Pipelined Genetic Architecture with Fitness on the Fly

F. Sáenz[1], A. Ibarra[2], J. Lanchares[2]
J. I. Hidalgo[2]

[1] *Dpto. de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Avda. Complutense s/n, E-28040 Madrid, Spain.*
e-mail: fernan@dacya.ucm.es

[2] *Dpto. de Arquitectura de Computadores y Automática, Universidad Complutense de Madrid, Avda. Complutense s/n, E-28040 Madrid, Spain.*
e-mail: ibaiba@dacya.ucm.es

Abstract

One of the main bottleneck in Genetics Algorithms is the fitness evaluation for each individual. In this work, we propose a new fitness evaluation method, which solve the bottleneck calculating a fitness on the fly.

Introduction

Finding a solution in the search space is a complex task when the number of variables and cardinalities of domains are large. Several heuristic techniques have been used, as simulated annealing [1], evolution programs [2], tabu search [3], Grasp (Greedy Randomized Adaptive Search Procedure) [4], and neural networks [5].

The main disadvantage of these techniques is the lack of completeness. Optimum is not guaranteed to be found. However, it is not always needed to find the best solution, but a good solution when, either computing time is a critical factor, or cost functions are not precise enough for justifying a complete search. In these cases, stochastic approaches are fully justified.

Genetic algorithms [6] are stochastic algorithms implementing a search procedure which model a biological phenomena: genetic inheritance, and Darwinian survival. A genetic algorithm maintains a population of individuals coding potential solutions. These individuals are mated and mutated through the application of genetic operators.

Performance of genetic algorithms can be effectively increased through the parallel evaluation of genetic operators. In this work, we enhance the performance of genetic algorithms, performing parallel evaluation of genetic operators through pipelining. For this goal, we firstly propose a novel pipelined genetic architecture to implement genetic algorithms which behaves different from a usual sequential algorithm [7]. Then, we implement the architecture and compare its efficiency against the sequential one, focusing on the total generations needed to obtain solutions.

Genetic Algorithms Background

Many of interesting problems have not reasonably fast algorithms to solve them. Most belongs to the class of NP-hard combinatorial optimisation problems, which are NP-complete decision problems requiring an algorithm of exponential complexity with the problem size.

Guided random search techniques are based on enumeration techniques but use additional information to guide the search. Moreover, the search has an stochastic component in order to explore randomly (and partially) the search space, which allows both to explore local optima and escape from them (unlike hill-climbing techniques). Evolutionary algorithms are examples of such techniques. Genetic algorithms (GAs) are instances of evolution algorithms [2], which manage raw data implemented as bit strings, instead of managing complex data structures as evolution algorithms do.

Modelling the problem means two issues: the first one is the coding of solutions, that is, coding chromosomes in a way each one may represent a solution. The second one is to design a cost function, which measures the fitness of the individual, that is, how good the solution is under a given criterium.

The fitness function is the link between the GA and the problem to solve. A fitness function takes a chromosome as input and returns a measure of its goodness. This function provides the information needed to select individuals for

mating.

GAs focus on local optimisation, but evolution allows them to jump to different points of the search space, which makes the optimisation more global. According to evolution, best individuals are likely to survive and generate offspring, therefore transmitting their best features to new generations. In this procedure there are two basic operators, known as genetic operators, involved in GA operation:

- *Crossover*, i.e., mating pairs of individuals.
- *Mutation* of each individual.

In addition to genetic operators, the fitness function, previously mentioned, must also be supplied. This function is used in the selection stage, which selects best individuals to survive.

With these operations, a generic GA Fig.(1) consists of the following:

1. Initialise a population. A set of coded binary strings is created.
2. Evaluate each chromosome in the population. The fitness function is applied to each chromosome.
3. Create new chromosomes by means of crossover and mutation. The first operator takes into account the fitness values (computed in the previous step) of each chromosome in order to select the best ones to mate together.
4. Evaluate the new chromosomes, select the best ones and overwrite the worst ones. This stage is known as selection.
5. Repeat from step 2 if more generations are required, else take the best chromosome and return it as the solution.

Above-mentioned operations have different ways to be implemented, considering different criteria and different parameters. In the following list they are briefly discussed:

- *Selection.* This is a weighted random selection of two or more individuals whose genetic material will be propagated to the next generation. The weight induces the probability for best chromosomes to be selected. Therefore, there is an unpredictable component in the next operator from selection, which is determined by the weight parameter.
- *Crossover.* From selected chromosomes, their genetic material is combined for generating the new ones. Typically, two chromosomes are mated, but more are allowed depending on the implementation. There are several ways to perform crossover. For instance, a bit of the string is selected for splitting the two chromosomes and after, the corresponding parts are swapped. Another alternative consists of doing so in a bit per bit approach.
- *Mutation.* After generating offspring, there is a random selection of new chromosomes for changing particular genes. This is guided by a parameter, which induces the probability of mutation. This operation is quite important in the evolution process, because it allows GAs to explore other points in the search space, escaping from local optima. Obviously, it must be adjusted in order to avoid a pure random search.
- *Fitness.* Each chromosome is characterised by its fitness, which is problem dependent. There are no GA parameters related to it.

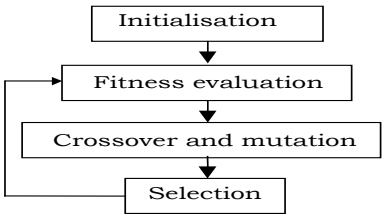

Figure 1: Structure of a Generic Genetic Algorithm.

GAs have been proven useful to tackle complex optimisation problems, even easily when coding is natural from the problem posed. Nonetheless, as any other stochastic procedure, there is no way of knowing whether optimum has been computed. Moreover, there is no clear stopping criterion for termination. However, when other approaches cannot be applied because of either restrictions (e.g., computation time in real time applications) or they are expensive (e.g., no-need of best solutions, but for reasonable ones), GAs are fully justified.

Pipelined Genetic Architecture

In this Section, we present the design of a pipelined genetic architecture. For other related parallel models [8],[9], it has been shown and analysed the high speedup of hardware implementations when compared with software implementations [9]. In this work, we propose a new approach which allows a better use of the pipeline.

Identifying Dependencies

Figure.(2) shows the basic Holland genetic algorithm (HGA), which embodies the fitness, selection, crossover, and mutation operations. In this algorithm as well as forthcoming ones, a pseudo-code has been used. The main loop iterates until a predefined number of iterations had been reached or a feasible solution had been found under the programmer's criteria. The population is the array X, with elements Xi representing each chromosome in the population. Each element is a record with two fields: *fitness* (devoted to hold chromosome's fitness), and the binary string representing genes in the chromosome (not shown in the Figure). The function *select* selects two chromosomes from X. The function *crossover* gives two new chromosomes (Yj and Yk, elements of an array Y) as their children. The function *mutation* mutates these two new chromosomes. The function *new_generation* assembles the new population X from new chromosomes in Y.

```
 1 .procedure HGA();
 2 .begin
 3 .  t:=0;
 4 .  for i:=1 to population_size do
 5 .    Xi := random_chromosome;
 6 .  repeat
 7 .    for i:=1 to population_size do
 8 .      Xi.fitness := fitness_evaluation(Xi);
 9 .    X := mean_fitness_evaluation(X);
10 .    for all_pairs do
11 .      select(Yj,Yk,X);
12 .      crossover(Yj,Yk);
13 .      mutation(Yj,Yk);
14 .    new_generation(X,Y);
15 .    t:=t+1;
16 .  until max_iteration(t) or feasible_solution;
17 .end;
```

Figure 2: Sequential Basic Holland Genetic Algorithm.

Figure.(3) shows a slight modification of the basic Holland genetic algorithm following [10], which makes explicit how new individuals are generated. Here, each chromosome has in addition a field *select* which indicates how many times it is selected for crossover. The way the selection is implemented is by using an array Y which holds as many copies of a chromosome Xi as its field *select* indicates. These copies are generated by the function *replicate*. Next, crossover and mutation are applied to pairs of chromosomes. Finally, creation of the new generation becomes as simple as $X:=Y$.

```
 1 .procedure HGA();
 2 .begin
 3 .  t:=0;
 4 .  for i:=1 to population_size do
 5 .    Xi := random_chromosome;
 6 .  repeat
 7 .    for i:=1 to population_size do
 8 .      Xi.fitness := fitness_evaluation(Xi);
 9 .    X := mean_fitness_evaluation(X);
10 .    for i:=1 to population_size do
11 .      Xi.select := round(Xi/X );
12 .      if Xi.select>0 then replicate(Xi,Xi.select,Y);
13 .    for i:=1 to population_size/2 do
14 .      crossover(Yi,Y_{i+population_size/2});
15 .      mutation(Yi,Y_{i+population_size/2});
16 .    X:=Y;
17 .    t:=t+1;
18 .  until max_iteration(t) or feasible_solution;
19 .end;
```

Figure 3: Sequential Genetic Algorithm implemented following [10].

We depict in Figure.(4) the dependency graph for the main loop in Figure.(3), which is useful for identifying functional dependencies and express independent tasks. At this granularity level, there are no independent tasks, but it can be thought of a pipeline from a concurrent point of view. Tasks as pairing off, crossover, and mutation may work concurrently as the first one supplies pairs of chromosomes to the next task, mate, and this one, in turn, supplies offspring to the mutation task. This is the usual pipeline that can be found in the literature [8] [10], and it has been observed its good speedup [9].

Apart from resources considerations, we can split tasks in Figure.(4) into several subtasks since several genetic operations are known independent and can be applied to different chromosomes. Figure.(5) shows this split which yields to several parallel pipelines, in which n stands for population size. Firstly, fitness is computed independently for each chromosome. Next, the mean fitness is computed from all the chromosome's fitness in the population. The replica-

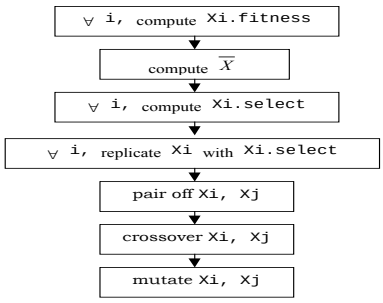

Figure 4: Dependency graph for the main loop in HGA.

tion phase comes next. Pairing off, crossover, and mutation conforms the main pipeline, which has been also replicated for the population; that is, there are $n/2$ pipelines working in parallel. However, there are also two main bottlenecks which disables a continuous data flow, namely, the computation of the mean fitness, and the replication.

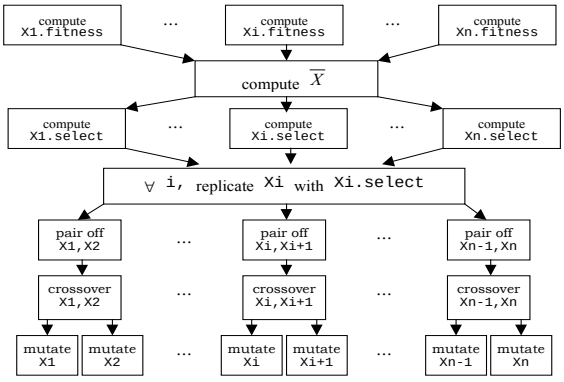

Figure 5: Deeper granularity dependency graph for the main loop in SGA.

In the next Section, we deal with the goal of eliminating these two bottlenecks.

Overcoming Dependencies Problems

Selection operation has been observed to be the main bottleneck in the pipelined architecture, which implies sequential loops slowing down the dataflow. Since selection relies on mean fitness computation (which depends on all the chromosomes in the population), we propose to compute it on the fly, that is, computing an incremental mean.

In order to overcome the wait for chromosome fitness availability, the mean can be computed incrementally (on the fly) by feeding new fitness values to mean fitness computation. Therefore, there is always an available value for $\overline{f}$, which is updated whenever a new selection have to be performed. In such a way, the pipeline must not wait for mean fitness computation. The incremental mean is defined inductively as follows:

$$\begin{aligned} \overline{f}_0 &= \overline{f} \\ \overline{f}_j &= \tfrac{1}{j}\left(\overline{f}_{j-1} \cdot (j-1) + f_{i_j}\right) \end{aligned} \tag{1}$$

where $\overline{f}$ is the mean fitness computed initially, f_{i_j} is the chromosome fitness (from f_1 to f_n, $i_j \in 1,...,n$) actually feeded, and j, the number of chromosome fitness feeded.

This approach has two drawbacks:

- The number of fitness values previously feeded must be kept. This implies to have enough room available for storing , and, more important, it must have an upper bound.
- Since generations are not synchronous anymore, there is no clear notion of generation fitness. That is, should all f_{i_j} be equally weighted as Equation.1 implies?

In this work, we compute the mean on the fly, avoiding the storing of j, and we use an empirically obtained weight for each f_{i_j}, so that the pipeline has a continuos data flow, only limited by the performance of each pipeline stage. The architecture which supports this idea is depicted in Figure.(6).

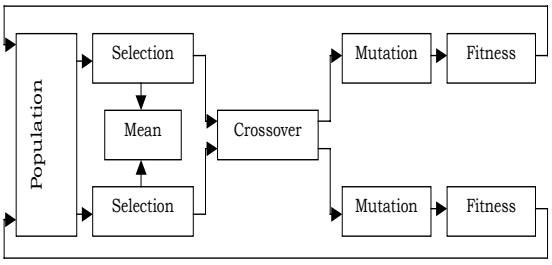

Figure 6: Architecture of the pipelined GA.

This approach has an additional advantage: there is no need of sorting chromosomes, as

another implementations need [10] in order to select best chromosomes to be passed to the crossover module. This task is accomplished in our model by comparing the chromosome fitness to the current mean, so that selection acts as a filter for worst chromosomes.

Implementation of the Pipeline

The modified GA for embodying the pipeline consists of the following: firstly, the initial mean fitness is computed as the standard GA does; then, the pipeline itself comes, which consists of selection, crossover, and mutation operation, which are covered in forthcoming sections.

Selection

From a given population, all chromosomes are quasi-randomly selected (i.e., paired off) for crossover, mutation, or rejection. We have adopted the following criterion: If both chromosomes are good enough, they are selected for crossover. If both chromosomes are not good enough, they are selected for mutation. If only one of them is good, then it is replicated once and the two clones are selected for mutation.

Goodness of chromosomes ($g(X_i, j)$) is determined by an empirically obtained parameter, the selection threshold (st), which indicates the lower bound for the ratio of chromosome fitness, and the mean fitness. This parameter must increase as better solutions are found. The increasing is guided by another parameter, the increase of selection threshold (Δst), which is also obtained empirically. The goodness is defined by:

$$g(X_i, j) = \begin{cases} true, & if \frac{f_i}{\overline{f}} \geq st(j) \\ false, & if \frac{f_i}{\overline{f}} < st(j) \end{cases}$$

The goodness is dependent not only on fitness, but also on "time" (here denoted by j)[1], since hopefully, as time passes by, better solutions are found and, therefore, one must be more strict in selecting chromosomes. Therefore, st is tuned correspondingly with (Δst). This tuning is defined as:

$$st(j) = \begin{cases} st(j-1), \\ \quad if wmf(j) < wmf(j-1) \\ st(j-1) + \Delta st, \\ \quad if wmf(j) \geq wmf(j-1) \end{cases}$$

$wmf(j)$ stands for the weighted mean fitness at time (See below).

Weighted Mean Fitness

We discard the incremental mean approach, and, instead, we compute a weighted mean with the value of chromosome fitness. The criterion applied for computing the weighted mean fitness is the following: If chromosome fitness is greater than the current weighted mean, then it must influence notably, and, conversely, if it is not, then it must slightly influence the current weighted mean. This slight influence is necessary in order to avoid to have chromosomes never selected because the current weighted mean fitness is too high. Both influence degrees are determined by empirically obtained parameters (n_i and s_i, respectively). The computation of the weighted mean fitness at time $j(wmf(j))$ is defined as:

$$wmf(0) = \overline{f}$$

$$wmf(j) = \begin{cases} f_i \cdot n_i + wmf(j-1) \cdot (1 - n_i), \\ \quad iff_i \geq wmf(j-1) \\ f_i \cdot s_i + wmf(j-1) \cdot (1 - s_i), \\ \quad iff_i < wmf(j-1) \end{cases}$$

Crossover

This operation is carried out from a classical point of view. A parameter determines the probability of crossover. If two chromosomes have to be breed, then a gene is randomly selected for splitting each chromosome and exchange the resulting partitions.

Mutation

Mutation may occur when the chromosome must be mutated (decided in the selection stage), or when it is randomly determined in terms of a mutation parameter. Mutation consists of the logical not of a randomly selected gene in the chromosome.

[1] In fact, j denotes here a cpo (complete partial order) between generations of particular pairs of chromosomes. The sequence $j = 1, 2, 3, ...$ identifies the order in which chromosomes are selected, i.e., $X_{i_1}, X_{i_2}, X_{i_3}, ...; i_j \in 1, ..., n$.

Performance Analysis of the Pipelined Genetic Architecture

This Section is devoted to the performance analysis of the proposed pipelined genetic architecture (hereinafter PGA) in terms of its adequacy in finding solutions. Since our proposal is different from the basic Holland's genetic algorithms we therefore compare both relating quality of solutions and number of generations needed to reach them. Moreover, we study the bottlenecks present in both algorithms for improving performance.

We consider the classical Holland's genetic algorithm (hereinafter HGA) characterised by sequential behavior.

Comparing PGA against HGA

We have considered three benchmarks (Hill Climbing, Prisoner's Dilemma, and Task Problem), with three sets of seeds, and three population sizes, which results in twenty seven different runs for each genetic algorithm instance (HGA and PGA). All the runs have two hundred generations long, and the string length for the chromosome is sixteen bits. The selection threshold, is 1.0, and the increase for this factor is 0.001. Crossover probability is 60%, and mutation probability is 2%.

Hill Climbing is an optimisation problem with one local maximum and another global in its search space. Progression to the optimum, once in the correct environment, is fast. Prisoner's Dilemma has one global maximum and several local maxima. Progression to the optimum is therefore slower. Convergence in Task problem is slower than previous problems since optimum is near other possible solutions.

For the sake of conciseness, we have shown in the following graphs the mean value of the three runs corresponding to the three different sets of seeds.

Figure.(7) to Figure.(9) show the results for the benchmark Hill Climbing, with a population size of 16, 32, and 64 chromosomes, respectively. Figure.(10) to Figure.(12) correspond to the benchmark Prisoner's Dilemma, and, finally, Figure.(13) to Figure.(15) correspond to the benchmark Task Problem.

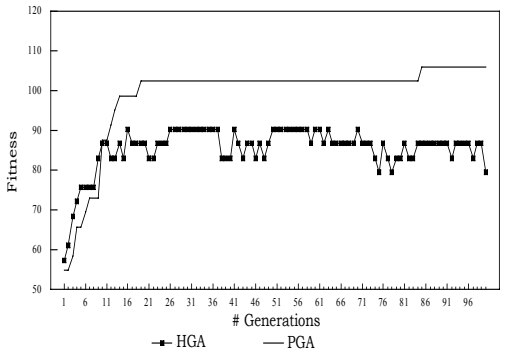

Figure 7: Mean fitness values for Hill Climbing with 16 chromosomes.

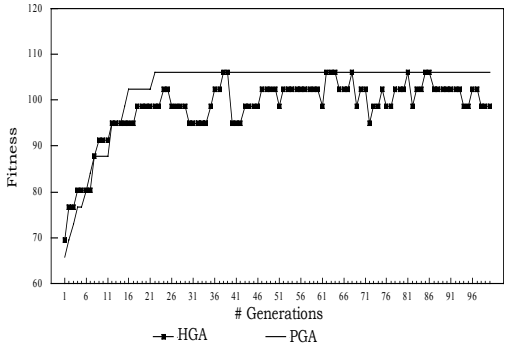

Figure 8: Mean fitness values for Hill Climbing with 32 chromosomes.

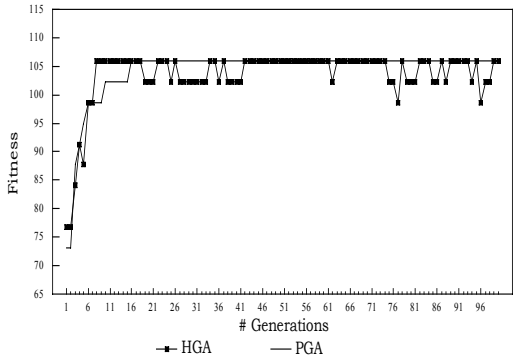

Figure 9: Mean fitness values for Hill Climbing with 64 chromosomes.

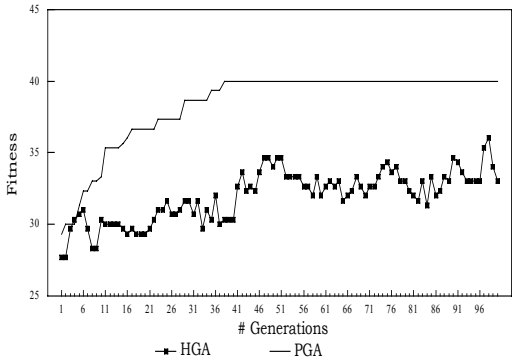

Figure 10: Mean fitness values for Prisoner's Dilemma with 16 chromosomes.

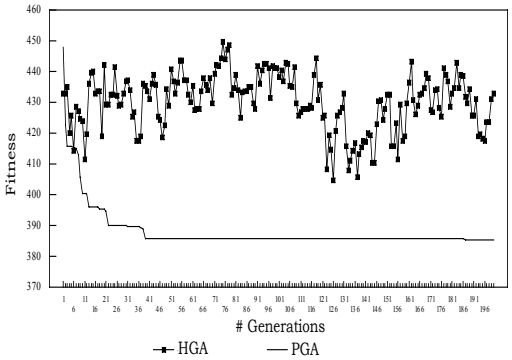

Figure 13: Mean fitness values for Task Problem with 16 chromosomes.

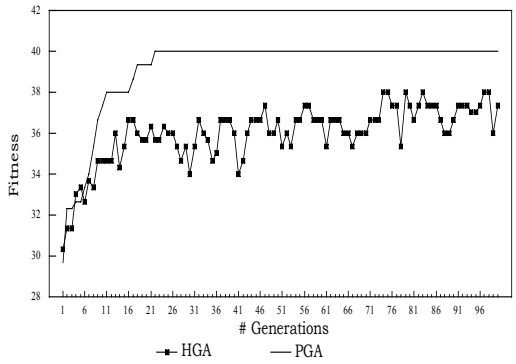

Figure 11: Mean fitness values for Prisoner's Dilemma with 32 chromosomes.

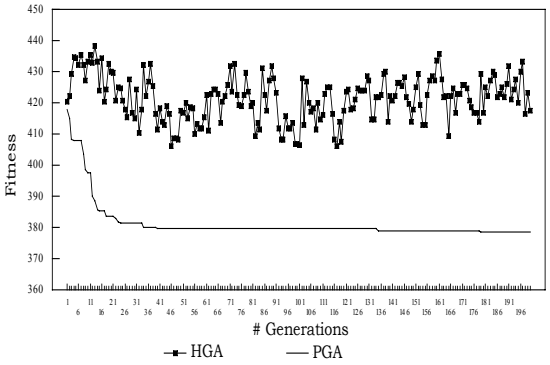

Figure 14: Mean fitness values for Task Problem with 32 chromosomes.

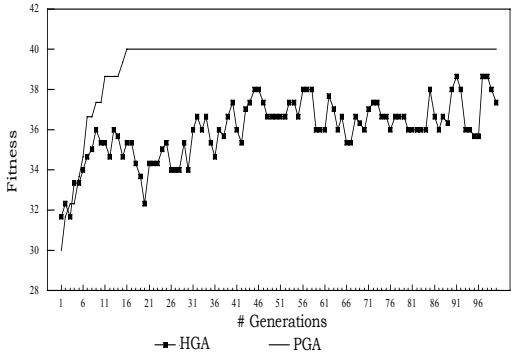

Figure 12: Mean fitness values for Prisoner's Dilemma with 64 chromosomes.

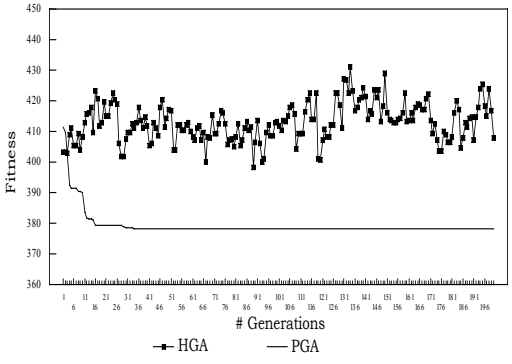

Figure 15: Mean fitness values for Task Problem with 64 chromosomes.

We can observe from graphs above that PGA behaves clearly better than HGA. Convergence speed increases notably when the population size is augmented from 16 to 32 chromosomes. It also increases (in less degree) when we augment it from 32 to 64.

Conclusion

In this work, we have developed a general novel pipelined architecture which is adequate, in particular, for solving the stochastic procedures needed in constraint propagation. This architecture has been shown to deal important speed-ups in early simulation experiments.

Results have been acquainted from C and VHDL analysis implementations. The former give us qualitative data in terms of number of generations needed for achieving a solution, whereas the latter, in addition, can be used as a platform to investigate the implementation of critical regions of the parallel system in programmable logic devices, such as FPGAs. Moreover, by embodying the needed (and approximated) delays we can carry out a high-level simulation regarding speed-up in terms of simulation time. This could be an analysis step in comparing implementations of the parallel system in a multiprocessor architecture and workstation networks.

References

[1] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, E. Teller, "Equation of State Calculations by Fast Computing Machines", Journal of Chemical Physics, Vol 21, pp. 1087-1092, 1953.

[2] Z. Michalewicz, "Genetic algorithms + Data Structures = Evolution Programs". Third Edition, Springer-Verlag, 1996.

[3] F. Glover and M. Laguna, "Tabu Search", Kluwer Academic Publishers, 1996.

[4] T.A. Feo and M.G.C. Resende, "A Probabilistic Heuristic for a Computationally Difficult Set Covering Problem", Operations Research Letters, 8:67-71, 1989.

[5] S. Haykin, "Neural Networks. A Comprehensive Foundation", IEEE Press, 1994.

[6] D.E. Goldberg, "Genetic algorithms in search, optimization, and machine learning", Addison-Wesley, Reading, MA, 1989.

[7] J.H. Holland, "Adaptation in Natural and Artificial Systems", University of Michigan Press, Ann Arbor, Michigan, 1975.

[8] S. D. Scott, A. Samal, and S. Seth. "HGA: A Hardware-Based Genetic Algorithm". Proc. of the 1995 ACM/SIGDA Third International Symposium on Field-Programmable Gate Arrays, pp. 53-59, 1995.

[9] P. Graham and B. Nelson. "Genetic Algorithms in Software and in Hardware - A Performance Analysis of Workstation and Custom Computing Machine Implementations". Dpt. of Electrical and Computer Engineering, Brigham Young University, 1995.

[10] I. M. Bland and G. M. Megson. "Implementing a Generic Systolic Array for Genetic Algorithms". Dpt. of Computer Science, University of Reading, U.K., 1996