Collective Memory Search

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

KEYWORDS

Genetic Programming
Collective Adaptation
Distributed Search

Abstract

Collective action has been examined to expedite search in optimization problems [2]. Collective memory has been applied to learning in multiagent systems [4]. We integrate the simplicity of collective action with the pattern detection of collective memory to significantly improve both the gathering and processing of knowledge. We augment distributed search in genetic programming based systems with collective memory. Four models of collective memory search are defined based on the interaction of the search agents and the process agents which manipulate the collective memory. We implement one of the collective memory search models and show how it facilitates "scaling up" a problem domain. A Passive-Active model, in which the gathered results are collated, is employed by the process agents to piece together the solution from the parts collected by the search agents.

1 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 the class of NP complete problems [3]. 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.

Our collective memory search is more than just the integration of the action and memory aspects of earlier research. It differs from the collective action research of Dorigo *et al.*, [2] in that agents need not communicate and a central agent can process the gathered knowledge. It differs from the collective memory research of Garland and Alterman [4] in that agents need not learn via the collective memory, agents need neither interact nor communicate, and the memory is centralized. 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) [10] population.

Genetic algorithms (GA) [8] 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 [7] and are considered to be autonomous in the sense that they do not typically interact to find a solution. They also implicitly cooperative since the more fit chromosomes of generation G_t are more likely to contribute genetic material to the chromosomes in generation G_{t+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 rulesets must be maintained. In the "Michigan approach" each chromosome is a rule and the population as a whole is the ruleset. In the "Pitt approach" each chromosome

is a ruleset, being comprised of multiple rules [1].

We investigate the addition of collective memory to GP-based learning systems. We allow the explicit reuse of knowledge from one generation to the next. We show how the knowledge retrieved by the agents can be integrated to form a whole greater than the parts.

The rest of the paper is organized as follows: Section 2 defines the four models of collective memory. Section 3 is an overview of the basics of genetic-based computational search systems. Section 4 reveals how collective memory can be utilized to improve search in a clique detection domain. Section 5 presents some experiments in implementing collective memory to scale up the the clique detector. Section 6 discusses some of the ramifications and drawbacks of collective memory within genetic programming. Section 7 concludes our discourse on the applicability of collective memory in distributed search. Section 8 examines further avenues of research in utilizing collective memory in distributed search.

2 COLLECTIVE MEMORY

Garland and Alterman present a distributed collective memory in their research; agents manipulate their own slice of the collective memory [4]. We present a centralized collective memory, which is a knowledge repository, not local to the agents. As agents gather knowledge, they deposit it into the collective memory. Agents can have read, write, and delete privileges. The write action cannot overwrite.

We define:

Search agents as those agents which retrieve knowledge from the search space. They have write privileges, do not have delete privileges, and may or may not have read privileges.

Collective memory as an area where the raw information retrieved by the search agents can be stored.

Process agents as those agents which collate and process the collective memory. They can have combinations of the different privileges. For example, collation is a composition of read 1, read 2, delete 1, delete 2, integrate 1 and 2 into A, and write A.

Process agents cannot directly 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 collective memory nor direct other search agents. Furthermore, a search agent cannot direct itself; once it has been assigned a task, it continues executing it until redirected by a process agent.

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 agents do not retrieve knowledge from the collective memory, 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 collective memory are:

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

Active-Active The collective memory is interactively accessed by the independent search agents. By manipulating the memory, the process agents can guide the search agents.

Passive-Passive This form of collective memory is actually no collective memory at all.

Passive-Active The collective memory does not interact with the search agents. They still gather and deposit knowledge into the collective memory, but they cannot retrieve knowledge from it. The collective memory is a repository from which process agents can manipulate the knowledge.

We explore the addition of both Active-Passive and Passive-Active collective memory to a society of search agents represented by GP chromosomes. 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.

3 GENETIC PROGRAMMING

Genetic programming is a machine learning technique used in the automatic induction of computer programs [10]. 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 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 [11]. 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 [7, 11]. The STGP variant mainly restricts the construction and reproduction of chromosomes; the basic algorithm is GP.

4 CLIQUE DETECTION

We have used clique detection as a benchmark for improving learning in GP systems [5, 6]. A collection of cliques in a graph can be represented as a list of a list of nodes 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. Our 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 $F = \{\mathbf{ExtCon}, \mathbf{IntCon}\}$ and $T = \{1, \ldots, \#nodes\}$. **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. We also ensure that we do 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 = # nodes 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.

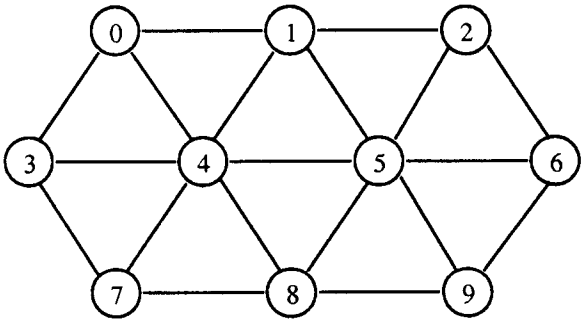

Figure 1: Example 10 node graph.

Figure 1 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 2. 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.

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 [5]. 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 (A coding segment is the material in the chromosome which contributes, either positively or negatively, to the evaluation of the chromosome. In this domain, the coding segments correspond to that material which was not stripped out.) we could significantly improve the search process.

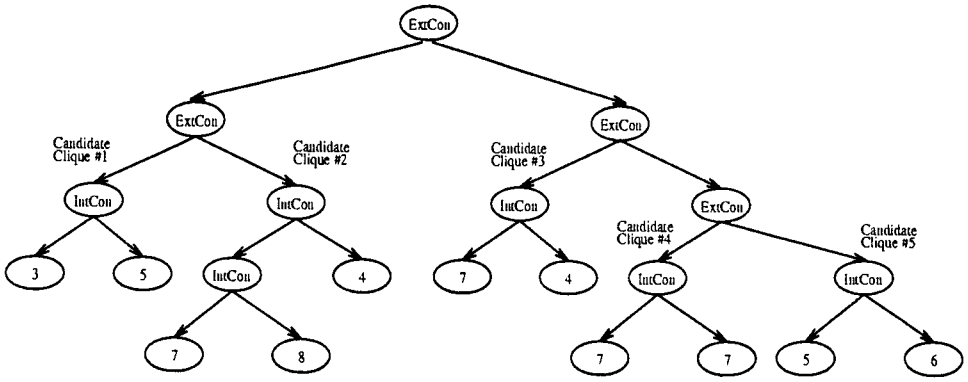

Figure 2: S-expression for 10 node graph.

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. It appears that the Active-Passive collective memory technique has failed to aid in the search process. We find if we instead adopt a Passive-Active collective memory technique in this domain, the search process is greatly facilitated.

With the Passive-Active collective memory we do not repair chromosomes which have no valid candidate cliques. Instead we gather candidate cliques in the collective memory, removing duplicates and candidates subsumed by larger candidates. In Figure 3 we present a comparison of three search techniques for clique detection (For all of our experiments, we set $\alpha = 10$ and $\beta = 9$ [6].). The noteworthy parameters for the STGP system were a max of 600 generations (Even if we find the optimal solution, we let the search continue on until the maximum number of generations had passed.) and a population size of 2000. Each curve shown in Figure 3 is an average of 10 different runs. Each of the methods extends the previous methods. The first method (R0) is a STGP system modified with the type inheritance presented in [6]. 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 354 for R0, 56 for R10Q7, and 8 for PACM. On the average, PACM is 7 times more efficient than R10Q7 and $44\frac{1}{4}$ times more efficient than R0. Finally, if we investigate how much the repair process is assisting the Passive-Collective memory, we see in Figure 4 that the addition of the duplication of coding segments repair is not significant. The PACMR10Q7 curve corresponds to the PACM curve in Figure 3, while the PACMR0 curve represents a Passive-Active collective memory which does not use the repair process.

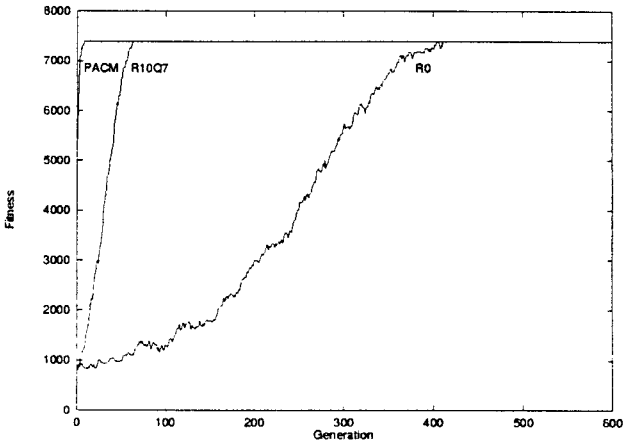

Figure 3: Comparison of best fitness per generation for no repair of chromosomes (R0), duplication of coding segments repair of chromosomes with a 10% return rate and 7 duplicates (R10Q7), and Passive-Active collective memory (PACM), which utilizes R10Q7 to drive the search agents.

5 EXPERIMENTS

The addition of Passive-Active collective memory to the search technique significantly improves the efficiency of the search process. 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 [9] 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.

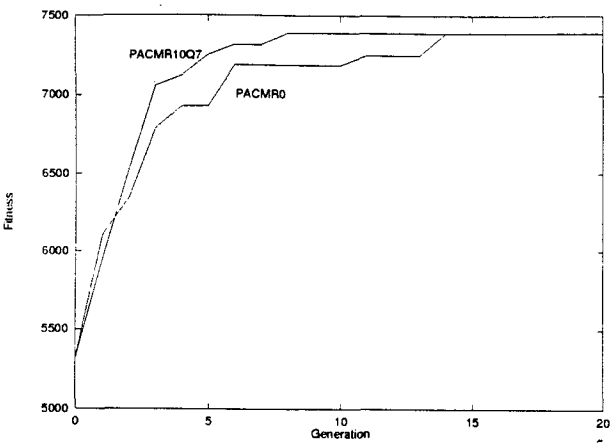

Figure 4: Comparison of best fitness per generation for Passive-Active collective memory (PACMR10Q7), which utilizes R10Q7 to drive the search agents, and Passive-Active collective memory (PACMR0), which utilizes no repair.

A shortcoming of these graphs is the results presented are for the maximal clique size found, if any, but no data is 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 [3]. To utilize graphs from the DIMACS repository to test the duplication of coding segments repair process, we must implement an algorithm to generate all of the cliques. A brute force algorithm is to build candidate cliques in increasing levels of size, k. Due to NP completeness [3], 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.

We 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. We present the results, in Figure 5, 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, we could allow the search to continue for more generations. We could also increase the population size. 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). We could allow for these agents to try to extend candidate cliques by either merging ones in the collective memory or by utilizing the deterministic brute force algorithm on a subset of the candidate cliques.

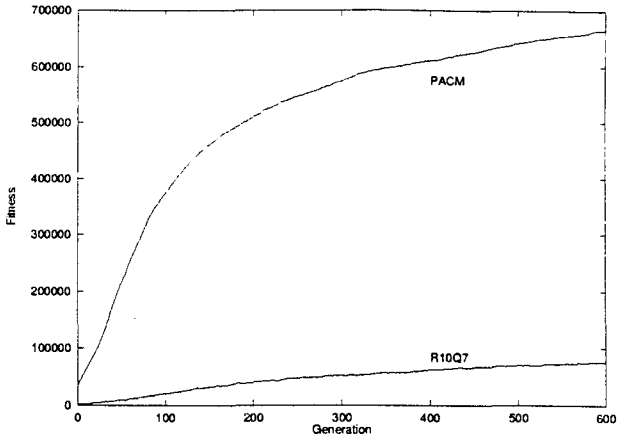

Figure 5: Passive-Active collective memory search applied to the hamming6-4 graph. In particular, comparison of best fitness per generation for duplication of coding segments repair of chromosomes with a 10% return rate and 7 duplicates (R10Q7), and Passive-Active collective memory (PACM), which utilizes R10Q7 to drive the search agents.

6 DISCUSSION

The models of collective memory search are effective if building blocks of the solution can be identified. In the clique detector domain, candidate cliques form the building blocks. The identification of building blocks in genetic programming is in general a difficult task [5, 12, 13]. In part this is due to the domain dependent nature of the alphabet; 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 [5]. The repair of chromosome by duplication of coding segments strategy holds promise in automating the detection of building blocks [5]. 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.

7 CONCLUSIONS

The collective memory search model 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 allow complex process agents to collate and process the

raw data. We could also employ simple process agents, capitalizing on the reduced search space.

We have shown collective memory can be used to augment distributed search. It can serve to be either a springboard from which further searches can be launched or as a central repository from which parts of the solution can be connected. We find the Passive-Active model significantly improves the search process. It also allowed scaling up in a problem domain, while our previous method failed to scale.

8 Future Work

In our current research, the Passive-Active model of collective memory search is applied to the clique detection domain. The process agent in this domain just collates the knowledge, removing duplicates. There is scope to improve the process agent such that it explores the search space by seeing if the candidate cliques can be merged to form larger candidate cliques. While the process agent is active since it collates, it can become even more energetic by exploring the collective memory space.

References

[1] Kenneth A. DeJong. Genetic-algorithm-based learning. In Y. Kodratoff and R. S. Michalski, editors, *Machine Learning, Volume III*. Morgan Kaufmann, Los Alamos, CA, 1990.

[2] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. The Ant System: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics*, 26(1):29–41, 1996.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

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

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

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

[11] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995. (Also published as BBN Technical Report #7866, Cambridge, MA, March 1994.).

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

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