Using a Genetic Algorithm to Solve the Traveling Salesman Problem

Report for the 2nd assignment of COSC 3P71

Maike Rees

November 2015

1 Objective and Problem Definition

The goal is to implement a genetic algorithm that finds valid solutions for the traveling salesman problem. With more and more generations, the fitness of the best solution and the population's average fitness should both improve.

For the example data provided by the university of Heidelberg (Germany) a good outcome is a solution with the best fitness that is smaller than 10000. (If you messure the fitness by the total distance travelled).

1.1 The idea behind my algorithm

I created the class TravelingSalesmanProblem which provides the basic structur and methods for any Traveling Salesman Problem. There are three classes that inherit from this class, two for testing purposes (EasyTsp, MultipleCitiesTsp) and one with the real data mentioned above (TspBerlin).

A TravelingSalesmanProblem-Object has a variable called *solution* where the current best solution is saved. It also saves the current population in the variable *parentPop*. And contains the generation Size *genSize*.

Every generation is one population. A Population contains n Individuals, whereas n is set in the RunMe class and transfered to the TSP-Constructor.

The genetic algorithm starts by creating a TravelingSalesmanProblem-Object and calling it's *findSolution()*-method. It reproduces the current population until the setted generation number is reached.

When reproducing the current population, first of all the algorithm grabs a certain amount of elite individuals and puts them directly into the child population. Then it selects two parents via tournament selection and creates two children with a crossover method. Bevor doing so, it checks for the crossover rate. If a parent is not allowed to reproduce, this parent will be put directly into the child generation. When a child is created, due to the mutation rate, it has a chance to mutate. After that is done (or not), it is put into the child population. Once, there are enough children in the child population, the child population becomes the parent population.

2 Summary of Parameters

There are various parameters that can be adjusted to find the best solution. The following will describe what the paramets stand for and where in the code you can adjust them.

2.1 Populationsize

The Populationsize describes how many individuals are included in one population. I'm using populationsizes between 50 and 200.

To adjust the populationsize, set the desired number as the value of the static int variable popSize in the RunMe class.

```
private static int popSize = 100;
```

2.2 Number of Generations

The Number of Generations describes how often the population will be reproduced. I'm using generation numbers between 600 and 1000.

To adjust the number of generations, set the desired number as the value of the static int variable genSize in the RunMe class.

```
private static int genSize = 800;
```

2.3 Crossoverrate

The crossoverrate describes how many individuals of one population are allowed to do the crossover and reproduce children. The crossoverrate is a float decimal between 0.0 and 1.0. If an individual is allowed to do the crossover, it reproduces one child. If not, the parent itself is put into the next generation.

You can adjust it by setting the desired decimal as the value of the static float variable crossoverRate in the RunMe class.

```
private static float crossoverRate = 0.9F;
```

2.4 Crossover Method

My genetic algorithm supports two crossover methods.

- UOX: Uniform Order Crossover
- PMX: Partially Mapped Crossover

To choose between the crossover methods you have to comment / not comment the appropriate lines in the method reproduce() in the class Population.

2.4.1 The adjusted method for the Uniform Order Crossover (UOX)

```
/**
 * Generate the offspring of the current population.
 * @param tsp
 * @return
*/
   public Population reproduce(TravelingSalesmanProblem tsp) {
      // grab the elite individuals
      Individual elite[] = grabElite();

      Population childrenPop = new Population(size, tsp, crossoverRate,
                 mutationRate, elitismRate);

      // insert elite individuals into new childPopulation
      for (int i = 0; i < elite.length; i++) {
         childrenPop.individuals[i] = elite[i];
      }

      Individual children[] = new Individual[2];
      int childrenCount = elite.length;
      Individual parent1, parent2;

      // generate as many children as needed until the
         // population size is reached
      while (childrenCount < this.size) {

         // grab 2 parents with tournament-selection
         parent1 = selectTournament();
         parent2 = selectTournament();


         // get 2 children

         // UOX:
            int mask[] = getRandomMask();
         // PMX:
         //int pointsForPmx[] = getTwoPointsForPMX(tsp.dimension);

         // think of crossover rate
         if (crossoverOk()) {
            // UOX:
```

```java
                children[0] = parent1.getChildUOX(mask, parent2);
                // PMX:
                // children[0] = parent1.getChildPMX(pointsForPmx[0],
                                // pointsForPmx[1], parent2);

                // mutate children
                // think of mutation rate
                if (mutationOk()) {
                    children[0].mutateReciprocalExchange();
                }

            } else {
                children[0] = parent1;
            }

            // put child (or parent if no crossover) into childrenPop
            childrenPop.individuals[childrenCount++] = children[0];

            // if there is still space in the childrenPop for another child,
                // get it
            if (childrenCount < this.size) {

                if (crossoverOk()) {
                    // UOX:
                    children[1] = parent2.getChildUOX(mask, parent1);
                    // PMX:
                    // children[1] = parent2.getChildPMX(pointsForPmx[0],
                        // pointsForPmx[1], parent1);

                    // mutate child
                    // think of mutation rate
                    if (mutationOk()) {
                        children[1].mutateReciprocalExchange();
                    }

                } else {
                    children[1] = parent2;
                }

                childrenPop.individuals[childrenCount++] = children[1];
            }
        }
        return childrenPop;
    }
```

2.4.2 The adjusted method for the Partially Mapped Crossover (PMX)

```java
/**
 * Generate the offspring of the current population.
 * @param tsp
 * @return
*/
    public Population reproduce(TravelingSalesmanProblem tsp) {
        // grab the elite individuals
        Individual elite[] = grabElite();

        Population childrenPop = new Population(size, tsp, crossoverRate,
                    mutationRate, elitismRate);

        // insert elite individuals into new childPopulation
        for (int i = 0; i < elite.length; i++) {
            childrenPop.individuals[i] = elite[i];
        }

        Individual children[] = new Individual[2];
        int childrenCount = elite.length;
        Individual parent1, parent2;

```

```
        // generate as many children as needed until the
            // population size is reached
        while (childrenCount < this.size) {

            // grab 2 parents with tournament-selection
            parent1 = selectTournament();
            parent2 = selectTournament();


            // get 2 children

            // UOX:
            // int mask[] = getRandomMask();
            // PMX:
            int pointsForPmx[] = getTwoPointsForPMX(tsp.dimension);

            // think of crossover rate
            if (crossoverOk()) {
                // UOX:
            // children[0] = parent1.getChildUOX(mask, parent2);
                // PMX:
                children[0] = parent1.getChildPMX(pointsForPmx[0],
                                    pointsForPmx[1], parent2);

                // mutate children
                // think of mutation rate
                if (mutationOk()) {
                    children[0].mutateReciprocalExchange();
                }

            } else {
                children[0] = parent1;
            }

            // put child (or parent if no crossover) into childrenPop
            childrenPop.individuals[childrenCount++] = children[0];

            // if there is still space in the childrenPop for another child, get it
            if (childrenCount < this.size) {

                if (crossoverOk()) {
                    // UOX:
                    //children[1] = parent2.getChildUOX(mask, parent1);
                    // PMX:
                    children[1] = parent2.getChildPMX(pointsForPmx[0],
                            pointsForPmx[1], parent1);

                    // mutate child
                    // think of mutation rate
                    if (mutationOk()) {
                        children[1].mutateReciprocalExchange();
                    }

                } else {
                    children[1] = parent2;
                }

                childrenPop.individuals[childrenCount++] = children[1];
            }
        }
        return childrenPop;
    }
```

2.5 Mutationrate

The mutation rate describes how many children will be mutated after they are created through the crossover. The mutation rate is a float decimal between 0.0 and 1.0.

To adjust mutation rate, set the static float variable mutationRate in the RunMe class to the desired decimal.

```java
private static float mutationRate = 0.1F;
```

2.6 Mutation Method

I implemented two different mutation methods.

- Reciprocal Exchange (RE)
- Modified Reciprocal Exchange (MRE)

The second of the above mentioned methods gives better results. The final results vary by approximately 2000 in the fitness. That's why I only used the Modified Reciprocal Exchange for the tests discussed in 3. Results. If you still want to use the Reciprocal Exchange, you only have to comment / non comment the appropriete lines in the method reproduce() in the Class Population.

The following code snippet is the version for the Modified Reciprocal Exchange

```java
/**
 * Generate the offspring of the current population.
 * @param tsp
 * @return
 */
public Population reproduce(TravelingSalesmanProblem tsp) {
...
        if (mutationOk()) {
//              children[0].mutateReciprocalExchange();
                children[0].mutateBetter();
            }
...
        if (mutationOk()) {
//                  children[1].mutateReciprocalExchange();
                    children[1].mutateBetter();
                }
```

2.6.1 How the Modified Reciprocal Exchange works

While at the original reciprocal exchange you change two random cities, at the modified version, you change all the cities in between those two points. You change the first point with the last point (within these boarders), the second point with the second last point and so on. Here is an example:

original chromosome: 0 1 2 3 4 5 6 7 8
two random generated points: 1, 5
mutated chromosome: 0 5 4 3 2 1 5 6 7

2.7 Elitismrate

The elitismrate describes how many of the elite individuals are grabbed out of the parent population and are put without crossover and mutation into the cild population. It is a float decimal between 0.0 and 1.0.

To adjust the elitismrate you have to set the static vloat variable elitismRate in the RunMe class to the desired decimal.

```java
private static float elitismRate = 0.1F;
```

2.8 Amount of selected individuals in the tournament selection

To pick the two parents for the crossover, my algorithm uses a tournamet selection strategy.

The amount of the selected individuals that are randomly picked for the tournament selection is set at the final int tournamentSelection in the class Population.

```
/**
 * The amount of Individuals that will be compared in the tournament
 * selection
 */
private final int tournamentNumber = 4;
```

2.9 Random seed

To be able to compare the results while changing single parameter values, I need a seed for my random number generator.

It is set in the class RunMe.

```
public static final int seed = 1000;
```

2.10 Amount of Cities

For testing purposes, I created a class TspMultipleCities. It represents a Traveling Salesman Problem, but you can decide yourself how many cities should be included by the individuals.

To adjust this number of cities, you just have to give this number into the constructor (at the position int dimension).

```
/**
 * A Tsp for testing purposes.
 * It uses real City - Data but only as much as you want it to.
 * @author Maike Rees
 *
 */
public class TspMultipleCities extends TravelingSalesmanProblem {

    public TspMultipleCities(int dimension, int popSize, int genSize,
                    float crossoverRate, float mutationRate,
                        float elitisimRate) {

        super(dimension, popSize, genSize, crossoverRate, mutationRate,
                    elitisimRate);
        TspBerlin arrayCopy = new TspBerlin(popSize, genSize,
                        crossoverRate, mutationRate, elitisimRate);
        initialize(arrayCopy.nodes);
    }
    ...
}
```

3 Results

3.1 Comparing two different Crossover methods

In the following experiments, I am testing two crossover methods with different parameters.

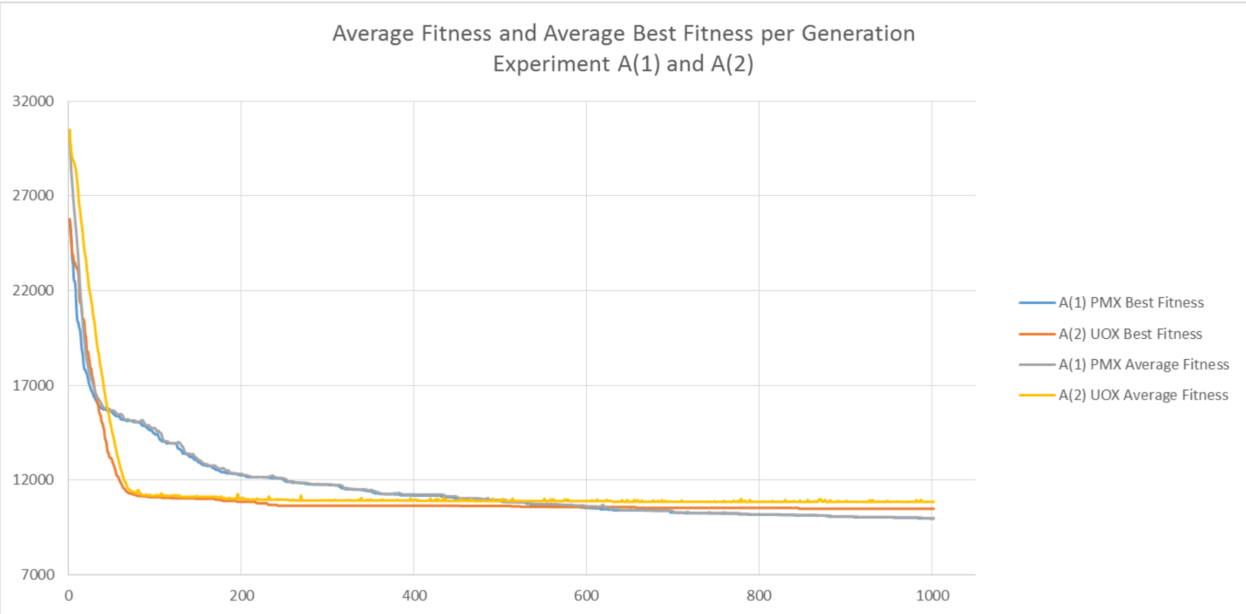
The parameters that are the same in all experiments are the following. I don't change my seed, because my program calls the solve() method multiple times without the need to start the programm new.

- Population Size: 150

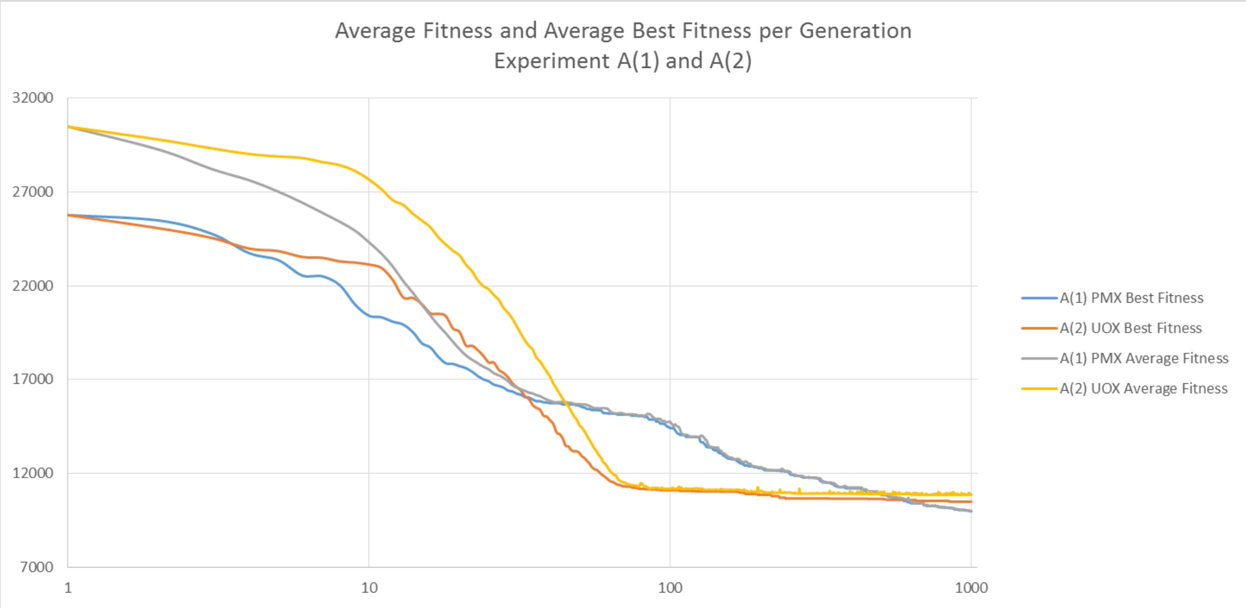
- Number of Generations: 1000
- Elitism Rate: 9
- Random seed: 500
- Number of Runs: 10
- Amount of selected individuals in the tournament selection: 5

Experiment	Crossover Rate	Crossover Method	Mutation Rate	Average best Fitness
A (1)	100%	PMX	0%	9978
A (2)	100%	UOX	0%	10496,2
B (1)	100%	PMX	10%	8721,4
B (2)	100%	UOX	10%	9563,6
C (1)	80%	PMX	0%	10606,4
C (2)	80%	UOX	0%	10828
D (1)	80%	PMX	10%	9029,8
D (2)	80%	UOX	10%	8898,6
E (1)	100%	PMX	50%	8723,6
E (2)	100%	UOX	50%	9340,8

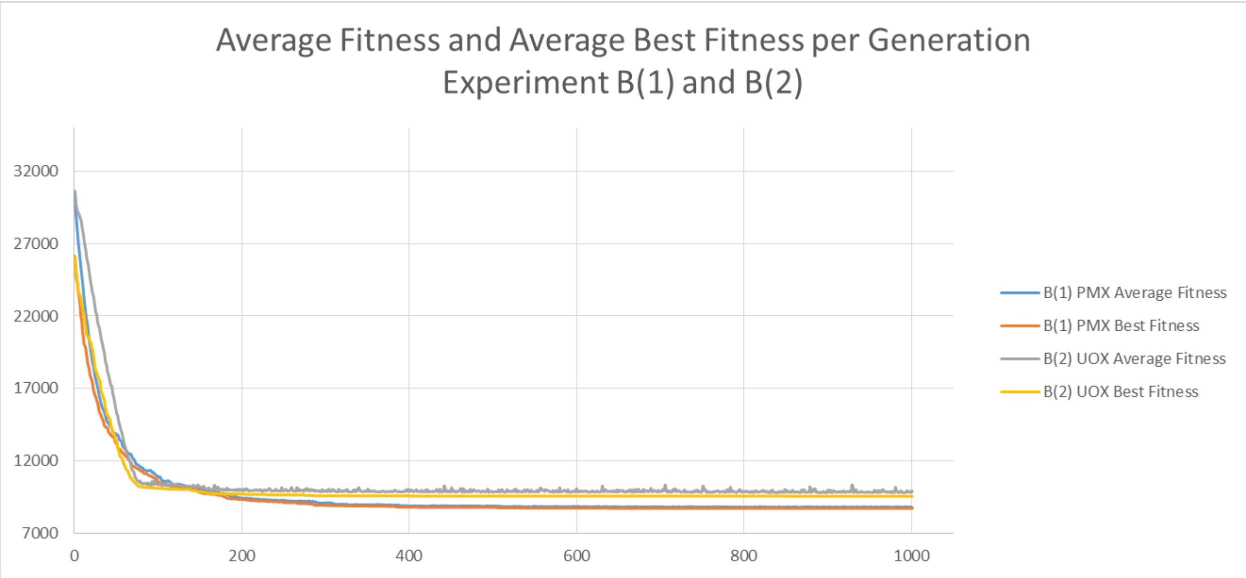
3.1.1 Experiment A



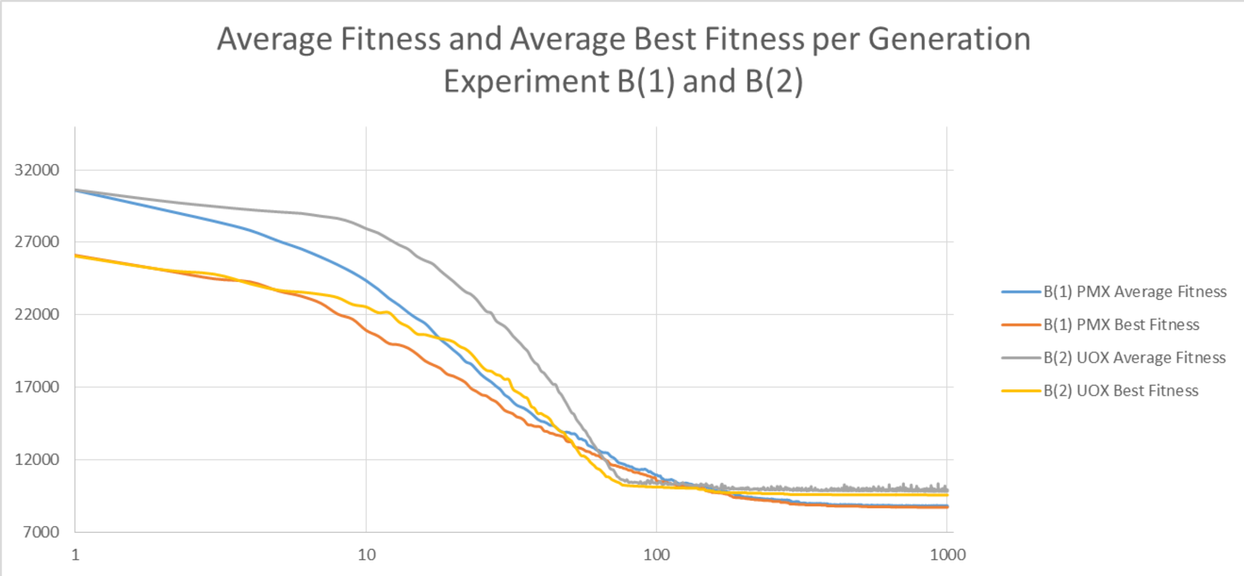
With a logarithmic scaled x-axis:



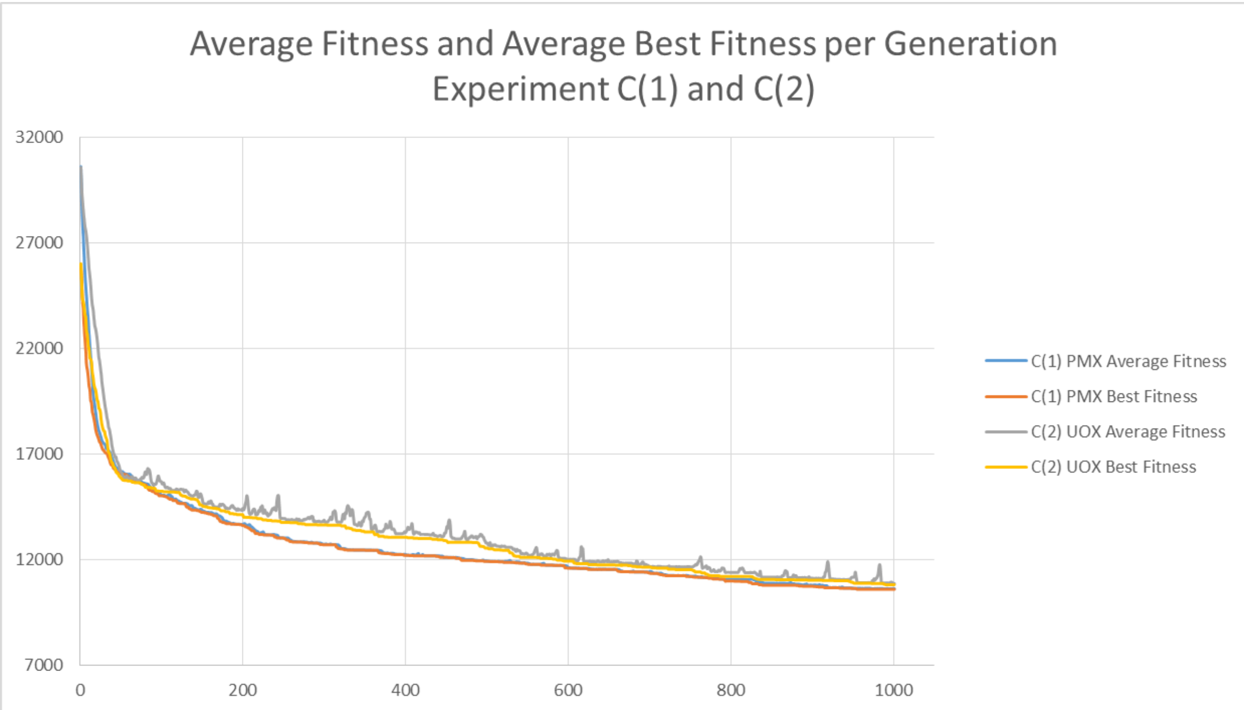
3.1.2 Experiment B



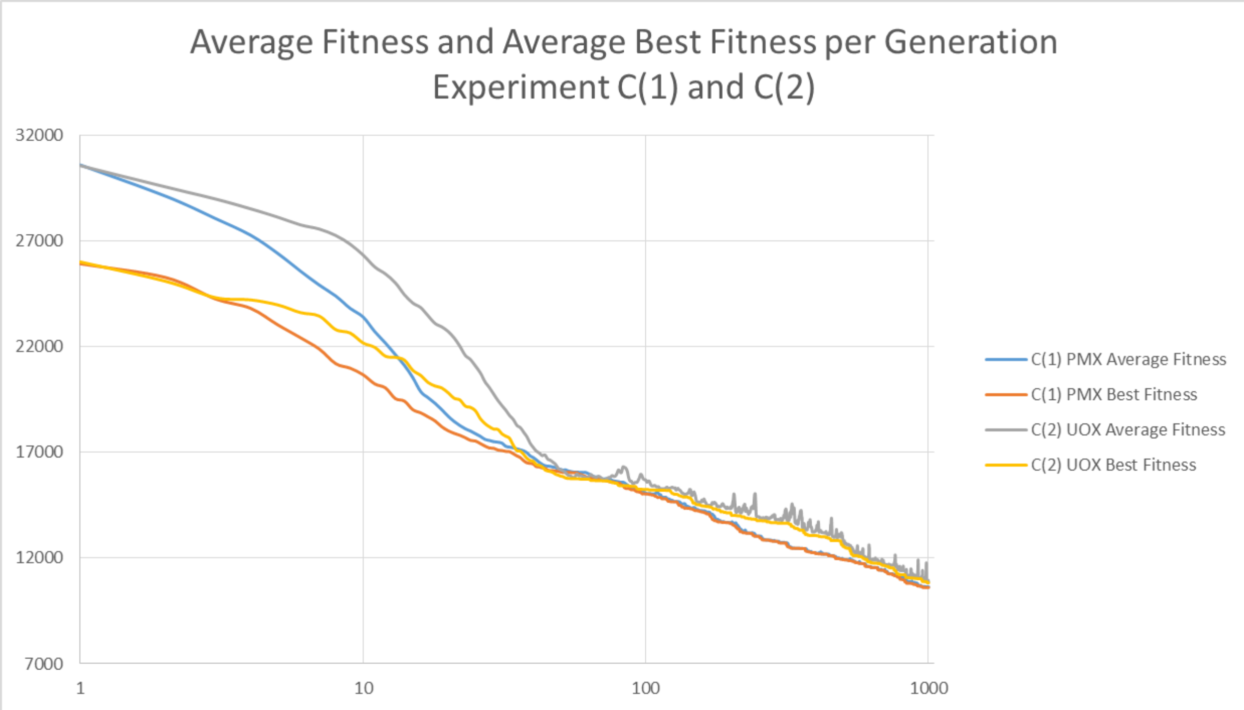
With a logarithmic scaled x-axis:



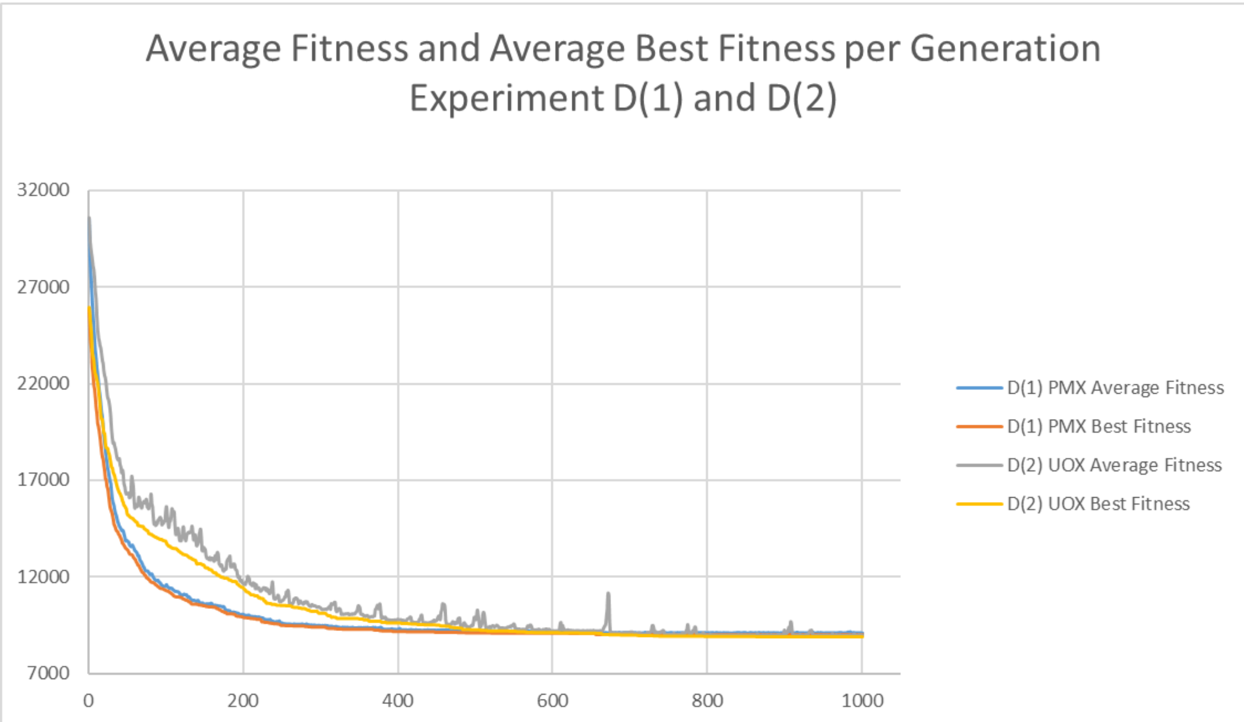
3.1.3 Experiment C



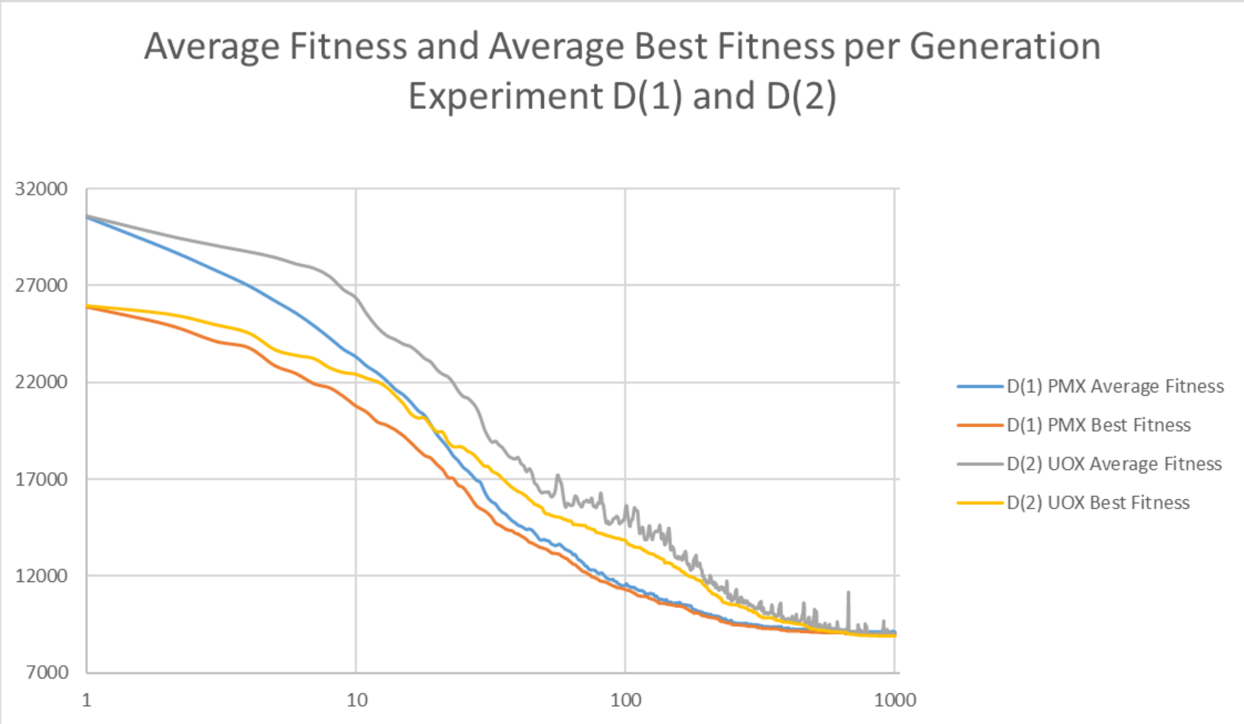
With a logarithmic scaled x-axis:



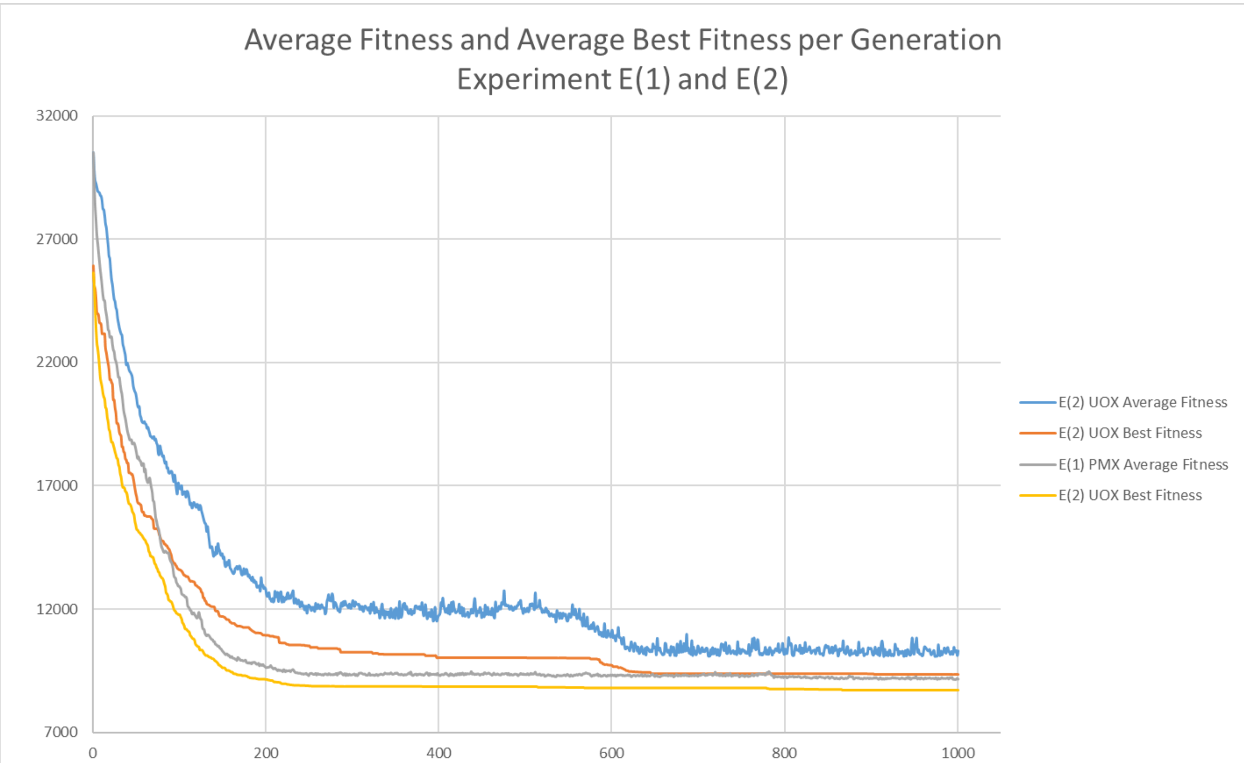
3.1.4 Experiment D



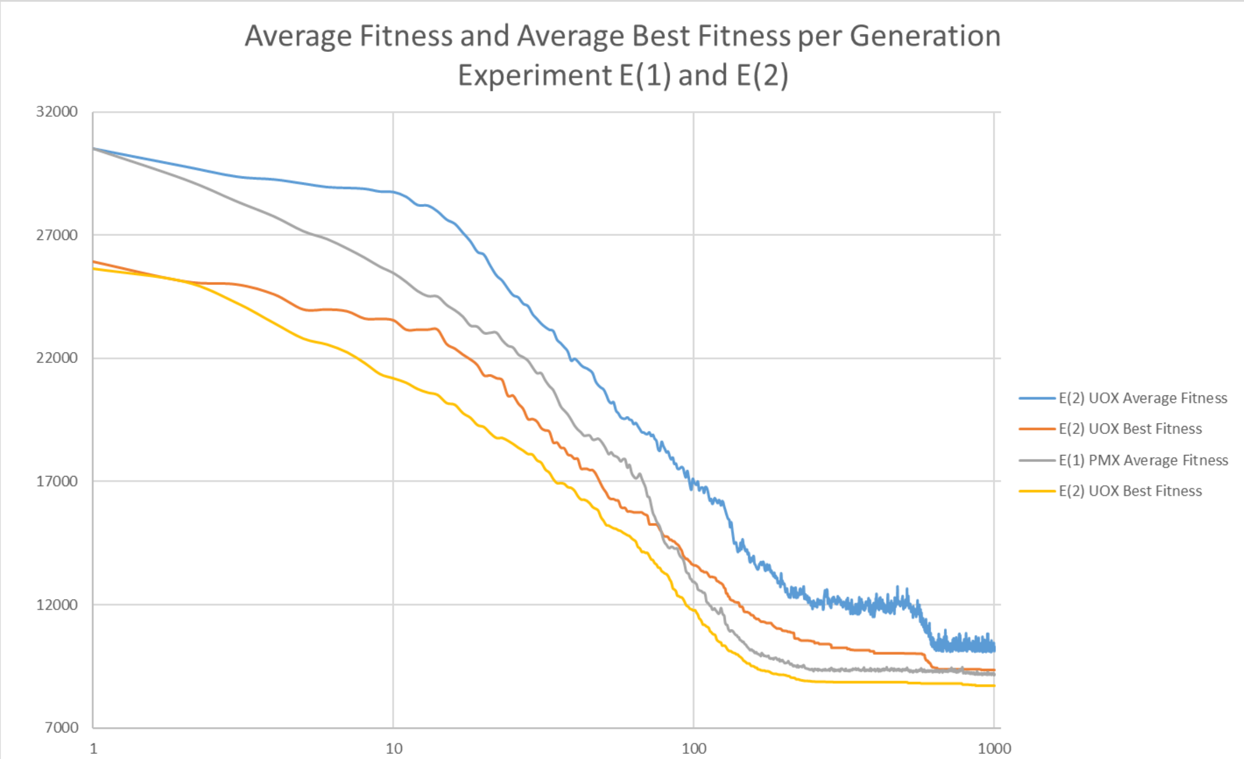
With a logarithmic scaled x-axis:



3.1.5 Experiment E



With a logarithmic scaled x-axis:



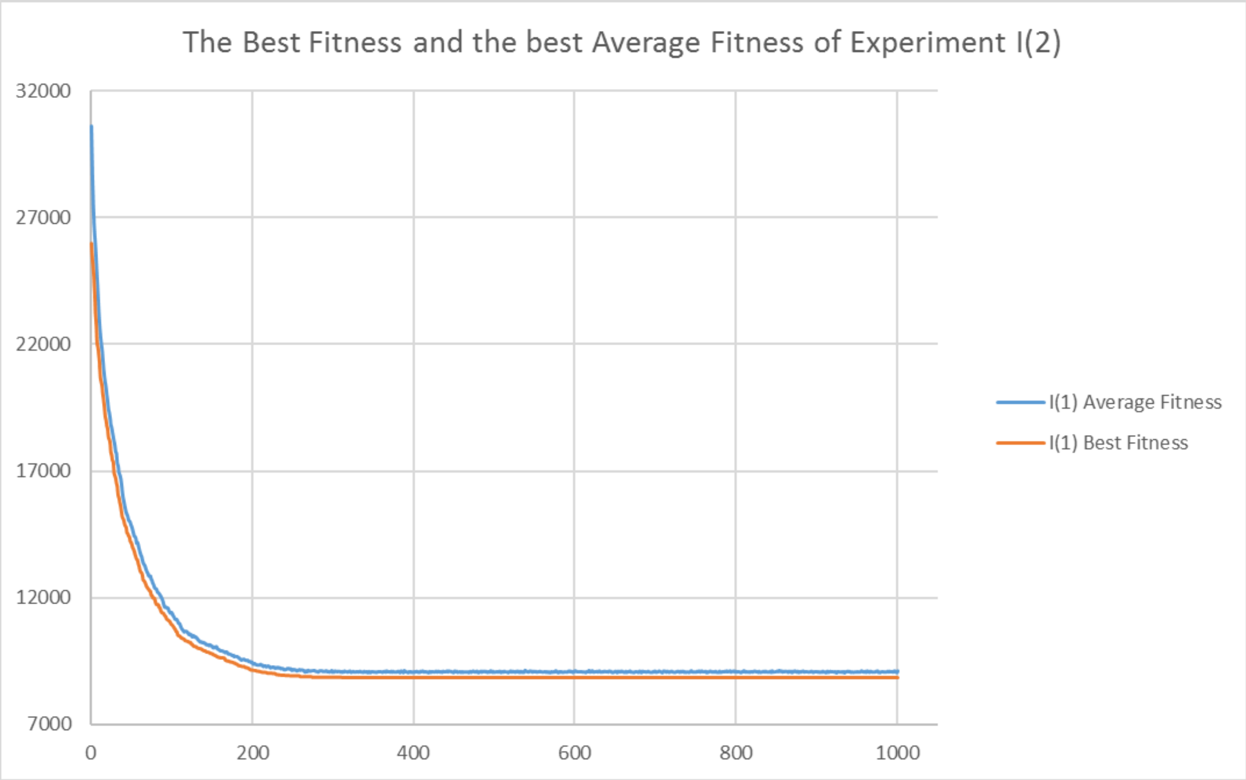
3.2 Comparing two different Mutation methods

Now I am comparing two different mutation methods (described in 2.6). I'm using the same parameters as described in 3.1. For the crossover, I am using the PMX, since this scored the better results on average, as shown in 3.1.

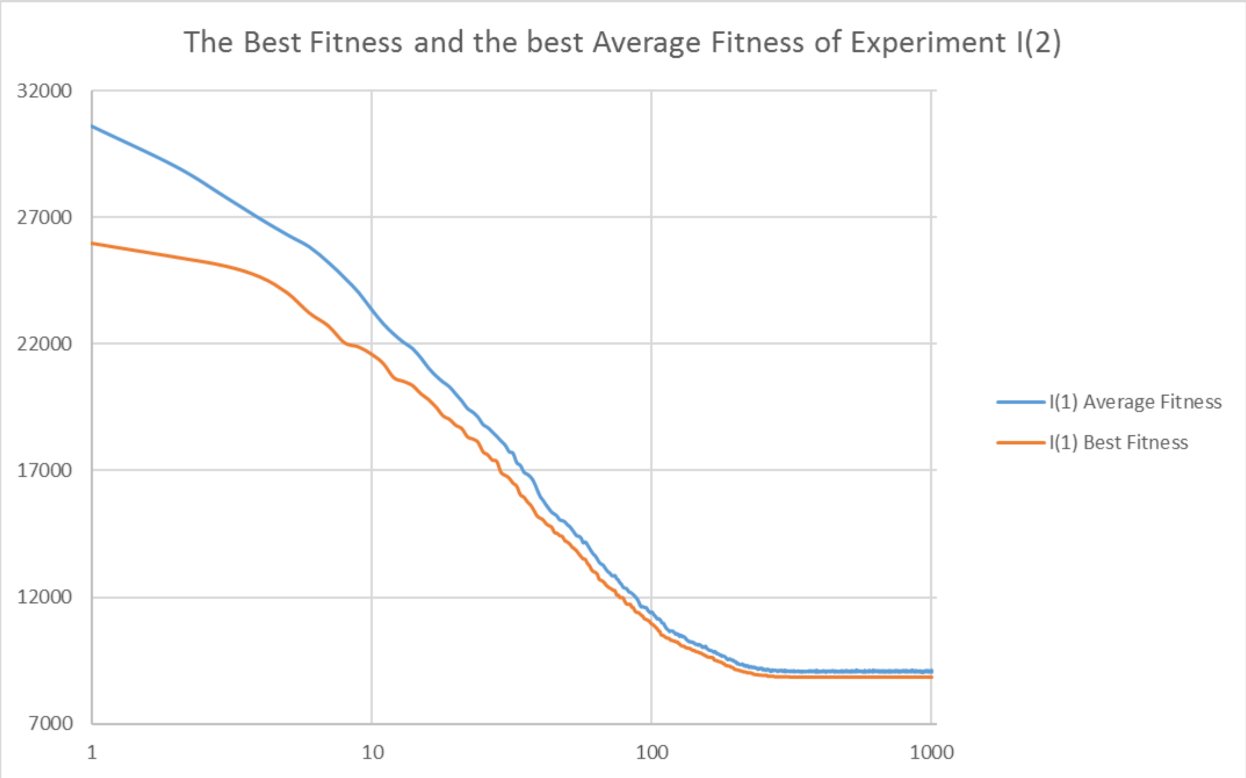
Here are the cases that I've tested:

Experiment	Crossover Rate	Mutation Rate	Mutation Method	Average best Fitness
F (1)	100%	10%	RE	11339,2
F (2)	100%	10%	MRE	8721,4
G (1)	80%	10%	RE	11568,2
G (2)	80%	10%	MRE	9029,8
H (1)	100%	50%	RE	10886,6
H (2)	100%	50%	MRE	9340,8
I (1)	50%	50%	RE	11561,8
I (2)	50%	50%	MRE	8846,4
J (1)	10%	50%	RE	10953
J (2)	10%	50%	MRE	8865,4

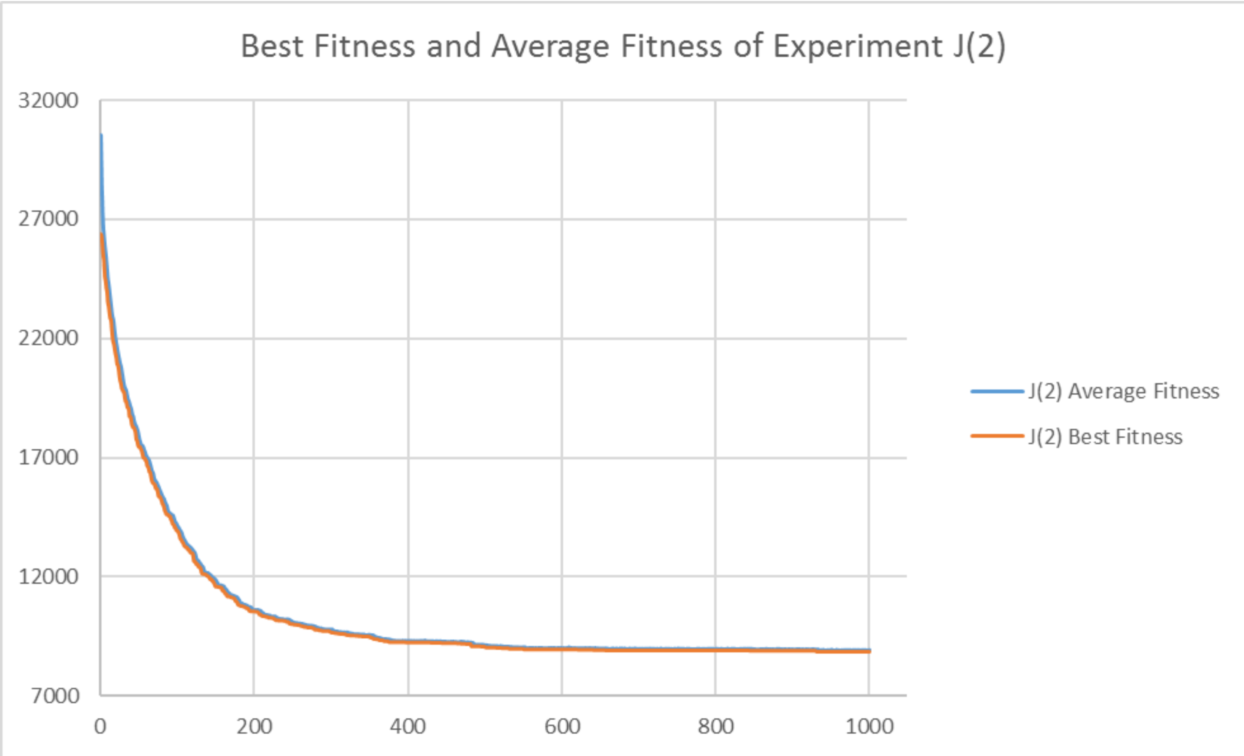
3.2.1 Experiment I



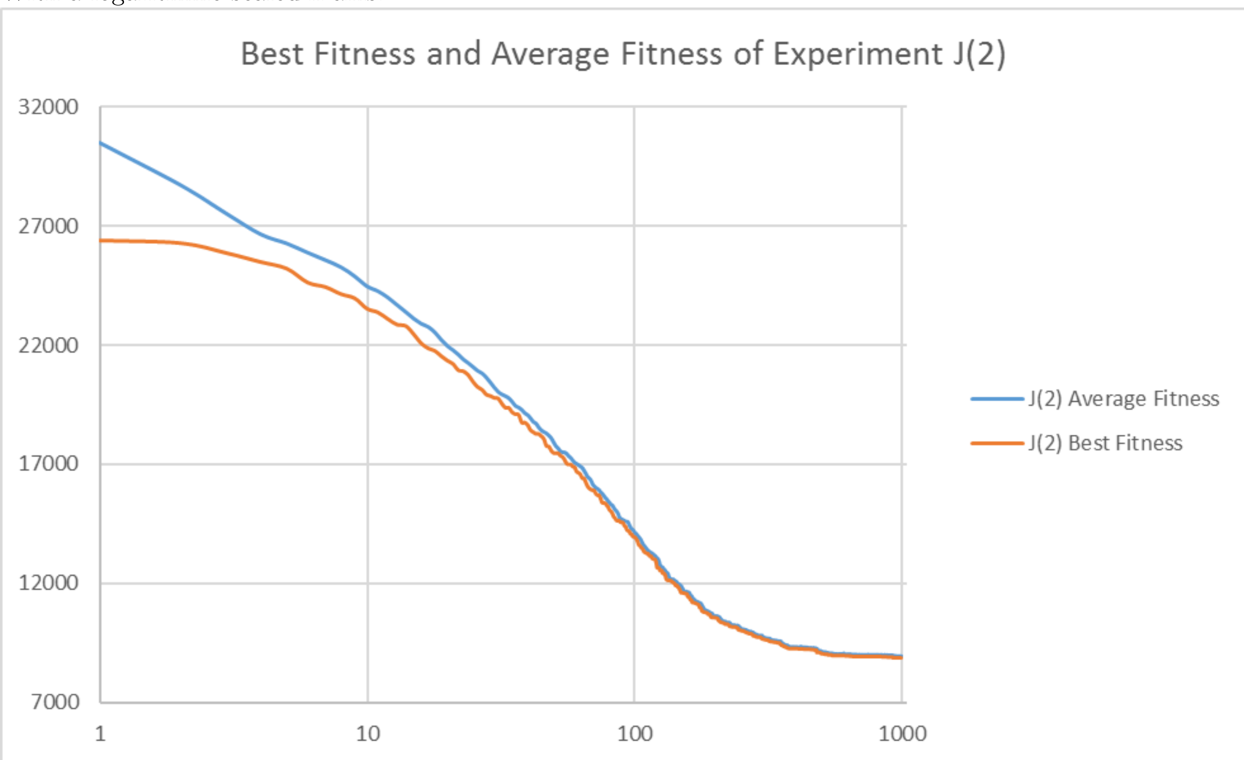
With a logarithmic scaled x-axis:



3.2.2 Experiment J



With a logarithmic scaled x-axis:



4 Discussion and Conclusion

from your results. You discussions should include issues like which crossover performed better than the other one, if more than one mutation type tried, which one performed better. If you included local search, did it help? How did the choice of GA parameters affect the final outcome etc?

4.1 Speed of finding a solution

4.2 Porximity of average and best fitness

4.3 Effect of the Mutation Rate

4.4 Effect of the Crossover Rate

4.5 Miscellaneous