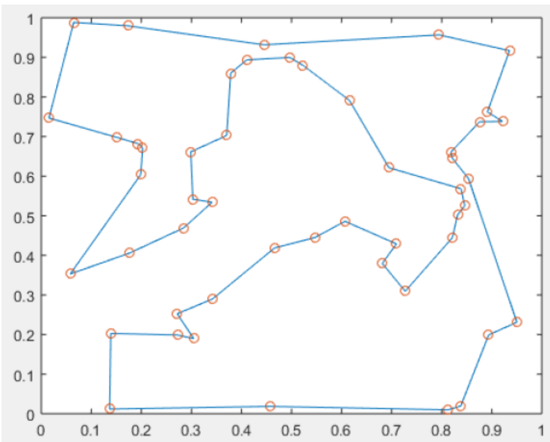
Using Genetic Algorithms to Find the Shortest & Longest Path in the Traveling Salesman Problem

By Perrin Jones (paj2117)

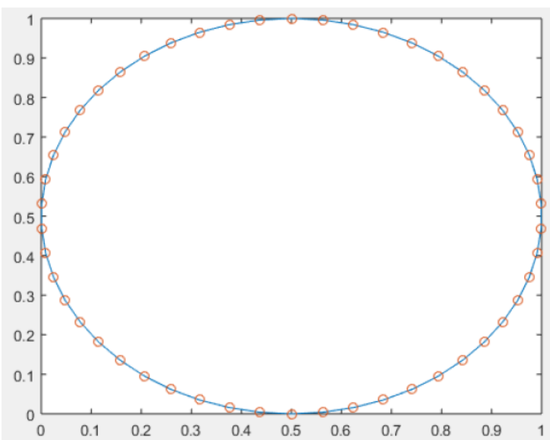


Data Set #1

Shortest Distance: 6.210332

Evaluations: 5,000,000

Clock Time: 6.3 seconds

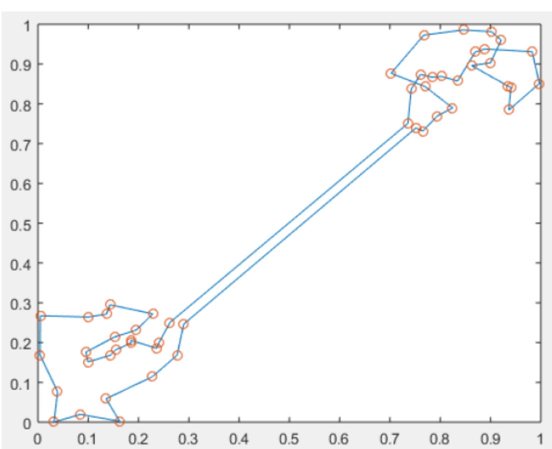


Data Set #2

Shortest Distance: 3.139579

Evaluations: 5,000,000

Clock Time: 5.4 seconds



Data Set #3

Shortest Distance: 4.122337

Evaluations: 5,000,000

Clock Time: 5.7 seconds

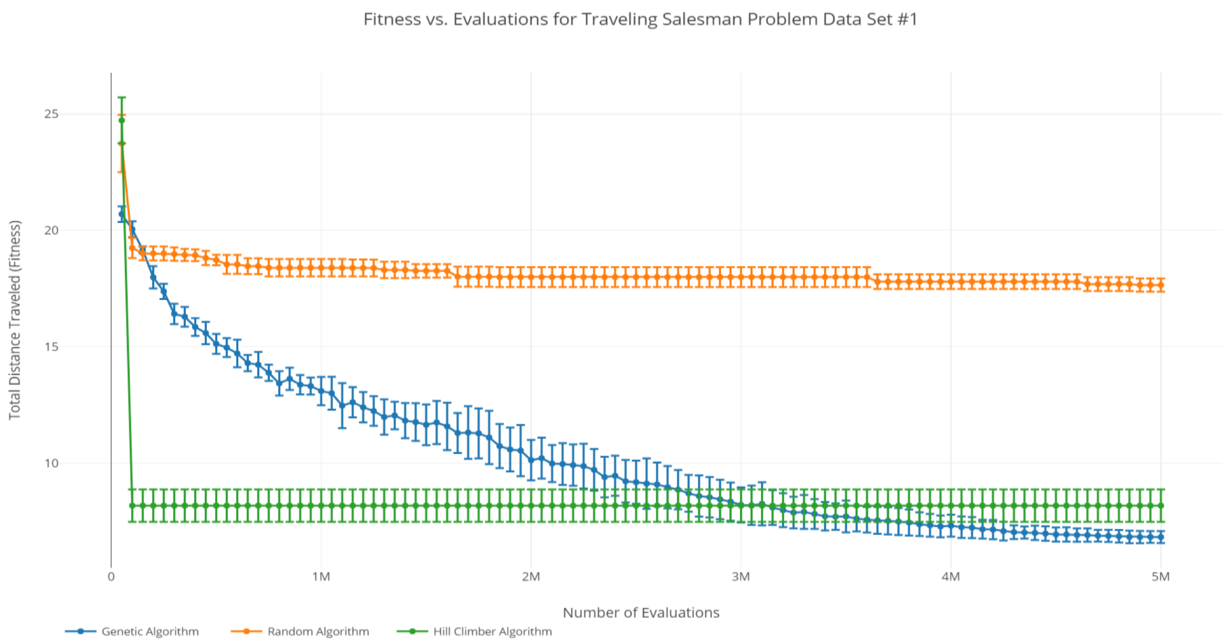
Data Set #1

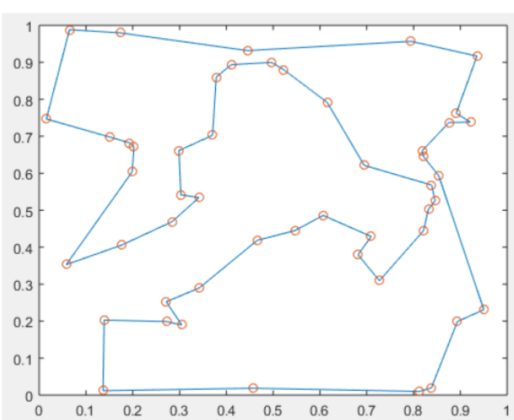
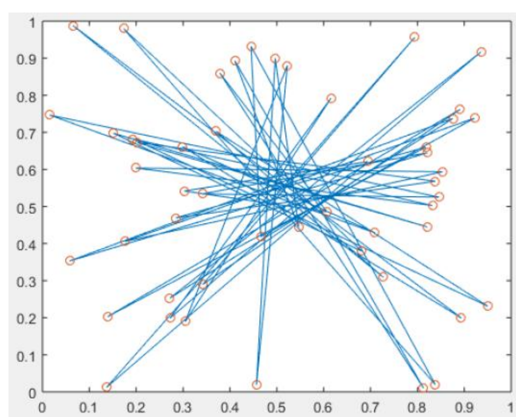
The genetic algorithm represented the cities as a set of points in an array, where the order of the array is the order the cities are visited. A population of these paths is put together, storing a generation's worth of possible paths. In testing, the best results came with a population of 1,000 and 100 generation runs. After that, any larger values did not produce results that were more optimal.

When evolving a population, crossover was performed by selecting a set of high performing parents in the population. A set of 10 parents would be randomly selected at a time, and the top, based on fitness, would move on to crossover. A set of two parents is used to create a child. A random range of cities is selected from the first parent and moved to the child in the same indices as the first parent. The cities not already represented in the child are loaded into the child based on the order they appear in the second parent. This child is added to the new population for the next generation

There is also the opportunity for mutation in every generation. Given a mutation rate, 0.1% yielded the best results in testing for this specific data set; 0.1% of each population is mutated each generation. Mutation is represented by a random swap in cities within a specific path.

In the data set below, we can see that the genetic algorithm successfully surpassed the hill climber algorithm after around 3 million evaluations, giving a more accurate estimate of the shortest path. The random generation algorithm did not compete with either algorithms after 10,000 evaluations. The points are dispersed almost evenly in the space, so the genetic algorithm takes longer to evolve a solution, since there isn't a specific set of characteristics (like two clusters of points together) that drive the shortest path.



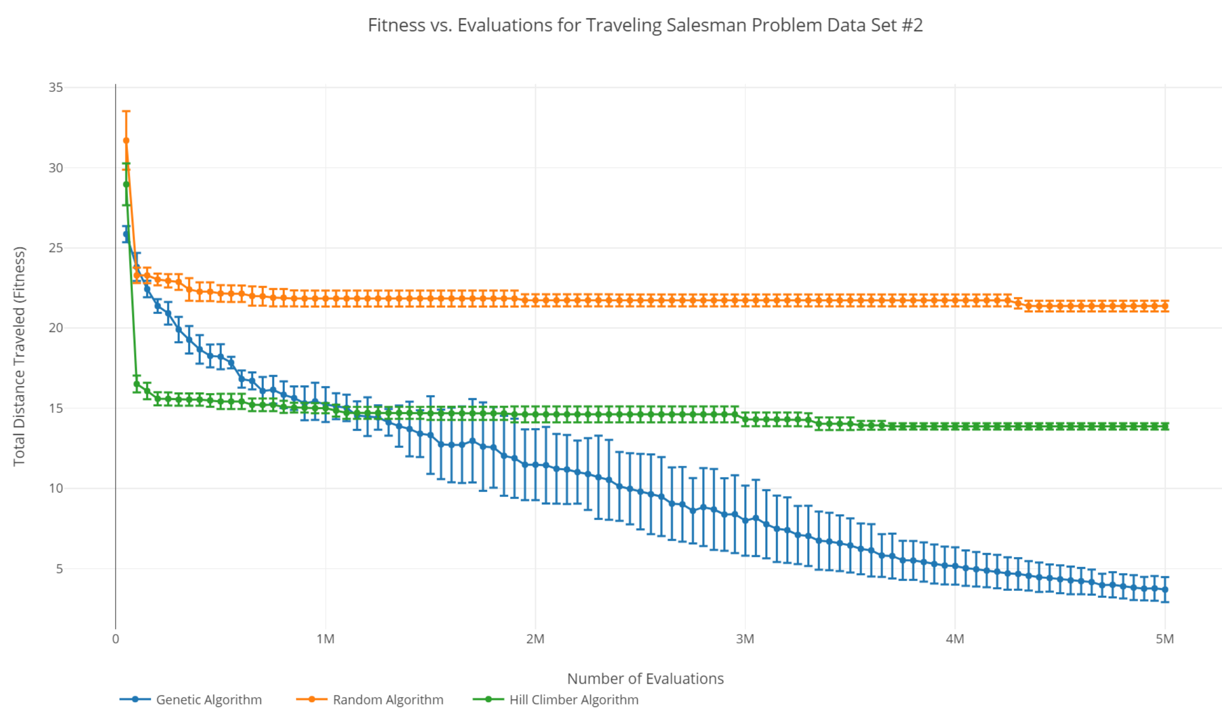
Data Set #2

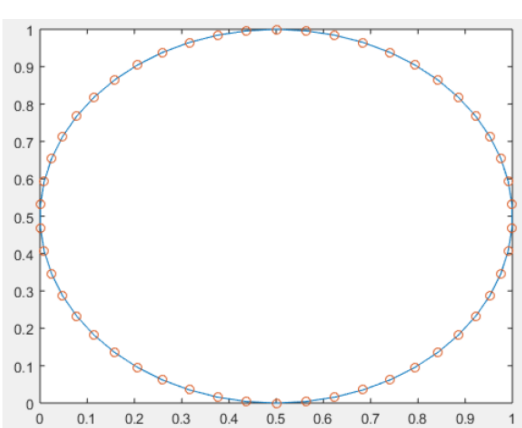
The genetic algorithm represented the cities as a set of points in an array, where the order of the array is the order the cities are visited. A population of these paths is put together, storing a generation's worth of possible paths. In testing, the best results came with a population of 1,000 and 100 generation runs. After that, any larger values did not produce results that were more optimal.

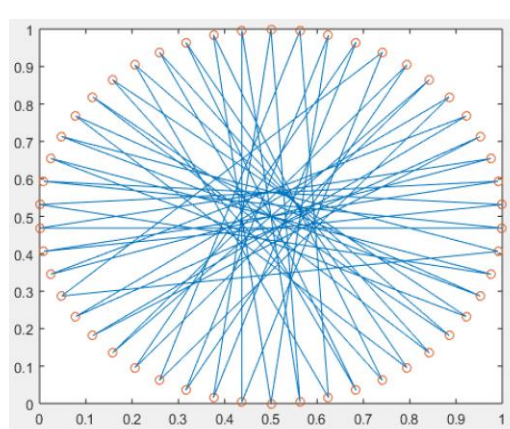
When evolving a population, crossover was performed by selecting a set of high performing parents in the population. A set of 10 parents would be randomly selected at a time, and the top, based on fitness, would move on to crossover. A set of two parents is used to create a child. A random range of cities is selected from the first parent and moved to the child in the same indices as the first parent. The cities not already represented in the child are loaded into the child based on the order they appear in the second parent. This child is added to the new population for the next generation

There is also the opportunity for mutation in every generation. Given a mutation rate, 3% yielded the best results in testing for this specific data set; 3% of each population is mutated each generation. Mutation is represented by a random swap in cities within a specific path.

In the data set below, we can see that the genetic algorithm successfully surpassed the hill climber algorithm after around 1 million evaluations, giving a more accurate estimate of the shortest path. The random generation algorithm did not compete with either algorithms after 10,000 evaluations. The points are dispersed in an almost perfect circle, giving the genetic algorithm a greater opportunity to evolve sets of points very close to each other. No set of points must visit a point farther away than their nearest neighbor. The high volume of close pairs of points survive evolution more easily due to their high fitness, therefore the genetic algorithm is more efficient than in Data Set #1.







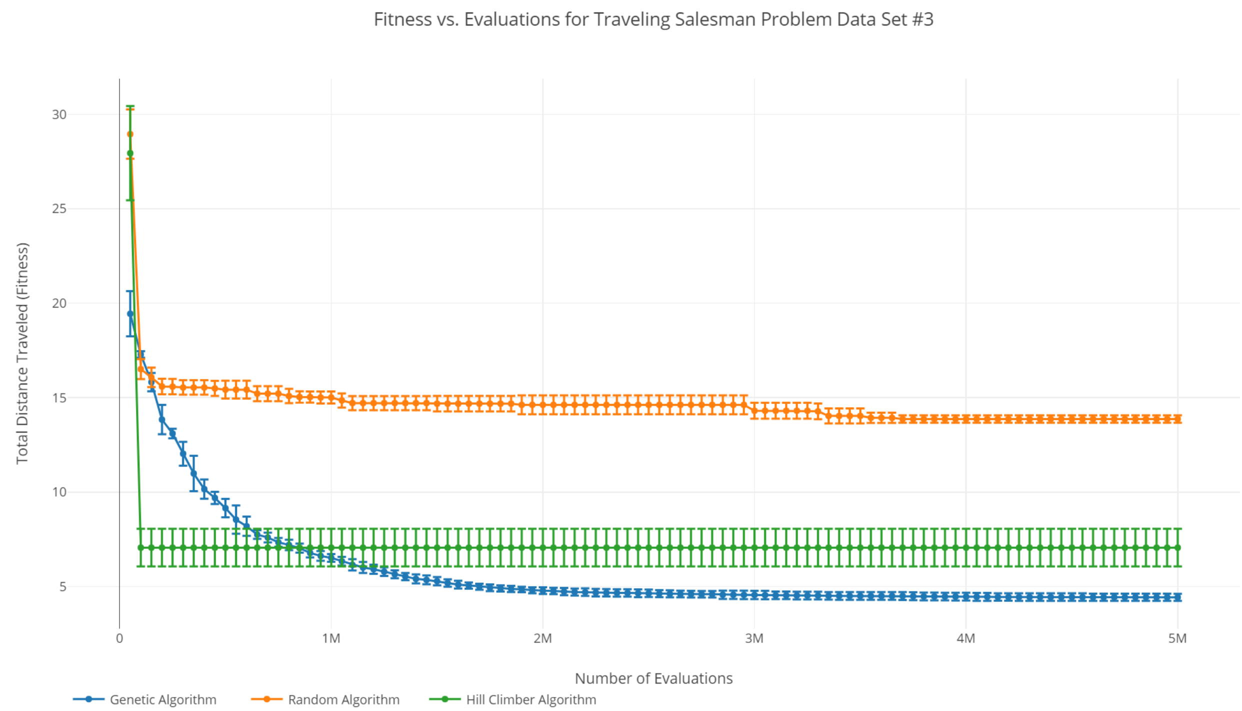
Data Set #3

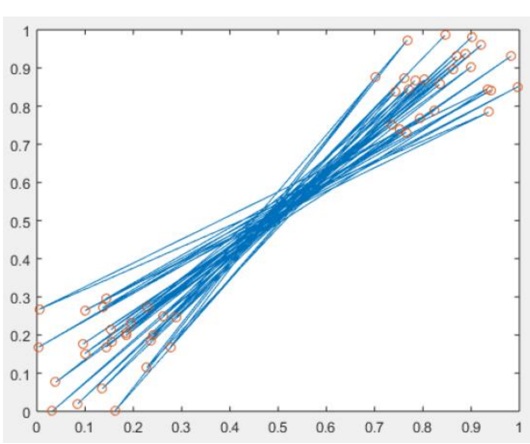
The genetic algorithm represented the cities as a set of points in an array, where the order of the array is the order the cities are visited. A population of these paths is put together, storing a generation's worth of possible paths. In testing, the best results came with a population of 1,000 and 100 generation runs. After that, any larger values did not produce results that were more optimal.

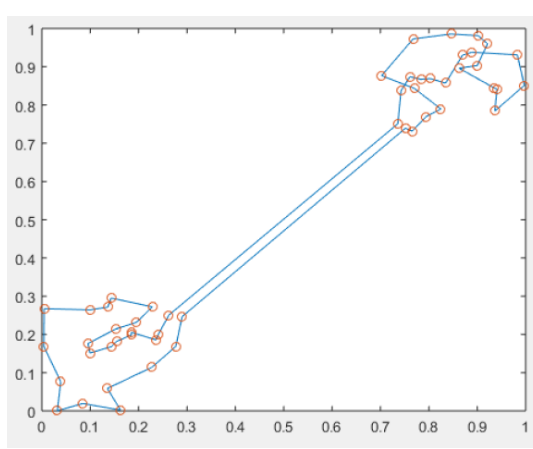
When evolving a population, crossover was performed by selecting a set of high performing parents in the population. A set of 10 parents would be randomly selected at a time, and the top, based on fitness, would move on to crossover. A set of two parents is used to create a child. A random range of cities is selected from the first parent and moved to the child in the same indices as the first parent. The cities not already represented in the child are loaded into the child based on the order they appear in the second parent. This child is added to the new population for the next generation

There is also the opportunity for mutation in every generation. Given a mutation rate, 1% yielded the best results in testing for this specific data set; 1% of each population is mutated each generation. Mutation is represented by a random swap in cities within a specific path.

In the data set below, we can see that the genetic algorithm successfully surpassed the hill climber algorithm after around 1 million evaluations, giving a more accurate estimate of the shortest path. The random generation algorithm did not compete with either algorithms after 10,000 evaluations. The points are closely dispersed in two clusters; thus, for the genetic algorithm the evolving population allows for the successful survival a path that only crosses clusters twice. The genetic algorithm closes in on a solution quickly as in Data Set #2, due to the clear high fitness paths. The algorithm then evens out as it tries different crossings between clusters and shorter paths within each cluster.







Appendix: Java Code

Path.java

```
import java.util.ArrayList;

import java.util.Collections;

import java.util.List;


public class Path {

	//Keep an instance of the points

	private ArrayList<Point> path = new ArrayList<Point>();

	//keep values stored so that I don't have to unnnecesarily reevaluate them

	private double distance = 0;

	private double fitness = 0;



	public Path (ArrayList<Point> newPath) {

		path = newPath;

		//Shuffle points so that each path generated it unique

		Collections.shuffle(path);

	}


	public Point getPoint(int pos) {

		return path.get(pos);

	}


	public void setPoint(int pos, Point point) {

		path.set(pos, point);

		this.resetValues();

	}


	//Returns distance of all the points

	public double getDistance() {

		if(distance == 0) {

			//Include distance between first and last point

			distance = path.get(0).getDistance(path.get(path.size()-1));

			//Iterate through all distances

			for (int i = 0; i < path.size()-1; i++) {

				distance += path.get(i).getDistance(path.get(i+1));

			}
```

```
		}
		return distance;
	}

	//Returns fitness of path for comparing with a genetic algorithm
	public double getFitness() {
		if(fitness == 0) {
			fitness = 1/this.getDistance(); //1 - 1/this.getDistance() for longest path
		}
		return fitness;
	}

	//Resets saved distance and fitness
	private void resetValues() {
		distance = 0;
		fitness = 0;
	}

	//Reshuffles path
	public ArrayList<Point> shufflePath() {
		Collections.shuffle(path);
		return path;
	}

	public int getPathLength() {
		return path.size();
	}

	//Completely clears a path for loading new information
	public void clearPath() {
		for(int i = 0; i < this.getPathLength(); i++) {
			this.path.set(i, null);
		}
	}

	//Checks if a point is within the path
	public boolean contains(Point point) {
```

```java
		return path.contains(point);
	}

	//Print all the points in the path
	public void print() {
		for(int i = 0; i < path.size(); i++) {
			System.out.println(path.get(i));
		}
	}

	//Swap points in path at index a and b
	public void swap(int a, int b) {
		Point temp = path.get(a);
		path.set(a, path.get(b));
		path.set(b, temp);
		this.resetValues();
	}
}
```

Population.java

```java
import java.util.ArrayList;
import java.util.Collections;

public class Population {
	//Stores a population of different paths
	//Population is a set size in this instance
	private Path[] paths;
	//Storage of points loaded from file
	private ArrayList<Point> points = new ArrayList<Point>();

	public Population(int size, ArrayList<Point> newPoints) {
		paths = new Path[size];
		points = newPoints;

		//Create a new randomized population
		//Populates it upon initialization
		for(int i = 0; i < size; i++) {
			ArrayList<Point> temp = (ArrayList<Point>) points.clone();
```

```java
                Collections.shuffle(temp);
                Path newPath = new Path(temp);
                paths[i] = newPath;
            }
        }

        public Path getPath(int pos) {
            return paths[pos];
        }

        public void setPath(int pos, Path newPath) {
            paths[pos] = newPath;
        }

        public int getPathsSize() {
            return paths.length;
        }

        //Searches for the most fit path and returns it
        public Path searchFit() {
            Path top = paths[0];
            for(int i = 1; i < this.getPathsSize(); i++) {
                if(top.getFitness() < paths[i].getFitness()) {
                    top = paths[i];
                }
            }
            return top;
        }
}
```

EvaluationManager.java

```java
//Manages counting the number of evaluations when the algorithm runs
public class EvaluationManager {
        static int numEval = 0;

        //bump increases the evaluation number tally by 1
        public static void bump() {
            numEval++;
```

```java
	}

	//prints the number of evaluations
	public static String print() {
		return "" + numEval;
	}
}
```

FileImpot.java

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Scanner;

public class FileImport {
	private Scanner reader;

	public FileImport (String filename) {
		//Reads file and stores in reader
		File newFile = new File(filename);
		try {
			reader = new Scanner(newFile);
		} catch (FileNotFoundException e) {
			System.out.println("ERROR WITH READING FILE");
			e.printStackTrace();
		}
	}

	public ArrayList<Point> getData() {
		//New array list to hold points
		ArrayList<Point> data = new ArrayList<Point>();

		//Loads array with doubles from points
		while(reader.hasNextDouble()) {
			double x = reader.nextDouble();
			double y = reader.nextDouble();
			data.add(new Point(x,y));
```

```
        }

        //Limits data by 50, since there are only 50 points in the file given
        //Had trouble with null points coming up after 50
        ArrayList<Point> finalData = new ArrayList<Point>(50);
        for (int i = 0; i < 50; i++) {
            finalData.add(data.get(i));
        }
        return finalData;
    }
}
```

Point.java

```
public class Point {
    private double x;
    private double y;

    //Stores x,y coordinate
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

    //gets distance away from another point
    public double getDistance(Point other) {
        EvaluationManager.bump();
        return Math.sqrt( Math.pow((this.getX()-other.getX()),2) + Math.pow((this.getY()-
other.getY()),2) );
    }
```

```
        @Override
        public String toString() {
            return x + "\t" + y;
        }
}
```

Algorithm.java

```
import java.util.ArrayList;
import java.util.Random;

public class Algorithm {
        //Variable to control how often mutation occurs
        private double numMutate;
        //Number of paths to compete
    private int numTourn;
    //Storage of points
    private ArrayList<Point> points;
    Random rand = new Random();

    public Algorithm(double mutate, int tourn, ArrayList<Point> newPoints) {
        //Load variables
        numMutate = mutate;
        numTourn = tourn;
        points = newPoints;
    }

    //Runs an evolution and returns the evolved population
    public Population evolve (Population population) {
        //Clone points to disconnect data structures
            ArrayList<Point> tempPoints = (ArrayList<Point>) points.clone();

            //Create a new population to be created from crossover
        Population nextPopulation = new Population(population.getPathsSize(), tempPoints);

        //Run a crossover after selecting a set of parents in a tournement
        for(int i = 0; i < nextPopulation.getPathsSize(); i ++) {
            Path path1 = compete(population);
            Path path2 = compete(population);
```

```
            Path pathHybrid = this.crossover(path1, path2);
            nextPopulation.setPath(i, pathHybrid);
        }

        //Control the mutation for every population set
        for(int j = 0; j < nextPopulation.getPathsSize(); j ++) {
            nextPopulation.setPath(j, this.mutate(nextPopulation.getPath(j)));
        }

        //Return new population
        return nextPopulation;
    }

    private Path crossover(Path path1, Path path2) {
            ArrayList<Point> tempPoints = (ArrayList<Point>) points.clone();
            //Create a new hybrid path to be the result of crossover
        Path pathHybrid = new Path(tempPoints);
        //Clear path to be loaded
        pathHybrid.clearPath();

        //Randomly choose where crossover starts and ends
        int start = (int) (rand.nextDouble() * path1.getPathLength());
        int stop = (int) (rand.nextDouble() * path1.getPathLength());

        //If start is larger than stop, swap them
        if(start > stop) {
            //swap
            int temp = start;
            start = stop;
            stop = temp;
        }

        //cross over from path1
        for(int i = 0; i < pathHybrid.getPathLength(); i++) {
            if(i > start && i < stop) {
                    pathHybrid.setPoint(i, path1.getPoint(i));
            }
        }
```

```
        //cross over what has not already been crossed over in the order in path2
        for(int i = 0; i < path2.getPathLength(); i++) {
            if(!pathHybrid.contains(path2.getPoint(i))) {
                for(int j = 0; j < pathHybrid.getPathLength(); j++) {
                    if(pathHybrid.getPoint(j) == null) {
                        pathHybrid.setPoint(j, path2.getPoint(i));
                        break;
                    }
                }
            }
        }

        return pathHybrid;
    }

    private Path compete(Population population) {
        //Selection process
            ArrayList<Point> tempPoints = (ArrayList<Point>) points.clone();
        Population tournement = new Population(numTourn, tempPoints);
        //Create a new population of a certain number and populate it with random spots in the
current population
        String help = "";
        for(int i = 0; i < numTourn; i++) {
            int spot = (int) (rand.nextDouble() * population.getPathsSize());
            tournement.setPath(i, population.getPath(spot));
            help += spot + ",";
        }
        //Return the best fitness path of this mini-population
        return tournement.searchFit();
    }

    private Path mutate(Path path) {
        //Runs a mutation if random double is less than numMutate
        Path newPath = path;
        for(int i = 0; i < newPath.getPathLength(); i++) {
            if(numMutate > rand.nextDouble()) {
                //If mutation is true, then randomly swap a random set of points in the
path
```

```java
                    int j = (int) (rand.nextDouble() * newPath.getPathLength());

                    //swap
                    Point temp = newPath.getPoint(i);
                    newPath.setPoint(i, newPath.getPoint(j));
                    newPath.setPoint(j, temp);
                }
            }
            return newPath;
    }
}
```

Runner.java (Genetic)

```java
import java.util.ArrayList;

public class Runner {

        public static void main(String[] args) {
            //Open file and store point values in arraylist
            String filename = args[0];
            FileImport fileSystem = new FileImport(filename);
            ArrayList<Point> points = fileSystem.getData();

            //Population size
            int populationSize = 1000;
            //Number of generation
            int generations = 200;
            //Mutation rate: ex 3%
            double mutate = .03;
            //Number to compete from population to be parents for crossover
            int competeNum = 10;

            //Create a new population, randomize values
            Population population = new Population(populationSize, points);

        //Send algorithm specific variables to be held in genetic algorithm
        Algorithm algo = new Algorithm(mutate, competeNum, points);
```

```
        //Evolve population for a set number of generations
                for(int i = 0; i < generations; i++) {
                        //System.out.println(i + "\t" + EvaluationManager.print() + "\t" +
population.searchFit().getDistance());
                        System.out.println(population.searchFit().getDistance());
                        population = algo.evolve(population);
                }
                //Prints points in order to plot best solution
        population.searchFit().print();
          }
}
```

Runner.java (Random)

```
import java.util.ArrayList;
import java.util.Random;

public class Runner {

          public static void main(String[] args) {
               //Load in file with points and store points
               String filename = args[0];
               FileImport fileSystem = new FileImport(filename);
               ArrayList<Point> points = fileSystem.getData();

               //Population of 1 for holding the best result of random
               int populationSize = 1;
               int generations = 10000000;
               Random r = new Random();

               Population population = new Population(populationSize, points);

               for(int i = 0; i < generations; i++) {
                       //Create a new random path
                       Path test = new Path(points);
                       double dist1 = test.getDistance();
                       //If new random path is short, keep stored in the single population
                       if(population.searchFit().getDistance() > dist1) {
                              population.setPath(0, test);
```

```
			}

			//Only print values at this interval
			if(i % 50000 == 0) {
				//System.out.println(i + "\t" + EvaluationManager.print() + "\t" + population.searchFit().getDistance());
				System.out.println(population.searchFit().getDistance());
			}
		}
	//population.searchFit().print();
	}
}
```

Runner.java (Hill Climber)

```
import java.util.ArrayList;
import java.util.Random;
//Runner used for Hill Climber
public class Runner {

	public static void main(String[] args) {
		//Load in points from file
		String filename = args[0];
		FileImport fileSystem = new FileImport(filename);
		ArrayList<Point> points = fileSystem.getData();

		//Population is 1, since you assume a solution and optimize it
		//Run for a certain number of generations
		int populationSize = 1;
		int generations = 10000000;
		//Use random to get random points to swap
		Random r = new Random();

		//Create a new population of 1
		Population population = new Population(populationSize, points);

	//For each generation, try swapping two points
		for(int i = 0; i < generations; i++) {
			int a = r.nextInt(50);
```

```
                        int b = r.nextInt(50);


                        double dist1 = population.getPath(0).getDistance();
                        population.getPath(0).swap(a, b);
                        //If the swap creates a population that is not shorter, swap back
                        if(population.searchFit().getDistance() > dist1) {
                                population.getPath(0).swap(a, b);
                        }


                        //Only print results at this interval
                        if(i % 50000 == 0) {
                                //System.out.println(i + "\t" + EvaluationManager.print() + "\t" +
population.searchFit().getDistance());
                                System.out.println(population.searchFit().getDistance());
                        }


                }
        //population.searchFit().print();
          }


}
```

Grapher.m

```
%Plots a path between a set of 2D points
plot(BIG1(:,1),BIG1(:,2))
hold on
scatter(BIG1(:,1),BIG1(:,2))
hold off
```