Adarsh Solanki
As5nr
4/26/12
CS 2150 Post-Lab 11

Time Complexity

The traveling salesman algorithm that I implemented has a time complexity that is a function of a few variables. The inputs that I foresee as having an effect on the time complexity of the algorithm are the number of cities in Middle-Earth and the number of cities visited. I will represent these as C (for cities) and V (for visited) from here on out.

C affects the program by creating more cities in the instance of Middle-Earth that is created. This affects the creation of Middle-Earth, and perhaps the computeDistance() function. Actually, it turns out that C does not have an impact on the time complexity, as the getDistance() function is an amortized constant time complexity because the distances are pre-computed and stored into an array.

That means that the important input is V, or the number of cities to be visisted. The program iterates while a permutation can be found of all of the cities (except for the start-point) so the complexity of that step is O((V-1)!). Next, in each of the permutations, during the computeDistance() function, there are a number of computations that depend on the size of the vector dests, as it iterates through the vector. This is linear time, so the time complexity of this step is O (V). Together, the parts multiplied together result in a time complexity of O (V*(V-1)!) or O(V!).

Space Complexity

In terms of space complexity, at first glance the algorithm seems to use the most space holding the instance of Middle-Earth itself. This is because regardless of the number of cities to visit, the instance of the world will always contain a vector containing all of the cities in the world, and also a 2d array holding the distances between each of the cities to each other. This itself is a linear space and a polynomial complexity, respectively, based on the number of cities in the world, or C as previously referred to. This is $O(C + C^2)$. Each additional city in the Middle-Earth instance leads to one more space in the vector of cities, but the 2d array increases in both dimensions.

Acceleration Techniques

In order to accelerate the implementation of the traveling salesman problem, there are a number of possible angles to begin optimization. The first and most obvious place to begin is to reduce extraneous distance calculations.

In the same way that the distances between individual cities are stored in a constant-time data structure, the same could be done for multiple-city paths. That is to say, once you compute the distance to Bree through Isengard from Minas Tirith, you should never compute it again. Instead, some data structure should hold all previously-computed path/distance combinations. This will probably end up taking a lot of space, but that is the cost to the increase in time-complexity and eviction of redundant clock-cycles. The space would need to be at worst factorial complexity because of the possibility of storing all of the paths.

Next, it could be possible to try to create a heuristic that would reject paths that pass through city-sequences that are over costly. More specifically, if one of the permutations “passes by” a city on the way to another (which can be determined by consulting the coordinate pairs of each city) then the permutation can be considered inefficient and iteration halted. In order to implement this pseudocode, perhaps one could calculate angles and create a threshold angle, say 40 degrees. If a city exists within 20 degrees on either side of the destination city at a lesser distance, than the permutation can be considered “inefficient” because it “passes by” a city. This threshold angle would probably need to be chosen carefully to optimize this optimization itself.

Lastly, another optimization of my traveling salesman implementation could be to pre-determine the optimal destinations from each given city and use that to determine if a permutation is “efficient” or not, similarly to the previous optimization. For example, if city A is on the corner of Middle-Earth, and has no near neighbor other than cities B and C, then it only makes sense for A to always go to B or C. The algorithm could check in a constant-time associative data structure that holds a collection of all possible destinations for each given city. If the next city is not an “approved destination”, then the iteration would halt and continue at the next permutation (if any remain). This could even be expanded to work in the other direction as well, that is to say, to store another associative data structure holding a collection of possible cities from which it is efficient to travel to the given city.