Gettting Started with Vitri:
A Simple Genetic Algorithm

About this Document

This document shows how to use and customize a simple genetic algorithm that is part of Vitri, an object-oriented framework implemented in pure Java. The original Vitri kernel was developed by John Baugh at North Carolina State University, but numerous graduate students and colleagues contributed to its subsequent growth and application as a large-scale decision support system. These include Sujay Kumar, Raghu Kurlagunda, Troy Doby, Ranji Ranjithan, and Downey Brill.

Releases beginning with Vitri 2.0 have been re-engineered from the original kernel. Previous users will notice some changes in new releases, which now include distribution of source code, but are absent some of the previous tools for decision support (like MGA) and visualization of runtime performance; these may appear again in future releases. Unfortunately, Vitri had grown so large that it became unwieldy and difficult to maintain. Releases 2.0 and beyond are a return to a smaller yet powerful kernel for using and exploring heuristic search on workstation clusters.

For additional information visit `http://www4.ncsu.edu/~jwb/vitri`, or contact:

John Baugh
Department of Civil Engineering
North Carolina State University
Raleigh, NC 27695
`john.baugh@ncsu.edu`

Further reading

1. *Vitri: a generic framework for engineering decision support systems on heterogeneous computer networks,* Sujay V. Kumar, Ph.D. Thesis, North Carolina State University, 2002 (available online at `http://www.lib.ncsu.edu`).

2. Vitri: A framework for environmental decision support on heterogeneous computer networks, Sujay V. Kumar, Troy Doby, John Baugh, Downey Brill, Ranji Ranjithan. In *Proceedings of the 2001 World Water and Environmental Resources Congress,* ASCE, 2001.

3. *An object-oriented framework for distributed genetic algorithms,* Raghu N. Kurlagunda. M.S. Thesis, North Carolina State University, 1999.

2/18/03

Object-Oriented Frameworks

Vitri is an object-oriented framework in the sense that it provides an overall organization and structure for a family of software abstractions. While frameworks can be specialized to produce custom applications, they differ from a typical software library in the so-called "inversion of control" that takes place. Instead of passively responding as libraries do, a framework coordinates and directs the flow of control within the application. For instance, an optimization framework like Vitri allows users to "plug in" their simulation models, acting as a front end that iteratively invokes the supplied model as needed to obtain a solution.

The methods of "attaching" components and models depend on the architectural design principles and implementation details adopted in the framework. These variable aspects that are identified in the application domain are often called *hot spots,* or *plug points.* An object-oriented approach, as is taken in Vitri, leverages on the orthogonal notions of data abstraction, relationships among types in a hierarchy or lattice, and various forms of ad hoc polymorphism. Realization of these concepts is afforded by the usual elements of object-oriented programming: (abstract) classes, methods, fields, inheritance, and interfaces.

Genetic Algorithms

Several modern optimization techniques, including genetic algorithms, neural networks, and simulated annealing, rely on analogies to natural processes. Genetic algorithms (GAs) are based on the principle of natural selection, or "survival of the fittest." To put this analogy to use, computer programs maintain a population of organisms (potential solutions) that are, ahem, "combined" to produce new organisms in the next generation. Because a selection scheme screens organisms, only the fittest are likely to have offspring in the subsequent generation. As generations pass, the average fitness of the organisms improves; eventually, the program converges and the most fit organism is returned as the solution.

A sample problem

Consider the *knapsack* problem, in which a burglar tries to maximize the *value* of his takings: Every object that he might place into the bag has a size (or cost) and a value (or benefit). The decision problem then is to determine which objects to place in the bag. For example, imagine a bag of size 10 and the following five objects from which to select:

Object:	1	2	3	4	5
Value:	1	13	12	19	1
Size:	7	3	5	7	9

One solution to the problem, though not a very good one, is to take items 2 and 3, which have a combined size of 8 and a value of 25. A GA must be capable of representing this and any valid solution to the problem. Such candidate solutions (i.e., organisms in a GA) can often represented as binary strings, or arrays (a so-called "binary encoding"). For example, the previous solution can be described by the binary array $\langle 01100 \rangle$, where a 1 means "take the object" and a 0 means "leave the object." For other problems, other encodings may be used to represent a solution; these include gray, tree, and real-valued encodings, among others.

Evaluating fitness

In addition to representing organisms in a population, a GA needs to have some way of determining how *fit* a solution is. For example, the organism $i = \langle 01010 \rangle$ should be considered more fit than the previous, since it has a value of 32. Although it looks like the fitness of i can be expressed as $i \cdot \mathit{Value}$, this does not take into account whether or not the objects will fit into the bag. For example, $j = \langle 11111 \rangle$, which has a value of 46, is *not feasible* since the combined size of all objects is 31. One way to deal with infeasible solutions is to penalize them so that they don't appear as attractive as feasible solutions. For this problem a reasonable *fitness function* might be $f(x) = x \cdot \mathit{Value} - 3g(x)$, where g is the excess size, e.g., $g(x) = \max(x \cdot \mathit{Size}, 10) - 10$, and 3 is an arbitrarily selected "penalty" for exceeding the capacity. Thus, for solutions i and j, $f(i) = 32$ (still) and $f(j) = -17$.

Producing offspring

The basic idea of GAs is to explore decision space by generating new solutions from previous ones. This process, known as "crossover," is analogous to the way chromosomes line up and then swap portions of their genetic code. For example, consider the following organisms:

$$\begin{array}{lccccccc} x_1: & \langle & 1 & 1 & 0 & 1 & 0 & \rangle \\ x_2: & \langle & 0 & 0 & 1 & 0 & 0 & \rangle \end{array}$$

When x_1 and x_2 are lined up with each other, as above, a point along the strings is selected at random and the portions to the right of that point are exchanged to produce *two* offspring. For example, if object 3 was randomly chosen as the crossover point, the following two new organisms would result:

$$\begin{array}{lcccc|ccccc} x_1': & \langle & 1 & 1 & & 1 & 0 & 0 & \rangle \\ x_2': & \langle & 0 & 0 & & 0 & 1 & 0 & \rangle \end{array}$$

The foregoing approach for generating offspring is referred to as "one-point crossover," for the obvious reasons. An alternative that sometimes improves convergence considers each position in turn and, using a random draw, decides whether or not to perform the swap. This latter approach is referred to as "uniform crossover."

Competing for survival

A straightforward way of incorporating crossover in a GA is to select two organisms to compete for the opportunity to produce offspring. This competition-based process is usually referred to as "probabilistic binary tournament selection." For example, if x_1 and x_2 are randomly selected organisms, then the more fit of the two is determined to be the winner. To avoid reaching a homogeneous population too quickly, however (and hence a poor solution), some diversity is maintained by letting the less fit organism sometimes win, e.g., with a probability of 25%. The tournament selection algorithm is as follows:

```
for each organism $i$ desired in the new generation do
    select two organisms, $x_1$ and $x_2$, from the old generation at random
    if ran.nextFloat() $< 0.75$ then
        let organism $i$ be the more fit of $x_1$ and $x_2$
    else
        let organism $i$ be the less fit of $x_1$ and $x_2$
```

where `ran` is an instance of class `java.util.Random`, and `nextFloat()` is a method that returns a random number r such that $0 \leq r < 1$. After filling a new generation with competition winners, those winners can be taken successively in pairs and replaced (using crossover) by their offspring.

A popular alternative to tournament selection, "stochastic universal selection," simulates the spin of a roulette wheel whose slots are proportional to each organism's fitness. After a single spin, markers, all equally spaced, are placed along the outside of the wheel. The number of markers landing in each slot (possibly zero) determines the number of copies of an organism that are placed in the next generation.

Regardless of the selection technique, an additional measure, mutation, is typically used to maintain a diverse population and to recover genes that may have been lost from the "gene pool." After populating a new generation, each bit of each organism is changed (from 0 to 1, or from 1 to 0) with a very small probability, e.g., 5 in 1000.

A simple genetic algorithm

Putting all of this together into a simple GA is not too difficult. The basic algorithm is as follows, in which P_t is the population in generation number t:

```
$t = 0$
new population ($P_t$)
**if** termination condition not met **then**
  $t = t + 1$
  $P_t$ = select($P_{t-1}$)
  crossover($P_t$)
  mutate($P_t$)
```

Programs that implement the algorithm can be structured in a variety of ways, however, and the design decisions that are made can profoundly affect the ease with which new problems can be accommodated. In the design that follows, two complementary approaches are taken to enhance flexibility:

- Provide a taxonomy of classes that allows customization at a different levels of abstraction. Selection schemes, crossover techniques, and even entirely different algorithms all fit into the same framework.
- Use abstraction techniques that require only that certain behavior be implemented without dictating how it is achieved. Importantly, any GA encoding technique, whether binary, real-valued, or otherwise, can then be accommodated since it is hidden.

Vitri

Using the principle of correspondence, programs are designed to mirror the problem domain, so we have classes representing populations and organisms, as well as a genetic algorithm itself, which keeps track of problem parameters and controls execution of the program. A diagram of the core Vitri classes is shown in Figure 1 using UML, a graphical modeling language for describing program structure. In the diagram, closed arrowheads indicate a subclass or "is-a" relationship and open ones with dashed lines indicate a "uses" relationship. For instance, a `SimpleGA` *is-a* `GeneticAlgorithm`, and a `Population` *uses* one or more `Organisms`. The abstract class `Population` leaves its selection scheme undetermined: a `BtsPopulation` is a population that provides a binary tournament selection scheme, whereas a `SusPopulation` is a population that uses stochastic universal selection.

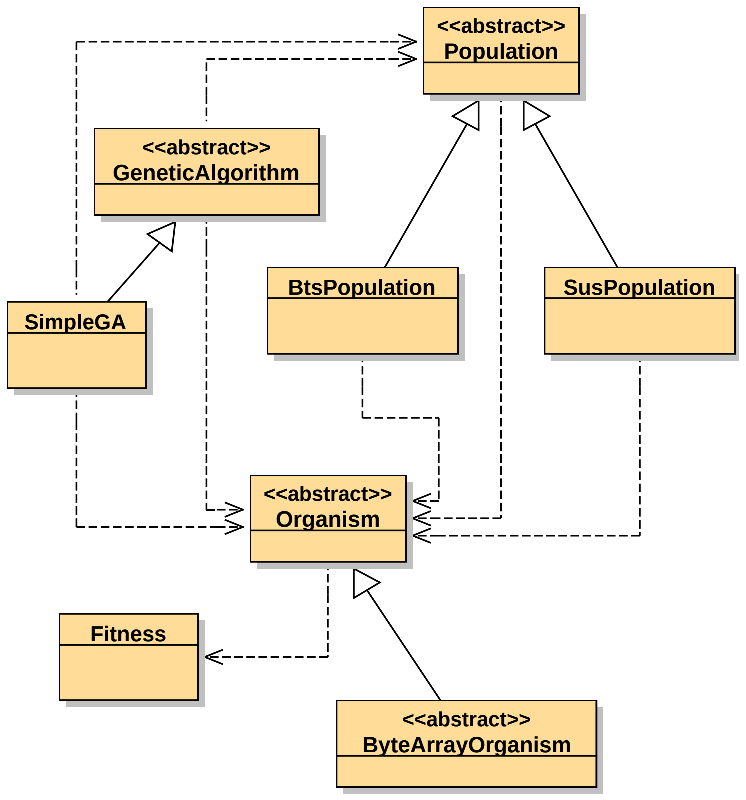

Figure 1: Class Structure used in Vitri

To make use of this program structure a user must do two things:

1. Extend class `Organism` with a concrete class that models the problem of interest.

2. Implement a main method that instantiates a population and runs a genetic algorithm.

As an alternative to item 1, users may choose to extend class `ByteArrayOrganism`, which partially implements class `Organism`. A `ByteArrayOrganism` supports binary encodings or other string-like encodings with each position assuming as many as 2^8 different states. Because this class implements basic genetic operators for crossover and mutation, the amount of coding for new problems is minimal.

To satisfy item 2 users implement a main method that sets parameters and initiates the search process. A tiny main program that uses default settings is shown below.

```
1.      public static void main(String[] args) {
2.            BtsPopulation pop = new BtsPopulation(100);
3.            pop.fill(new MyOrganism());
4.            pop.randomize();
5.            GeneticAlgorithm ga = new SimpleGA(pop);
6.            MyOrganism fittest = (MyOrganism) ga.run(50);
7.      }
```

In line 2 the program instantiates a population that uses binary tournament selection and is capable of holding 100 organisms. Line 3 fills that population with copies of an organism (`MyOrganism`) whose class is user-defined. Line 4 completes the setup of the initial population by randomizing each organism it contains. Lines 5 and 6 create and run a simple genetic algorithm on the population for 50 generations. On completion, the variable `fittest` contains the best solution found during the GA run.

To better understand how this program works, let's take several of the core Vitri classes in turn and describe their use and behavior. For a detailed listing of the signatures of fields, constructors, and methods, see Appendix A. The full API description in Javadoc is provided with the Vitri 2.0 distribution.

Class GeneticAlgorithm

This class is designed to accommodate multiple, coexisting algorithmic implementations of a genetic algorithm. Subclasses are required to implement a constructor, which sets the initial population, and the `run(int, int)` method. To the extent possible, subclasses should also update the `population` and `generation_number` fields during iteration; doing so enables, for instance, the use of dynamic or adaptive penalty functions in organism evaluation. They should also use the value of `print_level` to control the amount of detail printed during execution. By design, this class is intended to be *reentrant* in the sense that a genetic algorithm, say, can be run within a genetic algorithm without any interference between the two levels of search.

Class SimpleGA

This class implements a simple genetic algorithm that begins with an initial population, and then iteratively (a) selects a mating pool, (b) produces offspring by pairing members of the pool and performing crossover, and (c) mutates each of the offspring to produce a new generation. Termination of this iterative process is based on two limits: a specified number of generations, and a specified number of generations without improvement in the fittest organism. An incumbent organism — the fittest seen at any point in the computation — is maintained and is made available to the selection operation (implemented by a given population) for implementing an elitism policy.

Class Population

This class provides a skeletal implementation of a population, which maintains a collection of `Organisms`. Operations provided by `Population` are straightforward, and often just iterate over `Organism` instances. For example, the `mutate` method successively mutates each organism in the population.

Subclasses need provide only a selection policy to realize a concrete population; this is accomplished by implementing the abstract method `select(Organism)`, which creates a new population from an old one using a customized selection process. Note that such implementations should place organism *clones*, and not the organisms themselves, into the new population. Also, to improve convergence of the search process, a clone of the incumbent solution may be placed in the new population; it is given as an argument to the `select` method.

Class Organism

This class provides a skeletal implementation of an organism. A key consideration in its design is that the contractual obligations imposed on an organism's realization focus on the organism's behavior and not on its implementation. That is, an organism's encoding, e.g., 0/1 versus real-valued encodings, is hidden, allowing any encoding approach to be taken and used within this framework. Subclasses need implement only the following methods: `randomize`, `crossover`, `mutate`, and `evaluate`.

Because fitness evaluation can be costly in some applications, an organism is designed so that evaluation need happen only once, which is accomplished by caching evaluation results. This policy, however, places responsibility on an organism's storage-modifying operators to clear its cache, typically by calling the `unsetFitness` method.

Class Fitness

A fitness is a mutable value produced when evaluating an an organism. Its role is to act as a hook for GA schemes that need to manage or incorporate multiple values for each fitness, such as raw fitness values or multi-objective values.

Class ByteArrayOrganism

A ByteArrayOrganism provides a common implementation for Organisms that are represented as an array of genes in which each gene has a maximum of 2^8 (256) possible states. A subclass of `ByteArrayOrganism` need implement only a constructor and the following methods:

- `crossover`, which may be implemented in terms of two methods already provided by this class: the `uniformCrossover` and `onePointCrossover` methods,
- `mutate`, which may be implemented in terms of the `mutate(double)` method provided, and
- `evaluate`, which should return the fitness of the organism.

Customizing an Organism

Using Vitri for genetic search requires programming, but no changes are required to existing code. The `Organism` class provides the primary plug point for solving new problems, and other parts of the framework can be similarly used to customize the behavior of the genetic algorithm itself. Here we describe how to solve new problems by defining a subclass of `ByteArrayOrganism`. Subclassing `ByteArrayOrganism` rather than class `Organism` is appropriate for problems that have binary or other string-like encodings, since many of the GA operations are already provided by `ByteArrayOrganism`.

To solve a new problem using `ByteArrayOrganism` we need to do two things: extend class `ByteArrayOrganism`, and implement a main method that runs a GA. We can do both of these in a single class definition, a template of which is shown in Figure 2. The template displays two important features: an overall structure for adding a new problem type to Vitri, and places where customization may occur, indicated in *italics*. With respect to structure, the new class, `MyOrganism`, has a constructor, the three requisite methods for specialization (`crossover`, `mutate`, and `evaluate`), and the static `main` method for running the GA. Combined, the `MyOrganism` constructor and `evaluate` method encode the new problem; the other methods affect the running behavior of the GA.

The `MyOrganism` constructor is responsible for initializing the space used by a single organism. The `storage` field is a byte array (the organism's "chromosome") of length $\langle size \rangle$ that can represent problems with up to $8^{\langle size \rangle}$ different states, since each position is a byte, or 8 bits. Because problems often require fewer states, the `values` field may be set to $\langle states \rangle$, the number of possible states at each position, e.g., $\langle states \rangle$ is 2 for a typical binary encoding.

```
public class MyOrganism extends ByteArrayOrganism {

    public MyOrganism() {
        storage = new byte[⟨size⟩];
        values = ⟨states⟩;
    }

    protected Fitness evaluate() {
        calculate how fit this organism is
        return new Fitness(⟨fitness⟩);
    }

    public void crossover(Organism org) {
        uniformCrossover(org);    // or choose onePointCrossover
    }

    public void mutate() {
        mutate(⟨mutation probability⟩);
    }

    public static void main(String[] args) {

        GeneticAlgorithm.random.setSeed(⟨seed⟩);

        // or choose SusPopulation below
        BtsPopulation pop = new BtsPopulation(⟨population size⟩);
        pop.fill(MyOrganism());
        pop.randomize();

        GeneticAlgorithm ga = new SimpleGA(pop);
        System.out.println(ga);
        MyOrganism fittest = (MyOrganism) ga.run(⟨number of generations⟩);
    }
}
```

Figure 2: Customizing an Organism in Vitri

The `evaluate` method is where the organism's byte array is interpreted, and this is where the problem itself is introduced. It calculates a scalar, the relative level of fitness of a specific organism, by inspecting the byte array. That scalar value, ⟨*fitness*⟩ is not returned directly but rather is "wrapped" by a `Fitness` object, which is returned to and used by the GA.

Now that the problem has been encoded, we turn to customizing the GA's behavior. A `ByteArrayOrganism` provides both crossover and mutation methods that can be used to do so. In the template, uniform crossover is chosen by calling the `ByteArrayOrganism`'s `uniformCrossover` method, although one-point crossover could also have been used. Of course, a completely custom crossover technique could be implemented here by the user. With respect to mutation, the `mutate(double)` method in `ByteArrayOrganism` takes a parameter, ⟨*mutation probability*⟩, that sets the likelihood that a single position in the byte array is mutated. For problems with a `values` field greater than 2, the mutation of a particular position results in a random change to a different value. As with `crossover`, `mutate` can alternatively implement a completely custom technique.

Although a similar static `main` method has been described earlier, the implementation in Figure 2 shows a few additional details, such as how to set a seed for Vitri's random number generator by giving the ⟨*seed*⟩ parameter to `setSeed`. It also comments that `SusPopulation` can be used in place of `BtsPopulation` to perform the selection operation by stochastic universal selection. Finally, this implementation of `main` makes explicit the settings for ⟨*population size*⟩ and ⟨*number of generations*⟩, and adds a `println` statement that echos GA settings to the user's display.

The Knapsack Problem

To see how a specific problem is "plugged in" to the Vitri framework, we return to the knapsack problem. The implementation is shown in Figure 3, which essentially fills out the template in Figure 2.

Recall from earlier discussions that a specific knapsack can be represented as a binary string in which a 1 means "take the object" and a 0 means "leave the object." As a result, the `values` field is set to 2 in the `KnapsackOrganism` constructor. The size of the byte array is the number of objects that may potentially go into the knapsack; its value comes from the static `size` field, which is set to be the length of the `benefit` and `cost` arrays. The values in those arrays are taken from the earlier problem statement for the knapsack problem.

The `evaluate` method implements in Java the fitness function for knapsacks, although it relies on a dot-product method that must be implemented. Other parts of class `KnapsackOrganism` have already been described.

Experimenting with Knapsacks

Before trying to solve your own problem, try experimenting with an existing one to see how Vitri works. The `KnapsackOrganism` class in the Vitri distribution solves a larger instance of the knapsack problem than that described earlier. The problem has 40 items that may be placed into the knapsack, which has an upper limit on cost of 200. The optimal solution to the problem has a benefit of 306, which was found via integer programming.

Begin by compiling and running Vitri with this problem to see how well you can fill your knapsack. Also try using some of the different features of Vitri to see what effect they have on output and solution quality. Here are some things to try.

```
public class KnapsackOrganism extends ByteArrayOrganism {

    private static final int[] benefit = { 1, 13, 12, 19, 1 };
    private static final int[] cost = { 7, 3, 5, 7, 9 };
    private static final int size = benefit.length;

    private static final int PENALTY = 3;
    private static final int MAX_COST = 10;

    public KnapsackOrganism() {
        storage = new byte[size];
        values = 2;
    }

    public int dot(int[] a) {
        int sum = 0;
        for (int i = 0; i < storage.length; i++)
            sum += storage[i] * a[i];
        return sum;
    }

    protected Fitness evaluate() {
        int b = dot(benefit);
        int c = dot(cost);
        return new Fitness(b - (c > MAX_COST ?
                                PENALTY * (c - MAX_COST) :
                                0));
    }

    public void crossover(Organism org) {
        uniformCrossover(org);
    }

    public void mutate() {
        mutate(0.008);
    }

    public static void main(String[] args) {

        GeneticAlgorithm.random.setSeed(0);

        BtsPopulation pop = new BtsPopulation(50);
        pop.fill(new KnapsackOrganism());
        pop.randomize();

        GeneticAlgorithm ga = new SimpleGA(pop);
        System.out.println(ga);
        KnapsackOrganism fittest = (KnapsackOrganism) ga.run(25);
    }
}
```

Figure 3: Solving the Knapsack Problem in Vitri

1. Run `KnapsackOrganism` "as is" by compiling all files

   ```
   javac *.java
   ```

   and then invoking the class

   ```
   java KnapsackOrganism
   ```

   The output should look something like that shown in Figure 4 using the default print level (2). It includes not only the GA parameters that are used, but also, at each generation, the fittest solution seen so far (the "incumbent"), the fittest solution in the current generation, and the minimum, maximum, and average fitnesses of all organisms in the current generation. Individual solutions are shown with a particular syntax. For instance, the best solution found is

   ```
     [305.00] < 0 1 1 1 0 1 ... 0 1 1 1 1 1 > b = 305 c = 200
   ```

   which has a fitness (in square brackets) of 305, a binary array with values shown in angle brackets, and finally a benefit (b) of 305, and a cost (c) of 200.

2. Change the seed in the static `main` method of `KnapsackOrganism.java`, recompile it, and observe the different solutions obtained with different seeds. Some will be better and some will be worse. This behavior has to do with the non-deterministic nature of GAs.

3. Reduce the print level to 1 by adding the line

   ```
   ga.setPrintLevel(1);
   ```

   just above the line with `ga.run(50)`, which runs the GA for 50 generations. Recompile and re-run. Notice that intermediate solutions are not printed, only the final solution. A print level of 0 turns off all printing during a GA run, including display of the final solution. This behavior is useful when a GA is embedded in another program, which somehow uses the fittest organism returned by a call to `ga.run`. Note that the static `main` method of `KnapsackOrganism.java` simply ignores the result of that call and exits.

4. Look again at Figure 4 and examine the GA parameters that are echoed at the top of the output. It indicates that classes `Population` and `BtsPopulation` have some default settings that may be changed. Explore the Javadoc APIs provided in the Vitri distribution for these two classes and find descriptions of the following methods

   ```
   public void setCrossoverProbability(double p)
   public void setElitism(boolean b)
   public void setSelectFittestProbability(double p)
   ```

   Try invoking these methods from the static `main` method of `KnapsackOrganism` with different settings. As an example, the call

   ```
   pop.setElitism(true);
   ```

   changes the binary tournament scheme so that the incumbent organism receives special treatment during selection: if it happens not to be selected for the new generation then it is explicitly added to that generation anyway. This call may appear anywhere after population `pop` is defined but before the `ga.run` method is called.

```
---- Genetic Algorithm Parameters ----
Genetic algorithm class: SimpleGA
Print level: 2
Population class: BtsPopulation
  size: 100
  crossover_probability: 0.75
  elitism: false
  select_fittest_probability: 0.75
Organism class: KnapsackOrganism
  storage.length: 40
  values: 2
  crossover: uniform
  mutation_probability: 0.008

---- Generation 1 ----
Incumbent organism:
  [235.00] < 0 0 1 1 1 1 ... 0 1 0 1 1 1 > b = 235 c = 196
Fittest in current generation:
  [235.00] < 0 0 1 1 1 1 ... 0 1 0 1 1 1 > b = 235 c = 196
Min/Max/Avg fitness: -124.00/235.00/140.67
---- Generation 2 ----
Incumbent organism:
  [260.00] < 0 1 0 1 0 1 ... 1 1 1 0 1 1 > b = 260 c = 195
Fittest in current generation:
  [260.00] < 0 1 0 1 0 1 ... 1 1 1 0 1 1 > b = 260 c = 195
Min/Max/Avg fitness: -49.00/260.00/151.33

...

---- Generation 49 ----
Incumbent organism:
  [305.00] < 0 1 1 1 0 1 ... 0 1 1 1 1 1 > b = 305 c = 200
Fittest in current generation:
  [304.00] < 0 1 1 1 0 1 ... 0 1 0 1 1 1 > b = 304 c = 200
Min/Max/Avg fitness: 134.00/304.00/250.19
---- Generation 50 ----
Incumbent organism:
  [305.00] < 0 1 1 1 0 1 ... 0 1 1 1 1 1 > b = 305 c = 200
Fittest in current generation:
  [292.00] < 0 1 1 1 0 1 ... 0 1 1 0 1 1 > b = 292 c = 198
Min/Max/Avg fitness: 119.00/292.00/255.12
---- Final Solution ----
Organism:
  [305.00] < 0 1 1 1 0 1 ... 0 1 1 1 1 1 > b = 305 c = 200
Number of generations: 50
Evaluation count: 5100
```

Figure 4: Abbreviated Output from the Knapsack Problem

5. Try using stochastic universal selection by replacing `BtsPopulation` with `SusPopulation` as follows:

   ```
   SusPopulation pop = new SusPopulation(50);
   ```

Additional Exercises for the Reader

1. Solving some problems with a GA can take a long time, so effectiveness of the search process may warrant some scrutiny. If the total "computation cost" is expressed by the number of times the `evaluate` method is called, how can one best use available CPU cycles to get "good" solutions? That is, you might try to answer questions about the knapsack problem such as: How many evaluations are being used to produce a given solution? What is an appropriate ratio of the population size to the number of generations? What effect do mutation and selection rates have? Try experimenting a little, but remember: this is a *stochastic* problem, and it pays to think about statistical averages (using various seeds on the same problem instance).

2. Because constraints are only *penalized*, and not enforced, it is possible that a GA can return an *infeasible* solution. Try running the knapsack problem with various seeds to see if this happens for you (the actual costs exceeds the allowable cost in this situation). Although one might be inclined to increase the penalty, such a tactic can actually worsen convergence to good solutions. Try setting your penalty to 400, e.g.,

   ```
   private static final int PENALTY = 400;
   ```

   and see what happens to the quality of your solutions. An alternative is to start out with a low penalty and gradually increase it with each passing generation. Try making this change by modifying the `evaluate` method of class `KnapsackOrganism`. Can convergence be improved using this technique? Can the number of generations or the population size be reduced?

3. There's a very simple heuristic available for solving the knapsack problem. Ever hear of "bang for the buck" or "benefit-cost ratio?" Implement a procedure or shell script using a combination of benefit-cost ratio and sorting, and see how it compares with a GA.

4. Find a different problem to optimize. Describe the problem, implement a class (analogous to class `KnapsackOrganism`), and see how good a solution you can obtain. Be sure to present your results in a way that someone unfamiliar with your problem can understand.

Appendix A Class Summaries

Class GeneticAlgorithm

```
public abstract class GeneticAlgorithm
```

Field Summary

```
public static final java.util.Random random
protected Population population
protected int generation_number
protected int print_level
```

Constructor Summary

```
protected GeneticAlgorithm(Population p)
```

Method Summary

```
public Organism run(int max_generations)
public abstract Organism run(int max_generations,
                             int max_generations_wo_improvement)
public Population getPopulation()
public int getGenerationNumber()
public int getPrintLevel()
public void setPrintLevel(int level)
public String getSettings()
public String toString()
```

Class SimpleGA

```
public class SimpleGA extends GeneticAlgorithm
```

Method Summary

```
public SimpleGA(Population pop)
public Organism run(int max_generations,
                    int max_generations_wo_improvement)
```

Class Population

```
public abstract class Population
```

Field Summary

```
    private double crossover_probability
    protected Organism[] storage
```

Constructor Summary

```
    protected Population(int size)
```

Method Summary

```
    public void fill(Organism prototype)
    public void randomize()
    public Iterator iterator()
    public int size()
    public void setCrossoverProbability(double p)
    public double getCrossoverProbability()
    public abstract Population select(Organism incumbent)
    public void crossover()
    public void mutate()
    public Organism getRandomOrganism()
    public Organism getFittestOrganism()
    public double getAvgFitness()
    public double[] getFitnessStats()
    public String getSettings()
    public String toString()
```

Class Organism

```
public abstract class Organism implements Cloneable, Serializable
```

Field Summary

```
private static int evaluation_count
protected transient Fitness fitness
```

Method Summary

```
protected Object clone()
public abstract void randomize()
public abstract void crossover(Organism org)
public abstract void mutate()
protected abstract Fitness evaluate()
public static int getEvaluationCount()
public static int resetEvaluationCount()
public void unsetFitness()
public final double getFitness()
public Fitness getFitnessObject()
public void setFitnessObject(Fitness f)
public String getSettings()
public String toString()
```

Class Fitness

```
public class Fitness implements Cloneable, Serializable
```

Field Summary

```
protected double value
```

Constructor Summary

```
public Fitness(double value)
```

Method Summary

```
public Object clone()
public double getValue()
public void setValue(double value)
public String toString()
```

Class ByteArrayOrganism

```
abstract public class ByteArrayOrganism extends Organism implements Cloneable
```

Field Summary

```
protected byte[] storage
protected int values
```

Method Summary

```
public Object clone()
public void randomize()
public void uniformCrossover(Organism org)
public void onePointCrossover(Organism org)
public void mutate(double mutation_probability)
public String getSettings()
public String toString()
```