COSC 3P71 – FALL 2015
Artificial Intelligence – Assignment 2
Matt Laidman
5199807, ml12ef
November 6, 2015

Contents

Assignment Outline

Fall 2015 COSC **3P71 Artificial Intelligence: Assignment 2**

Instructor: B. Ombuki-Berman

TA & Tutorial leader: Justin Maltese

Assigned Date: Tuesday October 13th, 2015

Due: Friday, Nov. 6th, by 4:00 pm, **NO LATES.**

Goal: Genetic algorithm implementation and application.

Languages: Any programming language of your choice.

Task: Implement a genetic algorithm system for the Traveling Salesman Problem described in class using the data provided.

The overall procedure for a simple GA will be something like this:

Procedure simple GA
begin
Read problem instance data;
set GA parameters;
generate randomly an initial population POP of size Pop_Size;
    for gen = 1 to MAXGEN do
    evaluate fitness of the individuals of POP;
    select new population using some selection strategy;
    apply genetic operators, crossover and mutation;
    endfor;
end;

The GA's main modules are described as:(a) **Initial Population initializer:** creates a population of size Pop_Size of randomized individuals as described in class.

(b) **Reproduction:** use Tournament Selection

(c) **Crossover:** given two individuals, this creates two offspring. Implement your GA using the following crossover strategies independently:

(i) Uniform order crossover (UOX, with bit mask)

(ii) A crossover of your choice as discussed in class. Of course, the crossover must ensure valid offspring chromosomes for the TSP problem. Some examples are: partially mapped crossover (PMX), Order crossover (OX), cycle crossover (CX).

(d) **Mutator:** given an individual, this creates a mutated individual. Implement your GA using a mutation operator of your choice (from those discussed in class). Again, the mutation operator must ensure a valid chromosome for the TSP problem!

(e) **Fitness evaluation function:** Total distance traveled, and recall that the last city visited is connected to the first city (the shorter the distance, the more fit an individual is)

(f) **Genetic algorithm system:** This is the implementation of the GA system

(g) **user parameters:** population size, maximum generation span, probability of (crossover, mutation etc)

Your GA program should permit the user to easily define his/her own genetic parameters (e.g., crossover rate, mutation rate, population size, maximum generation span etc....). The program should allow testing for various problem sizes, i.e., size of cities, n.

Experimental Analysis

1) Run your GA to compare the performance of the two crossover operators mentioned above by using the following parameters (and include elitism in all cases):

a. Crossover rate = 100%, mutation = 0%

b. Crossover rate = 100%, mutation = 10%

c. Crossover rate = 80%, mutation 0%

d. Crossover rate = 80%, mutation 10%

e. Your own parameter settings

(PS. For elitism), first consider an elite strategy where only the best chromosome is replicated to the next generation. Next consider replication criteria where a certain percentage of individuals determined empirically (no greater than 10%) are allowed to replicate (include chromosome of best fitness value for this replication)

2) Incorporate into your experiments your own innovativeJ‟ idea; this would be a different initial population representation & creation, a different selection scheme, a different crossover or even mutation. Your idea could be introducing some local search into your GA etc (This will be for bonus mark of 2% of total course grade)

For each experiment mentioned above, run your GA at least 5 times. Use the data set “berlin52.tsp” obtained from TSP-online benchmark data at:

http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/tsp/

Output the following to a file or standard output:

a) All GA parameters, including random number seed

b) Per each generation: best fitness value, average population fitness value

c) Per each run: best solution fitness and its corresponding best solution chromosome

Finally compute for the multiple runs: average of best fitness per generation and average population fitness per generation. Using a graph drawing tool such as excel, plot well labeled graphs forexperiment 1, above including experiment 2 (if done). Types of graphs you plot for experiment 2 depend on your incorporated idea, if any. Feel free to experiment with different crossover and mutation rates. You will set your own Pop- Size and generation size.

Lastly, prepare a summarized report with sub-headings (a) Objective and problem definition b) Summary parameters used (c) Results: that is, summary tables and graphs, EXPLAIN your graphs in detail. (d) Discussions and conclusions 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? This report is very important, so be sure to include it. Start early, gathering the data and doing the experimental analysis will take much more time than coding the assignment.

Objective

Solve the Travelling Salesman Problem (TSP) by implementing a Genetic Algorithm (GA) capable of executing any number of input parameter combinations such that the user is able to explore strategic mutation rates, crossover rates, tournament k-values, crossover types, population sizes, and number of generations.

Analyze the data created by the program and discuss why and how different combinations of input parameters contributed to the different solutions to the problem.

Implementation

Software Usage

The program is packaged in a .jar file and can be executed with the following command:

```
  java -jar TSPSolve.jar
```

The expected input parameters are as follows. the parameters themselves are discussed in the next section.

```
  java -jar TSPSolve.jar "path/to/file.tsp" MAX_POP POP_SIZE  K_VALUE
MUTATION_RATE CROSSOVER_TYPE CROSSOVER_RATE SEED
```

Executing the program with no parameters will produce a summary of the input parameters and the value expected.

```
  $java -jar TSPSolve.jar

  Psst! Try:

  java -jar TSPSolve.jar "path/to/file.tsp" MAX_POP POP_SIZE  K_VALUE
MUTATION_RATE CROSSOVER_TYPE CROSSOVER_RATE SEED

    MAX_POP      Integer - Maximum generations before      quiting
    POP_SIZE   Integer - Population size of the chromosomes
    K_VALUE      Integer - 'k' value used in the         tournament
selection
    MUTATION_RATE  Double - Rate of occurrence of mutations
    CROSSOVER_TYPE Integer - 0 for Uniform Order Crossover, 1
  for Order Crossover
    CROSSOVER_RATE Double - Rate of occurrence of crossovers
    SEEDLong - Seed used for random number generator

    Note: .tsp file is expected to be in the following format:
```

```
NAME: berlin52
TYPE: TSP
COMMENT: 52 locations in Berlin (Groetschel)
DIMENSION: 52
EDGE_WEIGHT_TYPE: EUC_2D
NODE_COORD_SECTION
1 565.0 575.0
2 25.0 185.0
3 345.0 750.0
...
```

```
EDGE_WEIGHT_TYPE is not considered in this implementation
```

The execution script which was used in the gathering of data for the analysis portion of this report is included in both near the end of the report, and in the electronic submission, and can also be used as usage reference.

Program Parameters and Design Choices

This implementation of a genetic algorithm to solve the travelling salesman problem uses the following operators and functions:

- **Reproduction** – For reproduction a k-tournament selection was used, where the specific k-value used is a parameter supplied by the user.
- **Crossover** – For crossover two operators were implemented, and which is used is determined by user parameters:
  - Uniform Order Crossover
  - Order Crossover
- **Mutation** – Reciprocal Exchange Mutation was implemented for the mutation operator.
- **Evalutation** – The evaluation function evaluates the chromosomes (paths) for the shortest path in the population in that current generation. Elitism is implemented to ensure that the best path found up too any point will always be included in the next generation.

Parameters that are used in these functions and operators must be specified by the user at runtime. A description of the input parameters and the values expected can be found below.

- **"path/to/file.tsp"** – The location of the source problem file. The format is expected to be the same as the berlin52.tsp file used in this assignment.
- **MAX_POP** – The number of generations that will be generating throughout the execution of the algorithm. An integer value is expected.
- **POP_SIZE** – The size of each population to be generated at each iteration. An integer value is expected.
- **K_VALUE** – The k-value used in the K-Tournament selection during the reproduction stage, ie the size of the tournament. An integer value is expected.
- **MUTATION_RATE** – The rate of occurence of mutations after crossover has occurred

for each individual chromosome. A decimal value in the range [0.0, 1.0] is expected.
- **CROSSOVER_TYPE** – Indicated whether to use Uniform Order Crossover, or Order Crossover. A 0 or 1 is expected to indicate Uniform Order or Order, respectively.
- **CROSSOVER_RATE** – The rate of occurrence of crossover for each pair of sequential chromosomes in the population. A decimal value in the range [0.0, 1.0] is expected.
- **SEED** – The seed value used for the random number generator. Allows for reproduction of executions for experimental analysis. A long integer value is expected.

For the data collected for the plots in this report, the following data was used:

- Mutation Rate = 0.0, Crossover Rate = 1.0
- Mutation Rate = 0.1, Crossover Rate = 1.0
- Mutation Rate = 0.0, Crossover Rate = 0.8
- Mutation Rate = 0.1, Crossover Rate = 0.8
- Mutation Rate = 0.5, Crossover Rate = 0.9

Both population size and number of generations were set to 200, and a k-Value of 3 was used in every instance. These parameters were run five times each for both Uniform Order Crossover, and Order Crossover. The seeds used for the five different runs are 8766553, 7654332, 2356329, 4906877, and 7337890.

The values set for the parameters used in this report are discussed in the Discussions and Conclusions section.

Results

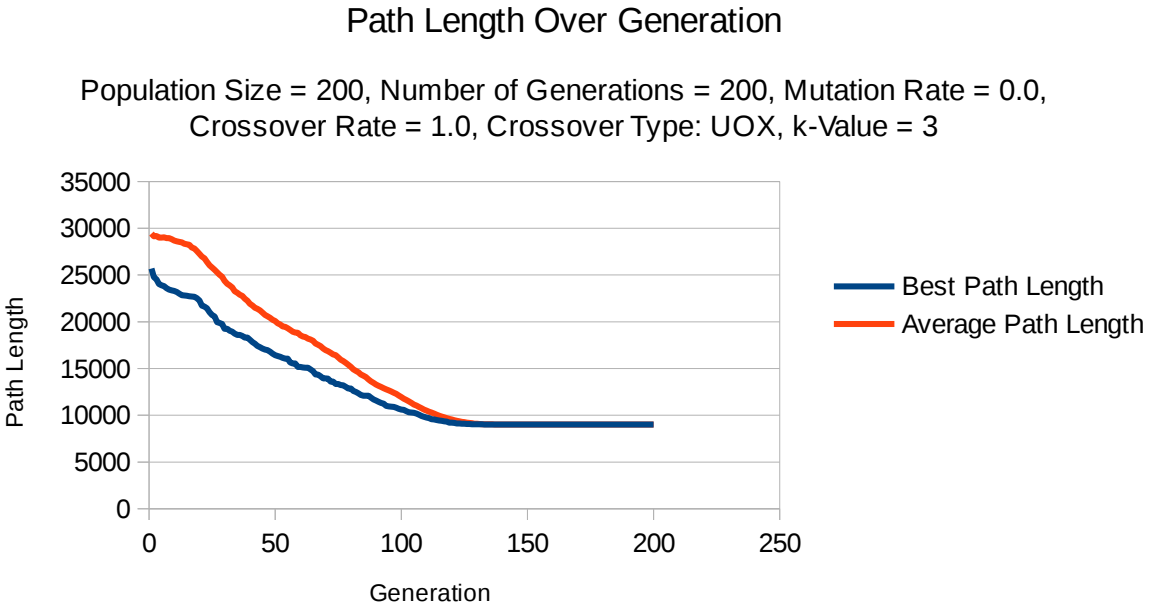
As above, the algorithm was run a total of 5 times for each paramater set. The best path length and the average path lengths for each generation was then averaged across the five iterations and plotted.

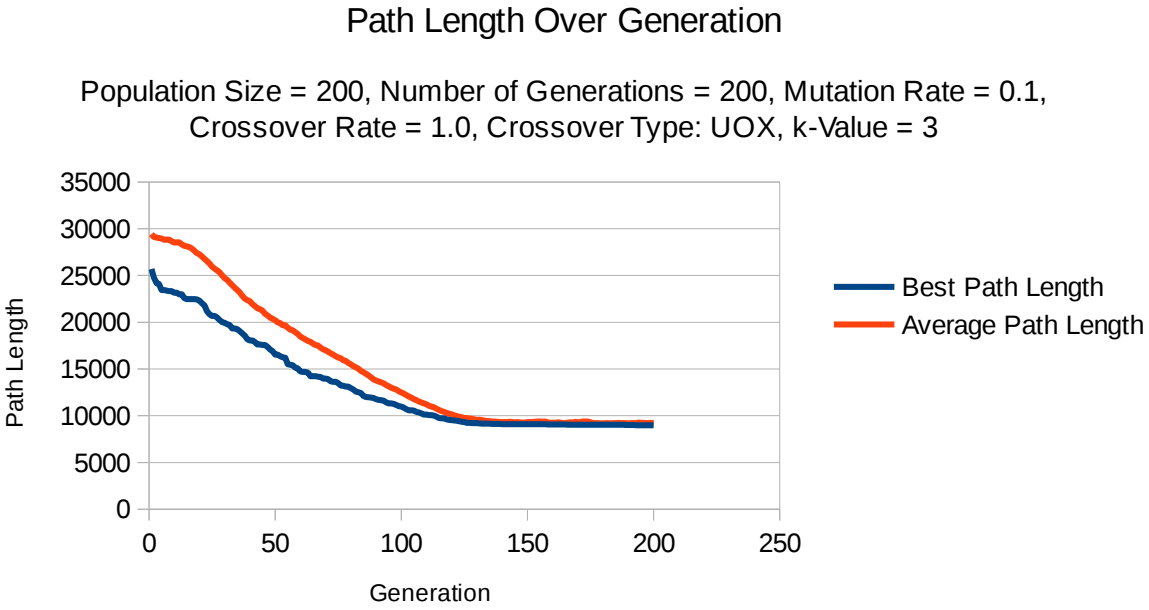
Due to the large size of the data, neither the raw data file or the spreadsheet used to work with the data have been included in this physical report. They have however been included in the electronic submission for this assignment.

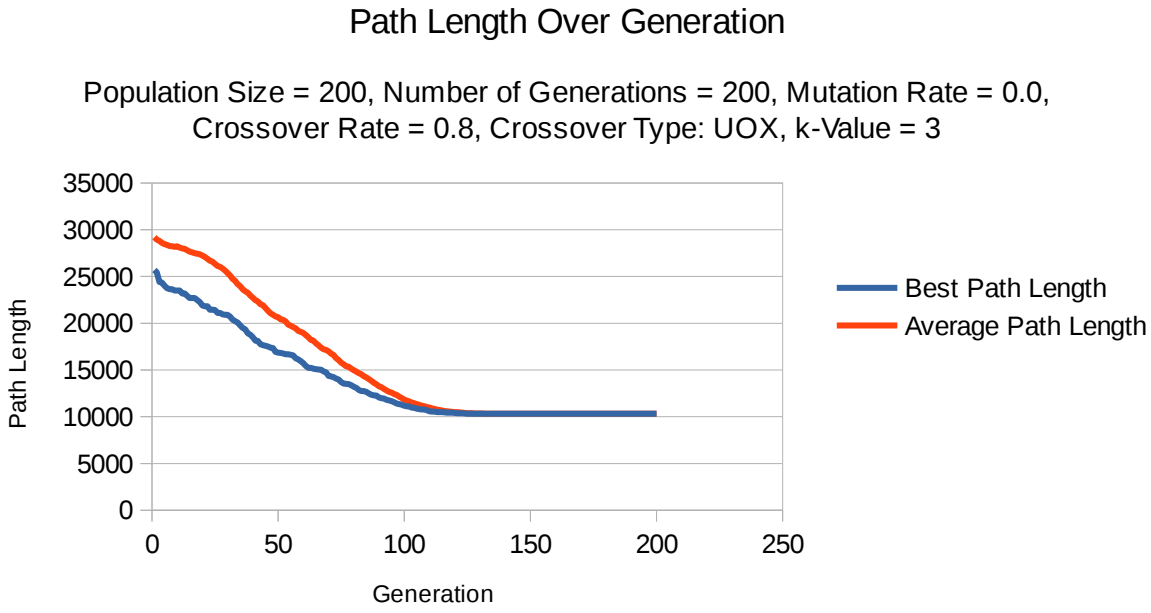
The plots can be seen below in the next section along with their respectice best path length and corresponding chromosome.

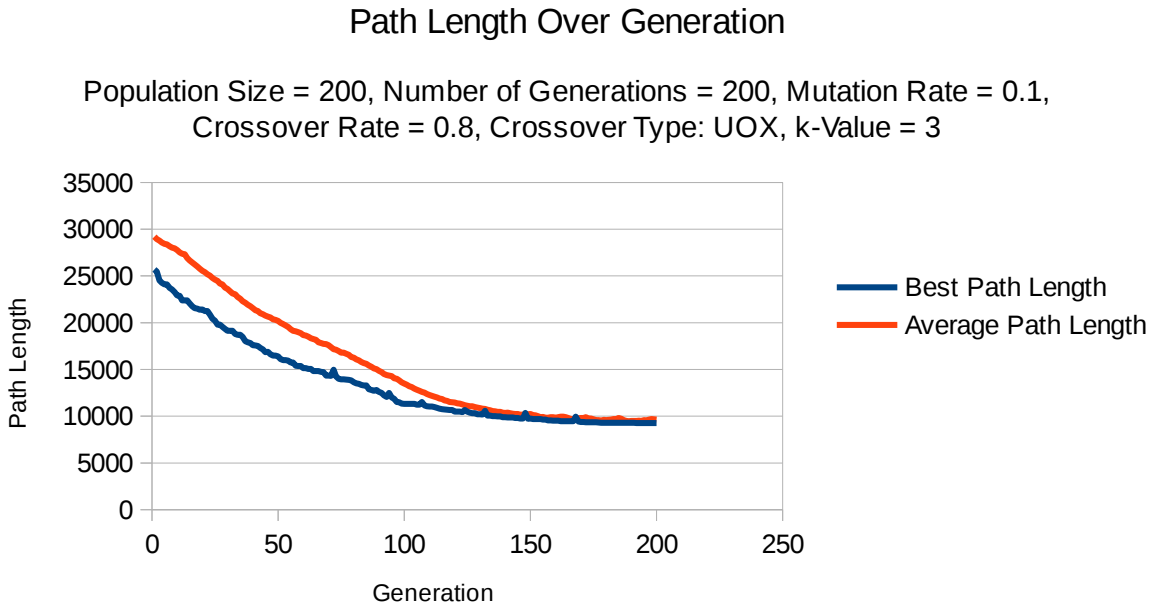
Plots

Below are the plots of the data obtained from running the genetic algorithm. In the sub-headings are the parameters that were used for that run, and in the captions are the best path lengths and their corresponding chromosomes.


Path Length Over Generation
Population Size = 200, Number of Generations = 200, Mutation Rate = 0.0,
Crossover Rate = 1.0, Crossover Type: UOX, k-Value = 3
Path Length
35000
30000
25000
20000
15000
10000
5000
0
0
50
100
150
200
250
Generation
Best Path Length
Average Path Length

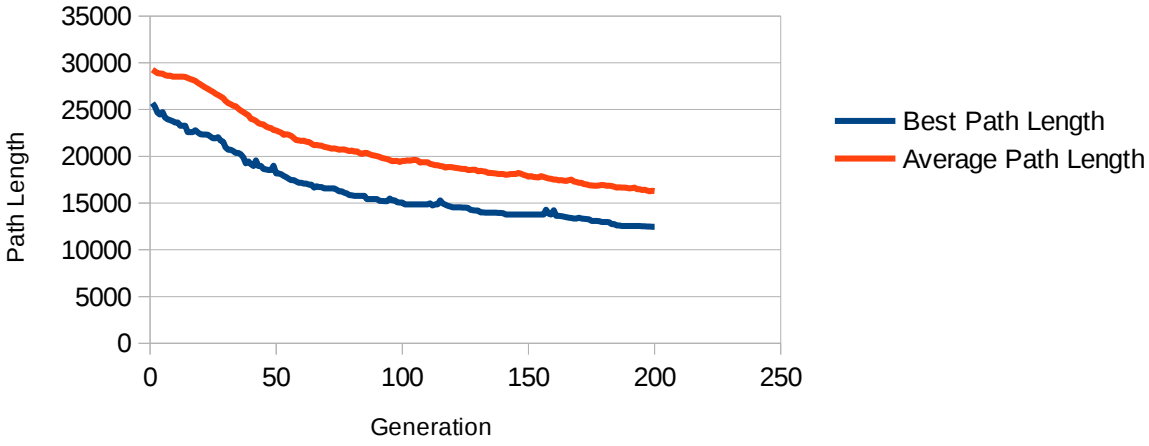

Path Length Over Generation
Population Size = 200, Number of Generations = 200, Mutation Rate = 0.1,
Crossover Rate = 1.0, Crossover Type: UOX, k-Value = 3
Path Length
35000
30000
25000
20000
15000
10000
5000
0
0
50
100
150
200
250
Generation
Best Path Length
Average Path Length


Path Length Over Generation
Population Size = 200, Number of Generations = 200, Mutation Rate = 0.0,
Crossover Rate = 0.8, Crossover Type: UOX, k-Value = 3
Path Length
35000
30000
25000
20000
15000
10000
5000
0
0
50
100
150
200
250
Generation
Best Path Length
Average Path Length


Path Length Over Generation
Population Size = 200, Number of Generations = 200, Mutation Rate = 0.1,
Crossover Rate = 0.8, Crossover Type: UOX, k-Value = 3
Path Length
35000
30000
25000
20000
15000
10000
5000
0
0
50
100
150
200
250
Generation
Best Path Length
Average Path Length

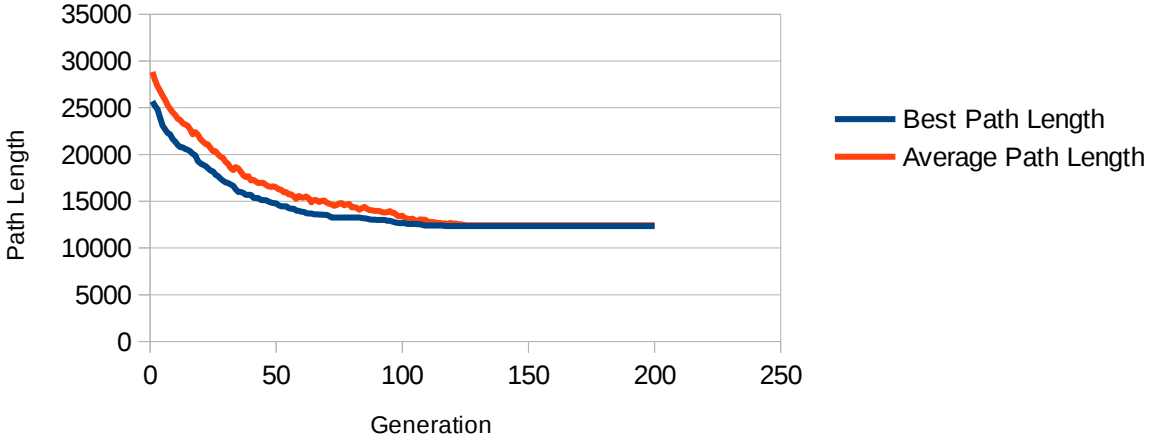
Path Length Over Generation

Population Size = 200, Number of Generations = 200, Mutation Rate = 0.5,
Crossover Rate = 0.9, Crossover Type: UOX, k-Value = 3



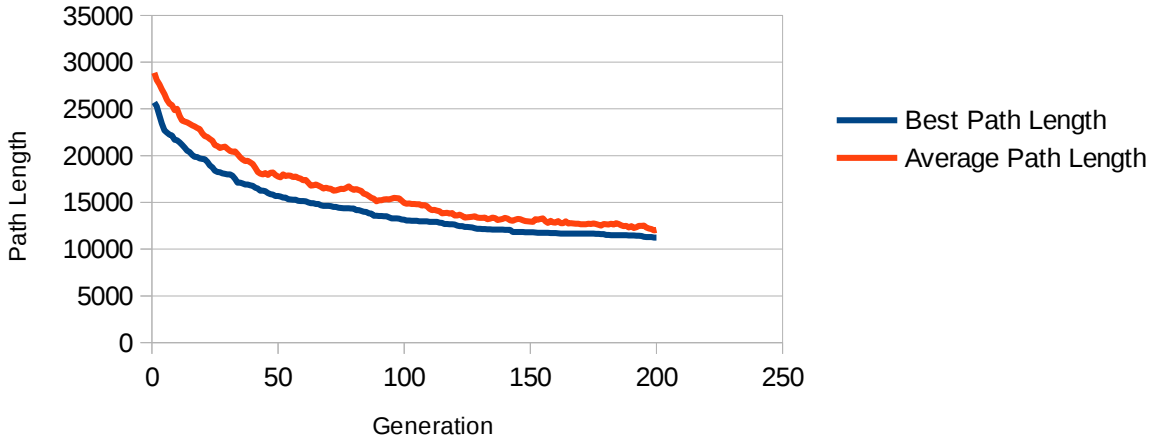
Path Length Over Generation

Population Size = 200, Number of Generations = 200, Mutation Rate = 0.0,
Crossover Rate = 1.0, Crossover Type: OX, k-Value = 3



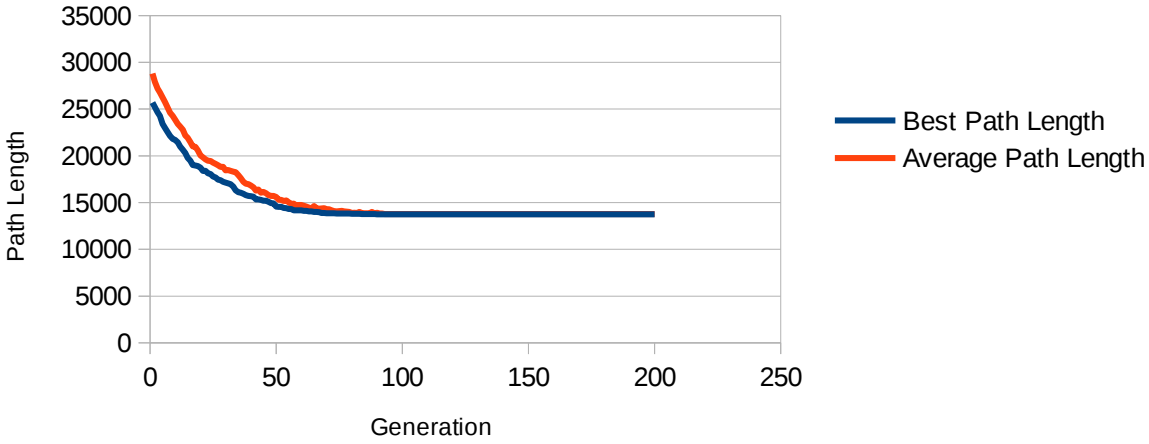
Path Length Over Generation

Population Size = 200, Number of Generations = 200, Mutation Rate = 0.1,
Crossover Rate = 1.0, Crossover Type: OX, k-Value = 3



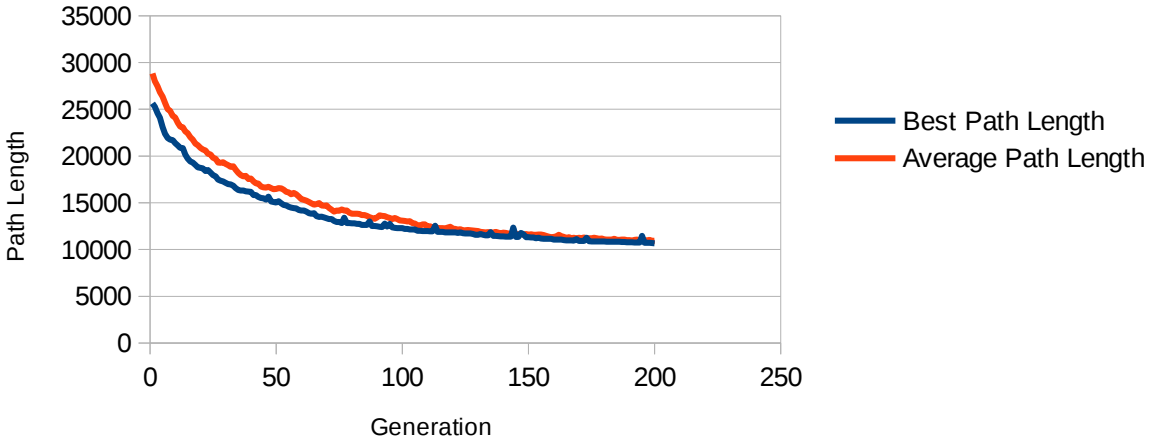
Path Length Over Generation

Population Size = 200, Number of Generations = 200, Mutation Rate = 0.0,
Crossover Rate = 0.8, Crossover Type: OX, k-Value = 3



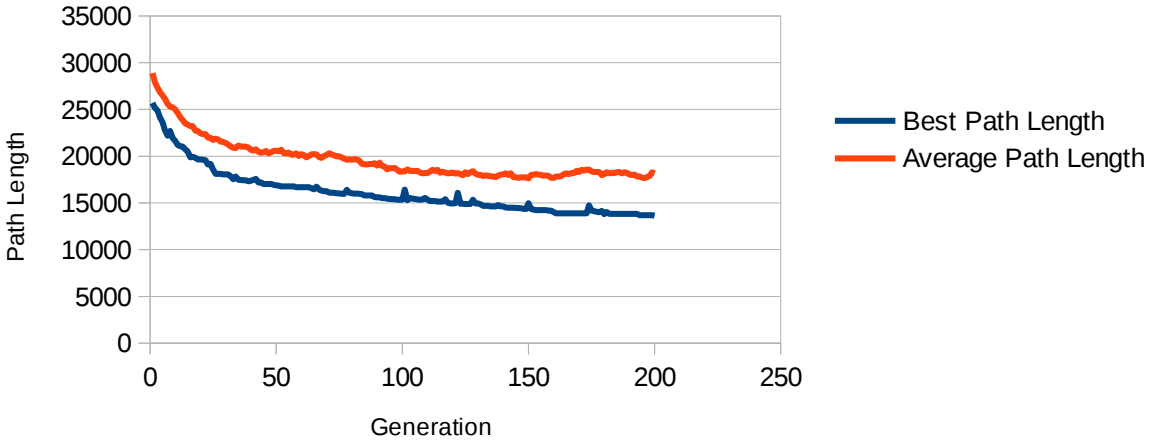
Path Length Over Generation

Population Size = 200, Number of Generations = 200, Mutation Rate = 0.1,
Crossover Rate = 0.8, Crossover Type: OX, k-Value = 3



Path Length Over Generation

Population Size = 200, Number of Generations = 200, Mutation Rate = 0.5,
Crossover Rate = 0.9, Crossover Type: OX, k-Value = 3



Analysis

The graphs shown plots the average generational best path length and average generational average path length of the five iterations over the 200 generations.

The first five plots all use uniform order crossover. The first four all come to similar results. However, the plots using a crossover rate of 1.0 had a lower result in the end.

The last five plots all use order crossover. Plots six through nine also all come to similar results with the plots using a mutation rate of 0.1 having a lower result on average. The five order crossover plots all have seemingly slightly worse results on average than those using uniform order crossover.

The fifth and tenth plots used a high mutation rate and compared to the rest of the results came out with longer path lengths in the end.

Discussion and Conclusions

The graphs obtained show that it would seem Uniform Order Crossover performed better on average than Order Crossover. This is regardless of whether or not there is no mutation or a small amount of mutation. If the mutation rate is too large then the results are significantly worse.

Order crossover came up with worse results on average. It did however perform better with some mutation versus none. This must be due to the fact that order crossover provides less diversity in the reproduction process (a subset from a section rather than dispersed throughout chromosome).

The parameters that were used for the data obtained and that is used in the plots were chosen for several different reasons. Population size was set to 200 as a larger population was shown to yield a lower result in less generations during testing. The number of generations was set to 200 as some iterations seemed to come to result quickly and then plateau around 100 generations, while other seemed to take much longer to plateau. 200 was a good compromise. A k-Value of 3 was used arbitrarily as there was not much of an apparent difference found during testing between values of 2, 3, 4, or 5. k-Values of 1 seemed to not perform as well.

TSPSolve Source Code

Main.java

```
import TSPSolve.Map.Map;
import TSPSolve.TSPSolve;
/**
 * COSC 3P71 - Artificial Intelligence
 * Assignment 2
 * Matt Laidman
 * 5199807, ml12ef
 * November 6, 2015
 */
/**
 * Main class is the command line "wrapper" calling the actual program
 * class, TSPSolve.java, with the parameters passed by the user.
 */
public class Main {
    public static void main(String[] args) {
        if (args.length < 8) {
            System.out.println("Psst! Try:\n");
            System.out.println("\tjava -jar TSPSolve.jar \"path/to/file.tsp\"
MAX_POP POP_SIZE K_VALUE MUTATION_RATE CROSSOVER_TYPE CROSSOVER_RATE, SEED\n");
            System.out.println("\t\tMAX_POP\t\t\tInteger - Maximum generations
before quiting");
            System.out.println("\t\tPOP_SIZE\t\tInteger - Population size of the
chromosomes");
            System.out.println("\t\tK_VALUE\t\t\tInteger - 'k' value used in the
tournament selection");
            System.out.println("\t\tMUTATION_RATE\tDouble - Rate of occurrence of
mutations");
            System.out.println("\t\tCROSSOVER_TYPE\tInteger - 0 for Uniform Order
Crossover, 1 for Order Crossover");
            System.out.println("\t\tCROSSOVER_RATE\tDouble - Rate of occurrence of
crossovers");
            System.out.println("\t\tSEED\tLong - Seed used for random number
generator\n");
            System.out.println("\t\tNote: .tsp file is expected to be in the
following format:\n");
            System.out.println("\t\t\tNAME: berlin52");
            System.out.println("\t\t\tTYPE: TSP");
            System.out.println("\t\t\tCOMMENT: 52 locations in Berlin
(Groetschel)");
            System.out.println("\t\t\tDIMENSION: 52");
            System.out.println("\t\t\tEDGE_WEIGHT_TYPE: EUC_2D");
            System.out.println("\t\t\tNODE_COORD_SECTION");
            System.out.println("\t\t\t1 565.0 575.0");
            System.out.println("\t\t\t2 25.0 185.0");
            System.out.println("\t\t\t3 345.0 750.0");
            System.out.println("\t\t\t...\n");
            System.out.println("\t\tEDGE_WEIGHT_TYPE is not considered in this
implementation");
        } else {
            new TSPSolve(new Map(args[0]), Integer.parseInt(args[1]),
```

```
Integer.parseInt(args[2]), Integer.parseInt(args[3]), Double.parseDouble(args[4]),
Integer.parseInt(args[5]),  Double.parseDouble(args[6]), Long.parseLong(args[7]));
        }
    }
}
```

TSPSolve.java

```
package TSPSolve;
import TSPSolve.Map.City;
import TSPSolve.Map.Map;
import TSPSolve.Map.Path;
import java.util.Random;
/**
 * TSPSolve class is the main program. Class initializes population to passed
parameters
 * and runs until the number of generations has been reached, doing the crossover
and
 * mutation operators each generation.
 */
public class TSPSolve {
    int MAX_POP; // Parameters passed from command line
    int POP_SIZE;
    int K_VALUE;
    double MUTATION_RATE;
    double CROSSOVER_RATE;
    int CROSSOVER_TYPE;
    long SEED;
    int dimension; // number of cities in path
    Map theMap;
    Random rnd;
    public TSPSolve (Map theMap, int MAX_POP, int POP_SIZE, int K_VALUE, double
MUTATION_RATE, int CROSSOVER_TYPE, double CROSSOVER_RATE, long SEED) {
        this.MAX_POP = MAX_POP;
        this.POP_SIZE = POP_SIZE;
        this.K_VALUE = K_VALUE;
        this.MUTATION_RATE = MUTATION_RATE;
        this.CROSSOVER_TYPE = CROSSOVER_TYPE;
        this.CROSSOVER_RATE = CROSSOVER_RATE;
        this.SEED = SEED;
        rnd = new Random(SEED);
        this.theMap = theMap;
        dimension = theMap.getDimension();
        doSolve(initPop(theMap));
    } //constructor
    private void doSolve (Path[] population){ // main "program loop"
        Path bPath = new Path(theMap, dimension);
        Path tBPath;
        double totalLength;
        System.out.println("Number of generations: " + MAX_POP + "\tPopulation
Size: " + POP_SIZE + "\tk-Value: " + K_VALUE);
        System.out.print("Mutation Rate: " + MUTATION_RATE + "\t\tCrossover Type:
");
        if (CROSSOVER_TYPE == 0) {
            System.out.print("UOX");
        } else {
            System.out.print("OX");
        }
        System.out.println("\tCrossover Rate: " + CROSSOVER_RATE);
        System.out.println("Random Number Seed: " + SEED + "\n");
```

```
        for (int cPop = 0 ; cPop < MAX_POP ; cPop++) { // generate "n" generations
            totalLength = 0;
            if (cPop != 0) { // enforce elitism
                for (int i = 0 ; i < dimension ; i++) {
                    if (population[i].getPathLength() == bPath.getPathLength()) {
                        break;
                    } else if (population[i].getPathLength() <
bPath.getPathLength()) {
                        population[i] = bPath;
                        break;
                    }
                }
            }
            tBPath = evalPop(population); // evaluate
            if (cPop == 0 || tBPath.getPathLength() < bPath.getPathLength()) {
                bPath = tBPath;
            }
            population = doMutation(doCrossover(doTourny(population, K_VALUE),
CROSSOVER_TYPE)); // get next gen
            // print
            System.out.print(bPath.getPathLength() + "\t");
            for (int i = 0 ; i < POP_SIZE ; i++) {
                totalLength+=population[i].getPathLength();
            }
            System.out.println(totalLength/POP_SIZE);
        }
        // print
        System.out.println();
        for (int i = 0 ; i < dimension ; i++) {
            System.out.print(bPath.getCity(i).getCityNum() + " ");
        }
        System.out.println();
        System.out.println(bPath.getPathLength());
    }
    private Path[] doMutation (Path[] population) { // Reciprocal exchange mutation
        int i, j;
        City temp;
        for (int p = 0 ; p < POP_SIZE ; p++) { // for each path in pop
            if (rnd.nextDouble() < MUTATION_RATE) {
                i = (int)(rnd.nextDouble()*dimension); // pick 2 random numbers
                j = i;
                while (j == i) { // ensure j and i are different
                    j = (int)(rnd.nextDouble()*dimension);
                }
                temp = population[p].getCity(i); // swap cities in i and j
                population[p].addCity(population[p].getCity(j), i);
                population[p].addCity(temp, j);
            }
        }
        return population;
    }
    private Path[] doCrossover (Path[] population, int CROSSOVER_TYPE) {
        if (CROSSOVER_TYPE == 0) {
            return doUOX(population);
        } else {
```

```
            return doOX(population);
        }
    }
    private Path[] doUOX (Path[] population) {
        Path[] newPop = new Path[POP_SIZE];
        int k;
        for (int i = 0 ; i < POP_SIZE ; i+=2) { // for every 2 chromosomes in
population
            if (rnd.nextDouble() < CROSSOVER_RATE) {
                newPop[i] = new Path(theMap, dimension);
                newPop[i+1] = new Path(theMap, dimension);
                for (int j = 0 ; j < dimension ; j++) { // for each city in path
                    if ((int)(rnd.nextDouble()*2) == 1) { // if random "mask"
generates 1
                        // copy city to new index for both ( 1 to 1, 2 to 2)
                        newPop[i].addCity(population[i].getCity(j), j);
                        newPop[i+1].addCity(population[i+1].getCity(j), j);
                    }
                }
                k = 0;
                for (int j = 0 ; j < dimension ; j++) { // for every city in old
path 1
                    if (!newPop[i+1].contains(population[i].getCity(j))) { // if it
doesnt exist in new path 2
                        while (k < dimension && newPop[i+1].getCity(k) != null)
{ // add to first available place
                            k++;
                        }
                        if (newPop[i+1].getCity(k) == null) {
                            newPop[i+1].addCity(population[i].getCity(j), k);
                        }
                    }
                }
                k = 0;
                for (int j = 0 ; j < dimension ; j++) { // for every city in old
path 2
                    if (!newPop[i].contains(population[i+1].getCity(j))) { // if it
doesnt exist in new path 1
                        while (k < dimension && newPop[i].getCity(k) != null) { //
add to first available place
                            k++;
                        }
                        if (newPop[i].getCity(k) == null) {
                            newPop[i].addCity(population[i+1].getCity(j), k);
                        }
                    }
                }
            } else {
                newPop[i] = population[i];
                newPop[i+1] = population[i+1];
            }
        }
        return newPop;
    }
    private Path[] doOX (Path[] population) {
```

```
        Path[] newPop = new Path[POP_SIZE];
        int setSize = (int)(rnd.nextDouble()*dimension);
        int setStart = (int)(rnd.nextDouble()*dimension);
        int k;
        for (int i = 0 ; i < POP_SIZE ; i+=2) { // for every 2 chromosomes in
population
            if (rnd.nextDouble() < CROSSOVER_RATE) {
                newPop[i] = new Path(theMap, dimension);
                newPop[i+1] = new Path(theMap, dimension);
                for (int j = setStart ; j < setStart+setSize ; j++) {
                    newPop[i].addCity(population[i].getCity(j%dimension), j
%dimension); // copy from parent 1 to child 1
                    newPop[i+1].addCity(population[i+1].getCity(j%dimension), j
%dimension); // copy from parent 2 to child 2
                }
                k = setStart+setSize;
                for (int j = setStart+setSize ; j < setStart+setSize+dimension ; j+
+) { // for every remaining city in old 1
                    if (!newPop[i+1].contains(population[i].getCity(j%dimension)))
{ // copy to next spot in new 2 if it doesnt exist
                        while (newPop[i+1].getCity(k%dimension) != null) {
                            k++;
                        }
                        newPop[i+1].addCity(population[i].getCity(j%dimension), k
%dimension);
                        k++;
                    }
                }
                k = setStart+setSize;
                for (int j = setStart+setSize ; j < setStart+setSize+dimension ; j+
+) { // for every remaining city in old 2
                    if (!newPop[i].contains(population[i+1].getCity(j%dimension)))
{ // copy to next spot if it doesnt exist in new
                        while (newPop[i].getCity(k%dimension) != null) {
                            k++;
                        }
                        newPop[i].addCity(population[i+1].getCity(j%dimension), k
%dimension);
                        k++;
                    }
                }
            } else {
                newPop[i] = population[i];
                newPop[i+1] = population[i+1];
            }
        }
        return newPop;
    }
    private Path[] doTourny (Path[] population, int kValue) { // k-tournament
        Path[] tourny = new Path[kValue];
        Path[] newPop = new Path[POP_SIZE];
        Path tBestPath = new Path(theMap, dimension);
        double tLength, tBestLength;
        for (int i = 0 ; i < POP_SIZE ; i++) { // for each path in population
            newPop[i] = new Path(theMap, dimension);
```

```
            for (int k = 0 ; k < kValue ; k++) { // pick k randomly
                    tourny[k] = population[(int)(rnd.nextDouble()*POP_SIZE)];
            }
            tBestLength = -1;
            for (int k = 0 ; k < kValue ; k++) { // for each of k random paths
                if (tBestLength == -1) { // if first, its best so far...
                    tBestPath = tourny[k];
                    tBestLength = tourny[k].getPathLength();
                } else { // otherwise compare
                    tLength = tourny[k].getPathLength();
                    if (tLength < tBestLength) {
                        tBestPath = tourny[k];
                        tBestLength = tLength;
                    }
                }
            }
            newPop[i] = tBestPath; // add best to new
        }
        return newPop;
    }
    private Path evalPop (Path[] population) { // get the best city from the
population
        Path tBestPath  = new Path (theMap, dimension);
        double tLength, tBestLength = -1;
        for (int i = 0 ; i < POP_SIZE ; i++) {
            if (tBestLength == -1) {
                tBestPath = population[i];
                tBestLength = population[i].getPathLength();
            } else {
                tLength = population[i].getPathLength();
                if (tLength < tBestLength) {
                    tBestPath = population[i];
                    tBestLength = tLength;
                }
            }
        }
        return tBestPath;
    }
    private Path[] initPop (Map theMap) { // initializes the population randomly
        Path[] population = new Path[POP_SIZE];
        int index;
        for (int i = 0 ; i < POP_SIZE ; i++) { // for each path
            population[i] = new Path(theMap, theMap.getDimension());
            for (int j = 0 ; j < dimension ; j++) { // for each city in path
                index = (int)(1+rnd.nextDouble()*dimension);
                while (population[i].contains(theMap.getCity(index))) { // add a
random unused city
                    index = (int)(1+rnd.nextDouble()*dimension);
                }
                population[i].addCity(theMap.getCity(index), j);
            }
        }
        return population;
    }
}
```

City.java

```
package TSPSolve.Map;
public class City {
    private int cityNum;
    private double x, y;
    public City (int cityNum, double x, double y) {
        this.cityNum = cityNum;
        this.x = x;
        this.y = y;
    }
    public int getCityNum ( ) {return cityNum;}
    public double getX ( ) {return x;}
    public double getY ( ) {return y;}
}
```

Map.java

```
package TSPSolve.Map;
import java.io.BufferedReader;
import java.io.FileReader;
public class Map {
    private int dimension;
    private City[] theMap;
    public Map (String fileName) {
        genMap(fileName);
    }
    public City getCity (int cityNum) {return theMap[cityNum-1];}
    public double getDistance (int c1, int c2) {
        // Return distance between two cities (coordinates)
        return Math.sqrt(Math.pow(theMap[c2-1].getX()-theMap[c1-1].getX(),
2)+Math.pow(theMap[c2-1].getY()-theMap[c1-1].getY(), 2));
    }
    public int getDimension ( ) {return dimension;}
    private void genMap (String fileName) {
        String line;
        int ni, xi, yi;
        int cityNum;
        double x, y;
        try {
            FileReader fr = new FileReader(fileName);
            BufferedReader br = new BufferedReader(fr);
            br.readLine();
            br.readLine();
            br.readLine();
            line = br.readLine();
            dimension = Integer.parseInt(line.substring(11, line.length()));
            br.readLine();
            br.readLine();
            theMap = new City[dimension];
            while((line = br.readLine()) != null) {
                if (line.equals("EOF")) {
                    break;
                }
                ni = 0;
                while (line.charAt(ni) != ' '){
                    ni++;
                }
                xi = ni + 1;
                while (line.charAt(xi) != ' '){
                    xi++;
                }
                yi = xi + 1;
                while (yi != line.length()){
                    yi++;
                }
                cityNum = Integer.parseInt(line.substring(0, ni));
                x = Double.parseDouble(line.substring(ni + 1, xi));
                y = Double.parseDouble(line.substring(xi+1, yi));
                addCity(new City(cityNum, x, y));
```

```
            }
            br.close();
        } catch(Exception e) {
            System.out.println("Invalid problem source file.");
            e.printStackTrace();
        }
    }
    private void addCity (City c) {
        int cityNum = c.getCityNum();
        if (cityNum > dimension) { // Check that city number is within dimension
(array bounds)
            throw new CityNumOutOfBounds();
        } else {
            theMap[cityNum-1] = c; // Cities are mapped to their number - 1
        }
    }
    private class CityNumOutOfBounds extends RuntimeException {
        CityNumOutOfBounds ( ) {
            super("City Number out of array bounds.");
        }
    }
}
```

Path.java

```
package TSPSolve.Map;
public class Path {
    private Map theMap;
    private City[] thePath;
    public Path (Map theMap, int dimension) {
        this.theMap = theMap;
        thePath = new City[dimension];
    }
    public void addCity (City c, int index) {
        thePath[index] = c;
    }
    public City getCity (int index) {return thePath[index];}
    public double getPathLength ( ) {
        double length = 0;
        for (int i = 0 ; i < thePath.length-1 ; i++) {
            length = length + theMap.getDistance(thePath[i].getCityNum(),
thePath[i+1].getCityNum());
        }
        length = length + theMap.getDistance(thePath[thePath.length-
1].getCityNum(), thePath[0].getCityNum());
        return length;
    }
    public boolean contains (City c) {
        for (City city : thePath) {
            if (city != null && c.getCityNum() == city.getCityNum()) {
                return true;
            }
        }
        return false;
    }
}
```