Ashley Nguyen - apn2my
Postlab11 - postlab11.pdf
4/27/16

Complexity Analysis

Pre-lab - topological.cpp

- *Time complexity:*
  - The main function in topological.cpp reads in two strings from the file and searches the adjacency list (implemented by a vector of nodes) to determine if the nodes match the first string read in from the file, this is a Big-Theta(n) operation for each step by step search through the list for n nodes. If the string is already in the list vector, then the push_back() operation is performed on the node, which takes Big-Theta(n) even in the possible case where the entirety of the vector must be copied over and repopulated by nodes. These two operations are repeated when reading in the second string from the file. Thus this is a running time of Big-Theta(n^2).
  - The topological sorting function was implemented using a stack and pushing nodes onto the stack. When pushing n number of nodes, this operation is Big-Theta(n). The running time for the loop to add adjacent nodes to another node depends on the number of nodes there are to add to the stack, thus this number is essentially constant. Therefore, the overall running time for the topological sort is Big-Theta(n^2).
- *Space complexity:*
  - graphNode Class
    - String = 4 bytes
    - vector<graphNode*> of 100 elements = 4 bytes * 100 = 400 bytes
    - Int indegree = 4 bytes
    - Total = 408 bytes
  - Main
    - vector<graphNode*> of 100 elements = 4 bytes * 100 = 400 bytes
    - 2 boolean checks = 1 bit + 1 bit = 2 bits
    - 2 int resetters = 4 bytes * 2 = 8 bytes
    - Total = 408.2 bytes
  - topSort
    - Stack = ~40 bytes (variable depending on number of nodes)
  - Total = 856.2 bytes

In-lab - traveling.cpp

- *Time Complexity*
  - Since the number of elements in the vector is known (33 total cities in Middle Earth), it can be assumed that the vectors creating Middle Earth will not have to be resized, thus push_back() operation has a running time of Big-Theta(1). Since the 2D array accesses two vectors, xpos & ypos, that are known values thus this array runs at Big-Theta(1) also. Therefore, the constructor operates at running time of Big-Theta(n).
  - The next_permutation call is quite a costly operation, contributing the most to the overall running time. Since there are n! permutations where n is the number of cities to consider

when creating an itinerary of cities to travel to (in an attempt to find the shortest path), the overall running time is Big-Theta(n!).

- *Space Complexity*
  - Middle Earth Class
    - Int num_city_names + int xsize + int ysize = 4 bytes * 3 = 12 bytes
    - vector<float> xpos = 4 bytes * xpos
    - vector<float> ypos = 4 bytes * ypos
    - vector<string> cities = 4 bytes * num_cities
    - Float distances = 4 bytes
    - map<string, int> indices = 4 bytes * 4 bytes * indices
    - Total = (4*xpos) + (4*ypos) + (4 * num_cities) + (16 * indices) + 16
  - Main
    - MiddleEarth = (4*xpos) + (4*ypos) + (4 * num_cities) + (16 * indices) + 16
    - vector<string> dests = 4 bytes * num_cities
    - String s1 = 4 bytes
    - String s2 = 4 bytes
    - vector<string> final = 4 bytes * num_cities
    - Float smallestDist = 8 bytes
    - Float distance = 8 bytes
    - Total = 24 + 2(4 * num_cities)
  - computeDistance()
    - Float result = 8 bytes
    - String start = 4 bytes
    - vector<string> dests [i] = 4 bytes
    - Total = 16 bytes
  - Total = 56 + (4*xpos) + (4*ypos) + (16*indices) + 3(4*num_cities) bytes

Accelerations techniques (for TSP problem)

1. Approximation Technique - Nearest Neighbor

This heuristic technique depends on the current city visited. From that city, the shortest edge is determined to connect the current city to the next unvisited city. Thus, the next city is dependent on the previous city because it is determined by finding the nearest city from the previous city. This algorithm requires some way to keep track of visited and unvisited cities. Once all the cities have been marked as visited, the route effectively created is quite short although it is not the fastest algorithm. This has a running time for Big-Theta(log |n|) which is quite an improvement on my code that runs at Big-Theta(n!).

2. Acceleration Technique - Large-step Markov chains

The fundamental principle of this technique implements a local search algorithm to make the inputs more manageable. Mini itineraries are created by constructing locally efficient tours based on prior routes rather than random ones. Overall, the intention is to create an all-encompassing route that prioritizes the

cities by local minimum distances (the shortest length). This is repeated for each step in the Markov chain, bias is given to local "tours" of the shortest length which contributes to creating the entire chain of the route of the Traveling Salesperson. By embedding optimized local searches into a complete optimized solution, the run time is roughly Big-Theta(n). My code implemented did not perform local searches and compared distances from all the cities Middle-Earth rather than compared local cities that the user was actually interested in. This acceleration technique reduces the need for the random next_permutation call by localizing the search as a step in the chain.

3. Prim's algorithm for Minimum Spanning Tree

The minimum spanning tree intends to reduce the number of stops visited in the route. Although any city can be the root, it is most intuitive to make the starting city as the root. The edge with the smallest weight (for TSP purposes, most likely the shortest distance) is connected to a known vertex from an unknown one. The process of removing smallest weight edges is until all the vertices have been deemed "known". The result is a minimum spanning tree that will be traversed in preorder recursively to refine the shortest route for the traveling salesperson. Although there are many algorithms to construct a minimum spanning tree, the one described here is Prim's algorithm. The resulting distance is far more optimal than the brute-force code implemented in the in-lab however, the time is not as efficient as the large step Markov chain. The running time for this algorithm is Big-Theta(e log(n)), which is quite efficient compared to my code. This is because different paths are constructed in a MST that do not require to be rechecked when calculating the most efficient route, whereas the brute-force method constantly checks each path and the distance as a city is added onto the vector.

Sources

http://www.complex-systems.com/pdf/05-3-3.pdf
https://en.wikipedia.org/wiki/Travelling_salesman_problem#Exact_algorithms
http://demonstrations.wolfram.com/GreedyAlgorithmsForAMinimumSpanningTree/