Eza Rasheed

er6qt

04-24-19

postlab11.pdf

Time and Space Complexity:

Pre-Lab

Implementation/Time Complexity: My implementation for the pre-lab included first, making a class called graphCourses that was basically a class for a vector node. Each vertex had a variable int edges that was the number of indegrees coming into a node, a string vertex which stored the name of the vertices and a vector called adjList that consisted of graphCourses* node pointers which stored the adjacent nodes of the particular node being passed in. Then, in my main method, I made a vector of node pointers called graph that stored the nodes(vertices). Using this, I read in the input file and for the 2 string passed in, I iteratively went through each line in the vector of nodes to check if a vertex in the vector was the same as the string passed in, and if so, I set the vertex to that position in the vector. Since it was done iteratively, it has big theta run-time of **O(n)**. If the vertex was not present for the 2 strings read in, I created a new node for the vertex and updated the adjacency list vector for each new node created by using push_back on the vector. This as well has a linear run time of **O(n)**. This was done for both the first string read in, and the 2nd, so it will have a run time of **O(n^2)**. At the end of the main, I did my implementation for topological sort on that vector so that it can sort the nodes in the vector and print out the course names of the vertices when they got to an indegree(edge) of 0. For my topological sort implementation, which I modified using Bloomfield's slides, I had a queue of vectors that checked to see when a vertex reached an indegree of 0 so that it could be pushed onto the stack of vertices (course names) that also had an indegree of 0, and then sort them in order. For the first vertex pointer v*, I iteratively ran through each element in the vector graph, which held the list of nodes, and if the vertex had an indegree of 0, I pushed it onto the queue. This had a linear time **O(n)**. From there, I checked that while the queue was not empty, the value of the front of the queue is set as vertex v, and then popped from that queue so that it could be print out the course name of that vertex as we continue to go through the queue of vertices with indegrees of 0. Then, I had a for loop that went through each of the adjacent nodes in the adjList vector and gave the elements of the vertex v to the adjacent vector w, and decremented the amount of indegrees coming into vertex w now. If the indegree for vertex w equaled 0, then it was also pushed onto the queue holding the course names in order. The runtime for adding the adjacent nodes in the vector depends on the leftover number of adjacent nodes to add onto the queue, so this number is constant. Therefore, the overall runtime of the topological sort is **O(n^2)**.

The user time it took to get the output for the input prelab-test-small.txt was 0.000s, while the time for the input prelab-test-full.txt took 0.002s.

Space Complexity:

For the space complexity of the data structures in the pre-lab, my graphCourses class consisted of int edges, which is 4 bytes, a string vertex, which can be 1-4 bytes, and a vector of adjacent nodes, in which each node pointer is 4 bytes, and the vector is compromised of these element nodes that are 4 bytes each, so it's the size of the vector(whatever that is since it's varied) multiplied by 4 bytes. Adding all these bytes together gets the space complexity of the class. For the main method, it consisted of a vector called graph that stored the list of nodes, so the space complexity for that is once again calculated by multiplying the varied size of the vector by 4 bytes. Then I had 2 boolean variables, which is a total of 2 bytes, and then 2 int variables used as resetters, which is a total of 8 bytes. Lastly in the main, I had the topsort method which consisted of 2 pointers w* and v*, so 8 bytes total, and then the size of the queue to implement the topsort method, which has a varied size depending on the number of nodes in it. Adding all these bytes together gets the space complexity for my main method.

In-Lab

Implementation/Time Complexity: In my inlab implementation, I used the skeleton code given to us for the TSP problem. For starters we had a computeDistance method that we had to fill in the code implementation for. This method computed the full distance of a cycle that starts at the start parameter and goes to each of the cities in the destinations vector in order, and ends back at the start parameter. To implement this, I iteratively went through each of the elements (cities) in the destinations vector, which has a linear **O(n)** running time, and added the distance between the current city being passed through and the next city at some iteration index "i". Then I added the distance back from the final location back to the start to get the full cycle's distance. In the other method given to us, printRoute, we had to pint the entire route, staring and ending at the 'start' parameter. The implementation for this was very simple. In a for loop, I iterated through each element in the destinations vector, which has a linear **O(n)** runtime as well, and printed out the destination name at each index "i" and then printed out the start at the end, since the ending city is the same as the starting city. In the main, we had to take in 5 command parameters: x-size(width) of the world, y-size(height) of the world, the number of cities in the world, the random seed, and the number of cities to visit. Then, in order to create the world and select your itinerary, they gave us code in which a MiddleEarth object me was created by taking in all the above parameters, except for the number of cities to visit. Then, a vector of the list of destinations was created by using getItinerary. Then I created a copy of this destinations vector so that I could use and update it after I print out the shortest path found in the permutations. I then initialized a flat variable to set the shortest path found. Next, before going through next_permutation, you have to sort the vector, which takes **nlog(n)** time to sort a vector that is of length n. After doing this, I went through the permutations in a while loop, which iterated from the beginning of the vector to the end. Next_permutation has a run time of (**n!)**, which makes sense because a large n number of elements take a long time to run. As each possible path was being calculated, I set the shortest path variable to be the shortest one found while iterating if the found distance was less than the previously computed shortest distance. Next, I updated the vector of destinations to hold the shortest distance path. Lastly, I called printRoute on the

updated vector to print out the path that had the shortest distance, which gives you the route of the destinations to visit. In all, since there is an n! number of permutations, as mentioned before, for n number of cities, the time complexity for the TSP problem is **O(n!)**.

The user times it took to get the outputs for a random seed of 14, world size of 20x20 with 20 cities, and various path lengths(1-10) were:

1. 0.002s
2. 0.000s
3. 0.002s
4. 0.002s
5. 0.003s
6. 0.010s
7. 0.056s
8. 0.483s
9. 4.445s
10. 48.963s

Space Complexity:

For the space complexity of the data structures in the in-lab implementation, the MiddleEarth object me consists of int width(4 bytes), int height(4 bytes), int num_cities(4 bytes) and int rand_seed(4 bytes). Therefore, in total, the MiddleEarth object has a space complexity of 16 bytes. In the main method, the vector that stores the list of destinations has a space complexity of the size of the vector(varied?) multiplied by the size of the string elements consisted within it, which are of size 1-4 bytes each. To find the shortest path(find_dist) and set the shortest path(shortest_dist), I had two float variables in my main method, which are 8 bytes each, and then in the computeDistance method itself, I had another float variable that would get the distances between the cities(get_dist), which also takes up 8 bytes. Adding up the space taken up for all these individual data structures gets you the space complexity of the whole program.

Acceleration Techniques:

Nearest Neighbor: Nearest neighbor is a constructive heuristic, which is under the category of heuristic and approximation algorithms. In this technique, the shortest edge is found to connect the first city, which is the start city, to the next city. Then, that city is marked off as found and in order to find the next city, you look at the marked city to determine what the closest point/city is from there, which would be the shortest path/distance. At the end, when you have marked off all the visited and unvisited cities, you can look at the visited cities to gather information on the shortest path possible and return back to the starting city. This technique has a worst-case runtime of O(n^2).

Branch and Bound: Branch and bound is an algorithm under the category of exact algorithms. In this technique, you find the best solution by looking at all the branches from the root of the tree. At each branch, you examine and compare the upper and lower bounds of the best solution with the previously found best solution, and if it is better, you can trash the previous tree branch. It does this by looping through the branches that contain the solutions and comparing and replacing

a newly found better branch with the best branch found previously. This is efficient in getting rid of extra space in memory as it deletes the unneeded branches and therefore, the memory in which where they were stored in, as it continues to find the next new best solution. As well, it is a reliable solution as it stops and checks at each branch before for the best solution before arriving to a final solution, unlike the nearest neighbor algorithm. The runtime for this is typically exponential, and it may require exploring all possible permutation in their worst case.

Particle Swarm Optimization: Particle swarm optimization is an optimization algorithm under the category of heuristic and approximation algorithm, and under the subcategory of randomized improvement. In this optimization technique, a population of particles is used as individuals and each particle is a representation of a feasible solution for the shortest path/distance possible to travel through. In implementing this, the positions of the particles change by flying in a multidimensional search space, and the index of the best particle(solution) is where the swarm moves to. This is then updated as better positions are found by other particles. After much research, I could not find the runtime of this optimization algorithm.

Of these three techniques, I believe branch and bound would be the best acceleration technique to optimize my in-lab code. It looks at the whole program before finding a solution. The TSP problem implemented in the in-lab looks at all the possible path combinations, and branch and bound looks at each branch one by one, determining the possible best branches, and comes up with one final solution. This and the aforementioned advantages of branch and bound would make my code and its runtime faster.

Sources:

https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm

http://demonstrations.wolfram.com/TheTravelingSalesmanProblem3NearestNeighborHeuristic/

https://en.wikipedia.org/wiki/Branch_and_bound

https://www.geeksforgeeks.org/branch-and-bound-algorithm/

https://en.wikipedia.org/wiki/Particle_swarm_optimization

https://www.sciencedirect.com/topics/computer-science/particle-swarm-optimization

.