Multi-Agent Based Chess Move Generator System: Taking into Account Local Environments

Juan F. Rodríguez Hervella
Univ. Carlos III de Madrid
Av. Universidad, 30, Edif. Torres Quevedo. E-28911 Leganés (Madrid)
Tel: (+34) 91-624-8859
E-mail: jrh@it.uc3m.es

Abstract

Chess has spawned a great deal of successful research in the artificial intelligence (AI) community. The strength of some chess programs can be compared to top-level human players. Nevertheless, most of the AI research has been focused on the refinement of evaluation functions which are used to search the NP-complete solution space. Some optimizations have also been made to the arrangement in which possible moves are examined by the search algorithm. In this article we take a nouvelle approach to the generation of chess moves with the help of multi-agent based systems. We design a multi-agent system which represents the pieces of a chess battlefield and we feed the search algorithm with the most promising move(s) from the point of view of the individual agents. Thus, we take into account the individual decisions (tactics) to define the strategy, as well as cutting out the search space. We plan to make a prototype implementation of the multi-agent based chess move generator system (MAB-CMGS) using the FIPA-OS architecture to test the overall system behaviour and achievable chess power.

1. Introduction

Strategy can be defined as a coordinated plan of actions for reaching a particular goal [1]. A strategy always relies on the possibility of having a view as global as possible of the current situation, and making sure that the resources will perform the desired actions. Strategy is a synonym of "action plan" or "intelligence", which can be viewed as the set of steps that an agent carry out to solve a problem.

Although chess is a game of "perfect information", which means that both players are aware of the entire state of the game at all times, the search space is so huge that brute-force techniques [1]can not be directly applied to it. Thus we usually talk about the "intelligence" of chess programs because they have to take decisions based on the current state, in a similar way human players do, because both humans and computers can not examine all the possibilities to let them choose the best move.

The traditional methodology of AI has consisted in the decomposition of intelligence into functional information processing modules whose combinations provide overall system behaviour [2]. In this model, the reasoning process becomes trivial once the search space can be managed and the domain of knowledge is well understood.

AI solutions for playing chess are founded in the hope that "intelligence" will somehow emerge from good heuristic evaluation functions. The search space is reduced by replacing the full problem with a simplified version. Rather than computing the full tree of all possible moves for the remainder of the game, a more limited tree is computed, with the tree being pruned at a certain number of moves, and the remainder of the tree being approximated by the evaluation function.

On the other hand, some AI methodologies have faced the chess battlefield from a totally different point of view [3]. These methodologies base the decomposition of "intelligence" into individual behaviour modules, whose coexistence and co-operation let more complex behaviours emerge [2].

At the time of writing, both methodologies have not achieved yet the secrets of the "holy grail of AI" [2], which aims to adquire human level intelligence equivalence. Regarding the chess problem, heuristic solutions have demostrated superior performance than individual decision processes [16].

In this paper we try to put together both ways of tackling with the chess problem, namely the heuristic based computation and the interaction of (chess) agents with the enviroment, in order to achieve a more precise resemblance of human chess players' behavior. In the next section (section 2) we introduce the current chess programming tecniques. We continue with a section (section 3) devoted to the MAB-CMGS system description. Section 4 outlines the prototype implementation that we plan to develope and we finish with some conclusions and future works in section 5. We hope to validate our statements when our first prototype comes to light. Until then, we plan to utilize incremental refinements until a complete and detailed description can be fully introduced. This paper should be view as the first of a series with the final objective of joining determinitic and behavioural based tools to simply create a grandmaster.

[1] a **brute-force search** consists of systematically enumerating every possible solution of a problem until a solution is found, or all possible solutions have been exhausted.

2. Chess programming overview

In order to play chess, a computer needs a certain number of software components. It needs some way to represent a chess board (Figure 1) in memory, so that it knows what the sate of the game is. It also needs rules to determine how to generate legal moves, a tecnique to choose the "best" move to make amongst all legal possibilities, and some way to compare moves and positions, as well as some sort of user interface [4]. In the following sections, we describe the most important functions of a chess program without explaining them all. We refer the reader to the bibliography to get a deeper understanding of the different methods.

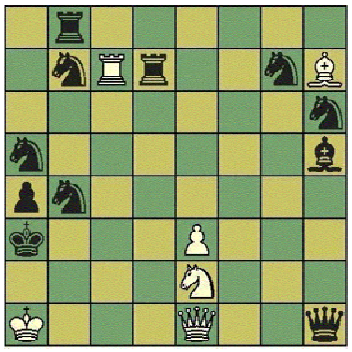
Figure 1: Typical chess board

It is also worth noting that the following concepts have already been applied to distributed computation, for example in the "chessbrain" project [8].

2.1. Move Generation

A chess grandmaster thinks by "intuitively" selecting 2 or 3 moves and then analyzing them several moves deep. Chess programs attempting this "selective generation" have generally failed, as it is extremely difficult to put a grandmaster's intuition into a program. The selective generation of moves is also known as "forward pruning".

Today nearly all programs use incremental (or complete) generation of chess movements, which consists on:

- Finding all of the legal moves available in a position.
- Ordering them in some way, hopefully speeding up the search by picking an advantageous order.
- Searching them all one at a time, until all moves have been examined.

The classification of moves is a fundamental strategy to improve the performance of the algorithm. Therefore, many programs generate captures first, often starting with those of highly valuable pieces. Some other methods are described in [4].

2.2. Searching algorithms

We need to search the solution space basically because we are not smart enough to play chess without it. A really bright program might be able to look at a board position and determine who is ahead, by how much, and what sort of strategy should be implemented to drive the advantage to a win. Unfortunately, there are so many patterns and so many rules as well as exceptions, that it is not feasible for a computer program to obtain that knowledge. Hence, we look at searches as an easy way to "teach" the machine about relatively complicated tactics. Some efforts have been addressed to find alternatives to the searching methods, as it is explained in [1] and [15].

The basic idea underlying all typical chess search algorithms is the "minimax" principle [5]. The strategy behind the minimax algorithm is that the computer assumes that both players will play to the best of their ability. So, if the opponent has the choice of a bad move or a good move, the computer will have the opponent choose the good move. The player moving a piece tries to maximize his gain according to the evaluation function, the adversary tries to minimize it. The interplay of both intentions results in a way to explore the game tree. If the solution space of chess could be efficiently searched, we could use this algorithm to create a "perfect" player. Unfortunately, this exploration can not be achieved in a valid computational time.

Improvements to make the minimax algorithm feasible have been developed, like Alpha-Beta pruning [6], and some other improvements to speed up the search are currently being applied, like ordering the moves and the iterative deepening alphabeta concept [7].

2.3. Evaluation functions

While search techniques are pretty much universal and move generation can be deducted from a game's rules and no more, evaluation requires a deep and thorough analysis of strategy. Some of the evaluation metrics commonly used are material balance, mobility and board control, development, pawn formations, and king safety [4]. It is still an open problem how to weight each one.

Evaluation functions appear because we can not apply the minimax principle to its final consequences. Programs usually stop searching and assign a value to the last search node (up to date leafs). Good evaluation functions are mostly developed and improved using trial and error techniques. For a more detailed description of evaluation functions we refer the reader to the extensive bibliography available.

3. Multi-Agent Based Chess Move Generator System (MAB-CMGS)

To build a system that is intelligent it is necessary to have its representations grounded in the physical world [2]. The idea outlined in this paper relies on the emergence of more global behavior from the interaction of smaller behavioral units. As with heuristics, there is no "a priori" guarantee that this will always work. However, as it is been shown in [1], careful design of the simple behaviors and their interactions can often produce systems with useful and interesting "emergent properties".

We utilize the previous work done inside MARCH project [1] to create a chess program that is driven by the more promising moves that appear when a set of agents start interacting between them.

The approach consists in viewing each chess piece as an autonomous agent, with its own behavior and its own perception area.

3.1. The overall process

Every chess piece has to communicate with the rest of the agents, both friends and enemies. Besides, the agents have to send the evaluation of its moves to their respective King agent, which controls the strategy of the game. The King agent implements a typical search algorithm to look up the chess space, using minimax and alpha beta techniques. The agents simply inform the King agent, which ultimately is the agent in charge of deciding which move to make next, in order to advance the game. This is a totally different approach to the multi agent chess system described in [1], because we do not allow the agents to select the next move based on the agent's assessments. The next move is based on a heuristic search with a limit on the number of branches, which depends on the information provided to the King agent. Thus we can think of the King agent as the "head of our soldiers".

The idea of examining a few moves in order to play chess was first stated by *Mikahil Botvinnik*. He was convinced that the only way for a computer to ever play grandmaster-level chess was to play like a grandmaster, i.e., examine only a few moves, but in a great depth and detail. We identify and implement the sort of high-level plans which a world class player might come up with during a game, with the help of a multi-agent system platform, as it will be explained later on. Another variation regarding to the *Botvinnck's* thoughts is that we do not aim to obtain the "best" move. The King agent dynamically decides which moves are the more promising to be searched based on the information provided by every agent. We even could end up in a situation where the agents' evaluations were not enough to make any selection, which would mean that the algorithm would have to consider all the possible moves.

At this point is not clear if the agent based selection of moves could be effectively applied for each ply[2] of the search algorithm, instead of only for the first move. We plan to test both approaches.

3.2. The detailed process

The multi-agent system is modelled following the description of [1] plus the introduction of the objectives that every agent needs to achieve. Hence, we face a similar testbed in which every place of the chessboard knows the chess piece that is lying on it and holds two values called "whiteStrengh" and "blackStrength". Every square also holds the two differences between these values, respectively called "whiteDiff" and "blackDiff". In order to fully understand our genuine decision process, the algorithm which appears in [1] is directly copied here. Later on we explain the modifications we have made.

The way a turn is played on [1] is the following:

1. Asking each chess piece (agent) to
   a. Propagate a constant value on the places it directly threatens.
   b. Inform the enemy's pieces it directly threatens that they are threatened by itself.
2. Asking each threatened pieces to propagate its material value (1 for pawns, 3 for knights and bishops, 4 for rooks, 10 for queen, and infinite for the king) on the places situated between itself and the pieces that threaten itself.
3. Asking each piece to give a mark to each place onto which it could move. The mark is computed as follows:
   a. Initially the mark is whiteDiff or blackDiff, depending on the piece color.
   b. If there is any enemy piece on the place, its material value, multiplied by two, is added to the mark.
   c. The material value of each enemy piece it would threaten when located on this place is added to the mark, as well as the material value of each allied piece it would protect (except that of the king).
   d. Finally, the material value of the piece and the "whiteDiff" or "blackDiff" value of its place is removed from the mark.
4. Choosing randomly a piece among the pieces whose mark are the greatest and asking it to move on the related place.

Our approach extends and modifies this algorithm introducing a novel definition of the forth point. We

[2] One ply is one move by one side

have already stated that the actual movement is taken by the King agent based on a heuristic search. Hence, instead of choosing randomly a piece with the greatest mark to move on the related position, we just send the mark information to the King agent, which selects the movements with greatest marks to start the search algorithm. Upon timer expiration the King agent is forced to evaluate the most promising move, and it sends a message to the proper agent to make the move actually occur, and the turn is given to the correspondent player (opponent's King agent) and finally the process starts again, having this time the "hot potato" on the other side.

4. Implementation of MAB-CMGS in the FIPA-OS architecture

We plan to make a first implementation of the concepts outlined in this paper using the FIPA-OS multi-agent platform. FIPA is a non-profit association whose purpose is to "promote the success of emerging agent-based applications, services and equipment" [4]. FIPA's goal is to make available specifications that maximize interoperability across agent-based systems. FIPA operates through the open international collaboration of member organizations, which are companies and universities that are active in the field. Figure 2 shows the abstract FIPA architecture and emphasizes the different actual implementations of the FIPA's standards.

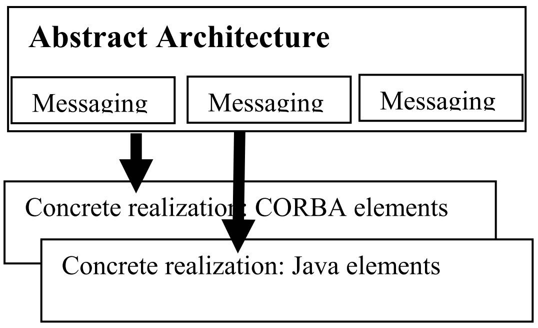
Abstract Architecture
Messaging
Messaging
Messaging
Concrete realization: CORBA elements
Concrete realization: Java elements

Figure 2: FIPA architecture and its realization

The FIPA abstract architecture [9] defines a high-level organisational model for agent communication and core support (minimum support required) for communication such as a directory service, message transport service and namespace. The abstract architecture is neutral with respect to any particular directory service or the use of a particular network protocol for message transport.

Figure 3 shows the overall agent platform defined by FIPA. It is worth noting that some parts of the platform are modelled as agents while some others are non-agents elements. The decision about where a specific feature belongs to has been made through a iterative process throughout the years. For instance, in early FIPA specifications the "Message Transport Service" (MTS) was modelled as an agent [14].

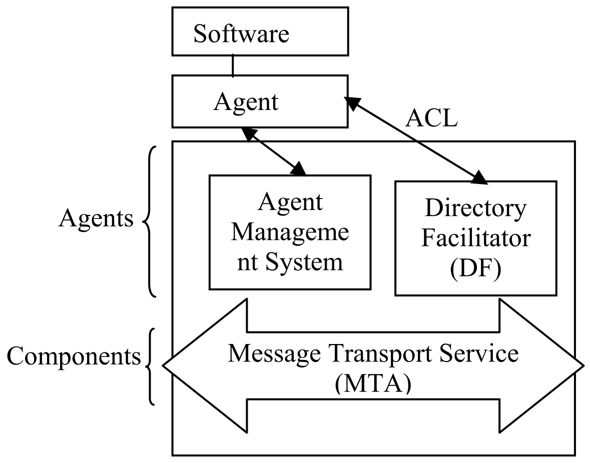

Figure 3: FIPA reference model

We have chosen to implement our system using a FIPA based agent platform because the FIPA set of standards focus on the specification of external communications between agents rather than the (internal) processing of the communication at the receiver. This is the same thing as saying that the FIPA specifications attempt to cover generalisations and high-level neutral abstractions. Thus, using a FIPA based platform allows us to assure that our development will be compatible and focused on inter-operability between different kinds of agents.

In the following subsection we describe the concrete FIPA platform called "FIPA-OS". Afterwards we describe the real state of the development of the MAB-CMGS system.

4.1. FIPA-OS multi-agent platform

FIPA-OS is an agent development toolkit and a FIPA compatible platform [13]. The FIPA standards aim to improve agent interoperability by providing standards for agent communication language (ACL), agent life cycle and how agents interface the platform management and directory functions [12]. FIPA-OS provides the basic high level communication and conversation management functionality, the ability to create tasks and subtasks, the basic platform services specified by FIPA (Agent Management System and Directory Facilitator, respectively spelled AMS and DF in short) as well as the lower level transport system that supports a set of transports such as RMI [10] and CORBA [11].

The AMS provides platform management functionality, such as monitoring agent lifecycles and ensuring a correct behaviour of entities within, and upon, the platform. The DF provides "yellow pages" services to other agents.

FIPA-OS uses a stack-based layered design, where each agent consists of a number of components that are placed in a stack. The stack is used within the agent, and in addition a transport stack resides within the MTS. Figure 4 presents an abstract diagram of the FIPA-OS agent stack.

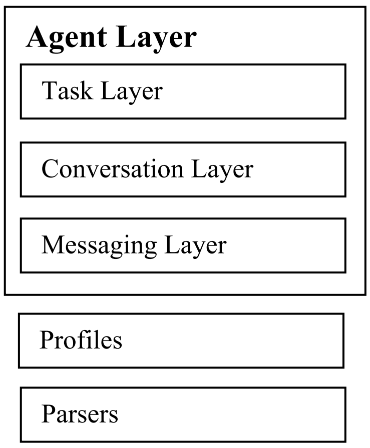

Figure 4: FIPA-OS architecture stack

FIPA-OS consists of the task and conversation management components, messaging service and the transports. The platform components required by FIPA (AMS and DF) are implemented as agents, with the exception of the Agent Communication Channel (ACC) that handles inter-platform communications. More information about the platform structure can be obtained from the project home page at [13].

4.1.1. Installation instructions

The latest package of FIPA-OS dates from 2003-03-18. There is plenty of documentation about FIPA-OS, for example the instructions concerning the installation of the platform can be found in [12].

There are some hardware requirements (Pentium 166 Mhz processor, 64 Mbytes of memory and 4 Mbytes of disk space), but the only software requirement consists in the availability of Java support (runtime environment and collection and swing classes). The FIPA-OS platform has been developed with Java 1.2.2, but we have not found any problem with the latest version of Java at the time of this writing, which is Java-1.4.2. The installation has been carried on a FreeBSD-4.10 machine.

The package comes with a self-extraction JAR file that contains the Java classes that take care of the process of installing the platform and starting the FIPA-OS wizard. It is enough to execute:

```
Java –jar FIPA_OSv2_1_0_Installer.jar
```

Once the wizard shows up, you only have to choose the installation directory and you can safely accept the default values of the rest of the windows.

The platform has the following directory structure:

Path and filename	Description
\bat	Script files for launching agents and configuration
\certificates	Security certificate keys for agents. Only used by RMI over SSL.
\classes\ FIPA_OSv2_1_0.jar	Compiled FIPA-OS core without diagnostic support
\databases	Default location where persistent databases will be stored by agents
\imports\	Third-party components (e.g. Xerces)
\src	Source code
\javadocs	Documentation of classes
\docs	Distribution notes
\docs\licenses	License notes
\tools	Associated tools
\profiles	Agent profiles for each of the agents included in the distribution (AMS, DF)
\examples	Example ACL messages

Once the installation wizard has finished, simply running the “bat\|StartFIPAOS” script and the system will turn up. Figure 5 shows the look and feel of the platform. Both the AMS and the DF provide a graphical interface, as most of the example agents usually do.

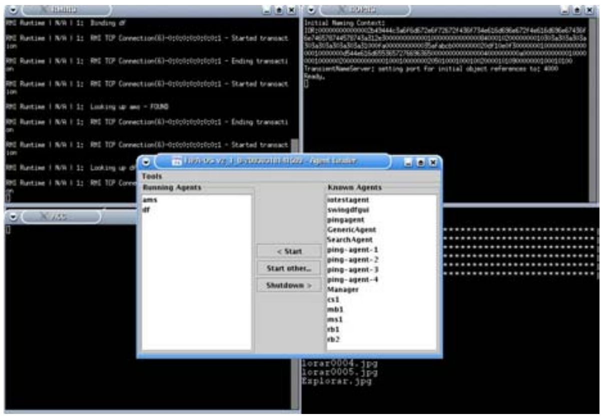

Figure 5: FIPA-OS look and feel

Figure 6 shows a more detailed view of the Agent Loader. It is recommended that the Agent Loader be used to start all agents. This enables agents to be managed by a human user as required. Another advantage of using the Agent Loader is that the user can control the lifecycle of the agents at runtime.

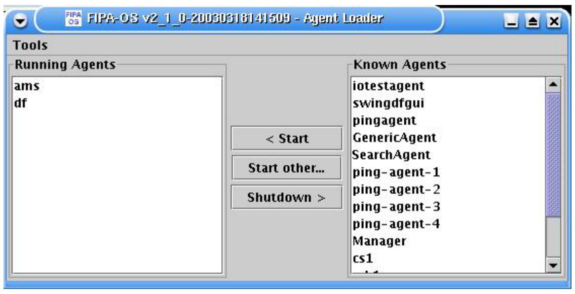

Figure 6: Agent Loader

Figure 7 shows the DF agent user interface. When the DF starts, it creates a graphical user interface to enable registration of the DF with other DF's and vice-versa. This allows creating a chain of searches. The DF interface it is only activated by double-clicking on the name of the DF agent.

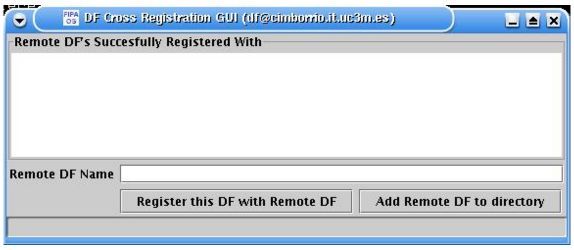
DF Cross Registration GUI (df@cimborrio.it.uc3m.es)
Remote DF's Succesfully Registered With
Remote DF Name
Register this DF with Remote DF
Add Remote DF to directory

Figure 7: Directory Facilitator GUI.

Finally, the FIPA-OS platform comes with an agent implementation that gives the user a way to interact with DFs. It is possible to subscribe to a home platform DF and the gui will show the agents registered in the corresponding home DFs. It is also possible to retrieve information about the agents subscribed to remote DFs as well. Finally, the tool also supports the visualization of DF's agent descriptions. Figure 7 shows the main window.

Added new DF: df@cimborrio.it.uc3m.es

Figure 8: Swing DF GUI main window

It is possible to download a tutorial for learning how to program your own agents from the official web page of FIPA-OS [13]. Through an incremental learning, from the simplest agent which does nothing to a simulation of auctions in a marketplace made of sellers and buyers, the tutorial enables the user to get the fundamentals of agent's development in the FIPA-OS platform.

4.2. MAB-CMGS implementation

We are in the early steps of the implementation effort that implies the MAB-CMGS system. We hope to have a first functional version in a couple of months. Although we have talked about the role of the King agent as being the centralized figure who actually executes the chess moves and decides which move seems to be the best, these actions could be performed by any other type of agent. We think that the figure of the King is the best one to represent the head of the army and that it fits very well with the mythology surrounding this ancient game. On the other hand, if we implemented the King agent as it is been stated, we would have to construct an agent with two totally different behaviours, one as being just another piece of the game, and the other one as being the commander of the troops. Thus, we think that it is better to introduce the figure of the "God" agent, which will take over the functions of the analytical search and will perform the actual moves based on the information provided by the rest of the agents, including the information provided by the King agent which will stand up just as any other agent, with it is local perception of the environment and its own evaluation of its moves.

5. Conclusions and Future Works

This paper introduces the novel idea of having into account local decisions and a local perception of the environment in order to narrow the search space of a program that wants to play chess in a similar way human beings usually do. The objective surrounding the idea is to utilize local evaluations to improve the efficiency of the minimax search. This is achieved by the reduction of the number of branches that the search algorithm has to take into consideration. Hence we construct which has been call a Multi-Agent Based Chess Move Generator System.

Human beings intuitively select a couple of interesting moves and later start a deep thinking to evaluate which of the selected moves is the best one. Human beings do not examine all possible moves, because we are "intelligent" enough to automatically discard those moves that are not useful at all. Although previous research in this field had concluded that machines can not emulate this human behavior [15], we rely on the key concept stated in [1], which is that strategies can emerge from tactical behaviors.

We are positive that commonly used search techniques are not enough to make a very strong chess player program. This is due to the following facts:

- Evaluation functions are heuristics: we can not cope with the complete search tree in a reasonable amount of time, so we are force to guess which position seems better.
- Time constrains: we are bound to the time we have to play the game, so it is a waste of time analyzing all plies of a given position, but we are force to do so in order to not eliminate a (possible) winner subtree.
- Machines play overly cautious games: the minimax principle states that both players should make their best moves. That implies that machines do not take risks to deceive the opponent, though most of the times the opponent would be teased. If the computer has to decide between either making a move that forces a draw, or making a (bad) move that could let it win if the opponent made another mistake, it will always play conservative.

On the other hand, we combine both analytical and behavioral methods to play chess with the hope that it will make the computer player a more robust opponent. We do not have analytical results yet but we think that a combined solution has the following benefits:

- Careful design of simple behaviors and their interactions can often produce systems with useful and interesting emergent properties [2]: we use these "expectations" to feed the search algorithm to narrow the set of possible moves to be considered.
- Improving the response time and obtaining a kind of adaptation to the game style: in fast games we could rely on the best move the agents select, and in slow time controls we could dynamically change the number of moves to take into account, especially in complex positions.
- Human player behavior equivalence: local agent decisions can make computers play as human players would do, using tricks and playing risky even if the selected move is worse than playing for a draw. This means that the influence of the environment can guide to a suboptimal strategy, but that is what humans players normally do.

It is worthwhile remembering that we do not rely either on analytical-only based tools or on agents based decisions on their own. We try to get the best of both worlds to improve chess programs where they look weaker than humans.

After releasing the first stable implementation, we plan to test the MAB-CMGS with both human and computer players. We are sure that the local agent decisions and some other variables will have to be tuned, as it is not realistic to think that the first implementation will beat Kasparov. We do not aim to make that, instead we are focused on studying the combination of local decisions in a restricted perceived environment with a global strategy driven by an agent with a more complete view and knowledge of the domain. We would also like to investigate the influence of local decisions in the definition of global strategies in other domains (as social domains), which are often analyzed "a posteriori" and thus are likely to be inaccurate. [1] demonstrates that most of the plans that we use are partially built by us and partially built by our immediate environment. We conclude that the environment should also be considered when playing chess games and we further refine the work begun within [1].

References

[1] Alexis Drogoul, "When Ants Play Chess (Or Can Strategies Emerge From Tactical Behaviours)", 1995.

[2] R. A. Brooks, "Elephants Don't Play Chess", USA Robotics and Autonomous Systems (1990), MIT Artificial Intelligence Laboratory, Cambridge,

[3] Brief chess history reference: http://www.cs.nott.ac.uk/~gxw/chesshis.html

[4] Good chess programming article with a Java example implementation: http://www.GameDev.net, "Chess programming Series".

[5] Minimax search: http://www.xs4all.nl/~verhelst/chess/search.html

[6] Alpha Beta pruning: http://www.cs.mcgill.ca/~cs251/OldCourses/1997/topic11/

[7] Iterative Deepening Alpha Beta pruning: http://www.seanet.com/~brucemo/topics/iterative.htm

[8] ChessBrain project: http://www.chessbrain.net/

[9] FIPA Abstract Architecture Specification. FIPA00001. Foundation for Intelligent Physical Agents (2000).

[10] RMI: http://java.sun.com/products/jdk/rmi/

[11] CORBA: http://www.cs.wustl.edu/~schmidt/corba.html

[12] FIPA-OS Developer Guide: http://www.emorphia.com/research/about.htm

[13] FIPA-OS Tutorial http://www.emorphia.com/research/about.htm

[14] S. Poslad and P. Charlton, "Standardizing Agent Interoperability: The FIPA approach", Multi-

Agent Systems and Applications, 9th ECCAI Advanced Course, Prague, Czech Republic, July 2001.

[15] P. W. Frey, “An Introduction to Computer Chess”, in “Chess Skills in Man and Machine”, Springer-Verlag, New York, 1977.

[16] Deep blue match:
http://www.research.ibm.com/deepblue/