Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games (CIG 2007)

A Multi-Agent Architecture for Game Playing

Ziad Kobti, Shiven Sharma

Abstract— **General Game playing, a relatively new field in game research, presents new frontiers in building intelligent game players. The traditional premise for building a good artificially intelligent player is that the game is known to the player and pre-programmed to play accordingly. General game players challenge game programmers by not identifying the game until the beginning of game play. In this paper we explore a new approach to intelligent general game playing employing a self-organizing, multiple-agent evolutionary learning strategy. In order to decide on an intelligent move, specialized agents interact with each other and evolve competitive solutions to decide on the best move, sharing the learnt experience and using it to train themselves in a social environment. In an experimental setup using a simple board game, the evolutionary agents employing a learning strategy by training themselves from their own experiences, and without prior knowledge of the game, demonstrate to be as effective as other strong dedicated heuristics. This approach provides a potential for new intelligent game playing program design in the absence of prior knowledge of the game at hand.**

I. INTRODUCTION

In game playing, one of the most important aspect is the ability of the player to make intelligent, legal moves during game play. Many different approaches have been explored in this area, and much research and potential still exists to develop intelligent game players.

A. General Game Playing

The field of General Game Playing (GGP) is an important part of Artificial Intelligence (AI) research, and provides an important leap in the direction and approach of the construction of intelligent agent systems. In the past, much of the emphasis in the creation of intelligent systems was on the system being intelligent in its behaviour only for the task it was constructed to perform well in. GGP systems, as the name implies, are far more general. They are able to accept descriptions of any game, and are able to play them. The importance of this research lies in the fact that GGP systems provide a step from intelligent systems giving an illusion of intelligence to intelligent systems that act in an intelligent manner.

Though pure General Game Playing capabilities have not entirely been implemented, systems have been designed which display a general behaviour with respect to a specific class of games.

This work was supported in part by an NSERC Discovery grant.
Z. Kobti is with the School of Computer Science at the University of Windsor, Windsor, ONT, Canada N9B-3P4. (Phone: +1-519-253-3000; fax: +1-519-973-7093; e-mail: kobti@uwindsor.ca). S. Sharma, is with the School of Computer Science at the University of Windsor, Windsor, ONT, Canada N9B-3P4. (e-mail: sharmaw@uwindsor.ca).

B. Early attempts at General Game Playing: Positional Games

One class of games where general game playing has been investigated are *positional games.* These type of games were formalised by Koffman [6]. Banerji [1], Citrenbaum, Pitrat [3], and Banerji and Ernst [2] have studied these class of games. Some examples of position games include Tic-Tac-Toe, Hex, the Shannon switching games.

A position game can be defined by three sets, P, A, B. Set P is a set of *positions*; with set A and B both containing subsets of P. In other words, sets A and B represent a collection of subsets of P, with each subset representing a specific positional situation of the game. The game is played with two players, with each player alternating in moves, which consist of choosing an element from P. The chosen element cannot be chosen again. The aim for the first player is to construct one of the sets belonging to A, whereas the aim for the second player is to construct one of the sets belonging to B.

Programs that are capable of accepting rules of positional games, and, with practice, learn how to play the game have been developed. Koffman constructed a program that is able to learn important board configurations in a 4 X 4 X 4 Tic-Tac-Toe game. This program plays about 12 times before it learns and is effectively able to play and start defeating opponents. A set of board configurations are described by means of a weighted graph.

C. General Game Playing Architecture

For our purposes, we use the GGP architecture developed at Stanford University [9]. A GGP system consists of an agent designated as the Game Player (GP) and a Game Manager (GM). The GM is responsible for sending to the GP, initially, the rules of the game, and subsequently, the moves being made at each stage, and upon termination of the game, a termination message. The responsibility of the GP is to accept all the messages sent by the GM and take the appropriate action. Currently, Stanford University maintains at their website for GGP a GM to which GP's can connect and play games. They also maintain a rich resource base detailing the model of communication between GP's and the GM, the types of games that are playable in GGP and a set of game descriptions. The game descriptions are written in prefix Knowledge Interchange Format (KIF) [4]. They are written in such a manner that it is possible to use them and generate a set of legal moves from a given game state. In

order to facilitate and generate interest in GGP, a competition in GGP was held at AAAI 2005 and AAAI 2006, which pitched different GP's against one another [5].

D. Multi-Agent Social Environment for Game Playing

The credibility of a General Game Player lies in its ability to make not just legal moves, but intelligent moves. From the game descriptions, it is possible to generate legal moves from a game state. However, the challenge lies in the selection of a single move that can be considered *intelligent* from this set. Many different approaches are possible to allow a GP to make intelligent moves. Traditional search based approaches involved generating a tree representing the different possible outcomes from a given game state and searching the tree to come up with the best possible solution (or move) from that game state to the next. The problem with this approach lies in the fact that for games with large state spaces, such as chess, the trees can becoming astronomical in size, and therefore, become impossible to search exhaustively. In order to tackle this problem, trees are generated only up to a certain point in the space. Techniques such as Alpha-Beta search and Minimax search are commonly used in this paradigm.

We implement a new approach to facilitate a GP to make intelligent moves, and in order to allow it to handle general games, information regarding which game is going to be played is not provided. Our main GP, which is connected to the GM, before deciding on a legal move to make, enlists the aid of several sub-players (agents) to help it decide on an intelligent move to make. This is done by each of the agents assuming the role of the GP in the game and playing the game (exhaustively for games with a small search space, and up to a certain limit for games with a large search space). Each of these agents record their experience learnt during the play and share it with each other, and consequently converge to a series of strategies which they deem best to play. These are sent to another agent which decides which strategy to use, and that is sent back to the main GP.

In the following sections, we proceed to describe the underlying architecture of the multi-agent GP and the algorithms used in evolving strategies. Section II presents the architecture. In section III we discuss the underlying strategies and algorithms for generation of the training data and its evolution and the final selection of the move to be sent. Section VI presents the results from testing various forms of the multi-agent GP by playing Tic-Tac-Toe against a player using minimax as a search technique. In section V we outline the limitations and assumptions made in order to test the architecture. Finally in section VI we give an overview of future work in this area. The game description for Tic-Tac-Toe is available from Stanford University's GGP project website [9], and is used for our purposes.

II. Multi-Agent Environment

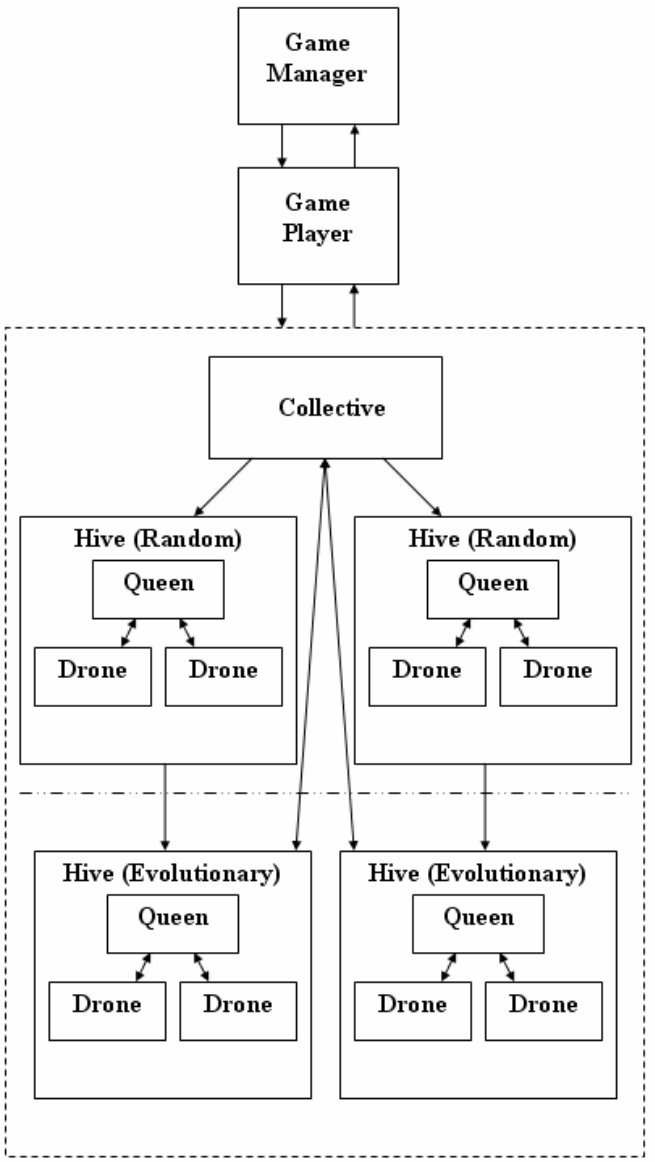

Fig 1. This figure shows the architecture of the multi-agent game playing system. The Game Player (GP) requests the Collective to send an intelligent move. The Collective first creates multiple Hives which make random moves and generate the training set.. Then the Collective creates multiple evolutionary hives that use the set and evolve strategies for making intelligent moves. Then the Collective selects the best move from these and sends it back to the GP.

Our game playing system consists of the main Game Player (GP), which communicates with the Game Manager (GM). Associated with the GP is the multi-agent environment. The simplest agent in this environment is called a *Drone*, which represents a player. Each Drone assumes a single role from the set of roles allowed by the game rules. A set of Drones, each having a unique role, are controlled by a virtual GM, called the *Queen*. It is the Queen's responsibility to accept moves from Drones, check their validity, inform the Drones about the moves made by the other Drones, and update the state of the game. In essence, the Queen is the analogue to the GM in the GGP

environment, and each Drone is an analogue to the GP. A grouping of the Queen and Drones, with each Drone representing the unique role allowed by the game rules, is called a *Hive*.

The Hive can be thought of being a single agent playing a game. In order to facilitate multi-agent playing, we create a finite number of Hives, each of which plays the game independently from the other Hives. A larger agent, the *Collective*, creates a finite number of Hives, and uses the results from each of these Hives to decide on the best move to make. It is the Collective which is in direct communication with the main GP.

There are two types of Hives that we use in our environment. One type is the *Random Hive*, in which the Drones play the game making random legal moves. The move sequences thus generated constitute the *training set*. The second type is the *Evolutionary Hive*. These use the aforementioned training set to evolve more intelligent sequences. They then report these back to the Collective which then selects the best of these and reports it back to the GP.

Fig. 1 illustrates this environment. The GP, when faced with the decision to make a move, calls on the Collective. The Collective then goes ahead and creates a finite number of Hives. The Hives play the game and communicate their results back to the Collective. The Collective then uses these results to decide which move is most likely to lead to a winning state. It then communicates this back to the GP, which in turn notifies the GM about it.

The GP, while playing the game, coordinated by the GM, maintains in its memory, apart from the game rules, the current state of the game. As the GM sends to the GP a message containing all the moves made by all the players in the previous turn, it updates its states in memory in order to reflect the new game state. It is these game states that are communicated to the Collective, which in turn passes these states to the each Hive it creates. The GP is only interested in the best move to make from the current game state. The previous moves do not affect this decision, except to the point that the previous moves led to the new game state. Therefore, each Hive plays the game from the new game state, oblivious to the previous moves made. If there are *n* Hives created by the Collective, then *n* random sequences of moves are generated by them (each Hive can be modified to return multiple sequences also). Since the GP has a limited time period in order to decide on the move and send it back to the GM, this time period is used to constrain the level to which each Hive plays the game from the new state. For small games, in which only a few sequence of moves leads to the terminal state, it is possible, given a sufficient time period, to play the game till the end. However, in games with a large sequence of moves, it may not be possible. For our purposes for testing, we use tic-tac-toe in which the entire sequence of moves from a given state till the termination of the game can be generated.

III. EVOLUTIONARY STRATEGIES AND ALGORITHMS

In this section we focus on the generation of the Hives, the generation of the training data set and the subsequent evolution intelligent move sequences from it.

A. Generation of the Random and Evolutionary Hives

Fig. 2 illustrates the algorithm for the creation of both Random and Evolutionary Hives in a sequential manner and in which each hive returns only one sequence. When the main GM requests the main GP to make a move, the GP requests the Collective to provide it with an intelligent, legal move. The information conveyed to the Collective is the current game state. When the Collective receives this request, it starts by creating a number of Hives. The first set of Hives created is the Random Hives, which lack intelligence, but always make legal moves in the given state. These Hives generate a sequence of moves of game play from the given state till the termination of the game. Only those sequences which are most likely to lead to a win are stored into a knowledge base. This constitutes the *training set*. The main purpose the training set serves is to provide some form of basic play information of the game, using which the multi-agent environment can generate intelligent moves (or move sequences).

```
ALGORITHM: CREATE-HIVES
INPUT: current Game State

FOR i ← 1 to m do
  create Random-Hive
  Move-Sequence-R ← sequence of random moves
                    returned from Random-Hive
  if (Move-Sequence-R is a winning sequence)
    store Move-Sequence-R in Knowledge-Base
END-FOR

FOR i ← 1 to n do
  create Evolutionary-Hive
  Move-Sequence-E ← sequence of intelligent moves
                    returned from Evolutionary-Hive
  if (Move-Sequence-E is a winning sequence)
    store Move-Sequence-E in Knowledge-Base
END-FOR

select best sequence from Knowledge-Base and return
the first move from that sequence
```

Fig 2. This figure gives the basic sequential algorithm for creating both types of Hives. The Random Hives return random sequences of legal moves from the current game state till the termination of the game. These are used by the Evolutionary Hives, which return intelligent sequences. The Collective then selects the best move and returns it to the GP. Note that both n and m can be either the same or different.

Once the training set is ready, the Collective now creates

a number of Evolutionary Hives and passes on the training data to them. The purpose of these Hives is to use the training data to create more intelligent moves for playing the game from that given state. The details on how they do this will be given in the next section.

Over time the Evolutionary Hives converge upon a sequence of more intelligent moves, and pass on this information to the Collective. The Collective then selects the best of these sequences and sends the first move of the selected sequence back to the GP. Details on this will also be explained in the next section.

B. Evolutionary Hives and Collective algorithms

In this section we describe the manner in which the move sequences generated by Random-Hives are evolved to produce more intelligent sequences, and how the Collective then uses these to select a move for the main GP.

Fig. 3 illustrates the algorithm used by the Evolutionary-Hive agents. Each Hive randomly selects one of the random sequences generated by Random Hives from the knowledge base. Once a sequence is selected, the hive proceeds to play the game by selecting every move from that sequence. In other words, it makes its moves at step i by selecting the move at position i in the sequence. If the move it makes cannot be made because it is not valid or legal, then it selects the next move in the sequence and makes it. If the end of the sequence is reached and there are still moves to be made, it makes random moves. In case of such a scenario, then upon game termination, the new sequence, which consists of the original moves and the newly selected random moves, is stored in the knowledge base (if it is a winning sequence).

ALGORITHM: EVOLUTIONARY-HIVES
INPUT: current Game State and Knowledge-Base containing random sequences of moves

FOR each Evolutionary-Hive, **do**
  Pick a sequence K_i from Knowledge-Base. K_i has l moves.
  FOR each move from m_1 to m_l in K_i, **do**
    if (move m_k is legal)
      make it
    else
      make move m_{k+1}
    if (end of sequence is reached)
      make random move
  END-FOR
  if (sequence played is different from K_i)
    store it in Knowledge-Base
  if (probability of crossover is met)
    Pick two sequences from Knowledge-Base
    Find matching point, and crossover the two sequences at that point.
    Store the new sequences in Knowledge-Base
  Examine Knowledge-Base to find patterns in the sequences and store them in Pattern-Base
END-FOR

Fig 3. This figure gives the basic algorithm used by the Evolutionary Hives to converge upon intelligent sequences from the random sequences generated by the Random-Hives.

The Evolutionary-Agents perform a crossover operation on the sequences with a probability p. In our testing, we selected p to be 0.20. This is performed by first randomly selecting two sequences from the knowledge base. Once selected, a random crossover point is selected. Crossover is then performed at that point. To illustrate, if sequence S_i and S_j are selected of lengths m and n respectively, and the crossover point is k, then subsequence $S_{i\{1,\ k\}}$ is appended to subsequence $S_{j\{k+1,\ m\}}$ and subsequence $S_{j\{1,\ k\}}$ is appended to subsequence $S_{i\{k+1,\ n\}}$. The new subsequences are stored in the knowledge base, along with the parent sequences. Note that the parent sequences can be removed as well, and only current generations be stored.

With the two aforementioned ways of exploring the initial random sequences, the agents aim to recognise patterns in the winning sequence. As the population of sequences grows, the agents try to recognise consistencies in the moves in the winning patterns. To illustrate this, consider the sample sequences represented by the vector of moves <*a, b, c, d, e*> and <*a, b, c, f, g*>. The agents recognise the pattern as <*a, b, c, *, **>, where the * represents any random move. Once the patterns are recognised, they are sent back to the Collective, which then selects the best and returns it to the main GP.

The Collective in our testing of the architecture uses a basic statistical count to select the best sequence. From the set of sequences returned, the Collective finds which sequence occurs the most, and then returns that back to the main GP. There are two options available to the Collective regarding what to return. Since the main GP is only interested in making a single move, the Collective can select only the first move from the sequence and return that to the GP. The second option is that the Collective return the entire sequence itself. The advantage of the latter is that the GP has the entire sequence, and therefore when it needs to make more moves (after making the current move), it can look at the pattern it received and select the next move in the sequence. On the other hand, if the next move is not legal, it can either chose to make a random move, or call the Collective again and ask it to return to it another move (or sequence).

IV. TEST RESULTS AND CONCLUSION

Here we present the results of different competitions of Tic-Tac-Toe by different versions of the Game Player (GP) against an opponent using minimax to search for moves. The GP and the Collective are both unaware of the game being sent. The first competition focuses on a match between a player making random, legal moves by selecting a move randomly from the legal move set, and the opponent. Then, we pit the opponent against a player playing with a heuristic specifically tailored for Tic-Tac-Toe. Next we play with the opponent and the Collective without using the Evolutionary

Hives, and finally we play with the opponent and the Collective using Evolutionary Hives. We compare the results of each of these competitions and evaluate the difference of having agents using a basic evolutionary strategy to use random sequences on intelligent general game playing. Our aim is to see how the Collective using Evolutionary Hives performs in the absence of game knowledge compared to the heuristic player which has prior game knowledge in the form of game specific heuristics.

A. Minimax vs. Random player

Here our player simply uses the current state to generate a random move which is legal in that state. The results of playing are shown in Fig. 4. for 30 games.

Player	Wins	Draws
Random Player	2	0
Minimax	28	0

Fig 4. Results of 30 games of Tic-Tac-Toe for a random move making player against Minimax player.

As can be seen, a player using Minimax has a definite advantage over one making simple random moves, as the former uses a well defined search to make moves, and therefore wins a vast majority of the games.

B. Minimax vs. Heuristic player

Here our player uses a specific heuristic defined for Tic-Tac-Toe. The heuristic is such that the player aims to mark as many adjacent pairs in the grid as possible. In other words, the player aims to mark squares [*i*, *j*] and [k, l], where *i* and *k* correspond to rows on the game grid and *j* and *l* correspond to columns on the grid, such that the squares are adjacent to each other either in a row, column or a diagonal. The player also aims to always check for such pairs and try to make an entire row, column or diagonal marked in order to win the game, while simultaneously looking for the opponents adjacent pairs and preventing it from making an entire row, column or diagonal and thus winning the game. The results of playing are shown in Fig. 5. for 30 games.

Player	Wins	Draws
Heuristic Player	30	0
Minimax	0	0

Fig 5. Results of 30 games of Tic-Tac-Toe for a player using specific heuristics for Tic-Tac-Toe against Minimax player.

The heuristic quickens the search for moves than a minimax search, while also playing *double tricks*, in which more than one winning sequence exists in a given game state, thereby enabling the player to get winning combinations quickly. Therefore, the heuristic player wins all the games.

C. Minimax vs. Collective without Evolutionary Hives

Here our player employs the Collective, but the Collective only uses the Random Hives, and selects a move statistically in the way describe above for Evolutionary Hives, i.e. it selects the sequence that occurs the most in the knowledge base and returns the first move of that sequence. The results of playing are shown in Fig. 6. for 30 games.

Player	Wins	Draws
Collective without Evolutionary Hives	5	20
Minimax	5	20

Fig 6. Results of 30 games of Tic-Tac-Toe for a player using the Collective architecture, which is not using Evolutionary Hives, against Minimax player.

The results show that even a simple statistical selection of random sequences enables the player to perform well enough to draw most of the games against the opponent. In the next sub-section we see how adding the Evolutionary Hives enhances the Collective.

D. Minimax vs. Collective with Evolutionary Hives

Here our player employs the Collective, using Evolutionary Hives. For our testing purposes, the Collective only returned the first move of the sequence it selected. The results of playing are shown in Fig. 7. for 30 games.

Player	Wins	Draws
Collective with Evolutionary Hives	30	0
Minimax	0	0

Fig 7. Results of 30 games of Tic-Tac-Toe for a player using the Collective architectur using Evolutionary Hives against Minimax player.

The addition of the Evolutionary Hives enhances the playing abilities of the player to such an extent that it performs almost as well as the player using heuristics. Since the player using heuristics is sure to perform exceedingly well, given that it has a well defined way of making moves, the fact that the Collective with Evolutionary Hives almost matches the performance of the heuristic player shows that the addition of the Evolutionary Hives makes a great difference in how the Collective selects moves.

E. Conclusion

The graph in Fig. 8 tabulates the results of all four aforementioned competitions, each having 30 games in total.

It can be seen that the player using the Collective with the Evolutionary Hives performs almost as well as the player using specific heuristics, even in the absence of knowledge regarding the game before the start of play. Also, the Collective is able to generate more specialized sequences of moves for Tic-Tac-Toe from a given game state than the Collective not using Evolutionary Hives. This is because the Hives are able to use simple evolutionary techniques to explore the sequences received from the Random Hives and recognize patterns in the winning sequences, thereby

generating more specialized and intelligent sequences.

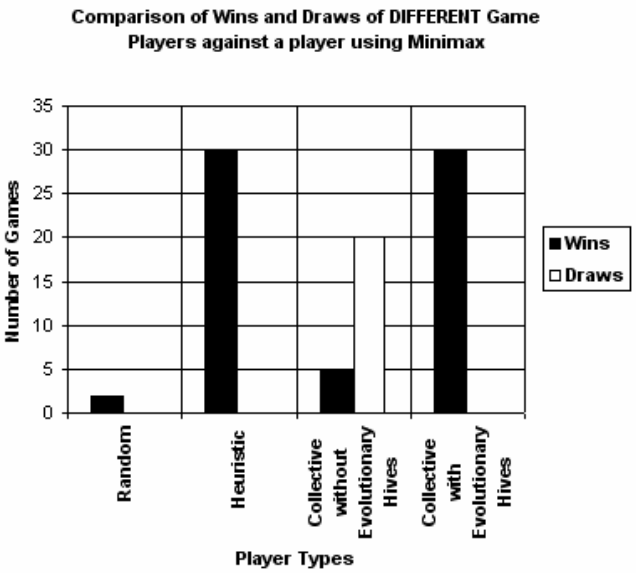

Fig 8. Comparison of the results of four different game players against a player using minimax search to play Tic-Tac-Toe.

We can conclude from our experiments that a player using the Collective architecture with the Evolutionary Hives for specialising game sequences can play, in an intelligent manner, in the current scenario, Tic-Tac-Toe, given the game description in the KIF format, and perform almost as well as a player which has prior knowledge of the game. Building upon this, we hope to establish that the Collective should be able to play any game given the game descriptions in KIF, thus emerging as a general game player.

V. Limitations and Assumptions

During the creating and testing of the Collective, we made two assumptions. First of all, it is assumed that the game being played can be played from a given game state till the end. Secondly, it is assumed that the player has enough time to generate enough training examples from the Random Hives, and the Evolutionary Hives also have enough time to generate specialized sequences to return to the Collective.

The current architecture also has a few limitations. The Hives are created sequentially, and that entails a large amount of processing time. Consequently, the Collective takes a large amount of time to return an intelligent move to the Game Player (GP). This problem can be avoided if the Hives operate in a parallel instead of a sequential manner. The Evolutionary Hives in this architecture use only a very basic evolutionary strategy to generate intelligent sequences and patterns. This can be enhanced by using more complex strategies. Also, the strategy employed by the Collective to select a sequence (or move) to return to the GP is a simple statistical selection. Again, more complex and efficient strategies can be used in this selection process. Finally, the techniques used for testing purposes work only for games with simple search spaces. The training set generated in our present architecture consists of sequences of moves from a game state till the end of the game. In games that require a large sequence of moves before termination, this may not be possible.

VI. Future Work

The architecture presented here is a new and novel approach to train general game players to learn how to play games. So far the tests have been conducted on a single game. However, the Collective was able to play the game without having prior knowledge of the game. In future tests, we intend on making the Collective play a wider range of games. We hope to establish that, using more complex strategies for move sequence generation and selection, the Collective architecture can be enhanced to create agents sepcialised in certain games or certain classes of games. This aspect of creating specialised agents over time is an important guiding factor in our work. Future work involves developing these strategies. Also, this architectural framework provides different ways to explore agent interaction in social agent communities and ways to share, represent and update knowledge learnt by each agents. Finally, we plan on improving the architecture to handle games in which move sequences till the game termination cannot be generated. For this we plan on evolving certain heuristics or rules that can assign a score to the sequences without actually knowing the final outcome of the game. This is an important aspect to be worked on, since given the absence of game knowledge prior to the start of play, determining the fitness value of move sequences that do not end in a terminal game state can be challenging.

References

[1] R. B. Banerji, “Theory of Problem Solving: An approach to Artificial Intelligence.”, 1969
[2] R. B. Banerji , G. W. Ernst, “Changes in representation which preserve strategies in games.”, 1971
[3] R. L. Citrenbaum, “Strategic pattern generation: a solution technique for a class of games.”, 1972
[4] M. Genesereth, N. Love, “General Game Playing: Game Description Language Specification”, 2005
[5] M. Genesereth, N. Love, B. Pell “General Game Playing” *AI Magazine*, 2005
[6] E. G. Koffman, “Learning through pattern recognition applied to a class of games”, 1967
[7] P. C. Jackson Jr., *Introduction to Artificial Intelligence*, 1985
[8] J. Pitrat, “A general game playing program” 1971
[9] http://games.standford.edu
[10] J. Liu, C. Yao, “Rational competition and cooperation in ubiquitous agent communities”, *Knowledge-Based Systems*, 2004
[11] D A. Ostrowski, T. Tassier, M. Everson, R. G. Reynolds, “Using Cultural Algorithms to Evolve Strategies in Agent-Based Models”, *IEEE*, 2002
[12] Z. Kobti, R. G. Reynolds, T. Kohler, “The effect of kinship cooperation learning strategy and culture on the resilience of social systems in the village multi-agent simulation”, 2005
[13] A. Namatame, T. Sasaki, “Competitive Evolution in a Society of Self-interested Agents”, 1998