TSP Project Report - CS325 Project 5

Group 14 - Chris Sanford, Erin Marshall, Jonathan Flessner
5 December 2014

Description of the algorithm:
This algorithm is a modification of simulated annealing (SA) to solve the travelling salesman problem (TSP). SA was chosen to be modified for this assignment because even though some other approaches may work better for finding the absolute best solution, SA works well at getting a good solution in a reasonable amount of time. The algorithm starts by generating a route (tour) of every city in the given input file. The way the program does this is different from a traditional SA method where a completely random route is always chosen. In this algorithm, the route is chosen between a greedy/nearest neighbor approach, a modified greedy approach, or a totally random route. For the modified greedy approach, a random start value is generated, then the two closest cities to that start value are found. One of them becomes the front value, the other the back value. From there, each of those cities (front and back) alternate finding the next closest unvisited city to them until the circuit is complete. This approach was chosen over a traditional greedy method, where the next closest unvisited city is always chosen, after outperforming it and a strictly random route in testing. Still, to get the most varied results, these are weighed and any can be chosen, with the modified greedy having the best odds, followed by the greedy and lastly the random approach. After this first route is completed, we begin the SA process.

The SA process involves a starting temperature, an ending temperature, and a cooling rate. Then, while the temperature is still hot enough, two cities are randomly swapped, and then the value is testing to see if it has improved (meaning the total cost has decreased). The current starting temperature is set to 75,000; more testing is still needed to try and determine an optimal value. Unfortunately, one of the reasons this problem is so difficult is because with each set of cities, it appears a different temperature is optimal. Because of that, 75,000 is a current working best estimate. I am using an ending temperature of just zero, and cooling by 1 each time. Many SA approaches use a cooling factor while this approach just simply decrements the temperature by one on each iteration to cool. When a value is found, that is better than the currently held best distance, the temperature is reheated to the starting value. Once the temperature is all the way cooled, that entire route is discarded and we restart the process with a new route.

One of the major optimizations of this program is how the distances are calculated. On loading the program, a triangular matrix is created holding the distance of every city to every other city. That way, the total route distance can be calculated much more quickly as it just needs to sum the distances between each city instead of calculating them and summing them each time. The triangular matrix was a strong optimization also, taking less than half the time to load as the full matrix over a large number of cities.

Usage:

```
python saTemp.py <infile>
```

Pseudocode:

```
get cities from input file
calculate in triangular matrix the distance between each city
    set route using one of the methods
    swap cities
    determine if the swap decreased the cost
    swap back if the cost was not useful
    keep the swap if it was useful and update the distances
    cool the temperature
    repeat until the temperature has completely cooled
    restart with a new route
loop infinitely until time is expired or there is a keyboard interrupt
```

Improvement wish list:

This is a monumental problem, and a ton more time could obviously be spent making improvements. A first immediate desire to improve would be to implement a function that calculates the distance of the route after having swapped two cities, by only looking at the distances that were affected by swap. This optimization would noticeably improve running time over large problem sets.

Spending more time on the temperature and cooling factor would be the next (and much larger) improvement to make. Currently, this implementation is serviceable, but could certainly be improved. It would be nice to have a dynamic system based on the number of cities input, or even their average distances to each other.

It would be great to add the option to accept a swap, even if it was slightly worse than the previous value, but the temperature was also low. This would possibly allow for avoiding more false peaks in the results to get better results.

We would have also loved to have been able to hone our ant colony optimization (see below) to get better results as this was a very interesting algorithm and very promising.

Switching Algorithms:

When we started this project we set our sights on implementing an ant colony optimization algorithm. We were successful and tried many different modifications. Unfortunately, it just wasn't giving us the results that we wanted. So, we began work on the SA algorithm. This allowed us to get better results on a more varied set of inputs. Our ant colony was only able to outperform the SA in example 1. The ACO produced far worse results in the other examples. The Ant Colony program (ACO.py) is included to show our work.