Analyzing the Effects of Module Encapsulation on Search Space Bias

Ozlem O. Garibay
Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL 32816-2362
ozlem@cs.ucf.edu

Annie S. Wu
Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL 32816-2362
aswu@cs.ucf.edu

ABSTRACT

Modularity is thought to improve the evolvability of biological systems [18, 22]. Recent studies in the field of evolutionary computation show that the use of modularity improves performance and scalability of evolutionary algorithms for certain applications. [5, 12, 15, 16, 17]. The effects of introducing modularity to evolutionary search, however, are not well understood. This paper focuses on analyzing the effects of modularity on evolutionary computation. In particular, we analyze the effects of modular representations on the search space bias.

Categories and Subject Descriptors

I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods —*Representations (procedural and rule-based)*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*

General Terms

Algorithms

Keywords

Modularity, Search Algorithms, Genetic Algorithms

1. INTRODUCTION

In this paper, we analyze the effects of module encapsulation on the structure of search spaces. The introduction of modules in a problem representation can alter the structure of the corresponding search space, which in turn may affect how easy or hard it is for a evolutionary algorithm (EA) to find a solution. Specifically, we want to study the following aspects of search space structure: the size of the search space and the average distance to a solution.

Previous research on modularity in EAs focuses primarily on developing effective empirical techniques to discover, encapsulate and use modules in EAs. Modularity is considered to be an important factor in achieving complexity and scalibility [5, 12, 15, 16, 17]. When incorporating modularity into the representations of an EA, the two fundemantal questions that we need to answer are: how many modules are there and what are the compositions of each of the modules. The definition of modules can be defined statically at the beginning or dynamically during a search. Approaches that have been studied include keeping the number and content of modules static [6, 7, 12, 13], keeping the number of modules static but evolving the content of modules dynamically [15, 19], and evolving both the number of the modules and the content dynamically [2, 20, 21]. Static definition of the number and content of modules requires a priori knowledge of the solution in order to define useful modules.

Previous studies have used both explicit and implicit methods to discover the composition of modules. Explicit methods introduce special mechanisms into an evolutionary algorithm for module discovery and encapsulation. For example, Rosca and Ballard [21] use frequency and fitness evaluation to discover candidate modules in their population. Alternatively, candidate modules may be selected and encapsulated randomly [3, 4]. In both cases, the modules are then evaluated based on their usage and performance and the better ones are kept and the rest are released into their original form. Cooperative coevolution uses subpopulations each of which evolves a single module [20]. Representative modules from each subpopulation are then combined into a complete solution for fitness evaluation. Implicit methods, on the other hand, use problem representations and operators that allow modules to emerge in the representation [10, 23]. For example, the use of dynamically evolved non-coding regions in representations is one way to achieve this. The insertion of the non-coding regions during the evolutionary process changes the relative distance of the encoded information. Genes that are closer together are less likely to be split apart by crossover and implicitly form modules.

In this paper, we provide a basic combinatorics analysis of the search space structure before and after module encapsulation. First, we develop a framework for modular seach spaces. In this framework, encapsulating a module is seen as a search space transformation. Using this framework, we analyze the changes in the distance to a solution. Using the results of our analysis, we devise an indicator that can be used to predict beneficial module encapsulation. Our empirical analysis confirms that this indicator is able to predict the effects of encapsulating a module. Based on this analysis, we are able to predict, under certain assumptions, when and why encapsulating a module will be beneficial for the search.

GECCO'07, July 7–11, 2007, London, England, United Kingdom.

2. BASIC DEFINITIONS

We will begin by defining a framework for modular search spaces upon which we can build our mathematical model. Using this framework, we can analyze search space differences between a canonical GA and a modified GA augmented with modules. The definitions and analyses assumes a binary representation, but can be extended to multicharacter representations. In our definitions, we use standard set theory and formal languages notation, i.e. see [11].

2.1 Search Spaces

The basic components of our framework are the genotype space, the phenotype space, and the rules that map from the genotype to the phenotype space. *Primitives* are the atomic components of problem representation that are used to encode solutions. *Modules* are substrings of interest and may contain two kinds of symbols: primitives and previously defined module names. Our module definition is based on the previous work **??**

The *genotype space* or *representation space* is the space of all the strings that encode candidate solutions. The elements of the genotype space are strings over the alphabet of primitives and module names. A *genotype space* $\mathcal{S}_g$ is a 5-tuple:

$$\mathcal{S}_g = \langle \mathcal{P}, \mathcal{M}, \Sigma_g, l, \mathcal{R} \rangle$$

where, $\mathcal{P}$ is a set of primitive symbols; $\mathcal{M}$ is a set of module symbols; $\Sigma_g \subseteq \mathcal{P} \cup \mathcal{M}$ is the genotype space alphabet; l is the length of all genotype strings when expanded; and $\mathcal{R}$ is the set of module defining rules.

The *phenotype space* $\mathcal{S}_p$ is the space of all possible candidate solutions. The elements of the phenotype space are strings over the alphabet of primitives only and are fixed in length.

For a given genotype space $\mathcal{S}_g = \langle \mathcal{P}, \mathcal{M}, \Sigma_g, l, \mathcal{R} \rangle$, the set of module defining rules, $\mathcal{R}$, is a set of rewriting rules:

$$\mathcal{R} = \{M_1 \rightarrow w_1, ..., M_i \rightarrow w_i, ...\}$$

where, $M_i \rightarrow w_i$ is a rewriting rule defining module M_i; $M_i \in \mathcal{M}$ is a symbol naming the module; $w_i \in \{\mathcal{P} \cup \{M_1, M_2, ..., M_{i-1}\}\}^*$ is the module defining string; and $|w_i| \leq l$, since we consider modules to be substrings of candidate solutions. There is one defining rule in $\mathcal{R}$ for each module symbol in $\mathcal{M}$, hence $|\mathcal{R}| = |\mathcal{M}|$. We define the *size* of a module to be the length of its defining string $|w_i|$. A module is of *order* zero if its defining substring consist solely of primitives, and it is of order n if its defining substring consist of primitives and symbols naming modules of at most order $n-1$.

Module defining rules are used to expand a genotype into a phenotype through an iterative process of replacing module names with their corresponding definitions until a candidate solution consisting of only primitives is obtained. Let us assume $\mathbf{s}$ be a string over $\{\mathcal{P} \cup \mathcal{M}\}$ for some genotype space $\mathcal{S}_g = \langle \mathcal{P}, \mathcal{M}, \Sigma_g, l, \mathcal{R} \rangle$. We define the expanded form of string $\mathbf{s}$ as $\text{Expand}_{\mathcal{R}}(s)$, where $\text{Expand}_{\mathcal{R}} : \{\mathcal{P} \cup \mathcal{M}\}^* \mapsto \mathcal{P}^*$ is the expanding function for module defining rules $\mathcal{R}$. The expanding function applies the rewriting rules in $\mathcal{R}$ to its input $\mathbf{s}$ until a string solely over $\mathcal{P}$ is obtained. That string is the output of $\text{Expand}_{\mathcal{R}}$. In this case, we say that $\text{Expand}_{\mathcal{R}}(s)$ has been *generated* from $\mathbf{s}$ using rewriting rules $\mathcal{R}$. Notice that, our definition of $\mathcal{R}$ guarantees that for any string $\mathbf{s} \in \{\mathcal{P} \cup \mathcal{M}\}^*$, $\text{Expand}_{\mathcal{R}}(s) \in \mathcal{P}^*$ can be computed in a finite amount of rewriting rule applications. Thus, the elements of the genotype space $\mathcal{S}_g$, denoted by $L(\mathcal{S}_g)$, are all strings over the genotype space alphabet Σ_g that expand to strings of length l over $\mathcal{P}^*$:

$$L(\mathcal{S}_g) = \{s \in {\Sigma_g}^* \mid |\text{Expand}_{\mathcal{R}}(s)| = l\}$$

Expanding all of the elements in genotype space into their phenotype form gives us the phenotype space $\mathcal{S}_p$. We define the elements of the the phenotype space, $\mathcal{S}_p$, denoted by $L(\mathcal{S}_p)$ as the following multiset [1].

$$L(\mathcal{S}_p) = \{| \text{ Expand}_{\mathcal{R}}(s) \mid s \in L(\mathcal{S}_g) |\}$$

where, $L(\mathcal{S}_g)$ be the elements of genotype space $\mathcal{S}_g$. $L(\mathcal{S}_p)$ is the multiset of all strings in the genotype space $\mathcal{S}_g$ in their expanded form. For the remainder of this paper, references to *search space* will refer to the phenotype space.

As we will see, multiple elements in $L(\mathcal{S}_p)$ may expand to the same element in $L(\mathcal{S}_p)$ The multiplicity [2] of each element in the phenotype space is determined by the number of genotypes that expand to the same phenotype. Hence, by definition, the size of the genotype space is equal to the size of the phenotype space, $|L(\mathcal{S}_g)| = |L(\mathcal{S}_p)|$. Notice that, also by definition, the elements of $L(\mathcal{S}_p)$ are fixed length l strings over $\mathcal{P}$ and the elements of $L(\mathcal{S}_g)$ are variable length strings over Σ_g. Clearly, $\text{Expand}_{\mathcal{R}}$ is the function used to map the genotype to the phenotype on our modular search spaces. Therefore, for this paper, the genotype to phenotype mapping is a generative process determined solely by the module creating rules in $\mathcal{R}$.

2.2 Module Encapsulation

Module encapsulation or *module creation* is the process of naming a substring of interest with a new alphabet symbol. This process changes the structure of the search space by adding a new element to the genotypic alphabet and, more fundamentally, by adding a new rule to $\mathcal{R}$. *Encapsulation* of a module, $\mathcal{E} : \mathcal{S}_g \times R_k \mapsto \mathcal{S}_g$ is defined as follows:

$$\mathcal{E}(\mathcal{S}_g, M_k \rightarrow w_k) = \langle \mathcal{P}, \mathcal{M} \cup \{M_k\}, \Sigma_g \cup \{M_k\}, l, \mathcal{R} \cup \{M_k \rightarrow w_k\} \rangle$$

where, $\mathcal{S}_g = \langle \mathcal{P}, \mathcal{M}, \Sigma_g, l, \mathcal{R} \rangle$ is a genotype space, $M_k \rightarrow w_k$ is the rewriting rule defining the new module to be encapsulated, M_k is a new module symbol, w_k is a string over $\{\mathcal{P} \cup \mathcal{M}\}$, and $|w_k| \leq l$.

Given a module length l_m, the complete m-module set is the set of all possible permutations of primitives of length l_m. The complete-$\mathcal{P}$ concept is used to define Complete m-module set.

We can calculate the size of a search space that includes σ_m modules in its alphabet using the following equation.

$$|L(\mathcal{S})| = \sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} 2^{l - l_m n} \sigma_m^n \binom{l - (l_m - 1)n}{n} \qquad (1)$$

where $\lfloor \frac{l}{l_m} \rfloor$ is the maximum number of modules length of l_m that a string of length l can include. Notice that, the formula applies only on search spaces where all modules have size l_m. The proof for Equation 1 is given in [8].

[1] A multiset or a bag is a set that allow repeated elements and it is denoted by $\{| \, |\}$

[2] Let N be a multiset and $m \in N$, The multiplicity of m, denoted by $|N|_m$, is the number of times the element of m is repeated in multiset N.

2.3 Analysis Metric

For this study, we will examine the effects of module encapsulation on the average Hamming distance from all elements in a search space to the solution. We focus this analysis on binary representations and make the assumption that there is fitness distance correlation (FDC) in the problem representation. Hamming distance is suitable to measure the distance in binary representations. Using the FDC framework with Hamming distance is one way to predict the performance of the mutation based algorithms such as genetic algorithms [1, 14]. Therefore, using Hamming distance is appropriate to analyze the effect of module encapsulation on the search.

Suppose $p_x, p_y \in L(\mathcal{S}_p)$, then the distance between p_x and p_y in phenotype space is

$$\Delta(p_x, p_y) = Hamming\ distance(p_x, p_y)$$

where, the Hamming distance between two strings is simply the number of positions at which the strings differ in value. The average Hamming distance to the solution, denoted by $\Delta_{avg}(s)$, is a measure of the average distance between a solution string $\Delta_{avg}(s)$ and every other string in the search space. In a uniformly distributed binary search space, it is easy to see that the expected value of $\Delta_{avg}(s)$ will be $l/2$. If $\Delta_{avg}(s)$ decreases after a search space transformation, the elements of the new search space are, on average, closer to the solution. If $\Delta_{avg}(s)$ increases, the elements of the new search space are, on average, further away from the solution.

Let us define the average Hamming distance to solution formally. Assume that $p, s \in L(\mathcal{S}_p)$ and that s is the solution string. The average Hamming distance to solution s is:

$$\Delta_{avg}(s) = \frac{\sum_{\{p \in L(\mathcal{S}_p)\}} \Delta(p, s)}{|L(\mathcal{S}_p)|}$$

Next, let us assume that $\mathcal{S}_{p1}$ is a search space with a solution string s, and $\mathcal{S}_{p2} = \mathcal{E}(\mathcal{S}_{p1})$. In addition, $\Delta_{avg1}(s)$ and $\Delta_{avg2}(s)$ are the average Hamming distances to solution in $\mathcal{S}_{p1}$ and $\mathcal{S}_{p2}$, respectively. We consider three cases. $\mathcal{S}_{p2}$ has a positive relative search space distance bias or, simply, a positive distance bias with respect to $\mathcal{S}_{p1}$ iff

$$\Delta_{avg1}(s) > \Delta_{avg2}(s)$$

$\mathcal{S}_{p2}$ has a negative distance bias with respect to $\mathcal{S}_{p1}$ iff

$$\Delta_{avg1}(s) < \Delta_{avg2}(s)$$

$\mathcal{S}_{p2}$ has no distance bias with respect to $\mathcal{S}_{p1}$ iff

$$\Delta_{avg1}(s) = \Delta_{avg2}(s).$$

In other words, assuming $\mathcal{S}_{p2} = \mathcal{E}(\mathcal{S}_{p1})$, a positive distance bias of the resulting space relative to the initial space indicates that the module encapsulation results in a space with elements that are, on average, closer to the solution. On the contrary, a negative distance bias indicates that the module encapsulation results in a space with elements that are, on average, farther away from the solution.

We give a definition for the probability density function of the distance to the solution. The probability density function, r_d, is defined as:

$$i_d = \frac{\Sigma_{p \in L(\mathcal{S}_p)}[\delta_d(p, s)]}{|L(\mathcal{S})|}$$

where

$$\delta_d(p, s) \begin{cases} 1 & \text{if } \Delta(p, s) = d \\ 0 & \text{otherwise} \end{cases}$$

and $\Delta(p, s)$ is the Hamming distance between an arbitrary individual p and the solution string s. r_d is the probability of a string having a Hamming distance of d from the solution string.

Last, we use the following equation to calculate average Hamming distance:

$$\Delta_{avg} = \sum_{d=0}^{l} P(d)d \tag{2}$$

where $P(d) = i_d$ is the probability of a randomly chosen string having a Hamming distance of d from the solution.

3. THEORETICAL ANALYSIS

We study the effects of fixed length modules on distance bias in a search space in two ways. First, if we do not know which modules of a given length are good and which modules are bad, one approach is to make available all modules of a given length in the problem representations. This analysis examines the complete module set as defined in section 2.2. A more ideal situation is when we have information about the problem that allows us estimate what could be *good* and *bad* modules. We repeat the same analysis for the two cases in which we encapsulate only good modules and only bad modules to study the effects of using modules that are or are not partial solutions.

3.1 Complete Module Set Encapsulation

If we cannot distinguish good and bad modules, one approach is to include all modules and allow the search algorithm to dynamically select which ones to use. Let $\mathcal{S}_0$ and $\mathcal{S}_1$ be the search spaces before and after the encapsulation, respectively. Let $\Sigma_0 = \{0, 1\}$ be the alphabet of search space $\mathcal{S}_0$. $\mathcal{E}(\mathcal{S}_0) \rightarrow \mathcal{S}_1$. Let Σ_1 be the alphabet of $\mathcal{S}_1$ and $\Sigma_1 = \{0, 1, M_1, M_2, ..., M_k\}$ where $k = 2^{l_m}$ and $|M_i| = l_m$ for $1 \leq i \leq k$. Also, let l be the phenotype length and l_m be the module length.

LEMMA 1. *The average Hamming distance to solution in search space $\mathcal{S}_1$ can be calculated using the following formula:*

$$\Delta_{avg} = \frac{\sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \binom{l-(l_m-1)n}{n}}{|L(\mathcal{S})|} \sum_{d=0}^{l} \binom{l}{d} d$$

PROOF. By Equation 2

$$\Delta_{avg} = \sum_{d=0}^{l} P(d)d$$

where $P(d) = i_d$.

Let us derive i_d for a search space where a complete module set is encapsulated and included in the alphabet. $\binom{l}{d}$ gives the number of ways to choose d bits that differs from the solution string in a string of length l and $\sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \binom{l-(l_m-1)n}{n}$ gives the number of ways to place n modules of length l_m in a string of length l_g. Multiplication of these two expressions gives the number of all possible strings that have n modules, for all n $0 \leq n \leq \lfloor \frac{l}{l_m} \rfloor$, and are d Hamming distance

away from the solution. Note that $\lfloor \frac{l}{l_m} \rfloor$ gives the maximum number of modules that can be in a string to keep the phenotype string length l constant. Dividing the number of strings that have a Hamming distance of d from the solution by the search space size, $L(\mathcal{S})$, gives the ratio of strings that have a Hamming distance of d from the solution in the search space.

Thus, we can write i_d as:

$$i_d = \frac{\binom{l}{d}}{|L(\mathcal{S})|} \sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \binom{l-(l_m-1)n}{n}$$

If we rewrite Δ_{avg} formula using i_d, we obtain:

$$\Delta_{avg} = \frac{\sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \binom{l-(l_m-1)n}{n}}{|L(\mathcal{S})|} \sum_{d=0}^{l} \binom{l}{d} d.$$

QED. □

LEMMA 2. *Encapsulating the complete set of modules of length l_m and including them into the alphabet of primitives without replacement does not change the distance bias in the search space.*

PROOF. Let Δ_{avg_0} and Δ_{avg_1} be the average Hamming distance in search spaces $\mathcal{S}_0$ and $\mathcal{S}_1$, respectively.

$$\Delta_{avg_0} = \sum_{d=0}^{l} \frac{\binom{l}{d}}{2^l} d$$

For $\mathcal{S}_1$,

$$\Delta_{avg_1} = \frac{\sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \binom{l-(l_m-1)n}{n}}{|L(\mathcal{S}_1)|} \sum_{d=0}^{l} \binom{l}{d} d.$$

and

$$|L(\mathcal{S}_1)| = \sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} 2^{l-l_m n} \sigma_m^n \binom{l-(l_m-1)n}{n}.$$

where σ_m is the number of modules in the alphabet and $\sigma_m = 2^{l_m}$. We can rewrite Δ_{avg_1} by using $|L(\mathcal{S}_1)|$:

$$\Delta_{avg_1} = \frac{\sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \binom{l-(l_m-1)n}{n}}{\sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} 2^{l-l_m n} \sigma_m^n \binom{l-(l_m-1)n}{n}} \sum_{d=0}^{l} \binom{l}{d} d.$$

We can rearrange the expression in the following way:

$$\Delta_{avg_1} = \frac{\sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \binom{l-(l_m-1)n}{n}}{2^l \sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \binom{l-(l_m-1)n}{n}} \sum_{d=0}^{l} \binom{l}{d} d.$$

This expression reduces to:

$$\Delta_{avg_1} = \sum_{d=0}^{l} \frac{\binom{l}{d}}{2^l} d$$

which is equal to Δ_{avg_0}.

$$\Delta_{avg_1} = \Delta_{avg_0}$$

In summary, there is no distance bias change when encapsulating a complete set of modules of size l_m and adding them to the alphabet without replacing the primitives. QED. □

3.2 Encapsulation of a Module Fully Included in the Solution

We expect that encapsulating a good module that is fully included in the solution will reduce the distance to the solution by promoting the strings that are closer to solution in Hamming distance. This promotion should contribute to a shorter average Hamming distance in the search space. Such case is illustrated in the following example.

Let $\mathcal{S}_0 = \{000, 001, 010, 011, 100, 101, 110, 111\}$ be the genotype space before encapsulation where $\Sigma_0 = \{0, 1\}$ and $l = 3$ and

$$\mathcal{S}_1 = \{000, 001, 010, 011, 0\mathcal{M}, 100, 101, 110, 111, 1\mathcal{M}, \mathcal{M}0, \mathcal{M}1\}$$

be the genotype space after the encapsulation of $M = 11$. The alphabet now includes M as well as 0 and 1 and the phenotype length remains fixed at l. Assume that the solution string is 111. i_d gives the ratio of the elements that are d distance away from the solution. For instance, $i_2 = \frac{3}{8}$ means that $\frac{3}{8}$ of the strings in the search space are a Hamming distance of two from the solution. Similarly, $i_0 = \frac{1}{8}$, $i_1 = \frac{3}{8}$, $i_2 = \frac{3}{8}$, $i_3 = \frac{1}{8}$. In $\mathcal{S}_1$, $i_0 = \frac{3}{12}$, $i_1 = \frac{5}{12}$, $i_2 = \frac{3}{12}$, $i_3 = \frac{1}{12}$. In this example, we can observe that the distance to solution decreases with the encapsulation of module $\mathcal{M}$. The ratio of strings that are at Hamming distance of zero from the solution in $\mathcal{S}_1$ is twice as large as the one in $\mathcal{S}_0$. Similarly, the ratio of the number of strings to search space size after module encapsulation for one step away is larger than the one in the original search space, $\mathcal{S}_0$. Thus, the ratio of the strings that are closer to the solution string increases after the encapsulation of a good module. The ratio of the strings with larger distance to solution, on the other hand, becomes smaller as a result of encapsulation.

Let us define the probability density function i_d. i_d applies to problems whose solutions have the property of location independent modularity. In other words, an encapsulated "good" module is fully valid at any location in the solution. The OneMax problem, on which we focus this paper, has this property. We define the probability density function for the problem class with location independent modular problems as follows:

$$i_d = \frac{1}{|L(\mathcal{S})|} \sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \binom{l-l_m n}{d} \binom{l-(l_m-1)n}{n} \qquad (3)$$

where $\binom{l-(l_m-1)n}{n}$ gives all possible ways to place n modules in a string and $\binom{l-l_m n}{d}$ gives the number of ways to choose d bits that differ from the solution string of length l. The n modules of length l_m occupies $l_m n$ number of bits and this part of the string does not contain the any of the d unmatching bits. Therefore, these d bits are located in the remaining $l - l_m n$ length of the string. Summing the multiplication of these two terms over all n gives all possible strings with Hamming distance d to the solution string. If we divide this value by the search space size, we obtain the probability density function of d which we denote with i_d in this paper.

Using Equation 2, we can derive the average Hamming distance for good module encapsulation:

$$\Delta_{avg} = \sum_{d=0}^{l} i_d d$$

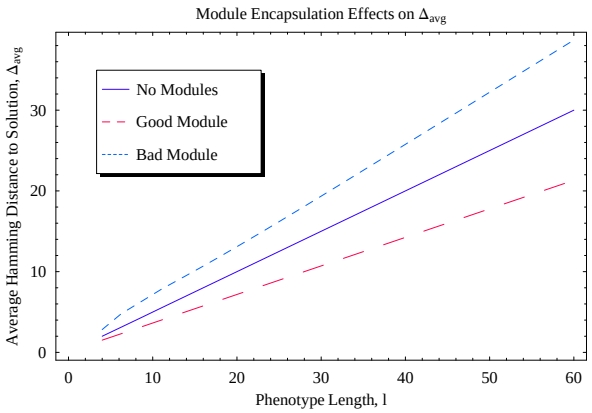

Figure 1: Comparison of the average Hamming distance to the solution in the three structurally different search spaces: the alphabets which consist of primitives and a good module of length two, no modules, and a bad module of length two. The x-axis shows the phenotype length and the y-axis shows the average Hamming distance to the solution. The distance is smaller when a good module is encapsulated and larger when a bad module is encapsulated. The larger the phenotype length, more prominent the effect of module encapsulation on the average Hamming distance to solution measure.

Rewriting Δ_{avg} using the formula of i_d given above:

$$\Delta_{avg} = \sum_{d=0}^{l} d \frac{1}{|L(\mathcal{S})|} \sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \binom{l - l_m n}{d} \binom{l - (l_m - 1)n}{n}$$

gives us a general formula for calculating average Hamming distance in a search space that includes all primitives and a module that is fully included in the solution string. If we replace the values of l_m in the formula for the case of encapsulating module M size of 2, we obtain:

$$\Delta_{avg} = \sum_{d=0}^{l} d \frac{1}{|L(\mathcal{S})|} \sum_{n=0}^{\lfloor \frac{l}{2} \rfloor} \binom{l - 2n}{d} \binom{l - n}{n}$$

To determine the effect of good module encapsulation on average Hamming distance, we compare the average Hamming distance before and after encapsulation. Assume Δ_{avg_0} and Δ_{avg_1} are the average

Hamming distance before and after the encapsulation, respectfully. To compare, we evaluate Δ_{avg_0} and Δ_{avg_1}, for values of $l \geq l_m$. We exclude values for $l < l_m$, since l can not be negative and for nonnegative values of l that are smaller than the module length, Δ_{avg_0} and Δ_{avg_1} are equal because there are no modules. Figure 1 shows the comparison between Δ_{avg_0} and Δ_{avg_1}. Δ_{avg_0} is larger than Δ_{avg_1} for all l values. This result support our initial argument that the average Hamming distance after encapsulation of a good module is smaller than the one before the encapsulation.

3.3 Encapsulation of a Module Fully Excluded From the Solution

We expect that encapsulating a bad module will have a counter effect, increasing average Hamming distance to the solution. Bad modules increase the average Hamming distance because they result in a larger search space with new strings that increase the average Hamming distance to the solution because they are at least bad module length distance or further away from the solution. For instance, assume that $\mathcal{S}_0$ and $\mathcal{S}_1$ are the search spaces before and after the encapsulation, respectively.

$$\mathcal{S}_0 = \{000, 001, 010, 011, 100, 101, 110, 111\}$$

where $\Sigma_0 = \{0, 1\}$ and

$$\mathcal{S}_0 = \{000, 001, 010, 011, 0\mathcal{M}, 100, 101, 110, 111, 1\mathcal{M}, \mathcal{M}0, \mathcal{M}1\}$$

where $\Sigma_0 = \{0, 1, \mathcal{M}\}$. We assume that the solution string is 111 and the module $M = 00$. Note that module M is not a part of the solution string. Let us look at the probability density of each distance value d in each search space. In $\mathcal{S}_0$, $i_{0_0} = \frac{1}{8}$, $i_{1_0} = \frac{3}{8}$, $i_{2_0} = \frac{3}{8}$, $i_{3_0} = \frac{1}{8}$ In $\mathcal{S}_1$, $i_{0_1} = \frac{1}{12}$, $i_{1_1} = \frac{3}{12}$, $i_{2_1} = \frac{5}{12}$, $i_{3_1} = \frac{3}{12}$ The probability density of the shorter distances decreases as we observe in $i_{0_1} < i_{0_0}$ and $i_{1_1} < i_{1_0}$. On the other hand, the probability density of longer distances increases as seen in $i_{2_1} > i_{2_0}$ and $i_{3_1} > i_{3_0}$. We can calculate the average Hamming distance of each case for this example.

$$\Delta_{avg_0} = \sum_{d=0}^{l} i_{d_0} d = 1.5$$

and

$$\Delta_{avg_1} = \sum_{d=0}^{l} i_{d_1} d = 1.83$$

Hence, $\Delta_{avg_1} > \Delta_{avg_0}$. In other words, the average Hamming distance of the search space with a bad module is larger than that of the search space without a bad module. Let us define the probability density function for the search space that has a bad module in its alphabet.

$$i_d = \frac{1}{|L(\mathcal{S})|} \left(\binom{l}{d} + \sum_{n=1}^{\lfloor \frac{l}{l_m} \rfloor} f(n) \binom{l - (l_m - 1)n}{n} \right) \quad (4)$$

where $f(n)$ is

$$f(n) = \begin{cases} \binom{l - l_m n}{d - l_m n} & \text{if } d > l_m \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

where $\binom{l-(l_m-1)n}{n}$ gives all possible ways to place n modules in a string of length l. If $d < l_m$, which is true only when there are no modules in the string (since a module introduces l_m number of unmatching bits into the string), $\binom{l}{d}$ gives all the possible strings that do not include any module, and d is the distance away from the solution. The rest of the expression is zero since it enumerates the strings that have at least one module. If $d \geq l_m$, we can include strings that have modules as well as the ones that do not have any modules. The term $\binom{l}{d}$ enumerates the strings that have no modules and are d distance away from the solution. The second term, $\sum_{n=1}^{\lfloor \frac{l}{l_m} \rfloor} f(n) \binom{l-(l_m-1)n}{n}$ gives all possible strings that

include at least one module and are d distance away from the solution. $l_m * n$ of d unmatching bits come from the n modules. The rest of the d unmatching bits, $d-l_m n$, are enumerated by $\binom{l-l_m n}{d-l_m n}$. Multiplying $\binom{l-l_m n}{d-l_m n}$ with $\binom{l-(l_m-1)n}{n}$ gives all possible strings that are d bits away from the solution and have n modules. Summing up the number of strings for all $n \geq 1$ gives $\sum_{n=1}^{\lfloor \frac{l}{l_m} \rfloor} f(n)\binom{l-(l_m-1)n}{n})$ gives all possible strings that are d distance away from the solution and include at least one module.

Using Equation 2 and Equation 4, we can derive the expression to calculate the average Hamming distance to the solution:

$$\Delta_{avg} = \sum_{d=0}^{l}(\frac{d}{|L(\mathcal{S})|}(\binom{l}{d} + \sum_{n=1}^{\lfloor \frac{l}{l_m} \rfloor} f(n)\binom{l-(l_m-1)n}{n}))$$

where $f(n)$ is given in Equation 5.

How does the average Hamming distance change when a bad module is encapsulated? In order to answer this question, we compare Δ_{avg} before and after the encapsulation. We use the OneMax problem for this mathematical analysis. Assume that $\mathcal{S}_0$ is the search space before the encapsulation of M and $\Sigma_0 = \{0, 1\}$ is the alphabet of the search space $\mathcal{S}_0$. Assume also that $\mathcal{S}_1$ is the search space after the encapsulation of M and $\Sigma_1 = \{0, 1, M\}$ is the alphabet of the search space $\mathcal{S}_1$. Module M is a substring of length l_m consisting of all 0's. It is not a part of the solution string of the OneMax problem which consists of all 1s. Let us write Δ_{avg_0}.

$$\Delta_{avg_0} = \sum_{d=0}^{l} \frac{d}{|L(\mathcal{S}_\prime)|}\binom{l}{d}$$

where $|L(\mathcal{S}_\prime)| = 2^l$. If we place this into the formula above:

$$\Delta_{avg_0} = \sum_{d=0}^{l} \frac{d}{2^l}\binom{l}{d} = \frac{l}{2}$$

Let us write Δ_{avg_1}

$$\Delta_{avg_1} = \sum_{d=0}^{l} d(\frac{1}{|L(\mathcal{S}_1)|}(\binom{l}{d} + \sum_{n=1}^{\lfloor \frac{l}{l_m} \rfloor} \binom{l-l_m n}{d-l_m n}\binom{l-(l_m-1)n}{n}))$$

where $|L(\mathcal{S}_1)| = \sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} 2^{l-l_m n}\binom{l-(l_m-1)n}{n}$. Replacing $|L(\mathcal{S}_1)|$ by its equivalence, we can rewrite Δ_{avg_1} as:

$$\Delta_{avg_1} = \sum_{d=0}^{l} d\frac{1}{\sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} 2^{l-l_m n}\binom{l-(l_m-1)n}{n}} (\binom{l}{d} + \sum_{n=1}^{\lfloor \frac{l}{l_m} \rfloor} \binom{l-l_m n}{d-l_m n}\binom{l-(l_m-1)n}{n})$$

If, for example, the module encapsulated is size of $l_m = 2$,

$$\Delta_{avg_1} = \sum_{d=0}^{l}(\frac{1}{|L(\mathcal{S}_1)|}(\binom{l}{d} + \sum_{n=1}^{\lfloor \frac{l}{2} \rfloor} \binom{l-2n}{d-2n}\binom{l-n}{n}))d$$

where $|L(\mathcal{S}_1)| = 2^l \sum_{n=0}^{\lfloor \frac{l}{2} \rfloor}(\frac{1}{4})^n\binom{l-n}{n}$. Using Equation [9], $|L(\mathcal{S}_1)|$ can be rewritten as follows:

$$|L(\mathcal{S}_1)| = \frac{2^l}{\sqrt{2}}((\frac{1+\sqrt{2}}{2})^{l+1} - (\frac{1-\sqrt{2}}{2})^{l+1}).$$

If we rewrite Δ_{avg_1} using the equation above, we obtain:

$$\Delta_{avg_1} = (\frac{\sum_{d=0}^{l}(\binom{l}{d} + \sum_{n=1}^{\lfloor \frac{l}{2} \rfloor}\binom{l-2n}{d-2n}\binom{l-n}{n})d}{\frac{2^l}{\sqrt{2}}((\frac{1+\sqrt{2}}{2})^{l+1} - (\frac{1-\sqrt{2}}{2})^{l+1})}$$

Figure 1 gives the comparison of Δ_{avg_0} and Δ_{avg_1}. Δ_{avg_1} is larger than Δ_{avg_0} for all $l \geq 0$. In other words, encapsulating a bad module increases the average distance to the solution in the phenotype space for the OneMax problem. Therefore, we can conclude that, for the OneMax problem, encapsulating a bad module increases the average Hamming distance to solution in the phenotype space.

4. EXPERIMENTAL ANALYSIS

In Sections 2 and 3, we present a simple analysis of module encapsulation on the bias of the search space. Next, we compare values calculated from those equations with an empirical validation using the OneMax problem.

4.1 Method

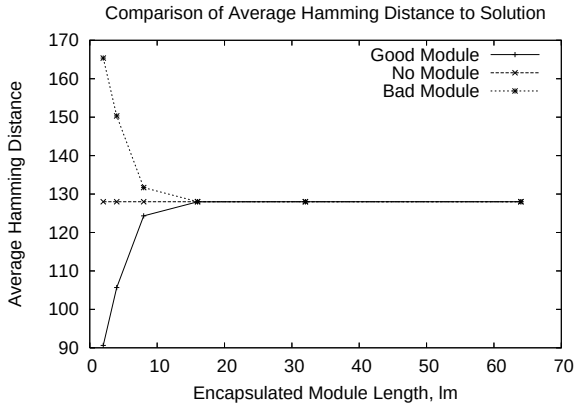

Figure 2: Comparision of the theoretical results of the average Hamming distance to a solution for search spaces with a good module, no modules, and a bad module. The x-axis shows the encapsulated module length $l_m = \{2, 4, 8, 16, 32, 64\}$ and the y-axis shows the average Hamming distance.

We first use our equations from Section 3 to calculate the average distance to solution for a 256-bit OneMax problem with no modules, one good module, and one bad module. A good module is a subsequence of all ones; a bad module is a subsequence of all zeros. Figure 2 shows the theoretical results for the average distance to bias metrics for the module size, $l_m \in \{2, 4, 6, 8, 16, 32, 64\}$. As expected, the theoretical data show that encapsulating a good module decreases the average distance to solution while encapsulating a bad module increases the average distance to solution.

In comparison, our empirical validation runs evaluate the fitness of a GA also on a 256-bit OneMax problem. We use the following parameter values for our GA runs: the mutation rate is 0.01, the selection type is tournament with size 4, the population size is 500, and the number of generations is 500. We perform 100 trials for all experiments and report average values with their 95% confidence interval. Our

fitness function is the ratio of the number of 1s over the individual length. We run GA using the same module sizes as the theoretical data above and observe if the varations in the experiment results correlate with the changes we obtained in our theoretical results.

As our theoretical analysis focuses only on the search space and does not take into account the dynamics and the characteristics of any specific search algorithm, we expect only qualitative verification of our results.

4.2 Results

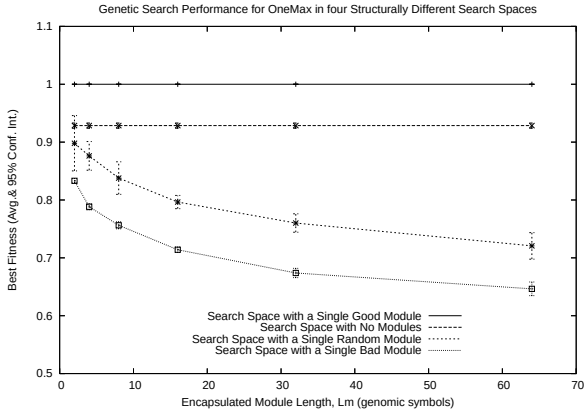

Figure 3: Perfomance comparison of a GA in four structurally different search spaces: a search space with a good module, a search space with no modules, a search space with a bad module, and a search space with a random module. Performance is measured in terms of best fitness and shown on the y-axis for each encapsulated module length $l_m = \{2, 4, 8, 16, 32, 64\}$ shown on the x-axis.

Figure 3 shows the results of our empirical runs. We compare the performance of a GA with no modules, with one good module, with bad module, for module length, $l_m \in \{2, 4, 6, 8, 16, 32, 64\}$. In addition, we also include GA runs using one randomly generated module. The y-axis shows the best fitness and the x-axis shows the module length, l_m. The GA with a good module outperforms the GA with no modules and the GA with no modules outperforms the GA with a bad module for all l_m values. It is evident that the performance of GA with no modules is not affected by the module size changes. The performance of the GA with a good module also remains unaffected by the module length. While increasing the good module length may improve the ability to find a solution, it may also make it more difficult to evolve a solution of the correct length. As expected, the performance of the GA with bad module degrades as module length increases. The GA with a random module case is out of the scope of this paper. It is, however, given to compare its performance with the other three cases. These results are comparable to the performance predicted by our theoretical results.

Also, as module length increases linearly, the relative decrease in performance from slows down. We observe a similar behavior in our theoretical results where the larger l_m has smaller effect on the average distance to solution.

5. CONCLUSION

In this paper, we investigate the effects of module encapsulation on the average Hamming distance to a solution. Encapsulating a module increases the size of the alphabet and consequently the size of the search space. We define a theoretical framework on which to describe search spaces for binary representations and analyze the impact of adding a complete set of modules, a good module, and a bad module to a binary search space. We then compare our theoretical results to empirical runs of a GA on the OneMax problem.

Our theoretical results indicate that encapsulating a complete module set does not affect the average Hamming distance in the search space. In other words, encapsulating without favoring a good or a bad module does not change the average hamming distance to solution. Encapsulating a good module, however, decreases the average Hamming distance solution while encapsulating a bad module increases it. These results correlate with our experimental results which show that encapsulating a good module improves the performance of GA while encapsulating a bad module decreases performance. These results also underscore the importance of selecting modules intelligently when using modular representations for search.

6. REFERENCES

[1] L. Altenberg. Fitness distance correlation analysis: An instructive counterexample. In T. Baeck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97)*, pages 57–64, San Francisco, CA,, 1997. Morgan Kaufmann. http://dynamics.org/Altenberg/PAPERS/FDCAAIC/.

[2] P. J. Angeline and J. B. Pollack. The evolutionary induction of subroutines. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, Bloomington, Indiana, USA, 1992. Lawrence Erlbaum.

[3] P. J. Angeline and J. B. Pollack. Evolutionary module acquisition. In D. Fogel and W. Atmar, editors, *Proceedings of the Second Annual Conference on Evolutionary Programming*, pages 154–163, La Jolla, CA, USA, February 25-26 1993.

[4] E. D. De Jong and T. Oates. A coevolutionary approach to representation development. In *Proc. of the ICML-2002 WS on development of rep.*, page 1, 2002.

[5] E. D. De Jong, D. Thierens, and R. A. Watson. Defining modularity, hierarchy, and repetition. In *GECCO 2004 Workshop Proceedings*, 2004.

[6] I. I. Garibay, O. O. Garibay, and A. S. Wu. Effects of module encapsulation in repetitively modular genotypes on the search space. In *GECCO '04: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1125–1137, 2004.

[7] O. O. Garibay, I. I. Garibay, and A. S. Wu. The modular genetic algorithm: Exploiting regularities in the problem space. In *Proceedings of ISCIS 2003 The International Symposium on Computer and Information Systems*, LNCS, pages 584–591. Springer-Verlag, 2003.

[8] O. O. Garibay, I. I. Garibay, and A. S. Wu. No free luch theorem for modular genomes. Technical report, University of Central Florida, 2004.

[9] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Boston, MA, 1994.

[10] D. G. Green, D. Newth, D. Cornforth, and M. Kirley. On evolutionary processes in natural and artificial systems. In P. Whigham et al., editors, *Proceedings of the 5th Australia-Japan Joint Workshop on Intelligent and Evolutionary Systems*, pages 1–10, 2001.

[11] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[12] G. S. Hornby. Measuring, enabling and comparing modularity, regularity and hierarchy in evolutionary design. In H.-G. Beyer and U.-M. O'Reilly, editors, *GECCO '05: Proceedings of the 2005 genetic and evolutionary computation conference*, pages 1729–1736, New York, NY, USA, 2005. ACM Press.

[13] G. S. Hornby and J. B. Pollack. Creating high-level components with a generative representation for body-brain evolution. In *Artificial Life 8*, pages 223–246. MIT, 2002.

[14] T. Jones and S. Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 184–192, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[15] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA, 1994.

[16] H. Lipson. Principles of modularity, regularity, and hierarchy for scalable systems. In *GECCO 2004 Workshop Proceedings*, 2004.

[17] H. Lipson, J. B. Pollack, and N. P. Suh. Promoting modularity in evolutionary design. In *Proceedings of DETC'01: 2001 ASME Design and Engineering Technical Conferences*, Pittsburg, Pennsylvania, USA, September 9-12 2001.

[18] H. Meinhardt. Pathways and building blocks. *Nature*, 430:970–970, 2004.

[19] M. A. Potter and K. A. De Jong. A cooperative coevolutionary approach to function optimization. In H.-P. S. Y. Davidor and R. Mnner, editors, *PPSN III: Proceedings of the 3th International Conference on Parallel Problem Solving from Nature*, pages 249–257, Berlin, 1994. Springer.

[20] M. A. Potter and K. A. De Jong. Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation*, 8(1):1–29, 2000.

[21] J. P. Rosca and D. H. Ballard. Learning by adapting representations in genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[22] G. Schlosser and G. P. Wagner, editors. *Modularity in Development and Evolution*. The University of Chicago Press, 2004. QH 491.M59 2004.

[23] A. S. Wu and R. K. Lindsay. Empirical studies of the genetic algorithm with non-coding segments. *Evolutionary Computation*, 3(2):121–147, 1995.