

MONDAY, FEBRUARY 23, 2009

## Introduction to Genetic Algorithms with JGAP

Out of interest I am familiarizing myself in genetic algorithms, in short GA. My interest in GA came when I first heard about the JGAP project. As mentioned on the project's site "JGAP (pronounced "jay-gap") is a Genetic Algorithms and Genetic Programming component provided as a Java framework.". For a newcomer I found it difficult to get a good overview about all the concepts introduced in genetic algorithms. Before diving into JGAP, I think it is essential that these concepts are well understood. This post is an introduction to genetic algorithms (GA) with JGAP and is explained with a concrete example. In one of my next posts I will demonstrate solving a problem with genetic programming (GP).

So what is a genetic algorithm? Given is the following definition from John R. Koza:

*The genetic algorithm is a **probabilistic search algorithm** that iteratively transforms a set (called a **population**) of mathematical objects (typically fixed-length binary character strings), each with an associated **fitness value**, into a new population of offspring objects using the Darwinian principle of **natural selection** and using operations that are patterned after naturally occurring **genetic operations**, such as **crossover** (sexual recombination) and **mutation**.*

In genetic algorithms, a **potential solution** is called a **chromosome**. A chromosome consists of a fixed length of **genes**. A gene is a distinct component of a potential solution. During the **evolution** of the genetic algorithm, multiple solutions (chromosomes) are combined (**crossover** and **mutation**) to form, potentially, better solutions. The evolution is done over a population of solutions. The population of solutions is called a **genotype** and consists of a fixed-length of chromosomes. During each evolution, **natural selection** is applied to determine which solutions (chromosomes) make it to the next evolution. The input criteria for the selection process is the so-called **fitness** of a potential solution. Solutions with a better fitness value are more likely to appear in the next evolution than solutions with a worse fitness value. The fitness value of a potential solution is determined by a **user-supplied fitness function**.

Although it is possible to implement the above concepts yourself, JGAP already took care of this. Because the best way to learn is by example, let me first introduce the problem domain which I am going to solve with genetic algorithms. During the example, the concepts mentioned above are further clarified.

Consider a moving company which is specialized in moving boxes (with things in it) from one location to another. These boxes have varying volumes. The boxes are put in vans in which the boxes are moved from location to location. To reduce transport costs, it is crucial for the moving company to use as minimal vans as possible.

**Problem statement:** given a number of boxes of varying volumes, what is the optimal arrangement of the boxes so that a minimal number of vans is needed? The following example shows how to solve this problem with genetic algorithms and JGAP.

First: with the arrangement of the boxes I mean the following: consider 5 boxes with the following volumes (in cubic meters): 1,4,2,2 and 2 and vans with a capacity of 4 cubic meters. When the boxes are put in the vans based on the initial arrangement, the distribution of the boxes in the vans is like this:

Van	Boxes	Space wasted
Van 1	Box 1	3
Van 2	Box 4	0
Van 3	Box 2, Box 2	0
Van 4	Box 2	2

Fitness value =  $3+2 * 4 = 20$ . See section [fitness function](#) for an explanation of the fitness function for this particular problem.

A total of 4 vans is needed. But when the number of vans needed is calculated, which is the total volume of the boxes divided by the volume of the vans, the optimal number of vans is:  $11 / 4 = 2.75$ . Because no partial vans can be used the optimal number of vans needed is 3. The optimal arrangement of the boxes is the following: 1,2,2,2,4. Based on this arrangement the distribution looks like this:

Van	Boxes	Space wasted
Van 1	Box 1, Box 2	1
Van 2	Box 2, Box 2	0
Van 3	Box 4	0

Fitness value of  $1 * 3 = 3$ .

Before implementing the actual solution, the following *preparatory steps* must be taken. These preparatory steps are always needed if genetic algorithms is used to solve a particular problem.

1. Define the *genetical representation* of the problem domain. The boxes which must be put in the vans are represented by an array of Box instances. The genetic algorithm must find the optimal arrangement in the array as how to put the boxes in the vans. A chromosome is a potential solution and consists of a fixed-length of genes. A potential solution in this example consists of a list of indexes where each index represents a Box in the box array. To represent such index, I use an IntegerGene. As mention earlier, a gene is a distinct part of the solution. In this example, a solution (chromosome) consists of as many genes as there are boxes. The genes must be ordered by the genetic program in such a way that it represents a (near) optimal arrangement. For example: if there are 50 boxes, a chromosome with 50 IntegerGene's is constructed, where each gene's value is initialized to an index in the box array, in this case from 0 to 49.
2. Determine the *fitness function*. The fitness function determines how good a potential solution is compared to other solutions. In this problem domain, a solution is fitter when fewer vans are needed so less space is wasted.
3. Determine the *parameters* used for the run. For the run I use a population size of 50 and a total number of 5000 evolutions. So the genotype (the population) initially consists of 50 chromosomes (potential solutions). These values are chosen based on some experimentation and can vary based on the specific problem.
4. Determine the *termination criteria*. The program ends when 5000 evolutions are reached or when the optimal number of vans needed is reached. The optimal number of vans can be calculated by dividing the total volume of the

boxes by the capacity of the vans and rounding the result up (because no partial vans can be used).

### Initialization

The Box class has a volume. In this example 125 boxes are created with varying volumes between 0.25 and 3.00 cubic meters. The boxes are stored in an array. The following code creates the boxes:

```
Random r = new Random(seed);
this.boxes = new Box[125];
for (int i = 0; i < 125; i++) {
    Box box = new Box(0.25 + (r.nextDouble() * 2.75));
    box.setId(i);
    this.boxes[i] = box;
}
```

Before we configure JGAP we must first implement a fitness function. The fitness function is the most important part in GA as it determines which populations potentially make it to the next evolution. The fitness function for this problem looks like this:

```
package nl.jamiecraane.mover;

import org.jgap.FitnessFunction;
import org.jgap.IChromosome;

/**
 * Fitness function for the Mover example. See this
 * {@link #evaluate(IChromosome)} for the actual fitness function.
 */
public class MoverFitnessFunction extends FitnessFunction {
    private Box[] boxes;
    private double vanCapacity;

    public void setVanCapacity(double vanCapacity) {
        this.vanCapacity = vanCapacity;
    }

    public void setBoxes(Box[] boxes) {
        this.boxes = boxes;
    }

    /**
     * Fitness function. A lower value means the difference between
     * the total volume of boxes in a van is small, which is better. This means
     * a more optimal distribution of boxes in the vans. The number of vans
     * needed is multiplied by the size difference as more vans are more expensive.
     */
    @Override
    protected double evaluate(IChromosome a_subject) {
        double wastedVolume = 0.0D;

        double sizeInVan = 0.0D;
        int numberOfVansNeeded = 1;
        for (int i = 0; i < boxes.length; i++) {
            int index = (Integer) a_subject.getGene(i).getAllele();
            if ((sizeInVan + this.boxes[index].getVolume()) <=
```

```

vanCapacity) {
    sizeInVan += this.boxes[index].getVolume();
    } else {
        // Compute the difference
        numberOfVansNeeded++;
        wastedVolume += Math.abs(vanCapacity -
sizeInVan);
        // Make sure we put the box which did not fit in
this van in the next van
        sizeInVan = this.boxes[index].getVolume();
    }
}
// Take into account the number of vans needed. More vans
produce a higher fitness value.
return wastedVolume * numberOfVansNeeded;
}
}

```

The above fitness function loops through all the genes in the supplied potential solution (where each gene in the chromosome represents an index in the box array) and calculates how many vans are needed for this arrangement of boxes to fit in the vans. The fitness value is based on the space wasted in every van when a new van is needed, called the wasted volume. The total volume wasted is multiplied by the number of vans needed. This is done to create a much worse fitness value when more vans are needed. In the above, simplified, example the fitness value of the first solution is 20 and the fitness value of the second, optimal, solution is 3. One term deserves more explanation and that is allele. In the above code the `getAllele` method on the gene is called. Allele is just another word for the value of the gene. Because all genes are `IntegerGene`'s, the value of each gene is of type `Integer`.

Next it is time to setup JGAP:

```

private Genotype configureJGAP() throws InvalidConfigurationException {
    Configuration gaConf = new DefaultConfiguration();
    // Here we specify a fitness evaluator where lower values means a better
fitness
    Configuration.resetProperty(Configuration.PROPERTY_FITEVAL_INST);
    gaConf.setFitnessEvaluator(new DeltaFitnessEvaluator());

    // Only use the swapping operator. Other operations makes no sense here
// and the size of the chromosome must remain constant
    gaConf.getGeneticOperators().clear();
    SwappingMutationOperator swapper = new SwappingMutationOperator(gaConf);
    gaConf.addGeneticOperator(swapper);

    // We are only interested in the most fittest individual
    gaConf.setPreservFittestIndividual(true);
    gaConf.setKeepPopulationSizeConstant(false);

    gaConf.setPopulationSize(50);
    // The number of chromosomes is the number of boxes we have. Every
chromosome represents one box.
    int chromeSize = this.boxes.length;
    Genotype genotype;

    // Setup the structure with which to evolve the solution of the problem.
// An IntegerGene is used. This gene represents the index of a box in the
boxes array.
    IChromosome sampleChromosome = new Chromosome(gaConf, new

```

```

IntegerGene(gaConf), chromeSize);
gaConf.setSampleChromosome(sampleChromosome);
// Setup the fitness function
MoverFitnessFunction fitnessFunction = new MoverFitnessFunction();
fitnessFunction.setBoxes(this.boxes);
fitnessFunction.setVanCapacity(VOLUME_OF_VANS);
gaConf.setFitnessFunction(fitnessFunction);

// Because the IntegerGenes are initialized randomly, it is necessary to
set the values to the index. Values range from 0..boxes.length
genotype = Genotype.randomInitialGenotype(gaConf);
List chromosomes = genotype.getPopulation().getChromosomes();
for (Object chromosome : chromosomes) {
    IChromosome chrom = (IChromosome) chromosome;
    for (int j = 0; j < chrom.size(); j++) {
        Gene gene = chrom.getGene(j);
        gene.setAllele(j);
    }
}

return genotype;
}

```

In the above code we setup the JGAP library. The provided Javadoc should be self-explanatory. A population (genotype) of 50 potential solutions (chromosomes) is created where every chromosome consists of the same number of genes as there are boxes. Because in this example a lower fitness value is better, the `DeltaFitnessEvaluator` is used.

Next, it is time to evolve the population. The population is evolved 5000 times. When the optimal amount of vans is reached earlier, the run ends. The following code demonstrates the evolution of the problem solution:

```

private void evolve(Genotype a_genotype) {
int optimalNumberOfVans = (int) Math.ceil(this.totalVolumeOfBoxes /
VOLUME_OF_VANS);
LOG.info("The optimal number of vans needed is [" + optimalNumberOfVans +
"]");

double previousFittest =
a_genotype.getFittestChromosome().getFitnessValue();
numberOfVansNeeded = Integer.MAX_VALUE;
for (int i = 0; i < NUMBER_OF_EVOLUTIONS; i++) {
    if (i % 250 == 0) {
        LOG.info("Number of evolutions [" + i + "]");
    }
    a_genotype.evolve();
    double fitness = a_genotype.getFittestChromosome().getFitnessValue();
    int vansNeeded =
this.numberOfVansNeeded(a_genotype.getFittestChromosome().getGenes().size(
));
    if (fitness < previousFittest && vansNeeded < numberOfVansNeeded) {
        this.printSolution(a_genotype.getFittestChromosome());
        previousFittest = fitness;
        numberOfVansNeeded = vansNeeded;
    }

// No more optimal solutions
if (numberOfVansNeeded == optimalNumberOfVans) {

```

```

        break;
    }
}
IChromosome fittest = a_genotype.getFittestChromosome();

List<Van> vans = numberOfVansNeeded(fittest.getGenes());
printVans(vans);
this.printSolution(fittest);
}

```

Because we set the `preserveFittest` property on the `JGAP` configuration object to `true`, we have access to the most fittest chromosome with the `getFittestChromosome()` method. The fittest chromosome consists of 125 genes, the indexes of the boxes in the array, in the arrangement of how to put the boxes in the vans. The actual evolution is performed by `JGAP`. The fitness value determines which populations have the highest chance to make it to the next evolution. Eventually a (near) optimal solution is formed. This indicates the importance of a well chosen fitness function as it is used in the selection process of the chromosomes. Below is the output of a sample run:

```

The total volume of the [125] boxes is [210.25989987666645] cubic metres.
The optimal number of vans needed is [49]
Number of evolutions [0]
Fitness value [4123.992085987977]
The total number of vans needed is [63]
Fitness value [3458.197333300851]
The total number of vans needed is [61]
Fitness value [3138.2899569572887]
The total number of vans needed is [60]
Fitness value [2865.5105375433063]
The total number of vans needed is [59]
Fitness value [2562.282028584251]
The total number of vans needed is [58]
Fitness value [2267.7135196251966]
The total number of vans needed is [57]
Fitness value [1981.8050106661412]
The total number of vans needed is [56]
Fitness value [1704.5565017070858]
The total number of vans needed is [55]
Fitness value [1479.769464870246]
The total number of vans needed is [54]
Number of evolutions [250]
Fitness value [1215.9278601031112]
The total number of vans needed is [53]
Fitness value [1002.6487336510297]
The total number of vans needed is [52]
Number of evolutions [500]
Fitness value [774.5352329142294]
The total number of vans needed is [51]
Number of evolutions [750]
Number of evolutions [1000]
Fitness value [535.1696373214758]
The total number of vans needed is [50]
Number of evolutions [1250]
Number of evolutions [1500]
Number of evolutions [1750]
Number of evolutions [2000]
Number of evolutions [2250]
Fitness value [307.8713731063958]
The total number of vans needed is [49]
Van [1] has contents with a total volume of [4.204196540671411] and

```

```
contains the following boxes:
Box:0, volume [2.2510465109421443] cubic metres.
Box:117, volume [1.9531500297292665] cubic metres.
Van [2] has contents with a total volume of [4.185233471369987] and
contains the following boxes:
Box:17, volume [1.0595047801111055] cubic metres.
Box:110, volume [0.5031165156303853] cubic metres.
Box:26, volume [2.6226121756284955] cubic metres.
Van [3] has contents with a total volume of [4.312990612147265] and
contains the following boxes:
Box:91, volume [1.8897555340430203] cubic metres.
Box:6, volume [2.423235078104245] cubic metres.
...Further output omitted
```

As seen in the above output, the optimal number of vans is reached between 2250 and 2500 evolutions. The program also outputs the distribution of the boxes in the vans. The complete source code of the above example can be downloaded from <http://code.google.com/p/jc-examples/>. The down-loadable artefact is called `ga-moving-example-1.0.jar`.

## Conclusion

GA is an exciting technology but with a lot of concepts which are hard to grasp if you are new to the field. This post is an introduction of the concepts for genetic algorithms and showed how to implement a GA solution with JGAP. In one of my next posts, genetic programming is explained with a concrete example.

The silver bullet rule applies to genetic algorithms as well. To give you an impression, the following problems are good candidates to solve with GA:

- Problems where it is hard to find a solution but once a solution is found, measure how good this particular solution is.
- Problems where the search space is very large, complex or poorly understood.
- Problems where a near optimal solution is acceptable.
- Problems where no mathematical analysis is available.

Although I am not an expert on the subject, feel free to ask your questions and I will try to answer them as best as possible.

## Resources

[1] [Field guide to genetic programming](#)

[2] [JGAP](#)

[3] [http://en.wikipedia.org/wiki/Genetic\\_algorithm](http://en.wikipedia.org/wiki/Genetic_algorithm)

[5] <http://www.genetic-programming.org/>

[6] <http://code.google.com/p/jc-examples/>.

[7] [http://en.wikipedia.org/wiki/Knapsack\\_problem](http://en.wikipedia.org/wiki/Knapsack_problem).

[8] [http://en.wikipedia.org/wiki/Bin\\_packing\\_problem](http://en.wikipedia.org/wiki/Bin_packing_problem).