CS3510 Traveling Salesman Problem
By Ahan Shah and Varun Saxena

PseudoCode:

We have three helper methods to first start

The first one generates the greedy solution.

Greedy Solution

Set tour length = 0.
Create an array called tour sequence and store 1 in it.

Make a set called visited_cities.
Set the current node to 1.
Add this (1) to the visited_cities.

Create a while loop with the termination condition being all the cities are in the visited cities set

Set a variable shortest distance as the largest size possible for the system.
Set the shortest node as None.

Now we will iterate through every city.

Check if the city being looked at is already in the visited set.
If so then we move on to look at the next city.

Otherwise we find the Euclidean distance between the current node city and the i^{th} city we are indexing through and check to see if the distance is less than the shortest distance.
If it is, we found a new shortest distance and a new shortest node.

After the for loop make current equivalent to the shortest node. Then add this shortest node to the visited cities set.

Tour length will increase by this shortest distance and we will append the node to the tour sequence.

At the end of this while loop we will add the distance between the current node and the first node (this is because we need the traveler to return back) and add the shortest node to the tour sequence.
We return the tour length and the tour sequence.

Get Neighbor Path

The next helper method is called getNeighborPath which will reverse the neighbor nodes/paths. The parameters are path and two city nodes c1 and c2.
If both are equivalent then return path.

Otherwise check if c1 is greater than c2. If so then flip them.
Outside the conditional make path1 the path from beginning to the first node inclusive.
Make path2 the path from c1 to c2 (c2 inclusive) and then reverse it.

Make path3 the rest of the path meaning everything past c2.
Return the new path which is path1 + path2 + path3.

Get Tour Length

Another helper method is getTourLength which the only parameter is the path.

Initialize tour length to zero.
Iterate through every city node and add the Euclidean distance to the tour length.
Return the tour length

THE MAIN METHOD

We need to initialize the start_time to the time of the system.
Make a variable called args that will be the arguments passed into the system.

The first argument will be the file that holds the input coordinates.
The second argument is where the output will go to.
Create a variable called time allowed and that will be the third argument passed into the system and this will be reduced by 0.5 seconds to account for writing to the output at the end.

We read the input coordinates file and read the first line into a variable called line.
We initialize a cities dictionary.
Create a while loop that terminates if line is an empty space.

- Split the line on an empty space.
- In the cities dictionary we will store the key as the first element of the line array we created by splitting and we store the value as coordinates which are the second and third elements of the array of line. Convert the strings to floats then integers for faster arithmetic
- Now read the next line of input file.

First we will start with the greedy route.
Generate the path length and the path sequence from the generateGreedy method we created already.

Now we need to create a “temperature” and that will be our time_left variable. Set this to the start time – current time + allowed time.

We need to make a while loop that run until the current time minus the initial time is less than the time allowed

- We need two random cities so set that as index1 and index2.
- Get the neighbor path of these two indexes.
- Get the length of this path as well.

- If this length is less than the current path length.
  - Then set the path length as the neighbor path length.

Set the path sequence to the actual neighbor path.

Otherwise

Set a variable temperature to time_left times 10.

If e^((difference in path length and neighbor length) / temperature) is greater than a random probability.

Set path length to neighbor length

Set path sequence as neighbor sequence

Lower the “temperature” each time by finding the new time left. Continue through while loop.

Write each path sequence to the output file separated by a space.

Why we chose this algorithm?

We chose the simulated annealing algorithm after creating a greedy solution approach. Obviously after writing up a greedy solution we realized that the code would be way too slow and would not be able to get the correct path length in time. Therefore, after talking to Professor Dovrolis we saw that the Greedy Algorithm can be adapted so we aren’t choosing the shortest next distance (which might not be the best path in long term). It was a very similar topic to something we learned in Intro Artificial Intelligence where we choose to explore a different path rather than what is the shortest. After doing a lot of research we found a very interesting concept of Simulated Annealing. This process models the process of heating a material and slowly lowering the temperature to a state of low energy where they are very strong.

In our code this basically means that we keep the current values but at each step we pick a random coordinate points and see if this path is shorter than the path sequence we have currently. If so then we replace the alternate path sequence with this path. Even if it’s not then with some random probability we will still accept this as the new path sequence. We decided that this alternate path would basically take the path sequence between the two indexes and reverse it. This new path can be anything but we found this to be the one that gave us good results.

To generate the probability, we made our “temperature” a function of time in this case because we have to stop at certain time as per the requirements of this project. To figure out the optimal function we had to mess with different values for temperature. We came to the conclusion that multiplying the time left by 10 would get us good temperature values. The probability function, e to the difference in the current path length and the neighbor length over the temperature, was used to compare to a random probability. The bigger the difference in path lengths, the less likely the worse path would be chosen, but the higher the temperature, the more likely the algorithm is to take risks. This is a key point because this willingness to take risks is what helps the algorithm to not get stuck at local optima. Finally, we change the time left variable to the new time of the program which again, helps stay within the project constraints, as well as decreases the temperature of the system.

Through many test runs of the code, we found that our implementation can find an optimal tour length of 27602 pretty consistently in about 120 seconds.