# Final Project Report

Megan Cromis, Tad Kile, James Mallon

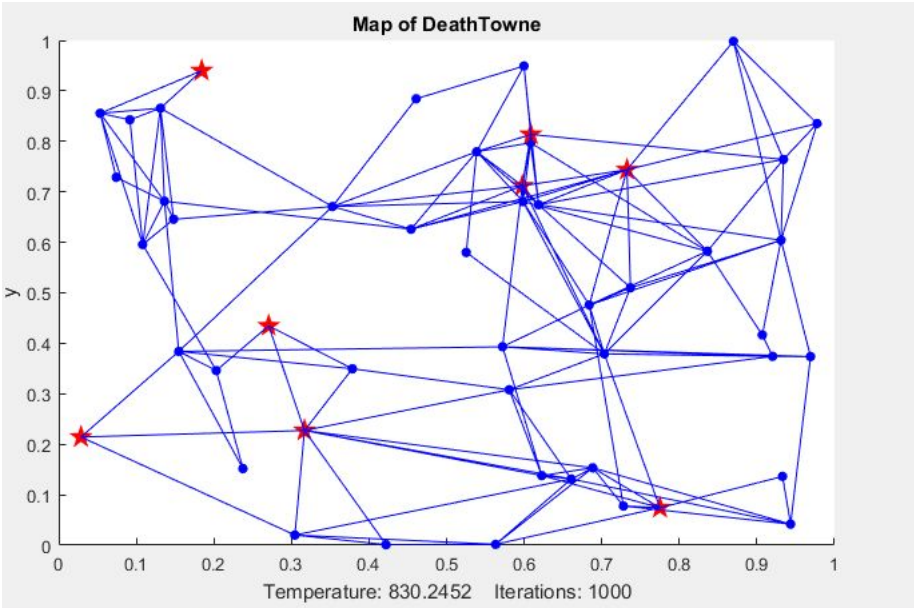
## Introduction

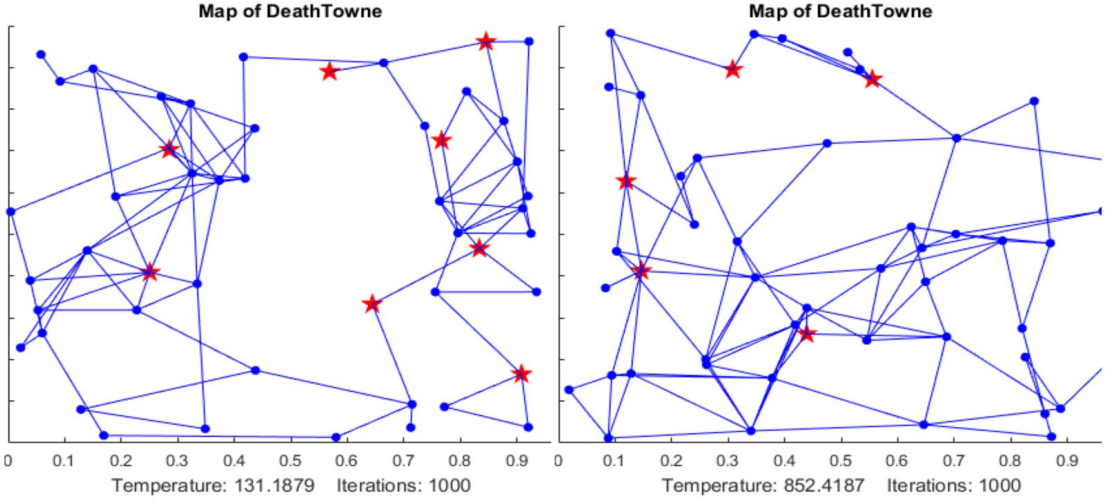
Nurr incorporated, was approached by both Not Black Market inc. and Totally Legit Business LLC to help them with their issue of optimizing their pick-up service. Nurr was tasked with optimizing their organ collection business from a small city with a remarkably high mortality rate, “Deathtown”. Deathtown is a collection of roads, and includes 100 intersections.

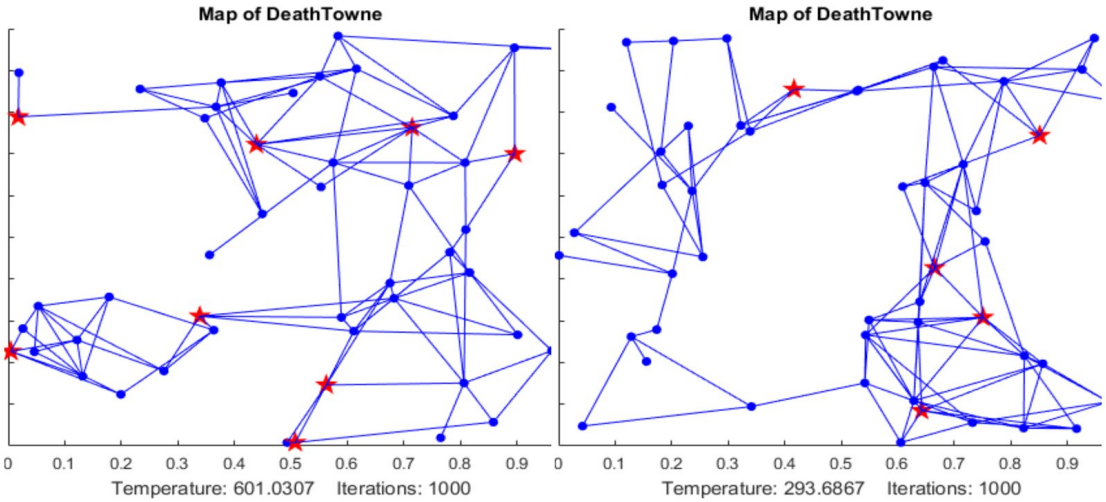
In order to entice the many organ courier services in Deathtown into a bidding war, Nurr incorporated developed an algorithm to take in the intersections where there were organs available for pick up and find the optimal path around the city to pick up each organ.

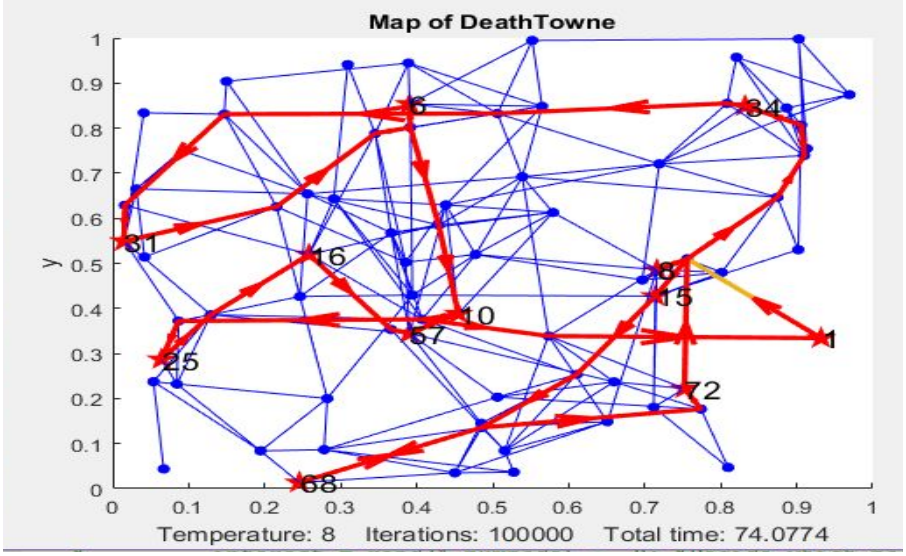
## Methods

Before we could implement our algorithms, Nurr had to develop a map of Deathtowne. Our first idea of how to accomplish this was to plot a 10 x 10 grid to simulate roads, but after a chat with Dr. D, we decided that this was oversimplified. Instead, we implemented a randomized system which took in number of nodes, which we chose to be 50, and placed 50 random intersections on an arbitrary 1 x 1 map. The code then randomly connected nodes to create the road system and chose several random nodes to be points of interest (places for organ pick-up). We then ran that code roughly 100 times, and picked our 5 favorite iterations to be our possible Deathtowne. These 5 options are shown below, with the largest being the chosen map:


Map of DeathTowne
y
Temperature: 830.2452 Iterations: 1000


Map of DeathTowne
Temperature: 131.1879 Iterations: 1000
Map of DeathTowne
Temperature: 852.4187 Iterations: 1000


Map of DeathTowne
Temperature: 601.0307 Iterations: 1000
Map of DeathTowne
Temperature: 293.6867 Iterations: 1000




We chose option 1 because we thought it looked like the best for our simulation. In order to access this map, we saved the workspace as a .mat file, and loaded it into our final code. We later added option 6 because we wanted to see how well our code would work with a slightly more difficult problem.

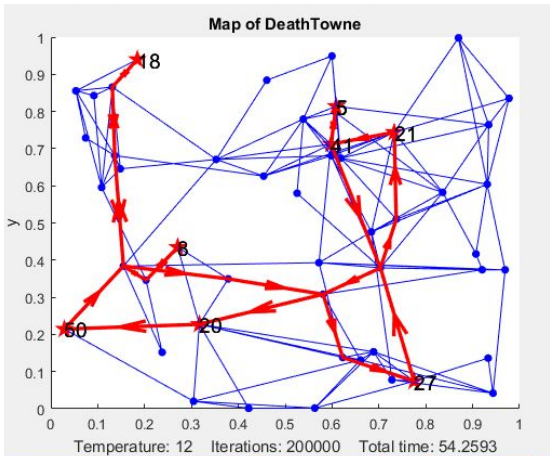
The next section of code was implementing Dijkstra's algorithm to find the optimal path in between each of the points of interest. To do this, the algorithm began with one waypoint (and through multiple iterations, would end up starting with all waypoints), and then look at the distance between it and all connected nodes. These time values are recorded, and the smallest time becomes "visited", and then all connections to it are looked at, and the time to them plus the time to the visited node are added the nodes' "shortest time". Once all nodes have been "visited", the algorithm is ended, and the shortest times to every node from the starting point are found.

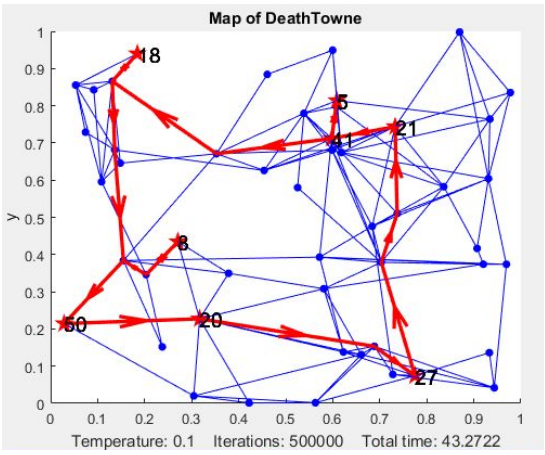
The last section was simulated annealing. Based upon the classic traveling salesman problem, our code takes in the data received from Dijkstra's algorithm, and optimizes the order in which the courier travels. Our first draft of the S.A. process took about 1.75 million iterations, but we eventually got it down to approximately 500,000 by adjusting the starting temperature and increasing the time spent at lower temperatures.

## Results

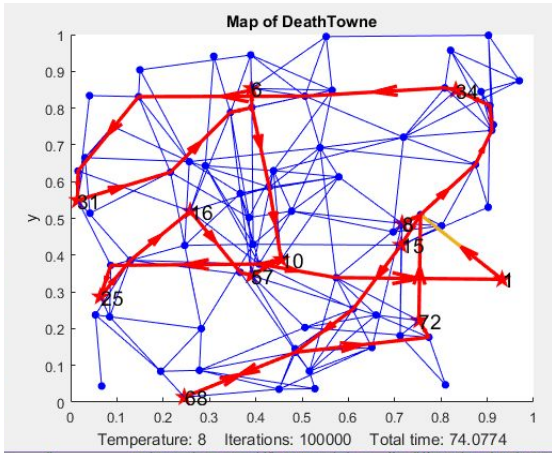
Once we had all three sections working, we tweaked the code slightly to get them to run as a cohesive unit. The results for each scenario are shown below, the end solution on the right.

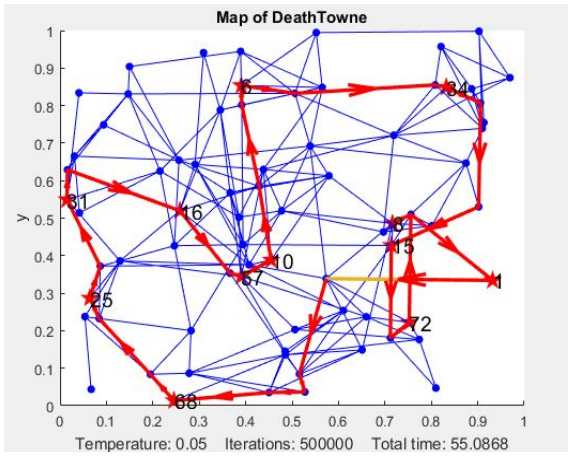
Map 1:


Map of DeathTowne
18
5
21
41
8
50
20
27
Temperature: 12 Iterations: 200000 Total time: 54.2593


Map of DeathTowne
18
5
21
41
8
50
20
27
Temperature: 0.1 Iterations: 500000 Total time: 43.2722

Map 2:


Map of DeathTowne
5
34
31
16
8
15
10
57
1
25
72
68
Temperature: 8 Iterations: 100000 Total time: 74.0774

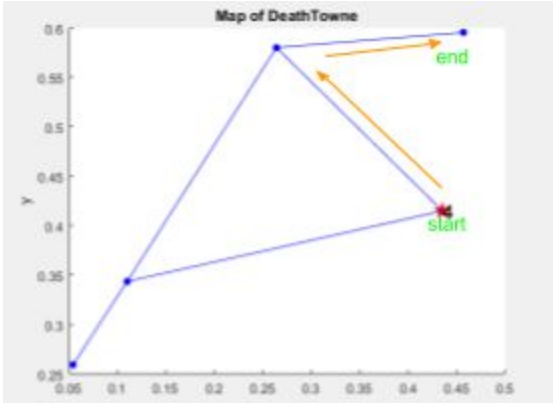

Map of DeathTowne
5
34
31
16
8
15
10
57
1
25
72
68
Temperature: 0.05 Iterations: 500000 Total time: 55.0868

## Validation

No validation was needed for the mapping section, because it would either work or not work.

For Dijkstra's algorithm, we ran a simple map of only 5 nodes (which makes it extremely easy to discern an optimal path with the naked eye), ran the algorithm, and checked to make sure the given answer made sense. The test case is shown below.

We took nodebox matrix, and found the optimal paths from the waypoint to each point, and it matched the obvious solution. Ex to get to the left corner, the nodebox matrix tells us to go up to node 2, then node 1.




```
nodebox =

    3.0000    4.0000    4.0000    4.0000    2.0000
    5.4470    3.9662    3.3641         0    5.3658
    1.0000    1.0000    1.0000    1.0000    1.0000
```

We also wrote some code to check if any nodes only had paths to the point, to make sure that no nodes were left stranded.

Simulated annealing also needed to be tested to ensure its accuracy. To do this, we included a brute forcing section that attempts every possible order of waypoints, and then finds the smallest time from that. This was tested using the 50 node city, as perms(8) was only 5040 different variations, which meant it could be tested quickly.

After running the brute force, and comparing it to our simulated annealing code, it was found that the time and path for both optimal routes were the same, thus validating our simulated annealing code.

```
EDU>> wayorder

wayorder =

     5    21    27    20    50     8    18    41     5

EDU>> totaltime

totaltime =

   43.2722
```

Output from our simulated annealing on the 50 node city.

```
totaltime =

   43.2722

EDU>> allpossible(pathvalue,:)

ans =

     5    41    18     8    50    20    27    21     5
```

Output from our brute force solver on the 50 node city.

## Conclusion

Overall, our code was successful! Each of the three sections worked as planned, and the integration of the sections together was, for the most part, seamless. Not only this, but the *exact* solution was found! However there are several things that we could have improved upon to make our code run more intuitively.

First, our mapping system, while adequate, still has a readability issue after about 100 intersections. Second the simulated annealing has room to be improved, as the optimal iteration and temperature values are dependant on the map present, so code could be added to automatically shift the simulated annealing values.