```
                    // Loop over genome
                    for (int geneIndex = 0; geneIndex < parent1.
                    getChromosomeLength(); geneIndex++) {
                          // Use half of parent1's genes and half of
                          parent2's genes
                          if (geneIndex < swapPoint) {
                                offspring.setGene(geneIndex,
                                parent1.getGene(geneIndex));
                          } else {
                                offspring.setGene(geneIndex,
                                parent2.getGene(geneIndex));
                          }
                    }

                    // Add offspring to new population
                    newPopulation.setIndividual(populationIndex,
                    offspring);
               } else {
                    // Add individual to new population without applying
                    crossover
                    newPopulation.setIndividual(populationIndex, parent1);
               }
          }

          return newPopulation;
}
```

Note that while we've made no mention of elitism in this chapter, it's still represented above and in the mutation algorithm (which was unchanged from the previous chapter).

Single point crossover is popular both because of its favorable genetic attributes (preserving contiguous genes), and because it's easy to implement. In the code above, a new population is created for the new individuals. Next, the population is looped over and individuals are fetched in order of fitness. If elitism is enabled, the elite individuals are skipped over and added straight into the new population, otherwise it's decided whether to crossover the current individual based on the crossover rate. If the individual is chosen for crossover, a second parent is picked using tournament selection.

Next, a crossover point is picked at random. This is the point at which we'll stop using parent1's genes and start using parent2's genes. Then we simply loop over the chromosome, adding parent1's genes to the offspring at first, and then switching to parent2's genes after the crossover point.

Now we can invoke crossover in the RobotController's main method. Adding the line "population = ga.crossoverPopulation(population)" resolves our final TODO, and you should be left with a RobotController class that looks like the following:

```
package chapter3;

public class RobotController {

	public static int maxGenerations = 1000;

	public static void main(String[] args) {

		Maze maze = new Maze(new int[][] {
			{ 0, 0, 0, 0, 1, 0, 1, 3, 2 },
			{ 1, 0, 1, 1, 1, 0, 1, 3, 1 },
			{ 1, 0, 0, 1, 3, 3, 3, 3, 1 },
			{ 3, 3, 3, 1, 3, 1, 1, 0, 1 },
			{ 3, 1, 3, 3, 3, 1, 1, 0, 0 },
			{ 3, 3, 1, 1, 1, 1, 0, 1, 1 },
			{ 1, 3, 0, 1, 3, 3, 3, 3, 3 },
			{ 0, 3, 1, 1, 3, 1, 0, 1, 3 },
			{ 1, 3, 3, 3, 3, 1, 1, 1, 4 }
		});

		// Create genetic algorithm
		GeneticAlgorithm ga = new GeneticAlgorithm(200, 0.05,
		0.9, 2, 10);
		Population population = ga.initPopulation(128);

		// Evaluate population
		ga.evalPopulation(population, maze);

		int generation = 1;

		// Start evolution loop
		while (ga.isTerminationConditionMet(generation,
		maxGenerations) == false) {
			// Print fittest individual from population
			Individual fittest = population.getFittest(0);
			System.out.println("G" + generation + " Best solution
			(" + fittest.getFitness() + "): " + fittest.
			toString());

			// Apply crossover
			population = ga.crossoverPopulation(population);
```

```
                // Apply mutation
                population = ga.mutatePopulation(population);

                // Evaluate population
                ga.evalPopulation(population, maze);

                // Increment the current generation
                generation++;
            }
            System.out.println("Stopped after " + maxGenerations + "
            generations.");
            Individual fittest = population.getFittest(0);
            System.out.println("Best solution
            (" + fittest.getFitness() + "): " + fittest.toString());
        }
}
```

Execution

At this point, your GeneticAlgorithm class should have the following properties and method signatures:

```
package chapter3;

public class GeneticAlgorithm {

        private int populationSize;
        private double mutationRate;
        private double crossoverRate;
        private int elitismCount;
        protected int tournamentSize;

        public GeneticAlgorithm(int populationSize, double mutationRate,
        double crossoverRate, int elitismCount, int tournamentSize) { }
        public Population initPopulation(int chromosomeLength) { }
        public double calcFitness(Individual individual, Maze maze) { }
        public void evalPopulation(Population population, Maze maze) { }
        public boolean isTerminationConditionMet(int generationsCount,
        int maxGenerations) { }
        public Individual selectParent(Population population) { }
        public Population mutatePopulation(Population population) { }
        public Population crossoverPopulation(Population population) { }

}
```

If your method signatures don't match the above, or if you've accidentally missed a method, or if your IDE shows any errors, you should go back and resolve them now.

Otherwise, click Run.

You should see 1,000 generations of evolution, and hopefully your algorithm ended with a fitness score of 29, which is the maximum for this particular maze. (You can count the number of "Route" tiles – represented by "3" – in the maze definition to get this number.

Recall that the purpose of this algorithm wasn't to solve a maze, it was to program a robot's sensor controllers. Presumably, we could now take the winning chromosome at the end of this execution and program it into a physical robot and have a high level of confidence that the sensor controller will make the appropriate maneuvers to navigate not just this maze, but *any* maze without crashing into walls. There's no guarantee that this robot will find the most efficient route through the maze, because that's not what we trained it to do, but it will at the very least not crash.

While 64 sensor combinations may not seem too daunting to program by hand, consider the same problem but in three dimensions: an autonomous flying quadcopter drone may have 20 sensors rather than 6. In this case, you'd have to program for 2^{20} combinations of sensor inputs, approximately one million different instructions.

Summary

Genetic algorithms can be used to design sophisticated controllers, which may be difficult or time consuming for a human to do manually. The robot controller is evaluated by the fitness function, which will often simulate the robot and its environment to save time by not needing to physically test the robot.

By giving the robot a maze and a preferred route, a genetic algorithm can be applied to find a controller that can use the robot's sensors to successfully navigate the maze. This can be done by assigning each sensor an action in the individual's chromosome encoding. By making small random changes with crossover and mutation, guided by the selection process, better controllers are gradually found.

Tournament selection is one of the more popular selection methods used in genetic algorithms. It works by picking a pre-decided number of individuals from the population at random, then comparing the chosen individual's fitness values to find the best. The individual with the highest fitness value "wins" the tournament, and is then returned as the selected individual. Larger tournament sizes lead to high selection pressure, which needs to be carefully considered when picking an optimal tournament size.

When an individual has been selected, it will undergo crossover; one of the crossover methods that could be used is single point crossover. In this crossover method, a single point in the chromosome is picked at random, then any genetic information before that point comes from parent A, and any genetic information after that point comes from parent B. This leads to a reasonably random mix of parent's genetic information, however often the improved two-point crossover method is used instead. In two-point crossover, a start and end point are picked which are used instead to select the genetic information that comes from parent A and the remaining genetic information then comes from parent B.

Exercises

1. Add a second termination condition that terminates the algorithm when the route has been fully explored.
2. Run the algorithm with different tournament sizes. Study how the performance is affected.
3. Add a selection probability to the tournament selection method. Test with different probability settings. Examine how it affects the genetic algorithm's performance.
4. Implement two-point crossover. Does it improve the results?

Traveling Salesman

Chapter 4

Introduction

In this chapter, we are going to explore the traveling salesman problem and how it can be solved using a genetic algorithm. In doing so, we will be looking at the properties of the traveling salesman problem and how we can use those properties to design the genetic algorithm.

The traveling salesman problem (TSP) is a classic optimization problem, studied as far back as the 1800s. The traveling salesman problem involves finding the most efficient route through a collection of cities, visiting each city exactly once.

The traveling salesman problem is often described in terms of optimizing a route through a collection of cities; however, the traveling salesman problem can be applied to other applications. For example, the notion of cities can be thought of as customers for certain applications, or even as soldering points on a microchip. The idea of distance can also be revised to consider other constraints such as time.

In its simplest form, cities can be represented as nodes on a graph, with the distance between each city represented by the length of the edges (see Figure 4-1). A "route" or a "tour", simply defines which edges should be used, and in what order. The route's score can then be calculated by summing the edges used in the route.

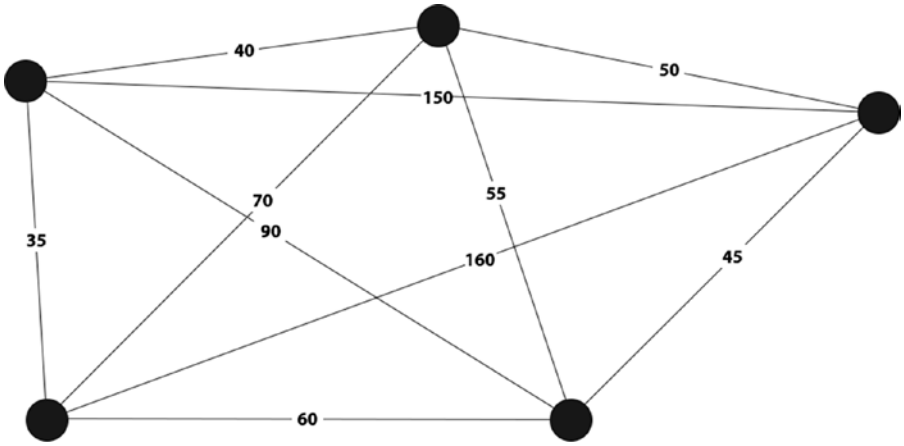

Figure 4-1. Our graph showing the cities and the respective distances between them

During the 1900s, the traveling salesman problem was studied by many mathematicians and scientists; however to this day the problem remains unsolved. The only guaranteed method to produce an optimal solution for the traveling salesman problem is by using a brute force algorithm. A brute force algorithm is an algorithm designed to systematically try every possible solution. Then you find the optimal solution from the complete set of candidate solutions. Attempting solving the traveling salesman problem with a brute force algorithm is an extremely hard task because the number of potential solutions experiences factorial growth as the number of cities is increased. Factorial functions grow even faster than exponential functions, which is why it's so hard to brute force the traveling salesman problem. For example, for 5 cities, there are 120 possible solutions (1x2x3x4x5), and for 10 cities that number will have increased to 3,628,800 solutions! By 15 cities, there are over a trillion solutions. At 60 cities, there are more possible solutions than there are atoms in the observable universe.

Brute force algorithms can be used to find optimal solutions when there are only a few cities but they become more and more challenging as the number of cities increase. Even when techniques are applied to remove reverse and identical routes, it still becomes quickly infeasible to find an optimal solution in a reasonable amount of time.

In reality, we know that finding an optimal solution usually isn't necessary because a good enough solution will typically be all that's required. There are a number of different algorithms that can quickly find solutions that are likely to be within only a couple of percent of optimal. One of the more commonly used algorithms is the nearest neighbor algorithm. With this algorithm, a starting city is picked at random. Then, the next nearest unvisited city is found and picked as the second city in the route. This process of picking the next nearest the unvisited city is continued until all cities have been visited and a complete route has been

found. The nearest neighbor algorithm has been shown to be surprisingly effective at producing reasonable solutions that score within a fraction the optimal solution. Better still, this can be done in a very short time frame. These characteristics make it an attractive solution in many situations, and a possible alternative to genetic algorithms.

The Problem

The problem we will be tackling in this implementation is a typical traveling salesman problem in which we need to optimize a route through a collection of cities. We can generate a number of random cities in 2D space by setting each city to a random x, y position.

When finding the distance between two cities we will simply use the shortest length between the cities as the distance. We can calculate this distance with the following equation:

$$Distance = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$$

Often, the problem will be more complex than this. In this example we are assuming a direct ideal path exists between each city; this is also known as the "Euclidean distance". This usually isn't going to be a typical case as there could be various obstacles making the actual shortest path much longer than the Euclidean distance. We are also assuming that traveling from City-A to City-B takes just as long as traveling from City-B to City-A. Again, in reality this is rarely the case. Often there will be obstacles such as one-way roads that will affect the distance between cities while traveling in a certain direction. An implementation of the traveling salesman problem where the distance between the cities changes depending on the direction is called an *asymmetric traveling salesman problem*.

Implementation

It is time to go ahead and tackle the problem using our knowledge of genetic algorithms. After setting up a new Java/Eclipse package for this problem, we'll begin by encode the route.

Before You Start

This chapter will build on the code you developed in Chapter 3. Before you start, create a new Eclipse or NetBeans project, or create a new package in your existing project for this book called "chapter4".

Copy the Individual, Population, and GeneticAlgorithm classes over from Chapter 3 and import them into chapter4. Make sure to update the package name at the top of each class file! They should all say "package chapter4" at the very top.

Open the GeneticAlgorithm class and delete the following methods: calcFitness, evalPopulation, crossoverPopulation, and mutatePopulation. You'll rewrite those methods in the course of this chapter.

Next, open the Individual class and delete the constructor with the signature "public Individual(int chromosomeLength)". There are two constructors in the Individual class, so be careful to delete the correct one! The constructor to delete is the one that randomly initializes the chromosome; you'll rewrite that as well in this chapter.

The Population class from Chapter 3 requires no modifications other than the package name at the top of the file.

Encoding

The encoding we choose in this example will need to be able to encode a list of cities *in order*. We can do this by assigning each city a unique ID then referencing it using a chromosome in the order of the candidate route. This type of encoding where sequences of genes are used is known as *permutation encoding* and is very well suited to the traveling salesman problem.

The first thing we need to do is assign our cities unique IDs. If we have 5 cities to visit, we can simply assign them the IDs: 1,2,3,4,5. Then when our genetic algorithm has found a route, our chromosome may order the city IDs to look like the following: 3,4,1,2,5. This simply means we would start at city 3, then travel to city 4, then city 1, then city 2, then city 5, before returning back to city 3 to complete the route.

Initialization

Before we start optimizing a route, we need to create some cities. As mentioned earlier, we can generate random cities by picking random x,y coordinates and using them to define a city location.

First, we need to create a City class, which can create and store a city as well as calculate the shortest distance to another city.

```
package chapter4;
public class City {
	private int x;
	private int y;
```

```
            // Initialize population
            Population population = ga.initPopulation(cities.length);

            // TODO: Evaluate population

            // Keep track of current generation
            int generation = 1;

            // Start evolution loop
            while (ga.isTerminationConditionMet(generation,
            maxGenerations) == false) {
                    // TODO: Print fittest individual from population

                    // TODO: Apply crossover

                    // TODO: Apply mutation

                    // TODO: Evaluate population

                    // Increment the current generation
                    generation++;
            }

            // TODO: Display results
        }
}
```

Hopefully, this procedure is becoming familiar; we're once again beginning to implement the pseudocode presented at the beginning of Chapter 2. We've also generated an array of City objects that we'll use in our evaluation methods, much like how we generated a Maze object to evaluate individuals against in the last chapter.

The rest is rote: initialize a GeneticAlgorithm object (including population size, mutation rate, crossover rate, elitism count, and tournament size), then initialize a population. The individuals' chromosome length must be the same as the number of cities we wish to visit.

We get to reuse the simple "max generations" termination condition from the previous chapter, so we're left with only six TODOs and a working loop this time. Let's begin, as usual, with the evaluation and fitness scoring methods.

Evaluation

Now we need to evaluate the population and assign fitness values to the individuals so we know which perform the best. The first step is to define the fitness function for the problem. Here, we simply need to calculate the total distance of the route given by the individual's chromosome.

```
            // Increment the current generation
            generation++;
        }

        // Display results
        System.out.println("Stopped after " + maxGenerations + "
        generations.");
        Route route = new Route(population.getFittest(0), cities);
        System.out.println("Best distance: " + route.getDistance());
    }
}
```

At this point, we can click "Run" and the loop will go through the motions, printing the same thing 3,000 times but showing no change. This, of course, is expected; we need to implement crossover and mutation as our two remaining TODOs.

Termination Check

As we have already learned, there is no way to know if we have found an optimal solution to the traveling salesman problem unless we try every possible solution. This means the termination check we use in this implementation cannot terminate when the optimal solution has been found, because it simply has no way of knowing.

Seeing as we have no way to terminate when the optimal solution has been found, we can simply allow the algorithm to run for a set number of generations before finally terminating, and therefore we are able to reuse the isTerminationConditionMet method in the GeneticAlgorithm class from Chapter 3.

Note, however, that in situations like these—where the best solution can not be known—that there are many sophisticated techniques for termination beyond simply setting an upper bound on the number of generations.

One common technique is to measure the improvement of the population's fitness over time. If the population is still improving rapidly, you may want to allow the algorithm to continue on. Once the population stops improving, you can end the evolution and present the best solution.

You may never find the globally optimum solution in a complex solution space like the traveling salesman problem's, but there are many strong local optima, and a plateau in progress typically indicates that you've found one of these local optima.

There are several ways you can measure the progress of a genetic algorithm over time. The simplest method is to measure the number of consecutive generations where there has been no improvement in the best individual. If the number of generations with no improvement has crossed some threshold, for instance 500 generations with no improvement, you can stop the algorithm.

```
                // Add child
                newPopulation.setIndividual(populationIndex, offspring);
            } else {
                // Add individual to new population without applying
                crossover
                newPopulation.setIndividual(populationIndex, parent1);
            }
        }

        return newPopulation;
    }
```

In this method, we first create a new population to hold the offspring. Then, the current population is looped over in the order of the fittest individual first. If elitism is enabled, the first few elite individuals are skipped over and added to the new population unaltered. The remaining individuals are then considered for crossover using the crossover rate. If crossover is to be applied to the individual, a parent is selected using the selectParent method (in this case, selectParent implements tournament selection as in Chapter 3) and a new blank individual is created.

Next, two random positions in the parent1's chromosome are picked and the subset of genetic information between those positions are added to the offspring's chromosome. Finally, the remaining genetic information needed is added in the order found in parent2; then when complete, the individual is added into the new population.

We can now implement our crossoverPopulation method into our "main" method in the TSP class and resolve one of our TODOs. Find "TODO: Apply crossover" and replace it with:

```
// Apply crossover
population = ga.crossoverPopulation(population);
```

Clicking "Run" at this point should result in a working algorithm! After 3,000 generations, you should expect to see a best distance of approximately 1,500. However, as you may recall, crossover alone is prone to getting stuck in local optima, and you may find that the algorithm plateaus. Mutation is our way of randomly dropping candidates in new locations on our solution space, and can help improve long-term results at the cost of short-term gain.

Mutation

Like with crossover, the type of mutation we use for the traveling salesman problem is important because again, we need to ensure the chromosome is valid after it has been applied. A method in which we randomly change a single value of a gene would likely cause repeats in the chromosome and as a result, the chromosome would be invalid.

Mutation can be implemented in a similar manner. Instead of choosing a random number for a random gene in the chromosome, we can create a new *random but valid individual* and essentially run uniform crossover to achieve mutation! That is, we can use our Individual(Timetable) constructor to create a brand new random Individual, and then select genes from the random Individual to copy into the Individual to be mutated. This technique is called *uniform mutation*, and makes sure that all of our mutated individuals are fully valid, never selecting a gene that doesn't make sense. Add the following method anywhere in the GeneticAlgorithm class:

```
public Population mutatePopulation(Population population, Timetable
timetable) {
        // Initialize new population
        Population newPopulation = new Population(this.populationSize);

        // Loop over current population by fitness
        for (int populationIndex = 0; populationIndex < population.size();
        populationIndex++) {
                Individual individual = population.
                getFittest(populationIndex);

                // Create random individual to swap genes with
                Individual randomIndividual = new Individual(timetable);

                // Loop over individual's genes
                for (int geneIndex = 0; geneIndex < individual.
                getChromosomeLength(); geneIndex++) {
                        // Skip mutation if this is an elite individual
                        if (populationIndex > this.elitismCount) {
                                // Does this gene need mutation?
                                if (this.mutationRate > Math.random()) {
                                        // Swap for new gene
                                        individual.setGene(geneIndex,
                                        randomIndividual.getGene(geneIndex));
                                }
                        }
                }

                // Add individual to population
                newPopulation.setIndividual(populationIndex, individual);
        }

        // Return mutated population
        return newPopulation;
}
```

G, H

I, J, K, L, M

N, O

P, Q

R

S

T

U

V, W, X, Y, Z