# Large Scale Taxi Distribution With Simulated Annealing.

Nicholas Rutherford - 11081742

February 9, 2014

## Contents



## 1 Problem Description

The problem we focused on is to deliver a set of packages in a city. We have the following parameters:

**N** The number of packages to be delivered

**K** The number of vehicles used to deliver the packages

**M** The number of nodes in our graph of the city

Each package has a source node, and destination node. Each truck can only move one package at a time, so you can think of them like taxis moving people rather than a delivery truck delivering 100s of packages at one time. Each of the trucks starts and ends at the garage node. The city is constructed as a graph with possible locations being nodes and roads between those locations being weighted edges. The goal is to delivery all of these packages in an efficient manner.

# 2 Solution Description

The main problem can be broken down into smaller problems.

1. Finding the best path from one location to another in the city.
2. Finding the best order to deliver a set of packages that a truck is responsible for delivering.
3. Finding the best set of packages for each truck.

To find the best path from location A to location B is a problem that can easily be solved with shortest path finding algorithms. We chose to use $A^\star$for this problem as it finds the optimal path in reasonable time. Addressing problems 2 and 3 is a bit more tricky. We could through everything into one large $A^\star$ search, however at each step in our search we would need to generate $b^K$ new search nodes where $b$ is the maximum degree of node[1]. This would result in a ludicrous amount of nodes being generated at each step, but it would give you the optimal if it ever managed to halt.

Instead we decided to use simulated annealing. This is where you generate an end state, then find a similar but slightly different end state. Then you test whether the new state is cheaper, if it is you move to this new state and repeat the process. If the new state is more expensive you have a chance at moving to it, but the chance of you moving to a higher cost state decreases as the program runs. Simulated annealing does not run into the state explosion that $A^\star$ experiences, however it is not optimal which means it will return a good result, but not necessarily the best result. Another benefit of simulated annealing is that you can get better results by allowing the program to run longer.

# 3 Implementation Description

## 3.1 Generating the Test City

The first thing we need is a city to deliver packages in. For this we are using the *geographical_threshold_graph* from the NetworkX python package which is described as:

> The geographical threshold graph model places $n$ nodes uniformly at random in a rectangular domain. Each node $u$ is assigned a weight $w_u$. Two nodes $u, v$ are connected with an edge if $w_u + w_v \geq \theta r^\alpha$ where $r$ is the Euclidean distance between $u$ and $v$, and $\theta, \alpha$ are parameters.

This graph generating method tends to generate interesting graphs which means we can easily test our solution on complicated life-like graphs. However this

[1] If you imagine a basic grid most nodes would have $b = 4$.

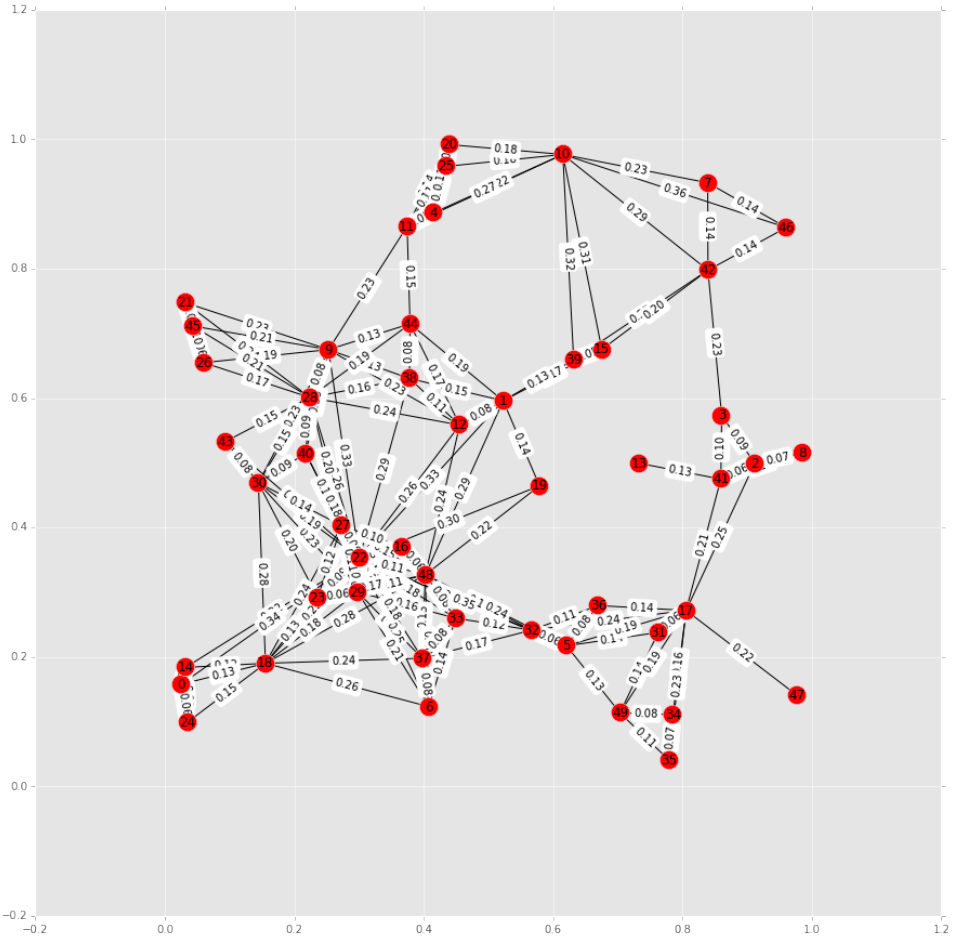

Figure 1: An example of a randomly generated graph with 50 nodes.

graph method can generate unconnected graphs so we simply loop until we find one that is connected[2].To see an example generated by this function see figure 1.

Once we have our city graph we randomly generate $N$ packages with random source and destination nodes. Then we pick a random node as our garage location. Lastly we generate our $K$ trucks and randomly assign our packages to the trucks. This gives our trucks a random ordering of nodes to visit, and an order to visit those nodes. Our example city is now set up and is ready to be solved.

[2]If a graph is unconnected it could result in a package needing to be delivered to a node that is completely isolated

### 3.2 Calculating Costs

We need a way to measure how *"good"* a solution is, so we need to define the cost function. It feels natural to define the cost as simply the sum of distance traveled, however minimizing this cost will result in just sending out one truck to deliver all of the packages. This is because if we send out more then one truck they will have to drive out from, and back to the garage, resulting in extra distance that one truck wouldn't have to travel. With this in mind we define the cost of any given state $S$ as a function of the maximum distance traveled by any truck added to the average cost for each truck:

$$\text{cost}\,(S) = \max\left(\{t_{\text{dist}} | t \in S\}\right) + \frac{1}{N}\sum_{t \in S} t_{\text{dist}}$$

We calculate the distance that each truck travels by using $A^\star$ to find the shortest path between each node in its path, then summing these calculations. For each random state we generate we need to calculate the cost of that state. This is a very expensive operation for large values of $N, K$, and $M$ since we would have to compute so many $A^\star$ searches of the city. We changed this so that for each truck we store the total cost, which means that unless we change the packages it is delivering or the order it is delivering them we don't need to run the repeated $A^\star$ searches on them. This means that independent of $N, K$, and $M$ we only need to recalculate the distance for two trucks per permutation of the state since we are only swapping one package.

### 3.3 The Annealing Processes

Now that we have our cost function and our test city we can finally start to solve the problem using simulated annealing. One key aspect of the annealing processes is the function that will generate a random neighbor of your current state. For this process we define this neighbor generating function as:

1. Select a random truck that has packages, $T_1$.
2. Select a random package from $T_1$, call that package $P_1$.
3. Select another random truck denoted $T_2$. (Note: it is possible that $T_1 = T_2$)
4. Select a random package $P_2$ from $T_2$ and insert $P_1$ in front of it.

Now that everything is in place we can initiate our annealing function which generates random states, moving to the new state if it is cheaper. If the state is more expensive there is a chance that we will move to this new state, but this chance decreases over time.

## 4 Results

Please see table 1 on the next page for the raw results and figure 2 on page 6 for the graphed results.

| KMAX | Cost | Time(s) |
|---|---|---|
| 1 | 5.937538 | 0.000781 |
| 2 | 6.829176 | 0.000845 |
| 4 | 6.627195 | 0.001868 |
| 8 | 6.405918 | 0.003769 |
| 16 | 6.064325 | 0.009314 |
| 32 | 5.385607 | 0.01594 |
| 64 | 5.965227 | 0.03133 |
| 128 | 5.437922 | 0.059354 |
| 256 | 5.767973 | 0.121463 |
| 512 | 5.014664 | 0.240791 |
| 1024 | 5.014664 | 0.474074 |
| 2048 | 5.014664 | 0.987716 |
| 4096 | 5.014664 | 2.008543 |
| 8192 | 5.014664 | 3.911809 |
| 16384 | 5.014664 | 7.6286 |
| 32768 | 5.014664 | 16.223122 |
| 65536 | 5.014664 | 35.080539 |
| 131072 | 5.014664 | 65.147481 |
| 262144 | 5.014664 | 121.595374 |
| 524288 | 5.014664 | 250.779898 |

(a) N=5, K=2, M=15

| KMAX | Cost | Time(s) |
|---|---|---|
| 1 | 14.477362 | 0.001536 |
| 2 | 14.477362 | 0.001902 |
| 4 | 11.744479 | 0.005885 |
| 8 | 14.477362 | 0.011669 |
| 16 | 10.662326 | 0.016833 |
| 32 | 10.895049 | 0.039689 |
| 64 | 10.461457 | 0.077516 |
| 128 | 10.924679 | 0.154631 |
| 256 | 9.367373 | 0.282156 |
| 512 | 9.506347 | 0.660316 |
| 1024 | 8.458292 | 1.216206 |
| 2048 | 8.283734 | 2.427771 |
| 4096 | 8.329476 | 4.907448 |
| 8192 | 8.461348 | 9.784679 |
| 16384 | 8.12729 | 19.968251 |
| 32768 | 8.137235 | 49.458115 |
| 65536 | 7.840349 | 82.088683 |
| 131072 | 8.023004 | 159.398873 |
| 262144 | 7.840349 | 340.257495 |
| 524288 | 8.023004 | 692.649834 |

(b) N=10, K=3, M=30

| KMAX | Cost | Time(s) |
|---|---|---|
| 1 | 18.077664 | 0.01247 |
| 2 | 18.003993 | 0.017245 |
| 4 | 18.017484 | 0.021412 |
| 8 | 17.927887 | 0.052412 |
| 16 | 16.75227 | 0.115286 |
| 32 | 17.228147 | 0.22375 |
| 64 | 16.921698 | 0.482851 |
| 128 | 16.248993 | 0.883003 |
| 256 | 17.470642 | 1.884079 |
| 512 | 16.319192 | 3.788475 |
| 1024 | 15.728387 | 7.565779 |
| 2048 | 14.129385 | 14.732667 |
| 4096 | 14.48704 | 30.578264 |
| 8192 | 14.321801 | 59.678774 |
| 16384 | 13.766023 | 125.15633 |
| 32768 | 13.330526 | 238.39519 |
| 65536 | 13.306941 | 450.856941 |
| 131072 | 12.962471 | 900.656528 |
| 262144 | 12.234322 | 1834.385617 |
| 524288 | 12.574684 | 3961.158168 |

(c) N=30, K=5, M=150

| KMAX | Cost | Time (s) |
|---|---|---|
| 1 | 30.205686 | 0.496578 |
| 2 | 30.205686 | 0.933685 |
| 4 | 30.205686 | 1.983287 |
| 8 | 30.205686 | 3.522272 |
| 16 | 29.208911 | 7.618322 |
| 32 | 29.208911 | 15.242345 |
| 64 | 28.78838 | 29.166524 |
| 128 | 29.138745 | 56.460736 |
| 256 | 29.720409 | 112.18948 |
| 512 | 28.126214 | 222.930921 |
| 1024 | 27.616447 | 436.264031 |
| 2048 | 28.724179 | 929.159583 |
| 4096 | 25.419794 | 1827.639881 |
| 8192 | 23.953923 | 3634.95182 |
| 16384 | 22.5278 | 7413.342096 |
| | | |
| | | |
| | | |
| | | |
| | | |

(d) N=300, K=15, M=5000



Table 1: Raw Results

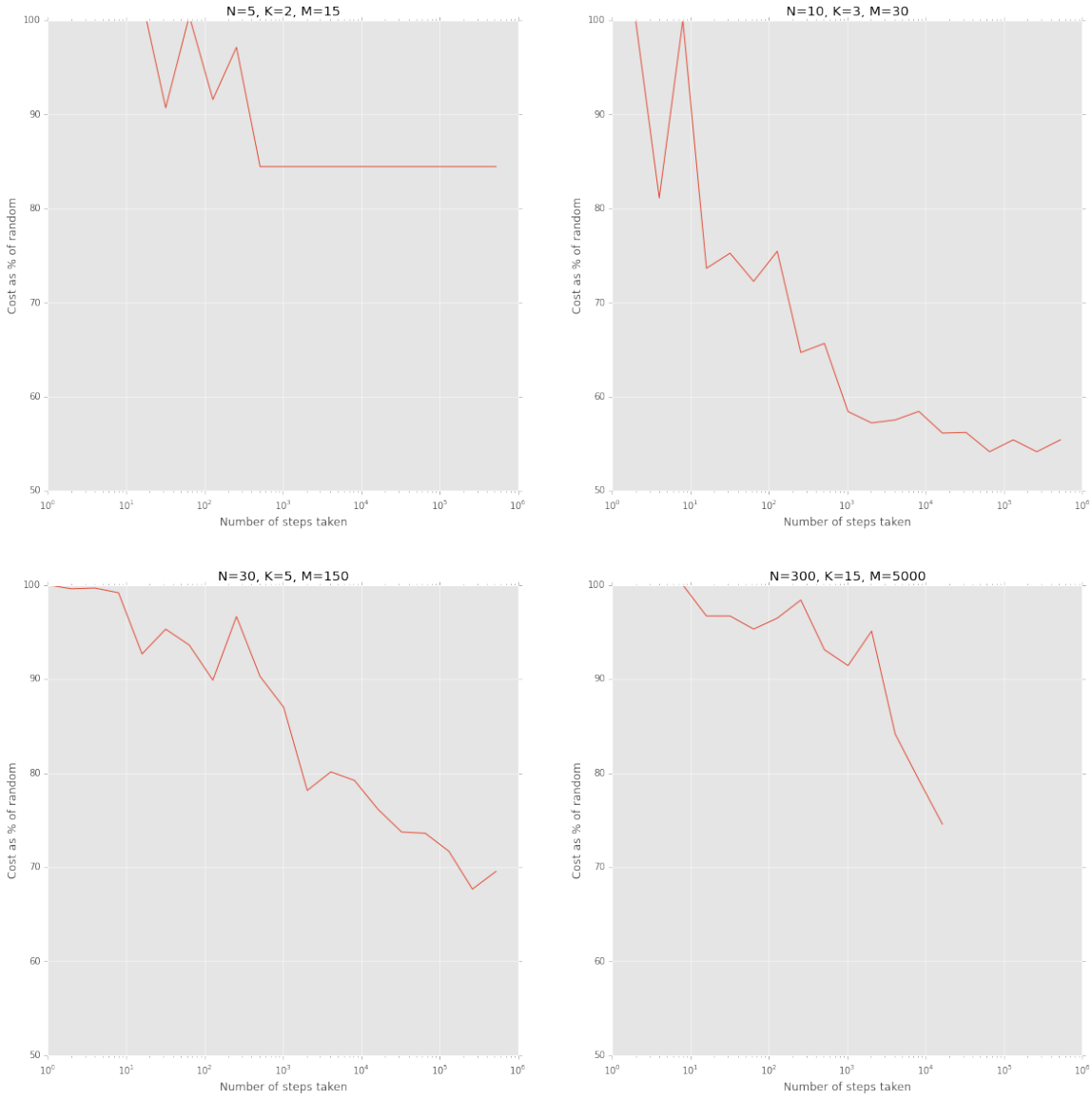


Figure 2: The cost as a percentage of the random initial state compared to the log of the number of steps taken.

# 5 Conclusions

From the results we can see that it does reduce the cost over time, which means that it at least works. However the cost can be prohibitive. For the real world scenario we selected 300 packages, 15 trucks and a city with 5000 nodes. In order to get a solution that is $\approx 75\%$ the cost of the random initialization it took two hours of run time. It looks like the cost wasn't about to flatten out so better results would be possible if you were willing to wait longer, however it is worth reiterating that these plots are log plots so if you wanted to continue that downward trend you'll be waiting for 4 hours or longer. However the majority of the memory cost is just storing the city graph in memory, which means if you had a faster computer or were willing to wait longer you could make $N, K,$ and $M$ much larger.

# 6 Implementation