used random, time is approx. n, space is n

**Fill initial cost matrix -** time is n^2, space is n^2

**Reduce initial matrix -** time is n^2, space is n^2

**While queue is not empty and time < allowed: -** time and space could be n! but will be limited by time limit

**Pop item off queue -** time is nlogn for priority queue, space is n^2

**If cost is greater than bssf, skip**

**Get current city index**

**For all possible destinations from this city: -** worst case n repeats, will depend on branching factor

**Expand subproblem -** time n^2, space n^2

**Check cost, prune if greater**

**If depth is max:**

**Convert to possible solution -**  time is n, space is n

**Check cost, if better update bssf, else don’t**

**Else:**

**Put new state on queue -** time is nlogn for priority queue

**Return values**

**Discussion: (n = number of cities, b= branching factor (0.2), S = number of states)**

**Priority Queue** – nlogn time for insert and delete (used python heapq). Space is n^2 \* #of states on queue which could be bn^n but most will be pruned.

**SearchStates** – Each state has time complexity n^2 to be computed because the RCM must be reduced. It has space complexity of n^2 for storing the rcm

**RCM** – updating this takes n^2 time because for each row and each column, the min is found and subtracted from the row. The space complexity is n^2

**Initial BBSF –** I used the random function for initialization which chooses a random possible path. This has time complexity of n because it is just creating a random permutation of the order of cities. It is possible that this permutation will not be valid. The probability of this is b^n but since n is small in practice, it is considered a constant factor and overall complexity is n. Space complexity for this is also n for storing the solution. I run the random algorithm 5 times and take the best solution but this is constant factor and so is ignored.

**Expanding SearchStates** - time complexity worse case is n\*n^2 because each search state could lead to n more search states with n^2 time complexity to calculate new rcm. In practice, the branching factor reduces this to nb\*n^2 because not all cities are connected. This takes n\*n^2 additional space as n new subproblems each with rcm of n^2 space.

**Branch and Bound –** The whole branch and bound algorithm is hard to fit into a complexity class. The worst case time complexity is S\*L where S is the number of states and L is the complexity of creating the computing the lower bound. For our algorithm S could be up to (n+1)! Possible states and computing each state is n^2 so overall worst case of O(n^2\*(n+1)!). In practice this will be much less because the initial BSSF will allow automatic pruning of hopefully over half the states and this will be improved upon quickly. It will depend on how much of the tree is explored, how large the queue is allowed to be, and how good of an initial bssf is found. Since we are taking the best of 5 random bssf to start, we can hope that the start will be at least an average solution if not better than average. This will mean pruning can begin immediately. Also since not all cities are connected, (each city is only connected to bn other cities where b was 0.2 for our case) a large portion of the tree will never be expanded at all as there are not valid paths.

The overall space complexity is similar with O(L\*size of queue). In our implementation the queue is not limited in size so could grow to (n+1)! And L is n^2 so space complexity worst case is O(n^2(n+1)!) however like above this is very unlikely to occur and in practice the time and space is much less.

# 2. Data Structures

2.1 SubProblem Representations

I create a class to represent the subProblems that contained all important information for each state. The RCM was stored along with the lower bound for that state which would allow the creation of other substates later. The level of the tree was also stored and used in computing the priority given to each subproblem in the queue. The path used so far was stored along with the current city id so that outgoing paths could be looked at. The class also had functions for getting the length of the current path and overridden comparison function so that it could be sorted in the priority queue.

2.2 Priority Queue

I used the python heapq module to implement the priority queue. This is a binary tree style heap implementation like the one that we wrote for a previous project. It has time complexity of nlogn for inserts and nlogn for deletions both because of the bubbling that must be done to maintain the sorted nature of the binary tree. It works by sorting the array so that it represents a binary tree with higher priority items at the top. The top item is the highest priority and will be returned first after which the last item is put into the first items place and bubbled down until it is sorted again.

# 3. Initial BSSF

For the initial BSSF I used the already implemented random path algorithm. It was very fast especially for small number of cities. This meant that I could run it several times and take the solution with the shortest path as my initial bssf. I considered writing a function to dynamically change how many times the random algorithm was run at the beginning but ended up just defaulting to 5. A dynamic algorithm could have accounted for the fact that for very large n, it is probably not worth getting 5 different random paths as it may take a long time to find 5 valid random paths. For very small problems it is not worth running the random algorithm too many times either as it is unlikely to be much faster than just doing Branch and Bound. I did not figure out a good way to optimize this however so for smallish problems n<20 I use 5 the best of r random bssf as the initial and for n>20 I just do 1.

# 4. Results

**Table of Results**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **# Cities** | **Seed** | **Running time (sec.)** | **Cost of best tour found (\*=optimal)** | **Max # of stored states at a given time** | **# of BSSF updates** | **Total # of states considered** | **Total # of states pruned** |
| 15 | 20 | 0.756 | 10534\* | 74 | 19 | 31426 | 27730 |
| 16 | 902 | 1.5305 | 7954\* | 89 | 7 | 68657 | 60924 |
| 10 | 1 | 0.0525 | 9357\* | 30 | 3 | 1132 | 934 |
| 12 | 1 | 0.115851 | 9151\* | 48 | 6 | 4001 | 3394 |
| 20 | 1 | 4.824 | 10733\* | 143 | 11 | 218494 | 199091 |
| 22 | 1 | 12.829 | 10733\* | 179 | 24 | 618767 | 568447 |
| 23 | 3 | 60 | 12658 | 195 | 13 | 2470778 | 2267743 |
| 25 | 1 | 60 | 12399 | 233 | 3 | 2701011 | 2502587 |
| 30 | 1 | 60 | 15540 | 341 | 7 | 2741992 | 2562582 |
| 40 | 1 | 60 | 17713 | 619 | 10 | 3885140 | 3732847 |
| 50 | 1 | 60 | 19025 | 983 | 4 | 3963989 | 3839692 |

**Discussion**

These results are interesting for several reasons. The algorithm with the hyperparameters that I set seemed to do very well for up to 22 cities. It found the optimal path very quickly. Everything above 22 was harder and often was not solved in the 60 second limit. I assume that a beam search would have improved on this since optimal solutions were not guaranteed anyway because of time out, it would have increased the number of improved bssf and hopefully gotten closer. The time complexity theoretically increasing exponentially for each new city so it is reasonable that one new city would make such a large difference. Lower numbers show an almost doubling in time for each new city added.

It was interesting to compare the final bssf for the higher numbers of cities and see how it was increasing far from the 9k-10k range of all the optimal solutions found down below. While this is partially explained by an increased number of cities without complete connectedness, it also shows a rough measure of how far off the optimal solution each was when it timed out.

The ration of pruned states to considered states stayed about constant which was promising and makes sense as hopefully the majority of states are being pruned in all cases because of a decent initial bssf and quick digging deep to improve it.

**Optimizations**

I tried several different priority functions to optimize the state space search and get it to dig deeper more quickly. I did this my calculating the priority of each subproblem as it’s lower bound – it’s level \* an adjustment. This meant that cities at a lower level would be prioritized before cities at a higher level. The adjustment factor allowed me to tune to what extend this prioritization was done. Interestingly, forcing it to always dig deeper with an adjustment of 200 or more did not improve on the results at high numbers of cities. This makes sense as the algorithm would immediately dig down to the bottom layer taking the shortest path possible but then stay at the bottom exploring all possible last steps before going up one more layer etc. It therefore would have to work it’s way through all the possible states backwards to find the correct optimizations early on that lead to the optimal solution. I found that an adjustment factor of 100 worked fairly well as this would prioritize lower level states if there was a discrepancy of less than 100 but it would stop prioritizing them if there were other paths at higher levels that were more than 100 shorter. This seemed to lead to decent results at all the numbers of cities that I tried where the bssf was being updated but not too many times and a large number of states were being pruned.

# Appendix: Commented Source Code

#!/usr/bin/python3  
import random  
  
from which\_pyqt import PYQT\_VER  
  
if PYQT\_VER == 'PYQT5':  
 from PyQt5.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 \*  
from heapq import \*  
import itertools  
  
  
def reduceMatrix(arr, lb): # Function to reduce the given matrix and update the lower bound  
 n = len(arr)  
 for row in range(n): # Check rows for minimum value  
 minItem = np.min(arr[row])  
 if minItem == float('inf'): # this row has been used and can be ignored  
 continue  
 lb += minItem # add the minimum to the lower bound (will often be 0)  
 arr[row] -= minItem # reduce the values in this row  
 for col in range(n): # Check columns for minimum value  
 minItem = np.min(arr[:, col])  
 if minItem == float('inf'): # this row has been used and can be ignored  
 continue  
 lb += minItem # add the minimum to the lower bound (will often be 0)  
 arr[:, col] -= minItem # reduce the values in this column  
 return arr, lb  
  
class BBsubProblem: # An object to store a subinstance of the branch and bound problem  
 def \_\_init\_\_(self, rcm, priority, lb, level, curPath, cityID):  
 self.rcm = rcm  
 self.priority = priority  
 self.lb = lb  
 self.level = level  
 self.path = curPath  
 self.cityId = cityID  
  
 def getDepth(self):  
 return len(self.path)  
  
 def \_\_gt\_\_(self, other): # overriden greater than function so that objects can be stored in priority queue  
 if self.priority > other.priority:  
 return True  
 elif self.priority < other.priority:  
 return False  
 elif self.level >= other.level: # if priority is the same, use level  
 return True  
 else:  
 return False  
  
  
def calcPriority(lb, level, ncities): # calculate the priority for subproblems  
 # if level > 0:  
 # adjustFactor = (10\*ncities\*level) \* np.log(ncities/level) # Enhanced entropy function. Will return high values for levels in the middle of the total number of cities  
 # print("Using adjust %d for level %d with %d cities" % (adjustFactor, level, ncities))  
 # else:  
 # adjustFactor = 0  
 adjustFactor = 10000 \* level # *todo test* return lb - adjustFactor # include level in priority so that higher level subProblems get precedence  
  
  
def expandSubProb(subProb, source, dest): # expand a problem with given source and destination  
 newRCM = subProb.rcm.copy()  
 newRCM[source] = float('inf') # set row to inf  
 newRCM[:, dest] = float('inf') # set col to inf  
 newLB = subProb.lb + subProb.rcm[source][dest]  
 newRCM, newLB = reduceMatrix(newRCM, newLB)  
 level = subProb.level + 1  
 key = calcPriority(newLB, level, len(subProb.rcm))  
 curPath = subProb.path + [source] # build list of cities indexs that have been used  
 newProb = BBsubProblem(newRCM, key, newLB, level, curPath, dest)  
 return newProb  
  
  
def initialRunsAlgo(ncities): #Return how many times to run the random algorith for initial bssf. Depends on #cities  
 return 5 # *TODO update this with more interesting function depending on problem size*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  
 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>  
 '''  
  
 def greedy(self, time\_allowance=60.0):  
 pass  
  
 ''' <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):  
 results = {}  
 cities = self.\_scenario.getCities()  
 ncities = len(cities)  
 updatesToBSSF = 0  
 numStatesCreated = 0  
 numLeavesFound = 0  
 numPruned = 0  
 maxQueueSize = 1  
 bssf = self.defaultRandomTour().get("soln") # *TODO use gready?* for i in range(initialRunsAlgo(ncities)): # Run a faster algorithm n times and take most optimal solution as initial bssf  
 solution = self.defaultRandomTour().get("soln")  
 if solution.cost < bssf.cost:  
 bssf = solution  
 start\_time = time.time()  
 rcm = np.full((ncities, ncities), float('inf'), dtype=float) # *todo faster if int* rcm1 = [[float('inf') for x in range(ncities)] for y in range(ncities)]  
 # Fill cost matrix  
 for i in range(ncities):  
 for j in range(ncities):  
 rcm[i][j] = cities[i].costTo(cities[j]) # *TODO convert to int?* # Calculate reduced cost matrix  
 lb = 0  
 rcm, lb = reduceMatrix(rcm, lb)  
 level = 0  
 key = calcPriority(lb, level, ncities) # includes the level in priority for better depth  
 curPath = []  
 root = BBsubProblem(rcm, key, lb, level, curPath, 0) # start at city 0 always  
 numStatesCreated += 1  
 hq = []  
 # Start priority queue  
 heappush(hq, root)  
  
 while len(hq) > 0 and time.time() - start\_time < time\_allowance:  
 # Take top from queue  
 maxQueueSize = max(maxQueueSize, len(hq))  
 subProb = heappop(hq)  
 # Prune? Final?  
 if subProb.lb > bssf.cost:  
 numPruned += 1  
 continue # Skip this subproblem  
  
 # Expand into subproblems  
 cityIndex = subProb.cityId  
 for to in range(ncities): # check all possible desitantions  
 if subProb.rcm[cityIndex][to] + subProb.lb > bssf.cost: # ignore cities where path to is greater than current bssf  
 numPruned += 1  
 continue  
 newProb = expandSubProb(subProb, cityIndex, to) # create new subproblem  
 numStatesCreated += 1  
 if newProb.lb >= bssf.cost:  
 numPruned += 1 # pruned because solution had too high a lower bound after reduction  
 continue  
 if newProb.getDepth() == ncities: # if it could be possible solution (max depth)  
 numLeavesFound += 1  
  
 possibleSolution = TSPSolution([cities[x] for x in newProb.path]) # create the solution  
 cost = possibleSolution.cost  
 if cost != float('inf') and cost < bssf.cost: # compare to bssf and update if better  
 bssf = possibleSolution  
 updatesToBSSF += 1  
 else:  
 heappush(hq, newProb) # push new problem onto queue  
  
 # Return values  
 end\_time = time.time()  
 results['cost'] = bssf.cost  
 results['time'] = end\_time - start\_time  
 results['count'] = updatesToBSSF  
 results['soln'] = bssf  
 results['max'] = maxQueueSize  
 results['total'] = numPruned + numStatesCreated  
 for leftOverState in hq: # include the number of unused states on the queue that would have been pruned  
 if leftOverState.lb > bssf.cost:  
 numPruned += 1  
 results['pruned'] = numPruned  
 return results  
  
 ''' <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