**Traveling Salesman – Brute Force Approach**

Christopher McClure

Electrical and Computer Engineering

Speed School of Engineering

University of Louisville, USA

[cmcclure1](mailto:NameID@louisville.edu)7@gmail.com

1. **Introduction**

This project focused on the perhaps the most rudimentary of traveling salesman problem (TSP) solving algorithms. This simplistic approach provided a good introduction to the TSP, showed some of the problems inherent to probabilistic artificial intelligence systems, and offered a good worst-case baseline for algorithm run-times when faced with high algorithmic complexity.

1. **Approach**

When finding the full path solution to the TSP using a brute force approach, it is necessary to find every possible path that the salesman could take. However, it must be taken into account that the paths taken by the traveling salesman are all circular (i.e. a solution that starts at a point and completes a full Hamiltonian cycle eventually returning to the initial point). Therefore, it was not necessary to find all possible permutations of the cities given. Rather, it was only necessary to find all possible circular permutations [1]. Based on this, my algorithm performed as follows.

First, it was necessary to parse the given .tsp files into usable data. This provided us with the total number of cities (given to us by the dimension entry) and the data for each city (each city being listed as a numerical node and given x and y Cartesian co-ordinates). The first city is taken as the start and finish point for each TSP path. The rest of the cities then arrayed in every possible permutation [2]. After appending the first city to the start and end of the path, the distance was calculated, and the shortest path was reported.

1. **Results**

My initial approach when using this algorithm was to collect all of the possible paths in a single, increment through the array calculating each path’s length, and store the optimal solution that is found. This worked well for all of the sets except those in Random11.tsp and Random12.tsp. These sets resulted in too many permutations for my javascript heap to store, and caused a memory error.

As a solution to this, I modified the program to keep only a “running” best path stored, updating the path whenever a more optimal permutation was found. This allowed me to solve each of the data sets provided.

While a solution was found for each set, it is fairly obvious that this algorithm does not scale well. Aside from the memory constraints, the time it took to solve for each set grew substantially longer. This makes sense, as the number of total paths that needed to be processes will expand radically (O(n!)) [1].

* 1. **Data**

Files were provided (in .tsp format) containing lists of cities and their respective co-ordinates. Each of the provided files was subjected to the brute force algorithm, and the results compiled in the Results section below.

* 1. **Results**
* Random4.tsp

Optimal path found:1,4,2,3,1

Distance required to transverse:215.08553303209044

* Random5.tsp

Optimal path found:1,2,5,3,4,1

Distance required to transverse:139.1335417499496

* Random6.tsp

Optimal path found:1,2,3,4,5,6,1

Distance required to transverse:118.96891407553862

* Random7.tsp

Optimal path found:1,2,7,3,6,5,4,1

Distance required to transverse:63.863031874767636

* Random8.tsp

Optimal path found:1,6,8,4,5,2,3,7,1

Distance required to transverse:310.98207974423167

* Random9.tsp

Optimal path found:1,7,6,3,5,2,9,4,8,1

Distance required to transverse:131.02836613987677

* Random10.tsp

Optimal path found:1,2,7,6,8,5,9,10,4,3,1

Distance required to transverse:106.78582021866472

* Random11.tsp

Optimal path found:1,6,10,11,8,9,7,5,3,4,2,1

Distance required to transverse:252.6844344550543

- Random12.tsp

Optimal path found:1,8,2,3,12,4,9,5,10,6,7,11,1

Distance required to transverse:66.08484401133855

1. **Discussion**

Unfortunately, I was unable to get the provided concorde interface working to test my results. However, the algorithm seemed to run as expected. I was able to verify that smaller sets generated all paths, and the time increase when running the program on larger sets seemed to indicate that the code was correctly scaling to find all permutations.

The language chosen for this code is node.js (a offshoot of javascript). As such, to execute the algorithm, you will need to have the ability to execute node.js scripts. To run brute force algorithm, navigate to the folder containing the tsp.js file, and then type in your command terminal:

node tsp.js Random#.tsp

where # is the respective number of the .tsp file you are testing. The algorithm is set to run any of the .tsp file located in the “bruteTSP/assignment-files” folder. If the files are not located properly, or if the file name is mistyped, an error should be generated.

1. **References**

[1] [Weisstein, Eric W.](http://mathworld.wolfram.com/about/author.html) "Circular Permutation." From [*MathWorld*](http://mathworld.wolfram.com/)--A Wolfram Web Resource. <http://mathworld.wolfram.com/CircularPermutation.html>

[2] Permutation base code adapted from code by SiGanteng: http://stackoverflow.com/questions/9960908/permutations-in-javascript