28 Genetic and Evolutionary Computing

Chapter Objectives

A brief introduction to the genetic algorithms
- Genetic operators include
  - Mutation
  - Crossover

An example GA application worked through
- The WordGuess problem

Appropriate object hierarchy created
Generalizable to other GA applications
Exercises emphasize GA interface design

Chapter Contents

28.1 Introduction

The genetic algorithm (GA) is one of a number of computer programming techniques loosely based on the idea of natural selection. The idea of applying principles of natural selection to computing is not new. By 1948, Alan Turing proposed "genetical or evolutionary search" (Turing 1948). Less than two decades later, H.J. Bremmermann performed computer simulations of "optimization through evolution and recombination" (Eiben and Smith 1998). It was John Holland who coined the term, *genetic algorithm* (Holland 1975). However, the GA was not widely studied until 1989, when D.E. Goldberg showed that it could be used to solve a significant number of difficult problems (Goldberg 1989). Currently, many of these threads have come together under the heading *evolutionary computing* (Luger 2009, Chapter 12).

28.2 The Genetic Algorithm: A First Pass

The Genetic Algorithm is based loosely on the concept of natural selection. Individual members of a species who are better adapted to a given environment reproduce more successfully. They pass their adaptations on to their offspring. Over time, individuals possessing the adaptation form a new species that is particularly suited to the environment. The genetic algorithm applies the *metaphor* of natural selection to optimization problems. No claim is made about its biological accuracy, although individual researchers have proposed mechanisms both with and without a motivating basis from nature.

A candidate solution for a genetic algorithm is often called a *chromosome*. The chromosome is composed of multiple *genes*. A collection of

chromosomes is called a *population*. The GA randomly generates an initial population of chromosomes, which are then ranked according to a *fitness function* (Luger 2009, Section 12.1).

Consider an example drawn from structural engineering. Structural engineers make use of a component known as a *truss*. Trusses come in many varieties, the simplest of which should be familiar to anyone who has noticed the interconnected triangular structures found in bridges and cranes. Figure 28.1 is an example of the canonical 64-bar truss (Ganzerli et al. 2003), which appears in the civil engineering literature on optimization. The arrows are loads, expressed in a unit known as a Kip. Engineers would like to minimize the volume of a truss, taken as the cross-sectional area of the bars multiplied by their length.

To solve this problem using a GA, we first randomly generate a population of trusses. Some of these will stand up under a given load, some will not. Those that fail to meet the load test are assigned a severe penalty. The ranking in this problem is based on volume. The smaller the truss volume, after any penalty has been assigned, the more fit the truss. Only the fittest individuals are selected for reproduction. It has been shown that the truss design problem is NP-Complete (Overbay et al. 2006). Engineers have long-recognized the difficulty of truss design, most often developing good enough solutions with the calculus-based optimization techniques available to them (Ganzerli et al. 2003).

By the late nineties, at least two groups were applying genetic algorithms to very large trusses and getting promising results (Rajeev and Krishnamoorthy 1997), (Ghasemi et al. 1999). Ganzerli et al. (2003) took this work a step further by using genetic algorithms to optimize the 64-bar truss with the added complexity of load uncertainty. The point here is not simply that the GA is useful in structural engineering, but that it has been applied in hundreds of ways in recent years, structural engineering being an especially clear example. A number of other examples, including the traveling salesperson and SAT problems are presented in Luger (2009, Section 12.1). The largest venue for genetic algorithm research is *The Genetic and Evolutionary Computation Conference* (GECCO 2007). Held in a different city each summer, the papers presented range from artificial life through robotics to financial and water quality systems.

Despite the breadth of topics addressed, the basic outline for genetic algorithm solvers across application domains is very similar. Search through the problem space is guided by the *fitness-function*. Once the fitness-function is designed, the *GA* traverses the space over many iterations, called *generations*, stopping only when some pre-defined convergence criterion is met. Further, the only substantial differences between one application of the GA and the next is the representation of the chromosome for the problem domain and the fitness function that is applied to it. This lends itself very nicely to an object-oriented implementation that can be easily generalized to multiple problems. The technique is to build a generic *GA* class with specific implementations as subclasses.

WordGuess Example

Consider a simple problem called *WordGuess* (Haupt and Haupt 1998). The user enters a target word at the keyboard. The GA guesses the word. In this case, each letter is a gene, each word a chromosome, and the total collection of words is the population. To begin, we randomly generate a sequence of chromosomes of the desired length. Next, we rank the generated chromosomes for fitness. A chromosome that is identical with the target has a fitness of zero. A chromosome that differs in one letter has a fitness of 1 and so on. It is easy to see that the size of the search space for *WordGuess* increases exponentially with the length of the word. In the next few sections, we will develop an object-oriented solution to this problem.

Suppose we begin with a randomly generated population of 128 character strings. After ranking them, we immediately eliminate the half that is least fit. Of the 64 remaining chromosomes, the fittest 32 form 16 breeding pairs. If each pair produces 2 offspring, the next generation will consist of the 32 parents plus the 32 children.

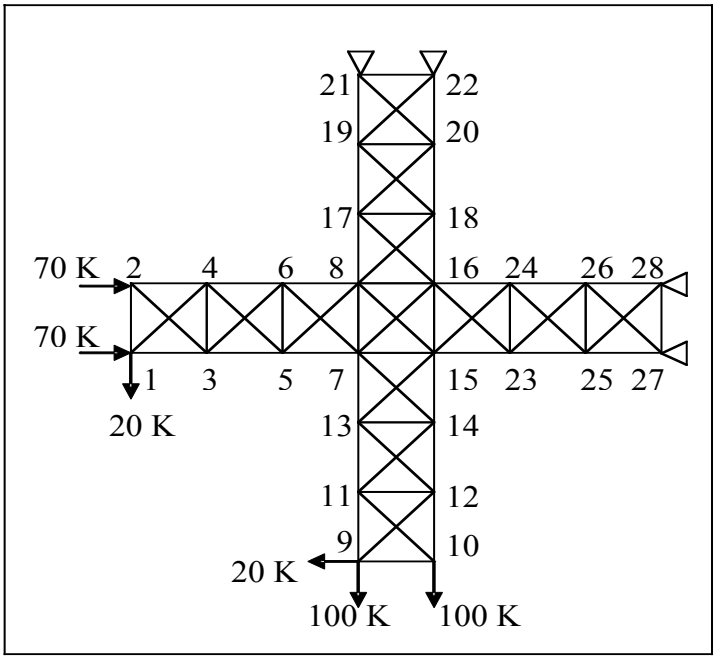

Figure 28.1 A system of trusses to be optimized with a set of genetic operators.

Having decided who may reproduce, we mate them. The GA literature is filled with clever mating strategies, having more or less biological plausibility. We consider two, *TopDown* and *Tournament*. In TopDown, the fittest member of the population mates with the next most fit and so on, until the breeding population is exhausted. Tournament is a bit more complex, and slightly more plausible (Haupt and Haupt 1998). Here we choose a subset of chromosomes from the breeding population. The fittest chromosome within this subset becomes Parent A. We do the same thing again, to find its mate, Parent B. Now we have a breeding pair. We continue with this process until we have created as many breeding pairs as we need.

Mating is how each chromosome passes its genes to future generations. Since mating is an attempt to simulate (and simplify) recombinant DNA, many authors refer to it as *recombination* (Eiben and Smith 2003). As with pairing, many techniques are available. *WordGuess* uses a single technique called *Crossover*. Recall that each chromosome consists of *length*(*chromosome*) genes. The most natural data structure to represent a chromosome is an array of *length*(*chromosome*) positions. A gene—in this case an alphabetic character—is stored in each of these positions. Crossover works like this:

1. Generate a random number n, 0 <= n < length(chromosome). This is called the Crossover Point.
2. Parent A passes its genes in positions 0 … n to Child 1.
3. Parent B passes its genes in positions 0 … n to Child 2.
4. Parent A passes it genes in positions n + 1 … length(chromosome – 1) to the corresponding positions in Child 2.
5. Parent B passes its genes in positions n + 1 … length(chromosome – 1) to the corresponding positions in Child 1

Figure 28.2 illustrates mating with $n = 4$. The parents, PA and PB produce the two children CA and CB.

After the reproducing population has been selected, paired, and mated, the final ingredient is the application of random mutations. The importance of random mutation in nature is easy to see. Favorable (as well as unfavorable!) traits have to arise before they can be passed on to offspring. This happens through random variation, caused by any number of natural mutating agents. Chemical mutagens and radiation are examples. Mutation guarantees that new genes are introduced into the gene pool. Its practical effect for the GA is to reduce the probability that the algorithm will converge on a local minimum. The percentage of genes subject to mutation is a design parameter in the solution process.

The decision of when to stop producing new generations is the final component of the algorithm. The simplest possibility, the one used in *WordGuess*, is to stop either after the GA has guessed the word or 1000 generations have passed. Another halting condition might be to stop when some parameter P percent of the population is within Q standard deviations of the population mean.

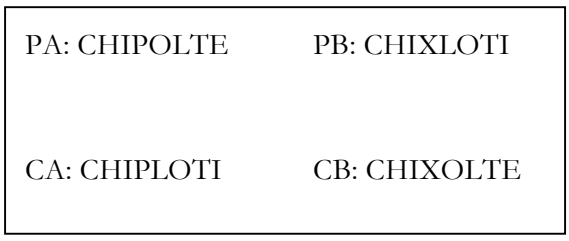

Figure 28.2 Recombination with crossover at the point n = 4.

The entire process can be compactly expressed through the while-loop:

```
GA(population)
{
  Initialize(population);
  ComputeCost(population);
  Sort(population);
  while (not converged on acceptable solution)
  {
     Pair(population);
     Mate(population);
     Mutate(population);
     Sort(population);
     TestConvergence(population);
  }
}
```

28.3 A GA Implementation in Java

`WordGuess` is written in the Java programming language with object-oriented (OO) techniques developed to help manage the search complexity. An OO software system consists of a set of interrelated structures known as classes. Each class can perform a well-defined set of operations on a set of well-defined operands. The operations are referred to as *methods*, the operands as *member variables*, or just *variables*.

The Class Structure

The classes interrelate in two distinct ways. First, classes may inherit properties from one another. Thus, we have designed a class called `GA`. It defines most of the major operations needed for a genetic algorithm. Knowing that we want to adapt `GA` to the problem of guessing a word typed at the keyboard, we define the class `WordGuess`. Once having written code to solve a general problem, that code is available to more specific instances of the problem. A hypothetical inheritance structure for the genetic algorithm is shown in Figure 28.3, where the upward pointing arrows are inheritance links. Thus, `WordGuess` inherits all classes and variables defined for the generic `GA`.

Second, once defined, classes may make use of one another. This relationship is called *compositionality*. `GA` contains several component classes:

- `Chromosome` is a representation of an individual population member.
- `Pair` contains all pairing algorithms developed for the system. By making `Pair` its own class, the user can add new methods to the system without changing the core components of the code.
- `Mate` contains all mating algorithms developed for the system.
- `SetParams`, `GetParams`, and `Parameters` are mechanisms to store and retrieve parameters.

- `WordGuessTst` sets the algorithm in motion.

Finally, class `GA` makes generous use of Java's pre-defined classes to represent the population, randomly generate chromosomes, and to handle files that store both the parameters and an initial population. `GA` is character-based. A Graphical User Interface (GUI) can be implemented with Java's facilities for GUIs and Event-Driven programming found in the `javax.swing` package (see Exercise 28.3).

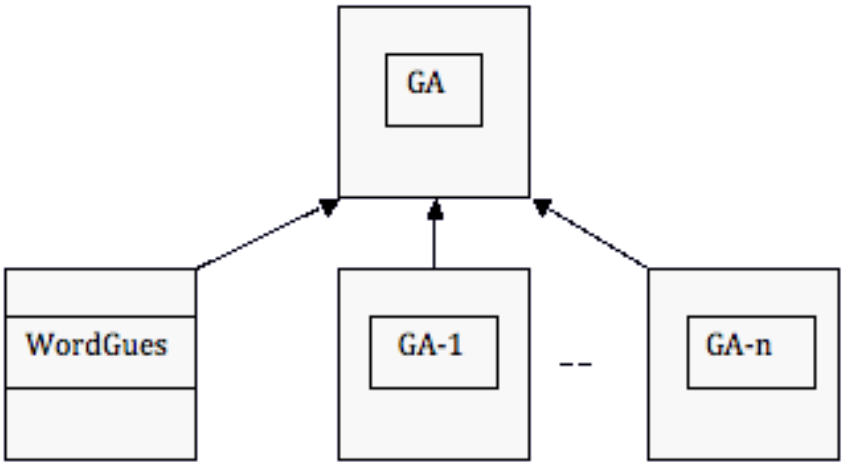

Figure 28.3 The inheritance hierarchy for implementing the GA.

The Class Chromosome

The variables reflect what a class knows about itself. Class `Chromosome` must know how many genes it has, its fitness, and have a representation for its genes. The number of genes and the fitness of the chromosome can be easily represented as integers. The representation of the genes poses a design problem. For `WordGuess`, a character array works nicely. For an engineering application, we might want the chromosome to be a vector of floating point variables. The most general representation is to use Java's class `Object` and have specific implementations, like `WordGuess`, define their own chromosomes (see Exercise 28.4).

The methods describe what a class does. Class `Chromosome` must be able to set and return its fitness, set and return the number of its genes, display its genes, and determine if it is equal to another chromosome. The Java code that implements the class `Chromosome` follows.

```
public class Chromosome
{
  private int CH_numGenes;
  protected int CH_cost;
  private  Object[] CH_gene;
  public Chromosome(int genesIn)
  {
     CH_numGenes = genesIn;
     CH_gene = new char[CH_numGenes];
  }
```

```
    public int GetNumGenes()
    {
       return CH_numGenes;
    }
    public void SetCost(int cost)
    {
       CH_cost = cost;
    }
    public void SetGene(int index, Object value)
    {
       CH_gene[index] = value;
    }
    public boolean Equals(String target)
    {
       for (int i = 0; i < CH_numGenes; i++)
       if (CH_gene[i] != target.charAt(i))
          return false;
       return true;
    }
}
```

Classes Pair and Mate

Chromosomes must be paired and mated. So that we can experiment with more than a single pairing or mating algorithm, we group multiple versions into classes `Pair` and `Mate`. Since pairing and mating are done over an entire population, before we define `Pair` and `Mate` we must decide upon a representation for the population. A population is a list of chromosomes. Java's built-in collection classes are contained in the `java.util` library and known as the *Java Collection Framework*. Two classes, `ArrayList` and `LinkedList` support list behavior. It is intuitively easy to conceive of a population as an array of chromosomes. Accordingly, we use the class `ArrayList` to define a population as follows:

```
ArrayList<Chromosome> GA_pop;
GA_pop = new ArrayList<Chromosome>();
```

The first line defines a variable, `GA_pop` as type `ArrayList`. The second creates an instance of `GA_pop`.

`WordGuess` implements a single paring algorithm, `TopDown`. Tournament pairing is left as an exercise. `Pair` has to know the population that is to be paired and the number of mating pairs. Since only half of the population is fit enough to mate, the number of mating pairs is the population size divided by 4. Here we can see one of the benefits of using pre-defined classes. `ArrayList` provides a method that returns the size of the list. The code for `Pair` follows:

```
public class Pair
{
  private ArrayList<Chromosome> PR_pop;
```

```
  public  Pair(ArrayList<Chromosome> population)
  {
     PR_pop = population;
  }
  public int TopDown()
  {
     return (PR_pop.size() / 4);
  }
}
```

Class `Mate` also implements a single algorithm, `Crossover`. It is slightly more complex than `Pair`. To implement `Crossover`, we need four chromosomes, one for each parent, and one for each child. We also need to know the crossover point, as explained in Section 28.2, the number of genes in a chromosome, and the size of the population. We now present the member variables and the constructor for `Mate`:

```
public class Mate
{
  private Chromosome MT_father,
     MT_mother,
     MT_child1,
     MT_child2;
  private  int   MT_posChild1,
     MT_posChild2,
     MT_posLastChild,
     MT_posFather,
     MT_posMother,
     MT_numGenes,
     MT_numChromes;
  public   Mate(ArrayList<Chromosome> population,
                int numGenes, int numChromes)
  {
     MT_posFather  = 0;
     MT_posMother  = 1;
     MT_numGenes   = numGenes;
     MT_numChromes = numChromes;
     MT_posChild1  = population.size()/2;
     MT_posChild2  = MT_posChild1 + 1;
     MT_posLastChild = population.size() - 1;
     for (int i = MT_posLastChild;
           i >= MT_posChild1; i--)
        population.remove(i);
     MT_posFather = 0;
     MT_posMother = 1;
```

```
    }
                    // Remaining method implemented below.
}
```

`Mate` takes a population of `chromosome` as a parameter and returns a mated population. The `for`-loop eliminates the least fit half of the population to make room for the two children per breeding pair.

`Crossover`, the only other method in `Mate`, is presented next. It implements the algorithm described in Section 28.2. Making use of the `Set/Get` methods of `Chromosome`, `Crossover` blends the chromosomes of each breeding pair. When mating is complete, the breeding pairs are in the top half of the `ArrayList`, the children are in the bottom half.

```
public ArrayList<Chromosome> Crossover(
  ArrayList<Chromosome> population, int numPairs)
{
  for (int j = 0; j < numPairs; j++)
  {
     MT_father =  population.get(MT_posFather);
     MT_mother =  population.get(MT_posMother);
     MT_child1 =  new Chromosome(MT_numGenes);
     MT_child2 = new Chromosome(MT_numGenes);
     Random rnum = new Random();
     int crossPoint = rnum.nextInt(MT_numGenes);
                                          // left side
     for (int i = 0; i < crossPoint; i++)
     {
        MT_child1.SetGene(i,
           MT_father.GetGene(i));
        MT_child2.SetGene(i,
           MT_mother.GetGene(i));
     }
                                          // right side
     for (int i = crossPoint;
            < MT_numGenes;i++)
     {
        MT_child1.SetGene(i,
           MT_mother.GetGene(i));
        MT_child2.SetGene(i,
           MT_father.GetGene(i));
     }
     population.add(MT_posChild1,MT_child1);
     population.add(MT_posChild2,MT_child2);
     MT_posChild1 = MT_posChild1 + 2;
     MT_posChild2 = MT_posChild2 + 2;
```

```
            MT_posFather = MT_posFather + 2;
            MT_posMother = MT_posMother + 2;
        }
        return population;
    }
```

The GA Class

Having examined its subclasses, it is time to look at class `GA`, itself. We never create an instance of `class GA`. `GA` exists only so that its member variables and methods can be inherited, as in Figure 28.3. Classes that may not be instantiated are called *abstract*. The classes higher in the hierarchy are called *superclasses*. Those lower in the hierarchy are called *subclasses*. Member variables and methods designated `protected` in a super class are available to its subclasses.

`GA` contains the population of chromosomes, along with the various parameters that its subclasses need. The parameters are the size of the initial population, the size of the pared down population, the number of genes, the fraction of the total genes to be mutated, and the number of iterations before the program stops. The parameters are stored in a file manipulated through the classes `Parameters`, `SetParams`, and `GetParams`. We use object semantics to manipulate the files. Since file manipulation is not essential to a GA, we will not discuss it further. The class declaration `GA`, its member variables, and its constructor follow.

```
public abstract class GA extends Object
{
  protected int  GA_numChromesInit;
  protected int  GA_numChromes;
  protected int  GA_numGenes;
  protected double  GA_mutFact;
  protected int  GA_numIterations;
  protected ArrayList<Chromosome> GA_pop;
  public GA(String ParamFile)
  {
     GetParams GP  = new GetParams(ParamFile);
     Parameters P   = GP.GetParameters();
     GA_numChromesInit = P.GetNumChromesI();
     GA_numChromes  = P.GetNumChromes();
     GA_numGenes    = P.GetNumGenes();
     GA_mutFact  = P.GetMutFact();
     GA_numIterations  = P.GetNumIterations();
     GA_pop   = new ArrayList<Chromosome>();
     }
            //Remaining methods implemented below.
}
```

The first two lines of the constructor create the objects necessary to read the parameter files. The succeeding lines, except the last, read the file and

store the results in class **GA**'s members variables. The final line creates the data structure that is to house the population. Since an `ArrayList` is an expandable collector, there is no need to fix the size of the array in advance.

Class **GA** can do all of those things common to all of its subclasses. Unless you are a very careful designer, odds are that you will not know what is common to all of the subclasses until you start building prototypes. Object-oriented techniques accommodate an iterative design process quite nicely. As you discover more methods that can be shared across subclasses, simply push them up a level to the superclass and recompile the system.

Superclass **GA** performs general housekeeping tasks along with work common to all its subclasses. Under housekeeping tasks, we want a super class **GA** to display the entire population, its parameters, a chromosome, and the best chromosome within the population. We also might want it to tidy up the population by removing those chromosomes that will play no part in evolution. This requires a little explanation. Two of the parameters are `GA_numChromesInit` and `GA_numChromes`. Performance of a GA is sometimes improved if we initially generate more chromosomes than are used in the GA itself (Haupt and Haupt 1998). The first task, then, is to winnow down the number of chromosomes from the number initially generated (`GA_numChromesInit`) to the number that will be used (`GA_numChromes`).

Under shared tasks, we want the superclass **GA** to create, rank, and mutate the population. The housekeeping tasks are very straightforward. The shared method that initializes the population follows:

```
protected void InitPop()
{
  Random rnum = new Random();
  char letter;
  for (int index = 0;
        index < GA_numChromesInit; index++)
  {
     Chromosome Chrom =
        new Chromosome(GA_numGenes);
     for (int j = 0; j < GA_numGenes; j++)
     {
        letter = (char)(rnum.nextInt(26) + 97);
         Chrom.SetGene(j,letter);
     }
     Chrom.SetCost(0);
     GA_pop.add(Chrom);
  }
}
```

Initializing the population is clear enough, though it does represent a design decision. We use a nested `for` loop to create and initialize all genes

within a chromosome and then to add the chromosomes to the population. Notice the use of Java's pseudo-random number generator. In keeping with the object-oriented design, `Random` is a class with associated methods. `rnum.nextInt(26)` generates a pseudo-random number in the range [0..25]. The design decision is to represent genes as characters. This is not as general as possible, an issue mentioned earlier and addressed in the exercises. We add 97 to the generated integer, because the ASCII position of 'a' is 97. Consequently, we transform the generated integer to characters in the range ['a'..'z'].

Ranking the population, shown next, is very simple using the `sort` method that is part of the static class, `Collections`. A static class is one that exists to provide services to other classes. In this case, the methods in `Collections` operate on and return classes that implement the *Collection Interface*. An interface in Java is a set of specifications that implementing classes must fulfill. It would have been possible to design `GA` as an *Interface* class, though the presence of common methods among specific genetic algorithms made the choice of `GA` as a superclass a more intuitively clear design. Among the many classes that implement the methods specified in the *Collection* interface is `ArrayList`, the class we have chosen to represent the population of chromosomes.

```
protected void SortPop()
{
 Collections.sort(GA_pop, new CostComparator());
}
private class CostComparator
   implements Comparator <Chromosome>
{
   int result;
   public int compare(Chromosome obj1,
      Chromosome obj2)
   {
      result = new Integer(obj1.GetCost()).
         compareTo(new Integer(obj2.GetCost()));
         return result;
   }
}
```

`Collections.sort` requires two arguments, the object to be sorted—the `ArrayList` containing the population—and the mechanism that will do the sorting:

```
Collections.sort(GA_pop, new CostComparator());
```

The second argument creates an instance of a helper class that implements yet another interface class, this time the *Comparator* interface. The second object is sometimes called the *comparator object*. To implement the Comparator interface we must specify the type of the objects to be compared—class `Chromosome`, in this case—and implement its `compare` method. This method takes two chromosomes as arguments,

uses the method `GetCost` to extract the cost from the chromosome, and the `compareTo` method of the *Integer* wrapper class to determine which of the chromosomes costs more. In keeping with OO, we give no consideration to the specific algorithm that Java uses. Java documentation guarantees only that the Comparator class "imposes a total ordering on some collection of objects" (Interface Comparator 2007).

`Mutation` is the last of the three shared methods that we will consider. The fraction of the total number of genes that are to be mutated per generation is a design parameter. The fraction of genes mutated depends on the size of the population, the number of genes per chromosome, and the fraction of the total genes to mutate. For each of the mutations, we randomly choose a gene within a chromosome, and randomly choose a mutated value. There are two things to notice. First, we never mutate our best chromosome. Second, the mutation code in `GA` is specific to genetic algorithms where genes may be reasonably represented as characters. The code for `Mutation` may be found on the Chapter 28 code library.

28.4 Conclusion: Complex Problem Solving and Adaptation

In this chapter we have shown how Darwin's observations on speciation can be adapted to complex problem solving. The GA, like other AI techniques, is particularly suited to those problems where an optimal solution may be computationally intractable. Though the GA might stumble upon the optimal solution, odds are that computing is like nature in one respect. Solutions and individuals must be content with having solved the problem of adaptation only well enough to pass their characteristics into the next generation. The extended example, `WordGuess`, was a case in which the GA happens upon an exact solution. (See the code library for sample runs). This was chosen for ease of exposition. The exercises ask you to develop a GA solution to a known NP-Complete problem.

We have implemented the genetic algorithm using object-oriented programming techniques, because they lend themselves to capturing the generality of the GA. Java was chosen as the programming language, both because it is widely used and because its syntax in the C/C++ tradition makes it readable to those with little Java or OO experience.

As noted, we have not discussed the classes `SetParams`, `GetParams`, and `Parameters` mentioned in Section 28.3. These classes write to and read from a file of design parameters. The source code for them can be found in the auxiliary materials. Also included are instructions for using the parameter files, and instructions for exercising `WordGuess`.

Chapter 28 was jointly written with Paul De Palma, Professor of Computer Science at Gonzaga University, Spokane Washington.

Exercises

1. The traveling salesperson problem is especially good to exercise the GA, because it is possible to compute bounds for it. If the GA produces a solution that falls within these bounds, the solution, while probably not optimal, is reasonable. See Hoffman and Wolfe (1985) and Overbay, et al.

(2007) for details. The problem is easily stated. Given a collection of cities, with known distances between any two, a tour is a sequence of cities that defines a start city, C, visits every city once and returns to C. The optimal tour is the tour that covers the shortest distances. Develop a genetic algorithm solution for the traveling sales person problem. Create, at least, two new classes `TSP`, derived from `GA`, and `TSPtst` that sets the algorithm in motion. See comments on mating algorithms for the traveling salesperson problem in Luger (2009, Section 12.1.3).

2. Implement the Tournament pairing method of the class `Pair`. Tournament chooses a subset of chromosomes from the population. The most fit chromosome within this subset becomes Parent A. Do the same thing again, to find its mate, Parent B. Now you have a breeding pair. Continue this process until we have as many breeding pairs as we need. Tournament is described in detail in Haupt and Haupt (1998). Does `WordGuess` behave differently when Tournament is used?

3. As it stands, `GA` runs under command-line Unix/Linux. Use the `javax.swing` package to build a GUI that allows a user to set the parameters, run the program, and examine the results.

4. Transform the java application code into a java applet. This applet should allow a web-based user to choose the `GA` to run (either `WordGuess` or `TSP`), the pairing algorithm to run (Top-Down or Tournament), and to change the design parameters

5. `WordGuess` does not make use of the full generality provided by object-oriented programming techniques. A more general design would not represent genes as characters. One possibility is to provide several representational classes, all inheriting from a modified `GA` and all being super classes of specific genetic algorithm solutions. Thus we might have `CHAR_GA` inheriting from `GA` and `WordGuess` inheriting from `CHAR-GA`. Another possibility is to define chromosomes as collections of genes that are represented by variables of class Object. Using these, or other, approaches, modify `GA` so that it is more general.

6. Develop a two-point crossover method to be included in class `Mate`. For each breeding pair, randomly generate two crossover points. Parent A contributes its genes before the first crossover and after the second to Child A. It contributes its genes between the crossover points to Child B. Parent B does just the opposite. See Haupt and Haupt (1998) for still other possibilities.