Lab 11 Post-Lab Report

Name: Peter Haynes
Student id: prh7yc
Date: 12/4/19
Section: Tuesdays 12:30

Pre-Lab:

Time:
The big-theta runtime of my Topsort algorithm is $O(n^2)$ where n is the number of distinct elements to be sorted. This draws from the while loop. Since the loop pops exactly one element from the queue in each loop and pops n total elements, it must iterate through the loop n times. Additionally, within the loop there is a for loop that iterates through once for every element in the vector (n times). Therefore, the algorithm loops n^2 times as it contains an n-times loop within an n-times loop. This gives the big theta runtime.

Space: The algorithm requires 6 total variables:

1. vector< vector<int> > graph - First vector requires 24 bytes plus space for every element which in this case are more vectors (of which there are n) which require 24 initial bytes plus 4 bytes for every integer of which there are n for a total space usage of:
    a. Space = $24 + n(24 + n(4)) = 24(n + 1) + 4n^2$ bytes where n is the number of unique elements.
2. int indx = 4 bytes
3. int size = 4 bytes
4. unordered_map<string, int> indices = 40 bytes for the unordered_map plus 36 for every element within the map (24 for the string, 4 for the int, 8 for the pointer between them) for a total of :
    a. Space = 40 + 36n bytes where n is the number of unique elements
5. unordered_map<int, string> backind = same as indices: Space = 40 + 36n bytes
6. unordered_map<string, int> ndegree = same as indices: Space = 40 + 36n bytes

In-Lab

Time: The big-theta runtime of the traveling salesman algorithm is O(n·n!) where n is the number of cities to be visited. This number was gained by looking at the while loop. Since the while loop iterates through every permutation of the n cities to be visited it must loop n! times as there are n! possible iterations of n cities. Additionally, within each permutation the loop runs computeDistance which has O(n) runtime. This gives a total of n·n! operations. Thus, the big-theta runtime of the algorithm is O(n·n!) .

Space: The algorithm itself uses very little space. The only space the algorithm uses is the space to store the length of the final path and the space to store the final path itself. This totals out to 4 bytes for the float, 24 bytes for the vector to store the cities in the path, and 24 bytes for every string stored in the vector. All totaled up, this algorithm requires 28 + 24n bytes of space where n is the number of cities to visit to report the final length and the final path. However, to run the algorithm, the algorithm needs three additional variables:

1. A middleEarth object - The storage for this object breaks down into 8 objects
    a. int num_city_names - 4 bytes
    b. int xsize - 4 bytes
    c. int ysize - 4 bytes
    d. vector<float> xpos - 24 bytes for the vector plus 4 for every float stored of which there are n (one for each city in the map)
    e. vector<float> ypos - same as xpos
    f. vector<string> cities - 24 bytes for the vector plus 24 bytes for each of the n cities
    g. float *distances - 8 bytes for the pointer, 4 for the float
    h. map<string, int> - 24 bytes for the map plus 36 for every city included in it (24 for the string, 4 for the int, 8 for the pointer)
    i. This gives a total storage space of
        i. Size = 4 + 4 + 4 + 24 + 4n + 24 + 4n + 24 + 24n + 12 + 24 + 36n
            1. = 120 + 68n bytes for a MiddleEarth object where n is the number of cities in the map.
2. A string for the start city - 24 bytes
3. A vector of the cities to visit - 24 bytes for the vector + 24 bytes for every city to be visited for a total of 24(n + 1) bytes.

Accelerating Techniques

1. Acceleration technique - Taking a bottom up approach that cuts out repetitive calls using dynamic programming otherwise known as T**he Bellman-Held-Karp algorithm** - Rather than starting at the start node and iterating through every possible path to the bottom node to find the bottom path, this method starts with the final path - the path back to the start node - and iterates backwards by recursively finding the shortest path through the powerset of the cities to be visited. For instance, the first path requires n steps as each path as you have to map every city that isn't the start city to the start city. Additionally, the second path requires n - 1 moves for each existing path as it has to check every city that isn't the start city or the city mapped to the start city. However, from here we can start cutting down our recursive calls. Say your first city was city x and your second was city y, this leaves you to check for the shortest path of the subset that doesn't include x and y to attach to the end of the path from x to y to get the shortest path. However, you can use this same path to build the shortest path added to the path from y to x. Thus, we have cut off one recursive call. We can continue to do this for every pairing and for every layering. For instance in the next step say we have cities x, y, and z. Now, we have cut off two recursive calls for each grouping as we can find it once and use that recursive call for all three orderings of the cities. As the function continues, this strategy cuts out more and more recursive calls, eventually cutting our big o runtime from factorial to exponential - ($O(n^2 2^n)$). While this still essentially iterates through every path, it cuts down the number of steps required to iterate through every path greatly thus accelerating the program. While with small values of n this won't make much of a difference, in larger values of n it will make significant impact. For instance, if we're visiting just 10 cities, this cuts down our number of operations from 3,628,800 to just 102,400 which would speed up our program by a factor of 35.44.

2. Approximation technique - Approximation techniques are techniques that don't necessarily give the exact right answer, but that give one that's pretty good. For instance, the **Nearest Neighbor** algorithm is a way of finding the shortest path by simply going to the closest neighboring city every time you leave a city. This approximation gives a path that isn't necessarily the shortest but on average yields a path that is only 25% longer than the actual optimal path. While it often does return an answer close to the best answer, there are certain arrangements of cities where this gives a very poor answer or possibly even the worst possible answer. However, there are also arrangements that end up actually giving the shortest path. Admittedly, the description of this algorithm is drawn mostly from Wikipedia. However, with such a simple algorithm it's hard to really expand upon the idea and give a higher-level approach to the problem understanding. Since this is a fairly simple algorithm, simply finding the closest unvisited city at each

stop and going there, the runtime of the algorithm is much better than our brute-force algorithm. Since there are n cities, the algorithm must make n steps. At each step, it must consider a maximum of n - 1 other cities to decide which city to go to next. Thus, for all n cities, the maximum number of steps at that city is (n-1) meaning the algorithm runtime is certainly upper bounded by n^2 which means the algorithm is an element of $O(n^2)$ and in practice, the number of steps is the summation of all numbers from 1 to n with n being the number of cities to visit.. Again, with small numbers of cities (say 3 or 4) this wouldn't make a *huge* difference in our runtime. However, with large inputs the runtime would be much much lower. For instance, using our same methodology as the last optimization, looking at ten cities to visit, we would cut the number of steps down from 3,628,800 to just 55 steps. While this wouldn't produce the exact correct path, reducing our runtime by a factor of 65, 978 while producing what will likely be a *pretty good* path seems to be a fair trade off.

3. Approximation technique - **Ant colony algorithms** are relatively new algorithms initially proposed by Marco Dorigo in 1993 based around the behavior of ants following the pheromone trails of other ants to find food sources, shelter, etc. By following other ant's paths these ants are able to quickly find an efficient route to whatever they are searching for. Applying this concept to computer science, any colony algorithms send a group of virtual ants out into a path to search for a particular node. When that node has been found, the ants release pheromones along their path (i.e. the computer weights their path more) in inverse correlation to the length of the path. Essentially, in each group of ants, the ant who found the shortest path weights their path the most while the ant with the longest path weights theirs the least. From this data, future ants going into the program are able to make better decisions on which paths to follow as they know which edges have typically given previous ants shortest routes - for instance, if an edge between two particular nodes has been on the shortest path in 8 out of 10 trials, it's likely that this edge is truly in the shortest path and thus the ants are more likely to follow it. After many groups of ants have been released, there exists a most heavily weighted path that acts as the working shortest path for the Traveling Salesman Problem. This is an approximation algorithm so it doesn't guarantee that the path the ants found is truly the shortest (all though it could be!). However, they do create near-optimal answers. It's difficult to find good asymptotic runtime analysis on these problems, however what I have found is an asymptotic runtime of $O(\frac{1}{p}\ n^2\log(n))$ for a set of n cities to visit, an evaporation rate (of the pheromone trails) or p, and using n ants to explore. This is difficult to conceptualize within our context as the pheromone evaporation rate changes from case to case and we certainly never did anything with an evaporation rate in our study of the problem.

However, using the $n^2\log(n)$ given in the big O analysis and our typical analysis of 10 cities, it would cut our operations down from 3,628,800 to 230 operations (using the natural log as our logarithmic element) which would be a reduction by a factor of 15,760 making our algorithm run significantly faster. Yes, the evaporation rate will make this number of steps go up as the evaporation rate will be < 1, but the final result will still be significantly shorter than the brute-force method we constructed.