Augmenting Collective Adaptation with Simple Process Agents

Thomas Haynes
Department of Mathematical & Computer Sciences
600 South College Ave.
The University of Tulsa
Tulsa, OK 74104-3189
e-mail: haynes@euler.mcs.utulsa.edu

Abstract

We have integrated the distributed search of genetic programming based systems with collective memory to form a collective adaptation search method. Such a system significantly improves search as problem complexity is increased. However, there is still considerable scope for improvement. In collective adaptation, search agents gather knowledge of their environment and deposit it in a central information repository. Process agents are then able to manipulate that focused knowledge, exploiting the exploration of the search agents. We examine the utility of increasing the capabilities of the centralized process agents.

Introduction

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 of the individual agent. Such bounds are caused by the combinatorial explosion found in either search or optimization of NP complete problems (Garey & Johnson 1979). Collective memory is defined as the combined knowledge gained by the interaction of the agents with both themselves and their environment. We combine the raw power of collective action with the expressiveness of collective memory to enhance a distributed search process. The integration of action and memory leads to a distributed society of search agents which interact via collective memory; allowing for either agent communication or for a centralized search of the gathered knowledge. We consider simple computational search agents, which are chromosomes in a genetic programming (GP) (Koza 1992) population.

Genetic algorithms (GA) (Holland 1975) are a class of distributed search algorithms inspired by biological evolutionary adaptation. GP is an off-shoot of GA's, and is typically used in the automatic induction of programs. Both GA and GP represent search strategies in a population of chromosomes. Each chromosome in the population can be searching different parts of the search space or fitness landscape. Each chromosome can be considered to be a behavioral strategy to control an agent (Haynes *et al.* 1995) and are considered to be autonomous in the sense that they do not typically interact to find a solution. They are also 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.

We have found that collective adaptation, which is the addition of collective memory to a GP-based learning system, significantly improves the search process as problem complexity is increased (Haynes 1997). We believe that this improvement is a direct result of the change of focus from strict competition to cooperation. However, there is still considerable room for improvement. In this paper, we investigate increasing the processing power of process agents, which are computational agents external to the GP system, to improve the collective adaptation.

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.

We wish to minimize the complexity of the agents in this society. We believe that the knowledge gained from the interactions of the simple agents will be greater than the sum of the knowledge of those same individual agents. To that end, we wish to limit communication, as it can place too much of a burden on the agents. Further, we limit the amount of state that an agent can posses. Our basic model is that of the social insect; near mindless individuals which interact to form an intelligent group. We need to strike a balance between parallel and distributed search. If our agents are simple enough, then they will be cheap in terms of execution time. If we are able to detect the redundant results, we can allow redundant search. Finally, to reduce communication, we restrict it to either before or after an agent is actively searching.

Our goal is to utilize simple computational agents to retrieve knowledge from the problem space, store that knowledge in an information center, and allow other computational agents to manipulate that knowledge in the information center. To that end, we define:

Information center as 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 as those agents which retrieve knowledge from the problem space. They may not communicate with other agents outside of the information center. They may add knowledge to the information center, but they may not delete from it.

Process agents as those agents which manipulate the knowledge stored in the information center. They may delete knowledge from the repository.

The computational agent society is depicted in Figure 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.

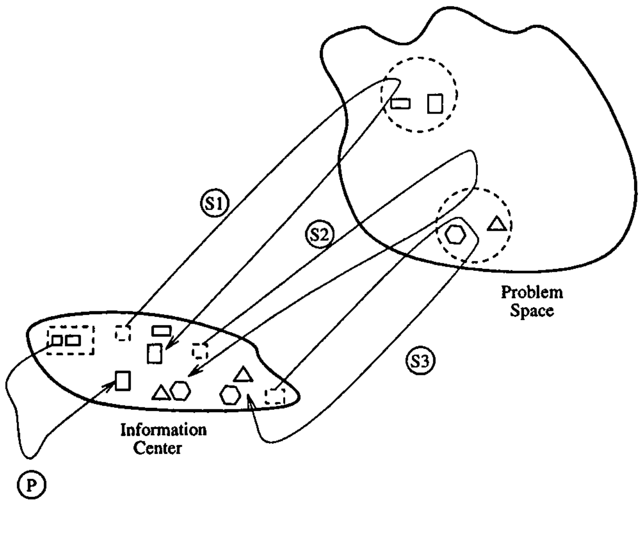

Figure 1: Computational agent society employing an information center. Process agents are labeled with a **P** and search agents with a **S**.

Process agents cannot manipulate the search space; they must direct the search agents in order to sense and manipulate the search space. The search agents can neither manipulate the information center nor direct other search agents. The interactions of both the process and search agents with the information center form two orthogonal dimensions of access. Both dimensions can take on one of two discrete values: passive and active. Passive agents do not retrieve knowledge from the information center, while active agents can retrieve knowledge. We reference a tuple in these dimensions by *Interactivity-Processing*, where Interactivity denotes the state of the search agents and Processing denotes the state of the process agents.

The four models of access are:

Active-Passive The information center is interactively accessed by the independent search agents. They gather knowledge and deposit it into the information center. Before a new search is started, a search agent can retrieve and utilize knowledge from the information center to guide and shape the search.

Active-Active The information center is interactively accessed by the independent search agents. By manipulating the repository, the process agents can guide the search agents.

Passive-Passive This form of access is actually parallel search as there is no communication.

Passive-Active The search agents do not interact with the information center. They still gather and deposit knowledge into the repository, but they cannot retrieve knowledge from it. The information center is a focusing of the problem space from which process agents can manipulate the knowledge.

We examine the coordination of knowledge of loosely-coupled, heterogeneous, and initially simple agents. The agents can adapt during the search process, eventually becoming quite complex.

Genetic Programming

Genetic programming is a machine learning technique used in the automatic induction of computer programs (Koza 1992). A GP system is primarily comprised of three main parts:

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

In implementing a system for a new problem, 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 offspring of the current generation: parents are randomly selected in proportion to their fitness evaluation. 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 states that all functions must be able to handle all inputs, i.e., division can handle a 0 denominator. Sufficiency requires that the domain be solvable with the given function and terminal sets. One ramification of closure is that all functions, function arguments, and terminals have just one typality. Hence, closure means any element can be a child node in a parse tree for any other element without having conflicting data types.

Montana claims that closure is a serious limitation to genetic programming. He introduces a variant of GP in strongly typed genetic programming (STGP), in which variables, constants, arguments, and returned values can be of any type (Montana 1995). 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. It has been shown that STGP can significantly reduce the search space (Haynes *et al.* 1995; Montana 1995). The STGP variant mainly restricts the construction and reproduction of chromosomes; the basic algorithm is GP.

Clique Detection

Clique detection has been used as a benchmark for improving learning in GP systems (Haynes 1996; Haynes, Schoenefeld, & Wainwright 1996). 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. A clique is denoted by the set of vertices in the complete subgraph and the goal is to find all cliques of G. Since the subgraph of G induced by any subset of the vertices of a complete subgraph of G is also complete, it is sufficient to find all maximal complete subgraphs of G. A maximal complete subgraph of G is a maximal clique. Each chromosome in a STGP pool will represent sets of candidate maximal cliques. The function and terminal sets are $F = \{\textbf{ExtCon}, \textbf{IntCon}\}$ and $T = \{\mathit{1,\ldots,\#vertices}\}$. **ExtCon** "separates" two candidate maximal cliques, while **IntCon** "joins" two candidate cliques to create a larger candidate.

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 maximal cliques, each represented by an ordered list of vertex labels.
- Throw away any duplicate candidate maximal cliques and any candidate maximal cliques that are subsumed by other candidate maximal cliques.
- Throw away any candidate maximal cliques that are not complete subgraphs.

The fitness formula is

$$F = \alpha c + \sum_{i=1}^{c} \beta^{n_i},$$

where $c = \#$ of valid candidate maximal cliques and $n_i = \#$ vertices in clique C_i. Both α and β are configurable by the user. β has to be large enough so that a large clique contributes more to the fitness of one chromosome than a collection of proper subcliques contributes to the fitness of a different chromosome.

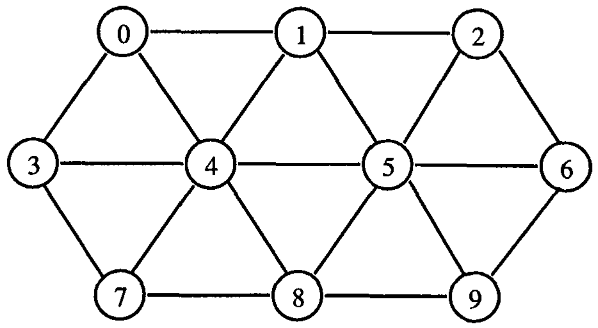

Figure 2: Example 10 node graph.

Figure 2 is a ten node graph we have used in our previous research to test the clique detection system. There are exactly 10 cliques: $C = \{\{0,3,4\}, \{0,1,4\}, \{1,4,5\}, \{1,2,5\}, \{2,5,6\}, \{3,4,7\}, \{4,7,8\}, \{4,5,8\}, \{5,8,9\}, \{5,6,9\}\}$. An example chromosome for the 10 node graph is presented in Figure 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 vertices, i.e. vertex 7 is repeated; #3 is subsumed by #2; and, #1 is not completely connected.

This example graph exhibits nice regularities which allows for the efficient comparison of results across different test runs. We have utilized these regularities to identify and enumerate the building blocks, i.e., the connected components (Haynes 1996). We repaired chromosomes by stripping out all invalid candidate cliques. We investigated various rates of return of repaired chromosomes into the population. We found that by duplicating the coding segments we could significantly improve the search process.

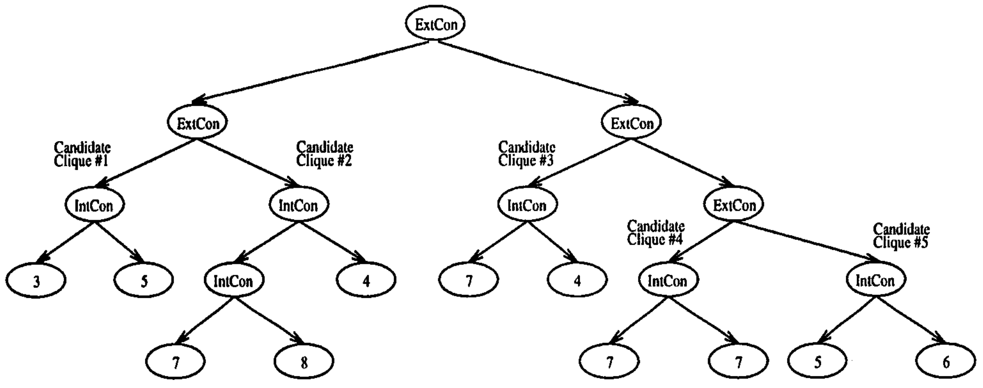

Figure 3: S–expression for 10 node graph.

If a chromosome contained no valid candidate cliques, we tried a repair strategy of injecting the set of all valid cliques found to date. We found that such a repair strategy led to premature convergence in a non–optimal section of the search space. We find if we instead adopt a Passive-Active collective adaptation technique in this domain, the search process is greatly facilitated. With the Passive-Active collective adaptation we do not repair chromosomes which have no valid candidate cliques. Instead the search agents gather candidate cliques into the information center and the process agent removes duplicates and candidates subsumed by larger candidates.

The addition of Passive-Active collective adaptation to the search technique significantly improves the efficiency of the search process (Haynes 1997). We want to leverage that improvement to allow clique detection in more realistic graphs. The ten node graph we use to illustrate the clique detection is contrived and thus facilitates 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 & Trick 1993) 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.

We now examine 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. We present the results, in Figure 4, 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 PA, i.e., add Passive-Active collective adaptation to piece together the set of all cliques.

The addition of Passive-Active collective adaptation

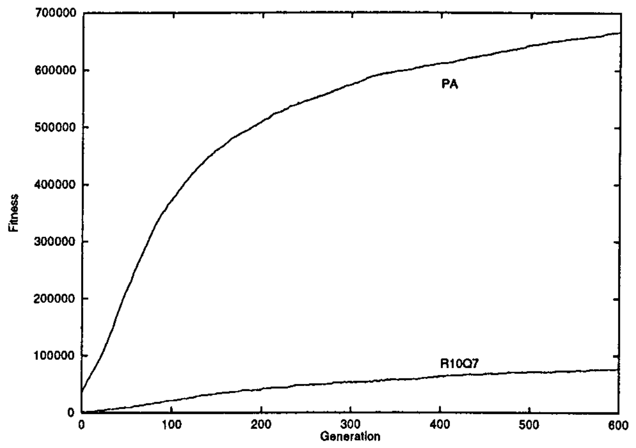

Figure 4: Passive-Active collective memory search applied to the hamming6-4 graph. In particular, comparison of best fitness per generation for duplication of coding segments repair of chromosomes with a 10% return rate and 7 duplicates (R10Q7), and Passive-Active collective adaptation (PA), which utilizes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

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, we could allow the search to continue for more generations. We could also increase the population size.

Experiments

Both methods fail to address our implicit desire to effectively search the space in both minimal time and memory. A possible extension is bestow further computational effort to the process agent(s) (The process agent in this domain just collates the knowledge, removing duplicates.). The information center is a rich storehouse of knowledge and the process agents should be able to exploit the exploration of the search agents.

Imagine the information center 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 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 detector, we can 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 and the set of all vertices, S_i:
   (a) If a candidate clique can not be expanded by the addition of one vertex, then add it 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. $S_i = S_{i+1}$.

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.

Our first experiment adds an additional process agent to our computational agent society. Each generation, after both the search agents 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 information center (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 can not be used for exploration, i.e., the four candidate cliques, of size 3, of a candidate clique of size 4, C_{4i}, can not be used to find potential candidate cliques of size 4, $C_{4j} (j \neq i)$, which have three vertices in common with C_{4i}; and 3) 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. 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). 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.

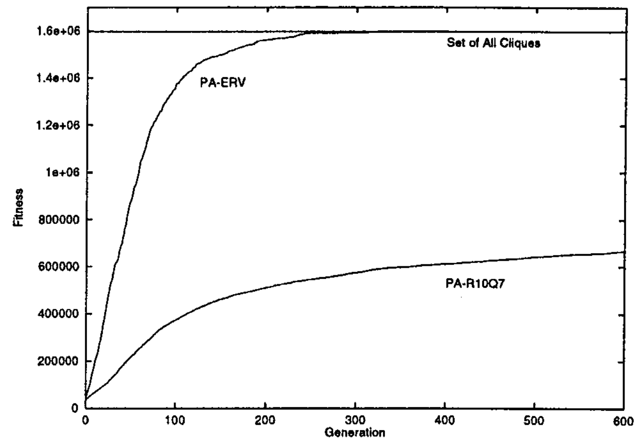

Figure 5: Comparison of fitness per generation for Passive-Active collective adaptation with two levels of activity on the part of the process agents: 1) a simple collating agent (PA-R10Q7), and 2) an agent which, after collation, extends by one randomly selected vertex each generation (PA-ERV). The 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 innocuous seeming extension so effective? The search space is narrowed into the information center space. The process agent is 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, we 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 seems feasible, but we find that it actually performs worse than Passive–Active collective adaptation, seeFigure 6. Why? The process agent which merges the candidate cliques is forming larger candidate cliques than the agent employing 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. By maximizing locality, this algorithm ensures that multiple mergers can not 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.

We can test our hypothesis by considering a third algorithm, Merge Random Candidate Clique, MR. We 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 information center. As can be seen from Figure 6, 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 it is taking too big a step during the merge process.

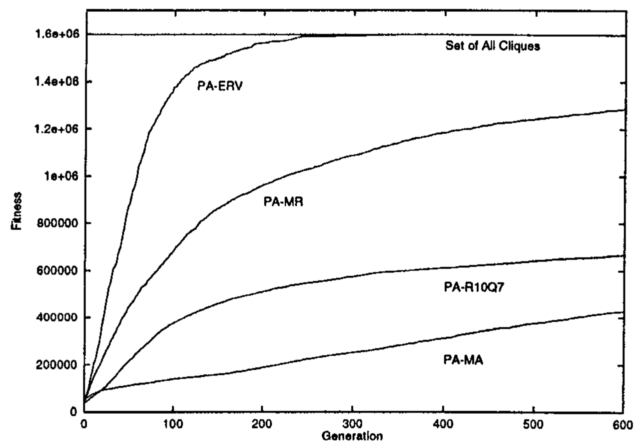

Figure 6: Comparison of fitness per generation for Passive-Active collective adaptation with different levels of activity on the part of the process agents: 1) a simple collating agent (PA-R10Q7), 2) agents, which after collation, sort then merges adjacent candidate cliques if they are connected (PA-MA), 3) agents, which after collation, sort and merge one randomly selected candidate clique with all other compatible candidate cliques in the information center, 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).

Conclusions

Collective adaptation is applicable in integrating results from loosely-coupled agents. Simple search agents are effective in gathering knowledge. 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.

References

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

Haynes, T.; Wainwright, R.; Sen, S.; and Schoenefeld, D. 1995. Strongly typed genetic programming in evolving cooperation strategies. In Eshelman, L., ed., *Proceedings of the Sixth International Conference on Genetic Algorithms*, 271–278. San Francisco, CA: Morgan Kaufmann Publishers, Inc.

Haynes, T.; Schoenefeld, D.; and Wainwright, R. 1996. Type inheritance in strongly typed genetic programming. In Kinnear, Jr., K. E., and Angeline, P. J., eds., *Advances in Genetic Programming 2.* MIT Press. chapter 18.

Haynes, T. 1996. Duplication of coding segments in genetic programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence.*

Haynes, T. 1997. Collective memory search. In *Proceedings of the 1997 ACM Symposium on Applied Computing.* ACM Press.

Holland, J. H. 1975. *Adpatation in Natural and Artificial Systems.* Ann Arbor, MI: University of Michigan Press.

Johnson, D. S., and Trick, M. A. 1993. Cliques, coloring, and satisfiability: The second DIMACS challange. (to appear).

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

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