Evolving Cooperative Groups: Preliminary Results

Maria Gordin, Sandip Sen, and Narendra Puppala
Department of Mathematical & Computer Sciences
The University of Tulsa
600 South College Avenue,
Tulsa, OK 74104-3189
e-mail:[gordin,sandip,puppala]@euler.mcs.utulsa.edu

Abstract

Multi-agent systems require coordination of sources with distinct expertise to perform complex tasks effectively. In this paper, we use co-evolutionary approach using genetic algorithms to evolve multiple individuals who can effectively cooperate to solve a common problem. We concurrently run a GA for each individual in the group. In this paper, we experiment with a room painting domain which requires cooperation of two agents. We have used two mechanisms for evaluating an individual in one population: (a) pair it randomly with members from the other population, (b) pair it with members of the other population in a shared memory containing the best pairs found so far. Both the approaches are successful in generating optimal behavior patterns. However, our preliminary results exhibit a slight edge for the shared memory approach.

1 Introduction

Our goal is to generate behavior strategies for cooperative agents that have distinct capabilities. A coordinated group effort is necessary to successfully complete the task. Performance of an agent depends on its performance in a group. Ability of agents to adopt to each other and cooperate becomes critical in this context. We have been investigating co-evolutionary methods to find such strategies in an agent group consisting of two agents. We use simultaneous evolution of two genetically distinct populations. Each population consists of rules (behavioral strategies) for one of the agents. The fitness of an individual in either of the populations depends on its ability to perform the task in a group with an individual from another population. Later we use a shared memory to store the n best couples found so far. Simultaneous evolution approachs have been successfully used in competitive environments. In Rosin [3] it was successfully applied in developing efficient strategies for simple two players games: Tic-Tac-Toe, Nim and Go. SAMUEL system [1] [2] applies co-evolution approach in simple domain of robots competing for food. For cooperative agents environment SAMUEL successfully develops strategies using evaluation of individuals from one of the populations against random members of the set of previous champions from another population. Our work can be considered where the best agent groups seen so far are stored (rather than best individual in each population) as another approach to co-evolution. Grefenstette's approach is more suited to competitive domains, whereas our approach is biased towards cooperative problems.

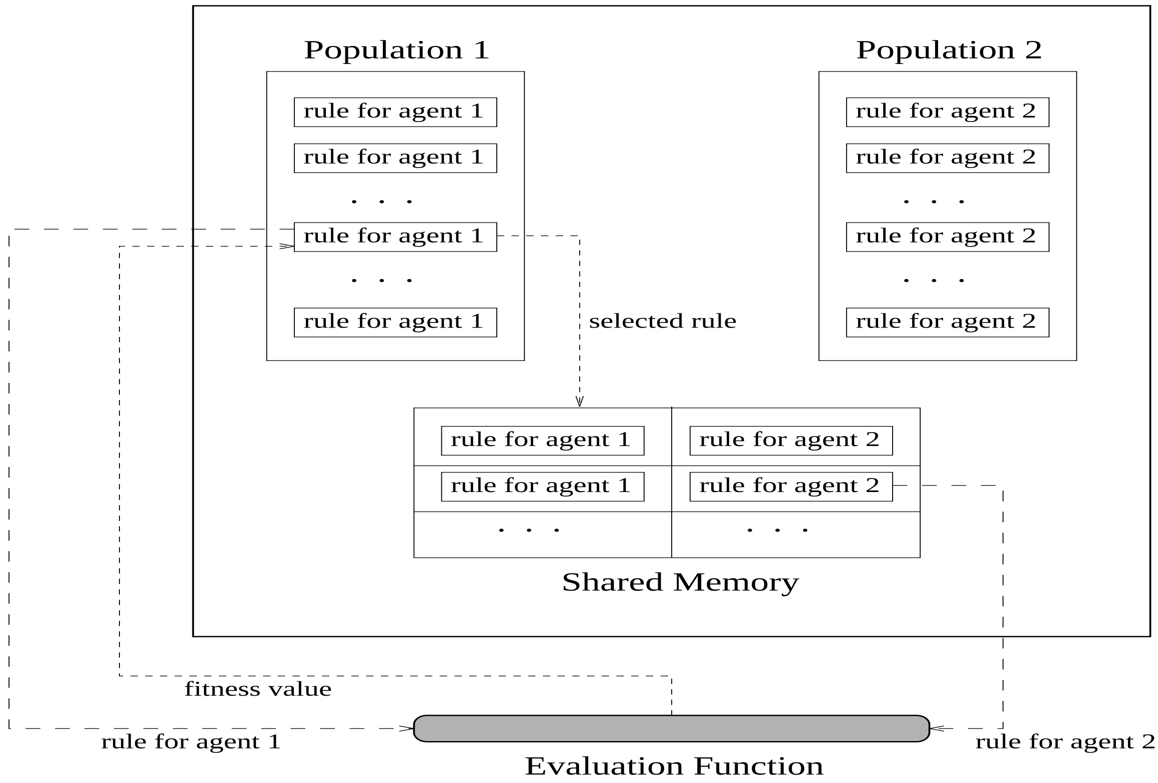

Figure 1: Using shared memory with two GA populations. The figure illustrates the process of evaluating an individual from Population 1.

2 Shared Memory in GA.

Though our shared memory approach can be used with larger groups, in this paper we use a domain with two cooperative agents. It takes efforts of both agents to complete a shared task. Each of the agents has unique capabilities necessary to complete the job. We would like to use a Genetic Algorithm (GA) to find rules for each agent so that the task gets completed most efficiently. Performance of agents as a group depends not only on the ability of each agent to perform its share of the task, but also on the ability of the agents to adopt to each other and cooperate effectively. This means that evaluation of an agent's performance needs to be done in the context of a group. In our experiments, we ran two different GA populations (one for each agent) concurrently.

To evaluate an individual from the first population, we paired it with an individual from the second population. One way of doing it was to randomly pick an individual to construct a pair. However in this approach, we did not have a way of remembering the successful pairs as only the best individuals from each population was available and the best individuals paired may not provide the best pair. This can be achieved with the shared memory approach to the problem, where the shared memory is a collection of the most successful pairs observed so far.

Another approach to evaluating individuals would be to pair it with one or more good individuals from the other population. Others have suggested using a random collection of the best individual from the other population over all generations for this purpose [1]. A variation of this approach would be to select a collection of best partners from the other generation. Rather than explicitly storing the best individuals separately, we chose to store the best pairs observed so far in a shared memory.

Direction to move	North	South	East	West	Distance
State of environment	N S E W	N S E W	N S E W	N S E W	
Whitewasher rule	0 2 2 2	1 0 2 2	1 1 0 2	1 1 1 0	03
Painter rule	3 2 2 2	4 3 2 2	4 4 3 2	4 4 4 3	

Figure 2: Rule representation.

In fact, if a pair performs better than any other pair, it could be because one of them perform really well, while another is not very effective. To improve the performance of such pair, only one of the players needs to be replaced. It means that we need to have a way of remembering the successful individual from one of the populations, pair it with different individuals from another population hoping to improve the result of the group. This can be achieved with shared memory approach to the problem. In a competitive setting, the evaluation of an individual should be some kind of average over its evaluation when pitted against each of the members from the other population [1, 3], as we want to evolve individuals that are robust. In the cooperative setting we assigned the highest evaluation from all pairings with an individual to be the evaluation of that individual. This is because we are interested in finding a pair who performs the best. So, an individual need not be robust, it just has to effectively co-adapt with another individual.

Shared memory consists of a collection of pairs of individuals that have demonstrated the best performance so far. Figure 1 illustrated the process of evaluation of an individual from Population 1. To get a fitness value for an individual from Population 1, we get a fitness value for a pair formed by that individual and an individual of Population 2 stored in shared memory. In our experiments we evaluated an individual from population 1 by pairing it with each of the individuals from Population 2 stored in shared memory. The maximum evaluation from all such pairs is used as the evaluation of the individual from population 1. Shared memory gets updated if we encounter a fitness value higher than at least one of the fitness values of the pairs stored in shared memory. In that case the pair with the minimum value gets replaced with the new pair.

3 Room Painting Domain.

For evaluating our proposed cooperative mechanism, we have defined a problem of room painting. A room is divided into many empty squares which needs to be painted. Two agents are required to do the job: whitewasher and painter. A square can be painted only after it has been whitewashed. If not painted after it is whitewashed, the square turns back empty after some period of time (three time ticks in our experiments). Both agents perform their tasks as they move. They can move in four directions: north, south, east, west, or hold (stay where they are). Whitewasher moves twice as fast as the painter. To complete the job in a reasonable time, the painter needs to learn to follow the whitewasher and the whitewasher needs to learn to wait for the painter if it gets too far ahead.

We now explain the working of the simulation used for executing agent behaviors. We use the following alphabet to represent states of a square: 0=empty, 1=painted or whitewashed, 2=anything, 3=whitewashed, 4=empty or painted.

The last two digits of the whitewasher's rule represents the maximum distance allowed between the agents. The whitewasher holds if the actual distance exceeds that number. Other than that the structure of the rules for the whitewasher and painter is the same.

The sensor function returns a pattern of four digits that represent the state of the environment respectively in the North, South, East and West. If the pattern returned by the sensor functions matches the pattern represented by the first four digits of the agent's rule, the agent moves North;

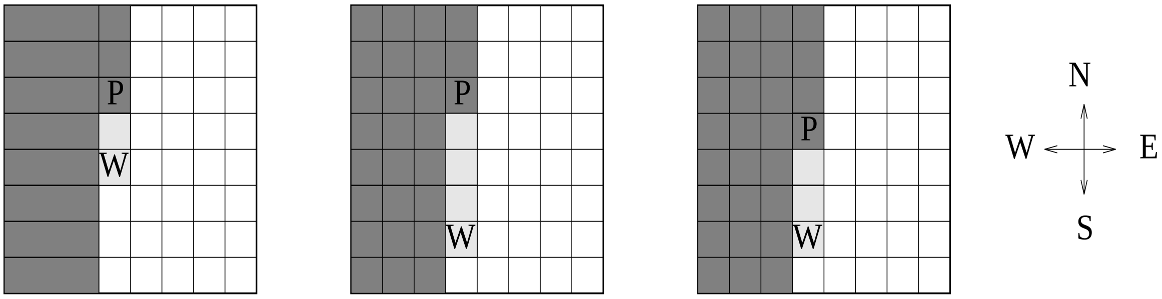

Figure 3: Example of room painting using rules in Figure 2

if it matches the second four digits, the agent moves South, etc. The rule structure is illustrated in Figure 2. For example, consider the movement of the whitewasher when the sensor finds that the square to the North is whitewashed, the square to the South is empty, the square to the West is painted and the square to the East is empty. The first part of the rule is tried first: 0 2 2 2. This rule states that if the square at the North is empty and the squares at the South, West and East can be anything, the whitewasher moves North. The square at the North is whitewashed, therefore this rule does not match the state of environment. Next part of whitewasher's rule is 1 0 2 2. It says that if the square at the North is painted or whitewashed, the square at the south is empty and the squares at the West and at the East are anything, the whitewasher moves South. This rule matches the state of environment, as the sensor has found that the square at the North is whitewashed and the square at the South is empty. The last two parts of the rule are ignored. If none of the four digit patterns matches the pattern returned by the sensor function, we switch to "global" mode. In "global" mode, the sensor looks at every square in specified direction up to the wall. For example the first part of the rule in "global" mode implies there is an empty square any where to the North, move North.

A set of optimal rules (rules with which the two agents can paint all squares for this domain room are the following:

Whitewasher:

0 2 2 2 1 0 2 2 1 1 0 2 1 1 1 0 0 3

Painter:

3 2 2 2 4 3 2 2 4 4 3 2 4 4 4 3

Figure: 3 shows how the agents move using rules in figure 2. Painter follows the whitewasher, the maximum distance between the agents is 3. In the figure whitewasher moves first. The second part of the rule matches the output of the sensor function, which finds an empty square at the south. the whitewasher moves south. Painter moves next. The sensor function finds whitewashed square at the south. It matches the second part of the painters rule. The painter moves south following the whitewasher.

Evaluation function. To get a fitness value for a pair, we run the simulator using a rule for whitewasher and a rule for painter. The simulator calculates the fitness value based on how much of the room is painted after a given number of ticks. We would like to maximize the number of squares painted. At the same time, we want to minimize the number of squares washed but not painted, because washed squares turn back empty and it is a waste of effort to wash too many squares as the painter won't be able to catch up. We used the following formula in our experiments: $p - (w - p) = 2 * p - w$ where, p is number of squares painted and w is the number of squares washed. (w - p) is number of squares washed but not painted (every square was washed before painted). The maximum fitness value for 8 x 8 room is 128.

4 Preliminary results and Future work

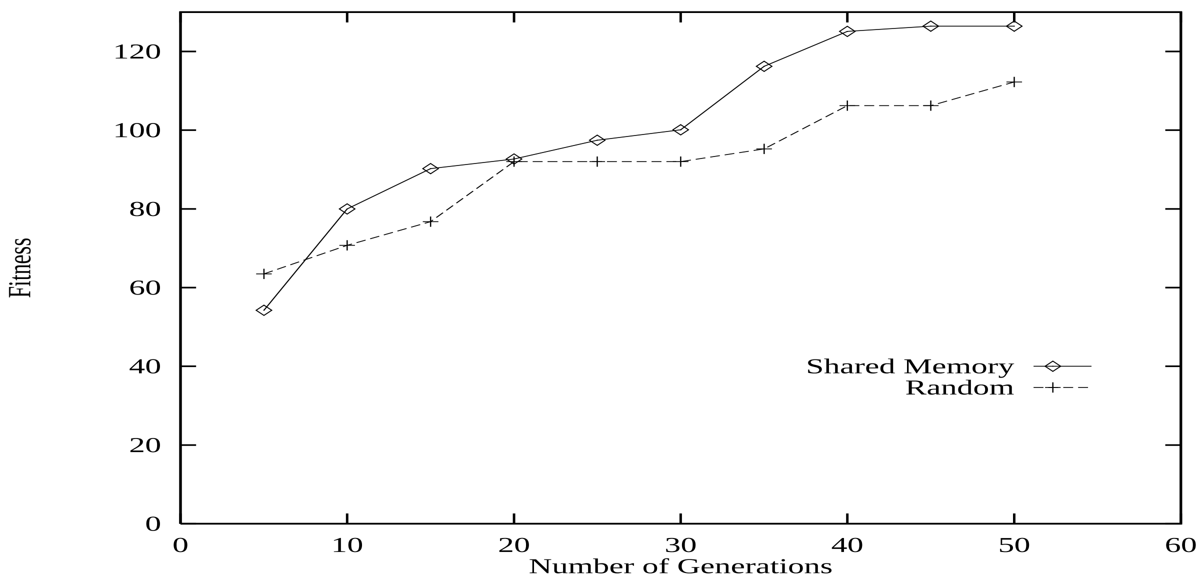

Figure 4: Shared Memory Vs Random

To evaluate the shared memory approach for this domain we ran two separate GAs: one for the evolving whitewasher rules and one the other for evolving painter rules. We kept pairs that performed best in shared memory. To get a fitness value for a whitewasher rule, we paired it with each of the painter rules stored in shared memory and ran the simulator on each of the pairs.

We now explain the outcome of our experiments conducted using the following parameters: Population size: 50, Crossover rate: 0.9, Mutation rate: 0.01, Shared Memory Size: 5. We ran two sets of 10 experiments each: one with random pairings and one with shared memory. We did not allow duplicate pairs in the shared memory. Results obtained from these preliminary experiments have been promising, with optimal rule pairs being consistently discovered by both approaches. Figure 4 shows the online average of the populations. The experiments showed that, while both the shared memory and the random technique generated close to optimal rule pairs, the former approach generates better rule pairs more consistently later in the runs.

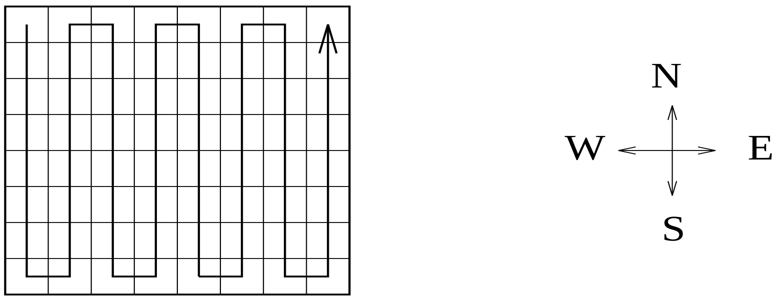

Figure 5: An example trace of the whitewasher and painter using a set of optimal rules generated by GA: 1102 1011 2221 2220 02 and 3223 4244 2434 2223.

All traces of solutions that demonstrate optimal or close to optimal performance are similar to the one on Figure 5. There are two reasons for that: (1) our rules have fixed order of preferences:

North, South, West, East; (2) we place both agents at position (0, 0) at the beginning of simulation. For future work we plan to make order of preferences a part of the rule and let GA find the best order. We also plan a series of experiments placing the agents to different initial positions.

Acknowledgments:

This work has been supported, in part, by an NSF grant IRI-9410180.

References

[1] John J. Grefenstette and Robert Daley. Methods for competitive and cooperative co-evolution. In *AAAI-96 Spring Symposium on Adaptation, Co-evolution and Learning in Multiagent Systems*, pages 45–50, 1996.

[2] John J. Grefenstette Mitchell A. Potter, Kenneth A. De Jong. A coevolutionary approach to learning sequential decision rules. In *Sixth International Conference on Genetic Algorithms*, pages 366–372, 1995.

[3] Christopher D. Rosin and Richard K. Belew. Methods for competitive co-evolution: Finding opponents worth beating. In *Sixth International Conference on Genetic Algorithms*, pages 373–380, 1995.