A Serial Population Genetic Algorithm for Dynamic Optimization Problems

Lars Zwanepol Klinkmeijer LARS@DROIDS.NL
CKI, Utrecht University, Heidelberglaan 6, 3584 CS, Utrecht, The Netherlands

Edwin de Jong DEJONG@CS.UU.NL
Marco Wiering MARCO@CS.UU.NL
Department of Computer Science, Utrecht University, Padualaan 14, 3508 TB, The Netherlands.

Abstract

The increase in number of papers being published on Evolutionary Algorithms for dynamic optimization problems shows the growing interest in this field. In this paper we introduce a powerful new genetic algorithm, the Serial Population Algorithm (SPA), which uses memory to solve dynamic tasks. We compare SPA to two other algorithms: a Hypermutation- and a Diploid Genetic Algorithm. Our results show that SPA outperforms these algorithms on a recurrent, discontinuous, dynamic optimization task.

1. Introduction

Over the past two decades the field of Evolutionary Algorithms (EA's) has matured and shown itself to be a useful and powerful engineering tool. For tasks where the underlying mechanisms are not well understood or optimization tasks that have many interdependent parameters an EA is often capable of finding good solutions. For the most part, the research on EA's has focused on static, non-changing tasks. This is understandable for two reasons. Firstly, the inner workings of the algorithms, such as takeover time[1] and the accumulation of building blocks, are easier to analyze. Secondly, in the 'old-days' computing power was limited so the problems to be solved needed to remain simple in order for the experiments to remain practical. In recent years the frequency of papers being published concerning non-static problems has increased. Many of those papers will tell you dynamic environments is an interesting research area, because many real world problems are dynamic in nature and the area is largely unexplored.

Goldberg and Smith (1987) are two of the earlier researchers who looked into Genetic Algorithms (GA's) for dynamic optimization problems. They experimented with a diploid GA on a dynamic 0/1 knapsack problem which oscillated between two states. In such algorithms the solutions are encoded by two chromosomes instead of the regular single (haploid) chromosome. Not all genes are used to encode the solution and it is this redundancy in genes that is thought to give the diploid GA its power to tackle the changes in the fitness landscape.

Another well-known GA for dynamic fitness environments is the (Triggered) Hypermutation algorithm by Cobb (1990). She applied this algorithm on a continually changing environment attempting to track an optimal solution. Over the years this algorithm has been shown to perform well (Lewis, Hart & Ritchie, 1998; Morrison & K. de Jong, 2000; Simões & Costa, 2003).

In this paper we introduce a new algorithm that uses subpopulations as a mechanism for storing good solutions. Between changes only one of those subpopulations will be used for evolutionary exploration, leaving the others unchanged. The paper is structured as follows; in section two we discuss related work, in section three we describe our Serial Population Algorithm (SPA), section four describes some dynamic characteristics of environments and we consider which GA's could be preferred for such dynamics, in section five we describe our experimental setup, in section six we discuss our results and we give our conclusions in section seven. The tables with experimental results are shown in the appendix.

2. GA's for Dynamic Environments

In order to apply Genetic Algorithms to dynamic environments we need to make some adjustments to the standard GA used for static environments. One can adopt several kinds of strategies, each having its own advantages and disadvantages. Jürgen Branke (2001) gave a nice overview which we will summarize here accompanied by a more in-depth look into two algorithms; a diploid GA and a Triggered Hypermutation GA.

[1] The time it takes for the population to be filled with the fittest individual only.

2.1 Strategies for Dynamic Environments

In the literature four broad strategies can be distilled:

1. Increase diversity after change
2. Maintaining diversity throughout the run
3. Memory
4. Multiple subpopulations

2.1.1 Increase diversity after change

Triggered Hypermutation (Cobb, 1990) and Variable Local Search (VLS) (Vavak, Jukes & Fogarty, 1998) are two well-known examples of this strategy. Once a change in environment has been detected the mutation-rate will be increased. This can happen either in one dramatic burst followed by a period of decay as with Hypermutation or by gradually increasing the rate as with VLS.

The (Triggered) Hypermutation GA (HMGA) is an elegantly simple algorithm that works well on dynamic environments. For most of the time the algorithm works like a regular GA using mainly crossover and selection to search for a good solution. When a change in the environment is detected the algorithm increases the amount of variation in the population by raising the mutation rate to a very high level. Cobb changed the rate from 0.001, which is very low, up to rates of 0.5. The increase in mutation rate is followed by a period of decay where the rate decreases back to its base rate. Very high mutation bursts like 0.5 are similar to reinitializing the population. Lower levels of mutation bursts retain some parts of the old solution and so are better capable of adapting to smaller changes. Furthermore Morrison and K. de Jong (2000) showed that larger hypermutation bursts track the optimum better when the environmental changes are frequent while lower hypermutation levels perform better when the changes are less frequent.

Cobb tried detecting the change by monitoring the fitness of the best performer in the population. When this value would decline a burst of hypermutation is triggered. Not all changes are detectable this way. Adding peaks in the multiple peaks problem or raising the maximum allowed weight in a knapsack problem may go undetected leaving the algorithm struggling on a suboptimal solution.

Over the years this algorithm has shown itself not only to work well on continuously changing environments but also on discontinuous environments which show large changes to the optima (Lewis, et al., 1998; Morrison & K. de Jong, 2000; Simões & Costa, 2003b). Problems may occur when the algorithm fails to detect the change in the environment or when the change is too large (Grefenstette, 1992).

2.1.2 Maintaining diversity throughout the run

Diversity was already important in GA's used for static environments in order to avoid getting stuck in suboptimal solutions. In dynamic environments this importance is amplified. If a change occurs, your once optimal solution is destined to become suboptimal at best. Diversity maintenance mechanisms such as Fitness Sharing, Random Immigrants (Grefenstette, 1992) and Crowding are common examples of this strategy. Ensuring the population holds no multiple instances of the same solution is another example.

2.1.3 Memory

Two types of memory can be distinguished:

- Explicit memory
- Implicit memory

GA's incorporating explicit memory usually have strategies for storing solutions and reintroducing them on later occasions during the run (Louis & Xu, 1996; Ramsey & Grefenstette, 1993; Eggermont & Lenaerts, 2002). GA's incorporating implicit memory usually incorporate some form of redundancy in their genetic representation. The most common example is using a diploid genetic structure. (Branke, 2001; Calabretta, Calbiati, Nolfi & Parisi, 1996; Lewis et al., 1998; Ng & Wong, 1995)

Applying memory serves two functions: First; it provides diversity by retaining former good solutions which otherwise would have been lost in the selection process and reintroducing (parts of) these solutions on a later occasion. Second; reintroducing former solutions in repetitive environments can enable the algorithm to quickly retrieve the previously encountered optimum.

A diploid GA is different from a regular GA by the fact it has two sets of chromosomes instead of the common single set (haploid). The consequence of this is that two genes compete for the same phenotypic trait in the same individual. In order to solve this dilemma a dominance mapping is devised labeling some genes as dominant and others as recessive. If a dominant gene is paired with a recessive gene, only the former is expressed in the phenotype leaving the recessive gene unexpressed. Dominant genes are thus able to protect less fit recessive genes from being discarded by selection. Formerly fit genes can piggyback ride the fitter dominant genes they are paired with, hopefully coming into expression again when the environment is more favorable. It is this mechanism that is thought to give the GA a form of *implicit* memory.

Apart from this it is also possible for two dominant or two recessive genes to be paired. What happens in this case differs between the dominance mappings used by different researchers. Although over the years many researchers (Callabretta et al., 1996; Hollstein, 1971; Ng & Wong, 1995; Ryan, 1997) have devised their own dominance mappings there is one mapping that is commonly referred to; the triallelic dominance mapping.

The triallelic dominance mapping was first developed by Hollstein (1971) for static environments and made popular by Goldberg and Smith (1987) who first used it for a dynamic environment. The genetic strings use a trinary [0,1,2] representation instead of the regular binary [0,1].

Table 1. A schematic view of the triallelic dominance mapping where the first row and column denote the genetic values.

	0	***1***	***2***
0	0	0	1
1	0	1	1
2	1	1	1

The first row and column in table 1 show the genetic values (alleles) and the rest of the table shows the resulting phenotypic expression. In this mapping there is a clear bias for expressing 1's. Alternative mappings (Ng & Wong, 1995, Ryan, 1997) have been proposed to eliminate this bias. Their representations have four alleles where the probabilities of generating 0's and 1's are equal. Lewis, Hart and Ritchie (1998) showed that a diploid structure alone is not enough for a diploid GA to adapt to changing environments. Frequently switching the values from dominant to recessive and vice versa was needed to give acceptable results.

2.1.4 MULTIPLE SUBPOPULATIONS

The most common way of using subpopulations is to have one part of the population track the best solution present and have other parts of the population search for and track sub-optimal solutions (Branke, Klaussler, Schmidt & Schmeck, 2000; Ursem, 2000).

3. Serial Population Algorithm (SPA)

At the start of an evolutionary run a population is created and initialized randomly. The population is divided into a predefined number of subpopulations. All subpopulations are evaluated and the best[2] subpopulation is selected. Until a change in environment is detected only this subpopulation will be used. When a change has been detected all subpopulations will be evaluated on the new environment and again the best subpopulation will be selected. It is through this serial use of the subpopulations that we hope to create a form of memory. This memory-function will perform optimally when the number of subpopulations is equal to the number of optima.

For the detection of environmental changes we use a system similar to what Eggermont and Lenaerts (2002) used for their algorithm. We store the best individual and its fitness value at the end of each generation. We then evaluate this individual again at the beginning of a new generation. If its fitness value has changed we know a change in environment has occurred and in our case it triggers the algorithm to reevaluate all the subpopulations.

[2] The best subpopulation is either the subpopulation containing the individual with the highest fitness or the subpopulation with the highest *average* fitness. Either qualification seems to work fine.

When SPA has decided on which subpopulation to use, a child population will be created. To generate the child population we repeatedly select two parents from the subpopulation through tournament selection. These parents are recombined using two point crossover with chance *P*c followed by mutation. The resulting two children are placed in the child population.

From the child population the new subpopulation is selected, once again using tournament selection. By using elitist selection we ensure both the best parent and the best child are added to the new subpopulation. This new subpopulation is not allowed to contain any double instances thus ensuring the needed diversity in the population.

```
until maximum number of generations:
    if change detected
        evaluate total population;
        choose best subpopulation;
    end
    else continue with same subpopulation;

    until child population is full:
        select two parents with tournament
        selection;
        perform:
            crossover with chance Pc;
            mutation;
        add kids to child population;
    end
    add the best parent and the best child to new
    subpopulation;
    until new subpopulation is full:
        select child with tournament selection;
        add child to new subpopulation;
        remove any double instances;
    end
    replace the old subpopulation with the new;
end
```

Figure 1. Pseudo code for SPA

4. Characteristics of Dynamic Environments

Dynamic environments come in many different flavors that can have dramatic effects on the functionality of the algorithms used. Therefore it is important one first gets an idea of what the problem looks like and how it behaves before deciding on what algorithm and genetic operators to use. We now give two distinctions by which you can characterize the dynamics and we will discuss their consequences for what type EA to use:

Recurrent	vs.	Non-recurrent
Continuous	vs.	Discontinuous

Recurrent environment have states that are revisited during the evolutionary run. This can happen either periodically/cyclic or a-periodically. In general you could say that such dynamics are well suited for GA's that incorporate some form of memory.

Non-recurrent environments have no states that are revisited or at most merely by accident. Here, applying memory will serve little more function than adding some diversity to the population. GA's that either maintain or introduce diversity seem to have better chances of succeeding.

Continuous environments, in a strict sense, change every timestep by a small margin. They require only small genetic changes to be made to the previous found optimum in order to find the next. Such environments are state dependent functions where the next state is dependent on the previous state. Maintaining diversity throughout the run seems to be a good strategy to handle this problem. If the environment is both *continuous* and *recurrent* the amount of related yet distinct states may be too large for a memory system to be a feasible option (Cobb, 1990).

Discontinuous environments switch from one state to the next in relatively large steps. This may cause problems for some diversity maintenance GA's and diversity introducing GA's when the adjustments are too big (Grefenstette, 1992). Combined with a recurrent environment the amount of states to be found is likely to be small thus a paradise for memory incorporating GA's including SPA.

5. Experimental Setup

For our experiments we compared three different algorithms: a diploid GA using the triallelic dominance mapping, a Hypermutation GA and our Serial Population Algorithm. The environment we used for the bulk of our experiments was a dynamic 0/1 knapsack problem that can be classified as a recurrent, discontinuous optimization problem. Additionally we used a dynamic 0/1 knapsack problem where we swapped two items for a slightly more continuous and non-recurrent environment.

5.1 Dynamic 0/1 Knapsack Problem

The Dynamic 0/1 Knapsack Problem is an oscillatory version of the standard 0/1 Knapsack Problem. The task is to fill a 'knapsack' with a subset of items. Each item has both a *weight* and a *value*. The aim is to maximize the value of the content of the knapsack without exceeding the maximum allowed weight. This weight constraint is enforced by a penalty function identical to the one used in Smith and Goldberg (1992):

$$P = C(\Delta W)^2$$

Where:

P = the Penalty on the fitness value.

ΔW = the overweight of the individual.

C = 20.

Negative scores will be rated as zero.

At the start of the experiment we generate several sets of items. After a predefined number of generations we pick a different set of items thus giving it its dynamic character. This method of changing the knapsack problem is different from what most researchers do, who change the maximum allowed weight (Goldberg & Smith, 1987; Smith & Goldberg, 1992; Lewis, et al., 1998; Simões & Costa, 2003b). Raising the maximum allowed weight has the disadvantage that the change may not be detected, as stated earlier. In our case we can be 100% certain of detecting a change.

In order to investigate SPA's basic characteristics we performed several different experiments, each lasting 2000 generations. For the bulk of our experiments we used three different set *sizes* containing 17, 50 and 150 items each. With each set size we altered two conditions:

- The *amount* of sets used; using 2 and 5 sets of items each containing weights and values.
- The *duration* of the *stationary period*; P = 10, P = 25 or P = 50 generations.

Additionally we performed experiments using only one set with size 50 where we generated a change by swapping two items in the sets. This created a genotypic difference with a hamming distance of size 2 while leaving the optimal knapsack value unchanged. Our aim was to create a somewhat more continuous environment although it is not continuous in the strict sense for it still changes periodically.

5.2 The Algorithms

For each of the three algorithms we use the same genetic operators as much as possible in order to keep things as equal as possible. The only differences are: a triallelic encoding for the diploid algorithm, a hypermutation phase for the Hypermutation algorithm and the use of serial subpopulations for SPA. The parts that are equal are: child- and parent population sizes, tournament sizes, crossover probability, mutation rate, and not allowing multiple instances in the parent population. In the case of five sets with size 50 we also ran a basic GA that used only these operators.

5.2.1 General Settings

We set the following parameters for all algorithms:

- (sub) population size = 50
- child population size = 150
- tournament size:
  - mating selection = 4
  - replacement selection = 8
- crossover probability Pc = 0.9
- mutation rate = 1/setsize

This means that if SPA uses five subpopulations, its total population size will be 250. This appears to give SPA a major advantage over the other algorithms but in reality this is limited.

The following three equations give the average number of evaluations for each algorithm:

Eq. 1: $\frac{(S-1)*I}{P}+1+I+C=E$ SPA

Eq. 2: $1+I+C=E$ HMGA

Eq. 3: $I+C=E$ Diploid GA

Where:

S = number of subpopulations used in SPA.

I = number of individuals in each (sub-) population.

C = number of individuals in each child population.

P = duration of stationary period, measured in generations.

E = average number of evaluations in each generation.

After each detected change SPA's entire population will be evaluated. On average this results in Eq. 4 evaluations per generation. The detection system itself accounts for one extra evaluation per generation and evaluating the parent- and child populations are equal for all GA's (Eq. 3). The Hypermutation Algorithm does not have the subpopulations but it does have the detection system, resulting in Eq. 4 fewer evaluations per generation than SPA.

Eq. 4: $\frac{(S-1)*I}{P}$

The Diploid GA has no detection system nor does it use the subpopulations resulting in eq. 5 fewer evaluations per generation then SPA.

Eq. 5: $\frac{(S-1)*I}{P}+1$

This means that if SPA uses five subpopulations and the stationary period is fifty generations, it will have four more evaluations than HMGA and five more than the Diploid GA. The number of evaluations per generation will go up either by increasing the number of subpopulations or by increasing the frequency of environmental change. To counterbalance this advantage of SPA we increased the populations of the Hypermutation GA and the Diploid GA by the appropriate numbers. So for this example HMGA would have 54 individuals and the Diploid GA 55.

5.2.2 The Hypermutation Algorithm

We altered the Hypermutation algorithm slightly compared to what is common. Normally, between mutation bursts, HMGA uses a very low base mutation rate and depends mostly on crossover and a large population size to find solutions. In our experiments we used the base rate of 1/L where L is the length of the chromosome. Because our population is smaller than normally used in hypermutation experiments the extra mutation is needed to compensate the loss of variation due to the population size. The mutation burst is set to be roughly 35%. This is comparable to the burst size used in Lewis et al. (1998) and to the theory that high frequencies require high mutation rates (Morrison, K. de Jong, 2000). The burst is followed by a period of linear decay. Two generations after the initial burst the mutation rate is back on the base rate. Also the detection system is different than the one used by Cobb. Our detection system has a 100% chance of detecting a change in environment. This is in part caused by the way we change our environment. If we would have altered the maximal allowed weight there would have been a chance that the change would go unnoticed. For these experiments we used the same detection system as SPA that we described earlier.

5.2.3 The Diploid Genetic Algorithm

The diploid genetic algorithm uses Hollstein's triallelic dominance scheme as described earlier. Apart from this and the fact it doesn't use any subpopulations the algorithm is the same as SPA.

5.3 The Performance Measures

We use two criteria to measure the performances of the algorithms; Accuracy and Adaptability as described by Simões & Costa (2003a & 2003b) but with a slight alteration. Accuracy measures the difference between the optimal value of that period and the best individual in the last generation before the change. We altered this slightly by taking this difference as a percentage of the optimum. This is especially useful when comparing results of tests with large differences in set size what can result in large differences in optimal values. Adaptability is similar to what is commonly known as the mean fitness error. We measure the difference between the best individual of each generation with the optimum value of that period. It gives us an indication of the speed of recovery of the algorithm. These two measurements should be as close to zero as possible. The values that are shown in the tables are the averages over 10 evolutionary runs per experiment.

6. Results

The details of the results are given in the appendix.

6.1 Discontinuous recurrent dynamics

The experiments show that the use of serial subpopulations has the effect that each subpopulation tends to converge toward a single optimum. This enables SPA to quickly regain the former solution in recurrent problems. If the subpopulation had previously found the optimum and it has not been used to search for a different optimum it is even capable of retrieving the old optimum in the first generation after the change. This results in an adaptability score of zero in the later parts of the evolutionary run. Of course this is helped by the fact that the number of subpopulations is set equal to the number of optima.

Considering both the accuracy and adaptability HMGA performs equally well on environments with five optima as on those with two optima although on the latter it does slightly better. SPA on the other hand does show a significant difference. This is caused by the fact that when using two subpopulations for two optima the subpopulations quickly develop a preference for one of the optima. With more subpopulations this development of preference tends to take longer as sometimes a subpopulation is used for more than one optimum. Given sufficient time and revisitations to the optima this double use of a subpopulation ceases.

Whereas HMGA doesn’t lag too much behind when we consider the accuracy measurement, SPA clearly is king when we look at the adaptability.

Some preliminary explorations using either more or fewer subpopulations showed that in the former case the accuracy and adaptability do not suffer although there are a lot of useless extra evaluations after each change. In the latter case some subpopulations are used for more than one optimum whereas others still manage to converge to a single solution thus retaining some of its memory.

The Diploid GA clearly is the worst performer of all tested algorithms, both with the accuracy and the adaptability measure. Even when we compared it to a basic GA that was similar to SPA using only one undivided population, it still performed worse. This is most likely caused by the slow convergence of the diploid GA and by the fact that even when all individuals were unique it still was able to converge to a single phenotype, losing much of its selective pressures.

6.2 Small changes, non-recurrent dynamics

Using only one set of items for the knapsack and changing them around between the stationary periods result in small changes. Because in SPA subpopulations tend to converge to one specific set it now only uses one subpopulation. Any additional subpopulations will not be used and become a burden because of the extra evaluations after each detected change. HMGA now does a lot better and comes very close to the accuracy and adaptability performances of SPA. HMGA may very well exceed SPA’s performances in this case when properly tuned.

7. Discussion

7.1 Strengths and weaknesses of SPA

Strengths:

Adaptability is clearly the strongest point of SPA. The more an optimum is revisited the more powerful SPA will become. The dedication of a single subpopulation to an optimum also causes SPA to increase its accuracy because former search results are not lost.

Weaknesses:

- Higher frequency of change causes an increase in the average number of evaluations per generation.

- Larger numbers of subpopulations cause an increase in the average number of evaluations per generation.

Both these weaknesses may be lessened by sampling the subpopulations instead of evaluating the entire population.

- The efficiency of SPA is dependent on whether the number of optima and subpopulations match.

This may be resolved by introducing more complex rules, such as starting out with one subpopulation and introducing more subpopulations when this is required by the environment.

7.2 Future Work

SPA works extremely well on recurrent, discontinuous optimizations tasks. Given a number of subpopulations greater or equal to the number of optima it makes near perfect use of memory. Developing a system for SPA that does not require knowing the amount of optima, for instance by gradually introducing extra subpopulations, may make SPA a more widely applicable algorithm.

References

Branke, J. (2003). Evolutionary approaches to dynamic optimization problems - introduction and recent trends. In J. Branke, editor, *GECCO Workshop on Evolutionary Algorithms for Dynamic Optimization Problems*, pages 2-4.

Branke, J., Kaußler, T. & Schmidt, C. & Schmeck, H. (2000). A multi-population approach to dynamic optimization problems. In *Adaptive Computing in Design and Manufacturing 2000*. Springer.

Cobb, H. G. (1990). An investigation into the use of hypermutation as an adaptive operator in genetic algorithms having continuous, time-dependent nonstationary environments. Technical Report AIC-90-001, Naval Research Laboratory, Washington, USA.

Eggermont, J. & Lenaerts, T. (2002) Dynamic Optimization using Evolutionary Algorithms with a Case-based Memory. In *Proceedings of the 14th Belgium Netherlands Artificial Intelligence Conference (BNAIC'02)*

Goldberg, D. E. & Smith, R. E. (1987). Nonstationary function optimization using genetic algorithms with dominance and diploidy. In J. J. Grefenstette, editor, *International Conference on Genetic Algorithms*, pages 59-68. Lawrence Erlbaum Associates.

Grefenstette, J. J. (1992). Genetic algorithms for changing environments. In R. Maenner and B. Manderick, editors, *Parallel Problem Solving from Nature 2*, pages 137-144. North Holland.

Lewis, J., Hart, E. & Ritchie. G. (1998).A comparison of dominance mechanisms and simple mutation on non-stationary problems. In A. E. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature,* number 1498 in LNCS, pages 139-148. Springer.

Morrison, R. W. & De Jong, K. A. (2000). Triggered hypermutation revisited. In *Congress on Evolutionary Computation*, pages 1025-1032.

Ng, K. P. & Wong, K. C. (1995). A new diploid scheme and dominance change mechanism for non-stationary function optimization. In *Proceedings of the Sixth International Conference on Genetic Algorithms, Algorithms*, pages 159-166. Morgan Kaufmann.

Simões, A. & Costa, E. (2003a). An immune system-based genetic algorithm to deal with dynamic environments: Diversity and memory. In D. W. Pearson, N. C. Steele, and R. Albrecht, editors, *Proceedings of the Sixth international conference on neural networks and genetic algorithms (ICANNGA03)*, pages 168-174. Springer.

Simões, A. & Costa, E. (2003b). A comparative study using genetic algorithms to deal with dynamic environments. In D. W. Pearson, N. C. Steele, and R. Albrecht, editors, *Proceedings of the sixth international conference on neural networks and genetic algorithms (ICANNGA03)*, pages 203-209. Springer.Smith, R. E. & Goldberg, D. E. (1992). Diploidy and Dominance in Artificial Genetic Search. In *Complex Systems*, Vol. 6, pages. 251-285.

Ursem, R. K. (2000). Multinational GA optimization techniques in dynamic environments. In D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee, and H.-G. Beyer, editors, *Genetic and Evolutionary Computation Conference,* pages 19-26. Morgan Kaufmann.

Vavak, F., Jukes, K. A. & Fogarty, T. C. (1998). Performance of a genetic algorithm with variable local search range relative to frequency for the environmental changes. In Koza et al., editor, *International Conference on Genetic Programming*. Morgan Kaufmann.

Appendix

Accuracy:

Table 2. Accuracy (%) on 5 sets of size 17

	P = 10	P = 25	P = 50
Diploid	0.8515	0.5484	0.3646
HMGA	0.1087	0.0010	0.0000
SPA	0.0222	0.0011	0.0355

Table 3. Accuracy (%) on 2 sets of size 17

	P = 10	P = 25	P = 50
Diploid	1.1878	0.3108	0.1261
HMGA	0.0128	0	0
SPA	0.0021	0	0.009

Adaptability:

Table 9. Adaptability (%) on 5 sets of size 17

	P = 10	P = 25	P = 50
Diploid	11.6612	5.8761	3.4162
HMGA	4.6372	1.8022	1.0003
SPA	0.3821	0.2250	0.2233

Table 10. Adaptability (%) on 2 sets of size 17

	P = 10	P = 25	P = 50
Diploid	9.3527	5.2123	2.9160
HMGA	3.7169	1.5001	0.7430
SPA	0.0586	0.0523	0.0785

Table 4. Accuracy (%) on 5 sets of size 50

	P = 10	P = 25	P = 50
Diploid	7.988	1.8547	0.7381
HMGA	3.65	0.2609	0.0151
SPA	0.305	0.045	0.0059
Basic GA	5.701	0.40087	0.0354

Table 5. Accuracy (%) on 2 sets of size 50

	P = 10	P = 25	P = 50
Diploid	6.4238	2.0139	0.8077
HMGA	4.0406	0.2884	0.0151
SPA	0.0565	0.0122	0.0015

Table 6. Accuracy (%) on 5 sets of size 150

	P = 10	P = 25	P = 50
Diploid	21.414	8.333	3.440
HMGA	9.985	3.985	2.020
SPA	1.789	1.194	0.688

Table 7. Accuracy (%) on 2 sets of size 150

2 optima	P = 10	P = 25	P = 50
Diploid	15.1685	4.1251	2.1364
HMGA	9.6863	3.5759	1.8728
SPA	0.6822	0.4651	0.3254

Table 8. Accuracy (%) for ***small changes*** on size 50

	P = 10	P = 25	P = 50
HMGA	0.6907	0.0347	0.0048
SPA	0.3108	0.0246	0.0021

Table 11. Adaptability (%) on 5 sets of size 50

	P = 10	P = 25	P = 50
Diploid	19.16	15.56	9.37
HMGA	13.56	6.73	3.45
SPA	0.73	0.60	0.48
basic GA	15.26	5.29	5.28

Table 12. Adaptability (%) on 2 sets of size 50

	P = 10	P = 25	P = 50
Diploid	19.33	16.91	10.96
HMGA	15.28	7.30	3.71
SPA	0.15	0.16	0.17

Table 13. Adaptability (%) on 5 sets of size 150

	P = 10	P = 25	P = 50
Diploid	25.6305	26.5077	23.4982
HMGA	13.5355	13.8279	8.8153
SPA	2.0354	2.1057	1.8599

Table 14. Adaptability (%) on 2 sets of size 150

	P = 10	P = 25	P = 50
Diploid	18.4185	18.5358	19.0245
HMGA	13.7836	13.9357	8.8824
SPA	0.7521	0.7691	0.7160

Table 15. Adaptability (%) for ***small changes*** on size 50

	P = 10	P = 25	P = 50
HMGA	1.3635	0.5520	0.3275
SPA	0.7235	0.3234	0.1813