2008

Alayzing The Effects Of Modularity On Search Spaces

Ozlem Garibay
University of Central Florida

STARS Citation

Garibay, Ozlem, "Alayzing The Effects Of Modularity On Search Spaces" (2008). *Electronic Theses and Dissertations, 2004-2019*. 3513.
https://stars.library.ucf.edu/etd/3513

Analyzing the Effects of Modularity on Search Spaces.

by

Özlem Özmen Garibay
B.S. Middle East Technical University, 1998
M.S. University of Central Florida, 2001

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2008

Major Professor:
Annie S. Wu

Abstract

We are continuously challenged by ever increasing problem complexity and the need to develop algorithms that can solve complex problems and solve them within a reasonable amount of time. Modularity is thought to reduce problem complexity by decomposing large problems into smaller and less complex subproblems. In practice, introducing modularity into evolutionary algorithm representations appears to improve search performance; however, how and why modularity improves performance is not well understood. In this thesis, we seek to better understand the effects of modularity on search. In particular, what are the effects of module creation on the search space structure and how do these structural changes affect performance? We define a theoretical and empirical framework to study modularity in evolutionary algorithms. Using this framework, we provide evidence of the following. First, not all types of modularity have an effect on search. We can have highly modular spaces that in essence are equivalent to simpler non-modular spaces. This is the case, because these spaces achieve higher degree of modularity without changing the fundamental structure of the search space. Second, for the cases when modularity actually has an effect on the fundamental structure of the search space, if left without guidance, it would only crowd and complicate the space structure resulting in a harder space for most search algorithms. Finally, we have the case when modularity not only has an effect in the search space structure, but most importantly, module creation can be guided by problem domain knowledge. When this knowledge can be used to estimate the value of a module in terms of its contribution toward building the solution, then modularity is extremely effective. It is in this last case that creating high value modules or low value modules has a direct and decisive impact on performance. The results presented in this thesis help to better understand, in a principled way, the effects of modularity on search. Better understanding the effects of modularity on search is a step forward in the larger issue of evolutionary search applied to increasingly complex problems.

To my parents, Öznur and Yusuf Özmen

Acknowledgments

I would like to thank my committee members: Dr. Guha, Dr. Lang, Dr. Georgiopoulos, and Dr. Wu. In particular, I would like to thanks my advisor, Annie S. Wu for all her support on these years. This work would have not been possible without the support of Tom O'Neal at the UCF Office or Research and Commercialization who allowed me to continue my educational goals. A big thanks for my husband, Ivan, his constant encouragement and support make it all possible, and my daughter Lia Idil that contributed by allowing Mommy to work while she play. Finally, I am forever grateful to my mom and dad who thought me my first science classes and more importantly encouraged me to pursue my dreams.

TABLE OF CONTENTS

LIST OF FIGURES . . . . **xii**

LIST OF TABLES . . . . **xviii**

I PRELIMINARIES **1**

CHAPTER 1 INTRODUCTION **2**

1.1 Complex Systems, Evolution and Modularity 3

1.2 Framework to Study Modularity 5

1.3 Strategy and Methodology 6

1.4 Contributions 9

1.5 Overview 11

CHAPTER 2 BACKGROUND **14**

2.1 Evolutionary Computation Overview 14

2.2 Related Work 15

2.2.1 Modularity Overview 15

2.2.2 Modularity in Evolutionary Computation 16

2.3 Modularity in Biology 20

CHAPTER 3 FRAMEWORK 30

3.1 Search Spaces 30

3.2 Search Space Size, Reachability and Redundancy 33

3.3 Module Encapsulation 35

II GENERAL EFFECTS OF MODULARITY IN THE SEARCH SPACE STRUCTURE 40

CHAPTER 4 INVARIANCE OF SEARCH SPACE UNDER A CLASS OF MODULE CREATION TRANSFORMATIONS 41

4.1 Search Space Reachability 42

4.2 Search Space Redundancy 46

4.3 Search Space Invariance Under the Strict-Encapsulation Transformation . . . 49

4.4 Summary 51

CHAPTER 5 EXPERIMENTAL ANALYSIS: MODULE ENCAPSULATION EFFECTS ON SEARCH SPACE REACHABILITY, REDUNDANCY AND PERFORMANCE 53

5.1 Experimental Settings 54

5.2 Part 1: Qualitative Validation of Search Space Invariance 56

5.3 Part 2: Relevance of Reachability and Redundancy 60

5.4 Part 3: Biasing Search Space by Changing Reachability and Redundancy . . 70

5.5 Summary 76

III PARTICULAR EFFECTS OF MODULARITY IN MUTATION BASED SEARCH 79

CHAPTER 6 FRAMEWORK REVISITED 80

6.1 Assumptions 80

6.2 Definitions 81

6.3 Module Encapsulation Instances 82

CHAPTER 7 ANALYSIS OF SEARCH SPACE SIZE 84

CHAPTER 8 EFFECTS OF MODULE ENCAPSULATION ON SEARCH SPACE COMPOSITON 88

8.1 Complete Module Set Encapsulation 89

8.2 Encapsulation of a Module Fully Included in the Solution 92

8.3 Encapsulation of a Module Fully Excluded From the Solution 95

8.4 Summary 97

CHAPTER 9 EFFECTS OF MODULE ENCAPSULATION ON SEARCH SPACE CONNECTIVITY 98

9.1 Analyzing Average Hamming Distance After Module Encapsulation 100

9.1.1 Complete Module Set Encapsulation 101

9.1.2 Encapsulation of a Module Fully Included in the Solution 103

9.1.3 Encapsulation of a Module Fully Excluded From the Solution 106

9.1.4 Comparison of Average Hamming Distances 109

9.2 Experimental Analysis 115

9.2.1 Methodology 115

9.2.2 Initialization Issues 117

9.2.3 Experimental Setting 118

9.2.4 Results 119

9.3 Summary 123

CHAPTER 10 DISCUSSION 126

IV CONCLUSIONS 129

CHAPTER 11 FUTURE WORK 130

CHAPTER 12 CONCLUSIONS 132

12.1 Limitations 136

12.2 Contributions 136

LIST OF REFERENCES 139

LIST OF FIGURES

2.1 Parcellation and integration, two methods of achieving modularity are illustrated. To obtain modularity, pleiotropic effects between the character groups are suppressed in parcellation and promoted in integration [Adapted from Wagner, 1995]. 23

2.2 The timeline for developmental and evolutionary processes. 24

2.3 This figure represents the genotype-phenotype mapping. The gene group consisting of G1, G2 and G3 primarily determines the character set A, B, C and D. Characters A, B, C and D constitutes the function F1. Genes G1, G2 and G3 have only a pleiotropic effect on function F2. Genes G4, G5 and G6 form another gene group which determines characters E, F and G that are combined into function F2. Gene G4, G5 and G6 have only a pleiotropic effect on function F1. 28

3.1 Genotype space $\mathcal{S}_g = \langle \mathcal{P}, \mathcal{M}, \Sigma_g, l, \mathcal{R} \rangle$, phenotype space and their mapping. . 31

3.2 This figure shows an example of module encapsulation for $\mathcal{S}_{g_2} = \mathcal{E}(\mathcal{S}_{g_1}, M \rightarrow 11)$. 36

3.3 This figure shows an example of strict-encapsulation, $\mathcal{S}_{g_2} = \mathcal{E}_s(\mathcal{S}_{g_1}, l_m = 2)$. . 38

4.1 Relations between genotype and phenotype spaces before and after strict-encapsulation as established by Lemmas 1 and 2, $\mathcal{S}_{g_2} = \mathcal{E}_s(\mathcal{S}_{g_1}, l_m)$. 42

5.1 Results for Experiment 1: GA performance for the five levels of the modularity representation pyramid described in Table 5.1. X-axis shows (A) Absolute best fitness, and (B) Generations to absolute best fitness and y-axis shows the five levels of the pyramid. The first four levels perform similarly but the last level performs slightly worse than the first four levels. 58

5.2 Experiment 2, x-axis in (A) shows the absolute best fitness and x-axis in (B) shows the generations needed to obtain this absolute best fitness, when increasing the search space size while reachability and redundancy are kept unchanged. Y-axis shows the search space size corresponding to $1CMS$, $2CMS$, $4CMS$ and $8CMS$. Changes in the search space size without any structural or redundancy bias does not affect the performance. 63

5.3 Experiments 3, x-axis in (A) shows the absolute best fitness and x-axis in (B) shows the generations needed to obtain this absolute best fitness, when removing a random set of modules. Y-axis shows the fraction of the complete module set names included in the alphabet, $1CMS$, $0.75CMS$, $0.5CMS$ and $0.25CMS$. The performance decreases as we remove larger random sets of module names from the alphabet. 65

5.4 Experiment 4, x-axis in (A) shows the absolute best fitness and x-axis in (B) shows the generations needed to obtain this absolute best fitness, when increasing the number of copies of a random subset of the alphabet. Y-axis shows the number of copies, R, of the random subset with $R = \{0, 1, 2, 4, 8, 16\}$. The performance decreases with the increase in redundancy. 69

5.5 Results for Experiment 5: Reachability in targeted parts of a search space. X-axis in (A) and (C) shows the absolute best fitness and in (B) and (D) shows the generations needed to obtain this absolute best fitness. Y-axis shows the fraction of the complete module set names included in the alphabet, $1CMS$, $0.75CMS$, $0.5CMS$ and $0.25CMS$. Experiment 5(i), (A) fitness and (B) generations when removing non-promising modules (not similar to solution). The performance increases as we remove larger non-promising sets of module names from the alphabet. Experiment 5(ii), (C) fitness and (D) generations when removing promising modules (similar to solution). The performance decreases as we remove larger promising sets of module names from the alphabet. 72

5.6 Results for Experiment 6: Redundancy in targeted parts of a search space. X-axis in (A) and (C) shows the absolute best fitness and in (B) and (D) shows the generations needed to obtain this absolute best fitness. Experiment 6(i), (A) fitness and (B) generations when increasing the number of copies of a promising module (similar to solution). Y-axis shows the number of copies, R, of a promising module name with $R = \{0, 1, 2, 4, 8, 16\}$. The performance increases as we increase the redundancy of a promising module name in the alphabet. Experiment 6(ii), (C) fitness and (D) generations when increasing the number of copies of a non-promising module (not similar to solution). Y-axis shows the number of copies, R, of a non-promising module name with $R = \{0, 1, 2, 4, 8, 16\}$. The performance decreases as we increase the redundancy of a non-promising module name in the alphabet. 75

8.1 Elements of the genotype space before ($L(\mathcal{S}_{g_O})$), and after ($L(\mathcal{S}_{g_{CM}})$, $L(\mathcal{S}_{g_{GM}})$, $L(\mathcal{S}_{g_{BM}})$) the encapsulation of a (A) complete module set, (B) good module, and (C) bad module. For all cases, $l = 3$, $l_m = 2$ and the solution string is 111. The genotype elements mapping to the solution string are circled. . . . 91

9.1 Comparison of theoretical results of average Hamming distance to solution in three structurally different search spaces: a search space with a good module of size two, with no modules and with a bad module of size two. The plot shows Equations 9.2, 9.4 and 9.7 with fixed $l_m = 2$ and various values of l, l-bits OneMax. The x-axis shows the phenotype length, l, and the y-axis shows the average Hamming distance to the solution for the three cases. The Hamming distance is smaller when a good module is encapsulated and larger when a bad module is encapsulated. The larger the phenotype length, the more prominent the effect of module encapsulation is on the average Hamming distance to solution. 111

9.2 Comparison of theoretical results of average Hamming distance to solution in three structurally different search spaces: a search space with a good module of various sizes, with no modules, and with a bad module of various sizes. The plot shows Equations 9.2, 9.4 and 9.7 with various values of l_m and fixed $l = 256$, 256-bit OneMax. 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 to solution for the three cases. The Hamming distance is smaller when a good module is encapsulated and larger when a bad module is encapsulated. The larger the module length, the less prominent the effect of module encapsulation is on the average Hamming distance to solution. 112

9.3 Comparison of the normalized average Hamming distance to solution 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 random modules. The modules are of various lengths. The target problem is 20-bit OneMax. The x-axis shows the encapsulated module length $l_m = \{2, 4, 8, 16, 20\}$ and the y-axis shows the average Hamming distance. (A) Theoretical results from Equations 9.2, 9.4 and 9.7. (B) Experimental results for a GA initial generation. The average Hamming distance is calculated as (1 − average best fitness) and reported with 95% confidence intervals. (C) Experimental results for a GA final generation. 120

LIST OF TABLES

5.1 Details for the modularity representation pyramid A_1 used in experiment 1. . 56

5.2 Elements of the complete module set of module size 4, $\mathcal{CMS}_k(\{0,1\},4)$. . . . 61

5.3 Elements of the alphabets in Experiment 3 and the first part of Experiment 5. 66

5.4 Elements of the alphabet for Experiment 4. 67

5.5 Elements of the alphabet for Experiment 6. $x = 15$ for first part and $x = 0$ for the second part of Experiment 6. 68

5.6 Elements of the alphabet for the second part of Experiment 5. 71

5.7 Elements of the alphabet for Experiment 6. $x = 15$ for first part and $x = 0$ for the second part of Experiment 6. 74

9.1 Probability density of each distance value d in search spaces $\mathcal{S}_O$, $\mathcal{S}_{GM}$ and $\mathcal{S}_{GM}$ which are search spaces before module encapsulation, after *good* module encapsulation and after *bad* module encapsulation respectively. 104

Part I

PRELIMINARIES

CHAPTER 1
INTRODUCTION

In the systems that occur in nature, or are designed by man, not all components interact strongly with other components. Most such systems are, in fact, nearly completely decomposable. That is to say, they can be subdivided into blocks in such a way that all strong interactions occur among elements in individual blocks, and only weak interactions between the blocks.

–Herbert A. Simon, 1997.

We are continuously challenged by ever increasing problem complexity and the need to develop algorithms that can solve complex problems and solve them within a reasonable amount of time. Modularity is thought to reduce problem complexity by decomposing large problems into smaller and less complex subproblems. In practice, introducing modularity into evolutionary algorithm representations appears to improve search performance [GGW03, HP03, Koz94, HLP01, HP02]; however, how and why modularity improves performance is not well understood. In this thesis, we seek to better understand the effects of modularity on search. The goal of this dissertation is to study the following question. How and why is that modularity in representations help improve evolutionary search performance? More concretely, what are the effects of module creation on the search space structure and how these structural changes affect performance. We hypothesize that modularity has a significant effect on the search space structure and that this effect can be tailored toward a performance gain, but only when there is a priori information regarding the problem class subject to the search.

1.1 Complex Systems, Evolution and Modularity

As our society evolves and grows, the phenomena that we try to understand in the sciences and the problems that we face in engineering are increasingly complex. Complex systems are usually thought of as systems with a massive number of components, but quantity alone does not amount to complexity in behavior. The complex behavior that characterize complex systems is due not only to the large amount of components but also to the nonlinear interactions among them. It is this nonlinearity that makes complex systems difficult to understand and to predict. In fact, these systems challenge ordinary mathematical methods, our computational systems required to simulate them, and more often than not our intuition.

One encouraging thought on the path towards better understanding complex systems is that most observed complex systems have a hierarchical modular structure [Wat03, CR05]. In fact, it was Herbert A. Simon that pointed out the fact that most of the world's complex systems are made up of complex subsystems which can also contain complex subsystems, and so forth [SA61, Sim69, Sim05]. Of course, these subsystems interact with each other to form the entire system, but the most interesting part of Simon's perspective, is the way in which these subsystems are known to interact. The frequencies of interactions among elements inside a particular subsystem of a system are typically one or two orders of magnitude higher than the frequencies of interaction between the subsystems. Simon called systems with this property *nearly decomposable systems* [CR05, SA61]; we simply call them *modular systems*. With this perspective, it is, perhaps, easier to observe how ubiquitous modularity is in natural and engineered systems. In nature, for example, organisms contain organs, organs consist of tissues, tissues contain cells, cells are made of molecules, and so on. A similar hierarchically modular organization can be observed in engineered systems. For instance, computer systems have modules such as memory, processing unit, and storage. These modules contain lower level modules such as integrated circuits, which in turn are made of modules such as transistors and so forth.

In Evolutionary Computation we seek to evolve solutions to complex problems. If we cast these solutions themselves to be complex systems, then we are performing evolutionary search in the realm of hierarchical modular solutions, in a sense, the very same space explored by natural biological evolution. Now, the obvious question arises. Can evolutionary algorithms be refined to better target hierarchically modular search spaces? Empirically, the answer seems to be yes, since, there are many examples in which modularity inducing mechanisms improve evolutionary search [GGW03, HP03, Koz94]; however, there is little theoretical understanding of this phenomena. In this thesis, we provide a simple theoretical framework to study modularity in search.

1.2 Framework to Study Modularity

This study focuses on linear problem representations consisting of an alphabet of symbols. A module is simply defined as a sequence of symbols of interest. Module encapsulation is the process of module creation by which a module is assigned to be represented by a new alphabet symbol. This new symbol is then incorporated into the alphabet of the representation and can appear in the encoded solutions. As a result, modules can protect and promote the reuse of their subsequences in the represented solutions. The actual working solutions are obtained by recursively processing the encoded solutions to "undo" modules and replace them by their corresponding sequences until all symbols are non-module symbols.

More formally, we model nearly decomposable subsystems as modules at the genotypic level that produce a phenotype after a process of development. Genotypes are strings in a formal language. Module definitions are rewriting rules, and the derivation of a genotype under a set of rewriting rules produces a phenotype. In addition, we adopt the view that module encapsulation or module creation is a search space transformation that is equivalent to creating a nearly decomposable subsystem. When a module is encapsulated, the net effect in the search space is to isolate the elements in that module from interactions outside the module. This isolation is evident at the genetic search level, where a module may be included or not, but it can not be altered by the genetic operators. This isolation is more subtle but present at the representation level. After a search space undergoes several module creation transformations, the resulting space is inherently hierarchical and modular. Therefore, a representation design for a given problem has to give meaning to hierarchically modular structures. Of course, the representation designer may choose to ignore these features. We work under the assumption that these underlying features are welcomed and exploited by the representation designer.

1.3 Strategy and Methodology

In order to better understand the effects of modularity on search, we study the effects of module creation on the search space structure and how those structural changes affect performance. We choose to focus on modularity in representations because most of the successful applications of modularity in evolutionary search use modular representations [HP01a, LPS01, JJ06]. In addition, modularity in representations naturally injects into the search space properties such as repetition, isolation of subcomponents and reuse. Alternative approaches such as cooperative coevolving modules [PD94, PD00, WP06, PD05] where modularity is introduced in sub-population level are beyond the scope of this work.

Our strategy is two fold. First, we analyze whether or not all module creation transformations produce an impact on the search space structure, or if there are some transformations under which the search space is invariant. This is the *invariance analysis*. Second, when the search space is not invariant, we study the changes in the search space structure and their impact on the search performance. We focus on how these changes can be designed to obtain a performance increase. This is the *change and impact analysis*. We conduct these two analyses at two different levels: *general search space structure* and *particular to a problem search space structure.* In order to conduct the invariance analysis and the change and impact analysis, it is crucial to define adequate metrics at these two levels. These metrics need to be not only descriptive of the search space state but also need to be relevant from a search performance perspective. In the first level, we analyze the properties of search space structure independent of a particular problem domain and only associated with a particular representation class. The properties of a search space structure that we study in this level are *reachability* and *redundancy.* Reachability simply determines the elements that can be accessed in a given search space. After a transformation, some elements may become unreachable or new elements may become reachable. It is simply a metric of what elements are included in a search space. Redundancy quantify how many times elements are present in a search space. After a transformation, changes in redundancy of some elements but not others can significantly bias the search towards the overrepresented sections of the space. In the second level, we analyze the properties of search space structure associated with a particular problem domain. We use a traditional domain for theoretical studies: OneMax problem. The properties of a search space structure that we study in this level are *composition* and *connectivity.* Composition measures the density of solution elements in a given search space and connectivity measures the average number of mutation steps between each elements of the search space and the solution. We can think of these metrics as measuring similar properties at two levels: general and particular to a problem class. Composition can be seen as a problem particular version of redundancy. This is the case because, once we have information of the problem class, we can refine "redundancy" to focus on the solution redundancy or solution density. In a similar way, connectivity can be seen as a problem particular version of "reachability". This is the case because, once information about the problem class is available, and if the solution is reachable, we can calculate the average distance to "reach" the solution from any other point in the search space.

Using our framework and the metrics, we can now post the specific questions we answer in this thesis. At the level of general search space structure:

- Does module creation, by itself, always results in a change in reachability and redundancy of elements in the search space? Or do reachability and redundancy of the search space remain invariant after the creation of some types of module sets?
- Are reachability and redundancy relevant structural aspects of search spaces in terms of search performance? If so, can changes in search space structure be targeted in order to improve performance?

At the level of search space structure as associated with OneMax problem:

- Does module creation, by itself, always results in a change in composition and connectivity of elements in the search space, or do the composition and the connectivity of the search space remain invariant?
- Can module creation be targeted using problem domain knowledge in order to achieve predictable increase or decrease in search performance due to changes in composition and/or connectivity of the search space?

We use mathematical and empirical methods in our analysis. In our mathematical analysis, we use standard set theory and formal language notation to define our theoretical framework. We also use basic combinatorics and probability theory to derive formulas for composition and connectivity metrics. Using these theoretical tools, we prove invariance and give equations to calculate the changes. We empirically validate these results and provide empirical performance analysis.

1.4 Contributions

This thesis contributes to the field of machine learning and evolutionary computation in the following ways:

- It provides a theoretical framework to study modularity in search. This framework defines the representation space (genotype space), the search space (phenotype space), module defining production rules, and module creation transformations. This framework allow an explicit analysis of the effects of module encapsulation in evolutionary computation. It is based on the search space structure and is independent of the search algorithm used. Module creations and deletions are viewed as a search space transformations that in effect create nearly decomposable subsystems.
- Using this framework, it provides the following metrics to analyze changes in search space structure: reachability, redundancy, composition and connectivity. It provides a theoretical analysis that shows how these metrics change with module encapsulation

transformations and provides experimental analysis to show that these metrics are strongly correlated with the search performance in genetic algorithms.

- It provides a No Free Lunch theorem for module creation at the representation level. This theorem states that systematically encapsulating lower level modules into higher level counterparts, by itself, does not benefit any search strategy, and provides proof of search space structure invariance under a particular class of module creation transformations. In other words, there are some module sets, the creation of which, do not change reachability, redundancy, composition or connectivity of the resulting search space. It provides an experimental analysis that validates this theoretical result.

- It provides experimental evidence of the existence of module creation transformations that do change search space reachability and redundancy, which result in a predictable increase or decrease in search performance. In addition, it provides experimental evidence that, at the general level, reachability and redundancy changes in random areas of the search space produce a detrimental effect in performance. At the particular problem level, reachability decrease of unfavorable areas of a search space or redundancy increase of favorable areas results in an improvement in performance.

- It provides a theoretical and experimental study describing the effects of the module creation transformation for three types of modules: good quality modules, bad quality modules and complete module sets. The quality of a module depends on its contribution towards building the solution. Under a set of assumptions, the following is established. Encapsulating a complete module set has a neutral effect on our metrics and on search performance. Encapsulating a good module is always advantageous and encapsulating a bad module is always detrimental in terms of our metrics and search performance.

1.5 Overview

The remainder of this thesis proceeds as follows. The first three chapters introduce evolutionary computation and modularity providing a review of the existing literature. Chapter 2 gives an introduction to field of evolutionary computation, and provides a literature search on modularity in evolutionary computation and modularity in biology. Chapter 3 describes a theoretical framework which defines the representation space, the search space, module defining production rules, and module creation transformations.

The next two chapters provide an analysis of the effects of module creation transformations on the general search space structure. Chapter 4 gives a theoretical analysis of invariance of search space under a class of module creation transformations. We prove that under a class of module creation transformations, the search space structure does not change in terms of reachability and redundancy metrics. Chapter 5 provides an experimental analysis on the effects of module encapsulation on search space reachability, redundancy and performance. We provide an empirical validation of invariance of search space structure under a class of module creation transformations. We also provide an experimental analysis on the effects of arbitrary module encapsulation and encapsulation of a “good” and “bad” modules on search space reachability, redundancy and performance. We show that arbitrary module encapsulation is detrimental in terms of reachability, redundancy and performance. We also show that encapsulating a “good” module improves the reachability, redundancy and performance of the search space while encapsulating a “bad” module is detrimental on all these three metrics. This chapter also shows that reachability and redundancy are relevant metrics to performance in search.

The next five chapters provide an analysis of the effects of module creation transformations on search space structures where we focus on a particular problem class. Chapter 6 describes the non-standard framework which is slightly different that the standard framework given in Chapter 3. Chapter 7 gives an analysis on the search space size after module creation transformation. We show that the search space size always increases after module creation transformations when the new module names are added to the alphabet without replacing the existing elements of the alphabet. Chapter 8 provides a theoretical analysis of module creation transformations on search space composition. We show that the search space remains invariant in terms of search space composition under a class of module creation transformations. We also show that encapsulating a “good” module improves search space composition while encapsulating a “bad” module is detrimental in terms of search space composition. Chapter 9 provides a theoretical and experimental analysis of module creation transformations on search space connectivity. We show that the search space remains invariant in terms of search space connectivity under a class of module creation transformations.

We also show that encapsulating a “good” module improves search space composition while encapsulating a “bad” module is detrimental in terms of search space composition. Chapter 10 discusses the theoretical and experimental results we obtain in Chapter 8 and Chapter 9 Finally, Chapter 11 discusses possible future directions of research and Chapter 12 gives concluding remarks.

CHAPTER 2
BACKGROUND

2.1 Evolutionary Computation Overview

Evolutionary computation (EC) is a field in computer science inspired by the Darwinian concept of evolution by natural selection [Dar59]. EC algorithms or Evolutionary Algorithms (EA) are iterative stochastic parallel search methods. The idea of using genetic evolution for computation was first suggested by Turing more than half century ago [Tur48]. The *schemata theorem*, the foundation of evolutionary algorithms, was outlined by Holland [Hol62, Hol75] a few decades later. The *schemata theorem* states that the lower level short highly fit strings (in other words, partial solutions) are combined into high level strings that are also likely to have higher fitness.

Evolutionary algorithms sample the search space with a population. The use of a population of individuals allows EAs to perform a parallel search. The evolutionary cycle consists of evaluation, selection and breeding of the next generation through evolutionary operators. The initial population is randomly generated. Individuals are evaluated and undergo a fitness based selection process. The selected individuals create the next generation via evolutionary operators such as mutation and crossover. This evolutionary cycle iterates until a solution is found or a termination criterion is met.

Evolutionary Algorithms have been successfully applied to many computationally difficult problems such as job scheduling, parameter optimization and electronic circuits design. Although, the field is tending to unification, there are four main historical subfields: genetic algorithms (GAs) (see for example, Holland [Hol75], De Jong [De 75] and Goldberg [Gol89]), evolutionary strategies (ES) (see for example, Schwefel [Sch02, Rec65]), evolutionary programming (EP) [Fog62, Fog64, LW66] and genetic programming (GP) [Koz92, Koz99, Koz99, Koz03].

2.2 Related Work

2.2.1 Modularity Overview

Modularity is a common characteristic of many artificial and natural systems. Systems with separable or nearly separable units are called modular. Modules are repetitively used design units [Mei04] that are easy to dissociate, recombine, and reuse in different systems [NM04]. Alternatively, a module is a part of a system that has an independent function, but, at the same time, it is weakly related to the other parts of the system. The degree of modularity [SW04] depends on the interdependency between the modules and decreases with the increase of interdependency [Lip04, LPS01, DTW04, WP05]. In this thesis, a module is simply a sequence of genomic primitives. Genomic primitives include actual system primitives as well as other lower level modules. This way of defining modules relates to Holland's building blocks hypothesis [Hol75]. Modules, however, differ from building blocks in their robustness against genetic operators. For example, an encapsulated module can not be disrupted by recombination operators while building blocks can.

Modular systems are advantageous in their easy assembly, focused repair, and flexible arrangement of components. Natural and artificial systems with complex design often have modular structure [Sim05]. Having such structure can be beneficial in terms of increasing evolvability and adaptation [Wag95, WA96]. Evolvability and adaptability are improved by reducing the pleiotropic effect of mutation between the modules [Alt05] and decreasing the complexity of the complex adaptive systems via decomposability [GNC01, FMV99]. In addition, modularity can also be beneficial in terms of increasing robustness. In fact, although the origin of modularity in biology is unknown, one hypothesis states that modularity is product of natural selection for robustness [CR05]. Because these advantageous and others, modularity has been studied in many fields such as complex systems, engineering design, evolutionary biology and developmental biology. In this thesis, we focus on modularity from the evolutionary computation perspective.

2.2.2 Modularity in Evolutionary Computation

The concept of modularity can be intuitively thought as forming and reusing high level building blocks from lower level building blocks. The modularity of a problem is usually assumed to be preserved in its problem representation [GGW04a]. Studies on modularity in evolutionary computation (EC) have used different types of representations including linear and tree structures (cf. [AP94, AP92, GGW04b, Koz94]). What is common for all representation types is that a module is a subcomponent of the representation. This module has meaning independent of the rest of the representation and is created by encapsulating the subcomponent into an atomic unit that cannot be disrupted by genetic operators.

Previous studies have shown that modularity can improve the performance of the EC search process [DO02, GGW03, OR96, PG01, SFH03]. This improvement is attributed to several features that emerge in modular representations: scalability, reuse, and robustness [GB02, HP01b, HP02, Hor05, LPS01]. Scalability is the ability of a search algorithm to find solutions that can solve a problem when the size of the problem changes. Modular problems can be divided into smaller and less complex problems which are computationally less expensive to solve than the problem as a whole. Modularity improves scalability by reducing the complexity of a problem by dividing it into smaller and less complex problems [Hor05, HP03, JJ06, PD00]. Once formed, modules are reusable. If a module appears multiple times in a solution string, module encapsulation eliminates the need for evolving again a segment of the solution that has already been evolved. Reuse of a module indirectly reduces the size of a problem and therefore can improve scalability (cf. [AP93, Hor05, Koz94, KSK03, DTW04]). Robustness refers to the fact that the partial solutions that have been evolved are less likely to be disrupted by the genetic operators [OYR04, VML04]. Forming a module and encapsulating it into an atomic unit makes the subsequence represented by the module difficult to disrupt. Solutions containing modules are robust against search operators because the content of a module can not be changed by the operators unless a module is expanded. Modules can, however, be replaced by other modules or by non-module symbols.

Creating a module usually introduces new elements into a search space, therefore, increases redundancy of some of the search space elements. Redundancy generally increases the evolvability by increasing accessibility and added connectivity between the phenotypes [ESS01]. Elements with higher redundancy spread more rapidly in a population [Tob05], but it can improve performance only if a-priori information of the optimal solution is available [RG03, Rot02].

The formation of modules in EC systems can be classified as *implicit* or *explicit* depending on whether modules emerge as a result of indirect forces or are explicitly created. Implicit mechanisms form modules by rearranging the encoded information [GK01, GNC01, HG96, KP03, Bon02]. Over time, the search process evolves individuals in which related information is arranged in closer proximity [GKD89, HG96, Har97]. Because regions that are in close proximity are less likely to be disrupted by genetic operators, we say that they implicitly form a module [WL95]. Implicit modularity also plays a role in self-similar genomes and self-organizing modularization. In the former case, genomic information self-organize into segments or modules that are self-similar with respect to fitness [GWG06, WG02]. In the latter case, the authors include selection pressure to favor individuals that meet their particular definition of modularity [DU05].

Explicit mechanisms form modules by explicitly encapsulating modular sequences into atomic units. Module encapsulation is the process of selecting a substring of interest, renaming it with a new symbol, and adding the new symbol to the alphabet. Once encapsulated, modules may be kept throughout the evolutionary search process or may be revisited periodically to determine whether they should be retained or released. A simple mechanism for revisiting modules is to select a module randomly and evaluate it based on some performance metric such as fitness or usage. For example, Angeline and Pollack select segments of an individual randomly for encapsulation and evaluate them on the reproductive advantage they provide to individuals [AP92, AP93]; Rosca and Ballard select modules by looking for commonalities in the better fit individuals in a population [RB94]; and De Jong and Oates select the most frequent building block and make an encapsulation decision based on its performance contribution [DO02].

In explicit modularity, the quantity and the content of modules can be defined statically or dynamically. The quantity of modules refers to the number of modules in the alphabet and the content of a module is the substring represented by that module. In static modularity, both the quantity and the content of modules are pre-defined before the EC search begins and remain unchanged throughout a run [Hor05]. In dynamic modularity, the quantity and the content of the modules are determined during the search [AP92, AP93, AP94, RB94] For example, Koza's Automatically Defined Functions (ADFs) are modules whose content and quantity evolve dynamically along with the content of the main function [Koz94]; Potter and De Jong's cooperative coevolution decomposes problems into separate components that are similar to modules. Early studies allowed only the content of these components to emerge [PD94], but more recent work allowed both the content and the quantity of these components to evolve dynamically [PD00].

Despite many examples of modularity improving search performance and many successful approaches to incorporating modularity in EC problem representations, there is not theoretical explanation that we are aware of on how modularity affects the structure of the search space and the search process [GW08]. We attempt to start filling this gap with this study. We focus on explicit modularity in general and we expect our results to be applicable to both dynamic and static modularity.

2.3 Modularity in Biology

The concept of modularity is one of the most important concepts in biology [Gil06]. Modularity has been studied across different fields. Most relevant to this thesis, it has been studied in developmental and evolutionary biology.

Each field defines modularity and modules in a different way based on their scope. Although the concept of modularity is simple, finding a unified definition is difficult. Bolker [Bol00] defines modularity as a biological entity which has internal integration and external connectivity. Schlosser & Wagner [SW04] define modules in a similar way: modules are integrated autonomous units. Raff [Raf96] describes modules as dynamic units of a system rather than only a part of it. Wagner [Wag96] defines a module as a part of phenotype that is relatively independent from the rest but remains incorporated via pleiotropy. In a similar way, Callebaut et al. [CR05] defines modules as semi-independent parts of an organism such as arms, kidneys, heart and so on. These parts are tightly integrated within a module, but loosely integrated with other parts of the organism. For instance, a hand together with its fingers compose a module. While the hand is connected to other organs, it is more tightly connected to its fingers in terms of location and purposeful function. On the contrary, the ties between kidney and fingers are not as strong as the ties between hand and fingers, although they are connected through veins, etc. Due to the various definitions and multiple levels of modularity, it is not always easy to distinguish between what is a module and what is not a module in biology [SW04]. Determining whether a part of a system is a module or not often depends on the context in which we analyze the system.

It has been argued that modules are necessary for adaptive evolution. Lewontin and Bonner [Lew78, Bon88] suggest that modules exist because they are helpful and probably necessary for evolution. Similarly, Brandon [Bra05] argues the existence of modules, and bases his argument on fossil data. For example, the earliest mammalian tetrapod has forelimbs and hind limbs which were not developmentally and functionally different. As a result of evolution; however, the forelimbs changed and became flippers on a whale, wings on a bat and hand of a human. Although forelimbs change into very different forms in different species, some other characteristics remain relatively similar. For instance, the circulatory systems of whales and bats are very similar, while their forelimbs, flippers and wings, are different. The evolution of forelimbs occur independently from the rest of the organism including the hind limbs. This example suggest that the forelimbs and hind limbs are separate evolutionary modules.

Altenberg [WA96, Wag95] investigates module formation and propose two possible methods to achieve modularity. Altenberg defines the partitioning of a high level organism as *parcellation*, and the combination of low level building blocks as *integration*. Parcellation achieves modularity by restraining pleitropy among gene groups, and integration builds modules by establishing pleitropy between the gene groups. Parcellation and integration are illustrated in Figure 2.1. To obtain modularity, the pleiotropic effects between the character groups are suppressed in parcellation, and are promoted in integration. Because, it is more likely that modularity emerges due to evolutionary modification rather than being a basic feauture of all living beings, Wagner and Altenberg [WA96, Wag95] suggest that parcellation is the most likely candidate to explain module formation in nature.

2.3.0.1 Developmental Modularity

Developmental modules are autonomous parts of an embryo that can individually develop to become complete or almost complete structures. An example of a developmental module is a limb bud, because it develops independently once its developmental process is started [Wag04]. Developmental modules may also be considered as phenotypic expressions of genes in an environment [Spe02].

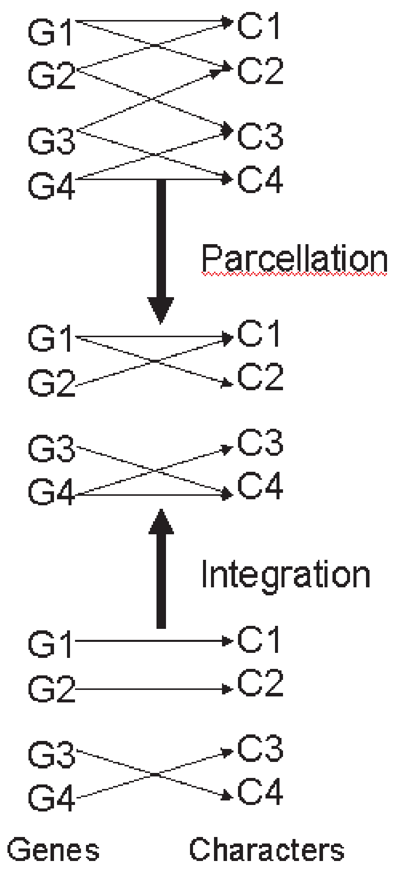


Figure 2.1: Parcellation and integration, two methods of achieving modularity are illustrated. To obtain modularity, pleiotropic effects between the character groups are suppressed in parcellation and promoted in integration [Adapted from Wagner, 1995].

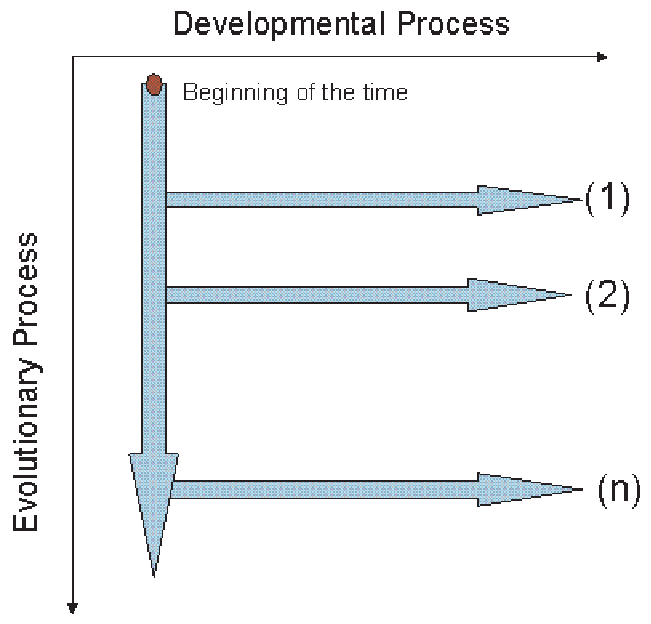


Figure 2.2: The timeline for developmental and evolutionary processes.

Developmental biology studies the development of an organism from an embryo. The stages of organ development from a single cell and cell groups are the main focus of this field. Modularity is considered to be a key component of development, because organisms are highly modular [SW04]. Organisms consist of units that are functionally nearly independent: organs. Organs can be divided into smaller subcomponents such as tissue and so on. Developmental biology studies how lower level organizational and functional units are combined into higher level units or organisms. In other words, it takes a bottom-up approach and characterizes modules as lower level parts that contributes to a whole or a higher level function [Bol00].

The time scale for developmental and evolutionary processes are demonstrated in Figure 2.2. A developmental process may last up to a lifetime of an organism. An evolutionary process, on the other hand, takes millions of years. In order to explain the developmental and the evolutionary points of view of modularity, we will take limb buds as an example for both evolutionary and developmental modularity [Wag04]. The timeline for the developmental process is shown in the horizontal line while the timeline for the evolutionary process is shown in the vertical line. The developmental process of a limb bud starts with an embryo and continues until the limb buds are fully developed. Considering that the limb bud has changed throughout the evolution, the developmental process at different evolutionary stages of a limb bud would be slightly different. For instance, development of a limb bud in the evolutionary stage shown with the horizontal line (1) may be different than the developmental process in the evolutionary stage shown with line (2). Each horizontal line represents the developmental process of a limb bud at a particular evolutionary stage. The vertical line shows the evolution of a module: how it has changed throughout the evolutionary process to become the one currently at work.

Developmental modularity is considered to be advantageous for development and evolution of organisms. In fact, Gilbert [Gil06] states that "development depends on modularity". Developmental modularity increases the capacity of development by increasing robustness, flexibility, and complexity of the organism under development [SW04, Gil06]. Modular systems can usually remain functional in case of one or more of their subsystems becomes defective [Gil06]. In non-modular systems, on the other hand, a defective unit may cause whole system to fail. Thus, modularity increases robustness and flexibility. Gilbert [Gil06], also states that systems are made from preassembled subsystems. Without these intermediate subsystems, it would not be possible to construct complex systems. The subsystems can be used in different context which allows generation of more complex systems. In other words, recombination of developmental modules increases complexity [SW04]. For example, left and right arms are two distinct developmental modules, but they are from same evolutionary module. Thus, the arm is evolved once and reused. Finally, developmental modules may contribute to increase evolvability [Sch04]. The reusability of developmental modules is a key property that facilitates evolvability [Ste95]. Also, developmental modularity may facilitate evolvability by increasing the interdependency between the module fitness contribution [Sch04].

2.3.0.2 Evolutionary Modularity

Evolutionary modules are subcomponents of an organism that have undergone evolutionary change relatively independently from the rest of the organism [Bra99]. More precisely, " ... an evolutionary module is some feature of an organism that has a unitary ecological function and genetic/developmental architecture that allows it to evolve in a "quasi-independent" way from other features" [Bra05]. In order for a part of an organism to be an evolutionary module, it must have a function to contribute to the organism's fitness, and it needs to be quasi-independent from other traits so that it can evolve independently. For example, forelimb is an evolutionary module [Bra05], because it contributes to the organism fitness and, as explained earlier, it has evolved almost independently.

In contrast with developmental biologists, evolutionary biologists take a top-bottom approach at modularity and characterize modules as subcomponents of a whole, or of a higher level function [Bol00]

Evolutionary modules are tightly related to their underlying genes [WA96]. In fact, the genetic representations of modular structures or functional units are also modular at the genomic level [Bra99]. These representations are not completely independent because of the inter-modular connectivity due to pleiotropy. Figure 2.3 shows a genotype to phenotype map. Wagner & Altenberg [WA96] use this figure to explain how evolutionary modularity is tightly related to the underlying genes. In Figure 2.3, a particular group of genes determines a particular character. This genotype to phenotype mapping is shown in Figure 2.3. The gene group consisting of G1, G2 and G3 primarily determines the character set A, B, C and D. Characters A, B, C and D form the function F1. Gene G3 also affects character E which is a component of function F2. Genes G4, G5 and G6 form another gene group which determines characters E, F and G that combine to form function F2. Genes G4 and G5 also affect function F1 to a lesser degree via relations to characters B and C. As a result, we can say that genes G1, G2, and G3 have a direct effect on function F1 and a pleiotropic effect on function F2. The solid lines show the strong relation between the characters and the functions. The dotted lines show the weak relations. Similarly, genes G4, G5, and G6 have a direct effect on function F2 and a pleiotropic effect on function F1. In general, genomes have group of genes that work together and each group maps onto a certain function. As a result, modularity in the genotype is preserved in the phenotype.

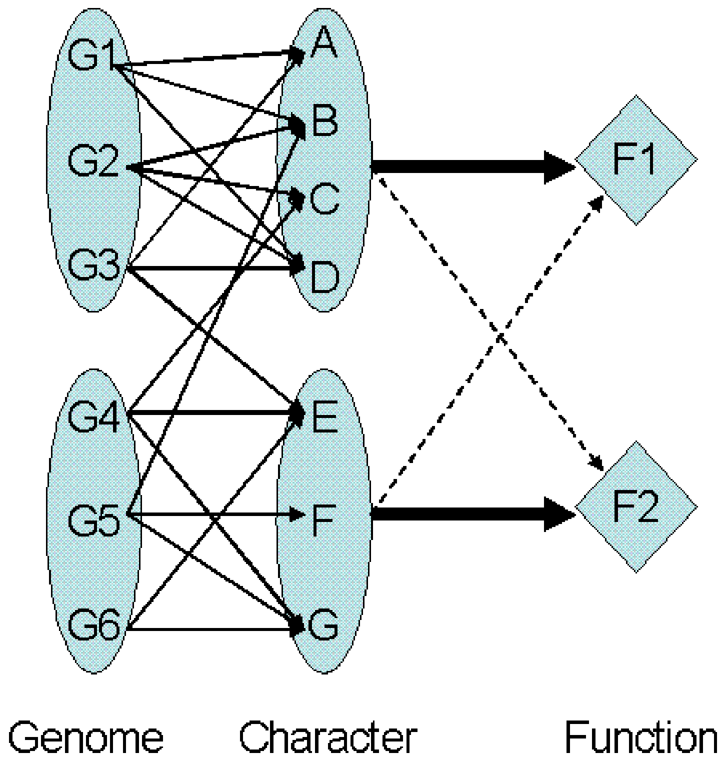


Figure 2.3: This figure represents the genotype-phenotype mapping. The gene group consisting of G1, G2 and G3 primarily determines the character set A, B, C and D. Characters A, B, C and D constitutes the function F1. Genes G1, G2 and G3 have only a pleiotropic effect on function F2. Genes G4, G5 and G6 form another gene group which determines characters E, F and G that are combined into function F2. Gene G4, G5 and G6 have only a pleiotropic effect on function F1.

Furthermore, the mapping between genotype and phenotype is also modular [Wag95, Alt05] and Altenberg [Alt05] goes as far as to state that the modularity is the most important property of the genotype-phenotype mapping.

CHAPTER 3
FRAMEWORK

We 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 traditional search spaces and search spaces augmented with modules.

3.1 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 similar to definitions in previous work [GW07, DTW04, Hor05].

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:

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

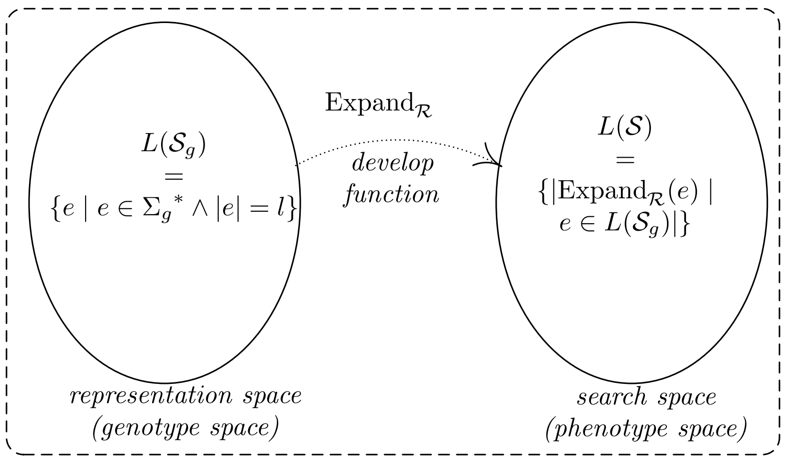

$L(\mathcal{S}_g)$
$=$
$\{e \mid e \in {\Sigma_g}^* \wedge |e| = l\}$
$\text{Expand}_{\mathcal{R}}$
develop function
$L(\mathcal{S})$
$=$
$\{|\text{Expand}_{\mathcal{R}}(e) \mid e \in L(\mathcal{S}_g)|\}$
representation space (genotype space)
search space (phenotype space)

Figure 3.1: Genotype space $\mathcal{S}_g = \langle \mathcal{P}, \mathcal{M}, \Sigma_g, l, \mathcal{R} \rangle$, phenotype space and their mapping.

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; and $\mathcal{R}$ is the set of module defining rules. Figure 3.1 shows a graphical depiction of a genotype space, phenotype space and their mapping.

The *phenotype space* or *search space*, $\mathcal{S}$, is the space of all possible candidate solutions. The elements of the phenotype space are strings over the alphabet of primitives only.

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 e is 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 e as $\text{Expand}_{\mathcal{R}}(e)$, 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 e 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 $\text{Expand}_{\mathcal{R}}(e)$ has been *generated* from e using the rewriting rules $\mathcal{R}$. Notice that, our definition of $\mathcal{R}$ guarantees that for any string $e \in \{\mathcal{P} \cup \mathcal{M}\}^*$, $\text{Expand}_{\mathcal{R}}(e) \in \mathcal{P}^*$ can be computed in a finite amount of rewriting rule applications.

The set of elements of the genotype space $\mathcal{S}_g$, denoted by $L(\mathcal{S}_g)$, are all strings over the genotype space alphabet Σ_g of length l:

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

Expanding all of the elements in the genotype space into their phenotype form gives us the phenotype space $\mathcal{S}$. For the remainder of this thesis, references to search space refers to the phenotype space We define the elements of the phenotype space or search space, $\mathcal{S}$, denoted by $L(\mathcal{S})$ as the following multiset[1]

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

where, $L(\mathcal{S}_g)$ denotes the elements of genotype space $\mathcal{S}_g$. $L(\mathcal{S})$ is the multiset of all strings in the genotype space $\mathcal{S}_g$ in their expanded form. We denote the size of genotype and phenotype space to be $|L(\mathcal{S}_g)|$ and $|L(\mathcal{S})|$, respectively.

3.2 Search Space Size, Reachability and Redundancy

Multiple elements in $L(\mathcal{S}_g)$ may expand to the same element in $L(\mathcal{S})$. The multiplicity 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:

$$|L(\mathcal{S}_g)| = |L(\mathcal{S})|$$

Notice that, also by definition, the elements of $L(\mathcal{S}_g)$ are fixed length strings of length l over Σ_g and the elements of $L(\mathcal{S})$ are potentially variable length strings over $\mathcal{P}$. Clearly, $\text{Expand}_{\mathcal{R}}$ is the function used to map the genotype to the phenotype on our modular search spaces. Therefore, for this thesis, the genotype to phenotype mapping is a generative process

[1] Multisets are sets that allow repeated elements, denoted by $\{\!|\ |\!\}$. The number of times an element is repeated in a given multiset is called the *multiplicity* of that element and the size of a multiset is the summation of the multiplicities of all its elements. For instance, in $\{\!|A, A, B|\!\}$, element A multiplicity is two, element B multiplicity is one, and the size of the multiset is three. Two multisets are said to be equal if they have the same elements and their elements have the same multiplicities. If they only have the same elements, we call them *set-equal*, $\overset{set}{=}$. For example, $\{\!|A, A, B|\!\}$ and $\{\!|A, B|\!\}$ are set-equal but not multiset equal.

determined solely by the module creating rules in $\mathcal{R}$. The size of the search space is $|L(\mathcal{S})| = |\Sigma_g|^l$. This is a direct consequence of our definition of $L(\mathcal{S}_g)$.

We are interested in the bias produced by module definitions. The reachability of elements of a search space can be biased after module creation by changing the availablity of some elements in a search space. If $L(\mathcal{S})$ contains all possible strings of primitives of a given length t, then we say that the search space has no bias in the reachability of elements for length t because there are no unreachable strings of primitives of that length. The search space has a bias in reachability otherwise. The redundancy of elements of a search space can be biased after module creation because module creation results in multiple elements of the genotype space mapping to identical strings in the phenotype space. As a result, some strings of primitives appear multiple times in the search space. For instance, the genotype space $\mathcal{S}_g = \langle \mathcal{P} = \{0, 1\}, \mathcal{M} = \{A\}, \Sigma_g = \{1, 0, A\}, l = 8, \mathcal{R} = \{A \rightarrow 11\}\rangle$ has no bias in reachability of elements of search space $\mathcal{S}$ for $t = l$ since all strings of primitives of length 8 can be reached; however it has a bias in redundancy since, for instance, the string "00000000" can only be derived from "00000000" and the string "111111111" can be derived from "A1111111", "1A111111", "11A11111", etc.

Formally, for a given search space $\mathcal{S}$:

- if

$$L(\mathcal{S}) \not\supseteq \{e \mid e \in \mathcal{P}^* \wedge |e| = t\}$$

is true, we say that there is a bias in the reachability of the elements of $\mathcal{S}$ for length t. There is no bias in reachability otherwise.

- if

$$\exists_{e_1,e_2 \in L(\mathcal{S}_g)}[(\text{Expand}_{\mathcal{R}}(e_1) = \text{Expand}_{\mathcal{R}}(e_2)) \wedge (e_1 \neq e_2)]$$

  is true, we say that there is a bias in the redundancy of the elements of $\mathcal{S}$. There is no bias otherwise.

3.3 Module Encapsulation

Module encapsulation or *module creation* is the process of selecting and 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 $\mathcal{R}$. The *encapsulation* of a module, $\mathcal{E} : \mathcal{S}_g \times R_k \mapsto \mathcal{S}_g$ is defined as follows:

$$\begin{aligned} &\mathcal{E}(\mathcal{S}_g, M_k \to w_k) = \\ &\qquad \langle \mathcal{P}, \mathcal{M} \cup \{M_k\}, \Sigma_g \cup \{M_k\}, l, \mathcal{R} \cup \{M_k \to w_k\} \rangle \end{aligned}$$

where, $\mathcal{S}_g = \langle \mathcal{P}, \mathcal{M}, \Sigma_g, l, \mathcal{R} \rangle$ is a genotype space, $M_k \to w_k$ 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 $|w_k| \leq l$. We view the process of module creation as a search space transformation. Figure 3.2 shows an encapsulation example. Given a module of length l_m, we define *Complete Module*

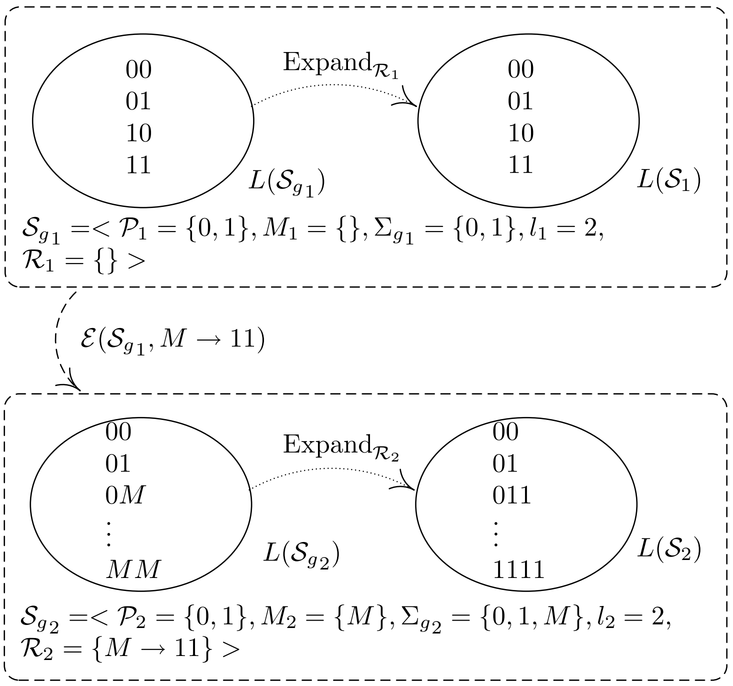

00
01
10
11
Expand$_{\mathcal{R}_1}$
00
01
10
11
$L(\mathcal{S}_{g_1})$
$L(\mathcal{S}_1)$
$\mathcal{S}_{g_1} =< \mathcal{P}_1 = \{0,1\}, M_1 = \{\}, \Sigma_{g_1} = \{0,1\}, l_1 = 2,$
$\mathcal{R}_1 = \{\} >$
$\mathcal{E}(\mathcal{S}_{g_1}, M \to 11)$
00
01
$0M$
$\vdots$
MM
Expand$_{\mathcal{R}_2}$
00
01
011
$\vdots$
1111
$L(\mathcal{S}_{g_2})$
$L(\mathcal{S}_2)$
$\mathcal{S}_{g_2} =< \mathcal{P}_2 = \{0,1\}, M_2 = \{M\}, \Sigma_{g_2} = \{0,1,M\}, l_2 = 2,$
$\mathcal{R}_2 = \{M \to 11\} >$

Figure 3.2: This figure shows an example of module encapsulation for $\mathcal{S}_{g_2} = \mathcal{E}(\mathcal{S}_{g_1}, M \to 11)$.

Set, $\mathcal{CMS}$, as:

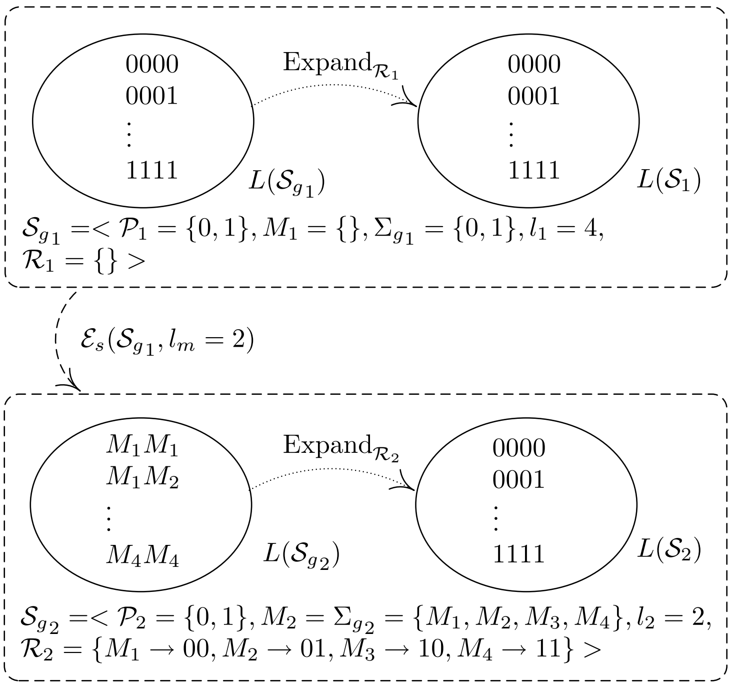
$$\mathcal{CMS}(\Sigma_g, l_m) = \{(M_i \rightarrow w) | w \in {\Sigma_g}^* \wedge |w| = l_m \wedge i \textit{ is a unique index for } w\}$$

In other words, a *complete module set* is the set of modules defined by all strings of length l_m over $\mathcal{P}$. In effect, there will be $|\mathcal{P}|^{l_m}$ modules in this set, one for every permutation of length l_m of the primitive symbols.

Strict-encapsulation is the process of creating a complete module set of size l_m over the current genomic alphabet Σ_g and then replacing the current alphabet with the newly created module names. This process changes the genotype space by adding a set of modules and by completely replacing the genomic alphabet with module symbols associated with the newly added modules. Notice that there are $|\Sigma_g|^{l_m}$ modules of size l_m that can be created over Σ_g. The new individual length is l/l_m, since individuals consist only of new modules encapsulating sub-strings of size l_m. Formally, *strict-encapsulation*, $\mathcal{E}_s : \mathcal{S}_g \times l_m \mapsto \mathcal{S}_g$ is defined as follows:

$$\mathcal{E}_s(\mathcal{S}_g, l_m) = \langle \mathcal{P}, \eta(\mathcal{CMS}(\Sigma_g, l_m)) \cup \mathcal{M}, \eta(\mathcal{CMS}(\Sigma_g, l_m)), l/l_m, \mathcal{CMS}(\Sigma_g, l_m) \cup \mathcal{R} \rangle$$

where $\mathcal{S}_g = \langle \mathcal{P}, \mathcal{M}, \Sigma_g, l, \mathcal{R} \rangle$ is a genotype space, l_m is an arbitrary module size, l/l_m is an integer, and the function $\eta : \mathcal{R} \mapsto \mathcal{M}$, is defined as $\eta(\mathcal{R}') = \{M_x \mid (M_x \rightarrow w_x) \in \mathcal{R}'\}$. The function η takes a module rule or a module rule set as parameter and returns the module module name or the set of the module names respectively. Figure 3.3 shows a strict-encapsulation example. *Strict-encapsulation without replacement* of a genotype space $\mathcal{S}_g$ is the process of creating all possible modules of a given size l_m over the current genomic



0000
0001
⋮
1111
$L(\mathcal{S}_{g_1})$
$\text{Expand}_{\mathcal{R}_1}$
0000
0001
⋮
1111
$L(\mathcal{S}_1)$
$\mathcal{S}_{g_1} =< \mathcal{P}_1 = \{0,1\}, M_1 = \{\}, \Sigma_{g_1} = \{0,1\}, l_1 = 4,$
$\mathcal{R}_1 = \{\} >$
$\mathcal{E}_s(\mathcal{S}_{g_1}, l_m = 2)$
M_1M_1
M_1M_2
⋮
M_4M_4
$L(\mathcal{S}_{g_2})$
$\text{Expand}_{\mathcal{R}_2}$
0000
0001
⋮
1111
$L(\mathcal{S}_2)$
$\mathcal{S}_{g_2} =< \mathcal{P}_2 = \{0,1\}, M_2 = \Sigma_{g_2} = \{M_1, M_2, M_3, M_4\}, l_2 = 2,$
$\mathcal{R}_2 = \{M_1 \rightarrow 00, M_2 \rightarrow 01, M_3 \rightarrow 10, M_4 \rightarrow 11\} >$

Figure 3.3: This figure shows an example of strict-encapsulation, $\mathcal{S}_{g_2} = \mathcal{E}_s(\mathcal{S}_{g_1}, l_m = 2)$.

alphabet Σ_g and then adding to the current alphabet the newly created module names. The difference with the previous definition is that in strict-encapsulation the newly created module names replace the alphabet, and in this definition they are just added to the alphabet. Also in this case the genome length is not affected.

Formally, *Strict-encapsulation without replacement*, $\mathcal{E}_s : \mathcal{S}_g \times l_m \mapsto \mathcal{S}_g$ is defined as follows:

$$\mathcal{E}_s(\mathcal{S}_g, l_m) = \langle \mathcal{P}, \eta(\mathcal{CMS}(\Sigma_g, l_m)) \cup \mathcal{M}, \eta(\mathcal{CMS}(\Sigma_g, l_m)) \cup \Sigma_g, l, \mathcal{CMS}(\Sigma_g, l_m) \cup \mathcal{R} \rangle$$

where $\mathcal{S}_g = \langle \mathcal{P}, \mathcal{M}, \Sigma_g, l, \mathcal{R} \rangle$ is a genotype space, and l_m is an arbitrary module size.

Part II

GENERAL EFFECTS OF MODULARITY IN THE SEARCH SPACE STRUCTURE

CHAPTER 4
INVARIANCE OF SEARCH SPACE UNDER A CLASS OF MODULE CREATION TRANSFORMATIONS

Module encapsulation changes the genotype space, $\mathcal{S}_g = \langle \mathcal{P}, \mathcal{M}, \Sigma_g, l, \mathcal{R} \rangle$, by changing the genomic alphabet, Σ_g, and by changing the rewriting rules set $\mathcal{R}$. These changes in the genotype space may or may not result in changes on the associated search space structure. We are interested in investigating whether or not there is a class of module encapsulation transformations under which the search space structure is invariant in terms of reachability, redundancy and search space size. In order to establish the search space invariance under a class of transformations, we need to prove that reachability, redundancy and the search space size are invariant under that class of transformations. We establish search space invariance under a particular class of module creation transformations: strict-encapsulation. We use the definitions introduced in the previous section to prove two lemmas. In the first one, we prove that the search spaces before and after strict-encapsulation are set-equal [1] —they have the exact same elements present, but with potentially different multiplicities. We use this lemma to establish invariance of search space reachability under strict-encapsulation. Having established reachability, we proceed to work on redundancy and introduce a second lemma. In this lemma, we prove that the representation spaces before and after strict-encapsulation are one-to-one and onto. We use this second lemma to establish invariance

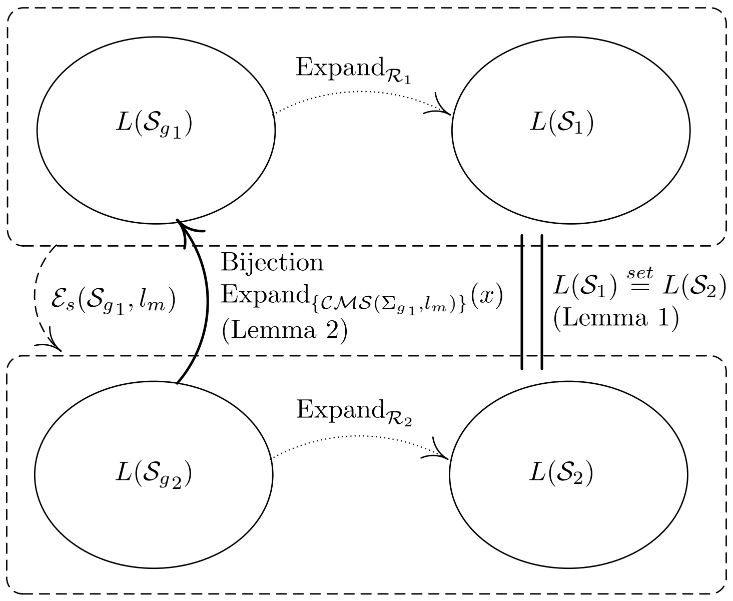

$L(\mathcal{S}_{g_1})$
$\text{Expand}_{\mathcal{R}_1}$
$L(\mathcal{S}_1)$
$\mathcal{E}_s(\mathcal{S}_{g_1}, l_m)$
Bijection
$\text{Expand}_{\{\mathcal{CMS}(\Sigma_{g_1}, l_m)\}}(x)$
(Lemma 2)
$L(\mathcal{S}_1) \overset{set}{=} L(\mathcal{S}_2)$
(Lemma 1)
$L(\mathcal{S}_{g_2})$
$\text{Expand}_{\mathcal{R}_2}$
$L(\mathcal{S}_2)$

Figure 4.1: Relations between genotype and phenotype spaces before and after strict-encapsulation as established by Lemmas 1 and 2, $\mathcal{S}_{g_2} = \mathcal{E}_s(\mathcal{S}_{g_1}, l_m)$.

of search space size and redundancy under the strict-encapsulation transformation. Finally, we use both lemmas to prove search space invariance under strict-encapsulation. Figure 4.1 offers a graphical depiction of the implications of strict-encapsulation.

4.1 Search Space Reachability

In order to establish invariance of search space reachability under the strict-encapsulation transformation, we prove in the folowing lemma that the search spaces before and after strict-encapsulation are set-equal. In essense, if the search spaces are set-equal, they contain the same elements. Therefore, if the search space before strict-encapsulation contains all possible elements of a given length, so does the search space after strict-encapsulation. Similarly, if the search space before strict-encapsulation does not contain all possible elements of a given length, so does not the search space after strict-encapsulation.

Lemma 1 *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be search spaces such that $\mathcal{S}_{g_2} = \mathcal{E}_s(\mathcal{S}_{g_1}, l_m)$, then*

$$L(\mathcal{S}_1) \overset{set}{=} L(\mathcal{S}_2)$$

Proof: To prove the set equality, we will consider both multisets to be just sets and proceed to construct $L(\mathcal{S}_1)$ from $L(\mathcal{S}_2)$. Let $\mathcal{S}_{g_1} = \langle \mathcal{P}_1, \mathcal{M}_1, \Sigma_{g_1}, l_1, \mathcal{R}_1 \rangle$. Using our strict-encapsulation definition, we can state the following:

$$\begin{aligned} \mathcal{S}_{g_2} = \mathcal{E}_s(\langle \mathcal{P}_1, \mathcal{M}_1, \Sigma_{g_1}, l_1, \mathcal{R}_1 \rangle, l_m) = \quad \langle \quad & \mathcal{P}_2 = \mathcal{P}_1, \\ & \mathcal{M}_2 = \eta(\mathcal{CMS}(\Sigma_{g_1}, l_m)) \cup \mathcal{M}_1, \\ & \Sigma_{g_2} = \eta(\mathcal{CMS}(\Sigma_{g_1}, l_m)), \\ & l_2 = l_1/l_m, \\ & \mathcal{R}_2 = \mathcal{CMS}(\Sigma_{g_1}, l_m) \cup \mathcal{R}_1 \\ \rangle \quad & \end{aligned} \tag{4.1}$$

Using our search and representation space definitions, $L(\mathcal{S}_2)$ can be written as:

$$L(\mathcal{S}_2) = \{\text{Expand}_{\mathcal{R}_2}(e) \mid e \in {\Sigma_{g_2}}^* \wedge |e| = l_2\}$$

using Σ_{g_2} and l_2 from (4.1), we obtain:

$$L(\mathcal{S}_2) = \{\text{Expand}_{\mathcal{R}_2}(e) \mid e \in \{\eta(\mathcal{CMS}(\Sigma_{g_1}, l_m))\}^* \wedge |e| = l_1/l_m\} \tag{4.2}$$

Note that, by definition, $\mathcal{CMS}(\Sigma_1, l_m)$ is the set of rules defining all possible strings over ${\Sigma_{g_1}}^*$ of size l_m. In fact, it is trivial to show that the following is true

$$\text{Expand}_{\mathcal{CMS}(\Sigma_{g_1},l_m)}[\eta(\mathcal{CMS}(\Sigma_{g_1}, l_m))] = \{w \mid w \in {\Sigma_{g_1}}^* \wedge |w| = l_m\} \tag{4.3}$$

Consider the following set

$$\{e \mid e \in \{\eta(\mathcal{CMS}(\Sigma_{g_1}, l_m))\}^* \wedge |e| = l_1/l_m\} \tag{4.4}$$

This is the set of all possible strings over alphabet $\{\eta(\mathcal{CMS}(\Sigma_{g_1}, l_m))\}$ of length l_1/l_m. Combining (4.3) and (4.4), it is easy to see that

$$\begin{aligned}\{\text{Expand}_{\mathcal{CMS}(\Sigma_{g_1},l_m)}(e) \quad &| \quad e \in \{\eta(\mathcal{CMS}(\Sigma_{g_1}, l_m))\}^* \wedge |e| = l_1/l_m\} \\ &= \quad \{w \mid w \in {\Sigma_{g_1}}^* \wedge |w| = l_1\}\end{aligned}$$

Applying $\text{Expand}_{\mathcal{R}_1}()$ to both sets, we obtain:

$$\begin{gathered}\{\text{Expand}_{\mathcal{R}_1}(\text{Expand}_{\mathcal{CMS}(\Sigma_{g_1},l_m)}(e)) \quad | \\ e \in \{\eta(\mathcal{CMS}(\Sigma_{g_1}, l_m))\}^* \wedge |e| = l_1/l_m\} \\ = \{\text{Expand}_{\mathcal{R}_1}(w) \mid w \in {\Sigma_{g_1}}^* \wedge |w| = l_1\}\end{gathered} \tag{4.5}$$

From (4.1), $\mathcal{R}_2 = \mathcal{CMS}(\Sigma_{g_1}, l_m) \cup \mathcal{R}_1$. Given that, by definition, all rules in $\mathcal{CMS}(\Sigma_{g_1}, l_m)$ are of higher order than rules in $\mathcal{R}_1$, the two rule sets can be applied consecutively without altering the result. More precisely:

$$\text{Expand}_{\mathcal{R}_2}(x) = \text{Expand}_{\mathcal{R}_1}(\text{Expand}_{\mathcal{CMS}(\Sigma_{g_1},l_m)}(x))$$

Using the above expression, we can rewrite (4.2) as follows

$$L(\mathcal{S}_2) = \{\text{Expand}_{\mathcal{R}_1}(\text{Expand}_{\mathcal{CMS}(\Sigma_{g_1}, l_m)}(e)) \mid$$

$$\mid e \in \{\eta(\mathcal{CMS}(\Sigma_{g_1}, l_m))\}^* \wedge |e| = l_1/l_m\}$$

Finally, using (4.5) in the above expression, we obtain:

$$L(\mathcal{S}_2) = \{\text{Expand}_{\mathcal{R}_1}(e') \mid e' \in {\Sigma_{g_1}}^* \wedge |e'| = l_1\} = L(\mathcal{S}_1)$$

$\dashv$

Corollary 1 *The strict-encapsulation transformation of a search space preserves search space set-reachability.*

Proof: Let A be an arbitrary set, and $\mathcal{S}_1$, $\mathcal{S}_2$ be search spaces such that $\mathcal{S}_{g_2} = \mathcal{E}_s(\mathcal{S}_{g_1}, l_m)$, we need to prove that

$$L(\mathcal{S}_1) \supseteq A \quad \rightarrow \quad L(\mathcal{S}_2) \supseteq A$$

$$L(\mathcal{S}_1) \not\supseteq A \quad \rightarrow \quad L(\mathcal{S}_2) \not\supseteq A$$

For the first implication, we have $L(\mathcal{S}_1) \supseteq A$. It suffices to observe that using Lemma 1 (above), $L(\mathcal{S}_1) = L(\mathcal{S}_2) \supseteq A$. The same observation is true for the second implication. $\dashv$

Corollary 2 *The strict-encapsulation transformation of a search space preserves any bias in the reachability of the search space.*

Proof: For $\mathcal{S}_1$, $\mathcal{S}_2$ search spaces such that $\mathcal{S}_{g2} = \mathcal{E}_s(\mathcal{S}_{g1}, l_m)$, we need to prove that

$$\begin{aligned} \textit{reachability biased } L(\mathcal{S}_1) &\rightarrow \textit{reachability biased } L(\mathcal{S}_2) \\ \textit{not reachability biased } L(\mathcal{S}_1) &\rightarrow \textit{not reachability biased } L(\mathcal{S}_2) \end{aligned}$$

By definition, reachability biased $L(\mathcal{S}_1)$ iff $L(\mathcal{S}_1) \not\supseteq \{e \mid e \in \mathcal{P}^* \wedge |e| = t\}$. It suffices to observe that this is a special case of the corollary above, where $A = \{e \mid e \in \mathcal{P}^* \wedge |e| = t\}$.$\dashv$

4.2 Search Space Redundancy

In order to establish invariance of search space size and redundancy under strict-encapsulation, we prove that the representation spaces before and after strict-encapsulation are one-to-one and onto. First, we establish invariance of search space size. We know that, by definition, the representation space and its associated search space are always of the same size. We also know that if two sets are one-to-one and onto, then they are of the same size. Therefore, if the representation spaces before and after strict encapsulation are one-to-one and onto, then their associated search spaces are also equal in size. Second, we establish invariance of search space redundancy. Remember that: $\text{Expand}_{\mathcal{R}_2}(x) = \text{Expand}_{\mathcal{R}_1}(\text{Expand}_{\mathcal{CMS}(\Sigma_{g1}, l_m)}(x))$. Thus, after strict-encapsulation, the function mapping elements of the representation space to the search space can be seen as first mapping the encapsulated elements of the resulting representation space to the original elements in the original representation space and then mapping these elements to the search space using the original mapping function. Therefore, if the representation spaces are one-to-one and onto, two elements mapping to equal phenotype strings in the original search space will also map to equal phenotype strings in the resulting search space. Similarly, two elements mapping to different phenotype strings in the original search space will also map to different phenotype strings in the resulting search space after strict-encapsulation. For example, assume $A, B \in L(\mathcal{S}_{g1})$ and $C, D \in L(\mathcal{S}_{g2})$ where genotype A maps to phenotype a ($a = \text{Expand}_{\mathcal{R}_1}(A)$), genotype B maps to phenotype b ($b = \text{Expand}_{\mathcal{R}_1}(B)$), genotype C maps to phenotype c ($c = \text{Expand}_{\mathcal{R}_2}(C)$) and genotype D maps to phenotype d ($d = \text{Expand}_{\mathcal{R}_2}(D)$). Assume also that A and C are the corresponding genotypes before and after strict-encapsulation ($A = \text{Expand}_{\mathcal{CMS}(\Sigma_{g1}, l_m)}(C)$) and B and D are the corresponding genotypes before and after strict encapsulation ($B = \text{Expand}_{\mathcal{CMS}(\Sigma_{g1}, l_m)}(D)$). If $L(\mathcal{S}_{g1})$ and $L(\mathcal{S}_{g2})$ are one-to-one and onto, then the following is true: if $a = b$ then $c = d$ and if $a \neq b$ then $c \neq d$. Thus, in order to prove the invaraince of redundancy, it is sufficient to prove that the representation spaces before and after strict-encapsulation are one-to-one and onto.

Lemma 2 *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be search spaces such that $\mathcal{S}_{g2} = \mathcal{E}_s(\mathcal{S}_{g1}, l_m)$, then there is a one-to-one correspondence (bijection) between $L(\mathcal{S}_{g1})$ and $L(\mathcal{S}_{g2})$ defined by the function $f : L(\mathcal{S}_{g2}) \mapsto L(\mathcal{S}_{g1})$,*

$$f(x) = \mathit{Expand}_{\{\mathcal{CMS}(\Sigma_{g1}, l_m)\}}(x)$$

Proof: We will prove that f is one-to-one (injective) and onto (surjective).

One-to-one:

Suppose by the way of contradiction that $x_1, x_2 \in L(\mathcal{S}_{g2})$ such that $x_1 \neq x_2$ but $f(x_1) = f(x_2)$. Let $x_1 = x_{11} \cdot x_{12} \cdot x_{13} \ldots x_{1(l_1/l_m)}$ and $x_2 = x_{21} \cdot x_{22} \cdot x_{23} \ldots x_{2(l_1/l_m)}$, where $x_{ij} \in$

$\eta(\mathcal{CMS}(\Sigma_{g1}, l_m))$. By our definition of $\mathcal{CMS}$, there is a unique module symbol in $\eta(\mathcal{CMS}(\Sigma_{g1}, l_m))$ for each string of length l_m over ${\Sigma_{g1}}^*$. Since $x_1 \neq x_2$ the two strings must differ in at least one symbol: $\exists_j (x_{1j} \neq x_{2j})$. Since all symbols in $\eta(\mathcal{CMS}(\Sigma_{g1}, l_m))$ map to unique strings when expanded, we reach a contradiction: $f(x_1) \neq f(x_2)$.

Onto:

Suppose by the way of contradiction that $x \in L(\mathcal{S}_{g2})$ and $y \in L(\mathcal{S}_{g1})$ such that $f(x) = y$ does not exists. Let $x = x_1 \cdot x_2 \cdot x_3 \ldots x_{(l_1/l_m)}$ and $y = y_1 \cdot y_2 \cdot y_3 \ldots y_{(l_1/l_m)}$, where $x_i \in \eta(\mathcal{CMS}(\Sigma_{g1}, l_m))$ and each y_i is a string of length l_m over Σ_{g1}. By definition, $\mathcal{CMS}(\Sigma_{g1}, l_m)$ contains one module defining rule for each string of length l_m over Σ_{g1}. Therefore, for every $y = y_1 \cdot y_2 \cdot y_3 \ldots y_{(l_1/l_m)}$, there is a string of module symbols $x = x_1 \cdot x_2 \cdot x_3 \ldots x_{(l_1/l_m)}$ such that $f(x_1 \cdot x_2 \cdot x_3 \ldots x_{(l_1/l_m)}) = y_1 \cdot y_2 \cdot y_3 \ldots y_{(l_1/l_m)}$ $\dashv$

Corollary 3 *The strict-encapsulation transformation of a search space preserves the search space size.*

Proof: For $\mathcal{S}_1$, $\mathcal{S}_2$ search spaces such that $\mathcal{S}_{g2} = \mathcal{E}_s(\mathcal{S}_{g1}, l_m)$, we need to prove that $|L(\mathcal{S}_1)| = |L(\mathcal{S}_2)|$. Using $|L(\mathcal{S}_g)| = |L(\mathcal{S})|$ and Lemma 2: $|L(\mathcal{S}_1)| = |L(\mathcal{S}_{g1})| = |L(\mathcal{S}_{g2})| = |L(\mathcal{S}_2)|$ $\dashv$

Corollary 4 *The strict-encapsulation transformation of a search space preserves the search space redundancy bias.*

Proof: For $\mathcal{S}_1$, $\mathcal{S}_2$ search spaces such that $\mathcal{S}_{g_2} = \mathcal{E}_s(\mathcal{S}_{g_1}, l_m)$, we need to prove that

$$\begin{aligned} \textit{redundancy biased } L(\mathcal{S}_1) &\rightarrow \textit{redundancy biased } L(\mathcal{S}_2) \\ \textit{not redundancy biased } L(\mathcal{S}_1) &\rightarrow \textit{not redundancy biased } L(\mathcal{S}_2) \end{aligned}$$

By definition, there is a bias in redundancy of $L(\mathcal{S}_1)$ iff $\exists_{e_1,e_2 \in L(\mathcal{S}_{g_1})}[(\text{Expand}_{\mathcal{R}_1}(e_1) = \text{Expand}_{\mathcal{R}_1}(e_2)) \wedge (e_1 \neq e_2)]$. It suffices to observe that according to Lemma 2 there is a one-to-one correspondence between $L(\mathcal{S}_{g_1})$ and $L(\mathcal{S}_{g_2})$, and that $\text{Expand}_{\mathcal{R}_2}(x) = \text{Expand}_{\mathcal{R}_1}(\text{Expand}_{\mathcal{CMS}(\Sigma_{g_1}, l_m)}(x))$.

$\dashv$

4.3 Search Space Invariance Under the Strict-Encapsulation Transformation

We use Lemma 1 and 2 to prove that the search space is invariant under strict-encapsulation. In order to prove the search space invariance, we need to prove that the search space size and any bias in reachability and redundancy are preserved under strict encapsulation. Lemma 1 is used to establish invariance of search space reachability under the strict-encapsulation and Lemma 2 is used to establish invariance of search space redundancy and search space size.

Theorem 1 *Transforming a search space by strictly-encapsulating lower-order modules into a complete set of higher-order modules does not change the search space size, reachability bias or redundancy bias.*

Proof: Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two search spaces such that $\mathcal{S}_{g2} = \mathcal{E}_s(\mathcal{S}_{g1}, l_m)$. We need to prove that

$$\begin{aligned}
|L(\mathcal{S}_1)| &= |L(\mathcal{S}_2)| \\
\textit{reachability biased } L(\mathcal{S}_1) &\rightarrow \textit{reachability biased } L(\mathcal{S}_2) \\
\textit{not reachability biased } L(\mathcal{S}_1) &\rightarrow \textit{not reachability biased } L(\mathcal{S}_2) \\
\textit{redundancy biased } L(\mathcal{S}_1) &\rightarrow \textit{redundancy biased } L(\mathcal{S}_2) \\
\textit{not redundancy biased } L(\mathcal{S}_1) &\rightarrow \textit{not redundancy biased } L(\mathcal{S}_2)
\end{aligned}$$

All statements above are proved in the corollaries of Lemmas 1 and 2. $\dashv$

Continuously strictly-encapsulating a genotype space produces a hierarchy of genotype spaces. At the bottom of this hierarchy, we have a genotype space with individuals of size l and trivial modules of size one. As we move up the hierarchy, we have genotype spaces with more modules and larger module sizes. At the top of this hierarchy, we have individual's genomes of size one consisting of only a single module of size l. We call this type of hierarchy a *modularity representation pyramid* for $\mathcal{S}_g$ because the representation spaces change but their search spaces remain invariant. Formally, let $\mathcal{S}_g$ be a genotype space with only primitives. Let us recursively define the modularity representation pyramid, $A = \{a_0, a_1, ...\}$, for $\mathcal{S}_g$ as:

$$\begin{aligned}
a_0 &= \mathcal{S}_g \\
a_{i+1} &= \mathcal{E}_s(a_i, l_m)
\end{aligned}$$

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), or is not longer divisible by l_m (for which strict-encapsulation is not longer defined).

Corollary 5 *Let $\mathcal{S}_g$ be a genotype space with only primitives and A be a modularity representation pyramid for $\mathcal{S}_g$. Then the following is true for all levels of the pyramid:*

1. *all search spaces are of equal size: $|\Sigma_g|^l$*
2. *all search spaces have no bias in reachability or redundancy.*

Proof: Base case: $L(\mathcal{S})$ has, trivially, no bias in reachability or redundancy, and is of size $|\Sigma_g|^l$. Recursive step: by the previous theorem, a_{i+1} and a_i are of the same size and if a_i has no bias in reachability or redundancy, so does not a_{i+1}. $\dashv$

4.4 Summary

While arbitrary module creation significantly changes the structure of the search space in ways that can favor or disfavor a given class of search strategies, we have to be careful to remember that while these changes are possible, they are not guaranteed. There are combinations of module creation transformations that can effectively neutralize each other and produce a search space not changed in any meaningful way, as witnessed by the strict-encapsulation transformations described in this Section. In this section, we establish that there is a class of module creating transformations under which search space structure as measured by size, reachability and redundancy is invariant: the strict-encapsulation transformations. Furthermore, we establish that there is a hierarchy of genotype spaces with invariant search space structure produced by this class. We see this result as a starting point for future study. It indicates that while arbitrary module creation significantly changes the structure of the search space in ways that can favor or disfavor a given class of search strategies, we have to be careful to remember that while these changes are possible, they are not guaranteed. There are combinations of module creation transformations that can effectively neutralize each other and produce a search space not changed in any meaningful way, as witnessed by the strict-encapsulation transformations described in this Section.

Simply stated, just introducing modularity at the structural level in a search space, in and on itself, could help, hinder, or cause no effect, depending of the characteristics of the modules introduced.

CHAPTER 5
EXPERIMENTAL ANALYSIS: MODULE ENCAPSULATION EFFECTS ON SEARCH SPACE REACHABILITY, REDUNDANCY AND PERFORMANCE

Our theoretical study presented in Chapter 4, does not claim anything about performance of a given algorithm in a given search space. In fact, our theorem does not make any assumptions about search strategies. It only claims that, under certain module creating transformations, the search space does not change in three structural aspects: size, reachability and redundancy. Although, we have found some evidence of correlation between search space structure and algorithmic performance, we certainly do not claim a direct correlation between search space structure and arbitrary search strategy performance. What we do claim is that, in general, the search space structure is one of many factors that contribute to the final algorithm performance and that in some cases is the overriding factor.

With this observation in mind, in our experimental analysis, we investigate the correlation between our theoretical results and the performance obtained by a genetic algorithm acting on these seach spaces. First, we empirically show that the performance of a GA acting on these search spaces remains invariant under strict-encapsulation, which qualitatively verify Theorem 1. As previously discussed, this theorem states that, under strict-encapsulation, the search space structure remains invariant. Second, we analyze whether the three search space structural aspects that we study—size, reachability and redundancy—are relevant in terms of performance. We analyze how each one of them in isolation affects GA search performance. Finally, we investigate how we can bias the search towards or away from a solution by biasing the search space structure in terms of reachability and redundancy.

5.1 Experimental Settings

This experimental study has three parts. The first part includes Experiment 1 where we give an empirical confirmation of Theorem 1. The second part includes Experiments 2, 3 and 4. In this part, our goal is to show that both redundancy and reachability are relevant in terms of performance by observing performance changes while keeping one constant and changing the other. In fact, we analyze the effects of changes in each of the three aspects of the search space structure: search space size, reachability and redundancy. The third part of this experimental analysis includes Experiment 5 and Experiment 6. In this part, we analyze how we can bias the search toward or away from a solution by changing the bias in reachability or redundancy in targeted parts of the search space.

For all experiments, we use a traditional genetic algorithm (GA) with a binary representation, but augmented with module symbols when necessary. All module creation transformations use the binary alphabet as the starting point for module creation. We use the following common parameters: population size and number of generations are 500, crossover rate, P_c, is 0.9, mutation rate, P_{m_p}, is kept constant at the phenotype level at 0.01. The selection type is tournament with size 4. We use the Generalized OneMax problem[1] in Experiments 1, 2, 3 and 4, and we use the standard OneMax in Experiments 5 and 6. In standard OneMax, the goal is to maximize the number of 1's in a solution string. The string with all 1's has the highest fitness value. In generalized OneMax, the goal is to maximize the number of matching bits between a candidate solution string and a fixed target string. The candidate string with a perfect match has the highest fitness value. We use the following module, genotype and phenotype lengths. For Experiment 1 we use $l_m = \{1, 2, 4, 8, 16\}$, $l = \{16, 8, 4, 2, 1\}$ and $l_p = 16$. For Experiments 2, 3, 4, 5, and 6, we use l_m=4, l=32, and $l_p = 128$. We perform 100 trials for all experiments and report average values with their 95% confidence intervals[2].

We analyze performance changes in each experiment. The results are presented in terms of the absolute best fitness and the number of generation to obtain the absolute best fitness. When the absolute best fitness values are statistically undistinguishable, we rely on the number of generations to obtain the absolute best fitness in order to determine performance differences. The less the number of generations to obtain the same absolute best fitness value the better the performance. When the absolute best fitness values are different, we consider that the number of generations to obtain them are not comparable.

[1]maximizing $f(x) = n - \text{HammingDistance}(x, x^*)$, where phenotypes $x, x^* \in \{0, 1\}^n$ and x^* is the fixed target phenotype. For $x^* = 111...1$, the problem reduces to the traditional OneMax

[2] We perform additional experiment sets with the following length values. For Experiment 1 we also use $l_m = \{1, 2, 4, 8\}$, $l = \{8, 4, 2, 1\}$ and $l_p = 8$. For Experiments 2, 3, 4, 5, and 6, we also use l_m=2, $l = 64$, $l_p = 128$; l_m=8, $l = 16$, $l_p = 128$; l_m=4, $l = 16$, $l_p = 64$; and l_m=4, $l = 64$, $l_p = 256$. All additional experiment sets have identical qualitatively behavior and support the conclusions drawn from the experiment sets reported.

	Alphabet size	l_m **in terms of primitives**	l	P_{m_g}
Level 1 (a_0)	2	1	16	0.01
Level 2 (a_1)	4	2	8	0.02
Level 3 (a_2)	16	4	4	0.04
Level 4 (a_3)	256	8	2	0.08
Level 5 (a_4)	65536	16	1	0.16

Table 5.1: Details for the modularity representation pyramid A_1 used in experiment 1.

5.2 Part 1: Qualitative Validation of Search Space Invariance

Experiment 1 is a qualitative empirical validation of Theorem 1. In order to do so, we use the small modularity representation pyramid, A_1, shown in Table 5.1. Because our theorem does not make any assumptions about search strategies, we expect the GA dynamics to have an impact on performance. In particular, we expect an impact because genetic operators act at the changing representation level and not at the resulting invariant search space level. Most importantly, we also expect the structure of the search space be the overriding factor on performance. As a result, we expect to see a qualitative validation of Theorem 1 in the form of roughly comparable performance at all pyramid levels due to the underlying search space structural invariance.

We define the modularity representation pyramid A_1 for $\mathcal{S}_{g1} = \langle \mathcal{P} = \{0,1\}, M = \{\}, \Sigma_g = \{0,1\}, l = 16, \mathcal{R} = \{\}\rangle$ as follows:

$$
\begin{aligned}
a_0 = \quad \mathcal{S}_{g1} \quad &= \langle \mathcal{P} = \{0,1\}, M = \{\}, \Sigma_g = \{0,1\}, l = 16, \mathcal{R} = \{\}\rangle \\
a_1 = \mathcal{E}_s(a_0, 2) = &\qquad \langle \mathcal{P} = \{0,1\}, M_{a_1}, \Sigma_{g_{a_1}}, l = 8, \mathcal{R}_{a_1}\rangle \\
a_2 = \mathcal{E}_s(a_1, 2) = &\qquad \langle \mathcal{P} = \{0,1\}, M_{a_2}, \Sigma_{g_{a_2}}, l = 4, \mathcal{R}_{a_1}\rangle \\
a_3 = \mathcal{E}_s(a_2, 2) = &\qquad \langle \mathcal{P} = \{0,1\}, M_{a_3}, \Sigma_{g_{a_3}}, l = 2, \mathcal{R}_{a_1}\rangle \\
a_4 = \mathcal{E}_s(a_3, 2) = &\qquad \langle \mathcal{P} = \{0,1\}, M_{a_4}, \Sigma_{g_{a_4}}, l = 1, \mathcal{R}_{a_1}\rangle
\end{aligned}
$$

In this pyramid, using our strict-encapsulation definition, we obtain the following for a_1:

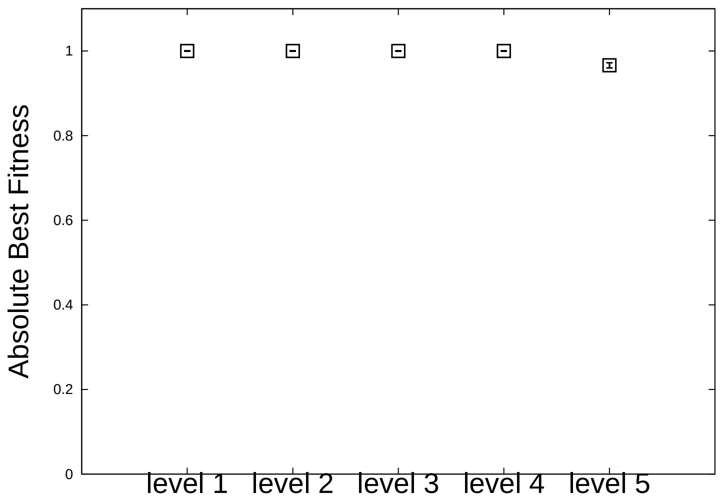
$$
\begin{aligned}
M_{a_1} &= \{M_0, M_1, M_2, M_3\} \\
\Sigma_{g_{a_1}} &= \{M_0, M_1, M_2, M_3\} \\
\mathcal{R}_{a_1} &= \{M_0 \rightarrow 00, M_1 \rightarrow 01, M_2 \rightarrow 10, M_3 \rightarrow 11\}.
\end{aligned}
$$

Similarly, we can also obtain the following for a_2:

$$
\begin{aligned}
M_{a_2} &= \{M_0, M_1, M_2, M_3, M_{0.0}, M_{0.1}, M_{0.2}, \ldots, M_{3.3},\} \\
\Sigma_{g_{a_2}} &= \{M_{0.0}, M_{0.1}, M_{0.2}, \ldots, M_{3.3},\} \\
\mathcal{R}_{a_2} &= \{M_{0.0} \rightarrow M_0M_0, M_{0.1} \rightarrow M_0M_1, M_{0.2} \rightarrow M_0M_2, \ldots, M_{3.3} \rightarrow M_3M_3\}
\end{aligned}
$$

We can obtain all the values for the remining levels in a similar way. Each level of the pyramid is described in Table 5.1. Each level is the strict-encapsulation of the previous level using new modules consisting of two previous level modules. In Table 5.1, l_m is the module size in terms of number of primitives that a module ultimately produces.

Figure 5.1 summarizes the results for this first part of our experimental analysis. Figure 5.1(A) plots the absolute best fitness versus all five levels of the modularity representation pyramid

(A)


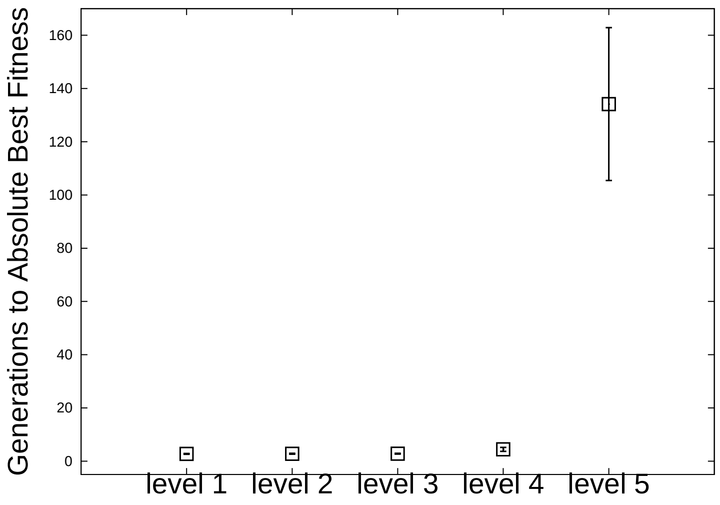
(B)


Figure 5.1: Results for Experiment 1: GA performance for the five levels of the modularity representation pyramid described in Table 5.1. X-axis shows (A) Absolute best fitness, and (B) Generations to absolute best fitness and y-axis shows the five levels of the pyramid. The first four levels perform similarly but the last level performs slightly worse than the first four levels.

in Table 5.1. The absolute best fitness value ranges between 0.0 and 1.0, 1.0 being the highest fitness. The absolute best fitness value is 1.0 for Levels 1 to 4, and it is slightly lower at 0.97 for Level 5. As expected, the absolute best fitness is the same in all levels of the pyramid except for the extreme case in the last level. In Level 5, there are 65,536 strings of size 1 in the search space. Each string consists of one module of size 16. As a result, the search in this setting becomes a random search where there is only one solution and the search operators do not help accumulate information toward the solution. For example, a string that matches with the solution string in 15 bits but differs in only one bit may become a string where all 16 bits differ from the solution string after a single mutation step. Therefore, even though at all levels, the search space has not changed in size, reachability bias or redundancy bias; the associated representation spaces, where the genetic operators act, have substantially changed. At the top level of the pyramid, the overriding factor to determine performance is not the search space, but the representation space structure that is basically constraining the GA into performing a random search. Thus, we do not expect the last level of a pyramid to agree with our theorem[3]. In Figure 5.1(B), we report performance in terms of generations to obtain the absolute best fitness. In the first four levels, the absolute best fitness is obtained in the first five generations. The difference between the number of generation that the absolute best fitness obtained is statistically insignificant between Levels 1 to 4. In Level 5, the absolute best fitness is obtained in generation number 146 which is significantly higher than the other four levels.

[3]We also perform this experiment on a four-level pyramid. We observe identical qualitatively results to the ones reported on this thesis. The first three levels perform similarly and the last level performs slightly worse than the first three levels.

5.3 Part 2: Relevance of Reachability and Redundancy

In this part, we have three experiments: Experiment 2, 3 and 4. In these experiments, we analyze how changes in search space size, reachability and redundancy, respectively, affect performance. For all experiments in this part, we define various module sets to include into or remove from the genomic alphabet in order to change search space size, reachability or redundancy. We use the generalized OneMax problem and generate a new random solution string for each run. As a result, the reachability and redundancy changes affect random regions of the search space with respect to the solution string.

In Experiment 2, we analyze the impact of changes in search space size on performance. We compare the performance of a GA running on search spaces with different sizes but equal reachability and redundancy. We change the search space size by changing the elements in the genomic alphabet. The original genomic alphabet consists of module names of a complete module set. We increase the search space size by duplicating all the elements of the alphabet. As defined in Section 3.3, a complete module set is a set containing one module for each possible genomic string of a given size. For example, for modules of size 2, the complete module set is $\mathcal{CMS}(\{0,1\},2) = \{M_0 \rightarrow 00, M_1 \rightarrow 01, M_2 \rightarrow 10, M_3 \rightarrow 11\}$. We duplicate the elements of the alphabet by adding copies of all the modules in the complete module set but with different names. For example, $\{M_0 \rightarrow 00, M_1 \rightarrow 01, M_2 \rightarrow 10, M_3 \rightarrow 11, M_4 \rightarrow 00, M_5 \rightarrow 01, M_6 \rightarrow 10, M_7 \rightarrow 11\}$. Duplication of all the alphabet elements increases the search space size without introducing any bias in redundancy because the proportion of the

Module Name	**Corresponding Substring**
M_{0_k}	0000
M_{1_k}	0001
M_{2_k}	0010
M_{3_k}	0011
M_{4_k}	0100
M_{5_k}	0101
M_{6_k}	0110
M_{7_k}	0111
M_{8_k}	1000
M_{9_k}	1001
M_{10_k}	1010
M_{11_k}	1011
M_{12_k}	1100
M_{13_k}	1101
M_{14_k}	1110
M_{15_k}	1111

Table 5.2: Elements of the complete module set of module size 4, $\mathcal{CMS}_k(\{0,1\},4)$.

elements in the search space does not change. It certainly does not change reachability either, because it does not eliminate any existing elements from or add any new elements to the search space.

We use Table 5.2 to explain the content of the alphabet for Experiments 2, 3 and 4. Table 5.2 shows the module names and their corresponding substrings. The index k in the module names gives the number of copies of that module. For instance, M_{0_1} is the first copy of the corresponding substring 0000 in an alphabet. The highest number for k gives the total number of modules that corresponds to the same substring. For example, in an alphabet where the highest value of k is 5 for M_{0_k}, there are in total five module names that correspond to substring 0000. We denote the set of module names that correspond to the same substring with $S_{M_{i_k}}$, where i is in $[0 : 2^{l_m} - 1]$. For example, for $i = 1$ and $l_m = 4$,

$$S_{M_{1_k}} = \{M_{1_1}, M_{1_2}, M_{1_3}, M_{1_4}, ..., M_{1_k}\}$$

where, k is the total number of copies of module name M_{1_k} which corresponds to substring 0001. The module names of a complete module set, $\eta(\mathcal{CMS}_k(\{0,1\}, l_m))$, can be written in terms of $S_{M_{i_k}}$ as follows:

$$\eta(\mathcal{CMS}_k(\{0,1\}, l_m)) = S_{M_{0_k}} \cup S_{M_{1_k}} \cup S_{M_{2_k}} \cup .. \cup S_{M_{2^{l_m}-1_k}}$$

We compare the performance of a GA with four different alphabet sizes. In the first case, the GA has an alphabet consisting of names of a complete module set of module length 4, $\eta(\mathcal{CMS}_k(\{0,1\}, 4))$. From hereafter, we use CMS to refer $\eta(\mathcal{CMS}_k(\{0,1\}, 4))$ for simplicity. In the second case, we doubled the elements of the alphabet of the first case. Thus, in the second case, $k = 2$ and we refer to this case as $2CMS$ in our results. For the third and forth cases, k are 4 and 8, and are referred as $4CMS$ and $8CMS$ respectively.

Figure 5.2(A) and (B) summarize the results for Experiment 2. Figure 5.2(A) shows the changes in the absolute best fitness versus four different search space sizes. The search space sizes for each case are: $(1 * 2^{l_m})^{l_p}$, $(2 * 2^{l_m})^{l_p}$, $(4 * 2^{l_m})^{l_p}$ and $(8 * 2^{l_m})^{l_p}$. The results reported in this figure are for $l_m = 4$ and $l_p = 128$. The absolute best fitness values obtained in all four cases are the same and equal to 1.0. Figure 5.2(B) shows the changes in the number of generations to obtain absolute best fitness versus the search space sizes. The number of generations to obtain the absolute best fitness is statistically the same for all four search

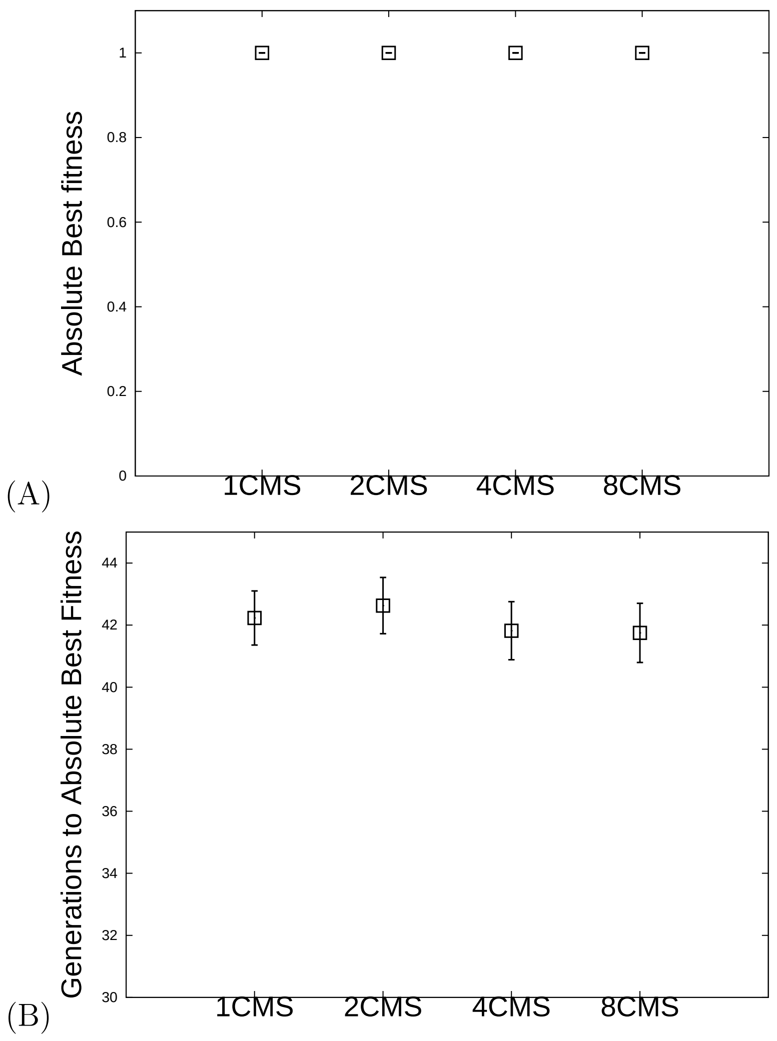


Figure 5.2: Experiment 2, x-axis in (A) shows the absolute best fitness and x-axis in (B) shows the generations needed to obtain this absolute best fitness, when increasing the search space size while reachability and redundancy are kept unchanged. Y-axis shows the search space size corresponding to $1CMS$, $2CMS$, $4CMS$ and $8CMS$. Changes in the search space size without any structural or redundancy bias does not affect the performance.

space sizes. Thus, in our experiments, changes in search space size without any change in reachability or redundancy do not affect the performance of the GA search.

In Experiment 3, we analyze the impact of reachability on performance. Similar to the previous experiment, the original alphabet consists of module names of a complete module set. In order to change reachability, we remove a percentage of the elements from the alphabet which results in a search space where some of the elements from theoriginal search space do not exist. This change decreases the search space size. It does not, however, introduce any bias in redundancy because there are only single copies of each search space element before and after removing the alphabet. In Experiment 2, we show that changes in search space size without any change in reachability and redundancy do not affect performance. Therefore, we atribute performance changes in this experiment to changes in reachability.

As previously stated, the initial alphabet contains a CMS. Next, we remove the first 25%, 50% and 75% of CMS. The content of the alphabet for each case is given in Table 5.3. We refer to the case where alphabet contains 75% of CMS after removing 25% of its elements as $0.75CMS$ in our results. Similarly, $0.5CMS$ and $0.25CMS$ refers to the cases where we remove 50% and 75% of the elements of the alphabet respectively. We compare the performance of a GA before and after a percentage of elements is removed from the alphabet to determine how reachability affects the search.

The results of Experiment 3 are reported in Figure 5.3(A) and (B). Figure 5.3(A) illustrates the changes in the absolute best fitness when we change reachability in the search space by removing 25%, 50% and 75% of the alphabet elements. When all the alphabet

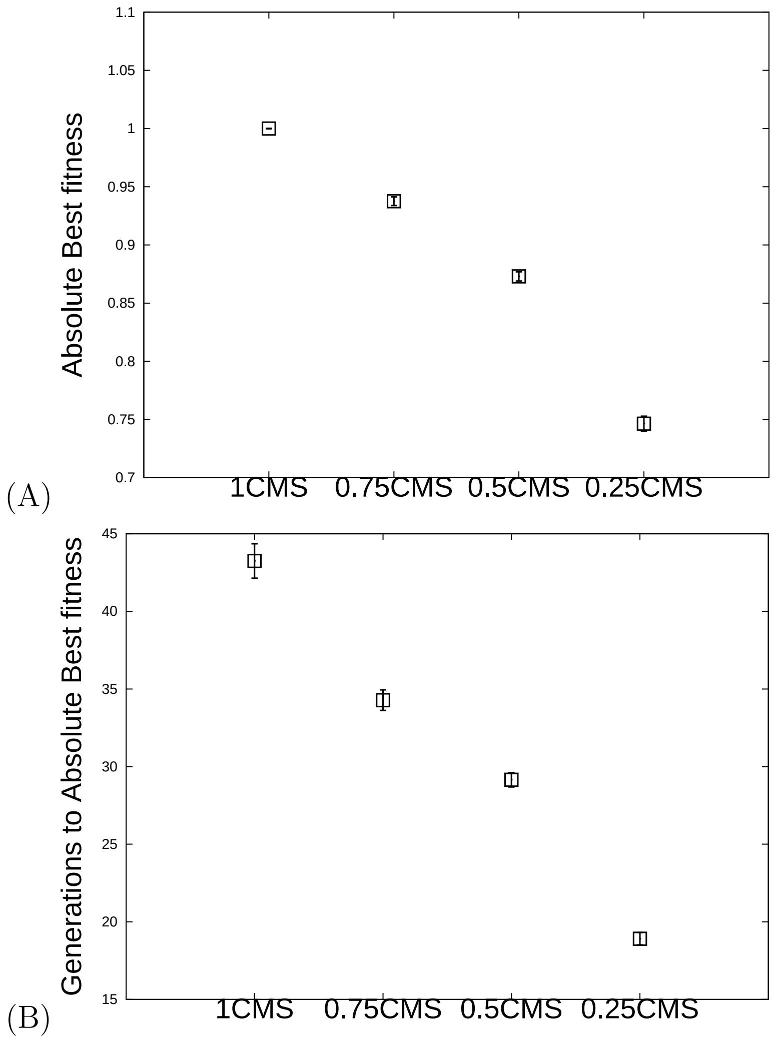


Figure 5.3: Experiments 3, x-axis in (A) shows the absolute best fitness and x-axis in (B) shows the generations needed to obtain this absolute best fitness, when removing a random set of modules. Y-axis shows the fraction of the complete module set names included in the alphabet, $1CMS$, $0.75CMS$, $0.5CMS$ and $0.25CMS$. The performance decreases as we remove larger random sets of module names from the alphabet.

1CMS	**0.75CMS**	**0.5CMS**	**0.25CMS**
M_{0_1}	M_{4_1}	M_{8_1}	M_{12_1}
M_{1_1}	M_{5_1}	M_{9_1}	M_{13_1}
M_{2_1}	M_{6_1}	M_{10_1}	M_{14_1}
M_{3_1}	M_{7_1}	M_{11_1}	M_{15_1}
M_{4_1}	M_{8_1}	M_{12_1}	
M_{5_1}	M_{9_1}	M_{13_1}	
M_{6_1}	M_{10_1}	M_{14_1}	
M_{7_1}	M_{11_1}	M_{15_1}	
M_{8_1}	M_{12_1}		
M_{9_1}	M_{13_1}		
M_{10_1}	M_{14_1}		
M_{11_1}	M_{15_1}		
M_{12_1}			
M_{13_1}			
M_{14_1}			
M_{15_1}			

Table 5.3: Elements of the alphabets in Experiment 3 and the first part of Experiment 5.

elements are present ($1CMS$), the absolute best fitness obtained is 1. The absolute best fitness values after removing 25% ($0.75CMS$), 50% ($0.5CMS$) and 75% ($0.25CMS$) of the alphabet elements are 0.94, 0.875 and 0.74, respectively. As expected, when the search space reachability is reduced by removing elements, we observe a decrease in the absolute best fitness. These results indicate that the performance changes with reachability.

In Experiment 4, we analyze the impact of redundancy on performance. Similar to the previous experiment, the original alphabet consists of module names of a complete module set. In order to change redundancy, we add redundant copies of a percentage of elements of the original alphabet. This change increases the search space size but does not change the reachability in the seach space. Reachability does not change because adding redundant copies of elements to the alphabet does not eliminate any existing element from or add any

R=0	**R=1**	**R=2**	**R=4**	**R=8**	**R=16**
CMS					
	M_{0_2}	M_{0_2}	M_{0_2}	M_{0_2} M_{0_6}	M_{0_2} M_{0_6} $M_{0_{10}}$ $M_{0_{14}}$
	M_{1_2}	M_{1_3}	M_{1_2}	M_{1_2} M_{1_6}	M_{1_2} M_{1_6} $M_{1_{10}}$ $M_{1_{14}}$
	M_{2_2}	M_{2_2}	M_{2_2}	M_{2_2} M_{2_6}	M_{2_2} M_{2_6} $M_{2_{10}}$ $M_{2_{14}}$
	M_{3_2}	M_{3_2}	M_{3_2}	M_{3_2} M_{3_6}	M_{3_2} M_{3_6} $M_{3_{10}}$ $M_{3_{14}}$
		M_{0_3}	M_{0_3}	M_{0_3} M_{0_7}	M_{0_3} M_{0_7} $M_{0_{11}}$ $M_{0_{15}}$
		M_{1_3}	M_{1_3}	M_{1_3} M_{1_7}	M_{1_3} M_{1_7} $M_{1_{11}}$ $M_{1_{15}}$
		M_{2_3}	M_{2_3}	M_{2_3} M_{2_7}	M_{2_3} M_{2_7} $M_{2_{11}}$ $M_{2_{15}}$
		M_{3_3}	M_{3_3}	M_{3_3} M_{3_7}	M_{3_3} M_{3_7} $M_{3_{11}}$ $M_{3_{15}}$
			M_{0_4}	M_{0_4} M_{0_8}	M_{0_4} M_{0_8} $M_{0_{12}}$ $M_{0_{16}}$
			M_{1_4}	M_{1_4} M_{1_8}	M_{1_4} M_{1_8} $M_{1_{12}}$ $M_{1_{16}}$
			M_{2_4}	M_{2_4} M_{2_8}	M_{2_4} M_{2_8} $M_{2_{12}}$ $M_{2_{16}}$
			M_{3_4}	M_{3_4} M_{3_8}	M_{3_4} M_{3_8} $M_{3_{12}}$ $M_{3_{16}}$
			M_{0_5}	M_{0_5} M_{0_9}	M_{0_5} M_{0_9} $M_{0_{13}}$ $M_{0_{17}}$
			M_{1_5}	M_{1_5} M_{1_9}	M_{1_5} M_{1_9} $M_{1_{13}}$ $M_{1_{17}}$
			M_{2_5}	M_{2_5} M_{2_9}	M_{2_5} M_{2_9} $M_{2_{13}}$ $M_{2_{17}}$
			M_{3_5}	M_{3_5} M_{3_9}	M_{3_5} M_{3_9} $M_{3_{13}}$ $M_{3_{17}}$

Table 5.4: Elements of the alphabet for Experiment 4.

new element into the search space. As a result, we attribute performance changes in this experiment to changes in redundancy. More concretely, we introduce redundancy to the search space by replicating a subset of the alphabet elements 1, 2, 4, 8 or 16 times. We call this subset the *redundant subset*. In our results, R gives the number of copies of the redundant subset in the alphabet. When $R = 0$, the alphabet contains only CMS. For $R > 0$, The alphabet contains CMS and R copies of the redundant subset. We choose $0.25CMS$ to be the redundant subset. Table 5.4 shows the content of the alphabets for $R = \{0, 1, 2, 4, 8, 16\}$ for Experiment 4. We compare the performance of the GA before and after adding these redundant subsets of elements to the alphabet to determine how redundancy affects the performance. Notice that because the solution string is randomly

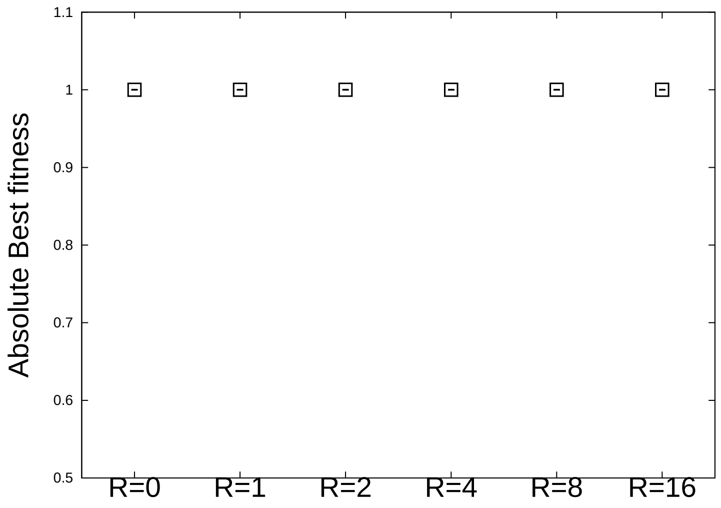
R=0	**R=1**	**R=2**	**R=4**	**R=8**	**R=16**
CMS					
	M_{x_2}	M_{x_2}	M_{x_2}	M_{x_2}	M_{x_2}
		M_{x_3}	M_{x_3}	M_{x_3}	M_{x_3}
			M_{x_4}	M_{x_4}	M_{x_4}
			M_{x_5}	M_{x_5}	M_{x_5}
				M_{x_6}	M_{x_6}
				M_{x_7}	M_{x_7}
				M_{x_8}	M_{x_8}
				M_{x_9}	M_{x_9}
					$M_{x_{10}}$
					$M_{x_{11}}$
					$M_{x_{12}}$
					$M_{x_{13}}$
					$M_{x_{14}}$
					$M_{x_{15}}$
					$M_{x_{16}}$
					$M_{x_{17}}$

Table 5.5: Elements of the alphabet for Experiment 6. $x = 15$ for first part and $x = 0$ for the second part of Experiment 6.

generated, the redundant copies of elements target the random regions of the search space with respect to the solution.

Figure 5.4(A) and (B) summarize the results for Experiment 4. Figure 5.4(A) shows the changes in the absolute best fitness versus the number of copies of $0.25CMS$ for $R = \{0, 1, 2, 4, 8, 16\}$. The absolute best fitness for all six cases are the same and equal to 1.0. The results for the number of generations to obtain the absolute best fitness for all six cases are given in Figure 5.4(B). When we increase the redundancy, the number of generations to obtain the absolute best fitness increases which indicates a performance decrease. These results indicate that the performance changes with redundancy.

(A)



(B)

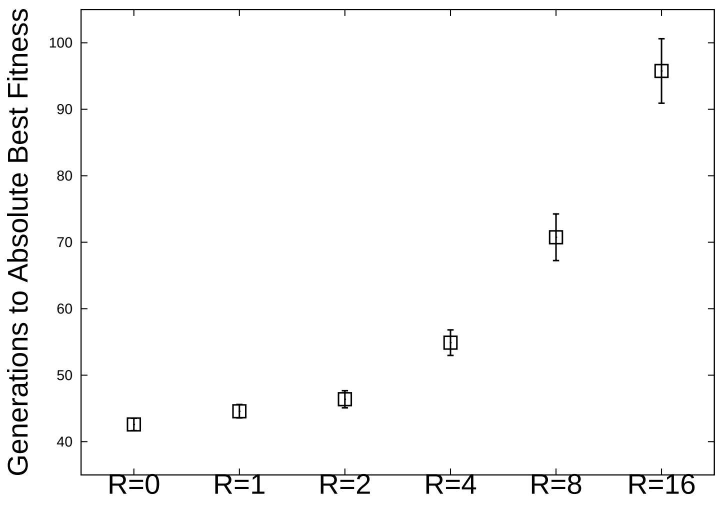


Figure 5.4: Experiment 4, x-axis in (A) shows the absolute best fitness and x-axis in (B) shows the generations needed to obtain this absolute best fitness, when increasing the number of copies of a random subset of the alphabet. Y-axis shows the number of copies, R, of the random subset with $R = \{0, 1, 2, 4, 8, 16\}$. The performance decreases with the increase in redundancy.

In this part, our experiments indicate that reachability and redundancy are relevant metrics with respect to performance, while the search size is not.

5.4 Part 3: Biasing Search Space by Changing Reachability and Redundancy

In this part, we have two experiments: Experiment 5 and 6. In these experiments, we analyze how biasing a search space by changing reachability and redundancy in targeted regions affects performance. We target reachability and redundancy changes to particular regions of a search space by identifying promising and non-promising modules and including them into or removing them from the genomic alphabet. We determine if a module is promising or not simply by comparing its similarity with the target solution string. For clarity, let us select a target solution string off all 1s which is the solution string in OneMax. In this case, a promising module is composed of mostly 1s and a non-promising module is composed of mostly 0s. For all experiments in this part, we use standard OneMax and these definitions of promising and non-promising.

In Experiment 5, we bias reachability on targeted regions of the search space with respect to the solution. Initially, the alphabet consists of a CMS. We analyze two cases: 5(i) and 5(ii). In Experiment 5(i), we remove the first 25%, 50% and 75% of the CMS as ordered in Table 5.2. In Table 5.2 CMS is in ascending order by interpreting module defining substrings as binary numbers. Because of this ordering, the module names removed in this case correspond to the substrings containing less 1s as compared to the rest of the module

1CMS	**0.75CMS**	**0.5CMS**	**0.25CMS**
M_{0_1}	M_{0_1}	M_{0_1}	M_{0_1}
M_{1_1}	M_{1_1}	M_{1_1}	M_{1_1}
M_{2_1}	M_{2_1}	M_{2_1}	M_{2_1}
M_{3_1}	M_{3_1}	M_{3_1}	M_{3_1}
M_{4_1}	M_{4_1}	M_{4_1}	
M_{5_1}	M_{5_1}	M_{5_1}	
M_{6_1}	M_{6_1}	M_{6_1}	
M_{7_1}	M_{7_1}	M_{7_1}	
M_{8_1}	M_{8_1}		
M_{9_1}	M_{9_1}		
M_{10_1}	M_{10_1}		
M_{11_1}	M_{11_1}		
M_{12_1}			
M_{13_1}			
M_{14_1}			
M_{15_1}			

Table 5.6: Elements of the alphabet for the second part of Experiment 5.

names in the alphabet. Alphabet elements are the basic building blocks of the elements in the search space and they are now biased to include less 1s. Therefore, in Experiment 5(i), we effectively remove elements of the search space that are less similar to the solution string. The content of the alphabet in this case is the same as in Experiment 3 and is given in Table 5.3. In Experiment 5(ii), we remove the last 25%, 50% and 75% of the CMS as ordered in Table 5.2. The module names removed in this case correspond to the substrings containing more 1s as compared to the rest of the module names in the alphabet. Therefore, in Experiment 5(ii), we effectively remove elements of the search space that are more similar to the solution string. The content of the alphabet for this case is given in Table 5.6.

Figures 5.5(A) and (B) summarize the results of Experiment 5(i) and Figures 5.5(C) and (D) summarize the results of Experiment 5(ii). Figure 5.5(A) shows the absolute best fitness

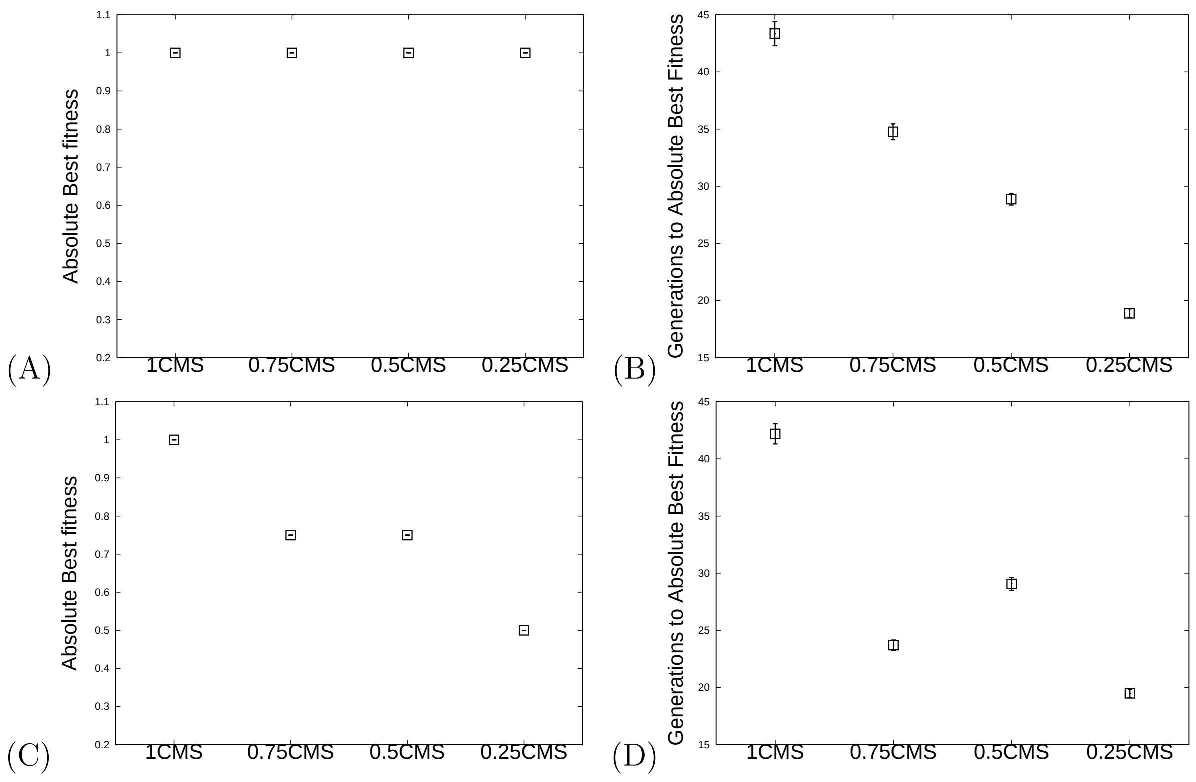


Figure 5.5: Results for Experiment 5: Reachability in targeted parts of a search space. X-axis in (A) and (C) shows the absolute best fitness and in (B) and (D) shows the generations needed to obtain this absolute best fitness. Y-axis shows the fraction of the complete module set names included in the alphabet, $1CMS$, $0.75CMS$, $0.5CMS$ and $0.25CMS$. Experiment 5(i), (A) fitness and (B) generations when removing non-promising modules (not similar to solution). The performance increases as we remove larger non-promising sets of module names from the alphabet. Experiment 5(ii), (C) fitness and (D) generations when removing promising modules (similar to solution). The performance decreases as we remove larger promising sets of module names from the alphabet.

versus the four cases of bias in reachability where we remove the first 0% ($1CMS$), 25% ($0.75CMS$), 50% ($0.5CMS$) and 75% ($0.25CMS$) of the alphabet elements. The absolute best fitness value for all four cases is 1.0. The results for the number of generations to obtain the absolute best fitness versus the same four cases are given in Figure 5.5(B). The number of generations to obtain the absolute best fitness for cases $1CMS$, $0.75CMS$, $0.5CMS$ and $0.25CMS$ are 43.4, 34.8, 29 and 19, respectively. Therefore, the performance increases as we increase the bias in reachability by removing increasingly more search space elements that are less similar to the solution. Figure 5.5(C) shows the absolute best fitness versus the four cases of bias in reachability where, this time, we remove the last 0% ($1CMS$), 25% ($0.75CMS$), 50% ($0.5CMS$) and 75% ($0.25CMS$) of the alphabet elements. The absolute best fitness values for cases $1CMS$, $0.75CMS$, $0.5CMS$ and $0.25CMS$ are 1, 0.75, 0.75 and 0.5 respectively. Therefore, the performance decreases as we increase the bias in reachability by removing search space elements that are similar to the solution.

In Experiment 6, we increase the redundancy of elements located on targeted parts of the search space by adding redundant copies of promising or non-promising elements into the alphabet. The initial alphabet consists of CMS. We analyze two cases: 6(i) and 6(ii). In Experiment 6(i), we choose the last element of CMS, $M_{15_k} = 1111$, to be the redundant subset. Note that this elements is fully included in the solution string for OneMax. Increasing the redundancy of alphabet elements that are included in the solution increases the number of search space elements that are the same as or similar to the solution string. In Experiment 6(ii), we choose the first element of CMS, $M_{0_k} = 0000$, to be the redundant subset. Note

R=0	**R=1**	**R=2**	**R=4**	**R=8**	**R=16**
CMS					
	M_{x_2}	M_{x_2}	M_{x_2}	M_{x_2}	M_{x_2}
		M_{x_3}	M_{x_3}	M_{x_3}	M_{x_3}
			M_{x_4}	M_{x_4}	M_{x_4}
			M_{x_5}	M_{x_5}	M_{x_5}
				M_{x_6}	M_{x_6}
				M_{x_7}	M_{x_7}
				M_{x_8}	M_{x_8}
				M_{x_9}	M_{x_9}
					$M_{x_{10}}$
					$M_{x_{11}}$
					$M_{x_{12}}$
					$M_{x_{13}}$
					$M_{x_{14}}$
					$M_{x_{15}}$
					$M_{x_{16}}$
					$M_{x_{17}}$

Table 5.7: Elements of the alphabet for Experiment 6. $x = 15$ for first part and $x = 0$ for the second part of Experiment 6.

that this elements is fully excluded from the solution string for OneMax. Increasing the redundancy of alphabet elements that are fully excluded from the solution string increases the number of search space elements that are less similar to the solution. In both cases, the alphabet contains CMS and R copies of the redundant subset. Table 5.7 show the content of the alphabets for $R = \{0, 1, 2, 4, 8, 16\}$ for Experiment 6.

Figure 5.6(A) and (B) summarize the results for Experiment 6(i) and Figure 5.6(C) and (D) summarize the results for Experiment 6(ii). Figure 5.6(A) shows the absolute best fitness versus the six cases of bias in redundancy. The absolute best fitness value for all six cases is 1.0. The results for the number of generations to obtain the absolute best fitness versus the six cases of bias in redundancy are given in Figure 5.6(B). The number of generations

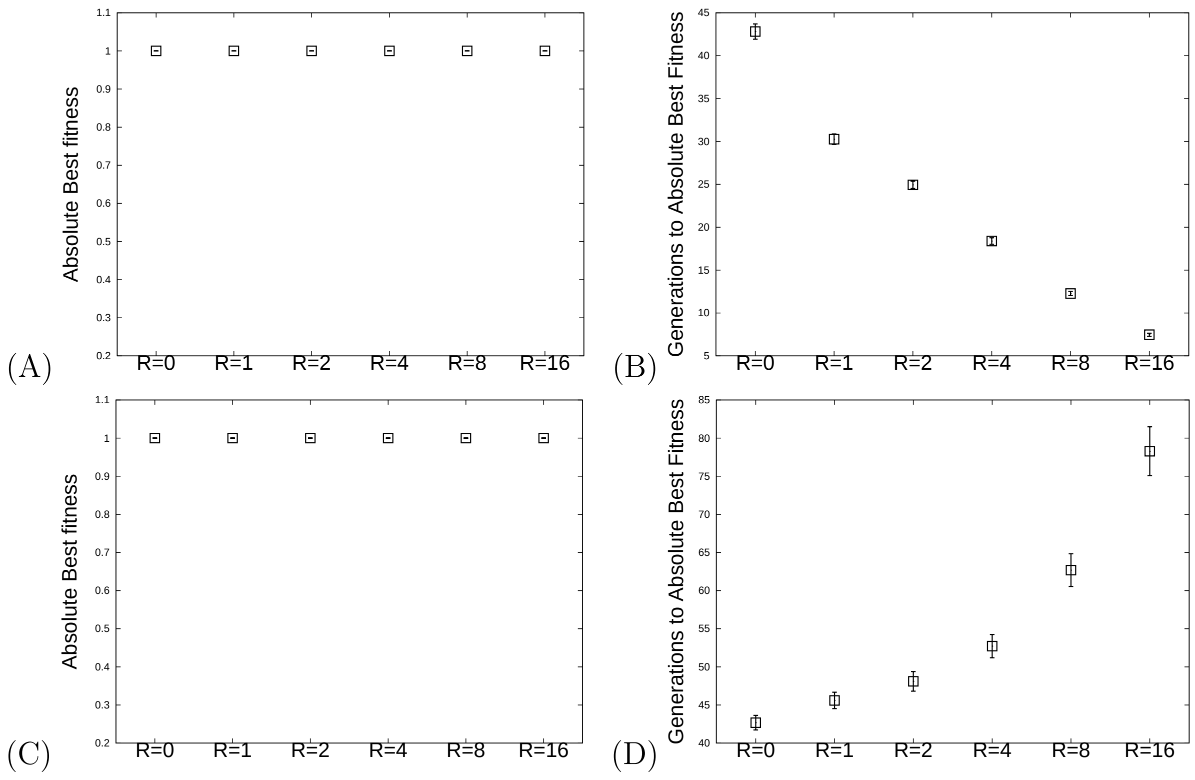


Figure 5.6: Results for Experiment 6: Redundancy in targeted parts of a search space. X-axis in (A) and (C) shows the absolute best fitness and in (B) and (D) shows the generations needed to obtain this absolute best fitness. Experiment 6(i), (A) fitness and (B) generations when increasing the number of copies of a promising module (similar to solution). Y-axis shows the number of copies, R, of a promising module name with $R = \{0, 1, 2, 4, 8, 16\}$. The performance increases as we increase the redundancy of a promising module name in the alphabet. Experiment 6(ii), (C) fitness and (D) generations when increasing the number of copies of a non-promising module (not similar to solution). Y-axis shows the number of copies, R, of a non-promising module name with $R = \{0, 1, 2, 4, 8, 16\}$. The performance decreases as we increase the redundancy of a non-promising module name in the alphabet.

to obtain the absolute best fitness for cases $R = 0$, $R = 1$, $R = 2$, $R = 4$, $R = 8$ and $R = 16$ are 43, 30, 25, 18, 12 and 8, respectively. Hence, as we increase the redundancy of elements similar to the solution, the performance increases. In Experiment 6(ii), we increase the redundancy of the elements that are less similar to the solution. Figure 5.6(C) shows the absolute best fitness versus the six cases of bias in redundancy. The absolute best fitness is 1.0 for all six cases. The number of generations to obtain the absolute best fitness for cases $R = 0$, $R = 1$, $R = 2$, $R = 4$, $R = 8$ and $R = 16$ are given in Figure 5.6(D) and are 43, 45.6, 48, 53, 63 and 78, respectively. Therefore, increasing the redundancy of the elements that are less similar to the solution decreases the performance.

In this part, our experimental results indicate that reducing the reachability in the non-promising regions of a search space affects the performance in a favorable way while reducing the reachability of the promising regions affects the performance in a disfavorable. Our experimental results also indicate that increasing redundancy of promising regions of a search space affects the performance in a favorable way while increasing the redundancy of non-promising regions affects the performance in a disfavorable way.

5.5 Summary

In our experimental analysis, we first analyze the effects of the strict-encapsulation transformation on the performance of a GA. Next, we analyze the effects of various module encapsulation transformations on the three search space features that we consider when analyzing the invariability of search spaces under strict-encapsulation: search space size, reachability and redundancy. We analyze the effects of the bias in reachability and redundancy on the performance of the search by changing only the reachability or redundancy, first in random parts and then in targeted regions of the search space. The results we obtain in our experimental analysis indicate that:

- The strict-encapsulation transformation, where search space size, reachability and redundancy are kept unchanged, does not affect the performance of the search.
- Changing search space size without any reachability or redundancy changes does not affect the performance of the search.
- Changing the bias in reachability or redundancy affects the performance of the search. We can affect the performance in a favorable way if we change the reachability or redundancy in targeted regions of the search space.
- Starting from a fully reachable search space, decreasing reachability of random sectors while keeping redundancy constant, causes a drop in performance. Decreasing reachability of promising sectors produce a larger drop in performance. Decreasing reachability of non-promising sectors, produce a performance increase.
- Starting from a non-redundant search space, increasing redundancy randomly while keeping reachability constant causes a drop in performance. Increasing redundancy of promising sectors produces an increase in performance. Increasing redundancy of

non-promising sectors, produces a decrease in performance, but not as large as the performance drop due to increase redundancy in random sectors.

Part III

PARTICULAR EFFECTS OF MODULARITY IN MUTATION BASED SEARCH

CHAPTER 6
FRAMEWORK REVISITED

In the previous part of this thesis, we have focus search space structure analysis at a general level, without focusing on a particular algorithm or problem class. In the next chapters, we focus on a more particular level of analysis for mutation-based algorithms and the OneMax problem. The required additional assumptions and definitions follow.

6.1 Assumptions

For the reminder of this work, our analysis makes the following assumptions.

1. We focus on linear binary representations but our theoretical results can easily be extended to any arity.

2. The distance between two solutions is measured as the Hamming distance or the number of unequal characters between the two solutions. Neighbors are solutions which are a Hamming distance of one from each other.

3. There is a positive correlation between the form and quality of candidate solutions.

4. We use the OneMax problem which maximizes the number of 1's in a solution string. We expect, however, this study to apply to problems whose solutions consist of repeated patterns that are location independent.

Because we calculate distance as the number of differing characters between two solutions, this work applies primarily to mutation based search algorithms which are particularly sensitive to the single step connectivity of a search space. We use the term *solution* to refer to the target solution.

6.2 Definitions

When using a set of rewriting rules to map from an initial genotype string to a resulting phenotype string, we have to choose among three alternatives with respect to modeling string lengths. First, both strings could be of variable length. This case will introduce unnecessary complications to the mathematical analysis. Second, we can keep the length of the genotype strings constant and let the resulting phenotype strings, after the mapping, be variable. This is the case we presented in Chapter 3. We call this case the *standard model* because a traditional genetic algorithms use fixed length genotypes. Third, we can keep the length of the resulting phenotype strings constant and let the genotype strings be variable. In contrast with the previous case, we call this case the *non standard model* (NSM). Because the standard and non standard models are, in essence, equivalent and because the non standard model facilitates the mathematical analysis, for this Part we use the non standard model.

Our framework can be easily extended to account for the NSM by introducing two alternative definitions.

Formally, a *genotype space* $\mathcal{S}_g$, in the non standard model version, is a 5-tuple:

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

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

The set of elements of the *genotype space* $\mathcal{S}_g$, denoted by $L(\mathcal{S}_g)$, in the non standard model version is:

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

We keep the phenotype length fixed to limit the size of the search space.

6.3 Module Encapsulation Instances

In the following chapters, we consider three instances of module encapsulation.

- Encapsulation of a module that is fully included into a solution. We call such module a "*good*" module, because it is part of the solution.
- Encapsulation of a module that is fully excluded from a solution. We call this type a "*bad*" module, because it is not in any part of the solution.

- Strict-encapsulation without replacement of a module set. As previously described, this encapsulation creates all modules of a complete module set of size l_m. This set consists of all possible modules of length l_m that can be created using elements of Σ. In this case, there are no distinctions of good or bad modules, simply all modules of a given length are created.

In the rest of this work, we denote the search space before module encapsulation with $\mathcal{S}_O$ and we denote the search space after any number of arbitrary module encapsulations with $\mathcal{S}_M$. If a good module is encapsulated, we denote the search space after module encapsulation with $\mathcal{S}_{GM}$. If a bad module is encapsulated, we denote the search space after module encapsulation with $\mathcal{S}_{BM}$. Finally, if a complete module set is encapsulated, we denote the search space after module encapsulation with $\mathcal{S}_{CM}$.

CHAPTER 7
ANALYSIS OF SEARCH SPACE SIZE

Using the above framework, we can now mathematically express the size of a search space and how it changes with the encapsulation and addition of one or more module to the alphabet. First, we derive an equation to calculate the size of a search space with modules. Then, we show that the search space size increases with module encapsulation by comparing the size of a search space before module encapsulation with that after module encapsulation.

We denote the size of a search space $\mathcal{S}$ by $|L(\mathcal{S})|$. Let l_g be the genotype length, l_m be the module length, and l be the phenotype length. Also, let n be the number of modules in a string. The number of different ways to place n modules in a string of length l_g, C_n, can be written as:

$$C_n = \binom{l_g}{n}$$

where, $l_g = l - (l_m - 1)n$. Thus, we can rewrite C_n as follows:

$$C_n = \binom{l - (l_m - 1)n}{n} \tag{7.1}$$

Let σ_m be the number of modules in the alphabet and N be the number of strings that have n modules. We can write N as:

$$N = 2^{l_g - n} \sigma_m^n C_n$$

where, σ_m^n gives the number of different combinations of all σ_m different modules located in n places in the string length of l, and 2^{l_g-n} gives the number of different combinations of two primitives located in the rest of the string which is $l_g - n$. Remember that we consider only the binary primitives. By using Equation 7.1, N can be written as:

$$N = 2^{l-l_m n}\sigma_m^n \binom{l-(l_m-1)n}{n} \tag{7.2}$$

If we sum N for all n, we obtain the search space size.

$$|L(\mathcal{S})| = \sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} N$$

Using Equation 7.2, we can rewrite the search space size as:

$$|L(\mathcal{S})| = \sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} 2^{l-l_m n}\sigma_m^n \binom{l-(l_m-1)n}{n} \tag{7.3}$$

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. Note that the formula applies only to search spaces where all modules have the same length of l_m.

Next, we investigate the effects of encapsulating σ_m number of modules of length l_m on the search space size by comparing the search space before and after the encapsulation. Assume that the genotype space alphabet before the encapsulation, Σ_O, includes solely binary primitives and the alphabet after the encapsulation, Σ_M, includes the binary primitives as well as σ_m number of modules of length l_m.

Lemma 3 *Let $\mathcal{S}_O$ and $\mathcal{S}_M$ be the search spaces before and after encapsulation, respectively, and let $\mathcal{S}_{g_O}$ and $\mathcal{S}_{g_M}$ be their corresponding genotype spaces such that $\mathcal{S}_{g_M} = \mathcal{E}(\mathcal{S}_{g_O}, M \to w)$. For $l > l_m$, the following statement is true:*

$$|L(\mathcal{S}_M)| > |L(\mathcal{S}_O)| \text{ for } l > l_m$$

Proof: Let $\Sigma_O = \{0, 1\}$ be the alphabet of the genotype space $\mathcal{S}_{g_O}$ and let $\Sigma_M = \{\mathcal{P} \cup \mathcal{M}\}$ be the alphabet of the genotype space $\mathcal{S}_{g_M}$. Let $M_i \in \mathcal{M}$ and $M_i = \mathcal{P}^*$ and $|M_i| = l_m$. The search space size for $\mathcal{S}_O$ is [Mit98]:

$$|L(\mathcal{S}_O)| = 2^l$$

We can define the search space size for $\mathcal{S}_M$ by using Equation 7.3:

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

where σ_m is the number of modules and l_m is the module length. We can rewrite $|L(\mathcal{S}_M)|$ as follows:

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

If we evaluate the summation for $n = 0$, we obtain the following:

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

Since the expression $2^l \sum_{n=1}^{\lfloor \frac{l}{l_m} \rfloor} (\frac{1}{2})^{l_m n} \binom{l-(l_m-1)n}{n}$ is positive, it is obvious that:

$$|L(\mathcal{S}_M)| > 2^l$$

Hence, we can conclude that:

$$|L(\mathcal{S}_M)| > |L(\mathcal{S}_O)| \ for\ l > l_m$$

QED. $\dashv$

As expected, module encapsulation increases the search space size. These results hold true for encapsulating any type and number of modules including good modules, bad modules and a complete module set. Also, because a solution length of l can not include a module length of l_m that is larger than the length of a solution, this result applies when $l > l_m$.

Module encapsulation clearly and obviously increases the size of a search space. What is less clear is how module encapsulation changes the structure of a search space. We next examine two aspects of search space structure: composition and connectivity.

CHAPTER 8
EFFECTS OF MODULE ENCAPSULATION ON SEARCH SPACE COMPOSITON

Composition examines the density of solutions in a search space. We examine how module encapsulation changes the ratio of the number of solutions to the total number of elements in a search space. This ratio can be said to be one measure of how difficult a problem is. The higher the ratio, the more likely that a random sampling of the search space includes a solution. This observation applies particularly to algorithms which initialize or use a sampling of the search space in any part of the search process. For example, a GA is a population based algorithm and the sampling of the initial population is important for the performance of the algorithm.

We define this ratio to be the solution density.

Definition 1 (Solution density) *Let $\mathcal{S}$ be a search space and n_s be the number of solution strings in the search space $\mathcal{S}$. The solution density in search space $\mathcal{S}$ is defined as:*

$$\rho_s = \frac{n_s}{|L(\mathcal{S})|} \tag{8.1}$$

where $|L(\mathcal{S})|$ is the size of the search space $\mathcal{S}$.

In order to analyze how encapsulating a module changes the solution density in a search space, we compare solution density before module encapsulation with solution density after module encapsulation. Let us first derive the solution density for the search space before module encapsulation. For our analysis, we assume that the optimal solution is known.

Let $\mathcal{S}_{g_O} = \langle \mathcal{P}_O, \mathcal{M}_O, \Sigma_O, l, \mathcal{R}_O \rangle$ be the genotype space before module encapsulation. The alphabet associated with the original search space, $\mathcal{S}_O$, contains only the binary primitives, $\Sigma_O = \{0, 1\}$, therefore, $|\Sigma_O| = 2$. We define the set of elements of the original search space, $L(\mathcal{S}_O)$, to be a set of strings with a single solution. Thus, the number of solutions in the original search space is one. Hence, the solution density of the original search space, ρ_{s_O}, can be written as:

$$\rho_{s_O} = \frac{1}{|L(\mathcal{S}_O)|}$$

where $|L(\mathcal{S}_O)| = |\Sigma_O|^l$ and $|\Sigma_O| = 2$. Thus,

$$\rho_{s_O} = \frac{1}{2^l} \tag{8.2}$$

We next analyze how module encapsulation affects solution density in a search space by comparing Equation 8.2 with corresponding measurements of search spaces in which there is module encapsulation. Specifically, we examine three cases: encapsulating a complete module set, encapsulating a good module and encapsulating a bad module.

8.1 Complete Module Set Encapsulation

In many cases, one may not know which modules are useful and which ones are not. In such cases, we might want to allow all modules of a given length and allow the search algorithm to dynamically decide which ones to use and which ones to ignore. A complete module set is a set containing one module for each possible genomic string of a given length. In order to analyze the effects of a complete module set, we first derive a formula to calculate the solution density after encapsulating a complete module set and then, we compare the solution density before and after a complete module encapsulation. Let $\mathcal{S}_O$ be the search space before encapsulation where $\Sigma_O = \{0, 1\}$ and let $\mathcal{S}_{CM}$ be the search space after encapsulation of a complete module set and $\Sigma_{CM} = \{0, 1, M_1, M_2, ..., M_k\}$ where $k = |\Sigma_O|^{l_m}$ and $l_m = |M_i|$ for $1 \leq i \leq k$. Before the analysis, let us illustrate what happens in a search space when we encapsulate a complete module set in an example. Assume $l = 3$, $l_m = 2$ and the solution string is 111. Given $l_m = 2$, the modules in the complete module set are: $M_1 = 00$, $M_2 = 01$, $M_3 = 10$, $M_4 = 11$. Figure 8.1(A) shows the corresponding sets of elements of the genotype space before and after complete module set encapsulation, $L(\mathcal{S}_{g_O})$ and $L(\mathcal{S}_{g_{CM}})$, respectively. In $L(\mathcal{S}_{g_O})$, the number of solutions is 1 and the search space size is 8; therefore, the solution density in $\mathcal{S}_O$ is $\frac{1}{8}$. In $L(\mathcal{S}_{g_{CM}})$, the number of solutions is 3 and the search space size is 24. Thus, the solution density of $\mathcal{S}_{CM}$ is $\frac{1}{8}$. In this example, the solution density of $\mathcal{S}_{CM}$ is equal to the solution density of $\mathcal{S}_O$.

Now, let us derive a formula to calculate the solution density after encapsulating a complete module encapsulation. By Equation 8.1, the solution density of $\mathcal{S}_{CM}$ is:

$$\rho_s = \frac{n_s}{|L(\mathcal{S}_{CM})|}$$

n_s, the number of solution strings in the search space $\mathcal{S}_{CM}$, can be calculated by calculating the number of ways to place n modules of length l_m in a string of length l, where $0 \leq n \leq \lfloor \frac{l}{l_m} \rfloor$. $\sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \binom{l-(l_m-1)n}{n}$ gives the number of ways to place n modules of length l_m in a string of length l. Note that $\lfloor \frac{l}{l_m} \rfloor$ gives the maximum number of modules that can be in a string to

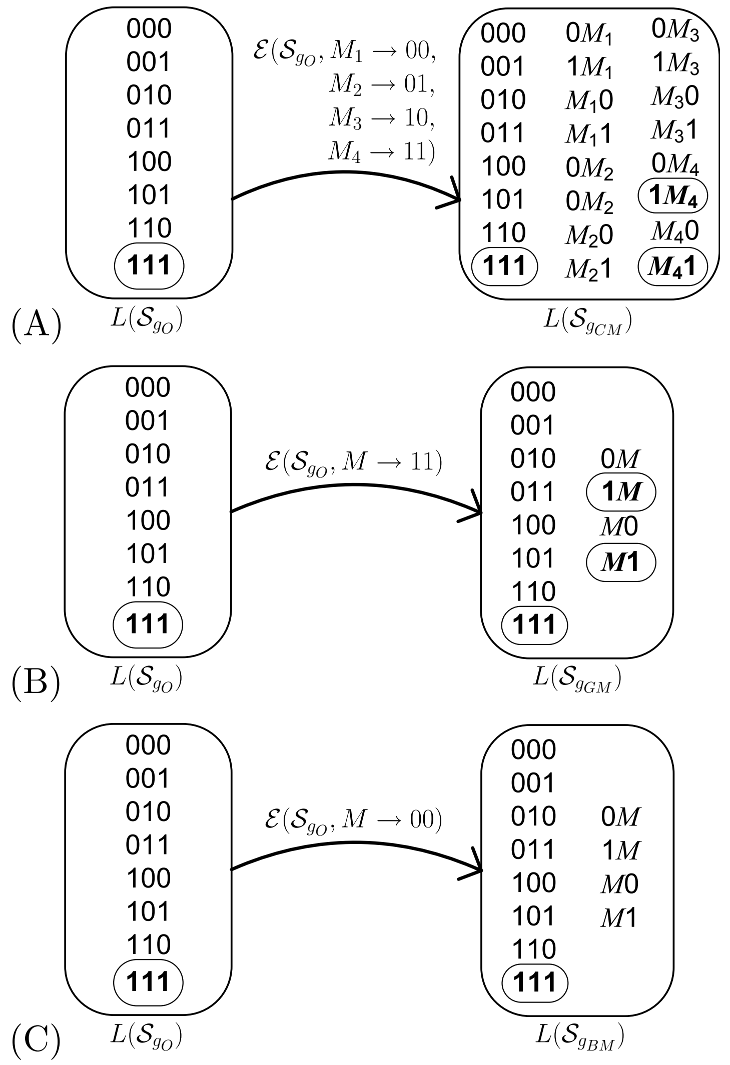

(A)
000 001 010 011 100 101 110 111
$L(\mathcal{S}_{g_O})$
$\mathcal{E}(\mathcal{S}_{g_O}, M_1 \rightarrow 00, M_2 \rightarrow 01, M_3 \rightarrow 10, M_4 \rightarrow 11)$
000 001 010 011 100 101 110 111
$0M_1$ $1M_1$ M_10 M_11 $0M_2$ $0M_2$ M_20 M_21
$0M_3$ $1M_3$ M_30 M_31 $0M_4$ $1M_4$ M_40 M_41
$L(\mathcal{S}_{g_{CM}})$
(B)
000 001 010 011 100 101 110 111
$L(\mathcal{S}_{g_O})$
$\mathcal{E}(\mathcal{S}_{g_O}, M \rightarrow 11)$
000 001 010 011 100 101 110 111
$0M$ $1M$ $M0$ $M1$
$L(\mathcal{S}_{g_{GM}})$
(C)
000 001 010 011 100 101 110 111
$L(\mathcal{S}_{g_O})$
$\mathcal{E}(\mathcal{S}_{g_O}, M \rightarrow 00)$
000 001 010 011 100 101 110 111
$0M$ $1M$ $M0$ $M1$
$L(\mathcal{S}_{g_{BM}})$

Figure 8.1: Elements of the genotype space before ($L(\mathcal{S}_{g_O})$), and after ($L(\mathcal{S}_{g_{CM}})$, $L(\mathcal{S}_{g_{GM}})$, $L(\mathcal{S}_{g_{BM}})$) the encapsulation of a (A) complete module set, (B) good module, and (C) bad module. For all cases, $l = 3$, $l_m = 2$ and the solution string is 111. The genotype elements mapping to the solution string are circled.

keep the phenotype string length l constant. We can write the solution density as follows:

$$\rho_{s_{CM}} = \frac{\sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \binom{l-(l_m-1)n}{n}}{|L(\mathcal{S}_{CM})|}$$

where l is the phenotype length. Let us replace $|L(\mathcal{S}_{CM})|$ with its equivalent given in Equation 7.3:

$$\rho_{s_{CM}} = \frac{\sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \binom{l-(l_m-1)n}{n}}{\sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} 2^{l-l_m n} \sigma_m^n \binom{l-(l_m-1)n}{n}}$$

where σ_m is the number of modules in the alphabet and $\sigma_m = 2^{l_m}$ for encapsulation of a complete module set case. If we replace σ_m with 2^{l_m}, we obtain the following:

$$\rho_{s_{CM}} = \frac{\sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \binom{l-(l_m-1)n}{n}}{2^l \sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \binom{l-(l_m-1)n}{n}}$$

This expression reduces to:

$$\rho_{s_{CM}} = \frac{1}{2^l}$$

Notice that $\rho_{s_{CM}}$ given above is equal to the solution density of the original search space given in Equation 8.2. This result shows that the solution density of a search space remains unchanged after the encapsulation of a complete module set. Thus, our comparison of solution density before and after encapsulation of a complete module set indicates that solution density does not change with the encapsulation a complete module set.

8.2 Encapsulation of a Module Fully Included in the Solution

We expect encapsulating a good module to increase the solution density of a search space. Let $\mathcal{S}_O$ be the search space before encapsulation where $\Sigma_O = \{0, 1\}$ and $\mathcal{S}_{GM}$ be the search space after the encapsulation of M where $\Sigma_{GM} = \{0, 1, M\}$. Before the analysis, let us illustrate the changes that occur when encapsulating a good module in an example. Assume $l = 3$, $l_m = 2$, the solution string is 111 and the module is $M = 11$. Figure 8.1(B) shows the genotype space before and after encapsulation of a good module, $L(\mathcal{S}_{g_O})$ and $L(\mathcal{S}_{g_{GM}})$, respectively. In $L(\mathcal{S}_{g_O})$, the number of solutions is 1 and the search space size is 8. Thus, the solution density in $\mathcal{S}_O$ is $\frac{1}{8}$. In $L(\mathcal{S}_{g_{GM}})$, the number of solutions is 3 and the search space size is 12. The solution density in $\mathcal{S}_{GM}$ is $\frac{3}{12}$ which is larger than the solution density in $\mathcal{S}_O$. In this example, we observe an increase in the solution density due to encapsulation of module M.

In order to analyze the changes in the solution density after encapsulation of a good module, M, we derive a formula to calculate solution density of the search space $\mathcal{S}_{GM}$ and compare it with the solution density of search space $\mathcal{S}_O$ given in Equation 8.2. We use Equation 8.1:

$$\rho_{s_{GM}} = \frac{n_s}{|L(\mathcal{S}_{GM})|}$$

We can calculate n_s, the number of solution strings in a search space $\mathcal{S}_{GM}$, by calculating the number of ways to place n modules of length l_m in a string of length l, where $0 \leq n \leq \lfloor \frac{l}{l_m} \rfloor$. $\sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \binom{l-(l_m-1)n}{n}$ gives the number of ways to place n modules of length l_m in a string of length l. The upper boundary $\lfloor \frac{l}{l_m} \rfloor$ is the maximum number of modules that can be in a string to keep the phenotype string length l constant. We can write the solution density as:

$$\rho_{s_{GM}} = \frac{\sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \binom{l-(l_m-1)n}{n}}{|L(\mathcal{S}_{GM})|}$$

If we replace $|L(\mathcal{S}_{GM})|$ with its equivalent given in Equation 7.3:

$$\rho_{s_{GM}} = \frac{\sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \binom{l-(l_m-1)n}{n}}{\sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} 2^{l-l_m n} \sigma_m^n \binom{l-(l_m-1)n}{n}}$$

where $\sigma_m = 1$ because we encapsulate only one module. Thus:

$$\rho_{s_{GM}} = \frac{\sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \binom{l-(l_m-1)n}{n}}{2^l \sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \frac{1}{2^{l_m n}} \binom{l-(l_m-1)n}{n}} \tag{8.3}$$

Next, we analyze the effects of encapsulating a good module on the solution density of the search space. Let $f(n) = \binom{l-(l_m-1)n}{n}$. Clearly, $(\forall n \in \mathbb{N}^+)[f(n) \geq 0]$. The following is true:

$$f(n) > \frac{f(n)}{2^{l_m n}} \quad \forall n > 0 \textit{ and } \forall l_m > 0$$

because $(\forall n > 0 \wedge \forall l_m > 0)[2^{l_m n} > 1]$. Clearly, $l_m \in \mathbb{N}^+$ and $l_m > 1$ because the content of a module of size 1 would be identical to one of the elements of the alphabet. Summing each side of the inequality for all n where $0 < n < \lfloor \frac{l}{l_m} \rfloor$ does not change the direction of the inequality.

$$\sum_{n=1}^{\lfloor \frac{l}{l_m} \rfloor} f(n) > \sum_{n=1}^{\lfloor \frac{l}{l_m} \rfloor} \frac{f(n)}{2^{l_m n}}$$

When we add $f(0)$ to both sides of the inequality, the direction of the inequality remains the same.

$$f(0) + \sum_{n=1}^{\lfloor \frac{l}{l_m} \rfloor} f(n) > f(0) + \sum_{n=1}^{\lfloor \frac{l}{l_m} \rfloor} \frac{f(n)}{2^{l_m n}}$$

We can rewrite this expression as follows

$$\sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} f(n) > \sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \frac{f(n)}{2^{l_m n}}$$

Let us divide both sides by $2^l \sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \frac{f(n)}{2^{l_m n}}$:

$$\frac{\sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} f(n)}{2^l \sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \frac{f(n)}{2^{l_m n}}} > \frac{1}{2^l}$$

Because $(\forall n > 0 \wedge \forall l_m > 0)[2^l \sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \frac{f(n)}{2^{l_m n}} > 0]$, the direction of the inequality does not change. Notice that the right hand side of the inequality is equal to the solution density of the original search space, ρ_{s_O}, given in Equation 8.2 and the left hand side of the inequality is equal to the solution density of the search space after module encapsulation given in Equation 8.3. Thus,

$$\rho_{s_{GM}} > \rho_{s_O}$$

Therefore, we can conclude that the solution density increases after we encapsulate a good module.

8.3 Encapsulation of a Module Fully Excluded From the Solution

We analyze the effects of encapsulating a bad module on the solution density of a search space. We expect the solution density to decrease as a result of an encapsulation of such a module. As explained in Chapter 7, encapsulating a module increases the search space size. All of the new elements of the search space include at least one module. Thus if the new module is not part of the solution, none of the new elements can be a solution. The number of solutions remains the same while the search space size increases. Let $\mathcal{S}_O$ be the search space before encapsulation where $\Sigma_O = \{0, 1\}$ and $\mathcal{S}_{BM}$ be the search space after the encapsulation of M where $\Sigma_{BM} = \{0, 1, M\}$.

Before the analysis, let us illustrate the changes that occur when encapsulating a bad module in an example. Assume $l = 3$, $l_m = 2$, the solution string is 111 and the module is $M = 00$. Figure 8.1(C) shows the corresponding sets of elements of the genotype space before and after encapsulation of a bad module, $L(\mathcal{S}_{g_O})$ and $L(\mathcal{S}_{g_{BM}})$, respectively. Recall that we assume the number of solutions in the original search space to be one. The solution density in $\mathcal{S}_O$ is $\frac{1}{8}$ and the solution density in $\mathcal{S}_{BM}$ is $\frac{1}{12}$ which is smaller than the solution density in $\mathcal{S}_O$.

Next, we derive a formula to calculate solution density after encapsulating a bad module and we show that encapsulating such modules reduces the solution density in a search space. As we stated earlier, all of the new elements in $L(\mathcal{S}_{g_{BM}})$ includes at least one module and, therefore, cannot be a solution. Thus, the number of solutions remains unchanged at 1 and $n_s = 1$. Using Equation 8.1, we can write the solution density of the search space after module encapsulation as:

$$\rho_{s_{BM}} = \frac{1}{|L(\mathcal{S}_{BM})|}$$

If we replace the search space size with its equivalence given in Equation 7.3, we obtain the following:

$$\rho_{s_{BM}} = \frac{1}{\sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} 2^{l-l_m n} \sigma_m^n \binom{l-(l_m-1)n}{n}}$$

In Chapter 7, we have shown that the following is true:

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

We can rewrite this inequality as follows:

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

The left and right hand side of this inequality give $\rho_{s_{BM}}$ and ρ_{s_O} respectively. Hence,

$$\rho_{s_{BM}} < \rho_{s_O}$$

In other words, encapsulating a bad module decreases the solution density in a search space.

8.4 Summary

Solution density gives us a measure of the redundancy of the solutions in the search space. The assumption is that search spaces with higher solution redundancy are more advantageous in general for search algorithms [RG03]. We analyze the changes in the solution density in a search space by comparing solution density before and after module encapsulation. We analyze three cases: encapsulating a complete module set, encapsulating a good module and encapsulating a bad module. We show that encapsulating a complete module set does not change the solution density in the search space. We also show that the solution density increases after we encapsulate a good module while it decreases when encapsulating a bad module.

CHAPTER 9
EFFECTS OF MODULE ENCAPSULATION ON SEARCH SPACE CONNECTIVITY

The solution density gives a somewhat narrow view of the search space, because it focuses only on solutions. To get a better picture of how the search space changes with module encapsulation, we also analyze the connectivity of the elements in a search space. Connectivity examines the average distances from search space elements to a solution. We define the distance between two elements in the search space to be the number of characters at which they differ. Hamming distance is a simple but appropriate metric to use.

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

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

We examine the effects of module encapsulation on the average Hamming distance of a search space. The average Hamming distance of a search space denoted by Δ_{avg} is the average of the Hamming distances between a solution string, and every other string in the search space. In a uniformly distributed binary search space, the expected value of Δ_{avg} will be $l/2$. If Δ_{avg} decreases after a search space transformation, the elements of the new search space are, on average, closer to the solution. If Δ_{avg} increases, the elements of the new search space are, on average, further away from the solution.

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

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

Next, let us assume that $\mathcal{S}_O$ is a search space with a solution string and the corresponding genotype space $\mathcal{S}_{g_O}$, and $\mathcal{S}_{g_M} = \mathcal{E}(\mathcal{S}_{g_O}, M)$, with the corresponding search space $\mathcal{S}_M$. In addition, Δ_{avg_O} and Δ_{avg_M} are the average Hamming distances to solution in $\mathcal{S}_O$ and $\mathcal{S}_M$, respectively. There can be three outcomes of a module encapsulation in terms of average Hamming distance in a search space:

1. The average Hamming distance decreases.

   $$\Delta_{avg_O} > \Delta_{avg_M}$$

   In other words, the module encapsulation transformation results in a space with elements that are, on average, closer to the solution.

2. The average Hamming distance increases.

   $$\Delta_{avg_O} < \Delta_{avg_M}$$

   The module encapsulation results in a space with elements that are, on average, farther away from the solution.

3. The average Hamming distance remains the same.

   $$\Delta_{avg_O} = \Delta_{avg_M}$$

The module encapsulation results in a space with elements that are, on average, in the same distance from the solution as before the encapsulation.

Δ_{avg}, the average Hamming distance between an arbitrary individual and the solution string, is calculated using the following equation:

$$\Delta_{avg} = \sum_{d=0}^{l} d i_d \tag{9.1}$$

where d is step distance away from the solution and ranges from 0 to l and for a given d, i_d is the probability density function which calculates the ratio of the number of strings that are d distance away from the solution in a search space. We calculate i_d as follows:

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

where

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

Notice that i_d is the only part that can be different in each of the three cases analyzed in the following section. Therefore, we can derive i_d for $\mathcal{S}_{CM}, \mathcal{S}_{GM}$ and $\mathcal{S}_{BM}$ and plug it into Equation 9.1 to calculate Δ_{avg} for each case.

9.1 Analyzing Average Hamming Distance After Module Encapsulation

We can now study how connectivity changes by comparing the average Hamming distance after a module encapsulation. In this section, we first derive i_d for $\mathcal{S}_O$, the search space before encapsulation, and then use i_d to calculate the average Hamming distance before encapsulation, Δ_{avg_O}. $\binom{l}{d}$ gives the number of ways to choose d bits that differs from the solution string in a string of length l. Dividing $\binom{l}{d}$ by the search space size 2^l gives the density of the elements that are d distance away from the solution in the search space. Thus, i_d is:

$$i_d = \frac{\binom{l}{d}}{2^l}$$

Plugging this into Equation 9.1, we obtain Δ_{avg_O}:

$$\Delta_{avg_O} = \sum_{d=0}^{l} d\frac{\binom{l}{d}}{2^l} \tag{9.2}$$

This can be simplified to:

$$\Delta_{avg_O} = \frac{l}{2}$$

As we mentioned earlier, this is the expected value of the Δ_{avg_O}.

In the following subsections, we derive i_d for $\mathcal{S}_{CM}, \mathcal{S}_{GM}$ and $\mathcal{S}_{BM}$. In each case, we use the i_d that we derive to calculate $\Delta_{avg_{CM}}, \Delta_{avg_{GM}}$ and $\Delta_{avg_{BM}}$ which we then compare with Δ_{avg_O} to analyze how search space connectivity changes with module encapsulation.

9.1.1 Complete Module Set Encapsulation

We expect to observe no change in the average Hamming distance in a search space after encapsulation of a complete module set. Because we encapsulate one module for each possible genomic string of a given length, we do not expect the new elements to change the ratio of the existing elements for a given distance. Let Σ_O and Σ_{CM} be the alphabets before and after complete module encapsulation. where $\Sigma_O = \{0, 1\}$ and $\Sigma_{CM} = \{0, 1, M_1, M_2, ..., M_k\}$ where $k = 2^{l_m}$ and $l_m = |M_i|$ for $1 \leq i \leq k$. 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 are d Hamming distance away from the solution and have n modules, for all n where $0 \leq n \leq \lfloor \frac{l}{l_m} \rfloor$. 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, $|L(\mathcal{S}_{CM})|$, 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 plug i_d into Equation 9.1, we obtain:

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

where l is the phenotype length. This is the formula for calculating the average Hamming distance in a search space after a complete module set encapsulation.

Next, we compare the average Hamming distance before and after complete module set encapsulation. The search space size after encapsulating a complete module set can be calculated by using Equation 7.3

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

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

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

We can rearrange the expression in the following way:

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

This expression reduces to:

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

which is equal to Δ_{avg_O} given in Equation 9.2.

$$\Delta_{avg_{CM}} = \Delta_{avg_O}$$

Thus, we can conclude that there is no change in the average Hamming distance when encapsulating a complete module set and adding them to the original alphabet without replacing the primitives.

9.1.2 Encapsulation of a Module Fully Included in the Solution

We expect encapsulating a *good* module to reduce the average distance to the solution because it adds strings that are closer to the solution in Hamming distance. Such a case is

		d=**0**	d=**1**	d=**2**	d=**3**
No module encapsulation	i_{d_O}	.125	.375	.375	.125
Good module encapsulation	$i_{d_{GM}}$	.250	.417	.250	$.08\bar{3}$
Bad module encapsulation	$i_{d_{BM}}$	$.08\bar{3}$	.250	.417	.250

Table 9.1: Probability density of each distance value d in search spaces $\mathcal{S}_O$, $\mathcal{S}_{GM}$ and $\mathcal{S}_{GM}$ which are search spaces before module encapsulation, after *good* module encapsulation and after *bad* module encapsulation respectively.

illustrated in the following example. Assume $l = 3$, $l_m = 2$, the solution string is 111 and the module is $M = 11$. The sets of the elements of the genotype space before and after the encapsulation are shown in Figure 8.1(B). Note that of the four new strings added to the genotype space, two are solutions. The probability density values for this example are given in Table 9.1. Recall that i_d gives the ratio of the elements that are a distance of d 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. In this example, we compare the values from row 1 (no module encapsulation) and row 2 (good module encapsulation). Notice that strings which are at a Hamming distance zero, $d = 0$, are the solution strings. The ratio of solution strings clearly increases after good module encapsulation: $i_{0_{GM}} = \frac{3}{12} = .250$ is twice as much as $i_{0_O} = \frac{1}{8} = .125$ Similarly, if we compare row 1 and row 2 for $d = 1$, we observe that probability density after a good module encapsulation, $i_{1_{GM}} = .417$, is larger than the probability density of no module encapsulation, $i_{1_O} = .375$. In other words, the ratio of the number of strings that are one step away from a solution in a search space after module encapsulation is larger than the one in the original search space. For the farther distances, $d = 2$ and $d = 3$, the probability density becomes smaller when encapsulating a good module.

For $d = 2$, the probability density of a good module encapsulation is $i_{2_{GM}} = .250$ and is smaller than the probability density of no module encapsulation, $i_{2_O} = .375$. Similarly, For $d = 3$ the probability density of a good module encapsulation is $i_{3_{GM}} = .08\bar{3}$ and is smaller than the probability density of no module encapsulation, $i_{3_O} = .125$. Thus, the ratio of the strings which are closer to the solution string increases after the encapsulation of a good module. The ratio of the strings that are farther away from the solution, on the other hand, becomes smaller as a result of a good module encapsulation.

We can calculate the average Hamming distance of each case for this example.

$$\Delta_{avg_O} = \sum_{d=0}^{l} d i_{d_O} = 1.5$$

and

$$\Delta_{avg_{GM}} = \sum_{d=0}^{l} d i_{d_{GM}} = 1.084$$

Hence, $\Delta_{avg_{GM}} < \Delta_{avg_O}$. In other words, the average Hamming distance of the search space with a good module is smaller than that of the search space with no modules.

Let us define the probability density function i_d for $\mathcal{S}_{GM}$.

$$i_d = \frac{1}{|L(\mathcal{S}_{GM})|} \sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \binom{l - l_m n}{d} \binom{l - (l_m - 1)n}{n} \tag{9.3}$$

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 any of the d unmatched bits. Therefore, these d unmatched 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 thesis.

We can plug i_d we derive into Equation 9.1 to obtain $\Delta_{avg_{GM}}$:

$$\Delta_{avg_{GM}} = \sum_{d=0}^{l} d \frac{1}{|L(\mathcal{S}_{GM})|} \sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \binom{l - l_m n}{d} \binom{l - (l_m - 1)n}{n} \tag{9.4}$$

where $|L(\mathcal{S}_{GM})|$ is given in Equation 7.3. We compare $\Delta_{avg_{GM}}$ with Δ_{avg_O} in Section 9.1.4.

9.1.3 Encapsulation of a Module Fully Excluded From the Solution

We expect that encapsulating a bad module will have the opposite effect, increasing average Hamming distance to the solution. *Bad* modules increase the average Hamming distance by adding new strings that are at least *bad* module length away from the solution into the search space. For instance, assume $l = 3$, $l_m = 2$, the solution string is 111 and the module is $M = 00$. The sets of the elements of the genotype space before and after the encapsulation are shown in Figure 8.1(C). Note that of the four new strings added to the genotype space, none of them is solution.

Table 9.1 shows that the probability density of each distance value d in each search space. We compare the values from row 1 (no module encapsulation) and row 3 (bad module encapsulation). The ratio of the elements that are $d = 0$ distance away from the solution to the search space size clearly decreases after a bad module encapsulation: $i_{0_{BM}} = .08\bar{3}$ is smaller than $i_{0_O} = .125$. When we compare row 1 and row 3 for $d = 1$, we observe that probability density after a bad module encapsulation, $i_{1_{BM}} = .250$, is smaller than the probability density of no module encapsulation, $i_{1_O} = .375$. In other words, the ratio of the number of strings that are one step away from a solution in a search space after module encapsulation is smaller than the one in the original search space. For the farther distances, $d = 2$ and $d = 3$, the probability density becomes larger when encapsulating a bad module. For $d = 2$, the probability density of a bad module encapsulation is $i_{2_{BM}} = .417$ and is larger than the probability density of no module encapsulation, $i_{2_O} = .375$. Similarly, for $d = 3$ the probability density of a bad module encapsulation is $i_{3_{BM}} = .250$ and is larger than the probability density of no module encapsulation, $i_{3_O} = .125$. Thus, the ratio of the strings which are closer to the solution string decreases after the encapsulation of a bad module. The ratio of the strings that are farther away from the solution, on the other hand, becomes larger as a result of a bad module encapsulation.

We can calculate the average Hamming distance of each case for this example.

$$\Delta_{avg_O} = \sum_{d=0}^{l} d i_{d_O} = 1.5$$

and

$$\Delta_{avg_{BM}} = \sum_{d=0}^{l} d i_{d_{BM}} = 1.83$$

Hence, $\Delta_{avg_{BM}} > \Delta_{avg_O}$. In other words, the average Hamming distance of the search space with a bad module is larger than that of the search space with no modules.

Let us define the probability density function for the search space that has a bad module in its alphabet.

$$i_d = \frac{1}{|L(\mathcal{S}_{BM})|}(\binom{l}{d} + \sum_{n=1}^{\lfloor \frac{l}{l_m} \rfloor} f(n)\binom{l-(l_m-1)n}{n}) \tag{9.5}$$

where $f(n)$ is

$$f(n) = \begin{cases} \binom{l-l_m n}{d-l_m n} & \text{if } d > l_m \\ 0 & \text{otherwise} \end{cases} \tag{9.6}$$

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 unmatched bits into the string), $\binom{l}{d}$ gives all possible strings that do not include any module, and d is the distance away from the solution. The rest of the expression is zero because 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 unmatched bits come from the n modules. The rest of the d unmatched 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 $n \geq 1$, $\sum_{n=1}^{\lfloor \frac{l}{l_m} \rfloor} f(n)\binom{l-(l_m-1)n}{n}$ gives all possible strings that are d distance away from the solution and include at least one module.

By inserting the above equation for i_d into Equation 9.1, we obtain the average Hamming distance to the solution:

$$\Delta_{avg_{BM}} = \sum_{d=0}^{l} \left(\frac{d}{|L(\mathcal{S}_{BM})|} \left(\binom{l}{d} + \sum_{n=1}^{\lfloor \frac{l}{l_m} \rfloor} f(n) \binom{l-(l_m-1)n}{n} \right) \right) \tag{9.7}$$

where $f(n)$ is given in Equation 9.6. If we replace $|L(\mathcal{S}_{BM})|$ with its equivalence given in Equation 7.3, we obtain the following:

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

We compare $\Delta_{avg_{BM}}$ with Δ_{avg_O} in Section 9.1.4.

9.1.4 Comparison of Average Hamming Distances

We can now compare Equations 9.2, 9.4 and 9.7 given in Section 9.1 to observe the changes in the average Hamming distance in a search space when encapsulating a good module and a bad module. The equations are restated below for convenience of exposition in the following order: average Hamming distance with no modules, average Hamming distance with a good module and average Hamming distance with a bad module.

$$\Delta_{avg_O} = \sum_{d=0}^{l} d \frac{\binom{l}{d}}{2^l} \tag{9.2}$$

$$\Delta_{avg_{GM}} = \sum_{d=0}^{l} d \frac{1}{|L(\mathcal{S}_{GM})|} \sum_{n=0}^{\lfloor \frac{l}{l_m} \rfloor} \binom{l - l_m n}{d} \binom{l - (l_m - 1)n}{n} \tag{9.4}$$

$$\Delta_{avg_{BM}} = \sum_{d=0}^{l} (\frac{d}{|L(\mathcal{S}_{BM})|} (\binom{l}{d} + \sum_{n=1}^{\lfloor \frac{l}{l_m} \rfloor} f(n) \binom{l - (l_m - 1)n}{n})) \tag{9.7}$$

There are two variables in these equations: the solution length, l, and the module length, l_m. First, we keep l_m constant and compare these three equations for $l \geq l_m$. We exclude values for $l < l_m$, because a string of length l cannot include a module that is larger than the length of the string. Second, we keep l constant and compare them for $l_m \geq 2$.

In the first case, l_m is kept constant at a small value, $l_m = 2$. Figure 9.1 illustrates the comparison of calculated values of Δ_{avg_O}, $\Delta_{avg_{GM}}$ and $\Delta_{avg_{BM}}$. This figure shows that $\Delta_{avg_{GM}}$ is smaller than Δ_{avg_O}. In other words, encapsulating a good module decreases the average Hamming distance in a search space. On the other hand, $\Delta_{avg_{BM}}$ is larger than Δ_{avg_O}, which means that encapsulating a bad module increases the average Hamming distance in a search space. We also observe that for all cases the average Hamming distance increases with the problem size l. Thus, the larger the phenotype length, the more prominent the effect of module encapsulation is on the average Hamming distance to solution.

In the second case, we keep the problem size constant at $l = 256$. Figure 9.2 plots the calculated values of Δ_{avg_O}, $\Delta_{avg_{GM}}$ and $\Delta_{avg_{BM}}$ for various module lengths $l_m \in \{2, 4, 6, 8, 16, 32, 64\}$. When a bad module is encapsulated, the average Hamming distance is 167 for $l_m = 2$ and decreases as the module length increases. At $l_m = 16$, the average Hamming distance is slightly larger than 128 and while it continues to decrease as the module length increases, it remains larger than 128. When no modules are encapsulated, the average Hamming dis-

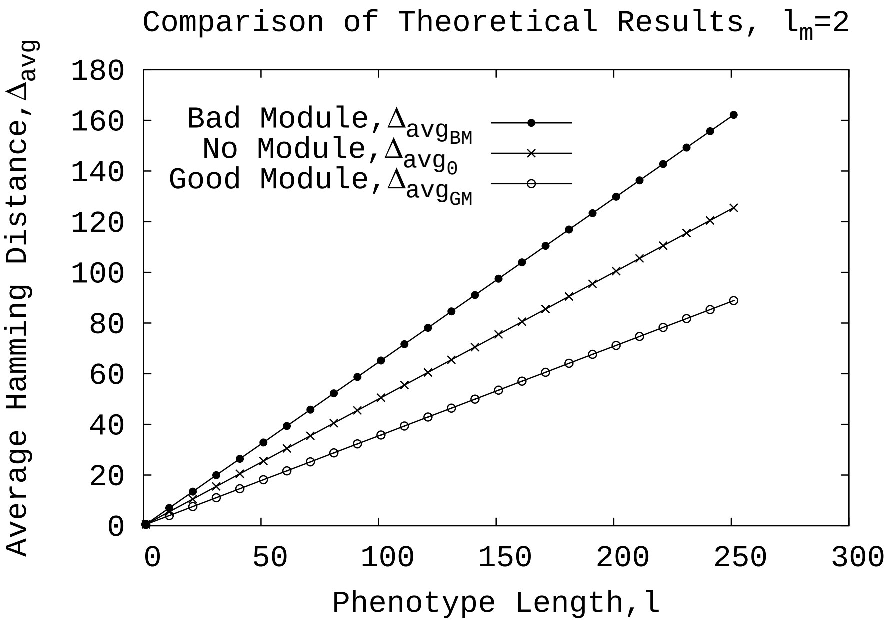


Figure 9.1: Comparison of theoretical results of average Hamming distance to solution in three structurally different search spaces: a search space with a good module of size two, with no modules and with a bad module of size two. The plot shows Equations 9.2, 9.4 and 9.7 with fixed $l_m = 2$ and various values of l, l-bits OneMax. The x-axis shows the phenotype length, l, and the y-axis shows the average Hamming distance to the solution for the three cases. The Hamming distance is smaller when a good module is encapsulated and larger when a bad module is encapsulated. The larger the phenotype length, the more prominent the effect of module encapsulation is on the average Hamming distance to solution.

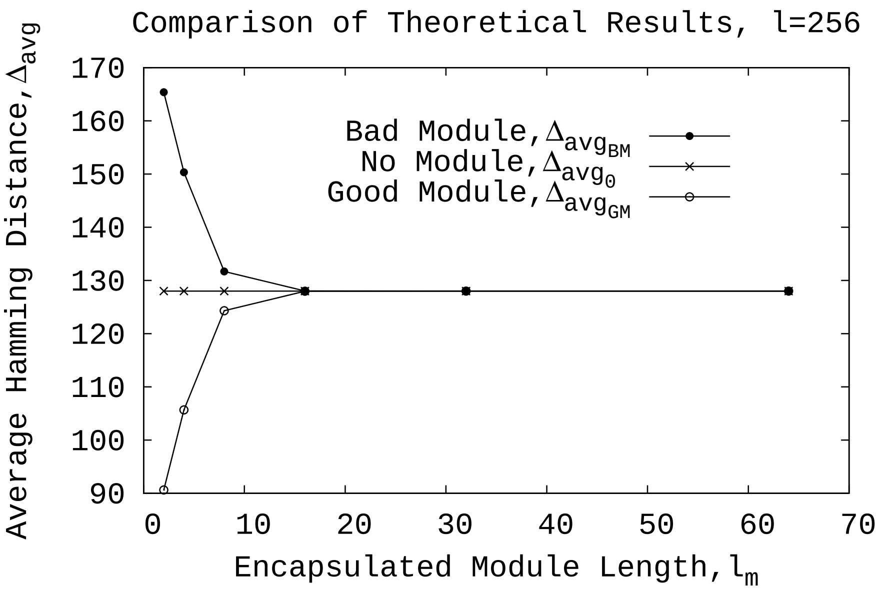


Figure 9.2: Comparison of theoretical results of average Hamming distance to solution in three structurally different search spaces: a search space with a good module of various sizes, with no modules, and with a bad module of various sizes. The plot shows Equations 9.2, 9.4 and 9.7 with various values of l_m and fixed $l = 256$, 256-bit OneMax. 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 to solution for the three cases. The Hamming distance is smaller when a good module is encapsulated and larger when a bad module is encapsulated. The larger the module length, the less prominent the effect of module encapsulation is on the average Hamming distance to solution.

tance is 128 and remains the same as the module length increases. This result is obvious because, in this case, there is no modules in the alphabet. Therefore, the module length has no effect in a search space with no modules. When a good module is encapsulated, the average Hamming distance is 90 for $l_m = 2$ and increases with the module length. At $l_m = 16$, the average Hamming distance is slightly smaller than 128. It continues to increase with the module length, but remains smaller than 128. Thus, we can conclude that in this example, $\Delta_{avg_{GM}}$ is smaller than Δ_{avg_O} and $\Delta_{avg_{BM}}$ is larger than Δ_{avg_O}. In other words, encapsulating a good module decreases the average Hamming distance in a search space and encapsulating a bad module increases the average Hamming distance. This observation is obvious from looking at the figure for small values of l_m, but not for larger l_m values. As l_m increases, the impact of module encapsulation decreases. In the example given in Figure 9.2, the average Hamming distance converges around $l_m = 16$ and after this point, in practice, we can consider that there is no impact or that the impact is minimal. The larger the module length, the less prominent the effect of module encapsulation is on the average Hamming distance to solution.

The question of why the average Hamming distance is affected by module encapsulation in a significant way only for small modules is not so obvious. Encapsulating a small module results in a larger number of new elements introduced into a search space. The phenotypes of these new elements duplicates some of the existing elements in a search space. Thus, modularity creates redundancy in a search space. By definition, all of the new elements must include at least one module. If the encapsulated module is a good module, encapsulation will increase the redundancy of the elements that are closer to the solution in Hamming distance. If the encapsulated module is a bad module it will increase the redundancy of the elements that are farther away from the solution. Increasing the encapsulated module length reduces the number of new elements introduced into the search space therefore it reduces redundancy.

For example, let us assume that the phenotype length is 3. The search space without any module has the following elements $\{000, 001,$
$010, 011, 100, 101, 110, 111\}$. Encapsulating a module of length 2 adds the following new elements into this search space: $\{0M, M0, 1M, M1\}$. Each of these new elements is a copy of an element in the original search space. The new search space that includes these new elements has the redundant copies of four of its existing elements. Assume that M is 11. The new search space contains the two copies of elements 011 and 110 and three copies of the element 111. If we encapsulate a module length of 3, the new search space will have one new element: $\{M\}$. Assuming that M is 111, the new search space contains the two copies of elements 111. As we see in this example, the redundancy in a search space decreases with the module length. Lower the redundancy, lower the effect of a module encapsulation is on the average Hamming distance.

In summary, the theory shows that encapsulating a good module always produces a decrease in the average Hamming distance to the solution and encapsulating a bad module always produces an increase. This effect, however, is noticeable only for small modules. Encapsulating a small module results in a higher level of redundancy and, therefore, increases the impact of module encapsulation on the average Hamming distance to solution. Encapsulating a larger module results in a lower level of redundancy and, therefore, decreases the impact of module encapsulation. This impact decreases rapidly with the increase of module size l_m and for large values of l_m, the impact is, in practice, negligible.

9.2 Experimental Analysis

In order to show validity of our theoretical study presented in Section 9.1, we perform a complementary experimental analysis to be compared with our theoretical results.

9.2.1 Methodology

We use a standard Genetic Algorithm with mutation as the only variation operator and measure the average fitness value of the randomly generated initial population and of the final population. The fitness value we measure in our experiments falls in the range $[0 : 1]$. Our test problem is the OneMax which maximizes the number of 1's in a solution string. In order to be parallel with our theoretical study, we define a good module as a subsequence of all ones and a bad module as a subsequence of all zeros. The OneMax problem has the property of having a positive correlation between the form and quality of candidate solutions. In other words, higher fitness indicates lower Hamming distance to solution which means elements of a search space in average move closer to the solution. Reciprocally, lower fitness indicates higher Hamming distance. Therefore, we expect correlated behavior between the

Hamming distance calculated theoretically and the fitness value obtained experimentally. In fact, for the OneMax problem, we can directly compare theoretical and experimental Hamming distance simply by noticing that for our experiments:

$$Normalized\ Average\ Hamming\ Distance = \ 1 - \ Average\ Fitness$$

In our theoretical study, we calculate average hamming distance of a search space by taking into account every element of the search space. In other words, we analyze the whole search space. A GA, however, operates on a population which is a, typically very small, subset of the search space. As a result, and in contrast with our theoretical work, our experimental study takes into account only the elements in that small subset of the entire search space. The elements of the initial population are randomly sampled elements from the search space due the initialization process. Because we analyze the whole search space in our theoretical analysis, we expect these results to be comparable to the experimental results obtained from the randomly sampled initial population. After the first population is initialized, the GA creates new populations by applying genetic operators and selection to the current population. These evolved populations no longer resemble the search space structure.

In the first part of our experimental study, Experiment 1, we compare our theoretical results with the experimental results where we obtain average fitness of the initial population. As expected, we observe that these two results correlate.

In the second part of our experimental study, Experiment 2, we compare our theoretical results with the experimental results where we run a GA and obtain the average fitness of the final generation. We, however, did not observe correlation between the results of this second experiment and our theoretical results because the genetic operators alter the population structure.

9.2.2 Initialization Issues

Let us begin with a discussion on the bias in the initial population introduced by the canonical GA initialization. Because, the phenotypes in our mathematical model are of fixed size, the size of the genotype can vary. Genotypes that include modules are smaller because a module expands into a sequence of two or more primitives. We implement the same genotype to phenotype mapping model in our empirical study. If we use a simple "coin toss" random sampling of the space, we observe a uniformly distributed population when we have a binary alphabet only and we observe a bias when we have a module in the alphabet. Elements containing modules have greater probability to have the modules located toward the beginning of the string. We explain the reason with an example. Assume that the phenotype length is 3 and the module length is 2. The alphabet contains 0, 1 and module M. We randomly select an element from the alphabet where each element has the same probability of being selected as long as the length of the symbol is permissible in the location it is selected for. In our example, * represents 0 or 1. The probabilities of having string of

M*, *M and *** are $\frac{1}{3}$, $\frac{2}{9}$ and $\frac{4}{9}$ respectively. Ratios of strings of M*, *M and *** in a search space[1] in our theoretical analysis are $\frac{1}{6}$, $\frac{1}{6}$ and $\frac{8}{12}$ respectively. Clearly, the probabilities of strings in a population differs from their corresponding ratios in a search space.

In order to avoid introducing a bias, we initialize GA using an exhaustive method. We first enumerate all the elements of the search space and randomly sample this enumeration to produce the initial population in an unbiased fashion. No doubt faster initialization methods can be developed but that is not the focus of this thesis. This exhaustive method ensures that the initial population is unbiased but limits the size of the problem due to the computational power required for this exhaustive initialization approach.

9.2.3 Experimental Setting

Because enumerating all the elements of a search space is computationally costly, we limit the test problem used in our experiments to the same instance of the OneMax problem presented in previously but with length of 20. Our fitness function is the ratio of the number of ones over the individual length. All experiments use the following parameter values: the mutation rate is 0.01 and the selection type is tournament with size 4. We run the GA using the same module lengths as the theoretical data in Figure 9.2: $l_m = \{2, 4, 8, 16\}$; but we include the experimental limit length of 20 and exclude module sizes larger than this limit. We report the normalized average Hamming distance calculated directly from the average fitness values

[1]The set of elements of the search space of phenotype length 3 and alphabet {0, 1,M}: $\{000, 001, 010, 011, 100, 101, 110, 111, M0, M1, 0M, 1M\}$

collected. We perform 100 trials for all experiments and report average values with their 95% confidence intervals. For Experiment 1, the population size is 100. For Experiment 2, the population size [2] and the number of generations are both 20.

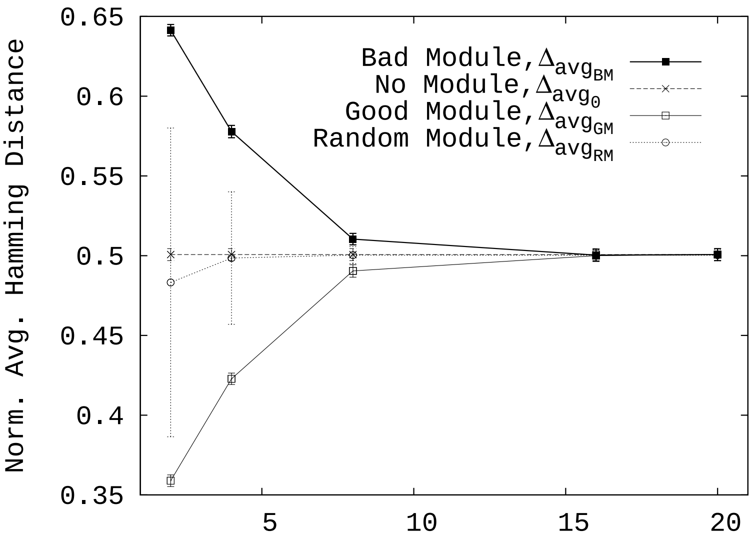
9.2.4 Results

Figure 9.3 compares our theoretical analysis with our experimental results from Experiment 1 and Experiment 2. In all three plots, the x-axis shows increasing module length and the y-axis gives the normalized Hamming distance. In our theoretical analysis, average Hamming distance falls in the range $[0 : l]$. In order to make the comparison between the fitness value and the average Hamming distance clearer, we normalize the average Hamming distance by dividing it by the phenotype length, l. As a result, the normalized average Hamming distance and the average fitness now both range between 0 and 1 with 1 representing the farthest distance or highest fitness respectively.

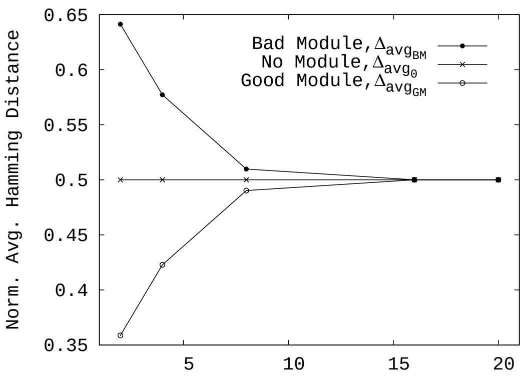
Figure 9.3(A) plots the theoretical average Hamming distance calculated from Equations 9.2, 9.4 and 9.7. As we saw in Figure 9.2, for the no module case, the average Hamming distance remains constant with the module length, because the alphabet does not contain modules. For bad module case, the normalized average Hamming distance is 0.65 for module length 2 and decreases down to nearly 0.5 around module length 16. For the good module case, the normalized average Hamming distance is 0.35 for module size 2 and increases up

[2] We use a larger population for Experiment 1 because all data is collected from the initial population.

(B)



(A)



(C)

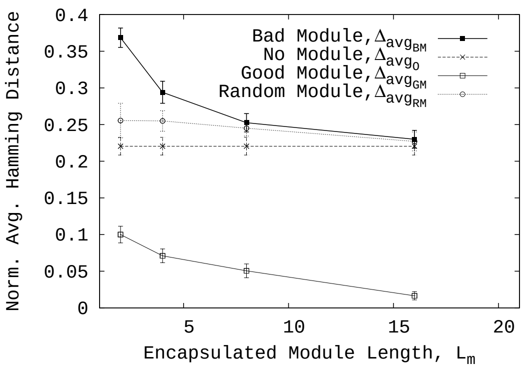


Figure 9.3: Comparison of the normalized average Hamming distance to solution in four structurally different search spaces: a search space with a good module, a search space with no modules, a search space with a bad module, and a search space with random modules. The modules are of various lengths. The target problem is 20-bit OneMax. The x-axis shows the encapsulated module length $l_m = \{2, 4, 8, 16, 20\}$ and the y-axis shows the average Hamming distance. (A) Theoretical results from Equations 9.2, 9.4 and 9.7. (B) Experimental results for a GA initial generation. The average Hamming distance is calculated as $(1 - \text{average best fitness})$ and reported with 95% confidence intervals. (C) Experimental results for a GA final generation.

to nearly 0.5 around module length 16. It remains nearly constant after module size 16 for both good and bad module cases. Thus, average Hamming distance when encapsulating a bad module is significantly higher than the average Hamming distance when encapsulating a good module when the module length is smaller than 16 and the difference is marginal for larger module length.

Figure 9.3(B) shows the results from Experiment 1 which examines the average Hamming distance of the initial population where we sample a uniformly distributed search space. For the no module case, the average Hamming distance remains constant with the module length. For bad module case, the normalized average Hamming distance is 0.65 where the module length is 2 and decreases down to approximately 0.5 around module length 16. For good module case, the normalized average Hamming distance is 0.35 where module size is 2 and increases up to approximately 0.5 around module length 16. It remains nearly unchanged after module size 16 for both good and bad module cases. Thus, similar to our theoretical results given in Figure 9.3(A), results of Experiment 1 given in Figure 9.3(B) shows that average Hamming distance when encapsulating a bad module is significantly higher than the average Hamming distance when encapsulating a good module when the module length is smaller than 16 and the difference is marginal for larger module lengths. The theoretical results in which we analyze the normalized average Hamming distance in a whole search space correlate with our experimental results where we analyze the normalized average Hamming distance of the randomly generated initial population. The GA with a random module case is included in Experiment 1 as a baseline for the comparison. As expected, the results from

Experiment 1 show that the normalized average Hamming distance of the random module case is larger than the normalized average Hamming distance of the good module case and smaller than the normalized average Hamming distance of the bad module case.

Figure 9.3(C) shows the results from Experiment 2 which examines the average Hamming distance of the final population. For the no module case, similar to the results in Figure 9.3(A) and (B), the average Hamming distance remains constant with the module length. For the bad module case, the normalized average Hamming distance is 0.37 and decreases down to 0.22 at module length 16 and the change in the average Hamming distance slows down as the module length increases. For the good module case, the normalized average Hamming distance is 0.1 and decreases down to 0.02 at module length 16 and the change in the average Hamming distance slows down as the module length increases. Also, the normalized average Hamming distance obtained when we encapsulate a good module is smaller than the normalized average Hamming distance obtained when we encapsulate a bad module.

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. For the no module and the bad module cases, the results from Experiment 2 given in Figure 9.3(C) correlate with our theoretical results given in Figure 9.3(A). For the good module case, however, the results in Figure 9.3(C) do not correlate with our theoretical results. While the normalized average Hamming distance increases in our theoretical results, the experimental results indicate a decrease in the normalized average Hamming distance. One possible reason for this difference from the theoretical results is the selection pressure acting among the available elements in the population. Although, longer modules result in less redundancy, the new elements in the search space include at least one module and the longer the good module is the higher the fitness of an individual containing this module and smaller the Hamming distance of that individual to the solution. Hence, the new elements added to a search space as a result of a good module encapsulation are smaller in number but higher in fitness. The GA selects for higher fitness. Therefore the average fitness increases and the average Hamming distance decreases with the longer modules. Similar to Experiment 1, we also include a GA with a random module. The results are also similar to the results that we obtain in Experiment 1 where the normalized average Hamming distance of the random module case is larger than the normalized average Hamming distance of the good module case and smaller than the normalized average Hamming distance of the bad module case.

9.3 Summary

We analyze the effects of encapsulating a module on search space connectivity where connectivity is measured in terms of the average Hamming distance to the solution. Our analysis indicates the following results:

- The average Hamming distance is unchanged when encapsulating a complete module set.

- The average Hamming distance decreases when encapsulating a good module. A decrease in average Hamming distance indicates that, on average, the elements of the search space are closer to the solution string in mutation steps.

- The average Hamming distance increases when encapsulating a bad module. An increase in the average Hamming distance to solution indicates that, on average, the elements of the search space are further away from the solution string in mutation steps.

We provide an experimental validation of our theoretical results. Our analysis indicates that the results obtained from the uniformly distributed sampling of a search space correlate with our theoretical results for all three cases: a search space with no modules, a search space with a good module and a search space with a bad module. In the empirical studies, when we look at the population after the search algorithm, in this case a GA, has had a chance to run, we find that: our results correlate with the theoretical results for the search spaces with no modules and with bad modules. Our results, however, do not correlate for the good module case. We speculate that this is due to the selection pressure introduced by the GA dynamics. Our theoretical analysis is a static analysis based on search space structure and general in the sense that it does not take the dynamics of a particular search algorithm into account. We also note that, as module length increases linearly, the relative change in the normalized average Hamming distance slows down in both our experimental and theoretical results. In other words, larger modules have a smaller impact on the average Hamming distance to solution than smaller modules.

The correlation between our experimental and theoretical results suggest that although additional methods might be needed and our results limited to the problems with positive correlation between form and quality of candidate solutions and to the mutation based search algorithms, our theoretical analysis predicts the effect of an encapsulation on the performance of a search in general. It also accurately predicts the random sampling changes in the search space structure due to modularity.

CHAPTER 10
DISCUSSION

Our study focuses on two metrics: search space composition and search space connectivity. We have analyzed the composition and connectivity for search spaces before and after module encapsulation. In our analysis, we consider two classes of modules: *good* and *bad*. We call a module good when it is included in the solution string and bad otherwise. As a baseline, we also analyze the case in which we encapsulate a *complete module set*. A complete module set is a set containing one module for each possible genomic string of a given length. Throughout our study we have made following assumptions: the genomes are linear; Hamming distance is an appropriate metric for search space connectivity; there is a correlation between the form and quality of candidate solutions; and the target problem is OneMax. Module encapsulation is defined as assigning a new alphabet symbol to a string of interest. Therefore, for all cases, the search space after module encapsulation is larger than the original search space. The introduction of new elements into the search space changes the composition and the connectivity of the search space. We use these two metrics to study whether encapsulation is advantageous or detrimental for the search algorithm.

For the complete module set case, our results indicate that there is no change in the composition or connectivity of the search space. These results were expected by the way we defined complete module set and underline the fact that, simply increasing the degree of modularity of search space does not necessarily affect the performance of the search as measured by our two metrics. For the good module case, our results indicate that in terms of composition, there is an improvement in the search space after module encapsulation: the number of solutions to search space size increases. In terms of connectivity, there is also an improvement: average Hamming distance between the solution and every other element in the search space decreases, which means that the elements in the new search space on average are closer to a solution as measured by mutation steps. For the bad module case, our results indicate the complete opposite. Composition and connectivity metrics show a detrimental effect in the search space after encapsulation. The ratio of solutions to all search space elements decreases and the average Hamming distance to the solution increases.

Even though our theoretical results are valid for any module length smaller than the phenotype length, in practice, we observe that the impact of the module encapsulation decreases exponentially with the module length and increases linearly with phenotype length. From these observations, we can conclude that, for module encapsulation to have a meaningful detrimental or advantageous effect, the module length should be significantly smaller than the phenotype length. We provide equations to help determine under what conditions we can expect module encapsulation to have a significant effect. These equations can be used to guide module encapsulation in a search algorithm. In general, we can think of these results as analyzing the extreme cases of good and bad modules with the perspective of expanding the analysis into a complete spectrum of module qualities from good to bad based on how much of the module string is contained in the solution and how many modules there are. In this sense, our results hint at the possibility that the more modular a solution is, the more advantageous the encapsulation of good quality modules is.

In summary, we have answered our question of whether a module encapsulation is advantageous or detrimental as follows, good modules are always advantageous, bad modules are always detrimental, but their effect is marginal unless the module length is sufficiently smaller than the phenotype length.

Part IV

CONCLUSIONS

CHAPTER 11
FUTURE WORK

There are not many theoretical studies of modularity effects on search. Understanding hierarchically modular search spaces better, can help us understanding systems with this underlying property better. These systems include biological systems, their development and evolution, as well as complex systems in general.

Modularity in search is a new area of research with many promising directions for future work:

- Our analysis can be expanded to not only account for the two extreme cases of completely good and completely bad quality modules, but for all the spectrum of modules in between.

- Out framework can be expanded to account for tree structures at the genotype and phenotype level. Currently, our framework account only for linear structures for genotypes and phenotypes. We believe that this extension will not fundamentally change our results. While linear structures are by far the most widely used structures for evolutionary computation, three structures are the second most common structures. We should also note that while our genotypes and phenotypes are linear structures, the mapping between them is a derivation three produced by the module creation rewriting rules.

- Our analysis can be extended to account for module quality when there is only partial problem domain knowledge. The second part of our analysis uses full information about the problem domain. This is the case because the focus is on better understand the phenomena of modularity in search and its implications. We can shift the focus to better design algorithms that automatically identify and create promising modules in order to speed up the search. With no or little problem domain information, we can use our metrics and analysis to guide module encapsulation, but only if a method to estimate module quality can be defined.

CHAPTER 12
CONCLUSIONS

Most complex systems observed in nature happen to be modular from the point of view of near-decomposability. These systems also happen to be the product of evolution, in the broader sense, and for the case of biological complex systems, in the Darwinian sense. From the evolutionary computation perspective, we are interested in the evolution of complex systems as solutions to challenging problems. From the near-decomposability point of view, evolutionary search seeking to produce complex systems may not need to target arbitrary search spaces but only those search spaces that enable modular structures. As a result, we study search spaces with built-in features to represent modular solutions.

In this thesis, we define module creation as a type of search space transformation and study some of its effects on search. The underlying assumption is that when a module is created, its components are isolated from interactions and, in effect, creates a structure that enables the representation of nearly decomposable subsystems. In order to do this, we define a formal languages based framework for the study of modularity in the context of genotype to phenotype mapping and development in evolutionary search. Using this framework, we present theoretical results. These results are validated and complemented by an experimental study using the Modular Genetic Algorithm (MGA). The MGA is a traditional genetic algorithm augmented with the ability to create modules in accordance with our theoretical framework. These theoretical and experimental results allow us to answer the questions posted at the beginning of this thesis as follows:

- *Does module creation, by itself, always result in a change in reachability and redundancy of elements in the search space? Or do reachability and redundancy of the search space remain invariant after the creation of some types of module sets?* Theorem 1 shows that not all module creation transformations change the structure of the search space (Section 4.3). There are sets of modules, that when created together, neutralize the effects of each other and keep the search space invariant in terms of reachability and redundancy. We call these sets complete module sets, and their module creation transformation strict encapsulation. Furthermore, we experimentally show that for a genetic algorithm, when the search space fundamental properties of reachability and redundancy are kept constant, this particular type of module creation does not affect search performance either (Section 5.2). In contrast, we experimentally show that arbitrary module creation affects search space structure and performance (Section 5.3).

- *Are reachability and redundancy relevant structural aspects of search spaces in terms of search performance? If so, can changes in search space structure be targeted in order to improve performance?* Based on our experimental analysis, reachability and redundancy are relevant aspects of search space structures. In fact, in our experiments, they seem to be the overriding factors driving performance. In general, reachability or redundancy changes in random parts of a search space is detrimental to search performance (Section 5.3). If we introduce information about the target problem and use it to decrease reachability of unfavorable areas or increase redundancy of favorable areas, the result is a performance boost (Section 5.4).

- *Does module creation, by itself, always results in a change in composition and connectivity of elements in the search space, or does the composition and the connectivity of the search space remain invariant?* The search space structure is also invariant under the strict-encapsulation transformation when using composition and connectivity as the metric of change (Sections 8.1, 9.1.1 and 9.2.4). These results expand Theorem 1 for the case where the problem domain information is available.

- *Can module creation be targeted using problem domain knowledge in order to achieve predictable increase or decrease in search performance due to changes in composition and/or connectivity of the search space?* Module creation can be targeted to predictably impact search performance using composition and connectivity as the prediction metrics. Composition and connectivity are refined versions of redundancy and reachability using domain knowledge for the OneMax problem. We assume that the quality of a module depends on its contribution towards building the solution. We theoretically show that encapsulating good quality modules improve search space composition (Section 8.2) and connectivity (Section 9.1.2). Based on these results, for the good module case, we predict an improvement in search performance. We theoretically show also that encapsulating bad quality modules is detrimental for search space composition (Section 8.3) and connectivity (Section 9.1.3). Based on these results, for the bad module case, we predict a detrimental result on search performance. We present experimental evidence that confirms our expected theoretical results (Section 9.2.4). Experimentally, good modules are beneficial and bad modules are detrimental for performance. In addition, theoretical and experimental results indicate that the search performance impact is more prominent for module sizes that are significantly smaller in comparison with the problem size.

On the light of these answers, we now go back to address our original motivation: How and why does modularity help improve evolutionary search performance? First, not all types of modularity have an effect on search. We can have highly modular spaces that in essence are equivalent to simpler non-modular spaces, because they achieve higher degree of modularity without changing the fundamental structure of the search space. Second, for the cases when modularity actually has an effect on the fundamental structure of the search space, if left without guidance, it would only crowd and complicate the space structure resulting in a harder space for most search algorithms. Last and in our view the most relevant case, is when modularity not only has an effect on search, but also module creation can be guided by problem domain knowledge. When this knowledge can be used to estimate the value of a module in terms of its contribution toward building the solution, then modularity is extremely effective. It is in this last case that creating high value modules or low value modules has a direct and decisive impact on performance.

We see the answering of these basic questions about the nature of modular search spaces as steps contributing toward the better understanding of evolutionary search in complex systems.

12.1 Limitations

Our approach is limited to linear structures. When translating linear genotypes into a linear phenotypes, the derivation itself can be a tree structure but we do not consider the case when phenotypes are tree structures themselves. We believe that an extension of a framework to accommodate phenotypic tree structures does not change our fundamental results such as invariance under strict module encapsulation, and the performance effects of the good and bad quality modules . The first part of this thesis focuses on the search space structure regardless of search strategy and regardless of problem domain. Because of its generality, it has limited prediction abilities. We focus only on the fundamental structural changes that can override the algorithm and problem domain factors. The second part of this thesis focuses on the OneMax problem. In this part, the theoretical and experimental results provided are subject to this domain. In order to extend this study to any other problem domain, two things must be considered: first the equations for composition and connectivity need to be adjusted. Second, a way to measure the quality of a module needs to be redefined.

12.2 Contributions

This thesis contributes to the field of machine learning and evolutionary computation in the following ways:

- It provides a theoretical framework to study modularity in search. This framework defines the representation space (genotype space), the search space (phenotype space), module defining production rules, and module creation transformations. This framework allow an explicit analysis of the effects of module encapsulation in evolutionary computation. It is based on the search space structure and is independent of the search algorithm used. Module creations and deletions are viewed as a search space transformations that in effect create nearly decomposable subsystems.

- Using this framework, it provides the following metrics to analyze changes in search space structure: reachability, redundancy, composition and connectivity. It provides a theoretical analysis that shows how these metrics change with module encapsulation transformations and provides experimental analysis to show that these metrics are strongly correlated with the search performance in genetic algorithms.

- It provides a No Free Lunch theorem for module creation at the representation level. This theorem states that systematically encapsulating lower level modules into higher level counterparts, by itself, does not benefit any search strategy, and provides proof of search space structure invariance under a particular class of module creation transformations. In other words, there are some module sets, the creation of which, do not change reachability, redundancy, composition or connectivity of the resulting search space. It provides an experimental analysis that validates this theoretical result.

- It provides experimental evidence of the existence of module creation transformations that do change search space reachability and redundancy, which result in a predictable increase or decrease in search performance. In addition, it provides experimental evidence that, at the general level, reachability and redundancy changes in random areas of the search space produce a detrimental effect in performance. At the particular problem level, reachability decrease of unfavorable areas of a search space or redundancy increase of favorable areas results in an improvement in performance.

- It provides a theoretical and experimental study describing the effects of the module creation transformation for three types of modules: good quality modules, bad quality modules and complete module sets. The quality of a module depends on its contribution towards building the solution. Under a set of assumptions, the following is established. Encapsulating a complete module set has a neutral effect on our metrics and on search performance. Encapsulating a good module is always advantageous and encapsulating a bad module is always detrimental in terms of our metrics and search performance.

LIST OF REFERENCES

[Alt05] Lee Altenberg. "Modularity in Evolution: Some Low-Level Questions." In Callebaut and Rasskin-Gutman [CR05]. foreword by Herbert A. Simon.

[QH366.2 .M63 2005.]

[AP92] 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.

[AP93] 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*, pp. 154–163, La Jolla, CA, USA, February 25-26 1993.

[AP94] P. J. Angeline and J. B. Pollack. "Coevolving high-level representations." In Christopher G. Langton, editor, *Artificial Life III*, volume XVII of *SFI Studies in the Sciences of Complexity*, pp. 55–71, Santa Fe, New Mexico, June 15-19 1994. Addison-Wesley.

[Bol00] Jessica A. Bolker. "Modularity in Development and Why It Matters to Evo-Devo." *Integr. Comp. Biol.*, **40**(5):770–776, 2000.

[Bon88] John T. Bonner. *The Evolution of Complexity by Means of Natural Selection.* Princeton University Press, Princeton, NJ, 1988.

[Bon02] J. C. Bongard. "Evolving modular genetic regulatory networks." In *Proceedings of the 2002 IEEE Conference on Evolutionary Computation (CEC2002)*, pp. 1872–1877, Piscataway, NJ, USA, 2002. IEEE Press.

[Bra99] Robert N. Brandon. "The Units of Selection Revisited: The Modules of Selection." *Biology and Philosophy*, **14**:167–180, 1999.

[Bra05] Robert N. Brandon. "Evolutionary Modules: Conceptual Analysis and Empirical Hypotheses." In Callebaut and Rasskin-Gutman [CR05]. foreword by Herbert A. Simon.

[QH366.2 .M63 2005.]

[CR05] Werner Callebaut and Diego Rasskin-Gutman, editors. *Modularity : understanding the development and evolution of natural complex systems.* The Vienna series in theoretical biology. MIT Press, Cambridge, Mass, 2005. foreword by Herbert A. Simon.

[QH366.2 .M63 2005.]

[Dar59] Charles Darwin. *The Origin of Species.* J.M. Dent and Sons Ltd, 1859.

[De 75] Kenneth De Jong. *Analysis of Behevior of a Class of Genetic Adaptive Systems.* PhD thesis, Technical University of Berlin, Germany, 1975.

[DO02] Edwin D. De Jong and T. Oates. "A coevolutionary approach to representation development." In *Proc. of the ICML-2002 WS on development of rep.*, p. 1, 2002.

[DTW04] E. D. De Jong, D. Thierens, and R. A. Watson. "Defining Modularity, Hierarchy, and Repetition." In *GECCO 2004 Workshop Proceedings*, Seattle, Washington, USA, July 26-30 2004.

[DU05] Peter Dauscher and Thomas Uthmann. "Self-Organized Modularization in Evolutionary Algorithms." *Evolutionary Computation*, **13**(3):303–328, 2005.

[ESS01] Marc Ebner, Mark Shackleton, and Rob Shipman. "How neutral networks influence evolvability." *Complex.*, **7**(2):19–33, 2001.

[FMV99] Koen Frenken, Luigi Marengo, and Marco Valente. "Interdependencies, nearly-decomposability and adaption." In Thomas Brenner, editor, *Computational Techniques for Modelling Learning in Economics*, Advances in computational economics ; v. 11, pp. 145–165. Kluwer Academic Publishers, 1999. ISBN 0792385039.

[http://www-ceel.gelso.unitn.it/papers/papero99_03.pdf.]

[Fog62] L.J. Fogel. "Autonomous Automata." *Industrial Research*, **4**:14–19, 1962.

[Fog64] L.J. Fogel. *On the Organization of Intellect.* PhD thesis, University of California at Los Angeles, 1964.

[GB02] Timothy G. W. Gordon and Peter J. Bentley. "Towards Development in Evolvable Hardware." In *EH '02: Proceedings of the 2002 NASA/DoD Conference on Evolvable Hardware (EH'02)*, p. 241, Washington, DC, USA, 2002. IEEE Computer Society.

[http://portal.acm.org/citation.cfm?id=787884&jmp=cit&coll=Portal&dl=GUIDE&CFID

[GGW03] O. O. Garibay, I. I. Garibay, and A. S. Wu. "The modular genetic algorithm: exploiting regularities in the problem space." In *Proc. of ISCIS 2003*, pp. 578–585, 2003.

[GGW04a] Ivan I. Garibay, Ozlem O. Garibay, and Annie 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*, pp. 1125–1137, 2004.

[GGW04b] Ozlem O. Garibay, Ivan I. Garibay, and Annie S. Wu. "No Free Lunch for Module Encapsulation." In *GECCO 2004 Workshop Proceedings*, Seattle, Washington, USA, July 26-30 2004.

[Gil06] Scott F. Gilbert. *Developmental Biology.* Sinauer Associates, Inc., 2006.

[GK01] John S. Gero and Vladimir Kazakov. "A Genetic Engineering Approach to Genetic Algorithms." *Evolutionary Computation*, **9**(1):71–92, 2001.

[GKD89] D.E. Golberg, B. Korb, and K. Deb. "Messy genetic algorithms: Motivation, analysis, and first results." *Complex Systems*, **3**(5):493–530, 1989.

[GNC01] 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*, pp. 1–10, 2001.

[http://www.csse.monash.edu.au/ dgreen/dggpubs05.htm.]

[Gol89] David E. Goldberg. *Genetic algorithms in search, optimization, and machine learning.* Addison Wesley, 1989.

[GW07] Ozlem O. Garibay and Annie S. Wu. "Analyzing the effects of module encapsulation on search space bias." In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pp. 1234–1241, New York, NY, USA, 2007. ACM Press.

[GW08] Ozlem O. Garibay and Annie S. Wu. "On the Effects of Modularity in Mutation Based Search." Manuscript submitted to Genetic Programming and Evolvable Machines Journal, 2008.

[GWG06] Ivan Garibay, Annie S. Wu, and Ozlem Garibay. "Emergence of genomic self-similarity in location independent representations: favoring positive correlations between the form and quality of candidate solutions." *Genetic Programming and Evolvable Machines*, **7**:55–80, 2006.

[Har97] Georges R. Harik. *Learning gene linkage to efficiently solve problems of bounded difficulty using genetic algorithms.* PhD thesis, University of Michigan, Ann Arbor, MI, 1997.

[HG96] Georges R. Harik and David E. Goldberg. "Learning Linkage." In *FOGA*, pp. 247–262, 1996.

[HLP01] Gregory S. Hornby, Hod Lipson, and Jordan B. Pollack. "Evolution of Generative Design Systems for Modular Physical Robots." In *IEEE International Conference on Robotics and Automation*, 2001.

[Hol62] John H. Holland. "Ouline for a logical theory of adaptive systems." *ACM*, **9**:297–314, 1962.

[Hol75] John H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI, 1975.

[Hor05] Gregory S. Hornby. "Measuring, enabling and comparing modularity, regularity and hierarchy in evolutionary design." In Hans-George Beyer and Una-May O'Reilly, editors, *GECCO '05: Proceedings of the 2005 genetic and evolutionary computation conference*, pp. 1729–1736, New York, NY, USA, 2005. ACM Press.

[HP01a] G. S. Hornby and J. B. Pollack. "The Advantages of Generative Grammatical Encodings for Physical Design." In Jong-Hwan Kim et al., editors, *Proceedings of the 2001 Congress on Evolutionary Computation: CEC2001*, pp. 600–607. IEEE Press, 2001. Digital Object Identifier 10.1109/CEC.2001.934446.

[HP01b] G. S. Hornby and J. B. Pollack. "Body-Brain Co-evolution Using L-systems as a Generative Encoding." In Lee Spector et al., editors, *GECCO '01: Proceedings of the 2001 genetic and evolutionary computation conference.* Morgan Kaufmann, 2001.

[HP02] G. S. Hornby and J. B. Pollack. "Creating High-Level Components with a Generative Representation for Body-Brain Evolution." In *Artificial Life 8*, pp. 223–246. MIT, 2002.

[HP03] G. S. Hornby and J. B. Pollack. "Generative Representations for the Automated Design of Modular Physical Robots." *IEEE Transactions on Robotics and Automation*, **19**(4):703–719, 2003.

[JJ06] Rieffel John and Pollack Jordan. "An Endosymbiotic Model for Modular Acquisition in Stochastic Developmental Systems." In *Proceedings of the Tenth International Conference on the Simulation and Synthesis of Living Systems (ALIFE X)*, 2006.

[http://www.demo.cs.brandeis.edu/papers/rieffel-alife-06.pdf.]

[Koz92] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, 1992.

[Koz94] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge, MA, 1994.

[Koz99] John R. Koza. *Genetic Programming III: darwinian invention and problem solving.* Morgan Kaufmann, San Francisco, CA, 1999.

[Koz03] John R. Koza. *Genetic Programming IV: routine human-competitive machine intelligence.* Kluwer Academic Publishers, Boston, 2003.

[KP03] Vladimír Kvasnicka and Jirí Pospíchal. “Emergence of Modularity in Genotype-Phenotype Mappings.” *Artificial Life*, **8**:295–310, 2003.

[KSK03] John R. Koza, Matthew Streeter, and Martin Keane. “Automated synthesis by means of Genetic Programming.” In *2003 AAAI Spring Symposium Series*, pp. 138–145, 2003.

[Lew78] R C. Lewontin. “Adaptation.” *Scientific America*, **239**(3):156–169, 1978.

[Lip04] H. Lipson. “Principles of Modularity, Regularity, and Hierarchy for Scalable Systems.” In *GECCO 2004 Workshop Proceedings*, Seattle, Washington, USA, July 26-30 2004.

[LPS01] Hod Lipson, Jordan B. Pollack, and Nam 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.

[LW66] A. Owens Lawrence J. Fogel and M. Walsh. *Artificial Intelligence through Simulated Evolution.* John Wiley & Sons., New York, NY, 1966.

[Mei04] Hans Meinhardt. “Pathways and building blocks.” *Nature*, **430**:970–970, 2004.
[http://www.nature.com/nature/journal/v430/n7003/pdf/430970a.pdf.]

[Mit98] Melanie Mitchell. *An Introduction to Genetic Algorithms.* The MIT Press, 1998.

[NM04] Aurora M. Nedelcu and Richard E. Michod. “Evolvability, Modularity, and Individuality During the Transition to Multicellularity in Volvocalean Green Algae.” In Schlosser and Wagner [SW04]. QH 491.M59 2004.
[http://eebweb.arizona.edu/michod/Downloads/Evolvability%20modularity%20final.pdf.]

[OR96] Una-May O'Reilly. "Investigating the Generality of Automatically Defined Functions." In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pp. 351–356, Stanford University, CA, USA, July 28-31 1996. MIT Press.

[OYR04] Una-May O'Reilly, Tina Yu, Rick Riolo, and Bill Worzel. *Genetic Programming Theory and Practice II.* Springer, Cambridge, Massachusetts, 2004.

[PD94] Mitchell A. Potter and Kenneth A. De Jong. "A Cooperative Coevolutionary Approach to Function Optimization." In *PPSN III: Proceedings of the International Conference on Evolutionary Computation. The Third Conference on Parallel Problem Solving from Nature*, pp. 249–257, London, UK, 1994. Springer-Verlag.

[http://cs.gmu.edu/ mpotter/pubs/.]

[PD00] Mitchell A. Potter and Kenneth A. De Jong. "Cooperative Coevolution: An Architecture for Evolving Coadapted Subcomponents." *Evolutionary Computation*, **8**(1):1–29, 2000.

[PD05] Elena Popovici and Kenneth De Jong. "Understanding cooperative coevolutionary dynamics via simple fitness landscapes." In *GECCO '05: Proceedings of the 2005 genetic and evolutionary computation conference*, pp. 507–514, New York, NY, USA, 2005. ACM Press.

[PG01] Martin Pelikan and David E. Goldberg. "Escaping Hierarchical Traps with Competent Genetic Algorithms." In Lee Spector et al., editors, *GECCO '01: Proceedings of the 2001 workshops on Genetic and evolutionary computation conference*, pp. 511–518, 2001.

[http://www.cs.umsl.edu/ pelikan/publications.html#2001003.]

[Raf96] Rudolf A. Raff, editor. *The Shape of Life.* The University of Chicago Press, 1996.

[RB94] 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.

[Rec65] I. Rechenberg, editor. *Cybernetic Solution Path of an Experimental Problem.* Translation 1122, Royal Aircraft Establishment Library, 1965.

[RG03] Franz Rothlauf and David E. Goldberg. "Redundant representations in evolutionary computation." *Evolutionary Computation*, **11**(4):381–415, 2003.

[Rot02] Franz Rothlauf, editor. *Representations for Genetic and Evolutionary Algorithms.* Physica-Verlag Heidelberg, Germany, 2002.

[SA61] Herbert A. Simon and Albert Ando. “Aggregation of Variables in Dynamic Systems.” *Econometrica*, **29**(2):111–138, October 1961.

[Sch02] Hans-Paul Schwefel. *Evolutionsstrategie und numerishe Optimierung.* PhD thesis, University of Michigan, Ann Arbor, 2002.

[Sch04] Gerhard Schlosser. “The Role of Modules in development and Evolution.” In Schlosser and Wagner [SW04], chapter 23. QH 491.M59 2004.

[ISBN:0-226-73855-8.]

[SFH03] Kisung Seo, Zhun Fan, Jianjun Hu, Erik D. Goodman, and Ronald C. Rosenberg. “Dense and Switched Modular Primitives for Bond Graph Model Design.” In Erick Cant-Paz et al., editors, *GECCO ’03: Proceedings of the 2003 genetic and evolutionary computation conference*, LNCS series, pp. 1764–1775. Springer-Verlag, 2003.

[Sim69] Herbert Simon. “The Architecture of Complexity.” In *The Sciences of the Artificial.* MIT Press, 1969.

[Sim05] Herbert Simon. “The structure of Complexity in an Evolveing World: The role of Near Decomposability.” In Callebaut and Rasskin-Gutman [CR05]. foreword by Herbert A. Simon.

[QH366.2 .M63 2005.]

[Spe02] Dan Sperber. “In Defense of massive modularity.” In Emmanuel Dupoux, editor, *Language, Brain, and Cognitive Development: Essays in Honor of Jacques Mehle*, pp. 47–57. The MIT Press, 2002.

[http://www.dan.sperber.com/modularity.htm.]

[Ste95] Kim Sterelny. “The Adapted Mind.” *Biology and Philosophy*, **10**(3):365–380, 1995.

[SW04] Gerhard Schlosser and Günter P. Wagner, editors. *Modularity in Development and Evolution.* The University of Chicago Press, 2004. QH 491.M59 2004.

[ISBN:0-226-73855-8.]

[Tob05] Lehrstuhl Fur Mikroelektronik Tobias Blickle, Lothar Thiele. “Genetic Programming and Redundancy.” In Jörn Hopf, editor, *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94)*, pp. 33–38, Max-Planck-Institut für Informatik (MPI-I-94-241), 2005.

[Tur48] Alan M. Turing. “Intelligent Machinery.” *National Physical Laboratory*, **5**:3–23, 1948.

[VML04] Evan A. Variano, Jonathan H. McCoy, and Hod Lipson. "Networks, Dynamics, and Modularity." *Physical Review Letters*, **92**(18):188701, 2004.

[WA96] Günter P. Wagner and Lee Altenberg. "Complex Adaptations and the Evolution of Evolvability." *Evolution*, **50**(3):967–976, 1996. http://dynamics.org/Altenberg/FILES/GunterLeeCAEE.pdf.

[Wag95] Günter P. Wagner. "Adaptation and the Modular Design of Organisms." In F. Morán, A. Morán, JJ. Merelo, and P. Chacón, editors, *Advances in Artificial Life*, volume 929, pp. 317–328. Springer Verlag, 1995. Third European Conference on Artificial Life, Granada, Spain, June 4 - 6, 1995 Proceedings http://pantheon.yale.edu/

[ISBN: 3-540-59496-5.]

[Wag96] Günter P. Wagner. "Homologues, natural kinds and the evolution of modularity." *American Zoologist*, **36**:36–43, 1996. citeseer.ist.psu.edu/wagner96homologues.html.

[Wag04] Günter P. Wagner. "The Role of Genetic Architecture Constraints in the origin of Variational Modularity." In Gerhard Schlosser and Günter P. Wagner, editors, *MOdularity in Development and Evolution*, pp. 338–358. The University of Chicago Press, 2004.

[ISBN: 0-226-73853-1.]

[Wat03] R. A. Watson. "Hierarchical Module Discovery." In *2003 AAAI Spring Symposium Series*, pp. 262–267, 2003.

[WG02] Annie S. Wu and Ivan Garibay. "The Proportional Genetic Algorithm: Gene Expression in a Genetic Algorithm." *Genetic Programming and Evolvable Machines*, **3**:157–192, 2002.

[WL95] Annie S. Wu and Robert K. Lindsay. "Empirical studies of the genetic algorithm with non-coding segments." *Evolutionary Computation*, **3**(2):121–147, 1995.

[WP05] Richard Watson and Jordan B. Pollack. "Modular Interdependency in Complex Dynamical Systems." *Artificial Life*, **11**:445–457, 2005.

[WP06] R. Paul Wiegand and Mitchell A. Potter. "Robustness in cooperative coevolution." In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pp. 369–376, New York, NY, USA, 2006. ACM.