Evolutionary Algorithms: Final report

Jorrit Willaert (r0652971)

December 30, 2021

1 Metadata

- **Group members during group phase:** Lukas De Greve and Thomas Vanhemel
- **Time spent on group phase:** 10 hours
- **Time spent on final code:** 50 hours
- **Time spent on final report:** 16 hours

2 Peer review reports

2.1 The weak points

1. Our initial recombination operator was a **simplified version of the edge crossover operator** [3]. This version did not prioritize common edges between parents, but only chose an entry which itself had the shortest list. Hence, not enough exploitation of the parents' features follows.
2. The $(\kappa + \mu)$-elimination (without any type of diversity promotion) puts a **lot of selective pressure** on the population.
3. Our **mutation operator does not scale** to larger problem sizes, since it only swaps two random locations. As a consequence, the mutation operator will have an even smaller impact on the solution when the problem size increases.
4. Due to the elimination strategy, together with the chosen mutation operator, **premature convergence** was observed.

2.2 The solutions

1. The simplified version of the edge crossover operator was first replaced with the **proper edge crossover operator** [1]. However, once the running times of edge crossover were compared with the ones of order crossover, it was apparent that **order crossover** ran much faster. For this reason, edge crossover was abandoned in favor for order crossover.
2. The $(\kappa + \mu)$-elimination scheme was kept for quite some time, but was supplemented with **fitness sharing**. The high selective pressure of $(\kappa + \mu)$-elimination was therefore largely mitigated with the introduction of this diversity promotion scheme. However, to further reduce the selective pressure present in the $(\kappa + \mu)$-elimination, the operator was swapped for the **k-tournament elimination** operator in the final version of the algorithm (along with the same fitness sharing technique).
3. The mutation operator has been changed from swap mutation to **inversion mutation**. Hence, the effect of the mutation operator is constant for rising problem sizes.
4. With the introduction of fitness sharing, along with the inversion mutation operator, **premature convergence** was largely **avoided**.

2.3 The best suggestion

Both groups suggested to modify the simplified edge crossover operator to the 'proper' one. Although this certainly is a useful suggestion, another suggestion by one group is even more useful in my opinion, given that I was going to change the crossover operator in the individual phase anyway. The other suggestion is about changing the mutation operator from swap mutation to inversion mutation. In the group phase part of the project, we hadn't really observed the shortcoming of the swap mutation, which only became clear after their suggestion has been made.

3 Changes since the group phase

1. The simplified edge recombination operator was first replaced with the **proper edge recombination operator**, after which it eventually was replaced by the **order recombination operator** due to massive speed gains. An elaboration on the recombination operators is provided in Section 4.7.
2. **Fitness sharing** has been introduced in the elimination step, as further explained in Section 4.10.
3. A **local search operator** (2-opt) has been introduced to increase the fitnesses of the newly created offsprings (Section 4.9).
4. Simple random initialization has been replaced by **greedy** and **legal initialization**, as further explained in Section 4.4.
5. The mutation operator has been changed from swap mutation to **inversion mutation**. This is more elaborated in Section 4.6.
6. The elimination operator has eventually been changed from $(\kappa + \mu)$-elimination to **k-tournament elimination**, for reasons explained further in Section 4.8.
7. Numerous **optimizations** have been made in almost all parts of the algorithm to significantly increase the execution speed.

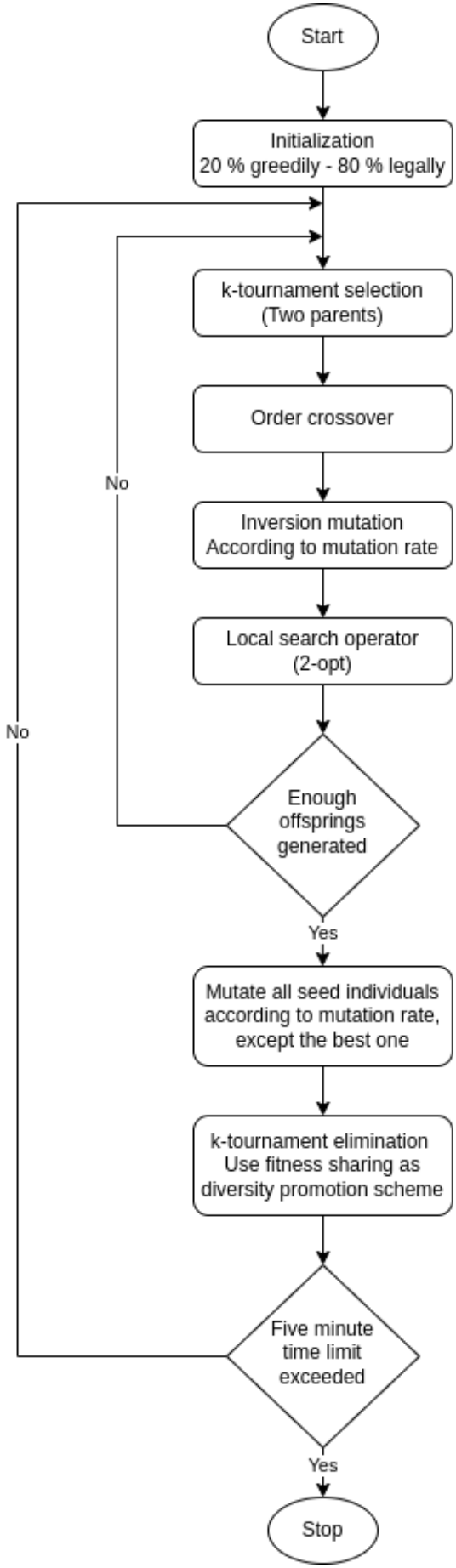
4 Final design of the evolutionary algorithm

4.1 The three main features

1. **Fitness sharing** has been used in the elimination step of the algorithm. This diversity promotion scheme is of crucial importance to **avoid premature convergence**, and hence makes sure that far better solutions can be found, instead of letting all individuals converge to one local minima.
2. By introducing the 2-opt **local search operator**, far better solutions were found more quickly. Without the local search operator, much more iterations were required to find the same fitness values, along with a necessarily larger population. Although this operator is inherently extremely computationally expensive, it turns out to be pivotal for the algorithm. Especially in this operator, optimizations such as making use of **dynamic programming** and using **Numba** were decisive in making the operator computationally feasible.
3. One last crucial improvement is the introduction of the **greedy** and **legally initializations**. Greedy initialization starts from a random node, and chooses the next one according to the smallest distance. The details of this initialization scheme are elaborated in Section 4.4, along with a consideration of the introduced biases. Furthermore, legal initialization simply chooses the next node from a random neighbor that has an existing road between them.

4.2 The main loop

Figure 1: Main loop



4.3 Representation

Possible solutions are represented as **permutations** and are written down in **cycle notation**. For example, the permutation (1423) starts at 1, then goes to 4, then 2, then 3, and returns to 1. An advantage of this notation is that no cycles are present as long as we initialize the representations as a permutation.

This representation is implemented in as a Numpy array with its length equal to the number of cities in the problem. Each element in the array represents a city number.

4.4 Initialization

In the group-phase part of the project, individuals were generated by a **random permutation**, with their size determined from the distance matrix. However, especially for the larger problem sizes, quite a lot of paths were **nonexisting** or **extremely long**. Hence, random initialization of all individuals yielded almost always individuals where none of them represented a valid path.

Two new initialization schemes were introduced, legal and greedy initialization, where the objective of legal initialization is to not create nonexisting paths when initialing an individual. This while not introducing certain biases, such as individuals who take over the population immediately. The objective of greedy initialization, on the other hand, is introducing locally optimal individuals with a high fitness, in a computationally inexpensive way. Here, special care has been taken that the individuals don't introduce high biases and won't take over the population immediately.

4.4.1 Legal initialization

When initializing an individual legally, one makes sure that the generated path **exists**. Therefore, a city is chosen at random, after which the next city in the tour is chosen to be a random one out of all the neighbors with a non-infinite path cost. If, however, no existing neighbors are available anymore as next cities, the whole procedure restarts. The pseudo-algorithm is given in Algorithm 1.

Algorithm 1 Legal initialization

```
while True do
    Choose the initial city at random
    while individual not entirely generated do
        Generate all legal possibilities
        if no legal possibilities then
            Break out of inner loop
        end if
        Choose a random city to be the successor of the previous city
    end while
    if not broken out of inner loop and path from the last city to the first one is not infinity then
        Return newly generated individual
    end if
end while
```

This legal initialization scheme **does not introduce undesirable biases**. In theory, also the most optimal route could have been initialized, and the only bias present is a bias to not have nonexisting paths. I therefore can't think of any disadvantage this initialization scheme entails.

4.4.2 Greedy initialization

Besides the legal initialization scheme, another initialization scheme named greedy initialization is used. The algorithm resembles the legal initialization scheme, with the only change that the successor city is chosen to be the **closest neighbor**, instead of a random legal neighbor. The pseudo-algorithm is given in Algorithm 2.

This initialization scheme **does introduce certain biases**, which could result in some individuals taking over the population immediately. Therefore, one has to take preventions against this bias, by for example only initializing a small fraction of all the individuals with this scheme.

The introduced bias is that all individuals are locally optimal, where in theory the maximum number of different solutions is given by the problem size. Given that it may be difficult to escape local minima, one must consider the usefulness of this initialization scheme.

However, after experimenting with this initialization scheme, it became apparent that **good solutions** for the problem where found much faster, while still **maintaining a lot of diversity**, along with a **smooth convergence**. With only a small portion of the individuals being initialized with this scheme, being totally stuck in a local

Algorithm 2 Greedy initialization

```
while True do
    Choose the initial city at random
    while individual not entirely generated do
        Generate all legal possibilities
        if no legal possibilities then
            Break out of inner loop
        end if
        Choose the closest neighboring city to be the successor of the previous city
    end while
    if not broken out of inner loop and path from the last city to the first one is not infinity then
        Return newly generated individual
    end if
end while
```

minima was not observed, and for me it brought huge advantages, since the given five minutes could now be used to start searching in some way more interesting regions of the search space.

4.4.3 General aspects

The distance matrix can be given in such a way that greedy initialization gets stuck in an **infinite loop**, because greedy initialization may always construct a dead path (due to always taking the closest neighbor), starting from each node. To create a legal path in special cases, it should instead sometimes take sub-optimal paths to a neighbor to not end up in a dead path near the end of the initialization. To prevent the whole algorithm from crashing, a **time constraint** on the initialization of one individual has been introduced. Once an individual takes longer than two second to initialize, simple random initialization of that individual ensues.

An individual also gets assigned a random α value, which represents the probability that the individual will mutate in the mutation step of the algorithm. This way, a suitable mutation rate is determined by **self-adaptivity**.

The initial value of α is given by Formula 1.

$$\alpha = \max(0.04, 0.20 + 0.08 \cdot (X \sim \mathcal{N}(0,\, 1))) \tag{1}$$

After some testing with population sizes, a size of 15 had been chosen. Furthermore, as mentioned in Section 4.4.2, only a fraction of the population should be initialized greedily. Given that the population size is 15, I found that greedily initializing 20 % of the individuals (i.e. 3 individuals) worked out quite well in practice. The remaining 80 % of the individuals are initialized legally.

For large problem sizes, initialization could take up to 10 seconds. Since the initialization of an individual is totally independent of the other individuals, **multiprocessing** has been added to this step, which entailed a factor five speed improvement on a machine with four physical cores (eight virtual cores).

4.5 Selection operators

The **k-tournament selection operator** from the group phase part has been kept. This selection operator is computationally inexpensive, since only k fitness values have to be computed, while this would require μ fitness values in fitness-based methods. Furthermore, sigma-scaled selection would for example not have been an appropriate choice, since the greedy initialization scheme introduces some very good individuals in the population. These individuals would dominate in such a scheme, since their selection probability would be very high.

A k-value of 5 has been chosen after numerous experiments.

4.6 Mutation operators

The mutation operator used for the final implementation is the **inversion mutation**, whereby a random sub-vector is chosen and its order is reversed. The swap mutation operator that was used in the group phase part did not scale well to larger problems, since it only swaps two random locations. That mutation operator, as a consequence, had a relatively even smaller impact on the solution when the problem size increased.

Inversion mutation does not suffer from this scaling problem, since the cities that determine the sub-vector are randomly chosen. Hence, the effect of the mutation operator is constant for rising problem sizes.

Self-adaptivity has been used for the mutation rate, which is hence specific to each individual. It is initialized as in Formula 1, and it changes in crossover as described in Formula 2 and Formula 3.

$$\beta = 2 \cdot (X \sim \mathcal{N}(0,\, 1)) - 0.5 \tag{2}$$

$$alpha = \max(0.04, \alpha_{parent\_1} + \beta \cdot (\alpha_{parent\_2} - \alpha_{parent\_1})) \tag{3}$$

As a last note, **elitism** is used to prevent the best seed individual from mutating.

4.7 Recombination operators

In the group phase part of the project, a **simplified version** of the **edge crossover operator** was used as the recombination operator, for which the process is described in Algorithm 3 [3]. This recombination results in a new path where almost all the edges of the child were present in at least one of the parents. It does however not prioritize edges present in both parents over edges present in a single parent.

Algorithm 3 Simple edge recombination operator [3]

```
Let K be the empty list
let N be the first node of a random parent
while length(K) < length(Parent) do
    Append N to K
    Remove N from all neighbor lists
    if N's neighbor list is not empty then
        let N* be the neighbor of N with the fewest neighbors in its list (or a random one, in case of multiples)
    else
        let N* be a randomly chosen node that is not in K
    end if
    N ← N*
end while
```

This algorithm is very simple and was the weakest part of the genetic algorithm. However, the algorithm still has some desirable features despite its simplicity. Edges present in both parents have a relatively high probability of propagating to the child, so important features are mostly preserved. On the other hand, when the parents are very different, the child will look fairly different from both parents. Hence, this operator moves a bit more to the exploration side then other operators.

The reason this simplified algorithm was implemented, instead of the proper one from Eiben & Smith [1], was due to the belief that the computational cost of this algorithm was (much) lower than the one from Eiben & Smith.

For the individual phase of the project, an analysis was made between order crossover and the proper edge crossover algorithm of Eiben & Smith. After some research, with a lot of contradictory advices, an arbitrary choice has been made to first try out the **proper edge crossover algorithm**.

Implementation wise, quite a lot of effort has been made to catch all the corner cases of the algorithm, along with achieving relatively optimized code. The algorithm was kept a long time thereafter, until it was noticed that for larger problem sizes, crossover took an **extremely long time** (up to 95 % of the total runtime was spend in the edge crossover operator).

Due to this slow execution time, **order crossover** [1] has been implemented as well (Algorithm 5).

This crossover algorithm is inherently much cheaper to calculate and takes only about 5 % of the total execution time in the final algorithm. This is exactly the reason why this crossover operator was eventually used.

In hindsight, one reason for the slow execution time of the edge crossover operator is probably due to the usage of sets in the operator. The edge table was basically one list with sets, where a minus denoted a double entry. Sets were used because it was desirable to check quickly if an edge was present in the edge table. However, since for each element, maximum four edges could be present, lists would probably have sufficed. Given that also quite some bookkeeping was required with the sets, one more point can be made for using lists (e.g. deleting a positive entry if it occurred for the second time, to insert it afterwards with a minus in front of it).

Another reason for the big performance gap is the fact that the order crossover operator was able to use **Numba** for compiling the Python code and running it way faster, by using the decorator $@jit(nopython = True)$. This because the order crossover operator only uses operations on Numpy arrays (which Numba handles perfectly well), while Numba threw hundreds of compile errors in the edge crossover implementation, because Numba (in the $nopython = True$ mode) couldn't create new Numpy arrays, had difficulties with working on sets, and wasn't able to infer the $dtype$'s most of the time.

Algorithm 4 'Proper' edge recombination operator [1]

```
Construct the edge table
Pick an initial element at random and put it in the offspring
while the offspring is not entirely constructed do
    Set the variable current_element = entry
    Remove all references to current_element from the table
    if there is a common edge then
        Pick that to be the next one
    else
        Pick the entry in the list which itself has the shortest list (split ties at random)
    end if
    if an empty list is reached then
        if the other end has not yet been examined then
            Examine the other end for extension
        else
            Choose a new element at random
        end if
    end if
end while
```

Algorithm 5 Order crossover operator [1]

Choose two crossover points at random
Copy the segment between them from the first parent into the offspring
Starting from the second crossover point in the second parent, copy the remaining unused numbers into the child in the order they appear in the second parent, wrapping around at the end of the list.

The mutation rate of the offspring is determined with Formula 2 and Formula 3..

4.8 Elimination operators

For a long time, the **(κ + μ)-elimination** from the group phase part was kept in the algorithm. However, for the smaller problem sizes, it was noted that the population converged extremely quickly, even with the fitness promotion scheme present (as further discussed in Section 4.10). After some research, it became apparent that the (κ + μ)-elimination operator actually puts quite a lot of selective pressure. A **k-tournament operator**, in contrast, can mitigate this selective pressure, hence the (κ + μ)-elimination operator has been exchanged for the k-tournament operator.

To combine the k-tournament operator with the fitness sharing operator, Algorithm 6 is for the k-tournament operator (along with some preparatory computations), while Algorithm 8 is invoked each time for the fitness sharing diversity promotion scheme itself.

Algorithm 6 Elimination [1]

```
Calculate the fitnesses for all the individuals
Add the best individual to the list 'survivors'
for i from 0 to λ − 2 do
    Calculate the new fitness values, based on fitness sharing (Algorithm 8)
    for j from 0 to k_elimination − 1 do
        Randomly sample a fitness value ('fit') from the new list of fitness values
        if 'fit' is the lowest fitness found then
            Store the index of the individual that corresponds to this fitness under 'idx'
        end if
    end for
    The new survivor is the individual located at 'idx'
    Append the new survivor to the 'survivors' list
    Delete the survivor from the list containing all the individuals
end for
```

After numerous experiments, a k-value of 8 has been chosen.

4.9 Local search operators

The **2-opt local search operator** has been implemented, which swaps every two possible edges in a given cycle. In a first version of this algorithm, the fitness was recalculated for every possible neighbor of the given individual, which entailed an unacceptable high computational cost, especially for the larger problem sizes. After some investigation, patterns were detected in the computation of the fitness. Hence, instead of recalculating the fitness for every neighbor, some kind of **dynamic programming** approach was undertaken. For every individual, there is a sort of preprocessing step, whereby so-called 'cumulatives' are created. These cumulatives capture the path length from the first city to that corresponding city in the cumulative array. The same process applies for the calculation of the path length from the last city to the corresponding city in the array (i.e. in reverse order, whereby the return cost of the last city to the first city is also incorporated). It is clear that the calculation of these cumulatives is done in $O(N)$, where N is the number of cities in the problem size.

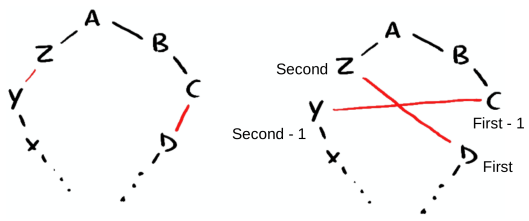
Now, calculations of fitnesses of individuals are simply a matter of bookkeeping. The process is explained in Algorithm 7.

Algorithm 7 Local search operator

Let the best fitness be the fitness of the original individual
Let the best combination be (0, 0)
Build cumulatives
for first from 1 to (length - 3) **do**
  $fit_{first\ part} = forward\_cumulative[first - 1]$
  $fit_{middle\ part} = 0.0$
  for second from (first + 2) to (length - 1) **do**
    $fit_{middle\ part}$ += $distanceMatrix[order[second - 1]][order[second - 2]]$
    $fit_{last\ part} = backward\_cumulative[second]$
    $first\_bridge = distanceMatrix[order[first - 1]][order[second - 1]]$
    $second\_bridge = distanceMatrix[order[first]][order[second]]$
    $fitness = fit_{first\ part} + first\_bridge + fit_{middle\ part} + second\_bridge + fit_{last\ part}$
    if $fitness < best\_fitness$ **then**
      Let the new best combination be (first, second)
      Let the new best fitness be the newly calculated fitness
    end if
  end for
end for
Swap the order of the individual from the best first to the best second, obtained from the best combination.

As further illustrated in Figure 2, the first part of the tour is simply the same as the previous iteration, with one extra cost added from 'first - 1' to 'first'. The same reasoning applies for the last part of the tour, where in this case the total cost decreases each time by the cost from 'second - 1' to 'second'. Finally, also the middle part can be build up in an analogous way. This way, the total cost of the 2-opt local search algorithm is only $O(N^2)$, where N denotes the total number of cities.

Figure 2: Local search operator (2-opt) [2]



It should also be noted that by using Numba with the command $@jit(nopython = True)$ above the method declarations, the local search operator runs **745 times as fast**. Numba can make these huge improvements due to the compilation of these methods, where especially the loops can be exploited.

4.10 Diversity promotion mechanisms

The used diversity promotion scheme is **fitness sharing elimination**, which changes the fitnesses of the individuals that are in the σ-neighborhood of the already chosen survivors. The fitness sharing elimination operator is explained in Algorithm 8.

Algorithm 8 Fitness sharing algorithm

for i from 0 to length of population - 1 **do**
  for j from 0 to length of survivors - 1 **do**
    distances[i][j] = distance_from_to(population[i], population[j])
  end for
end for
Calculate 'shared_part' as $1 - (distances/\sigma)^{alpha}$
Only keep the values in 'shared_part' if the distance is less than σ
Sum 'shared_part' along the first axis ('sum_shared_part')
Calculate the shared fitness values as the original fitness values times 'sum_shared_part'

The sub-method 'distance_from_to' calculates the distance between two individuals, measured by the number of **common edges** between the two individuals. To do this efficiently, the edges of each individual are calculated and stored in a set, at the moment of initialization. For measuring the distance between two individuals, the **intersection** between the sets is calculated.

Calculating the intersection of a lot of individual pairs over and over, turned out to be quite computationally expensive. An improvement that has been made is to store all the calculated distances in a **hashmap**, which gave a decent improvement, given that quite some individuals stay in the population for more than just one iteration. A precaution against thrashing has been taken, by simply emptying the hashmap if the memory usage of the system exceeds 95%.

4.11 Stopping criterion

Not a lot of effort has been put in implementing a stopping criterion, since all the larger problems stayed converging after running for five minutes. Given that even if the best fitness stayed for a very long time fixed, it occurred that due to a well chosen mutation/crossover operation, suddenly the algorithm can proceed even further. Hence, the stopping criterion is simply the time limit of five minutes.

4.12 Parameter selection

The **population** and **offspring size** have largely been determined by the computational cost from the algorithm. The largest computational cost per iteration for large problem sizes is the fitness sharing elimination step. This since the matrix with the number of common edges between all the individuals and the survivors grows quadratically, and the computation of one entry in this matrix also grows linearly. Hence, the computational costs grew too large if a lot more than 15 individuals in the population (and 15 more individuals as offsprings) were taken. Much less than 15 individuals is, naturally, also not desirable, since an evolutionary algorithm depends on having some diverse individuals.

The **k-tournament parameters** were determined by a **hyperparameter search**, where values ranging from 2 to 10 have been tried. Given that these two parameters are highly correlated, a grid-search or random search were required. To make the hyperparameter search feasible, a random search for these values was undertaken, which yielded a k-tournament value of selection equal to 5, and a value of 8 for elimination.

When the matrix with the number of common edges was printed, it became apparent that either a lot of entries were either the maximal problem size, or zero. Hence, after some experimentation, a σ value of 50% of the problem size has been taken, with 0.25 as α value. This low alpha is in my opinion also better, given that really 'close' solutions should be penalized way more than solutions with still quite some different edges.

1. The population size = 15
2. The offspring size = 15
3. The k-tournament parameter of the selection operator = 5
4. The k-tournament parameter of the elimination operator = 8
5. The α-value of fitness-sharing = 0.25
6. The σ-value of fitness-sharing = half of the problem size

4.13 Other considerations

As discussed in Section 4.10, quite a speedup was attained by storing the distances between individuals in a hashmap. For the same reason, a hashmap to store the fitness values of all the individuals has been introduced. Since in each iteration, it is known which individuals are removed from the population (if mutation is applied, local search yields a new individual, or individuals are killed in the elimination step), their value can easily be

removed from the hashmap as well. This way, the size of the hashmap stayed always the same, which is more performant, since there is no restart after the memory level exceeds its threshold, and there is no time wasted for the garbage collector that needs to kick in.

5 Numerical experiments

5.1 Metadata

These experiments were conducted on an Intel Core i7-6700HQ CPU, with a clock frequency of 3.60GHz and 8 virtual cores. The systems contains 16 GB of main memory, and Python version 3.8 was used for the tests. The used hyperparameters are summarized in Section 4.12.

All convergence graphs were plotted semi logarithmically, since the changes at the start are much bigger than the changes near the end of the algorithm.

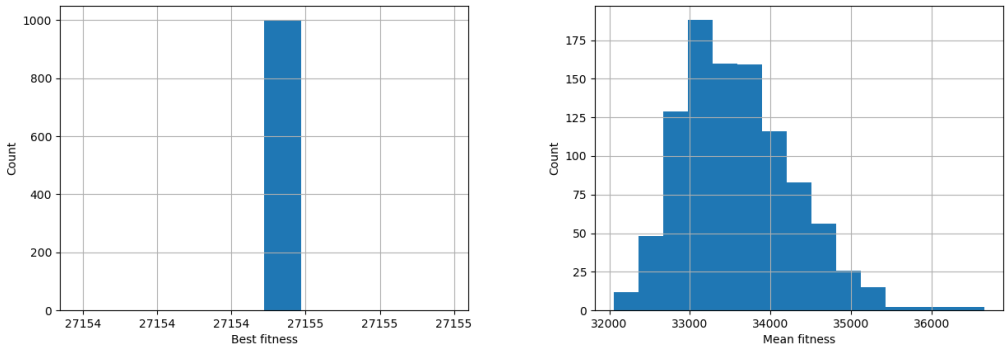
5.2 tour29.csv

The best tour found had a fitness of 27154.5, and has the following tour sequence:

[0,1,4,7,3,2,6,8,12,13,15,23,24,26,19,25,27,28,22,21,20,16,17,18,14,11,10,9,5]

The convergence graph is shown in Figure 4a.

Here, convergence of the best fitness happens extremely quickly. The mean fitness, however, does not converge, which means that the population stays diverse due to the fitness promotion elimination scheme.



(a) Best Fitness

(b) Mean fitness

Figure 3: Histograms for the best and mean fitness, after 1000 runs on tour 29

As follows from Figure 3, the (probably optimal) fitness value of 27154.5 has been found each time. Furthermore, the mean fitness possesses some variation due to the diversity promotion scheme. It is never equal to the best fitness value, which means that diversity is always preserved. Note that for this experiment, a time limit of 10 seconds per run was used to make the computation feasible.

The mean value of the thousand runs for the mean fitnesses is equal to 33551.6, with a value of 657.6 for the standard deviation,

5.3 tour100.csv

The best tour found had a fitness of 219948.4, and has the following tour sequence:

[0,83,72,78,40,67,5,4,21,63,14,93,56,82,11,70,76,64,81,54,34,44,99,55,18,77,66,13,35,85,94,10,89,7,32,19,38,95,80,30,28,23,22,86,

The convergence graph is shown in Figure 4b.

The algorithm clearly performs great, by quickly jumping to the most interesting regions of the search space, after which the best fitness stays converging until the five minute time constraint is passed. As can be seen from the mean fitness, the population stays diverse and hence has the ability to keep exploring for better solutions.

It is difficult to make an estimate of how close this solution is to the most optimal solution, since only a heuristic value was known, which this algorithm clearly outperforms.

5.4 tour500.csv

The best tour found had a fitness of 110814.2, and has the following tour sequence:

[0,167,223,386,235,199,172,479,311,405,266,140,399,108,432,265,198,231,312,454,205,61,323,383,291,57,420,482,193,451,379,6

The convergence graph is shown in Figure 4c.

The algorithm clearly performs great again, by quickly jumping to the most interesting regions of the search space, after which the best fitness stays converging until the five minute time constraint is passed. As can be seen from the mean fitness, the population stays diverse and hence has the ability to keep exploring for better solutions.

It is difficult to make an estimate of how close this solution is to the most optimal solution, since only a heuristic value was known, which this algorithm clearly outperforms.

5.5 tour1000.csv

The best tour found had a fitness of 194955.2, and has the following tour sequence:

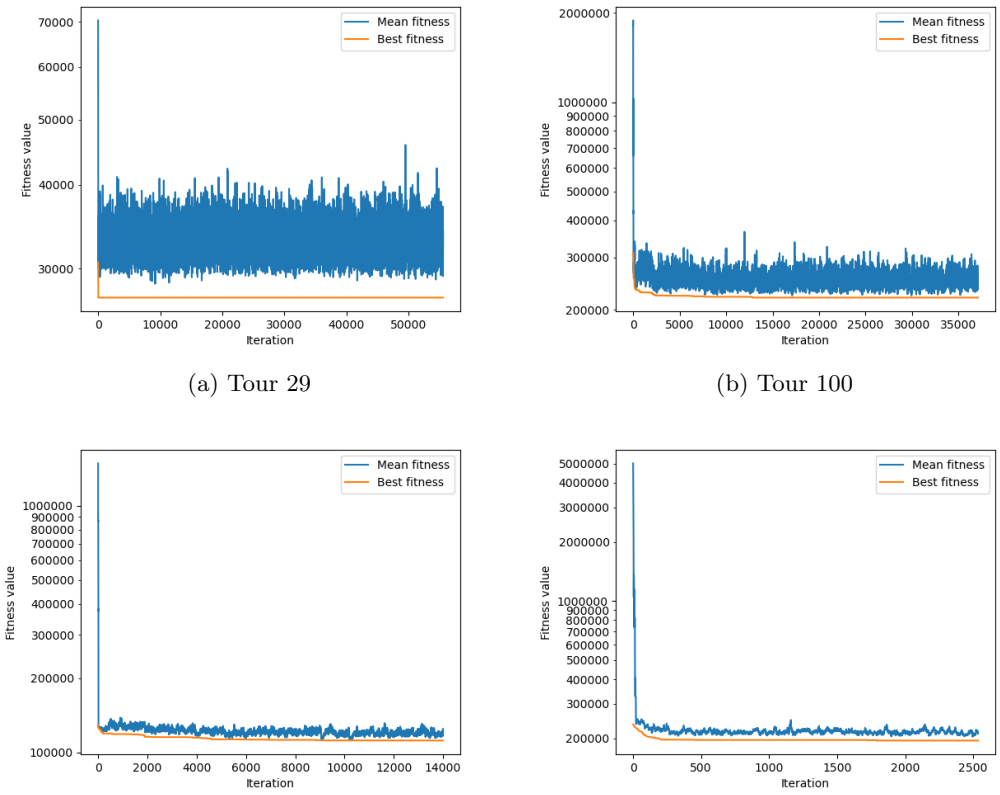
[0,846,28,344,518,762,924,390,541,18,155,735,965,810,318,938,971,152,215,398,268,497,992,291,165,650,470,709,325,194,691,9

The best fitness again stays converging until the five minute time limit has been exceeded. Once more can be derived from the mean fitness that the population stays diverse.

It is difficult to make an estimate of how close this solution is to the most optimal solution, since only a heuristic value was known, which this algorithm clearly outperforms.

5.6 Convergence graphs

The convergence graph is shown in Figure 4d.



(a) Tour 29

(b) Tour 100

(c) Tour 500

(d) Tour 1000

Figure 4: Convergence graphs of benchmarks (Semi logarithmically)

6 Critical reflection

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

1. By using a population-based metaheuristic, a tradeoff can be made between **exploration** and **exploitation**. This tradeoff turned out to be instrumental in finding good solutions for the travelling salesman problem. Pure random search is computationally way too expensive (it would even take an infinity before a 'decent',

simply greedy solution would have been found). Pure local optimizers, on the other hand, would quickly converge to one suboptimal solution, which would most likely also be relatively far from the most optimal solution. Evolutionary algorithms provide a way to find good sub-optimal solutions in a decent amount of time.

2. Evolutionary algorithms are in principle relatively easy to **parallelize**, since the population can explore the search space concurrently in many directions. Although in this project, not a lot of parallelization had been used (only in the initialization step), it would probably be the next most beneficial task to undertake. Trying to use multiprocessing has been undertaken, but the execution time when using multiprocessing was more than a thousand times slower, compared to simple sequential execution on one CPU core. This probably came due to a lot of interprocess communication (IPC), since for example the list of individuals was shared, which resulted in a lot of locks and semaphores, that clearly dominated the execution time.
3. A last thing that struck me about evolutionary algorithms, is the fact that they in essence optimize a function **without using derivatives** (only by using the fitness values, leaving aside the local search operator for the moment). Especially for non-differentiable functions may an evolutionary algorithm offer salvation.

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

1. Evolutionary algorithms work well, as long as they are **carefully designed** with reasonable parameters. For example, take a population size of 100 instead of 15, and the algorithm struggles to make some iterations. This of course due to the fitness sharing elimination, which has a cubic complexity in terms of the population (and offspring) size. Furthermore, a lot of other parameters have to be chosen to obtain reasonable performance. If for example the k-tournament parameters (from selection and elimination) are not considerably chosen, either the population converges immediately, or the overall fitnesses of the individuals hardly increase. This, in the end, comes at no surprise, since, according to the no-free-lunch theorem, there cannot exist any algorithm for solving all problems that is generally superior to any competitor.
2. The representation used in this project clearly yields valuable offsprings after crossover. If one reasons about the representation, it is apparent that subvectors of the parents are probably relatively close to each other, since they come after each other in the tour of the solution. When doing crossover, these **features** may now be combined in a more optimal way, so the offsprings have clearly a relatively high chance on having a low fitness value as well. However, for a lot of other problems, such representations and crossover operators are not that clear (i.e. features can't be extracted from the parents), and hence evolutionary algorithms can't offer a lot to these problems.
3. Evolutionary algorithms are **computationally very expensive**. Since a lot of exploration is undertaken, inherently a lot of meaningless individuals are constructed while searching.

6.3 Describe the main lessons learned from this project. Do you believe evolutionary algorithms are appropriate for the problem studied in this project? Why (not)? What surprised you and why? What did you learn from this project?

I think that the travelling salesman problem fits perfectly for the project related to this course. Finding an exact solutions is of course infeasible, hence the usage of metaheuristics becomes necessary, and they prove themselves to work extremely well. Considering that a lot of real-world applications require good approximations of the optimal solutions, it is obvious that these metaheuristics (and evolutionary algorithms in particular) are an essential tool to possess.

As a side-effect of this course, I would like to note that my Python coding skills are somewhat deepened, especially for computationally expensive tasks. I learned to appreciate the value of libraries such as **Numba**, that can speed up Python code massively by compiling the code. Surely, not all operations are allowed, but given that the Numba community is evolving rapidly, I will keep an eye on it.

One kind of drawback of evolutionary algorithms are the way you have to test certain improvements. Due to the inherent **randomness** of evolutionary algorithms, quite a lot of **variation** can be observed between separate runs. Certainly if improvements of hyperparameters only yield a relatively small difference, the experiments have to be repeated several times before the best parameter becomes apparent.

References

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

[2] Nick Vannieuwenhoven. Local Search Operators. Technical report, 2021.

[3] Darrell Whitley, Timothy Starkweather, and Daniel Shaner. The Traveling Salesman and Sequence Scheduling: Quality Solutions Using Genetic Edge Recombination. page 18.