Complex Systems **8** (1994) 1–24

Covers: A Theory of Boolean Function Decomposition

Scott E. Page
Division of Humanities and Social Sciences 228-77,
California Institute of Technology, Pasadena, CA 91125, USA

Abstract. In this paper, we develop a theory of covers for functions defined over boolean strings. Given a function, a cover is a decomposition, though not necessarily a partition, of the bits into subproblems that can be solved in parallel. In the paper, we formulate the notion of *cover size*, which equals the size of the largest subproblem in a decomposition for a particular function. Cover size is defined relative to a function's upper contour sets. An implication of cover theory is that a problem's difficulty, as measured by its cover size, necessarily decreases as the function value improves. We also show how cover theory lends insight to the performance of hillclimbing algorithms and genetic algorithms, and present data from simulations that support a covers-based explanation for genetic algorithm performance.

1. Introduction

This paper introduces the theory of covers for functions defined over boolean strings. Cover theory serves three purposes. First, covers formalize the familiar notion that complex problems defined over many variables can be decomposed into simpler subproblems each containing fewer variables. Each of the subproblems can then be solved in parallel, thereby decreasing computation time. Practical applications of the benefits from decomposition include the parallel architecture of supercomputers, the divisionalization of firms, and the decentralization of economic activity [24]. In this paper, we show how cover theory might be applied to the multiple public projects problem.[1] Second, covers provide a measure of decomposability with respect to upper contour sets. Nonlinear effects that occur near the optimum are more relevant than nonlinear effects arbitrarily located in the domain. Third, covers can be used to analyze the performance and optimize classes of search algorithms. This last point requires clarification. Locating a cover is difficult. We do not mean to imply that locating a cover is an efficient optimization algorithm. What cover theory can do is shed light on the performance of

[1] See [20] for a more complete analysis of the multiple public projects problem.

several search techniques including genetic algorithms as discussed later in this paper.

In the problems we consider, a decision maker wants to maximize a function V that maps boolean strings of fixed length onto the real numbers.[2] A *cover* represents a decomposition of the string into substrings that can be solved in parallel. To explain the notion of a cover, we present a simple example. This problem could be solved by enumeration rather easily. The purpose of this example is to explain what a cover is and not to demonstrate the concept's usefulness.

Suppose that a city is considering three public projects: an airport (a), a botanical garden (b), and a cable car system (c). Suppose that the city's value function is well defined and denoted by V, where V maps the power set of $\{a, b, c\}$ onto the real numbers. Let the empty set $\emptyset$ denote the status quo and let "ab" denote the state of the world where the airport and the botanical garden are provided, but the cable car system is not. Suppose that V satisfies the following inequalities:

$$V(c) < V(\emptyset) < V(bc) < V(b) < V(a) < V(ab) < V(ac) < V(abc)$$

If the objective function V is not known ex ante but is revealed through cost-benefit analysis, a complete decomposition of the three project decisions so that each decision is made independently might not lead to the optimal choice of projects. In this scenario, the decision on the cable car system is problematic: the cable car system should not be built if the airport is not built:

$$V(c) < V(\emptyset) \quad \text{and} \quad V(bc) < V(b)$$

However, if the airport is built, then the cable car system should be built as well. It follows that coordination on decisions is required to guarantee the optimal choice over subsets of projects. Suppose that we decompose the set of projects into the sets $\{a, c\}$ and $\{b\}$ and make decisions on these two sets independently. Consider first the decision on the botanical garden. The inequalities below follow from above:

$$V(\emptyset) < V(b) \quad V(c) < V(bc) \quad V(a) < V(ab) \quad V(ac) < V(abc)$$

These inequalities show that regardless of the decision on the other projects, the botanical garden should always be provided. Consider second the decision on the other set of projects:

$$\max\{V(\emptyset), V(a), V(c)\} < V(ac)$$
$$\max\{V(b), V(ab), V(bc)\} < V(abc)$$

These two inequalities show that providing both the airport and the cable car system is the preferred alternative, regardless of the decision on the botanical garden. Combining the decisions on the subproblems $\{a, c\}$ and $\{b\}$ leads to

[2]Formally, all that we require is that range of the function be a completely ordered set.

the optimal decision over the set of all projects, that is, providing all three projects. More generally, a decomposition into subproblems *forms a cover* if (i) each decision (variable) belongs to at least one subproblem and (ii) the optimal decisions on the subproblems "agree" with the optimal decision on the larger problem. In the example, $\{a, c\}$ and $\{b\}$ form a cover and also partition the set of decisions. The latter need not be true in general; the same variable may belong to more than one subproblem.

This example can also be used to demonstrate how covers measure decomposability relative to the function's upper contour sets.[3] Suppose that the airport's value is known to be greater than the status quo value and that the airport's external effect on the other two projects is thought to be positive. Suppose further that, preliminarily, the airport is assumed to be provided so that the starting point for optimization is "a." In this case, the problem can be decomposed into the sets $\{a\}$, $\{b\}$, and $\{c\}$. To understand why, first consider the decision on the cable car system.

$$V(a) < V(ac)$$
$$V(ab) < V(abc)$$

The value of the airport together with the cable car system, $V(ac)$, exceeds the value of the airport alone, $V(a)$. And the value of all three projects, $V(abc)$, exceeds the value of the airport and the botanical garden, $V(ab)$. It follows that the optimal decision on the subproblem $\{c\}$ is to provide the cable car system. A similar argument establishes that the optimal decision on the botanical garden is to provide that project as well. Finally, the optimal decision on the airport is to not reverse the earlier decision. Regardless of the decisions on the other two projects, the airport is always worth providing. Thus, we say that the sets $\{a\}$, $\{b\}$, and $\{c\}$ form a cover for V on the contour set above "a."

In the formal model presented later in this paper, the *cover size* of a function equals the number of variables in the largest subproblem of a decomposition. This measure is most relevant if the subproblems are to be solved in parallel: the time required to solve the problem equals the time required to solve the largest subproblem. Alternatively, if the function is to be solved sequentially, then another measure may be more appropriate. In a companion paper, we show how cover theory can be used to formulate a measure of *ascent size*, which captures the difficulty of solving a problem sequentially [21].

Given this measure of cover size, we see in the example that by starting optimization from a better initial set of projects ($V(a) > V(\emptyset)$) the cover size decreased. The maximum number of decisions in any one subproblem was reduced from two to one. Interpreting cover size as a measure of the difficulty of optimizing, we can say that the problem becomes less difficult near the optimum. Later in this paper we show that this is a generic phenomenon:

[3]The upper contour set above θ consists of all elements of the domain whose values are greater than or equal to θ.

cover size decreases as the initial point of search improves for *any function* defined over binary strings.

The remainder of this paper is organized as follows. In sections 2 and 3 we define covers for functions defined over binary strings and construct a decomposability vector. We also compare cover size with other measures of complexity and present data from test functions. In section 4 we use covers to select optimal parameters for a class of hillclimbing algorithms and offer an alternative explanation for the performance of genetic algorithms. The latter discussion focuses on the "building block hypothesis" and its interpretation through the lens of covers. In our discussion of genetic algorithm performance, we also present data from test functions that support a cover theory interpretation. In the conclusion, we discuss a more general notion of covers mentioned by Richardson [23].

2. Binary strings

A cover decomposes a problem into subproblems that can then be solved in parallel. This decomposition is relative to the objective function's upper contour sets. We begin with some basic definitions.

2.1 Preliminaries

We refer to each binary variable as a *bit* and to a decision on each binary variable as a *string.*

Definition 1. The set of *bits*, $N = \{1, 2, 3, \ldots, n\}$.

Definition 2. The set of *strings*, $S = \{s \mid s = s_1 s_2 \ldots s_n \text{ with } s_i \in \{0, 1\}\}$.

We assume that we are trying to maximize V, which belongs to F, the set of all functions whose domain can be encoded as binary strings of length n and whose range is the real numbers.

Definition 3. The set of *objective functions*, $F = \{V \mid V : S \rightarrow \Re\}$.

A class of subsets of N called hyperplanes play a prominent role in the analysis. A hyperplane can be represented by a ternary string of length n over the set $\{0, 1, *\}$.

Definition 4. The set of *hyperplanes*, $H = \{h \mid h = h_1 h_2 \ldots h_n \text{ with } h_i \in \{0, 1, *\}\}$.

For ease of exposition, the ternary variables h_i are also referred to as bits. A bit in a hyperplane is defined if it takes on the value 0 or 1.

Definition 5. The *defined bits* of h, $d(h) = \{i \mid h_i \in \{0, 1\}\}$.

A string lies in a hyperplane if the string and the hyperplane have identical values on the defined bits of the hyperplane. We let $S(h)$ equal the set of binary strings that belong to h.

Definition 6. The *set of binary strings belonging to* h, $S(h) = \{s \mid s_i = h_i \text{ if } h_i \in \{0,1\}\}$.

Example 1. $S(0*1*) = \{0010, 0011, 0110, 0111\}$.

The size of h equals the number of defined bits of h.

Definition 7. The *size* of h, $\sigma(h) = |d(h)|$

According to this measure, a hyperplane's size equals its co-dimension. A hyperplane with a larger size contains fewer strings.

2.2 String dominant hyperplanes

The idea underlying the definition of a cover is that "good" hyperplanes can be combined to form "good" strings. This same idea is the basis for the Schema Theorem described in section 4. We begin by defining the projection operator $\wedge$, which combines hyperplanes.

Definition 8. The *projection operator* $\wedge : H \times H \rightarrow H$ satisfies the following rule: $h \wedge \tilde{h} = y$, where

$$y_i = \begin{cases} \tilde{h}_i & \text{if } h_i = * \\ h_i & \text{if } h_i \in \{0,1\} \end{cases}.$$

Example 2. $0** \wedge 1*1 = 0*1$.

The set of strings S is contained in H, therefore $\wedge$ is also a map from the Cartesian product of H and S into S. We think of $h \wedge s$ as moving the string s into the hyperplane h while making the minimal number of changes in bit values.

Claim 2.1. *The operator* $\wedge$ *is associative but not commutative.*

Proof. $\wedge$ *associative:* $h \wedge (\tilde{h} \wedge \hat{h}) = y$, where

$$y_i = \begin{cases} \hat{h}_i & \text{if } h_i = \tilde{h}_i = * \\ \tilde{h}_i & \text{if } h_i = * \text{ and } \tilde{h}_i \in \{0,1\} \\ h_i & \text{if } h_i \in \{0,1\} \end{cases}$$

It is straightforward to show that $y = (h \wedge \tilde{h}) \wedge \hat{h}$, which completes the proof.

$\wedge$ *not commutative:* Let $h = 00*$ and $\tilde{h} = *1*$. Then $h \wedge \tilde{h} = 00*$, but $\tilde{h} \wedge h = 01*$. ∎

Recall that a motivation for covers is that "good" hyperplanes can be combined to form "good" strings. A strong notion of "good" hyperplane is Greffenstette and Baker's [7] *dominant hyperplane.*

Definition 9. A hyperplane h is *dominant* for V if $\forall s \in S(h)$ and $\forall \hat{s} \notin S(h)$, $V(s) \geq V(\hat{s})$.

Definition 10. A hyperplane h is *strictly dominant* for V if $\forall s \in S(h)$ and $\forall \hat{s} \notin S(h)$, $V(s) > V(\hat{s})$.

Claim 2.2 below states that, given two strictly dominant hyperplanes, one must be a subset of the other. This implies that there cannot exist strictly dominant hyperplanes with nonintersecting sets of defined bits. Therefore, it is nonsensical to speak of combining strictly dominant hyperplanes to form a good string.

Claim 2.2. *For any h, $\hat{h}$ strictly dominant for V, if $h \neq \hat{h}$ and $\sigma(h) \leq \sigma(\hat{h})$, then $S(\hat{h}) \subset S(h)$.*

Proof. (by contradiction) Suppose $\exists \hat{s} \in S(\hat{h})$ such that $\hat{s} \notin S(h)$. It follows that $\exists s \in S(h)$ such that $s \notin S(\hat{h})$. But h strictly dominant for V implies $V(s) > V(\hat{s})$, while $\hat{h}$ strictly dominant for V implies $V(\hat{s}) > V(s)$, a contradiction. ∎

A consequence of Claim 2.2 is that if h and $\hat{h}$ are dominant but not strictly dominant hyperplanes for V, then the function V must take an identical value for all strings that belong to exactly one of the hyperplanes. It appears then, that requiring a hyperplane to be dominant is too restrictive for our purposes. As an alternative to dominant hyperplanes, we propose the weaker notion of *string dominance*, which is sufficiently weak to allow for hyperplanes to be combined but strong enough to ensure that the hyperplanes combine to form the optimal string. A hyperplane h is said to be string dominant on a subset of strings T if the value of a string in T does not decrease when moved into hyperplane h by the operator $\wedge$. Formally,

Definition 11. A hyperplane h is *string dominant* for V on T if $V(h \wedge s) \geq V(s)$, $\forall s \in T$.

Definition 12. A hyperplane h is *strictly string dominant* for V on T if $V(h \wedge s) > V(s)$, $\forall s \in T \backslash h$ where $T \backslash h = \{s \mid s \in T, s \notin S(h)\}$.

Claim 2.3 below states that the operator $\wedge$ preserves string dominance.

Claim 2.3. *If h and $\hat{h}$ are string dominant for V on T and $\hat{h} \wedge s \in T$ for all $s \in T$, then $h \wedge \hat{h}$ is string dominant on T.*

Proof. If $\hat{h}$ string dominant for V on T, then $V(\hat{h} \wedge s) \geq V(s)$, $\forall s \in T$. By assumption, $\hat{h} \wedge s \in T$. Therefore, given that h is string dominant for V on T, it follows that $V(h \wedge (\hat{h} \wedge s)) \geq V(\hat{h} \wedge s)$, which by the associativity of $\wedge$ implies that $V((h \wedge \hat{h}) \wedge s) \geq V(s)$, which completes the proof. ∎

3. Covers

3.1 Definition of covers

In this section we formally define a cover. Before doing so, we need to define the contour sets for the objective function V. To simplify the analysis, we assume that no two strings have the same value under V. This assumption allows us to ordinally define the upper contour sets. The extension to cardinal characterization of the upper contour sets and non-injective objective functions is straightforward.

Assumption 1. $\forall s, \hat{s} \in S$, if $s \neq \hat{s}$, then $V(s) \neq V(\hat{s})$.

Given Assumption 1, the strings can be ordered from 1 to 2^n according to their value under V.

Definition 13. *S ordered by V* $= \{s^1, \ldots, s^{2^n}\}$ where $V(s^i) > V(s^{i+1})$ for $i = 1$ to $2^n - 1$

Definition 14. The *upper contour set including s^α*, $T(\alpha) = \{s^\beta \mid \beta \leq \alpha\}$

The next claim states that string dominant hyperplanes map an upper contour set onto itself.

Claim 3.1. *If h is string dominant for V on $T(\alpha)$, then $h \wedge s \in T(\alpha)$, $\forall s \in T(\alpha)$.*

Proof. If h is string dominant for V on $T(\alpha)$, then $V(h \wedge s) \geq V(s)$, $\forall s \in T(\alpha)$. If $s \in T(\alpha)$ and $V(h \wedge s) \geq V(s)$, then by assumption $h \wedge s \in T(\alpha)$. ∎

Corollary 3.1 follows directly from Claim 2.3 and Claim 3.1.

Corollary 3.1. *If $h, \hat{h}$ are string dominant for V on $T(\alpha)$, then $h \wedge \hat{h}$ is also string dominant for V on $T(\alpha)$.*

We now define a cover for V. A cover is a finite set of string dominant hyperplanes, the union of whose defining bits contains all variables.

Definition 15. The collection of hyperplanes, $C = \{h^1, h^2, \ldots, h^m\}$, forms a *cover* for V on T if (i) and (ii) hold:

(i) h^i is string dominant for V on T for all i;

(ii) $\bigcup_{i=1}^{m} d(h^i) = N$.

This definition allows for two hyperplanes in a cover to be defined on the same bit. The example below shows that a cover is not necessarily a partition.

Example 3. Let $n = 3$ and $V(s) = 3s_1 + s_2 + s_3 - 2s_1s_2 - 2s_1s_3$. It is straightforward to show that $C = \{10*, 1*0\}$ is a cover for V on S.

Given Assumption 1, it can be shown that two hyperplanes in a cover must agree on any bits that are defined for both hyperplanes. Another consequence of this definition is the following.

Claim 3.2. *If C is a cover for V on $T(\alpha)$, then C is a cover for V on $T(\beta)$, $\forall \beta \leq \alpha$.*

Proof. If h^i is string dominant for V on $T(\alpha)$, then h^i is also string dominant for V on $T(\beta)$, $\forall \beta \leq \alpha$, which completes the proof. ∎

The next result below states that any string belonging to every hyperplane in a cover for V must optimize V. In other words, the optimal string can be located by forming a cover for V. A consequence of Claim 2.1 is that the order in which the hyperplanes are located is irrelevant.

Claim 3.3. *If $C = \{h^1, h^2, \ldots, h^m\}$ is a cover for V on $T(\alpha)$, then $h^1 \wedge (h^2 \wedge (\ldots h^m \wedge (s)\ldots)) = s^1$, $\forall s \in S$.*

Proof. By Claim 3.2, C is a cover for V on $T(1) = \{s^1\}$. By Corollary 3.1, if h^i is string dominant for V on $T(\alpha)$, then $V(h^1 \wedge (h^2 \wedge (\ldots h^m \wedge (s^1)\ldots))) = V(s^1)$. It follows that

$$h^1 \wedge (h^2 \wedge (\ldots h^m \wedge (s^1)\ldots)) = s^1.$$

Therefore, by (ii) in the definition of a cover,

$$h^1 \wedge (h^2 \wedge (\ldots h^m \wedge (s)\ldots) = h^1 \wedge (h^2 \wedge (\ldots h^m \wedge (s^1)\ldots) = s^1 \quad \forall s \in S,$$

which completes the proof. ∎

We now describe how the notion of a cover captures decomposability. If the hyperplanes that compose the cover are defined on a small number of bits—say if all the hyperplanes in a cover on S are of size one—then each bit value can be determined in isolation, and the problem can be solved quickly by optimizing each bit in parallel. Of course, the cover size would have to be known in order to guarantee that this locates the optimal string. If, on the other hand, several hyperplanes in a cover have a large number of defined bits, then the time required to solve the subproblems may be substantial. To capture the intuition that a problem is as difficult as its largest subproblem, we define a cover's size to be the maximal number of defined bits in any hyperplane which belongs to the cover.

Definition 16. The *size* of a cover $C = \{h^1, h^2, \ldots h^m\}$ for V on S is given by $Z(C) = \max_i\{\sigma(h^i)\}$.

Example 4. $C = \{1***, *00*, **01\}$ is a cover of size 2.

Let $n(\alpha)$ equal the size of the smallest cover on $T(\alpha)$. From Claim 3.2 it follows that $n(\alpha)$ is monotonically increasing in α.

Definition 17. $n(\alpha) = \min\{Z(C) \mid C \text{ is a cover on } T(\alpha)\}$

Let $\alpha_j(V)$ equal the number of strings in the largest upper contour set that has a cover of size j. Each $\alpha_j(\cdot)$ can be thought of as a functional that maps functions defined over binary strings into the set $\{1, \ldots, 2^n\}$.

Definition 18. $\alpha_j(V) = \max\{\alpha \mid n(\alpha) \leq j\}$

Example 5. $\alpha_1(V) = 2^{n-1}$ implies there exists a cover of size 1 for V on the upper contour set consisting of all strings with function values above the median.

Claim 3.4 states that for any function $V \in F$, $\alpha_j(V)$ is weakly increasing in j. In other words, as the function value improves, the minimal cover size decreases.

Claim 3.4. *$\forall V \in F$, the following hold:*

(i) $\alpha_{j+1}(V) \geq \alpha_j(V)$;

(ii) $\alpha_n(V) = 2^n$.

Proof.

(i) Let $\hat{C}$ be a cover for V of size j on $T(\alpha_j(V))$. Trivially, $Z(\hat{C}) \leq j + 1$. The result follows.

(ii) $C = \{s^1\}$ is a cover of size n on $T(2^n)$. ∎

An implication of Claim 3.4 is that covers distinguish between encoded nonlinear interactions, those that may affect optimization, and relevant nonlinear interactions, those that do. A function may contain nonlinear terms but still have a cover of size one. A similar distinction between potential and relevant nonlinearities has been made in economics by Buchanan and Stubblebine [3]. If an encoded nonlinear effect does not create problems for optimization, then heuristics, optimization techniques, mechanisms, and algorithms developed to overcome the nonlinear effect may be unnecessary.

The following claim addresses the simplest cover:

Claim 3.5. *$\alpha_1(V) = 2^n$ if and only if $\exists C = \{h^1, h^2, \ldots, h^n\}$ forming a cover for V on S that satisfies:*

(i) $h_i^i \in \{0, 1\}$

(ii) $h_i^j = *$ *for* $i \neq j$

Proof. Suppose $\alpha_1(V) = 2^n$. Let $\hat{C} = \{\hat{h}^1, \hat{h}^2, \ldots, \hat{h}^n\}$ form a cover of size one for V on S. Choose $\tau \in \Phi$, the permutation group on m elements, such that $\hat{h}_i^{\tau(i)} \in \{0,1\}$. It follows that $C = \hat{C}$. The other direction follows immediately from the definition. ∎

Claim 3.5 can be interpreted as a decentralization (or parallel processing) result. Beginning with any string, maximizing each bit with respect to that string leads to the optimal string. Decisions as to which values to assign to bits need not be coordinated. This does not mean that V contains no nonlinear effects. In the next section, we construct functions with nonlinear terms that nonetheless satisfy the assumptions of Claim 3.5. Such functions have been characterized by Liepins and Vose [14] as *easy*.

The $\alpha_j(\cdot)$'s can be combined to form the *decomposability vector*, which measures the size of the upper contour sets that have covers of various sizes.

Definition 19. The *decomposability vector* $\alpha(V) = (\alpha_1(V), \alpha_2(V), \ldots, \alpha_n(V))$.

The decomposability vector $\alpha(V)$ can be considered as a functional that maps the set of all functions defined on S into integer-valued vectors of length n. Functions mapped to decomposability vectors with *larger* values are less difficult, that is, more decomposable, as measured by cover size, than those mapped to vectors with smaller values. Some simple examples demonstrate how $\alpha(V)$ measures decomposability.

Example 6.

$$V_1(s) = 8s_1 + 4s_2 + 2s_3 + s_4 - 10s_1 \cdot s_4$$
$$\alpha(V_1) = (4, 16, 16, 16)$$

$$V_2(s) = s_1 + s_2 + s_3 + s_4 + 8(1 - s_1) \cdot (1 - s_2) \cdot (1 - s_3)$$
$$\alpha(V_2) = (2, 2, 16, 16)$$

$$V_3(s) = 8(s_1 \cdot s_2 \cdot s_3 + s_1 \cdot s_2 \cdot s_4 + s_2 \cdot s_3 \cdot s_4) - s_1 - s_2 - s_3 - s_4$$
$$\alpha(V_3) = (5, 5, 16, 16)$$

Four features merit attention. First, V_1 has a cover of size two on all of S while the other two functions have covers of size three on S. Thus, at least according to this measure, V_1 appears most decomposable. Second, the function V_3 has a cover of size one on a larger upper contour set than either V_2 or V_1, which implies that as the function value improves, V_3 becomes easiest. Third, while V_2 has a unique cover of size 3 on S, V_3 has multiple covers of size 3. Finally, although the vector $\alpha(\cdot)$ does not create a complete ordering of functions, a function V might be said to be more decomposable than a function $\hat{V}$ if $\alpha_i(V) \geq \alpha_i(\hat{V})$, for $i = 1, \ldots, n$. According to this criterion, V_1 and V_3 could be said to be less decomposable than V_2, but no such comparison can be made between V_1 and V_3.

At this point, we should clarify that cover size measures the decomposability of a particular function. In the current formulation, we make no attempt to extend this measure to classes of functions. Our intuition suggests that members of a class of functions might have similar cover sizes; simulations described later in this paper strongly support such an hypothesis.

3.2 Decomposability as a measure

Cover size differs from other measures of nonlinearity/complexity by focusing on decomposability. Standard nonlinearity/complexity measures count the number and size of encoded nonlinear effects [11, 14]. We refer to these measures as *domain based*. In this section, we show that domain-based measures can be misleading. On the one hand, simple nonlinear interactions may combine to form complex problems. On the other hand, complicated nonlinear interactions may collapse to form easy problems. Decomposition size is perhaps the simplest domain-based complexity measure for functions defined over binary strings. Before we define decomposition size, we need to introduce the decomposition basis coefficients, which attach a value to each subset of $N = \{1, 2, 3, \ldots, n\}$ [15]. If $O(s)$ equals the subset of bits in s that have the value 1, then the value of a string equals the sum of the values of the subsets contained in $O(s)$.

Definition 20. Given $V \in F$, the *decomposition basis coefficients* $(\beta_{V,\emptyset}, \ldots, \beta_{V,I}, \ldots \beta_{V,N}) \in \Re^{2^n}$ satisfy

$$V(s) = \sum_{I \subset O(s)} \beta_{V,I} \qquad \text{where } O(s) = \{i \mid s_i = 1\}.$$

The decomposition size equals the size of the largest subset I that has a nonzero coefficient.

Definition 21. The *decomposition size* of V, $\text{size}_d(V) = \max\{|I| \mid \beta_{V,I} \neq 0\}$.

Using decomposition size as a measure, it is possible for simple nonlinear effects to combine to form problems that would be difficult to optimize for many search algorithms. In the example below, we construct a function with decomposition size equal to two that forms a problem with a cover of size n. This example can be understood in the context of a multiple public projects model [20]. Suppose there are n potential public projects and that project values are interdependent as in the introductory example with the airport, botanical garden, and cable car system. Decisions on public projects can be modeled as discrete choices, where "yes" is denoted by 1 and "no" is denoted by 0. If we let s_i represent the decision on project i, then a string represents a decision on each project.

In the example below, each individual project has a negative isolated value and each pair of projects has a positive complementarity. By inspection, the

decomposition size of this problem equals two. However, only when all n projects are undertaken does the combined value of the complementarities outweigh the negative project values, that is, the problem has a cover of size n.

Example 7. Assume $n > 2$, and choose the decomposition basis coefficients as follows:

$$\beta_{V,I} = \begin{cases} -1 & \text{if } |I| = 1 \\ \dfrac{1}{n-1} + \dfrac{1}{n-2} & \text{if } |I| = 2 \\ 0 & \text{if } |I| > 2 \end{cases}$$

Choose $s \in S$. It is straightforward to show that $V(s) < 0$ if $s \cdot s = k < n$ and that $V(s) > 0$ if $s \cdot s = n$. Therefore, the highest-valued string of projects equals the set of all projects, and the second highest-valued string of projects equals the set of no projects. It follows that n equals the minimal cover size for any upper contour set containing more than *two strings.*

Along similar lines, Goldberg [6] has shown that complex problems can arise from simple interactions. Using the Walsh Basis, Goldberg combines nonlinear interactions of decomposition size two and three to form problems that are deceptive for genetic algorithms. Roughly speaking, a problem is deceptive if hyperplanes whose strings have above average value do not contain the optimal string.

Measuring a function's difficulty by encoded effects may also *overstate* the amount of relevant nonlinearity. In the decomposition basis, maximum difficulty occurs when the coefficient of the subset of all variables has a positive coefficient, that is, when $\text{size}_d(V) = n$. The example below is a function of the decomposition size n which has a minimal cover over S of size one.

Example 8. Choose $V \in F$ such that $\beta_{V,I} > 0, \forall I \subset N$. It is straightforward to show that $\text{size}_d(V) = n$, but that V has a cover of size 1.

Though these examples suggest that cover size may be a more appropriate measure of the difficulty of decomposing a problem in order to solve it in parallel, they do not imply that cover size is a better measure than decomposition size regardless of the particular search algorithm.

3.3 Test functions

In this section, we report data from numerical simulations on test functions. We created two classes of test functions each with a decomposition size of two. We then measured the cover size of randomly drawn functions from the two classes of functions. Functions from each class have decomposition sizes

equal to two with probability one. The difference between the test functions is that the first class has smaller nonlinear terms.

$$V_1(s) = \sum_{i=1}^{n} \beta_i \cdot s_i + \sum_{i=1}^{n} \sum_{j=1}^{n} \beta_{ij} \cdot s_i \cdot s_j \qquad \beta_i \in [-2, 2],\ \beta_{ij} \in [-1, 1]$$

In the simulations both β_i and β_{ij} were uniformly distributed. We varied n, the number of bits, and found that cover size tended to increase with n. The table below shows that for $n = 6$, only 10% of the randomly generated functions of type V_1 had a cover size of six or greater, but that for $n = 10$, 88% of the functions had a cover size of six or greater. This occurs because the number of nonlinear effects increases more than linearly with n. The data from one hundred simulations are given below:

% of functions of type V_1

# Bits	Cover Size					
n	1	2	3	4	5	≥ 6
5	1	25	34	30	10	0
6	1	11	25	31	22	10
7	0	2	10	27	34	27
8	0	0	3	24	29	44
9	0	0	2	8	9	81
10	0	0	2	2	8	88

100 trials in each row

In the second class of functions, the nonlinear terms are only one fourth as large as the linear terms:

$$V_2(s) = \sum_{i=1}^{n} \beta_i \cdot s_i + \sum_{i=1}^{n} \sum_{j=1}^{n} \beta_{ij} \cdot s_i \cdot s_j \qquad \beta_i \in [-4, 4],\ \beta_{ij} \in [-1, 1]$$

As before, β_i and β_{ij} are uniformly distributed. The data from test functions randomly drawn from the second class of function are given below:

% of functions of type V_2

# Bits	Cover Size					
n	1	2	3	4	5	≥ 6
5	13	37	27	17	6	0
6	4	33	33	17	12	1
7	2	16	19	30	21	12
8	0	7	27	25	20	21
9	0	2	11	24	24	39
10	0	2	4	7	30	57

100 trials in each row

As might be expected, the smaller nonlinear effects result in smaller cover sizes and as before, the probability of the cover size being greater than or equal to six increases with the number of bits. Two hypotheses may be formulated from these data. First, we see that decomposition size and cover size may differ substantially. Second, for a fixed n, functions drawn from the same class have similar cover sizes. In the next section, we discuss how these similarities may be exploited.

4. Covers and optimization theory

In this section, we show how covers can be used to select optimal parameters for a class of hillclimbing algorithms. We also provide an alternative explanation for the performance of genetic algorithms in terms of covers.

4.1 ALGO(r, p)

In this section we define a class of hillclimbing algorithms called ALGO(r, p) where r equals the maximum number of bits selected in one iteration and p equals the probability that each bit is switched. The term ALGO is borrowed from the work of Reiter and Sherman [22]. The algorithms used here differ slightly from their algorithms. We choose to investigate simple hillclimbing algorithms for several reasons. First, hillclimbing algorithms enjoy widespread use because of their simplicity and effectiveness. For example, Lin [16] has applied variants of hillclimbing algorithms to the traveling salesperson problem with great success. Second, the number of bits that have to be flipped in combination is a crude approximation of a function's decomposability. Finally, many other algorithms such as simulated annealing rely on hillclimbing algorithms. Insights gained from analyzing the latter may also apply to the former.

In our formulation, ALGO(r, p) begins with a randomly generated string.

Definition 22. The search algorithm ALGO(r, p) is defined as follows:

Step 1: Choose $s^A \in S$
Step 2: Randomly select $I \subset N$ such that $|I| = r$
Step 3: Create s^t as follows:

$$\begin{aligned} s_i^t &= s_i^A && \text{if } i \notin I \\ s_i^t &= (1 - s_i^A) \text{ with probability } p && \text{if } i \in I \\ s_i^t &= s_i^A \quad \text{with probability } (1-p) && \text{if } i \in I \end{aligned}$$

Step 4: If $V(s^t) > V(s^a)$, then $s^A := s^t$
Step 5: Go to Step 2

ALGO(r, p) compares the current best string, s^A, to a chosen string, s^t, which differs by at most r bits. A string s^* belongs to the set of local optima with respect to ALGO(r, p) if any string differing by r bits or fewer has a strictly lower value under V.[4]

[4]Recall that we assume that no two strings have the same value.

Definition 23. A string s^* belongs to the *set of local optima* with respect to ALGO(r,p), denoted LO(r,p), if $\forall s \neq s^*$, $|\{i : s_i \neq s_i^*\}| \leq r$ and $V(s) < V(s^*)$.

Claim 4.1 below provides a sufficient condition for an ALGO(r,p) to converge to a string that belongs to a string dominant hyperplane.

Claim 4.1. *For any $h \in H$, if $\sigma(h) \leq r$ and h is string dominant for V on S, then* LO$(r,p) \subseteq S(h)$.

Proof. Choose $s^* \in$ LO(r,p). By assumption $\sigma(h) \leq r$. Therefore, it follows that $\{i \mid h \wedge s_i^* \neq s_i^*\} \leq r$. The proof is by contradiction. Suppose that $s^* \notin S(h)$. By assumption, h is string dominant. Therefore, $V(h \wedge s*) > V(s^*)$, a contradiction. ∎

A corollary of Claim 4.1 is that if a function has a cover of size less than or equal to r, then ALGO(r,p) locates the optimal string.

Corollary 4.1. *If V has a cover of size j and $j \leq r$, then* LO$(r,p) = \{s^1\}$, *the optimal string.*

Proof. Follows directly from Claim 4.1. ∎

From Corollary 4.1 we see that the vector $\alpha(V)$ reveals information about how r should be chosen. A function V_1 that satisfies $\alpha_1(V_1) = 2^n$ can be solved with ALGO$(1,p)$ for any positive p. Alternatively, a function V_2 such that $\alpha_{20}(V) = 1$ probably would not be optimized by ALGO(r,p) for r small.

An implication of Corollary 4.1 is that if the distribution of cover sizes for a particular class of functions is known or approximated, then $\alpha(V)$ can be used to compute the probability that a local optimum is global. In the previous section, the data suggested that 99% of functions of type V_2 have covers of size n or less. Suppose further that 99% of the functions of type V_2 satisfy $\alpha_2(V) > |S|/10$. A search algorithm that randomly generates thirty strings and applies ALGO$(2,p)$ to the string with the highest value would yield a global optimum with a probability strictly greater than $(.99) \cdot [1 - (.9)^{30}] = .948$. This lower bound is strict given that each new local optima moves the search to a higher contour set and increases the probability that ALGO$(2,r)$ finds the global optimum.

Cover size may help to explain why simple algorithms are often effective at finding solutions. For example, finding an optimal tour for a traveling salesperson problem (TSP) is NP-hard. Reiter and Sherman [22] and Kauffman and Levin [12] attempted to solve TSPs using algorithms similar to our ALGOs. Their algorithms switch cities within a tour. Both studies found that increasing the number of cities switched in a given iteration improved performance but that these improvements dropped off sharply. For instance, Kauffman and Levin showed that a variant of ALGO$(3,1)$ performed almost

as well as a variant of ALGO(4, 1). In other words, their TSPs may have had covers of size 3 for large portions of the domain.

The performance of ALGO(r, p) depends both on the parameter p and on r, the maximal number of bits switched. The following example shows how changing r changes the probability of locating a particular hyperplane.

Example 9. Let $n = 5$ and suppose $h = 11***$ is string dominant for V on S. Let $s^A = 00000$ and consider the performance of ALGO$(2, 1/2)$:

ALGO$(2, 1/2)$:

In Step 2, $\Pr\{d(h) \subset I\} = 0.1$

In Step 3, $\Pr\{s_i^t = 1 \Leftrightarrow i \in d(h)\} = 0.25$

Therefore, after Step 3, $\Pr\{s_i^t \in h\} = 0.025$.

Claim 4.3 states a more general result. The idea of the claim is that to maximize the probability of switching bit values on a given subset of size k, the optimal r for a given ALGO$(\cdot, p)$ varies directly with k and inversely with p. We first define the probability of switching a given subset using ALGO(r, p).

Definition 24. The *probability of switching* h in ALGO(r, p) is given by

$$\Pr(h, \text{ALGO}(r,p)) = \Pr\{d(h) = \{i : s_i^t \neq s_i^A\} \mid \text{ALGO}(r,p)\}.$$

Claim 4.2. *If* $\sigma(h) = k$ *and* $r \geq k$, *then*

$$\Pr(h, \text{ALGO}(r,p)) = p^k \cdot (1-p)^{r-k} \cdot \frac{r!\,(n-k)!}{n!\,(r-k)!}.$$

Proof. In Step 2 of ALGO(r, p),

$$\Pr\{\sigma(h) \subset I\} = \frac{r!\,(n-r)(n-k)!}{n!\,(n-r)!\,(r-k)!} = \frac{r!\,(n-k)!}{n!\,(r-k)!}$$

In Step 3 of ALGO(r, p), $\Pr\{s_i^t = s_i^A \Leftrightarrow i \in d(h)\} = p^k \cdot (1-p)^{(r-k)}$, which completes the proof. ∎

Claim 4.3. *If* $\sigma(h) = k$, *then* $\operatorname{argmax} \Pr(h, \text{ALGO}(\cdot, p)) = \lfloor k/p \rfloor$, *where* $\lfloor a \rfloor$ *equals the greatest integer less than or equal to* a.

Proof. From Claim 4.2,

$$\Pr(h, \text{ALGO}(r,p)) = p^k \cdot (1-p)^{r-k} \cdot \frac{r!\,(n-k)!}{n!\,(r-k)!}.$$

Let

$$g(r, k, p) = p^k \cdot (1-p)^{r-k} \cdot \frac{r!}{(r-k)!}.$$

It follows that

$$\Pr(h, \text{ALGO}(r,p)) = g(r,k,p) \cdot \frac{(n-k)!}{n!}.$$

Therefore, it suffices to maximize $g(r,k,p)$ with respect to r. Let

$$G(r) = \frac{g(r,k,p)}{g(r-1,k,p)} = r \cdot \frac{1-p}{r-k}.$$

Therefore, $G(r) \geq 1 \Leftrightarrow r \leq k/p$, and further, $G'(r) < 0$, which completes the proof. ∎

The corollary below states that p should be set equal to one and r equal to $\sigma(h)$ to maximize $\Pr(h, \text{ALGO}(r,p))$.

Corollary 4.2. *If* $\sigma(h) = k$, *then* $\Pr(h, \text{ALGO}(k,1)) \geq \Pr(h, \text{ALGO}(r,p))$ *for all* $r \in N$ *and* $p \in [0,1]$.

Proof. From Claim 4.3,

$$\Pr(h, \text{ALGO}(r,p)) = g(r,k,p) \cdot \frac{(n-k)!}{n!}$$

where

$$g(r,k,p) = p^k \cdot (1-p)^{r-k} \cdot \frac{r!}{(r-k)!}.$$

Note that $g(r,k,p) = (1/k!) \cdot f(k : r,p)$, where $f(k : r,p)$ is the standard binomial distribution. It follows immediately that $r = k$ and $p = 1$ maximizes f, which completes the proof. ∎

Corollary 4.2 does not prove how quickly a particular algorithm will locate the optimal string. It merely provides intuition as to how r should be chosen in $\text{ALGO}(r,p)$ in conjunction with the decomposability vector $\alpha(V)$. If a function V has a cover on S of size 8, and a cover on $T(2^{n-2})$ of size 4, then initially, r should be large to find the hyperplanes of larger size. Once these hyperplanes are located and the function value has improved, r should be decreased. This same idea underpins the simulated annealing algorithm [18, 4] in which a temperature is lowered over time. The decrease in temperature decreases the number of bits that are likely to be flipped. This idea is discussed at length in Page [21].

4.2 Genetic algorithms

A genetic algorithm (GA) is a constant-size population based search algorithm [8, 5]. Recently, GAs have been used in economics [1, 17, 9], political science [13], and in the study of inductive learning [10]. In this section we provide a brief introduction to GAs and use cover theory to develop an alternative explanation for their performance.

A GA begins with a population of M strings that it transforms into a new population. Each iteration of a GA is called a generation. We denote the population in generation t by p_t. The transition from p_t to p_{t+1} occurs in three stages: reproduction, crossover, and mutation. Reproduction chooses a new population of M strings from the existing population according to the function values of the strings. The idea is that better (more "fit") strings are reproduced with greater frequency, thus increasing the fitness of the population. Let $p_{t'}$ be the population after reproduction. The reproduction we describe relies on a tournament to select strings.

Tournament Selection: M pairs of strings from p_t are randomly created. The higher valued of the two strings in each pair belongs to $p_{t'}$.

A crossover operator creates new strings by "crossing" strings from among those reproduced. The analogy to be kept in mind is genetic recombination with the bit values thought of as alleles. The bit value of a string created during crossover may come from either parent. Crossover randomly creates M pairs of strings, and each pair independently crosses bits with probability p (typically $p \in [.25, .75]$). The crossing or exchanging of bit values occurs on a subset of the positions. The positions which cross bit values can be chosen in many ways. Three typical crossover rules are *one point*, *two point*, and *uniform crossover*. In the simulations described later in this section, we employ uniform crossover.[5]

Uniform crossover: A "switching rule" x is created by selecting a string from the set S. If $x_i = 1$, then the strings s and t switch their ith bit values. If $x_i = 0$, then the strings s and t do not switch their ith bit values.

Bit:	$x_1\ x_2\ x_3\ x_n$
Switching rule:	0 1 1 0
new s	$s_1\ t_2\ t_3\ s_n$
new t	$t_1\ s_2\ s_3\ t_n$

[5]There are two reasons for this decision. First, with one- or two-point crossover, hyperplanes with large distances between defined bits are more likely to get destroyed, making the distance between defined bits a more natural definition of hyperplane size. With uniform crossover, it can be shown that the more defined bits in a hyperplane, the measure used in cover size, the greater the probability that the hyperplane is destroyed during crossover. Second, uniform crossover treats all bits symmetrically, one and two-point crossover have endpoint bias effects. For a more complete discussion of crossover rules see Goldberg [5].

Crossover can "destroy" hyperplanes. A hyperplane h is destroyed during crossover if one of the strings belonged to h prior to crossover, but neither string belongs to h after crossover.

Example 10. *Suppose* $h = *0*1$. *Let* $s = 0011$ *and* $t = 0110$. *It follows that* $s \in h$ *and* $t \notin h$. *Suppose that* s *and* t *are uniformly crossed using the switching rule* 1100. *Neither resulting string,* $s' = 0010$ *or* $t' = 0111$ *belongs to* h.

Finally, mutation may occur at the bit level or the string level. Bit mutation switches the value of each bit of each string with a very small probability ($p < 0.1$). Bit mutation, which is analogous to biological mutation, creates both short and long term effects. In the short term, a mutant represents a single random search for a better string. In the long term, a bit mutation may spread through the population and slow the rate of convergence of bit values. In contrast to the milder bit mutation, string mutation creates an entirely random string. Typically, the probability of string mutation is also small ($p < 0.05$). String mutation allows for random searches in entirely different regions of the domain, which prevents the GA from getting stuck at a local peak. After mutation, the new population, p_{t+1}, is completed; reproduction, crossover, and mutation are reapplied to create p_{t+2}. The randomness created in the mutation stage guarantees that the population never converges to a single string replicated M times.

The performance of GAs has been the focus of a great deal of research in recent years. To date, the most important result in the theory of GA performance is the Schema Theorem [8], which says that hyperplanes whose strings consistently have higher than average function values will increase in number in the population proportionate to their fitness advantage in the population. Decreasing a hyperplane's size (or defining length for one and two-point crossover) increases the probability that a hyperplane survives crossover.

One interpretation of the Schema Theorem is the "building block" hypothesis, which says that GAs increase the number of hyperplanes of small size whose strings have above average value and combine them to form above average strings. These hyperplanes are commonly referred to as "building blocks." A hyperplane is above average if the strings in the population that lie in the hyperplane have, on average, higher value than the strings not in the hyperplane.

Cover theory offers an alternative perspective on GA performance. Claim 4.3 states that to maximize the probability of switching exactly the defined bits in a hyperplane h, the optimal size of r ALGO(r, p) depends upon the number of defined bits of h. Claim 3.4 states that for any function, the cover size decreases as the function value improves. Taken together these two claims suggest that a good search algorithm should switch fewer bits as the search climbs up the contour sets and the cover size decreases.

Ideally, a search algorithm would adapt the number of bits that it switches as it climbs the upper contour sets. Optimal adaptation of the number of bit

flips requires information about the function's contour sets. We hope to show that a GA learns the function's contour sets and adapts the number of bits it switches. To summarize, there are two hypotheses that we investigate: first, that a genetic algorithm reduces the number of bits switched as it moves up a function's contour sets; and second, that the rate at which this decrease occurs depends on the function. Specifically, if the cover size is small or if it decreases rapidly as the function value increases, then a genetic algorithm should decrease the number of bits switched.

We construct three test functions and one class of random functions on binary strings of length fifty. The function V_1 is linear, so it has cover of size one on all of S. We would expect that the number of bits a genetic algorithm switches would decrease rapidly as the function value improves. To investigate arbitrarily difficult functions we create a class of random functions, V_2. Functions in V_2 assign values to strings randomly using the uniform distribution on $[0, 1]$. The cover size for such functions should be close to n, the number of bits, for fairly large upper contour sets. The number of bits switched by a genetic algorithm should decrease slowly as the function value improves. The linear function and the class of random functions by themselves are not very informative. Their value is in providing benchmarks that let us compare a genetic algorithm's rate of convergence on the other two functions.

The other two functions we consider are constructed from a linear function, $f(s)$, and a nonlinear function $g(s)$.

$$f(s) = \sum_{i=1}^{50} s_i$$

$$g(s) = \sum_{i=1}^{8} 3 \cdot s_{2i} \cdot s_{2i+1}$$

The function V_3 equals $f - g$ if the value of f is less than or equal to $n/2$ and f otherwise:

$$V_3(s) = \begin{cases} f(s) - g(s) & \text{if } f(s) \leq n/2 \\ f(s) & \text{if } f(s) > n/2 \end{cases}$$

The function V_3 is linear at the top but rugged for smaller function values. In contrast, the function V_4 is linear for small values and rugged near the peak. The function V_4 equals $f - g$ if the value of f is greater than $n/2$ and f otherwise:

$$V_4(s) = \begin{cases} f(s) & \text{if } f(s) \leq n/2 \\ f(s) - g(s) & \text{if } f(s) > n/2 \end{cases}$$

The data presented comes from a tournament selection genetic algorithm using uniform crossover with probability 0.5 and bit mutation with probability 0.04. We have carried out computations with many other parameter choices and similar findings obtain. Table 1 contains means over 100 trials with standard errors of the distribution, *not of the estimated mean*, in parentheses. The data is shown graphically in Figure 1.

Gen #	Linear V_1		Random $V \in V_2$		Low Rugged V_3		Up Rugged V_4	
1	12.119	(1.075)	12.295	(0.871)	11.435	(1.217)	11.342	(2.225)
10	5.652	(1.213)	10.558	(1.026)	7.774	(2.105)	8.480	(1.063)
20	1.373	(0.842)	9.288	(1.011)	1.871	(0.284)	7.424	(0.438)
30	0.143	(0.223)	8,748	(1.212)	0.711	(0.653)	5.074	(1.796)
40	0.163	(0.213)	8.206	(1.158)	0.407	(0.706)	5.099	(2.334)
50	0.241	(0.234)	8.069	(1.352)	0.516	(0.383)	3.284	(1.574)
60	0.282	(0.298)	7.859	(1.352)	0.289	(0.269)	4.200	(2.356)
70	0.270	(0.293)	7.786	(1.312)	0.529	(0.804)	3.140	(2.002)
80	0.285	(0.333)	7.890	(1.286)	0.435	(0.142)	1.925	(0.956)
90	0.269	(0.279)	7.864	(1.371)	0.377	(0.318)	1.813	(0.979)
100	0.315	(0.330)	1.908	(1.164)	0.520	(0.373)	1.320	(0.452)
110	0.329	(0.332)	7.860	(1.287)	0.601	(0.399)	1.643	(1.459)
120	0.332	(0.306)	7.714	(1.362)	0.536	(0.531)	1.065	(1.308)
130	0.357	(0.295)	7.873	(1.381)	0.184	(0.193)	1.475	(2.253)
140	0.355	(0.323)	7.820	(1.361)	0.382	(0.352)	1.415	(2.319)
150	0.353	(0.321)	7.771	(1.296)	0.346	(0.300)	1.104	(1.912)
160	0.302	(0.330)	7.712	(1.361)	0.379	(0.483)	1.049	(1.752)
170	0.282	(0.303)	7.563	(1.324)	0.224	(0.204)	1.356	(2.348)
180	0.283	(0.303)	7.765	(1.311)	0.215	(0.290)	1.633	(2.829)
190	0.339	(0.323)	7.721	(1.302)	0.128	(0.160)	1.800	(3.118)
200	0.344	(0.310)	7.768	(1.358)	0.342	(0.346)	0.975	(1.631)

Table 1: Number of bits switched.

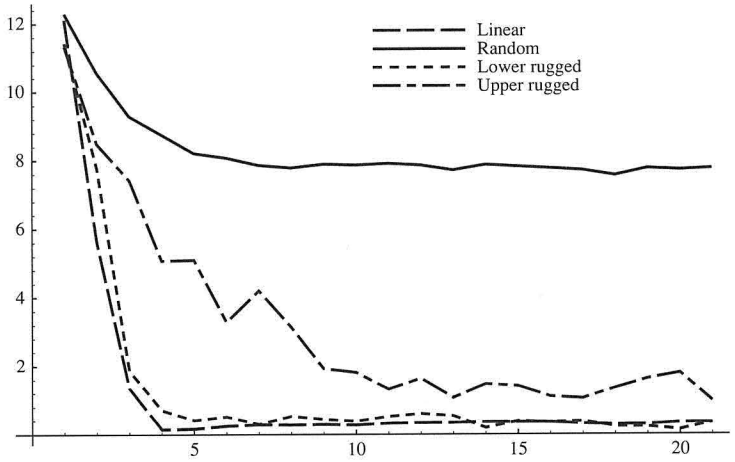

Figure 1: Average number of bits switched.

These data capture several features worth mentioning. Most obvious is that the average number of bits flipped decreases for all functions while the average value of strings in the population increases for all functions (data not shown). As is normally the case, the genetic algorithm switches fewer bits as the function value improves. From Claim 3.4, we know that the cover size decreases as the function value improves, which partially explains why genetic algorithms have been so successful. More importantly, these data support our hypothesis that a genetic algorithm adapts the number of bits switched during crossover in response to the function's cover size.

The number of bits switched is largest for the random functions in V_2, which have the largest cover size over S, and smallest for the linear function V_1 as predicted. The number of bits switched on average during crossover for functions V_3 and V_4 is less than the number switched for the random function but greater than the number switched for the linear function. A comparison between the number of bits flipped by these two functions also yields the expected result: more bits are flipped for function V_4 than for function V_3. Applied to V_3, the genetic algorithm locates strings above the rugged foothills ($f(s) > n/2$) and within seventy generations, the number of bits switched averages approximately one-half, which is only double the number switched for the linear function. Applied to V_4, the genetic algorithm continues to switch many bits for many generations. In generation seventy, on average about three bits are switched during each application of the crossover operator. This occurs because of the ruggedness in the upper contour sets. Although these simple examples give reason to be optimistic about a relationship between the decomposability vector and the rate of convergence of a genetic algorithm, more test functions should be examined before drawing any firm conclusions.

5. Discussion

In this paper, we have shown how cover theory formalizes the idea that functions can be decomposed into simpler subproblems that can be solved in parallel. Covers can be used to construct a decomposability vector that measures a function's difficulty relative to its contour sets. Finally, we have seen that covers can be used to explain the performance of search algorithms. A potential shortcoming of cover theory is the severity of its assumptions. In response, Richardson [23] has advanced the idea of ϵ-covers, a less restrictive construction. An ϵ-cover consists of "almost" string dominant hyperplanes—hyperplanes that are string dominant except on a small subset. To formalize this notion, define h as ϵ-string dominant on $T(\alpha)$ if, for any $\hat{\alpha} \leq \alpha$, the proportion of strings in $T(\hat{\alpha})$ that satisfy $V(h \wedge s) \geq V(s)$ is greater than $(1 - \epsilon)$. A consequence of this definition is that the number of strings for which an ϵ-string dominant hyperplane is not string dominant must decrease as the search moves to higher contours. An ϵ-cover is a collection of ϵ-string dominant hyperplanes the union of whose defined bit is N.

Acknowledgments

The author would like to thank Stan Reiter, David Richardson, and Jim Jordan for their insights.

References

[1] J. Arifovic, "Learning by Genetic Algorithms in Economic Environments," Santa Fe Institute Working Paper 90-001 (1989).

[2] A. Bethke, "Genetic Algorithms as Function Optimizers," Doctoral Dissertation, The University of Michigan. *Dissertation Abstracts International*, **41**(9) (1988) 3503B.

[3] J. Buchanan and C. Stubblebine, "Externality," *Economica*, (1962) 371–384.

[4] L. Davis, *Genetic Algorithms and Simulated Annealing* (San Mateo, CA: Morgan Kauffman, 1987).

[5] D. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning* (Reading, MA: Addison-Wesley, 1989).

[6] D. Goldberg, "Construction of High-order Deceptive Functions Using Low-order Walsh Coefficients," IlliGAL Report No. 90002, Department of General Engineering, University of Illinois (1990).

[7] J. Greffenstette and J. Baker, "How Genetic Algorithms Work: A Critical Look at Implicit Parallelism," in *Proceedings of the Third International Conference on Genetic Algorithms*, edited by J. Schaffer (San Mateo, CA: Morgan Kauffman, 1989).

[8] J. Holland, *Adaptation in Natural and Artificial Systems* (Ann Arbor: University of Michigan Press, 1975).

[9] J. Holland and J. Miller, "Artificial Adaptive Agents in Economic Theory," Proceedings of the American Economic Association (1991).

[10] J. Holland, K. Holyoak, R. Nisbett, and P. Thagard, *Induction: Processes of Inference, Learning, and Discovery* (Cambridge, MA: MIT Press, 1989).

[11] S. Kauffman, "Adaptation on Rugged Fitness Landscapes," pages 527–619 in *Lectures in the Sciences of Complexity* (Reading, MA: Addison-Wesley, 1989).

[12] S. Kauffman and S. Levin, "Towards a General Theory of Adaptive Walks on Rugged Landscapes," *Journal of Theoretical Biology*, **128**(11) (1987).

[13] K. Kollman, J. Miller, and S. Page, "Adaptive Parties in Spatial Elections," *American Political Science Review*, **86**(4) (1992) 929–937.

[14] G. Liepins and M. Vose, "Representational Issues in Genetic Optimization," *Journal of Experimental and Theoretical Artificial Intelligence*, **2**(2) (1990) 4–30.

[15] G. Liepins and M. Vose, "Polynomials, Basis Sets, and Deceptiveness in Genetic Algorithms," *Complex Systems*, **5** (1991) 45–62.

[16] S. Lin, "Computer Solutions of the Traveling Salesman Problem," *The Bell System Technical Journal* (1965) 2245–2269.

[17] R. Marimon, E. McGrattan, and T. Sargent, "Money as a Medium of Exchange in an Economy with Artificially Intelligent Agents," *Journal of Economic Dynamics and Control* (1990).

[18] R. Otten and L. van Ginnekan, *The Annealing Algorithm* (Boston: Kluwer, 1989).

[19] S. Page and D. Richardson, "Walsh Functions, Schema Variance, and Deception," *Complex Systems*, **6** (1992) 125–135.

[20] S. Page, "A Bottom-up Efficient Algorithm for Allocating Public Projects with Positive Complementarities," California Institute of Technology working paper #885 (1994).

[21] S. Page, "Two Measures of Difficulty," California Institute of Technology (1994).

[22] S. Reiter and G. Sherman, "Discrete Optimizing," *Journal of the Society of Industrial and Applied Mathematics*, **13**(3) (1965).

[23] D. Richardson, private communication 1991.

[24] H. Simon, *The Sciences of the Artificial* (Cambridge, MA: MIT Press, 1969).