Genetic Algorithms, and Hardness

BART RYLANDER
School of Engineering
University of Portland
Portland, Or 97203
U.S.A.
rylander@up.edu

JAMES FOSTER
Department of Computer Science
University of Idaho
Moscow, Idaho 83844
U.S.A.
foster@cs.uidaho.edu

Abstract: - Genetic algorithm (GA) researchers have long desired to describe the key characteristics of GA-hardness. Unfortunately this goal has remained largely unmet. In this paper, we describe a new way for evaluating GA-hardness that is based on the foundations of formal complexity theory. In particular, a GA-reduction is introduced. Using this reduction, we show that a reasonable definition of "GA-hardness" is essentially the same as the definition of "hardness" in complexity theory. We then provide a definition called "GA-semi-hard" that describes a problem that takes longer to solve with a GA than it does using any other method. The problem "Sort" is shown to meet this criterion. Finally, implications and possible future research directions are discussed.

Key-Words: -Genetic Algorithms, Hardness, GA-hard, Complexity

1 Introduction

The GA is a biologically inspired search method that seeks to converge to a solution using an evolutionary process. It is typically used when a more direct form of programming cannot be found. GAs have successfully been applied to problems from such diverse fields as economics, game theory, genetics, and artificial intelligence, to name a few [1].

This broad success has prompted researchers to question whether there may be problems that do not lend themselves so easily to GAs. The goal of finding such problems has been informally described as the search to identify what is "GA-hard" [2][3]. Unfortunately, though this search has been in progress for over twenty years, efforts have largely failed to achieve acceptance for identifying what is truly GA-hard. This, in part, is due to the fact that GA-hard has never been formally defined. Instead, most work has concentrated on trying to find problem characteristics that cause a GA to fail to converge, or at least to converge more slowly. This, despite the fact that "converge more slowly" has never been defined as the *essence* of GA-hard. Perhaps because of this, most efforts have only produced results in the analysis of representations and not in the identification of a universal *GA-hard*. While this is significant, it should be noted that features of a particular representation are not necessarily features of a *problem*. Knowing what features cause a representation to be GA-hard may not imply that the underlying problem is GA-hard. To the contrary, it is well known that there are an infinite number of representations for every problem instance. Given this, it is easy to see that not all representations for a given problem instance necessarily exhibit the same degree of epistasis or deception [2].

Another difficulty lies with the fact that there has been little attempt to distinguish between a *problem*, and a problem instance (usually referred to simply as an "instance"). A *problem* is a mapping from problem instances onto solutions [4]. For example, consider the Maximum Clique (MC) problem (see Definition 2), which is to find the largest complete subgraph H of a given graph G. An *instance* of MC would be a particular graph G, whereas the *problem* MC is essentially the set of all pairs (G, H) where H is a maximal clique in G.

Our main contention then, is that most current approaches to GA-hardness actually explain why a GA is unlikely to converge quickly to a solution on a particular problem instance given a particular representation. Since this is clearly inadequate, it seems pressing that a new approach be described. This paper represents a shift to a more formal approach to identifying GA-hardness. Using the techniques of formal computational complexity theory, we define what a GA-hard problem is. Section 2 introduces basic concepts of computational complexity theory. Section 3 applies these concepts to the analysis of the GA, introducing several definitions specific for GAs. Finally, section 4 is a discussion of the implications of this work.

2 Complexity and Hardness

Informally, computational complexity is the study of classes of problems based on the rate of growth of space, time, or other fundamental unit of measure as a function of the size of the input. It is a study that seeks to classify problems based on their relative computational intractability. Since the search for GA-hardness is in fact a search for degrees of intractability specifically for a GA, we begin by first describing what *intractable* means in general.

2.1 Complexity Classes

Since many problems that GAs are used to solve are optimization problems, PO and NPO seem the most relevant complexity classes to describe. Though P and NP may be more familiar, PO and NPO are in fact the optimization equivalents of P and NP, which are classes of decision problems [5].

> **Definition 1:** The class PO is the class of optimization problems that can be solved in polynomial time with a Deterministic Turing Machine (DTM) [5].

An example of such a problem is:

Minimum Partition:
Given: A = $\{a_1 \ldots a_n\}$ of non-negative integers such that for all i>j, $a_i \geq a_j$.
Find: Partition of A into disjoint subsets A1 and A2 which minimizes the maximum of $\left(\sum_{a\in A1} a, \sum_{a\in A2} a\right)$

> **Definition 2:** The class NPO is the class of optimization problems that can be solved in polynomial time with a nondeterministic Turing Machine (NTM) [5].

For example,
Maximum clique (MC):
Given: Graph G
Find: Largest complete sub-graph of G.

The class NPO contains the class PO. This means that problems that can be done in polynomial time with a DTM can also be done in polynomial time with an NTM. These two classes are by no means the only interesting or important ones. Nonetheless, they are sufficient for our purposes, since NPO most likely includes all tractable optimization problems. That is, these problems are computable in a reasonable amount of time.

2.2 Hardness

Given a complexity class, the formal way to define a *hard* problem is to show that every problem in the class reduces to the *hard* problem. In other words, a reduction R is a mapping from problem A onto problem B in such a way that one can optimize any problem instance *x* of A by optimizing B(R(*x*)). For example, there is a polynomial time computable transformation from an instance of Min-partition, which is a list of non-decreasing non-negative integers A, into an instance of MC, which is a graph G with the interesting property that if one finds the maximum clique in G one can recover the minimum partition of A. *Hardness* then, is an upper bound relative to the reduction R, since one need never do more work to solve any problem in the class than to transform the instance and solve the instance of the hard problem. As an example, if MC were reducible to Min-partition, this would place an easy (PO) upper bound on a hard (NPO) problem—implying that MC wasn't so hard after all. The formal definition of hardness is:

> **Definition 3:** A problem H is Hard for class C if, for any problem $A \in C$, and any instance *x* of A, optimizing R(A(x)) optimizes x [4].

Hardness in this sense, is a rigorously defined concept that can only correctly be used in conjunction with a specified class of problems and a specified reduction. This contrasts with the usually subjective notion of the term in current GA theory.

3 GA-Hardness

Having a formal understanding of classical hardness, we can begin to provide a definition of GA-hardness. It's important at this point to note what an appropriate definition should be. A useful notion will single out problems that are upper bounds on the complexity of some interesting class. For example, equating "GA-hard" with "nonrecursive" problems, or with "needle in a haystack" problems, would be singularly uninformative. Before this can be done however, there are a few items that need to be addressed. In the first place, we must describe a method for evaluating a problem specifically for a GA. Fortunately, this has already been addressed. A brief description of such a method is provided here, a more complete one can be found in [6].

3.1 Minimum Chromosome Length (MCL)

The goal for a representation is to minimize the number of bits in a chromosome that still uniquely identifies a solution to the problem [6]. In so doing, one minimizes the search space that must be examined to find an answer.

Definition 4: For a problem P, and D_n the set of instances of P of size n, let MCL(P,n) be the least *l* for which there is an encoding $e:S^l \rightarrow D_n$ with a domain dependant evaluation function g, where g and e are in FP (the class of functions computable in polynomial time).

That is, MCL(P,n) measures the size of the smallest chromosome for which there is a polynomial time computable representation and evaluation function. Notice that for a fixed problem P, MCL(P,n) is a function of the input instance size n, and so is comparable to classical complexity measures. Also, since a smaller representation means a smaller number of solutions to search, there must be a limit as to how small the representation can become. So MCL(P,n) is well-defined for every P and n.

It is important to stress that the domain dependent evaluation function must be feasible. Otherwise, it would be possible to skew the analysis by having an exponential time or even uncomputable function in the main loop of the GA. By bounding this function with a polynomial we ensure that the evaluation does not materially increase the worst-case analysis in terms of problem classes.

Given this, we can now provably describe the complexity of a problem instance as implemented with a GA. However, since complexity is typically based on the rate of growth of a fundamental unit of measure as a function of the input size, there is another step in our analysis. We must show how the growth of the problem instance size, n in our definition, correlates with the MCL.

As an example, consider the graph G with 5 nodes in Figure 1. We label the nodes 0 through 4. The maximal clique in G is the set of nodes {1,2,3}. For an n-node graph, we have an n-bit representation that designates subgraph H, where the ith bit of the chromosome is 1 (0) if node i is (is not) in H. 01110 represents the maximum clique in our example. This representation has been used several times for this problem [7][8].

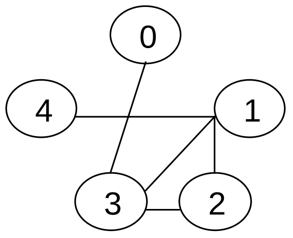


Fig.1, example instance of MC, with solution {1,2,3}.

This representation is an MCL for the problem Maximum-clique, since any smaller representation would be unable to represent all 2^n subsets of nodes for an n-node graph, and each subset is a maximal clique for some input instance. In fact, MCL(MC,n)=n is easy to verify.

Now, consider what happens when you add a sixth node to this graph. You must expand your chromosome to six bits. This means that the MCL grows by one as the size of the input instance grows by one. This also means that the search space has increased to 2^6. Clearly, as the size of the input increases linearly, the MCL grows linearly and the search space grows exponentially. This is the one hallmark of a hard problem. In the case of MC or any other NP hard problem, there is no known deterministic algorithm to search this exponentially growing space in polynomial time.

Recent research has shown that the MCL growth rate can be used to bound the worst case complexity of a problem for a GA [6]. If a problem is in NPO, then the MCL growth rate will be no more than linear, since an NP algorithm can search a space which grows no more quickly than exponentially, and linear MCL growth implies exponential search space growth. Conversely, if MCL grows slowly enough, then the search space will be small enough to place the problem in PO.

Now that there is a method for evaluating the complexity of a problem specifically for a GA, it is possible to address what a reduction for a GA must be.

> **Definition 5:** GA Reduction
> Let A and B be subsets of Σ^* and T* and let *f* be some function from Σ^* to T*. If |MCL(A,n)| = |MCL(B,n)| and |MCL(A,n+1)| = |MCL(B,n+1)|, we say that *f* reduces A to B, in case f^{-1}(MCL(B,n))=MCL(A,n).

This means a GA-reduction is a length - preserving translation from one problem's representation to another. Given this, it is easy to see that the same definition for hardness is appropriate for GAs as it is classically. (see Definition 3, brought forward).

> **Definition 3:** Complexity, Hard
> A problem H is Hard for class C if, for any problem $A \in C$, and any instance *x* of A, optimizing R(A(x)) optimizes x.

3.2 GA-semi-hard

One of the requirements for being GA-hard is that all other problems in a particular class must reduce to it. But this doesn't describe those problems that do not fit the classical definition of hardness and yet are inherently less efficient than if solved using a method other than a GA. More specifically, suppose a problem such as Min-partition, for which there is a known polynomial time algorithm, takes exponential time if solved with a GA (remembering of course that the problem is not simply solved in the encoding or decoding functions). Problems such as these are called GA-semi-hard (Definition 6).

> **Definition 6:** GA-semi-hard
> An optimization problem P is called Semi-hard if there is a DTM M that solves it in polynomial time and $MCL(P,n) \in O(n)$.

In fact, these problems may be the most interesting, since they are clearly ones for which a GA would be a poor selection for application.

Such problems are not just conjecture. Consider the problem of Sort (i.e. given a list of n elements, arrange the list in some predefined manner). In order for a GA to perform this task, as a minimum, there must be a representation that includes an ordination of each element. For example, let our list comprise of 8,4,3,and 7. One possible representation is to produce a chromosome where 4 bits are required for each element. In this case, 1000 0100 0011 0111 represents all elements. However, it is possible to provide for an ordinal representation of the elements allowing for a translation in the decoding function. Since there are only 4 elements, you would only need 3 bits per element. In this case, the chromosome would be 100 010 001 011 (for 8,4,3,7 being translated to 4,2,1,3, their respective rank in terms of size). This results in a search space of 2^{12}. By adding a 5th element, the chromosome must grow by 3 bits making the search space 2^{15}. Clearly this is not polynomial growth. In addition, as the number of elements becomes 8 or greater, the number of bits per element has to increase to 4. So the growth rate actually increases as the instance size does. Conducting an experiment, the results are verified empirically (see table 1).

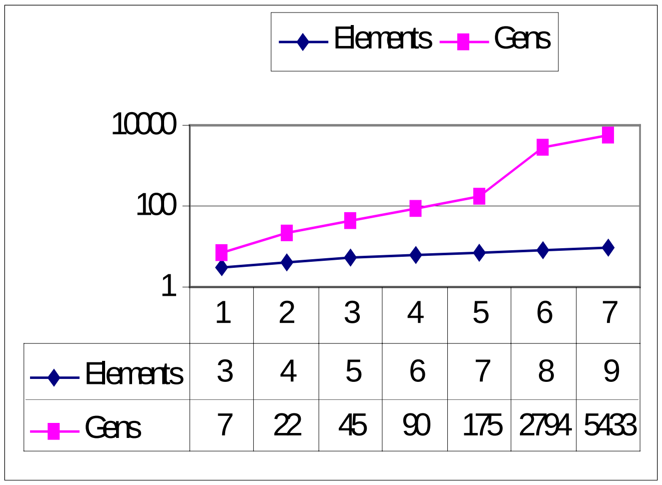
Elements	3	4	5	6	7	8	9
Average Generation	7	22	45	90	175	2794	5433

Table 1, the average number of generations to converge per number of elements to sort.

Note that for each extra element the number of generations to converge roughly doubles. This is with the exception of the jumps from 3 to 4 elements and from 7 to 8 elements. This is

because in the latter, in addition to adding an element, each individual element increased in size by 1 bit, thus causing a larger than expected jump in search space size as well as convergence time.

The experiment was conducted using a fixed population of 10 chromosomes. There was random single-point crossover, 1-% mutation, 50% elitism (thus 50% survival rate) and selection based randomly from the surviving chromosomes. Each instance size was tested 5000 times to produce the average number of generations per convergence per instance size. As can clearly be seen, Sort as implemented with a GA takes exponential time. Below is a logarithmic graph of the average convergence times.



	1	2	3	4	5	6	7
Elements	3	4	5	6	7	8	9
Gens	7	22	45	90	175	2794	5433

Fig. 2, logarithmically scaled graph of the average convergence time per number of elements for Sort.

It may be possible to produce a smaller representation for Sort. Suppose for example, that ignoring crossover problems, you could represent each ordinal element by its own smallest representation. That is, for a 4-element list, using 100 11 10 1 as the representation. But even with this representation, each additional element requires additional bits in the chromosome. And, as the number of elements increases, the largest element will require increasingly more bits for its representation. So even with this representation, the GA growth rate for Sort is exponential. Since we know that Sort is in PO, it seems clear that Sort is GA-semi-hard.

6 Conclusions

We have introduced the concepts of formal computational complexity to the search for GA-hardness. In particular, we described how a GA-reduction is accomplished. With this, we gave reasons why GA-hardness is essentially the same as traditional complexity hardness. We introduced the idea of GA-semi-hard to describe problems that take longer for a GA to solve than other methods, but yet don't meet the criterion to be GA-hard. In addition, it was demonstrated that the problem Sort is GA-semi-hard, and thus probably not a problem one would choose to solve with a GA.

By introducing these ideas, we hope to have provided a framework from which practitioners can evaluate problems for GA implementation, hopefully providing insight into which problems are suitable and which problems are not. As a result, some questions remain unanswered. For example, it might be interesting to evaluate problems on the high end of the complexity hierarchy such as NP-complete problems, or even Pspace problems to see if GAs are perhaps more efficient for problems that are inherently difficult. In the converse, it seems that the evolutionary baggage that GAs must lug may prevent them from being prudent choices for problems in PO.

Interestingly, we did not have to introduce a new model of computation for GAs in order to do this analysis. This makes it more likely that traditional proof strategies may be applicable, and it also makes it very likely that this theory is applicable to search algorithms other than GAs.

This paper raises interesting questions that need to be answered. Rather than enumerating them here, we encourage the interested researcher to find a question that interests them, then let us know the answer.

References:

[1] Rawlins, G.J.E. (1991). Introduction. *Foundations of Genetic Algorithms*, Morgan Kaufman. pp. 1-10.

[2] Whitley, L.D. (1991). Fundamental Principles of Deception in Genetic Search, *Foundations of Genetic Algorithms*, Morgan Kaufman. pp. 221-242.

[3] Liepins, G. And Vose, M. (1990), Representation Issues in Genetic Optimization, J. Experimental and Theoretical Art. Intell. 2(1990) pp. 101-115.

[4] Balcàzar, J., Díaz, J., and Gabarró, J. (1988). *Structural Complexity Theory I*, Springer-Verlag.

[5] Bovet, D., and Crescenzi, P. (1994). *Computational Complexity*, Prentice Hall.

[6] Rylander, B. and Foster,J. Computational Complexity and Genetic Algorithms, Manuscript, submitted to WSES 2001.

[7] Marconi, J., Foster, J. (1998). Finding Cliques in Keller Graphs with Genetic Algorithms, Proc. Int. Conf. on Evolutionary Computing.

[8] Soule, Terence, James A. Foster, and John Dickinson. (1996). Using genetic programming to approximate maximum cliques, Proc. Genetic Programming. pp. 400-405.