COMP 3201 Final Project Report

Daniel Power, Rebekah Pynn, Nina John, Zachary Northcott

Github Link: https://github.com/ZacharyMN709/COMP-3201---TSP-Evolutionary-Algorithm

Introduction

The Travelling Salesman Problem (TSP)

The TSP is a problem where, given a graph with weighted edges, the goal is to find the shortest hamiltonian circuit through the graph. Note that the TSP is an NP hard problem, meaning there is no simple way to programmatically determine the optimal solution without the aid of a brute force solution.

Our Task and Problem Representation

Our task was to use an evolutionary algorithm (EA) to find the shortest route that starts at a city, passes through each city, and returns to the initial city. An EA is an effective way to approximate, or even find, the optimal path. Through crossover, it combines sections of good paths together, and through mutation, remove deficiencies, like loops, from the route. Additionally, it can attempt to find multiple solutions simultaneously, and then select the best of these discovered solutions.

For our EA, each city can be considered a vertex of a graph, where an edge exists between every two cities, and the edge weights are the Euclidean distances between the cities. Our individuals are a list of integers, which represents the order the vertices are visited, where each integer represents a city. Additionally, since each city is only visited once (as we implicitly return from the final city to the first city), our individual can be represented as a permutation of integers. Our fitness evaluation function is the sum of the distances between cities. Due to our representation, we were able to optimize the algorithm by pre-calculating all of the city distances, and keeping them in a table to avoid redundant computation, as the integer representing each city can be used to index its location and distance between other cities.

Going Further

Rather than simply stop at our given task, we made our goal to create a program which could easily redefine the EA methods to be used, instead of continuously optimising a single algorithm. This allows for flexibility in what can be tested. Taking advantage of this flexibility, we implemented multiple heuristics in an attempt to optimize our solutions, including Christofides' heuristic as well as a clustering heuristic. In addition to designing the code to be extensible, we also support the representation of individuals using lists, numpy arrays and C arrays, making our code flexible and easy to integrate with other projects.

Methods

General Architecture and Code Modularity

The beauty of our design lies in that our evolutionary algorithm (EA) is broken down into 6 steps; initialise a population, select parents, generate offspring, mutate offspring, select survivors, and manage the population. The 'manage the population' step is an addition of our own. Each of these

steps is run by a single function call, and each of those function calls gets their information from a class designed to handle that step exclusively. Each class contains all of the functions related to that step, along with a reference to all of the parameters for the EA. This makes modifying the EA as simple as changing the parameters and function pointers, something which can be set-up to change mid-run based on whatever criteria the user wants.

Defining a population and calculating fitness are contingent on the problem being solved. However, selecting parents, generating offspring, applying mutations, selecting survivors, and managing the population are almost entirely independent of the representation of the data, and so the same code can be reused for each EA instance. This means to solve a new problem, only the population generation and fitness evaluation need to be redefined. For example, by writing new classes which define individuals and fitness for the 8-queens problem, and loading those into the EA shell, the EA would solve that problem instead. Furthermore, since the problem is defined through classes, changing the problem to be solved can be as simple as changing which modules are imported.

Testing and the EA Factory

Another benefit of modularity is the ease of running tests. To facilitate tests, we wrote a class which can generate multiple instances of the EA, and set each to use a different suite of methods. Since the methods are selected using integers, automating this through loops was incredibly simple. Another benefit of this class, is that the 'factory' can take in all the information related to the problem itself, then give the EAs references to it, rather than copies. This is *particularly* useful since each EA doesn't end up with identical data, which for some problems - like Canada - can be several hundred megabytes, or more, when in memory. When large enough, that overhead can ruin any attempt at parallelization of different instances of the algorithm.

Population Management to Combat Staling

`Annealing`

One of the advanced techniques that was implemented in our program was the simulated annealing technique, which is based on annealing in metallurgy. The appeal of this technique is that, in order to avoid being trapped in a local optima, the algorithm temporarily accepts a worse individual. By taking a less optimal solution temporarily, we hope the new individual is worse enough than the current, it is more likely to escape the local optima. The more generations the algorithm has progressed through, the less likely it is that optimal answers will be replaced with new individuals. The replacement rate can be modified through variables.

`Entropy`

The entropic stabilization technique's goal is to prevent the population from becoming too homogeneous. This, again, is to increase diversity in hopes of finding the true optimum rather than local ones. The more individuals that share the current best fitness in the population, the more likely it is that some of the best individuals will be randomly replaced with new individuals. It only begins to replace individuals if the number of individuals with the best fitness has surpassed a certain threshold variable.

Ouroboros

As the algorithm grows and stabilizes, the ouroboros technique works by in a sense, 'eating itself'. It does this is by limiting the percent of individuals who are able to share the maximum fitness. It can been seen as a less stochastic version of the entropy method described above. Again, a population threshold is used to determine if the number of individuals who share the same best fitness is beyond a certain point so that action can be taken.

Engineering

The engineering technique is inspired by genetic engineering. When the population becomes too self-similar, this method modifies only a single fittest individual. A random number is chosen and each allele is swapped with another allele that random number ahead of it. With each swap the fitness is checked to see if it improves. If it does not, then it reverts the change and moves to the next allele. This process is incredibly aggressive, and was designed to try and 'unwind' some of the loops that existed in paths we plotted.

Results

Preamble

Like the TSP itself, our algorithm can be arranged in many different ways. Rather than comparing every possible combination with every other combination, we chose a set of methods that we thought were responsible for the majority of the variance in the results of the different combinations: the initialization methods, population management methods, and mutation methods. From there, we selected one of these particular steps of the algorithm, and tested each method we had implemented for said step to compare the effects of each directly. This has two benefits: comparing all the combinations of all the methods would be confusing and difficult to gauge, and given each run takes a substantial amount of time, we're able to collect data with less variance by running the same tests more often. The data from running tests has produced many graphs of large size, and as such we've placed them in the appendix, rather than in the write-up.

Changes in Initialization

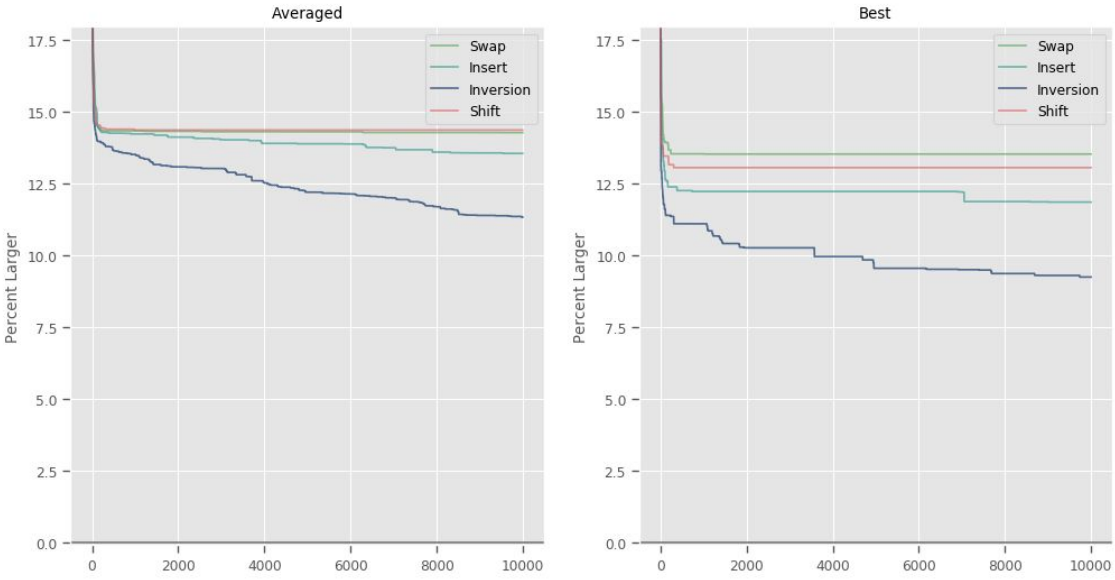
We expected that changing the population initialization through the use of heuristics would drastically change the EA, and we seem to be justified in thinking so. While the effect on smaller datasets is minimal, if not altogether negative, the effect on larger datasets is a massive improvement. We discuss the heuristics in detail, with graphs of data, further below.

Changes in Mutation Method

The inversion method tended to perform better than the swap, insert, and scramble mutations and generally was the best mutation method. In inversion, we randomly select two positions in the chromosome and reverse the order of everything between those positions. By doing this, we are in essence, breaking the chromosome in only two places, as inverting the center preserves the path, except for two new links made at the broken end. Since swap, insert, and scramble work by making larger changes to the order that the alleles occur in, this often causes an large number of

links to be broken. Since this problem is highly sensitive adjacency changes, this is a likely factor for why the inversion method performed at a higher level than the other methods.

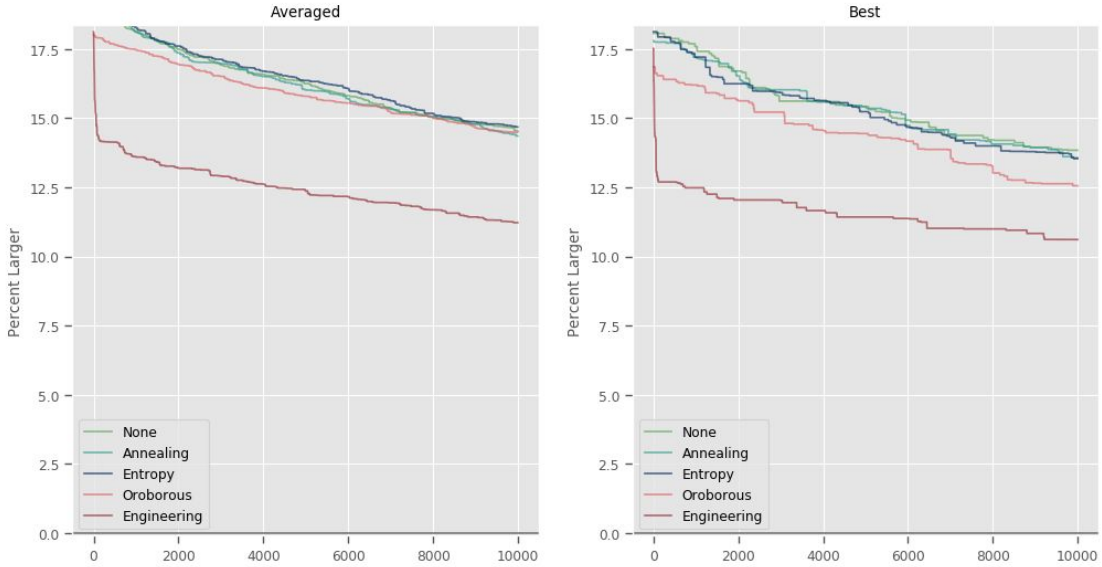
Figure 1: Mutation Comparisons - Uruguay Dataset



Changes in Population Management

We implemented various population management methods in an effort to combat population converging at a local optima, and being unable to escape it. The various techniques that we used for this were the annealing, entropy, orborous, and engineering techniques as previously mentioned. For the most part, changes between techniques were fairly minimal, especially for the Sahara Dataset. However, the engineering method was the highest performing for both the Uruguay and Canada dataset. On an average run, it performed 3 to 5 percent better than the other methods, and showed rapid improvements in the first hundred generations or so.

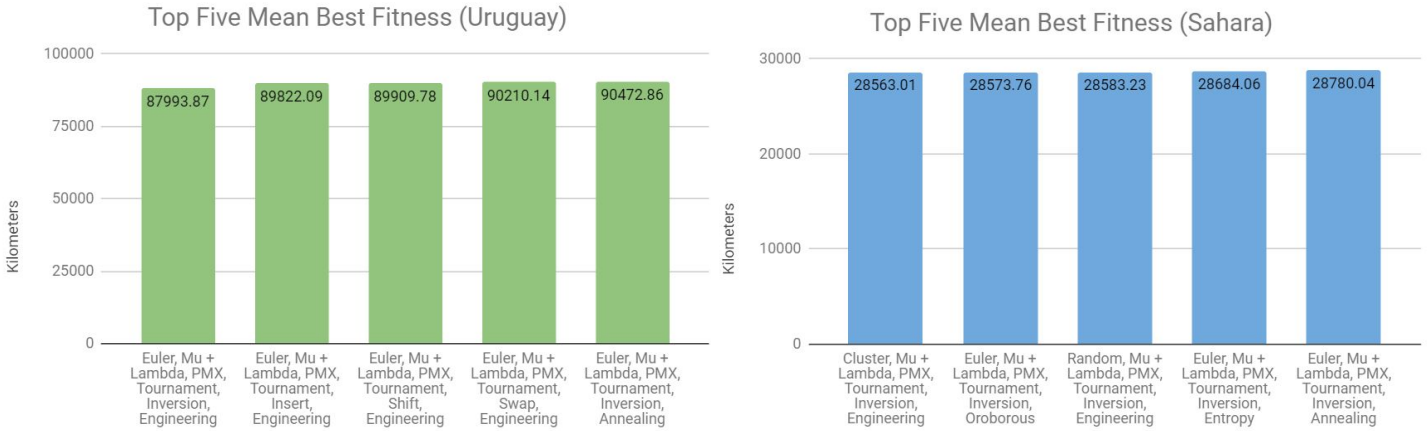
Figure 2: Population Management Comparisons - Uruguay Dataset



Considerations on Best Overall Combination

In order to decide which combination of methods worked best, we used the measurement mean best fitness (MBF). Initially we also had two other criteria, success rate (SR) - defined as a solution being within 10% of the optimum - and average evaluations to a solution (AES). However, for the Sahara dataset almost all runs were within 10%, while for the Uruguay dataset only a single SR was non-zero. Furthermore, the Uruguay dataset never converged, the Sahara dataset converged only 2 out of 40 total times, rendering AES meaningless. It is worth noting, however, that the EA came consistently close to meeting the success criterion for the Uruguay dataset, with several combinations producing a MBF within 15% of the optimum.

Figure 3: Best Mean Best Fitnesses for Uruguay and Sahara



After analyzing the data, it appears that the best overall combination depends on the size of the data. For the Sahara dataset, using the cluster initialization allows the algorithm to more easily escape local optima, and this paired with selection mechanisms Mu + Lambda and Tournament, mutation methods PMX and Insert, and population management method Engineering, gave the best mean fitness of 28,563 km, a mere 962 km longer than the optimum. For the Uruguay dataset, the Euler initialization narrowed the search space extremely quickly - starting at fitnesses other methods fail to reach - allowing it to make better use of the set 10,000 generation computation limit. Euler initialization, Mu + Lambda and Tournament selection, PMX and Inversion mutation, and Engineering population management gave the best mean fitness of 87,994 km, which is 8,860 km over the optimum. While these combinations specifically work best for the Sahara and Uruguay datasets respectively, it is worth noting that the Euler, Mu + Lambda, PMX, Tournament, Inversion, and Annealing combination gave high MBF's with both datasets, indicating it consistently performs well even as the problems scale up. Data from other combinations not mentioned, as well as statistics regarding MBF and SR, can be found in the appendix.

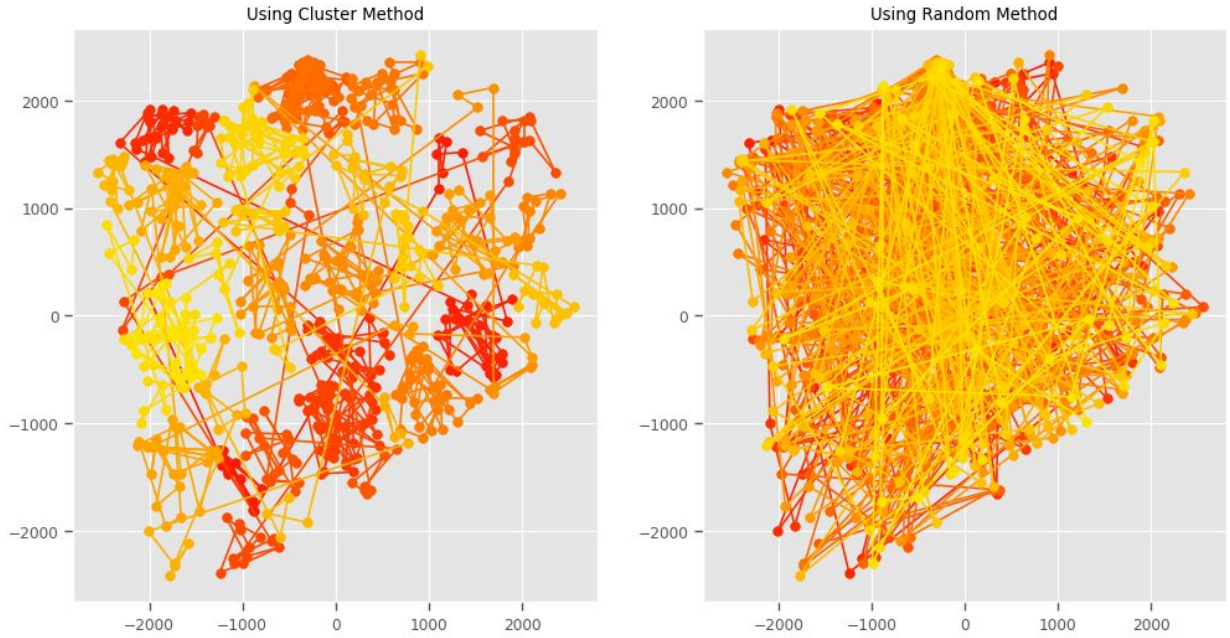
Discussion

Deficiencies of Random Initialization

While randomness helps ensure that the individuals don't converge into a local optimum, simple

random initialization is lacklustre. When we first started the problem we plotted the cities, and a random path connecting them. We quickly realized how poor of a start this was for our algorithm. As reference, using the Uruguay dataset, it typically takes upwards of 7500 generations of evolution for a randomly created individual to approach the fitness level of an individual made by the cluster heuristic *(see Figure 3, below)*. Below is a graph comparing a random individual to an individual created through our clustering heuristic (explained further below) showing the immediate improvement:

Figure 4: Graph of Uruguay Random Individual and Uruguay Cluster Individual



One thing we did notice however, is that for small datasets *(see Appendix - Graph 1)* the heuristic isn't as beneficial. However, the larger the tour, the greater the improvement the heuristic seems to make *(see Appendix - Graph 4)*.

Grid and Cluster Heuristic

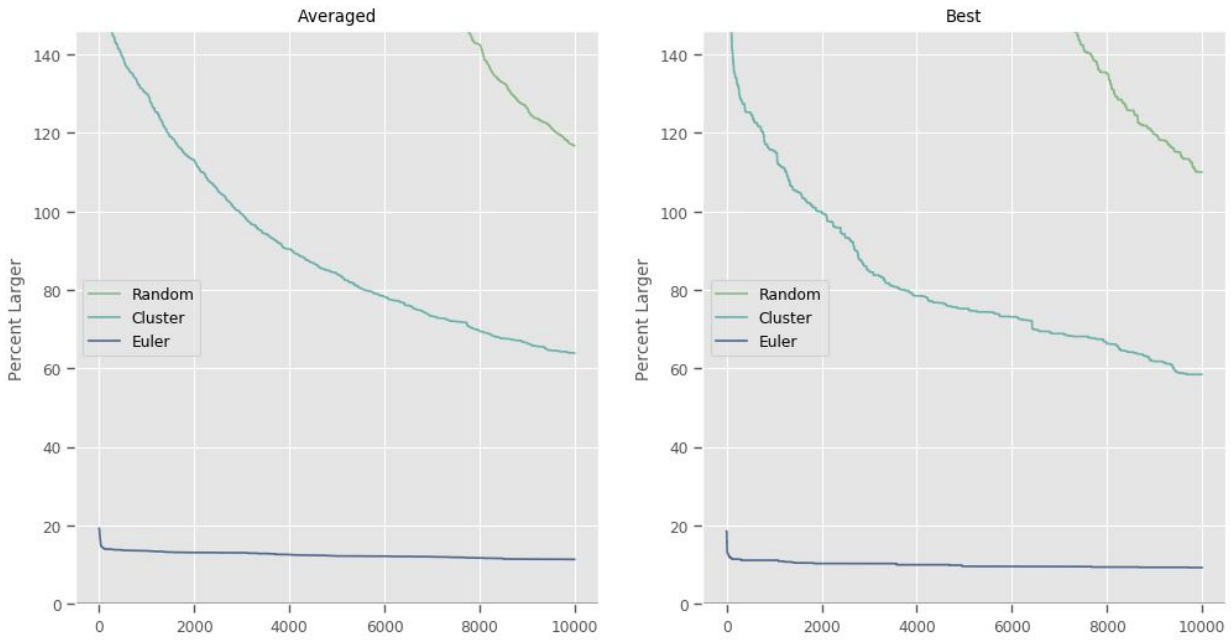
We had two similar ideas for simple initialization heuristics. The first, was to choose a random city, then cluster close cities together, choosing new cities until all the cities are added to clusters. To make an individual city, clusters are shuffled and then appended together. To further help diversity, we also shuffle the order of the clusters. One added benefit of this particular implementation, is that the clustering is only done once. Each new individual is essentially a formatted copy of the raw cluster data, meaning no redundant computation.

The other heuristic we considered, was to split the map into grids and add cities similarly to clustering, but order the grids by following the path of Hilbert's space-filling curve. The benefit of this heuristic over clustering, was to prevent two clusters on opposite sides of the map connecting. Unfortunately this heuristic was never implemented.

Christofides' Heuristic (Euler)

The Euler initialization method was based on Christofides' heuristic, which is currently the best known approximation algorithm for brute force computing solutions to the TSP. Because it is a '3/2 approximation algorithm', it never computes a solution 50% longer than the optimum. However, once implemented, the algorithm generally was within 20% of the optimum, giving us very good initial individuals to seed a population. The general idea behind the algorithm is to create the shortest possible Hamiltonian cycle, which as mentioned previously is itself an NP hard problem. We work around this by first creating a minimum spanning tree using the cities and distances between them, then perfect matching the odd vertices in the tree to create an an Euler cycle, which is then directly converted to a Hamiltonian cycle by deleting repeated vertices.

Figure 5: The Average and Best Fitness per Generation - Uruguay Dataset



Calculating the minimum spanning tree (MST) was the most time consuming process during initialization, and was the part of the algorithm we could improve computation speed for. When creating a new set of individuals, since the MST, and consequently all of its odd edges, are the same for each instance of a problem, we loaded the saved MST from a file, rather than computing it. Our EA also has pre-calculated distances between cities, so the Euler method simply accesses this data instead of computing the distances, further reducing redundant computation.

Despite the MST always being the same, the algorithm is actually stochastic. This is because the odd vertices are matched randomly, and each matched edge is randomly put back into the tree, allowing us to compute several unique individuals from different local optimas. This stochasticity, however, is also the heuristic's downfall. Because of the random perfect matching, it often creates sub-optimal routes by connecting vertices that are far from each other. While the heuristic supplies good individuals, the EA locates these poorly matched vertices and creates better routes, typically bringing the solution to within 10% of the optimum for the Sahara dataset, and 15% for the Uruguay dataset.

The Christofide heuristic also performs much worse on the Sahara dataset than the cluster heuristic or completely random. This is likely because it hits a difficult to escape local optimum. As can be seen below, over the 40 runs, the other initializations not only did better on average, but also had at least one run where it converged to the global optimum. However, on larger data sets the heuristic always out-performed the cluster and random initializations.

Figure 6: Graph of Uruguay Christofide Individual and Uruguay Cluster Individual

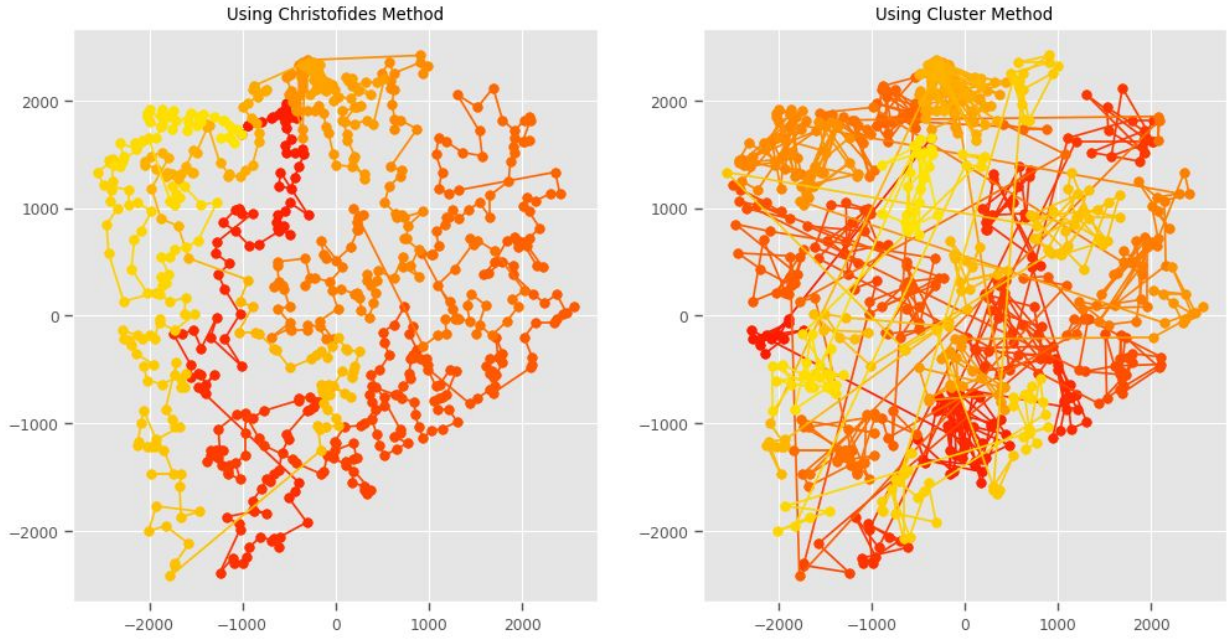
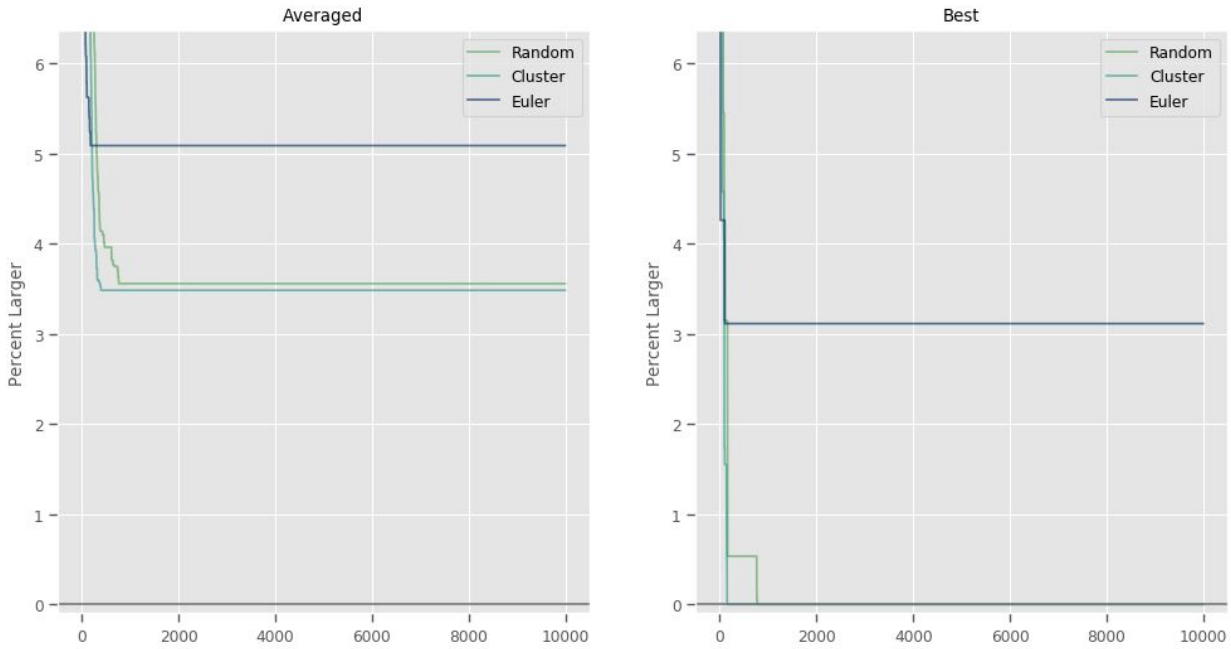


Figure 7: The Average and Best Fitness per Generation - Sahara Dataset



Population Management Methods

As mentioned previously, we decided to add an extra step to our EA: population management. For EA combinations with an aggressive selection mechanism, or an initially fit population, the algorithm has a tendency to get stuck in local optima in the fitness landscape, and converge before finding the optimal solution. To combat this "staling" of the population, we used several methods: Annealing, Entropy, Ouroboros, and finally Engineering. Each attempts to inject randomness into the population to escape these local optima, as well as prevent the algorithm from converging too early by monitoring the percentage of highly fit individuals. We noticed a difference particularly with Engineering on larger datasets, over a period of 5 runs it typically came within 11.5% of the optimum, approximately 3.25% closer than other population management methods, on average.

Storing Run Results

After finishing the EA shell, we added in an object to save the run statistics, and used the 'pickle' module to save the objects. This was a short term solution that allowed runs with the same methods and generation limit to be easily merged. Another benefit was that the object could be expanded with functions to automatically calculate the stats, meaning little code had to be written to do further analysis. Unfortunately, this storage method comes with a limitation - it cannot save objects of a large size.

Due to this limitation, we have started moving towards implementing databases to store runs. While we can store runs in databases at present, the implementation is new, and not well tested, so undefined behaviour may still occur when running the program. As well, the databases are automatically generated, and could benefit from further planning and design, so that results are better organized, and statistics are easier to compute.

Numba and Other Potential Improvements

Numba is a 'Just in Time' compiler for Python and the numpy library which greatly increases the speed of the code by converting it into machine language and caching it for future use. This is something we would like to implement in the future.

Multithreading would be a welcome addition to our program. Originally, we did have some multithreading capabilities, but they caused more complications than they added benefit. When adding multithreading inside the main code to run the EA, the overhead of making the new threads ended up giving a larger net runtime. We think that through more careful memory allocation (keeping empty arrays to add offspring into, rather than re-allocation memory at call), we could improve the run-time, but it would require testing, and likely, a significant overhaul of the code. However, the more successful attempt at multithreading was to run different combination of methods in parallel. The code allowed to run all the tests for method comparisons simultaneously, but due to differences in operating systems, this caused a series of issues - and after fixing one, another always sprung from it. We unfortunately had to remove the code for this, but it is something that we would like to look into in the future.

The recent addition of using databases for storage and JSON objects for initialising runtime variables made us consider another alternative to multithreading. While unfortunately we didn't have time for it, we had hoped to create a bash (or command line) script which would create multiple instances of the program, each initialized with different variables. By doing this, we could let the OS take care of the multithreading naturally, and database locking would automatically ensure that the run data doesn't get trampled, or trample existing data.

Final Remarks

Heuristics Help

Thanks to the modularity of our code, we were able to test out small, but diverse, problem configurations. From our runs we saw that, as we expected, heuristics can provide significant boosts to the efficacy of an EA. However, while we knew heuristics sometimes provided little benefit, we did not expect that they could be detrimental to solving a problem. While we found aggressive heuristics typically helped with larger problems, it actually made the performance of smaller problems worse.

Exploration and Exploitation

As well, we found more aggressive implementations tend to do better not only faster, but also on average. As a consequence, aggressive algorithms are faced with 'staling' occurring in the population more often. This conclusion is part of the reason we added the population management step into the EA, but even with a constant injection of new, randomly generated individuals, it was difficult to improve as we approached the global optimum.

Despite the aggressive algorithm's benefits, in small datasets we found its best run was worse than the other, more random method's best runs. This leads us to believe implementations with more randomness have a better chance of finding the true optimum, as they have more freedom to explore. However, there is a trade off in that they take significantly longer to run, because they explore with less direction.

Finding Balance

Finally, and most importantly, we discovered that the balance between selective and random seems to depend on the problem. The larger the problem, the more aggressive the solution can be, at the risk of getting stuck more frequently. We feel that when using an EA, some basic testing should be done, to determine a combination that works well for the problem, and which should then be decided on a case to case basis.

References

Annealing

Lundy, M. & Mees, A. Mathematical Programming (1986) 34: 111. https://doi.org/10.1007/BF01582166

Corana, A., Marchesi, M., Martini, C. & Ridella, S. ACM Transactions on Mathematical Software (1989) 15: 287.
https://dl.acm.org/citation.cfm?id=29864

Christofide's Heuristic

Frederickson, G., Hecht, M. & Kim, C. Approximation algorithms for some routing problems (1976)
https://ieeexplore.ieee.org/abstract/document/4567906