**Traveling Salesman**

Tyler Hamilton

CECS, SID 1713904

Speed School of Engineering

University of Louisville

jthami05[@louisville.edu](mailto:NameID@louisville.edu)

1. **Introduction** (What did you do in this project and why?)

In this project, I implemented a brute-force approach to solving the Traveling Salesman Problem (TSP). The purpose of this assignment was to introduce us to to the TSP, and to devise a simple process for solving the problem. Part of the learning process for this assignment was realizing that the brute-force approach to solving the TSP is extremely inefficient, and with even moderately large data sets, can bring a modern computer to its knees.

1. **Approach** (Describe algorithm you are using for this project)

The algorithm I used to output every permutation of the inputted cities in the problem was a recursive Heap's Algorithm. Heap's Algorithm, when used recursively is simple to implement but frustratingly difficult to completely visualize. The algorithm takes in an integer representing the number of elements in the array, and the array object itself. We run a for() loop continuously up to the number of elements in the array (N) minus 1, and inside this loop we call the algorithm function again with N-1, and then determine whether N is even or odd. If N is even, we swap the i'th element with the N-1'th element. If N is odd, we swap the 0'th element with the N-1'th element. Outside of the loop, we once again call the algorithm function with N-1 again. If the algorithm is called and N = 1, then the array will be outputted in its current order.

This process will output every permutation of the array. In my program, rather than using an array, I used a vector (of floats), as I was coding in C++ for the duration of this project. Because of this, I added elements to the vector with push\_back() and removed elements with erase().

1. **Results** (How well did the algorithm perform?)

The algorithm was highly inefficient. Every single possible route was calculated, and the shortest distance was calculated after all of these routes were determined. Because of this, the algorithm ran factorial(N-1) permutations, and was incredibly slow for datasets greater than 7 cities, and got progressively worse. For my program, the total lengths of each individual route were stored in a vector of floats. This vector was read every time a call to my algorithm finished executing, and compared to the previous shortest distance. If it was less than that, it is set as the new shortest distance. Thus, the last execution of the algorithm will output the shortest route distance of the dataset.

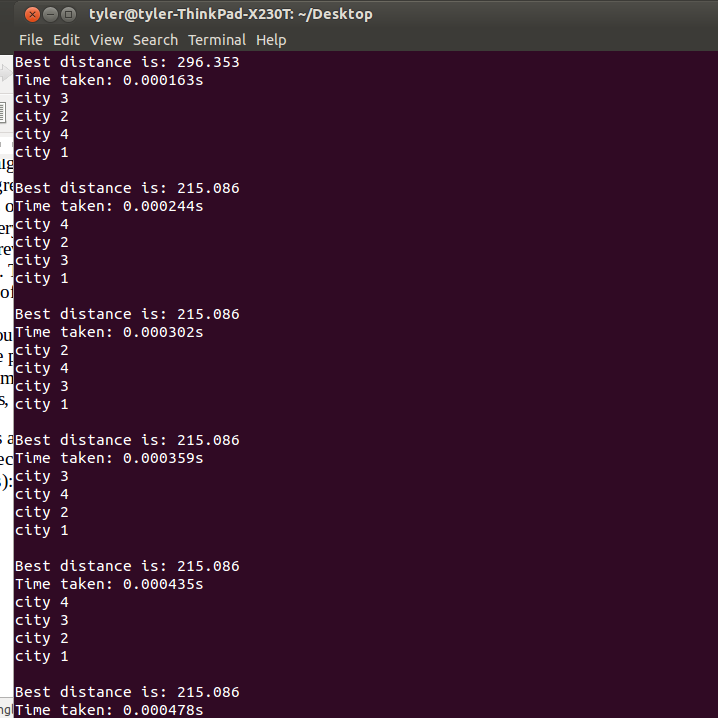
* 1. **Data** (Describe the data you used.)

I used .tsp files given in the project. The name of the file to be read was passed in as a command line argument. I ignored the first sections of the files and read the remaining lines in, as IDs, x-values, and y-values for *city* objects.

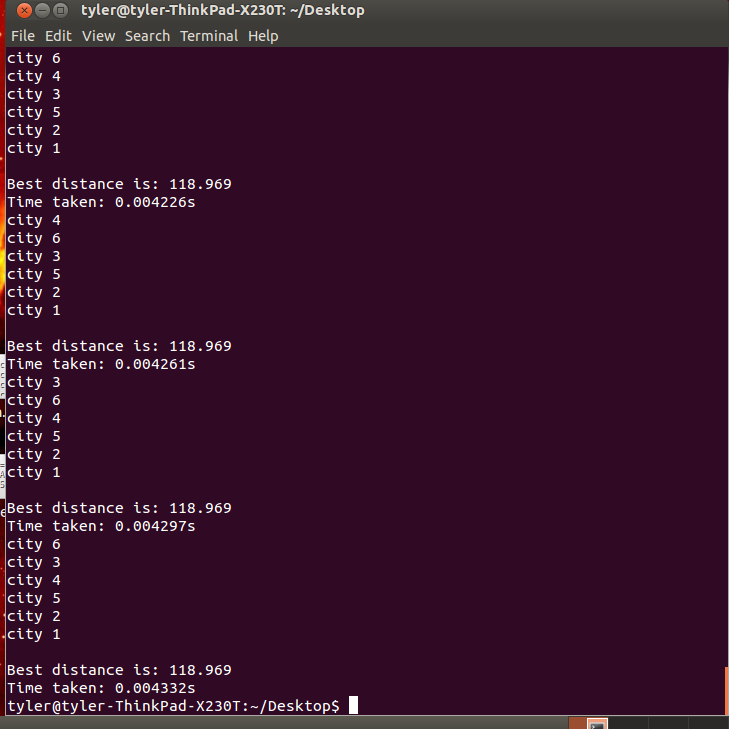
* 1. **Results** (Numerical results and any figures or tables.)

The program works as expected.

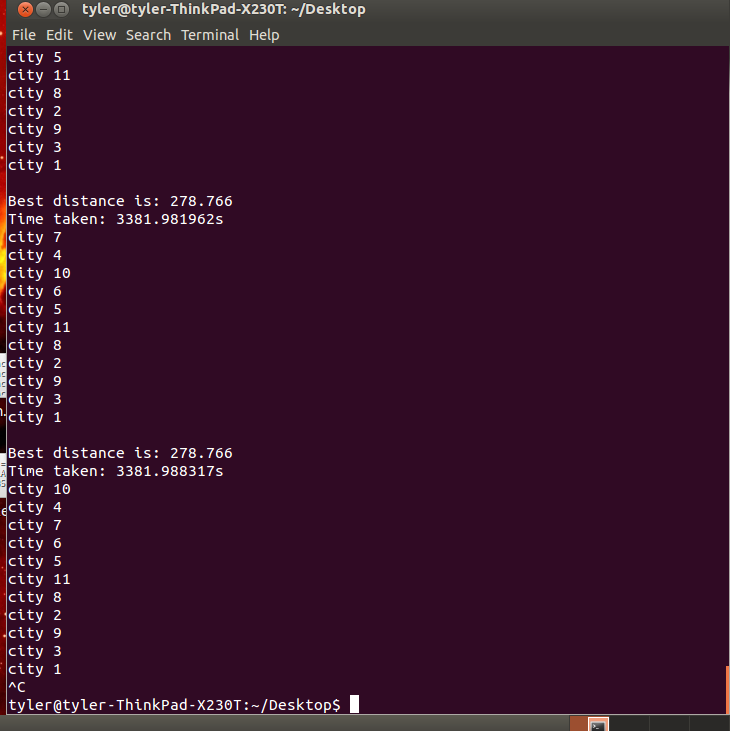
With Random4.tsp (4 cities):

****

With Random6.tsp (6 cities):



with Random11.tsp (11 cities):



1. **Discussion** (Talk about the results you got and answer any specific questions mentioned in the assignment.)

Bugs and memory errors were my biggest hurdle with this project. Eventually, the results I received were logical, but indexing errors, difficulty tracking bugs through the algorithm, and unintuitive debugging messages made this an arduous task.

The distances I received are all set as floats. The numbers below are the IDs of each city in order in the quickest permutation. The times are in seconds, often fractions of a second.

The basic process of my program is as such: The dataset is read in as a command line argument, and separated into city objects with x and y coordinates. The brute force algorithm is then called on the list of cities recursively, until every possible combination has been exhausted, and the total distance per route is placed in a list of distances. Lastly, the best distance is calculated from the list of total distances.

**Random4.tsp**: city 1

city 4

city 3

city 2

city 1

Best distance is: 215.086

Time taken: 0.000310s

**Random5.tsp**: city 1

city 3

city 4

city 5

city 2

city 1

Best distance is: 139.134

Time taken: 0.002581s

**Random6.tsp**: city 1

city 6

city 3

city 4

city 5

city 2

city 1

Best distance is: 118.969

Time taken: 0.004426s

**Random7.tsp**: city 1

city 5

city 6

city 3

city 4

city 7

city 2

city 1

Best distance is: 63.863

Time taken: 0.025662s

**Random8.tsp**: city 1

city 8

city 3

city 4

city 5

city 6

city 7

city 2

city 1

Best distance is: 310.982

Time taken: 0.416415s

**Random9.tsp**: city 1

city 7

city 8

city 3

city 4

city 5

city 6

city 9

city 2

city 1

Best distance is: 131.028

Time taken: 7.961648s

**Random10.tsp**: city 1

city 10

city 3

city 4

city 5

city 6

city 7

city 8

city 9

city 2

city 1

Best distance is: 106.786

Time taken: 429.768815s

**Random11.tsp**: Waited 3381 seconds and stopped

**Random12.tsp**: It wasn't gonna happen

1. **References** (If you used any sources in addition to lectures please include them here.)

Wikipedia page on Heap's Algorithm:

https://en.wikipedia.org/wiki/Heap's\_algorithm