Usage Instructions

The usage of the program closely follows what was specified in the coursework pdf.
To run it, simply run

python3 tsp_solver.py [PATH TO .TSP FILE]

This was tested with having the .tsp files in the same folder or a subfolder relative to the python script. If any issues arise during the execution of the program due to OS path problems, consider trying to place both files in the same folder.

If you want to enable progress output, there is a boolean at the top of the python file that handles that.

On top of that, there are a lot of command-line arguments to specify different hyperparameters. If you run the program without any other arguments (other than the tsp file), the default values set by me will be used. The table in the hyperparameters section will give you an overview of available command-line arguments and their accepted values, as well as a more detailed explanation of what the effect of their respective hyperparameters on the overall algorithm is.

Introduction

The assigned task was to solve the TSP, a famous NP-complete problem, using stochastic methods like local search and genetic algorithms. Given a fully connected graph, the goal is to find the hamilton tour with the sum of the edge cost minimized - or at least one that approximates the optimal solution. After reading a lot of associated research articles and playing around with different approaches, my final implementation uses the general framework of Genetic Algorithms while applying concepts from local search algorithms to

help with convergence. This combination managed to get satisfactory results on mid to large-sized TSP problem instances.
Even though I am content with the performance of the algorithm so far, I still think there are potential improvements that could be made to the current version.

When reading this report, it will become clear that some of the ideas (2opt, ERO, OX) used were not my own, but rater commonly used methods for implementing GAs for problems similar to TSP. Discussing this with the TA, he confirmed that it was okay to use those, as long as I did the actual implementation myself. I tried my best to add my own twists whenever possible and at the same time learn about the commonly applied methods for solving problems like these.

Hyperparameters

The following table should serve as an overview of all of the available hyperparameters. The meaning of the more esoteric ones should become clearer when reading the associated sections of the report. Due to this large number of hyperparameters, extensive optimization in this regard was infeasible. Hence, I chose the default values according to what proved itself to be beneficial during the development of the GA, providing me with satisfactory results in the end.

Name	Command Line Argument	Values	Default	Explanation
population_size	-p	Int, >= 0	50	The number of individuals created in each iteration of the GA.
fitness_budget	-e	Int, >= 0	40000	The number of allowed fitness evaluations in total
elitism	-f	Int, >= 0	5	The number of best individuals kept between each generation.

k_size	-k	Int, >= 0	10	The size of the randomly sampled pool during tournament selection.
ero_rate	-er	Float, >= 0.0 or -1	-1	The proportion of ERO crossovers when breeding a new generation.
initial_mutation_strength	-ms	Int, >= 0	400	The maximum number of 2-opt swaps to be made during mutation.
mutation_tries	-mt	Int, >= 0	400	The maximum number of subsequent unyielding comparisons allowed while searching for 2-opt swaps.
random_mutation_rate	-mrr	Float, >= 0.0	0.0	The number of vertices randomly swapped after performing the 2-opt mutation.
nn_rate	-nnr	Float, >= 0.0	0.0	The chance of an initial population member being optimized with the nearest neighbor heuristic.
twoopt_rate	-toptr	Float, >= 0.0	0.0	The chance of an initial population member having a random subpath optimized with 2-opt local search.
twoopt_slice_size	-toptl	Int, > 0	50	The size of the path being optimized with a 2-opt local search.

fine_tuning_budget	-fb	Int, > 0	30000	The maximum allowed number of 2-opt comparisons during the fine-tuning of the final model.

Fitness and representation

The basic building blocks of the GA consist of the way that the candidate solutions are represented and the fitness function. For the fitness function, the inverse of the summed tour distance seemed like the most reasonable, as we were trying to minimize distance traveled.

$$f(tour) = \frac{1}{\sum_{i=0}^{n} d(tour[i], tour[i+1 \bmod n])}$$

(with n being the tour length and d() the two dimensional euclidian distance.)

It is to be noted that every individual that is created is immutable and gets the fitness value of its tour calculated instantly. This made the handling of the fitness budget a lot easier, as one created individual costs one point of the fitness evaluation budget.

For the representation of the solutions, I choose to use ordered lists (permutations) of nodes, with implicit edges between neighboring nodes and the first and final node ([1,4,6,8,9, …]). This approach had two main advantages:

1. It made debugging a lot easier as the representation was quite intuitive and identifying invalid individuals was easy.
2. Ordered crossover, the crossover predominately used for tackling larger problem instances, was easy and efficient to implement.

Implementation wise, both the nodes (Locations) and the tours each got their own class to represent them. The purpose of these classes was mainly to make the program more structured and testing/debugging easier down the road.
There are many other possible representations found in common literature (distance matrices, sets of edges) and some even seem promising in regards to the performance of the crossover operators. At least in my case, testing different representations would have required changes in a variety of other areas in the program. As the performance and runtime of my approach were acceptable, I reserved experimentation with other representations for future approaches to this problem.

Due to the nature of the problem, I also considered allowing invalid representations to be created and assigning them with a considerably bigger penalty in their fitness value. As this would have significantly raised the complexity of the algorithm and the rate of invalid solutions for larger problem sets would be very high, I decided against deploying such a strategy.

Initialization

I found the initialization to be one of the more controversial parts of the assignment.
One the one hand, initializing the first population with traditional, non-stochastic algorithms would lead to very strong results in the end with little computational effort. However, doing it like this would relegate the stochastic algorithm to simply finetuning results gathered by traditional heuristics by a small margin, which would somewhat cheat the assignment. As there were no official statements about this, I decided two develop two approaches and have them be controllable with a hyperparameter for grading.

1. Random initialization with some optional 2opt local optimization of subpaths. The idea was that each individual of the initial population could get at least one good sub-path to pass onto new offspring. This would make the initial population stronger while also retaining the stochastic nature of the entire approach. This method of creating the initial population is the default one.

If you take a look at the relevant part of the “initialize” method of the Genetics class, you can see that for each member initialized, there is a chance that a subpath of predefined length is cut out and optimized using 2-opt.

2. Initializing randomly and giving each initial member a chance to be fully optimized with the nearest neighbor heuristic. On big problems, the genetic algorithm struggled to beat solutions produced by NN in a reasonable time. One idea was to initialize the population with very good solutions and use the GA to finetune these. However, NN tended to create solutions near a local minimum, with all edges being pretty short with one long one, making finetuning even harder. I decided against using it for the leaderboard as I felt it would not be in the spirit of the assignment to make a regular heuristic do most of the work. This initialization can be optionally enabled via hyperparameter.
   Using both NN and 2opt subpaths showed no improvement over NN alone.

The Genetic Algorithms Loop

As stated before, the algorithm follows the framework laid out by common genetic algorithms. After creating an initial population, I create new generations by selecting suitable parent-tours for each member of the new generation (or keep some in the case of elitism), combine their permutations of nodes via crossover operators, mutate the offspring permutation and create a new tour object from the result. For each member of the new population, I deduct one point of the fitness budget, as the fitness evaluation happens at creation. The fittest tour over all generations is always saved, as to not lose the previous best solution. As far as the Implementation goes, as soon as a “Genetic” object has been instantiated with all of its hyperparameters, calling the run method on it would use up the entire assigned budget to run the described optimization process. As I tried to make each method name descriptive in regard to their role in the process, you can see the different parts of the loop if you take a look at the source code for the “run”, “next_population” and “mate” methods.

Selection

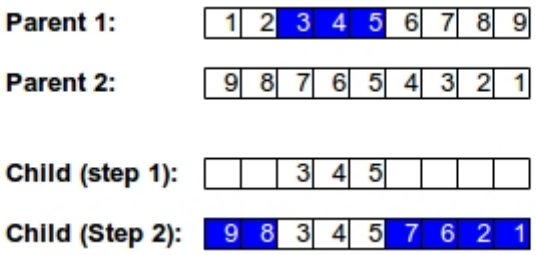
When it came to selection, I choose to keep it simple as to better see the effects of other parts of the GA. I implemented a tournament selection just like it was presented in class: From the pool of possible parents, I randomly sampled a certain amount of candidates(as specified by a hyperparameter) and picked the best one from that bunch. Moreover, I also kept an archive of top-performing solutions from each generation and just carried them over to the next generation unchanged.
While I initially planned on experimenting with different selection mechanisms, I choose to stick with the described model as I was satisfied with the performance it was providing across all different GA setups. Also, seeing how both low runtime and the rate of convergence were really important aspects when tackling bigger problems, the relatively simple tournament selections speed and the converging properties of elitism proofed to be beneficial in that regard.

Crossover

Crossovers for the TSP offer a somewhat peculiar challenge, as each location can only be visited once during each tour. This eliminates crossovers like One Point and requires operators specifically tailored to permutation problems.

Ordered Crossover

My first approach exclusively used the ordered crossover operator (OX) to produce offspring from two parents. It randomly picks two indices and cuts out the corresponding sub-path from one of the parents and adds the remaining nodes from the other parent while keeping them in their original order (with the nodes present in the cutout omitted). This is one f the simplest ways of doing a crossover that only produces children that do not violate the rules for route representations. The following figure shows an example crossover done via OX.



Source: http://dtnewman.github.io/finch/genetic_algorithms.html

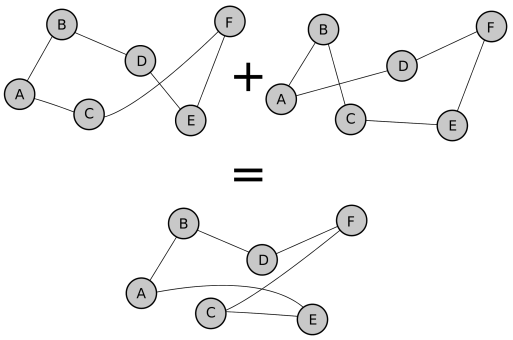
My implementation differs from the shown figure by always appending the leftover nodes from the second parent at the end of the cut-out part. However, seeing as the starting point of the route is not important for the solution, it is safe to assume that this does not change the crossover in any significant way.
You can find the associated code in the crossover_ox method of the “Genetic” class.

Issues with OX

While the performance of OX was decent and the runtime was quite fast, I was not satisfied with the number of iterations it took to converge. I also witnessed early plateauing on the bigger problem instances, leading me to assume that the destructive properties of the crossover (ripping apart the paths from the second parent) were preventing the solutions from improving after certain points.
After talking with the TA about these issues, he pointed me to some Wikipedia articles and I ultimately choose to implement the “Edge Recombination Crossover” .

Edge Recombination Crossover

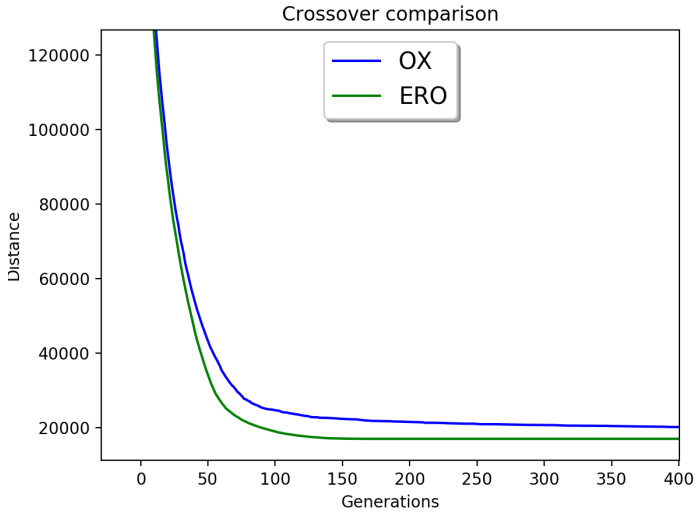
ERO solves the “destructive” properties of OX by trying to introduce very little new connections/paths, as they could have detrimental effects on the fitness of new offspring. It achieves this by using as many of the existing edges as possible.



Source: https://en.wikipedia.org/wiki/Edge_recombination_operator

ERO first creates a mapping that keeps track of all neighbors for each node across both graphs combined. It starts out with a randomly selected node. As long as not all nodes have been added to the child, it appends the selected node onto the child graphs and deletes it from the list of potential members of each node. It then proceeds to select the next node to add by looking at all of the neighbors of the previously added node and choosing the one that has the least neighbors itself. In the case of a tie, a random neighbor is chosen. If the previously added node does not have any more potential neighbors (meaning all are already present in the tour) a random node not already part of the child-tour is chosen as the next node.
For a more in-depth explanation of the algorithm, the Wikipedia article (linked at the source of the above figure) can help.

The figure below shows the improved performance of ERO when compared to OX. Not only does ERO converge in fewer iterations, but it also plateaus at better optima when running for more generations. The data points were averaged over 20 GA runs for each crossover method used. The mutation strength was lower than the final default value (20 vs 400), as to not skew the comparison between the crossovers. The benchmarks I did to compare different GA setups (here and in the following sections) were all done with the rd400.tsp problem instance from tsplib.

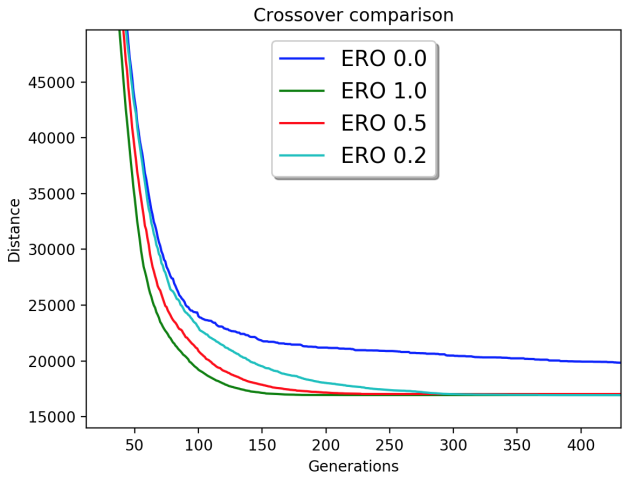


Issues with ERO

However, I ran into problems with ERO when it came to very large problem instances. Seeing that the operator iterates over representations of the graph much more often than OX, it is evident that ERO is significantly more computability expensive (at least in my implementation).
That left me with a dilemma of sorts. ERO showed much better performance in terms of convergence, but using ERO for the entire population would be infeasible for large problem instances.

ERO/OX Hybrid

While rewriting the way routes are represented could have reduced the amount of computation needed, I ultimately decided to solve this issue in another fashion.
I theorized that using ERO to produce only a portion of the offspring could already have beneficial impacts on the convergence of the GA. So by only using ERO for some of the crossovers and relying on the less resource-intensive OX otherwise, I could keep computation time low while also taking benefit of the properties ERO offers. The following figure shows that even though reduced rates of ERO crossovers take longer to converge, they eventually reach similar fitness to the GAs using EROS for their entire population.

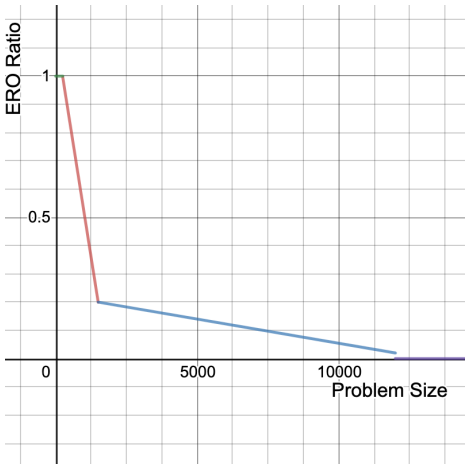


If you want to have a more detailed look into the implementation of the crossover procedure, I annotated the relevant parts in the code to make them easier to understand.

Fine-tuning the ERO/OX ratio for every specific problem size would be tedious and I did not know what size the assignment would be evaluated on. To tackle these issues, I choose to add a simple function that estimates a suitable ratio for any given problem size.
During development, I took note of the approximate ratios that would produce good results for the examined problem instances while also having acceptable runtimes. These were the ones for my highly tested problem instances:

Problem Size	ERO Rate
400	0.9
1500	0.2
12000	0.02

Going from this small sample, I build a rough estimator using linear functions that would connect these points, as is shown below. For the exact linear equations, please refer to the ero_rate_normalizer method.



It is evident that this is only a very naive approximation and more sophisticated methods along with more testing could produce better performance/runtime tradeoffs. For the scope of this assignment, however, I found that the ratios this function would provide were decent enough.

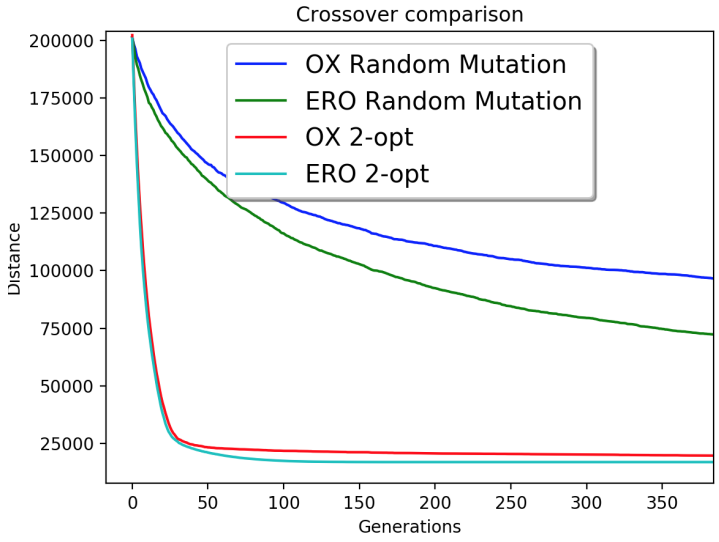
Mutation

After offspring are created by the described crossover, I apply a “pseudo-mutation”.
This mutation uses the core concept of the 2-opt local search. 2-opt looks at different edge pairs and decides if crossing the edges (i.e. AC & BD -> AB & CD, see the figure below for a visualization) could reduce the sum of both the edge costs/lengths. These swaps make use of the fact that vertices that are connected in a way in which the edges cross each other always produce tours that are longer than the non-crossing way to connect them.

Source http://pedrohfsd.com/2017/08/09/2opt-part1.html

The mutation operator mimics this in a more random fashion: It picks two random edges and “swaps” them if the swap would result in a reduced sum of the examined edge lengths. This process is repeated until either the swap limit (hyperparameter) is reached or the maximum amount of edge pairs (hyperparameter) are compared without any viable swaps being found.

I call this method “pseudo-mutation”, as the mutation only results in better or equally good solutions compared to the original offspring. The randomness mainly comes from the random selection of edges to compare. This combined with the randomness that the crossovers introduce seems to provide the algorithm with a fair amount of diversity. Considering this is not a traditional mutation, this would ultimately make the approach more of a light hybrid between local search and GA.
The figure below shows how this mutation with a strength of 50 clearly dominates just randomly swapping in terms of convergence. Concerns about too little exploration can be made, but the time the random mutation would take to converge seems infeasible, especially when applied to larger problem sets.



One notable comment has to be made at this point: This offspring somewhat cheats the fitness budget for the algorithm. It introduces many comparisons during the mutation phase that are not accounted for in the fitness budget. However, as these are very cheap and do not scale with the problem size (being only dependant on the chosen hyperparameters), I found this to be acceptable. Also, I decided to add a bit of fitness cost to each generation, treating the elites as if they were new members from a fitness budget perspective. This should make the overall fitness budget more representative of the number of comparisons made.

Fine Tuning

While using heuristics for initialization might assign a secondary role to the GA, I felt that letting the GA do the main portion of the optimization and using regular heuristics to improve that solution afterward could very well be considered in the spirit of this assignment.
After the fitness budget is used up, I made it optional to further fine-tune the solution by running it through a fully-fledged 2-opt local search. This local search would check all possible pairs of edges to see if a swap would be beneficial and continues to do this until no more improvements can be made or the specified fine-tuning budget (number of allowed comparisons) runs out. While the improvements achieved by this finetuning were never hugely significant, it allowed to slightly improve the final solution with even very little extra budget, making its inclusion worthwhile.

Evaluation

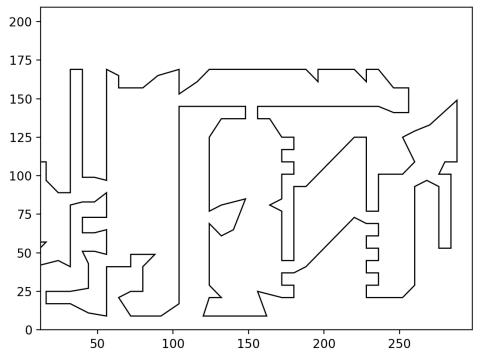
Apart from the leaderboard benchmark (which was the main focus), I also compared the performance of my GA to some of the solutions provided by tsplib.

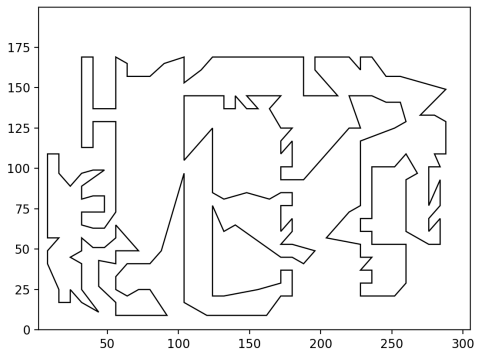
With the default hyperparameters, my GA managed to find the correct solution to the berlin52 problem instance in around 8 generations on average. In the cases that the optimum was not reached, I could witness deviations of at most 3% in distance.

With the same setup, the GA also showed satisfactory performance on the a280 set. While it rarely reached the real solution, it managed to get into around (on average) 12% of the best

solution within 50-100 generations. With 0.1 nearest neighbor initialization, I managed to cut that down to 10% in 32 generations.

To get a better feeling for the performance of my algorithms, I also visualized the finished tours after optimization. If you compare the optimal a280 tour (left) to one of the results of my GA (right), you can see that there were actually some optimal subpaths found. Also, you can see the effect that both the 2-opt mutation and fine-tuning had on the graph, as it has no crossing edges at all.





Possible Improvements

While my approach reached acceptable performance in the scope of this assignment, there are still many areas that I could look into to further improve my results.

Mutation

The 2opt pseudo-mutation-operator may have even more potential when dealing with big TSP instances. One idea would be to apply it to fully optimize smaller subroutes in the graph instead of doing random singular edge swaps. This would give each child a guarantee of having at least one part worth carrying over to the next generation. As the effects of this approach on diversity are not clear, further testing would be needed.

Edge Assembly Crossover

When it comes to crossover operators, I discovered the “Edge Assembly Crossover” proposed by Yuichi Nagata and Shigenobu Kobayashi while researching. Going by the results they have presented over multiple papers, this crossover operator seems to have huge potential when dealing with large TSP instances, making convergence much faster. However, as the efficient implementation of that particular crossover is tricky, I decided that it would be better to focus my time on other parts of the assignment. Understanding it, however, was a great exercise in seeing how complex some crossovers can get and I will definitely keep it in mind for future GA based projects.

Implementation Details

Implementation wise, there are still some minor cases of code reuse and some areas that could benefit from refactoring. I will for sure come back to these issues if I decide to reuse the code as a part of a bigger project in the future. Also, seeing how one of my main operations is the swapping of edges, a representation as a set of edges rather than a permutation of vertices could be beneficial for the overall performance of the algorithm.

Adaptive Hyperparameters

Looking at the adaptive hyperparameters, there is obvious room for improvement. Of course, manually fitting the hyperparameters to the problem size would produce the best results. However, more sophisticated methods for adaptive (depending on the size) or even dynamic assignment of the OX/ERO ratio at runtime could also be beneficial for performance. As the amount of OX and ERO performed also has an influence on the evolutionary pressure of the algorithm, fine-tuning these values automatically even has the potential to improve the convergence rate of the entire algorithm.

Testing/Evaluation

Finally, a dedicated/stronger testing suite for different GA setups would have been a huge help in choosing the right hyperparameters and gauging the performance of the final model. As this was not a main part of the assignment, I focused more on the GA itself, but looking

back, the time invested in such a framework could have made certain parts of development easier.

Conclusion/Leaderboard

For my submission to the leaderboard, I let the GA run for about 3600 generations using the default hyperparameters (with a respectively higher fitness budget). I managed to achieve a final distance score of 6 907 393, while the GA passed the 8.000.000 mark in around 727 generations.