Evolutionary Algorithms: Final report

Arthur Warlop (r0812080)

December 31, 2023

1 Metadata

- **Group members during group phase:** Tim Lauwers and Vincent Render
- **Time spent on group phase:** 15 hours
- **Time spent on final code:** 150 hours
- **Time spent on final report:** 20 hours

2 Changes since the group phase (target: 0.25 pages)

1. Represented the individuals as objects of the *Individual* class. (Section 3.3)
2. Initialized a part of the population with the *Nearest Neighbour* heuristic to lead the search into more promising areas of the search space. (Section 3.4)
3. Implemented a backtracking algorithm to find random but valid individuals. (Section 3.4)
4. Applied the *3-opt* local search operator to a part of the population during initialisation, to enrich the population. (Section 3.4)
5. Implemented scramble mutation, insert mutation and swap mutation. (Section 3.6)
6. Introduced randomness into the selection process for the mutation operator, to increase diversity. (Section 3.6)
7. Introduced self-adaptivity for the mutation rate. (Section 3.6)
8. Introduced the island model. (Section 3.10)
9. Replaced the Edge Crossover (EX) operator by Partially Mapped Crossover (PMX) in island 1 and Order Crossover (OX) in island 2. (Section 3.7)
10. Added fitness sharing during the elimination step to promote diversity and prevent premature convergence. (Section 3.10)
11. Applied the *3-opt* local search operator during variation. (Section 3.9)
12. Introduced elitism to prevent the best seed individual from mutating. (Section 3.6)
13. Every part of the algorithm has been optimized, resulting in a significant increase in execution speed and performance.

3 Final design of the evolutionary algorithm (target: 3.5 pages)

3.1 The three main features

1. By introducing the 3-opt local search operator, the population is enriched during initialisation and variation, yielding significantly improved solutions within a five-minute time span. This operator is the most impactful performance improvement for the algorithm, and is crucial to find high-quality individuals with a low fitness value. Through strategic code optimizations, the operator became computationally feasible and was successfully applied to all individuals in the seed population during variation.
2. To counterbalance this strong 3-opt operator and reduce the selective pressure, it is imperative to incorporate robust diversity promotion mechanisms into the algorithm. Increasing population diversity is achieved through the utilization of an island model and fitness sharing during elimination. This balances the exploitation and exploration and will help the algorithm to escape local minima while consistently finding high-quality individuals.
3. To guide the search process from the start towards more interesting areas of the search space, a diverse but rich population is initialised. The majority of the population is initialized randomly, while 20% is initialized using the *Nearest Neighbour* heuristic. Following this, the population is further enriched by applying the 3-opt operator to 25% of the entire population.

3.2 The main loop

3.3 Representation

Candidate solutions are represented by **permutations** of integers. Each element in the permutations corresponds to a city, and the order of these elements indicates the visiting order of the cities. This representation is intuïtive, compact and straightforward to utilize. The permutation is implemented as a numpy array that is stored in the *permutation* field of the ***Individual* class**. Each individual in the population is represented as an object of this class. Next to the *permutation* field, three other attributes are included in the *Individual* class. The *alpha* attribute stores the mutation rate of the individual and is used for self-adaptivity. The *fitness* attribute stores the objective value of the individual. The *fitness share* attribute stores the temporary fitness value during fitness sharing.

3.4 Initialization

To ensure sufficient **diversity** of the candidate solutions, the majority of the population is randomly initialized. Two method have been considered to do this. The first method generates **random** permutations, resulting in numerous invalid permutations with an infinite fitness value. The second method utilizes a backtracking algorithm to find random **valid** paths of finite length. This algorithm starts in a random city, and randomly selects an adjacent unvisited city as the next city to visit. This is done iteratively until the permutation is complete, or no available adjacent cities remain. In the latter case, the algorithm backtracks, and another random city is selected in the previous iteration. With this method, the search started with more promising individuals while maintaining diversity, and it yielded better fitness values within the five-minute time span. Therefore, this method was selected.

To **enrich** the population, the remaining individuals are initialized using the ***Nearest Neighbour (NN)*** heuristic. The *NN* algorithm starts by selecting a random city and iteratively travels to the closest city until all cities have been visited. Similar to the aforementioned algorithm, the algorithm backtracks if no available adjacent cities remain, and the second closest city is chosen in the previous iteration. To prevent an immediate takeover by the enriched individuals and avoid early convergence, the *NN* algorithm is employed to initialize only **20%** of the population.

Finally, to guide the search process towards more interesting areas of the search space, a **local search operator**, described in 3.9, is applied to **25%** of the whole population.

After applying code optimizations, the initialisation execution time was reduced by a factor of three and the initialisation is executed in less than **10 seconds** for the biggest problem (tour1000). The determination of λ and μ values is discussed in 3.12.

In summary, the initialisation strategy divides the population into four distinct parts: 37,5 % random individuals, 12.5 % good (random + lso) individuals, 15 % very good (NN) individuals, and 5 % excellent (NN + lso) individuals. This created a diverse population with rich and promising individuals, which resulted in a smooth convergence towards individuals with a lower fitness values, with enough diversity to escape local minima.

3.5 Selection operators

The ***k-tournament*** selection operator was implemented, where k denotes the number of individuals selected without replacement for a tournament. Tuning this parameter is essential to balance selective pressure and diversity. A higher k-value increases the likelihood of selecting superior individuals, emphasizing exploitation, while a lower k-value allows the selection of individuals with lower fitnesses, resulting in more exploration. Self-adaptivity was not applied to the k-value. The determination of the optimal k-value is discussed in 3.12

3.6 Mutation operators

Four distinct mutation operators have been implemented: **insert** mutation, **swap** mutation, **inversion** mutation, and **scramble** mutation. According to literature [1], inversion mutation is more commonly used, resulting in fewer broken links. However, this observation holds true for the **symmetric** Traveling Salesman Problem. Given the **asymmetric** nature of the problem in this project, all links in the reversed section will also be broken, resulting in an inversion mutation with a similar effect to the scramble mutation. To introduce more randomness and diversity, the mutation operator is selected **randomly**. This resulted in a more diverse population and better solutions. **Elitism** is applied to prevent the best k seed individuals from being mutated, with k being a tunable parameter set to 1. The α-value, denoting the mutation rate, is part of the representation, as discussed in part 3.3, and controlled by **self-adaptivity**. Self-adaptivity allows the algorithm to adjust the alpha value dynamically based on the performance and characteristics of the evolving population. For example, if there is early convergence to a local minimum, the mutation rate might increase to promote exploration. The use of self-adaptivity has significantly enhanced the algorithm's performance. The minimum value for α is set to 0.05. When crossover is applied to two parents p_1 and p_2, the α-value of the resulting child is determined as follows:

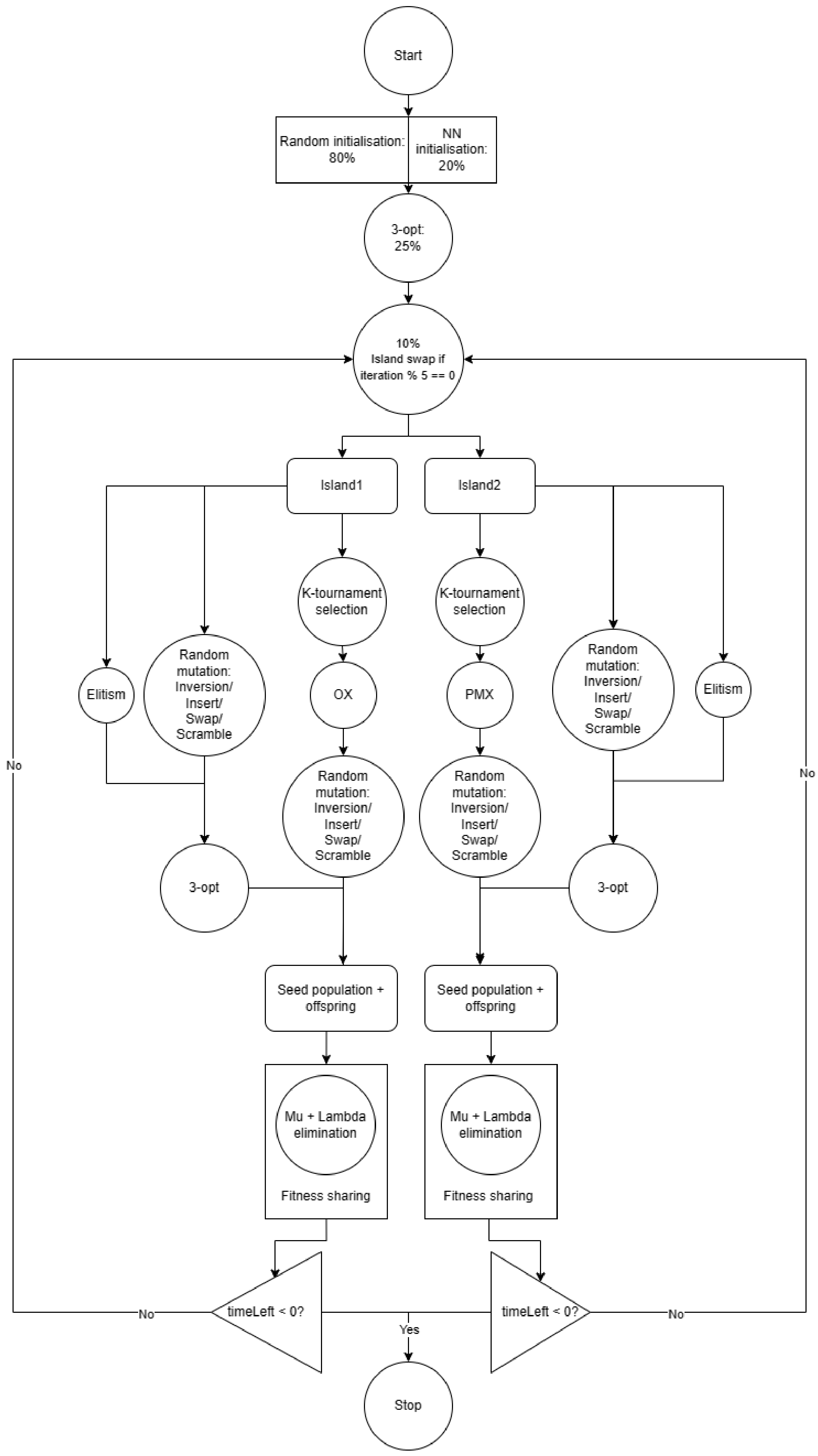

Figure 1: Main loop.

$alpha_{child} = max(\alpha_{p1} + \beta * (\alpha_{p2} - \alpha_{p1}), 0.05)$. Here, β is a random float in the half-open interval $[0.5, 1.5)$.

3.7 Recombination operators

Multiple crossover operators have been implemented: **Partially Mapped Crossover (PMX)**, **Cycle Crossover (CX)** [1], **Order Crossover (OX)** and **Complete Subtour Order Crossover (CSOX)** [4]. Since both Order Crossover and Partially Mapped Crossover resulted in a better performance than the other operators, they are both used in one of the two islands.

Although literature suggests that Edge Crossover [1] and Complete Subtour Order Crossover (CSOX) [4] are more suitable for the Traveling Salesman Problem, experiments have revealed that utilizing Partially Mapped Crossover (PMX) and Order Crossover (OX) for the different islands resulted in consistent superior performance.

The **Partially Mapped Crossover (PMX)** starts by selecting two random crossover points on both parent permutations. It then copies the segment between the chosen crossover points from the first parent into the corresponding segment of the first offspring. Starting from the first crossover point, cities are identified in the copied segment of the second parent that have not been copied to the offspring. For each of these cities, the offspring is checked to find the city that has taken its place from the first parent. The current city is then placed into the position occupied by this city in the offspring. If the position occupied by this city has already been filled in the offspring by another city, the current city is placed in the position occupied by that other city. This process is repeated until the current city is copied into the offspring. Once the elements from the crossover segment have been handled, the remaining positions in the offspring are filled with cities from the second parent. This process is repeated with the parents reversed to create the second offspring.

The PMX operator contributes to the exploration-exploitation balance. The resulting recombination produces offspring that share characteristics with both parents, combining segments directly inherited from the first parent with segments from the other, potentially resulting in a solution that combines the best features of both parents and promoting exploitation. The operator also allows the introduction of new information, potentially leading to the discovery of better features that might not exist in the parents, promoting exploration.

The **Order Crossover (OX)** operator begins by selecting a subset of elements from one parent and copies it to the offspring. The remaining positions in the offspring are then filled by adding the missing elements from the second parent, maintaining the order of appearance in the second parent.

The superior performance of the OX operator is due to the nature of the problem addressed in this project. Since the travelling salesman problem in this context is **asymmetric**, the visiting order is important. Moving from city A to city B does not result in the same cost as moving from city B to A. The OX operator, by respecting the order of cities in the parents, strives to retain this order as much as possible in the offspring, even if there is little overlap between the parents. In an asymmetric context, the information from parent permutations is optimally inherited by the offspring. In contrast, edge crossover more frequently reverses the order of two visiting cities, leading to more broken edges in the asymmetric Traveling Salesman Problem.

3.8 Elimination operators

Three elimination operators were explored: $(\mu + \lambda)$ elimination, k-tournament elimination and (μ, λ) elimination. After careful consideration, the $(\mu + \lambda)$ elimination was selected as the optimal choice. During $(\mu + \lambda)$ elimination, the λ best individuals are promoted to the next iteration, which introduces significant selective pressure and can lead to a loss in diversity. To counterbalance this, **fitness sharing** diversity promotion was integrated, as detailed in 3.10. The combination of $(\mu + \lambda)$ elimination with fitness sharing achieved the best results and was more consistent than the other considered operators. The combination of fitness sharing with k-tournament elimination was also implemented, but resulted in a slower convergence.

3.9 Local search operators

After an in-depth exploration of the available approaches in the literature [3][2], three local search operators were implemented: *Nearest Neighbour*, *2-opt* and *3-opt*. *Nearest Neighbour* was discussed in section 3.4.

The **2-opt** operator iteratively evaluates all possible edge swaps within a permutation. For each pair of edges, the operator calculates the change in fitness value that would result from the swap. If the swap leads to a shorter tour, the edges are reversed in the permutation. This process continues until no further improvements can be made. In the context of the **symmetric Traveling Salesman Problem (STSP)**, 2-opt is ideal, as only two edges are broken per swap. The fitness change introduced by a 2-opt swap is only determined by the lengths of the replaced and new edges. If the new edges are shorter, the individual has been improved. However, for the **asymmetric Traveling Salesman Problem (ATSP)** addressed in this project 2-opt is not the optimal choice. By swapping two edges, all the cities between them are visited in reverse order, breaking all the edges between those cities due to the inherent asymmetry of the problem. Consider the 2-opt swap example on figure 2. Every city between B and C is visited in reverse order after the swap compared to before the swap. Consequently, each swap breaks a consistent number of edges, creating a completely different permutation. This causes the 2-opt operator to optimize over a large area of the search space, contrary to the intended local search effect. Additionally, the distance of all broken edges must be computed in each iteration to calculate the fitness change introduced by the swap, which is computationally very expensive.

To tackle the asymmetric TSP problem in this project, the **3-opt** operator is better suited. This operator performs a three-edge swap to locally improve solutions. Seven possible ways to swap three edges exist, as displayed in figure 3. The first six swaps introduce the same issue as the 2-opt operator, reverting parts of the permutation as well. However, the seventh swap, as displayed on the figure, avoids this problem by maintaining the visiting order of the cities in the unswapped edges. This swap does not radically transform the permutation and focus on local optimization rather than searching a large area. Consequently, it has been implemented as a local search operator for this project. The fitness change introduced by this swap is only determined by the lengths of the replaced and new edges, making the calculation of the fitness change computationally inexpensive. However, due to the substantial number of possible three-edge combinations, the application of a 3-opt operator remains computationally expensive and unfeasible without code optimizations. Several **optimizations** have been implemented for this purpose. The first and most impactful optimization involves **early breaking** when the first new edge is invalid and has an infinite length, or when the fitness change introduced by this new edge is not greater than the already found fitness change. By introducing this optimization the operator became feasible, even though the resulting solution may be sub-optimal. The second optimization involves early breaking when the other new edges have an infinite length in the last for loop. The third optimization consists of returning the new edges when one of the old edges is infinite in the final loop, resulting in an infinite fitness change that cannot be improved upon. A final optimization involves using **numba**, which significantly decreased the execution time. Similar optimizations have been implemented for the 2-opt algorithm; however, as mentioned previously, the 2-opt operator resulted in a lower performance due to the asymmetric nature of the problem.

The 3-opt local searche operator is very strong and is the most significant performance improvement for the algorithm. It is applied on the seed population after mutation. With the code optimizations it is possible to apply it to every individual of the seed population. The individual removed from the seed population due to elitism is reintroduced before the 3-opt operation to ensure that it can also be enhanced. While experimenting, applying the 3-opt operator to the offspring was tested but resulted in quicker convergence and stagnation, as the offspring dominated the population.

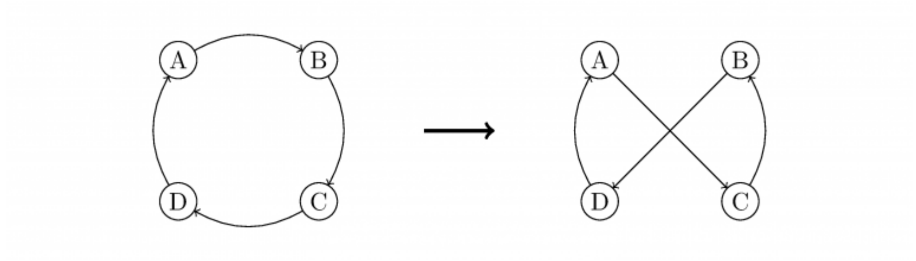

Figure 2: 2-opt local search operator.

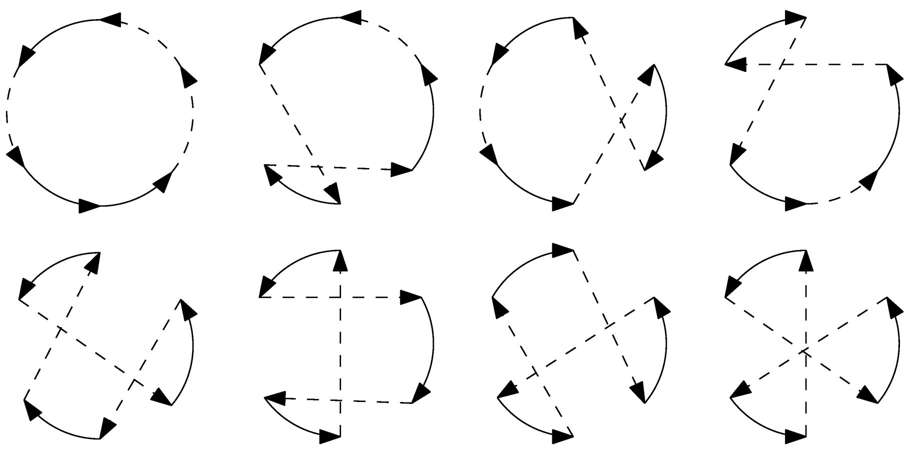

Figure 3: 3-opt local search operator.

3.10 Diversity promotion mechanisms

With this very strong local search operator described in section 3.9, achieving a balance between exploitation and exploration necessitates a strong diversity promotion algorithm. Four different methods were implemented: **Crowding**, **Fitness sharing** during **elimination**, Fitness sharing during **selection**, and the **Island model**. After extensive experimentation and performance comparison, the Island model and Fitness sharing during elimination proved to be the most effective.

Fitness sharing decreases the fitnesses of the individuals that are close to the individuals that are already selected for the next iteration. Fitness sharing has been combined with both k-tournament elimination and the $(\mu+\lambda)$ elimination, as explained in section 3.8. The **Hamming** distance was selected as the most suitable metric for comparing two permutations and evaluating the distance between individuals in the context of fitness sharing. To reduce the execution time of a fitness sharing iteration, a code **optimization** has been implemented. Instead of calculating the distance between the entire population and the promoted population in each iteration to update all the fitness sharing values, the distance is now calculated only between the newly promoted individual and the individuals that have not yet been promoted, and the fitness sharing value is updated according to that distance. Additionally, another optimization involves leveraging *Numba* for efficiently computing the distance between two individuals. The fitness sharing mechanism has two tunable parameters. The σ parameter determines the region around each individual where the effects of sharing are significant. The α parameter controls the rate at which the fitness increases as the distance between individuals decreases.

To further increase diversity in the population and emphasize exploration, the population was divided into **two islands**, each using its own crossover operator. The first island utilizes order crossover, while the second island employs partially mapped crossover. Each island evolves independently. At intervals of every **5 iterations**, **10%** of the population individuals are randomly chosen within each island and exchanged between them. The number of islands was determined based on the available physical CPU cores that will be available during testing. **Multithreading** has been implemented to evolve each island using a distinct thread. However, this increased the execution time, as the allocation of work to threads became a bottleneck for performance, and the speedup gained from multithreading did not adequately compensate for it.

3.11 Stopping criterion

As the most challenging problems did not converge within the first **five minutes**, no additional stopping criteria have been implemented for them. In contrast, due to the rapid convergence of the smallest problem (tour50), a specific stopping criterion has been introduced for it, terminating the run if the best individual remains unchanged for **1000 iterations**.

3.12 Parameter selection

With the code optimizations introduced in every part of the algorithm, the population and offspring size were not a bottleneck. Multiple population and offspring sizes have been tested, ranging from 50 to 100. Since no values outperformed the others consistently, the lambda and mu values were both set to 50 and 100, respectively. All the other parameters were determined with a hyperparameter search. The hyperparameter search was applied on the three biggest benchmark problems, since it was assumed that the optimal solution was achieved on the three smaller problems, as all the different tests applied on these benchmarks resulted in exactly the same fitness value found. The parameters resulting from the parameter selection are described in section 4.

4 Numerical experiments (target: 1.5 pages)

4.1 Metadata

The experiments were executed in Python 3.10.12 on 4 physical CPU cores (8 logical cores) of a machine containing an Intel Core i7-1165G7 CPU, with a clock speed of 2.8 GHz and 16GB of main memory.

The parameters that were chosen for the experiments are listed below. Self-adaptivity has only been applied to the mutation rate.

- λ = 50: size of the population
- μ = 100: size of the offspring
- p_l = 1: the share of the population to which 3-opt has been applied during variation
- p_i = 0.25: the share of the population to which 3-opt has been applied during initialisation
- p_k = 0.2: the share of the population that has been initialised with the NN heuristic.
- p_s = 0.1: the share of the population that is swapped during island swapping
- i_s = 5: the number of iterations that are executed between each island swap

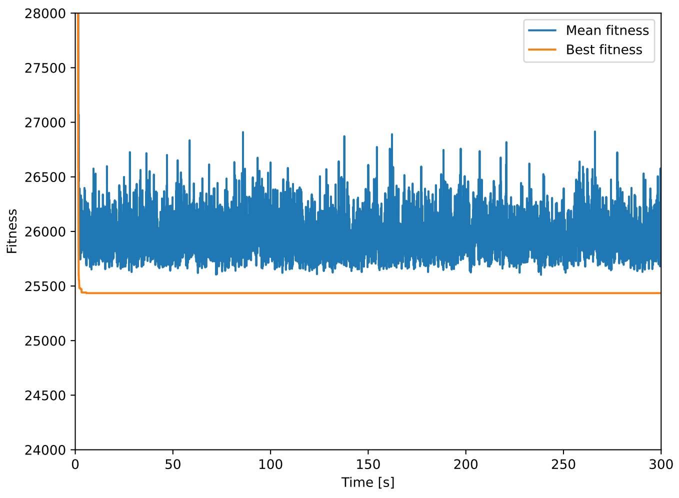

Figure 4: Mean and best fitness values over time for tour50.

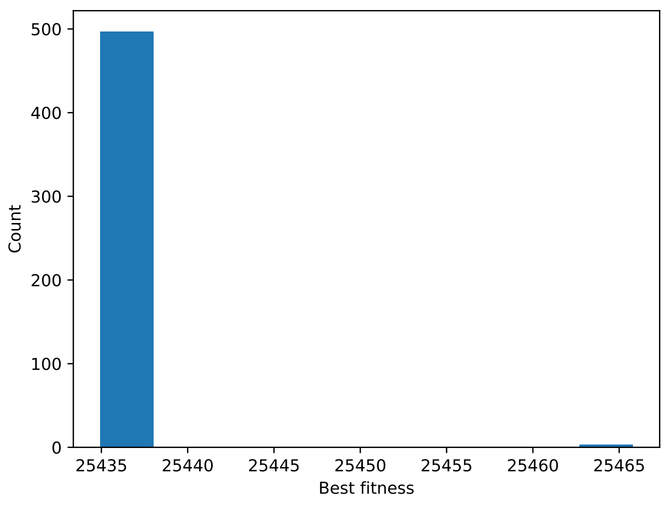

Figure 5: Histogram of final best fitnesses for 500 runs

- k_s = 3: k-value used during k-tournament selection
- k_e = 1: the number of best individuals that are prevented from being mutated due to elitism
- α_f = 1: the alpha value used during fitness sharing
- σ_f = 1: the share of the permutation length that is considered as the neighborhood during fitness sharing

4.2 tour50.csv

The best tour length found for benchmark tour50 is **25434**.918404427266. The corresponding sequence of cities is

[4,15,47,42,11,3,19,38,10,13,7,41,14,1,43,30,9,48,6,22,49,20,18,24,45,2,31,29
,37,39,25,12,27,28,36,23,44,17,40,33,34,21,32,0,26,5,8,46,35,16]

The convergence graph, presented on Figure 4, depicts a rapid convergence, accomplished almost instantly (within 3 seconds). The mean does not converge and is oscillating until the end, highlighting the diversity within the population. In the histogram displayed in Figure 5, it can be observed that in all but one run, the algorithm finds the same fitness value, 25434, which is probably the optimal one. On the histogram of the mean fitness values, displayed on figure 6, the diversity of the population can be observed. The mean value is consistently higher than the best fitness, highlighting the diversity and the presence of suboptimal individuals within the population.

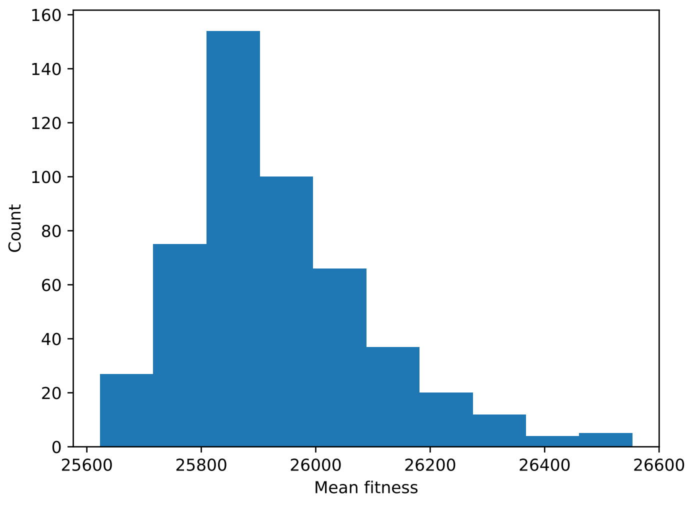

Figure 6: Histogram of final mean fitnesses for 500 runs

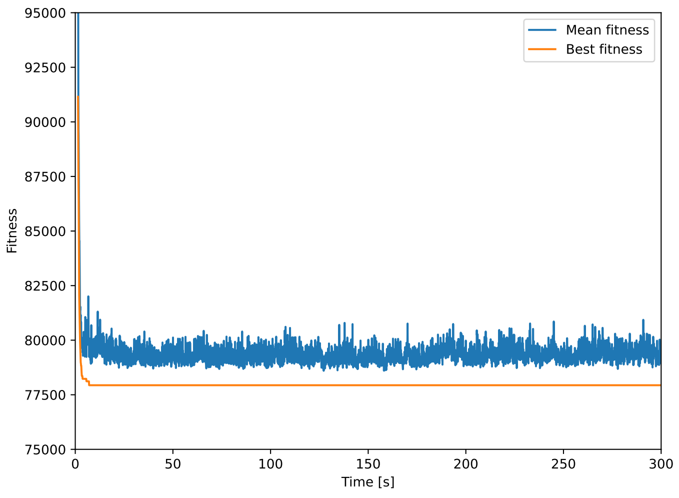

Figure 7: Mean and best fitness values over time for tour100.

4.3 tour100.csv

The best tour length found for benchmark tour100.csv is **77936**.83035451944

For this benchmark, the convergence also happens very fast, as showed on figure 7. As can be seen from the curve of the mean fitness, there is a good preservation of the diversity. The mean value does not converge and offers the possibility to escape from a possible local minimum. Since the best fitness does not change even with the high diversity, it is assumed that the optimal value has been found.

4.4 tour100.csv

The best tour length found for benchmark tour500.csv is **125624**.64017341478. From the convergence graph on figure 8 it can be observed that the best fitness is still improving after 5 minutes and that the algorithm hasn't converged yet. The mean fitness, shows multiple jumps representing the promotion of diversity introduced by the fitness sharing and island model. Since the algorithm still has not converged yet, the found fitness value is not the optimal one. However, since the fitness is very slowly decreasing at the end, it is assumed that the optimal solution is not far away.

4.5 tour1000.csv

The best fitness found four benchmark tour1000.csv is **152187**.53244790164.

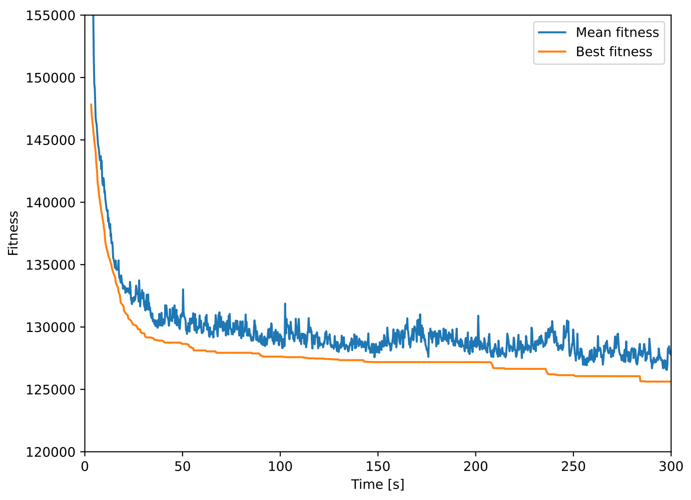

Figure 8: Mean and best fitness values over time for tour500.

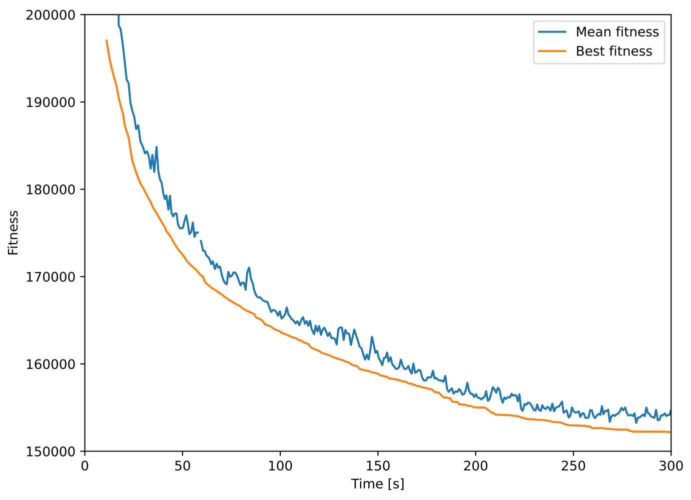

Figure 9: Mean and best fitness values over time for tour1000.

From the convergence graph on figure 9 it can be observed that the best fitness is still improving after 5 minutes and that the algorithm hasn't converged yet. Once again a diverse population can be observed since the mean fitness stays higher than the best fitness.

5 Critical reflection (target: 0.75 pages)

What are the three main strengths of evolutionary algorithms in your experience?

1. Evolutionary Algorithms (EAs) can be applied to a wide range of problems and may be the preferred approach when an optimal solution is not necessary, but a suboptimal solution suffices.
2. EAs are very customizable and can be adapted in many ways with hyperparametertuning, the introduction of new methods,...
3. There is a lot of literature on EAs accessible online. This is very helpful when conducting research on this topic.

What are the three main weak points of evolutionary algorithms in your experience?

1. The hyperparameter search is one of the weak points of Evolutionary Algorithms (EAs). Because there is no silver bullet and the parameters have to be determined for each problem independently, the whole

hyperparameter search procedure has to be repeated for each new problem. Additionally, there are a lot of parameters to tune, and this is very time consuming.

2. For almost every problem addressed by EAs, alternative methods that are both faster and more effective are available.
3. Evolutionary algorithms (EAs) come with a significant computational cost. To find one good individual, a vast amount of individuals has to be created and deleted afterwards.

References

[1] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. 2 edition, 2015.

[2] Goran Martinovic and Drazen Bajer. Impact of nna implementation on ga performance for the tsp. 2022.

[3] Gerard Sierksma. Hamiltonicity and the 3-opt procedure for the traveling salesman problem. 2022.

[4] Thanan Toathom and Paskorn Champrasert. The complete subtour order crossover in genetic algorithms for traveling salesman problem solving. 2022.