Emergent Cooperation in Reactive Multi-agent Systems

Marcelo Ladeira[1]
UnB / CPGCC/UFRGS
mladeira@inf.ufrgs.br

Lucia Maria Martins Giraffa[2]
II/PUCRS / CPGCC/UFRGS
giraffa@inf.pucrs.br

Ricardo Azambuja Silveira[3]
CPGCC/UFRGS
rsilv@inf.ufrgs.br

Luis Otávio Campos Alvares[4]
CPGCC/UFRGS
alvares@inf.ufrgs.br

Rosa Maria Viccari[5]
CPGCC/UFRGS / Leeds University
rosa@inf.ufrgs.br/ rosav@cbl.leeds.ac.uk

Abstract. We study the emergent cooperative behavior in a reactive Multi-Agent System (MAS) using the tileworld as a testbed. We take into consideration two strategies to simulate the behavior of the agents in their search for the tile pushing position: 'minimal distance' and 'clockwise movement'. The results are compared with emergent cooperative behavior by genetic programming.

1. Introduction

Distributed Artificial Intelligence studies the cooperative resolution of problems through a process or a group of descentralized agents. An agent is an entity that operates with some degree of autonomy and intelligence because they should have some reasoning and learning capacity and a way to cooperate and negotiate with other agents. Agents that explicitly have a capacity for reasoning and learning and are able to cooperate and negotiate with other agents are called cognitive agents. The cognitive agent approach needs an explicit representation of knowledge. A different approach is the so-called reactive approach. Reactive agents use a representation of behavior instead of knowledge. The activity of a reactive agent is produced by an interaction between agent and environment and not by an internal reasoning process [3].

In this paper, we focus on emergent cooperation in a reactive Multi-Agent System (MAS) and we use two tileworlds as a testbed (Figure 1 and 2). As defined by Pollack and Ringuette [1], the tileworld is a chessboard-like grid on which there are agents (denoted A_i, i=1,..,N), tiles (T), obstacles (#), and holes (V). An agent is a square unit which is able to move Up, Down, Left, or Right, one cell at a time unless, by doing so, it runs across an obstacle or finds one of the world's boundaries. When a tile is in a adjacent cell, the agent can push it by moving in that direction. A tile is a square unite which "slides", so rows of tiles can be pushed by the agent. An obstacle is a group of grid cells which are immovable.

In the original Tileworld [1], a hole is a group of grid cells with a tile "fill in" capacity, i.e., a cell that can be filled in by a tile when this tile is moved on top of it. The tile and the hole cell in question thus disappear, leaving a blank cell in their place. The agent scores points when all the

---

[1] Assistance Professor - UnB - Brazil - PhD student - CPGCC/UFRGS
[2] Associate Professor - Informatic Institute - PUCRS - Brazil - PhD student - CPGCC/UFRGS
[3] PhD student - CPGCC/UFRGS
[4] Associate professor - Informatic Institute - CPGCC/UFRGS - Brazil
[5] Associate professor - Informatic Institute - CPGCC/UFRGS - Brazil and Associate Researcher, Leeds University - England

cells in a hole are filled in. The agent knows ahead of time how valuable the hole is and its overall goal is to score as many points as possible by filling in holes. For sake of simplicity, we have omitted this criteria of hole capacity and score. In this way, the goal of the agent becomes to push all tiles as quickly as possible into the holes.

Manderick et al. [2] have studied emergent of cooperative behavior in the tileworlds in figure 1 through Genetic Programming (GP) and they have adopted the same simplification with regard to the hole's unlimited capacity and no scoring by the agent. The situation depicted in the TW1 can be used to illustrate a case where cooperation is needed to achieve the goal effectively. It would take more movements for agents A_0 or A_1 to perform the entire task on their own than it would if they worked together. The former needs 17 movements assuming A_1 is not on its way, and the latter needs 16 movements[1]. If the agents cooperate with each other the task will take them the minimum number of movements (i.e. 12) in each one of the both cases TW1 e TW2.

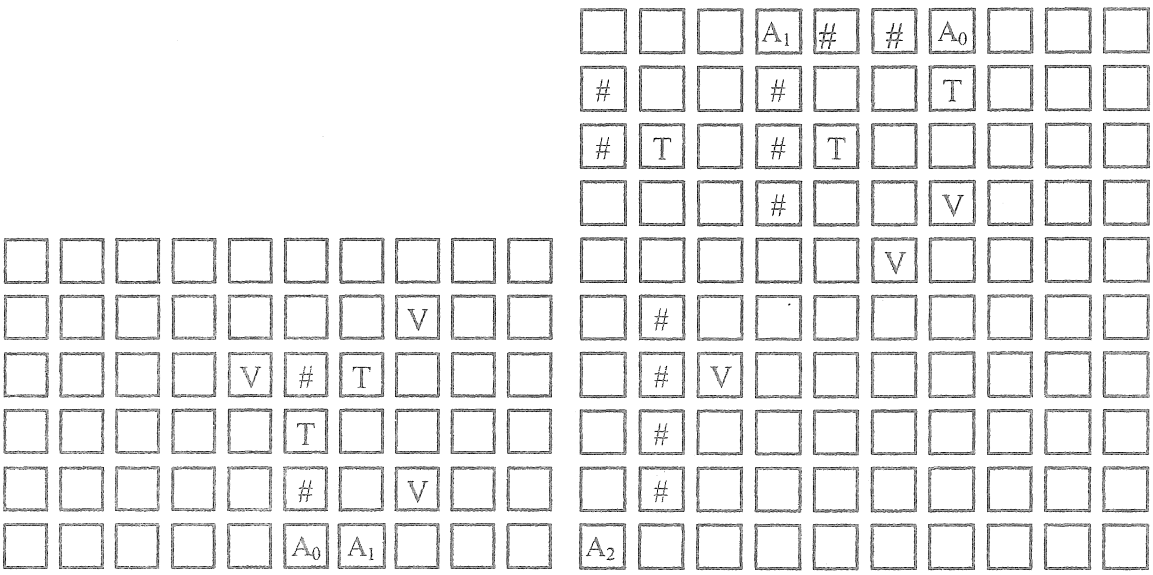


Figure 1 - The Initial State of the Tileworlds TW1 (left) and TW2 (right)

This paper starts by introducing Manderick et al.'s genetic programming for the Tileworld. Then the reactive agent modeling for the same Tileworld is presented. Finally, we discuss the results we obtained.

2. Genetic Programming for the Tileworld

Since the beginning of the 70s, algorithms based on metaphors of some natural evolution processes of living creatures have been suggested for the resolution of problems. Genetic Algorithms

[1] The agent A_1's movements, assuming the origin (row 0, column 0) at up left corner, are: (5,6), (4,6), (3,6), (2,6), (3,6), (3,5), (3,4), (4,4), (4,3), (3,3), (3,2), (2,2), (2,3), (1,3), (1,4), (1,5), (1,6)

(GA) make use of concepts such as natural evolution, population, genotypes, reproduction, mutation, crossover, and generation in the area of resolution of problems. Cooperative behavior can evolve through GA as follows [2]. First, allow the multi-agent system to behave according to each element of its population and evaluate its performance through a fitness function. Select the best members of the previous generation according to their fitness values (the higher their fitness value, the higher their probability to reproduce will be) and recombine their genotypes by using mutation and/or crossover. Make room for the new resulting genotypes (this can possibly be done by removing the worst fitness genotypes). Generate the next generation of the population including the new genotypes. Repeat this process until a behavior is found that allows the MAS to solve the task at hand, or the time limit (i.e. the maximum iteration number) is reached, whatever happens first.

Genetic Programming (GP) is an evolutionary computation technique where the genotypes are parse trees of Lisp-functions called GP trees. As Manderick et al. point out, GP starts with a randomly generated population de GP trees, each of which is assigned a fitness value. Next, some members of the previous generation are selected based on their higher fitness values. Finally, the selected GP trees are recombined using mutation and/or crossover to produce new GP trees. A new generation is formed when some GP trees are removed to make room for the new ones. This fitness evaluation-selection-recombination process continue until a solution for the task at hand is found or some stop criterion is met.

The mutation and crossover genetic operators change the GP trees in such a way that the resultant GP trees are always semantically meaningful. The mutation of a GP tree T consists of selecting, at random, of a subtree ST of T and replacing it with a new randomly generated tree ST'. The crossover of a pair of GP trees T_1 and T_2 consists of selectioning, at random, the subtrees ST_1 and ST_2, in T_1 and T_2 respectively. ST_2 then replaces ST_1 in T_1 and ST_1 replaces ST_2 in T_2, thus resulting the trees T_1' and T_2', which are called the offspring.

Manderick et al. represented the behavior of an agent via a parse tree of a Lisp function, in order to apply GP to the Tileworld. The Lisp function output is a vector v which has to be mapped on one of the possible actions an agent can take: STAY or move one step RIGHT, UP, LEFT or DOWN, depending on the direction of v. The agent must STAY where it is if the norm_of v is less than or equal to a certain parameter. The Lisp functions are built up with basic symbols like that Tile (the vector from the agent to the nearest tile), Hole (the vector from the agent to the nearest hole), etc.

In the case of the Tileworld, the fitness function of a GP tree was assigned in such a way that the following conditions are met: a) give a high score to a behavior which causes the agent to push a tile into a hole; b) give a higher score to a behavior which causes the agents to finish the task quickly; and c) if there are tiles left after the execution of the behavior, then give a higher score when tiles have been moved closer to some holes. The fitness function used by Manderick et al. is given by:

$$f = Bonus \times ft + Speed \times (Eval - t_F) + C_T \times \sum_{t \in LT} d(og(t), nr(t)) - d(cr(t), nr(t))$$

Where:
Bonus, Speed and C_T are parameters whose values are, respectively 3000, 80 and 100, in order to meet the three conditions above;
ft is the number of tiles pushed into the holes;
Eval = 50 is the fixed limit of time steps given to the agents to finish the task;
t_F is the *time* steps required to push all tiles into the holes. It is the same as *Eval* if the task is not completed;
LT is the set of tiles not pushed into holes;
d is the distance function;
og(t), cr(t) and nr(t) are the original position, the current position, and the position of the nearest hole to tile *t*, respectively.

These authors have considered three different breeding strategies to evolutionary cooperative behavior in the Tileworld through GP: homogeneous, heterogeneous, and the co-evolutionary breeding strategy.

One population of 1024 single behaviors evolves in the *homogeneous breeding strategy*. The agents execute a single behavior and this determines the effectiveness of the MAS. The resulting behavior may vary from agent to agent, since each agent's local environment is different, e.g. the nearest tile or the nearest hole might be different for different agents. Each member of the population represents a single behavior which corresponds to a GP tree and is evaluated by allowing all agents to execute the corresponding behavior and then taking their fitness values into account.

Once again, one population of 1024 individuals is evolved in the *heterogeneous breeding strategy* but now each member specifies a series of behaviors, namely one for each agent in the MAS. Each member of the population is a multi-branched GP tree with one branch for each agent in the MAS. Each agent executes its corresponding behavior in the series and job specialization among agents can occur. Crossover is restricted to corresponding branch pairs, during breeding. Each member of the population is evaluated by allowing each agent to execute its corresponding branch and then taking its fitness value into account.

Now the population is divided in subpopulations, one for each agent in the MAS. In this way, multiple subpopulations of single behaviors are evolved, one for each agent. A MAS is set up by selecting a behavior for each agent from its corresponding subpopulation. Manderick at al. point that now there is a problem of credit assignment because we know the effectiveness of the MAS as a whole and we now need to find a way to determine the relative contribution of each individual behavior since it is this that will determine which behaviors are selected from the subpopulations. They have solved this problem by randomly choosing the agents' behavior for the initial generation and then combine the best behaviors then evolved up to that moment in each Agent-type subpopulation, in later generations. In order to provide useful behavioral building blocks for the agents, the initial population of 1024 individuals is divided in N+1 subpopulations, where N is the number of agents in the MAS. One subpopulation, called Homo-type, evolves a program under the homogeneous strategy. The N other subpopulations, called Agent-type, evolve a specialized program for a particular agent. At each generation, M = 10 individuals from the Homo-type subpopulation emigrate to the Agent-type subpopulation. The best behavior evolved up to

that moment in Agent-type subpopulations is taken into account in the evaluation of the fitness values of all other members of all other subpopulations.

Table 1 presents the comparison of performance averaged over 20 runs, each one consisting of 100 generations. The average number to complete the task and the average best fitness values (between brackets) in the final generation are shown. As the minimum number of time steps to complete the task at the TW1 or at the TW2 is 12, then according to the fitness function, the maximum fitness are 9040 and 12040, respectively.

Manderick et al. conclude that the *homogeneous breeding strategy* gave satisfactory results for both Tileworlds but the acquired solutions were not necessarily optimal. The agents did not always cooperate effectively or did not work together at all. The optimal solution was often acquired for TW1 in the case of the *heterogeneous breeding strategy* but it gave worse performance on TW2 than for the other two strategies. The reason for that seems to be that this strategy requires a large number of subpopulations (3x1024 trees for TW2) which might degrade the GP search in the case of many agents. The *co-evolutionary breeding strategy* gave better results than the homogeneous one but worse results than the heterogeneous breeding on TW1. Its performance on TW2 was better than for the other two strategies. Cooperation only emerged for co-evolutionary breeding strategy. Another advantage is that this strategy allows for job specialization to occur without increasing the complexity of its population members.

Table 1 - Comparison of Performance averaged over 20 runs

Breeding strategy	Homogeneous	Heterogeneous	Co-evolutionary with Homo-type	Co-evolutionary w/out Homo-Type
Tileworld TW1	27.35 (7892.00)	17.81 (8655.00)	20.66 (8427.14)	36.16 (7187.00)
Tileworld TW2	39.89 (9889.00)	49.98 6081.09	20.19 (11465.00)	48.93 (9165.00)

Reference: Manderick et al. [2]

The results presented in the last column were obtained when the quoted authors [2] ran the same experiments for the co-evolutionary breeding strategy without migration from the Homo-type subpopulation to the whole Agent-type subpopulation. These results are much poorer than when migration is permitted.

3. Reactive Agent Modeling for the Tileworld

The implementation of the reactive agents in the worlds TW1 and TW2 was made using C Language in a C++ environment.).

Each world is composed by cells, each one with its own coordinates (x,y). These cells are either filled with: an agent ('a'), an obstacle ('#'), a hole ('h'), a tile('t') or are empty(' ').The gradient field of each hole is a system of coordinates (x,y) centered in the hole itself.

There are three ways to explore the world: exploring ('x'), pushing ('e') and find tile pushing position ('b'). There are four possible movements: exploring, random, pushing and find tile pushing position. Each Agent has three variables: mode, actual position and tile pushing position. The agent starts in the exploratory mode.
The program ends when all the tiles are pushed into the holes or when the iteration limit is reached.
The movements of the agents were modeled through the following algorithms:

- exploratory movement:

1. Evaluate surroundings, i.e. cells UP,RIGHT,DOWN and LEFT
2. If a tile is found go to pushing mode
3. If there are empty cells around perform random movement
4. Otherwise keep still

- random movement:

5. Choose the direction of movement through a selection process: 0(left), 1 (right), 2 (up) or 3(down).
6. Verify if the movement of the agent will hit a boundary or an obstacle.
7. The agent moves if everything is OK
8. Otherwise another selection is made (excluding the impossible direction) until a free path is selected and the agent is thus able to move.

- pushing mode:

1. Evaluate surroundings.
2. If there is not any tile in the surrounding tile, go to exploratory mode
3. If there is more than one tile, pick one at random
4. Select the hole with the same alignment as the tile. If there is none, then select the closest one. If there is more than one choice, select one at random
5. If a hole is selected then push the tile in order to reduce its distance from a hole
6. Go to exploratory mode if the tile is pushed into the hole
7. If the tile movement does not reduce the distance from the tile to a hole then determine the pushing position and go to pushing-position mode.

- tile pushing position mode:

1. Evaluate surroundings.
2. If there is a tile go to pushing mode.
3. If there are no free cells keep still.
4. Move according to selected strategy for behavior of the agents in the search for the tile pushing position: minimal distance or clockwise movement.
5. If there is more than one possible direction choose one at random.
6. If the tile pushing position is found, change to pushing mode.

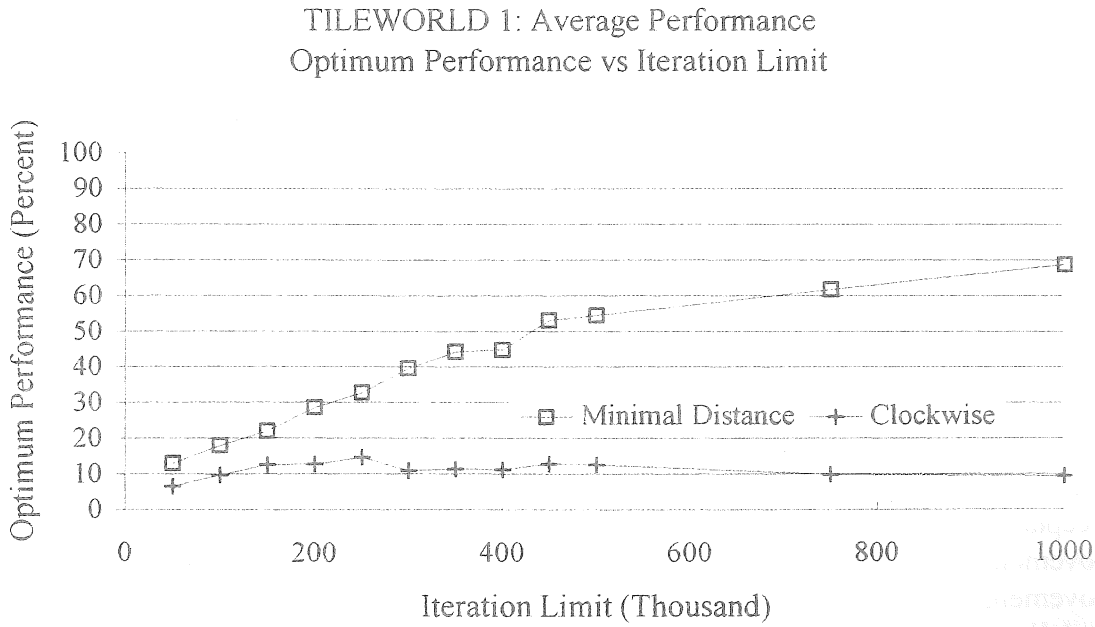
In the minimal distance strategy, the agent moves in the free direction that minimizes the actual distance to the hole. We have also modeled a behavior to avoid obstacles so that if an agent finds an obstacle, it just performs movement at random.

We have used a subsumption architecture [4] to prioritize the actions as follows. Note that the lower layers have more priority than the upper ones :
random movement
find tile pushing position
pushing movement
exploratory movement
avoid obstacles.

Thus, avoiding obstacles has the highest priority and random movement has the lowest.

4. The results

To evaluating the algorithm performance we are using the same fitness function and its parameters as Manderick et. al. (see Section 2). We averaged the values presented in Figures 2 and 3 running 200 experiments. These Figures presented the average performances of the algorithm considering the different tileworlds, TW1 and TW2, and the two possible strategies of agent's behavior when it is searching the tile pushing position. These average performances are maximal performance percentage. The maximal performance is obtained with agent cooperation when the task take the minimum number of movements



(i.e.12).

Figure 2 - Average Performance in the TW1

The world configuration affects the strategy performance. The minimal distance strategy presents better results than the clockwise strategy in the tileworld TW1 (as we can see in Figure 2) but not in the tileworld TW2. On the other hand, the clockwise strategy presents better results than the minimal distance strategy in tileworld TW2 (Figure 3). In the TW2 the objects were spread over a wide area and this feature makes the agent rounding easier or more likely to happen.

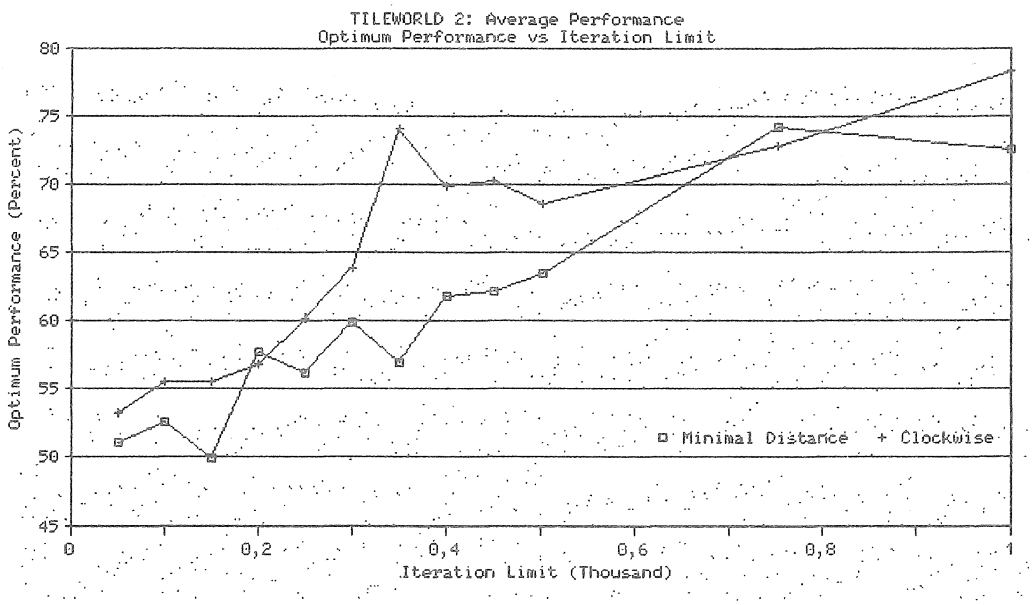


Figure 3 - Average Performance in the TW2

Firstly, the agents spend many iterations finding tiles randomly. After that, with lower iteration limit, the agents have not more time to complete its task. Consequently they present lower average performance. Initially the average performance increase sharply with the growth of the iteration limit and after that it increase slightly. The former behavior means more tiles pushed into holes. The later behavior means if the agents could push tiles into holes, it takes it around 500 iterations. If it was not successful it spend the remaining iterations doing redundant movements.

5. Conclusions

The solution presented by Genetic Programming is optimal but it is specific for each tileworld configuration. If you change the tileworld configuration you can not use this solution anymore. Therefore the solution is not robust. In addition, the solution by reactive agent is not optimal but acceptable. This approach is more robust than the other because we can use the same basic movements (e.g. random, find the pushing-position, pushing, exploring, and avoid obstacles movement) in different tileworlds configurations. To sum up, we can maintain the same agent's modeling in different tileworlds.

In spite of this, contrasting Genetic Programming and reactive agents approach, only Genetic Programming can evolve real cooperation behavior among the agents. But we believe that evolve cooperation by Genetic Programming for more complex tileworld will be more difficult because the complexity of the genotypes will be higher and it will be hard to find the optimal solution. Also we believe that evolve cooperation by reactive agent approach does not suffer these effect.

References

1. Martha E. Pollack and Marc Ringuette. Introducing the Tileworld: Experimentally Evaluating Agent Architectures. In *Proceedings of the 8th National Conference on Artificial Intelligence AIII-90*, pages 183-189, Boston, MA, 1990.

2. Bernard Manderick and Hitoshi Iba. Emergent Cooperation in Multi-agent Systems by Genetic Programming. Communication presented at the 13th Brazilian Symposium on Artificial Intelligence SBIA 96, Curitiba, Brazil, 1996.

3. Peter Wavish. Exploiting Emergent Behavior in Multi-Agent Systems. In Eric Werner and Yves Demazeau, Editors, *Decentralized Artificial Intelligence*, Elsevier Science Publishers, 1992.

4. R. A Brooks. Robust Layered Control System for a Mobile Robot. IEEE *Journal of Robotics and Automation*, 2:14-23, 1986