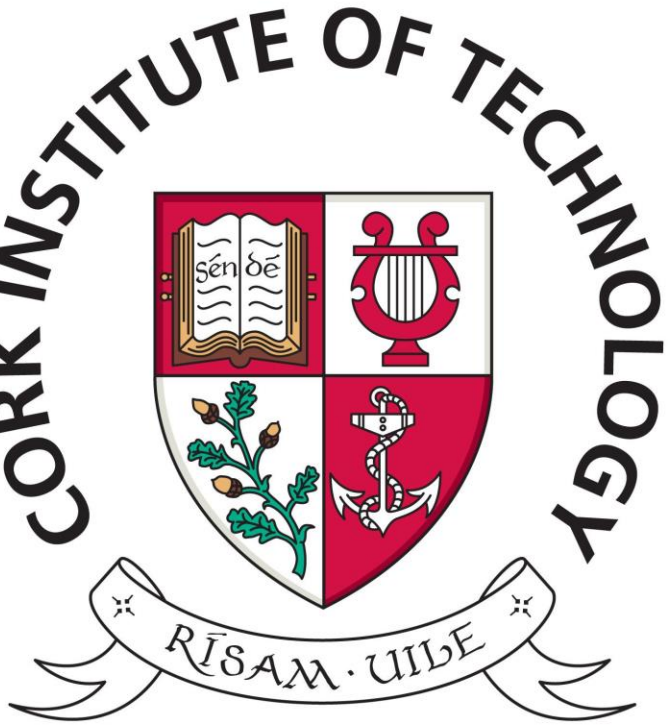


Mike Leske
R00183658

COMP9058 - Metaheuristic Optimization
Assignment 1 - Complexity & Genetic Algorithms

Table of Contents

1 NP Completeness

1.1 SAT to 3SAT Reduction

Note: Using d) as last digit of my student id is 8.

$F = (q_1 \vee q_4) \wedge (\neg q_1 \vee q_2 \vee q_3 \vee \neg q_4 \vee \neg q_5)$

In order to convert a formula F in SAT to 3SAT, each clause in F has to be converted individually.

$C_1 = (q_1 \vee q_4)$

Case: K = 2
=> 1 extra variable: y_1
=> 2 clauses

$Z_{11} = (q_1 \vee q_4 \vee y_1)$
$Z_{12} = (q_1 \vee q_4 \vee \neg y_1)$
$Z_1 = Z_{11} \wedge Z_{12} = (q_1 \vee q_4 \vee y_1) \wedge (q_1 \vee q_4 \vee \neg y_1)$

$C_2 = (\neg q_1 \vee q_2 \vee q_3 \vee \neg q_4 \vee \neg q_5)$

Case: K = 5
=> 2 extra variables: y_2 , y_3
=> 3 clauses

$Z_{21} = (\neg q_1 \vee q_2 \vee y_2)$
$Z_{22} = (\neg y_2 \vee q_3 \vee y_3)$
$Z_{23} = (\neg y_3 \vee \neg q_4 \vee \neg q_5)$
$Z_2 = Z_{21} \wedge Z_{22} \wedge Z_{23}$
$= (\neg q_1 \vee q_2 \vee y_2) \wedge (\neg y_2 \vee q_3 \vee y_3) \wedge (\neg y_3 \vee \neg q_4 \vee \neg q_5)$

The final 3SAT formula F' is

$F' = Z_1 \wedge Z_2 = (q_1 \vee q_4 \vee y_1) \wedge (q_1 \vee q_4 \vee \neg y_1) \wedge (\neg q_1 \vee q_2 \vee y_2) \wedge (\neg y_2 \vee q_3 \vee y_3) \wedge (\neg y_3 \vee \neg q_4 \vee \neg q_5)$

Define a solution for F':

q_1 = True q_2 = True q_3 = False q_4 = False q_5 = False y_1 = False y_2 = False y_3 = False	$F' = Z_1 \wedge Z_2 =$ $(q_1 \vee q_4 \vee y_1) \wedge$ $(q_1 \vee q_4 \vee -y_1) \wedge$ $(-q_1 \vee q_2 \vee y_2) \wedge$ $(-y_2 \vee q_3 \vee y_3) \wedge$ $(-y_3 \vee -q_4 \vee -q_5)$	$F' = Z_1 \wedge Z_2 =$ $(T \vee F \vee F) \wedge$ $(T \vee F \vee T) \wedge$ $(F \vee T \vee F) \wedge$ $(T \vee F \vee F) \wedge$ $(T \vee T \vee T)$ $= T$

Verify that solution for F' also satisfies F:

q_1 = True q_2 = True q_3 = False q_4 = False q_5 = False	$F = (q_1 \vee q_4) \wedge$ $(-q_1 \vee q_2 \vee q_3 \vee -q_4 \vee -q_5)$	$F = (T \vee F) \wedge$ $(F \vee T \vee F \vee T \vee T)$ $= T$

We conclude that the assignment of $q_1 = q_2 =$ **True** and $q_3 = q_4 = q_5 =$ **False** satisfies both F and F'. Therefore, we successfully reduced SAT to 3SAT.

1.2 3Col Graph

Note: Using first two clauses from F' as the first letter of my first name is M.

Convert subclauses from F' to a 3Col graph.

Subclauses:

$$Z_1 = (q_1 \vee q_4 \vee y_1) \wedge (q_1 \vee q_4 \vee -y_1)$$

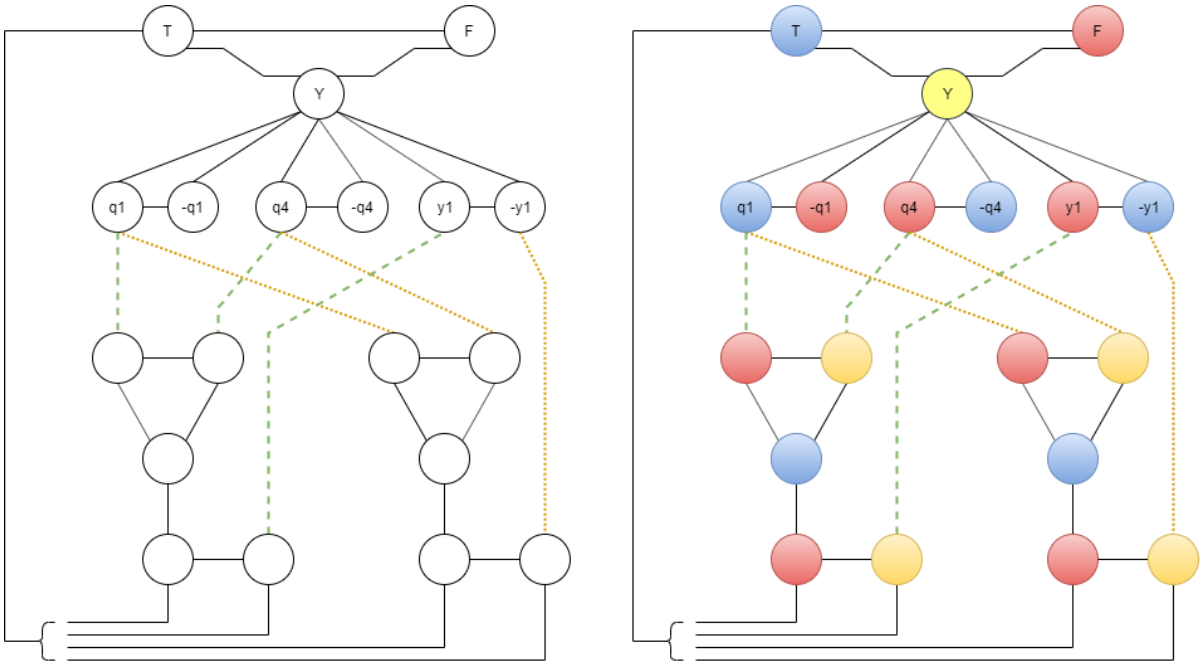


Figure 1 3Col graph and 3Col coloring for selected subclauses

2 Genetic Algorithms

To complete this part of Assignment 1 the students were provided with a general Genetic Algorithm (GA) framework based on which we will implement additional methods to enhance the GA framework with the following functionalities:

- Initial population
  - Heuristic initial population (based on nearest neighbor)
- Crossover
  - Uniform order-based Crossover
  - Partially Mapped Crossover (PMX)
- Mutation
  - Reciprocal Exchange Mutation
  - Inversion Mutation
- Selection
  - Stochastic Universal Sampling (SUS)

2.1 Overview

In this assignment we use Genetic Algorithms including the aforementioned components to optimize solutions to the Travelling Salesman Problem (TSP), where a list of cities is provided including (x, y) coordinates, based on which the distance between individual cities can be calculated. The main objective of TSP is finding a tour through all cities with the minimum overall distance. No city must be visited twice; except the start city, to which the travelling salesman must return to complete his tour.

In terms of Genetic Algorithms any valid tour is considered an **individual** within a **population** of multiple individuals. The sequence of cities each individual is defined by is known as the individuals' **chromosome**. The elements of the chromosome, i.e. city id's, are known as **genes**.

TSP is a category NP-Hard problem, because we cannot proof if a given solution is indeed the shortest route in polynomial time without solving the TSP itself which is known to require exponential time.

2.2 Problem Instances

As the first letter of my surname is L, I will use the following TSP instances:

- inst-4.tsp
- inst-6.tsp
- inst-16.tsp

2.3 Implementation of GA methods

The following sections elaborate on some of the most essential aspects of Genetic Algorithms that are to be used during the assignment experiments. Special focus is provided on those methods to be implemented by the students.

1. Objective Function
2. Population Initialization
3. Selection
4. Crossover
5. Mutation
6. Replacement & Elitism

2.3.1 Objective Function & Fitness Function

The objective of the Travelling Salesman Problem is minimizing the total cost of the route through all cities. Therefore, the cost of a solution, or better an individual's chromosome, can be defined as the total distance that one has to travel on straight lines between subsequent cities in a given chromosome.

As Genetic Algorithms are typically used for maximization problems, we define an individual's fitness F as the inverse of its objective Function: F = 1/objectiveFunction. [2]

2.3.2 Population Initialization

For a GA to be able to evolve, an initial population is required out of which a first generation of offsprings will be generated. For the experiments to be executed, two methods of initial population creation are to be considered:

1. Random Initial Population
2. Heuristic Initial Population

The GA framework already provided a method to create random individuals for the initial population.

2.3.2.1 Random Initial Population

With respect to the TSP a naive approach is to bring the list of cities to be visited in a random order and add this chromosome to the initial population. For a population of size P, P random routes through all cities would be generated.

2.3.2.2 Heuristic Initial Population

For the TSP the nearest neighbor heuristic method for creating the initial population of the GA will be used for some experiments of this assignment. This heuristic method considers "some" problem/domain specific knowledge, for example the Euclidean distance between the cities. Such a distance-based heuristic can be used as the coordinates of the cities are provided. In addition, knowing that the distance between any two cities on a map is no shorter than the straight connection between them, the nearest neighbor heuristic method represents an admissible heuristic.

In order to create a single heuristics-based individual the following steps are executed:

1. Select a random city (gene) from the total pool of cities to visit
2. Add selected city to solution path (chromosome). Remove city from pool.
3. From all remaining cities (genes) in pool, select the city which is closest to the last city selected.
4. Go to 2 until all remaining cities have been added to the solution path.

To generate a whole initial population using this heuristic, we loop over the outlined procedure `self.popSize` times.

```
    def heuristicPopulation(self):
        def load_data(f):
            count = np.loadtxt(f, delimiter=' ', max_rows=1)
            cities = np.loadtxt(f, delimiter=' ',
                                dtype={'names': ('id', 'x', 'y'),
                                       'formats': ('int', 'float', 'float')},
                                skiprows=1)
            return count, cities

        count, cities = load_data(self.fName)

        for _ in range(0, self.popSize):
            idx = int(random.randint(1, count))
            not_visited = cities[cities['id'] != idx]['id'].tolist()
            tsp_list = [idx]

            while not_visited:
                cur_x = cities[cities['id'] == idx]['x']
                cur_y = cities[cities['id'] == idx]['y']
                remaining_cities = cities[np.isin(cities['id'], not_visited)]
                distance = np.array([
                    remaining_cities['id'],
                    np.around(np.sqrt(np.power(remaining_cities['x'] - cur_x, 2) +
                              np.power(remaining_cities['y'] - cur_y, 2)))
                ])
                tsp_list.append( distance[0][distance[1].argmin()].astype(int) )
                not_visited.remove(distance[0][distance[1].argmin()])
                idx = int(distance[0][distance[1].argmin()])

            individual = Individual(self.genSize, self.data, _genes=tsp_list)
            individual.computeFitness()
            self.population.append(individual)
            self.updateBest(individual)

        print ("Best initial sol : ", self.best.getFitness())
        print ("Best initial cost: ", self.best.getCost())
```

2.3.3 Selection

Selection methods are an important component of Genetic Algorithms and specify how parent chromosomes from a population are selected for mating. Once the objective and fitness functions for a population have been properly defined, the selection process should give especially fit individuals of a population higher chances of participating in the mating process for creating offsprings. But at the same time, also the worst individuals of a population should have some probability of being selected for mating instead of being ignored. The underlying idea is that an offspring of two very fit parents may result in an even fitter individual. [1] uses the term “selection pressure” for improving the overall population by giving higher mating chances to fitter individuals.

2.3.3.1 Random Sampling

With Random Sampling all individuals are considered equal when selecting parents for mating. Each individual's fitness is completely ignored so that each individual has the same odds of being selected for mating. Every time two parents are selected for mating, two random individuals from the population are selected. Therefore, in a population of size K, each individual of the population is chosen with probability 2/K when two mating partners are chosen.

Random Sampling was already implemented as part of the GA framework provided.

2.3.3.2 Stochastic Universal Sampling (SUS)

Stochastic Universal Sampling (SUS) picks up the idea of "selection pressure" by giving especially fit individuals higher chances of being selection for mating, but also ensuring that worse individuals have their odds.

The process works as follows:

1. SUS normalizes the individuals' fitness by dividing each individual's fitness by the total population fitness, so that the sum of all normalized fitnesses adds up to 1. This results in especially fit individuals occupying a bigger range in the interval [0,1] relative to worse individuals.
2. Let ***r*** be a random uniform variable in the range [0, 1/ƛ], where **ƛ** specifies the intended number of individuals to be promoted into a mating pool.
3. Iterate **ƛ** times with step size ***r*** through the list of normalized fitnesses. If the step ends in the normalized fitness range of individual **S**, add **S** to the list **L** of potential mating partners. Fit individuals will likely be added multiple times as their fitness is expected to be n times ***r***.
4. Select two random individuals from **L**.

One should note that in a population of roughly equally fit individuals, SUS effectively represents a random selection.

```
def stochasticUniversalSampling(self):
    """
    Your stochastic universal sampling Selection Implementation

    Algorithm implemented based on pseudocode from: [3] - Slide 41
    """
    sortedPop = sorted(self.population, key=lambda x: x.getFitness(), reverse=True)

    totalFitness = 0
    for ind in sortedPop:
        totalFitness += ind.getFitness()

    weightedFitness = [ ind.getFitness()/totalFitness for ind in sortedPop ]
    weightedPool = []

    r = random.uniform(0, 1/self.popSize)
    idx = 0
    while len(weightedPool) < self.popSize:
        while r <= weightedFitness[idx]:
            weightedPool.append(sortedPop[idx])
            r += 1/self.popSize
        r -= weightedFitness[idx]
        idx += 1
```

```
        indA = weightedPool[ random.randint(0, self.popSize-1) ]
        indB = weightedPool[ random.randint(0, self.popSize-1) ]
        return [indA, indB]
```

A debug of SUS on a small toy TSP instance is documented in Appendix A.

2.3.4 Crossover

After two individuals have been selected for mating, a crossover operation generates 1 or 2 offsprings, which each inherit genetic input from either parents' chromosome. Crossover is the major operation when creating the next generation of a population by recombining existing chromosomes from a given population.

For optimization problems based on permutation representation like the Travelling Salesman Problem the crossover operation must carefully ensure the offsprings represent domain specific valid solutions. Simple crossover operations like 1-point or n-point crossover are not suitable for this problem. We instead use 2 crossover methods that ensure no single city will exist twice in the offspring's chromosome:

1. Uniform order-based Crossover
2. Partially Mapped Crossover

2.3.4.1 Uniform order-based Crossover

With Uniform order-based Crossover typically two offsprings are created. A random common mask is created that defines which genes of the parent chromosomes are copied into one of the childs' chromosomes at exactly the same position, e.g. masked genes of parentA are copied into childA and masked genes of parentB are copied into childB.

Next, the unassigned genes in each child's chromosome are filled with genes from the **other parent in the order they appear**, excluding those genes that the child originally inherited from the first parent. This way, genes that were originally close in the second parent's chromosome remain relatively close inside the child's chromosome after crossover.
One could implement a `crossoverRate` parameter which acts as a threshold for which genes from a given parent are copied into the related child. The code below simply uses a 0.5 threshold.

```
    def uniformCrossover(self, indA, indB):
        """
        Your Uniform Crossover Implementation
        """
        childA = []
        childB = []

        tmpA = indA.genes.copy()
        tmpB = indB.genes.copy()

        mask = [round(random.random()) for i in range(self.genSize)]

        if self.offsprings == 1:
            for x in range(len(mask)):
                if mask[x]:
                    childA.append(indA.genes[x])
                    tmpB.remove(indA.genes[x])
                else:
                    childA.append(None)
```

```
            for x in range(len(mask)):
                if not mask[x]:
                    childA[x] = tmpB.pop(0)
        else:
            for x in range(len(mask)):
                if mask[x]:
                    childA.append(indA.genes[x])
                    childB.append(indB.genes[x])
                    tmpA.remove(indB.genes[x])
                    tmpB.remove(indA.genes[x])1111
                else:
                    childA.append(None)
                    childB.append(None)

            for x in range(len(mask)):
                if not mask[x]:
                    childA[x] = tmpB.pop(0)
                    childB[x] = tmpA.pop(0)

        return childA, childB
```

A debug of Uniform order-based Crossover on a small toy TSP instance is documented in Appendix A.

2.3.4.2 Partially Mapped Crossover (PMX)

Partially Mapped Crossover (PMX) is another crossover operation that, like Uniform order-based Crossover, is tailored to create 2 offsprings while also ensuring permutation optimization specific aspects.

PMX randomly chooses a sequence of consecutive genes in the parents' chromosomes that are copied into a child: while selected genes from parentA are copied into childB, the same chromosome locations from parentB are copied into childA. The same randomly selected sequences of consecutive genes are kept for mapping purposes.

Next, the unassigned genes in each child's chromosome are filled with genes from the **other parent in the order they appear**, unless this gene is already included in the child's chromosome. In that case the before-mentioned mapping between parentA and parentB genes is used for looking up the original parent's gene at the position that now collides with the other parents' gene. This lookup might result in multiple recursive operations until the mapping reports an unassigned gene.

This way, genes that were originally close in either parents' chromosome remain relatively close inside the child's chromosome after crossover.
One could implement a `crossoverRate` parameter which regulates the distance between the start and stop genes defining the gene sequence used for block swap.

```
    def pmxCrossover(self, indA, indB):
        """
        Your PMX Crossover Implementation
        """
        childA = [None] * self.genSize
        childB = [None] * self.genSize

        idx1, idx2 = sorted(random.sample(range(0, self.genSize), 2))
        mapA = childA[idx1:idx2+1] = indB.genes[idx1:idx2+1].copy()
        mapB = childB[idx1:idx2+1] = indA.genes[idx1:idx2+1].copy()

        def geneMapping(child, gene, m1, m2):
            mapping = m2[m1.index(gene)]
            while mapping in child:
                mapping = geneMapping(child, mapping, m1, m2)
            return mapping

        if self.offsprings == 1:
            for idx in range(len(childA)):
                if not childA[idx]:
                    if not indA.genes[idx] in childA:
                        childA[idx] = indA.genes[idx]
                    else:
                        childA[idx] = geneMapping(childA, indA.genes[idx], mapA, mapB)
        else:
            for idx in range(len(childA)):
                if not childA[idx]:
                    if not indA.genes[idx] in childA:
                        childA[idx] = indA.genes[idx]
                    else:
                        childA[idx] = geneMapping(childA, indA.genes[idx], mapA, mapB)

                    if not indB.genes[idx] in childB:
                        childB[idx] = indB.genes[idx]
                    else:
                        childB[idx] = geneMapping(childB, indB.genes[idx], mapB, mapA)

        return childA, childB
```

A debug of PMX on a small toy TSP instance is documented in Appendix A.

2.3.5 Mutation

After the crossover operation created 1 or more offsprings, typically a mutation operation is performed on each offspring individually. In contrast to the crossover operation, which recombines available genetic information from the parents, the mutation operation has the power to introduce new information into the population for most optimization problems based on binary or discrete representation. For optimization problems based on permutation representation like the TSP, the mutation operation has to account for domain specific constraints that would break the solutions validity, such as introducing a new city ID that is not part of the original optimization problem. Therefore, mutation operations for permutation representation mainly rely on:

1. Swapping individual genes of a potential solution
2. Inverting a sequence of genes

According to [1] the design and usage of mutation operations should take into consideration the following aspects:

1. **Ergodicity**: The mutation operator is technically able to reach any valid solution within the search space.
2. **Validity**: The mutation operator should as much as possible consider and respect domain specific constraints.
3. **Locality**: The mutation operator should allow to be applied to a controllable range of the chromosome. Locality then ensures that changes is a very restricted subset of the chromosome also result in small changes to the solution fitness, while a mutation on a larger range of the chromosome is expected to have a larger impact on the solution fitness.

Genetic Algorithms usually define a “mutation rate” parameter that defines the impact of the mutation operation on the solution. A larger mutation rate allows the operation to change many genes, whereas a small mutation rate restricts the mutation operation to a very limited subset of genes only. It is generally recommended to rely on small mutation rates, as too large mutation rates can quickly render otherwise good solutions completely invaluable.

2.3.5.1 Reciprocal Exchange Mutation (REM)

The Reciprocal Exchange Mutation (REM) implements a simple gene swap between two randomly selected genes. All genes between the selected ones remain unchanged. The `self.mutationRate` parameter specifies the probability of an offspring facing mutation.

The current implementation allows for both selected genes to be equal resulting in no mutation.

General considerations for the selection of `self.mutationRate` especially in respect for optimizing the TSP are documented in 2.3.5.3.

```
    def reciprocalExchangeMutation(self, ind):
        """
        Your Reciprocal Exchange Mutation implementation
        """
        if random.random() > self.mutationRate:
            return ind

        idx1 = random.randint(0, self.genSize-1)
        idx2 = random.randint(0, self.genSize-1)

        tmp = ind.genes[idx1]
        ind.genes[idx1] = ind.genes[idx2]
        ind.genes[idx2] = tmp

        return ind
```

A debug of REM on a small toy TSP instance is documented in Appendix A.

2.3.5.2 Inversion Mutation (IM)

The Inversion Mutation (IM) is - like REM - a mutation operation that is suitable for the TSP optimization as it does not introduce new information, but instead inverses a given sequence of genes within a chromosome. The `self.mutationRate` parameter specifies the probability of an offspring facing mutation.

The inversion operation usually should not affect the solution quality within the inverted genes as their relative order remains intact for the TSP optimization (e.g. cost(B => C) is equal to cost(C=>B)).

General consideration for the selection of `self.mutationRate` especially in respect for optimizing the Travelling Salesman Person problem are documented in 2.3.5.3.

```
    def inversionMutation(self, ind):
        """
        Your Inversion Mutation implementation
        """
        if random.random() > self.mutationRate:
            return ind

        idx1 = random.randint(0, self.genSize-1)
        idx2 = random.randint(0, self.genSize-1)

        if idx1 < idx2:
            ind.genes[idx1:idx2+1] = ind.genes[idx1:idx2+1][::-1]
        else:
            ind.genes[idx2:idx1+1] = ind.genes[idx2:idx1+1][::-1]
        return ind
```

A debug of IM on a small toy TSP instance is documented in Appendix A.

2.3.5.3 General Aspects of Mutation Rate for TSP

Especially for the TSP optimization a small `self.mutationRate` seems useful when the solution already converged towards a potential optimum, where a large `self.mutationRate` would easily result in the inverse effect of breaking a good solution. In contrast, for solutions that are based off random initialization, larger `self.mutationRate` values seem acceptable in the beginning.

2.3.6 Replacement and Elitism

To complete the creation of a new generation of individuals in GA, a decision must be made about which individuals to keep. A naive approach creates as many offsprings as the GA is configured to produce and completely drop the previous generation. This approach strongly runs the risk that the fittest chromosomes within the population are inadvertently lost and the new generation in total shows weaker fitness.

“Elitism” in contrast refers to the concept where the **e** fittest individual from generation **G** are guaranteed to survive and taken into generation **G+1**. Using the concept of elitism ensures that the fittest individual in subsequent generations cannot be worse than the ones from previous generations. Elitism is expected to work nicely with fitness-proportional selection methods like SUS to ensure higher mating chances for fit individuals. Multiple variations of elitism exist, of which some also keep very weak individuals of a population as those still might contain useful genetic information. The amount of how many best solutions from a generation are kept for the next generation can have a large impact on the overall performance of the GA search due to pre-mature convergence.

Elitism in GA searches strongly follows the Darwinian evolution theory of the “survival of the fittest”.

2.4 Experiments

After implementing the requested crossover, mutation and selection operators a set of experiments have to be executed to evaluate the GA performance against provided TSP problem sets.

According to [1] the performance of a metaheuristic, which a Genetic Algorithm is, is measured by considering three major parts:

1. **Experimental Design,** in which we define the goal of our experiments (i.e. what should be achieved), specify the problem instances to be used and the search-specific parameters we modify. These parameters are often known as hyper-parameters as they steer how the search behaves.
2. **Measurements**, in which we define the exact measures that are computed in order to validate whether a certain experiment achieves the goal.
3. **Reporting**, in which we document and present findings and analyses of the experiment results. A proper report allows a third party to run the same experiments with the same results (**reproducibility**).

With respect to our assignment,

- The **goal** is to let the GA find good solutions, i.e. short path lengths, for the TSP **for specified TSP instances** provided.
- Certain parameters and search components have already been fixed in so called **configurations**, specifying the type of population initialization, crossover and mutation operator to use, mating selection, population size and mutation rate. The assignment also requires executing additional experiments based on additional parameters (e.g. elitism) and by altering population size and mutation rate.
- The measurement to take is TSP specific. We have to minimize the objective function (length of the TSP path). This is equivalent to the maximization of the fitness function, which is defined by: F = 1/objectiveFunction.
- The **reporting** is provided with this document. Reproducibility is achieved by specifying all hyper-parameters as configurations and fixing the random number generator with a student-specific seed value.

2.5 Basic Evaluation

For the basic evaluation section of the assignment we measure the GA performance against 2 different configurations. Each configuration uses a random initial solution and random parent selection. The exact requested configurations are documented below. Most notable no elitism is used. Each configuration is executed against any of the three instances to be used.

Configuration	Initial Solution	Crossover	Mutation	Selection
1	Random	Uniform Crossover	Inversion Mutation	Random Selection
2	Random	PMX Crossover	Reciprocal Exchange	Random Selection

Figure 2 Overview of experiment configurations 1 and 2

The following parameters were used:

- Population size: 100
- Mutation Rate: 0.1
- Maximum Iterations: 500
- Offsprings: 1
- Elite: 0
- Childs: 1
- Runs: 5

When evaluating the results especially in regards of time spent to find new solutions one has to take into consideration the number of cities in a given TSP:

- inst-4: 190
- inst-6: 823
- inst-16: 302

2.5.1 Results & Analysis

The following high-level insights can be taken from the pure search results:

Table 1 Results for experiments with configuration 1 and 2

Instance	Config.	Best solution	Average best solution	Search time per 1 GA run
4	1	#4 - 20526281	20802992	00:00:50, 0.1s / iteration
4	2	#1 - 20563018	20738168	00:01:06, 0.132s / iteration
6	1	#3 - 335138175	336428735	00:04:53, 0.586s / iteration
6	2	#2 - 333027156	335680881	00:10:44, 1.288s / iteration
16	1	#5 - 102734039	103572115	00:01:25, 0.17s / iteration
16	2	#5 - 101827691	102732312	00:02:09, 0.258s / iteration

"**Best solution**": The best solution the GA found in 5 runs á 500 iterations.
"**Mean solution**": Average of the best solution found in 5 runs.
"**Search time**": For each iteration a timestamp was recorded to draw conclusions on time efficiency.

As the **performance** per iteration diagrams below indicate both configurations represent more or less a random search as best solutions do not survive, the initial population is created randomly and also parents for mating are selected totally random. It takes a lot of luck to find a better solution than the best one found before. Overall one notices that both the mean fitness and the best fitness per iteration represent a quasi-flat line. In addition, from a performance perspective neither configuration 1 nor 2 outperform each other.

The diagrams below show for each TSP instance the statistics for the run with the best solution found:

a) the best fittest chromosome and
b) the average fitness of the population.

Diagrams for configuration 2 are omitted due to no added value.

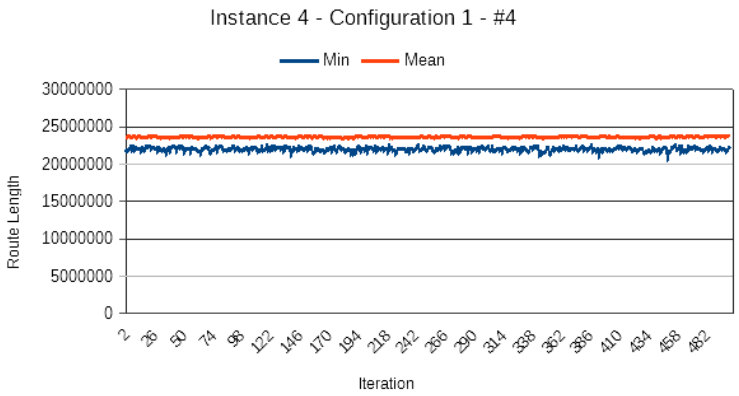


Figure 3 Visualization of results: Instance 4, configuration 1

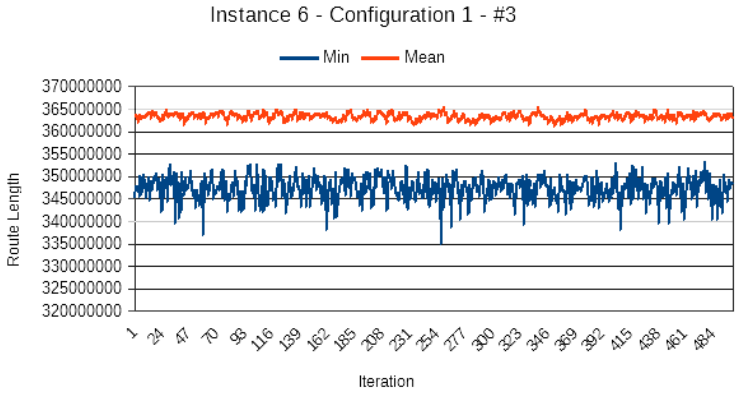


Figure 4 Visualization of results: Instance 6, configuration 1

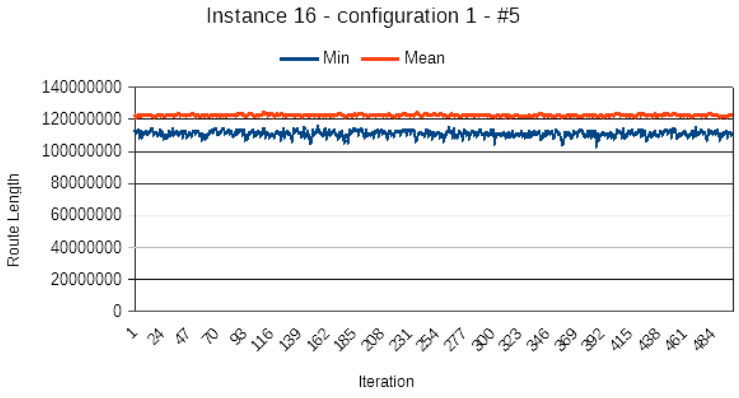


Figure 5 Visualization of results: Instance 16, configuration 1

Among tests for the same configuration one notices a linear increase in **search time** for configuration 1 as the instances grow in size. For configuration 2 instance 6 with its 823 cities experiences a very long run time.

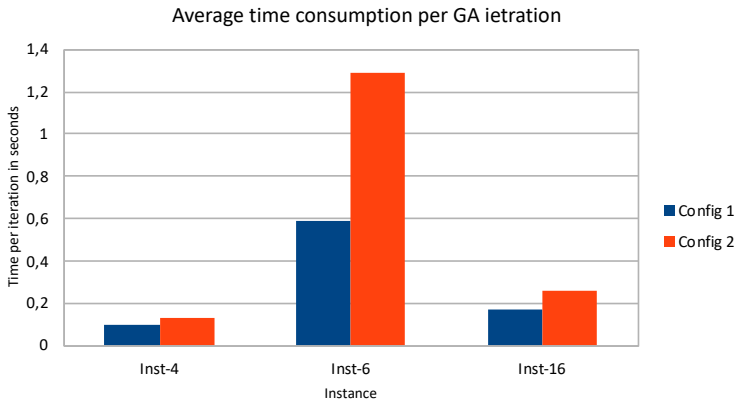


Figure 6 Visualization of time requirements for configurations 1 and 2

While the Reciprocal Exchange Mutation should be independent of the chromosome length (swap of two genes should be of consistent time complexity), especially the mapping part of the PMX crossover operation may be affecting severely the run time of GA with longer chromosomes as the mapping table may be consulted multiple times recursively.

Lastly, let us visualize the best solution found for instance 4. The red dot represents the start and stop location of the tour. It becomes obvious that neither configuration 1 nor configuration 2 produced high quality solutions as very often long distances between cities are traveled.

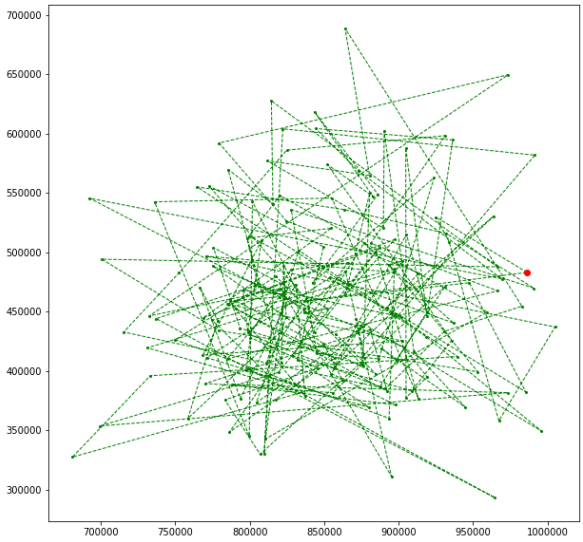


Figure 7 Visualization of the best configuration 1 solution for instance 4

(Note: Instance 4 was chosen for visualization due to being the smallest size tour and therefore offering a less convoluted drawing.)

2.6 Extensive Evaluation

Extensive evaluation has to be carried out against 6 other configurations. All 6 of them now rely on Stochastic Universal Sampling (SUS) to create a mating pool from which two partners for mating are selected.

Configurations 3 to 6 still use a random initial population together with SUS for selecting mating partners for subsequent generations, but also validate all permutations of the two crossover and two mutation techniques implemented as part of this assignment.

Lastly, configurations 7 and 8 are essential similar runs like configurations 4 and 6, but instead of random initialization use the method for heuristically creating the initial population.

The exact requested configurations are documented below. Most notable no elitism is used. Each configuration is executed against any of the three instances to be used.

Configuration	Initial Solution	Crossover	Mutation	Selection
3	Random	Uniform Crossover	Reciprocal Exchange	Stochastic Universal Sampling
4	Random	PMX Crossover	Reciprocal Exchange	Stochastic Universal Sampling
5	Random	PMX Crossover	Inversion Mutation	Stochastic Universal Sampling
6	Random	Uniform Crossover	Inversion Mutation	Stochastic Universal Sampling
7	Heuristic	PMX Crossover	Reciprocal Exchange	Stochastic Universal Sampling
8	Heuristic	Uniform Crossover	Inversion Mutation	Stochastic Universal Sampling

Figure 8 Overview of experiment configurations 3 to 8

The following parameters were used:

- Population size: 100
- Mutation Rate: 0.1
- Maximum Iterations: 500
- Offsprings: 1
- Elite: 0
- Childs: 1
- Runs: 5

2.6.1 Results, Analysis & Interpretation

The following insights can be taken from the pure search results:

Table 2 Results for instance 4 experiments with configurations 3 to 8

Instance	Config.	Best solution	Average best solution	Search time per 1 GA run
4	3	#4 - 20350580	20692957	00:01:02, 0.124s / iteration
4	4	#5 - 20552095	20745685	00:01:09, 0.138s / iteration
4	5	#4 - 20437803	20650256	00:01:09, 0.138s / iteration
4	6	#2 - 20222645	20739594	00:01:03, 0.126s / iteration
4	7	#2 - 3885429	3906724	00:02:13, 0.266s / iteration
4	8	#2 - 3885429	3893158	00:01:02, 0.124s / iteration

Table 3 Results for instance 6 experiments with configurations 3 to 8

Instance	Config.	Best solution	Average best solution	Search time per 1 GA run
6	3	#1 - 333370115	336010704	00:04:50, 0.580s / iteration
6	4	#1 - 330585799	334612366	00:10:47, 1.294s / iteration
6	5	#5 - 332974439	335815044	00:10:56, 1.312s / iteration
6	6	#1 - 333602402	335978040	00:04:56, 2.63s / iteration
6	7	#5 - 12890328	12936326	00:21:55, 0.592s / iteration
6	8	#4 - 12904481	12934039	00:05:06, 0.612s / iteration

Table 4 Results for instance 16 experiments with configurations 3 to 8

Instance	Config	Best solution	Average best solution	Search time per 1 GA run
16	3	#2 - 102450537	103219816	00:01:30, 0.180s / iteration
16	4	#4 - 100190138	101807903	00:02:13, 0.266s / iteration
16	5	#2 - 100190138	101905831	00:02:13, 0.266s / iteration
16	6	#3 - 100783459	102130318	00:01:30, 0.180s / iteration
16	7	#2 - 7398331	7416659	00:03:59, 0.478s / iteration
16	8	#2 - 7398331	7410783	00:01:57, 0.234s / iteration

“**Best solution**”: The best solution the GA found in 5 runs á 500 iterations.
“**Mean solution**”: Average of the best solution found in 5 runs.

"**Search time**": For each iteration a timestamp was recorded to draw conclusions on time efficiency.

Observations for GA results for different configurations executed against the three TSP instances:

- Configuration 6 matches Configuration 1 with the addition of SUS as selection method (random init, uniform crossover, inversion mutation)
- Configuration 4 matches Configuration 2 with the addition of SUS as selection method (random init, PMX crossover, reciprocal exchange mutation)
- With respect to the search time measured for configurations 4 and 6 we can conclude, that the introduction of SUS only led to a minimal increase in GA iterations time due to added complexity in creating a fitness-proportionate mating pool.
- Comparing the search time between configurations 3, 4, 5 and 6 we can conclude that the selection of the mutation method had no impact on the GA runtime. Rather the selection of the crossover operator impacts the runtime. Runtimes with a given crossover operation are pretty constant between all configurations.
- One runtime exception is configuration 7, which most notably relies on a heuristic initial population. I suspect that the initial set of heuristic solutions might show a huge overlap in chromosome structure and therefore the PMX crossover used in this configuration becomes especially costly.
- One also has to note that the addition of SUS does not noticeably improve the GA results for configurations 3, 4, 5 and 6 compared to configurations 1 and 2. This most likely is due to the random initialization and therefore all individuals being of more or less the same - poor - fitness and the GA has to rely on crossover and mutation to find quality offsprings.
- Both configurations 7 and 8 use a heuristic initial population and the results reflect that very fit solutions have been found during the search. Figure 9 below shows the best and mean fitness per iteration for the best configuration 8 run with instance 4. It reveals that those best fitness solutions have been generated by the heuristic and were lost after the first iteration. This has been seen consistently for all three problem instances.
- Using a heuristic initial population introduces some overhead due to more complex code to execute and create initial chromosomes of good fitness. This time is **not** included in the search times listed in the tables 2 to 4 and is in the range of seconds. Generally, larger instances require more time than shorter instances.
- Like for configurations 3 to 6, SUS initially has little influence on the mating pool as with the use of heuristics to generate the initial population most individuals are of relatively equal fitness.
- The lack of elitism is suspected to cause the results seen:
    - For configurations 3 to 6 randomly found fit individuals are mostly likely lost in the next iteration. Hence, SUS will never have the chance to promote fit chromosomes in mating selection and the whole GA process remains pretty much a random search.
    - For configurations 7 and 8 the heuristic provides extremely fit chromosomes. Random crossover and mutation is expected to break this fitness with a large probability. If we do not keep a certain amount of the very fit heuristics-generated chromosomes, the GA search will even out at random search.
- Key assumption: The number of chromosome permutations possible with our TSP problem sets is far too large for a GA search to succeed without elitism.

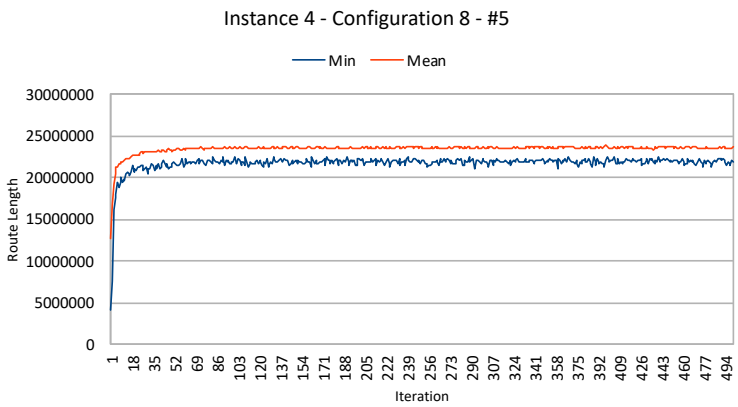
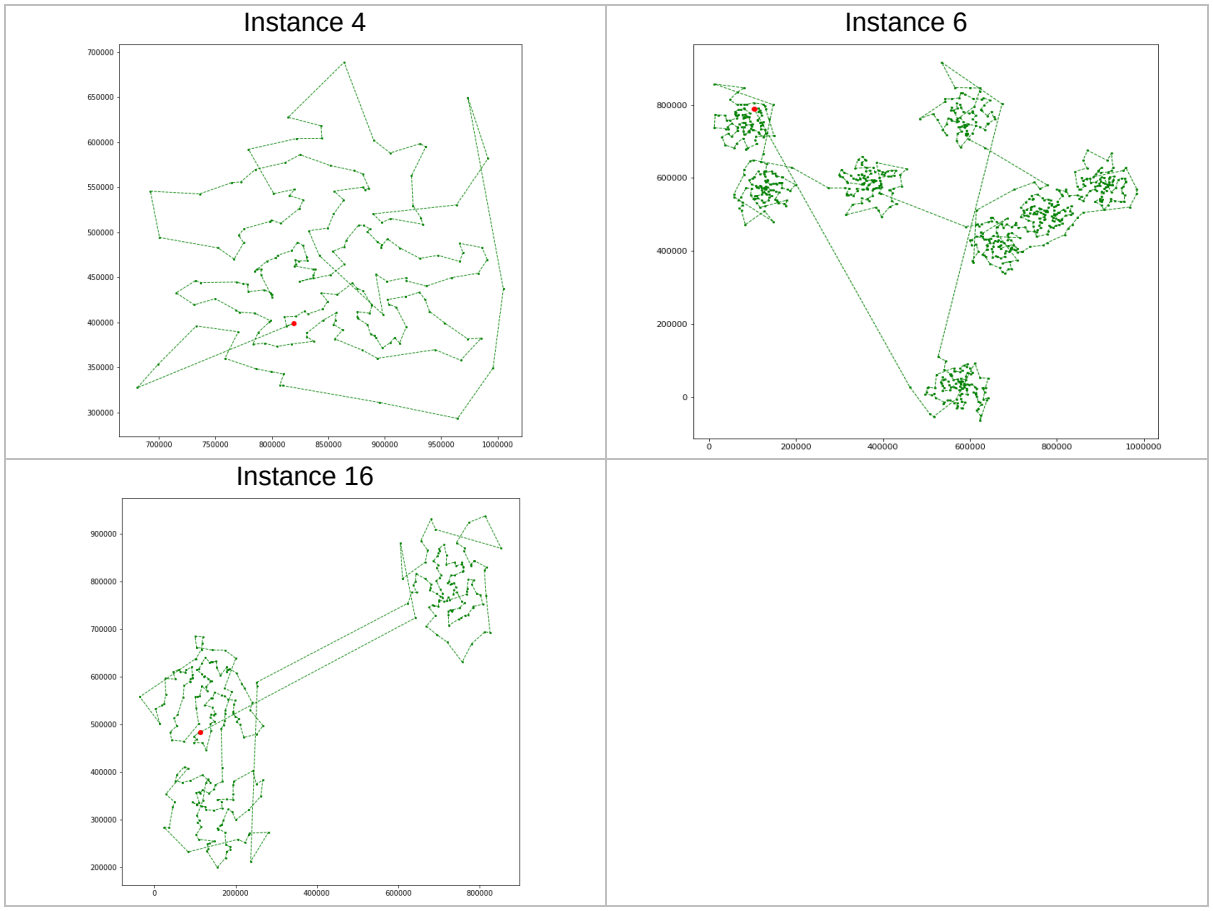


Figure 9 GA search performance for instance 4 with configuration 8

Due to the comparable results of configurations 3 to 6 to those of configurations 1 and 2 further diagrams are **omitted**.

Finally, let's visualize for each problem instance the **best heuristic created chromosome** for the initial populations.



2.7 Additional Experiments

Additional experiments are to be defined and executed to further enhance the search results of our GA implementation. Those experiments will be based on configuration 6 for further random initialization experiments and configuration 8 for heuristics-based initial populations.

As the different configurations executed so far have shown consistent behavior between different size instances, I will focus on working with instance 16 (medium size, 302 cities) as a compromise.

Configuration 6 has shown to provide “good” results across all instances with less time consumption than configurations 4 or 5 and for heuristics-based experiments configuration 7 seems prohibitive with time requirements in mind.

Based on the results collected so far for configurations 1 to 8, **elitism** will be the first additional element to include with the expectation that this will greatly boost the GA performance.

Next, the population size and iterations of a single search will be increased.

Then, mutation rate will be modified, followed by maximum iterations.

Search components to modify:

1. **Elitism** will be evaluated against 3 different values: [1, 10, 25]
2. **Population size** will be evaluated against 2 different values: [250, 500]
3. **Mutation rate** will be evaluated against 3 different values: [0.05, 0.01, 0.2]
4. **Maximum iterations** will be evaluated against 2 different values: [1000, 1500]

Initial parameters carried through experiments until modified:

- Population size: 100
- Mutation Rate: 0.1
- Max iterations: 500

2.7.1 Improve GA Search for TSP instance 16

Now the GA is run against a set of additional configurations that are based on initial configurations 6 and 8. To avoid a full mesh of experiments, I will iteratively search for the best parameter for a given component, before experimenting with the next one.

Table 5 Overview of additional experiments with random initialization

Config	Base Config	Add / Modify	Min Results	Avg Results
6			100783459	102130318
6a	6	Elitism: 1	100037955	101346454
6b	6	Elitism: 10	80036263	85528681
6c	6	Elitism: 25	54588248	57345839
6d	6c	PopSize: 250	88247502	90797972
6e	6c	PopSize: 500	74110072	91699758
6f	6c	Mut Rate: 0.05	46468290	52096781
6g	6c	Mut Rate: 0.01	33406980	41564834
6h	6c	Mut Rate: 0.2	* Skipped	* Skipped
6i	6g	Iterations: 1000	23671827	44834477
6j	6g	Iterations: 2000	21340705	34576252

* Configuration 6h was skipped.

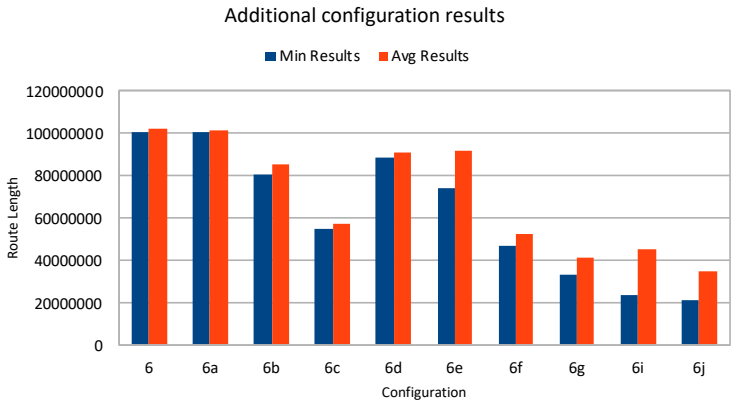


Figure 10 Min and average results for additional experiments with random initialization

By analyzing the GA results 6a (keep single best individual), 6b (keep 10 best individuals) and 6c (keep 25 best individuals) one can immediately recognize the overall improvement of the genetic algorithm. While keeping a single best solution in an otherwise random population (configuration 6a) shows close to no impact, configurations 6b and 6c highlight the benefit of keeping the n best solutions. Interestingly, Figure 11 shows a large gap between the best individual of a population and the average population

fitness for configuration 6b, whereas with configuration 6c the population's average fitness decreases steadily together with the fitness of the best individuals per iteration.

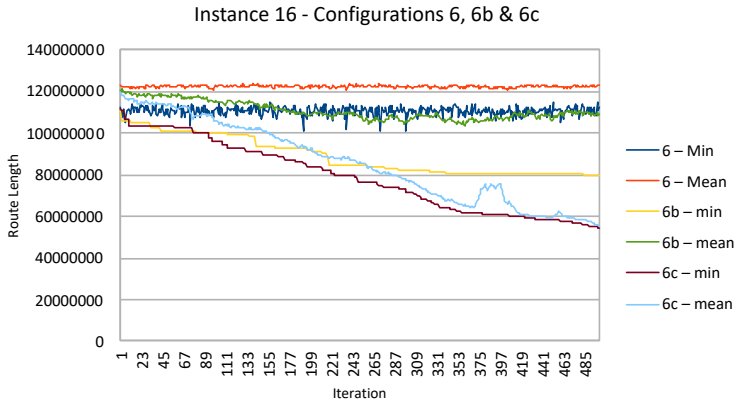


Figure 11 Impact of Elitism on GA results

Figure 10 indicates worse results for configurations 6e and 6f, where based on configuration 6c the population size is increased to either 250 or 500. Likely this is due to the best solutions now again having less chances of getting selected for mating (similar to configuration 6b). Potentially a larger elite group could solve this dilemma.

Next, based on configuration 6c two new mutations rated where tried out: 0.05 (6f) and 0.01 (6g). Both mutation rates showed positive impact on the search results. As seen in Figure 10 configuration 6g produced the best individual chromosomes for TSP instance 16 and became the baseline configuration to finally try more GA iterations: 1000 (6i) and 2000 (6j). The results show that configuration 6j produced the fittest chromosome seen, but like configuration 6i experiences a relatively large variance in the fittest chromosome found across multiple GA search. Figure 10 visualizes this with the 80-100% higher average best fitness across 5 GA searches. This underlines the importance of running multiple searches and also highlights how strongly a GA execution can be affected by random number generators with all other parameters being equal.

Figure 12 visualizes the GA performance of configurations 6c, 6g and 6j. It turns out that while in absolute numbers configuration 6j finds the fittest chromosome seen in all tests, the relative improvement after iteration 800 becomes negligible.

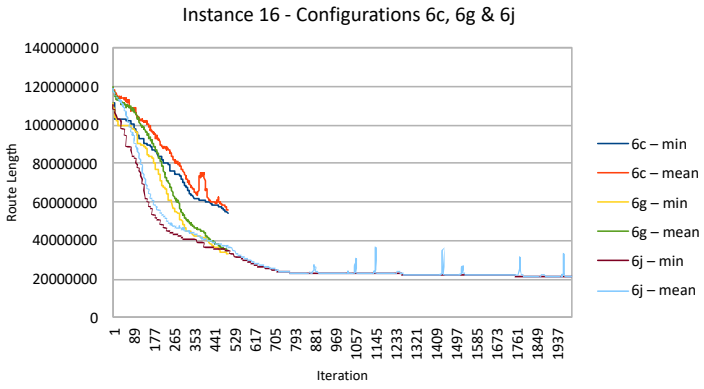


Figure 12 Impact of mutation rate and number of GA iterations

Table 6 Overview of additional experiments with heuristic initialization

Config	Base Config	Add / Modify	Min Results	Avg Results
8			7398331	7410783
8a	8	Elitism: 1	7398331	7410783
8b	8	Elitism: 10	7338467	7394063
8c	8	Elitism: 25	7210408	7273334
8d	8c	PopSize: 250	7327868	7357900
8e	8c	PopSize: 500	** Skipped	** Skipped
8f	8c	Mut Rate: 0.05	7302041	7349158
8g	8c	Mut Rate: 0.01	7390791	7409700
8h	8c	Mut Rate: 0.2	7134217	7260228
8i	8h	Iterations: 1000	7003682	7062205
8j	8h	Iterations: 2000	6762611	6842359

** Experiment 8e was skipped.

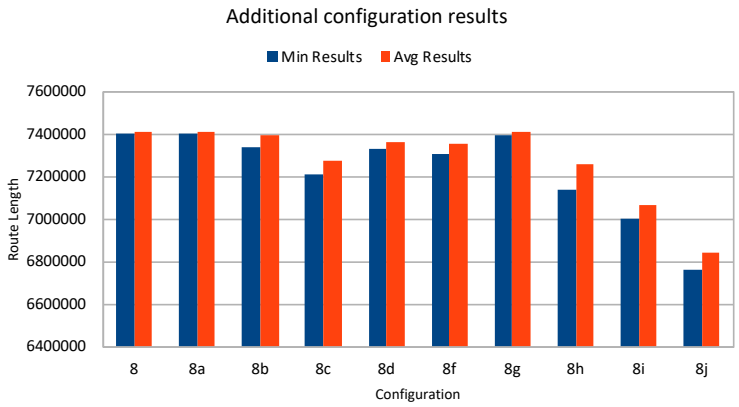


Figure 13 Min and average results for additional experiments with random initialization

The results of the experiments executed against the baseline configuration 8 show a similar pattern like those configurations that enhanced configuration 6. Most notably, configuration 8 creates the initial population based on a heuristic and the earlier configuration 8 test against TSP instances 4, 6, and 16 showed that the best chromosome for an entire GA search was created by the heuristic.

Again, we notice that simply the addition of elitism has a positive effect on the heuristics initiated GA search.

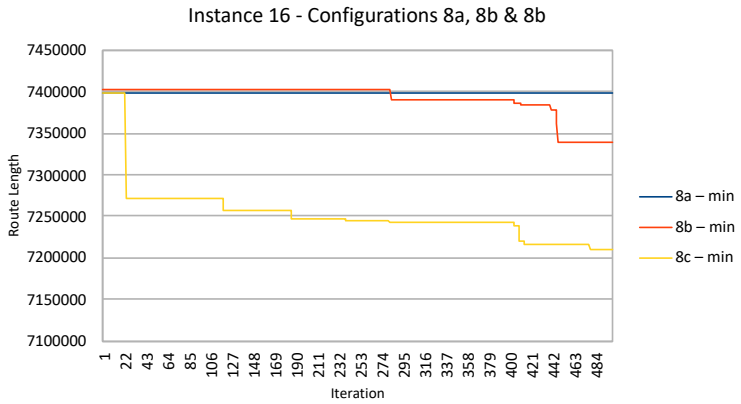


Figure 14 Impact of Elitism on GA results (with heuristics initialization)

Next, similar to the experiments based on configuration 6, the Figure 13 shows the increase of population size had an adverse effect so that the positive effect of elitism disappeared. Also, the previous mutation rate modifications of 0.05 (8f) and 0.01 (8g) did erase out some of the improvements introduced by elitism. Therefore experiment 8h with a mutation rate of 0.2 was executed, which proved useful and improved the overall search performance for finding the fittest chromosome.

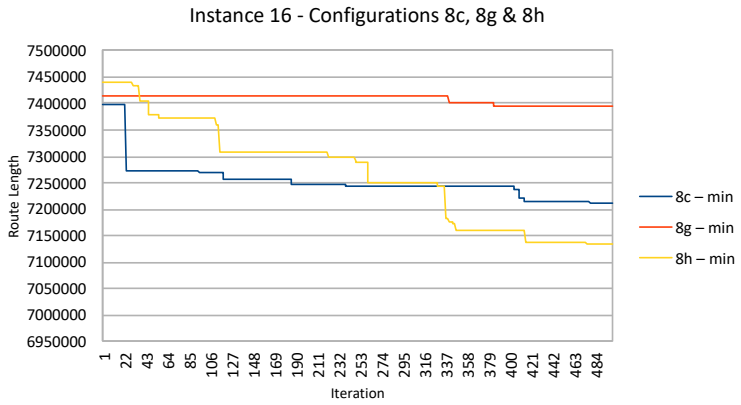


Figure 15 Comparison of different mutation rates with heuristic initialization

Lastly, experiments 8i and 8j increased the number of iterations per GA search and finally resulted in the fittest solution found during all experiments for TSP instance 16.

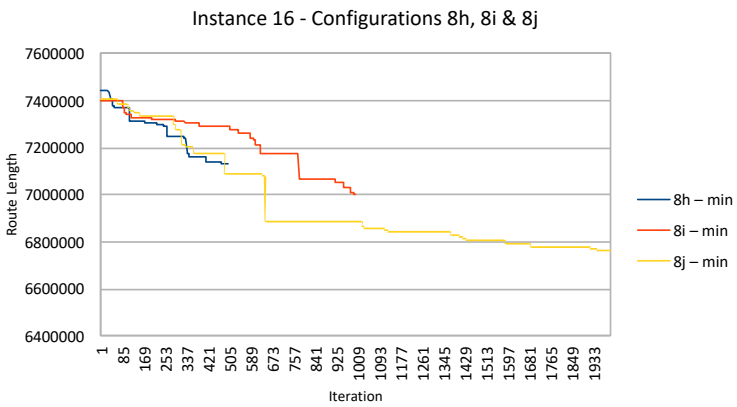


Figure 16 Impact of number of GA iterations

Now, that experiment 8j found the best TSP solution for instance 16, let's visualize the path.

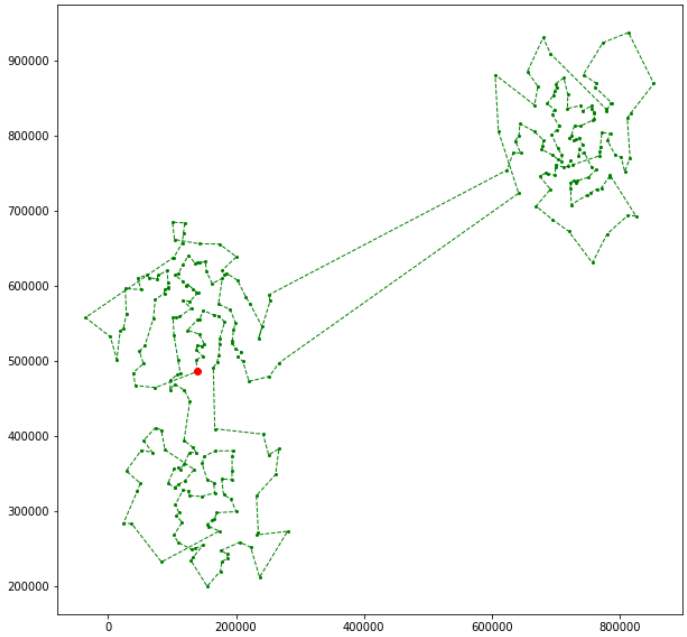
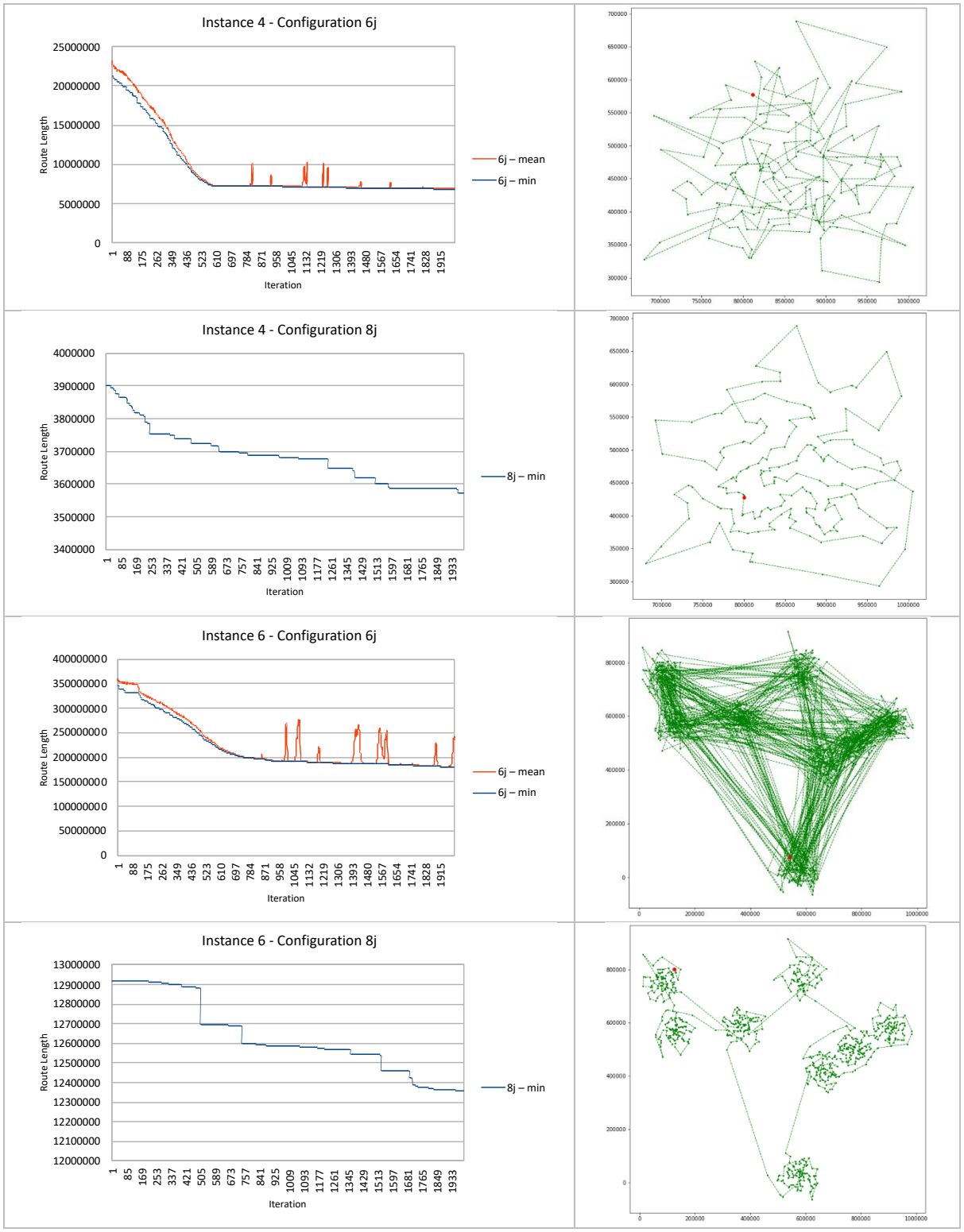


Figure 17 Best TSP solution found for instance 16

Finally, lets apply configurations 6j and 8j against the other TSP instances 4 and 6 and verify whether the settings validated against instance 16 improve GA performance for both random and heuristic initial populations.



Conclusion

In this assignment we used a Genetic Algorithm to find “good” solutions to a set of three Travelling Salesman Problem instances of different sizes by using a multitude of components guiding the GA search: Crossover operations, mutation operations, populations sizes, mutations rates, selections methods and elitism.

It seems that for very large permutation optimization problems like TSP instances the exclusion of elitism cannot be covered by modification of other components of the GA search. Especially with a random population initialization the GA search will remain random. With heuristics based initial populations crossover and mutation operators quickly renders the population unfit in the absence of elitism.

With the usage of elitism interesting results can be seen depending on the initialization strategy. With random initialization large improvement jumps can be seen, but especially for very large TSP instances results remain mediocre, e.g. see instance 6 with configuration 6j.

Using elitism on a heuristically generated initial population together with empirically selected other GA search component parameters has found the fittest solution for all three TSP instances analyzed in this report.

Appendix A: Operator Debugs

A small 10 cities test instance was used to debug the operation of the implemented debugs

Uniform order-based Crossover

```
Create a mask:
Mask:   [1, 1, 0, 1, 1, 1, 0, 0, 1, 0]

Copy parent genes:
inA:    [5, 3, 7, 6, 8, 9, 4, 1, 2, 10]
inB:    [1, 5, 9, 10, 3, 8, 2, 4, 6, 7]

Keep input genes where mask == 1.
childA: [5, 3, None, 6, 8, 9, None, None, 2, None]
childB: [1, 5, None, 10, 3, 8, None, None, 6, None]

Remove kept genes from other parent:
inA:    [7, 9, 4, 2]
inB:    [1, 10, 4, 7]

Final Childs after filling other parents' genes into free positions:
childA: [5, 3, 1, 6, 8, 9, 10, 4, 2, 7]
childB: [1, 5, 7, 10, 3, 8, 9, 4, 6, 2]
```

PMX

```
idx: 4 8
parA: [5, 3, 7, 6, 8, 9, 4, 1, 2, 10]
parB: [1, 5, 9, 10, 3, 8, 2, 4, 6, 7]
childA: [None, None, None, None, 3, 8, 2, 4, 6, None]
childB: [None, None, None, None, 8, 9, 4, 1, 2, None]
mapA: [3, 8, 2, 4, 6]
mapB: [8, 9, 4, 1, 2]
childA: [5, 9, 7, 1, 3, 8, 2, 4, 6, 10]
childB: [6, 5, 3, 10, 8, 9, 4, 1, 2, 7]
```

Reciprocal Exchange Mutation

```
Indices: 0 8
Before mutation: [2, 3, 9, 4, 8, 10, 6, 1, 5, 7]
After mutation:  [5, 3, 9, 4, 8, 10, 6, 1, 2, 7]
```

Inversion Mutation

```
Indices: 8 4
Before mutation: [1, 5, 9, 10, 3, 8, 2, 4, 6, 7]
After mutation:  [1, 5, 9, 10, 3, 4, 2, 8, 6, 7]
```

Stochastic Universal Sampling

```
r = 0.0476222053022189 0.05

Population objective (not ordered)
[1183873.523661946, 1209503.6380992886, 939767.3602230138, 1295476.0402412578,
1068844.736588869, 1127684.8025816525, 1161806.4528580622, 1001809.8866203869,
1155160.380880315, 1212873.4515187985, 1026299.0507202675, 962869.6610759405,
1122222.5364307954, 1077184.7844862356, 1029264.91319358, 1142048.5393775753,
1201291.57704648, 966090.6716563664, 1097828.5521486925, 968560.9532131364]

Population fitness (not ordered)
[0.04598331256335721, 0.04500889832756457, 0.057927555880544414, 0.042021947595335084,
0.05093202446574808, 0.048274505561662566, 0.0468567084819601, 0.05434007689590565,
0.04712629274260978, 0.04488384687278035, 0.0530434343048694, 0.056537689860535374,
0.048509475176973876, 0.05053768588088143, 0.052890587812927596, 0.04766734897598983,
0.04531658034918863, 0.05634918944067168, 0.04958736604862602, 0.05620547276186851]

Resulting mating pool: We see chromosome with fitness 0.057927555880544414 is inserted
twice into mating pool (objective 939767.3602230138)
[1209503.6380992886, 939767.3602230138, 939767.3602230138, 1068844.736588869,
1127684.8025816525, 1161806.4528580622, 1001809.8866203869, 1155160.380880315,
1212873.4515187985, 1026299.0507202675, 962869.6610759405, 1122222.5364307954,
1077184.7844862356, 1029264.91319358, 1142048.5393775753, 1201291.57704648,
966090.6716563664, 1097828.5521486925, 968560.9532131364, 968560.9532131364]
```

References

[1] El-Ghazali Talbi, Metaheuristics - From Design to Implementation,
[2] Varshika Dwivedi et. al., Travelling Salesman Problem using Genetic Algorithm
[3] Evolutionary Algorithms, Dr. Sascha Lange, Albert-Ludwigs-Universität Freiburg