The DEA: A Framework for Exploring Evolutionary Computation

Mark M. Meysenburg
Department of Information Science and Technology
Doane College
mmeysenburg@doane.edu

Abstract

The DEA (Doane Evolutionary Algorithm) is a general-purpose framework of classes, allowing easy exploration of many different kinds of evolutionary computation (EC) algorithms, all in a single context. Written in Java, the DEA contains abstract evolutionary computation classes, such as Individual, Operator, and Problem, that can be extended to implement genetic algorithm systems, genetic programming systems, or any system in which populations of Individuals move from generation to generation according to one or more Operators.

The DEA framework could be used either in a situation where the users are interested in applying EC techniques to numerous types of problems, or in a situation where the users are interested in experimenting with the development and implementation of different EC techniques.

Introduction

The Doane Evolutionary Algorithm (DEA) framework of classes is a general-purpose system for experimenting with evolutionary computation (EC) algorithms. This paper discusses the packages, classes, and applications that make up the DEA framework, and how students interested in exploring EC techniques might use the framework.

In order to set the stage for discussing the DEA, a brief introduction to the EC technique is appropriate.

A brief introduction to EC

EC algorithms seek to solve various types of difficult problems through simulated evolution. In a typical EC algorithm, a population of individuals is created at random. Each individual in the population represents a potential solution to the problem at hand. In some problem-specific manner, each individual is assigned a fitness value, which represents the degree to which the individual satisfies or solves the problem. However, since the first population of individuals is created at random, the individuals in the population are unlikely to have high fitness.

Once the initial population has been created, the population is moved from generation to generation via several genetic operators, which mimic evolutionary processes. For instance, EC algorithms typically have operations that simulate sexual reproduction, biological mutation, and natural selection. The operators are applied in sequence to produce a new generation of the population. In a properly configured EC algorithm, individuals in the population become better and better solutions to the problem at hand, until at least one of them reaches an acceptable fitness level.

Two broad categories of EC algorithms exist: genetic algorithms (GAs) and genetic programming (GP) systems. GAs can be thought of as evolving a specific solution to a problem. GAs typically work with linear strings of bits, ints, or floats. Each of these strings, called chromosomes, encodes a solution to the problem.

GP systems, rather than evolving a specific solution instance, can be thought of as evolving programs capable of solving a problem. GP systems often work with program trees, represented in Lisp-like syntax.

Regardless of the type of EC algorithm, several commonalities can be found: problems, individuals, and genetic operators, for instance. These commonalities form the foundation of the DEA framework.

A brief introduction to the DEA framework

The DEA is a general-purpose framework of classes, allowing easy exploration of many different kinds of EC algorithms, all in a single context. Written in Java, the DEA contains abstract evolutionary computation classes, such as Individual, Operator, and Problem, that can be extended to implement GA systems, GP systems, or any system in which populations of individuals move from generation to generation according to one or more operators.

The DEA framework includes several sample derived classes. For example, Individuals and Operators are included for fixed-length, bit-string chromosome GAs; for fixed-length, floating-point chromosome GAs; and for fixed-length, integer chromosome GAs. The framework also includes Individuals and Operators for program-tree-based GP systems. Several of the Operators are general, and can be applied without change to Individuals of any type. For example, the Operator implementing tournament selection can be applied to any population of Individuals, without regard to their specific structure.

The DEA framework could be used either in a situation where the users are interested in applying EC techniques to numerous types of problems (less programming intensive), or in a situation where the users are interested in experimenting with the development and implementation of different EC techniques (more programming intensive).

In the first case, users can employ the existing DEA classes to solve different problems with a minimum of programming effort. In essence, the users here only have to extend the Problem class to describe the problem they are trying to solve. To do this, the users must implement two methods: one that knows how to create new Individuals that represent potential solutions to the problem at hand, and one that knows how to evaluate the fitness of these Individuals. Once this class has been created, a simple main function can be written (based on examples in the framework) to actually execute the algorithm.

In the second case, users interested in the exploration, development, and implementation of EC techniques can use the DEA as a starting point. New types of Individuals or Operators are easy to implement. The use of inheritance in the framework allows users to easily switch between different Individuals or Operators, without having to re-write any of the supporting code. The framework is flexible enough to support GUI interfaces, parallelized EC algorithms, and more.

Origins

The DEA framework was developed over a period of time, primarily to support the author's research into the effect of random number generator quality on the performance of genetic algorithms. In this research, we ran a simple GA on a collection of almost fifty different test problems, using several different pseudo-random number generators (PRNGs) of varying quality.

In the process of this work, it was desirable to have a single system that could easily switch between problems, with minimal code changes or recompilation. The dynamic class loading capabilities of Java, and the structure of the DEA framework, made this possible; as a result, we could focus on our research question without becoming bogged down in code issues. GA execution speed was not an overriding factor in our work. In addition, portions of the research were carried out by second-year undergraduate students, and so simplicity of the GA system was a must.

Although the DEA framework was initially designed for research purposes, its flexibility and simplicity make it a good tool for experimenting with a wide variety of EC techniques.

DEA structure

The following sections describe the overall structure of the DEA framework, its packages, and its classes.

DEA packages

The package structure of the DEA framework is shown in Figure 1.

The two high level packages of the framework are **dea** and **prng**. The **prng** package is a relic of the research questions which drove the initial development of the framework. This package contains several different PRNGs, of varying quality. Some of the generators in this package require the Colt library of Java classes for high-speed numerical computation in Java. [1] The **prng** package is not further discussed in this paper.

The **dea** package contains the fundamental base classes of the DEA framework, and two packages: **apps** and **kits**. The **apps** package is designed to hold applications that utilize the DEA framework; as of now, it contains only the package **tests**, which holds demonstration applications showing how the framework can be employed.

The **kits** package holds derived classes, specializations of the base DEA classes for specific types of GA or GP systems. These specializations are contained in the several packages of the **kits** package: **binarychromosome**, **floatchromosome**, **generaloperators**, **generalreporters**, **intchromosome**, and **ptreechromosome**.

The **binarychromosome** package contains classes for the “classic” GA: a GA working with fixed-length, binary string chromosomes.

The **floatchromosome** package contains classes for GAs that employ fixed-length chromosomes of floats.

The **generaloperators** package contains evolutionary operators that can be applied to populations of any type individual.

The **generalreporters** package contains specializations of the Reporter class that can be used by any EC implementation.

The **intchromosome** package contains classes for GAs that use fixed-length chromosomes of ints.

The **ptreechromosome** package contains classes for genetic programming (GP) applications that evolve program trees.

Fundamental classes of the DEA framework

The root-level **dea** package contains the fundamental classes and interfaces required to capture the essence of an EC algorithm. The contents of the **dea** package are the DEA class, an extension of the Java Thread class; the abstract classes Individual, Operator, and Problem; and the Reporter interface.

Classes that extend the Problem base class represent problems that can be solved by an EC algorithm. Classes derived from Problem must define two methods. One of these methods, getInstanceOfIndividual( ), must produce new, randomly-created instances of Individual which represent solutions to the problem. The other method, evaluateInstanceOfIndividual( ), must compute the fitness of a given Individual. The fitness must be returned as a double, with higher values representing higher fitness. Users of the DEA framework interested in solving problems, rather than experimenting with EC techniques or representations, need only to create classes derived from Problem, tailored to their needs.

Classes derived from the abstract class Individual represent the chromosomes used in EC algorithms. There are pre-built classes derived from Individual, in the **kits** sub-packages, for common EC chromosome types: fixed-length binary strings, fixed-length float arrays, fixed-length integer arrays, and GP program trees. Users interested in experimenting with EC techniques can easily create new, custom-tailored representations by creating new classes derived from Individual.

Classes derived from the abstract class Operator represent the genetic operators used in the EC algorithm to move from generation to generation. Three common types of genetic operators are crossover, which models sexual reproduction; mutation, which models biological mutation; and selection, which models natural selection or survival of the fittest. In general, crossover and mutation operators are specific to the chromosome representation used in the EC algorithm, while selection operators are typically independent of the representation. Thus, there are pre-built crossover and mutation operators for each of the derived Individual types in the **kits** sub-packages, and a general-purpose tournament selection operator in the **kits.generaloperators** package.

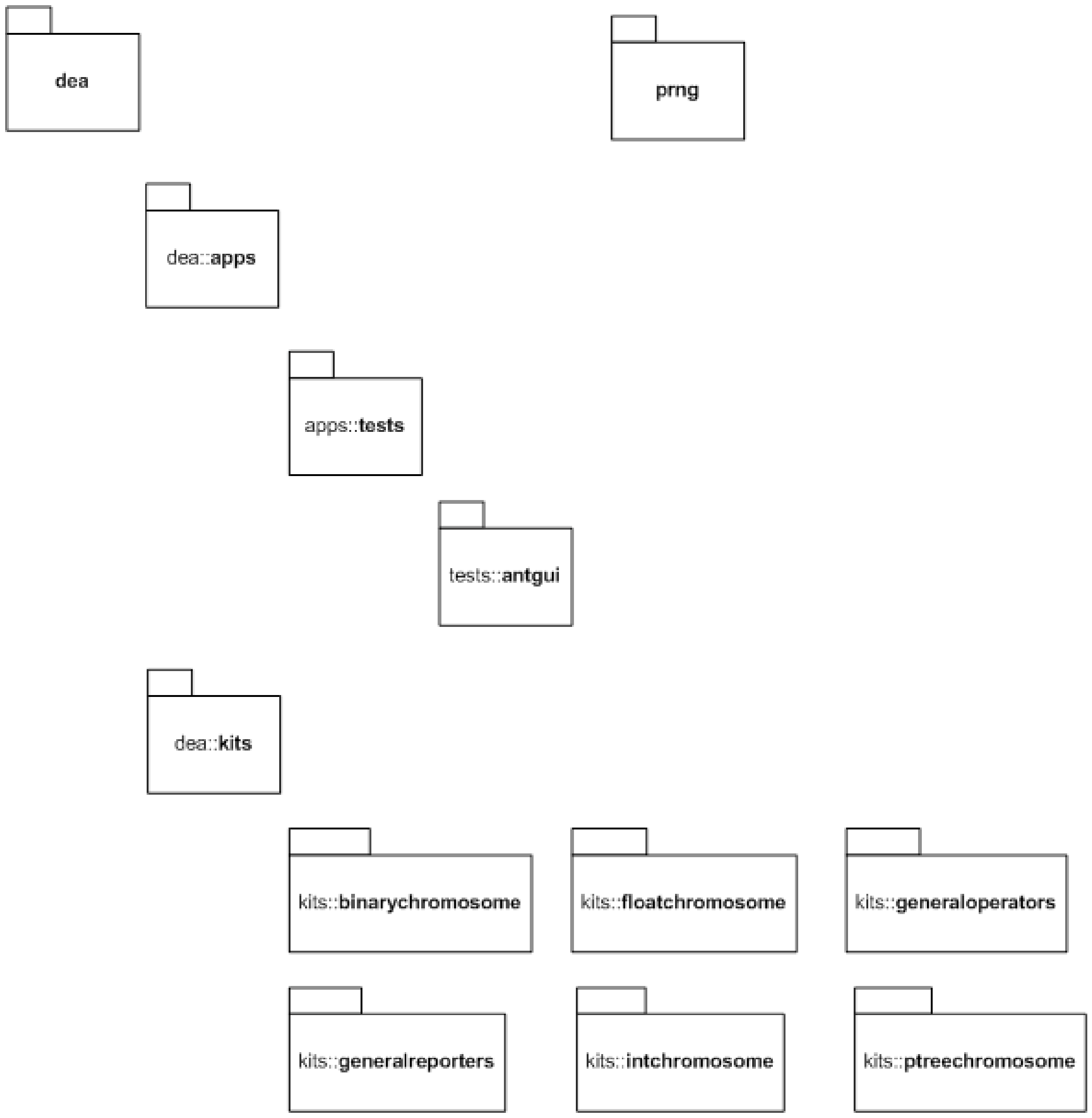

Figure 1 DEA framework package structure

Classes that implement the Reporter interface aren't strictly part of the EC process; instead, they are used to report end-of-generation statistics during the EC run. For instance, end-of-generation best, worst, and average fitness may be reported. The **kits.generalreporters** package contains implementations of Reporter to dump these statistics to the console or to a text file. Another class, in the **tests.antgui** package, shows how a Reporter can be used to connect the EC algorithm to a GUI.

The DEA class, derived from the Java Thread class, ties the above classes and interfaces together and actually executes an EC run. An EC run follows this sequence:

1. An initial population is created, as a Java Vector filled with a specified number of randomly created Individuals. These Individuals come from the user-defined instance of the Problem class, via the getInstanceOfIndividual() method.

2. Repeat the following until a specified fitness target is reached, or until a specified maximum number of generations is reached:

    a. Apply each of the Operators for this EC algorithm in sequence. Instances of the Operator class are stored in an array; each Operator is called in turn to perform its operation on the Vector of Individuals. A typical sequence of operators would be: mutation, crossover, evaluation (where the new fitness values of each Individual are computed), and selection.

    b. Use the Reporter object to report end-of-generation statistics.

3. Use the Reporter object to report end-of-run statistics. For example, the best-ever solution to the problem might be reported at the end of the run.

How the DEA framework might be used

The DEA framework could be used either by students interested in applying EC techniques to real-world problems, or by students interested in implementing new EC representations or operators. This section provides examples of how each of these uses might be realized.

Applying the DEA framework

To provide an example of how a student might use the DEA framework to solve a problem, with a minimal amount of coding, we shall consider the "Bug Bomb" example that is provided with the Generator Macro package [2], an easy-to-use GA for the Microsoft Excel spreadsheet application.

In the Bug Bomb problem, there are 12 wasp nests in an attic, and we have three insecticide bombs to try to destroy the wasps. Each nest has a different number of wasps in it, and a different location on a 100 by 100 unit grid that represents the attic. The locations of the nests, and the number of wasps in each nest, are shown in Table 1. The task of the GA is to choose the x and y coordinates for the three bug bombs, in order to maximize the number of wasps killed.

The representation for the Bug Bomb problem is very simple: six integers in the range [1, 100] that form the (x, y) coordinate locations for each of the three bug bombs. The fitness of an individual is the number of wasps killed by the bug bombs. The number of wasps killed is given by the following formula:

$$\sum_{i=1}^{12} \min\left(\sum_{j=1}^{3} \frac{141.42 \cdot P_i}{20 \cdot \sqrt{\left(L_{ix} - B_{jx}\right)^2 + \left(L_{iy} - B_{jy}\right)^2} + 0.0001},\quad P_i \right).$$

In the formula, P_i is the number of wasps in nest i, (L_{ix}, L_{iy}) is the (x, y) coordinate of nest i, and (B_{jx}, B_{jy}) is the (x, y) coordinate of bug bomb j. Java code showing how the bug bomb problem might be implemented is shown in Appendix A.

Table 1: wasp nest locations for Bug Bomb problem

Nest number	x coordinate	y coordinate	Population
1	25	65	100
2	23	8	200
3	7	13	327
4	95	53	440
5	3	3	450
6	54	56	639
7	67	78	650
8	32	4	678
9	24	76	750
10	66	89	801
11	84	4	945
12	34	23	967

Extending the DEA framework

As an example of how the DEA framework can easily be extended by students wishing to experiment with EC techniques, let us consider adding a hill-climbing operator for the bug bomb problem.

The new operator takes place each generation, immediately before selection. It finds the fittest individual in the population, and then uses a hill-climbing technique to move that individual to a nearby local maximum. To do this, the operator examines all the possible single-step coordinate moves for each of the bug bombs. New fitness values are computed for each of these, and the coordinates are changed to the best fitness value. This process repeats until none of the steps produces higher fitness; that is, until a local maximum is reached, or until 50 steps have been taken. Sample code showing an implementation of this operator is shown in Appendix B.

Conclusion

The DEA framework is an attempt to distill the nature of an EC algorithm into base components: the Problem to solve, Individuals that represent solutions, and Operators that move populations of Individuals closer and closer to optimal solutions. The framework is flexible enough to accommodate a wide range of EC algorithms, from simple binary string GAs through tree-based GP systems, all within the same context. The framework is useful either for students who wish to use EC techniques for problem solving, or for students who wish to experiment with new EC techniques of their own invention.

References

1. The Colt Distribution Open Source Libraries for High Performance Scientific and Technical Computing in Java, link verified as of 4 March 2004, http://hoschek.home.cern.ch/hoschek/colt/index.htm .
2. Generator package demo download page, link verified as of 4 March 2004, http://www.nli-ltd.com/products/genetic_algorithms/demos.htm .

Appendix A: Bug Bomb problem

```
package edu.doane.mmeysenburg.dea.apps.tests.bugbomb;

import edu.doane.mmeysenburg.prng.*;
import edu.doane.mmeysenburg.dea.*;
import edu.doane.mmeysenburg.dea.kits.intchromosome.*;
import edu.doane.mmeysenburg.dea.kits.generaloperators.*;
import edu.doane.mmeysenburg.dea.kits.generalreporters.*;

public class BugBomb extends Problem {

    private static int[] P = {100, 200, 327, 440, 450, 639, 650, 678,
                              750, 801, 945, 967};

    private static int[][] L = { {25, 65}, {23, 8}, {7, 13}, {95, 53},
                                 {3, 3}, {54, 56}, {67, 78}, {32, 4},
                                 {24, 76}, {66, 89}, {84, 4}, {34, 23} };

    /**
     * Create a new instance of the BugBomb problem, using the specified
     * random number generator.
     *
     * @param prng Random number generator to use for creating new
     * Individuals.
     */
    public BugBomb(PRNG prng) {
        super(prng);
    }

    /**
     * Evaluate the fitness of an Individual representing a solution to
     * this problem.
     *
     * @param ind IntChromosome with six ints in [1, 100], representing
     * the (x, y) coordinates for bug bomb placement.
     *
     * @return A double, representing the number of wasps killed by the
     * three bug bombs.
     */
    public double evaluateInstanceOfIndividual(Individual ind) {
        double fitness = 0.0;

        int[] coords = (int[])ind.getRepresentation();

        for(int i = 0; i < P.length; i++) {
            double sum = 0.0;

            for(int j = 0; j < 3; j++) {
                double num = P[i] * 141.42;
                double den = 20 *
                    Math.sqrt(Math.pow(L[i][0] - coords[j * 2], 2) +
                    Math.pow(L[i][1] - coords[j * 2 + 1], 2))
                    + 0.0001;
                sum += num / den;
            }

            fitness += Math.min(sum, P[i]);
        }

        return fitness;
    }

    /**
     * Return a new, randomly created IntChromosome that represents a
     * solution to the bug bomb problem.
     *
     * @return IntChromosome with six ints in [1, 100], representing
     * the (x, y) coordinates for bug bomb placement.
     */
    public Individual getInstanceOfIndividual() {
        return new IntChromosome(6, 1, 100, prng, formatter);
    }

    /**
     * Return the name of this problem.
     *
     * @return "BugBomb"
     */
    public String toString() {
        return "BugBomb";
    }

    /**
     * Execute the DEA on the Bug Bomb problem, from the command line.
     * Command line arguments are: population size, maximum number of
     * generations, crossover rate, mutation rate, and tourney size
     * for the tournament selection operator.
     */
    public static void main(String[] args) {

        try {
            PRNG prng = new Mersenne();

            Problem prob = new BugBomb(prng);

            Operator[] ops = new Operator[4];
            ops[0] =
             new SinglePointCrossover(Double.parseDouble(args[2]), prng);
            ops[1] =
             new SinglePointMutation(Double.parseDouble(args[3]), prng);
            ops[2] = new Evaluate(prob);
            ops[3] =
             new TournamentSelection(prng, Integer.parseInt(args[0]), true);

            Reporter rep = new StandardOutStringReporter();

            DEA dea = new DEA(prob,
                Integer.parseInt(args[0]),
                ops,
                rep,
                Integer.parseInt(args[1]),
                6947);

            dea.start();
        }
        catch(Exception e) {
            System.err.println("usage: " +
            "java edu.doane.mmeysenburg.dea.apps.tests.bugbomb.BugBomb " +
            "<size> <gens> <chi> <mu> <k>");
        }

    }
}
```

Appendix B: Bug Bomb hill-climbing operator

```
package edu.doane.mmeysenburg.dea.apps.tests.bugbomb;

import java.util.Vector;
import edu.doane.mmeysenburg.dea.*;
import edu.doane.mmeysenburg.dea.kits.intchromosome.*;

/**
 * Hill climbing operator for the Bug Bomb problem. Finds the best individual
 * in the population, and uses hill climbing to move it to a local maxima.
 *
 * @author Mark M. Meysenburg
 * @version 03/06/2004
 */
public class HillClimb extends Operator {

    private Problem prob;

    private int[] stepDirs = new int[6];

    private int[] stepCoords;

    private int[] startCoords = new int[6];

    public HillClimb(Problem prob) {
        this.prob = prob;
    }

    public void operate(Vector pop) {
        hillClimb((IntChromosome)getBest(pop));
    }

    private void hillClimb(IntChromosome ind) {
        double bestFitness = prob.evaluateInstanceOfIndividual(ind);
        stepCoords = (int[])ind.getRepresentation();

        for(int i = 0; i < 6; i++) {
            stepDirs[i] = -1;
            startCoords[i] = stepCoords[i];
        }

        int steps = 0;
        while(stepDirs[0] != 0 && stepDirs[1] != 0 && stepDirs[2] != 0 &&
            stepDirs[3] != 0 && stepDirs[4] != 0 && stepDirs[5] != 0 &&
            steps < 50) {

                for(int x1 = -1; x1 <= 1; x1++)
                    for(int x2 = -1; x2 <= 1; x2++)
                        for(int x3 = -1; x3 <= 1; x3++)
                            for(int y1 = -1; y1 <= 1; y1++)
                                for(int y2 = -1; y2 <= 1; y2++)
                                    for(int y3 = -1; y3 <= 1; y3++) {
                                        stepCoords[0] = startCoords[0] + x1;
                                        stepCoords[1] = startCoords[1] + y1;
                                        stepCoords[2] = startCoords[2] + x2;
                                        stepCoords[3] = startCoords[3] + y2;
                                        stepCoords[4] = startCoords[4] + x3;
                                        stepCoords[5] = startCoords[5] + y3;
                                        double f = 
prob.evaluateInstanceOfIndividual(ind);

                                        if(f > bestFitness) {
                                            bestFitness = f;
                                            stepDirs[0] = x1;
                                            stepDirs[1] = y1;
                                            stepDirs[2] = x2;
                                            stepDirs[3] = y2;
                                            stepDirs[4] = x3;
                                            stepDirs[5] = y3;
                                        }
                                    }

                for(int i = 0; i < 6; i++)
                    startCoords[i] += stepDirs[i];
        }
    }

}
```