Genetic Algorithm Optimization in Maze Solving Problem

Thomas Pasquier, Julien Erdogan

Institut Superieur d'Electronique de Paris
thomas.pasquier@isep.fr julien.erdogan@isep.fr

Abstract. In this essay, we studied crossing selection and local optimization for Genetic Algorithm. In particular we will study how to enhance performance of Genetic Algorithm for Maze Resolution Problem. We will study the impact of different operators, crossover and mutation, over the diversity and the performance. We will also try to determine the parameters which provides the best results.

1 Introduction

Genetics is the science which deals with the molecular structure and the function of genes. It studies the behaviour of genes in the context of cells or organism, the pattern of inheritance from parents to offspring and the genes distribution, variation and change in a population. In nature genes correspond of four difference DNA nucleotides sequences encoding a particular behaviour. If genes play a large role in the definition of an individual, it is the combination of genetics and individual experience that determines the ultimate outcome. More fitted individuals had more chance to survive and transmit their genes, so the population slowly evolves towards more fitted individuals. Genetic mutations generate new behaviours which if positive may be transmitted from generation to generation and become a dominant feature.

Genetic algorithm is an heuristic algorithm that mimics natural evolution process. It's an algorithm commonly used to generate solutions to optimization and search problems. A population of candidate solution to an optimization or search problem called individuals encoded by byte sequences called genome evolve towards an optimal solution. At each generation, the fitness of each individuals is evaluated, and some individuals are stochastically selected(based on their fitness) and modified (combined and randomly mutated) to generate a new population. This new population is used in the next iteration of the algorithm. The algorithm terminate when a maximum number of iteration is reached, a satisfactory solution is found or the genes converge.

To implement a genetic algorithm one needs to define a fitness function to evaluate the solution domain and a genetic representation of this solution domain. The fitness function return a single numerical fitness, which is supposed

to be proportional to the utility or the ability of the individual that chromosome represents[3].

When a Genetic Algorithm is correctly implemented, populations evolved such that the average fitness and the best individual fitness in each generation increase towards an optimum value. Convergence is the progression towards increasing uniformity. A gene is said to converge when 95% of the population share the same value[7].

It is very hard to know what values of GA parameters (such as population size, mutation operator, probability of mutation, etc.) to use to solve a problem. This problem has been investigated by Kalyanmoy Deb and Samir Agrawal[6], they highlighted the following facts :
- Population either too small or too large are detrimental. For a population too small the number of generation required to solve a problem is too large. If the population is too large, the number of generation needed to find an optimum is too large.
- GA with crossover and mutation operator perform better than implementation using only crossover or only mutation.
- For complex problem with multi-modality[1] and high chance of deception the crossover operator is the most important factor as mutation alone fail miserably to solve those kind of problems.

Deception is formally described[4] as : when the average of fitness of schemata which are not contained in the global optimum is greater than the average of those which are. A problem is defined as fully deceptive if all low-order schemata containing a suboptimal solution are better than other competing schemata. Deceptive problem are very hard to solve.

Our problem fall under the category of problem with multi-modality and high deception rate. So, we should put a strong emphasis on the cross-over operator while the mutation operator will play a less important role.

2 Rules

Our maze is composed of 4 types of cases:
- maze entrance
- maze exit
- path
- wall

The entrance and the exit are situated respectively at the top left corner and the bottom right one(see fig 1.). A path can't form what we call rooms (structures formed of m*n contiguous path, where n ¿ = and m ¿ = 2).

[1] Those category of problem contain a large number of false attractor.

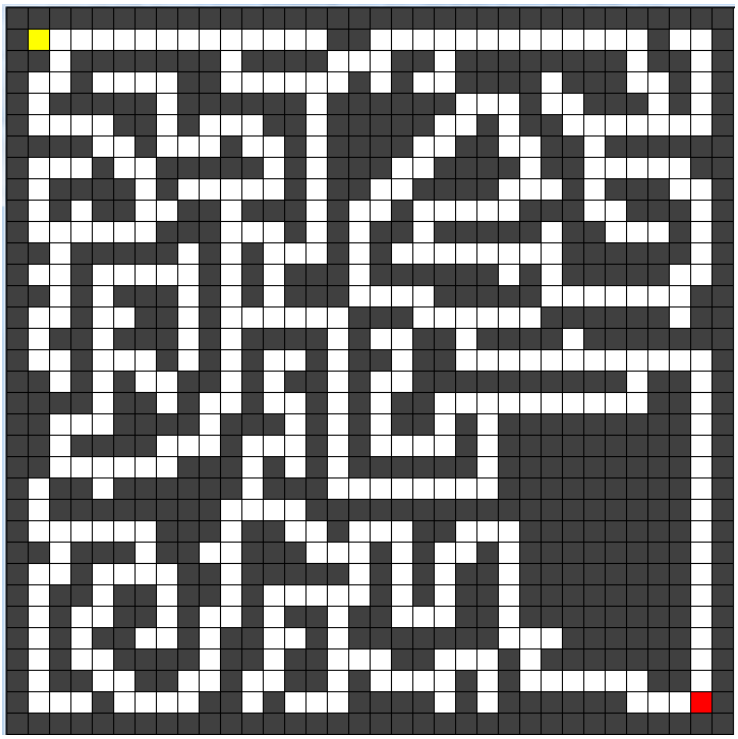

Fig. 1. Representation of a maze

3 How to measure genetic algorithm performance

3.1 Diversity

Diversity allow us to measure the current level of convergence. The diversity is the number of gene different between two individuals divided by the total number of genes. The closer the diversity is to 0, the closer we are from a solution (being the optimal solution or a suboptimal one). Diversity is one of the most important factors, the process is said to converge when the individuals of the gene pool are identical or near to be so. Once this event occurs, the crossover operator cease to generate new individuals, the algorithm allows all its trials to a very small subspace. Sadly, it may occurs before the true optimum as been found, it's called premature convergence[8].

3.2 On-line and Off-line Performance

There is two common performance measures used to evaluate the effectiveness of function optimizer : the on-line and off-line performance. On-line performance is the mean of all trials (or generation), while off-line performance is the mean of

the best individual of each generation[8]. On-line performance is suited when the learning process is done while performing the task such as gambling or economics. Off-line performance only evaluates the best individual and is better suited to problem whose objective is to find a solution or to create a model.

$$\text{On-line performance} \quad x_e(t) = \frac{1}{t} \cdot \sum_{i=1}^{t} f_e(i)$$

$$\text{Off-line performance} \quad x_e^*(t) = \frac{1}{t} \cdot \sum_{i=1}^{t} f_e^*(i)$$

$$\text{Best so far} \quad f_e^*(i) = \max_{j=1,i} f_e(j)$$

Fig. 2. Equation of on-line and off-line performance.

In our case, the most important measure will be the off-line performance as we are trying to find a solution for our maze.

4 Diversity, Mutation and Crossover Operator

During the reproductive phase of a GA, individuals are selected among the population and recombined producing offspring which will compose the next generation. Parents are selected randomly from the population using a scheme which favours the fittest individuals (good individuals may reproduce several time, while poor individual may never reproduce)[3].

When two parents has been selected, their chromosomes are recombined by using the mechanisms of crossover and mutation. Crossover is basically taking parts of both parent genome to create a new offspring genome. Mutation alter each gene with a small probability.

4.1 Many Villages Algorithm

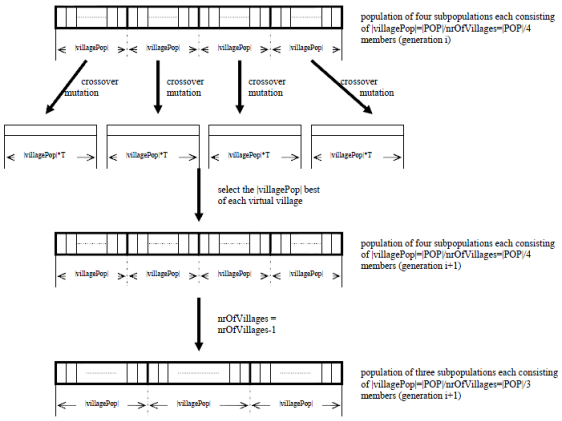

Fig. 3. How many villages algorithm works

Many Villages Algorithm divide the population in several separated subgroups, called villages, which merge after a certain number of generation (as described in the above schema).

It has been demonstrated[2, 1] that Many Villages Algorithm or SEGA algorithm (for Segregative Genetic Algorgrithm) avoids too early convergence by preserving more genetic diversity. Many Villages Algorithm also allow to paralelize computation and reduce computation runtime for an equivalent sized population.

Many Villages algorithm is inspired by the natural phenomena of the emergence of sub-species if sub-population of a same species are separated from each other.

We discussed the diversity measure earlier. In case of many villages we consider the diversity to be the average of the diversity of all villages.

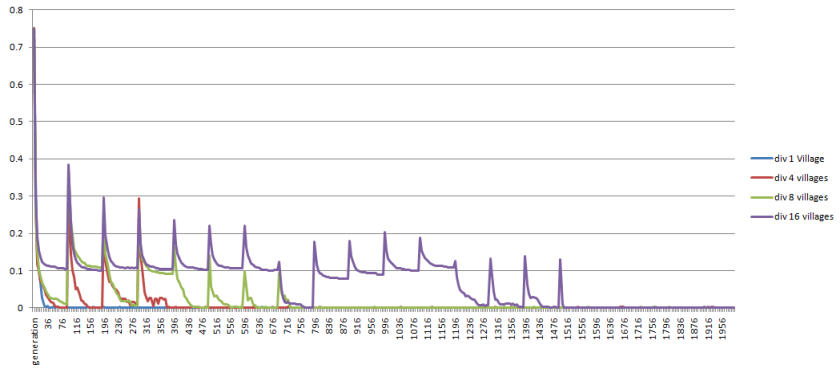

Fig. 4. Diversity in regard of the number of villages

We see clearly here that many villages algorithm maintain a rather high diversity over a longer period. With the same population size, when the population is not divided into village (normal GA implementation) convergence occurs in around 40 generations, while with this same population is divided between sixteen villages we converge much later.

An early convergence may occur inside a village, but when those village are merged the diversity grew up again allowing to eventually find a better solution to the problem.

It is also interesting to note that on the same machine using the same operators, it needs 20 minutes to compute 2000 generation if the population is not divided, while it takes only 6 when there is 16 villages.

4.2 Crossover Operator

```
public byte[][] cross(Individual father, Individual mother) {
    byte[] m = mother.getGenome();
    byte[] f = father.getGenome();
    byte[] child = new byte[Maze.SIZE*Maze.SIZE];
    byte[] child2 = new byte[Maze.SIZE*Maze.SIZE];


    for(int i=0; i<child.length; i++){
        int rd = rand.nextInt(2);
        if(rd==0){
            child[i]=m[i];
            child2[i]=f[i];
        }else{
           child[i]=f[i];
           child2[i]=m[i];
        }

    }
    byte[][] results = new byte[2][];
    results[0]=child;
    results[1]=child2;
    return results;
}
```

Fig. 5. Java implementation of Improved Segment Crossover Operator.

Simple Crossover Operator randomly choose each gene from the mother or the father.

```
public byte[][] cross(Individual father, Individual mother) {
    byte[] m = mother.getGenome();
    byte[] f = father.getGenome();
    byte[] child = new byte[Maze.SIZE*Maze.SIZE];
    byte[] child2 = new byte[Maze.SIZE*Maze.SIZE];

    int pos1 = rand.nextInt(child.length-1);
    int pos2 = rand.nextInt(child.length-pos1)+pos1;
    int i;
    // copy mother gene between 0 and pos1
    for(i=0; i<pos1; i++){
        child[i] = m[i];
        child2[i] = f[i];
    }
    // copy father gene between pos1 and pos2
    for(;i<pos2;i++){
        child[i]=f[i];
        child2[i] = m[i];
    }
    // copy mother gene between pos2 and size
    for(;i<child.length;i++){
        child[i]=m[i];
        child2[i] = f[i];
    }
    byte[][] results = new byte[2][];
    results[0]=child;
    results[1]=child2;
    return results;
}
```

Fig. 6. Java implementation of Segment Crossover Operator.

Segment Crossover Operator divide the genome in three parts (of random size). The offspring take segment from the mother or the father randomly.

```
public byte[][] cross(Individual father, Individual mother) {
    byte[] m = mother.getGenome();
    byte[] f = father.getGenome();
    byte[] child = new byte[Maze.SIZE*Maze.SIZE];
    byte[] child2 = new byte[Maze.SIZE*Maze.SIZE];

    int pos1;
    if(father.getPath().size()>mother.getPath().size())
        pos1 = father.getPath().size();
    else
        pos1 = mother.getPath().size();

    int pos2 = rand.nextInt(child.length-pos1)+pos1;
    int i;
    // copy mother gene between 0 and pos1
    for(i=0; i<pos1; i++){
        child[i] = m[i];
        child2[i] = f[i];
    }
    // copy father gene between pos1 and pos2
    for(;i<pos2;i++){
        child[i]=f[i];
        child2[i] = m[i];
    }
    // copy mother gene between pos2 and size
    for(;i<child.length;i++){
        child[i]=m[i];
        child2[i] = f[i];
    }
    byte[][] results = new byte[2][];
    results[0]=child;
    results[1]=child2;
    return results;
}
```

Fig. 7. Java implementation of Improved Segment Crossover Operator.

Improved Segment Crossover operator behave like the Segment Crossover Operator except that the first segment size is the size of the best working genes sequence between the two parents.

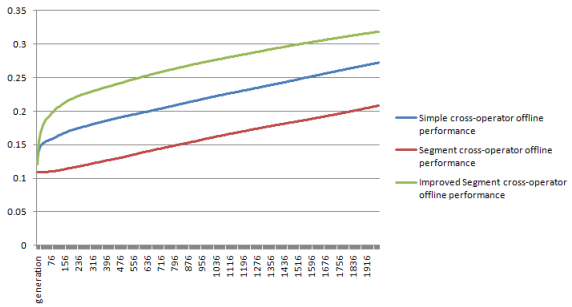

Fig. 8. Cross over operator off-line performance.

It appears clearly that the "'Improved Segment Crossover Operator"' give the best off-line performance as it does not destroy working genome.

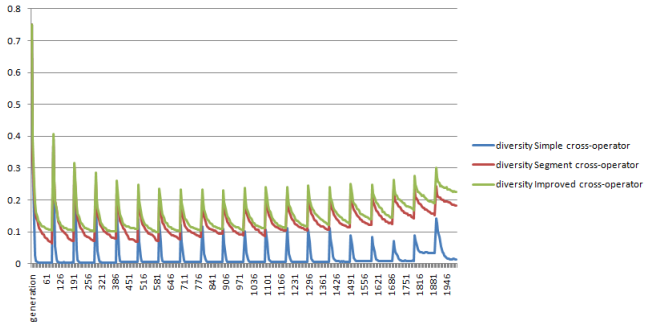

Fig. 9. Influence of the crossover operator on diversity.

It is also clear that Crossover operator based on segment algorithm maintain an higher diversity than simple crossover algorithm. A simple conjecture is that simple crossover operator generates a lot of not working genome, while segment based crossover operator preserve working genes sequence.

5 Local Optimization

As mentioned by Kalyanmoy Deb and Samir Agrawal[6], Cross Over Operator is the most important operator in multi-modal problems with high deception. The role played by mutation is very low as mutations are likely to create bad individuals which would not survive. From this fact, we decided to introduce Local Optimization which modify genome using knowledge on the problem.

5.1 Change Last Operator

```
public byte[] improve(Individual individual) {
    ArrayList<Integer> path = individual.getPath();
    if(path.isEmpty() || path.size() < 2){
        return individual.getGenome();
    }
    int wrongPosition = path.size();
    byte[] genome = individual.getGenome();
    int pos = path.get(path.size()-1);
    int x = pos%Maze.SIZE;
    int y = pos/Maze.SIZE;
    Maze maze = Maze.getInstance();
    int dir=0;
    boolean isInside=false;
    int count = 0;
    do{
        dir = rand.nextInt(4);
        if(dir == Individual.EAST){
            x=+1;
        }else if(dir == Individual.WEST){
            x=-1;
        }else if(dir == Individual.NORTH){
            y-=1;
        }else if(dir == Individual.SOUTH){
            y+=1;
        }
        isInside=true;
        if(x<0 && x>=Maze.SIZE)
            isInside=false;
        if(y<0 && y>=Maze.SIZE)
            isInside=false;
        // we can't find a solution we exit
        if(count++ > 15)
            return individual.getGenome();
    }while(!isInside || maze.getGridValue(x, y)==Maze.WALL_FIELD || path.get(path.size()-2) == x+y*Maze.SIZE);
    // is not inside the maze || go through a wall || go back to the previous cases
    genome[wrongPosition] = (byte)dir;
    return genome;
}
```

Fig. 10. Java implementation of the change last operator.

The purpose of this algorithm is to replace the first wrong gene, the one which lead to a wall, by a gene which lead to a valid path into the maze.

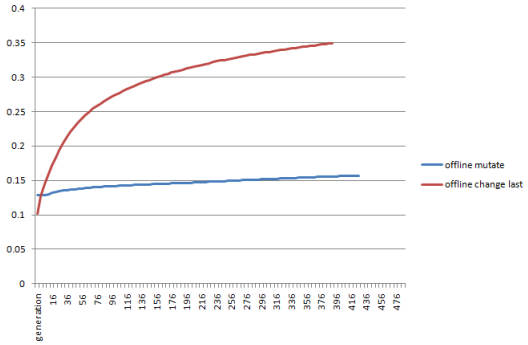

Fig. 11. Change-last operator influence on off-line performance.

By using this algorithm we see a clear improvement. It's due to the simple fact that each generation is very likely to see a more fitted individual than those in the previous generation.

5.2 Out of Dead End Operator

```
public byte[] improve(Individual individual) throws Exception{
  ArrayList<Integer> path = individual.getPath();
  if(path.isEmpty() || path.size() == 1)
    return individual.getGenome();
  Maze maze = Maze.getInstance();
  int posx = path.get(path.size()-1)%Maze.SIZE;
  int posy = path.get(path.size()-1)/Maze.SIZE;
  int av = maze.getAvailablePath(posx, posy);
  if(av == 1){
    ArrayList<Integer> crossRoad = new ArrayList<Integer>();
    ArrayList<Integer> genPosition = new ArrayList<Integer>();
    int i = 0;
    // we look if there is several way avalable at starting position
   for(int j = 0; j<path.size()-2; j++){
      int p = path.get(j);
      posx = p%Maze.SIZE;
      posy = p/Maze.SIZE;
      // search if their is more than one path available
      if(maze.getAvailablePath(posx, posy)>2){
        crossRoad.add(p);
        genPosition.add(i);
      }
      i++;
    }
    if(crossRoad.isEmpty())
      return individual.getGenome();
    int a = rand.nextInt(crossRoad.size());
    int positionInGenome = genPosition.get(a);
    int choosenCross = crossRoad.get(a);
    int val=choosenCross;
    int dir = 0;
    byte[] genome = individual.getGenome();
    do{
      dir = rand.nextInt(4);
      if(dir == Individual.EAST)
        val+=1;
      else if(dir == Individual.WEST)
        val-=1;
      else if(dir == Individual.SOUTH)
        val+=Maze.SIZE;
      else if(dir == Individual.NORTH)
        val-=Maze.SIZE;
    }while(path.get(positionInGenome-1)==val || path.get(positionInGenome+1)==val || val<0);
    byte[] r = new byte[1];
    r[0]=(byte)dir;
    int size = path.size()-positionInGenome;
    genome = this.replace(genome, positionInGenome, size, r);
    return genome;
  }else{
    return individual.getGenome();
  }

}
```

Fig. 12. Java implementation of the out of dead end operator.

Firstly, the algorithm check if the genes sequence leads to a dead end, then if it is the case it identify any other potential path, choose one randomly and modify the genome to take this path.

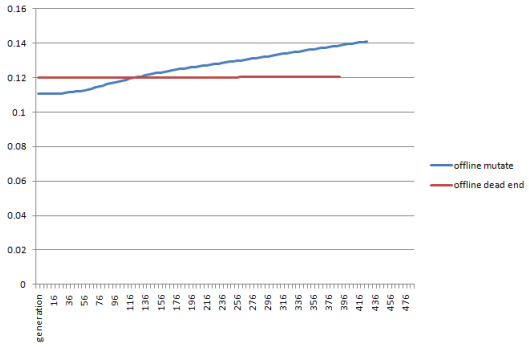

Fig. 13. Out of dead end operator influence on off-line performance.

We can see that Out Of Dead End Operator has no influence on performance used alone.

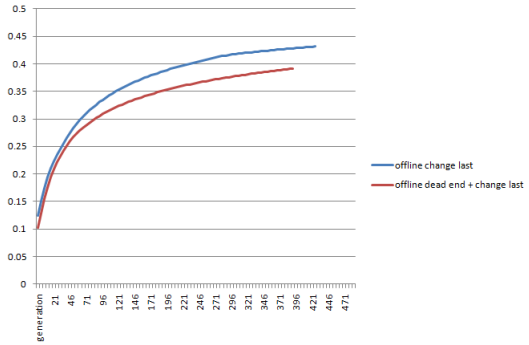

Fig. 14. Comparison between Change Last Operator applied alone and Change Last + Out Of Dead End operators

Change Last Operator seems to decrease the performance achieved by our genetic algorithm. However this algorithm should not be ignored as empirical test prove that it seems to help to find the optimum solution and avoid a lot of sub-optimal one. A new test protocol may need to be created to evaluate its impact over the problem resolution.

5.3 Remove Loop Operator

```
public byte[] improve(Individual individual) {
   byte[] newGenome = individual.getGenome().clone();
   for(int i=0; i < newGenome.length; i++){
      Vector vector = new Vector();
      byte val = newGenome[i];
      if(val == Individual.EAST)
         vector.x+=1;
      else if(val == Individual.WEST)
         vector.x-=1;
      else if(val == Individual.NORTH)
         vector.y-=1;
      else if(val == Individual.SOUTH)
         vector.y+=1;
      for(int j=i+1; j < newGenome.length; j++){
         val = newGenome[j];
         if(val == Individual.EAST)
            vector.x+=1;
         else if(val == Individual.WEST)
            vector.x-=1;
         else if(val == Individual.NORTH)
            vector.y-=1;
         else if(val == Individual.SOUTH)
            vector.y+=1;
         byte[] element = new byte[1];
         try{
            if(vector.x == 1 && vector.y == 0){
               element[0]=Individual.EAST;
               newGenome = this.replace(newGenome, i, j-i, element);
               return newGenome;
            }else if(vector.x == -1 && vector.y == 0){
               element[0]=Individual.WEST;
               newGenome = this.replace(newGenome, i, j-i, element);
               return newGenome;
            }else if(vector.x == 0 && vector.y == 1){
               element[0]=Individual.SOUTH;
               newGenome = this.replace(newGenome, i, j-i, element);
               return newGenome;
            }else if(vector.x == 0 && vector.y == -1){
               element[0]=Individual.NORTH;
               newGenome = this.replace(newGenome, i, j-i, element);
               return newGenome;
            }
         }catch(Exception e){
            System.out.println(e.getMessage());
            System.exit(-1);
         }
      }
   }
   return individual.getGenome();
}
```

Fig. 15. Java implementation of the remove loop operator.

In this algorithm, the genome is explored to detect loop. We construct a vector by browsing to all the genes, when a unit vector is detected we remove the sequence of gene constructing this vector and replace it by a single gene which equals to the same unit vector.

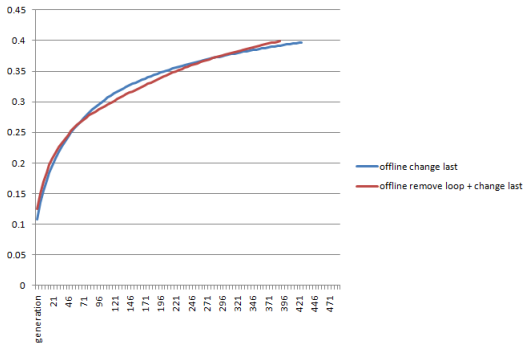

Fig. 16. Comparison between Change Last Operator applied alone and Change Last + Remove Loop Operators

Remove loop as a slight effect on algorithm performance. A complexity study should be made to evaluate if the CPU consumption is worth the slight improvement.

5.4 Conclusion

We have seen that local optimization are more effective than a simple mutation. For the rest of this paper we will consider as the most effective algorithm the following:
- Cross over operator : Improved Segment Cross Over
- Improvement / Mutation Operator : Change Last Operator + Remove Loop Operator + Out Of Dead End Operator

6 Selecting the alpha parameters

```java
public double computeFitness(Individual individual) {
    this.setA(individual.getConfiguration().getAlpha());
    double factorA=0;
    double factorB=0;
    Maze maze = Maze.getInstance();
    ArrayList<Integer> path =individual.getPath();
    if(path.size() <= 0)
        return 0;
    int pos = path.get(path.size()-1);
    int x = pos%maze.getSize();
    int y=pos/maze.getSize();

    double distance_x = (double)maze.getSize() - (double)x;
    double distance_y = (double)maze.getSize() - (double)y;
    double distance = Math.sqrt(distance_x*distance_x + distance_y*distance_y);
    factorA = 1 - (distance/Math.sqrt((double)maze.getSize()*(double)maze.getSize()*2));

    if(!path.isEmpty()){
        factorB = computeFactorB(path.size(), maze.getSize());
     }else{
        factorB = 0;
     }
    return a* factorA + (1-a) * factorB;
}

public double computeFactorB(double pathSize, double mazeSize){
    double factorB = 0;
    double optiPathSize = 2*mazeSize-1;
    double diff = optiPathSize - pathSize;

    double square = Math.pow(diff, 2);
    double max = Math.pow(optiPathSize, 2);
    factorB = (max-square)/max;
    return factorB;
}
```

Fig. 17. Java Implementation of the Fitness Algorithm.

Our fitness function rely on two factor. One is the distance from the shortest path possible, the second one is the distance from the exit.

The alpha parameter influence how the fitness value is computed. Fitness value has direct influence over the diversity. The most effective way to evaluate if our alpha value as been well chosen is to vary this alpha value and to verify which one provide us with the highest diversity over time.

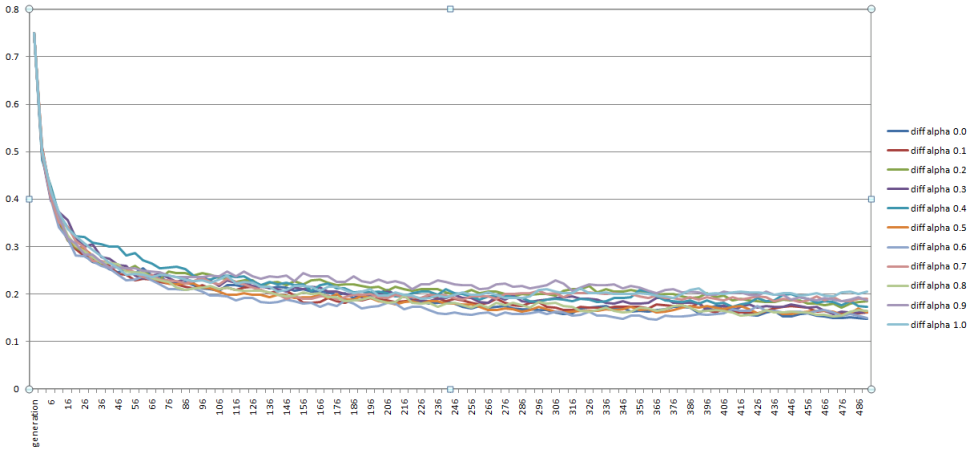

Fig. 18. Influence of alpha parameter on diversity.

We see that after the first hundred generations the diversity vary between 15% and 25%. An alpha parameters of 0.9 seems to offer the highest diversity over the whole period. We will compare how the off-line performance vary with the four alpha parameters offering the highest diversity.

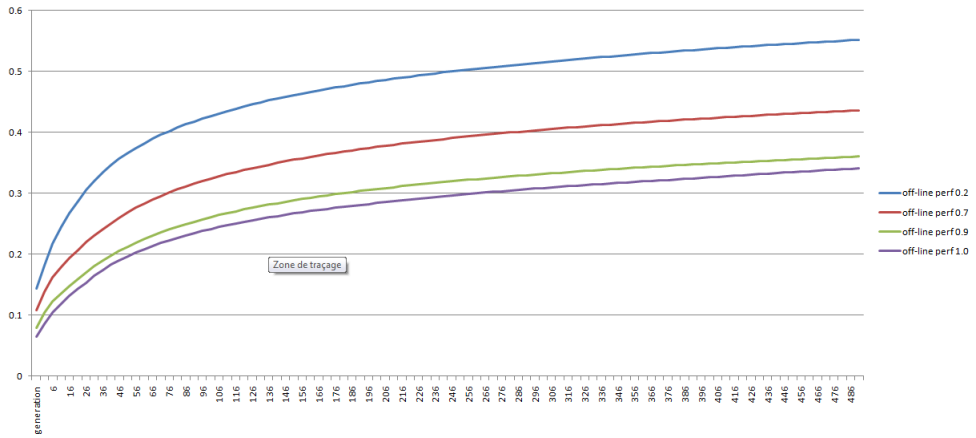

Fig. 19. Influence of alpha parameter on off-line performance.

We cannot compare the value directly, we have to compare the shape of the curve. We can see that all the curve have exactly the same shape. We decided to choose an alpha parameters of 0.7, as it does not put too many emphasis on one of the two factor of our fitness function and provides good performance.

7 Population size

So far we have selected our operators,our alpha parameter, we now have to select the optimum population size. We will study the influence of the population size on the diversity and the off-line performance. We will test with twenty villages of the following size each : 10, 20, 50, 100. A too large population will prevent a convergence to happen, a too small one will decrease the diversity and lead to an early convergence.

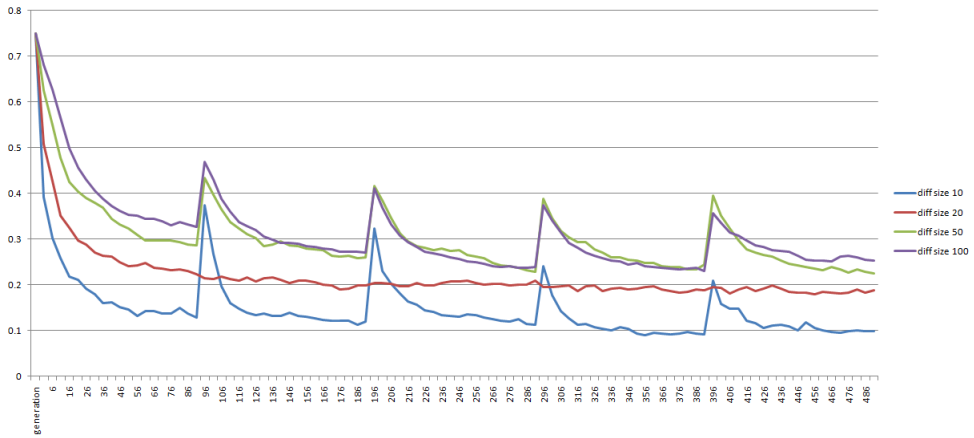

Fig. 20. Influence of population size on diversity.

We can see that the larger the population, the higher the diversity. For an unknown reason with a Population of size 20, the diversity curve has a totally different shape (reproducible phenomena).

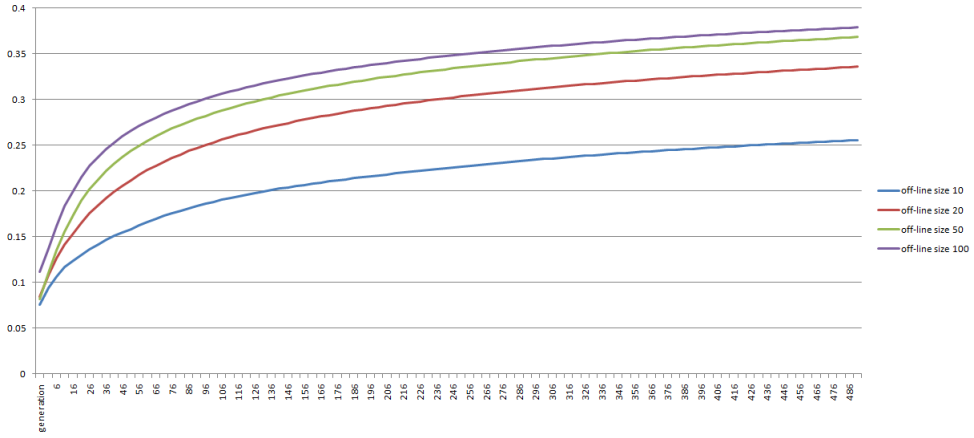

Fig. 21. Influence of population size on off-line performance.

The larger the population, the better the performance. It is also important to note that computation time is increased by the size of the population (a very simple estimation of the complexity is O(m*n*n) with m the number of village and n the population size). A population size of 50 seems to be a good compromise as we see that a population of 100 does not increase performance or diversity that much.

8 Conclusion

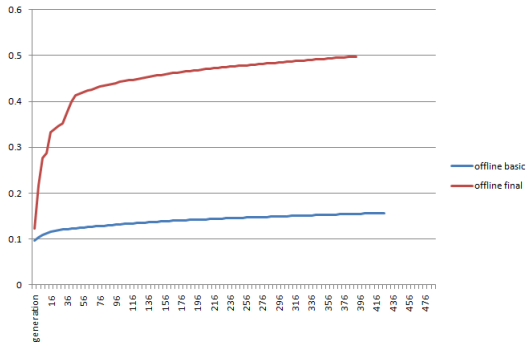

Fig. 22. Comparison between the basic algorithm and the final solution.

The curve above demonstrate the clear performance improvement we have been able to make through our study. The performance has been multiplied by 4. Even if we cannot find a solution to all maze, we are way more likely to find one in a reasonable time.

The code developed during this project allow to perform a large set of test, to study impact of cross operator, mutation operator and many other factor. Some simple development can make it even more powerful (configuration of maze generator algorithm, configuration of mating algorithm etc.) and it can become the root of a powerful tool to test genetic algorithm in maze resolution. Further study could focus on such parameters and would increase performance even more.

An other path potentially interesting to explore is the parallelization of the matting process. It could be easily by using MapReduce[5] as follow :
- Input Key : village-id, Input Value : genome
- Intermediate Key village-id, Intermediate Value : genome
- Output Key village-id, Output Value : genome

The Map operator would filter the input using a fittest threshold (defined by the average fittest score of the previous generation). The Reduce operator would cross the genome who survived through the Map operator and generate the new generation put in the Output file. We call this algorithm recursively. It would without a doubt increase the performance by a very large factor.

An other interesting approach would be to create initial villages with different operators and parameters take those into consideration when crossing two individuals. Genetic selection may eliminate under performing combination and lead to operators and parameters combination fitted to the really maze we are trying to solve. It may lead to a two level genetic algorithm, one level will be the actual solution and the second a genetic algorithm over the parameter influencing the generation of the first one. In An Overview of Genetic Algorithms[4], advanced crossover operator learn where the crossover operator should be most probable to happened (which is a more evolved and less systematic approach of our Ïmproved Segment Crossover¨), other example of dynamic operator are described in the same paper.

Finally, further study of the algorithms used here could lead to simplification and complexity reduction, if done it makes no doubt that it will increase the performance by an important factor.

9 Note

Implementation sources are available through subversion at :
https://geneticmaze.googlecode.com/svn/trunk/
Manual, application and other documents are available here :

http://code.google.com/p/geneticmaze/downloads/list

9.1 How to use the program

To test the algorithms we used a Java program. To launch the program you need to install Java first. There is a link to download the installer for Windows : http://www.java.com/fr/download/.

The program uses xml files to perform the algorithms you want to solve the maze. This file is set as followed.

```
- <Configuration>
    <Display>org.maze.display.SwingDisplay</Display>
  - <Population>
      <Villages>20</Villages>
      <IndividualConfiguration>./config/behaviours/changelast.xml</IndividualConfiguration>
      <Name>ChangeLast</Name>
      <PopulationSize>20</PopulationSize>
    </Population>
    <Display>org.maze.display.SwingDisplay</Display>
  - <Population>
      <Villages>20</Villages>
      <IndividualConfiguration>./config/behaviours/removeloop.xml</IndividualConfiguration>
      <Name>RemoveLoop</Name>
      <PopulationSize>20</PopulationSize>
    </Population>
  </Configuration>
```

Fig. 23. XML behaviour file

The XML configuration file contains the information about the population to use, the number of village and the population size. The configuration of the behaviours is done through an other XML file. The user can tst more than one population at a time as it is shown above.

Each behaviours is detailed itself in an XML file. Here is an example of XML behaviour file.

```
- <Behaviours>
  - <CrossBehaviour>
      <Behaviour>org.maze.behaviours.ImprovedSegmentCrossBehaviour</Behaviour>
    </CrossBehaviour>
  - <ImprovementBehaviour>
      <Behaviour>org.maze.behaviours.OutOfDeadEndImprovementBehaviour</Behaviour>
      <Behaviour>org.maze.behaviours.RemoveLoopImprovementBehaviour</Behaviour>
      <Behaviour>org.maze.behaviours.ChangeLastImprovementBehaviour</Behaviour>
    </ImprovementBehaviour>
    <MutationBehaviour />
  - <FitnessBehaviour>
      <Behaviour>org.maze.behaviours.SimpleFitnessBehaviour</Behaviour>
    </FitnessBehaviour>
    <AlphaValue>0.7</AlphaValue>
  </Behaviours>
```

Fig. 24. Default configuration XML file

The XML behaviour file contains the cross behaviour, the improvements behaviours, the mutation behaviour, the fitness behaviour and the alpha value. If no xml file is passed to the program, it will use the default configuration XML file shown above.

To launch the program you must use the following line in the windows command prompt :
$java\ -jar\quad Maze.jar\quad ConfigurationFile.xml$

References

1. Michael Affenzeller. Segregative genetic algorithms (sega): A hybrid superstructure upwards compatible to genetic algorithms for retarding premature convergence. *SYSTEMS SCIENCE*, 2001.
2. Michael Affenzeller. A generic evolutionary computation approach based upon genetic algorithms and evolution strategies. *SYSTEMS SCIENCE*, 2002.
3. Ralph R. Martin David Beasly, David R. Bull. An overview of genetic algorithm : Part 1, fundamentals. *Inter-University Committee on Computing*, 1993.
4. Ralph R. Martin David Beasly, David R. Bull. An overview of genetic algorithm : Part 2, research topics. *Inter-University Committee on Computing*, 1993.
5. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.
6. Kalyanmoy Deb and Samir Agrawal. Understanding interactions among genetic algorithm parameters. In *in Foundations of Genetic Algorithms 5*, pages 265–286. Morgan Kaufmann, 1999.
7. K DeJong. The analysis and behaviour of a class of genetic adaptive systems. *University of Michigan*, 1975.
8. Michael L. Mauldin. Maintaining diversity in genetic search. *Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213*, 1984.