The Problem with a Self-Adaptative Mutation Rate in Some Environments

A Case Study using the Shaky Ladder Hyperplane-Defined Functions

William Rand and Rick Riolo
Center for the Study of Complex Systems
University of Michigan
4485 Randall Lab
Ann Arbor, MI, USA, 48109-1120
wrand@umich.edu

ABSTRACT

Dynamic environments have periods of quiescence and periods of change. In periods of quiescence a Genetic Algorithm (GA) should (optimally) exploit good individuals while in periods of change the GA should (optimally) explore new solutions. Self-adaptation is a mechanism which allows individuals in the GA to choose their own mutation rate, and thus allows the GA to control when it explores new solutions or exploits old ones. We examine the use of this mechanism on a recently devised dynamic test suite, the Shaky Ladder Hyperplane-Defined Functions (sl-hdf's). This test suite can generate random problems with similar levels of difficulty and provides a platform allowing systematic controlled observations of the GA in dynamic environments. We show that in a variety of circumstances self-adaptation fails to allow the GA to perform better on this test suite than fixed mutation, even when the environment is static. We also show that mutation is beneficial throughout the run of a GA, and that seeding a population with known good genetic material is not always beneficial to the results. We provide explanations for these observations, with particular emphasis on comparing our results to other results [2] which have shown the GA to work in static environments. We conclude by giving suggestions as to how to change the simple GA to solve these problems.

Categories and Subject Descriptors: F.2.m [Analysis of Algorithms] Misc. I.2.8 [Artificial Intelligence] Search

General Terms: Algorithms, Theory

Keywords: Self-Adaptation, Dynamic Environments, Hyperplane-Defined Functions, Genetic Algorithms

1. INTRODUCTION

Dynamic environments have periods of quiescence and periods of change. In periods of quiescence an algorithm trying to solve a problem should exploit good individuals while in periods of change the algorithm should explore new solutions. The Genetic Algorithm (GA) is based on evolution, which works in an inherently dynamic environment and thus the GA is a natural choice for dynamic environments. The GA can exploit good individuals by allowing them to reproduce accurately and in quantity which keeps a constant supply of good building blocks around. The GA can explore by creating new individuals that are very different from its current population.

In fact, the GA has been shown to work successfully in many dynamic environments [5] [6]. However, in a GA the balance between exploration and exploitation is controlled by a variety of parameters, most noticeably the level of mutation and in a simple GA this parameter is constant. Therefore adding a mechanism to the simple GA to allow it to change the level of mutation might be desirable. In fact, Holland originally suggested this very concept with regard to static environments, since even in static cases it is often beneficial to change the mutation rate [11]. There are many ways to change mutation rates that have been explored in dynamic environments [7] [10] [22]. However in this paper we choose to look at a mechanism called *self-adaptation*. In the past, this mechanism has been investigated in GAs on static environments [2] [19] but has primarily been examined in dynamic environments in evolutionary programming and evolutionary strategies [3] [1].

In order to examine the behavior of the GA we present a family of test functions that are similar to the dynamic bit matching functions utilized by Stanhope and Daida [20] among others. The difference between this new test suite and other dynamic test functions is that the underlying representation of this suite is schemata [11]. By utilizing functions that reflect the way the GA searches, the performance of the GA can be easily observed. This test suite is a subset of the test functions created by John Holland, the hyperplane-defined functions (hdfs) [12] which were extensions of the Royal Road functions developed by Mitchell et al [16]. We have further extended the hdfs to dynamic environments and we call this test suite the Shaky Ladder Hyperplane-Defined Functions (sl-hdf's) [17].

In the rest of this paper, we examine the mechanism of self-adaptation, then explain the development of the sl-hdf's. Finally we present some simple experiments using self-adaptation on the sl-hdf's and discuss the results.

2. SELF-ADAPTATION

Even within static environments, one of the hardest parameters to set accurately in a GA is the mutation rate. On the one hand mutation introduces novel bit patterns into the population. In fact in the standard GA it is the only source of novelty in the alleles. However, mutation also can have a deleterious effect because it can destroy good alleles that are present in the population.

In a dynamic environment, this problem is compounded. In the ideal case, when the environment is undergoing change the GA should rapidly adapt to the change, which requires it to have a high mutation rate. However in periods of quiescence the GA should exploit good alleles that it has already found and thus it should have a low mutation rate, so that those alleles are passed on to future generations. A high mutation rate at this time would result in the destruction of good alleles and would be harmful to the overall search.

One possible solution to this dilemma is to allow the GA to adapt its own mutation rate during the run. Mechanisms to allow this have been explored in the past. One such mechanism is *hypermutation*, originally discussed by Cobb [7]. In this mechanism, the mutation rate is increased dramatically if there is a drop in the time-averaged performance of the GA. However, such a mechanism limits the ability of the GA to control its mutation rate and requires it to select between predefined rates.

Additionally, Vavak has investigated the use of *Variable Local Search* (VLS) in a steady state GA [22]. When the time-averaged best performance of the population decreases it applies an operator which increases the range from which new individuals are drawn. It will continue to increase this range if performance continues to decrease eventually drawing from the whole search space.

Both hypermutation and VLS require a central controlling function. Something monitors the population of the GA to determine if the best individuals are failing. In natural systems when mutation rates change there is no cenrtal controlling function; instead individuals have higher mutation rates because natural selection results in those higher mutation rates being propagated. Mutation rates are handled on an individual level and it has been shown that at least in some situations the mutation rates of natural systems move toward a positive optimal value [14]. Even if having mutation rates controlled by individual performance is not a better mechanism for the GA, the fact that it exists in biological systems makes it interesting to study.

Another mechanism explored in GAs is *random immigrants*, originally discussed by Grefenstette [10]. This technique introduces a group of randomly generated individuals every generation. This technique does not allow the GA to really control the amount of exploration or exploitation going on but rather forces the GA explore new individuals every generation. In some environments this might be useful, but in slowly changing environments this would not always be productive, since it could result in the loss of good alleles in the population. Even if the random immigrants have a poor fitness there is a low but positive probability that they could be involved in a crossover that would destroy combinations of good alleles.

In evolutionary programming and evolutionary strategies the only operator is mutation and hence it makes sense to look to these fields to see what kind of operators they have used in dynamic environments. One of the mechanisms that has proven to be the successful is self-adaptation [3] [1]. This mechanism allows the mutation rate itself to be adapted in time as part of the individual being examined.

Bäck [2] investigated self-adaptation on the GA but in a static environment and showed it to be quite successful on some standard fitness functions. Besides using a dynamic fitness function, and a different form of the standard GA, the self-adaptation mechanism we describe below is similar to Bäck's mechanism for one mutation rate.

We chose this particular mechanism of individual self-adaptation because it seems like an excellent solution to maintaining an appropriate level of mutation in dynamic environments. First it does not use knowledge of the whole population and thus different individuals can be exploring different parts of the search space. Second, in the general case it allows the GA to choose any number of different mutation rates without any prior knowledge of the environment. Finally, it allows the GA to balance exploration and exploitation as opposed to forcing the GA to favor one over the other. All of these factors together allow the GA to be more adaptive to dynamic environments.

There have been other investigations into the use of self-adaptation in GAs, including the thesis work of Smith [19] who investigated self-adaptation in steady state GAs, using a similiar but different mechanism that involved local search using a separate GA. Bäck's mechanism appears to be a first-order approximation of mutation rate adaptation in haploid organisms, and thus we chose to investigate it. However, before we go into detail about how our mechanism works we first describe the environment we will be utilizing to test the mechanism.

3. SHAKY LADDER FUNCTIONS

The test functions that we will be utilizing to explore the GA in dynamic environments are a subset of the hdfs [12]. Holland created these functions in part to meet criteria developed by Whitley [23]. The hdfs are designed to represent the way the GA searches by combining building blocks, hence they are appropriate for understanding the operation of the GA. We begin by describing these functions, then we describe a subset called the building block hdfs (bb-hdfs). Finally, we describe the shaky ladder hdfs (sl-hdfs).

3.1 The Hyperplane-Defined Functions

Holland's hdfs are defined over the set of all strings (usually binary) of a given length n. The fitness of a string is determined by the *schemata* contained by the string. A *schema* is a string defined over the alphabet of the original string plus the character, $*$, and is also of length n. The $*$ represents a *wildcard* that will match either a 1 or a 0. Any position in the string which is not a wildcard is a *defining locus* or *defining bit*. The *length* of a schema $l(s)$ is the distance between the first and last defining loci, and the *order* $o(s)$ is the number of defining loci. A string x *matches* (or *contains*) a schema s if for every position from 1 to n, the character is the wildcard or matches the character at that same position in x. Each schema s is assigned a fitness contribution, or utility, $u(s)$ that is a real number. Thus we define the hdf fitness, $f(x)$, as the sum of the fitness contributions of all of the schemata x matches [12].

3.2 Building Block hdfs

Holland stated that the most interesting hdfs are those built from ground level schemata. We select a group of *elementary* (or *base* or *first-level*) schemata which have a short length relative to n and a low order. We then combine pairs of these schemata to create *second-level* schemata, and combine the second-level schemata and so on, repeating this process until we generate a schema with length close to n. We call these schemata combinants the *intermediate* schemata, We place the intermediate and base schemata within a set B and associate a positive fitness contribution with them. Members of the set B are sometimes called the *building block* schemata, since they are used to "build" progress toward the maximal fitness. We also generate a set P of schemata with negative fitness contributions called *potholes*, which are local depressions in the fitness landscape. The potholes are created by using one elementary schema as a basis and adding some of the defining bits from a second elementary schema. We will refer to the set of hdfs that are constructed from the sets B and P as the *building block hdfs* or *bb-hdfs*.

The problem with the bb-hdfs in the general case is that the optimal set of strings is not easily known given the sets B and P, and thus the absolute performance can not easily be measured. Moreover, there is no way to take one bb-hdf and create another that is similar to it, which would be useful when exploring dynamic environments.

3.3 Shaky Ladder hdfs

In this section we describe three conditions that restrict the set of all bb-hdfs to a set that does not have the two difficulties described in the previous subsection.

The first condition is the *Unique Position Condition* (UPC). It requires that all elementary schemata contain no conflicting bits. For instance if both schemata s_1 and s_2 have a defining bit at position i, they must specify the same value.

The second condition we call the *Unified Solution Condition* (USC). This condition guarantees that all of the specified bits in the positive-valued elementary level schemata must be present in the highest level schema. This condition also guarantees that all intermediate schemata are a *composition* of lower level schemata. A *composition* of two schemata is simply a new schema which contains all of the defining loci of both schemata[1]. The USC means (a) if bit n is specified by s_1 then it also must be specified in the highest level schema s_h, and (b) there can be only one highest level schema.

The third condition is the *Limited Pothole Cost Condition* (LPCC), which states that the fitness contribution of any pothole plus the sum of the fitness contributions of all the building blocks in conflict with that pothole must be greater than zero. A pothole, p, is in *conflict* with a building block, b, if both p and b specify a defining locus with the same value. Thus potholes are temporary barriers to the search, but a string is rewarded if it matches all of the building blocks in conflict with a pothole.

These three conditions guarantee that any string which matches the highest level schema must be a string with optimal fitness. By knowing the optimal set of strings we solve one of the problems with Holland's original hdfs.

[1]Given the UPC we do not have to worry about conflicting loci.

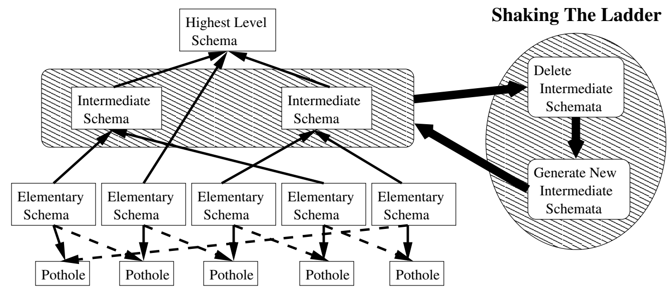

Figure 1: Illustration of Shaky Ladder Construction and the Process of Shaking the Ladder

3.4 The Algorithm

This section describes the algorithm that we used to generate examples of the sl-hdf problems. This is one algorithm out of many that could be used to generate instances of sl-hdf problems, but this algorithm guarrentees that the problem instances generated meet the three conditions specified above. We focus here on an approach which is as simple as possible, while following the general guidelines outlined by Holland [12].

The algorithm manipulates four major parts: elementary schemata, highest level schema, potholes, and intermediate schemata. Of course we also need to describe how we "shake the ladder" by changing the intermediate schemata. The process described below is illustrated pictorially in Figure 1 and more thoroughly described in previous work [17].

The Elementary Schemata: The elementary schemata must be created in such a way as to fulfill the UPC specified above. Holland also recommends that the elementary schemata have short lengths and low orders. Thus we need to be able to create random schemata with length l and order o that meet the UPC. Holland recommends a length equivalent to 1/10 the length of the string and an order of roughly 8. Due to the UPC it is fairly difficult to meet both of these additional requirements for an arbitrary set of schemata. Thus to simplify matters in all of the experiments in this paper we set the order of elementary schemata to 8 and did not worry about the schema length.

To begin with we create a random schema with order o (8 in this paper). We do this by choosing o random indices in the schema and with equal probability setting them to either a 1 or 0. We then create another list of indices which acts as a cumulative record of which loci can be assigned 1's, by examining the previous schema and adding to the list any places that are still wildcards or that are defined as ones. In a similar way we create a list for zeroes. We then shuffle these lists. To create subsequent schemata we flip a coin o times and if it is a 1 (0) we pull an index off the list for ones (zeroes), making sure we do not reuse any index we have already set. After we are done with this we add this schema to the list of schemata, update the cumulative list for 1's and 0's, and repeat the process for the next schema. When we are done we have a list of what we call *non-conflicting* schemata since they all meet the UPC. We assign a fitness contribution of 2 to each of them.

Highest Level Schemata: Given the elementary schemata, it is a simple matter to create the highest level schema since

Population Size	1000
Crossover Rate	0.7
Generations	1800
String Length	500
Selection Type	Tournament, size 3
Number of Elementary Schemata	50
Number of Runs	30

Table 1: Parameters of the GA and sl-hdf

Exp.	μ *Type & l_μ*	μ, $[\mu_{Min}, \mu_{Max}]$	*Seeded*	*Cross*
1	Fixed	0.001	No	Yes
2	Fixed	0.0005	No	Yes
3	Fixed	0.0001	No	Yes
4	Self, 10	$[0.0, 1.0]$	No	Yes
5	Self, 15	$[0.0001, 0.1]$	No	Yes
6	Self, 15	$[0.0001, 0.1]$	Yes	Yes
7	Self, 15	$[0.0001, 0.1]$	Yes	No

Table 2: Parameters of Experiments

according to the USC it contains all of elementary schemata. We examine each index in the string and see if any of the elementary schemata have it defined; if so we set the same location in the highest level schema equal to the same defining bit. If none of the elementary schemata have the bit set we leave the location as a wildcard in the highest level schema. We assign a fitness contribution of 3 to this schema.

Potholes: The potholes are created in a way inspired by the bb-hdfs. We simply go through the list of elementary schemata (ordered randomly) and use each one as a primary schema and its neighbor in the list as a supplementary schema, which creates one pothole for every elementary schema. We build a pothole by including all of the defined bits from the primary schema in the pothole, and then for each defining bit in the supplementary schema, we copy that bit value into the pothole with 0.5 probability. We assign a fitness contribution of -1 to all potholes. Note that it is possible to create a pothole that is simply the union of two elementary schemata, or a pothole that is exactly the same as the primary schema. That means that when examining the LPCC, we include the fitness contribution of one elementary schema and the highest level schema, and since $(2 + 3 + -1) > 0$ the LPCC is satified.

Intermediate Schemata: To create the intermediate schemata, we first decide how many schemata, *nNextLevel*, should be at the next level by taking the number of schemata at the previous level and dividing by two (rounding down). We draw two schemata (without replacement) from a shuffled list of schemata at the current level, and create a composition of these schemata which we add to the next level of schemata. We repeat this process *nNextLevel* times. We continue to create new levels until $nNextLevel \leq 1$

Shaking The Ladder: The four mechanisms described above allow us to generate hdfs that are similar. Once a set of elementary schemata have been established we already know the highest level schema. If we hold the elementary and highest level schemata constant, we can generate new hdfs by creating new intermediate schemata. To "shake the ladder" we first delete all of the previous intermediate schemata, and then we create new ones by repeating the process described in the previous paragraph. Since at each level we randomly select partners to create the combinants at the next level, we should get different intermediate schemata most of the time. For clarity we call a set of hdfs with the same elementary schemata and highest-level schema an *sl-hdf equivalence set.* The sl-hdf's fulfill the conditions that Holland [12] and Whitley [23] set out, but also have the benefit of an easily identifiable optimal fitness.

4. THE EXPERIMENTS

The basic setup for our experiments is a simple GA using the sl-hdf as its fitness function. The base GA presented here uses one-point crossover, per bit mutation, full population replacement, and is similar to the one described by Mitchell [15]. For the GA and sl-hdf we use the parameters in Table 1. In all experiments, we examine a control variable, t_δ, which specifies the number of generations before we change the environment. Every t_δ generations we shake the ladder and switch to another sl-hdf in the same equivalence set. We set $t_\delta = (1, 5, 10, 25, 50, 100, 900, 1801)$. In the last case the time between changes exceeds the run of the GA and thus it provides a benchmark of the performance of the GA on a static environment. We carried out 7 experiments which are described in Table 2.

The first column of this table presents the experiment number. The second column describes the type of mutation. "Fixed" is the traditional per-bit mutation with a fixed mutation rate as specified in the 3rd column. "Self" is the self-adaptive mechanism. In these experiments, self-adaptation is accomplished by adding a string of bits, called *mutation bits*, of length l_μ, to each individual in the population, and using these bits to determine the mutation rate, μ, for that individual. At the end of each generation, each individual is asked to mutate its own string. The individual accomplishes this by reading the mutation bits, and translating these bits into a decimal numerator. It then divides by the maximum possible numerator $(2^{l_\mu} - 1)$, to determine the mutation rate. This results in a value between 0 and 1, called μ_{Raw}. This number is then scaled by the maximum, μ_{Max}, and minimum μ_{Min}, mutation rates, to determine the final mutation rate, μ, i.e. $\mu = \{\mu_{Raw} * \{\mu_{Max} - \mu_{Min}\}\} + \mu_{Min}$. These scaling values are specified in the third column. μ is then used as a probability to determine whether each bit in the individual is changed, including the mutation bits themselves. This mechanism is meant to imitate that presented by Bäck [2] even though some of the details of the rest of the GA differ from Bäck's implementation.

Normally initial μ values were uniformly distributed. However, occasionally we seeded the population with a known good μ. In this case we initially fill the mutation bits with values that were as close as the binary representation could get to 0.001 which was found to be a good value in the fixed mutation case. Thus $Seeded = Yes$ indicates that we did this.

The final parameter we manipulated determines whether or not crossover was turned on at all. When crossover was turned on, individuals generated from crossover constituted 70% of the new population of each generation, but when $Crossover = No$, no crossover of individuals was done, thus individuals only changed through mutation.

The first three experiments used fixed mutation and are present as a baseline of comparison. The first of these experiments was explored in more detail in previous work [17]. In Experiment 1 (Fixed 0.001) $\mu = 0.001$, in Experiment 2 (Fixed 0.0005) $\mu = 0.0005$, and in Experiment 3 (Fixed 0.0001) $\mu = 0.0001$.

Experiment 4 (Self 0-1) is the first one to include the self-adaptation mechanism. We set $\mu_{Min} = 0.0$ and $\mu_{Max} = 1.0$. This allowed the μ to take any value possible.

In Experiment 5 (Self 0.0001-0.1), we decided to scale μ as described above, and we restricted it to $[0.0001, 0.1]$. μ_{Min} was an order of magnitude below the best fixed μ we had found and μ_{Max} was two orders of magnitude higher. This provided the GA with some guidance in finding a good mutation rate by limiting the space it had to explore. However, the upper limit proved to be non-limiting since the GA never approached it. We also allowed the GA to have more precise control over μ by increasing l_μ to 15.

We realized that with μ uniformly distributed initially, there would be some individuals that would be changing 1 out of 100 bits every generation, or 5 bits a generation on average. Thus in Experiment 6 (Seeded), we decided to seed the GA with the best μ we had found in the fixed mutation experiments. By seeding the population with good genetic material, we hoped to facilitate the performance of the GA.

The effects of crossover might confound the search for good μ values. Thus in Experiment 7 (NoCross), we decided to turn off crossover and observe the results.

5. RESULTS

In the results, we examine $t_\delta = (1, 25, 100, 900, 1801)$, since that set illustrates the most interesting phenomenom. We had planned to investigate the temporal dynamics of how the self-adaptive mutation rate changed over time but to do that we first wanted to make sure that the self-adaptitive GA matched the performance of the fixed GA. However, we were never able to achieve that level of performance, and hence most of the rest of this paper is spent examining why we were not able to achieve our first goal. We begin by looking at the best fitness found by the GA in the final generation, and average those results over 30 runs. The best fitness of the generation is representative of how the system can do in the current environment, regardless of the past. The optimal fitness for these sl-hdf's is 191. The results are displayed in Tables 3 and 4.

Finally, we examine the average μ of the population over time for $t_\delta = 100$ and average the results over 30 runs. These results, through generation 1200, are in Figure 2. Though there are some temporal differences for different t_δ's the general trend is similar for all five values, and thus we choose $t_\delta = 100$ as a representative.

Experiment 4 (Self 0-1): Experiments 1-3 (the fixed experiments) are provided mainly for comparison purposes and so we begin by examining the results of Experiment 4. The greatest improvement in the experiments was made by limiting the range of μ that the GA could select which is shown by the improvement of Experiments 5 through 7 over Experiment 4. One interesting result of Experiment 4 is that μ goes to 0 in the first 200 generations as illustrated by Figure 2. Since this experimental setup far underperformed the others it is clear that mutation is needed throughout the run in order to promote good performance. The simplest explanation of this is that mutation maintains diversity in the population and keeps the population from converging to local minima. This is well known, but an explanation as to

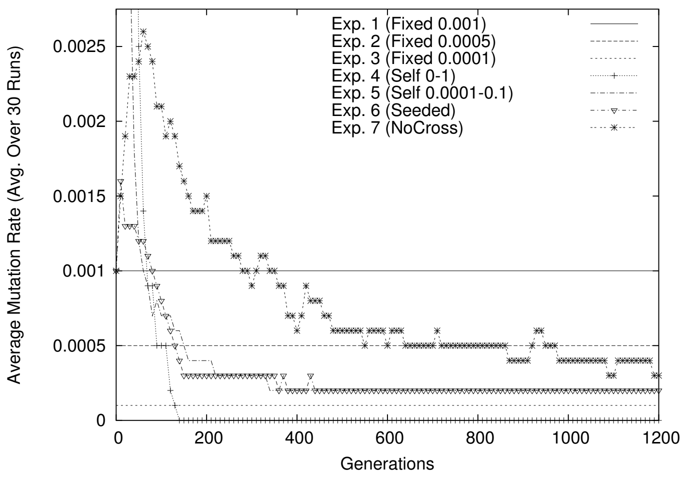

Figure 2: Avg. μ of the Population, $t_\delta = 100$.

why the GA chose not to use mutation is more interesting, and hence we investigate this phenomenom.

Experiments 5 (Self 0.0001-0.1) and 6 (Seeded): It is interesting that seeding the population (Experiment 6) with a known good μ had no significant effect on the overall performance over a non-seeded but limited μ (Experiment 5). As can be seen from Figure 2, these two experiments quickly wind up with the same μ, the only difference is at the beginning. In Experiment 5, the average μ at the beginning is 0.05005 (halfway between 0.1 and 0.0001), while in experiment 6 it is 0.001 the seeded μ value. However after generation 200 both of these experiments maintain roughly parallel μ values. The high initial μ of Experiment 5 may allow it to do more search at the beginning resulting in the two experiments having a long term equivalent performance. In fact Experiment 6 also initially increases its μ value but not nearly as high as Experiment 5 starts out.

Experiment 7 (NoCross): The fact that a high μ initially is good seems to be supported by the results of experiment 7. Experiment 7 shows a larger increase in the average μ value initially than Experiment 6. In the beginnning the GA needs diversity in order to search the large environment, but once it has found a few peaks it decreases its average μ in order to avoid deleterious mutations which destroy good alleles.

Experiment 7 is the only self-adaptation experiment that shows clear differences between the various values of t_δ. As in Experiment 1, the highest rates of fitness are attained by intermediate rates of change, $t_\delta = (25, 100)$; this is because with an intermediate shaking of the ladder the system is forced to maintain diversity and not rely too heavily on any particular set of building blocks [17]. In cases where the environment changes slowly the GA finds local peaks and stays near them. In cases where the environment changes too quickly the GA is never able to maintain a set of intermediate schemata.

The average μ results in Experiment 7 are also interesting. At the end of the run Experiment 7 still maintains an average μ above the minimum value when $t_\delta = 100$ (and

Experiment	$t_\delta = 1$		$t_\delta = 25$		$t_\delta = 100$	
	Best Fit.	*Std. Dev.*	*Best Fit.*	*Std. Dev.*	*Best Fit.*	*Std. Dev.*
1 (Fixed 0.001)	182.17	12.99	191.00	0.00	191.00	0.00
2 (Fixed 0.0005)	120.73	29.30	142.73	22.07	150.73	23.50
3 (Fixed 0.0001)	44.70	7.12	42.83	8.39	48.67	10.94
4 (Self 0-1)	30.90	8.15	32.40	8.50	31.90	10.00
5 (Self 0.0001-0.1)	67.13	13.93	74.90	24.49	76.27	18.42
6 (Seeded)	65.17	9.67	70.47	19.44	75.90	18.12
7 (NoCross)	53.30	11.39	107.87	28.76	81.03	29.15

Table 3: Best Fitness in Current Population after 1800 Generations Averaged over 30 Runs, $t_\delta = \{1, 25, 100\}$

Experiment	$t_\delta = 900$		$t_\delta = 1801$	
	Best Fit.	*Std. Dev.*	*Best Fit.*	*Std. Dev.*
1 (Fixed 0.001)	141.80	21.85	160.97	14.22
2 (Fixed 0.0005)	110.70	19.53	130.87	19.89
3 (Fixed 0.0001)	46.50	10.13	51.77	10.91
4 (Self 0-1)	32.00	10.35	37.50	11.18
5 (Self 0.0001-0.1)	70.87	15.81	83.57	18.51
6 (Seeded)	67.83	19.95	79.43	21.11
7 (NoCross)	61.93	19.80	72.23	19.31

Table 4: Best Fitness in Current Population after 1800 Generations Averaged over 30 Runs, $t_\delta = \{900, 1801\}$

for $t_\delta = 25$, not shown). When $t_\delta = (1, 900, 1801)$ the average μ is much lower than in the intermediate cases. The self-adaptative mechanism appears to be working better in Experiment 7 than the other experiments. It selects a higher average μ for the environments where it is possible to make improvements using mutation on a regular basis (the intermediate changing environments). In environments where the system has found a local optimum and there is not much chance for improvement, $t_\delta = (900, 1801)$, the system selects a lower average μ. In the environment where the system can do very little to adapt to changes, $t_\delta = 1$, the GA reduces its average μ to prevent losing the few building blocks it has found.

General Comments: In general, as is shown in the tables, self-adaptation fails to increase the performance of the best individual in the population in the long term over the best fixed mutation GA. From examining the course of both the average and best fitness, we can say that generally in all stages of exploration self-adaptation underperforms fixed mutation for this problem. However, in Experiment 3 where we set μ to the same as the μ_{Min} in the last three experiments, the GA's results are not significantly different from the self-adaptive results. The fact that the self-adaptation GA does not underperform is probably due to the fact that even though eventually the average μ winds up the same, initially the self-adaptive GAs have a higher average μ resulting in more diversity.

From Figure 2, we can start to see part of the reason why self-adaptation consistently fails to outperform fixed mutation. In the self-adaptive runs the GA lowers its average μ very quickly, and decreases its ability to create new alleles. In order for a population to perform well in a dynamic environment ideally the system should be able to change between higher μ and lower μ values to adapt to changes in the environment. It is possible for a system with a high average μ to find an individual with low μ. Moreover this individual (if successful) will remain in the population since an individual with a low μ is almost guarenteed to produce additional individuals that have a low μ. However, it is hard for a system that has a low average μ to create a large number of individuals with a high μ. Clearly the GA can quickly find an individual with a high μ by mutating the most significant bit of the μ. However the GA can not transition to a *population* with a high μ, because the offspring of the individual with a high μ are generally worse performers, and are not guarenteed to inherit the same high μ, because the μ will even change itself. Thus self-adaptation does not work well in these environments because in attempting to minimize the damage caused by mutation it also destroys its own ability to improve. This is further established by the initial jumps in the average μ as can be seen in several of the experiments. Early on the loss of good alleles due to a high μ is outweighed by the possibility of finding new building blocks but once those blocks are found individuals with lower μ's do better because they avoid self-destruction.

6. DISCUSSION

Why does the GA seem to prefer a low average μ? In biology, how and why natural systems change mutation rates is still an open question [4]. Some biologists have shown that under certain circumstances evolution favors positive optimal mutation rates [14]. Other biologists theorize that mutation rates can only decrease [24], which might explain why the GA favors low μ's.

However we know that in the environments we are looking at there are times when it is beneficial to have a positive μ, therefore we need to consider what incentives we can provide to increase μ at those times. An individual has an incentive to have a low μ since that is good for their offspring but this trend is bad for the population over time because the average μ will tend toward 0 which prevents the GA from solving the problem. Thus by changing the incentive structure we may be able to break the GA out of this situation.

Exploration and Exploitation: This seems to be very similar to the balance between exploration and exploitation that Holland described in his original work on GAs. In order to balance exploration and exploitation, the optimal allocation of trials establishes the representation of individuals in the population in proportion to the observed payoff over time [11]. In this case though we are describing mutation rates, which have a compound effect on the amount of exploration or exploitation being done within the GA. If the GA is on top of a fitness peak then individuals with low μ's are preferred because the GA should exploit the peak that it has found. If the GA is climbing a fitness peak then individuals with high μ's are preferred because the offspring may be higher up the fitness peak. Therefore since in the sl-hdf's there are local optima early on, the GA gets to the top of the fitness peaks and starts to prefer individuals with lower μ's. However as explained above once the GA has a population with low μ's, it is very difficult for it to create a large number of individuals with higher μ's, which makes it hard to jump to the next higher fitness peak.

Thus there are cases where self-adaptation may be useful to the GA. If the observed fitness of individuals with high μ's was greater than the average fitness of the population over time, then their representation in the population would increase. One example would be an environment in which the GA was contantly climbing a fitness peak.

Changes to the GA: It might also be possible to change the mechanism that we are using to vary μ. For instance, one could decrease the cost of switching between a high and low μ. If the GA simply had two preset μ's and could switch between them by flipping one bit, that might be enough to allow the system to maintain some individuals with a high μ and some with a low μ. This would be similiar to hypermutation [7] and variable local search [22] since ideally the GA would simply increase mutation when a change was encountered, but it would still allow each individual to control its μ and would not require global knowledge of the population. Thus current good individuals could maintain a low μ while poor performing individuals could switch to a higher μ. It might also make sense to investigate a GA that made use of a memory system like multiploidy [9] since it would be easier to maintain diversity.

Comparison to Bäck's Results: Why was Bäck able to get a similiar self-adaptation mechanism to work in static environments when it did not work in the static version of the sl-hdf? Bäck utilized two unimodal functions from De Jong and Schwefel (f_1, f_{15}) and one multimodal function from Törn et al (f_7) [13] [18] [21]. Bäck pointed out that self-adaptation had the strongest effect for unimodal functions. When the functions are unimodal it is to their benefit to produce offspring with a high μ because their offspring may be higher up the hill. If the function is multimodal like the sl-hdf's, when the GA finds the local optimum, individuals with a high μ are not preferred because they are likely to produce offspring that are no longer at the local optimum. In the one multimodal function that Bäck investigated self-adaptation only worked better than fixed mutation when he used a strong selection mechanism, unlike the one used here [2].

It also is interesting that μ in Bäck's experiments never goes to the minimum, unlike it does in the our results. One possible reason may be that in the unimodal problems the GA is always (until it finds the global optimum) on the side of a hill, therefore individuals with a positive μ will have more offspring, since some of their offspring will be higher up the hill. Moreover, the multimodal problem that Bäck investigated was a unimodal problem with a sine wave superimposed on top of it, and the potholes may not have been as great as they are in the sl-hdf's. This would allow a positive μ to be favored because individuals with a a positive μ could produce offspring that jumped over the potholes to higher fitness peaks [2].

Research has also shown that the problems Bäck was examining are amenable to easily being solved by a mutation hill-climber [8]. The hdf's on the other hand are designed to be deliberately resistant to hill-climbers [12]. The hdfs are mainly solved through recombination. Two different individuals find different building blocks and recombination puts them together to create a better individual. Thus mutation, though it is important, does not have as strong an effect as recombination. Thus it makes sense that the GA should favor a higher μ in Bäck's experiments than in the sl-hdf's.

Finally as mentioned Bäck only found the best results for self-adaptation when he used very strong selection. Bäck's method is an example of *extinctive* selection since at least one of the individuals in the current population do not have a positive probability of appearing in the next generation. The selection method that we use above (Tournament, size 3) is not extinctive, since even the lowest fitness individuals have some chance of surviving in each round. When Bäck utilizes a similiar selection method he also finds that his mutation rates drop to a much lower rate than when he uses the extinctive selection method.

The effect of this strong selection pressure, the different testing environments and some other minor differences between Bäck's experiments and our own, make it difficult to directly compare the results. Thus the fact that we do not observe positive μ values or an improvement in our results over fixed mutation are not in conflict with Bäck's results. We strongly suspect that the multimodal nature of our test environment is the reason for the differences and we plan to investigate this more in the future. Investigations into the different selection methods and a more indepth comparison to Bäck's multimodal fitness function are also warranted.

7. CONCLUSIONS AND FUTURE WORK

We have shown that one type of self-adaptation fails to improve the performance of the GA in some dynamic environments and for the static version of the environment. However there are environments where self-adaptation is useful, and we plan to investigate this more in the future. Moreover it may be that our implementation of self-adaptation is not the best choice and in the future we plan to examine different implementations [19]. It is important to note that it seems that some sort of dynamic μ would still be valuable in the sl-hdf environment since the ability to increase μ at periods of transition should be beneficial. Finally in the future we plan to investigate other mechanisms like hypermutation.

Regardless, we have observed several interesting phenomenon. For instance, mutation has a large positive influence on the GA throughout its run. In addition, in some cases seeding a population with what is known to be good genetic material (μ) did not improve the overall performance.

The overall goal of our project is to better understand how the GA works in dynamic environments, by conducting systematic controlled observations of the GA. We feel that this allows us to contribute to theory by providing a series of regular observations and to contribute to practice by providing suggestions for a rich set of environments.

Acknowledgements
We would like to thank the University of Michigan's Center for the Study of Complex Systems for the computer resources that they have provided to us without which this paper would not have been possible.

8. REFERENCES

[1] Angeline, P. J., Fogel, D. B., and Fogel, L. J. A comparison of self-adaptation methods for finite state machines in dynamic environments. In *Proceedings of the Fifth Annual Conference on Evolutionary Programming* (1996), L. J. F. et al, Ed., pp. 441–449.

[2] Bäck, T. Self-adaptation in genetic algorithms. In *Proceedings of the 1st European Conference on Artificial Life* (Cambridge, MA, 1992), F. Varela and P. Bourgine, Eds., MIT Press, pp. 263–71.

[3] Bäck, T. On the behavior of evolutionary algorithms in dynamic environments. In *1998 IEEE International Conference on Evolutionary Computation proceedings* (New York, 1998), IEEE, pp. 446–51.

[4] Bedau, M. A., and Packard, N. H. Evolution of evolvability via adaptation of mutation rates. *Biosystems 69* (2003), 143–62.

[5] Branke, J. Evolutionary algorithms for dynamic optimization problems: A survey. Tech. Rep. 387, Institute AIFB, University of Karlsruhe, February 1999.

[6] Branke, J. *Evolutionary Optimization in Dynamic Environments*. Kluwer Academic Publishers, 2001.

[7] Cobb, H. G. An investigation into the use of hypermutation as an adaptive operator in genetic algorithms having continuous, time-dependent nonstationary environments. Tech. Rep. AIC-90-001, Naval Research Laboratory, Washington, D.C., 1990.

[8] Davis, L. Bit-climbing, representational bias, and test suite design. In *Proc. 4th Int. Conference on Genetic Algorithms (ICGA)* (1991), L. Booker and R. Belew, Eds., pp. 18–23.

[9] Goldberg, D. E., and Smith, R. E. Nonstationary function optimization using genetic algorithms with dominance and diploidy. In *Proceedings of the Second International Conference on Genetic Algorithms* (1987), J. J. Grefenstette, Ed., pp. 59–68.

[10] Grefenstette, J. J. Genetic algorithms for changing environments. In *Parallel Problem Solving from Nature 2 (Proc. 2nd Int. Conf. on Parallel Problem Solving from Nature, Brussels 1992)* (Amsterdam, 1992), R. Männer and B. Manderick, Eds., Elsevier, pp. 137–144.

[11] Holland, J. *Adpatation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975.

[12] Holland, J. H. Building blocks, cohort genetic algorithms, and hyperplane-defined functions. *Evolutionary Computation 8*, 4 (2000), 373–391.

[13] Jong, K. D. *An Analysis of the Behaviour of a Class of Genetic Adaptive Systems*. PhD thesis, University of Michigan, 1975. Diss. Abstr. Int. 36(10), 5140B, University Microfilms No. 76-9381.

[14] Kimura, M. On the evolutionary adjustment of spontaneous mutation rates. *Genetical Research 57* (1967), 21–34.

[15] Mitchell, M. *An Introduction To Genetic Algorithms*. MIT Press, Cambridge, Massachusetts, 1997, ch. 1.

[16] Mitchell, M., Forrest, S., and Holland, J. H. The royal road for genetic algorithms: Fitness landscapes and GA performance. In *Towards a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life, 1991* (Paris, 11–13 1992), F. J. Varela and P. Bourgine, Eds., A Bradford book, The MIT Press, pp. 245–254.

[17] Rand, W., and Riolo, R. Shaky ladders, hyperplane-defined functions and genetic algorithms: Systematic controlled observation in dynamic environments. In *EvoWorkshops 2005 Proceedings* (2005), R. et al., Ed., Lecture Notes In Computer Science, Springer. In Press.

[18] Schwefel, H.-P. Evolutionary learning optimum-seeking on parallel computer architectures. In *Proceedings of the International Symposium on Systems Analysis and Simulation 1988, I: Theory and Foundations* (1988), A. Sydow, S. G. Tzafestas, and R. Vichnevetsky, Eds., pp. 217–25.

[19] Smith, J. E. *Self Adaptation In Evolutionary Algorithms*. PhD thesis, University of West England, Bristol, July 1998.

[20] Stanhope, S. A., and Daida, J. M. Optimal mutation and crossover rates for a genetic algorithm operatin in a dynamic environment. In *Evolutionary Programming VII* (1998), no. 1447 in LNCS, Springer, pp. 693–702.

[21] Törn, A., and Zilinskas, A. Global optimization. In *Lecture Notes In Computer Science*, vol. 350. Springer, 1989.

[22] Vavak, F., Jukes, K. A., and Fogarty, T. C. Performance of a genetic algorithm with variable local search range relative to frequency of the environmental changes. In *Genetic Programming 1998: Proceedings of the Third Annual Conference* (University of Wisconsin, Madison, Wisconsin, USA, 22-25 1998), J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, Eds., Morgan Kaufmann, pp. 602–608.

[23] Whitley, D., Rana, S. B., Dzubera, J., and Mathias, K. E. Evaluating evolutionary algorithms. *Artificial Intelligence 85*, 1-2 (1996), 245–276.

[24] Williams, G. C. *Adaptation and Natural Selection*. Princeton University Press, Princeton, NJ, 1966.