# ROB 537
# Learning Based Control
# HW#2 Search and Optimization
# TA: Shauharda Khadka

Eric Klinkhammer

October 16, 2017

## 1 Common Experimental Setup

### 1.1 Solution

The solution (or population of solutions) for my search algorithms was always a list of integers, representing an ordering of the cities. A solution is scored by the total Euclidean distance between all of the cities going in the order specified by the solution. A solution is considered strictly better if it has a lower score.

### 1.2 Mutation

The mutation operator used by all three of my algorithms was a random pairwise swap between two elements in the solution. There was no crossover.

### 1.3 Timing

The time reported is the total real time (using the UNIX time command) divided by the number of statistical runs. It is approximately the time of one iteration.

Code is not efficient. Lists are used in place of arrays, when the mutation operator is a swap of elements (so mutation is O(solution size), instead of O(1)). Does not utilize distance matrix as efficiently as it could.

### 1.4 Statistical Runs

All experiments are run with 100 statistical runs. The average run time, mean, best score, standard dev, and confidence internal are reported for all algorithms.

### 1.5 Simulated Annealing

My implementation of simulated annealing worked as follows. For each of the 1,000,000 generations, a single potential solution was generated. If that solution was better, it was chosen with probability 1. If that solution was not better, it was chosen with probability $e^{-\Delta E/T}$ where $\Delta E$ is the difference between the two solution's scores and T was $T_0 * 0.95^k$, with $T_0$ being the initial temperature (100) and $k$ being the current generation.

### 1.6 Evolutionary Algorithm

I designed an evolutionary algorithm that maintained a population of 50 solutions, used the same mutation operator as detailed above, selected with replacement with rank probability, and used an elitist generation strategy. The population evolved for 10,000 generations.

By selecting with rank probability, I mean that the probability a solution is selected is proportional to its relative fitness. For the individual with rank n (out of N), it would be selected with

probability $\frac{N-n}{\sum_{i=1}^{N} i}$. Selections are with replacement from a pool that consists of both the original solutions and the mutated versions (each original solution contributing exactly one mutated solution).

### 1.7 Own Algorithm Choice: $\epsilon$-greedy

For each of 10,000 epochs, 50 potential solutions were generated. If a generated solution is better, it is chosen. However, even if it is not better, the best of the generated solutions can still be chosen with probability $\epsilon$. All experiments were run with $\epsilon = 0.025$

## 2 15 City TSP

| Algorithm | Run Time | Mean | Min | Std Dev | Lower Bound 95% | Upper Bound 95% |
|---|---|---|---|---|---|---|
| SE | 2.95s | 5701 | 5113 | 404 | 5622 | 5780 |
| EA | 0.88s | 5506 | 5113 | 422 | 5423 | 5589 |
| $\epsilon$-Greedy | 0.89s | 5481 | 5113 | 394 | 5404 | 5559 |

Table 1: Algorithm performance on 15 city TSP.

## 3 25 City TSP

| Algorithm | Run Time | Mean | Min | Std Dev | Lower Bound 95% | Upper Bound 95% |
|---|---|---|---|---|---|---|
| SE | 3.48s | 6970 | 6006 | 510 | 6870 | 7070 |
| EA | 1.14s | 6524 | 5615 | 438 | 6438 | 6610 |
| $\epsilon$-Greedy | 1.19s | 6544 | 5735 | 394 | 6467 | 6621 |

Table 2: Algorithm performance on 25 city TSP.

## 4 100 City TSP

| Algorithm | Run Time | Mean | Min | Std Dev | Lower Bound 95% | Upper Bound 95% |
|---|---|---|---|---|---|---|
| SE | 7.83s | 20304 | 16794 | 1528 | 20004 | 20603 |
| EA | 2.89s | 16688 | 14085 | 1161 | 16460 | 16915 |
| $\epsilon$-Greedy | 2.89s | 16749 | 14215 | 1242 | 16505 | 16992 |

Table 3: Algorithm performance on 100 city TSP.

In general, the EA performed better than the $\epsilon$-greedy in terms of the absolute minimum reached, which performed better than simulated annealing. The EA most likely performed better than the single solution approaches because, as a population algorithm, it was able to avoid local minimum better and explore a larger portion of the search space. Similarly, the $\epsilon$-greedy approach considered a larger set of potential solutions at any given time (and selected among them). Additionally, unlike simulated annealing, the exploration percentage remained constant throughout the learning. However, the total number of unique solutions considered by the two algorithms is relatively similar, hence their similar means. They have near identical run times because they consider identical number of states (not necessarily unique).

That simulated annealing (as implemented) is particularly vulnerable to local minimum is further supported by the deteriorating relative performance as the number of cities (and potential local minimum) increases. The higher number of states being considered by simulated annealing (and thus the high run time) don't translate into higher performance because these considered states are likely to contain more duplicates.

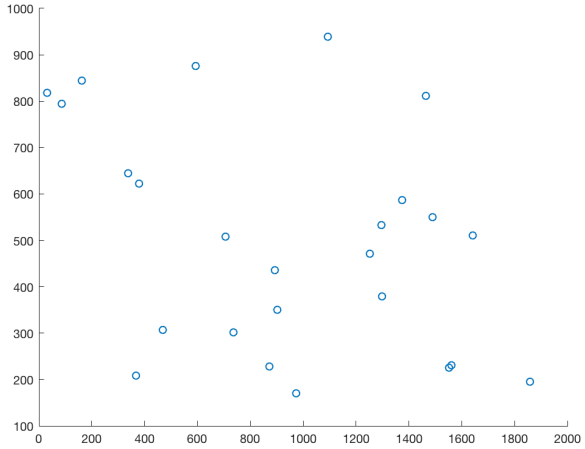


Figure 1: 25 cities placement (essentially random)

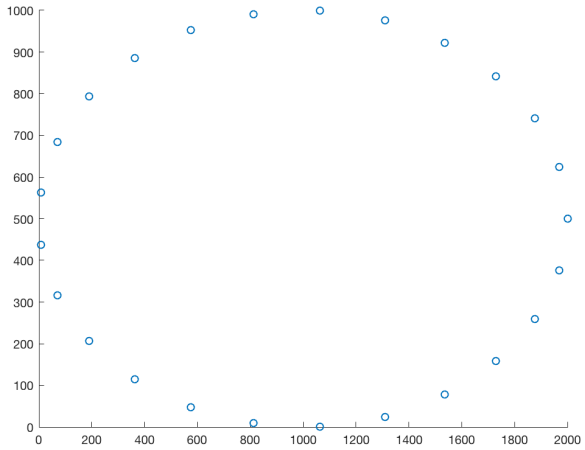


Figure 2: 25 cities A placement. Structured in a circle.

# 5 25 City - Map A TSP

| Algorithm | Run Time | Mean | Min | Std Dev | Lower Bound 95% | Upper Bound 95% |
|---|---|---|---|---|---|---|
| SE | 3.68s | 4831 | 4831 | 0 | 4831 | 4831 |
| EA | 1.16s | 4831 | 4831 | 0 | 4831 | 4831 |
| $\epsilon$-Greedy | 1.16s | 4870 | 4831 | 130 | 4844 | 4895 |

Table 4: Algorithm performance on 25 city TSP with cities arranged in a circle.

The alternate map's solution was an ordered list. For simulated annealing and the greedy approach, the algorithm starts with an ordered list as the solution, so it just happens to start with the right answer. However, the EA starts with a set of shuffled lists. It is able to consistently find the correct answer because of the geometry of the cities.

Because the cities are in a circle, there are no local minimum. Every swap that improves the solution beings the solution closer to the global optimum. Additionally, a swap only improves the solution if it reduces (or keeps constant) the total number of out of order pairs.

As an example, consider the subset of (5,3,4). There are two out of order pairs (5 and 3, 5 and 4). Swapping 3 and 4 increases the number of out of order pairs, and is a strictly worse solution.

Swapping 5 with 3 will result in only one out of order pair (5 and 4). You can confirm geometrically for all circular points that this is preferable. Alternatively, you can swap 5 with 4. This may or may not improve the solution, but doesn't change the number of swaps till the final solution is found.

These genetic algorithms are effectively implementing bubble sort. Bubble sort is an $O(n^2)$ algorithm, and in fact the total number of swaps can be bounded as less than 625. For a given solution, there are only 300 possible choices (25 choose 2) for a swap. This means the expected number of potential solutions needed is less than 62,500. In fact, it is substantially less, because for each required swap, the probability that the generated swap is valid increases. Assuming uniform swap selection, the expected number of selections is $\sum_i^{300} \frac{300}{i}$.

Because the genetic algorithms generate that many solutions, they converge to the solution (as expected).

# 6 Solution Scale

| Algorithm | 15 States Searched | 25 States Searched | 100 States Searched |
|---|---|---|---|
| SE | 1,000,000 | 1,000,000 | 1,000,000 |
| EA | 500,000 | 500,000 | 500,000 |
| $\epsilon$-Greedy | 500,000 | 500,000 | 500,000 |
| Exhaustive | $10^{12}$ | $10^{25}$ | $10^{157}$ |

Table 5: Solution space searched as compared to number of possible solutions.

The evolutionary search approaches explore only a fraction of the search space (the total number of states considered is most likely less than the numbers listed above, as duplicate states can be considered).

The number of solutions to the traveling salesman problem is the number of ways to permute the solution, which is on the order of the factorial of the number of cities. It is strictly less than that number because the solutions are equal under rotation, so the number of possible solutions, for n cities, should be close to n!/n, or (n-1)!