T H E U N I V E R S I T Y O F T U L S A
THE GRADUATE SCHOOL

COLLECTIVE ADAPTATION:
THE SHARING OF BUILDING BLOCKS

by
Thomas Dunlop Haynes

A dissertation submitted in partial fulfillment of
the requirements for the degree of Doctor of Philosophy
in the Discipline of Computer Science
The Graduate School
The University of Tulsa
1998

T H E U N I V E R S I T Y O F T U L S A
THE GRADUATE SCHOOL

COLLECTIVE ADAPTATION:
THE SHARING OF BUILDING BLOCKS

by
Thomas Dunlop Haynes

A DISSERTATION
APPROVED FOR THE DISCIPLINE OF
COMPUTER SCIENCE

By Dissertation Committee

_____, Chairperson

_____

_____

_____

_____

COPYRIGHT STATEMENT

ABSTRACT

Haynes, Thomas Dunlop (Doctor of Philosophy in Computer Science)
Collective Adaptation: The Sharing of Building Blocks
(178 pp. Chapter XIII)
Directed by Professor Sandip Sen
(431 words)

Weak search heuristics utilize minimal domain knowledge during the search process. Genetic algorithms (GA) and genetic programming (GP) are population based weak search heuristics which represent candidate solutions as chromosomes. The Schemata Theorem forms the basis of the theory of how GAs process building blocks during the domain independent search for a solution to a given problem. Building blocks are templates describing subsets of the chromosome which have a small defining length and are highly fit. The main differences between typical GP and GA implementations are a variable length tree versus a fixed length linear string representation and a n-ary versus a binary alphabet. A consequence of the differences is that what constitutes a building block has been difficult to answer for GP and has led to theories that the Schemata Theorem does not hold for GP.

This thesis defines building blocks to be coding segments, which are those subsets of the chromosome that contribute fitness to the evaluation of the chromosome. Building blocks can be extracted from chromosomes and stored in a collective memory, which becomes a repository of partial solutions for both recently discovered building blocks and those discovered earlier. The contributions of this thesis are the mechanisms by which building blocks can be effectively shared both inside and outside chromosomes.

The duplication of building blocks inside a chromosome is shown to increase the exploratory power of the weak search heuristics. The perturbation of a candidate solution will affect one copy of the building blocks and if the fitness of the perturbed copy is not better than the original, the duplicate copies may still maintain the overall fitness of the chromosome. The duplication of coding segments is significant in finding better partial solutions with the following weak search heuristics: GP, GA, random search (RS), hill climbing (HC), and simulated annealing (SA). Each algorithm is systematically validated in the clique detection domain against a particular family of graphs, which have the properties that the set of partial solutions is known, the set of partial solutions is larger than viable chromosome lengths, and pruning algorithms are not effective.

Collective adaptation is the addition of the collective memory to the weak search heuristic. The solution no longer has to be found inside the chromosomes; the chromosomes can collectively contribute partial solutions such that the overall solution is formed inside the collective memory. Strong search heuristics can extend the partial solutions inside the collective memory and these partial solutions can be transfered back into the chromosomes. The thesis empirically demonstrates that collective adaptation finds significantly better partial solutions with weak search heuristics (GP, GA, RS, HC, and SA).

ACKNOWLEDGMENTS

The first year of my doctoral studies were supported by Roger Wainwright and his OCAST Grant AR2004. The second year of my doctoral studies were supported by a departmental teaching assistantship and Sandip Sen and his NSF Research Initiative Award IRI–9410180. The third year of my doctoral studies were supported in part by the Department of Mathematics and Computer Science, the University of Missouri, St. Louis, St. Louis, MO. The fourth year of my doctoral studies were supported in part by equipment provided by the Department of Computer Science, Wichita State University, Wichita, KS.

I would like to thank my committee members for the invaluable help they have provided: Roger Wainwright for getting me started in genetic programming, Dale Schoenefeld for both introducing me to Garvey and Johnson and opening up the door to the detection of cliques in a graph, Peyton Cook for making me reevaluate my feeling towards mathematics, and Theresa Shaft for reminding me that other fields have different expectations towards research. Both Peyton and Theresa were instrumental in my statistical analysis of my data, but all errors, if any, are my own.

I would also like to thank Sandip Sen for serving as my advisor. While Sandip prefers multiagent learning over genetic algorithms, we were still able to bridge the gap between the two.

My wife, Stacy Lynn Champaign Haynes, has been my main source of inspiration during my doctoral studies. Her presence in my life has fostered confidence

that I lacked before I knew her.

Finally there is my son, Morgan William Haynes, who has taught me that responsibility is full-time job. While he gives his love freely to me, I earn it by giving mine freely to him.

Parts of Section 1.3 and Section 2.3 appeared as Thomas Haynes, Dale Schoenefeld and Roger Wainwright, "Type Inheritance in Strongly Typed Genetic Programming", in Kenneth E. Kinnear, Jr. and Peter J. Angeline, editors, *Advances in Genetic Programming 2*, chapter 18, MIT Press, 1996.

Parts of Chapter 2 appeared as 1) Thomas Haynes, "Clique Detection via Genetic Programming", Technical Report UTULSA-MCS-95-02, The University of Tulsa, April 24, 1995; 2) Thomas Haynes and Dale Schoenefeld, "Clique Detection via Genetic Programming", Technical Report UTULSA-MCS-96-05, The University of Tulsa, March 15, 1996; and 3) Thomas Haynes and Dale Schoenefeld, "Clique Detection via Genetic Programming", In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Proceedings of the First Genetic Programming Conference*, 1996.

Parts of Chapter 3 appeared in Thomas Haynes, "Duplication of Coding Segments in Genetic Programming" in the *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, August, 1996. I want to thank Cory Hoelting for some discussions on this research in Chapter 3. I thank Justinian Rosca of the University of Rochester, Sandip Sen of the University of Tulsa, and Annie Wu of the University of Michigan for reviews of that paper. The comments supplied

by the anonymous referees were also appreciated. I also thank Mark Lindsay for allowing me access to workstations in his computer lab.

Parts of Chapter 4 and 5 appeared as 1) Thomas Haynes, "Collective Memory Search", in Barrett Bryant, Janice Carroll, Dave Oppenheim, Jim Hightower, and K. M. George, editors, *Proceedings of the 1997 ACM Symposium on Applied Computing*, 1997; 2) Thomas Haynes, "Augmenting Collective Adaptation with a Simple Process Agent", in Sandip Sen, editor, *AAAI Workshop on Multiagent Learning*, 1997; 3) Thomas Haynes, "On-line Adaptation of Search via Knowledge Reuse", in John R. Koza and Kalyanmoy Deb and Marco Dorigo and David B. Fogel and Max Garzon and Hitoshi Iba and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, 1997; 4) Thomas Haynes, "A Comparison of Random Search versus Genetic Programming as Engines for Collective Adaptation", in V. William Porto, editor, *Proceedings of the Seventh International Conference on Evolutionary Programming*, 1998.

Parts of Appendix G appeared as Thomas Haynes, "Phenotypical Building Blocks for Genetic Programming", in Thomas Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms*, 1997.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOTATIONAL CONVENTIONS

Several notational conventions are used throughout this dissertation to assist the reader. These conventions follow standard ones used in compiler construction.

A guide to the different notational conventions:

1. Functions are in boldface, **addition**.

2. Terminals are in italics, *19.27*.

3. Types are in the typewriter family, `Reals`.

INTRODUCTION

A genetic algorithm (GA) employs weak search to find near optimal solutions to problems in a given domain. A weak search heuristic utilizes minimal domain knowledge during the search process. In contrast, a strong search heuristic depends heavily on domain dependent knowledge to find a solution. While strong search methods can speed up the search process, the drawback of such an approach is a loss of portability across problem domains. Weak search methods, on the other-hand, can be readily extended from one domain to another.

When using GAs we represent a candidate solution to a problem in a structure called a chromosome. Each allele, or position, in a chromosome, contains symbols from a domain dependent language. The chromosome is evaluated by a domain dependent function to determine its fitness, which is a measure of the "goodness" of the solution represented in that chromosome. For example, consider a binary encoding of length 5 and an objective function, $f(c) = c^2$. Given $c = 01101$, we have $f(c) = 169$. We can perturb this chromosome to get a new chromosome. We could change the first bit in c to get $c_a^+ = 11101$ and $f(c_a^+) = 841$. If we assume that we are trying to maximize $f(c)$, then clearly c_a^+ is more preferable than c. However, since we know nothing about the domain, it should be just as likely that we changed the second and not the first bit in c, yielding $c_b^+ = 00101$ and $f(c_b^+) = 25$. Clearly, c is more preferred to c_b^+.

A fitness landscape is a plot of the fitnesses of all possible chromosomes. One

possible landscape is shown in Figure 0.1. The goal of search in a fitness landscape is to find the global optimum, which could be either a minima or maxima. Based on our knowledge of the fitness of selected structures, we can employ several heuristics to guide the search through the fitness landscape.

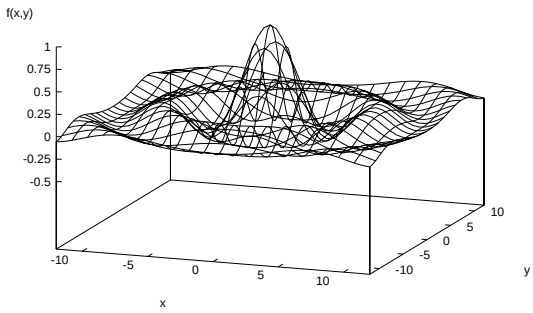

Figure 0.1: A fitness landscape.

With exhaustive search, we simply examine each possible structure and this process is guaranteed to find the global optimum. However, with large alphabets and chromosome lengths, such a search becomes unwieldy. In random search initial solutions are randomly generated and search moves randomly between structures in this space of solutions. Such a scheme jumps across the fitness landscape with the hope that with enough samples the global optimum will be reached. With infinite sampling, the global optimum will be found, but with finite sampling, there is no such guarantee.

Without a loss of generality, we can assume that we are trying to maximize fitness. We can then define a local optima to be a local maxima, which is a point in the fitness landscape such that its neighbors have the property that their fitness

is either less than or equal to its fitness. A global maxima is a point in the fitness landscape such that all other points have the property that their fitness is either less than or equal to its fitness. If we examine the fitness landscape in Figure 0.1, we see that it is "hilly"; i.e., there is an increasing gradient from less fit encodings to several local maxima.

The hill climbing heuristic takes advantage of this property by always following an increasing gradient. If $f(c^+) > f(c)$, then c^+ is selected as the current solution, else c is retained as the current solution. While it is not shown in the fitness landscape of Figure 0.1, sometimes it is advantageous to take a "side–step" if there is no increase in gradient, but several neighboring points have a fitness equal to the current point. We modify the algorithm such that if $f(c^+) >= f(c)$, then c^+ is selected as the current solution. A problem with hill climbing is that based on the initial randomly generated solution, the search might get stuck at a local optima. For example, in the fitness landscape of Figure 0.1, if the initial solution is in any of the four corners, the hill climbing algorithm quickly gets stuck at a local maxima. Indeed, unless the initial solution is already on the large hill at the center, the basic hill climbing algorithm can not find the global optimum for this problem.

Simulated annealing algorithms use a heuristic which allows for exploration away from local optimum. Like hill climbing, if $f(c^+) >= f(c)$, then c^+ is selected as the current solution. Otherwise, we accept the other solution as the current

solution with some probability,

$$p = e^{\frac{-|f(c^+)-f(c)|}{T}},$$

the worse solution. T is a parameter that models a decreasing temperature; as it decreases, the probability of accepting a worse solution also decreases.

The GA approach maintains a population of chromosomes. After each chromosome has been evaluated, reproduction, a domain independent process, drives the creation of new chromosomes for the next population of solutions. A cycle of evaluation and reproduction is called a generation. Chromosomes are selected to contribute to the new population based on their relative fitness; chromosomes which have higher fitness in a given generation are more likely to be selected. Reproduction by itself introduces no new solutions to the population of solutions. The basic GA algorithm employs two other operators, crossover and mutation, to change the solutions which have been selected via reproduction [Goldberg, 1989]. Instead of directly creating a population of k chromosomes, selection first chooses a pool of k parents with replacement from the current population. Two parents are then selected, without replacement, from this pool to exchange genetic material to form two children, which are formed via crossover, see Figure 0.2. An allele a_1 is selected in parent 1 and an allele a_2 is selected from parent 2. One child contains the material to the right of and up to a_1 from parent 1 and the material from the left of a_2 from parent 2. The other child contains the material to the right of and up to a_2 from parent 2 and the material from the left of the

a_1 from parent 1. After the new population has been formed from these children, the mutation operator can be applied; one or more symbols in the chromosome are randomly changed into new symbols from the language. The average fitness of the chromosomes per generation is likely to increase and over time the system converges to a "good", i.e., close to optimal, solution. The repetitive application of evaluation and selection has been shown to efficiently solve a variety of problems in many different domains [Davis, 1991; Goldberg, 1989; Goldberg, 1994; Koza, 1992].

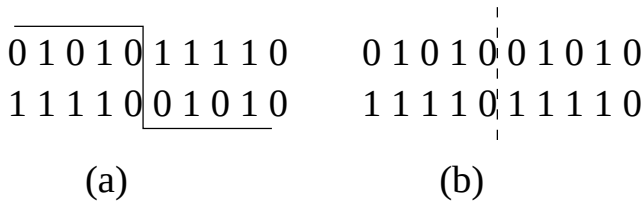

Figure 0.2: Single point crossover between two chromosomes.

The canonical GA chromosome, or string representation, utilizes a binary alphabet: $\{0, 1\}$. A schema (plural schemata) is a template describing subsets of strings within the string. If a `don't care` symbol, i.e., a symbol matching either 0 or 1, is utilized, we have the schemata alphabet $\{0, 1, *\}$. For example, the schema $s =$ `**0**1**` describes all strings that have a 0 in the third position and a 1 in the sixth. The *order* of a schema is the number of 0's and 1's present in the template ($o(s) = 2$). The *defining length* of a schema is the distance between the outermost non–`don't care` positions in the schema ($\delta(s) = 3$). *Building blocks* have a small defining length and are highly fit. By having a small defining length, building blocks are not as susceptible to destructive crossover, i.e., crossover which

disrupts the schema, as are schemata with larger defining lengths[1]. Being highly fit, once instantiated in the population, they are more than likely to increase their number of appearances in the population. The Schema Theorem relates how building blocks are combined to form better solutions over time [Holland, 1975; Goldberg, 1989].

Genetic programming (GP) is an off–shoot of GA research and the initial "basic theory" describing the operation of GP was borrowed from the GA Schema Theorem [Koza, 1992]. While GP utilizes the basic GA algorithm and also has the concepts of crossover and mutation, the canonical GP chromosome representation is a parse tree (S–expression) and the alphabet is n-ary. As seen in Figure 0.3, when two parents are selected for crossover, two nodes are randomly selected in each tree and these nodes form the roots of two subtrees which are exchanged between the parents. From Figure 0.3, we see that GP chromosomes do not have a fixed length like the GA chromosomes and that the same subtree structure may appear twice in the same tree (Subtrees labeled B and C in Child2 of Figure 0.3(b).).

If we examine the definition of a building block, we see that it is a highly fit schema which has a short defining length. For the parent chromosomes of Figure 0.3, a schema with short defining length is

$$H_1 \;=\; \{(IFTE \;\; * \;\; * \;\; (+ \;\; 6 \;\; 9))\}^2,$$

[1]Section 3.3 provides an overview of this process.

[2]Notice the difference between the schemata for the GA and the GP: in the GA schemata, we must denote all of the fixed positions, e.g., $s =$ `**0**1**`, whereas in GP schemata, we do

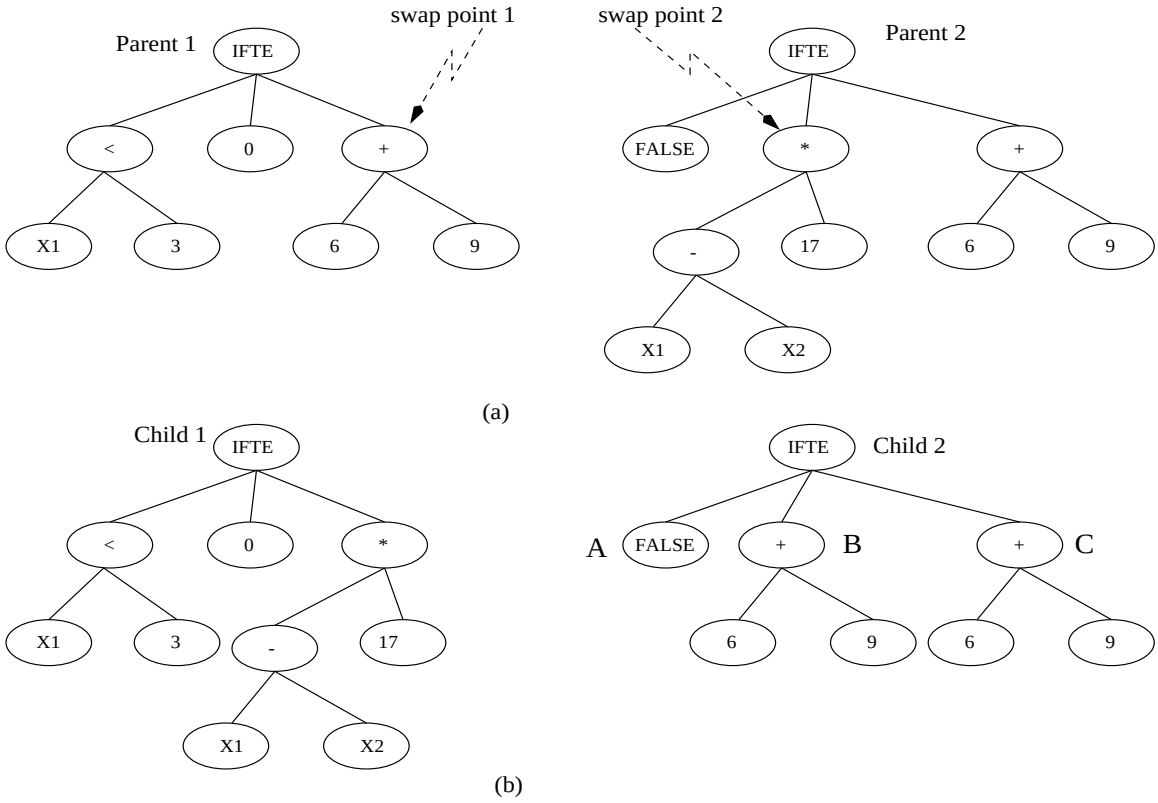

Figure 0.3: Subtree swap crossover between two chromosomes.

i.e., the function **IFTE**[3] which has three arguments with the third being instantiated as the subtree (+ 6 9). Is this schema highly fit? I claim it is highly fit; it appears in both chromosomes and it accounts for all the fitness awarded to Parent 2. However, the building block does not appear in Child 1 and appears once in Child 2. The schema

$$H_2 \;=\; \{(+\;\;6\;\;9)\}$$

not denote all possible combinations, i.e., each schema represents a rooted subtree which may appear anywhere in the chromosome.

[3]If the first subtree evaluates to *TRUE*, then return the evaluation of the second subtree, else return the evaluation of the third subtree.

is also a building block; it appears once in both parents, once in Child 1 and twice in Child 2. The Schema Theorem handles the first case but not the second case: during GA crossover, a schema can be disrupted in one or both children, but it will not appear twice in either child.

Also, if we examine Child 2, we see that the subtree labeled B will not be evaluated: the first argument to **IFTE** is *FALSE* and thus the subtree at C is evaluated. The subtree at B is defined to be a non–coding segment because its evaluation does not contribute either positively or negatively to the fitness evaluation of the *current* chromosome. The subtree at A is defined to be a coding segment because its evaluation can contribute either positively or negatively to the fitness evaluation of the *current* chromosome. The evaluation of a subtree is highly contextual.

Is the schema

$$H_3 \;=\; \{(IFTE \;\; * \;\; (+ \;\; 6 \;\; 9) \;\; *)\}$$

highly fit? In keeping with the earlier decision, it must be. For Child 2, we could change the value of node A from *FALSE* to *TRUE* and it would not change the fitness evaluation of the chromosome. But how will the GA Schema Theorem account for the multiple appearances of building blocks such as the schema H_2 in Child 2? Even though the first appearance does not contribute to the fitness of the current chromosome, it could be expressed in a descendant which did not

already have it. Also, are the following schemata equivalent:

$$\{(+\ 9\ 6)\} \equiv \{(+\ 6\ 9)\}?$$

The inability of the Schema Theorem to account for the multiple appearance of subtrees, i.e., building blocks, and their hierarchical recombination has led researchers to believe that the Schema Theorem does not hold for GP [O'Reilly, 1995; O'Reilly and Oppacher, 1995b]. Altenberg believes the Schema Theorem can not account for the proliferation of copies of subtrees and he applies Price's Theorem and introduces a "constructional fitness" to account for such proliferation [Altenberg, 1994]. The key to understanding constructional fitness is in his redefinition of a building block; a building block is not necessarily highly fit, instead it is a block which has a higher probability of increasing fitness in a child chromosome. Thus a block is not a building block because of its contribution to the current chromosome, but rather because of its potential contribution to descendants of the chromosome. The distinction is subtle, but critical for understanding the characteristics of building blocks in GP chromosomes.

GP building blocks are difficult to represent, are difficult to incorporate into the Schema Theorem, and are difficult to define. The common definition is based on the empirical observation of multiple copies of subtrees in the chromosome. I utilize a more concise definition of GP building blocks: they are those coding segments which contribute positively to the fitness of the chromosome. A building block can only be detected if and only if its fitness can be determined apart from

that of the chromosome. This definition removes the contextual requirements for determining if a given subtree is a building block. Indeed, a building block can be extracted from the chromosome such that it stands on its own and potentially it could be inserted into another chromosome.

I apply my definition of a building block to the detection of cliques in a graph. In the clique domain, the problem is given a graph G, to determine either the maximal complete subgraph of G with the highest cardinality, max clique (MC), or all maximal complete subgraphs of G, clique cover (CC) (Both problems are NP-complete.). A property of finding a maximal clique is that it is comprised of complete subgraphs. Furthermore, if we define a candidate clique as a complete subgraph, which may or may not be maximal, then for a clique of cardinality n, there are C_k^n candidate cliques of cardinality k. I expect to detect building blocks in the clique domain because it is an example of a domain for which there is data decomposition; the solution may be broken into sub–solutions. The unique candidate cliques in the chromosome form sub–solutions for cliques and as such they may be integrated to form the clique cover for a given graph.

The contributions of this dissertation are the sharing of building blocks both inside and outside of chromosomes. I find that by isolating coding segments, by increasing the probability they will combine to form larger segments, and by additionally allowing them to be combined outside of the chromosomes, I can complement the weak search heuristics such that better partial solutions can be found.

I validate my contributions by systematically investigating new techniques for the sharing of building blocks. My approach is geared in two phases: first I show the applicability of each technique against a single graph and then I analyze the results from testing the technique against the FC family of graphs. The graphs I utilize for testing the applicability of the techniques are either a hand–crafted graph of slight complexity or a graph with more complexity from a testbed [Johnson and Trick, 1996]. Each algorithm is systematically validated against the FC family of graphs, which have the properties that the set of partial solutions is known, the set of partial solutions is larger than viable chromosome lengths, and pruning algorithms are not effective.

I explore the sharing of building blocks inside the chromosome via the duplication of coding segments and outside of the chromosome via collective adaptation:

Duplication of coding segments: By determining all of the building blocks inside a chromosome, I also determine all of the non–coding segments inside that chromosome. From the previous example, we know that a non–coding segment might be a building block, e.g., if the subtree at B were evaluated, then it would contribute fitness to the evaluation of the chromosome. However, with the definition of a building block that I have adopted, I necessarily know that non–coding segments are *not* building blocks.

I can then replace, or repair, the non–coding segments with copies of the coding segments. (This process entails a mapping back from the phenotype to the genotype, which may not be an easy process as many genotypes may

evaluate to the same phenotype.) In the first part of this dissertation, I empirically demonstrate the effectiveness of varying both the rate of repair of the chromosome and the number of duplicates that are inserted into the chromosome. The weak search heuristics are able to find either the solution or better partial solution than can be found without the duplication of coding segments.

A previous conjecture in the literature was that non–coding segments only provided protection against destructive crossover. I show that the non–coding segments also provide a natural back–up mechanism of the material in the coding segments. The chromosome can explore, via crossover or mutation, with the original copy of the coding segments and keep a backup version for backtracking.

Collective adaptation: As I can detect and extract building blocks from one chromosome and the evaluation of a building block is independent of the chromosome, I can utilize the building blocks from any chromosome to replace non–coding segments in any other one. Furthermore, I need not restrict that only chromosomes in a given generation may contribute building blocks for that generation. In the second part of this dissertation, I systematically examine collective adaptation, which is the integration of building blocks via collective memory.

The collating of partial solutions is significant in piecing together the solution. Furthermore, I can extend the effect of collective adaptation by

allowing strong search heuristics to engage in local search within the collective memory. While this search heuristics prove to be intractable in the original search space, they are effective in the search space of partial solutions contained within the collective memory. I can also extend the search capability by transferring partial solutions back from the collective memory into the chromosomes. Such a transfer can refocus the neighborhood of search being examined by the weak search heuristics.

By sharing building blocks both inside and outside of the chromosome, I extend the processing power of both GP and GA. I also show that both techniques can be successfully applied to other weak search heuristics: random search, hill climbing, and simulated annealing. While the results I present are specific to the clique domain, my results hold in general; the theory of NP–completeness states there exists a polynomial time mapping from one NP–complete problem to any other. There are three basic NP-complete problems one can consider while detecting cliques in a graph [Garey and Johnson, 1979] (pages 193–194):

Partition into cliques: Given that $G = (V, E)$ and a positive integer $K \leq V$, can the vertices of G can be partitioned into $k \leq K$ disjoint sets $V_1, V_2, \ldots, V_k$ such that, for $1 \leq i \leq k$, the subgraph induced by V_i is a complete subgraph? *Or, can we partition the clique cover of G such that each clique does not share an edge with any other clique?*

Covering by cliques: Given that $G = (V, E)$ and a positive integer $K \leq E$, are there are $k \leq K$ subsets $V_1, V_2, \ldots, V_k$ of V such that each V_i induces

a complete subgraph of G and such that for each edge $\{u, v\} \in E$ there is some V_i that contains both u and v? *Or, can we determine all cliques of G?*

Clique: Given that $G = (V, E)$ and a positive integer $K \leq V$, does G contains a clique of size K or more, i.e., a subset $V' \subseteq V$ with $|V'| \geq K$ such that every two vertices in V' are joined by an edge in E? *Or, can we find the maximal cardinality clique of G?*

I can map any NP–complete problem into one of these three problems from the clique domain and then utilize duplication of coding segments (internal sharing of building blocks) and collective adaptation (external sharing of building blocks) to enhance the search for a solution.

The rest of this dissertation is laid out as follows: Chapter 1 is an overview of genetic programming, strongly typed genetic programming, and type inheritance. Chapter 2 introduces the clique detection problem and details how type inheritance is utilized to both allow collections of partial solutions for clique detection in a graph and to reduce the search space of possible encodings of partial solutions. Chapter 3 presents experimentation into the duplication of coding segments in a GP chromosome. Chapter 4 defines collective adaptation for a society of computational search agents. Chapter 5 presents experimentation into collective adaptation with a GP search heuristic. Chapter 6 validates the GP experiments with a family of graphs and also examines various other search heuristics as engines for collective adaptation.

CHAPTER I

Genetic Programming

1.1 Introduction

Holland's work on adaptive systems produced a class of biologically inspired algorithms known as genetic algorithms (GAs) that can manipulate and develop solutions to optimization, learning, and other types of problems [Holland, 1975]. In order for GAs to be effective, the candidate solutions should be represented as n–ary strings. Though GAs are not guaranteed to find optimal solutions, they still possess some nice provable properties (optimal allocation of trials to substrings, evaluating exponential number of schemas with linear number of string evaluations, etc.), and have been found to be useful in a number of practical applications [Davis, 1991].

Koza's work on genetic programming (GP) was motivated by the representational constraint, i.e., fixed length encodings, in traditional GAs [Koza, 1992]. His claim is that a large number of apparently dissimilar problems in artificial intelligence, symbolic processing, optimal control, automatic programming, empirical discovery, machine learning, etc., can be reformulated as the search for a computer program that produces the correct input–output mapping in any of these domains. To facilitate this search, he uses the traditional GA operators for selection and recombination of individuals from a population of structures, and

applies the operators on structures represented in a more expressive language than used in traditional GAs. The representation languages used in GPs are computer programs represented as Lisp S–expressions. GPs have attracted a large number of researchers because of the wide range of applicability of this paradigm, and the easily interpretable form of the solutions that are produced by these algorithms [Kinnear, Jr., 1994a; Koza, 1994; Angeline and Kinnear, Jr., 1996]. We assume the reader is familiar with the fundamentals of GAs and GPs.

A GP system is primarily comprised of three main parts:

- A population of chromosomes.
- A chromosome evaluator.
- A selection and recombination mechanism.

In implementing the system for a new problem domain, the designer must encode function and terminal sets, which will comprise the elements or genes of the chromosome, and implement a function which can evaluate the fitness, or applicability, of a chromosome in the domain.

Chromosomes are typically represented as parse trees. The interior nodes are functions and the leaf nodes are terminals. The first population of chromosomes is randomly generated. Each chromosome is then evaluated against a domain specific fitness function. The next generation is comprised of the off-spring of the current generation: parents are randomly selected in proportion to their fitness evaluation and create the children by exchanging subtrees during the crossover

process. Thus, more fit chromosomes are likely to contribute genetic material to successive generations. This generational process is then repeated until either a preset number of generations has passed or the population converges.

Two considerations for designing the function and terminal sets are *closure* and *sufficiency*. Closure requires all of the functions to accept arguments of a single data type (i.e., a float) and return values of that same data type. A consequence is that all functions must return values that can be used as arguments for any other function. Hence, closure entails any element can be a child node in a parse tree for any other element without having conflicting data types. An example of closure is given in the prefix expression **div** *Ralph 0*. Not only must the division operator handle division by *0*, it must also convert *Ralph* into a numeric value. Sufficiency requires that the domain be solvable with the given function and terminal sets.

A problem with using GP to solve large and complex problems is the considerable size of the state–space to be searched for generating good solutions. Even for small terminal and function sets and tree depths, search spaces of the order of $10^{30} - 10^{40}$ are not uncommon [Montana, 1995]. To address this pressing problem, researchers have been investigating various means to reduce the GP state–space size for complex problems. Notable work in this area include Automatically Defined Functions (ADF) [Kinnear, Jr., 1994b; Koza, 1994], module acquisition (MA) [Angeline, 1994; Kinnear, Jr., 1994b], and strongly typed genetic programming (STGP) [Montana, 1995]. The first two methods utilize function decomposition to reduce the state–space. The STGP method utilizes structuring of

the GP S-expression to reduce the state–space. We strongly agree with Montana's claim of the relative advantage of STGP over GP for complex problems [Montana, 1995].

1.2 Strongly Typed Genetic Programming

Montana claims that closure is a serious limitation to genetic programming [Montana, 1995]. Koza describes a way to relax the closure constraint using the concept of *constrained syntax structures* [Koza, 1992]. Koza used tree generation routines which only generated legal trees. He also only used operations on the parse trees which maintained legal syntactic structures.

Maintaining legal syntactic structures is at the heart of STGP. In STGP, variables, constants, arguments, and returned values can be of any type. The only restriction is that the data type for each element be specified beforehand. This causes the initialization process and the various genetic operations to only construct syntactically correct trees. A benefit of syntactically correct trees is that the search space is reduced. This has been shown to decrease the search time [Montana, 1995; Haynes *et al.*, 1995b].

One of the key concepts for STGP is the generic function, which is a mechanism for defining a class of functions, and defining generic data types for these functions. Generic functions eliminate the need to specify multiple functions which perform the same operation on different types. For example, one can specify a single generic function, **VECTOR–ADD**, that can handle vectors of different

dimensions, instead of multiple functions to accommodate vectors for each dimension. Specifying a set of argument types, and the resulting return type, for a generic function is called *instantiating* the generic function.

The STGP search space is the set of all legal parse trees. That is, all of the functions have the correct number of parameters of the correct type. Generally the parse tree is limited to some maximum depth. The maximum depth limit on a parse tree is one of the GP parameters. This keeps the search space finite and manageable. It also prevents trees from growing to an extremely large size.

Montana presented several different examples illustrating these concepts. STGP was used to solve a wide variety of moderately complex problems involving multiple data types. The examples showed that STGP was very effective in obtaining solutions to the problems compared to GP. Montana lists three advantages of STGP and generic functions:

1. Generic data types eliminate operations which are legal for some sets of data used to evaluate performance, but which are illegal for other possible sets of data.

2. When generic data types are used, the functions that are learned during the genetic programming process are generic functions.

3. STGP eliminates certain combinations of operations. Hence it necessarily reduces the size of the search space. In many cases the reduction is a significant factor.

One of Montana's examples presents a problem with a terminal set of size two,

and a function set of size 10. When the maximum tree depth was restricted to five, the size of the search space for the STGP implementation was 10^5, while the size of the GP search space was 10^{19}. In the same example when the maximum tree depth was increased to six, the size of the search space for the STGP implementation was 10^{11}, while the size of the GP search space was 10^{38} [Montana, 1995].

It has been shown that STGP can significantly reduce the search space. The STGP variant mainly restricts the construction and reproduction of chromosomes; the basic algorithm is GP. Thus, unless we are explicitly dealing with strong typing, we will utilize the term GP to refer to both untyped GP and STGP.

1.3 Type Inheritance

Strong typing is used to restrict the search space considered in the genetic programming paradigm. This is shown in both the paper in which Montana introduced STGP [Montana, 1995] and in our research into multiagent behavioral strategies [Haynes *et al.*, 1995b]. STGP is able to reduce the search space by only allowing syntactically correct programs to be generated and produced by the crossover and mutation operators. Montana types both the function return value and the arguments, and requires that the typing restrictions be honored by all operations on the S–expressions. Due to closure, standard GP has a "flat" type space of only one level, but generic functions allow STGP to have two levels of typing. We extend STGP by allowing a type hierarchy, which allows more than two levels of typing. In order to allow for a minimal function set, generic types

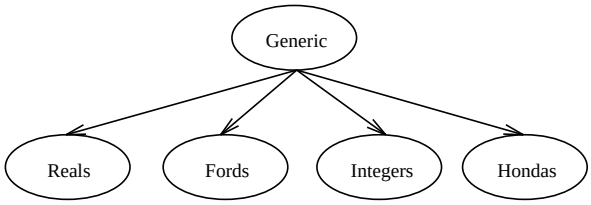

Figure 1.1: An example STGP type hierarchy.

are introduced. Generic types must be instantiated during node construction.

In effect, generic types allow for a two level type hierarchy, as shown in Figure 1.1. From object oriented programming, we know that it can be desirable to have more than two levels in a hierarchy. A simple example involving cars and numbers illustrates this desire. If we consider the class hierarchy shown in Figure 1.2, and assume the standard arithmetic operators of **addition**, **subtraction**, **division**, and **multiplication**, then we do not want to have specialized versions of these operators for `Reals` and `Integers`. The standard typing solution is to have a generic function for each of the operators, which can handle any type.

However, to type **addition** as `Generic`, we would have to ensure that **addition** is overloaded such that it makes sense in all contexts which can be instantiated from `Generic`. Failure to do so leads to the undesirable result that the program in Figure 1.3 is valid. What does it mean to add two `Fords`? Are we simply counting the cars, by type, that pass us on the highway? Or are we trying to add the qualities of one car to another to get a hybrid?

We would like to define the operator **addition** to be only valid in the class `Numbers`, appropriately overload it in class `Reals` and class `Integers`, and force the program in Figure 1.3 to be invalid. This redefinition further restricts the

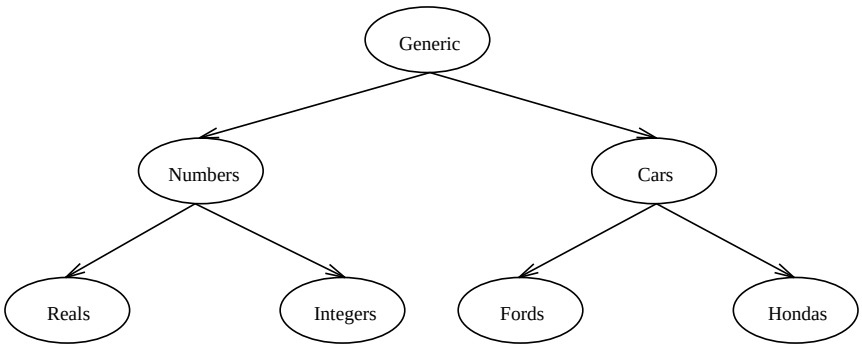

Figure 1.2: Example of a simple class/type hierarchy.

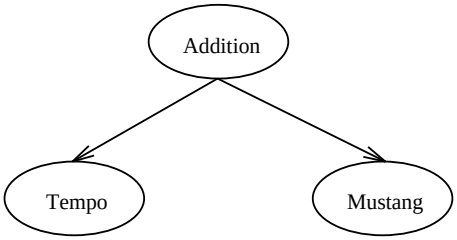

Figure 1.3: Addition of two `Fords`, which is not allowed.

allowed inputs while still reducing the total number of functions. We are extending the concept of a generic type for the tree to generic types for subtrees.

A generic type can be thought as a variable for types and can be instantiated with any of the other allowable types. This property requires that the `Generic` type be the root node of the type tree, as is shown in both Figure 1.1 and Figure 1.2. The extension we present in this chapter is to allow multiple levels of generic types, i.e., the `Numbers` and `Cars` nodes in Figure 1.2. We derived the term *type hierarchy* from the fact that in the tree form of the class hierarchy, as shown in Figure 1.2, the generic types are inherited just like the other class properties. Thus class hierarchies illustrate the inheritance of type.

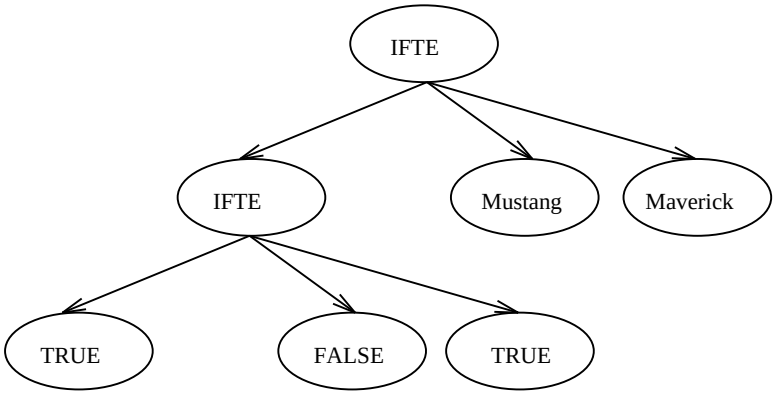

Figure 1.4: Instantiation of IFTE.

1.3.1 Generic Functionality

A classical example of a generic function is the **IFTE** function, which evaluates and returns its second argument if the first argument evaluates to true, otherwise it evaluates and returns the third argument. It can be typed as

$$\texttt{Generic}\ \textbf{IFTE}(\texttt{Boolean}\ A,\ \texttt{Generic}\ B,\ \texttt{Generic}\ C),$$

which allows **IFTE** to be reused by any type. Note that once B is instantiated, then both C and the return of **IFTE** must be instantiated to the same type.

Figure 1.4 illustrates the instantiation of the **IFTE** operator. The **IFTE** at the root of the tree has a return type of `Ford`, and the second **IFTE** has a return type of `Boolean`. Notice that each of the respective return types is typed the same as the respective B and C arguments.

The STGP algorithm differs from GP in that typed–based restrictions must be enforced at three points: initial tree generation, mutation, and crossover. As trees are generated, only child nodes with a return type matching the argument type

of the parent node can be instantiated into the tree. Also, even if a child node is type compatible with an argument, there has to be a check to make sure that the subtree represented by the child node does not violate the maximum depth restriction. A type table, showing valid types per depth level, is generated *a priori* to provide quick lookup to determine if this restriction has been violated. The expected type for the root node is an input parameter and domain dependent. It is during tree generation that generic functions are instantiated. Note that the standard mutation operator is a specialized case of tree generation.

During the crossover process, the return value type of the two subtrees selected for exchange must be tested to see if they are of the same type and if the resulting trees violate depth restrictions. If either check fails, then two new subtrees are selected. If, after a finite number of selections, no valid crossover points are found, then the two parent trees are copied into the pool for the next generation. At this point there can not be any generic types in the S–expressions, as they must have been instantiated.

1.3.2 Modifying STGP

Abstract classes do not contain any instantiated objects in the class. An example of this is the `Generic` class of STGP. In STGP, only one abstract class is allowed. We propose to allow multiple abstract classes. A concrete class is one in which a class can have objects instantiated in the class. Note that all concrete classes can be made abstract by simply adding another class under the concrete class, and putting all objects that were originally in the concrete class into this new

subclass.

The distinction can be illustrated by considering the `Cars` type in Figure 1.2:

1. Do we allow an instantiation of `Cars` to be the set of all automobiles minus the sets of `Fords` and `Hondas`?

2. Or is `Cars` the set of all automobiles including `Fords` and `Hondas`?

3. Or are there no other automobiles other than `Fords` and `Hondas`?

The last item reflects an abstract base class, and the first two are cases of concrete base classes.

An STGP algorithm which is modified for type inheritance has to perform additional checks during tree generation, crossover and mutation. During tree generation, base classes (types), both abstract and concrete, will also have to be instantiated. Abstract base classes do not have any objects that can be instantiated themselves, instead they serve as place-holders for collections of objects.

We construct a type tree to facilitate this checking. An example type tree is shown in Figure 1.2. Using the subtype principle, wherever a type may appear a descendant of it in the type tree may also appear. Following the example of generic functions, once an argument is instantiated to a specific type, then the remaining arguments are also typed to that instantiation. Thus an additional check must be performed to determine if a subtype is allowed.

Type trees may be considered by two cases: *non–branching* and *branching*. In a non–branching type tree the tree has a branching factor of 1. This gives a linear

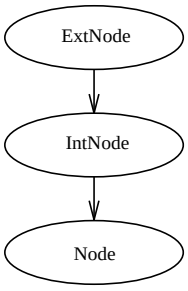

Figure 1.5: Non–branching type hierarchy for the clique detector, presented in Chapter 2.

type tree, as shown in Figure 1.5. A branching type tree has at least one node with a branching factor ≥ 2.

A non–branching type tree, as in Figure 1.5, is relatively simple to implement. Difficulties arise when we consider a branching type tree, as in Figure 1.2. In particular, assume we have the type tree shown in Figure 1.6, with a root type `TP` (Type Parent) which has left and right subtypes `STLC` (Sub–Type Left Child) and `STRC` (Sub–Type Right Child) respectively. If we have a function

$$\texttt{TP} \ \mathcal{F} \ (\texttt{TP} \ , \texttt{TP}),$$

and the first argument is typed as `STLC`, can the second be instantiated to type `STRC`? Using the subtype principle, this is permissible. But it is possible that `STLC` and `STRC` have some distinguishing characteristics that do not permit all relationships between them to be expressed.

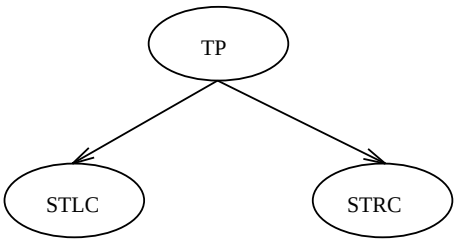

Figure 1.6: Example branching type hierarchy.

1.4 Conclusions

The generic functions of Montana's STGP do not allow for type inheritance. I have extended STGP to allow for type inheritance and in the next chapter I will utilize type inheritance to encode candidate cliques in a GP chromosome.

CHAPTER II

Clique Detection

2.1 Introduction

A collection of cliques in a graph can be represented as a list of a list of vertices which, in turn, can be represented by a tree structure. Given a graph $G = (V, E)$ a clique of G is a complete subgraph of G. We denote a clique by the set of vertices in the complete subgraph. As the subgraph of G induced by any subset of the vertices of a complete subgraph of G is also complete, it is sufficient to investigate the maximal complete subgraphs of G, i.e., the maximal cliques. We consider two NP–complete problems, finding either the maximum clique, max clique (MC), of G or the set of all cliques, clique cover (CC), of G. Furthermore, we define a candidate clique to be a complete subgraph which may or may not be maximal.

Each chromosome in the population represents a set of candidate maximal cliques. The function and terminal sets are $F = \{$**ExtCon**, **IntCon**$\}$ and $T = \{$*1,...,#vertices*$\}$. **ExtCon** "separates" two candidate maximal cliques, while **IntCon** "joins" two candidate cliques to create a larger candidate. Strong typing [Montana, 1995] and type inheritance [Haynes *et al.*, 1996b] are used to ensure that the parent of an **ExtCon** node is either the root or another **ExtCon** node.

The fitness evaluation rewards for clique size and rewards for the number of cliques in the tree. To gather the maximal complete subgraphs, the reward for

size is greater than that for numbers. The evaluation also does not reward for a clique either being in the tree twice or being subsumed by another clique. The first falsely inflates the fitness of the individual, while the second invalidates the goals of the problem. The algorithm for the fitness evaluation is:

- Parse the chromosome into a sequence of candidate subgraphs, each represented by an ordered list of vertex labels.
- Throw away any candidate subgraphs which duplicate any of the vertex labels.
- Throw away any candidate subgraphs that are not complete, which leaves only candidate cliques.
- Throw away any duplicate candidate cliques and any candidate cliques that are subsumed by other candidate cliques.

The fitness formula is

$$F = \alpha c + \frac{\sum_{i=1}^{c} \beta^{n_i}}{\beta^{\gamma * inc_{max}}},$$

where c is the number of valid candidate maximal cliques, n_i is the number of vertices in $clique_i$, inc_{max} is the maximum edges incident to any of the vertices, and α, β, and γ are all user defined parameters. γ is a scaling factor to allow fitness values to stay within the range of a C double.

β has to be large enough so that a large clique contributes more to the fitness of one S–expression than a collection of proper sub-cliques contributes to the fitness

of a different S–expression. For example, consider the graph in Figure 2.1. It is clear that there is only one maximal clique:

$$C_4 = \{1, 2, 3, 4\}.$$

However, there are four sub-cliques of cardinality three:

$$C_3 = \{\{1, 2, 3\}, \{1, 2, 4\}, \{1, 3, 4\}, \{2, 3, 4\}\}.$$

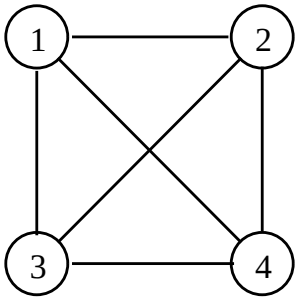

Figure 2.1: Example four node graph.

The various resultant fitnesses, as β is varied, $\gamma = 0$, and α is held constant, are shown in Figure 2.2. It is not until β is larger than the cardinality of the maximum clique that the desired result is found, see Table 2.1. Simply put, with the current fitness function, β must be chosen to respect the cardinality of maximum cliques in a graph. With other choices for α and β, our fitness function is more suited to determining the largest maximal clique in a graph, rather than the set of all maximal cliques.

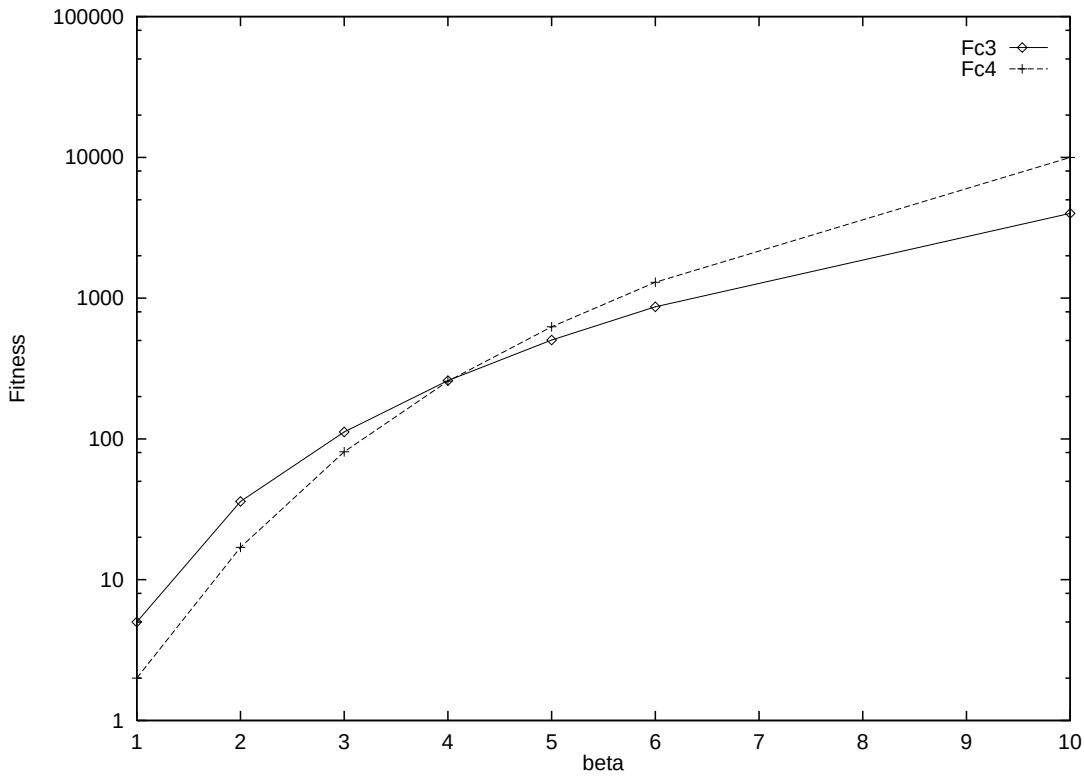

Figure 2.2: Fitness for four node graph. Fc_3 **has** $\beta = 3$ **and** Fc_4 **has** $\beta = 4$**. The y-axis is logarithmic.**

2.2 Prior Encodings

Various approaches have been taken within the GA community for clique detection [Bui and Eppley, 1995; Soule *et al.*, 1996; Soule and Foster, 1997]. Bui and Eppley used a binary encoding to solve for the max clique. Each position represented a vertex label from the graph and a '1' indicated that the vertex was present in the max clique. Soule *et al.* point out that such an approach fails to exploit building blocks. If two labels are connected, but not close together on the chromosome, then the defining length will not be short, resulting in a greater chance of disruption of the building block [Soule *et al.*, 1996].

GP has also been used to find the max clique [Soule *et al.*, 1996]. One function

α	β	F_{C_4}	F_{C_3}
1	1	2	5
1	2	17	36
1	3	81	112
1	4	257	260
1	5	626	504
1	6	1297	868
1	10	101001	4004

Table 2.1: Fitness for both the clique of cardinality four and four connected sets of cardinality three, for different β.

is used, **Union**, and the terminal set is the set of vertex labels. Soule *et al.* point out that the arguments provided by O'Reilly and Oppacher against a GP Schema Theorem fail for this domain [O'Reilly and Oppacher, 1995b]: subtrees do not interact adversely and subtrees represent hyperplanes of the fitness landscape.

Soule and Foster have used the max clique to investigate the relation between graph characteristics and GA hardness [Soule and Foster, 1997], i.e., how hard a particular problem is for a GA to solve. The GA encoding of Bui and Eppley could not maintain several candidate cliques in a single chromosome. Soule and Foster utilize a grouping GA [Falkenauer, 1995] to maintain multiple candidate cliques in the chromosome. They fix the number of initial groups in a chromosome to be initially 30 and slowly reduce it to 4. Since they are interested in max clique and not clique cover, only the largest group contributes to the fitness of the chromosome.

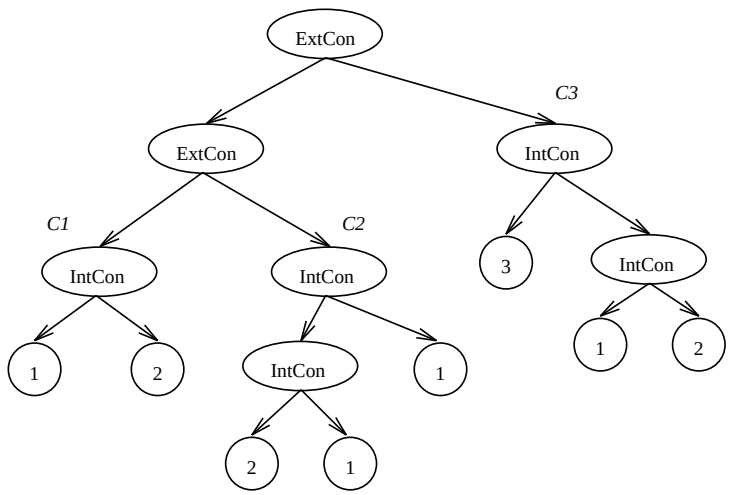

Figure 2.3: Example S–expression for the clique detection. Candidate cliques are labeled C1, C2, and C3.

2.3 Implementing Type Inheritance

In order to implement the clique detector, we have to introduce the type inheritance discussed in Section 1.3. The chromosome is mapped into a list of list of nodes. A chromosome is shown in S–expression form in Figure 2.3, and corresponds to the set

$$C = \{\{1, 2\}, \{2, 1, 1\}, \{3, 1, 2\}\}.$$

Let L_e be a list of candidate cliques, and L_c be a candidate clique, i.e. a list of nodes. It is evident that each element of L_e is an L_c. We want to ensure that the members of L_e and L_c are not L_e; i.e., neither a list of candidate cliques nor a candidate clique can have as a member a list of candidate cliques. This is shown in Figure 2.4, where an **ExtCon** is a child node of an **IntCon** node.

If the typing is the same as in Table 2.2, then S–expressions of the form shown in Figure 2.4 can be generated. These S–expressions correspond to incorrect

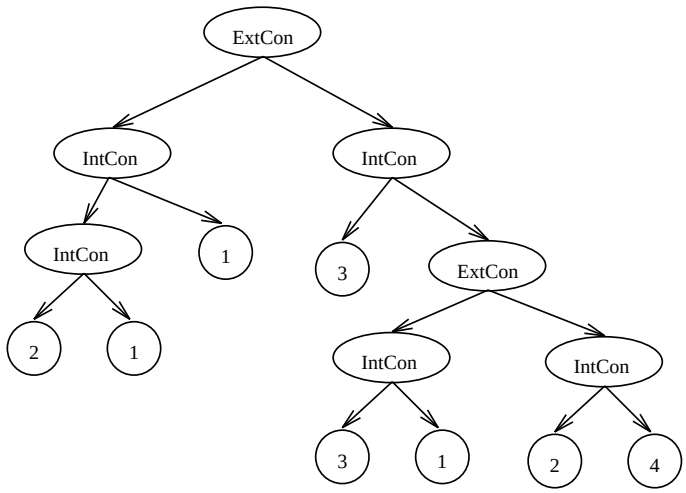

Figure 2.4: Bad S–expression corresponding to the first attempt at a type system.

Function/Terminal	Return Type	Argument Structure
Nodes	Node	
ExtCon	Node	Node A, Node B
IntCon	Node	Node A, Node B

Table 2.2: Naive attempt at typing for clique detection.

representations of the solution space. We do not want lists of cliques in which each clique can be a list of cliques.

From examining the desired form of the result, (see Figure 2.3) we derive the type system shown in Table 2.3. This system can not be implemented in a standard STGP package. It is representative of the non–branching type inheritance discussed in Section 1.3.2.

The two type levels represented in Table 2.3 are sufficient to solve the clique problem. However the types can be confusing to someone trying to understand the problem. Both **IntCon** and *Nodes* have a return type of `Node`. **ExtCon** is

Function/Terminal	Return Type	Argument Structure
Nodes	Node	
ExtCon	ExtNode	ExtNode A, ExtNode B
IntCon	Node	Node A, Node B

Table 2.3: Successful attempt at typing for clique detection.

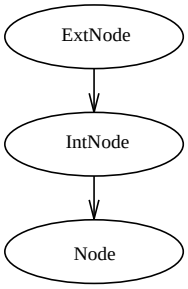

Figure 2.5: Non–branching type hierarchy for the clique detector.

a list building operator, **IntCon** effectively performs a Union on its child nodes, and *Nodes* returns a singleton set. Since **IntCon** is returning nodes, and *Nodes* is returning a node, the type of `Node` suggests they are returning the same object. Also, an **ExtCon** can have the terminal *Nodes* as a child node, with no interconnecting **IntCon**. This is the only way to specify a clique with only one member. Thus, `Node` is overloaded. To be correct, we should extend the type hierarchy to three levels, as shown in Figure 2.5.

2.3.1 Conclusion

The STGP as described by Montana [Montana, 1995] deals with genericity but not with other aspects of object oriented methods. We extend STGP to deal with type hierarchies, and in particular, with polymorphism and dynamic binding aspects of the object oriented paradigm. Of the branching and non–branching

cases of the type hierarchy, we have successfully implemented the non–branching case. In particular, we have shown how a non–branching type hierarchy can be used in clique detection, and how the standard STGP does not work for clique detection.

2.4 A GA Encoding

The GP encoding can be adapted to an encoding which can be used by RS, HC, SA, and GA. I fix a maximum number of positions, have a vertex encoding, and utilize a grouping GA to allow for a variable number of candidate cliques in a chromosome. Unlike Soule and Foster, I do not fix the number of groups in the chromosome. Instead I extend the alphabet and allow for a grouping marker, i.e., `-1`, which is never a valid vertex label, to indicate the ending of a group. I allow a grouping marker to appear with some probability p_m during both the initial random generation of chromosomes and mutation. The number of groups needed for a given graph is evolved along with the solution.

Figure 2.6 is an eight node graph which illustrates both max clique and clique cover. There are exactly 2 cliques: $C = \{\{0, 1, 2, 3\}, \{4, 5, 6, 7\}\}$. An example chromosome for the 8 node graph is

```
-1 3 3 -1 4 7 6 -1 0 1 2 5 6 -1 5 7 -1 -1 4 7 -1.
```

It has six candidate cliques, and the only cliques are #2 and #4:

$$C = \{\{4, 6, 7\}, \{5, 7\}\}.$$

The others are eliminated because they violate at least one of the rules: #1 contains duplicate vertices, i.e., vertex 3 is repeated; #3 is not completely connected; #5 contains no vertices; and, #6 is subsumed by #2.

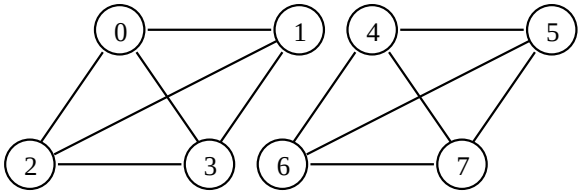

Figure 2.6: Example graph, consisting of 2 fully connected cliques of cardinality 4.

2.5 Conclusions

Encodings for detecting cliques in a graph for both a tree–based chromosome (to be used for GP) and a n–ary string chromosome (to be used for GA, RS, HC, and SA) were presented in this chapter. A common fitness evaluation can be applied to both representations. Now that the search heuristics, encodings, and fitness evaluation have been defined, we can start to investigate the sharing of building blocks between chromosomes.

CHAPTER III

Duplication of Coding Segments

3.1 Introduction

Research into the utility of non–coding segments, or introns, in genetic–based encodings has shown that they expedite the evolution of solutions in domains by protecting building blocks against destructive crossover [Levenick, 1991]. We consider a genetic programming system where non–coding segments can be removed, and the resultant chromosomes returned into the population. This parsimonious repair leads to premature convergence, since as we remove the naturally occurring non–coding segments, we strip away their protective backup feature. To avoid this problem, we duplicate the coding segments in the repaired chromosomes and place the modified chromosomes into the population. The duplication method significantly improves the learning rate in the domain we have considered.

3.2 Non–coding Segments

Non–coding segments model the intragenic regions reported in the biological literature and are the intron segments seen in the genetic based encoding (GBE) literature. They account for a large fraction of the DNA [Futuyma, 1986] and are believed to be backup material for the coding segments. For example, the frog *Xenopus laevis* has 450 copies of the gene codings for 18S and 28S rRNA

and 24,000 copies of the gene for 5S rRNA [Futuyma, 1986]. The non–coding sequences might also act as a library for adaptation. During RNA splicing the non–coding sequences are stripped, producing a smaller RNA molecule. As the gene can be spliced in a variety of ways, the non–coding sequence for one mRNA could be a coding sequence for another [Alberts *et al.*, 1989]. As a protein evolves to meet changes in the environment, it can also resort to the non–coding segments instead of evolving entirely new genetic material.

3.3 Genetic Algorithms

In the GA literature, the emphasis on non–coding segments has focused on how these extra bits provide a buffer against destructive crossover. The canonical GA chromosome, or string, representation utilizes a binary alphabet. If a `don't care` symbol is utilized, we have the schemata alphabet $\{0, 1, *\}$. A schemata is a template describing subsets of strings within the string. For example, the schema $s =$ `*1**0***` describes all strings that have a 1 in the second position and a 0 in the fifth. The *order* of a schema is the number of 0s and 1s present in the template. (In s, the order is 2.) The *defining length* of a schema is the distance between the outermost bits defined on the binary alphabet. (In s, the defining length is 3.) *Building blocks* have a small defining length and are highly fit. They are integral to the schema theorem, which defines how the implicit parallel search of a GA "builds" better solutions over time [Holland, 1975; Goldberg, 1989].

The addition of non–coding segments to chromosomes separates building blocks

and protects them from being sliced by crossover [Levenick, 1991]. GA chromosomes are typically of fixed length. With a string of length l, and a building block of defining length δ, any crossover operation has a probability

$$P_l = \frac{\delta}{l-1}$$

of destroying a building block [Goldberg, 1989]. If non–coding segments, of a total length of i are added, then the probability of destructive crossover breaking up a building block of defining length δ decreases to

$$P_{l+i} = \frac{\delta}{l+i-1}.$$

An example of non–coding segments is shown in Figure 3.1(a): there is a string of length $l = 15$ and a building block, $b1$, of defining length $\delta = 6$. The probability of crossover destroying $b1$ is $P_l = 0.43$. In Figure 3.1(b) a non-coding segment of length $i = 5$ is added and the probability of destructive crossover decreases to $P_{l+i} = 0.32$. Adding the non–coding segment to the chromosome's tail reduces the probability of destructive crossover, but does not aid the recombination of building blocks as much as placing the non–coding segments between the building blocks [Wu and Lindsay, 1995].

The key to inserting non–coding segments into the GA chromosome is that they reduce the chance of destructive crossover. An "artificial" constraint is that only the coding segments are examined to determine the fitness of a chromosome.

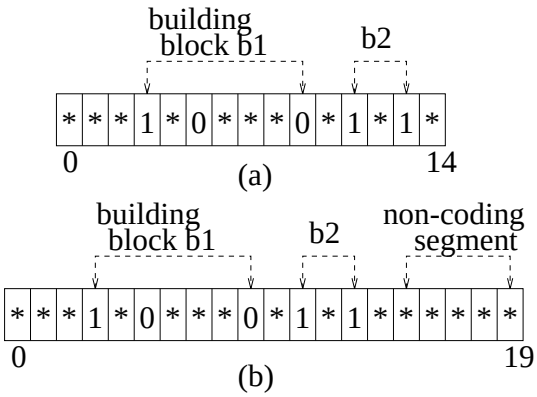

Figure 3.1: Non–coding segments in GA chromosomes prevent destructive crossover. (a) Without the non–coding segment. (b) With the non–coding segment.

Therefore material within a non–coding segment cannot be mixed with that within the coding segment. Thus the non–coding segment material is meaningless, and selection pressure does not drive it to be backup material.

Wu and Lindsay point out that there is a drawback to inserting non–coding segments; they retard the growth of building blocks [Wu and Lindsay, 1995]. It is hard for evolution to recombine the building blocks if non–coding segments are there to prevent destructive crossover. However, once those building blocks are formed, they are quite difficult to break up.

3.4 Genetic Programming

The "basic" theory of GP is borrowed from that of GA. Due to the difficulties in detecting building blocks in GP chromosomes, research is ongoing into formally connecting the theory as to why GP works with that of why GAs work [O'Reilly, 1995; Rosca and Ballard, 1996; Tackett, 1995]. The canonical GP chromosome representation is a parse tree (S–expression). The difference between GA and GP

is more than the fixed versus variable genotype representation. In GA there is a close relationship between the genotype and phenotype structure of a chromosome. Thus the building blocks of GAs are usually represented at the genotype level, and building blocks are relatively easy to detect. With the GP, building blocks are at the phenotype or semantical level, and are difficult to represent, detect, and capture. There can also be a duplication of building blocks in a GP chromosome, whereas there may not be any such duplication in a GA chromosome.

Tackett compares the difficulty in researching building blocks between GP and GA: different notations of schemata and a non–binary alphabet [Tackett, 1993]. He believes that small subtrees which appear frequently in S–expressions are GP's building blocks. These subtrees are prevalent due to their contribution to the fitness of the chromosomes in which they appear.

Altenberg believes duplications appear inside GP chromosomes due to two selection forces adding blocks of code to the population [Altenberg, 1994]. The genetic operators spread a block to different chromosomes, and an emergent selection pressure causes the formation of duplication within a chromosome. The duplication is a result of the fitness of the block being replicated.

Angeline reports while there is redundancy in chromosomes, the benefit of these semantically extraneous components is in the prevention of destructive crossover [Angeline, 1994]. He highlights a difference between GAs and GPs with regards to non–coding segments: in GAs they are added by design and in GPs they evolve naturally.

Nordin investigates the dynamics of non–coding segments in GP evolution [Nordin, 1996]. His chromosomes are comprised of linear genomes which are 32 bit strings and are binary code for a SUN–4 [Nordin, 1994]. Non–coding bits are defined to be those that when replaced by a `NOP` instruction do not change the semantics or phenotype of the chromosome. Using this capability, Nordin investigated the effects of non–coding segments on destructive crossover. He reached the same conclusions regarding the utility of non–coding segments as did the GA researchers. He reports promising preliminary results with the canonical representation.

3.5 Repair and Duplication

In my initial experiments in clique covering, I noticed that chromosomes had a high signal–to–noise ratio; i.e., the non–coding segments were expressed significantly more than the coding segments.

Under some chromosome encodings, certain genotypical combinations represent invalid states. For example, if we encode BCD numbers into binary, we need four bits to represent each digit. The bit combinations in the range $0000 \mapsto 1001$ are valid and the bit combinations in the range $1010 \mapsto 1111$ are invalid. Either crossover or mutation changes a valid combination into an invalid combination. During the translation from genotype to phenotype, we can deterministically map invalid combinations into valid combinations. With the BCD example, we could use modulo arithmetic to force all combinations to be legal. If we then take the phenotype and map it back into the genotype, we have *repaired* the chro-

mosome [Davis *et al.*, 1993; Orvosh and Davis, 1993]. Furthermore, *repair rate* denotes the percentage of repaired chromosomes that are returned into the population, overwriting the original. Thus, if a GA chromosome has invalid bits, and an algorithm can translate those bits into valid bits, then they can be repaired and the resultant chromosome evaluated to determine the fitness of the original chromosome. Repair is done at chromosome evaluation, not during the reproduction stage; there is no assurance that the repaired chromosome will even be selected for reproduction.

The translation from chromosome to a set of candidate cliques pares the chromosome into the coding segment(s). All chromosomes undergo this process and repair in this domain is the reverse process of translating the coding segment(s) back into the chromosome. The list of candidate cliques for a given chromosome succinctly encapsulates the content of that chromosome. Each candidate clique is a building block from which "better" chromosomes can be constructed. This paring down of the chromosome is similar to the RNA splicing in that non–coding segments are stripped out of the RNA transcript from DNA [Alberts *et al.*, 1989].

The evaluation function maps chromosomes from GP space to clique set space, i.e., genotype to phenotype. Repair maps the phenotype back into a genotype. Since the evaluation function removes nodes that do not contribute to the fitness, the resultant chromosome is likely to be smaller than the original. As an example consider the ten node graph, Figure 3.2, which I have used in my previous research

to test the clique covering system. There are exactly 10 cliques:

$$C = \{ \quad \{0,3,4\}, \{0,1,4\}, \{1,4,5\}, \{1,2,5\}, \{2,5,6\},$$
$$\{3,4,7\}, \{4,7,8\}, \{4,5,8\}, \{5,8,9\}, \{5,6,9\}\}.$$

A typical chromosome is presented in Figure 3.3. It has five candidate cliques, and the only cliques are #2 and #5: $C = \{\{4,8,7\},\{5,6\}\}$. The others are eliminated because they violate at least one of the rules: #4 contains duplicate nodes, i.e., node 7 is repeated; #3 is subsumed by #2; and, #1 is not completely connected. The fitness evaluation mapped the chromosome from GP space to clique set space. Selection of this chromosome for replacement produces the mapping back into GP space shown in Figure 3.4. Repair prunes dead branches of the S–expression.

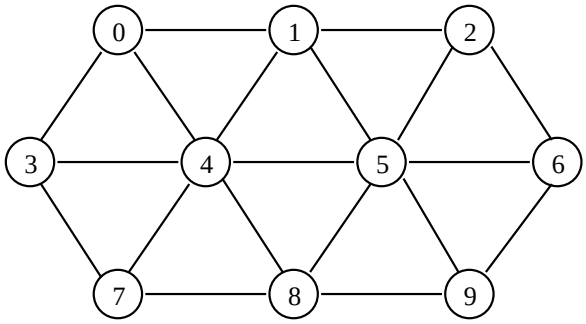

Figure 3.2: Example 10 node graph.

3.5.1 Simple Repair

The extraction of candidate cliques is a repair process and I investigate various rates of return of the repaired chromosomes into the population. My conjecture is that the genotypes of chromosomes which succinctly capture the phenotype of the

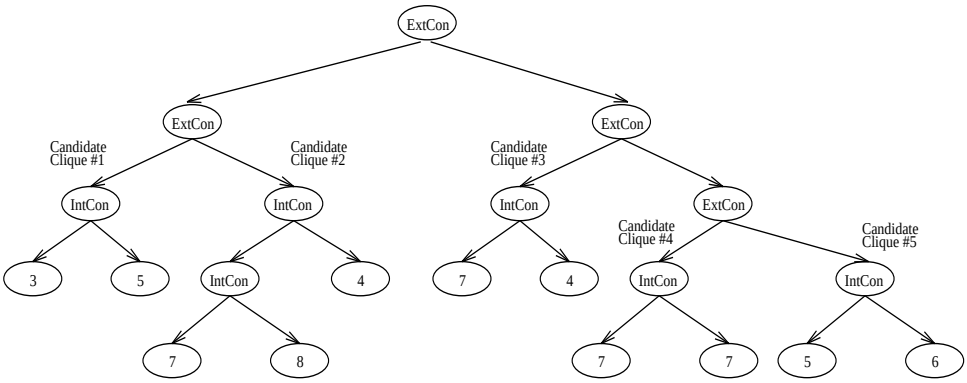

Figure 3.3: S–expression for 10 node graph.

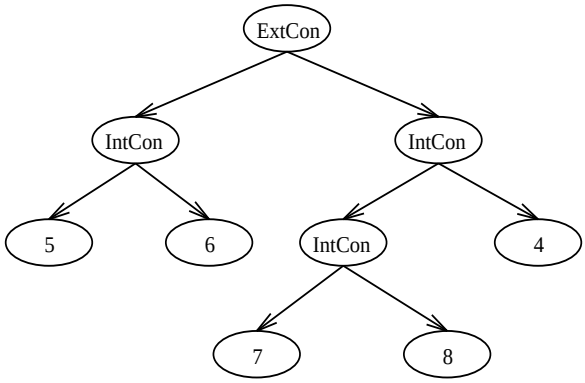

Figure 3.4: Repaired S–expression for 10 node graph.

chromosome are more elegant and natural. Non–coding segments can be inserted and deleted by evolution in DNA.

The experiments use a population size of 2000 and a generation size of 600, and are averaged over 10 trials. All statistical significance testing is done with a two-tail t-test, with a Student distribution, and a confidence level of 0.001. The 10 node graph (Figure 3.2) is used for clique covering. All chromosomes are repaired, and I investigate repair rates (the percentage of repaired chromosomes returned into the population) of 0%, 0.5%, 1.5%, 3%, 5%, and 10%. Repair rates greater than 0.5% (small repair rates are desirable [Orvosh and Davis, 1993])

either degrade the performance or cause premature convergence, see Figure 3.5. Why does repair work for GA, but not for GP?

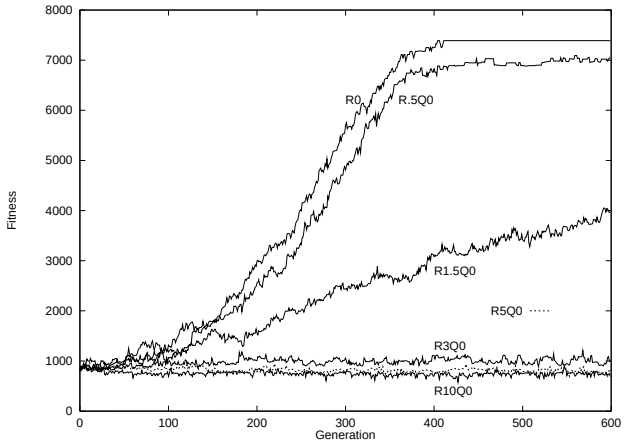

Figure 3.5: Best fitness for base case and repair rates of 0.5%, 1.5%, 3%, 5%, and 10%.

Repair removes "dead" or non–coding bits from the chromosome, i.e., those bits which do not contribute, either positively or negatively, to the calculation of the fitness. As a side effect, repair also removes genetic diversity. Finally, it removes any naturally occurring duplicate non–coding segments. Thus the protective backup feature of these segments is being negated. Genetic Based Encoding (GBE) research has shown that non–coding material protects building blocks from the effects of destructive crossover. I will discuss experiments in which a non–coding segments is inserted into the chromosome to investigate if there is a resulting increase in fitness.

3.5.2 Repair with Duplication

Further research is performed in which the repaired chromosome is duplicated before it is thrown back into the population. For example, the chromosome rep-

resented in Figure 3.6 has been duplicated into the chromosome in Figure 3.7. While the genotypes of these two chromosome are different, the phenotypes are exactly the same, i.e., both chromosomes evaluate to the same fitness. In effect, a non–coding segment has been added to the chromosome.

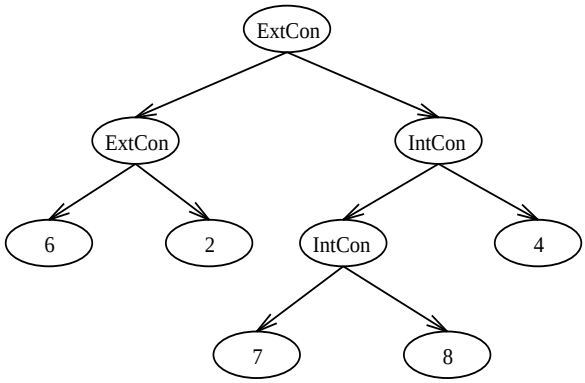

Figure 3.6: Best S–expression for generation 0.

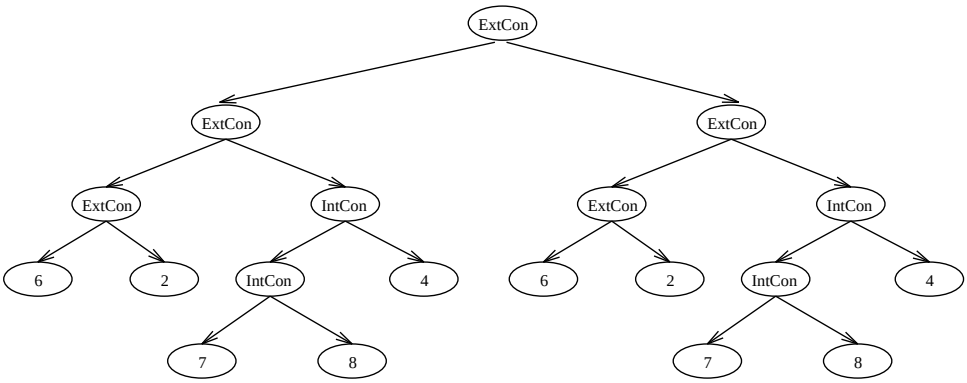

Figure 3.7: Generation 0's best S–expression doubled.

Crossover is destructive for the chromosome in Figure 3.6: any point selected for crossover will break up a building block. Crossover cannot be completely destructive for the chromosome in Figure 3.7; if any point to the left of the root is selected for crossover, then the right subtree will remain intact. The child which "inherits" the right subtree will have a fitness greater than or equal to that of the

parent. A similar argument holds for the right side. If the root is selected as the crossover point, then the child inheriting the whole tree still has a lower bound of the fitness of this parent. The non–coding segment is redundant in the parent, but it will only be redundant in the child if the other parent already contains the coding segment.

3.5.3 Experimental Results

The chromosome in Figure 3.7 should aid in the genetic search for all of the cliques in the graph, at least one of the children will be as fit as the repaired parent. The curve R0 in Figure 3.8 is the learning curve for the clique cover with no repairs taking place, with the solution found at about generation 354. The first experiment I conduct is to inject repairs with a 0.5% probability into the population. The curve R.5Q1 in Figure 3.8 is the result after adding one duplicate of the coding segment during the repair process. The solution is found at about generation 335.

The hypothesis of the utility of duplication appears to not have been significant. If we examine the process, we see that if the repaired chromosome is selected for crossover, the building block should last for at least one generation. Can the building block be forced to propagate through more than one generation? Yes, by adding more than one copy of the building block during repair.

If I assume that I can create only non–coding segments such that the total number of instances of the coding block is a power of two, then I can perform some worst and best case analysis as to the survivability of the coding segment.

In both cases, I assume that only one parent has copies of the coding segment. If we consider the tree formed by having the "roots" of the coding subtrees as terminals, we see it is a complete binary tree of depth $log_2 cs$, with cs being the number of instances of coding segments. The worst case is that the block will survive for $log_2 cs - 1$ generations (ignoring mutation). In the best case, the block will survive for a number of generations equal to the sum of the number of edges and the number of terminal nodes[1]. This is simply $3cs$.

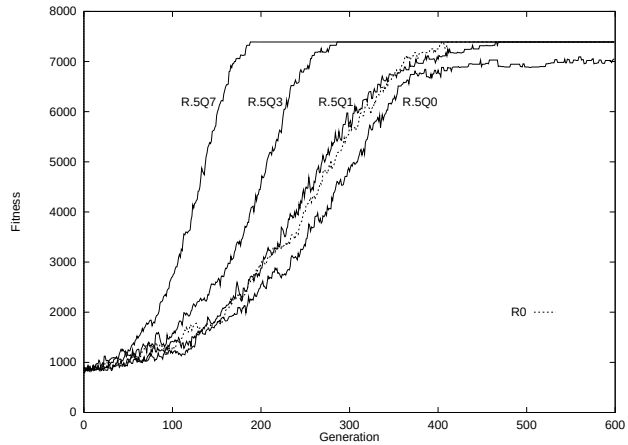

Figure 3.8: Best fitness for base case and a repair rate of 0.5% with 0, 1, 3, and 7 duplications.

I conducted further experiments by adding three and seven copies of the coding segment. The curve R.5Q3 in Figure 3.8 utilizes three backups of the coding segment. The solution appears around generation 246, a significant savings of 108 generations. The curve R.5Q7 utilizes seven duplicates. The solution appears around generation 171, a savings of 183 generations. Finally, in Figure 3.9, I present the results for a repair rate of 10%. At a repair rate of 10% and with 7

[1]Each block lasts until all of the crossover points above it have been chosen, and none have been chosen inside it.

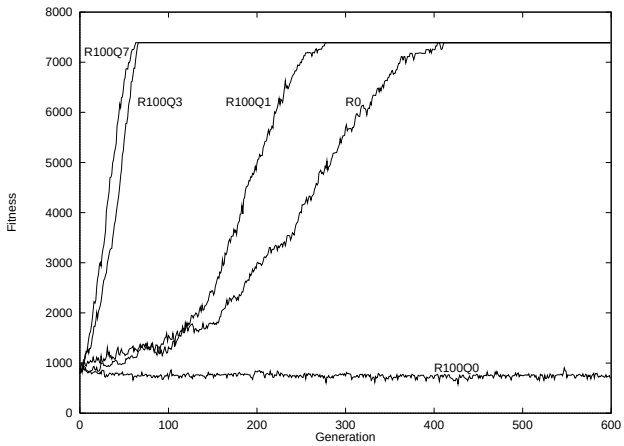

Figure 3.9: Best fitness for base case and a repair rate of 10% with 0, 1, 3, and 7 duplications.

duplicates of the coding segment, there is a significant savings of 298 generations over no repair, and 115 generations over 0.5% repair with 7 duplications.

I find that in general complete removal of non–coding segments causes premature convergence; increasing duplicates of the coding segment improves the learning; and, as the repair rate increases, and more than one duplicate of the coding segment is added to the chromosome, the learning increases. This contradicts the earlier findings reported in Orvosh and Davis [Orvosh and Davis, 1993].

3.5.4 Conclusions

I utilize the tree structure of GP chromosomes to conduct experimentation into duplication of coding segments. I have found that the duplication of three or more copies of the coding segments significantly speeds up the learning process for the clique covering problem. I have shown that with seven copies of the coding segment, I can at least halve the computational effort of finding the optimal

solution and at best I have shown an 84% increase in finding the optimal solution over no repair and duplication at all. While the detection of cliques in a graph readily lends itself to the study of building blocks in the GP chromosome, my results are not domain dependent.

Analysis shows that this method can work for any GP domain. Simple editing rules for GP chromosomes have been identified [Koza, 1992]. The methods used by compiler writers to optimize code are also applicable to "optimizing" the GP chromosome. An example of the repair and duplication process for other domains is shown in Figure 3.10. The parse tree to be evaluated is shown in Figure 3.10(a). The left subtree of the root node is *True*, which causes the middle subtree to be a coding segment and the right subtree to be a non–coding segment. The tree could be pruned, leaving only the middle subtree. The **IFTE** (IfThenElse) function can be used to add a duplicate of the coding segment [Angeline, 1994], as shown in Figure 3.10(b).

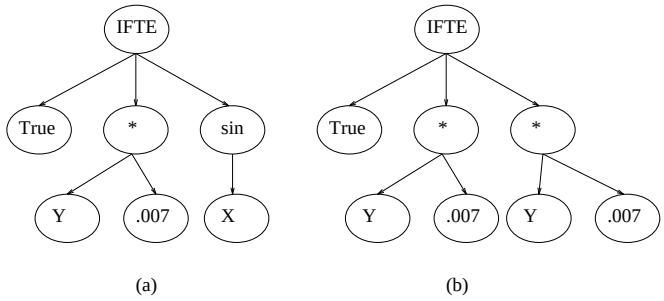

Figure 3.10: IFTE promotes duplication. (a) The right subtree of the IFTE node is non–coding. (b) A duplicate of the coding segment has been added.

3.6 Conclusions

I have shown that the duplication of building blocks does more than provide protection against destructive crossover; it also provides a natural backup of good genetic material. The duplication of building blocks enables the sharing of building blocks within the chromosome; the duplicates form redundant sections which the search heuristic's exploration operators can form new building blocks. In the rest of the dissertation, we see if we can leverage this internal separation of coding segments, duplication of coding segments, and joining of disparate coding segments, in a similar process external to the chromosome.

CHAPTER IV

Collective Intelligence

4.1 Introduction

My goal is to utilize simple computational agents to retrieve knowledge from the problem space, store that knowledge in a collective memory, and allow other computational agents to manipulate that knowledge in the collective memory. To that end, I define:

Information center : a centralized repository of knowledge. As the computational agents are simple and lack their own memory, this repository can act as a collective memory for the whole computational agent society.

Search agents : agents which retrieve knowledge from the problem space. They may not communicate with other agents outside of the collective memory. They may add knowledge to the collective memory, but they may not delete from it.

Process agents : agents which manipulate the knowledge stored in the collective memory. They may delete knowledge from the repository. They do not have access to the actual underlying search space.

Such a computational agent society is depicted in Figure 4.1. Note that search agents **S2** and **S3** retrieve the same knowledge. A task for one of the process

agents would be to eliminate redundant knowledge.

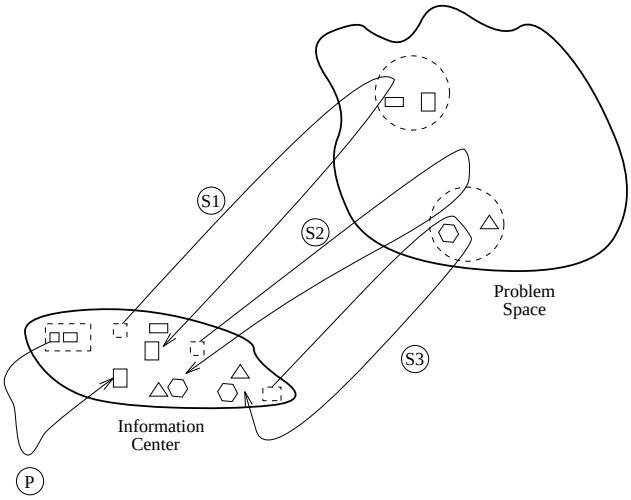

Figure 4.1: Computational agent society employing a collective memory. Process agents are labeled with a P and search agents with a S.

The basic principle of collective adaptation is that knowledge is gathered in a central location by search agents. Irrelevant portions of the search space are ignored in this focused copy of the search space. Process agents can manipulate that collective knowledge and, through it, influence the search agents. For example, they can eliminate some of the redundancy of the knowledge gathered by the search agents and direct the search agents to explore in potentially rich areas of the search space[1]. The collective memory allows the process agents to narrow their attention to relevant knowledge; it helps reduce the combinatorial explosion, allowing once prohibitive search strategies on the part of the process agents to be economical and productive.

[1]As will be shown later, direction is an overstatement: the process agents can "suggest" to the search agents where to focus their search, but the search agents are free to ignore the process agents.

A computational agent society can exhibit collective behavior in two dimensions: action and memory. Collective action is defined as the complex interaction that arises out of the sum of simpler actions by the agents. These simpler actions reflect a computational bound on either the reasoning power or memory storage or just the capabilities of the individual agents. Collective memory is defined as the combined knowledge gained by the interaction of the agents with both themselves and their environment. I combine the power of collective action with the expressiveness of collective memory to enhance a distributed search process.

The integration of collective action and memory leads to a distributed society of search agents which can interact via collective memory. The collective memory allows for either communication among the agents or for a centralized search of the gathered knowledge. I consider simple computational search agents, which are chromosomes in a GP population. Both GA and GP represent search strategies using a population of chromosomes. The chromosomes are considered to be autonomous in the sense that they do not typically interact to find a solution. They can be considered to be implicitly cooperative since the more fit chromosomes of generation G_i are more likely to contribute genetic material to the chromosomes in generation G_{i+1}.

Each chromosome is evaluated by a fitness function, which maps the chromosome representation into a given problem domain. The evaluation of one chromosome typically is independent of all others. A notable exception arises in genetic–based machine learning (GBML) systems: both rules and rule-sets

must be maintained. In the "Michigan approach" each chromosome is a rule and the population as a whole is the rule-set [Holland, 1986]. In these systems, the evaluation of a single chromosome is dependent on that of the population. If a rule is enacted only if the system is in a certain state, then it is dependent on other rules to get the system to that state. A major concern is how does the rule "reward" other rules for getting the system to that state, i.e., how is credit assignment handled?

Genetic algorithms are often interpreted as competitive learning systems: the driving force for exploring the fitness landscape is "survival of the fittest." Some GA applications can also be considered as cooperative learning systems [Cobb, 1993]: "Michigan style" GBML systems certainly fall into this category. Furthermore, Cobb views the fitness function as a mechanism to determine which solutions are to be shared among the chromosomes, with the crossover operator as the vehicle for which "partial solutions" are shared between chromosomes [Cobb, 1993].

4.2 Collective Memory

Collective memory is the body of common knowledge that a group shares. Common knowledge is knowledge which is either learned through interaction with the environment or explicitly communicated from one individual to all others in the group. Some examples of common knowledge include cars have four wheels, boiling water will burn you, and James Bond always gets the girl. Indeed, the central

thesis of the CYC project is that the inability of computational systems to effectively interact with humans (whether it be in direct communication or by reading encyclopedia articles) is that the computational systems lack the basic common knowledge that humans possess [Guha and Lenat, 1990].

Common knowledge models knowledge that each member of the group possesses and not the knowledge that an individual or subgroup possesses. Halpern and Moses point out a problem of a group learning a piece of common knowledge K: how does an individual A_i know whether another individual A_j has learned K or not [Halpern and Moses, 1990]? Furthermore, even if A_i is able to deduce that A_j knows K, how does A_i know that A_j knows A_i knows that fact? Without a base condition, this recursive question can stretch on *ad infinitum*. With my definition of common knowledge, each individual A_i knows that all other individuals $A_{j,j\neq i}$ have the knowledge and knows that every individual $A_{j,j\neq i}$ knows that every other individual $A_{k,k\neq j}$ knows that knowledge and for any deeper levels of nesting. We need not worry about whether A_i knows whether A_j knows the fact or whether A_i knows whether A_j knows whether A_i knows the fact, etc.

The collective memory can itself be either centralized or distributed (for examples of centralized and distributed blackboard architectures see [Corkill, 1989] and [Decker *et al.*, 1993]). Garland and Alterman present a distributed collective memory: agents manipulate their own slice of the collective memory [Garland and Alterman, 1995; Garland and Alterman, 1996].

4.3 Computational Agent Society

As problem spaces increase in complexity, the search for a solution can overwhelm a single computational agent. We can increase the exploratory power during the search process by introducing more agents. The first step is parallel search; the agents cannot communicate and thus are unable to coordinate their search efforts. We might assign n agents to the search, but instead of examining n different areas of the space, they might converge to one area, perhaps representing a local minimum. The next step is to allow communication between the agents, and thus move to distributed search. The agents are able to coordinate their actions, maximizing their coverage of the problem space.

I wish to minimize the complexity of the agents in this society. I believe that the knowledge gained from the interactions of a group of simple agents will be greater than the sum of the knowledge of those same individual agents. I limit the amount of memory that an individual agent can possess, but allow a group memory to which any individual in the society may access. As my model does not allow inter–agent communication and I am not concerned with the actual dispersal of knowledge through communication (and the resultant problems as reported in [Halpern and Moses, 1990]) all agent communication takes place to and from this collective memory, not between individuals.

We can model the collective memory as a blackboard production system [Fennell and Lesser, 1977; Nii, 1986; Corkill *et al.*, 1986]. With the blackboard architecture, any agent may read the partial solutions contained on the blackboard. An

agent can only write on the blackboard if it has possession of the "chalk". Likewise an agent may only delete partial solutions from the blackboard if it has the "eraser". When one agent writes a piece of knowledge on the blackboard, every agent has instant awareness of that knowledge. There need not be any point–to–point communication between two or more agents to transfer knowledge.

By limiting both the number of pieces of "chalk" and the number of "erasers", we can control access to the board. We can further add permission flags, much like file permission flags, to the agents and both the chalk and the eraser. Without write permission, an agent may not possess the chalk and hence may not add knowledge. Without delete permission, an agent may not possess the eraser and hence may not delete knowledge. Finally, it is also helpful to consider a read privilege which corresponds to the ability of an agent to read knowledge from the blackboard. Without such permission, it may not transfer knowledge from the blackboard.

However, if an agent updates a piece of a partial solution and wants to write it back to the global memory, questions arise as to whether the original memory has changed since the agent retrieved it, whether another agent is trying to read that partial solution during the update, how do agents get notified that the partial solution has changed, etc. Such issues are discussed in basic texts on operating systems [Tanenbaum, 1987] and computer architecture [Hwang and Briggs, 1985]. My model avoids these problems by restricting the number of pieces of chalk, i.e., by providing a semaphore to the write process, the number of erasers, i.e.,

by providing a semaphore to the delete process, the amount of local memory an agent possesses and by restricting agents to only being able to update their local memory at a common time.

Process agents cannot manipulate the search space; they must direct the search agents in order to sense the search space. To direct the search agents, they must add knowledge to the collective memory and wait for a search agent to read that knowledge. Since my model does not allow for the naming of agents, any message posted to the collective memory can be read and acted on by all search agents, which is in effect a broadcast. The search agents can neither search in the collective memory nor direct other search agents. They may add knowledge to the collective memory – resulting in directing other search or process agents. They may not however direct their communication to a specific agent. Thus they may not directly control the actions of another agent.

Earlier, we discussed three sets of permissions that an agent could have with the collective memory: read, write, and delete. Since we differentiate agents by the class they belong to and not as individuals, we assign these permissions to the class as a whole. By definition, search agents have the write permission and do not have the delete permission; they retrieve knowledge from the search space and store it into the collective memory. Also by definition, process agents have the read permission. In my model, if the process agent has the write permission, it also has the delete permission. Thus the two independent variables of access are whether the search agents have read permission and whether the process agents

have write permission.

The interactions of both the process and search agents with the collective memory form two orthogonal dimensions of access. Both dimensions can take on one of two discrete values: passive and active. Passive search agents may not read from the collective memory, while active ones may read from it. Passive process agents may not write to the collective memory, while active ones may write to it. I reference a tuple in these dimensions by *Interactivity-Processing*, where Interactivity denotes the read access of the search agents and Processing denotes the write access of the process agents.

The four models of access are:

Passive–Passive : Process agents may not write to the collective memory and search agents may not read from it. This models a two–tiered agent society in which both the process and search agents are engaged in parallel search. Also, the collective memory acts as a one way communication channel between the two agent classes: search agents may broadcast messages which only the process agents can receive. As the process agents cannot access the original search space, they are restricted to searching through the knowledge retrieved by the search agents.

Passive–Active : Process agents may write to the collective memory but search agents may not read from it. This represents a two–tiered agent society in which the process agents are engaged in distributed search while the search agents are engaged in parallel search. The collective memory acts

as a one way communication channel between the two agent classes. The process agents are able to communicate with each other through the collective memory. Note that since agents cannot be identified by name, any such communication is a broadcast.

Active–Passive : Process agents may not write to the collective memory and search agents may read from it. This models a two–tiered agent society in which the process agents are engaged in parallel search and the search agents are engaged in distributed search. The search agents are able to communicate with each other through the collective memory.

Active–Active : Process agents may write to the collective memory and search agents may read from it. This models an agent society in which both the process and search agents are engaged in distributed search. The process and search agents are able to communicate with each other through the collective memory. Note that since agents cannot be identified by name, such communication is a broadcast.

In this Section of my dissertation, I will explore the addition of Active–Passive[2], Passive–Active, and Active–Active collective memory to a society comprised of both a single search agent, the GP system, and of a variable number of process agents, which are simple local search algorithms. I will not explore a Passive–Passive collective memory. The GP chromosomes have the capability of

[2]I have previously shown Active–Passive interaction in a theorem proving domain [Haynes *et al.*, 1996a].

adaption during the search process, which can eventually allow them to become quite complex. I allow the explicit reuse of knowledge from one generation to the next; unlike selection, I do not confine that transfer of knowledge to be related to the fitness of an individual chromosome.

4.3.1 Prior Work

While the blackboard model is a useful metaphor for the collective memory employed by our system, there are many key differences between a classical blackboard system and collective adaptation. The most significant difference is the search agents are not independent Knowledge Sources (KS). Each agent is always executing and there is no triggering of execution via data added to the blackboard, which removes the need for a scheduling mechanism.

The A–Teams research employs a shared memory to effect communication between a team of autonomous agents [Talukdar *et al.*, 1983; de Souza and Talukdar, 1991]. Each agent is either a constructor or destructor; i.e., they either add or delete knowledge from the shared memory. My work differs from theirs in that my agents have synchronous execution and candidate solutions are dynamically modified during the execution of the heuristic's lifespan and not statically at the end of the agent's execution.

Blackboards have been used by an agent society to solve instances of the graph coloring problem [Hogg and Williams, 1993; Hogg and Williams, 1994]. In contrast to our system, information is selected via a triggering mechanism and each agent is executing the same search heuristic. Hogg and Williams determined while

the exchange of information was useful, if there are a large number of candidate solutions, few of which can be extended to full solutions, then there is a decrease in the ability of the heuristics to effectively prune unproductive branches [Hogg and Williams, 1994].

Finally, local search heuristics have been applied to improve the population–based search of both genetic algorithms [Hart, 1994; Hart and Belew, 1996] and genetic programming [O'Reilly and Oppacher, 1994; O'Reilly, 1995; O'Reilly and Oppacher, 1995a]. My work differs significantly from these research efforts in that I use a collective memory and the local search is applied within that collective memory and not to the chromosomes themselves.

4.3.2 Extracting Partial Solutions from the Chromosomes

The transfer of coding segments, i.e., partial solutions, from the chromosomes to the collective memory entails extracting phenotypical knowledge from the chromosomes. Such partial solutions can either be stored in the collective memory utilizing the alphabet employed for the domain, e.g., as subtrees of the chromosome if there is a mapping back from phenotype to genotype[3], or in a format better suited for the collective memory. The transfer of partial solutions from the collective memory to the chromosomes, i.e., directing the search pattern of the search agent in the next generation, necessitates the translation of phenotypical knowledge to genotypical format. If the partial solutions is stored as subtrees in the collective memory, the transfer to the chromosomes can entail minor modifi-

[3]Which is not always possible.

cations to "graft" the new subtree onto the chromosome. If the partial solution is stored in a format suited for the collective memory, then the transfer of partial solution process can involve a domain dependent translation from the collective memory format to a subtree, which would then be grafted onto the chromosome.

Genetic programming researchers have extracted knowledge from chromosomes, building "libraries" or "banks" from which material could be extracted at a later date [Tackett, 1995; Rosca and Ballard, 1996]. The knowledge is in the form of a subtree of the chromosome. These subtrees are chosen based on their perceived utility; i.e., they appear often in chromosomes which are highly fit. They are also conjectured to be the blocks from which the solution is built [Tackett, 1993]. However, the extracted knowledge is not processed; the partial solution contained in one subtree is not combined with the partial solution contained in other chromosomes. Tackett's *gene-bank* gathers statistical information about these subtrees. The collected subtrees are not allowed to return to the population [Tackett, 1995]. Rosca and Ballard consider an *adaptive representation* approach in which they extract small subtrees and allow them to be added back into the population by extending the alphabet [Rosca and Ballard, 1996]. Finally, Seront [Seront, 1995] considers a *concept library* system in which the knowledge gained from solving one problem, P_a, is applied to the solution of similar problems. The basic algorithm is to save the last generation used to solve problem P_a and then use that generation to bootstrap the new population. The new population is allowed to perform in crossover with the library.

My contribution to the extraction of partial solution from the chromosome is two–fold; I explicitly integrate the knowledge from the chromosomes and I find the solution from that integrated knowledge, i.e., the solution need not be expressed in a chromosome. The addition of cooperation changes the implicit distributed search of the GP to explicit distributed search. In GP research, the goal is for the solution to be encoded in the chromosome. The search is driven by the selection pressure formed by the combination of the fitness evaluation and the crossover function. From a random beginning, the chromosomes build the solution in their genotypical representation of the domain; potentially encoding more knowledge each generation.

With collective adaptation, the chromosome can remain relatively simple. Since the solution can be expressed in the collective memory and not the chromosome, the chromosome is no longer just a vehicle to represent the solution, rather it is a springboard from which richer parts of the search space can be explored. The chromosome can be comprised of small and simple building blocks, which can easily be combined.

4.4 Conclusions

In this chapter, a distributed search heuristic has been introduced. In subsequent chapters, this heuristic is applied such that a single search agent, employing one of the weak search heuristics, can extract coding from chromosomes into a collective memory. These partial solutions are then shared amongst not only the search

agent and all process agents, but also the chromosomes inside the search agent.

CHAPTER V

Exploiting an Information Center for Exploration

5.1 Introduction

I have conducted some experiments in distributed search in the clique cover domain and obtained promising results [Haynes, 1996; Haynes, 1997a; Haynes, 1997b; Haynes, 1997c; Haynes, 1997d]. All experiments were conducted with the GPengine package, written by myself for research presented in my master's thesis [Haynes, 1994; Haynes and Wainwright, 1995]. In this chapter I will discuss these experiments and their results. Finally I will present the key avenues of further research I have identified.

5.2 Passive–Active

If a chromosome contained no valid candidate cliques, I tried a repair strategy of injecting the set of all valid cliques found to date, which is an Active–Passive collective memory[1]. I found that such a repair strategy led to premature convergence in a non–optimal section of the search space. It appears that the Active–Passive collective memory technique has failed to aid in the search process. If I instead adopt a Passive–Active collective memory technique in this domain, the search

[1]This would correspond to the GP being the search agent and there being no process agent. As the search agent may read from the collective memory, a chromosome containing no valid candidate cliques is overwritten with the collective memory.

process is greatly facilitated.

With the Passive–Active collective memory I do not repair chromosomes which have no valid candidate cliques. Instead the search agent gathers candidate cliques in the collective memory and the process agent removes duplicates and candidates subsumed by larger candidates. In Figure 5.1, I present a comparison of three search techniques for clique cover (For all of my experiments, I set $\alpha = 10$, $\beta = 9$, and $\gamma = 0$ [Haynes *et al.*, 1996b].). The noteworthy parameters for the GP system were a max of 600 generations[2] and a population size of 2000. Each curve shown in Figure 5.1 is an average of 10 different runs. Each of the methods extends the previous methods. The first method (R0) is a strongly typed genetic programming (STGP) [Montana, 1995] system modified with the type inheritance presented in [Haynes *et al.*, 1996b]. Chromosomes are repaired during the fitness evaluation, but they are not returned into the population. The second search method (R10Q7) replaces the original chromosome with the repaired one with a probability of 0.1. The coding segment is duplicated seven times during the replacement process. The third method (PACM) adds Passive–Active collective memory to piece together the set of all cliques.

The average generation to discover the optimal solution is shown in Table 5.1. On the average, PACM is 7 times more efficient than R10Q7 and $44\frac{1}{4}$ times more efficient than R0. Finally, if I investigate how much the repair process is assisting the Passive-Collective memory, we can see in Figure 5.2 that the addition of the

[2]Even if the system found the optimal solution, I let the search continue on until the maximum number of generations had passed.

duplication of coding segments repair is not significant. The PACMR10Q7 curve corresponds to the PACM curve in Figure 5.1, while the PACMR0 curve represents a Passive–Active collective memory which does not use the repair process.

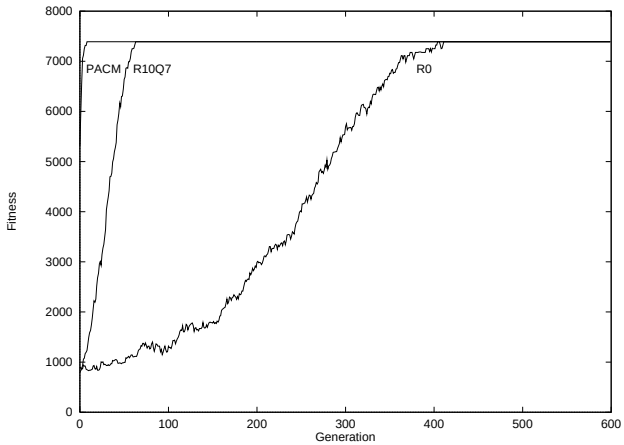

Figure 5.1: For the 10–node graph, comparison of best fitness per generation for no repair of chromosomes (R0), duplication of coding segments repair of chromosomes with a 10% return rate and 7 duplicates (R10Q7), and Passive–Active collective memory (PACM), which utilizes R10Q7 to drive the search agents.

Strategy	Generation of Appearance
R0	354
R10Q7	56
PACM	8

Table 5.1: Average appearance of optimal solution for different search strategies.

The addition of Passive–Active collective memory to the search technique significantly improves the efficiency of the search process. I want to leverage that improvement to allow clique covering in more realistic graphs. The ten node graph I use to illustrate the clique covering is contrived and thus facili-

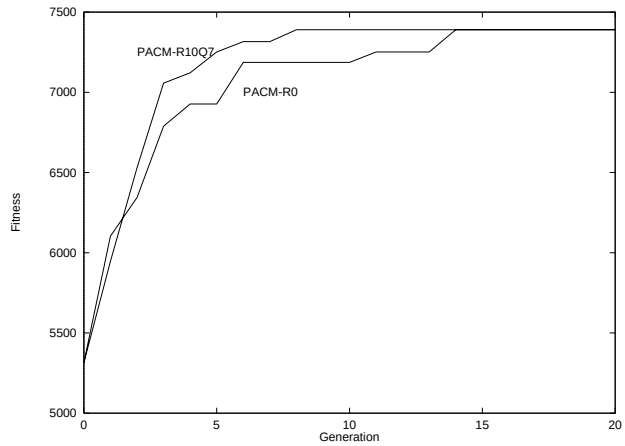

Figure 5.2: For the 10–node graph, comparison of best fitness per generation for Passive–Active collective memory (PACMR10Q7), which utilizes R10Q7 to drive the search agents, and Passive–Active collective memory (PACMR0), which utilizes no repair.

tates the search process, i.e., a known optimal solution exists. The search for the optimal solution for this graph is not trivial with either plain GP or STGP systems. In the Second DIMACS Challenge [Johnson and Trick, 1996] random graphs were generated as tests for the maximum clique detection problem (ftp://dimacs.rutgers.edu/pub/challenge). While the duplication of coding segments repair process is able to search such graphs, the plain STGP system will prematurely converge.

A shortcoming of the data reported for these graphs is the results presented are for the maximal clique size found, if any, but no data are presented for either the number and composition of all cliques in the graph. Both finding the maximum and all cliques in a graph are NP–complete [Garey and Johnson, 1979]. A brute force algorithm is to build candidate cliques in increasing levels of size, k. Due to NP–completeness, this algorithm is not guaranteed to be able to find a solution.

A viable search heuristic is to detect cliques from the Passive–Active collective memory.

I now examine the hamming6-4.clq dataset from the DIMACS repository, which has 64 nodes, 704 edges, and a maximum clique size of 4. From the brute force algorithm, we know that there are 464 cliques, with a maximum fitness of 1,597,424. I present the results, in Figure 5.3, of testing both R10Q7; i.e., replace the original chromosome with the repaired one with a probability of 0.1 and the coding segment is duplicated seven times during the replacement process, and PACM, i.e., add Passive–Active collective memory to piece together the set of all cliques.

The addition of Passive–Active collective memory is significant in improving the search process. However, the highest reported fitness of about 650,000 is only about 40% of the maximum fitness. As the learning curve has not stabilized at a plateau, I could allow the search to continue for more generations. I could also increase the population size. Both methods fail to address my implicit desire to effectively search the space in both minimal time and memory.

5.2.1 Conclusions

In this experiment, I found that the partial solution gathered by the chromosomes focused the search space into the collective memory. Process agents were then able to collate the knowledge gathered in the collective memory. This integration of knowledge lead to significant speed–up in the search process over the repair and duplication method. I then increased the complexity of the graph for which cliques

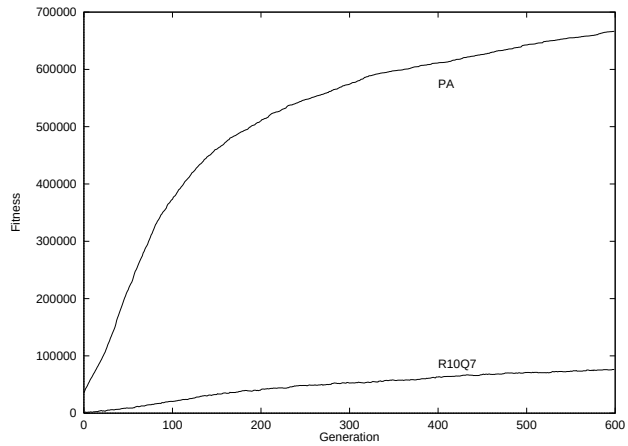

Figure 5.3: Passive–Active collective memory search applied to the hamming6–4 graph. In particular, comparison of best fitness per generation for duplication of coding segments repair of chromosomes with a 10% return rate and 7 duplicates (R10Q7), and Passive–Active collective memory (PACM), which utilizes R10Q7 to drive the search agents.

were being detected. The collective adaptation method was better able to explore the search space than the repair and duplication method.

5.3 Active–Process Agents

As mentioned in Section 5.2, the addition of Passive–Active collective adaptation is significant in improving the search process. However, the highest reported fitness of about 650,000 is only about 40% of the maximum fitness. As the learning curve has not stabilized at a plateau, I stated that we could either allow the search to continue for more generations or increase the population size. I rejected both methods as they fail to address my implicit desire to effectively search the space in both minimal time and memory.

A possible extension is to bestow further computational effort to the process

agent(s)[3]. The collective memory is a rich repository of knowledge and the process agents should be able to exploit the exploration of the search agents. Imagine the collective memory as a lens for focusing the search space into a more manageable space; the process agents are able to confine their search to the rich areas of the search space. The key point is that process agents are not working in the original search space, where confinement in a rich, but narrow, area might lead to an agent being trapped in a local minimum. As the search space has been refined for the process agents, they should be able to avoid the combinatorial explosion found in the original space. Thus, we can extend the process agents with simple algorithms, which might not be effective in the face of the combinatorial explosion.

In the context of the clique covering, we consider a brute force algorithm.

1. Set $i = 0$ and construct a set S_i of all candidate cliques of size 2, i.e., if there is an edge between two vertices, add them as a candidate clique.

2. Loop over both the set of all candidate cliques, S_i, and the set of all vertices.

   (a) If a candidate clique cannot be expanded by the addition of one vertex, then add it to the set S_{i+1}.

   (b) Else, for each vertex which expands the candidate clique, add a new candidate clique to the set S_{i+1}.

3. Increment i by one, and repeat until no new candidate clique is formed; i.e., $S_i = S_{i+1}$.

[3]The process agent in this domain just collates the knowledge, removing duplicates.

In the original search space, such an algorithm quickly becomes infeasible as the problem complexity scales up. However, it can remain feasible in the focused search space.

In my first set of experiments, I add an additional process agent to the computational agent society. Each generation, after both the search agent and the collating process agent execute, the new agent randomly selects a vertex and tries to extend each of the candidate cliques contained in the collective memory (Expand by Random Vertex, ERV). There are some subtle differences between this algorithm and the brute force one.

1. Not all vertices are guaranteed to be considered as expansion vertices.

2. Candidate cliques which are subsumed by larger cliques cannot be used for exploration, i.e., the four candidate cliques, of size 3, of a candidate clique of size 4, C_{4i}, cannot be used to find potential candidate cliques of size 4, $C_{4j}(j \neq i)$, which have three vertices in common with C_{4i}.

3. Most importantly, the ERV algorithm is not guaranteed to find all cliques, whereas the brute force algorithm can do so.

While point 2 is a weakness, it is also a strength; as problem complexity increases, the system does not need to remember everything, alleviating the combinatorial explosion in storage. The GP can be used in this case to facilitate exploration; as it is redundantly gathering knowledge, over generations as well as in the same generation, it can detect new combinations of candidate cliques. Indeed this feature discovery is the contribution of the GP subsystem.

The results of the Passive–Active collective adaptation with "energetic" process agents (PA-Energetic) are shown in Figure 5.4. For comparison, the results from our earlier Passive–Active experiments with just collation are also presented (PA-R10Q7). Finally, the fitness corresponding to the optimal solution is presented (Set of All Cliques)[4]. It is evident that the extension of the computational abilities of the process agent, with a simple rule, is significantly effective in reducing the computational effort in the distributed search. On the average, the optimal solution is found in generation 368.

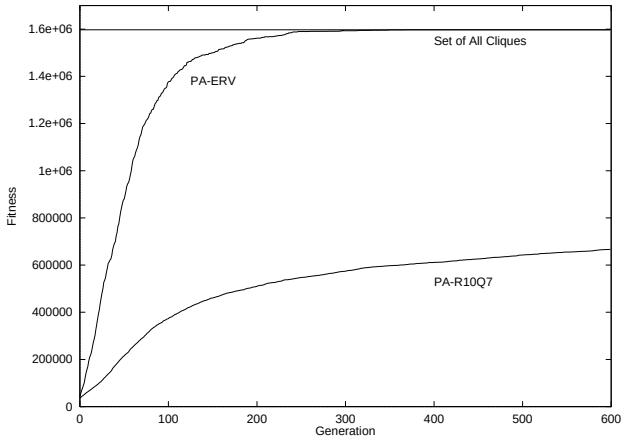

Figure 5.4: Comparison of fitness per generation for Passive–Active collective adaptation with two levels of activity on the part of the process agents: 1) a simple collating agent (PA-R10Q7), and 2) an agent which, after collation, extends by one randomly selected vertex each generation (PA-ERV). The underlying search engine is genetic programming with duplication of coding segments repair of chromosomes with a 10% return rate and 7 duplicates. All points represent the average of 10 runs. Also shown is the fitness associated with the set of all cliques (Set of All Cliques).

Why is this simple extension so effective? The answer lies in the narrowing of the search space into the collective memory space. The process agent is

[4]Determined by a brute force algorithm.

able to quickly explore the rich areas of the search space. Will the addition of process agents, employing simple algorithms, always lead to an improvement in learning? Even if we exclude bad algorithms, e.g., randomly delete one vertex from each candidate clique, the answer is still no. While not by design, the ERV algorithm minimizes its impact on building blocks, i.e., candidate cliques, and is quite ambitious in that the same expansion is tried on all candidate cliques. Each generation, the process agent employing the ERV algorithm is slowly expanding candidate cliques.

Consider instead a less ambitious algorithm, which maximizes locality in attempting to detect new candidate cliques. In the Merge Adjacent Candidate Cliques, MA, algorithm, I employ two additional process agents in conjunction with the collating one. After the collation, the first new process agent sorts all candidate cliques, based on vertex ordering within the candidate clique, and then the second one merges adjacent candidate cliques if the union of the vertices forms a new candidate clique.

The MA algorithm certainly seems feasible, but if we examine Figure 5.5, we find that it actually performs worse than Passive–Active collective adaptation. Why? The process agent which merges the candidate cliques is forming larger candidate cliques than the agent which employed the ERV algorithm. As a result smaller building blocks are not being exploited by the process agent. If n cliques of size k have a core candidate clique of size $i, i < k$, once one of the n cliques is found, the core candidate clique is not available for merging. Also, by maximizing

locality, this algorithm ensures that multiple mergers cannot take place unless the core candidate clique comprises the first i vertices of each candidate clique. It is not exploiting the exploration of the search agents.

I can test my hypothesis by considering a third algorithm, Merge Random Candidate Clique, MR. Once again I employ two process agents: one to sort and one to merge. However, now the merger randomly selects one of the candidate cliques and tries to merge it with every other candidate clique in the collective memory. As can be seen from Figure 5.5, this algorithm is significantly better than Merge Adjacent (MA) and worse than Expand Random Vertex (ERV). It performs better than MA because it does not maximize locality; each candidate clique has the opportunity to merge with the randomly selected one. It performs worse than ERV because, like MA, it is taking too big a step during the merge process.

5.3.1 Conclusions

We can increase the processing power of the search agents, but there might be physical or economical restrictions on the processing capabilities of the search agents. If there are such restrictions on the search agents, we can add simple algorithms to the process agents, capitalizing on the reduced search space. The advantage of considering a reduced search space is that simplistic algorithms, which are not *economical* in the original search space, can be used to effectively prune the search space farther.

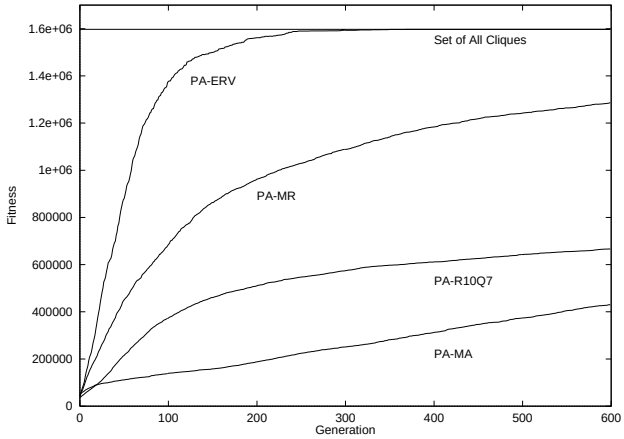

Figure 5.5: Comparison of fitness per generation for Passive–Active collective adaptation with different levels of activity on the part of the process agents: 1) a simple collating agent (PA-R10Q7), 2) agents, which after collation, sort then merges adjacent candidate cliques if they are connected (PA-MA), 3) agents, which after collation, sort and merge one randomly selected candidate clique with all other compatible candidate cliques in the collective memory, and 4) an agent which, after collation, extends by one randomly selected vertex each generation (PA-ERV). The underlying search engine is genetic programming with duplication of coding segments repair of chromosomes with a 10% return rate and 7 duplicates. All points represent the average of 10 runs. Also shown is the fitness associated with the set of all cliques (Set of All Cliques).

5.4 Random Search versus Genetic Programming

We can argue the majority of the savings might be from the addition of collective memory. Why then do we need genetic programming at all? Would not a blind search perform just as well? My intuitive answer is no; the GP algorithm guides the search agents in their exploration of the search space and it allows the agents to adaptively learn about their environment. While the search agent is no longer needed to discover the optimal solution, i.e., we need not wait until it is expressed in one of the chromosomes, it is needed to discover novel building blocks. As

a byproduct of striving to find the optimal solution, the GP selection pressure facilitates the discovery of such building blocks.

To confirm my expectations, I conducted a set of Passive–Active collective memory search experiments using random search as an engine for the search agent. In the first generation, chromosomes are constructed as in the GP system, i.e., a maximum depth of 4. The chromosomes are then evaluated by the same fitness function as in the GP system. The next generation is randomly constructed with a maximum depth of 10 and its chromosomes are evaluated. This process is repeated until the maximum number of generations has passed.

The results of the Passive–Active collective memory search with random search as an engine are shown in Figure 5.6. I also present the results from the Passive–Active experiments with R10Q7 and R0 as the engines. PA-RS is significantly better than both PA-R10Q7 and PA-R0, which is contrary to my expectation that the detection of building blocks, by the GP subsystem, can be exploited to guide the exploration of the search space.

My hypothesis fails because the solution need not be represented in the chromosomes. If we examine the fitness of the underlying search engines, in Figure 5.7, we see that R10Q7 outperforms both R0 and RS. Thus the R10Q7 search engine is able to detect building blocks and can express the solution inside the chromosome. Both the R0 and RS engines are not able to effectively detect building blocks.

While the PA-R0 performance curve is significantly worse than the PA-RS curve, the R0 performance curve is not significantly different from the RS. Why

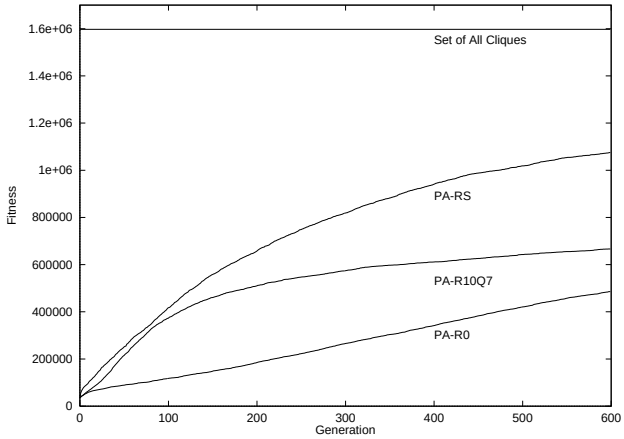

Figure 5.6: For the hamming6-4 graph, comparison of fitness per generation for Passive–Active search with three underlying search engines: 1) GP with duplication of coding segments repair of chromosomes with a 10% return rate and 7 duplicates (PA-R10Q7), 2) GP with no repair (PA-R0), and 3) random search (PA-RS). All points represent the average of 10 runs. Also shown is the fitness associated with the set of all cliques.

does the performance for PA-R0 and PA-RS not correlate with the underlying search engine? The random search subsystem is exploring the search space and the R0 is confined to a small section of the search space. Due to its stochastic nature, random search can make large jumps through the search space. While the GP algorithm is also stochastic in nature, the selection pressure caused by crossover between the more fit chromosomes, confines the jumps that R0 can make. It is the addition of the duplication of coding segments which allows the R10Q7 subsystem to effectively search via building blocks.

Also, when considering Figure 5.6, the reader should not jump to the conclusion that random search is better than GP. The findings in Figure 5.7 dispute this claim: if we just consider the underlying search engines, GP with repair significantly outperforms RS. It is only when we look at the addition of Passive–Active

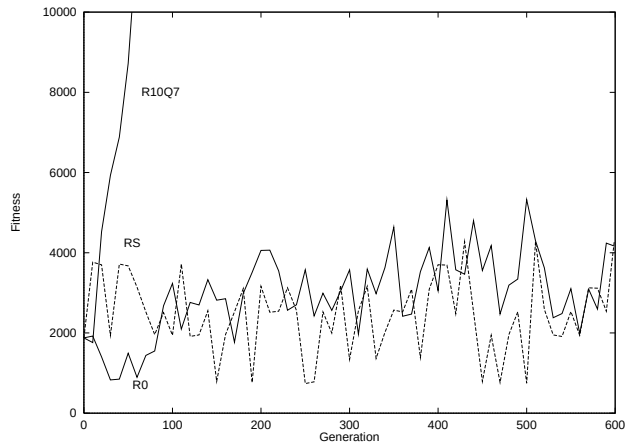

Figure 5.7: For the hamming6-4 graph, comparison of best fitness per generation for underlying search engines of three Passive–Active searches: 1) GP with duplication of coding segments repair of chromosomes with a 10% return rate and 7 duplicates (R10Q7), 2) GP with no repair (R0), and 3) random search (RS). All points represent the average of 10 runs. Also, only every tenth point is shown to aid visibility.

collective adaptation that the system employing random search is better than the one employing GP and repair.

5.4.1 Thought Experiment

I conjecture that the effectiveness of Passive–Active collective adaptation, with random search as its engine, will decrease as the problem complexity increases. The hamming6-4.clq graph is sparsely connected, has a small number of vertices, and a maximal clique size of four. I can illustrate my conjecture with a thought experiment; let us consider how many candidate cliques can be formed in completely connected graphs of size 8, 16, 32, and 64. I can enumerate all candidate cliques that can be expressed in all legitimate (i.e., they honor the type requirements for the alphabet employed) trees formed with a maximum depth of 3.

In Table 5.2, I present P_{8_i}, the percentage of candidate cliques of size 8 which can be generated from these trees for the various graphs. I also present E_{8_i}, the expected number of candidate cliques of size 8 which will be randomly generated in a population of 2000 chromosomes over the course of 600 generations ($E_{8i} = Pr_{8i} \times 2000 \times 600$). Finally, I make an assumption that all of these randomly generated candidate cliques are unique. I can now calculate the ratio of randomly generated candidate cliques of size 8 to the total possible number of candidate cliques of size 8. In Figure 5.8, we see that if we plot complete graph sizes against the ratio of expressed versus possible candidate cliques of size 8, we approximate exponential decay.

Graph Size	Percent Expressed	Expected	Ratio
8	$1.03E^{-4}$	124	$3.06E^{-3}$
16	$6.05E^{-3}$	7265	$1.40E^{-5}$
32	0.021	25120	$5.92E^{-8}$
64	0.036	42984	$2.41E^{-10}$

Table 5.2: For various completely connected graphs and all possible legitimate parse trees of depth 0 to 3, I report: 1) percentage of candidate cliques of size 8 which can be expressed in the trees, 2) expected number of unique candidate cliques of size 8 to appear over the course of random search, and 3) ratio of expected versus possible candidate cliques of size 8.

If I increase the maximum depth allowed in the parse trees, I will increase the number of candidate cliques of size 8 which can be expressed. Notice however if I set d to be the maximum depth, the limitation we observed for candidate cliques of size 8 will now apply for candidate cliques of size $2(d + 1)$. Also, as we increase d, we increase our memory requirements and we quickly run into the

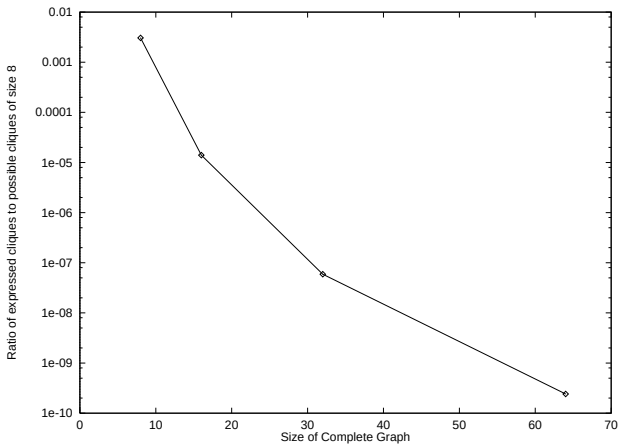

Figure 5.8: Comparison of completely connected graphs to the ratio of expressed candidate cliques of size 8 to possible candidate cliques of size 8, in parse trees of maximum depth 3. (Y-axis is log scale)

combinatorial explosion in memory space.

Thus my results from the analysis of the thought experiment still hold; we see that as we increase complexity (measured here by graph size), we decrease the probability of expressing candidate cliques via random search. I must return to the evolutionary pressure of GP to express candidate cliques as complexity increases. As a candidate clique is discovered, the GP is able to explore by extending the knowledge contained in that building block to detect other candidate cliques.

For example, consider a candidate clique of size 7, C_7, which is randomly generated in the initial population. The emergent pressure of the GP can use C_7 to search for all other viable candidate cliques of size 7 which are mainly comprised of nodes in C_7. Either the crossover or mutation operators can cause a few nodes to be changed and thus quite possibly discover a new candidate clique. Under my previous assumption of complete connectivity, this will always occur. Also, the GP can effectively utilize C_7 to search for all candidate cliques of size 8 for

which C_7 forms a core set of nodes. With random search, this potential to exploit exploration is lost.

The belief in emergent selection and the schema theorem [Holland, 1975], i.e., the building over time of the solution piece by piece from the elementary blocks, shields us from the fact that for cliques of maximum size 4 and parse trees of maximum depth 10, random search will effectively generate candidate cliques. That random generation will be very redundant, but given enough time will visit all possible candidate cliques. If we were mainly interested in expressing the solution in the chromosomes, then random search would fair poorly: the probability of expressing all 464 cliques in one chromosome is very small. But the addition of collective memory allows the search systems to distribute the discovery of candidate cliques to all of the chromosomes generated over time. Since we expect random search to generate all candidate cliques for the hamming6-4.clq graph, it should be no surprise that the addition of collective memory expresses the solution.

5.4.2 Fully Connected Graph of Size 16

I tested my thought experiment with a fully connected graph with 16 vertices. To make the calculations manageable in the thought experiment, we considered trees with a maximum depth of 3. Consistent with the parameters of the previous experiments, I utilize an initial generation maximum depth of 4 and subsequent generation maximum depth of 10. Both candidate cliques of size 16 and duplicate of the coding segments can readily be generated in trees of depth 10. It should be noted that with only one clique, the solution must be expressed inside a chro-

mosome. Since the process agents do not actively search within the collective memory, only the search agent can combine partial solutions.

In Figure 5.9 I present the results of Passive–Active search with both GP (PA-R10Q7) and random search (PA-RS) as the search engines[5]. The Passive–Active search with the GP based subsystem is significantly more effective than the random search based one. The PA-RS can discover small candidate cliques faster than PA-R10Q7, but the discovery of larger candidate cliques needs the direction given by PA-R10Q7. Furthermore, if we examine the performance of just the underlying search engines, see Figure 5.10, we see that the R10Q7 explores significantly better than RS. It is able to improve its solution at any generation by building on solutions from previous generations. (The spikes shown for RS are when one of the trials randomly expresses the clique of size 16.)

A potential critique against this experiment is that the solution must be expressed inside the chromosome and not in the collective memory. (As can be seen by the way the curve PA-R10Q7 in Figure 5.9 is tracked by the curve R10Q7 in Figure 5.10.) When we decrease the cooperation of the system, we will see a decline in the effectiveness of random search as an engine. The GP is still able to cooperate by sharing solutions via crossover. This critique falls apart if we consider a graph consisting of four fully connected cliques of size 16. Clearly if random search is not able to express one such clique during the alloted time, it will not be able to express four such cliques, even independently. The GP system

[5]Due to the range limits for doubles, I scaled the fitness calculation.

will however be able to begin to express each clique.

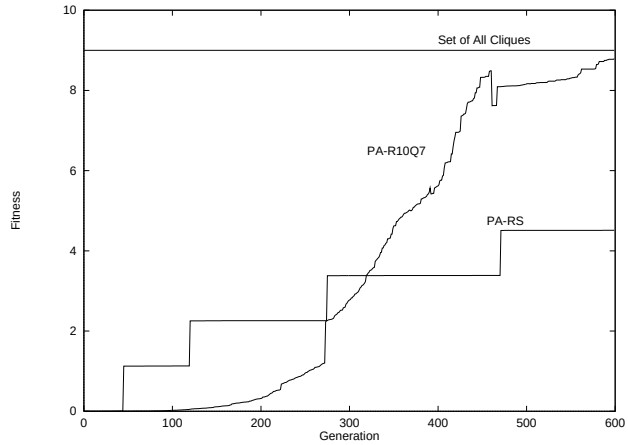

Figure 5.9: For the Fully Connect 16 graph, comparison of fitness per generation for Passive–Active search with two underlying search engines: 1) GP with duplication of coding segments repair of chromosomes with a 10% return rate and 7 duplicates (PA-R10Q7), and 2) random search (PA-RS). All points represent the average of 10 runs. Also shown is the fitness associated with the set of all cliques.

5.4.3 Conclusions

Collective adaptation utilizes cooperation to harness the competitive pressure of genetic programming. With collective adaptation, a GP based search engine is able to effectively search as problem complexity is increased. While the cooperation is necessary to exploit the solutions found by the underlying search engine, I have shown that the competition of GP is still needed to guide the exploration of the search process. A random search engine is more effective than a GP based one, but only at low problem complexity. As the complexity increases, the competitiveness of the GP search engine is more effective than the uncontrolled exploration of random search.

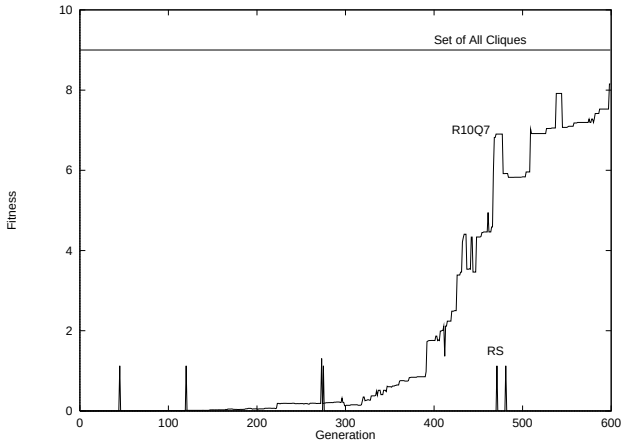

Figure 5.10: For the Fully Connect 16 graph, comparison of fitness per generation for underlying search engines of two Passive–Active searches: 1) GP with duplication of coding segments repair of chromosomes with a 10% return rate and 7 duplicates (R10Q7), and 2) random search (RS). All points represent the average of 10 runs. Also shown is the fitness associated with the set of all cliques.

5.5 Transfer of Control Knowledge

In my experiments to detect cliques with a Passive–Active collective memory search, I discovered while I can scale up the problem complexity, the effectiveness of the GP to guide the search agent in its exploration was initially in question. I conjecture that as the complexity rises, the need for improving the directed search on the part of the search agent, i.e., by the GP, will also rise. How then can I improve the exploration of the search space? My collective adaptation abstracts communication between both the agent classes and individuals both between and in those classes. Given the nature of my problem encoding, I rule out communication between individuals. Although it should be noted that this is effectively what the crossover process in the GP is accomplishing. I want to explore communication between the two agent classes: search and process.

Furthermore, I wish to restrict such communication to be only considered while the search agents are in the collective memory. There exists an analogy between the computational agent society and insect societies: the collective memory is the colony, the process agents are those insects which do not leave the confines of the colony, and the search agents are those insects which leave the colony. The search agent, as embodied by the chromosomes of the GP algorithm, leave the colony during fitness evaluation, return to it after the evaluations to deposit knowledge, the process agents manipulate both the new and old knowledge, and finally the search agent gets new directions. Until now, these directions came solely from the crossover process. It is at this point in the life cycle of the search agent that I

wish to introduce communication between the process and search agents.

Keeping with my desire to allow complex group level behavior to arise from simple individual behavior, I want to minimize the complexity that I add to the system. Thus I allow the process agents to direct the search agents by a transfer of knowledge during the crossover process. I allow a percentage of the crossover operations to be not between two individual chromosomes, but rather between one chromosome from the population and one created from the collective memory. Due to the stochastic nature of chromosome selection for crossover, I can have either none, one, or two chromosomes generated from the collective memory engage in any of the crossovers between two chromosomes. What I actually ensure is that a certain percentage of the chromosomes engaging in crossover come from the collective memory.

When such a transfer is to occur, I randomly select k candidate cliques (for the results reported in this section, I set $k = 2$) from the collective memory and construct an additional chromosome from them. The candidate cliques are duplicated in the chromosome following the repair strategy presented in Chapter 3 and in [Haynes, 1996]. The parameters for the duplication follow that for the duplication of coding segments used while repairing and replacing chromosomes during fitness evaluation; for our purposes this means seven duplicates of the set of k candidate cliques are added to the chromosome.

The graph I am testing is the hamming6-4.clq dataset from the DIMACS repository, which has 64 vertices, 704 edges, and a maximum clique size of 4. From a

brute force algorithm, we know that there are 464 cliques, with a maximum fitness of 1,597,424. The noteworthy parameters for the GP system are a maximum of 600 generations, a population size of 2000, probability of mutation is 0.1, probability of crossover is 0.9, a maximum depth of 4 for the initial generation, and a maximum depth of 10 after crossover or mutation[6]. For the clique covering, I set $\beta = 9$. In my experiments, I select crossover into the collective memory with a probability of $p = 0.1$ and then I randomly select $k = 2$ candidate cliques.

In Figure 5.11, I present the results of an experiment utilizing Active–Active collective adaptation which has a GP search engine with repair and duplication of chromosomes (AA-R10Q7). For comparative purposes, I also present my results with an equivalent Passive–Active collective adaptation system (PA–R10Q7). While it appears that the AA-R10Q7 does find the optimal solution, on the average it actually only attains 99.9% of the optimal solution. In 70% of the test cases, it does find the optimal solution.

My conjecture that knowledge transfer from process agents to search agents improves the overall search effort is well founded. Now we need to examine if there is an improvement in the performance of the underlying search engine, i.e., is there an improvement in the expression of the solution in the chromosomes? I report the performance of the underlying search engine for the Active–Active collective adaptation engine (R10Q7/AA) in Figure 5.12. I also present the results for the corresponding Passive–Active system (R10Q7/PA). It is clearly apparent that the

[6]My research is geared to improving the search process while holding the GP control parameters constant. As such, the reported parameters are not optimized for the task at hand and are actually taken from parameters I have used in another domain [Haynes *et al.*, 1995b].

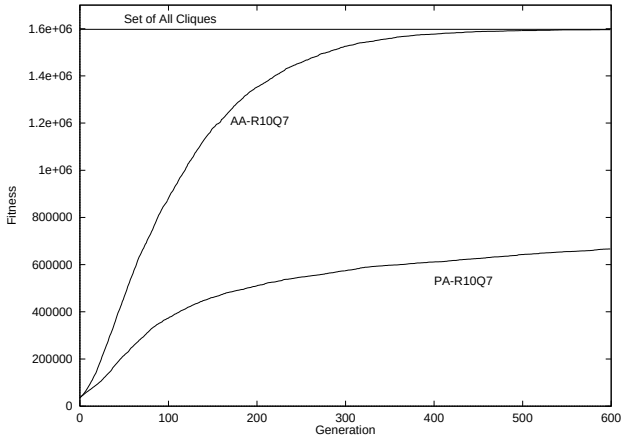

Figure 5.11: For the hamming6-4 graph, comparison of fitness per generation for Active–Active collective adaptation (AA-R10Q7) versus Passive–Active collective memory search with a simple collating agent (PA-R10Q7). The underlying search engine is genetic programming with duplication of coding segments repair of chromosomes with a 10% return rate and 7 duplicates. All points represent the average of 10 runs. Also shown is the fitness associated with the set of all cliques (Set of All Cliques).

knowledge transfer has increased the exploration of the chromosomes. While the optimal solution is still not close to being expressed inside the chromosomes, the increased exploration is magnified to allow the optimal solution to be expressed inside the collective memory.

5.5.1 Conclusions

By engaging the chromosomes in crossover with the collective memory, I transfer locally optimized knowledge back to the search agent. Hence, we can say the process agents are directing where in the search space the search agents should explore. While the improvement in the distributed search was not as great as the addition of local optimization by the process agents, it did significantly improve the exploration of the search agents and produced better results than no exchange

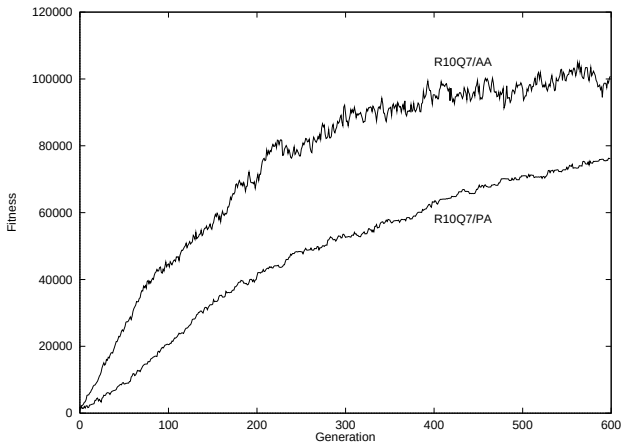

Figure 5.12: For the hamming6-4 graph, comparison of best fitness per generation for underlying search engine of Active–Active collective adaptation (R10Q7/AA) versus underlying search engine of Passive–Active collective memory search with a simple collating agent (R10Q7/PA). The underlying search engine in both cases is genetic programming with duplication of coding segments repair of chromosomes with a 10% return rate and 7 duplicates. The difference between the two systems is that the Active–Active collective adaptation transfers knowledge back into the GP subsystem. All points represent the average of 10 runs.

of control information.

5.6 Conclusions

5.6.1 Lessons in Scaling

In Figure 5.13 we present the big picture of what we have learned from detecting cliques as the problem complexity increases. In general we find that while the redundant collection and transfer of information from the search agents to the process agents via the collective memory are effective in exploring the search space, with just a little more effort on the part of either agent class, we can find the optimal solution. In particular, either the process agents must explore the rich areas of the search space as identified by the search agents or the search agents must allow themselves to be guided in their exploration by the process agents.

We also can see in Figure 5.13 that the addition of repair with duplication to the Passive-Active collective memory search is significant compared to no repair at all in the quality of the performance curve, i.e. PACM-R10Q7 versus PACM-R0. In Figure 5.2 of Section 5.2, we saw that this was not the case; i.e., previously the addition was not significant. As we also discussed in Section 5.4, as the complexity of the problem increases, the effectiveness of underlying search engine influences the effectiveness of the collective memory search. With low complexity problems, the collective memory search is able to exploit even weak underlying search engines.

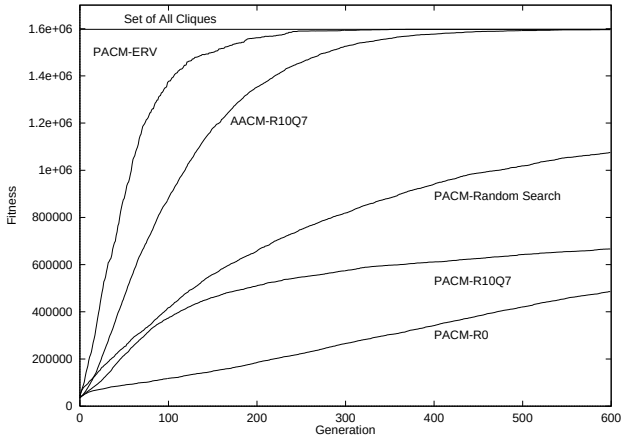

Figure 5.13: Comparison of fitness per generation for the five major search systems (unless explicitly noted, systems have an underlying GP engine with duplication of coding segments repair of chromosomes with a 10% return rate and 7 duplicates) : 1) Active–Active collective memory search (AACM-R10Q7), 2) Passive–Active collective memory search with an underlying GP engine with no repair (PACM-R0), 3) Passive–Active collective memory search (PACM-R10Q7), 4) Passive–Active collective memory search with an underlying Random Search engine (PACM-Random Search), and 5) Passive–Active collective memory search with process agents employing the ERV algorithm (PACM-ERV). All points represent the average of 10 runs. Also shown is the fitness associated with the set of all cliques (Set of All Cliques).

5.6.2 Applicability

The models of collective memory search are effective if building blocks of the solution can be identified. In clique detection, candidate cliques form the building blocks. The identification of building blocks in genetic programming is in general a difficult task [O'Reilly, 1995; Rosca and Ballard, 1996; Haynes, 1996]. In part this is due to the domain dependent nature of the alphabet, i.e. the members of the function and terminal sets[7]. As we saw in Chapter 3, the repair of chromosome by

[7]Building block are easier to find in GA chromosomes, but the typical string representation is the binary alphabet and of fixed length. As such, GA building blocks are at the structural level, whilst GP building blocks are at the semantical level [Haynes, 1996].

duplication of coding segments strategy holds promise in automating the detection of building blocks. If the system designer can identify function nodes that allow for addition of non-coding segments without changing the semantical meaning of the chromosome, the detection of building blocks can be automated.

CHAPTER VI

Collective Adaptation in Search Heuristics

6.1 Introduction

Royal Road functions manipulate the fitness landscape to provide controlled experiments into genetic algorithm (GA) theory [Mitchell *et al.*, 1992]. Variations of clique detection in a graph, e.g., finding both the max clique [Soule *et al.*, 1996] and the clique cover [Haynes, 1996], have been proposed as naturally occurring Royal Road functions. However, it has been shown that the clique domain does not serve as a Royal Road function for binary encoded GAs [Soule *et al.*, 1996]. If I vary the representation used in the chromosome, then the clique domain does satisfy the necessary criteria to be a Royal Road function for GA. Besides allowing a researcher to investigate the formation of building blocks, Royal Road functions can be used to test GAs against other paradigms [Mitchell *et al.*, 1992].

Soule and Foster have tried to positively correlate different graph families to various measures of GA hardness [Soule and Foster, 1997]. Soule and Foster have used a GA encoding to investigate the relation between graph characteristics and GA hardness [Soule and Foster, 1997]. They used graphs from the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) at Rutgers University. These graphs were generated as part of a DIMACS challenge on graph problems [Johnson and Trick, 1996]. Five measures of hardness were employed:

an epistasis measure [Davidor, 1991], a fitness distance correlation [Jones and Forrest, 1995]; graph size, graph density, and, relative clique size. They were unable to associate any of these measures with the difficulty of the GA to solve a particular graph.

I present a systematic approach to determining the relevant characteristics. I investigate the correlation between graph complexity and the ease of which various search heuristics, random search (RS), hill climbing (HC), simulated annealing (SA), and genetic algorithms, detect cliques in a given graph[1]. I investigate graphs for which simple search heuristics should easily determine both the max clique and the clique cover. I systematically vary both the number of cliques and their size in a graph. As expected, I find that the effectiveness decreases as I increase the number and size of cliques.

Within this chapter, I empirically validate that both duplication of coding segments and collective adaptation can be applied to all of the weak search heuristics. I investigate in detail changing the rate of repair for the GA heuristic and verify that a repair rate of $p = 0.1$ significantly improves the other weak search heuristics. For the collective adaptation, I examine two configurations: 1) the addition of collective memory with no duplication of coding segments, no process agents, and no transfer of partial solutions back into the chromosomes; and, 2) the addition of collective memory with the duplication of coding segments, process agents employing the ERV strong heuristic, and transfer of partial solutions back into

[1]The various search heuristics were presented in the Introduction. Note that the random search employed in this chapter is different than the one employed in Chapter 5

the chromosomes. As found in the earlier chapters, both duplication of coding segments and collective adaptation are instrumental in allowing the detection of better partial solutions.

6.2 FC Family of Graphs

I want to investigate the relationship between graph characteristics and GA hardness. Unlike Soule and Foster, I do not examine the families of graphs present in the DIMACS repository. I want to construct graphs for which we can characterize properties. With the FC family of graphs, I vary the number of cliques present and their cardinality. For any given graph, I restrict all cliques to have the same cardinality and consider sizes of 1, 2, 4, 8, 16, 32, and 64. I vary the number of cliques to be 1, 2, 4, 8, 16, 32, and 64. I consider the 49 graphs which result from the permutations of number and cardinality of cliques. I label a graph by first the number of cliques and then the cardinality. Thus fc2-64.clq refers to a graph with 2 cliques each of cardinality 64. With the FC family of graphs, I have set the following properties: 1) the number of cliques is known beforehand; 2) the cardinality of each clique is known beforehand, so the maximal clique size is also known; and, 3) the cliques are disjoint. I do not want to mislead the reader, because of property 3, the clique cover and max clique for each of these graphs can be found via greedy algorithms in linear time.

If an heuristic is aware of these properties, especially the last one, they can be exploited such that each of the three NP–complete problems become P:

Partition into cliques: Since we know that the cliques are disjoint, it follows immediately that this problem is solvable in polynomial time as illustrated by the greedy algorithm in Figure 6.1.

```
Given V, the set of vertices
Set P, the partition, to be empty
while (vertex v in V) {
      Set C, the current clique, to be empty
      Randomly select v, without replacement, a vertex from V
      Insert v into C
      Set W, the examined vertices, to be empty
      while (vertex u in V) {
          Randomly select u, without replacement, a vertex from V
          if ( ( union(u, C) ) is a complete subgraph ) then {
              Insert u into C
          } else {
              Insert u into W
          }
      }
      Insert C into P
      Insert W into V
}
```

Figure 6.1: Greedy algorithm for partition into cliques.

Covering by cliques: The greedy algorithm in Figure 6.1 also finds the clique cover.

Clique: After using the greedy algorithm in Figure 6.1 to determine the partition P, determine the cardinality of each clique in P, and return the maximum.

However, in general the greedy algorithm in Figure 6.1 will *not* run in polynomial time: the determination of the disjointness of the clique set is itself NP–complete and is actually the problem of Partition into cliques.

If examine the characteristics of a Royal Road function as put forth in [Mitchell *et al.*, 1992]:

1) All of the desired building blocks are known in advance.

2) The landscape can be varied systematically.

3) The global optimum, and all local optimum, can be enumerated.

we see that this family of graphs satisfy the conditions to make it a Royal Road function[2]. Also, it should be remembered that the original Royal Road functions were meant to be problems which the GA should find easy and which would allow the comparison of performance across multiple heuristics.

With the example graph shown in Figure 6.2, I can list all of the building blocks:

$$
\begin{aligned}
C = \{ \quad & \{0,1\},\{0,2\},\{0,3\},\{1,2\},\{1,3\},\{2,3\}, \\
& \{4,5\},\{4,6\},\{4,7\},\{5,6\},\{5,7\},\{6,7\}, \\
& \{0,1,2\},\{0,1,3\},\{0,2,3\},\{1,2,3\}, \\
& \{4,5,6\},\{4,5,7\},\{4,6,7\},\{5,6,7\}, \\
& \{0,1,2,3\},\{4,5,6,7\}\}.
\end{aligned}
$$

Since we know all of the candidate cliques, we can calculate the fitness for all interesting combinations of building blocks.

My goal in designing this family of graphs is to have graphs which are easy

[2]The connection between Royal Road functions and clique detection is explored in Appendix 5.

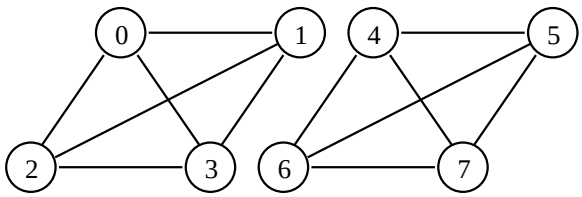

Figure 6.2: Example graph, consisting of 2 fully connected cliques of cardinality 4.

for hill climbers to solve and for which pruning heuristics will not work[3]. To facilitate hill climbers, I labeled the vertices such that all vertices in a clique were adjacent. The easiest pruning heuristics for max clique are to 1) discard vertices in a candidate clique which have less incidence than all other vertices in that candidate clique and, 2) discard those candidate cliques for which each node has less incidence than the cardinality of the maximum clique found so far. Since all vertices belong to a clique and all cliques have the same cardinality, neither of these pruning heuristics will succeed in reducing the size of the space of partial solutions.

6.3 Testing Hardness

In Table 6.1, I present 5 different hardness measures as applied to the 49 different benchmark graphs. If we first define, $n_c(G)$ to be the number of cliques in G and $c_c(G)$ to be the cardinality of the cliques in G, then for a graph G, I define:

$h_s(G)$ to be the hardness as measured by the graph size, which is simply the

[3]An irony of Royal Road functions is while they are designed to facilitate the hierarchical solving of problems via building blocks, in practice they hinder the integration of building blocks, e.g., hitchhiking [Mitchell *et al.*, 1992]. With clique detection, the space of partial solutions can become too large for building blocks to be effectively combined.

number of vertices in G. For this graph family,

$$h_s(G) = n_c(G) * c_c(G).$$

If $h_s(G_i) > h_s(G_j)$, then G_i should be harder to solve than G_j. Also note that graphs which share a bottom to top diagonal, e.g., fc4-1.clq, fc2-2.clq, and fc1-4.clq, will have the same value for $h_s(G)$.

$h_d(G)$ to be the hardness as measured by the graph density. If G were maximally complete, there would be an edge between each vertex in G, totalling C_2^N edges. Thus, $h_d(G)$ is the ratio of actual edges in G to the total possible. For this graph family,

$$h_d(G) = \frac{n_c(G) * C_2^{c_c(G)}}{C_2^N}.$$

If $h_d(G_i) < h_d(G_j)$, then G_i should be harder to solve than G_j. Also note that graphs which share a bottom to top diagonal, e.g., fc4-1.clq, fc2-2.clq, and fc1-4.clq, will have the same value for $h_d(G)$.

$h_{mcs}(G)$ to be the hardness as measured by the ratio of the max clique size to the graph size. For this graph family,

$$h_{mcs}(G) = \frac{c_c(G)}{h_s(G)}.$$

Notice that $c_c(G)$ will cancel out in both the numerator and the denomina-

tor, resulting in

$$h_{mcs}(G) = \frac{1}{n_c(G)}.$$

$h_{mce}(G)$ to be the hardness as measured by the ratio of max clique size to the number of edges in G. For this graph family,

$$h_{mce}(G) = \frac{c_c(G)}{n_c(G) * C_2^{c_c(G)}}.$$

If $h_{mce}(G_i) < h_{mce}(G_j)$, then G_i should be harder to solve than G_j. Also note that graphs which occupy a row will have the same value for $h_{mce}(G)$.

$h_{nce}(G)$ to be the hardness as measured by the ratio of the number of cliques to the number of edges in G. For this graph family,

$$h_{nce}(G) = \frac{n_c(G)}{n_c(G) * C_2^{c_c(G)}}.$$

Again, by canceling, we get

$$h_{nce}(G) = \frac{1}{C_2^{c_c(G)}}.$$

If $h_{nce}(G_i) < h_{nce}(G_j)$, then G_i should be harder to solve than G_j. Also note that graphs which occupy a column will have the same value for $h_{nce}(G)$.

For the FC family of graphs, $h_{mcs}(G)$ is independent of the max clique size and $h_{nce}(G)$ is independent of the number of cliques.

		1	2	4	8	16	32	64
1	h_s	1	2	4	8	16	32	64
	h_d	1	2	0.6667	0.2857	0.1333	0.06452	0.03175
	h_{mcs}	1	1	1	1	1	1	1
	h_{mce}	1	1	1	1	1	1	1
	h_{nce}	1	1	0.1667	0.03571	0.008333	0.002016	0.000496
2	h_s	2	4	8	16	32	64	128
	h_d	2	0.6667	0.2857	0.1333	0.06452	0.03175	0.01575
	h_{mcs}	1	0.1667	0.2143	0.2333	0.2419	0.246	0.248
	h_{mce}	0.5	0.5	0.5	0.5	0.5	0.5	0.5
	h_{nce}	1	1	0.1667	0.03571	0.008333	0.002016	0.000496
4	h_s	4	8	16	32	64	128	256
	h_d	0.6667	0.2857	0.1333	0.06452	0.03175	0.01575	0.007843
	h_{mcs}	0.1667	0.03571	0.05	0.05645	0.05952	0.06102	0.06176
	h_{mce}	0.25	0.25	0.25	0.25	0.25	0.25	0.25
	h_{nce}	1	1	0.1667	0.03571	0.008333	0.002016	0.000496
8	h_s	8	16	32	64	128	256	512
	h_d	0.2857	0.1333	0.06452	0.03175	0.01575	0.007843	0.003914
	h_{mcs}	0.03571	0.008333	0.0121	0.01389	0.01476	0.0152	0.01541
	h_{mce}	0.125	0.125	0.125	0.125	0.125	0.125	0.125
	h_{nce}	1	1	0.1667	0.03571	0.008333	0.002016	0.000496
16	h_s	16	32	64	128	256	512	1024
	h_d	0.1333	0.06452	0.03175	0.01575	0.007843	0.003914	0.001955
	h_{mcs}	0.008333	0.002016	0.002976	0.003445	0.003676	0.003792	0.003849
	h_{mce}	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625
	h_{nce}	1	1	0.1667	0.03571	0.008333	0.002016	0.000496
32	h_s	32	64	128	256	512	1024	2048
	h_d	0.06452	0.03175	0.01575	0.007843	0.003914	0.001955	0.000977
	h_{mcs}	0.002016	0.000496	0.0007382	0.0008578	0.0009173	0.000947	0.0009618
	h_{mce}	0.03125	0.03125	0.03125	0.03125	0.03125	0.03125	0.03125
	h_{nce}	1	1	0.1667	0.03571	0.008333	0.002016	0.000496
64	h_s	64	128	256	512	1024	2048	4096
	h_d	0.03175	0.01575	0.007843	0.003914	0.001955	0.000977	0.0004884
	h_{mcs}	0.000496	0.000123	0.0001838	0.000214	0.0002291	0.0002366	0.0002404
	h_{mce}	0.01563	0.01563	0.01563	0.01563	0.01563	0.01563	0.01563
	h_{nce}	1	1	0.1667	0.03571	0.008333	0.002016	0.000496

Table 6.1: Hardness factors for 49 fully connected graphs. Key: h_s is the graph size; h_d is the graph density; h_{mcs} is the ratio of max clique size to graph size; h_{mce} is the ratio of max clique size to number of edges; and, h_{nce} is the ratio of number of cliques to number of edges.

6.4 Experimental Setup

For the fitness function, I set $\alpha = 10$, $\beta = 9$, and $\gamma = 0$. For the GA and GP experiments the relevant parameters are as follows: the probability that a group marker would appear was $p_m = 0.1$, double point crossover, with a rate of 0.7 per pair of parents, mutation with a probability of 0.01 per bit, maximum number of generations of 256, and the population size was 64. I used linear scaling, with the maximum expected offspring being 2.0 [Goldberg, 1989]. The chromosomes for the GA were limited to 256 positions and those for the GP were fixed to a max depth size of 10, i.e., 2047 max nodes. The other heuristics had a probability of 0.01 per bit for mutation. Furthermore, I allow each of the non-GA heuristics to have 16,384 evaluations (generations multiplied by population size). For all of these heuristics, I report the results in terms of an epoch, i.e., each 64 evaluations.

With the given chromosome length, it is not possible to represent the clique covers for all of the graphs. However, it is possible to represent all of the max cliques for all of the graphs. Also, due to the fully connected nature of the cliques and each clique having the same cardinality, I could utilize a binary encoding and reduce the problem to the GA-easy 1s problem. However, I am trying to discover how these graphs are representative of naturally occurring graphs. As such, I pick an encoding that is not aware of the nature of the graphs. If we consider that a clique with cardinality 64 has $C_{32}^{64} = 1.83 \times 10^{10}$ possible candidate cliques of cardinality 32, we see that we must pick a reasonable chromosome length.

In each experiment, I present the results of 20 runs by each of the heuristics.

I had two observations I could present: the average of the last generation and the maximum average over all generation. I chose the latter result so that the results would not be biased towards hill climbers and away from random search. Instead of examining the fitness values, as I did in the previous chapters, I am interested in the max clique found and the ratio of the cover of cliques found to the clique cover. As both the max clique and clique cover are known for the FC family of graphs[4], I can state the search power of an heuristic via observing either variable.

6.5 The Base Heuristics

In Appendix A, I present the average of 20 runs for the various weak search heuristics for the FC family of graphs. The heuristics implement neither duplication of coding segments nor collective adaptation. I report the average of the max clique found for the 49 graphs in Table A.1 and the average of the clique cover in Table A.2. I am interested in determining the capabilities of each heuristic, i.e., which of the heuristics are better than the others. I analyze the max clique found, the clique cover, and the total time to run the experiment. The means and standard deviations are presented in Table 6.2.

Ignoring the variations produced by the different graph variations, we run a one-way ANOVA to test the hypothesis, using a significance level of $\alpha = 0.01$, that

H_0 All heuristics means are the same.

[4]But not for the DIMACS graphs.

	Max Clique	Clique Cover	Time
RS	4.42 (4.84)	0.22 (0.37)	21.58 (3.10)
HC	6.94 (7.90)	0.39 (0.46)	37.26 (18.97)
SA	6.83 (7.97)	0.36 (0.45)	38.41 (21.92)
GA	3.88 (4.42)	0.27 (0.41)	39.03 (17.28)
GP	3.16 (2.06)	0.23 (0.40)	207.31 (165.48)

Table 6.2: Base heuristics, mean of 20 runs, std. dev. in parenthesis.

H_a At least 2 heuristics have different means.

We reject H_0 for each of the max clique found, the clique cover, and the total time to run the experiment. We perform the Scheffe follow-up test to determine significant differences in means at a level of $\alpha = 0.01$.

Over the FC family of graphs, we find:

- HC and SA are better at detecting the max clique than RS, GA, and GP.
- RS is better at detecting the max clique than GA and GP.
- A Duncan follow-up test with $\alpha = 0.01$ reveals GA is better at detecting the max clique than GP and there is no significant difference between the GA and RS.
- HC and SA are better at detecting the clique cover than the other heuristics.
- GP takes significantly longer to run than all others.
- HC, SA, and GA take significantly longer to run than RS.

As the fitness evaluation code is the same for each heuristic, the time difference for the GP must be a result of having to recursively evaluate each node in the parse

tree. Since the SA approaches RS when the temperature is high and becomes a hill climber when the temperature decreases, it should be no surprise that HC and SA perform similarly when no local optima are present.

With the given representation, we expect building blocks to evolve and we do not expect the evolution of hitchhikers; for the clique cover, there are no hitchhikers. The poor performance of all heuristics can be attributed to the large search spaces; to integrate partial solutions, they must first be found. While increasing the chromosome length will boost the search, there exists a limit to the chromosome length for even moderately sized graphs.

6.5.1 Hardness

For each graph and each heuristic, we calculated the correlation coefficient and examined scatter plots. There was no correlation between the performance of any of the heuristics and any of the various hardness measures. If $h_{mce}(G)$ were a good measure, then as we increase the number of cliques in a graph while holding the cardinality constant, the solution quality should also stay constant. $h_s(G)$ and $h_d(G)$ are also poor measures of hardness. For example, fc8-16.clq and fc16-8.clq should be equally difficult to solve, yet we see in Table A.1 that HC comes closer to the max clique for fc8-16.clq (found 7.7 out of 8) than fc16-8.clq (found 5 out of 16). Also, fc8-16.clq finds 1.5×10^{-7} of the clique cover versus the 0.0011 of the clique cover for fc16-8.clq. Finally, if $h_{nce}(G)$ were a good measure of hardness, then as we increased the cardinality of the cliques in a graph while holding the number of cliques constant, the solution quality should also stay constant. From

observing the data in both Tables A.1 and A.2, it appears the actual hardness is a linear function of the clique size and the number of cliques, i.e., increasing both makes the resultant graph harder to solve.

We see that as we increase either cardinality or number of cliques in the graphs, performance decreases. Also, for a cardinality of 64, chromosome length does not seem to be a factor. For example, the numbers of vertices in both fc1-64.clq and fc2-64.clq are less than half the chromosome length. The average max clique found for HC was 41 and 29 respectively. The clique cover of the was 1.7×10^{-21} and 2.1×10^{-51} respectively. Even for complete solutions which could be expressed in the chromosome, the space of partial solutions can not be effectively represented in the chromosome.

6.5.2 Conclusions

I have examined the characteristics which make clique detection a good domain for testing various weak search heuristics: random search, hill climbing, simulated annealing, genetic algorithms, and genetic programming. Clique detection is difficult for all of these heuristics. I systematically vary the complexity of graphs and discover that neither relative maximum clique size, relative number of cliques, graph density, nor graph size are good measures of hardness for these search heuristics. While I can improve the heuristics by adding in either local search or a collective memory, I still would like to determine an accurate measure of hardness for either a given graph or family of graphs.

6.6 Duplication of Coding Segments

Except for the GP, each of the search methods utilize the same encoding for the chromosomes. Even though a tree format is not used, coding and non–coding segments can be determined, which will allow for repair via duplication of coding segments. Since each chromosome is of a fixed length, unlike a GP chromosome, I must ensure that each position is filled during the duplication process. Thus, unlike the GP algorithm, this heuristic inserts as many copies of the coding segments as possible. If there is not enough room for an additional entire copy of the coding segments, then only enough material as necessary is copied in for the last duplication.

The addition of duplicates of the coding segment drastically changes the nature of the search heuristics. With GA and GP, the repair and duplication is done before selection, thus there is no guarantee that a repaired chromosome will contribute genetic material towards the next generation. However, with RS, the repaired chromosome will be selected as the next candidate solution. As long as the fitness of the repaired chromosome, which had already been perturbed, is greater than that of the previous chromosome, it will be selected for both HC and SA. Finally, even if it has a lesser fitness, SA may still accept it as the new candidate solution.

Also, RS is no longer truly random; a backup copy of the coding segment can be kept, making RS a modified version of HC. Since RS perturbs the chromosome via the GA mutation operator, i.e., it does not completely regenerate the chro-

mosome, one or more copies of the coding segments can remain intact. If the new chromosome has a higher fitness, then it is selected and only if all of the copies of the coding segment are changed, and destructively at that, will a lower fitness chromosome be selected. On the average, after duplication of coding segments has taken place, the fitness of the candidate solution should tend to either stay the same or increase.

Since I do not control the number of duplicates injected into the chromosome, the key control parameter is the rate at which chromosomes are repaired. In Section 6.6.1 I present the results of adding duplication of coding segments to the various search heuristics and with a repair rate of $p = 0.1$. In Section 6.6.2 I present the results duplication of coding segments for GA chromosomes having repair rates of 0%, 0.5%, 1%, 5%, 10%, 25% and 50%.

6.6.1 The Heuristics

In Appendix B, I present the average of 20 runs for the various weak search heuristics for the FC family of graphs. The heuristics implement duplication of coding segments, with a repair rate of $p = 0.1$, but not collective adaptation. I report the average of the max clique found for the 49 graphs in Table B.1 and the average of the clique cover in Table B.2. I am interested in determining the capabilities of each heuristic, i.e., which of the heuristics are better than the others. I also want to know for which heuristics is the duplication of coding segments significant over the base heuristic. I analyze the max clique found, the clique cover, and the total time to run the experiment. The means and standard

	Max Clique	Clique Cover	Time
RRS	7.12 (5.70)	0.35 (0.44)	36.94 (17.87)
RHC	9.40 (8.56)	0.39 (0.46)	32.71 (16.26)
RSA	9.48 (8.56)	0.39 (0.46)	46.45 (22.90)
RGA	6.47 (7.73)	0.35 (0.45)	75.24 (23.69)
RGP	3.11 (1.91)	0.26 (0.41)	269.82 (202.04)

Table 6.3: Duplication of coding segment heuristics, mean of 20 runs, std. dev. in parenthesis.

deviations are presented in Table 6.3.

Ignoring the variations produced by the different graph variations, we run a one-way ANOVA to test the hypothesis, using a significance level of $\alpha = 0.01$, that

H_0 All heuristics means are the same.

H_a At least 2 heuristics have different means.

For this test, I utilize both the base and duplication of coding segments versions of each heuristic. We reject H_0 for each of the max clique found, the clique cover, and the total time to run the experiment. We perform the Scheffe follow-up test to determine significant differences in means at a level of $\alpha = 0.01$.

Over the FC family of graphs, we find:

- The addition of duplication of coding segments, with a repair rate of $p = 0.1$ to a heuristic, makes a heuristic better at detecting the max clique than the base heuristic. (Except for GP.)

- RRS is better at detecting the max clique than RGP, but not RGA.

- The addition of duplication of coding segments, with a repair rate of $p = 0.1$ to a heuristic, makes no difference for detecting the clique cover than the base heuristic. (Except for RS.)

- A Duncan follow-up test with $\alpha = 0.01$ reveals RGA is better at detecting the clique cover than GA.

- RGP takes significantly longer to run than all others.

- RGA takes significantly longer to run than RRS, RSA, and RHC.

Adding duplicates of coding segments allows both the RGA and RRS algorithms to detect the clique cover at the same level as RHC and RSA. The duplication of coding segments increased the number of partial solutions for each of these heuristics, but only RHC and RSA were better at expanding these partial solutions, i.e., they found higher cardinality max cliques.

The addition of duplication of coding segments improves all heuristics *except* GP! While I shall revisit this point later, I conjecture that the subtree crossover and mutation of GP is less powerful than the operators used by the other heuristics. I am also asking a different question here than in Chapter 3; the original experiments asked if the duplication of coding segments were significant in reducing the time until the best solution was found and these experiments determine if the maximum solution found is significantly different. These results follow those with the hamming6-4.clq graph for the GP; duplication of coding segments alone did not scale well with increased complexity.

	Max Clique	Clique Cover	Time
R0	3.88 (4.42)	0.27 (0.41)	39.03 (17.28)
R05	5.77 (5.67)	0.33 (0.44)	67.24 (16.88)
R1	6.09 (6.25)	0.35 (0.44)	66.02 (15.80)
R5	6.51 (7.45)	0.35 (0.44)	71.90 (20.35)
R10	6.47 (7.73)	0.35 (0.45)	75.24 (23.69)
R20	6.55 (8.08)	0.36 (0.45)	76.56 (26.93)
R25	6.56 (8.23)	0.36 (0.45)	77.27 (27.86)
R50	6.54 (8.28)	0.36 (0.45)	79.15 (30.61)

Table 6.4: Varying the repair rate for the GA heuristic, mean of 20 runs, std. dev. in parenthesis.

6.6.2 Varying GA Repair Rate

In Appendix C, I present the average of 20 runs for the GA heuristics, with various repair rates, for the FC family of graphs. I consider the following eight repair rates: R0 ($p = 0.0$), R05 ($p = 0.005$), R1 ($p = 0.01$), R5 ($p = 0.05$), R10 ($p = 0.10$), R20 ($p = 0.20$), R25 ($p = 0.25$), and R50 ($p = 0.50$). I report the average of the max clique found for the 49 graphs in Table C.1 and the average of the clique cover in Table C.2. I am interested in determining whether the addition of repair and duplication of coding segments are significant. I analyze the max clique found, the clique cover, and the total time to run the experiment. The means and standard deviations are presented in Table 6.3.

Ignoring the variations produced by the different graph variations, we run a one-way ANOVA to test the hypothesis, using a significance level of $\alpha = 0.01$, that

H_0 All repair rate means are the same.

H_a At least 2 repair rates have different means.

We reject H_0 for each of the max clique found, the clique cover, and the total time to run the experiment. We perform the Scheffe follow-up test to determine significant differences in means at a level of $\alpha = 0.01$.

Over the FC family of graphs, we find:

- The addition of duplicates of the coding segments results in the detection of larger max cliques than no duplication at all.
- Repair rates of $p = 0.20$, $p = 0.25$, and $p = 0.50$ result in better clique covers than all other repair rates.
- A Duncan follow-up test with $\alpha = 0.01$ reveals the addition of duplicates of the coding segments results in the detection of more of the clique cover than no duplication at all.
- The addition of duplicates of the coding segments results in longer run times than no duplication at all.
- Repair rates of $p = 0.20$, $p = 0.25$, and $p = 0.50$ take longer to run than all other repair rates.
- Repair rates of $p = 0.05$ and $p = 0.10$ take longer to run than $p = 0.0$, $p = 0.005$, and $p = 0.1$.
- A Duncan follow-up test with $\alpha = 0.01$ reveals that $p = 0.50$ takes longer to run than all other repair rates.

At worst, the repair and duplication algorithm forces the runtime to approximately double in time. As the performance results show, this penalty is well worth the effort. Also notice that since all chromosomes must be translated from genotype to phenotype, the bulk of the time must be spent in translating back from phenotype to genotype. As pointed out in the last section, I am ignoring the time taken to find the optimal solution. As Figures C.1-C.3 show for the fc4–8.clq graph, the rate at which the best fitness (Figure C.1), generational clique cover (Figure C.2), and max clique(Figure C.3) increase are influenced by the repair rate.

6.7 Collective Adaptation

The next step in our study of the weak search heuristics is to add collective adaptation. From the earlier research with GP, we saw that adding duplication of coding segments to collective adaptation was better than strict collective adaptation. As such, I first consider adding a collective memory, a process agent which collates, and the duplication of coding segments. I then investigate the various configurations of collective adaptation on just the GA heuristic. Finally, I examine collective adaptation on just the GP heuristic.

When comparing an observation for a heuristic which employs collective memory against a heuristic which does not, I simply compare the collective memory observation to the equivalent chromosomal observation. This comparison is fair in the sense that the best values found in the chromosomes are compared against

	Max Clique	Max Clique (CM)	Clique Cover	Clique Cover (CM)	Time
CMRS	7.12 (5.70)	7.72 (6.48)	0.35 (0.44)	0.40 (0.46)	1985.87 (10384.92)
CMHC	9.40 (8.56)	10.13 (9.42)	0.39 (0.46)	0.42 (0.46)	3893.12 (20262.36)
CMSA	9.48 (8.56)	10.22 (9.50)	0.39 (0.46)	0.42 (0.46)	4029.34 (18027.44)
CMGA	6.47 (7.73)	9.70 (11.43)	0.35 (0.45)	0.38 (0.45)	98.41 (313.07)
CMGP	3.10 (1.91)	10.45 (12.63)	0.25 (0.41)	0.37 (0.44)	266.15 (198.24)

Table 6.5: Collective Adaptation and duplication of coding segment heuristics, mean of 20 runs, std. dev. in parenthesis.

those in the collective memory. I also make sure to compare chromosomal values against each other.

6.7.1 The Heuristics

In Appendix D, I present the average of 20 runs for the various weak search heuristics for the FC family of graphs. The heuristics implement both duplication of coding segments, with a repair rate of $p = 0.1$, and collective adaptation. I report the average of the max clique found for the 49 graphs in Table D.1 (Table D.3 for inside the collective memory) and the average of the clique cover in Table D.2 (Table D.4 for inside the collective memory). I am interested in determining the capabilities of each heuristic, i.e., which of the heuristics are better than the others. I also want to know for which heuristics is collective adaptation significant over the base heuristic. I analyze the max clique found (both in the chromosome and the collective memory), the clique cover (both in the chromosome and the collective memory), and the total time to run the experiment. The means and standard deviations are presented in Table 6.5.

Ignoring the variations produced by the different graph variations, we run a

one-way ANOVA to test the hypothesis, using a significance level of $\alpha = 0.01$, that

H_0 All heuristics means are the same.

H_a At least 2 heuristics have different means.

For this test, I utilize both the collective adaptation with duplication of coding segments versions of each heuristic. We reject H_0 for each of the max clique found (both in the chromosome and the collective memory), the clique cover (both in the chromosome and the collective memory), and the total time to run the experiment. We perform the Scheffe follow-up test to determine significant differences in means at a level of $\alpha = 0.01$.

Over the FC family of graphs, we find:

- CMHC and CMSA are better at finding the max clique in the chromosome than all others.
- CMRS and CMGA are better at finding the max clique in the chromosome than CMGP.
- All heuristics are better at finding the max clique in the chromosome than CMRS.
- All heuristics are better at finding the clique cover in the chromosome than CMGP.
- All heuristics are equal at finding the clique cover in the collective memory.

- CMHC and CMSA take significantly longer to run than all others.
- A Duncan follow-up test with $\alpha = 0.01$ reveals CMRS takes significantly longer to run than both CMGA and CMGP.

Notice while CMRS is better at representing solutions inside the chromosome than is CMGP, CMGP is better at representing them inside the collective memory.

I also want to know for which heuristics is collective adaptation significant over the base heuristic. I analyze the max clique found (both in the chromosome and the collective memory), the clique cover (both in the chromosome and the collective memory), and the total time to run the experiment. For this test, I utilize both the base and the collective adaptation with duplication of coding segments versions of each heuristic. We reject H_0 for each of the max clique found, the clique cover, and the total time to run the experiment. We perform the Scheffe follow-up test to determine significant differences in means at a level of $\alpha = 0.01$.

Over the FC family of graphs, we find the addition of collective adaptation and the duplication of coding segments, with a repair rate of $p = 0.1$ to a heuristic

- make a heuristic better at detecting the max clique inside the chromosome than the base heuristic. (Except for GP.)
- make a heuristic better at detecting the max clique inside the collective memory than than the base heuristic inside the chromosome.

6.7.2 Genetic Algorithm

The truly interesting experiments to be conducted involve the addition of collective memory. For the heuristics I consider in this section, I utilize a collective adaptation system employing a GA as the weak search engine. I will present the results of the following experiments.

GA: The base GA system with neither duplication of coding segments nor collective memory.

CM: Collective memory is added, along with duplication of coding segments. However, the process agent just collates.

PA: Add an additional process agent employing the Expand by Random Vertex (ERV) heuristic presented in Section 5.3.

AA: Add communication back to the chromosomes in the form of crossover into the collective memory as defined in Section 5.5.

LG64: Two additional process agents are added to employ a Merge Random Clique to All, (MRCA), similar to the Merge Adjacent Candidate Cliques, MA, heuristic presented in Section 5.3. Instead of trying to merge adjacent cliques, MRCA randomly selects one clique in the collective memory and attempts to merge it with all cliques in the collective memory. They utilize the same heuristics employed by the process agents.

LG128: This experiment increases the population size from 64 to 128.

	Max Clique	Max Clique (CM)	Clique Cover	Clique Cover (CM)	Time
CA	3.88 (4.42)	3.82 (4.34)	0.27 (0.41)	0.27 (0.41)	39.03 (17.29)
PA	6.45 (7.75)	6.45 (7.75)	0.33 (0.44)	0.34 (0.45)	108.86 (448.03)
AA	6.47 (7.73)	9.70 (11.43)	0.35 (0.45)	0.38 (0.45)	98.41 (313.06)
LG64	11.70 (14.93)	11.71 (14.93)	0.47 (0.46)	0.47 (0.46)	73.13 (21.37)
LG128	11.85 (15.26)	11.85 (15.27)	0.52 (0.47)	0.52 (0.47)	147.11 (43.96)
MG64	18.09 (21.18)	18.09 (21.18)	0.65 (0.41)	0.67 (0.41)	86.46 (46.22)
MG128	18.14 (21.25)	18.14 (21.25)	0.70 (0.38)	0.74 (0.37)	193.58 (168.61)

Table 6.6: Collective Adaptation applied to GA, mean of 20 runs, std. dev. in parenthesis.

MG64: The same as LG64, except the chromosomes also engage in local search at the end of their evaluation.

MG128: The same as LG128, except the chromosomes also engage in local search at the end of their evaluation.

In Appendix E, I present the average of 20 runs for the GA with the various collective adaptation strategies outlined above and for the FC family of graphs. I report the average of the max clique found for the 49 graphs in Table E.1 (Table E.3 for inside the collective memory) and the average of the clique cover in Table E.2 (Table E.4 for inside the collective memory). I am interested in determining the cumulative addition of extra capabilities extends the search. I analyze the max clique found (both in the chromosome and the collective memory), the clique cover (both in the chromosome and the collective memory), and the total time to run the experiment. The means and standard deviations are presented in Table 6.6.

Ignoring the variations produced by the different graph variations, we run a one-way ANOVA to test the hypothesis, using a significance level of $\alpha = 0.01$, that

H_0 All heuristics means are the same.

H_a At least 2 heuristics have different means.

We reject H_0 for each of the max clique found (both in the chromosome and the collective memory), the clique cover (both in the chromosome and the collective memory), and the total time to run the experiment. We perform the Scheffe follow-up test to determine significant differences in means at a level of $\alpha = 0.01$.

Over the FC family of graphs, we find:

- MG64 and MG128 are better at finding the max clique in the chromosome than all other variants.
- LG64 and LG128 are better at finding the max clique in the chromosome than CA, PA, and AA.
- A Duncan follow-up test with $\alpha = 0.01$ reveals that CA and PA are better at finding the max clique in the chromosome than AA.
- MG64 and MG128 are better at finding the max clique in the collective memory than all other variants.
- LG64, LG128, CA, and PA are better at finding the max clique in the chromosome than AA.

- A Duncan follow-up test with $\alpha = 0.01$ reveals that LG64 and LG128 are better at finding the max clique in the chromosome than CA and PA.
- A Duncan follow-up test with $\alpha = 0.01$ reveals that PA are better at finding the max clique in the chromosome than CA.
- MG64 and MG128 are better at finding the max clique in the chromosome than all other variants.
- LG64 and LG128 are better at finding the max clique in the chromosome than CA, PA, and AA.
- CA and PA are better at finding the max clique in the chromosome than AA.
- A Duncan follow-up test with $\alpha = 0.01$ reveals that LG128 is better at finding the max clique in the chromosome than LG64.
- MG64 and MG128 are better at finding the max clique in the collective memory than all other variants.
- LG64 and LG128 are better at finding the max clique in the collective memory than CA, PA, and AA.
- CA and PA are better at finding the max clique in the collective memory than AA.
- A Duncan follow-up test with $\alpha = 0.01$ reveals that MG128 is better at finding the max clique in the chromosome than MG64.

- A Duncan follow-up test with $\alpha = 0.01$ reveals that LG128 is better at finding the max clique in the chromosome than LG64.
- MG128 takes significantly longer to run than all others.
- LG128 takes significantly longer to run than all others except MG128 and PA.
- PA, AA, and MG64 take significantly longer to run than CA and LG64.
- A Duncan follow-up test with $\alpha = 0.01$ reveals that LG128 also takes significantly longer to run than PA.
- A Duncan follow-up test with $\alpha = 0.01$ reveals that PA also takes significantly longer to run than LG64.

6.7.3 Conclusions

Each heuristic presented in this section increases the cooperation employed in the system and as we increase the cooperation, we almost always see an increase in performance. Large increases are seen in both the clique cover and the max clique, indicating that not only do these heuristics home in on the max clique, they are able to distribute the search process over all of the cliques in the graph.

6.8 Genetic Programming

As we saw in the previous sections, the base GP did not fare as well as the other algorithms. In this section, I will repeat the basic experiments of the earlier

chapters to see if the results hold with the FC family of graphs. I will present the results of the following experiments:

B64: The base GP system with neither duplication of coding segments nor collective memory. The population size is 64.

B256: The base GP system with neither duplication of coding segments nor collective memory. The population size is 256.

R64: The base GP system with duplication of coding segments and a repair rate of $p = 0.1$. The population size is 64.

R256: The base GP system with duplication of coding segments and a repair rate of $p = 0.1$. The population size is 256.

C64: Collective memory is added, along with duplication of coding segments. However, the process agent just collates. The population size is 64.

C256: Collective memory is added, along with duplication of coding segments. However, the process agent just collates. The population size is 256.

X64: Add communication back to the chromosomes in the form of crossover into the collective memory as defined in Section 5.5. The population size is 64.

X256: Add communication back to the chromosomes in the form of crossover into the collective memory as defined in Section 5.5. The population size is 256.

	Max Clique	Max Clique (CM)	Clique Cover	Clique Cover (CM)	Time
B64	3.16 (2.06)	3.16 (2.06)	0.23 (0.40)	0.23 (0.40)	207.31 (165.48)
B256	3.74 (2.51)	3.74 (2.51)	0.25 (0.41)	0.25 (0.41)	1036.88 (893.04)
R64	3.11 (1.91)	3.11 (1.91)	0.26 (0.41)	0.26 (0.41)	269.82 (202.04)
R256	3.60 (2.36)	3.60 (2.36)	0.28 (0.43)	0.28 (0.43)	1013.37 (728.42)
C64	3.10 (1.91)	10.45 (12.63)	0.25 (0.41)	0.37 (0.44)	266.15 (198.24)
C256	3.46 (2.16)	7.52 (7.50)	0.29 (0.42)	0.34 (0.44)	1348.35 (843.64)
X64	5.46 (5.44)	5.57 (5.85)	0.27 (0.41)	0.29 (0.42)	260.99 (186.34)
X256	4.83 (3.68)	4.85 (3.70)	0.31 (0.44)	0.34 (0.44)	736.08 (429.79)

Table 6.7: Base, duplication of coding segments, and collective adaptation applied to GP, mean of 20 runs, std. dev. in parenthesis.

In Appendix F, I present the average of 20 runs for the GA with the various GP variants outlined above and for the FC family of graphs. I report the average of the max clique found for the 49 graphs in Table F.1 (Table F.3 for inside the collective memory) and the average of the clique cover in Table F.2 (Table F.4 for inside the collective memory). I am interested in determining the cumulative addition of extra capabilities extends the search. I analyze the max clique found (both in the chromosome and the collective memory), the clique cover (both in the chromosome and the collective memory), and the total time to run the experiment. The means and standard deviations are presented in Table 6.7.

Ignoring the variations produced by the different graph variations, we run a one-way ANOVA to test the hypothesis, using a significance level of $\alpha = 0.01$, that

H_0 All heuristics means are the same.

H_a At least 2 heuristics have different means.

We reject H_0 for each of the max clique found (both in the chromosome and the collective memory), the clique cover (both in the chromosome and the collective memory), and the total time to run the experiment. We perform the Scheffe follow-up test to determine significant differences in means at a level of $\alpha = 0.01$.

Over the FC family of graphs, we find:

- X64 is better at finding the max clique in the chromosome than all other variants.
- X256 is better at finding the max clique in the chromosome than all other variants but X64.
- B256 is better at finding the max clique in the chromosome than B64, R64, and C64.
- A Duncan follow-up test with $\alpha = 0.01$ reveals that R256 is better at finding the max clique in the chromosome than B64, R64, and C64. With the Duncan test, there is no difference between R256 and B256.
- C64 is better at finding the max clique in the collective memory than all other variants.
- C256 is better at finding the max clique in the collective memory than all other variants but C64.
- X64 is better at finding the max clique in the collective memory than B64, B256, R64, and R256.

- X256 is better at finding the max clique in the collective memory than B64, R64, and R256.
- A Duncan follow-up test with $\alpha = 0.01$ reveals that X64 is better at finding the max clique in the collective memory than X256.
- A Duncan follow-up test with $\alpha = 0.01$ reveals that X256 is better at finding the max clique in the collective memory than B256.
- X256 is better at finding the clique cover in the chromosome than B64.
- A Duncan follow-up test with $\alpha = 0.01$ reveals that X256 is also better at finding the clique cover in the chromosome than B256, R64, and C64.
- A Duncan follow-up test with $\alpha = 0.01$ reveals that C256 is also better at finding the clique cover in the chromosome than B64.
- C64 is better at finding the clique cover in the collective memory than B64, B256, R64, R256, and X64.
- C256 is better at finding the clique cover in the collective memory than B64, B256, and R64.
- X256 is better at finding the clique cover in the collective memory than B64 and B256.
- A Duncan follow-up test with $\alpha = 0.01$ reveals that C256 is also better at finding the clique cover in the collective memory than R256 and X64.

- A Duncan follow-up test with $\alpha = 0.01$ reveals that X256 is also better at finding the clique cover in the collective memory than R64, R256, and X64.
- A Duncan follow-up test with $\alpha = 0.01$ reveals that X64 is better at finding the clique cover in the collective memory than B64.
- C256 takes significantly longer to run than all others.
- B256 and R256 take significantly longer to run than all but C256.
- X256 takes significantly longer to run than B64, R64, C64, and X64.
- A population size of 256 takes significantly longer to run than those with size of 64.

From the follow up tests, we see that the addition of collective memory produces better max clique sizes than no collective memory at all. Increasing the number of chromosomes did not significantly improve the search for either max clique or clique cover. This result indicates that either these population sizes are too small for the GP based search or that GP based search is in general ineffective. My conjecture is that the GP subtree based crossover operation is not as effective as the GA double point crossover operation. Recent research has shown that other GP crossover operators can be more effective than subtree based crossover [Haynes *et al.*, 1995a; Angeline, 1996; Angeline, 1997; Haynes and Sen, 1997].

6.9 Conclusions

Even though the Schema Theorem, i.e., the Building Block Hypothesis, has not been applied to RS, HC, and SA, we see that each of these heuristics can utilize building blocks, which I have defined as coding segments, to form larger sub–solutions for a problem.

CHAPTER VII

Conclusions

One problem in building a GP Schema Theorem is in the definition of a building block. GP building blocks differ from GA building blocks in that a schema s can be expressed multiple times in a given chromosome, each instance of s may not be highly fit, and multiple genotypical schemata may map into the same phenotypical schema. I have defined a GP building block as a coding segment which contributes positively to the fitness of the chromosome, which implies a building block can only be detected if and only if its fitness can be determined apart from that of the chromosome. My main contribution in this dissertation is to apply this definition to share building blocks both within the chromosome and outside of all chromosomes inside a collective memory.

The clique domain allows for carefully controlled experiments to test the sharing of building blocks. It encompasses three NP–complete problems, partition into cliques, covering by cliques, and clique, and has the property that any other NP–complete problem may be mapped to any of these three problems. A key characteristic of the clique domain is that the problems can be decomposed into independent sub–problems, each of which can be individually solved and then independent solutions can then be integrated to arrive at the final overall solution. Each of these solutions to sub–problems forms a building block in the phenotypical

space.

I have found that the duplication of coding segments, i.e., building blocks, facilitates the discovery of larger/other building blocks. During the first crossover, one copy acts as a springboard for further exploration, while the duplicates act as a backup memory, allowing the chromosome to at least keep its previous solution. I empirically demonstrate that building blocks of consistently above average fitness and resilience to disruption can be assured.

I have also found that collective adaptation, i.e., the gathering of building blocks in a collective memory, greatly facilitates the discovery of building blocks. In problems which can be decomposed into sub–problems, a common strategy to enhance the search is divide and conquer, i.e., assign the sub–problems to different processes and then integrate the results. Collective adaptation enhances the GP and GA search by allowing such cooperation. As building blocks are detected in the chromosomes, they are gathered into the collective memory. Chromosomes may then access the partial solutions gathered by other chromosomes.

Also, non–GP processes, e.g., strong search heuristics, can extend that knowledge, creating building blocks which were not constructed in the chromosomes. These new building blocks may cause a problem if the building blocks are phenotypical and genotypical: there may be more than one genotype which maps to a given phenotype. While a general method exists for mapping back building blocks found in the chromosomes[1], such a mapping may not be possible for every

[1]Simply store the first subtree found which corresponds to the phenotype.

domain.

As cooperation is increased in the collective adaptation search system, we see an increase in performance in both the underlying search engine and the overall search process. The detection and sharing of building blocks is integral to collective adaptation and central to this increase in performance. While the Schema Theorem itself will not account for the sharing of building blocks outside the chromosomes, the research presented in this dissertation certainly shows that building blocks do exist in GP and can be hierarchically combined.

CHAPTER VIII

Future Work

I have identified several promising avenues of research in which I can extend the work I have presented in this dissertation:

Adaptive GP crossover: My hypothesis as to why the GA system is able to perform so well with limited resources, as compared to the GP, is that the standard GA crossover is more effective than the standard GP crossover. In particular, the sub–tree swapping of GP is geared towards exchanging the leaf nodes of the chromosomes (On the average, half of the nodes are in the leaf nodes and three–fourths are in the leaf nodes and the level above.). I plan to investigate an adaptive probability distribution for selecting nodes for crossover. I believe I will see a significant increase in performance with such a mechanism.

Niching: The problems found in the clique domain naturally illustrate the need for niching in GA populations. For example, consider the max clique problem when there is a graph composed of disjoint cliques, 3 of cardinality 8, 1 of cardinality 9. I would expect that the greedy algorithm presented in Figure 8.1 would find the max clique about 27% of the time, depending on the initial node selected. If I increase the number of cliques of cardinality 8, I expect the success to decrease: 7: 14%, 15: 7%, 31: 4%, and 63: 2%. We

could modify the greedy algorithm such that each node is the initial node selected into C, but this algorithm will only work if the cliques are disjoint.

```
select a node in G and put it in C
put in S all other nodes
while ( nodes in S ) {
    select and remove a node from S and add it to C
    if the size of the clique in C increases, keep that node
    else discard it
}
```

Figure 8.1: Greedy algorithm for max clique.

Just as the greedy algorithm selects one "hill", i.e., clique, to explore, so do the hill climbing algorithms. As we increase both the number of cliques present and the cardinality of those cliques, as in the 49 benchmark graphs in Chapter 6, GA, without collective adaptation, also has a hard time both determining the clique cover and the max clique. This difficulty is a result of there being too many cliques to explore and an increase of destructive crossover.

One mechanism for distributing the search amongst all of the cliques would be to introduce niching via a sharing function [Goldberg, 1989]. The heuristic is to reduce the fitness of chromosomes which are similar. The clique domain presents problems in determining both genotypical and phenotypical similarities.

Extending the benchmark graphs: The 49 benchmark graphs presented in Chapter 6 were carefully designed such that the cliques were disjoint and

both the number of cliques and the cardinality of each clique were known beforehand. Some additional research could be performed by carefully removing the disjointness property and controlling the number and distribution of edges shared between cliques. Also, it would be beneficial to categorize the DIMACS graphs with respect to my benchmark graphs.

BIBLIOGRAPHY

[Alberts *et al.*, 1989] Bruce Alberts, Dennis Bray, Julian Lewis, Martin Raff, Keith Roberts, and James D. Watson. *Molecular Biology of the Cell.* Garland Publishing, Inc., 1989.

[Altenberg, 1994] Lee Altenberg. The evolution of evolvability in genetic programming. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 3, pages 47–74. MIT Press, 1994.

[Andre and Teller, 1996] David Andre and Astro Teller. A study in program response and the negative effects of introns in genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 12–20, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Andre, 1995] David Andre. The evolution of agents that build mental models and create simple plans using genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 248–255, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[Angeline and Kinnear, Jr., 1996] Peter J. Angeline and Kenneth E. Kinnear, Jr., editors. *Advances in Genetic Programming 2.* MIT Press, Cambridge, MA, 1996.

[Angeline, 1994] Peter J. Angeline. Genetic programming and emergent intelligence. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, pages 75–97. MIT Press, Cambridge, MA, 1994.

[Angeline, 1996] Peter J. Angeline. Two self-adaptive crossover operators for genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 5, pages 89–110. MIT Press, Cambridge, MA, USA, 1996.

[Angeline, 1997] Peter J. Angeline. Subtree crossover: Building block engine or macromutation? In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 9–17, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Bui and Eppley, 1995] Thang Nguyen Bui and Paul H. Eppley. A hybrid genetic algorithm for the maximum clique problem. In Larry Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 478–484, San Francisco, CA, 1995. Morgan Kaufmann.

[Cobb, 1993] Helen Cobb. Is the genetic algorithm a cooperative learner? In L. Darrell Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 277–296. Morgan Kaufmann Publishers, Inc., 1993.

[Corkill *et al.*, 1986] Daniel D. Corkill, Kevin Q. Gallagher, and Kelly E. Murray. GBB: A generic blackboard development system. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 1008–1014, Philadelphia, PA, August 1986. (Also published in *Blackboard Systems*, Robert S. Engelmore and Anthony Morgan, editors, pages 503–518, Addison-Wesley, 1988.).

[Corkill, 1989] Daniel D. Corkill. Design alternatives for parallel and distributed blackboard systems. In V. Jagannathan, Rajendra Dodhiawala, and Lawrence S. Baum, editors, *Blackboard Architectures and Applications*, pages 99–136. Academic Press, 1989. (Presented at the Second Workshop on Blackboard Systems, AAAI-88, St. Paul, Minnesota, August 24, 1988.).

[Davidor, 1991] Yuval Davidor. Epistasis variance: A viewpoint on GA-hardness. In Gregory J. E. Rawlins, editor, *Foundations of Genetic Algorithms*. Morgan Kaufmann Publishers, 1991.

[Davis *et al.*, 1993] Lawrence Davis, David Orvosh, Anthony Cox, and Yuping Qiu. A genetic algorithm for survivable network design. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 408–415, San Mateo, CA, 1993. Morgan Kaufman.

[Davis, 1991] Lawrence Davis. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, NY, 1991.

[de Souza and Talukdar, 1991] Pedro Sergio de Souza and Sarosh N. Talukdar. Genetic algorithms in asynchronous teams. In Rick Belew and Lashon Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 392–397, San Mateo, CA, 1991. Morgan Kaufman.

[Decker *et al.*, 1993] Keith S. Decker, Alan J. Garvey, Marty A. Humphrey, and Victor R. Lesser. Control heuristics for scheduling in a parallel blackboard system. *International Journal of Pattern Recognition and Artificial Intelligence*, 7(2):243–264, 1993.

[Falkenauer, 1995] Emanuel Falkenauer. Solving equal piles with the grouping genetic algorithm. In Larry Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 492–497, San Francisco, CA, 1995. Morgan Kaufmann.

[Fennell and Lesser, 1977] Richard D. Fennell and Victor R. Lesser. Parallelism in Artificial Intelligence problem solving: A case study of Hearsay II. *IEEE Transactions on Computers*, C-26(2):98–111, February 1977. (Also published in *Readings in Distributed Artificial Intelligence*, Alan H. Bond and Les Gasser, editors, pages 106-119, Morgan Kaufmann, 1988.).

[Futuyma, 1986] Douglas J. Futuyma. *Evolutionary Biology.* Sinauer Associate, Sunderland, MA, 1986.

[Garey and Johnson, 1979] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., San Francisco, CA, 1979.

[Garland and Alterman, 1995] Andrew Garland and Richard Alterman. Preparation of multi-agent knowledge for reuse. In David W. Aha and Ashwin Ram, editors, *Working Notes for the AAAI Symposium on Adaptation of Knowldege for Reuse*, Cambridge, MA, November 1995. AAAI.

[Garland and Alterman, 1996] Andrew Garland and Richard Alterman. Multiagent learning through collective memory. In Sandip Sen, editor, *Working Notes for the AAAI Symposium on Adaptation, Co-evolution and Learning in Multiagent Systems*, pages 33–38, Stanford University, CA, March 1996.

[Goldberg, 1989] David E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning.* Addison-Wesley, Reading, MA, 1989.

[Goldberg, 1994] David E. Goldberg. Genetic and evolutionary algorithms come of age. *Communications of the ACM*, 37(3):113–119, March 1994.

[Guha and Lenat, 1990] R. V. Guha and Douglas B. Lenat. Cyc: A midterm report. *AI Magazine*, 11(3):33–59, Fall 1990.

[Halpern and Moses, 1990] Joseph Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, 1990. A preliminary version appeared in *Proc. 3rd ACM Symposium on Principles of Distributed Computing*, 1984.

[Hart and Belew, 1996] William E. Hart and Richard K. Belew. Optimization with genetic algorithm hybrids that use local search. In Richard K. Belew and Melanie Mitchell, editors, *Adaptive Individuals in Evolving Populations: Models and Algorithms*, pages 483–496. Addison-Wesley, 1996. (SFI Studies in the Sciences of Complexity Vol. 26).

[Hart, 1994] William E. Hart. *Adaptive Global Optimization with Local Search.* PhD thesis, University of California, San Diego, May 1994.

[Haynes and Sen, 1997] Thomas Haynes and Sandip Sen. Crossover operators for evolving a team. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference.* MIT Press, 1997.

[Haynes and Wainwright, 1995] Thomas D. Haynes and Roger L. Wainwright. A simulation of adaptive agents in a hostile environment. In K. M. George, Janice H. Carroll, Ed Deaton, Dave Oppenheim, and Jim Hightower, editors, *Pro-*

ceedings of the 1995 ACM Symposium on Applied Computing, pages 318–323. ACM Press, 1995.

[Haynes *et al.*, 1995a] Thomas Haynes, Sandip Sen, Dale Schoenefeld, and Roger Wainwright. Evolving a team. In E. V. Siegel and J. R. Koza, editors, *Working Notes for the AAAI Symposium on Genetic Programming*, Cambridge, MA, November 1995. AAAI.

[Haynes *et al.*, 1995b] Thomas Haynes, Roger Wainwright, Sandip Sen, and Dale Schoenefeld. Strongly typed genetic programming in evolving cooperation strategies. In Larry Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 271–278, San Francisco, CA, 1995. Morgan Kaufmann Publishers, Inc.

[Haynes *et al.*, 1996a] Thomas Haynes, Rose Gamble, Leslie Knight, and Roger Wainwright. Entailment for specification refinement. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 90–97, Cambridge, MA, 1996. The MIT Press.

[Haynes *et al.*, 1996b] Thomas Haynes, Dale Schoenefeld, and Roger Wainwright. Type inheritance in strongly typed genetic programming. In Kenneth E. Kinnear, Jr. and Peter J. Angeline, editors, *Advances in Genetic Programming 2*, chapter 18. MIT Press, 1996.

[Haynes, 1994] Thomas D. Haynes. A simulation of adaptive agents in a hostile environment. Master's thesis, University of Tulsa, Tulsa, OK., April 1994.

[Haynes, 1996] Thomas Haynes. Duplication of coding segments in genetic programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, OR, August 1996.

[Haynes, 1997a] Thomas Haynes. Augmenting collective adaptation with a simple process agent. In Sandip Sen, editor, *AAAI Workshop on Multiagent Learning*. 1997.

[Haynes, 1997b] Thomas Haynes. Collective memory search. In *Proceedings of the 1997 ACM Symposium on Applied Computing*. ACM Press, 1997.

[Haynes, 1997c] Thomas Haynes. On–line adaptation of search via knowledge reuse. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 156–161, 1997.

[Haynes, 1997d] Thomas Haynes. Phenotypical building blocks for genetic programming. In Thomas Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97)*, San Francisco, CA, 1997. Morgan Kaufmann.

[Hogg and Williams, 1993] Tad Hogg and Colin P. Williams. Solving the really hard problems with cooperative search. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 231–236, Menlo Park, CA, 1993. AAAI Press.

[Hogg and Williams, 1994] Tad Hogg and Colin P. Williams. Expected gains from parallelizing constraint solving for hard problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 331–336, Menlo Park, CA, 1994. AAAI Press.

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

[Holland, 1986] John H. Holland. Escaping brittleness: the possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning, an artificial intelligence approach: Volume II.* Morgan Kaufmann, Los Alamos, CA, 1986.

[Hwang and Briggs, 1985] Kai Hwang and Faye A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill International, 1985.

[Johnson and Trick, 1996] David S. Johnson and Michael A. Trick, editors. *Cliques, Coloring, and Satisfiability*, volume 26 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science.* American Mathematical Society, 1996.

[Jones and Forrest, 1995] Terry Jones and Stephanie Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In Larry Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 184–192, San Francisco, CA, 1995. Morgan Kaufmann.

[Kinnear, Jr., 1994a] Kenneth E. Kinnear, Jr., editor. *Advances in Genetic Programming.* MIT Press, Cambridge, MA, 1994.

[Kinnear, Jr., 1994b] Kenneth E. Kinnear, Jr. Alternatives in automatic function definition: A comparision of performance. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, pages 119–141. MIT Press, Cambridge, MA, 1994.

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

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

[Levenick, 1991] James R. Levenick. Inserting introns improves genetic algorithm success rate: Taking a cue from biology. In Rick Belew and Lashon Booker,

editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 123–127, San Mateo, CA, 1991. Morgan Kaufman.

[Mitchell *et al.*, 1992] Melanie Mitchell, Stephanie Forrest, and John H. Holland. The royal road for genetic algorithms: Fitness landscapes and GA performance. In *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*, pages 245–254, Cambridge, MA, 1992. MIT Press.

[Montana, 1995] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[Nii, 1986] H. Penny Nii. Blackboard systems: The blackboard model of problem solving and the evolution of blackboard architectures. *AI Magazine*, 7(2):38–53, Summer 1986.

[Nordin, 1994] Peter Nordin. A compiling genetic programming system that directly manipulates the machine code. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*. MIT Press, 1994.

[Nordin, 1996] Peter Nordin. Explictly defined introns and destructive crossover in genetic programming. In P. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*. MIT Press, 1996.

[O'Reilly and Oppacher, 1994] Una-May O'Reilly and Franz Oppacher. Program search with a hierarchical variable length representation: Genetic programming, simulated annealing and hill climbing. In Yuval Davidor, Hans-Paul Schwefel, and Reinhard Manner, editors, *Parallel Problem Solving from Nature – PPSN III*, number 866 in Lecture Notes in Computer Science, pages 397–406, Jerusalem, 9-14 October 1994. Springer-Verlag.

[O'Reilly and Oppacher, 1995a] Una-May O'Reilly and Franz Oppacher. Hybridized crossover-based search techniques for program discovery. In *Proceedings of the 1995 World Conference on Evolutionary Computation*, volume 2, page 573, Perth, Australia, 29 November - 1 December 1995.

[O'Reilly and Oppacher, 1995b] Una-May O'Reilly and Franz Oppacher. The troubling aspects of a building block hypothesis for genetic programming. In L. Darrell Whitley and Michael D. Vose, editors, *Foundations of Genetic Algorithms 3*, pages 73–88, Estes Park, Colorado, USA, 31 July–2 August 1994 1995. Morgan Kaufmann.

[O'Reilly, 1995] Una-May O'Reilly. *An Analysis of Genetic Programming*. PhD thesis, Carelton University, Ottawa, Ontario, Canada, 22 September 1995.

[Orvosh and Davis, 1993] David Orvosh and Lawrence Davis. Shall we repair? Genetic algorithms, combinatorial optimization, and feasibility constraints. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, page 650, San Mateo, CA, 1993. Morgan Kaufman.

[Poli and Langdon, 1997a] Riccardo Poli and W. B. Langdon. An experimental analysis of schema creation, propagation and disruption in genetic programming. In Thomas Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97)*, San Francisco, CA, 1997. Morgan Kaufmann.

[Poli and Langdon, 1997b] Riccardo Poli and W. B. Langdon. A new schema theory for genetic programming with one-point crossover and point mutation. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 278–285, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Punch *et al.*, 1996] William F. Punch, Douglas Zongker, and Erik D. Goodman. The royal tree problem, a benchmark for single and multiple population genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 15, pages 299–316. MIT Press, Cambridge, MA, USA, 1996.

[Rosca and Ballard, 1996] Justinian P. Rosca and Dana H. Ballard. Discovery of subroutines in genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 9, pages 177–202. MIT Press, Cambridge, MA, USA, 1996.

[Rosca, 1997] Justinian P. Rosca. Analysis of complexity drift in genetic programming. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 286–294, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Russell and Norvig, 1995] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[Seront, 1995] Gregory Seront. External concepts reuse in genetic programming. In E. V. Siegel and J. R. Koza, editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 94–98, MIT, Cambridge, MA, USA, 10–12 November 1995. AAAI.

[Soule and Foster, 1997] Terence Soule and James A. Foster. Genetic algorithm hardness measures applied to the maximum clique problem. In Thomas Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97)*, San Francisco, CA, 1997. Morgan Kaufmann.

[Soule *et al.*, 1996] Terence Soule, James A. Foster, and John Dickinson. Using genetic programming to approximate maximum clique. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Conference*, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Spector, 1996] Lee Spector. Simultaneous evolution of programs and their control structures. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 7, pages 137–154. MIT Press, Cambridge, MA, USA, 1996.

[Tackett and Carmi, 1994] Walter Alden Tackett and Aviram Carmi. The donut problem: Scalability and generalization in genetic programming. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 7, pages 143–176. MIT Press, 1994.

[Tackett, 1993] Walter Alden Tackett. Genetic programming for feature discovery and image discrimination. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 303–309, University of Illinois at Urbana-Champaign, 17-21 July 1993. Morgan Kaufmann.

[Tackett, 1995] Walter A. Tackett. Mining the genetic program. *IEEE Expert*, 12(3):28–38, 1995.

[Talukdar *et al.*, 1983] S .N. Talukdar, S. S. Pyo, and T. Giras. Asynchronous procedures for parallel processing. *IEEE Transactions on PAS*, PAS–102(11), November 1983.

[Tanenbaum, 1987] A. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice Hall, Engelwood Cliffs, NJ, 1987.

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

APPENDIX A

Data for Base Heuristics

		1	2	4	8	16	32	64
1	RS	1	2	4	8	13	20	28
	HC	1	2	4	8	15	22	41
	SA	1	2	4	8	15	22	42
	GA	1	2	4	7.3	12	17	26
	GP	1	2	4	5.9	8.1	9.1	9.8
2	RS	1	2	4	6.4	7.9	9.8	11
	HC	1	2	4	8	14	19	29
	SA	1	2	4	8	13	20	28
	GA	1	2	3.9	5.7	6.8	8.3	8.7
	GP	1	2	3.5	4.8	5.6	5.5	5.6
4	RS	1	2	3.8	4.7	5.3	6	6
	HC	1	2	4	7.3	11	13	18
	SA	1	2	4	7.3	11	14	18
	GA	1	2	3.4	4	4.2	4.7	4.7
	GP	1	2	3.3	3.9	4.2	4	4
8	RS	1	2	3.1	3.8	4.2	4	4.4
	HC	1	2	4	6.4	7.7	9.3	10
	SA	1	2	4	6.1	8	8.8	9.7
	GA	1	2	3	3.1	3	3.5	3.6
	GP	1	2	2.8	3.5	3.8	3.1	3.2
16	RS	1	2	2.9	3.2	3	3.1	3.3
	HC	1	2	3.7	5	5.5	6	6.7
	SA	1	2	3.6	4.8	3.3	5.8	6.1
	GA	1	2	2.2	2.4	2.5	2.8	3
	GP	1	2	2.4	3.2	2.9	2.6	2.9
32	RS	1	2	2.3	2.6	2.8	2.6	2.7
	HC	1	2	3.3	3.7	4	4.8	4.4
	SA	1	2	3.3	3.9	3.8	4.5	4.3
	GA	1	1.8	2	2	2.1	2.4	2.3
	GP	1	2	2.1	2.4	2.5	2.5	2.5
64	RS	1	2	2	2.2	2.3	2.1	2.3
	HC	1	2	3	3	3.3	3.4	3.4
	SA	1	2	2.5	2.8	2.9	3.2	3.4
	GA	1	1.3	1.9	2	2	1.9	1.9
	GP	1	2	1.8	2.2	2	2.3	2.1

Table A.1: For comparing the different base versions of the heuristics the average maximal Generational Max Clique of Generation.

		1	2	4	8	16	32	64
1	RS	1	1	1	0.96	0.0014	1.2E-13	3.5E-54
	HC	1	1	1	1	0.48	5.8E-08	1.7E-21
	SA	1	1	1	1	0.4	2.3E-10	1.7E-21
	GA	1	1	1	0.53	7.1E-05	1E-17	5.4E-48
	GP	1	1	1	0.033	3E-08	2.1E-28	4.4E-82
2	RS	1	1	0.67	0.033	6.1E-09	1E-28	2E-80
	HC	1	1	1	0.97	0.035	3.4E-16	2.1E-51
	SA	1	1	1	0.9	0.0053	2.9E-08	7.3E-50
	GA	1	1	0.9	0.017	6.6E-10	8.1E-30	2E-83
	GP	1	0.93	0.42	0.002	4.6E-10	5.3E-33	6E-87
4	RS	1	0.71	0.24	0.00073	4.9E-12	4.7E-33	3.3E-87
	HC	1	1	1	0.16	6.1E-05	1.8E-23	4E-65
	SA	1	1	0.99	0.19	0.00084	5.3E-17	8.7E-61
	GA	1	0.99	0.35	0.00043	1.5E-12	2.9E-34	8.2E-88
	GP	1	0.4	0.15	0.00025	8.2E-13	6.9E-35	1.4E-88
8	RS	0.86	0.29	0.058	9.3E-05	4.8E-13	2E-35	3.5E-88
	HC	1	1	0.71	0.016	1.5E-07	2.1E-27	3.2E-76
	SA	1	0.99	0.62	0.0086	1.5E-07	1.4E-29	6.5E-80
	GA	1	0.68	0.11	8.5E-05	1.7E-13	3.6E-35	5E-89
	GP	1	0.17	0.043	7.9E-05	2.7E-13	5.3E-36	1.5E-89
16	RS	0.53	0.12	0.019	1.7E-05	2.4E-14	2.4E-36	5.8E-90
	HC	1	0.91	0.26	0.0011	6.6E-13	6E-34	1.2E-83
	SA	1	0.66	0.2	0.00046	4.8E-14	9.5E-34	3.7E-87
	GA	0.99	0.26	0.036	2.5E-05	5.2E-14	5.9E-36	1.1E-89
	GP	1	0.069	0.013	2.7E-05	4.1E-14	1.2E-36	3.9E-90
32	RS	0.27	0.045	0.0059	4.7E-06	8.4E-15	6.9E-37	1.4E-90
	HC	1	0.42	0.075	7.2E-05	1.2E-13	6.8E-35	4.4E-89
	SA	0.83	0.22	0.055	8.8E-05	1.6E-13	2.8E-35	1E-88
	GA	0.74	0.078	0.0098	7.5E-06	1.5E-14	1.9E-36	2.9E-90
	GP	1	0.034	0.0039	4.7E-06	9.1E-15	4.9E-37	1.2E-90
64	RS	0.14	0.018	0.0019	1.2E-06	2.8E-15	2.2E-37	4.6E-91
	HC	1	0.14	0.022	1.5E-05	3.2E-14	3.5E-36	7.8E-90
	SA	0.59	0.069	0.01	8.4E-06	1.9E-14	3.7E-36	5.5E-90
	GA	0.46	0.019	0.0025	2E-06	4E-15	4.1E-37	7.4E-91
	GP	1	0.016	0.0012	1.4E-06	1.7E-15	1.8E-37	3.3E-91

Table A.2: For comparing the different base versions of the heuristics the average maximal Max Clique Cover of Generation.

		1	2	4	8	16	32	64
1	RS	21.75	21.30	21.20	21.65	22.95	25.35	29.95
	HC	21.75	21.45	21.45	22.05	35.55	26.75	47.45
	SA	21.65	21.30	21.15	21.85	34.60	85.30	80.05
	GA	55.70	51.70	104.30	83.00	62.50	35.50	59.25
	GP	344.45	241.80	236.95	246.30	362.15	332.30	188.20
2	RS	21.05	20.70	20.90	21.00	21.20	21.30	21.75
	HC	21.55	34.70	69.20	23.60	35.55	81.65	33.40
	SA	21.45	21.25	82.90	41.90	36.75	78.85	34.30
	GA	53.35	36.85	26.50	28.30	47.70	29.40	31.20
	GP	270.40	238.55	171.20	190.50	218.30	212.20	199.95
4	RS	21.00	20.55	20.70	20.65	20.50	21.00	20.95
	HC	65.25	43.80	23.00	31.40	40.10	42.35	44.80
	SA	85.30	87.45	68.60	30.90	38.55	41.65	42.65
	GA	26.05	27.10	27.10	28.25	61.50	29.15	30.45
	GP	252.65	131.00	138.65	155.65	213.15	227.80	177.60
8	RS	21.00	20.50	20.60	20.45	20.50	20.80	20.90
	HC	22.25	22.85	27.25	34.75	38.75	80.80	88.65
	SA	21.70	22.05	25.90	31.85	36.35	75.40	82.85
	GA	27.10	26.65	28.15	38.30	61.60	30.65	32.20
	GP	244.30	128.95	189.35	234.90	189.60	180.75	128.10
16	RS	20.80	20.65	20.50	20.45	20.60	20.65	23.00
	HC	25.15	24.65	27.85	31.30	79.80	34.20	36.20
	SA	23.35	22.45	25.70	27.40	28.95	30.70	33.75
	GA	30.65	26.50	30.40	28.75	58.95	31.35	34.25
	GP	308.75	116.50	152.80	114.60	171.75	148.85	191.55
32	RS	20.95	20.20	38.05	20.45	20.50	20.65	20.65
	HC	35.30	23.30	25.15	26.80	28.80	57.35	58.40
	SA	26.30	21.30	30.60	25.35	24.40	50.40	53.30
	GA	35.05	26.20	27.25	28.45	63.15	32.65	39.15
	GP	526.45	128.15	151.40	141.15	100.25	179.95	140.35
64	RS	20.95	23.80	20.35	20.50	20.50	20.85	20.55
	HC	67.40	22.15	23.15	23.45	23.75	24.35	24.95
	SA	30.80	21.20	21.45	21.90	22.25	22.65	23.55
	GA	39.55	26.10	27.50	28.95	31.85	37.80	48.50
	GP	780.20	125.00	151.95	114.65	117.50	121.40	129.25

Table A.3: For comparing the different base versions of the heuristics Sum of Time Differences per Generation.

APPENDIX B

Data for Duplication of Coding Segments Heuristics

		1	2	4	8	16	32	64
1	RRS	1	2	4	8	14	20	27
	RSA	1	2	4	8	13	21	31
	RHC	1	2	4	8	14	22	33
	RGA	1	2	4	8	16	28	35
	RGP	1	2	4	5.6	7.9	8.4	9.1
2	RRS	1	2	4	7.8	12	16	20
	RSA	1	2	4	7.7	13	20	30
	RHC	1	2	4	7.5	13	20	32
	RGA	1	2	4	8	15	22	26
	RGP	1	2	3.9	5	5.3	4.8	5.3
4	RRS	1	2	4	7.6	10	12	14
	RSA	1	2	4	7.3	12	20	28
	RHC	1	2	4	6.8	12	18	27
	RGA	1	2	4	7.5	11	14	16
	RGP	1	2	3.4	3.9	4.2	4.1	4.2
8	RRS	1	2	3.9	6.8	9.6	10	11
	RSA	1	2	4	6.8	12	17	25
	RHC	1	2	4	6.5	12	17	23
	RGA	1	2	3.5	6	6.7	8.3	8.3
	RGP	1	2	3	3.3	3.4	3.3	3.3
16	RRS	1	2	3.7	6.3	8.3	9.2	10
	RSA	1	2	3.6	6.7	11	16	23
	RHC	1	2	3.5	6.3	11	16	20
	RGA	1	2	2.6	3.1	4.8	4.7	5.3
	RGP	1	2	2.5	2.9	2.9	2.8	2.8
32	RRS	1	2	3.3	6.2	8	8.9	9.3
	RSA	1	2	3.4	5.8	9.9	14	17
	RHC	1	2	3.3	5.7	9.8	14	18
	RGA	1	1.8	2.4	2.8	2.8	3	2.5
	RGP	1	2	2.3	2.4	2.2	2.4	2.4
64	RRS	1	2	3.4	6	7	7.8	7.8
	RSA	1	2	3	5.6	9.3	12	14
	RHC	1	2	3.1	5.6	8.7	12	15
	RGA	1	1.3	2	2.3	2.6	2	2.1
	RGP	1	2	2	2.1	2.1	2.1	2.2

Table B.1: For comparing repair on the heuristics the average maximal Generational Max Clique of Generation.

		1	2	4	8	16	32	64
1	RRS	1	1	1	1	0.0035	4.9E-16	2.2E-58
	RSA	1	1	1	1	0.056	1.2E-14	8E-45
	RHC	1	1	1	1	0.11	6.7E-12	3.3E-43
	RGA	1	1	1	1	1	0.00021	1.1E-39
	RGP	1	1	1	0.041	5.8E-09	9.4E-30	1.5E-80
2	RRS	1	1	1	0.5	6E-05	1.2E-20	1.3E-67
	RSA	1	1	1	0.8	0.0042	1E-14	8.1E-50
	RHC	1	1	1	0.83	0.0044	6.4E-14	1.4E-47
	RGA	1	1	1	0.95	0.23	4.2E-11	5.2E-56
	RGP	1	0.97	0.9	0.013	3.4E-11	8.9E-34	1.8E-86
4	RRS	1	1	0.79	0.19	5E-06	1.3E-25	1.4E-76
	RSA	1	1	1	0.41	0.00026	5.9E-14	3.7E-50
	RHC	1	1	1	0.22	0.0017	1.3E-15	1.4E-48
	RGA	1	0.99	0.87	0.19	2.5E-05	1.2E-19	1.2E-70
	RGP	1	0.7	0.46	0.0021	6.3E-12	2.1E-34	5.9E-88
8	RRS	1	0.99	0.32	0.04	4.7E-08	2.2E-27	5.3E-81
	RSA	1	1	0.85	0.11	0.0014	1.7E-18	5.3E-56
	RHC	1	1	0.89	0.057	3.1E-05	1.4E-18	1.3E-54
	RGA	1	0.7	0.31	0.014	4.3E-10	2.7E-29	6.4E-80
	RGP	1	0.23	0.13	0.00039	1.6E-12	5.3E-35	1.2E-88
16	RRS	1	0.58	0.13	0.011	1.3E-08	1E-27	7.7E-83
	RSA	1	0.9	0.43	0.034	8.9E-05	2.1E-21	1.8E-50
	RHC	1	0.92	0.39	0.027	2.1E-05	1.4E-18	1.8E-68
	RGA	1	0.23	0.059	0.0005	7.4E-11	5E-33	9E-86
	RGP	1	0.091	0.043	0.00011	3.4E-13	1.1E-35	2.3E-89
32	RRS	1	0.25	0.05	0.0059	6.4E-09	6.2E-31	2.2E-81
	RSA	1	0.42	0.14	0.0069	5.5E-07	5.2E-23	1.9E-67
	RHC	1	0.47	0.17	0.0072	3.8E-07	4.7E-22	4.9E-72
	RGA	0.95	0.073	0.015	0.00021	9.5E-14	3.1E-36	2.1E-89
	RGP	1	0.039	0.015	1.8E-05	3.5E-14	3.1E-36	6.7E-90
64	RRS	0.93	0.096	0.02	0.0017	2.7E-11	6.2E-33	5E-85
	RSA	1	0.12	0.047	0.0021	2.5E-08	3.2E-27	1.5E-74
	RHC	1	0.15	0.051	0.0028	3.3E-06	2.1E-26	3.5E-74
	RGA	0.73	0.018	0.0034	5.1E-06	1E-14	4.9E-37	1.6E-90
	RGP	1	0.017	0.0046	4.4E-06	8.3E-15	8.3E-37	1.5E-90

Table B.2: For comparing repair on the heuristics the average maximal Max Clique Cover of Generation.

		1	2	4	8	16	32	64
1	RRS	22.55	22.15	22.20	29.05	41.45	93.65	103.50
	RS	21.75	21.30	21.20	21.65	22.95	25.35	29.95
	RHC	14.80	21.65	30.50	37.60	33.70	56.70	28.90
	HC	21.75	21.45	21.45	22.05	35.55	26.75	47.45
	RSA	22.70	22.10	22.15	24.95	46.50	78.05	93.90
	SA	21.65	21.30	21.15	21.85	34.60	85.30	80.05
	RGA	54.95	53.65	54.35	57.30	71.05	113.30	147.45
	GA	55.70	51.70	104.30	83.00	62.50	35.50	59.25
	RGP	81.55	87.10	156.15	483.30	451.30	476.90	498.10
	GP	344.45	241.80	236.95	246.30	362.15	332.30	188.20
2	RRS	23.50	22.75	23.35	31.00	38.30	42.75	93.00
	RS	21.05	20.70	20.90	21.00	21.20	21.30	21.75
	RHC	16.00	19.50	16.15	31.60	94.90	69.70	55.55
	HC	21.55	34.70	69.20	23.60	35.55	81.65	33.40
	RSA	23.65	22.90	23.10	37.65	47.00	89.40	87.10
	SA	21.45	21.25	82.90	41.90	36.75	78.85	34.30
	RGA	57.00	55.60	56.20	59.80	87.65	105.15	119.45
	GA	53.35	36.85	26.50	28.30	47.70	29.40	31.20
	RGP	77.55	93.20	213.80	436.85	491.45	574.20	552.75
	GP	270.40	238.55	171.20	190.50	218.30	212.20	199.95
4	RRS	24.90	24.00	30.35	34.10	38.10	40.75	44.75
	RS	21.00	20.55	20.70	20.65	20.50	21.00	20.95
	RHC	14.70	14.40	16.05	33.30	31.15	31.05	34.35
	HC	65.25	43.80	23.00	31.40	40.10	42.35	44.80
	RSA	25.00	24.05	26.20	44.65	48.55	43.95	41.80
	SA	85.30	87.45	68.60	30.90	38.55	41.65	42.65
	RGA	61.00	60.20	59.55	64.05	77.55	84.45	96.80
	GA	26.05	27.10	27.10	28.25	61.50	29.15	30.45
	RGP	86.15	90.95	257.05	439.85	423.80	479.40	471.90
	GP	252.65	131.00	138.65	155.65	213.15	227.80	177.60
8	RRS	35.00	30.75	38.65	41.05	43.40	37.80	39.70
	RS	21.00	20.50	20.60	20.45	20.50	20.80	20.90
	RHC	17.65	16.45	29.25	37.50	37.60	54.85	54.15
	HC	22.25	22.85	27.25	34.75	38.75	80.80	88.65
	RSA	29.10	26.10	41.20	47.65	48.15	91.55	87.50
	SA	21.70	22.05	25.90	31.85	36.35	75.40	82.85
	RGA	68.15	61.60	64.95	62.65	73.90	74.85	74.60
	GA	27.10	26.65	28.15	38.30	61.60	30.65	32.20
	RGP	63.90	85.70	175.30	342.90	440.05	390.50	229.55
	GP	244.30	128.95	189.35	234.90	189.60	180.75	128.10
16	RRS	35.00	25.15	28.65	50.65	32.15	33.75	35.50
	RS	20.80	20.65	20.50	20.45	20.60	20.65	23.00
	RHC	26.65	19.30	33.75	41.50	37.85	30.50	32.55
	HC	25.15	24.65	27.85	31.30	79.80	34.20	36.20
	RSA	35.00	27.95	51.75	55.80	47.15	44.30	43.15
	SA	23.35	22.45	25.70	27.40	28.95	30.70	33.75
	RGA	84.30	58.75	65.55	70.00	69.30	76.45	76.25
	GA	30.65	26.50	30.40	28.75	58.95	31.35	34.25
	RGP	232.20	85.05	129.95	225.65	269.40	281.20	300.15
	GP	308.75	116.50	152.80	114.60	171.75	148.85	191.55
32	RRS	49.25	24.30	28.00	31.25	31.20	30.20	32.40
	RS	20.95	20.20	38.05	20.45	20.50	20.65	20.65
	RHC	32.75	19.95	35.15	36.25	35.45	27.80	30.55
	HC	35.30	23.30	25.15	26.80	28.80	57.35	58.40
	RSA	50.55	26.10	41.20	46.90	46.55	89.75	90.70
	SA	26.30	21.30	30.60	25.35	24.40	50.40	53.30
	RGA	115.35	56.45	61.30	63.90	68.15	74.40	87.55
	GA	35.05	26.20	27.25	28.45	63.15	32.65	39.15
	RGP	495.80	82.50	140.45	138.95	173.15	191.60	197.75

Continued on next page

		1	2	4	8	16	32	64
	GP	526.45	128.15	151.40	141.15	100.25	179.95	140.35
64	RRS	69.55	23.35	26.00	26.45	27.55	27.90	29.55
	RS	20.95	23.80	20.35	20.50	20.50	20.85	20.55
	RHC	58.10	15.40	22.25	25.15	28.25	31.15	32.60
	HC	67.40	22.15	23.15	23.45	23.75	24.35	24.95
	RSA	87.75	23.65	32.75	37.30	42.15	43.65	45.25
	SA	30.80	21.20	21.45	21.90	22.25	22.65	23.55
	RGA	155.30	54.45	58.60	62.80	68.25	80.65	101.75
	GA	39.55	26.10	27.50	28.95	31.85	37.80	48.50
	RGP	913.30	90.95	131.10	113.00	120.20	147.05	110.35
	GP	780.20	125.00	151.95	114.65	117.50	121.40	129.25

Table B.3: For comparing repair on the heuristics Sum of Time Differences per Generation.

APPENDIX C

Data for Varying the Repair Rate for GA

		1	2	4	8	16	32	64
1	R0	1	2	4	7.3	12	17	26
	R05	1	2	4	8	15	21	28
	R1	1	2	4	8	15	22	31
	R5	1	2	4	8	16	26	36
	R10	1	2	4	8	16	28	35
	R20	1	2	4	8	16	28	37
	R25	1	2	4	8	16	28	38
	R50	1	2	4	8	16	28	38
2	R0	1	2	3.9	5.7	6.8	8.3	8.7
	R05	1	2	4	7.9	12	14	17
	R1	1	2	4	8	14	17	19
	R5	1	2	4	8	15	21	24
	R10	1	2	4	8	15	22	26
	R20	1	2	4	8	16	23	28
	R25	1	2	4	8	16	23	29
	R50	1	2	4	8	16	24	28
4	R0	1	2	3.4	4	4.2	4.7	4.7
	R05	1	2	4	7.5	9.7	11	11
	R1	1	2	4	7.6	9.9	12	12
	R5	1	2	4	7.5	11	14	15
	R10	1	2	4	7.5	11	14	16
	R20	1	2	4	7.5	12	15	17
	R25	1	2	4	7.7	12	15	17
	R50	1	2	4	7.7	13	15	19
8	R0	1	2	3	3.1	3	3.5	3.6
	R05	1	2	4	5.7	6.3	7.8	8.3
	R1	1	2	3.7	5.9	7.2	7.7	8.5
	R5	1	2	3.9	6	6.9	7.9	8.9
	R10	1	2	3.5	6	6.7	8.3	8.3
	R20	1	2	3.5	5.5	7.5	8.3	8
	R25	1	2	3.5	5.3	7	8.2	8.7
	R50	1	2	3.5	5.7	6.7	8	8.3
16	R0	1	2	2.2	2.4	2.5	2.8	3
	R05	1	2	3.3	4.5	5.8	5.5	5.9
	R1	1	2	3.4	4.4	5.8	6	6.2
	R5	1	2	3.3	3.9	5.5	6	5.2
	R10	1	2	2.6	3.1	4.8	4.7	5.3
	R20	1	2	2.4	3	4.7	4	4.5
	R25	1	2	2.4	2.8	4.3	4	4.3
	R50	1	2	2.4	2.6	3.1	3.5	4.2
32	R0	1	1.8	2	2	2.1	2.4	2.3
	R05	1	1.8	2.6	3.3	3.4	4.1	4
	R1	1	1.8	2.5	3.5	3.8	3.6	4.5
	R5	1	1.8	3	2.8	3.5	3.6	3.3
	R10	1	1.8	2.4	2.8	2.8	3	2.5
	R20	1	1.8	2.3	2.4	2.6	2.5	2.6
	R25	1	1.8	2.1	2.3	2.2	2.5	2.6
	R50	1	1.8	2	2.1	2	2.5	2.4
64	R0	1	1.3	1.9	2	2	1.9	1.9
	R05	1	1.3	2.5	2.8	3.1	2.8	2.6
	R1	1	1.3	2.6	3.1	3	2.6	2.9

Continued on next page

		1	2	4	8	16	32	64
	R5	1	1.3	2.2	2.5	2.6	2.4	1.9
	R10	1	1.3	2	2.3	2.6	2	2.1
	R20	1	1.3	2.1	2.1	2.2	1.9	1.9
	R25	1	1.3	1.9	2.1	2.3	2.1	1.9
	R50	1	1.3	2	2.1	2.1	1.9	1.9

Table C.1: For comparing the effect on repair rate on the GA the average maximal Generational Max Clique of Generation.

		1	2	4	8	16	32	64
1	R0	1	1	1	0.53	7.1E-05	1E-17	5.4E-48
	R05	1	1	1	1	0.28	2.1E-09	6.9E-54
	R1	1	1	1	1	0.63	8.3E-11	1.6E-49
	R5	1	1	1	1	0.99	1.3E-07	5.5E-40
	R10	1	1	1	1	1	0.00021	1.1E-39
	R20	1	1	1	1	1	0.0032	2.4E-38
	R25	1	1	1	1	1	0.00033	5.8E-40
	R50	1	1	1	1	1	0.0033	1.4E-41
2	R0	1	1	0.9	0.017	6.6E-10	8.1E-30	2E-83
	R05	1	1	1	0.77	0.0037	4.6E-20	2.2E-70
	R1	1	1	1	0.89	0.035	5.3E-18	2.1E-63
	R5	1	1	1	0.88	0.14	6.5E-14	3.7E-60
	R10	1	1	1	0.95	0.23	4.2E-11	5.2E-56
	R20	1	1	1	1	0.34	2.9E-08	3.5E-54
	R25	1	1	1	1	0.33	7.8E-12	5.8E-53
	R50	1	1	1	1	0.47	5.9E-08	1.8E-54
4	R0	1	0.99	0.35	0.00043	1.5E-12	2.9E-34	8.2E-88
	R05	1	1	0.78	0.15	4.3E-06	7E-26	2.5E-78
	R1	1	1	0.82	0.18	5.1E-07	2.1E-25	1.9E-75
	R5	1	0.99	0.81	0.21	0.00017	6.4E-21	1.3E-74
	R10	1	0.99	0.87	0.19	2.5E-05	1.2E-19	1.2E-70
	R20	1	0.99	0.91	0.27	0.0017	6.4E-21	6.1E-68
	R25	1	0.99	0.87	0.34	0.00087	6.1E-21	2.6E-69
	R50	1	0.99	0.89	0.31	0.00097	1.2E-19	1.6E-66
8	R0	1	0.68	0.11	8.5E-05	1.7E-13	3.6E-35	5E-89
	R05	1	0.79	0.36	0.0042	1.2E-09	5.6E-31	4.7E-84
	R1	1	0.78	0.37	0.0071	1.5E-09	1.4E-30	3.6E-82
	R5	1	0.74	0.34	0.022	1.4E-07	2.3E-29	4.9E-81
	R10	1	0.7	0.31	0.014	4.3E-10	2.7E-29	6.4E-80
	R20	1	0.61	0.28	0.019	2.6E-09	1.3E-29	4.3E-81
	R25	1	0.64	0.27	0.024	4.9E-10	1.4E-29	9.5E-82
	R50	1	0.67	0.3	0.024	1.4E-09	1.3E-29	4.6E-81
16	R0	0.99	0.26	0.036	2.5E-05	5.2E-14	5.9E-36	1.1E-89
	R05	1	0.33	0.096	0.00038	1.9E-10	3.6E-34	2.8E-87
	R1	1	0.33	0.11	0.00024	6.2E-12	5.6E-33	1E-85
	R5	1	0.25	0.085	0.00016	1.5E-11	5.8E-30	2.1E-87
	R10	1	0.23	0.059	0.0005	7.4E-11	5E-33	9E-86
	R20	0.99	0.21	0.043	5.6E-05	6.2E-11	9.9E-35	3.9E-88
	R25	1	0.21	0.042	0.00016	8.6E-12	1.2E-34	9.1E-87
	R50	1	0.21	0.043	5.8E-05	5.4E-11	8.1E-35	2.1E-88
32	R0	0.74	0.078	0.0098	7.5E-06	1.5E-14	1.9E-36	2.9E-90
	R05	0.95	0.1	0.029	3.4E-05	8.1E-14	2.8E-34	1.6E-88
	R1	0.95	0.089	0.023	7.3E-05	3.8E-12	2E-35	6.7E-89
	R5	0.95	0.091	0.023	2.2E-05	5E-13	1.4E-35	2.5E-89
	R10	0.95	0.073	0.015	0.00021	9.5E-14	3.1E-36	2.1E-89
	R20	0.96	0.069	0.011	1.1E-05	8.4E-14	2.4E-36	4.7E-90
	R25	0.97	0.067	0.0094	9.5E-06	1.9E-14	2.9E-36	4.9E-90
	R50	0.97	0.067	0.0086	8.3E-06	1.6E-14	3.5E-36	4.7E-90
64	R0	0.46	0.019	0.0025	2E-06	4E-15	4.1E-37	7.4E-91
	R05	0.67	0.02	0.0071	8.1E-06	4.6E-14	5.6E-36	4.7E-90
	R1	0.67	0.024	0.0082	1.3E-05	3E-14	2E-36	4.4E-90
	R5	0.7	0.016	0.004	4.3E-06	1.2E-14	1.1E-36	1.1E-90
	R10	0.73	0.018	0.0034	5.1E-06	1E-14	4.9E-37	1.6E-90
	R20	0.78	0.016	0.0027	2.2E-06	5.2E-15	3.7E-37	8.1E-91
	R25	0.79	0.017	0.0023	2E-06	8.7E-15	7.1E-37	7.5E-91
	R50	0.83	0.016	0.0023	2.3E-06	3.7E-15	2.9E-37	8E-91

Table C.2: For comparing the effect on repair rate on the GA the average maximal Max Clique Cover of Generation.

		1	2	4	8	16	32	64
1	R0	55.70	51.70	104.30	83.00	62.50	35.50	59.25
	R05	52.95	52.05	52.70	55.30	69.70	83.55	93.40
	R1	52.95	52.15	52.50	54.85	66.95	82.10	96.10
	R5	54.20	53.35	54.35	57.10	70.35	101.10	130.10
	R10	54.95	53.65	54.35	57.30	71.05	113.30	147.45
	R20	55.05	54.35	54.65	57.75	72.35	118.75	156.25
	R25	55.50	54.20	54.80	57.65	72.90	123.35	160.35
	R50	56.35	55.05	55.50	58.45	73.95	128.50	172.00
2	R0	53.35	36.85	26.50	28.30	47.70	29.40	31.20
	R05	53.95	53.05	53.90	61.10	68.55	73.35	79.70
	R1	54.40	53.15	53.75	57.85	66.40	71.40	77.30
	R5	56.30	55.45	55.75	59.80	78.25	93.00	104.75
	R10	57.00	55.60	56.20	59.80	87.65	105.15	119.45
	R20	57.30	56.00	56.70	60.45	93.35	116.80	133.75
	R25	57.30	55.90	56.60	60.55	95.30	120.85	138.75
	R50	58.00	57.00	57.35	61.50	101.40	125.30	145.45
4	R0	26.05	27.10	27.10	28.25	61.50	29.15	30.45
	R05	56.30	55.30	57.90	62.60	89.25	67.90	71.60
	R1	57.20	55.55	56.60	60.35	66.10	68.55	69.85
	R5	60.55	58.35	60.00	62.75	71.40	76.95	85.15
	R10	61.00	60.20	59.55	64.05	77.55	84.45	96.80
	R20	61.65	59.80	60.00	65.00	82.20	91.55	104.35
	R25	60.90	59.55	60.15	66.45	81.30	96.45	109.20
	R50	62.30	60.25	60.65	66.45	86.90	104.00	119.25
8	R0	27.10	26.65	28.15	38.30	61.60	30.65	32.20
	R05	61.20	57.80	61.15	65.25	67.40	68.45	72.90
	R1	63.00	58.05	63.90	62.40	65.75	69.20	69.85
	R5	67.65	61.35	62.60	61.20	69.50	69.55	72.60
	R10	68.15	61.60	64.95	62.65	73.90	74.85	74.60
	R20	69.00	59.85	65.60	41.00	75.15	74.00	77.70
	R25	68.95	59.05	62.75	38.10	70.60	76.85	80.60
	R50	68.60	60.05	48.70	38.70	78.25	79.55	82.30
16	R0	30.65	26.50	30.40	28.75	58.95	31.35	34.25
	R05	73.90	55.95	60.75	64.25	67.00	71.30	76.80
	R1	77.50	57.05	61.90	63.80	65.55	69.10	74.45
	R5	83.40	58.85	63.10	65.70	66.55	71.25	76.15
	R10	84.30	58.75	65.55	70.00	69.30	76.45	76.25
	R20	84.50	58.60	64.30	71.35	68.75	76.95	78.80
	R25	85.10	58.70	64.60	71.15	72.10	79.85	80.55
	R50	85.75	59.35	65.05	71.10	78.00	82.45	81.15
32	R0	35.05	26.20	27.25	28.45	63.15	32.65	39.15
	R05	100.75	54.75	59.35	61.75	65.30	73.20	103.00
	R1	107.25	54.50	60.10	64.50	67.10	72.50	88.40
	R5	113.90	56.90	60.65	64.55	68.05	73.35	88.65
	R10	115.35	56.45	61.30	63.90	68.15	74.40	87.55
	R20	117.60	56.55	60.20	65.80	67.45	74.35	84.20
	R25	118.20	56.60	60.40	64.30	68.60	74.75	84.65
	R50	120.20	56.90	60.40	65.10	68.45	74.85	87.40
64	R0	39.55	26.10	27.50	28.95	31.85	37.80	48.50
	R05	136.10	53.85	57.75	61.70	68.25	57.85	53.20
	R1	135.00	53.60	57.60	61.45	67.60	46.25	51.65
	R5	148.25	54.25	58.75	63.10	69.20	81.85	73.20
	R10	155.30	54.45	58.60	62.80	68.25	80.65	101.75
	R20	165.65	54.45	58.25	62.55	68.80	79.50	102.55
	R25	166.10	54.75	58.65	62.65	68.35	79.35	102.10
	R50	174.50	54.85	58.55	62.65	68.10	79.50	102.50

Table C.3: For comparing the effect on repair rate on the GA Sum of Time Differences per Generation.

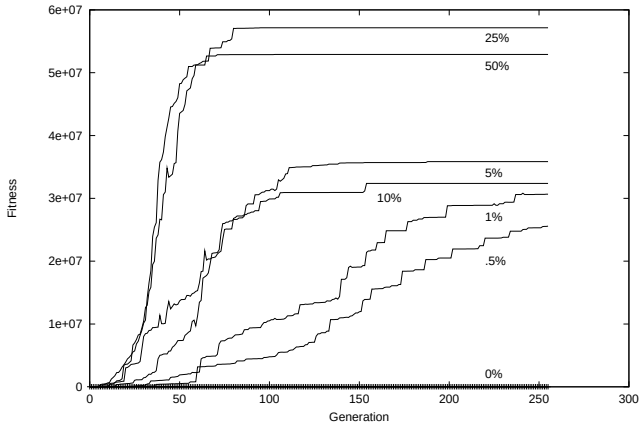

Figure C.1: For the GA–duplication of coding segments experiment and the fc4–8 graph, best fitness per generation. (Averaged over 20 runs)

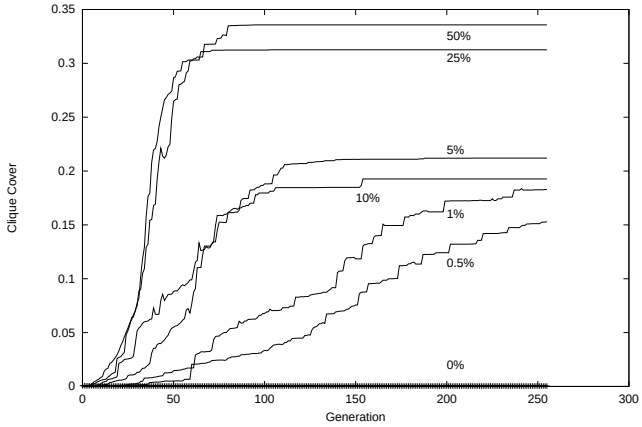

Figure C.2: For the GA–duplication of coding segments experiment and the fc4–8 graph, generational clique cover per generation. (Averaged over 20 runs)

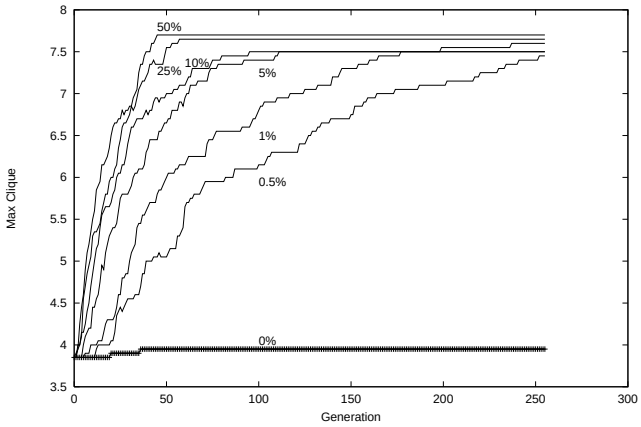

Figure C.3: For the GA–duplication of coding segments experiment and the fc4–8 graph, generational max clique per generation. (Averaged over 20 runs)

APPENDIX D

Data for Collective Adaptation and Duplication of Coding Segments for Heuristics

		1	2	4	8	16	32	64
1	CMRS	1	2	4	8	14	20	27
	CMHC	1	2	4	8	14	22	33
	CMSA	1	2	4	8	13	21	31
	CMGA	1	2	4	8	16	28	35
	CMGP	1	2	4	6.3	7.8	8.1	8.8
2	CMRS	1	2	4	7.8	12	16	20
	CMHC	1	2	4	7.5	13	20	32
	CMSA	1	2	4	7.7	13	20	30
	CMGA	1	2	4	8	15	22	26
	CMGP	1	2	4	4.9	5.2	5.1	5.5
4	CMRS	1	2	4	7.6	10	12	14
	CMHC	1	2	4	6.8	12	18	27
	CMSA	1	2	4	7.3	12	20	28
	CMGA	1	2	4	7.5	11	14	16
	CMGP	1	2	3.4	4	4.4	4	4
8	CMRS	1	2	3.9	6.8	9.6	10	11
	CMHC	1	2	4	6.5	12	17	23
	CMSA	1	2	4	6.8	12	17	25
	CMGA	1	2	3.5	6	6.7	8.3	8.3
	CMGP	1	2	3	3.3	3.7	3.5	3.1
16	CMRS	1	2	3.7	6.3	8.3	9.2	10
	CMHC	1	2	3.5	6.3	11	16	20
	CMSA	1	2	3.6	6.7	11	16	23
	CMGA	1	2	2.6	3.1	4.8	4.7	5.3
	CMGP	1	2	2.4	2.9	2.6	2.7	2.6
32	CMRS	1	2	3.3	6.2	8	8.9	9.3
	CMHC	1	2	3.3	5.7	9.8	14	18
	CMSA	1	2	3.4	5.8	9.9	14	17
	CMGA	1	1.8	2.4	2.8	2.8	3	2.5
	CMGP	1	2	2	2.4	2.1	2.3	2.5
64	CMRS	1	2	3.4	5.8	7	7.8	7.8
	CMHC	1	2	3.1	5.6	8.7	12	15
	CMSA	1	2	3	5.6	9.3	12	14
	CMGA	1	1.3	2	2.3	2.6	2	2.1
	CMGP	1	1.9	2	2.1	2.1	2.3	2.2

Table D.1: For comparing collective adaptation with duplication of coding segments on the heuristics the average maximal Generational Max Clique of Generation.

		1	2	4	8	16	32	64
1	CMRS	1	1	1	1	0.0035	4.9E-16	2.2E-58
	CMHC	1	1	1	1	0.11	6.7E-12	3.3E-43
	CMSA	1	1	1	1	0.056	1.2E-14	8E-45
	CMGA	1	1	1	1	1	0.00021	1.1E-39
	CMGP	1	1	1	0.15	1.2E-07	5.6E-31	2.3E-82
2	CMRS	1	1	1	0.5	6E-05	1.2E-20	1.3E-67
	CMHC	1	1	1	0.83	0.0044	6.4E-14	1.4E-47
	CMSA	1	1	1	0.8	0.0042	1E-14	8.1E-50
	CMGA	1	1	1	0.95	0.23	4.2E-11	3.8E-60
	CMGP	1	0.95	0.84	0.017	3.3E-11	1.6E-33	1.6E-86
4	CMRS	1	1	0.79	0.19	5E-06	1.3E-25	1.4E-76
	CMHC	1	1	1	0.22	0.0017	1.3E-15	1.4E-48
	CMSA	1	1	1	0.41	0.00026	5.9E-14	3.7E-50
	CMGA	1	0.99	0.87	0.19	2.5E-05	1.2E-19	1.2E-70
	CMGP	1	0.51	0.41	0.0015	1.2E-11	2.6E-34	4.7E-88
8	CMRS	1	0.99	0.32	0.04	4.7E-08	2.2E-27	5.3E-81
	CMHC	1	1	0.89	0.057	3.1E-05	1.4E-18	1.3E-54
	CMSA	1	1	0.85	0.11	0.0014	1.7E-18	5.3E-56
	CMGA	1	0.7	0.31	0.014	4.3E-10	2.7E-29	6.4E-80
	CMGP	1	0.26	0.13	0.00038	3.2E-12	4.4E-35	1E-88
16	CMRS	1	0.58	0.13	0.011	1.3E-08	1E-27	7.7E-83
	CMHC	1	0.92	0.39	0.027	2.1E-05	1.4E-18	1.8E-68
	CMSA	1	0.9	0.43	0.034	8.9E-05	2.1E-21	1.8E-50
	CMGA	1	0.23	0.059	0.0005	7.4E-11	5E-33	9E-86
	CMGP	1	0.094	0.041	0.00016	1.8E-13	1.1E-35	2.5E-89
32	CMRS	1	0.25	0.05	0.0059	6.4E-09	6.2E-31	2.2E-81
	CMHC	1	0.47	0.17	0.0072	3.8E-07	4.7E-22	4.9E-72
	CMSA	1	0.42	0.14	0.0069	6.3E-07	5.2E-23	1.9E-67
	CMGA	0.95	0.073	0.015	0.00021	9.5E-14	3.1E-36	2.1E-89
	CMGP	1	0.037	0.013	2E-05	3.4E-14	2.4E-36	7.2E-90
64	CMRS	0.93	0.096	0.02	0.0013	2.7E-11	6.2E-33	5E-85
	CMHC	1	0.15	0.051	0.0028	3.3E-06	2.1E-26	3.5E-74
	CMSA	1	0.12	0.047	0.0021	2.5E-08	3.2E-27	1.5E-74
	CMGA	0.73	0.018	0.0034	5.1E-06	1E-14	4.9E-37	1.6E-90
	CMGP	1	0.017	0.0038	4E-06	8.4E-15	6.7E-37	1.6E-90

Table D.2: For comparing collective adaptation with duplication of coding segments on the heuristics the average maximal Max Clique Cover of Generation.

		1	2	4	8	16	32	64
1	CMRS	1	2	4	8	14	20	28
	CMHC	1	2	4	8	15	24	36
	CMSA	1	2	4	8	14	23	36
	CMGA	1	2	4	8	16	28	38
	CMGP	1	2	4	8	16	31	55
2	CMRS	1	2	4	7.8	12	17	23
	CMHC	1	2	4	7.9	14	24	35
	CMSA	1	2	4	8	13	24	33
	CMGA	1	2	4	8	16	24	62
	CMGP	1	2	4	8	16	30	52
4	CMRS	1	2	4	7.6	11	15	22
	CMHC	1	2	4	7.8	14	22	30
	CMSA	1	2	4	8	14	23	30
	CMGA	1	2	4	8	14	20	21
	CMGP	1	2	4	8	15	27	39
8	CMRS	1	2	4	7.3	10	13	16
	CMHC	1	2	4	7.5	12	19	26
	CMSA	1	2	4	7.8	13	19	26
	CMGA	1	2	4	7.9	14	20	24
	CMGP	1	2	4	7.7	13	21	24
16	CMRS	1	2	4	6.5	8.5	10	11
	CMHC	1	2	4	7.3	12	17	20
	CMSA	1	2	4	7.3	12	17	24
	CMGA	1	2	4	7.3	11	15	16
	CMGP	1	2	4	7.1	10	13	15
32	CMRS	1	2	3.7	6.5	8.3	8.9	9.9
	CMHC	1	2	3.8	6.5	10	15	18
	CMSA	1	2	3.9	6.3	10	14	18
	CMGA	1	1.8	3.9	6.4	7.7	8.3	9.6
	CMGP	1	2	3.8	5.8	7.3	7.7	8.8
64	CMRS	1	2	3.5	5.8	7.1	8.2	8
	CMHC	1	2	3.6	5.8	8.9	12	15
	CMSA	1	2	3.5	5.8	9.3	12	15
	CMGA	1	1.3	2.9	4.5	4.5	4.8	4.6
	CMGP	1	1.9	3.3	4.1	5	4.5	5.3

Table D.3: For comparing collective adaptation with duplication of coding segments on the heuristics the average maximal Collective Memory Max Clique of Generation.

		1	2	4	8	16	32	64
1	CMRS	1	1	1	1	0.0069	5.4E-16	1E-53
	CMHC	1	1	1	1	0.5	9.4E-09	1E-36
	CMSA	1	1	1	1	0.35	1.7E-06	5.6E-40
	CMGA	1	1	1	1	1	0.0018	9.8E-30
	CMGP	1	1	1	1	0.9	0.6	1.3E-05
2	CMRS	1	1	1	0.96	0.00022	2.5E-19	6.2E-62
	CMHC	1	1	1	0.91	0.11	9E-07	1.2E-38
	CMSA	1	1	1	0.92	0.11	2.6E-05	6.8E-42
	CMGA	1	1	1	0.98	0.48	5E-05	0.13
	CMGP	1	1	0.99	0.52	0.5	0.1	1E-07
4	CMRS	1	1	1	0.46	7.1E-06	1.3E-15	2.6E-59
	CMHC	1	1	1	0.41	0.065	1.1E-09	1.4E-40
	CMSA	1	1	1	0.57	0.044	8.8E-07	1.4E-40
	CMGA	1	1	0.94	0.39	0.15	1.5E-08	3.3E-42
	CMGP	1	0.94	0.69	0.26	0.11	0.0004	1.2E-35
8	CMRS	1	1	0.81	0.12	2.8E-05	1.2E-18	3.1E-68
	CMHC	1	1	0.92	0.17	0.00038	7.7E-16	6.4E-52
	CMSA	1	1	0.88	0.22	0.015	3.7E-13	4.6E-53
	CMGA	1	0.74	0.42	0.12	0.021	5.2E-10	1.4E-56
	CMGP	1	0.63	0.29	0.099	0.0081	1.9E-11	1.4E-60
16	CMRS	1	0.92	0.47	0.027	1.6E-08	3.2E-26	1.9E-79
	CMHC	1	0.92	0.53	0.068	0.00051	3E-17	2.3E-67
	CMSA	1	0.9	0.56	0.062	0.0015	4.5E-20	1.7E-48
	CMGA	1	0.24	0.12	0.032	0.00021	4.8E-21	5.5E-71
	CMGP	1	0.24	0.098	0.029	2.9E-05	8.9E-25	2.8E-71
32	CMRS	1	0.47	0.2	0.01	7.2E-09	9.8E-31	2.3E-81
	CMHC	1	0.47	0.23	0.016	2.1E-06	3.2E-21	1.1E-70
	CMSA	1	0.42	0.2	0.014	3.3E-06	5.1E-22	1.9E-66
	CMGA	1	0.073	0.041	0.0066	3.9E-08	7.7E-29	1.6E-80
	CMGP	1	0.073	0.039	0.0028	9.7E-09	3.4E-30	1.1E-82
64	CMRS	1	0.16	0.058	0.0021	4E-11	1.6E-31	8.1E-85
	CMHC	1	0.15	0.058	0.0038	3.3E-06	1.2E-25	3.9E-72
	CMSA	1	0.12	0.059	0.0032	7.2E-08	1.5E-26	4.5E-74
	CMGA	0.94	0.019	0.0064	0.00027	1.8E-12	1.3E-32	2.5E-86
	CMGP	1	0.027	0.012	4.4E-05	3.8E-12	1.4E-35	3.1E-87

Table D.4: For comparing collective adaptation with duplication of coding segments on the heuristics the average maximal Collective Memory Clique Cover of Generation.

		1	2	4	8	16	32	64
1	CMRS	22.50	21.85	22.10	29.00	90.20	13049.50	71622.00
	CMHC	14.15	15.70	33.55	23.15	169.05	19966.30	137967.50
	CMSA	22.55	22.10	22.00	24.80	135.55	35776.00	109330.00
	CMGA	32.85	32.50	34.50	23.75	27.25	816.10	2041.95
	CMGP	88.20	95.40	169.85	454.25	521.60	521.25	521.00
2	CMRS	23.20	22.75	23.20	31.55	114.35	1319.45	8125.15
	CMHC	15.95	20.65	16.50	76.15	115.00	6522.60	18032.00
	CMSA	23.40	22.80	22.95	37.95	230.55	9201.10	32911.50
	CMGA	19.10	20.70	20.40	23.05	42.50	89.35	49.10
	CMGP	80.70	94.50	189.60	469.70	515.05	500.15	486.30
4	CMRS	24.95	24.05	30.50	35.30	82.25	330.20	754.05
	CMHC	14.40	14.50	16.25	35.30	93.45	1486.15	3478.05
	CMSA	25.00	24.30	26.20	45.90	171.10	1380.90	3701.45
	CMGA	20.90	23.25	24.50	27.80	36.50	63.00	45.60
	CMGP	85.40	85.55	238.05	440.45	418.15	452.15	507.35
8	CMRS	35.40	31.05	39.10	43.80	77.10	137.85	263.65
	CMHC	18.20	16.45	31.20	45.70	69.20	501.00	1080.40
	CMSA	28.35	26.30	41.35	49.30	117.80	784.95	1799.25
	CMGA	28.95	25.40	29.15	27.65	32.80	43.15	28.85
	CMGP	93.15	89.30	166.00	353.80	232.20	388.85	206.40
16	CMRS	35.10	25.25	29.35	33.35	44.95	70.45	95.75
	CMHC	23.20	18.05	35.10	40.45	49.20	81.30	128.20
	CMSA	35.15	28.20	52.35	52.90	65.50	130.80	224.80
	CMGA	31.65	22.65	59.50	67.75	65.10	52.95	53.65
	CMGP	237.25	83.10	128.75	227.65	274.35	276.15	290.70
32	CMRS	49.45	29.50	28.45	30.15	34.00	39.15	46.55
	CMHC	45.50	20.45	31.35	34.50	40.05	40.70	56.10
	CMSA	51.00	26.15	41.50	48.40	51.20	130.30	164.80
	CMGA	88.25	38.40	41.30	44.60	45.70	30.60	42.65
	CMGP	496.85	86.20	107.70	160.00	182.40	176.80	222.00
64	CMRS	70.25	23.25	26.15	77.75	28.75	30.55	33.40
	CMHC	59.65	16.25	23.55	27.85	31.15	37.40	34.35
	CMSA	88.35	23.95	32.90	37.95	44.50	48.55	52.85
	CMGA	98.15	35.85	40.25	43.90	46.85	59.60	82.10
	CMGP	910.45	107.30	106.50	113.80	127.60	151.80	109.50

Table D.5: For comparing collective adaptation with duplication of coding segments on the heuristics Sum of Time Differences per Generation.

APPENDIX E

Data for Varying Collective Adaptation for GA

		1	2	4	8	16	32	64
1	GA	1	2	4	7.3	12	17	26
	CM	1	2	4	8	16	28	35
	PA	1	2	4	8	16	28	35
	MG64	1	2	4	8	16	32	64
	MG128	1	2	4	8	16	32	64
	LG64	1	2	4	8	16	32	64
	LG128	1	2	4	8	16	32	64
2	GA	1	2	3.9	5.7	6.8	8.3	8.7
	CM	1	2	4	8	15	22	26
	PA	1	2	4	8	15	22	26
	MG64	1	2	4	8	16	32	64
	MG128	1	2	4	8	16	32	64
	LG64	1	2	4	8	16	32	64
	LG128	1	2	4	8	16	32	64
4	GA	1	2	3.4	4	4.2	4.7	4.7
	CM	1	2	4	7.5	11	14	16
	PA	1	2	4	7.5	11	14	16
	MG64	1	2	4	8	16	32	64
	MG128	1	2	4	8	16	32	64
	LG64	1	2	4	8	16	30	51
	LG128	1	2	4	8	16	31	54
8	GA	1	2	3	3.1	3	3.5	3.6
	CM	1	2	3.5	6	7.6	8.3	8.3
	PA	1	2	3.5	6	6.7	8.3	8.3
	MG64	1	2	4	8	16	32	64
	MG128	1	2	4	8	16	32	64
	LG64	1	2	4	8	15	23	28
	LG128	1	2	4	8	15	24	32
16	GA	1	2	2.2	2.4	2.5	2.8	3
	CM	1	2	2.6	3.1	4.3	4.7	5.3
	PA	1	2	2.6	3.1	4.8	4.7	5.3
	MG64	1	2	4	8	16	32	64
	MG128	1	2	4	8	16	32	64
	LG64	1	2	4	7.3	12	15	17
	LG128	1	2	4	7	11	15	17
32	GA	1	1.8	2	2	2.1	2.4	2.3
	CM	1	1.8	2.4	2.8	2.7	3	2.5
	PA	1	1.8	2.4	2.8	2.8	3	2.5
	MG64	1	1.9	4	8	16	32	64
	MG128	1	2	4	8	16	32	64
	LG64	1	1.9	4	6.3	8.3	9.8	9.9
	LG128	1	2	3.9	6	8	9.3	10
64	GA	1	1.3	1.9	2	2	1.9	1.9
	CM	1	1.3	2	2.3	2.6	2	2.1
	PA	1	1.3	2	2.3	2.6	2	2.1
	MG64	1	1.4	4	8	16	32	62
	MG128	1	1.8	4	8	16	32	64
	LG64	1	1.5	3.4	5	5.2	6	6
	LG128	1	2	3.4	4	5.3	5.8	5.6

Table E.1: For comparing the effect of collective adaptation with GA the average maximal Generational Max Clique of Generation.

		1	2	4	8	16	32	64
1	GA	1	1	1	0.53	7.1E-05	1E-17	5.4E-48
	CM	1	1	1	1	1	0.00021	1.1E-39
	PA	1	1	1	1	1	0.00021	1.1E-39
	MG64	1	1	1	1	1	1	1
	MG128	1	1	1	1	1	1	1
	LG64	1	1	1	1	1	1	1
	LG128	1	1	1	1	1	1	1
2	GA	1	1	0.9	0.017	6.6E-10	8.1E-30	2E-83
	CM	1	1	1	0.95	0.23	4.2E-11	5.2E-56
	PA	1	1	1	0.95	0.23	4.2E-11	3.8E-60
	MG64	1	1	1	1	1	1	1
	MG128	1	1	1	1	1	1	1
	LG64	1	1	1	1	1	0.95	0.38
	LG128	1	1	1	1	1	1	0.95
4	GA	1	0.99	0.35	0.00043	1.5E-12	2.9E-34	8.2E-88
	CM	1	0.99	0.87	0.19	2.5E-05	1.2E-19	1.2E-70
	PA	1	0.99	0.87	0.19	2.5E-05	1.2E-19	1.2E-70
	MG64	1	0.99	1	1	1	1	0.75
	MG128	1	1	1	1	1	1	0.75
	LG64	1	1	0.98	0.81	0.29	0.027	7.1E-17
	LG128	1	1	1	0.99	0.64	0.1	8.5E-10
8	GA	1	0.68	0.11	8.5E-05	1.7E-13	3.6E-35	5E-89
	CM	1	0.7	0.31	0.014	2.4E-08	2.7E-29	6.4E-80
	PA	1	0.7	0.31	0.014	4.3E-10	2.7E-29	6.4E-80
	MG64	1	0.67	0.88	0.91	0.94	0.79	0.38
	MG128	1	0.93	0.97	0.99	1	0.79	0.38
	LG64	1	0.68	0.44	0.15	0.028	7.5E-09	5.2E-52
	LG128	1	0.92	0.78	0.34	0.042	2.2E-08	7.2E-49
16	GA	0.99	0.26	0.036	2.5E-05	5.2E-14	5.9E-36	1.1E-89
	CM	1	0.23	0.059	0.0005	5.6E-11	5E-33	9E-86
	PA	1	0.23	0.059	0.0005	7.4E-11	5E-33	9E-86
	MG64	1	0.22	0.4	0.41	0.43	0.36	0.17
	MG128	1	0.45	0.6	0.7	0.69	0.4	0.18
	LG64	1	0.23	0.12	0.034	0.00021	2E-21	2.8E-71
	LG128	1	0.45	0.2	0.029	0.00021	4.6E-22	6.4E-70
32	GA	0.74	0.078	0.0098	7.5E-06	1.5E-14	1.9E-36	2.9E-90
	CM	0.95	0.073	0.015	0.00021	4E-14	3.1E-36	2.1E-89
	PA	0.95	0.073	0.015	0.00021	9.5E-14	3.1E-36	2.1E-89
	MG64	1	0.067	0.098	0.13	0.13	0.12	0.046
	MG128	1	0.14	0.19	0.25	0.26	0.18	0.086
	LG64	1	0.07	0.038	0.0064	6.9E-09	2.1E-24	1.6E-80
	LG128	1	0.15	0.048	0.0045	9.6E-10	5.2E-28	1.6E-80
64	GA	0.46	0.019	0.0025	2E-06	4E-15	4.1E-37	7.4E-91
	CM	0.73	0.018	0.0034	5.1E-06	1E-14	4.9E-37	1.6E-90
	PA	0.73	0.018	0.0034	5.1E-06	1E-14	4.9E-37	1.6E-90
	MG64	0.94	0.017	0.026	0.033	0.034	0.029	0.0024
	MG128	1	0.037	0.051	0.065	0.073	0.067	0.02
	LG64	0.93	0.018	0.0098	0.00027	3.4E-12	2.4E-31	3.8E-83
	LG128	1	0.041	0.012	2.6E-05	1.7E-11	1.3E-32	2.5E-85

Table E.2: For comparing the effect of collective adaptation with GA the average maximal Max Clique Cover of Generation.

		1	2	4	8	16	32	64
1	GA	1	2	4	7.3	12	17	26
	CM	1	2	4	8	16	28	35
	PA	1	2	4	8	16	28	38
	MG64	1	2	4	8	16	32	64
	MG128	1	2	4	8	16	32	64
	LG64	1	2	4	8	16	32	64
	LG128	1	2	4	8	16	32	64
2	GA	1	2	3.9	5.7	6.8	8.3	8.7
	CM	1	2	4	8	15	22	26
	PA	1	2	4	8	16	24	62
	MG64	1	2	4	8	16	32	64
	MG128	1	2	4	8	16	32	64
	LG64	1	2	4	8	16	32	64
	LG128	1	2	4	8	16	32	64
4	GA	1	2	3.4	4	4.2	4.7	4.7
	CM	1	2	4	7.5	11	14	16
	PA	1	2	4	8	14	20	21
	MG64	1	2	4	8	16	32	64
	MG128	1	2	4	8	16	32	64
	LG64	1	2	4	8	16	30	51
	LG128	1	2	4	8	16	31	54
8	GA	1	2	3	3.1	3	3.5	3.6
	CM	1	2	3.5	6	7.6	8.3	8.3
	PA	1	2	4	7.9	14	20	24
	MG64	1	2	4	8	16	32	64
	MG128	1	2	4	8	16	32	64
	LG64	1	2	4	8	15	23	28
	LG128	1	2	4	8	15	24	32
16	GA	1	2	2.2	2.4	2.5	2.8	3
	CM	1	2	2.6	3.1	4.3	4.7	5.3
	PA	1	2	4	7.3	11	15	16
	MG64	1	2	4	8	16	32	64
	MG128	1	2	4	8	16	32	64
	LG64	1	2	4	7.3	12	16	17
	LG128	1	2	4	7	11	15	17
32	GA	1	1.8	2	2	2.1	2.4	2.3
	CM	1	1.8	2.4	2.8	2.7	3	2.5
	PA	1	1.8	3.9	6.4	7.7	8.3	9.6
	MG64	1	1.9	4	8	16	32	64
	MG128	1	2	4	8	16	32	64
	LG64	1	1.9	4	6.3	8.3	9.8	10
	LG128	1	2	3.9	6	8	9.3	10
64	GA	1	1.3	1.9	2	2	1.9	1.9
	CM	1	1.3	2	2.3	2.6	2	2.1
	PA	1	1.3	2.9	4.5	4.5	4.8	4.6
	MG64	1	1.4	4	8	16	32	62
	MG128	1	1.8	4	8	16	32	64
	LG64	1	1.5	3.4	5	5.2	6	6
	LG128	1	2	3.4	4	5.3	5.8	5.6

Table E.3: For comparing the effect of collective adaptation with GA the average maximal Collective Memory Max Clique of Generation.

		1	2	4	8	16	32	64
1	GA	1	1	1	0.53	7.1E-05	1E-17	5.4E-48
	CM	1	1	1	1	1	0.00022	2.3E-39
	PA	1	1	1	1	1	0.0018	9.8E-30
	MG64	1	1	1	1	1	1	1
	MG128	1	1	1	1	1	1	1
	LG64	1	1	1	1	1	1	1
	LG128	1	1	1	1	1	1	1
2	GA	1	1	0.9	0.017	6.6E-10	8.1E-30	2E-83
	CM	1	1	1	0.95	0.23	4.4E-11	1E-55
	PA	1	1	1	0.98	0.48	5E-05	0.13
	MG64	1	1	1	1	1	1	1
	MG128	1	1	1	1	1	1	1
	LG64	1	1	1	1	1	0.95	0.38
	LG128	1	1	1	1	1	1	0.95
4	GA	1	0.99	0.35	0.00043	1.5E-12	2.9E-34	8.2E-88
	CM	1	1	0.9	0.19	2.8E-05	1.2E-19	1.2E-70
	PA	1	1	0.94	0.39	0.15	1.5E-08	3.3E-42
	MG64	1	0.99	1	1	1	1	1
	MG128	1	1	1	1	1	1	1
	LG64	1	1	0.98	0.81	0.29	0.027	7.1E-17
	LG128	1	1	1	0.99	0.64	0.1	8.5E-10
8	GA	1	0.68	0.11	8.5E-05	1.7E-13	3.6E-35	5E-89
	CM	1	0.74	0.34	0.014	2.4E-08	2.7E-29	6.4E-80
	PA	1	0.74	0.42	0.12	0.021	5.2E-10	1.4E-56
	MG64	1	0.67	0.88	0.91	0.94	0.94	0.89
	MG128	1	0.93	0.97	0.99	1	0.99	1
	LG64	1	0.68	0.44	0.15	0.028	7.5E-09	5.2E-52
	LG128	1	0.92	0.78	0.34	0.042	2.2E-08	7.2E-49
16	GA	0.99	0.26	0.036	2.5E-05	5.2E-14	5.9E-36	1.1E-89
	CM	1	0.24	0.064	0.00051	5.6E-11	5E-33	9E-86
	PA	1	0.24	0.12	0.032	0.00021	4.8E-21	5.5E-71
	MG64	1	0.22	0.4	0.41	0.43	0.45	0.27
	MG128	1	0.45	0.6	0.7	0.71	0.74	0.66
	LG64	1	0.23	0.12	0.034	0.00021	2E-21	2.8E-71
	LG128	1	0.45	0.2	0.029	0.00021	4.6E-22	6.4E-70
32	GA	0.74	0.078	0.0098	7.5E-06	1.5E-14	1.9E-36	2.9E-90
	CM	1	0.073	0.016	0.00021	4.2E-14	3.3E-36	2.1E-89
	PA	1	0.073	0.041	0.0066	3.9E-08	7.7E-29	1.6E-80
	MG64	1	0.067	0.098	0.13	0.13	0.12	0.047
	MG128	1	0.14	0.19	0.25	0.26	0.23	0.16
	LG64	1	0.07	0.038	0.0064	6.9E-09	2.1E-24	1.6E-80
	LG128	1	0.15	0.048	0.0045	9.7E-10	5.2E-28	1.6E-80
64	GA	0.46	0.019	0.0025	2E-06	4E-15	4.1E-37	7.4E-91
	CM	0.94	0.019	0.0035	5.2E-06	1.1E-14	5E-37	1.7E-90
	PA	0.94	0.019	0.0064	0.00027	1.8E-12	1.3E-32	2.5E-86
	MG64	0.94	0.017	0.026	0.033	0.034	0.029	0.0024
	MG128	1	0.037	0.051	0.065	0.073	0.071	0.021
	LG64	0.93	0.018	0.0098	0.00027	3.4E-12	2.4E-31	3.8E-83
	LG128	1	0.041	0.012	2.6E-05	1.7E-11	1.3E-32	2.5E-85

Table E.4: For comparing the effect of collective adaptation with GA the average maximal Collective Memory Clique Cover of Generation.

		1	2	4	8	16	32	64
1	BASE	55.70	51.70	104.30	83.00	62.50	35.50	59.25
	CMDU	56.90	21.85	18.85	21.85	56.85	247.55	3086.10
	CMAP	32.85	32.50	34.50	23.75	27.25	816.10	2041.95
	P64	57.05	57.35	57.10	58.20	66.10	82.10	157.35
	P128	115.00	112.90	112.60	117.90	134.85	165.25	308.10
	L64	56.65	55.05	55.55	59.10	66.25	81.65	132.95
	L128	113.15	110.65	111.95	119.20	131.70	163.95	157.70
2	BASE	53.35	36.85	26.50	28.30	47.70	29.40	31.20
	CMDU	17.40	17.30	19.20	21.95	37.20	75.70	160.45
	CMAP	19.10	20.70	20.40	23.05	42.50	89.35	49.10
	P64	60.25	58.45	57.30	58.60	65.45	80.50	125.75
	P128	118.95	115.00	96.25	118.35	131.85	161.60	286.80
	L64	58.00	56.70	57.80	58.35	64.50	77.30	97.75
	L128	116.55	114.30	98.30	116.95	129.45	155.55	219.40
4	BASE	26.05	27.10	27.10	28.25	61.50	29.15	30.45
	CMDU	58.80	19.95	19.90	40.50	80.85	62.80	73.20
	CMAP	20.90	23.25	24.50	27.80	36.50	63.00	45.60
	P64	61.35	59.60	59.15	61.55	69.55	88.10	160.65
	P128	122.80	119.65	119.30	123.90	138.80	179.55	395.25
	L64	60.30	58.90	58.25	60.30	65.20	72.75	81.85
	L128	121.30	118.00	118.00	121.65	134.40	153.05	179.30
8	BASE	27.10	26.65	28.15	38.30	61.60	30.65	32.20
	CMDU	62.90	60.15	66.70	66.10	70.40	33.60	29.80
	CMAP	28.95	25.40	29.15	27.65	32.80	43.15	28.85
	P64	68.05	61.65	64.30	67.80	77.95	72.80	133.80
	P128	116.80	132.15	131.05	137.45	156.65	101.75	433.20
	L64	66.30	60.65	63.05	64.25	67.75	72.00	80.30
	L128	118.10	128.90	127.70	131.30	138.05	149.45	165.70
16	BASE	30.65	26.50	30.40	28.75	58.95	31.35	34.25
	CMDU	33.65	23.70	28.00	32.05	69.35	56.30	55.35
	CMAP	31.65	22.65	59.50	67.75	65.10	52.95	53.65
	P64	84.60	59.45	63.90	68.10	78.60	101.65	219.10
	P128	172.65	132.95	138.95	129.90	166.45	251.05	1042.00
	L64	80.15	59.00	63.20	65.20	68.40	76.30	88.80
	L128	163.15	129.00	135.40	113.80	147.85	117.90	189.10
32	BASE	35.05	26.20	27.25	28.45	63.15	32.65	39.15
	CMDU	40.00	20.75	45.25	45.95	63.20	49.90	62.20
	CMAP	88.25	38.40	41.30	44.60	45.70	30.60	42.65
	P64	128.05	58.25	61.10	67.05	79.75	102.50	141.20
	P128	261.70	125.35	130.90	147.30	173.40	223.70	466.80
	L64	111.10	58.10	60.50	64.60	71.00	83.50	106.25
	L128	226.80	124.45	130.00	140.40	153.70	177.05	226.65
64	BASE	39.55	26.10	27.50	28.95	31.85	37.80	48.50
	CMDU	52.90	20.70	21.85	24.75	26.95	34.95	43.85
	CMAP	98.15	35.85	40.25	43.90	46.85	59.60	82.10
	P64	223.00	58.40	62.00	70.80	88.00	121.20	152.00
	P128	410.80	120.40	129.40	143.45	184.15	247.45	183.20
	L64	167.35	57.75	61.60	68.05	79.50	103.40	80.35
	L128	345.05	119.60	128.35	142.45	166.75	213.90	153.25

Table E.5: For comparing the effect of collective adaptation with GA Sum of Time Differences per Generation.

APPENDIX F

Data for Investigating GP and FC graphs

		1	2	4	8	16	32	64
1	B64	1	2	4	5.9	8.1	9.1	9.8
	B256	0.95	2	4	6.9	9.8	10	12
	R64	1	2	4	5.6	7.9	8.4	9.1
	R256	1	2	4	6.1	8.4	10	12
	C64	1	2	4	6.3	7.8	8.1	8.8
	C256	1	2	4	6.3	8.3	8.6	11
	X64	1	2	4	6.4	9.3	13	18
	X256	1	2	4	8	11	11	12
2	B64	1	2	3.5	4.8	5.6	5.5	5.6
	B256	1	2	4	6.2	7.5	6.8	6.9
	R64	1	2	3.9	5	5.3	4.8	5.3
	R256	1	2	4	5	6.2	6.8	6.9
	C64	1	2	4	4.9	5.2	5.1	5.5
	C256	1	2	4	5.1	5.8	6	6.4
	X64	1	2	4	5.9	9.5	12	20
	X256	1	2	4	7.7	7.7	7.4	8.2
4	B64	1	2	3.3	3.9	4.2	4	4
	B256	1	2	3.9	5	5.5	4.8	5.1
	R64	1	2	3.4	3.9	4.2	4.1	4.2
	R256	1	2	3.9	4.5	5	4.8	5.1
	C64	1	2	3.4	4	4.4	4	4
	C256	1	2	3.8	4.6	5.1	4.5	4.6
	X64	1	2	3.9	6.3	9.3	12	18
	X256	1	2	3.8	6.3	5.5	7.5	10
8	B64	1	2	2.8	3.5	3.8	3.1	3.2
	B256	1	2	3.5	4.3	4.5	3.8	3.8
	R64	1	2	3	3.3	3.4	3.3	3.3
	R256	1	2	3.4	4	4.3	3.8	3.8
	C64	1	2	3	3.3	3.7	3.5	3.1
	C256	1	2	3.3	4	4.5	3.8	4
	X64	1	2	3.8	5.4	7.5	9.6	11
	X256	1	2	3.7	5.2	6.3	8.1	11
16	B64	1	2	2.4	3.2	2.9	2.6	2.9
	B256	1	2	3.3	3.4	3.9	3.5	3.4
	R64	1	2	2.5	2.9	2.9	2.8	2.8
	R256	1	2	3.3	3.4	3.9	3.5	3.2
	C64	1	2	2.4	2.9	2.6	2.7	2.6
	C256	1	2	3.1	3.1	3.3	3.1	3.1
	X64	1	1.6	3.6	4	5	6.2	6.6
	X256	1	2	3.4	4.3	5.7	7.3	8.3
32	B64	1	2	2.1	2.4	2.5	2.5	2.5
	B256	1	2	2.7	3.1	2.9	2.7	2.9
	R64	1	2	2.3	2.4	2.2	2.4	2.4
	R256	1	2	2.7	3.1	2.9	2.7	2.9
	C64	1	2	2	2.4	2.1	2.3	2.5
	C256	1	2	2.5	2.5	2.7	2.5	2.7
	X64	1	1.5	3.3	3.5	3.6	3.8	4.1
	X256	1	2	3.2	4.2	5	3.6	5
64	B64	1	2	1.8	2.2	2	2.3	2.1
	B256	1	2	2.4	2.6	2.8	2.5	2.5
	R64	1	2	2	2.1	2.1	2.1	2.2

Continued on next page

		1	2	4	8	16	32	64
	R256	1	2	2.2	2.2	2.6	2.5	2.3
	C64	1	1.9	2	2.1	2.1	2.3	2.2
	C256	1.1	2	2.4	2.5	2.5	2.5	2.3
	X64	1	0.9	3	3.1	3.2	3.2	3.4
	X256	1	1.7	3.1	3.4	3.6	3.8	4.3

Table F.1: For comparing the effect of collective adaptation on the GP the average maximal Generational Max Clique of Generation.

		1	2	4	8	16	32	64
1	B64	1	1	1	0.033	3E-08	2.1E-28	4.4E-82
	B256	0.95	1	1	0.23	1.6E-06	2.5E-25	1.8E-80
	R64	1	1	1	0.041	5.8E-09	9.4E-30	1.5E-80
	R256	1	1	1	0.054	2.5E-08	2.5E-25	1.8E-80
	C64	1	1	1	0.15	1.2E-07	5.6E-31	2.3E-82
	C256	1	1	1	0.083	2.5E-08	0.037	8.3E-78
	X64	1	1	1	0.14	0.00022	2.9E-12	6.5E-57
	X256	1	1	1	1	0.00021	3.9E-27	8.9E-80
2	B64	1	0.93	0.42	0.002	4.6E-10	5.3E-33	6E-87
	B256	1	1	0.79	0.07	1.7E-09	5.4E-32	4.4E-86
	R64	1	0.97	0.9	0.013	3.4E-11	8.9E-34	1.8E-86
	R256	1	1	0.88	0.011	1.8E-10	5.4E-32	4.4E-86
	C64	1	0.95	0.84	0.017	3.3E-11	1.6E-33	1.6E-86
	C256	1	1	0.89	0.013	1E-10	4.5E-32	7.5E-85
	X64	1	1	0.91	0.08	0.05	4.6E-19	1.6E-55
	X256	1	1	1	0.46	4.8E-08	7.6E-32	1.9E-82
4	B64	1	0.4	0.15	0.00025	8.2E-13	6.9E-35	1.4E-88
	B256	1	0.65	4E-90	8.1E-89	1.6E-88	1.5E-34	9.1E-88
	R64	1	0.7	0.46	0.0021	6.3E-12	2.1E-34	5.9E-88
	R256	1	0.99	0.6	0.0028	4.4E-11	1.5E-34	9.1E-88
	C64	1	0.51	0.41	0.0015	1.2E-11	2.6E-34	4.7E-88
	C256	1	1	0.57	0.0033	0.037	3.3E-34	5.7E-88
	X64	1	0.78	0.45	0.081	0.0017	1.3E-15	7.9E-56
	X256	1	0.99	0.54	0.049	4.5E-10	5.8E-21	1.5E-66
8	B64	1	0.17	0.043	7.9E-05	2.7E-13	5.3E-36	1.5E-89
	B256	1	0.27	0.087	0.00019	3.5E-12	1.1E-35	3.5E-89
	R64	1	0.23	0.13	0.00039	1.6E-12	5.3E-35	1.2E-88
	R256	1	0.61	0.27	0.0017	4.4E-12	8.2E-35	1.9E-88
	C64	1	0.26	0.13	0.00038	3.2E-12	4.4E-35	1E-88
	C256	1	0.6	0.27	0.00079	0.00035	7.7E-35	0.037
	X64	1	0.34	0.16	0.027	2.8E-05	2.7E-17	1.5E-66
	X256	1	0.61	0.22	0.02	2.6E-05	3.7E-13	1.1E-70
16	B64	1	0.069	0.013	2.7E-05	4.1E-14	1.2E-36	3.9E-90
	B256	1	0.13	0.037	3.5E-05	2.4E-13	4.2E-36	7.3E-90
	R64	1	0.091	0.043	0.00011	3.4E-13	1.1E-35	2.3E-89
	R256	1	0.13	0.037	3.5E-05	2.4E-13	4.2E-36	7E-89
	C64	1	0.094	0.041	0.00016	1.8E-13	1.1E-35	2.5E-89
	C256	1	0.22	0.086	0.00023	9.1E-13	0.00042	6.9E-89
	X64	1	0.13	0.066	0.00018	6E-09	4.2E-24	2.1E-81
	X256	1	0.28	0.071	0.0036	1.9E-06	1.1E-27	3E-75
32	B64	1	0.034	0.0039	4.7E-06	9.1E-15	4.9E-37	1.2E-90
	B256	1	0.052	0.01	1.7E-05	2.8E-14	7.8E-37	2E-90
	R64	1	0.039	0.015	1.8E-05	3.5E-14	3.1E-36	6.7E-90
	R256	1	0.052	0.01	1.7E-05	2.8E-14	7.8E-37	2E-90
	C64	1	0.037	0.013	2E-05	3.4E-14	2.4E-36	7.2E-90
	C256	1	0.077	0.029	3.5E-05	1.4E-13	9.1E-36	1.9E-89
	X64	1	0.047	0.023	5.5E-05	7.1E-14	1.4E-35	2.7E-89
	X256	1	0.087	0.027	0.0018	3E-09	2.2E-32	5.9E-85
64	B64	1	0.016	0.0012	1.4E-06	1.7E-15	1.8E-37	3.3E-91
	B256	1	0.02	0.003	2.8E-06	6.6E-15	3.3E-37	5.2E-91
	R64	1	0.017	0.0046	4.4E-06	8.3E-15	8.3E-37	1.5E-90
	R256	1	0.028	0.0096	1.1E-05	3.3E-14	2.5E-36	6.1E-90
	C64	1	0.017	0.0038	4E-06	8.4E-15	6.7E-37	1.6E-90
	C256	1	0.028	0.0087	1.5E-05	3.6E-14	2.2E-36	5.2E-90
	X64	1	0.011	0.0077	1.1E-05	2.5E-14	1.8E-36	3.6E-90
	X256	1	0.029	0.01	1.2E-05	1E-13	9.9E-36	2.3E-86

Table F.2: For comparing the effect of collective adaptation on the GP the average maximal Max Clique Cover of Generation.

		1	2	4	8	16	32	64
1	B64	1	2	4	5.9	8.1	9.1	9.8
	B256	0.95	2	4	6.9	9.8	10	12
	R64	1	2	4	5.6	7.9	8.4	9.1
	R256	1	2	4	6.1	8.4	10	12
	C64	1	2	4	8	16	31	55
	C256	1	2	4	7.4	8.7	11	16
	X64	1	2	4	6.4	9.3	13	18
	X256	1	2	4	8	11	11	12
2	B64	1	2	3.5	4.8	5.6	5.5	5.6
	B256	1	2	4	6.2	7.5	6.8	6.9
	R64	1	2	3.9	5	5.3	4.8	5.3
	R256	1	2	4	5	6.2	6.8	6.9
	C64	1	2	4	8	16	30	52
	C256	1	2	4	7.1	10	17	27
	X64	1	2	4	5.9	9.6	13	22
	X256	1	2	4	7.7	7.7	7.4	8.2
4	B64	1	2	3.3	3.9	4.2	4	4
	B256	1	2	3.9	5	5.5	4.8	5.1
	R64	1	2	3.4	3.9	4.2	4.1	4.2
	R256	1	2	3.9	4.5	5	4.8	5.1
	C64	1	2	4	8	15	27	39
	C256	1	2	4	7.4	11	21	32
	X64	1	2	3.9	6.3	9.3	12	19
	X256	1	2	3.8	6.3	5.5	7.5	10
8	B64	1	2	2.8	3.5	3.8	3.1	3.2
	B256	1	2	3.5	4.3	4.5	3.8	3.8
	R64	1	2	3	3.3	3.4	3.3	3.3
	R256	1	2	3.4	4	4.3	3.8	3.8
	C64	1	2	4	7.7	13	21	24
	C256	1	2	4	6.3	9.9	20	21
	X64	1	2	3.8	5.5	7.5	9.7	11
	X256	1	2	3.7	5.2	6.3	8.1	11
16	B64	1	2	2.4	3.2	2.9	2.6	2.9
	B256	1	2	3.3	3.4	3.9	3.5	3.4
	R64	1	2	2.5	2.9	2.9	2.8	2.8
	R256	1	2	3.3	3.4	3.9	3.5	3.2
	C64	1	2	4	7.1	10	13	15
	C256	1	2	3.9	6.2	7.7	12	13
	X64	1	1.6	3.6	4	5.1	6.2	6.6
	X256	1	2	3.4	4.3	5.7	7.3	8.4
32	B64	1	2	2.1	2.4	2.5	2.5	2.5
	B256	1	2	2.7	3.1	2.9	2.7	2.9
	R64	1	2	2.3	2.4	2.2	2.4	2.4
	R256	1	2	2.7	3.1	2.9	2.7	2.9
	C64	1	2	3.8	5.8	7.3	7.7	8.8
	C256	1	2	3.8	5.7	6.7	7.9	8.1
	X64	1	1.5	3.3	3.5	3.6	3.8	4.1
	X256	1	2	3.2	4.2	5	3.6	5
64	B64	1	2	1.8	2.2	2	2.3	2.1
	B256	1	2	2.4	2.6	2.8	2.5	2.5
	R64	1	2	2	2.1	2.1	2.1	2.2
	R256	1	2	2.2	2.2	2.6	2.5	2.3
	C64	1	1.9	3.3	4.1	5	4.5	5.3
	C256	1.1	2	3.4	4.1	4.8	5.3	5.7
	X64	1	0.9	3	3.1	3.2	3.2	3.4
	X256	1	1.7	3.1	3.4	3.6	3.8	4.3

Table F.3: For comparing the effect of collective adaptation on the GP the average maximal Collective Memory Max Clique of Generation.

		1	2	4	8	16	32	64
1	B64	1	1	1	0.033	3E-08	2.1E-28	4.4E-82
	B256	0.95	1	1	0.23	1.6E-06	2.5E-25	1.8E-80
	R64	1	1	1	0.041	5.8E-09	9.4E-30	1.5E-80
	R256	1	1	1	0.054	2.5E-08	2.5E-25	1.8E-80
	C64	1	1	1	1	0.9	0.6	1.3E-05
	C256	1	1	1	0.61	1.5E-06	0.05	6.8E-54
	X64	1	1	1	0.22	0.00023	2.9E-12	6.5E-57
	X256	1	1	1	1	0.00021	4.4E-27	5.9E-79
2	B64	1	0.93	0.42	0.002	4.6E-10	5.3E-33	6E-87
	B256	1	1	0.79	0.07	1.7E-09	5.4E-32	4.4E-86
	R64	1	0.97	0.9	0.013	3.4E-11	8.9E-34	1.8E-86
	R256	1	1	0.88	0.011	1.8E-10	5.4E-32	4.4E-86
	C64	1	1	0.99	0.52	0.5	0.1	1E-07
	C256	1	1	1	0.27	0.0017	1.5E-12	7.1E-42
	X64	1	1	1	0.091	0.05	6.9E-19	2.4E-28
	X256	1	1	1	0.47	5E-08	2.1E-31	1.4E-81
4	B64	1	0.4	0.15	0.00025	8.2E-13	6.9E-35	1.4E-88
	B256	1	0.65	4E-90	8.1E-89	1.6E-88	1.5E-34	9.1E-88
	R64	1	0.7	0.46	0.0021	6.3E-12	2.1E-34	5.9E-88
	R256	1	0.99	0.6	0.0028	4.4E-11	1.5E-34	9.1E-88
	C64	1	0.94	0.69	0.26	0.11	0.0004	1.2E-35
	C256	1	1	0.84	0.17	0.05	4.2E-07	1.2E-35
	X64	1	0.93	0.62	0.085	0.0017	3E-14	2.1E-45
	X256	1	1	0.86	0.066	7.9E-10	6.1E-21	4.7E-66
8	B64	1	0.17	0.043	7.9E-05	2.7E-13	5.3E-36	1.5E-89
	B256	1	0.27	0.087	0.00019	3.5E-12	1.1E-35	3.5E-89
	R64	1	0.23	0.13	0.00039	1.6E-12	5.3E-35	1.2E-88
	R256	1	0.61	0.27	0.0017	4.4E-12	8.2E-35	1.9E-88
	C64	1	0.63	0.29	0.099	0.0081	1.9E-11	1.4E-60
	C256	1	0.96	0.48	0.05	0.008	9.6E-12	0.05
	X64	1	0.44	0.25	0.028	2.8E-05	2.8E-17	1.1E-63
	X256	1	0.84	0.47	0.021	2.6E-05	3.7E-13	1.4E-69
16	B64	1	0.069	0.013	2.7E-05	4.1E-14	1.2E-36	3.9E-90
	B256	1	0.13	0.037	3.5E-05	2.4E-13	4.2E-36	7.3E-90
	R64	1	0.091	0.043	0.00011	3.4E-13	1.1E-35	2.3E-89
	R256	1	0.13	0.037	3.5E-05	2.4E-13	4.2E-36	7E-89
	C64	1	0.24	0.098	0.029	2.9E-05	8.9E-25	2.8E-71
	C256	1	0.57	0.19	0.016	1.1E-06	0.05	1.3E-72
	X64	1	0.14	0.083	0.00027	6E-09	4.2E-24	2.1E-81
	X256	1	0.45	0.15	0.0038	2E-06	1.1E-27	3E-75
32	B64	1	0.034	0.0039	4.7E-06	9.1E-15	4.9E-37	1.2E-90
	B256	1	0.052	0.01	1.7E-05	2.8E-14	7.8E-37	2E-90
	R64	1	0.039	0.015	1.8E-05	3.5E-14	3.1E-36	6.7E-90
	R256	1	0.052	0.01	1.7E-05	2.8E-14	7.8E-37	2E-90
	C64	1	0.073	0.039	0.0028	9.7E-09	3.4E-30	1.1E-82
	C256	1	0.21	0.1	0.0039	5.1E-10	3.4E-31	6.6E-84
	X64	1	0.048	0.027	7.2E-05	1.4E-13	2.1E-35	5.5E-89
	X256	1	0.13	0.045	0.0019	3.3E-09	2.2E-32	5.9E-85
64	B64	1	0.016	0.0012	1.4E-06	1.7E-15	1.8E-37	3.3E-91
	B256	1	0.02	0.003	2.8E-06	6.6E-15	3.3E-37	5.2E-91
	R64	1	0.017	0.0046	4.4E-06	8.3E-15	8.3E-37	1.5E-90
	R256	1	0.028	0.0096	1.1E-05	3.3E-14	2.5E-36	6.1E-90
	C64	1	0.027	0.012	4.4E-05	3.8E-12	1.4E-35	3.1E-87
	C256	1	0.062	0.017	0.00016	1.4E-10	2.8E-34	5.4E-87
	X64	1	0.011	0.0081	1.4E-05	3.6E-14	3.7E-36	6.6E-90
	X256	1	0.034	0.014	2.2E-05	1.3E-13	1.4E-35	2.5E-86

Table F.4: For comparing the effect of collective adaptation on the GP the average maximal Collective Memory Clique Cover of Generation.

		1	2	4	8	16	32	64
1	B64	344.45	241.80	236.95	246.30	362.15	332.30	188.20
	B256	754.75	490.35	685.20	1436.30	2136.75	2091.65	1699.00
	R64	81.55	87.10	156.15	483.30	451.30	476.90	498.10
	R256	350.65	358.50	644.55	2193.90	2068.40	1325.60	781.00
	C64	88.20	95.40	169.85	454.25	521.60	521.25	521.00
	C256	345.85	352.15	650.75	1928.95	2131.20	2578.80	1601.45
	X64	68.50	84.10	162.30	454.55	458.90	517.50	582.25
	X256	305.15	373.20	768.80	847.10	1298.00	1457.25	1036.65
2	B64	270.40	238.55	171.20	190.50	218.30	212.20	199.95
	B256	523.70	315.75	505.30	892.05	970.75	1282.65	1286.55
	R64	77.55	93.20	213.80	436.85	491.45	574.20	552.75
	R256	308.85	374.95	953.30	2049.30	2005.10	839.50	1014.20
	C64	80.70	94.50	189.60	469.70	515.05	500.15	486.30
	C256	316.75	369.05	779.70	2064.60	2299.55	2415.85	2476.35
	X64	67.40	94.30	195.80	424.50	473.10	454.50	488.65
	X256	369.40	436.00	413.35	771.00	905.45	919.75	1011.95
4	B64	252.65	131.00	138.65	155.65	213.15	227.80	177.60
	B256	992.10	559.70	985.15	737.05	881.50	929.55	628.95
	R64	86.15	90.95	257.05	439.85	423.80	479.40	471.90
	R256	346.45	439.80	1438.95	1887.40	1833.10	665.65	630.40
	C64	85.40	85.55	238.05	440.45	418.15	452.15	507.35
	C256	334.50	449.25	1399.60	1924.70	1641.00	2097.05	2263.60
	X64	82.75	93.30	234.20	411.30	446.35	408.50	433.40
	X256	716.90	420.95	560.95	680.70	762.70	722.35	795.80
8	B64	244.30	128.95	189.35	234.90	189.60	180.75	128.10
	B256	984.40	789.30	693.15	1116.90	966.95	612.85	948.10
	R64	63.90	85.70	175.30	342.90	440.05	390.50	229.55
	R256	253.75	438.05	1295.90	1584.15	1027.15	1103.60	1057.10
	C64	93.15	89.30	166.00	353.80	232.20	388.85	206.40
	C256	254.80	440.35	1223.35	1871.20	1456.75	2197.75	1340.25
	X64	62.60	88.20	187.60	359.30	214.05	396.15	241.90
	X256	690.70	546.30	541.45	628.65	662.50	697.00	699.25
16	B64	308.75	116.50	152.80	114.60	171.75	148.85	191.55
	B256	1669.65	724.45	804.30	562.75	1005.75	844.55	1006.80
	R64	232.20	85.05	129.95	225.65	269.40	281.20	300.15
	R256	1395.30	493.90	592.30	602.10	733.30	581.90	2002.50
	C64	237.25	83.10	128.75	227.65	274.35	276.15	290.70
	C256	919.70	392.60	961.95	1687.85	1869.60	1665.20	2066.35
	X64	213.15	83.90	144.00	236.10	292.10	288.75	292.75
	X256	985.25	567.40	583.60	413.05	612.40	653.35	715.50
32	B64	526.45	128.15	151.40	141.15	100.25	179.95	140.35
	B256	2905.00	627.10	695.65	817.65	803.95	888.85	705.05
	R64	495.80	82.50	140.45	138.95	173.15	191.60	197.75
	R256	1974.40	637.65	695.55	562.40	805.95	624.00	705.75
	C64	496.85	86.20	107.70	160.00	182.40	176.80	222.00
	C256	1895.80	357.05	845.20	1235.65	1487.35	1682.50	1823.00
	X64	456.10	80.75	111.15	154.80	188.35	212.00	252.85
	X256	1641.60	454.80	548.55	612.15	624.65	622.30	785.20
64	B64	780.20	125.00	151.95	114.65	117.50	121.40	129.25
	B256	4797.80	825.55	705.85	705.95	961.80	840.15	1012.10
	R64	913.30	90.95	131.10	113.00	120.20	147.05	110.35
	R256	3073.05	344.70	523.80	840.85	988.75	889.65	1317.85
	C64	910.45	107.30	106.50	113.80	127.60	151.80	109.50
	C256	3212.60	328.10	459.65	656.30	1064.55	965.70	1287.50
	X64	847.75	78.90	104.30	125.90	153.90	167.80	117.25
	X256	2690.75	416.15	570.25	586.50	644.85	762.80	537.40

Table F.5: For comparing the effect of collective adaptation on the GP Sum of Time Differences per Generation.

APPENDIX G

Phenotypical Building Blocks

G.1 Introduction

The schema theorem and the building block hypothesis [Holland, 1975; Goldberg, 1989] provide intuition into the mechanics of the genetic algorithm (GA). They are sufficient in the same way the laws of Newtonian physics are for describing the physical world; on the surface, they prepare you for everyday life. We all can apply velocity, acceleration, equal and opposite force, and momentum to drive a car. But when we start to study the realm of the atom, we must resort to the laws of quantum physics. We can apply our Newtonian laws to get a "feel" for how things work, but we still need to adopt a counter-intuitive manner of thinking.

While problems with the schema theorem have been published [Altenberg, 1994; Mitchell *et al.*, 1992], it still provides a starting point for studying the theoretical workings of the GA. We do not really have that luxury when we examine the genetic programming (GP) paradigm [Koza, 1992]. Even though it is an off–shoot of GAs and it also borrows the concepts of selection, recombination, and mutation from the natural sciences, we are at a loss to provide a schema definition for the GP. O'Reilly and Oppacher argue that a GP Schema Theorem is not forthcoming [O'Reilly, 1995]. We do observe the formation of what we consider to be building blocks, e.g., highly fit subtrees of small defining length, but such

findings are empirical. Also, unlike GA building blocks, instances of a subtree may not be highly fit.

Royal Road functions are used by GA researchers to both investigate the formation of building blocks and test GAs against other paradigms [Mitchell *et al.*, 1992]. The GA researcher can utilize a Royal Road function to carefully craft a fitness landscape, which will allow for controlled experiments to test properties of the GA. Again, such an undertaking has not been done within the GP community. Some work has been done on presenting standard test-beds [Tackett and Carmi, 1994] and Punch *et al.* have even proposed a Royal Tree function [Punch *et al.*, 1996], which is presented more as a benchmark than as a tool for investigation of GP fitness landscapes.

My objective is to reconcile the conclusions drawn by O'Reilly and Oppacher against the empirical evidence for building blocks. I do so by showing how their GP schema fails to account for phenotypical building blocks. I then provide a brief survey of phenotypical building blocks in previous GP experiments. Finally I discuss additional characteristics I feel necessary for a GP schema to contain and illustrate these characteristics in a Royal Tree domain.

G.2 Building Blocks and GP

The "basic" theory of GP is borrowed from that of GA [Koza, 1992]. Due to the difficulties in detecting building blocks in GP chromosomes[1], research is ongoing

[1]See Section 3.2 for an introduction to building blocks.

into formally connecting the theory as to why GP works with that of why GAs work [O'Reilly, 1995; Rosca, 1997; Poli and Langdon, 1997b]. The canonical GP chromosome representation is a parse tree (S–expression). The fixed versus variable genotype representation has proven problematic in formulating a schema theorem for GP. O'Reilly and Oppacher carefully crafted a GP Schema Theorem, GPST, based on the expression of subtrees in the genotype of the chromosome. They just as carefully tore the resultant theory apart and questioned the existence of building blocks in GP systems [O'Reilly, 1995; O'Reilly and Oppacher, 1995b]. Their conclusions were:

> In this chapter we carefully formulated a Schema Theorem for GP using a schema definition that accounts for the variable length and the non–homologous nature of GP's representation. In a manner similar to early GA research, we used interpretations of our GP Schema Theorem to obtain a GP Building Block definition and to state a "classical" Building Block Hypothesis (BBH): that GP searches by hierarchically combining building blocks. We report that this approach is not convincing for several reasons: it is difficult to find support for the promotion and combination of building blocks solely by rigorous interpretation of a GP Schema Theorem; even if there were such support for a BBH, it is empirically questionable whether building blocks always exist because partial solutions of consistently above average fitness and resilience to disruption are not always assured; also, a BBH constitutes a narrow and imprecise account of GP search behavior. [O'Reilly, 1995] (pages 136–137)

Empirically however, the concept of introns have been discussed in GP literature for quite some time (see for example [Tackett, 1993; Angeline, 1994]). Introns are conjectured to guard against destructive crossover in GA chromosomes [Levenick, 1991]. Angeline conjectures the same role is played by introns

in GP chromosomes [Angeline, 1994], and Nordin *et al.* have shown this experimentally [Nordin, 1996]. Tackett however takes a slightly different stance: he believes that small subtrees which appear frequently in S–expressions and are especially expressed in the intron, are GP's building blocks. These subtrees are prevalent due to their contribution to the fitness of the chromosomes in which they appear [Tackett, 1993]. I adopt this frequent expression of subtrees in the chromosome as our working definition of building blocks in a GP system[2].

Altenberg applies Price's Theorem to account for the multiple appearance of a subtree within a chromosome [Altenberg, 1994]. He believes that the schema theorem can not account for the proliferation of copies of subtrees and introduces a "constructional fitness" to account for such proliferation. The key to understanding constructional fitness is in his redefinition of a building block; a building block is not necessarily highly fit; instead it is a block which has a higher probability of increasing fitness in a child chromosome. Thus a block is not a building block because of its contribution to the current chromosome, but rather because of its potential contribution to descendants of the chromosome. The distinction is subtle, but critical for understanding the characteristics of building blocks in GP chromosomes.

I was able to experimentally demonstrate Altenberg's constructional fitness by stripping all non-coding segments[3] out of the chromosome and creating a new

[2]The GPST of O'Reilly and Oppacher also accounts for the multiple appearance of building blocks in the chromosome.

[3]These are segments which do not contribute either positively or negatively to the evaluation of the chromosome.

chromosome which contained duplicates of the coding segment [Haynes, 1996]. By forcing the multiple appearance of the coding segment, I am able to effect multiple appearances of the building blocks. The duplicates were non–coding in that they had no direct influence on the fitness of the repaired chromosome. However, they had a significant influence on the fitness of subsequent generations. Empirically, I was able to show that building blocks could exist by assuring the existence of partial solutions of consistently above average fitness and resilience to disruption.

I believe that there is a contradiction between O'Reilly and Oppacher's theoretical model of GPST[4] and the experimental findings on the multiple appearances of subtrees in the chromosome (see for example [Tackett, 1993; Angeline, 1994]). I do not dispute O'Reilly and Oppacher's findings, but rather the assumptions that went into their model. Their formulation of building blocks is based on the chromosome's genotype. As an example, consider the GP-schema $H = \{((+\ 5\ 6), 2)\}$[5]; it has three instances in Figure G.1: (AC, AD, CD). The subtree at B does not help form three more instances since the order of the arguments do not match. If we consider only the chromosome's genotype this general schema definition is satisfactory.

However, if we consider the chromosome's phenotype, this schema definition fails. In the context of the parent nodes, e.g., the nodes labeled RCL and RCR, the contributions of the subtrees at the nodes labeled A, C, and D can be vastly different. For example, let the function at node RCR be addition. Are the subtrees

[4]Which is actually grounded in experimental research [O'Reilly, 1995].

[5]This reads as schema for which the subtree (+ 5 6) is expressed twice. The reader is referred to [O'Reilly, 1995] for the notation.

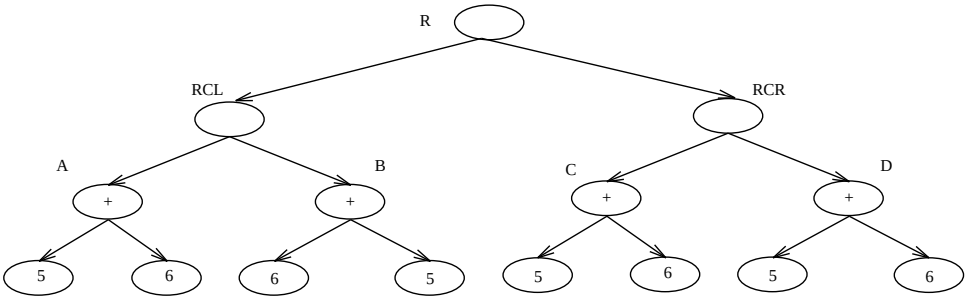

Figure G.1: The three instances of the GP-schema $H = \{((+\ 5\ 6), 2)\}$ in an S–expression.

at C and D necessary to derive the constant 11? Or are they backups of the subtree at node A, which is necessary in the context of node RCL? If the function at node RCL is any of the set $F_1 = \{$**addition**, **division**, **multiplication**, **subtraction**$\}$, the number returned by RCL is not dependent on the ordering in node B[6].

Even with the simple function set F_1, I have illustrated that a schema definition based solely on the chromosome's genotype is inadequate. Why then do O'Reilly and Oppacher focus only on the genotype and not the phenotype of the chromosome? They are building their GPST from the schema definitions employed in GAs, and in GAs there is a close relationship between the genotype and phenotype structure of a chromosome. Thus the schemata of GAs are usually represented at the genotype level and building blocks are relatively easy to detect. I argue that with the GP, schemata are at the phenotype or semantical level and the building blocks are difficult to represent, detect, and capture.

Poli and Langdon have recently proposed a different schema theorem based on their concept of one–point crossover [Poli and Langdon, 1997b; Poli and Langdon,

[6]Andre and Teller provide a more detailed discussion of the potential interactions of nodes in the chromosome [Andre and Teller, 1996].

1997a]. Their schema is also not commutative, i.e., it does not allow for the different ways to embed a phenotypical subtree in the genotype[7]. For example, (+ *5 6*) and (+ *6 5*) are not equivalent. They define a schema as a tree which is built from the union of both the function and terminal sets with the wild card "=", which represents a single element of the chromosome. While their schema theorem, and its representational power, is more complete than that of O'Reilly and Oppacher, it still fails to completely capture phenotypical interactions of building blocks.

Consider the problem of evolving a solution to the **XOR** binary function with the function set $F_2 = \{$**AND**, **OR**, **NOR**, **NAND**$\}$ and the terminal set $T_2 = \{A, B\}$[8]. Furthermore, without a loss of generality, let us restrict the maximum depth of trees to be 3. A possible solution is (**OR** (**AND** (**NOR** A A) B) (**AND** (**NOR** B B) A)). Notice that the schema (**NOR** = =) is not a building block given the current representation. There are four possible instantiations: (**NOR** A A), (**NOR** A B), (**NOR** B A), and (**NOR** B B). Two of the four are building blocks in the domain, but the other two are unfit.

We could extend the schema of Poli and Langdon by allowing for generic wild cards which become instantiated when expressed in a schema. For example, we could have the wild card set $W = \{=, \alpha, \beta\}$. Now the schema (**NOR** α α) is a building block since it is a subtree in which its first argument is the same as

[7]The commutative property is dependent on the properties of each function in the function set F.

[8]Poli and Langdon provide a detailed theoretical and experimental analysis of this domain and alphabet for trees of a maximum depth of 2 and also an experimental analysis of trees of depth 3 [Poli and Langdon, 1997a].

its second argument. Another observation is that (**NOR** α α) is a genotypical representation of the phenotypical building block (**NOT** =). With our depth restrictions, there are two possible genotypical representations of this phenotypical building block: (**NOR** α α) and (**NAND** α α).

G.3 Royal Roads

Royal Road functions represent fitness landscapes which facilitate the testing and understanding of how the GA works [Mitchell *et al.*, 1992]. By explicitly designing the fitness function to encourage building blocks, the performance of the GA can be tested. Consider the 1s problem, i.e., with a fixed binary string; the global optimum is having a 1 in each bit. A simple fitness function for this domain is to count the number of 1s expressed in the string. However, we would further like the 1s to be adjacent. Both s_1 = `*1**1***` and s_2 = `*11*****` are schemata of equal order which describe schema with at least two 1s present. However, s_2 is better because it is less susceptible to the destructive effects of crossover. In this domain, desirable building blocks maximize the number of adjacent 1s.

Mitchell *et al.* present the function R_1,

$$R_1(x) = \sum_{i=1}^{8} \delta_i(x) o(s_i), \text{where } \delta_{\mathrm{i}}(\mathrm{x}) = \begin{cases} 1 & \text{if } x \in s_i \\ 0 & \text{otherwise,} \end{cases}$$

where x is a bit string, and $o(s_i)$ is the order of the schema s_i. In Figure G.2, the goal is detect the 8 schemata and use them to build up the solution. By

selecting these building blocks, they are providing a detailed algorithm for solving the 1s problem. Their choice of building blocks dictates the learning curve for this problem. In R_2, Figure G.3, there are also the four schemata which correspond to 16 consecutive bits starting on 16 bit boundaries and the learning is drastically different [Mitchell *et al.*, 1992].

```
s1= 11111111************************************************************
s2= ********11111111****************************************************
s3= ****************11111111********************************************
s4= ************************11111111************************************
s5= ************************************11111111************************
s6= ****************************************11111111****************
s7= ************************************************11111111********
s8= ************************************************************11111111
```

Figure G.2: A set of schemata describing an instance of the Royal Road function R_1.

With the function R_2, replicated in Figure G.3, the goal is to detect and use the lower order schemata to detect the higher order schemata. For example, if crossover causes s_1 and s_2 to be joined, then we also have s_9. The desired effect is to force the application of the Schemata Theorem. The actual results indicated that the building blocks were not being exploited [Mitchell *et al.*, 1992].

A tempting assumption to make is that the 1s problem is purely genotypical. In Mitchell *et al.*'s description of R_1, the problem's domain is explicitly omitted. The intent is to simply describe a fitness landscape. However, I argue that the building blocks, s_i, are purely phenotypical in nature.

Let us consider an example for which we know more than the genotype: the

```
s1= 11111111*********************************************************
s2= ********11111111*****************************************************
s3= ****************11111111*******************************************
s4= ************************11111111**********************************
s5= *********************************11111111************************
s6= ******************************************11111111****************
s7= ************************************************11111111********
s8= ************************************************************11111111
s9= 1111111111111111*************************************************
s10= ****************1111111111111111*******************************
s11= *********************************1111111111111111****************
s12= ***************************************************1111111111111111
s13= 11111111111111111111111111111111*******************************
s14= *********************************11111111111111111111111111111111
s15= 11111111111111111111111111111111111111111111111111111111111111111
```

Figure G.3: A set of schemata describing an instance of the Royal Road function R_2.

XOR domain with the alphabet given by F_2 and T_2. In the last section, we did not discuss the fitness evaluation of a chromosome in this domain. Intuitively, we have four test cases, given by the truth table formed by the input pair AB and the output $A \oplus B$. The fitness evaluation could simply be the number of test cases that a chromosome correctly evaluated. Possible building blocks could be chromosomes which correctly identify the input pairs $\{A =\}$ and $\{= B\}$.

We can also construct a Royal Road function with the phenotypical building blocks: (1) (**NOT** =) and (2) (**AND** = (**NOT** =)). Since we are in the phenotypical space, and all functions are commutative, we allow (**AND** = (**NOT** =)) to also describe (**AND** (**NOT** =) =). These two phenotypical building blocks have the following genotypical building blocks:

1. (**NAND** α α)

2. (**NOR** α α)

3. (**AND** (**NAND** α α) =)

4. (**AND** = (**NAND** α α))

5. (**AND** (**NOR** α α) =)

6. (**AND** = (**NOR** α α))

We can enumerate all schemata containing these building blocks and assign relevant fitness points to each schema instantiation.

G.4 Phenotypical Building Blocks

My claim is that schemata actually occur in both the chromosome's genotype and phenotype. With the GA, the fixed length nature of the chromosome ensures the close relationship of genotype and phenotype. Building blocks in the phenotype are usually found in systems which have multi-level fitness functions; they are segments of the chromosome which solve the subtasks delimited by the multi-level fitness functions.

Multi–level fitness functions are typically considered in GP systems which emulate single or multi-agent systems [Andre, 1995; Haynes *et al.*, 1995b; Koza, 1992; Spector, 1996], although there are several non-agent systems that make use of multi–level fitness functions [Haynes, 1996; Punch *et al.*, 1996; Soule *et al.*, 1996]. Problem domains may not be explicitly designed to reward for different actions [Andre, 1995; Koza, 1992], but still encourage the development of building blocks to solve subtasks in the evaluation. For example, the Pac–Man game

rewards for an agent eating a pill and then the monsters while the pill is in effect [Koza, 1992]. In his encoding of the problem, Koza did not devise the problem so that this sub-task would be solved. It arose naturally from the way the video arcade game was designed. Likewise, this is the case with Spector's research [Spector, 1996] into the Wumpus World presented originally by Russell and Norvig [Russell and Norvig, 1995].

In other problem domains, the designers did explicitly define multi-level fitness functions to encourage the solving of sub-tasks, which when summed together would solve the global task [Haynes *et al.*, 1995b; Haynes, 1996; Punch *et al.*, 1996; Soule *et al.*, 1996]. For example, the predator/prey implementation by Haynes *et al.* first encourages predators to move closer to the prey, then to each take up and stay at a capture position, and finally to all stay at the capture position [Haynes *et al.*, 1995b]. The simple evaluation of always moving closer would also lead to capture. With the multi–level function they use, they have created a Royal Road function. The evaluation of a subtree is context dependent; i.e., its position in the chromosome readily changes its evaluation. However, no study has been done on the shape of the fitness landscape.

Three other functions which exhibit Royal Road characteristics have been proposed [Haynes, 1996; Punch *et al.*, 1996; Soule *et al.*, 1996]. The Royal Tree function proposed by Punch *et al.* is strictly based on the chromosome's genotype. While they utilize partial credit to implement the Royal Tree functionality, their function and terminal set are in effect the shape of the chromosome. As such,

their building blocks do not depend on context.

Both Haynes and Soule *et al.* consider variants of detecting cliques in a graph: Haynes detects all cliques in a graph and Soule *et al.* look for the maximum clique. Haynes utilized strong typing [Montana, 1995] and type inheritance [Haynes *et al.*, 1996b] to generate a collection of candidate cliques in a graph. He discussed how the valid candidate cliques are actually building blocks and shows how, by duplicating the candidate cliques in the chromosome, learning can be improved. Soule *et al.* do not utilize strong typing, instead they use an union operation to join nodes together into one candidate maximum clique. They also provide an overview of why the arguments provided by O'Reilly and Oppacher fail for this domain and show the representational problems that do not make clique detection a good candidate for a GA Royal Road function.

G.5 A GP Royal Road Function

In this section I illustrate how the definition of Royal Road functions must be changed to accommodate GP chromosomes. Soule *et al.* have shown that strong typing is not necessary; i.e., a plain GP system can not only detect cliques but also express building blocks inside the chromosome. However the identification of building blocks inside the chromosome is facilitated in our system by the combination of strong typing and the problem formulation of finding all the cliques.

Each candidate clique is a phenotypical building block. For example, the subtrees (**IntCon** *4 5*) and (**IntCon** *5 4*) are both genotypical representations of

the candidate clique $C_1 = \{4, 5\}$. This candidate clique can be utilized to form both of the maximal cliques $C_a = \{1, 4, 5\}$ and $C_a = \{4, 5, 8\}$. While Mitchell *et al.* needed to explicitly design their fitness function to reward for the expressing of the schemata s_i, with the clique detector fitness function the reward for the expression of the building block is implicit. To change the fitness landscape, I would either have to change the edges between the nodes or add additional nodes to the graph.

I employ Altenberg's definition of a building block [Altenberg, 1994] for the expression of candidate cliques in the chromosomes. For example, consider the third candidate clique in Figure 3.3. Since it does not directly contribute to the fitness of the chromosome, it does not fit the traditional definition of a building block. However, this subtree can potentially increase the fitness of a child if crossover either cuts it away from the subtree denoted by candidate clique #2 or splices it together with either of nodes 3 or 8.

While Mitchell *et al.* believe that one does not have to have every partial step included in the fitness function, I have observed that rewarding for increasing the size of candidate clique is effective in finding the optimal solution. The fitness landscape can be crafted by selecting the graph to be considered. For example, a graph with eight vertices, which are all connected, could correspond directly to the R_1 function. The fitness function could only reward the chromosome if either one of the eight candidate cliques of size seven is discovered.

The combination of the low cardinality candidate cliques into higher cardinal-

ity candidate cliques meets the characteristics of a Royal Road function as put forth by Mitchell *et al.*:

1) All of the desired building blocks are known in advance.

2) The landscape can be varied systematically.

3) The global optimum, and all local optimum, can be enumerated.

With the example graph shown in Figure G.4, I can list all of the building blocks:

$$\begin{aligned} C = \{ \quad & \{0,1\},\{0,2\},\{0,3\},\{1,2\},\{1,3\},\{2,3\}, \\ & \{4,5\},\{4,6\},\{4,7\},\{5,6\},\{5,7\},\{6,7\}, \\ & \{0,1,2\},\{0,1,3\},\{0,2,3\},\{1,2,3\}, \\ & \{4,5,6\},\{4,5,7\},\{4,6,7\},\{5,6,7\}, \\ & \{0,1,2,3\},\{4,5,6,7\}\}. \end{aligned}$$

Since I know all of the candidate cliques, I can calculate the fitness for all interesting combinations of building blocks. I vary the fitness landscape by adding or deleting edges. So, given that I construct a graph such that I know all of the possible candidate cliques, i.e., building blocks, I am forming Royal Road functions. Furthermore, I can vary the size and number of cliques such that a graph is not just a Royal Road function for GAs, but also for the other search heuristics.

As an example, consider the graph in Figure G.5, which has 3 cliques present: $C = \{\{0,1,2,3,4,5,6,7\},\{8,9,10,11\},\{12,13,14,15\}\}$. Ignoring the encoding, we see that any cycle in the graph represents a possible group. If the group

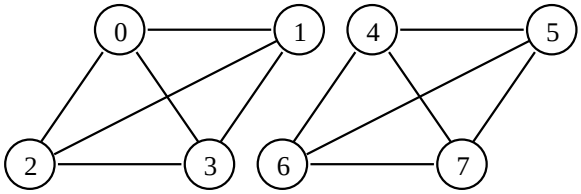

Figure G.4: Example graph, consisting of 2 fully connected cliques of cardinality 4.

contains duplicate vertex labels or if a dotted edge is traversed, then the group is not a candidate clique. Upon examination we see that each clique is a "hill", with the density of solid edges indicating the "steepness" of the hill. I expect that hill climbers might get stuck on either of the local maxima.

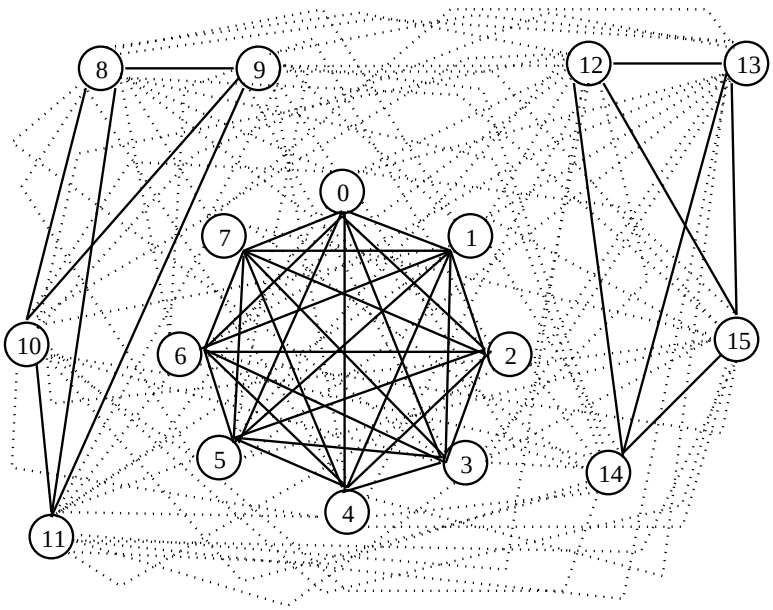

Figure G.5: Example graph, consisting of 3 fully connected cliques of cardinalities 4, 4, and 8. Actual connections are solid lines and possible connections are dotted lines.

G.6 Conclusions

The work of O'Reilly and Oppacher in developing a GP Schema Theorem [O'Reilly, 1995; O'Reilly and Oppacher, 1995b] is instrumental in proving why a schema theorem rooted in the genotype is not sufficient for GP. I do not however support their claim that such a theorem is not forthcoming and several theorems have been recently proposed [Rosca, 1997; Poli and Langdon, 1997a]. While I do not offer such a theory, I do reconcile the empirical observations of GP researchers, i.e., that GP does build hierarchical solutions by recombining small subtrees, with the findings of O'Reilly and Oppacher.

The building blocks expressed in GP systems are both in the genotype and phenotype of the chromosome. By designing multi–level fitness functions, GP researchers have been implicitly designing Royal Road functionality into their domains. Indeed several have reported the discovery of the multiple fitness criteria during the evolutionary process [Koza, 1992; Haynes *et al.*, 1995b].