Effects of Module Encapsulation in Repetitively Modular Genotypes on the Search Space

Ivan I. Garibay[1,2], Ozlem O. Garibay[1,2], and Annie S. Wu[1]

[1] University of Central Florida, School of Computer Science,
P.O. Box 162362, Orlando, FL 32816-2362, USA,
{igaribay,ozlem,aswu}@cs.ucf.edu,
WWW home page: http://ivan.research.ucf.edu
[2] University of Central Florida, Office of Research
Orlando Tech Center/ Research Park
12443 Research Parkway Orlando, FL 32826, USA.

Abstract. We introduce the concept of *modularity-preserving representations*. If a representation is modularity-preserving, the existence of modularity in the problem space is translated into a corresponding modularity in the search space. This kind of representation allows us to analyze the impact of modularity at the genomic level. We investigate the question of what constitutes a module at the genomic level of evolutionary search and provide a static analysis of how to identify good and bad modules based on their ability to reduce the search space, thus, biasing the search space towards a solution. We also prove, under a set of assumptions, that the systematic encapsulation of lower order modules into higher order modules does not change the size or bias of a search space and that this process produces a hierarchy of equivalent search spaces.

1 Introduction

The success of evolutionary algorithms for a given problem is heavily affected by the representation used [1–3]. The task of designing a good representation is, at this point, more an art than a science. In this paper, we analyze the effects that the choice of representation primitives and the encapsulation of primitives into modules can have on the size of a search space and bias towards a solution. We focus on two questions. Does the encapsulation and replacement of lower level modules or primitives with higher level counterparts, by itself, benefit evolutionary search? Under what circumstances does the encapsulation of lower level modules or primitives into higher level modules benefit evolutionary search and how? We offer a static mathematical analysis as well as experimental results to shed light on these two issues. We find that replacing lower level elements with encapsulated higher level counterparts, by itself, has no effect on the size or bias of the search space. This result is, in essence, a kind of No Free Lunch theorem [4] for genomic module encapsulation. We also find that the process of encapsulating primitives into modules has two static effects. It enlarges the

search space, because it introduces a new element into the search space alphabet; and it biases this extended search space towards solutions that contain the encapsulated primitives. As a result, there is a trade-off between the gains and losses of introducing a new module in terms of search space size and bias. We provide a closed form expression, under certain assumptions, whether the creation of a module will be beneficial in terms of the size of the resulting search space size. We show that this bias is governed by the module size and by how many times the module appears in the solution string.

2 Modularity

The concept of modularity has been studied extensively in complex systems and also in evolutionary computation. Recently, issues such improving the "innovativeness" and the scalability of evolutionary search have attracted renewed interest to the study of modularity in the evolutionary computation community. For instance see [5–7]. From our perspective, modularity implies not only the hierarchical organization of components from one level of complexity to the next, but also the ability to freely reuse components. In Evolutionary Computation, various techniques have been used to incorporate modularity into the evolutionary search, for example, *compress* and *expand* operators [8], automatically defined functions [9], speciation [10], repetitive modularity [11, 12], and coevolutionary methods [13], to name a few. All of these techniques seek to identify "good" modules for a given problem. In contrast with module definitions based on fitness [14, 15, 13], we study *location-independent genomic modularity*. For us, a *genomic module* is simply a pattern of consecutive genomic primitives or lower-level genomic modules occurring at any location in a genome.

We base our study of modular genomes on the following assumptions: first, that the class of problems with modular solutions is of interest; and second, that there exist representations that correlate modularity in solutions with corresponding modularity in genomes and that such representations can be found for any repetitively modular problem. We call this kind of representations *modularity-preserving representations(MPR)* and we call the second assumption the *Modularity-preserving representation hypothesis*. For the reminder of this paper, we assume that this hypothesis is true.

3 Mathematical Analysis

We introduce the following definitions using standard set theory and formal languages notation, i.e. see [16].

3.1 Modular Search Spaces

Primitives are the atomic components of problem representation that are used to encode a *candidate solution*. A candidate solution, also called an *individual*,

may consist of both primitives and modules. *Modules* are substrings of interest. A module may contain two kinds of symbols: primitives and previously defined modules. The *search space* of a problem is the space of all possible candidate solutions. The structure of a search space is determined by its alphabet which can consist of both primitive and module symbols, and by the length of its individuals. The elements of the search space are all possible strings of a given length over the search space alphabet.

Definition 1 (Search space). *A* search space $\mathcal{S}$ *is a 3-tuple:*

$$\mathcal{S} = \langle \Sigma, l, \mathcal{R} \rangle$$

Where, $\Sigma \subseteq \mathcal{P} \cup \mathcal{M}$ *is the search space alphabet;* $\mathcal{P}$ *is a set of primitive symbols, and* $\mathcal{M}$ *is a set of module symbols;* l *is the length of the individuals; and* $\mathcal{R}$ *is the set of module defining rules.*

Definition 2 (Module defining rules). *For a given search space* $\mathcal{S} = \langle \Sigma, l, \mathcal{R} \rangle$*, if* $r_i \in \mathcal{R}$ *then* r_i *is of the form:*

$$r_i = M_i \rightarrow w_i$$

Where, $M_i \in \mathcal{M}$ *is the module name;* $w_i \in \{\mathcal{P} \cup \{M_1, M_2, ..., M_{i-1}\}\}^*$ *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 $|\mathcal{R}| = |\mathcal{M}|$. Note that rules are hierarchically defined in terms of primitives and previously defined modules; hence, circularity in definitions is not possible. Let us define a module *size* as 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 $n-1$.

Definition 3 (Search space elements). *The elements of the search space* $\mathcal{S}$*, denoted by* $L(\mathcal{S})$*, are all strings of length* l *over the search space alphabet* Σ*:*

$$L(\mathcal{S}) = \{e \mid e \in \Sigma^* \wedge |e| = l\}$$

Since Σ can potentially contain primitives as well as modules, individuals can be *expanded* using the module definitions in $\mathcal{R}$. An individual is expanded by rewriting module names by module definitions until the individual consist of a string of only primitives.

Definition 4 (expanded form). *Let* $e \in L(\mathcal{S})$ *be an element of search space* $\mathcal{S} = \langle \Sigma, l, \mathcal{R} \rangle$*. We define the expanded form of element* e *as:*

$$e_{exp} = Expand_{\mathcal{R}}(e)$$

Where $Expand_{\mathcal{R}}$ *is the expanding function for module definitions* $\mathcal{R}$*. The expanding function applies the rewriting rules in* $\mathcal{R}$ *to its input* e *until a string solely over* $\mathcal{P}$ *is obtained,* $Expand_{\mathcal{R}}$ *outputs that string. In this case, we said that* e_{exp} *has been* derived *from* e *using rewriting rules* $\mathcal{R}$*.*

Notice that $e \in \Sigma^*$ and $|e| = l$, while $e_{exp} \in \mathcal{P}^*$ and $|e_{exp}| \geq l$. Also, if e does not contain any module symbols, then $e = e_{exp}$.

Definition 5 (expanded search space). *Let $\mathcal{S} = \langle \Sigma, l, \mathcal{R} \rangle$ be a search space and $L(\mathcal{S})$ its elements. We define the expanded search space $L_{exp}(\mathcal{S})$ as follows:*

$$L_{exp}(\mathcal{S}) = \{g \mid g = Expand_{\mathcal{S}}(e) \wedge e \in L(\mathcal{S})\}$$

$L_{exp}(\mathcal{S})$ is the set of all individuals in the search space $\mathcal{S}$ in their expanded form. Elements of $L_{exp}(\mathcal{S})$ are variable length strings over $\mathcal{P}$. It is easy to show that the length of the expanded individuals, l_{exp}, is bounded by: $l \leq l_{exp} \leq l^{|M|+1}$.

3.2 Search Space Size and Bias

The size of a search space $\mathcal{S} = \langle \Sigma, l, \mathcal{R} \rangle$, denoted by $|L(\mathcal{S})|$, is $|L(\mathcal{S})| = |\Sigma|^l$. Notice that it is possible for two different individuals to expand to the same string of primitives, therefore, $|L_{exp}(\mathcal{S})| \leq |L(\mathcal{S})|$. As we are interested in the bias produced by module definitions, we measure the bias of a search space using the expanded form. If the expanded search space $L_{exp}(\mathcal{S})$ contains all possible strings of primitives of a given length t, then we said that the search space has no *structural bias* for length t because there are no unreachable strings of primitives of length t. The search space is structurally biased otherwise. Note that even in an structurally unbiased search space where all strings of primitives are reachable, some strings of primitives may have multiple derivations and therefore be preferred. We call this later case a *modularly biased* search space. [3]

Definition 6 (Search space bias). *For a given search space $\mathcal{S} = \langle \Sigma, l, \mathcal{R} \rangle$, if*

$$L_{exp}(\mathcal{S}) \supseteq \{e \mid e \in \mathcal{P}^* \wedge |e| = t\}$$

is true, we say that $\mathcal{S}$ is not structurally biased for length t, or simply that it is not structurally biased in the case that $t = l$. $\mathcal{S}$ is structurally biased otherwise.

A set of module definition rules $\mathcal{R}$ are *complete-$\mathcal{P}$* with respect to a set of primitives $\mathcal{P}$ it the rules are able to derive every string of a given length t over the alphabet of primitives $\mathcal{P}$, starting from strings containing only module names from $\mathcal{R}$.

Definition 7 (Complete-$\mathcal{P}$). *Lets $\mathcal{R}$ and $\mathcal{P}$ be a set of module definition rules and a set of primitive symbols respectively. We say that $\mathcal{R}$ is complete-$\mathcal{P}$ with respect to $\mathcal{P}$ for a given length t if the following expression holds true:*

$$\{Expand_{\mathcal{R}}(e) \mid e \in \mathcal{M}^*\} \supseteq \{e \mid e \in \mathcal{P}^* \wedge |e| = t\}$$

Where $\mathcal{M}$ is the set of all module names from $\mathcal{R}$

[3] For instance, the search space $\mathcal{S} = \langle \Sigma = \{1, 0, A\}, l = 8, \mathcal{R} = \{A \rightarrow 11\} \rangle$ is structurally unbiased since all strings of primitives of length 8 can be reached; however it is modularly biased since, for instance, the string "00000000" can only be derived from "00000000" and the string "111111111" can be derived from "A1111111", "1A111111", "11A11111", etc.

Definition 8 (Complete m-module set). *$\mathcal{R}_c$ is the complete m-module set of Q iff:*

$$\mathcal{R}_c = \{(M_i \rightarrow w) \mid w \in Q^* \wedge |w| = m \wedge i \text{ is a unique index for } w\}$$

Where, Q is an arbitrary set of symbols, $\mathcal{R}_c$ is a set of module defining rules, m is the size of all modules in $\mathcal{R}_c$, and i is an arbitrary index for strings w over Q.

Since there is one module defined on $\mathcal{R}_c$ per each string of size m over Q, we have $|\mathcal{R}_c| = |Q|^m$

Lemma 1. *Lets $\mathcal{R}$ and $\mathcal{P}$ be a set of module definition rules and a set of primitive symbols respectively. The following two expressions are true:*

$$(\mathcal{R} \supseteq \text{complete m-module set of } \mathcal{P}) \rightarrow (\mathcal{R} \text{ is complete-}\mathcal{P}) \quad (1)$$

$$(\mathcal{R} \supseteq \text{complete m-module set of a complete-}\mathcal{P} \text{ set}) \rightarrow (\mathcal{R} \text{ is complete-}\mathcal{P}) \quad (2)$$

Lemma 2. *Let $\mathcal{S} = \langle \Sigma, l, \mathcal{R} \rangle$ be a search space. If $\mathcal{S}$ is not structurally biased then one of the following is true:*

1. *$\Sigma \supseteq \mathcal{P}$, or*
2. *$\Sigma \supseteq \mathcal{B}$, where $\mathcal{B}$ is the set of module names of a complete-$\mathcal{P}$ set of rules with respect to $\mathcal{P}$.*

It is not difficult to prove Lemmas 1 and 2

3.3 Search Space Altering Operations

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 module symbol to the alphabet.

Definition 9 (Encapsulation). *Module* encapsulation, *$\mathcal{E} : \mathcal{S} \times r_k \mapsto \mathcal{S}$ is defined as follows:*

$$\mathcal{E}(\mathcal{S}, M_k \rightarrow w_k) = \langle \Sigma \cup \{M_k\}, l, \mathcal{R} \cup \{M_k \rightarrow w_k\} \rangle$$

Where, $\mathcal{S} = \langle \Sigma, l, \mathcal{R} \rangle$ is a search 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 Σ, and $|w_k| \leq l$.

Strict-encapsulation of a search space $\mathcal{S}$ is the process of creating all possible modules of a given size m over the current search space alphabet Σ and then replacing the current alphabet with the newly created module names. This process evidently changes the structure of the search space by replacing completely the search space alphabet and the module defining rules. Notice that there are $|\Sigma|^m$ modules of size m that can be created over Σ. The new individual length is l/m, since individuals consist of only new modules of size m.

Definition 10 (Strict-encapsulation). Strict-encapsulation, $\mathcal{E}_s : \mathcal{S} \mapsto \mathcal{S}$ *is defined as follows:*

$$\mathcal{E}_s(\mathcal{S}) = \langle \Sigma', l/m, \mathcal{R}' \rangle$$

Where, $\mathcal{S} = \langle \Sigma, l, \mathcal{R} \rangle$ *is a search space;* $\mathcal{R}'$ *is a complete m-module set for* Σ *; and* Σ' *is the set of newly created module symbols from* $\mathcal{R}'$*.*

Lemma 3. *Let* $\mathcal{S}_1 = \langle \Sigma_1, l_1, \mathcal{R}_1 \rangle$ *and* $\mathcal{S}_2 = \langle \Sigma_2, l_2, \mathcal{R}_2 \rangle$ *be search spaces such that* $\mathcal{S}_2 = \mathcal{E}_s(\mathcal{S}_1)$*, then* $|L_(\mathcal{S}_1)| = |L(\mathcal{S}_2)|$*.*

Proof. We know that $|L_(\mathcal{S}_1)| = {\Sigma_1}^{l_1}$ and $|L_(\mathcal{S}_2)| = |\Sigma_2|^{l_2}$. Since $\mathcal{S}_2 = \mathcal{E}_s(\mathcal{S}_1)$ we have that $|\Sigma_2| = |\Sigma_1|^m$ and $l_2 = l_1/m$. Therefore $|\Sigma_2|^{l_2} = |\Sigma_1|^{m(l_1/m)} = |\Sigma_1|^{l_1}$. □

Lemma 4. *Let* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *be search spaces such that* $\mathcal{S}_2 = \mathcal{E}_s(\mathcal{S}_1)$*, then* $(\textit{not structurally biased}\, \mathcal{S}_1) \rightarrow (\textit{not structurally biased}\, \mathcal{S}_2)$*.*

Proof. Let $\mathcal{S}_1 = \langle \Sigma_1, l_1, \mathcal{R}_1 \rangle$ and $\mathcal{S}_2 = \langle \Sigma_2, l_2, \mathcal{R}_2 \rangle$ be search spaces. We need to prove that:

$$(L_{exp}(\mathcal{S}_1) \supseteq \{e \mid e \in {\mathcal{P}_1}^* \wedge |e| = l_1\}) \rightarrow (L_{exp}(\mathcal{S}_2) \supseteq \{e \mid e \in {\mathcal{P}_2}^* \wedge |e| = l_2\})$$

Let us assume that $\mathcal{S}_1$ is not structurally biased. We will prove that for that case $\mathcal{S}_2$ is also not structurally biased. According to Lemma 2, we would need to consider two cases:
Case 1: $\Sigma_1 \supseteq \mathcal{P}$ hence $\Sigma_1 = \{\mathcal{P} \cup \mathcal{U}\}$ for some possibly empty set $\mathcal{U}$. Since $\mathcal{S}_2 = \mathcal{E}_s(\mathcal{S}_1)$, $\mathcal{R}_2$ is a complete m-module set of $\Sigma_1 = \{\mathcal{P} \cup \mathcal{U}\}$. It is easy to show that for this case:

$$\mathcal{R}_2 \supseteq \text{complete m-module set of } \mathcal{P}_2$$

by Lemma 1 then, $\mathcal{R}$ is complete-$\mathcal{P}$. Therefore, $\mathcal{S}_2$ is also not structurally biased.
Case 2: $\Sigma_1 \supseteq \mathcal{B}$, where $\mathcal{B}$ is complete-$\mathcal{P}$. The prove is analogous to Case 1, but we need to use the second part of Lemma 1. □

Lemma 5. *Lets* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *be search spaces such that* $\mathcal{S}_2 = \mathcal{E}_s(\mathcal{S}_1)$*, then* $(\textit{structurally biased}\, \mathcal{S}_1) \rightarrow (\textit{structurally biased}\, \mathcal{S}_2)$*.*

The proof for Lemma 5 is analogous to that of Lemma 4 and omitted for space constrains. Further details can be found at [17].

3.4 Hierarchy of Modular Search Spaces

Theorem 1. *Strictly-encapsulating lower-order modules into a complete set of higher-order modules does not change the search space size or structural bias.*

Proof. Lets $\mathcal{S}_1$ and $\mathcal{S}_2$ be two search spaces such that $\mathcal{S}_2 = \mathcal{E}_s(\mathcal{S}_1)$. We need to prove that

$$|L_(\mathcal{S}_1)| = |L(\mathcal{S}_2)|$$
$$structurally\, bias\, \mathcal{S}_1 \rightarrow structurally\, bias\, \mathcal{S}_2$$
$$not\, structurally\, bias\, \mathcal{S}_1 \rightarrow not\, structurally\, bias\, \mathcal{S}_2$$

All three statements have been proved in Lemmas 3, 4, and 5. □

Continuously strictly-encapsulating a search space produces a hierarchy of search spaces. At the bottom of this hierarchy we have a search space with individuals of size l and trivial modules of size one (l primitives). As we move up the hierarchy, we have search spaces with more modules and larger module sizes. At the top of this hierarchy, we have individuals of size one consisting of only one module of size l. We call this type of hierarchy a *modularity representation pyramid* for $\mathcal{S}$.

Definition 11 (Modularity representation pyramid). *Let $\mathcal{S}$ be a search space with only primitives. Let us recursively define the modularity representation pyramid, $A = \{a_0, a_1, ...\}$, for $\mathcal{S}$ as:*

$$a_0 = S$$
$$a_{i+1} = \mathcal{E}_s(a_i)$$

The recursion terminates when the individual length for a given a_i is equal to one (the individuals can not be further encapsulated into modules).

Corollary 1. *Let $\mathcal{S}$ be a search space with only primitives and A be a modularity representation pyramid for $\mathcal{S}$. Then the following is true for all levels of the pyramid:*

1. *all search spaces are of equal size: Σ^l*
2. *all search spaces are structurally unbiased.*

Proof. Base case: $\mathcal{S}$ is trivially structurally unbiased and of size Σ^l. Recursive step: by the previous theorem, a_{i+1} and a_i are of the same size and if a_i is structurally unbiased so does a_{i+1}. □

4 What Makes Module Encapsulation Advantageous?

Replacing lower level modules with higher level modules does not produce any advantage unless there is some rationale for pruning some of the "undesired" higher level modules, hence reducing the search space. We define "good" modules to be modules that are present in a solution and "bad" modules to be modules that are not present in a solution. In this section, we analyze the effect on search space size of encapsulating good modules versus bad modules. Let us assume that the optimal solution is known. Consequently, the type and number of modules in the optimal solution is also known. Furthermore, we assume that the length of

the individuals in each search space equals the shortest optimal individual length. Based on these assumptions, we can derive the relationship that must be satisfied in order for encapsulation to be beneficial to a search. Let $\mathcal{S}_1 = \langle \Sigma_1, l_1, \mathcal{R}_1 \rangle$ and $\mathcal{S}_2 = \langle \Sigma_2, l_2, \mathcal{R}_2 \rangle$ be search spaces such that $\mathcal{S}_2$ is the product of encapsulating the module M into search space $\mathcal{S}_1$; hence, $\mathcal{S}_2 = \mathcal{E}(\mathcal{S}_1, (M \rightarrow w))$. Let the optimal solution for search space $\mathcal{S}_1$ be w_s. Let w_s contain x copies of the string w which defines module M. According to Definition 9, we have $\Sigma_2 = \Sigma_1 \cup \{M\}$, therefore:

$$|\Sigma_2| = |\Sigma_1| + 1 \tag{3}$$

Because the optimal solution can be expressed in terms of module $(M \rightarrow w)$ and because the optimal solution contains x copies of the module defining string w, then:

$$l_2 = l_1 - x(|w| - 1) \tag{4}$$

In order for the encapsulation of module $(M \rightarrow w)$ to produce an advantage in terms of search space size, we need:

$$\text{(Search space size with M)} \leq \text{(Search space size without M)}$$

Which it is:

$$|L(\mathcal{S}_2)| \leq |L(\mathcal{S}_1)|$$
$$|\Sigma_2|^{l_2} \leq |\Sigma_1|^{l_1}$$

Using (3) and (4) on the previous equation we have:

$$(|\Sigma_1| + 1)^{l_1 - C} \leq |\Sigma_1|^{l_1} \tag{5}$$

where,

$$C = x(|w| - 1)$$

is a constant which depends only on the modular properties of the solution string w_s. Applying logarithms to both sides of (5) and rearranging the terms gives us:

$$C \geq l_1 \frac{\ln(1 + 1/|\Sigma_1|)}{\ln(|\Sigma_1| + 1)} \tag{6}$$

Therefore, there are three possible outcomes when encapsulating a good module. If (6) is satisfied as an inequality, we predict an advantageous reduction of search space size when the module $(M \rightarrow w)$ is encapsulated. If it is satisfied as an equality, we predict no advantage or disadvantage because the search spaces will be of equal size. If (6) is not satisfied, then the encapsulation of module $(M \rightarrow w)$ will be disadvantageous since it will result in a larger search space. On the other hand, encapsulating a bad module is always disadvantageous. Since a bad module is not present in the optimal solution the value of the constant is $C = 0$, which renders (6) unsatisfiable.

	Alphabet size	Module length	Solution length
Level 1	2	1	8
Level 2	4	2	4
Level 3	16	4	2
Level 4	256	8	1

Table 1. Details for the modularity representation pyramid A_1 used in experiment 1.

5 Experimental Analysis

5.1 Objectives

In Sects. 3 and 4, we present a simple analysis of the effects of module encapsulation on the size and bias of a search space. We next present two experiments to empirically verify our conclusions. In both experiments, we evaluate the performance of a GA on a modular space in terms of the best fitness in the final generation. Performance evaluation is averaged over 40 runs. As our analyses focus only on the search space and do not take into account the characteristics or dynamics of any search algorithm—such as the GA that we use—, we expect only qualitative verification of our conclusions from the empirical tests.

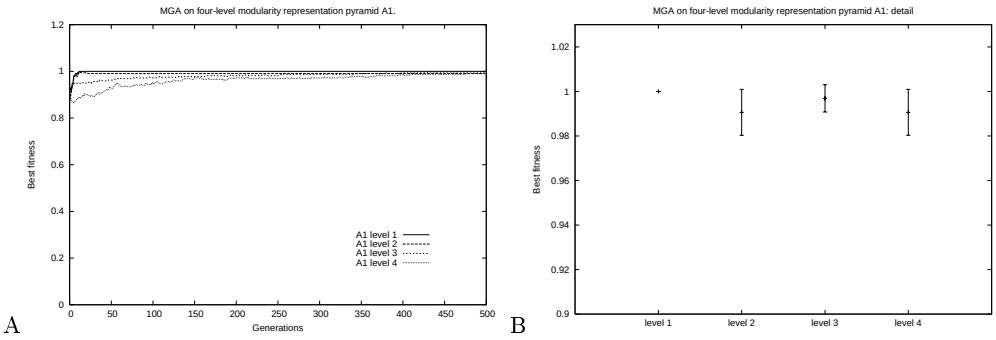
The first experiment is an empirical validation of Theorem 1. We select a small modularity representation pyramid A_1 for $\mathcal{S}_1 = \langle \Sigma = \{0,1\}, l = 8, \mathcal{R} = \{\}\rangle$. Each level of the pyramid is described in Table 1. Although GA dynamics at various levels of the search space may affect results to some degree. We expect to see a qualitative validation of Theorem 1 in the form of roughly comparable performance at all pyramid levels. In the second experiment, we attempt to validate (6) by comparing theoretically and experimentally obtained values of C. We follow the analysis in Sect. 4 using the following values: search space $\mathcal{S}_1 = \langle \Sigma_1 = \{0,1\}, l_1 = 64, \mathcal{R}_1 = \{\}\rangle$, module to be encapsulated $M \rightarrow 00010111$, optimal solution w_s contain x, for $x \in \{0,2,3,4,5,6\}$, copies of the string 00010111. We use (4) to calculate the value of the constant $C = x(|00010111| - 1) = 7x$. Equation 6 gives us the condition that must be satisfied for encapsulation to be beneficial: $C \geq 64\frac{ln(1+1/2)}{ln(2+1)} = 23.62$. From these equations, we can calculate the theoretical threshold value $x \geq 3.37$. Therefore, according to the analysis in Sect. 4, a target solution must contain more than 3 copies of 00010111 for the encapsulation of module $M \rightarrow 00010111$ to be beneficial. If the target solution contains more than three copies of the module M, then we expect the search space size to decrease and performance to improve. If the target solution contains fewer than three copies of M then we expect the search space size to increase and the performance to degrade.

5.2 Settings

For all experiments we use a modular version of a genetic algorithm (GA) [18, 19] called the *modular genetic algorithm* (MGA) [11]. MGA is a simple GA with the

ability to encapsulate modules as described in Sect. 3. MGA genetic operators are analogous to GA operators, but work with strings over alphabets of any size. The following parameter settings are common for all experiments: the crossover type is two-point, the crossover rate is 0.9, the mutation rate is 0.005, selection type is fitness proportional, population sizes of 50 for the first and 500 for the second set of experiments, and the number of generations is 500. We perform 40 trials for all experiments and report average values with their 95% confidence intervals.

5.3 Results



A B

Fig. 1. MGA on random pattern matching for each level of the modularity representation pyramid A_1: (A) best fitness by generation averaged over 40 runs, (B) fitness of best solution found averaged over 40 runs and 95% confidence intervals. This empirical result validates Theorem 1 since the performance is comparable for all levels.

Figure 1 shows the results from the first experiment. Figure 1(A) shows the best fitness for each generation averaged over 40 runs. Figure 1(B) shows the fitness of the best solution found averaged over 40 runs and 95% confidence intervals. For all four levels of the hierarchy of modular search spaces defined by the pyramid A_1 we obtain comparable performance as predicted by Theorem 1. Figure 2 shows the results from the second experiment. We compare the MGA with a traditional GA on a pattern matching problem. The goal is to generate a target solution w_s containing x copies of module M. Figure 2 shows the fitness of the best solution found averaged over 40 runs and 95% confidence intervals. These results indicate that GA performance is insensitive to the number of modules M contained in a solution. MGA performance with module M encapsulated is linearly correlated with x. We observe that the experimental threshold for x is approximately $x \geq 6$. This experiment validates qualitatively our analysis of what makes the encapsulation of a module advantageous. For the case $x = 0$, no good module is defined, encapsulation is disadvantageous, and a GA outperforms the MGA. For the cases of $x = 2, 3, 4, 5$ there is a good module defined

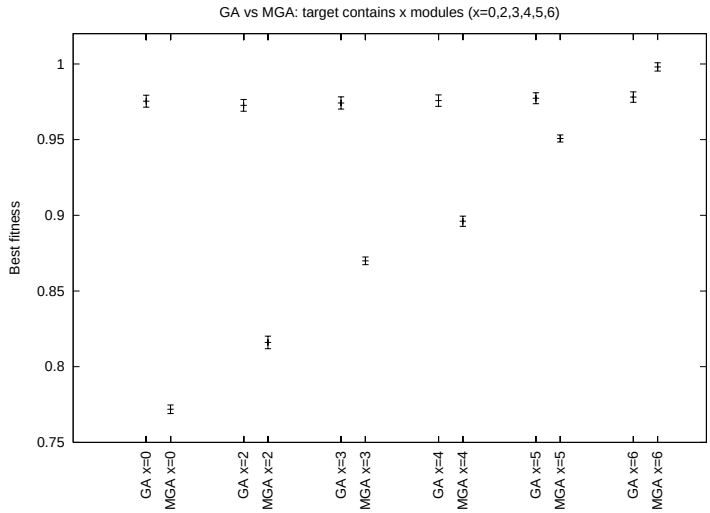


Fig. 2. MGA -vs- GA on pattern matching for target solution w_s containing $x \in \{0, 2, 3, 4, 5, 6\}$ copies of module M: Fitness of best solution found averaged over 40 runs and 95% confidence intervals.

but the experimental threshold has not been reached; therefore, encapsulation remains disadvantageous. Finally, for $x = 6$, there is a good module defined, the threshold has been met, and encapsulation in this case is advantageous—the MGA outperforms a traditional GA.

6 Conclusions

In this paper, we investigate the effects of module encapsulation in repetitively modular genomes on the search space size and bias. We introduce the concept of *modularity-preserving representations*. If a representation is modularity-preserving, the existence of modularity in the problem space is translated into a corresponding modularity in the search space. We hypothesize that such representations can be found for any given modular problem, which allows us study the impact of modularity at the genomic level. In Sect. 1 we pose two questions to focus our work. We now discuss our conclusions with respect to those questions:

1. Does the encapsulation and replacement of lower level modules or primitives with higher level counterparts, by itself, benefit evolutionary search?

We prove, under a set of assumptions, that systematically encapsulating lower order modules into higher order ones does not change the size of a search space or its structural bias: Theorem 1. We also provide an experimental analysis in support of this analytical result. Therefore, the encapsulation of modules alone does not benefit evolutionary search.

2. Under what circumstances does the encapsulation of lower level modules or

primitives into higher level modules benefit evolutionary search and how? Adding a module to a search space increases the size of the alphabet and, consequently, the size of the search space. This increase can be countered if appropriate modules are selected which decrease the length of an optimal solution. We define good modules as modules that are present in the optimal solution and bad modules as modules that are not. We provide an expression in (6) to determine when module encapsulation is advantageous. Using this expression we can answer this question as follows: encapsulating bad modules is always detrimental; encapsulating good modules is advantageous only if (6) is satisfied.

References

1. Clark, A., Thornton, C.: Trading spaces: Computation, representation, and the limits of uninformed learning. Behavioral and Brain Sciences **20** (1997) 57–90
2. Jones, T., Forrest, S.: Fitness distance correlation as a measure of problem difficulty for gas. In Eshelman, L.J., ed.: Proc. 6th Int'l Conf. on GAs. (1995)
3. Mathias, K., Whitley, L.D.: Transforming the search space with gray coding. In: Proc. IEEE Int'l Conference on Evolutionary Computation. (1994) 513–518
4. Wolpert, D.H., Macready, W.G.: No free lunch theorems for search. Technical Report SFI-TR-95-02-010, The Santa Fe Institute, Santa Fe, NM (1995)
5. Watson, R.A.: Hierarchical module discovery. In: 2003 AAAI Spring Symposium Series. (2003) 262–267
6. Koza, J.R., Streeter, M., Keane, M.: Automated synthesis by means of genetic programming. In: 2003 AAAI Spring Symposium Series. (2003) 138–145
7. Garibay, I.I., Wu, A.S.: Cross-fertilization between proteomics and computational synthesis. In: 2003 AAAI Spring Symposium Series. (2003) 67–74
8. Angeline, P.J., Pollack, J.: Evolutionary module adquisition. In: Proceedings of the second annual conference on evolutionary programming. (1993) 154–163
9. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge, MA (1994)
10. Darwen, P.J., Yao, X.: Speciation as automatic categorical modularization. IEEE Transactions on Evolutionary Computation **1** (1997) 101–108
11. Garibay, O.O., Garibay, I.I., Wu, A.S.: The modular genetic algorithm: exploiting regularities in the problem space. In: Proc. of ISCIS 2003. (2003) 578–585
12. De Jong, E.D., Oates, T.: A coevolutionary approach to representation development. In: Proc. of the ICML-2002 WS on development of rep. (2002) 1
13. De Jong, E.D.: Representation development from pareto-coevolution. In: Proceedings of GECCO 2003. LNCS series, Springer-Verlag (2003) 265–276
14. Goldberg, D.E., Korb, B., Deb, K.: Messy genetic algorithms: Motivation, analysis, and first results. Complex Systems **3** (1989) 493–530
15. Watson, R.A., Pollack, J.: Symbiotic combination as an alternative to sexual recombination in genetic algorithms. In: Proc. of PPSN VI. (2003) 262–267
16. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison Wesley (1979)
17. Garibay, O.O., Garibay, I.I., Wu, A.S.: No free luch theorem for modular genomes. Technical Report CS-TR-04-03, University of Central Florida (2004)
18. Holland, J.H.: Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, MI (1975)
19. Goldberg, D.E.: Genetic algorithms in search, optimization, and machine learning. Addison Wesley (1989)