Solving the partition problem using a genetic algorithm

Matthijs Dorst, #1380982, m.c.dorst@ai.rug.nl

Summary

Certain computational tasks require a great amount of processing time to complete using brute-force methods – i.e., computing all possible combinations and sifting through these to find the optimal solution.

A method is reviewed where genetic algorithms are used to incrementally approach a possible solution to a classic problem: given an array of numbers, find a partitioning such that two heaps of equal size emerge. Using brute-force techniques, such a problem would mean computing the sizes of each heap for 2^N possible combinations. While genetic algorithms may not always be able to ascertain the best solution has been found, they can generally find a good solution relatively quickly.

In the implementation of this algorithm several evolutionary techniques are tested with various settings. Once optimal settings have been found, the algorithm is then used to compute an optimal partitioning for 5.000 random numbers between 0 and 100, in less than two seconds on a normal pc.

1. Introduction

Genetic algorithms are based on the same principles that govern biological evolution: a population of individuals adept and evolve to increase their overall fitness. Though the solution is sometimes imperfect (take for example the blind-spot in the human eye), it is usually very good – especially considering the almost infinite amount of possible combinations.

In software this can be emulated by randomly creating a population and then letting it evolve by emulating established evolutionary processes such as mutation and procreation. While each generation will not necessarily be better than the last, it is possible to steer the evolution in such a way that, given enough generations, increasingly fitter individuals are created. Once this increase in fitness ceases the best individual can be chosen as an answer to the problem.

In this paper a genetic algorithm is discussed that has as goal to solve the classic partition problem: consider a stack of numbers and divide them in two groups, such that both groups are of equal size. While it is easy to compute all possible combinations, the amount of possibilities grows exponentially with each additional number to such extend that brute-force techniques do no longer suffice.

2. Implementation of the Genetic Algorithm

The algorithm used is written in JAVA. A complete listing of the source code is given later on in appendix A. Globally, the program is designed as follows:

2.1 Control Classes

A launcher class is called first upon execution. It keeps track of the program's runtime for statistical analysis and is able to catch all thrown exceptions and keep track of the algorithms return code. In this way valuable information about the algorithms global performance is obtained.

Called by the launcher is a general algorithm handler. Its goal is to retrieve source data (a stack of randomly generated numbers), create a new population and let it evolve till either a solution has been found or evolution stagnates. Once a solution is found it displays the result and stops.

2.2 Population Object

The population object handles the actual population: an array of individuals, each determined in turn by an array of Boolean values indicating specific gene values. It can create a new population of requested size in which each individual has randomly assigned gene values. Evolution methods can then be called to work on the population and evolve it.

To determine the fitness of specific individuals and a generation as a whole methods are available to calculate the difference between the heaps as determined by an individual's genotype. After each evolutionary phase all individuals are rated thus and compared to the best known overall individual to see if a better individual has been found. Finally, the new best individual's rating is compared to a predetermined value to check if our evolution has progressed sufficiently to present it as the final solution.

2.3 Evolutionary Processes

Though there are many known evolutionary processes, in this implementation only the basic three are used: survival of the fittest (otherwise known as selection), random mutation and cross-over. For each process a child class is created to modify the population's individuals array.

2.3.1 Survival of the Fittest

The process in which two individuals are tested against each other and only the strongest remains – alternatively called selection sometimes. In this implementation a relatively simple selection method is used: two random individuals are chosen and the genotype of the less fit individual is overwritten by that of the fitter individual. A more advanced implementation could for example base the survival chance of each individual on its rating, given fitter individuals a higher chance to survive. This is perhaps worth to investigate further at a later time.

2.3.2 Random Mutation

There is a base chance for each gene to mutate. Since genes can only have two values, true or false, mutation means the gene flips to the opposite value: false if the gene was true and vice versa. Increasing this chance means more dramatic population changes, while a lower mutation chance results in a more steady evolution. Finding an optimal setting for this value is discussed later on.

2.3.3 Cross-over

It is possible for two individuals to combine, choosing random genes from another parent individual to create a new child individual. In order to maintain a constant population size this new individual then replaces its parent. The amount of genes chosen from another individual, or the crossover-chance, is also determined by our general algorithm object and can be varied for optimal result.

3. Preliminary results

Using untested values for most settings average runtimes can be found for different population sizes.

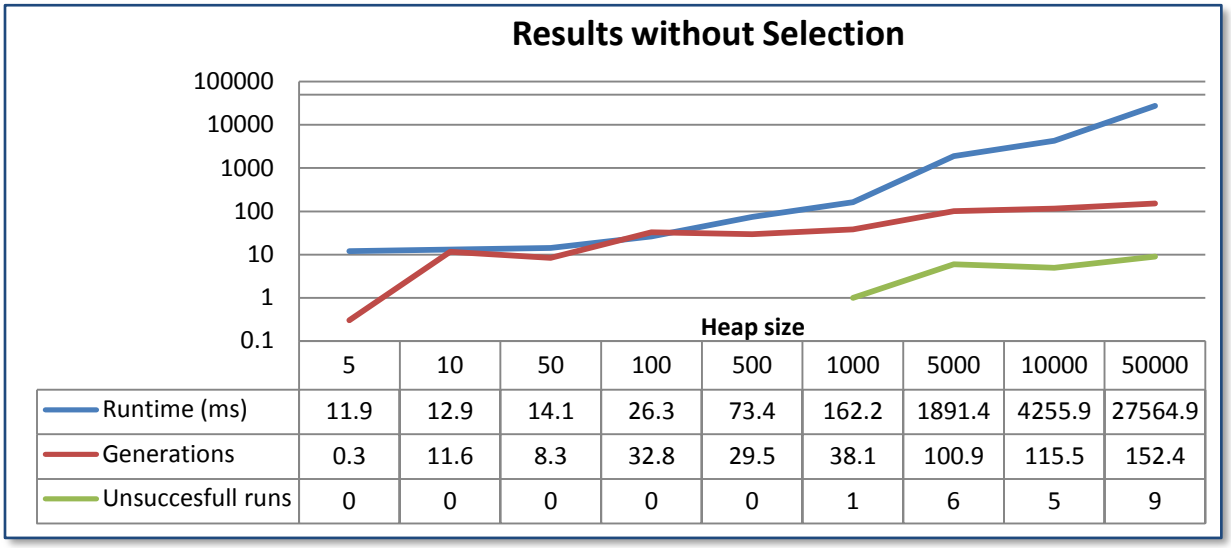
3.1 Test settings

The settings as used for the first set of trials:

Setting	Value	Description
Population size	30	Amount of individuals in each generation.
Mutation chance	5%	Chance a gene will randomly mutate.
Crossover chance	10%	Chance a gene will be taken from another individual.
Stagnation limit	100	If for this amount of generations no better individual has been found, the algorithm stops looking.

Using these settings the algorithm is used to solve the problem for a constant stack of numbers of increasing size. For each stack size the algorithm is ran 10 times to determine the average amount of required generations, the amount of imperfect solutions and the average execution time. To test the effect selection has on the results all tests are run two times: with and without selection.

3.2 Results without fitness selection

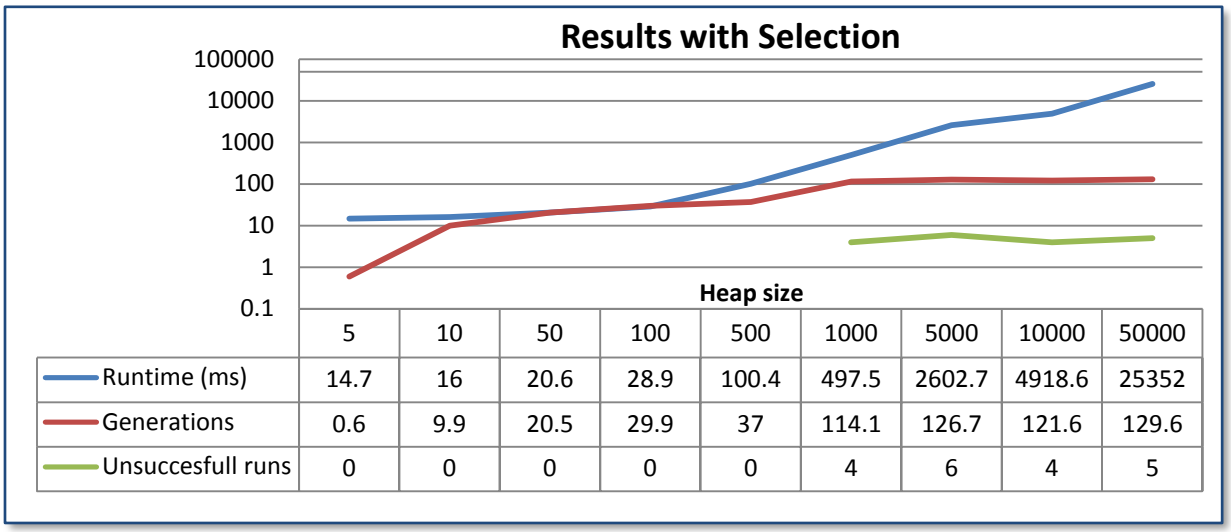


	5	10	50	100	500	1000	5000	10000	50000
Runtime (ms)	11.9	12.9	14.1	26.3	73.4	162.2	1891.4	4255.9	27564.9
Generations	0.3	11.6	8.3	32.8	29.5	38.1	100.9	115.5	152.4
Unsuccesfull runs	0	0	0	0	0	1	6	5	9

As can readily be seen the amount of source data is directly related to the time required for the algorithm to run. While the amount of generations shows a logarithmic growth, runtime seems to increase exponentially with increasing data size.

3.3 Results with fitness selection

With fitness selection active we see a similar result, yet with subtle differences:



	5	10	50	100	500	1000	5000	10000	50000
Runtime (ms)	14.7	16	20.6	28.9	100.4	497.5	2602.7	4918.6	25352
Generations	0.6	9.9	20.5	29.9	37	114.1	126.7	121.6	129.6
Unsuccesfull runs	0	0	0	0	0	4	6	4	5

3.4 Discussing results

While the algorithm without fitness selection on average finds a solution almost as often as when survival of the fittest selection is applied, the amount of generations required to find that solution keeps increasing with data size. When fitness selection is applied the amount of required generations seems to level off at about ~130 generations, independent of data size. This seems to indicate that for sufficiently small data sizes random mutations are more likely to produce an improved individual, yet once the algorithm is allowed to run for sufficient amount of time the selection procedure can help in finding a solution more quickly and more often.

4. Finding optimal constant values

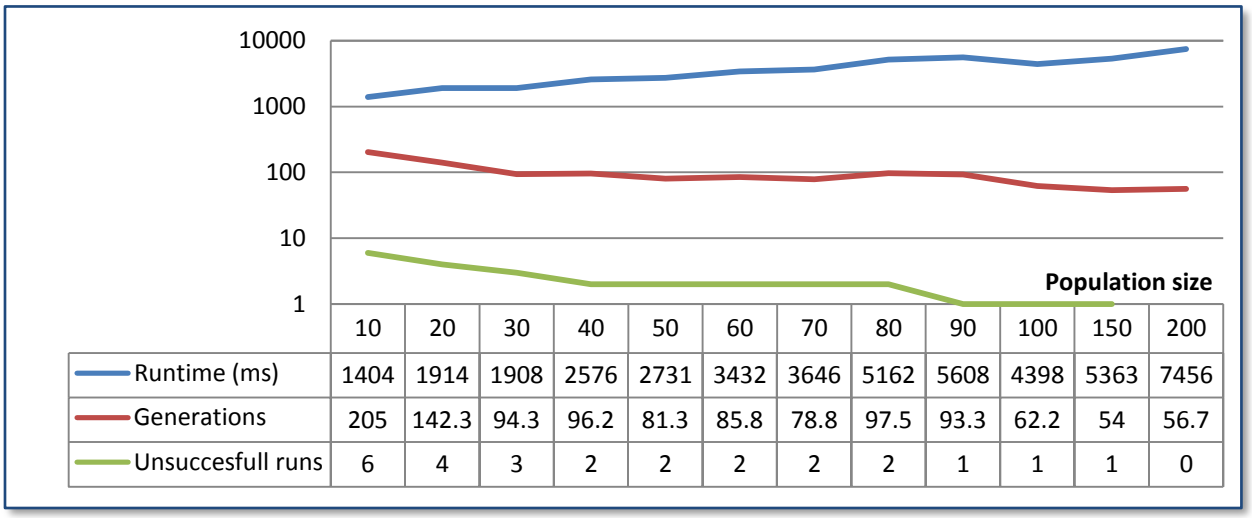
Using the original heap of source data the used settings can be altered to check for more optimal values. By altering each setting individually for a fixed data size information can be obtained to increase the algorithms performance.

4.1 Population size optimization

Using fixed values for all settings but population size, performance is measured. Values for the fixed settings are as follows:

Setting	Value	Description
Source data size	5000	Amount of numbers to sort.
Mutation chance	5%	Chance a gene will randomly mutate.
Crossover chance	10%	Chance a gene will be taken from another individual.
Stagnation limit	100	If for this amount of generations no better individual has been found, the algorithm stops looking.

Test results are the average of 6 trials for each setting. Runtime, number of generations required and number of unsuccessful runs plotted against population size:



	10	20	30	40	50	60	70	80	90	100	150	200
Runtime (ms)	1404	1914	1908	2576	2731	3432	3646	5162	5608	4398	5363	7456
Generations	205	142.3	94.3	96.2	81.3	85.8	78.8	97.5	93.3	62.2	54	56.7
Unsuccesfull runs	6	4	3	2	2	2	2	2	1	1	1	0

As can be expected the runtime increases with incremental population size. Likewise, the amount of required generations decreases with increasing population size, as does the amount of failed attempts.

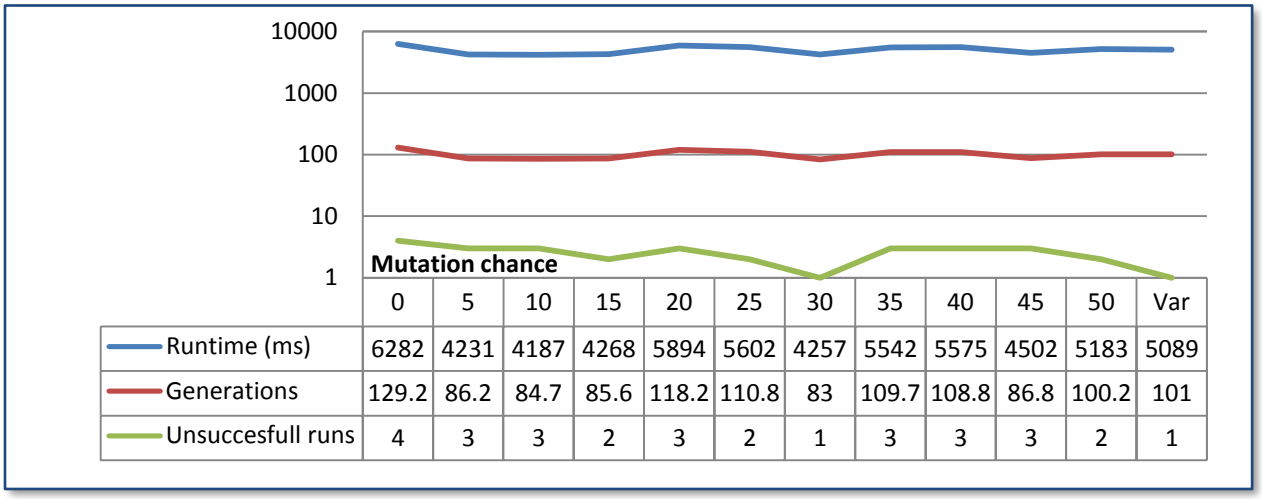
Based on this data, an optimal population size cannot be established: it depends on how accurate the algorithm should be as well as how fast it should be. In general a population size of 75, or 1.5% of the amount of source data, seems to be both reasonably fast as well as accurate enough for most purposes.

4.2 Mutation chance optimization

Using the optimized value for population size found in 4.1, the chance of random mutations can be varied to find an optimal setting. Independent variables are consequentially set thus:

Setting	Value	Description
Source data size	5000	Amount of numbers to sort.
Population size	75	Amount of individuals in each generation.
Crossover chance	10%	Chance a gene will be taken from another individual.
Stagnation limit	100	If for this amount of generations no better individual has been found, the algorithm stops looking.

Test results are again the average of 6 trials for each setting. Runtime, number of generations required and number of unsuccessful runs plotted against mutation chance:



Mutation chance	0	5	10	15	20	25	30	35	40	45	50	Var
Runtime (ms)	6282	4231	4187	4268	5894	5602	4257	5542	5575	4502	5183	5089
Generations	129.2	86.2	84.7	85.6	118.2	110.8	83	109.7	108.8	86.8	100.2	101
Unsuccesfull runs	4	3	3	2	3	2	1	3	3	3	2	1

As was to be expected, there is a direct correlation between number of generations required and runtime when only the mutation chance is varied. The last entry, var, denotes a variable mutation chance: mutation is set inversely to the amount of stagnation detected, so that when stagnation reaches a high level, mutation chance increases also. While this too has little effect on runtime and required generations, it does seem to increase the chance a successful solution is found.

Another noteworthy point is the case with 0% mutation chance – as might be expected there are many unsuccessful attempts, increasing the average runtime and number of required generations.

There also seems to be a dip in runtime and failed attempts at around 30% mutation chance. An obvious explanation cannot be given for this, yet repeated tests confirm this to be an optimum mutation chance.

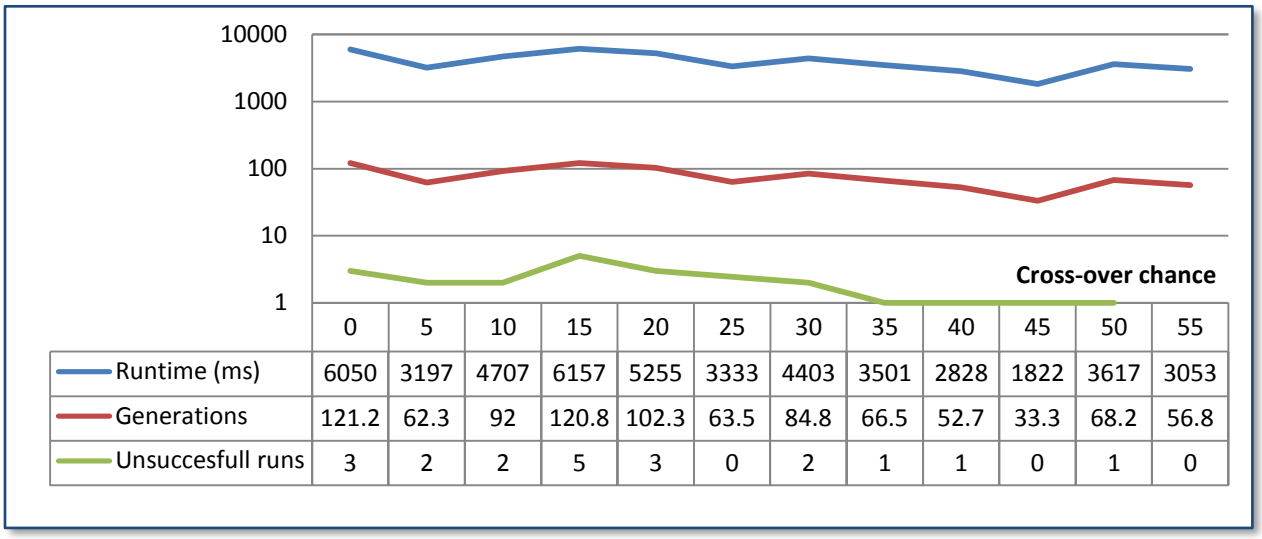
4.3 Cross-over chance optimization

The final setting to be optimized is the chance a gene is mutated from one individual to the next. A higher cross-over chance means a population will produce a wider range of offspring, yet it also means any high scoring individuals are lost when they combine with a low-scoring neighbor.

As in 4.1 and 4.2, the other variables are set constant:

Setting	Value	Description
Source data size	5000	Amount of numbers to sort.
Population size	75	Amount of individuals in each generation.
Mutation chance	30%	Chance a gene will randomly mutate.
Stagnation limit	100	If for this amount of generations no better individual has been found, the algorithm stops looking.

Test results are once more the average of 6 trials for each setting. Runtime, number of generations required and number of unsuccessful runs plotted against cross-over chance:



	0	5	10	15	20	25	30	35	40	45	50	55
Runtime (ms)	6050	3197	4707	6157	5255	3333	4403	3501	2828	1822	3617	3053
Generations	121.2	62.3	92	120.8	102.3	63.5	84.8	66.5	52.7	33.3	68.2	56.8
Unsuccesfull runs	3	2	2	5	3	0	2	1	1	0	1	0

As with varying mutation chance, the runtime depends only on the amount of generations, not on cross-over chance. This is logical since no additional calculations are made for different cross-over chance values.

Also, as with varying mutation chance, there is only a suggestion of a link between the dependent and independent variables. However, there is an indication an optimum can be found where cross-over chance results in an equal distribution of genes from both parents. Inexplicable though this may be, it is not without biological plausibility.

4.4 Optimal algorithm settings

What has so far not been discussed is the possibility that various constants are dependent upon others. Thus, it could be that while an optimum setting has been found for cross-over chance, using this setting could mean the originally assumed optimum for population size has become invalid. Interesting though this problem is, it is beyond the scope of this article to solve. Ironically, a fairly obvious way to solve this problem is either by brute force or using yet another genetic algorithm. For now it is assumed that our algorithm has been calibrated adequately and valid conclusions may now by drawn from its operation.

5. Conclusions

Consider once again the first test results. For a stack of 5.000 numbers the algorithm required 1891.4 milliseconds to run, 100.9 generations and failed to find the optimum solution in 60% of the tests. With calibration and optimization it now requires 1822 milliseconds to run yet taking only 33.3 generations and having a success rate of 100%.

What this means is that using a genetic algorithm, a problem with 2^{5000} possible solutions can now be solved in less than two seconds by a normal pc. Of course in these tests there has always been an optimum solution. When there definitely is no solution the algorithm will still keep trying to find one for many generations, making it far less efficient. Likewise, once such an imperfect solution has been found there is no way to determine whether it actually is the very best possible solution – just that is most likely quite close to that best possible one, if not it.

Thus, while the genetic algorithm as proposed here has amazing potential to solve extremely complex problems incredibly fast, its credibility is somewhat limited. This can be optimized by increasing population size and stagnation threshold at the expense of performance, yet only within limits. An infinite threshold would mean the final solution would be infinitely close to optimal but in imperfect scenarios it would also take an infinite amount of time to obtain.

In this paper settings are proposed that will often give an accurate result, yet will not take unproportionally long to run. It is up to the reader to accept these values or modify them to fit different situations.

Appendix A

Source code. In order:

algorithm package:

evolver package:

```java
package algorithm;

import  java.util.*;
import  java.io.*;

/**
 *  Our main genetic algorithm implementation. This
 *  class acts as a handler, not as such implementing
 *  many methods itself but calling them in separate objects
 *  designed for specific purposes.
 *
 *  Result of all this is that we can load a stack of
 *  numbers, use that as source data for a population of
 *  individuals capable of evolving towards a sufficiently
 *  close approximation to the requested solution, or
 *  alternatively an optimal solution when our evolution stagnates.
 *
 *  Or, to put it more simplistically: it'll run till we either find
 *  an answer or realize there isn't one. ;)
 *
 *  @author Matthijs
 */
public class GeneticAlgorithm implements Handler {

    public  int[]   heap;

    public  int     populationSize  = 75,
                    mutationChance  = 30,
                    crossoverChance = 55,
                    heapSize        = 5000,
                    previousRating  = 0,
                    stagnationCount = 0;


    /**
     *  Run our algorithm: we create a new population, populate it with random
     *  individuals (read in from file), then let it evolve for as long as we
     *  need it to - ie, till we either find an optimal solution or our
     *  evolution stagnates.
     *
     *  To detect the latter we keep a stack of the latest N individuals and
     *  compute each generation wether there has been any progress at all in
     *  that stack.
     *
     *  Once this is no longer the case or once we find an optimal solution
     *  (a difference of 1 or 0) we display the heap sizes we have obtained.
     */
    public int run () throws Exception {
        readHeap();
        Population mainGroup =  new Population(this);
        mainGroup.populate();
        int i = 0;

        do {
            System.out.print("Generation " + i++ + ":   ");
            mainGroup.evolve();
            mainGroup.rateIndividuals();
        } while (!mainGroup.isSufficient() && !isStagnant(mainGroup.highestRating));

        mainGroup.showBestIndividual();
        return 1;
    }


    /**
     *  We must know whether or not our evolution has stagnated. To that
     *  end we keep track of the best individual in each population so
     *  when they do not evolve anymore for several generations we can
     *  consider them the end result.
     *
     *  @param   rating      The highest rating in the current generation.
```

```
     *  @return  Either true (population IS stagnant and we should abort) or
     *           false (fittest is still increasing).
     */
    private boolean isStagnant (int rating) {
        if (rating == previousRating)
            stagnationCount++;
        else
            stagnationCount = 0;

        previousRating = rating;
        if (stagnationCount > 100)
            return true;
        return false;
    }


    /**
     *  Acquire a heap of numbers to be stacked. A file with 50.000 randomly
     *  generated numbers is available, out.txt, so we read in as many
     *  numbers as we need (ie, is set in this.heapSize).
     *
     *  Doing so means we always have the same source data for our algorithm
     *  so comparative analysis can be performed on its execution.
     *
     *  Note that this throws an IOException if an error occurs while reading
     *  the specified file.
     */
    private void readHeap () throws IOException {
        heap                    = new int[heapSize];
        File file               = new File("out.txt");
        BufferedReader buffer   = new BufferedReader(new FileReader(file));
        String line             = null;
        int pos                 = 0;

        while((line = buffer.readLine()) != null && pos < heapSize) {
            StringTokenizer tokens = new StringTokenizer(line);
            while (tokens.hasMoreTokens() && pos < heapSize)
                heap[pos++] = Integer.valueOf(tokens.nextToken());
        }
        buffer.close();
    }
}
```

```java
package algorithm;

/**
 * Provide an interface for all Handlers - these
 * are public classes that are designed to handle
 * a specific aspect of  our program.
 *
 * @author Matthijs
 */
public interface Handler {

    /**
     * A run method should, as the name implies, run whatever
     * algorithm a Handler is written to handle. As a main, it
     * should return an int indicating the result of that execution.
     *
     * @return  The status of the handled algorithm when it is complete.
    **/
    public abstract int run () throws Exception;
}
```

```java
package algorithm;

/**
 *  Launcher - this class is designed specifically to
 *  initiate the creation of actual handler classes
 *  that perform our code. It provides a save environment
 *  for our Handler so we can gracefully handle exceptions
 *  and relay status information to the user.
 *
 *  @author Matthijs
**/
public class Launcher {

    /**
     * Main function, called upon execution and handles the creation
     * of a new Handler object that proceeds to actually execute
     * our code.
     *
     * @param   args    Variable length argument that contains commandline
     *                  parameters.
     * @return  Void.
    **/
    public static void main(String[] args) {
        int     result          =   0;
        long    time            =   System.currentTimeMillis();

        try {
            Handler algorithm   =   new GeneticAlgorithm();
            result              =   algorithm.run();
        }
        catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
        time                    =   System.currentTimeMillis() - time;
        if (result == 0)
            System.out.println("Program failled to run succesfully.");
        else
            System.out.println("Program terminated succesfully.");
        System.out.println("Runtime: " + time + "ms");
    }
}
```

```java
package algorithm;

/**
 *  Population class - this basically governs the main
 *  affairs of all individuals in our population.
 *
 *  Upon creation a population object will create a new population
 *  consisting of random individuals. The size of each individual is
 *  determined by the parent class, as are other parameters like
 *  mutation chance and population size.
 *
 *  The parent class may then proceed to let the population evolve
 *  using the like named method and determine trough the highestRating
 *  parameter whether or not the population has evolved enough. A method
 *  to assist in this is isSufficient, which evaluates to true when a
 *  sufficiently close approximation has been found to the desired solution.
 *
 *
 *  @author Matthijs
 */
public class Population {

    private int[]               rating;

    public  GeneticAlgorithm    parent;
    public  int                 highestRating;
    public  boolean[]           bestIndividual;
    public  boolean[][]         individuals;


    /**
     *  Class constructor, assigns the reference to our parent object
     *  and initializes our data arrays with the sizes noted in our parent.
     *
     *  @param parent    GeneticAlgorithm object that holds various data members
     *                   required by a population.
     */
    public Population (GeneticAlgorithm parent) {
        this.parent         =  parent;
        this.individuals    =  new boolean[parent.populationSize][parent.heap.length];
        this.rating         =  new int[individuals.length];
    }


    /**
     *  Populate our population with random individuals. When this is
     *  done we rate the first individual to determine a baseline
     *  against which we can compare other individuals.
     */
    public void populate () {
        for (int i = 0; i < individuals.length; i++)
            individuals[i] = this.createIndividual();
        highestRating   =   rateIndividual(0);
    }


    /**
     *  Rate the 'fitness' of each individual and determine if we
     *  have found a 'fitter' individual than the best determined so
     *  far.
     */
    public void rateIndividuals () {
        for (int i = 0; i < individuals.length; i++) {
            rating[i]       =   this.rateIndividual(i);
            highestRating   =   highestRating >  rating[i] ? rating[i]              :
highestRating;
            bestIndividual  =   highestRating == rating[i] ? individuals[i].clone() :
bestIndividual;
        }
    }
```

```
/**
 *  Force the population (ie, stack of individuals) to evolve
 *  using various methods implemented in child classes. Currently
 *  three methods of evolution are implemented: mutation, crossover
 *  and survival of the fittest.
 *
 *  @throws Exception
 */
public void evolve () throws Exception {
    Handler crossover   = new evolver.Crossover(this);
    crossover.run();

    Handler fittest     = new evolver.Fittest(this);
    fittest.run();

    Handler mutation    = new evolver.Mutation(this);
    mutation.run();
}


/**
 *  Determine whether or not our population has advanced
 *  sufficiently to stop evolving. Typically, this should
 *  only be so when we have reached an optimal solution - which is
 *  either 0 or 1 (since we cannot optimize further once we achieve
 *  a difference in heapsizes of 1).
 *
 *  @return     Either true (population is sufficiently advanced) or
 *              false (it is not so).
 */
public boolean isSufficient () {
    System.out.println(highestRating);
    return highestRating < 2 ? true : false;
}


/**
 *  Determine a rating for a specific individual in the current
 *  population. This is done by calculating the absolute size
 *  difference between the two 'heaps', which is as good a rating
 *  as any.
 *
 *  @param   index   The index of the individual in the current populations
 *                   (in this.individuals) which we want to rate.
 *  @return  Int     The rating index for the specified individual.
 */
public int rateIndividual (int index) {
    int     primary     = 0,
            secondary   = 0;
    for (int i = 0; i < parent.heap.length; i++) {
        if (individuals[index][i] == true)
            primary   += parent.heap[i];
        else
            secondary += parent.heap[i];
    }
    return Math.abs(primary - secondary);
}
```

```
    /**
     *  Output information on the best individual in our population.
     *
     *  Basically, this just shows the sizes of the two heaps so we can
     *  see how big the difference is between them, relative to their
     *  size.
     */
    public void showBestIndividual () {
        int     primary     = 0,
                secondary   = 0;
        for (int i = 0; i < parent.heap.length; i++) {
            if (bestIndividual[i] == true)
                primary     += parent.heap[i];
            else
                secondary   += parent.heap[i];
        }
        System.out.println(primary + " / " + secondary);
    }


    /**
     *  Randomly create a new individual with the correct size.
     *
     *  @return  An array of boolean values the size of one individual.
     */
    private boolean[] createIndividual () {
        boolean[] individual = new boolean[parent.heap.length];
        for (int i = 0; i < individual.length; i++)
            individual[i] = Math.random() > 0.5 ? true : false;
        return individual;
    }
}
```

```java
package evolver;

import  algorithm.Handler;
import  algorithm.Population;


/**
 *  Cross-over evolver.
 *
 *  Implements evolution trough cross-over mutations. This resembles
 *  the biological procreation process somewhat in that genes from
 *  two individuals are combined (randomly) to create a 'new' individual
 *  with aspects of both parents.
 *
 *  Though this class can, on its own, simulate evolution of a population,
 *  it is best combined with other evolutionary processes such as mutation
 *  to create a more plausible similarity with real evolution.
 *
 *  @author Matthijs
 */
public class Crossover implements Handler {

    Population parent;


    /**
     *  Constructor, sets our parent reference.
     *
     *  @param parent       Reference to the parent Population object.
     */
    public Crossover (Population parent) {
        this.parent =   parent;
    }


    /**
     *  Performing method: this loops over all even-numbered individuals
     *  in our population and crosses them with the next individual using
     *  the cross method.
     *
     *  @return             Status integer: should be 1 for successful crossover.
     */
    public int run () throws Exception {
        for (int i = 1; i < parent.individuals.length - 1; i+= 2)
            this.cross(i);
        return 1;
    }


    /**
     *  Cross certain genes of the given individual with the next individual
     *  in our parent individuals array.
     *
     *  Note that the chance individual genes are crossed is determined by our
     *  geneticAlgorithm's crossoverChance parameter.
     *
     *  @param individual   Integer denoting the individual index we wish to cross
     *                      with the next individual.
     */
    private void cross (int individual) {
        for (int i = 0; i < parent.individuals[individual].length; i++)
            if ((Math.random() * 100) < parent.parent.crossoverChance)
                switchGenes(individual, i);
    }
```

```
    /**
     *  Switch the specified gen in the specified individual with that of the next
     *  individual. Basically, this is little more than a swap operation on our parent
     *  individuals array.
     *
     *  @param individual   Integer denoting the individual index we wish to cross
     *                      with the next individual.
     *  @param gen          Integer denoting the gen to swap.
     */
    private void switchGenes (int individual, int gen) {
        boolean original                        = parent.individuals[individual][gen];
        parent.individuals[individual][gen]     = parent.individuals[individual + 1][gen];
        parent.individuals[individual + 1][gen] = original;
    }
}
```

```java
package evolver;

import  algorithm.Handler;
import  algorithm.Population;

/**
 * Survival of the Fittest evolver.
 *
 * This class handles the evolutionary process of survival of
 * the fittest: 'weak' individuals are eliminated in favor of
 * stronger individuals.
 *
 * Note that if this would be the only evolutionary process in
 * the end our population would consist of a single individual.
 * To make it effective, this needs to be combined with other
 * processes such as cross-over or mutation.
 *
 * @author Matthijs
 */
public class Fittest implements Handler {

    private Population      parent;


    /**
     *  Constructor, sets our parent reference.
     *
     *  @param parent       Reference to the parent Population object.
     */
    public Fittest (Population parent) {
        this.parent = parent;
    }


    /**
     *  Performing method: this loops over all even-numbered individuals
     *  in our population and compares them with the next numbered individual
     *  in the parent individuals array. The weakest of the two is then eliminated
     *  and the DNA of the strongest individual is copied over his.
     *
     *  @return             Status integer: should be 1 for successful crossover.
     */
    public int run () throws Exception {
        for (int i = 0; i < parent.individuals.length - 1; i+= 2)
            this.killWeakest(i);

        return 1;
    }


    /**
     * We kill the weakest link - ie, of each pair we copy the DNA of our
     * strongest individual over that of the weaker one.
     *
     * @param index         Integer index denoting the first individual in
     *                      our parents individuals array we must compare and
     *                      compete with the second one.
     */
    private void killWeakest (int index) {
        int primary     =   parent.rateIndividual(index);
        int secondary   =   parent.rateIndividual(index + 1);

        if (primary < secondary)
            parent.individuals[index + 1]    = parent.individuals[index];
        else
            parent.individuals[index]        = parent.individuals[index + 1];
    }
}
```

```java
package evolver;

import  algorithm.Handler;
import  algorithm.Population;

/**
 *  The mutation evolver class handles evolution trough mutation: random
 *  gens are picked out and 'mutated', ie transformed, to create new
 *  individuals. This means that very well scoring individuals can degenerate
 *  and bad individuals evolve. Mutation chance is a decisive factor here: if
 *  it is too high we expect too much mutation and little evolution aside from
 *  occasional lucky guesses. If it is too low, our new generation will have little
 *  chance to contain a much better individual - 'big leaps' in evolution cannot
 *  occur.
 *
 *  Note that as with all evolution handlers, we need a reference to a parent
 *  population object to obtain data such as population size, mutation chance, etc.
 *
 *  @author Matthijs
 */
public class Mutation implements Handler {

    private Population      parent;


    /**
     *  Constructor, sets our parent reference.
     *
     *  @param parent   Reference to the parent Population object.
     */
    public Mutation (Population parent) {
        this.parent =   parent;
    }


    /**
     *  Performing method: this loops over all individuals in a
     *  population and calls the mutate method on each of them.
     *
     *  @return         A status int is returned, generally 1 when
     *                  mutation was successful.
     */
    public int run () throws Exception {
        for (boolean[] individual : this.parent.individuals)
            individual = mutate(individual);
        return 1;
    }


    /**
     *  Mutation method: this 'mutates' certain genes in a given
     *  individual. Note that genes are mutated depending on a
     *  set mutation chance: if a random number falls below that
     *  chance threshold, the gen is mutated.
     *
     *  Since we work with boolean values here we do not need to
     *  find a random new gen: we can simply flip the state from
     *  false to true or vice versa.
     *
     *  @param individual   A boolean array containing the individuals
     *                      genotype.
     *  @return             The transformed individual as boolean array.
     */
    private boolean[] mutate (boolean[] individual) {
        for (int i = 0; i < individual.length; i++)
            if ((Math.random() * 100) < parent.parent.mutationChance)
                individual[i] = !individual[i];
        return individual;
    }
}
```

Appendix B

Subversion repository URL:	http://svn.fragfrog.nl/java/geneticAlgorithm/	
Documentation URL:	http://documentation.fragfrog.nl/geneticAlgorithm/html/	
Authentication:	user:	genetic
	Password:	algorithm