Monte-Carlo Techniques: Applications to the Game of the Amazons

by

Julien KLOETZER

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Supervisor: Professor Hiroyuki Iida

School of Information Science
Japan Advanced Institute of Science and Technology

Preface

When I arrived here in Kanazawa in Japan three years and a half ago, nothing could have convinced me that things would change so much that I would like to stay here longer. My first wish was to come back in France after everything would be over, and now I'm hoping nothing will be really over. Life can change that much, and I think that is called growing. I hope I have grown during these these years, even if it is a little bit late. And anyway, although some times have been hard, I also had great times and experiences doing all this work in a country which is so different from my home country, in France.

There are so many people who helped me, encouraged me, taught me or just followed me during these past years working on my thesis that it's difficult to think about thanking everyone. It's hard because I fear I will forget people or will not thank others enough. If that would be the case, I'm sorry in advance for that and hope you would forgive me.

I would like to thanks Professor Hiroyuki Iida and Bruno Bouzy for giving me the opportunity of studying here in JAIST, getting through this thesis, coaching and helping me during more than three years. Their respective advices will surely be invaluable in the future. I would also like to thank people from my lab and in JAIST in general for their help with a good number of minor things, especially Alessandro Cincotti (thanks for revising my papers again and again), Jun'ichi Hashimoto and Tsuyoshi Hashimoto. Specifically coming from JAIST, although not from our lab, thanks also to Professor Kiyoaki Shirai and Professor Yoshimasa Tsuruoka, both for being part of my defense committee and providing me with invaluable advices, with a special thank to Professor Shirai for coaching me during a part of my time here. Thanks also to all the other people here in JAIST who indirectly or directly helped me going through this, Tim, Xavier, Mary-Ann, JC, the students from Iida lab... And I am sure I forget some. Finally, thanks to Richard Lorentz and Bruno Bouzy for having been part of my committee and taken the time to come from that far away, for your help and all the fruitful discussions we may have had during these years.

Not specifically related to this work, I would like also to thank all the people who supported me during these three years, eased my life in Japan and helped me at various moments. Times have sometimes been difficult here in Japan, so far away from home, and I am very grateful towards all of them. Some of them I already mentioned, since they helped me both with my work and my personal life. But thanks also to my parents for supporting me all this time, Laurent, Antoine, Laure, Marie-Gabrielle, Marie-Reine, Khin, Hashimoto and Hashimoto, Lauren, Tim, Sophie, Alessandro and Alessandro, Ayano, Sanae, Kais, Fabien, Mariella, Camille, Amandine, Anthony, Cecile, Julien, Jean, Nicolas, GITS and Yoko Kanno, Naoko, Shouko, Mikito, Madame Moine, Sunayama-san and lots of others. Thanks a bunch guys.

A final thanks to the Japanese government for providing me a scholarship. Although I have seen its amount reduced every year (bad time for the economy), they have been generous more than enoug and nothing would have been possible without it :)

Julien Kloetzer, March 2010

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Computers and Games

1.1.1 The grandfather of modern game programming: Computer Chess

In 1950, Claude Shannon was the first, in his paper *Programming a computer for playing Chess* [70], to mention the possibility of using a computer to play a board game. At the time, the target was the King of games, Chess. His idea got a lot of success, since after developping mostly military applications, making game programs seemed a quite natural way for using a computer. Although the near future was flooded with over-optimistic predictions [73], researchers started to work on what would become one of the longest computer-science experiments this world has seen (from Shannon in 1950 to the advent of Deep Blue in 1997).

However, although researchers were finding ingenious ways of integrating human intelligence into their Chess programs, it was soon discovered that, more than knowledge and smartness, processing power was king [74]. Thus, research towards a computer program that could beat Chess Grandmasters geared not toward having better or more intelligent algorithms but towards developing better search engines and creating more powerful machines to support them. Although this is arguably a form of intelligence, artificial intelligence was mostly toned down to simple artificial power.

Then at last, after a short loss against the world champion of that time Garry Kasparov with the machine Deep Thought, IBM researchers scored a victory against the very same Kasparov with their program Deep Blue. The program was running on one of the fastest machine existing at that time, processing up to 2-2.5 millions positions per second, and scored 3.5-2.5 against the human world champion after a six games match [22].

1.1.2 Games fall down to computers

During the meantime, Chess was not the only game to interest researchers and game programmers. Following the footsteps of Chess programs, very strong Checkers programs emerged and the world witnessed for the first time a game having a non-human world champion when the Checkers programs Chinook won the World Man-Machine championship in 1994 [68]. In a way, progress was quicker because of the lower complexity of the game, and they were so much quicker that the game of Checkers was solved a decade later in 2005 [67, 66], indisputably making Chinook the strongest programs in the world.

Other games fell down rapidly to the power of computers during the same decade, like the game of Othello which saw Logistello scoring a victory over human world champion Takeshi Murakami [19]. Stochastic games like Scrabble [71] or Backgammon [77] also benefited from the increasing power of computers. Although the search techniques and playing algorithms widely differ from game to game and from deterministic to stochastic games, the increase of computer power has been a valuable asset to defeating human world champions for almost any kind of board game.

1.1.3 Computer games nowadays

After the fall of several human Grandmasters over a few years, interest in computer programming shifted towards other kinds of games. Games like Poker, Go, Shogi or Amazons have attracted the computer-game community. Go, Shogi or Amazons only represent a new step in terms of complexity after Chess, while games like poker introduce aspects of randomness, bluff and opponent modeling which pose new challenges to the community. The run to power has also changed: there are strong indications that Moore's law will not be able to follow its first tendency any longer soon, but at the same time the use of multiple processors for the same computation, also known as parallelization, is a natural follow-up to the physical limit to Moore's law. Nevertheless, parallelizing requires specific kind of algorithms, one cannot only rely on waiting to get better, faster or larger processors and memory. This makes game programming more than ever a natural testbed for various algorithms, techniques or hardware, whether this stays in the field of artificial intelligence or touches other fields.

Some have been far enough to the point of creating games that should be naturally challenging for computers. That is the case of the game Arimaa [75]: its main challenge is obviously its branching factor, which has been estimated to 17.000 on average. Even if games of Arimaa are shorter than, for example, Go or Shogi games, the branching factor is such that relying on brute-force alone is near to impossible. Also, the game is designed so that there is no opening theory, rendering useless any potential Opening-Book. The same way, endgame databases cannot be used either. The Arimaa challenge [1] has been up for several years now, but humans still dominate the game.

Finally, game programming found itself lots of applications in games focusing on entertainment. Video games now represent a huge market worldwide, and programming AIs (more than just a concept, Artificial Intelligence has been turned into a noun, representing any kind of computer program displaying some sort of intelligence) has become a very important part of creating a video game. Customers do not ask any more to the AIs to be strong, but also entertaining, realistic, fun... Various customers ask for various features. Machine learning is now used to learn players' behaviors and adapt to them, scripts are created to make in-game characters feel more human... The task is now endless, but Turing's test is not yet to be solved.

1.2 Monte-Carlo methods

Monte-Carlo methods are a generic name for methods or algorithms using stochastic simulations. They are used in various fields, including mathematics and physics. In 1990, Abramson proposed a way to adapt Monte-Carlo methods to game programming [5] as a possible replacement to traditional minimax search with Alpha-Beta [47]. We will present in the next sections the main idea of Abramson's work as well as some recent improvements of Monte-Carlo.

1.2.1 General principle

The main idea behind Monte-Carlo for games is to play in a given position the move leading to the best average result, that is the move returning the highest probability to win the game. This goal, although easy to understand and human-like, is not that easy to achieve. In the work presented by Abramson the algorithm is the following:

- Given a position, fix an important number of simulations N.
- For each possible move in the given position, simulate N random games, possibly helped by some knowledge, and average their results. These random games are also named simulations, playouts or rollouts.
- Finally, play the move with the best expected result.

Given unlimited time, this algorithm gives us the result we want: thanks to the law of large numbers, the average result computed through simulations will converge to the expected average result of each move. With finite time - that is with finite N - this algorithm will give us an estimation of the average result of each move, from what one has only to hope that it is as close to the real average as possible.

Given its stochastic aspect, this algorithm has already been used for non-deterministic games like Poker [11], Scrabble [71] or Backgammon [77], where it usually gave honorable to very good results. However, as unnatural it may seems, it is also possible to apply this algorithm to deterministic games, exactly in the same way as was presented before. In this case, the stochastic aspects of the random games permits the program using Monte-Carlo to explore the search space; obviously, this is done mostly randomly so every simulation in itself is not very informative, but the sum of all them, in the end, is. Brügmann was the first to apply Monte-Carlo methods to deterministic games in 1993 with his Go-program Gobble [18].

There are several ways to improve Monte-Carlo. They often consist in one of two things:

- The first consists in changing the choice of the move for which to launch a simulation. Instead of evaluating each move through a huge number N of simulations, we can focus on moves looking more promising and ignore those which value is worse even after a few simulations [14]. This kind of improvements is very often inspired by algorithms used to solve the N-armed bandit problem [9].
- The second consists in changing the simulations used to make them pseudo-random. Instead of playing random moves until the end of the game for each simulation, it is possible to add knowledge and heuristics to bias the moves, still keeping a certain degree of randomness in the move selection process [16, 11, 77].

1.2.2 Monte-Carlo Tree-Search and UCT

Monte-Carlo Tree-Search (MCTS) is a term which appeared recently, and got its first recognized work in early 2006, when the Monte-Carlo Tree-Search Go program Crazystone won the 9x9 Go tournament of the eleventh Computer Olympiad [35]. Beyond the simple integration of Tree-Search into a Monte-Carlo framework, MCTS is a complete re-work of the traditional minimax framework in which the processes of backing-up values and node expansion as well as the evaluation and exploration paradigms are completely different. The basic idea is to apply techniques used to bias exploration towards more promising moves into a Monte-Carlo program at several levels of a game-tree, and not only at the root node.

The UCT algorithm [48], or UCB for Trees, gained popularity as an MCTS algorithm among the gaming community. It consists in using the UCB algorithm, an algorithm used to solved the N-armed Bandit Problem, at several level of the search in an MCTS based program. The UCT algorithm possesses several good properties for computer games, among which is a good balance between exploration of new variations and exploitation of previous results, and is proved to converge to the traditional minimax values of a game-tree if given unlimited time.

The game of Go, one of the remaining board games not beaten by computers, has greatly benefited from the appearance of MCTS. One of the main challenges for this game, alongside with a large branching factor coupled with a long depth, was the creation of a good evaluation function [15]. This task is necessary to be able to apply traditional Tree-Search techniques. However, MCTS and Monte-Carlo in general do not require any evaluation function to work, which made it an ideal framework to try for the game of Go. The fact that Go is a territory game in which pieces do not move, making the board more static than in other games, is very often pushed forward to explain the success of the MCTS approach for programming Go, alongside with the fact that it does not require an evaluation function. Using such techniques, Go programs play nowadays at a strong level on standard board sizes and can even match professionals in handicap games or on 9x9 boards [39, 4, 50].

1.3 Testbed and research questions

1.3.1 The game chosen: Amazons

The game we chose to study in this work is the game of the Amazons. This game is a young deterministic game with perfect information, a member of the class of territory games. While it is not as well known as some of its cousins like Chess, Checkers or Go, the game of the Amazons displays several interesting features [69], notably a very large branching factor which makes the use of traditional search techniques unappealing, even if some strong evaluation functions already exist.

With the goal of studying Monte-Carlo techniques, the game of the Amazons is also an intermediate between the game of Go and other kinds of games. It possesses a strong territory concept, like the game of Go, but at the same time has fast moving pieces, similar to those used in games such as Chess. While Monte-Carlo techniques show good results for the game of Go, their application to games with moving pieces seems more difficult. The study of MCTS on the game of the Amazons should provide insights into choices that have to be made to transpose MCTS techniques to a more general family of games.

1.3.2 Research questions

The game of Go, nowadays the main representative of the class of territory games, has greatly benefited from the development of Monte-Carlo techniques for computer games. On the other hand, and contrary to the game of Chess, classical methods such as minimax with Alpha-Beta and other traditional improvements have never really worked for Go. The playing level of such programs is usually close to that of average amateur players. The difference is perceptible between both games: on the one hand we have a game with fast moving pieces and in which the goal is very tactical (capture the opponent King), while on the other hand we have a game with non moving pieces flooding the board with a much more strategical aspect and a continuous but slow fight over territory. It could be easy to make a parallel between the use of minimax and of Monte-Carlo with the categories in which the game stand. Now, we also have the game of the Amazons, which takes a little bit from both worlds: the game uses fast moving pieces and it has been shown that minimax can give birth to strong programs, but at the same time the game pace is nearer to the one of Go and, like the latter, Amazons is fundamentally a territory game. From this observations we can raise the following question: **Is it possible to build a strong Amazons playing program using Monte-Carlo Tree-Search techniques ?**

Now, if the answer to the previous question would be positive, it remains to see how a Monte-Carlo program playing Amazons could be improved. Minimax and Monte-Carlo Tree-Search may both be wonderful and intuitive techniques, a game program using them has never really been far without improvements that could be either specific to the technique or to the game. Again, the game of the Amazons stands in an intermediate position. Lots of improvements have already been done to improve the level of Monte-Carlo Tree-Search based programs, but mostly for the game of Go. **Could such techniques used for the game of Go or other games be applied for an MCTS program playing Amazons?** Also, since work has already been done to improve the level of traditional minimax Amazons programs, **could traditional improvements for the game of the Amazons be used in a Monte-Carlo based program?**

Finally, while improving its general playing level is a necessary step to get a strong level program, to be able to beat the top one's program must usually possess specific features to deal with parts of the game for which the most knowledge can be gathered: the opening game and the endgame. The game of the Amazons makes no exception. For the former, the main and easiest answer is Opening-Book. Simply put, an Opening-Book for a game is a collection of annotated positions that can be reached during the first few moves of the game. Since, traditionally, the starting position of the game does not change, it is possible to pre-compute some form of knowledge that a program can use either to gain time - and thus get more computational power during the mid-endgame - or just to make it play better moves. Now, by comparing old fashioned minimax based programs to Monte-Carlo based programs playing the same game, whatever it is, one usually quickly notices that the playing style of Monte-Carlo based programs is different. To quote the Chinese Go professional player Guo Juan (5dan) when she saw the Go program Mogo play at the 12th Computer Olympiad in Amsterdam, *Mogo plays romantically.* Indeed, Monte-Carlo programs traditionally play more loosely and with more freedom than traditional programs which are designed with lots of knowledge in mind, and even more when there are many possible options in the game. The game of the Amazons being such kind (with a very high branching factor), we can expect a Monte-Carlo program playing Amazons to have such a style, or at least a different one from traditional programs. This makes the use of traditional techniques used to create Opening-Books look dangerous: while an Opening-Book can improve the level of a program, it can also hinder it by leading it to situations that the program does not well fully understand. Then, **is it possible to directly use standard Opening-Books of more traditional programs for Monte-Carlo based programs? If not, is it possible to adapt them to a different playing style? Is it possible to design more specific Opening-Books or methods to create them for Monte-Carlo based programs?**

Finally, on the point of the latter, endgames, Monte-Carlo based programs are often said to be less precise than their counterparts using traditional minimax search. Indeed, Monte-Carlo programs strength often comes from a good understanding of global positions, but the stochastic elements which form the core of such programs usually hampers them for more precise tasks such as playing skillfully to capture enemy pieces, create living positions in the game of Go or, more generally, playing endgames. Endgames are a very important part of the game of the Amazons and quite easy to play the wrong way: they usually involve playing several subgames in parallel. However and contrary to simple intuition, the move looking the biggest in a situation is often not the best considering other situations on the board. Now, **what about Monte-Carlo for Amazons? Is MCTS able to handle precise playing like one finds in endgame situations? Are specific improvements needed?**

1.3.3 Outline of the dissertation

Introduction (the present Chapter) and conclusion (in Chapter 6) set aside, this thesis consists of four chapters.

We will present in Chapter 2 the game of the Amazons. After a brief explanation of the rules, we will explain some strategic concepts of the game to help the reader in his/her understanding of our work. We will also mention in this chapter previous studies done about the game of the Amazons as well as present existing programs and competitions.

Chapter 3 will deal with our experiments in building an Amazons playing program using Monte-Carlo based methods. We will present first to the reader how Monte-Carlo methods can be applied to create game playing engines, not forgetting their recent evolution into Monte-Carlo Tree-Search. Specifically for the game of the Amazons, we will present simple adjustments that can - or need to - be done in order to create a Monte-Carlo Amazons program, followed by an evaluation of such a program. The playing level of the latter being very weak, we will propose a way to integrate knowledge in the form of an evaluation function in a Monte-Carlo Amazons program by shortening the stochastic process which is the basis of the Monte-Carlo method, and study the effect of shortening or lengthening this process to the performance of our MC Amazons program. Since a program using this improvement gets a fairly good playing level, we will next discuss several improvements that can be added to the Monte-Carlo search engine. More importantly, we will discuss the implementation of the AMAF/RAVE heuristic and show that, while it cannot be directly used as it is for other games such as Go, with the correct implementation it gets a Monte-Carlo Amazons program to an even higher level in terms of performance.

In Chapter 4, we will study the performance of several solving and playing techniques to the task of playing well Amazons endgames. These usually consist in the sum of several subgames, each one of them being a difficult problem, thus making the whole a difficult task to handle. We will show how it is possible to adapt traditional solving techniques to play Amazons endgames well but also that, even if Monte-Carlo methods lack the tactical precision which would allow them to play single subgame problems perfectly, they have a better strategical sense which gives them an advantage in playing well in a combination of subgames.

Finally, based on the observation that our Amazons program CAMPYA attained a good playing level, Chapter 5 will present our research towards building a correct Opening-Book for it. While some traditional automatic Opening-Book building methods exist, we will show that, even if they can be used to create reasonable Opening-Books for minimax based programs, the direct use of such Opening-Books with a Monte-Carlo based program leads to disastrous performance. We explain this by a difference in playing style between the two classes of programs. We then propose several adaptations of Monte-Carlo Tree-Search techniques to create Opening-Books, mostly based on the Meta-MCTS paradigm, consisting in interleaving two levels of Monte-Carlo search (or Tree-Search) to include some very high level knowledge. We finally show that Opening-Books created using such techniques are much more adapted to the playing style of Monte-Carlo based programs.

The conclusion of our work, as well as the presentation of some potential future work, follows in Chapter 6.

Chapter 2

The Game of the Amazons

The game of the Amazons, or in Spanish *El Juego de las Amazonas*, is a two-player board game that was created in 1988 by Walter Zamkauskas of Argentina. It is a perfect-information game, meaning that both players know at any instant everything about the state of the game. It is also a deterministic game: there is no randomness involved in it in the form of cards, dices etc. The game is also classified in the family of territory games, like the game of Go. This last point will be discussed in Section 2.2.

The next sections will present in turn the rules of the game (Section 2.1), some strategic concepts of the game (Section 2.2), several reasons why the game of the Amazons is worth studying (Section 2.3) and finally existing programs, research and activities around the game (Section 2.4).

2.1 Rules of the game

The game of the Amazons, or just simply Amazons, is usually played on a 10 by 10 square board using 4 pieces for each player. These pieces are called Amazons. To play the game, one also needs a set of other pieces that shall be used to represent arrows on the board. There are several variants, usually with smaller boards but the same number of pieces for each player, and the game consisting of the same rules but played on any board and with any number of pieces is generally called *Generalized Amazons* [37].

At the beginning of the game, the Amazons are placed like illustrated in Figure 2.1. Unless you are playing a variant, no arrows are yet placed on the board.

There is no real common agreement for the color of the pieces, so we will consider in this work two players White and Black (light gray and dark gray on the pictures) with White moving first.

Each player takes turn to move. One move consists of two steps:

- First, the player to move moves one of his Amazons the same way a Queen would move in Chess; that is, any number of squares, in any direction.
- Then, from the landing square of this Amazon, he[1] chooses a square that could be reached by a second Queen move, that is again any number of squares away in

[1]The reader should note that our player could be a man as well as a woman, needing us to write *he or she*. However, for reasons of simplicity and readability, we will only write *he* from now on, without regard for the player's gender.

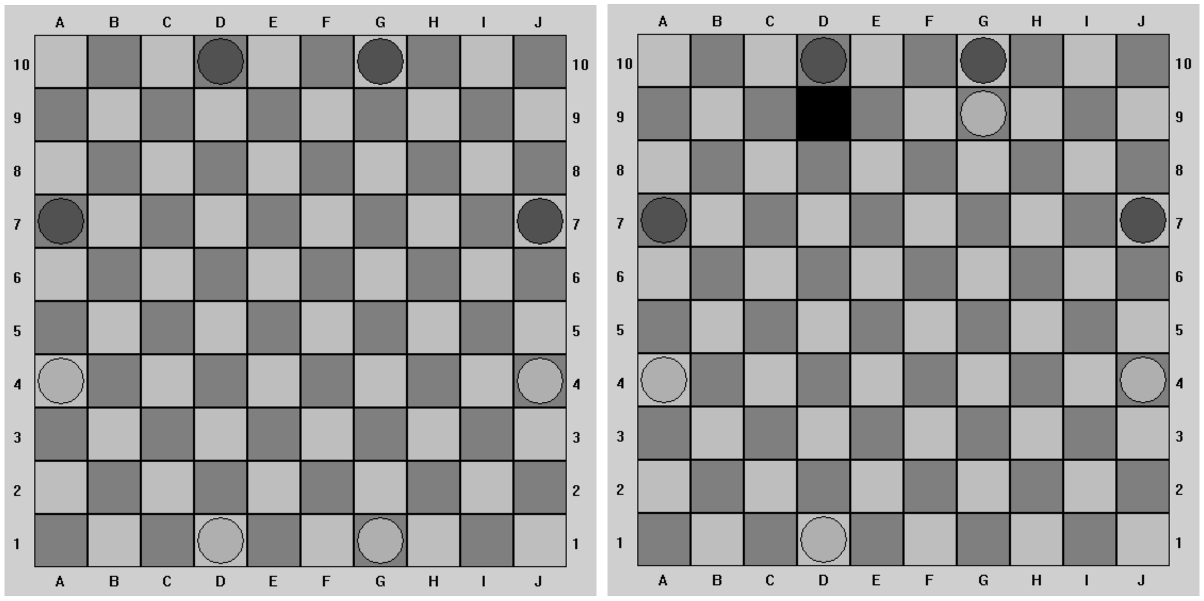


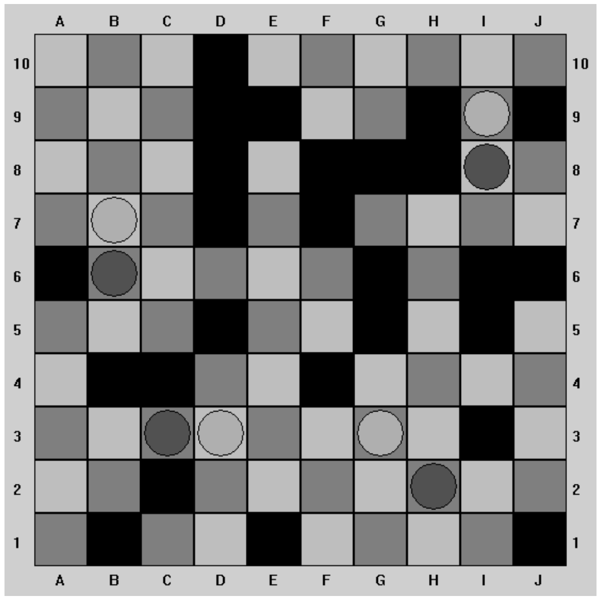
Figure 2.1: Left: Classical Amazons starting position; Right: A move from the starting position (G1G9-D9). The darkened squares is the burnt square.

any direction, and places an arrow on it. This choice can be made in the opposite direction in which the Amazon has just moved. We then say that the player *shoots an arrow* on that square. The term *burn* shall also be employed to describe this step.

An example of move from the opening position is given in Figure 2.1. From this moment and onward, the square on which the arrow is placed blocks every further Amazons move or arrow shot on or over it.

The two steps of the move are mandatory, and no pass is allowed. It follows that, since one square of the board will be burnt on each player's turn, at some point one of the players will be unable to move any of his Amazons. When that is the case, this player loses the game, while his opponent wins it. In other words, the last player who is able to move wins the game. At this point, it is usually agreed that the score of the game for the winning player is the number of moves he would be able to play after his opponent's pass.

While the game is supposed to be played until one player passes, the game usually reaches a point where both players move their Amazons without interacting any more with each other. Each player has then Amazons inside their own territories (see the right of Figure 2.2 for an example of such situation). Unless presents itself the case of defective territories (discussed in Section 2.3), the player who has the biggest territory will win because he will have more squares to burn before being forced to pass than his opponent. Human players usually then agree to stop the game, and the score is then the difference in size between territories. If both players disagree on it, it is possible to just continue the game until one player has to pass or both players finally agree on the score.



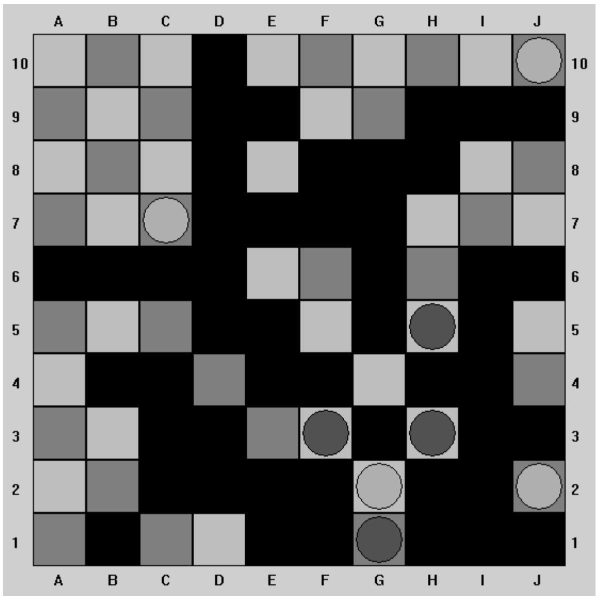


Figure 2.2: Left: An Amazons middle game position; Right: An endgame position; even though the White Amazon on G2 is still connected to its opponent's Amazons, it is entrapped thus all territories are isolated.

2.1.1 Integration of a Komi

Komi is a concept that has recently been incorporated for the game of Go, but is sometimes used in other games under the same name. It refers to a bonus given to the second player of a game when he is at a disadvantage. For example, since the first player in a game of Go (on standard size 19x19) has an advantage recognized by most human players, his opponent is usually awarded 6.5 or 7.5 points at the end of the game, depending on the ruleset.

Since Amazons is also a game which victory is tied to a score, it should also be possible to introduce a Komi. While the Komi in the game of Go also serves as a tie-breaker, there is no such need for the game of the Amazons as there is no draw. However, the fact remains that one has to choose between two possible implementations of this Komi rule:

- With a Komi of N points, at the end of the game, give the second player a bonus of N points (similar to Go)
- With a Komi of N points, allow the second player to pass freely N times at any point of the game. Only after those pass are used the second player would lose because of a pass.

Now, even if these rulesets are similar, the strategy they involve are different as is the treatment of zugzwangs (see Section 2.3.2 for a word on those). Also, while human players and statistics recognize that the first player has a clear advantage in the game of Go, there is no such evidence for the game of the Amazons. As such, most games are played without Komi, and the work presented in this dissertation will also consider no Komi to determine the score of games.

2.2 Strategic concepts of the game

Despite the apparent simplicity of its rules (which do not allow any pathologic case like there exist within some various rulesets of the game of Go, to name another game with simple rules), the game of the Amazons possesses several interesting strategic concepts. Three of them, territory, accessibility and mobility, are usually agreed on by most of the players, while a fourth, Amazons distribution, is more dubious. We will present these concepts in the next sections.

2.2.1 Territory

Territory is one of the most important concepts of the game, and what classes it into a larger game family. In the game of the Amazons, it usually becomes relevant near the end of the game, but could be sometimes relevant before, depending on the style of the player.

A player's territory can be defined as such: a set of connected squares of the board that only that player's Amazons have access to. That is, in a given situation, a square is part of a player's territory if his opponent cannot reach that square with any of his Amazons, even given an unlimited number of potential moves (since any arrow shot can be made onto the square an Amazon just left, it is even possible to just consider the path from an Amazon to a square in terms of moves and forget the shots).

In the endgame, when players do not interact with each other anymore, each player's territory becomes a pool of squares that he only will be able to burn. The bigger the territory the higher the number of burnable squares, and so the more moves the player has at his disposal. It follows quickly that, as is the case for any territory game, the player with the bigger territory will win the game (pathologic cases will be discussed in Section 2.3). In a computer-programming oriented way, since we need to differentiate between a win by 0 points and a loss by the same score, half a point is usually added to such scores. The latter scores then become respectively a win by half a point and a loss by half a point. For example, the position on the right of Figure 2.2 has six territories: three for White, of respective size (number of empty squares) 14 (top right), 1 (center) and 4 (bottom), and three for Black of respective size 13 (top left), 2 (bottom left) and 4 (bottom right); however, the bottom right territory has a deffect, meaning that in this situations Black will have to waste one square of it, thus making the effective score given by this territory 3. This makes $14 + 1 + 4 = 19$ vs $13 + 2 + 3 = 18$ with White to move first, meaning White will win by half a point.

Finally, since they are of no use to anyone, any square which is completely isolated from the rest of the board by arrows and thus will not be of any use to any player for the rest of the game is usually considered as neutral territory.

2.2.2 Accessibility

The accessibility is a concept somehow similar to the one of territory. It was introduced quite early in the days of Amazons game programming [42] and we can affirm with some confidence that it is nowadays used in every Amazons program.

Although the definitions provided by Hashimoto *et al.* in [42] and Lieberum in [51] slightly differ, the concept is very similar. We will in this work adopt the definition from

Lieberum: the accessibility of a player's Amazon A to a square of the board S is defined as the minimum number of moves needed for that player to place the Amazon A onto the square S. The player's accessibility for that square is then defined as the minimum of the accessibility for that square for all of his Amazons. An example of player's accessibility is given in Figure 2.3. If one player cannot access a square with any of his Amazons, his accessibility for that square is set to infinite, while it is set to zero for squares on which his Amazons are placed.

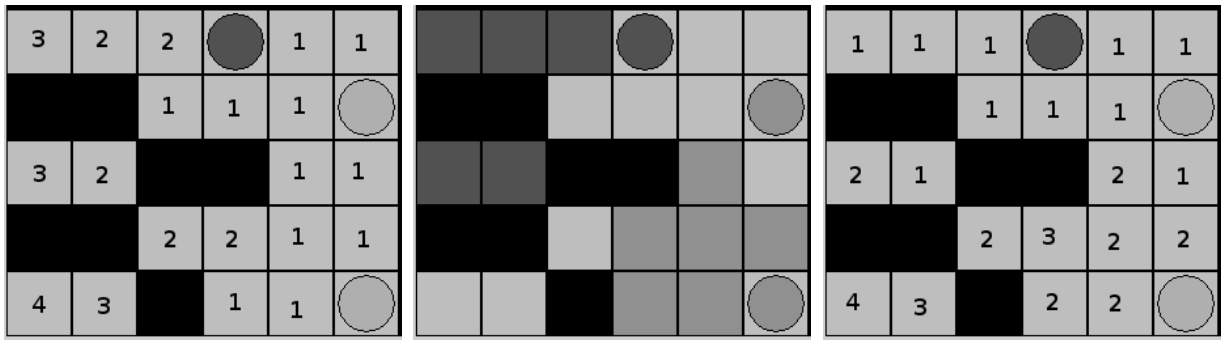


Figure 2.3: Left: accessibility for White; Right: accessibility for Black; center: territory map resulting of the comparison of accessibilities on every square.

Accessibility alone is not very informative. However, it becomes useful when we compare two player's accessibility values on the same square: if player A has a better (that is, smaller) accessibility than player B for a given square S, that means that player A should be able to place an Amazon onto square S before player B. If both accessibility are equal, the player to move has an advantage.

In the early and mid-game, having a better accessibility for a given square is important strategically, because it means that the player has a better chance to control (place an Amazon onto) that square. For some specific squares of the board - usually squares that give access to a big territory as defined in Section 2.2.1 - having better accessibility is a very valuable asset. Near the endgame, comparing both player's accessibility values on every square of the board provides a fair approximation of the territory of both players: having a better accessibility onto a square means that this square is more likely to become part of our territory than of our opponent's. Also, any square for which a player has a finite accessibility but not his opponent is part of that player's territory (although it does not mean that this territory will be accountable at the end of the game - see Section 2.3 for the case of defective territories).

2.2.3 Mobility

Mobility, albeit being an important concept of the game and quite natural to the eye of a human player, was not introduced into Amazons game programs until recently. Jens Lieberum made the community realize the importance of mobility by winning the 2002 and 2003 Amazons Computer Olympiad with his new program Amazong [53, 51].

There is no clear mathematical definition of mobility, but the main concept beneath it is to keep one's own Amazons in positions from which they can escape if threatened and/or from which they can access to other important areas of the board. This concept is different from accessibility in the sense that it does not take into account speed, but just general possibilities of movement. An extreme example is that of an Amazon which finds itself enclosed into a small area of the board near the beginning of the game. This indeed gives the Amazon access to a small territory, but at the same time makes the remaining of the board a battle of 4 Amazons against 3. This is so severe that some do not hesitate to consider such a situation as a 10 points handicap [51]. Figure 2.4 gives an example of an enclosed Amazon.

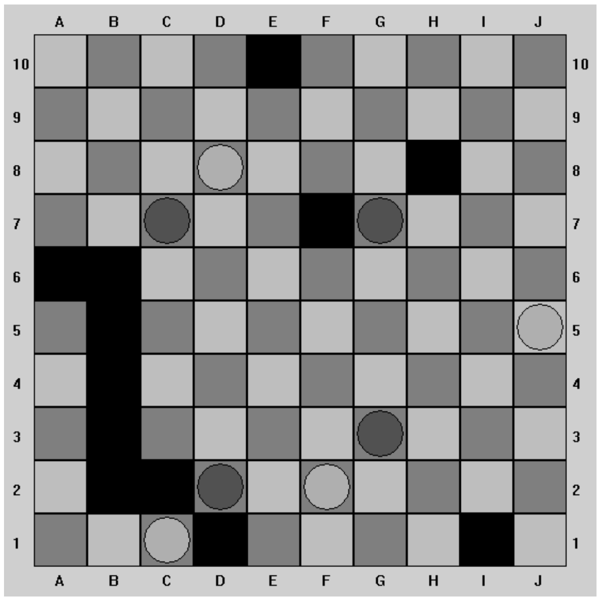


Figure 2.4: An enclosed Amazons; even though it captured a small territory, the opponent's Amazons fighting at 4 against 3 get an advantage.

A very simple way to define the mobility of an Amazon is to count the number of squares it can move to. One still has to apply a post-function to it since the penalty from having 5 possible moves only should be quite severe (in the beginning of the game at least), while there is no real difference between having 20 or 25 possible moves for a single Amazon.

2.2.4 Amazons Distribution

On the contrary to the three previous game concepts, Amazons distribution is a concept on which people do not fully agree. But it is still one of the very first simple rules to give to beginners of the game of the Amazons: *Have one Amazon in each quarter of the board.* The inherent idea is to counter one of the main weaknesses of the concept of accessibility (see Section 2.2.2): taking the minimum of accessibility of four Amazons does not take at all into account the repartition of those Amazons on the board. In the beginning of the game, the board is so open that moving Amazons around does not change that much the overall accessibility of each player on each square. However, distributing the Amazons over the board is definitely something that helps humans to get a grasp of the game, as well as computers, by avoiding bad blunders the like of *One arrow blocking three Amazons at the same time.* By distributing his Amazons, most people usually agree that it becomes more difficult for the opponent to thwart one's plans.

However, there are definitely some situations in which this rule should not be applied or could be ignored. A good example was given in the third match opposing the program Invader to the program 8 Queens Problem in the Amazons tournament of the 13th Computer Olympiad in Beijing. The position given in Figure 2.5 was attained after only 14 moves in the game and see Invader, as Black player, using no less than three Amazons to capture only one of his opponent's Amazon. The resulting position, while good in terms of mobility (Invader just knocked out one of 8 Queens Problem's Amazon into a 3 squares territory), is very bad in terms of Amazon distribution since three Amazons are grouped together. Despite that, Invader won this game, strongly helped by the fact that he got one of his opponent's Amazon trapped this early in the game [55].

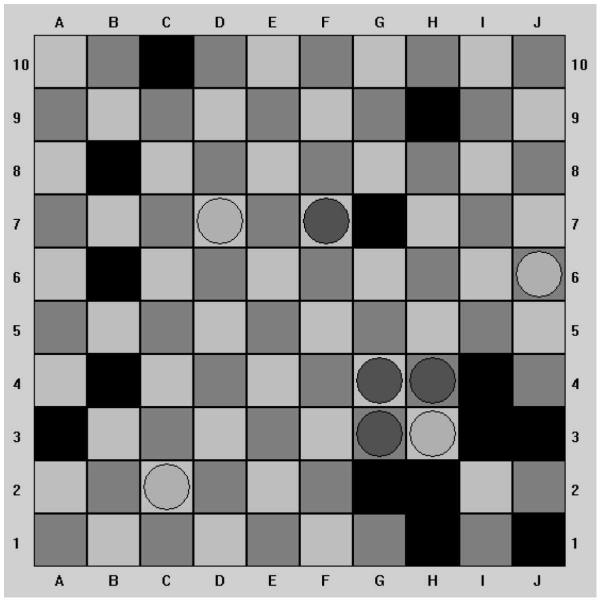


Figure 2.5: Three Amazons to capture a single one; the capture is of great benefit, but the placement of the Amazons is quite bad.

2.3 A good testbed for game programming algorithms

Despite being a game not so entertaining for humans because mostly of his huge number of look-alike moves, the game of the Amazons has several interesting aspects making it a worthy subject of research. We will present in the next sections several of these aspects.

2.3.1 The Amazons branching factor

The game of the Amazons is indisputably one of the board games with the highest branching factors. A comparison to other classical board games is given in Table 2.1.

Now, while it is true that the search depth is limited (92 at maximum for the traditional version of the game) and that the branching factor decreases quite rapidly after having increased for a few moves from the original position, it is still well over 1000 moves for the first 10 moves and usually over several hundreds during the middle game [10]. It is also true that some good evaluation functions exist, but to cope with such a high branching factor, one definitely has to find ways of selecting more promising moves, pruning parts

Game	Number of moves in the initial position	Average branching factor	Game-tree complexity	Reference
Checkers	7	2.8	10^{31}	[7]
Othello	4	10	10^{58}	[7]
Chess	20	35	10^{123}	[7]
Shogi	30	80	10^{226}	[56]
Go	361	250	10^{360}	[7]
Amazons	2176	>400	10^{165}	[10]

Table 2.1: Number of moves from the starting position and average branching factor for several games

of the game-tree or rapidly evaluating positions. This makes of the game of the Amazons a good testbed for Selective Search techniques.

However, compared to games such as Chess, the game of the Amazons possesses some good properties making it easier to analyze. Among those, the fact that the game is converging, that the game-tree is an acyclic graph, or the fact that the game cannot have draws; all these facts help both the person building a game program or the researcher trying to solve the game.

The game of the Amazons has also be proven to be part of the NP-Complete class of problems [37], while simple Amazons subgame solving is NP-equivalent [20].

2.3.2 Defective territories and zugzwangs

When considering endgames for the game of the Amazons, two situations require a special kind of attention. The first one is the case of defective territories: while it is true that, usually, the number of moves one can possibly make inside a given territory is equal to the number of empty squares into that territory, this is not always the case. Several configurations are such that a player cannot fill all the squares of his territory with the Amazons at his disposal. An extreme example of such territory is given in Figure 2.6.

Muller *et al.* investigated in [59] defective territories for the game of the Amazons and proposed a way of building a database for these. But while they are more often than not small, unlike the one presented in Figure 2.6, they are not that rare in game. This means that one should be careful when approximating the size of a territory with its number of empty squares. Fortunately, they are also easy to recognize, since they always involve some kind of tortuous maze in which a single Amazon just has a single way to move.

While not specific to the game of the Amazons, zugzwangs are also a problem one has to deal with when playing Amazons. Zugzwangs refer to situations in which a player would better pass than move, that is that he would win more (lose less) material or score if he can just choose to pass rather than moving. Examples of zugzwangs are given in Figure 2.7.

The first case of zugzwang of Figure 2.7 is pretty straightforward: the first player to move will let his opponent take some more territory, and the loser of the zugzwang is the player who has the smallest territory on other parts of the board.

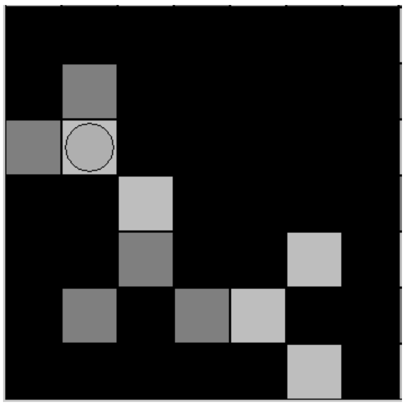


Figure 2.6: A highly defective territory; while it has nine empty squares, it is only possible to make 5 moves inside this territory.

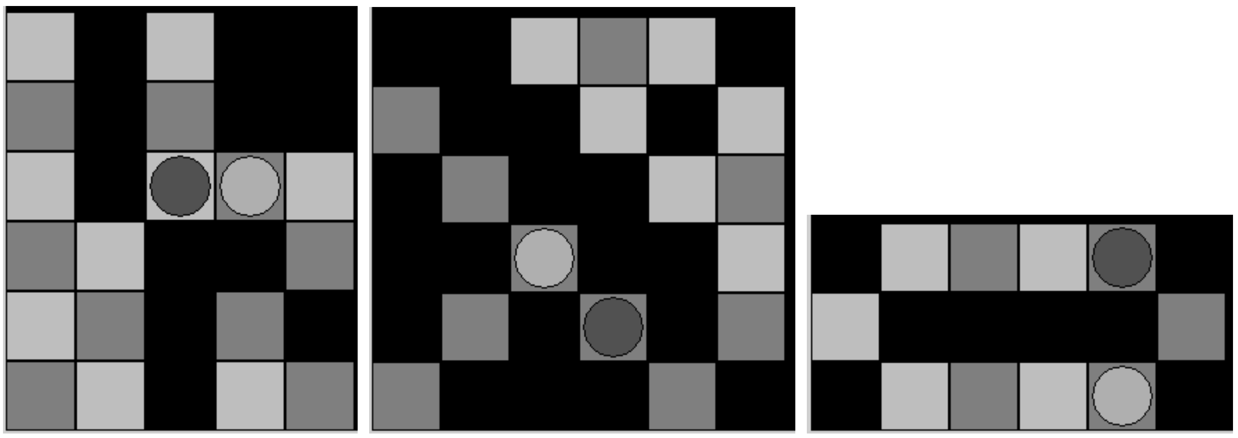


Figure 2.7: Left: A zugwzwang situation for both players; center: a zugzwang for one player only; right: a two-player zugzwang with a loop shape.

The second case is pretty similar, except that this time only one player has something to lose from this situation (White in this case). His opponent can forget about this situation, while the losing player should try to get a bigger territory on other parts of the board to win the zugzwang.

Although not rare, positions corresponding to these two cases are mainly pathological and not seen very often in real games. A variation of the first kind of zugzwang is represented in the third case of Figure 2.7. We are again in the case where both players would lose by playing first, but this time, even if the position is symetrical, the first player to move will not even get half of the existing territory. There are also some other kinds of topologies in which it is possible to move several times while the opponent passes, which gives them an outcome more complicated to compute. These cases are seen more often in games than the previous ones.

2.3.3 Amazons endgames: a peek into combinatorial game theory

The game of the Amazons, combining aspects of both piece moving games and territory games, is also a member of another family of games: the combinatorial games. At a certain point of the endgame, the board of the game of the Amazons is usually divided into several regions that can no longer interact with each others. The main difference here with territories is that we allow these regions to contain Amazons of both players. Each of these sub-positions is a single game, and it is possible to apply combinatorial game theory [31] to find the perfect solution of the global game from the single solutions of every sub-position.

2.4 Existing programs and research

2.4.1 Research

Although not as popular as other games such as Chess or Go, the game of the Amazons has attracted the interest of groups of people who worked on specific aspects of the game, on their programs, or on both.

Hashimoto *et al.* as well as Lieberum propose descriptions of the evaluation functions of their respective programs, ASKA and AMAZONG, in *An Evaluation Function for Amazons* [42] and *An Evaluation Function for the Game of Amazons* [51]. Although both are similar, Lieberum was the first to present the real power of some aspects of his evaluation and using it won twice the Amazons tournament in the Computer Olympiad.

In parallel to that, Lorentz *et al.* treated several aspects of the search for the game of the Amazons in the articles *Selective Search in an Amazons Program* [10] and Opening-Book creation in *Generating an Opening-Book for Amazons* [43].

Several studies also treated the combinatorial and endgames aspects of the game: Müller *et al.* studied in *Experiments in Computer Amazons* [59] defective territories and zugzwangs in the game of the Amazons, while Furtak *et al.* and Buro were more focused on the complexity of the problem in their respective papers *Generalized Amazons is PSPACE-Complete* [37] and *Simple Amazons Endgames and Their Connection to Hamilton Circuits in Cubic Subgrid Graphs* [20]. On a more practical side, Müller *et al.* and Kloetzer *et al.* studied the performance of several solving methods for Amazons endgames in the respective papers *Temperature Discovery Search* [58] and *A Comparative Study of Solvers in Amazons Endgames* [46].

Finally, more recently, Kloetzer *et al.* as well as Lorentz applied Monte-Carlo methods to the game of Amazons. These research efforts are described in *The Monte-Carlo Approach in Amazons* [45] and *Amazons Discover Monte-Carlo* [54] respectively.

2.4.2 Competitions

The main computer-Amazons event takes place at the annual Computer Olympiad. Since the fifth Computer Olympiad in London in the year 2000 and with the exception of the year 2006, there has been an Amazons tournament every year with 2 to 6 participants each time. The tournament is usually played in round-robin style with no Komi [53, 83, 55, 44].

Another tournament mixing human and computer players, the Jenazon cup, was also held three times between the years 2002 and 2004 by Ingo Althöfer from Germany. The games were played on the Internet with teams usually mixing human players and several programs, from which the human players chose their moves.

Taking part in all these tournaments, 8 Queens Problem (8QP) is the main reference in the world of Amazons. It is a traditional Alpha-Beta program using an evaluation function developed by game-programmer Johan De Köning. 8QP won five of the nine Amazons tournaments held at the Computer Olympiad, demonstrating its unrivaled performance.

To the side of 8QP, programs such as Amazong from Jens Lieberum [51] or Invader from Richard Lorentz [10, 43] are other traditional Alpha-Beta programs who got honorable performance during the several tournaments. Amazong, particularly, won the two Olympiads he entered into.

More recently, a new version of Invader (MCInvader) and Campya from the author are two programs developed using Monte-Carlo Tree-Search which both performed quite well in the latest Olympiads of 2008 and 2009. The new Invader also won his two first gold medals on this occasion.

Finally, other programs including Antiope from Theodore Tegos, Aska from Yoichiro Kajihara or TAS from Yoshinori Higashiuchi and Reijer Grimbergen should not be forgotten and all got honorable performance in the several tournaments organized.

Chapter 3

Monte-Carlo and the game of the Amazons

This chapter is an updated and abridged version of

1. J. Kloetzer, H. Iida, and B. Bouzy: "The Monte-Carlo approach in Amazons", Proceedings of the Computer Games Workshop 2007, pp. 185-192 (2007).

In this chapter, we will present our study of the Monte-Carlo method (MC) for the game of the Amazons. Section 3.1 will present the basic Monte-Carlo algorithm for games and several ways of improving a Monte-Carlo based program. Section 3.2.2 will present the more recent Monte-Carlo Tree-Search (MCTS) and several possible improvements. Next, Section 3.3 will deal with the adaptation of MCTS for the game of the Amazons, and study the performance of a simple Amazons program obtained using them. Finally, in the following sections, we will discuss the integration of an evaluation function into an MCTS Amazons program (Section 3.5) as well as several other improvements, including AMAF (Section 3.7) and grouping nodes (Section 3.6).

3.1 Monte-Carlo

Monte-Carlo is a very broad term used for very different ways to solve a variety of problems: physics (simulations), games (playing engine), mathematics (computation of integral functions) and so on. While the possible application field is vast, the use of the Monte-Carlo method has almost always one of these two goals in common: it is used either to make simulations or to compute values, functions or averages of those. This also makes randomness a common factor for all of those applications, and its defining factor. In fact, the term origin is from the Principality of Monaco which hosts a good number of casinos, making the link with randomness and probabilities.

3.1.1 Monte-Carlo theory

We will give here a basic example of an application using the theories behind Monte-Carlo to compute a value: making a survey about the percentage of people watching the channel A on the television at night. The method is quite well known:

- Call a given but important number of people randomly at their home

- Ask to any of those if they are watching the channel A at night

Obviously, to know the true value of this percentage, we need to ask every living soul on Earth, which is not practically possible. But using this method, the end result will be an approximation of the true value we are searching for. This results from the law of large numbers, which simply tells us that, with a significant number of tries, the average value we are computing will tend to converge towards the true value that we are searching; here, this is the percentage of people watching channel A at night.

For the Monte-Carlo method to give a good approximation of a value, we need two main factors:

- A number of samplings (in the previous example that is the number of calls) large enough so that we can trust the results with enough confidence.
- An unbiased random generator (in the previous example, the method to choose people to call). This is arguably the most critical aspect to apply the Monte-Carlo method. In the previous example already the generator was biased, since we excluded all the range of people without a telephone at home as well as those who could not be joined during working hours.

As long as these two points are taken in consideration, it is possible to apply Monte-Carlo with a good confidence to compute a broad number of values.

3.1.2 The basic algorithm

Abramson was the first to propose, in 1993, a way to use Monte-Carlo for playing games, with his expected outcome model [5]. This model was presented as a possible replacement for traditional Alpha-Beta search [47]. The main idea behind the algorithm is to play, in a given position, the move leading to the best average result. Now this idea of average result of a move must be defined, and a simple way to do that is to use for it the percentage of victory of this move. To compute this percentage of victory, he proposes to use the Monte-Carlo method. The way to compute this average result for a move is quite simple and elegant: we just need to launch a huge number of random games from the position attained by playing the move, and average the results of these random games. This simple algorithm is given in pseudocode in Figure 3.1.

Being a stochastic method, Monte-Carlo has been naturally applied for games which also inherently include a part of randomness, like scrabble [71], poker [11] or backgammon [77], in which it generally leads to good results (providing some specific improvements). However, that does not mean that the method cannot be used for deterministic games, and it has indeed been applied to such games, like the game of Go, in more recent years [17]. Since then and following good results for the game of Go, it has been applied for numerous other games, both deterministic and stochastic [84, 76, 78].

One of the good point of Monte-Carlo for games is that it is very easy to program and, in a way, knowledge free. With the exception of the rules of the games, which permit us to generate moves and play them as well as to evaluate endgame positions, this algorithm does not require us to include any other specific form of knowledge, be it an evaluation function, patterns, an Opening-Book or an endgame database. Of course, as we will see later, including those can be very beneficial for Monte-Carlo, but the point remains that it is possible to implement a program playing any game quite easily. For this reason,

```
function getBestMove(Position, NumberRandomGames)
  for each move m playable from Position
    averagevalue[m] = 0
    for i from 1 to NumberRandomGames
      Pos = copy(Position)
      play(Pos, m)
      while ( Pos is not ended position )
        play(Pos, random move playable from Pos)
      end while
      value[m] = value[m] + result(Pos)
    end for
    value[m] = value[m]/NumberRandomGames
  end for
  return(move m with highest value[m])
end function
```

Figure 3.1: Pseudocode for a basic Monte-Carlo playing engine

Monte-Carlo methods have also been used with great success in General Game-Playing competitions [36], the goal of which is to play a set of different games without knowing the rules beforehand.

Another good point of using Monte-Carlo for games is that the search is anytime. It is quite easy to change the code presented in Figure 3.1 not to launch N random games for each moves with a big number N, but instead play a random game for each move and repeat this process as long as there is time. Quite naturally, and thanks to the convergence property of Monte-Carlo, the longer the computation time, the bigger the number of games for each move and the more accurate the values computed. With unlimited time, Monte-Carlo will even return the best move, that is the move with the best average result.

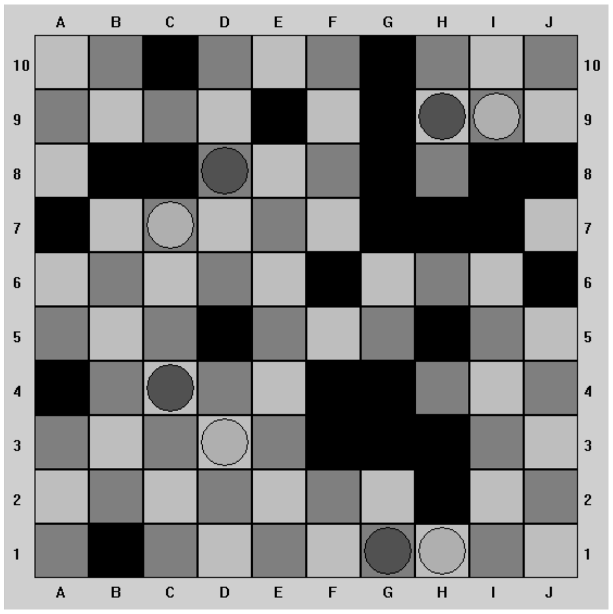


Figure 3.2: Example of a global position badly handled by simple Monte-Carlo

However, this simplicity comes at several costs. The first one is computing power:

while random games are quick to play, the basic Monte-Carlo algorithm for games needs to play a significant number of random games to get a correct estimation of the average result of each move. While this computation problem is common to a number of game programming methods, it is particularly acute here since a good portion of the computing time is allocated to bad moves and thus wasted.

Another cost is that the possible success of this application of Monte-Carlo to games is under the assumption that the move leading to the best average result is the best move to play in any given position. While this is a reasonable assumption and very human-like, it is definitely wrong in certain circumstances. To illustrate this, let us take a closer look at the Amazons position in Figure 3.2: the Black Amazon at D8 is defending a non-negligible territory. While there are probably several potentially good moves in this position, a simple Monte-Carlo program will choose D8D7-D6. While this looks like an opportunity to attack the center, a simple answer would be White playing C7D8-E8, thus stealing from Black all the territory on the top. A more reasonable move both attacking the center and defending the territory on the top would be D8D7-D8, or even D8E8-D8 for more safety.

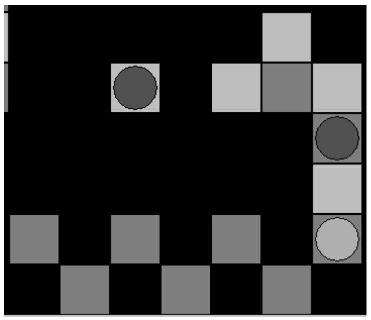


Figure 3.3: Example of a local position badly handled by simple Monte-Carlo

The position in Figure 3.3 gives us another example of a position badly handled by a Monte-Carlo program, this time a local position: while this is a win position for the White player if he is playing first, this kind of territory requires him to play moves in the correct order to win. On the other hand, the upper territory is easier to fill, and will give a better average result, so moving down for the White player will mostly lead to loss for him. On the other hand, moving up (and then shooting back, sealing his own territory by doing this) looks more attractive, because if the Black player makes the mistake of not shooting back at the entrance of his territory, he will lose more often than not. Since we are here dealing with random games, this outcome is quite probable. So in this situation, the White player will prefer to move upward while the only winning move begins by going downward.

The next two sections will present possible improvements to tackle these two particular problems.

Some other problems may appear depending on the game Monte-Carlo is applied to. One of the most acute of them is that, since Monte-Carlo gets its evaluations from random games played from a given position up to the end, it may have some troubles terminating for games without a clear ending condition. Converging games such as the game of the Amazons pose no problem in this respect. For others such as the game of Go, it is possible to enforce specific rule such as *Do not fill the eyes* to ensure that random games will terminate [17]. For lots of others however, such as Shogi [81] or Lines of Actions [84], specific rules or knowledge have to be used to ensure the termination of the random games.

3.1.3 N-armed bandit heuristics and progressive pruning for Monte-Carlo

The N-armed bandit problem, or multi-armed bandit problem, is a simple machine learning problem. It consists in the following:

- A gambler is given N slot machines to play with. Each of these machines has a different but stable reward distribution.
- At each round, the gambler uses one of these machines and gets a reward from it.
- At the end of the process, the goal is to maximize the cumulated reward obtained after an initial number of rounds. A sub-goal of this problem is quite obviously to find the best machine. With an unlimited number of tries, the goal is to maximize the average reward obtained.

This problem lead to the development of several solution algorithms, including some very simple ones: brute force (play all machines the same number of times; this only solves the sub-goal), greedy (always play the best machine), ϵ-greedy (play a random machine each round with the probability ϵ, otherwise play the best machine) etc. Auer *et al.* proposed in [9] a policy called Upper Confidence Bound, or UCB, to solve the multi-armed bandit problem. UCB consists in choosing at each round the machine M maximizing the following formula:

$$\frac{\sum_{i=1}^{N} R(M,i)}{V(M)} + C * \sqrt{\frac{ln(\sum_{machines\ m} V(m))}{V(M)}} \tag{3.1}$$

$V(m)$ measures the number of times that machine m was selected at the time the computation is made. R(m,i) denotes the reward given by machine m at the i^{th} iteration, while N denotes the number of iterations done at the moment of the computation. Finally, C is a parameter that has to be tuned. Using this policy, one will favor better machines (the first term of the formula) and exploit experience gained, and at the same time the second term (the bias term) will raise more and more for any machine which is not selected. This way, the policy will naturally go back to machines tried very few times to confirm their result, but will asymptotically select the best machine an increasing number of times more than all the others (the convergence is shown in [9]). We can also see that, because of the division of the bias term by $V(m)$, every machine has to be tried at least once before the policy can be used. New implementations of this policy such as UCB-tuned or flat-UCB were proposed and studied in [9, 60], the main principle of UCB remaining the same.

Back on the topic of Monte-Carlo, it is easy to see that simple Monte-Carlo for games as presented in Section 3.1.2 is in fact a multi-armed bandit problem: we are indeed presented with several machines, the moves to play, each leading to a given probability of winning (or given average results, whatever the evaluation used). Even if we will discard the final result of UCB, the cumulated reward, we will keep the sub-result consisting of the best average move. Using UCB we can deal with the first major problem of Monte-Carlo for games, the management of processing time, and increase the accuracy of the algorithm in the same amount of time.

Bouzy *et al.* also studied in [17] the concept of progressive pruning applied to Monte-Carlo Go. The main idea of progressive pruning is to prune moves which are shown statistically inferior to at least one other move with a given degree of confidence. In mathematical terms, each move has an average value $V(i)$ and a standard deviation $\sigma(i)$. Using these values we can compute, given a ratio τ, a left and a right expected average equal respectively to $V(i) - \sigma(i)\tau$ and $V(i) + \sigma(i)\tau$. The value of τ has to be tuned and will give us the degree of confidence we need. Then, after a number of random samples, if a move i is such that his right expected average is inferior to the left expected average of any other move j, it is relatively safe to prune i because it is statistically inferior to j. Then the process continues with fewer nodes, and finally can stop when there is only one node remaining (see Figure 3.4).

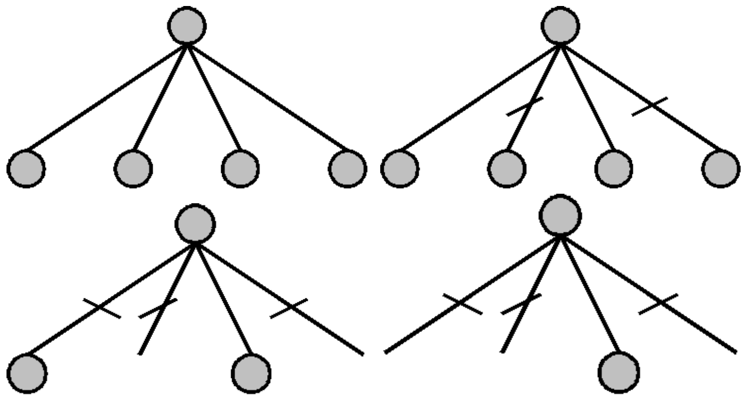


Figure 3.4: Progressive pruning for Monte-Carlo; after a handful of samples the second and fourth node are proven statistically inferior to the first, so they can be removed (top right). Then later, the first node is proven inferior to the third, so he is removed as well (bottom left), which leaves us with only the latter (bottom right).

3.1.4 Integrating knowledge in Monte-Carlo

Another way of improving basic Monte-Carlo for games is to integrate knowledge in it. This can be basically done in two places:

- Use knowledge in the upper part of Monte-Carlo to help choose moves to examine
- Use knowledge in the random games

The addition of knowledge to the Monte-Carlo part has been studied in [12, 16]. It mostly consists in pruning a good number of moves and after that to use Monte-Carlo to evaluate the few of them remaining. This can drastically reduce computation time: the approach studied by Bouzy in [12] concludes that one of the best settings is to keep the top 7 moves in 19x19 Go to get both a reasonable thinking time and good performance. This means at the beginning 7 moves out of 361, which provides in turn a 98% speed improvement (playing on boards of size 19x19 in Go would be impossible without such pruning). The use of knowledge has also been proposed to play the first few moves of the game automatically in [16], similarly to the use of an Opening-Book. Of course, more than making a program play better moves, the use of knowledge in any of these both ways also allows to get more thinking time for more critical parts of the game, which naturally turns into better performance [17].

In a completely different way, knowledge can also be used to change random games into what can be called pseudo-random games. These consist of replacing the uniform probability used to draw moves to play random games by a non-uniform probability, which hopefully plays good moves more often. Using such pseudo-random games have the main effect of biasing the average result computed by Monte-Carlo, which in turns changes the main paradigm *play the move leading to the best average result* into *play the move leading to the best average result according to a given policy*. They also more generally help to speed up the convergence of the average values used by Monte-Carlo to evaluate nodes. With the right knowledge inserted into those random games, one can hopefully overcome the weakness of Monte-Carlo directly resulting from this paradigm, as presented in Section 3.1.2. Bouzy studied this idea for the game of Go in [12], leading to much better performance for his program Indigo at the cost of computational power. Indeed, how much light the process of choosing a random move using a probability distribution which is not uniform could be, it will always be heavier than a simple random choice over a set of move. Of course, as shown by Bouzy, combining this approach with a time-saving approach such as described above, one can get an overall stronger program playing full games in a reasonable time.

3.2 Monte-Carlo Tree-Search

Monte-Carlo Tree-Search (MCTS) is a more recent term, which was introduced after the successful application of Tree-Search techniques to improve the level of Monte-Carlo programs. The next sections will describe the history of such techniques, a simple description of MCTS as it is used nowadays as well as a discussion of its possible good and bad points, and finally present the UCT algorithm.

3.2.1 A brief history of MCTS

One of the first applications of MCTS for deterministic games has been proposed in [13]. The first ideas were pretty simple and far from traditional Tree-Search method: using progressive pruning for Monte-Carlo (as described in Section 3.1.3), after a good number of iterations the algorithm would face situations where few nodes remain but it is difficult to separate them. Since the branching factor in these situations is low, it is possible to add a second level of search and apply the same progressive pruning techniques at depth two. Then, if the same problem appears at depth two, we can expand the tree one more level, and so on (see Figure 3.5). The use of this technique has a higher requirement in computations, but overall increases the level of a Monte-Carlo program and helps prevent situations such as the ones described in Section 3.1.2.

In 2006, Coulom proposed in [34] what would become one of the milestone of MCTS as it is used nowadays. His algorithm consists of a natural and elegant way to add Tree-Search into a Monte-Carlo program and how to combine these to get the most of

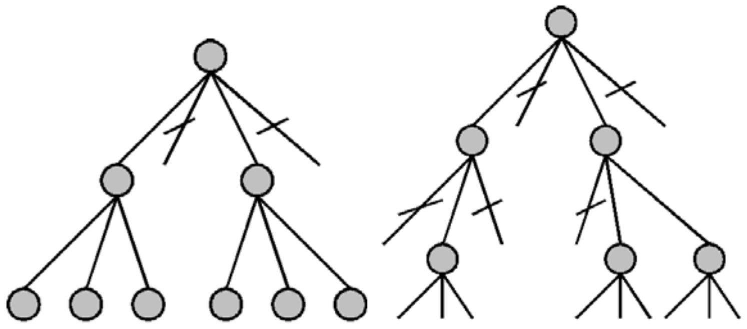


Figure 3.5: From the situation on the top-right of Figure 3.4, a second level of search is added, followed by a third, which will help to choose between both nodes at the root. Ultimately, only one node will remain at the root.

potential exploration and exploitation of the random games played using Monte-Carlo, as well as a very simple way to expand a game-tree to interact best with Monte-Carlo. He also proved the result of his research by winning the 2006 Computer Olympiad in Go on board size 9x9. A more detailed description of MCTS inspired by his work will be given in Section 3.2.2.

3.2.2 Basic MCTS

Monte-Carlo Tree-Search for games as it is known nowadays is an iterative process exploring a game-tree using an evaluation based on random simulations (random games). It is a best-first search algorithm. The basic process consists of repeating the four following operations: *selection* of a leaf node to explore, *expansion* of one of the children of this leaf node, *evaluation* of that child by a stochastic process, and finally *back-propagation* of this evaluation throughout the tree from the new leaf node up to the root. The process stops with a given ending condition, which can be a fixed number of iterations, a given time limit or any other more complex condition based on the results of the search. The pseudocode of this algorithm is given in Figure 3.6.

We will now describe more in detail the four steps of the process.

Step 1: Selection

This steps corresponds to the Tree-Search aspect of MCTS, and what makes it a best-first search algorithm. Starting from the root of the tree, that is the initial position given, the algorithm descends the tree by selecting at each level the most promising node (see Figure 3.7), until it finds a node which is not yet in the tree. The best node is chosen at each level according to a selection policy, given by the function *chooseChildOf* in the code of Figure 3.6. This task is in every respect similar to the choice of a move to explore in the basic Monte-Carlo algorithm (sections 3.1.2). Although this selection policy varies from program to program, the UCB strategy described in Section 3.1.3 is very often chosen for this task.

```
function getBestMove(Position, Endingcondition)
  initialize(tree, Position)
  while ( Endingcondition not satisfied )
    endingnode = tree.root
    Pos = copy(Position)
    while ( endingnode is in the tree ) \\ selection
      endingnode = chooseChildOf(endingnode)
      play(Pos, move leading to endingnode)
    end while
    tree.add(endingnode) \\ expansion
    while ( Pos is not ended position )
      play(Pos, random move playable from Pos)
    end while
    V = evaluation(Pos) \\ evaluation
    while( endingnode =/= tree.root ) \\ back-propagation
      update(endingnode, V)
      endingnode = endingnode.fathernode
    end while
    update(root, V)
  end while
  return(move m with highest value)
end function
```

Figure 3.6: Pseudocode for a basic Monte-Carlo Tree-Search playing engine

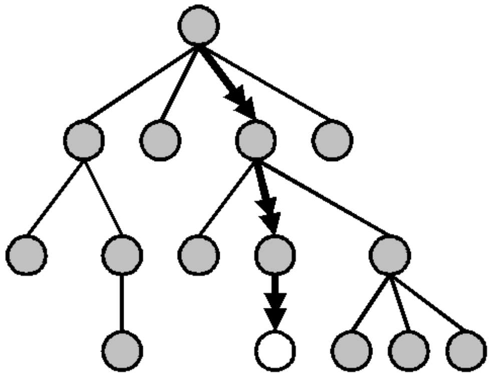


Figure 3.7: Selection of the first best node outside the game-tree. This is done in a best-first manner.

Step 2: Expansion

The second step is quite straightforward: it consists in adding nodes to the search tree. Coulom proposed in [34] to add only the first node which does not exist in the tree, and to discard every other node that one could get from the next step. This choice is quite sensible, as it allows the game-tree to grow at a nice pace, and at the same time includes in the tree the only really important new node, the one which will be evaluated. In some implementations, a node is added in the game-tree only if its parent node or itself has been visited only a sufficient number of time, which is sometimes a number as little as two. This allows for the tree to grow less rapidly and can be useful if memory space is a problem (especially if the thinking time is important, since it leads in turn to more iterations and so to more nodes added to the tree).

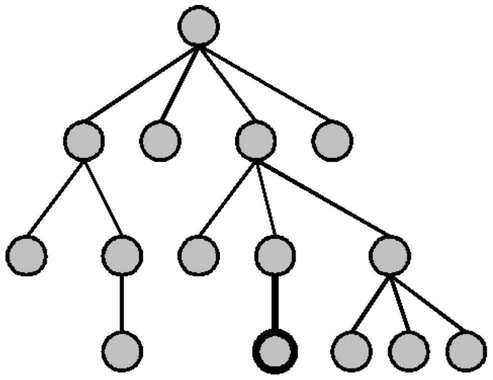


Figure 3.8: Expansion of the tree: the node found in step 1 is added to the tree.

Step 3: Evaluation

This is one of the most critical parts. In the basic Monte-Carlo framework, a node is evaluated with a single random game, and the evaluation becomes the result of that game. For games with a limited number of outcomes (win/loss, perhaps also including draw) the choice is simple. However, for games allowing a score it is not completely clear what option is the best, between evaluating a game by a single digit value (win/loss) or by its score. The first method will lead to a program playing the move leading to the best probability of winning, while the second will focus on moves leading to the best average score. The first method has been shown to lead to better performance for the game of Go (from discussions on the Computer-Go mailing-list [2]), but this result has not been generalized to other games. However, it seems quite clear that it leads to robust play and can be used at least as a first approach for any game.

Step 4: Back-propagation

Finally, the evaluation computed in step 3 (through a stochastic process, the random game) is backed-up from the node added at step 2 to the root of the tree. In a traditional MCTS implementation, each node holds two values: its average evaluation M and the number of times it was visited V. Following this, the update rule for each node with an evaluation E is given by the Formulas 3.2 and 3.3:

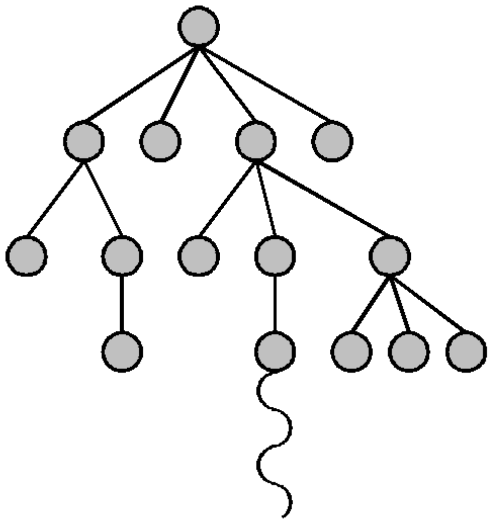


Figure 3.9: Evaluation through a stochastic process (random game).

$$M' = \frac{M * V + E}{V + 1} \tag{3.2}$$

$$V' = V + 1 \tag{3.3}$$

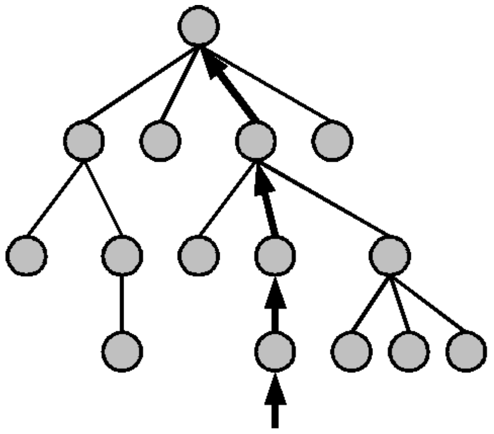


Figure 3.10: Back propagation up to the root of the evaluation found in step 3.

Advantages and drawbacks of MCTS

The addition of Tree-Search to Monte-Carlo using this framework is definitely more demanding both in terms of memory and processing power: while playing a random game is quite quick if no knowledge is added to it, the policy used for the selection process is usually the thing which slows the algorithm. However, this heavier requirement is lifted when one considers the added value of MCTS, that is its ability to read several moves deep. Even with fewer iterations of the process and so fewer evaluations done, MCTS will be able to use these evaluations much more intelligently than basic Monte-Carlo and, provided that these evaluations are of good quality and that the selection policy is good, lead to a much better game playing program than simple Monte-Carlo.

However, one of the bad points of MCTS becomes quite apparent when one tries to make an MCTS program. Traditional minimax-search programs are based on a very intuitive paradigm and, since they are deterministic, it is easy to get quick results of new improvements. MCTS on the other hand, due mostly to its stochastic property, needs a lot of testing for any potential improvement. Also, since its base paradigm is less intuitive, it happens very often that improvements thought good are in fact bad while ideas which seem bad at first appear to be good.

3.2.3 The Upper Confidence Bound method applied to Trees

The Upper Confidence Bound method applied to Trees (or UCT) is an algorithm proposed by Kocsis *et al.* in 2006 and described in [48]. It is an MCTS algorithm based on the UCB algorithm described in Section 3.1.3.

UCT stand for UCB for Trees. The main algorithm is very similar to the one described in Section 3.2.2. Its main interest is to provide a selection policy for MCTS, using UCB for that. This selection policy is described in the pseudocode of Figure 3.11. The function *compute_ucb* used in it corresponds to Formula 3.1.

```
function chooseChildOf(Node N)
  array nonvisitedchildren, ucbvalues;
  for each C children(N)
    if (C has no visits)
      add(nonvisitedchildren, C)
    else
      ucbvalues[C] = compute_ucb(C)
    end if
  end for
  if (not_empty(nonvisitedchildren))
    return random_element_of(nonvisitedchildren)
  end if
  return(node C with highest ucbvalues[C])
end function
```

Figure 3.11: The UCT selection policy

The authors also suggested, in the expansion step of MCTS, to add to the game-tree all the nodes explored in the random game following the selection. However, it is quite easy to see that most of these nodes will be of no use during the whole process and that the top few nodes are the ones that really matter. As of today, most programs switched to the one node expansion described in [34].

Good points of MCTS with UCT

MCTS with UCT (or simply UCT) is a very simple Tree-Search algorithm both in terms of understanding and of programming. It also has the very good property of needing very few knowledge about a game to work, making the programming of a UCT program for any game a simple task, on the contrary to traditional minimax search which needs a more complicated evaluation function. Of course, like for simple minimax search the program level can be very weak depending on the game; like any other program it needs refinements. MCTS with UCT is a simple solution, but definitely not the ultimate one.

One of the other very good points of UCB and what makes it attractive to include into MCTS is its property of convergence. Given unlimited time, the UCT value for a node has been proven to converge to its real minimax value [48]. Since the best child node of any node will be ultimately selected more than any other child and that the bias term of UCB converges to zero, this is quite intuitive. And even if several moves playable from a given position have the same optimal value, UCB will explore each one of them equally, making the average of their result still equal to the real minimax value of the position.

Issues of MCTS with UCT

Of course, some bad points remain. First, MCTS with UCT inherits several of the drawbacks of Monte-Carlo (like its problems to deal with games which do not finish clearly, cf Section 3.1.2) and of MCTS (the need of heavy testing mostly). But, more related to UCB is the fact that, even if UCB does indeed converge to an optimal choice, this convergence can be quite slow. The fact that every childr node has to be explored once for UCB to be applicable is another factor slowing down the whole process.

3.2.4 Parallelization of MCTS

Parallelization has become a hot topic in game programming as far as Monte-Carlo is concerned. While classical Chess programs have since long come with machines using several processors or clusters of processors (see Deep Blue [22]), the parallelization of traditional minimax search is not a trivial task. Parallelizing Monte-Carlo, on the other hand, is much easier to do. It has also been shown that, for 9x9 Go, doubling the processing power gives on average 100 ELO of gain to a Monte-Carlo program [2]. Now, since simply doubling the number of processors does not always equate to doubling the processing power, this observation does not hold for direct hardware doubling but is definitely a good sign.

Since it is not the main focus of this work, we will not provide much more details about parallelization of MCTS here. An interested reader could consult [25, 26, 28] for some state of the art work about MCTS parallel programming.

3.3 Adaptation to the game of the Amazons

In this section, we will deal with the adaptations that have to be made to play Amazons with MCTS and discuss several simple improvements. performance measurements will follow in section 3.4.

3.3.1 Speeding up the random games

One of the most straightforward changes that can be done for a Monte-Carlo or MCTS Amazons program consists in changing the choice of one single move at random in the random games to a double choice: first choose an Amazons movement at random, then choose a shoot. For true, the probability distribution obtained is not uniform anymore and thus biased towards playing Amazons having access to more potential movements.

However, the speed-up obtained more than compensates for that bias, since it is much easier to compute a list of movements and of shots than a list of full moves combining a movement and a shot.

3.3.2 Grouping nodes

Besides splitting moves in the random games, it is also possible to split them in the Tree-Search part of MCTS; this means that we will add a second layer of nodes in the game-tree of MCTS at each depth, grouping together all the moves that start with a given Amazon movement. Hopefully, this will allow us to handle more efficiently the huge branching factor of the game.

3.3.3 Evaluation changes

While using a single bit evaluation for the game of Go has proven its success against using the score of a game in the Evaluation phase of MCTS [2], it is not clear yet which is better for the game of the Amazon. For this reason we will investigate three potential evaluations:

- A simple classic evaluation (0 for a loss, 1 for a win)
- A score based evaluation, respectively positive and negative for a win or a loss, and with a 0.5 factor to differentiate wins by 0 point as described in Section 2.2.1
- A combined evaluation, based on the score biased with a factor larger than 0.5 to improve the effect of winning a game.

3.3.4 The program Campya

To test the efficiency of MCTS for Amazons as well as the possible improvements described before, the author created an Amazons playing program using Monte-Carlo called Campya. The program is written in C++ for consideration of speed as well as the portability of the code. More details about this program can be found in Appendix B.

3.4 Experimenting MCTS Amazons

We will present in this section the results of our first experiments using Monte-Carlo and Monte-Carlo Tree-Search in our program Campya.

3.4.1 Experimental methodology

Evaluating an MCTS Amazons program is a difficult task in multiple ways. First, MCTS program are said by most of the developing community [3, 2] to be more difficult to evaluate than traditional minimax search programs. Because of the stochastic aspect of the method, it is not possible to judge accurately if a new feature was indeed an improvement just by looking at the program playing one game. Also, new features in Monte-Carlo programs usually have both strange and counter-intuitive interactions with each other. For example, adding knowledge to random games seems a good idea at first, but one quickly discovers that it could lead to worse results, and that a minimum level of randomness is necessary [27, 72].

Next, there is the problem of the game itself. The game of the Amazons, while interesting in several aspects and a good testbed, has not attracted as much attention from the gaming community as other games such as Chess, Go or Othello. This means both that there is few opponents (humans or programs) to play against, and that there is no well known server or protocol to play games automatically to help the evaluation process of our program.

At the same time, self-play is a simple and not so bad evaluation, but does not take into account the possible variety of programs and of playing style. For this reason, we decided to extend the idea of self play and created our own private Amazons playing server. It does not use any specific kind of protocol, so it was not possible to plug in other programs (who would anyway need first to implement such a protocol), but we introduced versions of our program with different evaluations and playing style as well as traditional Alpha-Beta versions to help evaluate the improvements made to CAMPYA. Next to that, we used ELO ratings to both evaluate different versions of our program and help the process of pairing programs to play together (the reader not familiar with ELO ratings can refer to Appendix A). Finally, we used the program Bayeselo from Remi Coulom [32] to compute more accurate ELO ratings for the various versions of our program.

To help evaluating the tested features, we almost always played two versions of each program:

- The first one played with a fixed number of evaluations per move (one evaluation consisting in one iteration of the MC or MCTS process). We fixed this number at 40.000.
- The second one played with limited time for the whole game (fixed to 5 minutes per game).

The first version of each program would allow us to verify if the use of a new feature was indeed an improvement, while the results of the second would tell us if the inclusion of that improvement did not deteriorate the performance of the program in tournament conditions because of a higher need in processing power. For features not demanding in processing power, only one of the versions was tested.

3.4.2 Monte-Carlo and MCTS for Amazons

The first feature we tested was the simple inclusion of Tree-Search (using UCT) in our program, and compared it to a simple Monte-Carlo version of the program. To establish this comparison, we also used three different kinds of evaluations such as described in Section 3.3.3: a simple win/loss evaluation, a score-based evaluation and a biased score-based evaluation. We evaluated these versions of our program in our Amazons server such as described in Section 3.4.1. Results are summarized in Table 3.1.

We can see clearly from these results that the inclusion of Tree-Search in a Monte-Carlo Amazons program is necessary to get a better playing level. Also, it seems that, unlike other games, Amazons playing programs benefit more from an evaluation based on the score rather than on a win/loss ratio. And that, both for simple Monte-Carlo and for MCTS. The use of the biased evaluation showing better results than the simple score for

Algorithm	Setting	Evaluation		
		Win/Loss	Score	Biased score
MC	40.000 samples	33	67	58
MC	5 minutes	-60	62	61
MCTS	40.000 samples	537	711	780
MCTS	5 minutes	458	596	686

Table 3.1: ELO ratings for simple version of Campya

MCTS while showing similar results for MC suggests that this evaluation does provide more information that MCTS is able to exploit, while simple MC is not. At least, at this playing level.

However, even a program based on MCTS plus an evaluation based on the score (with a simple bias) has a very weak level. As we can see from the position in Figure 3.12, the best version of Campya so far cannot keep up with even an average level player. It lets its Amazons be enclosed without really noticing it (although crude attempts at this are still understood by the program, MCTS helping for that), and does not have any good understanding of territory. Most notably, it does not know where to shoot arrows correctly. The game ended with a win by around 40 points for the author.

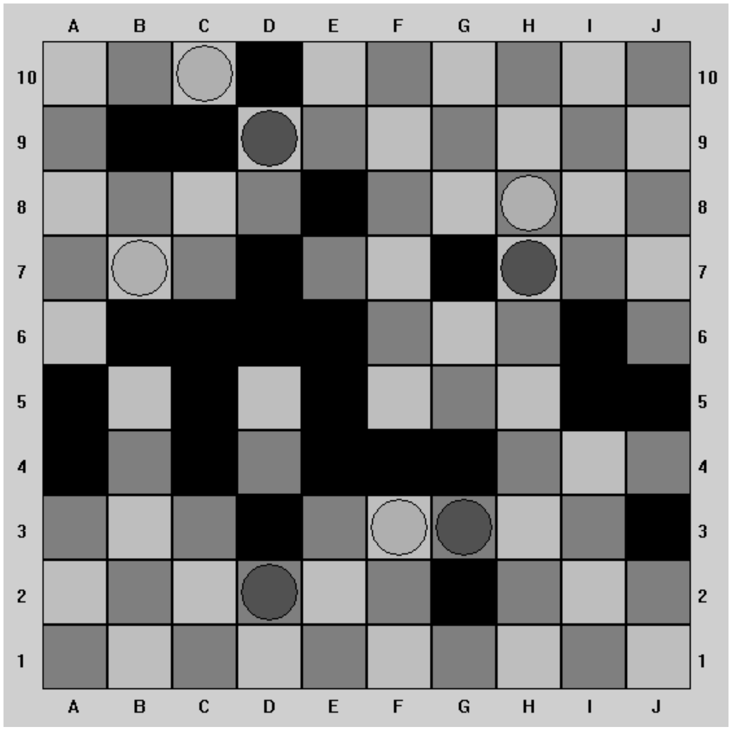


Figure 3.12: Game between the author (Black) and Campya (White)

3.5 Modifying the reward

This section will deal with our attempts to integrate an evaluation function into an MCTS Amazons program.

3.5.1 Using an evaluation function

Since simple Monte-Carlo Amazons programs are very weak, even with the addition of MCTS, it became apparent that we would need a very important improvement to make our program play at least at human level. For that, we would need to introduce concepts such as territory, mobility or Amazons distribution to our program. This means either modifying the Tree-Search engine or changing the random games to integrate such knowledge. Since some good evaluation functions already exist for the game of the Amazons, using one of them seemed like a natural way to include this knowledge we needed.

Our proposition to include such an evaluation consists in changing the basic Monte-Carlo paradigm. The basic stochastic evaluation used by Monte-Carlo programs consists in playing a random game (perhaps biased by some probability distribution) until the end of the game and then evaluate the final result, whatever way this is. We propose here to replace the endgame evaluation by an approximation using an evaluation function. This also removes the need to play random games until the end, thus these games can be stopped after a given depth. Since a good evaluation function would provide us already with a correct approximation of the result we are searching for, this improvement should both give us a better quality of evaluation and speed the convergence of the average value of a node to its correct value.

Experiments and results

We took inspiration of the work of Jens Lieberum [51] to design our evaluation function. More details about it can be found in Appendix B. To diversify our tests, we used three different evaluation functions:

- The first one (noted as *classic*) is a simple evaluation function based on the accessibility.
- The second one (noted *complex*) is an extension of the classic function with concepts such as mobility and Amazons distribution. This evaluation function is more computer intensive, but will hopefully give us a better player overall.
- The third one (noted as *ratio*) is obtained by applying a sigmoid function to the output of the complex evaluation function. This will give us a restriction to the $[-1; 1]$ interval, permitting us to obtain a behavior similar to standard MCTS using a win/loss evaluation.

We chose initially to cut the evaluation after 8 moves from the root of the game-tree whatever the depth of the leaf that had to be evaluated into the MCTS process. This means that to evaluate a leaf node at depth D, we would play a random game (we would still call them that way, even if they are too short to be worthy to be called games any more) for $8 - D$ moves. If the leaf node to be evaluated had a depth of more than 8, it would be evaluated straightaway without any random game played.

Algorithm	Setting	Endgame evaluation	Classic evaluation	Complex evaluation	Ratio evaluation
MC	40.000 samples	67	763	856	876
MC	5 minutes	62	830	957	985
MCTS	40.000 samples	780	1500	1741	1779
MCTS	5 minutes	686	1541	1809	1835

Table 3.2: ELO rating for each of the three evaluations as well as for the full random game evaluation, for MC and MCTS, with unlimited and limited time conditions

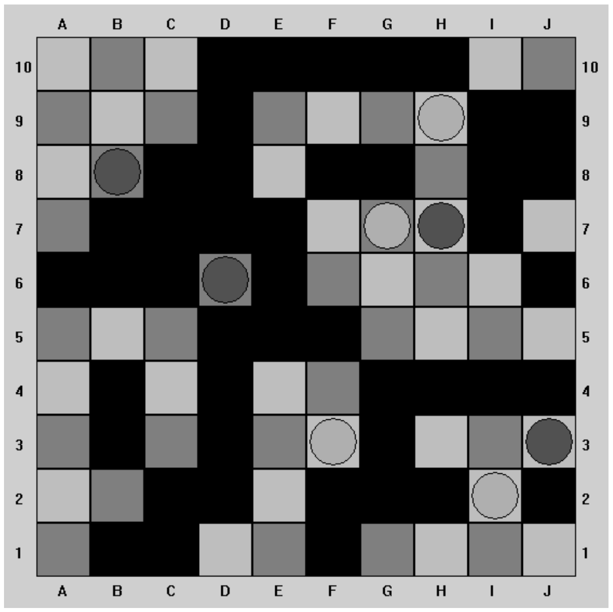
The results of this experiment are summarized in Table 3.2. As we can see from them, the addition of an evaluation function, whatever simple it is, is the boost to the strength of our program we needed: whether the program used is basic Monte-Carlo or MCTS, it provides us with a gain of a whooping 700-1200 ELO depending on the algorithm and the evaluation used. In fact, even though the evaluation function is quite demanding in terms of computer resources, playing a full random game is not much quicker than playing a short random game and then calling an evaluation function. Thus, the gain in quality of play obtained by using the evaluation function is quite clear. This is also confirmed by the fact that using the right evaluation function is more beneficial to MCTS than to basic Monte-Carlo.

It also seems that, according to the ELO ratings of the programs involved, applying a sigmoid function onto the result of the evaluation function is a good thing. However, the gain is not that clear, and both versions of the program using the complex and the ratio evaluation functions respectively tie more often than not when playing together. The small gain in performance of the sigmoid version comes to the fact that it has slightly better performance over other versions of our program in general.

Discussion

Now, one could wonder why evaluating a position at such a low depth can give much better result than traditional Monte-Carlo. While we do not have clear evidence to answer to this question, the answer seems pretty straightforward. In Monte-Carlo, whether it has Tree-Search or not, the stochastic evaluation of a node provided by playing a random game from that node should give us some estimation of the value of this node. While the stochastic aspect of this evaluation makes it unreliable on a single-game basis, on average, this evaluation should be representative of the position corresponding to the leaf node we want to evaluate. However, the game of the Amazons has a huge branching factor. This, combined with the fact that bad moves can be disastrous to the side to move, removes a lot of the reliability of completely random games to evaluate a position, unlike the game of Go where stones do not move and thus the board state has a higher probability to remain similar. The fact that the game can gain a sudden-death aspect when bad moves are played (meaning that the game ends with just a single bad move) does not help in this respect.

Let us illustrate this by an example. During the computation of MCTS, it is possible to draw a property table of the board. After each random game, we will count a square



	A	B	C	D	E	F	G	H	I	J
10	-42782	-45547	-42561						17130	13777
9	-46613	-48508	-44957		22796	26670	27886	32732		
8	-44503	-46476			22423			21423		
7	-39952					17876	22082	17119		-1922
6				-33762		12566	11728	-3511	1442	
5	-28451	-37752	-42971				6760	-165	-1660	6837
4	-24576		-36953		25581	18968				
3	-20105		-29718		-36108	37926		-11788	-8308	-36177
2	-18114	-20802			35407				21312	
1	-18292			32075	31524		17964	19218	14890	18499

	A	B	C	D	E	F	G	H	I	J
10	-49947	-49956	-49943						33923	32852
9	-49979	-49979	-49958		40225	42250	42012	41260		
8	-49979	-49991			29450			14967		
7	-49815					31873	23229	10710		-14006
6				-46952		13967	16317	-24726	-23100	
5	-45388	-48093	-49296				8829	-13397	-21103	-4222
4	-45208		-47987		43736	24810				
3	-44416		-45787		5267	47415		-9320	-6176	-33042
2	-44092	-44496			46953				41638	
1	-44095			45568	40928		28310	28593	29158	29757

Figure 3.13: Property tables for the top position for two MCTS player, with evaluation at the endgame (middle) and after 8 moves (bottom), after 50.000 iterations

positively if it is considered being in the White player's territory or if the White player shot an arrow on it, and negatively for the Black player. The sum of all these values computed for all squares of the board will give us the property table of the board. Simply put, it will show us how many times (relatively to the other player) a square was part of one player's territory during the computation. Figure 3.13 gives us a position as well as the property tables obtained for an MCTS player using the score at the end of the game as an evaluation (with a bias) and an MCTS player using the Complex evaluation function described in Section 3.5.1 after 50000 iterations of the MCTS process. We can see quite clearly that, while the former estimation of each player's territory is not so bad, it is much less precise than the later's estimation. In zones where both players can move the estimation of the first player is sometimes plain wrong (as for square E3, which definitely should not be part of the Black player's territory), and in territories already formed the estimation is less than optimal (see both territories on the left as well as the ones in the bottom and on the top right). Using an evaluation function after the random games, even if it does not provide an optimal value all the time, at least gives us a much better evaluation of the position.

Since, by adding a simple evaluation function (the Classic one) we were able to get a fairly good program, we decided for the rest of the experiment to fix its ELO value to 1500 when playing with 40000 iterations per move. Other ELO values obtained through experiments from this point on shall be compared with this reference value of 1500.

3.5.2 Removing Monte-Carlo ?

Since playing random games until their end is just so inefficient for Monte-Carlo Amazons, another option we studied is the complete removal of the stochastic process from MCTS with UCT, changing it in the end into UCT Tree-Search. Without its Monte-Carlo part, an MCTS engine just becomes a classical Tree-Search engine with a different move selection policy (UCB instead of argmax like in traditional minimax search) and a completely different process to backup leaf values to the root of the tree (average instead of minimum and maximum). The results of Section 3.5.1, showing that short random games are better for an MCTS program to be stronger, also suggest that the stochastic element of Monte-Carlo is one of his main weak points. For that reason, this implementation of UCT Tree-Search seemed advantageous in a way.

We tested UCT Tree-Search by simply removing random games from our program Campya without changing any of its other characteristics. We then used the three different evaluation functions described in Section 3.5.1 to get a better feeling of this removal. The results of this experiments are summarized in Table 3.3.

Removing the Monte-Carlo part of our MCTS engine cuts between 300 and 400 ELO points for each version of the program. The conclusion is clear: randomness is necessary.

While the random games played in our program are indeed very weak (as illustrated in Section 3.4.2), they provide two very important assets to our MCTS implementation:

- They allow the program to look deeper in the game-tree. Even if the information gained is weak on its own, with sufficient sampling it gets some significance.
- Most important, regardless of the depth at which random games are cut, these provide the program more exploration of the possible future states.

Setting	Classic evaluation	Complex evaluation	Ratio evaluation
40000 samples, classic cut	1500	1761	1784
5 minutes, classic cut	1545	1817	1848
40000 samples, no MC	1063	1475	1376
5 minutes, no MC	1192	1494	1490

Table 3.3: ELO ratings for each of the three implementations of UCT Tree-Search as well as for basic MCTS (with random games cut after 8 moves), with unlimited and limited time conditions

3.5.3 Influence of the depth

Since it became apparent that modifying the evaluation of MCTS to include an evaluation function without completely removing the stochastic aspect of random games was the right move, we wanted to study the effect of modifying the depth at which these games are stopped to the performance of our program.

Fixed depth from the root

In this first experiment, we decided to stick on our implementation consisting in stopping all the random games at the same depth. After some trial and error, we found previously the depth of 8 (that is 8 moves in each iteration including both moves made during the selection process of MCTS and those made in the random games) as being a correct depth. We will study here more precisely the effect of this depth by evaluating the performance of our program with the three evaluation functions presented in Section 3.5.1. Since modifying the depth so slightly does not change that much the speed of one iteration in the MCTS process (the computation of the evaluation and the selection process of MCTS are the two resource intensive parts of an MCTS Amazons program), we only experimented with unlimited time and 40000 playout per move. The results of this experiment are shown in Figure 3.14.

From what we can see, the value of 8 found by trial and error was indeed good for a program based on our simple evaluation function. However, it definitely does not fit a program using a more complex evaluation function, for which a smaller depth of cut (6 or 7) is preferable. We impute this result to the fact that longer random games offer a more important variance which is not compatible with the precision of the concepts introduced in our more complex evaluation function.

On another note, we can also conclude that it is necessary to find a good balance between the amount of information we can get and the relevance of this information. Long random games provide more information of less quality, while short games provide less information of better quality. But one thing is for sure, as we saw from the study of the effect of removing the random games in Section 3.5.2: this information is necessary.

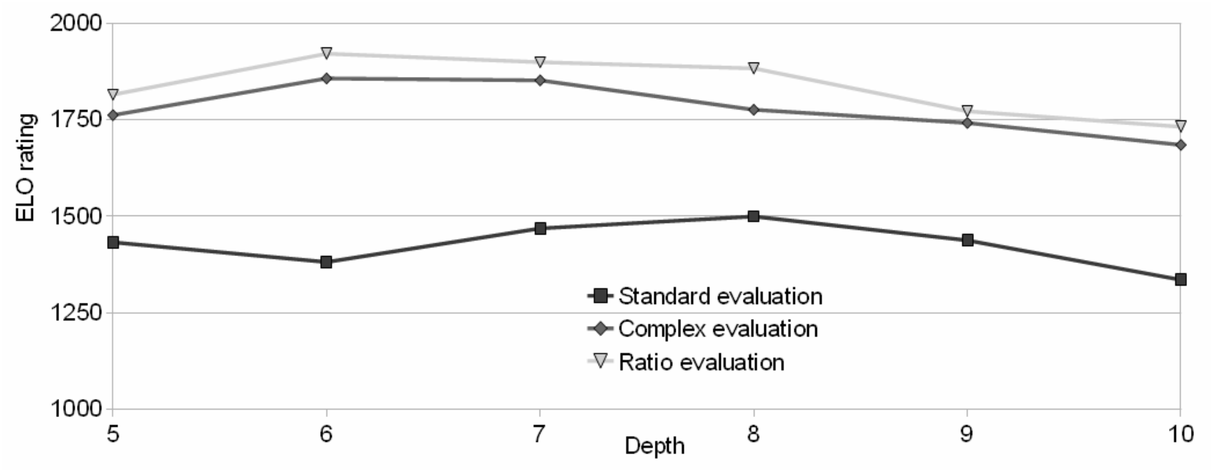


Figure 3.14: ELO ratings for various depth of cut and evaluations for the random games

Random games of fixed length

One possible weakness of our implementation consisting in halting random games at a fixed depth is that this depth cannot adapt to specific situations in which we would like the search to go deeper. If the selection process of MCTS finds a leaf node which depth is near or over the limiting depth, only very short random games are played, or perhaps even none. As we saw in Section 3.5.2, this could be prejudicial to our program. In this section, we will propose another implementation: instead of halting the random games at a fixed depth D, we will play random games of a fixed length L, meaning that the depth of the cut will adapt itself to the depth of the leaf node to be evaluated.

Since the optimal depth for halting random games in our first implementation lied between 6 and 8, it made sense that the optimal length of random games in the present implementation should be somewhat smaller. We then proceeded to the same experiment as Section 3.5.3 with various lengths for the random games. Results can be seen in Figure 3.15.

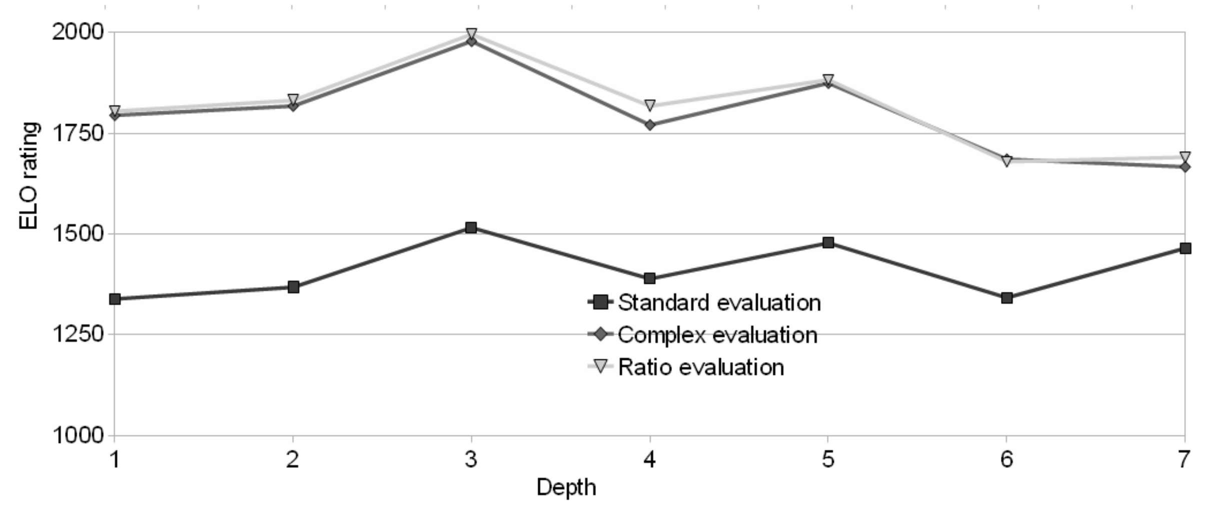


Figure 3.15: ELO ratings for random games length and evaluations

As for the previous implementation, we can clearly notice that there is a balance to find between quality and quantity of information, and that the optimum of the random game's length should be 3 or 5. This is consistent with the best depth of 6-8 found in the previous experiment if we consider leaf nodes located at a depth of around 1-3.

However, we are faced this time with a stranger phenomenon, since it appears that odd game lengths usually give better results than even game lengths. This can be usually explained by the fact that a program's style will be different if it ends its searches at an odd or even depth (ending the search with the player to move will usually lead to a more aggressive style, and vice-versa). However, since the search ends in the present case at a varying depth which can be both odd and even, this explanation does not hold. The fact that we did not observe this phenomenon in the experiments with a fixed depth of cut in Section 3.5.3 confirms that, except for the fact that the best value for the depth was found to be even, having the search stops at an odd or even depth does not really influence the performance of the program.

For this reason, we propose a third implementation, in which random games will get a fixed number of moves N or $N + 1$ to make them all end at an even depth. This way, all random games should end with the second player moving but still have a similar length. The same experiment was performed, and its results are summarized in Figure 3.16.

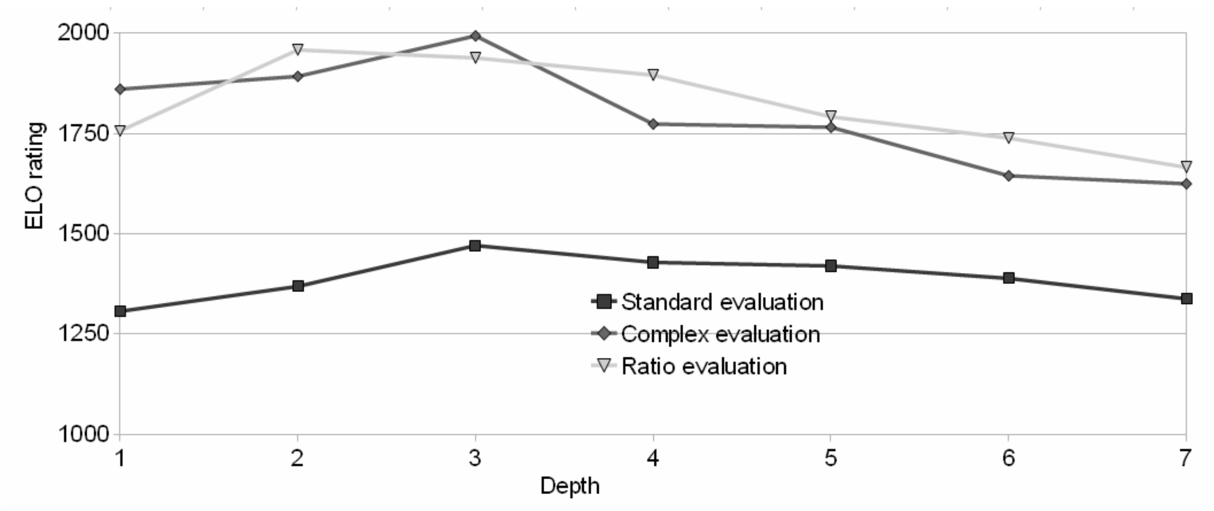


Figure 3.16: ELO ratings for various random games length and evaluations, with the games always ending at an even depth

As we expected, with this last implementation the odd-even effect observed is no more. However, it should also be noted that, with the possible exception of the complex evaluation function, the results of the best implementation for each evaluation function are slightly inferior with our current implementation than with the previous one.

For reasons of clarity, we also provide in Figure 3.17 the difference in ratings from the two previous implementations with the same settings. It seems quite clear from there that, while the addition of one random move to end random games at an even depth seems to help implementations with an even fixed game length, on the contrary it hinders implementations with an odd length for the random games, the latter having overall better performance. This is true for all kinds of evaluation functions. For this reason, we also experimented with a variable length for the random games, both ending the latter always at an even depth or not, but could not get any convincing result.

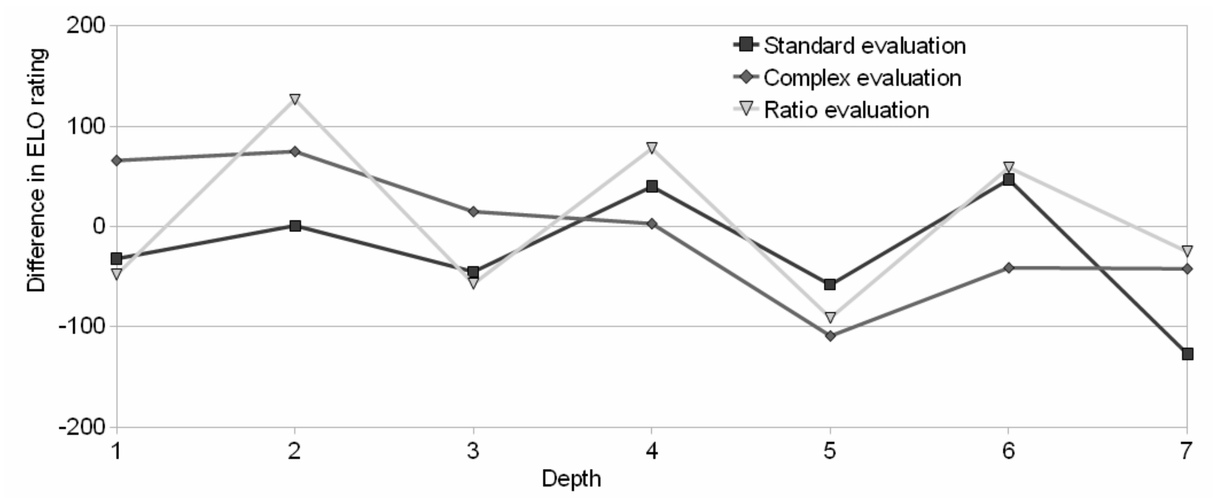


Figure 3.17: Difference in ELO ratings between a fixed random game length and the same length with a variance of one move to end games at an even depth

In the end, as expected at first, having a fixed length for the random games seems to be a better choice than halting all games at a fixed depth and there exist optimal values for this length, but one should be careful with possible local extrema when searching for these values, even in situations when one would not expect an odd-even effect to appear.

3.6 Grouping nodes

One of the aspects of the game of the Amazons which makes it so difficult for computer programs to handle is its high branching factor. With more than 2000 moves for the first player and more than 1000 on average for several more moves, the task proves itself very challenging. We present here a simple improvement which will help us to tackle this difficulty and thus help the search engine of our MCTS program: grouping nodes. The idea consists in expanding the game-tree structure by adding nodes to it representing a set of other nodes sharing a common criterion. While it should be possible to come up with various criterion for different games (proximity to a given square, piece or move, distance of a move, property of a move such as moving into a territory...), the game of the Amazons provides in itself a simple criterion to use: we will group together all nodes sharing a common movement.

As shown in Figure 3.18, grouping nodes has two main effects:

- It artificially reduces the branching factor of a game tree. However, that does not mean that we will be able to search much deeper, since all the nodes from the original tree are still present into the tree grouping nodes. The depth of the tree is increased by the same factor of reduction of the branching factor.
- It adds a set of meta nodes into the tree; in the present case, movement nodes.

In the end, the number of nodes in the tree is not reduced, quite the opposite. However, we hope that this change will permit an MCTS program to exploit more its results and thus in the end to increase its performance.

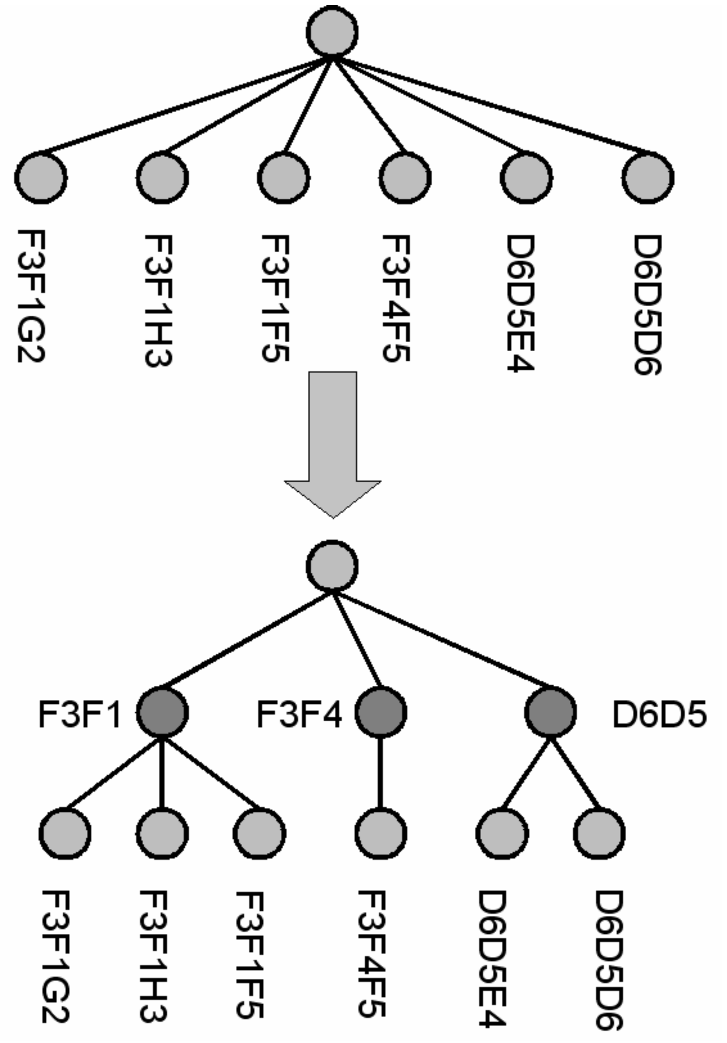


Figure 3.18: Grouping nodes by common movement

Experimental results

We studied the effect of grouping nodes in our MCTS implementation using various evaluation: two based on the result of a complete random game such as described in Section 3.3.3 (we did not use the ratio based evaluation since its performance was strictly worse than the two others), and the other three using an evaluation function after 8 moves such as described in Section 3.5.1. The performance of the programs using these evaluations in both limited and unlimited time are summarized in Table 3.4.

Evaluation after a full random game		
Setting	Score	Biased score
40000 samples, grouping	719	788
5 minutes, grouping	601	694
40000 samples, no grouping	448	617
5 minutes, no grouping	226	336

Evaluation after 8 moves			
Setting	Classic evaluation	Complex evaluation	Ratio evaluation
40000 samples, grouping	1500	1761	1784
5 minutes, grouping	1545	1817	1848
40000 samples, no grouping	1223	1632	1521
5 minutes, no grouping	1130	1476	1362

Table 3.4: ELO ratings for several different evaluations, with and without grpuping nodes, with unlimited and limited time conditions

From these results, it is quite clear that grouping nodes by movement is a huge improvement for our MCTS program. Whether the evaluation is precise or not, our program gained between 300 and 500 ELOs from grouping nodes in limited time. The gain from having to generate less important moves lists is quite clear, and shorter lists also means that the computation necessary for UCT is less demanding. Both lead in turn to more random games, and thus more effective programs.

What is more surprising is to see that even programs playing with unlimited time, and that again whatever the evaluation used, benefit a lot from grouping nodes, since we observe a gain of 150 to 300 ELO compared to the versions without grouping. This allows us to say that grouping moves by their common movement is not a bad heuristic, since it allows the UCT algorithm to be more effective in selecting more promising nodes: while without grouping the exploitation of UCT could not begin before 2176 random games from the standard Amazons starting position, using grouping nodes 80 games suffice to exploit their result. The lack of information from the remaining part of each move (the arrow shot) is largely compensated by that faster exploitation.

Other studies also provided some good results for grouping nodes for the game of Go [63, 30].

3.7 The All Moves As First heuristic

3.7.1 A simple concept: the similarity of positions

Some improvements for basic minimax Tree-Search already exploit the concept of similarity for position. We can cite killer moves [6], the history heuristic [65] or more recent techniques such as the context-killer heuristic [41]. All these methods suggest that if two positions P_1 and P_2 are similar, a good move to play in P_1 should also be good in P_2. While this is definitely not always true, it provides us with a good and simple heuristic to guide the search in a game-tree.

The same way, it is possible to use this concept in MCTS. Bouzy proposed in [17] what he called the All Moves As First heuristic (AMAF) for Monte-Carlo. In simple Monte-Carlo, after one random game is played, the only move whose value is updated with the evaluation taken from this game is the first move that was played. Using AMAF, we will not consider only the first move of the random game but every move played during the random game (or just a subset of it if the update process is too slow) and update all child nodes from the root reached by playing one of these moves, hence the name of the heuristic. This is an extreme use of the similarity concept, since it considers that all the position reached from the root by playing a random game are similar to the position of the root.

Later, Gelly and Silver proposed in [38] an adaptation of AMAF to the framework of MCTS. Although the spirit of the method is similar in MCTS, it has to be adapted, since it would be impossible to update all the nodes of the game-tree reachable from playing any move that appeared in a single random game (even if just part of it). The process they proposed functions in the following way:

- During the descent in the game-tree (the Selection part of MCTS), store all the nodes encountered from the root node to the leaf node in a set S
- Evaluate the leaf node by playing a random game, and get a set of moves played during this game (all of them, or just a subset); this evaluation shall be noted as E, and the set of moves M
- During the Update part of MCTS, on top of updating all nodes of S with E, update all nodes reachable by the nodes of S by playing any of the moves of M

In this implementation, summarized in Figure 3.19, two positions are considered similar if they were both reached during the same descent of the game-tree from the root down to a final node.

Now, one of the main differences between the traditional AMAF for Monte-Carlo and this implementation is that one will update separately the average result of the random games going through a node and the average result of the moves leading to it (although from different positions). The average result of random games will be the same as in traditional MCTS. However, following the basic assumption of AMAF that a move in a position similar to P should be good in position P too, the average result of moves will give us a crude indication of how good it is to a move. For this reason, and since this average value will be more quickly computed than the regular average due to more data updated at each iteration of the MCTS process, Gelly *et al.* called this adaptation Rapid Action Value Estimate (RAVE).

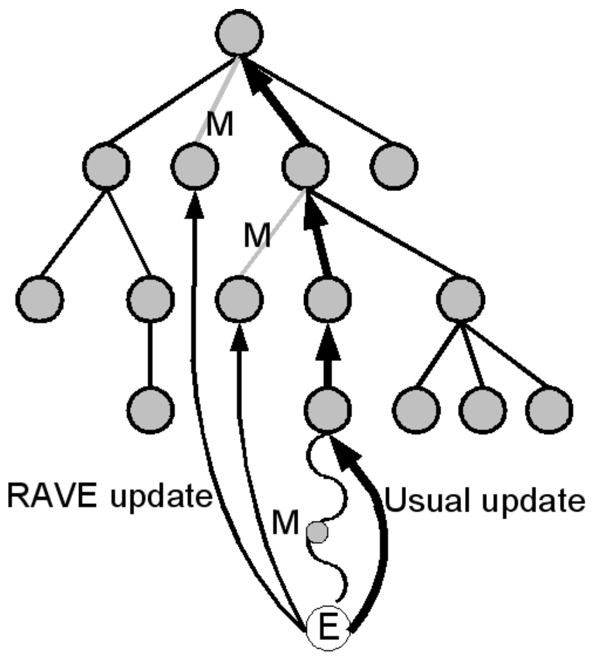


Figure 3.19: Update process of RAVE, showing the update for one move M

3.7.2 Implementing RAVE for the game of the Amazons

However good the RAVE heuristic could be for the game of Go - combined to other knowledge data, it even replaced at some point the bias term of the UCB formula in the Go program Mogo [3] - it is difficult to adapt as-is to the game of the Amazons. Beside some possible memory cost, the main reason is that while in Go a move played in a position P_2 reached from playing random moves from a position P_1 can very often be played in P_1, that is almost always except in cases of captures, this is not the case for the game of the Amazons since the main pieces - the Amazons - move a lot. And while some of them would indeed be playable in the previous position P_1, the computational costs of having to check if indeed the move is playable combined to the loss of data of un-playable moves makes a straight adaptation of RAVE to the game of the Amazons difficult.

For this reason, we decided to implement AMAF using what we will call a moves table, in which the updates of the RAVE value will be done. In a way, we are back to the traditional idea of AMAF for Monte-Carlo saying that all positions reachable from the root are similar since they share an ancestor. Of course, this means that some information is lost since we greatly enlarge the set of similar positions, but we hope that the computational gain of this approach will balance out this information's loss. Moreover, the similarity shall be kept in some way. Considering that random games played in an MCTS Amazons program are short, as described in Section 3.5 and that, for one move to be playable in two position reached during the search process, at least one Amazon should be on the same square and all the squares concerned by the move should be empty, the two positions should have some degree of similarity between them. We can thus see that all the idea of similarity is not lost using this implementation.

3.7.3 Integrating AMAF/RAVE into the MCTS framework

Gelly *et al.* proposed in [38], when evaluating a node in MCTS, to combine the classical UCB term based on the average result of random games to another UCB term based on the average result of the move leading to this node (from the point of view of the father node). This lead them to evaluate the position P reached from move M with the following formula:

$$Eval(m,k) = (1 - Q(N)) * UCB(P) + Q(N) * UCB(M) \tag{3.4}$$

with

$$UCB(M) = \frac{\sum_{i=1}^{N} R(i,M)}{V(M)} + C' * \sqrt{\frac{ln(\sum_{moves\ m} V(m))}{V(M)}} \tag{3.5}$$

where $UCB(P)$ is the classical UCB evaluation of the node representing position P and $UCB(M)$ the UCB evaluation of the move M, $R(i,m)$ the reward received by the move m at the i^{th} iteration, $V(m)$ the number of times move m was updated, and N the number of time the father node of P was visited; the latter also determines the weight $Q(N)$ of the RAVE term. Simply put, we compute a new UCB term based on the confidence we have for the move m, and combine it with the classical UCB term by giving the former a decreasing weight $Q(N)$. Ultimately, only the classical UCB term will matter.

Now, while this implementation gave great results for the game of Go in the program Mogo, we could not manage to get significant results from it for our program Campya. In the following sections, we will present two possible ways of integrating the RAVE value in our Amazons program to guide the search as well as the results of such implementations.

Adding the RAVE value

One of the most simple ways to include knowledge into a UCT based framework is to simply add a bias term based on said knowledge to the UCB formula to select moves. Since RAVE is a knowledge based evaluation, although computed online, we decided to try its efficiency by adding it in this simple way. The UCB formula then becomes:

$$\frac{\sum_{i=1}^{N} R(P,i)}{V(P)} + C * \sqrt{\frac{ln(\sum_{siblings\ s} V(s))}{V(P)}} + W * Rave(m) \tag{3.6}$$

with $Rave(m)$ being the RAVE value computed for move m leading to position P. W is a weight factor that has to be tuned experimentally.

We analyzed the performance of our program after adding the RAVE values and compared it to the previous version without the improvement. We used for that 3 versions of our program with the evaluation function described in Section 3.5.1, with both limited and unlimited time. Since the computations of the RAVE values are demanding in processing power, it made sense to also analyze the results in limited time.

The first thing striking from these results is the efficiency: even with a very simple implementation, that is adding the RAVE value to the UCB term to bias the search

Time	40000 samples			5 minutes		
W	Classic evaluation	Complex evaluation	Ratio evaluation	Classic evaluation	Complex evaluation	Ratio evaluation
0.2	-25	+35	+190	-105	+51	+35
0.5	+76	+48	+186	+20	+64	+93
1	+206	+91	+215	-11	+133	+125

Table 3.5: Evolution of the ELO rating of our program integrating a simple RAVE term with weight W, for various evaluation functions and time settings

towards more promising moves, our program gained between 100 and 200 ELOs depending on the evaluation function use. Furthermore, this gain in performance also appears in limited time, with an improvement of around 100 ELOs for the best evaluation functions. This means both that adding the RAVE value benefits to the search, and that computing this value is not computer intensive to the point that it would reduce the efficiency of this improvement to none in limited time.

We can also observe that adding the RAVE value is much more beneficial for more sophisticated evaluation functions, which seems quite natural since it then brings more meaningful information to drive the search. Finally, we were quite surprised by the scale of the weight of the RAVE value giving the best performance, since with a value of 1 it strictly competes with the value of the exploitation term in the UCB formula.

Introducing the RAVE value into the exploitation term

The first results from the use of RAVE for MCTS Amazons are promising. However, as noted in [38], the RAVE term included in the UCB formula should decrease with time. With a RAVE term having a fixed weight, the search could be stuck exploring some moves which look promising in some way but are sub-optimal, and thus lead to wrong results. Ultimately, the exploration term of the UCB formula would take over and allow the search engine to explore moves badly biased by the RAVE term, but the convergence to the min-max values could be hampered if not simply stopped by the RAVE term. For this reason, we proposed to include the RAVE term in the UCB formula in the following way:

$$\frac{\sum_{i=1}^{N}(R(P,i)) + W * Rave(m)}{V(P) + W} + C * \sqrt{\frac{ln(\sum_{siblings\ s} V(s))}{V(P)}} \tag{3.7}$$

with Rave(m) being the rave value computed for move m leading to position P. This implementation is directly inspired of the work of [29], but while the authors included in their formula a value coming from some pre-computed knowledge, we decided to include directly the RAVE term computed on the fly. The idea consists simply in considering that all nodes have been visited W times with an average result of $Rave(m)$ on top of their usual visits. W is a parameter that have to be tuned experimentally. Since it could happen that the RAVE value in this formula is not computed from a sufficient number of results, we also decided to set W in the formula to the number of times the move m from $Rave(m)$ has been evaluated in cases where the latter is inferior to the W used. We then performed the same experiment as in Section 3.7.3. Results are presented in Table 3.6.

Time	40000 samples			5 minutes		
W	Classic evaluation	Complex evaluation	Ratio evaluation	Classic evaluation	Complex evaluation	Ratio evaluation
5	+64	+156	+86	+37	+121	+81
10	+91	+159	+123	+79	+160	+74
15	+64	+163	+134	+65	+128	+132

Table 3.6: Evolution of the ELO rating of our program including the RAVE term as W pre-visits, for various evaluation functions and time settings

It is interesting to note that, unlike the previous implementation, this one only has beneficial effects whatever the value of W tested. This in a way confirms the fact that the bias term introduced in Formula 3.6 can lead to wrong results while in the current implementation, the RAVE term can disappear at some point leading to a better convergence to min-max values.

We also tried to separate the decreasing RAVE term from the classical exploitation term, but none of these experiments brought convincing results. This highlights another use of the RAVE term in this implementation: it reduces the possible variance of the exploitation term of the UCB formula when the number of simulations is low. This observation is also supported by the striking fact that this implementation (unlike the previous one) seems to have a greater effect on our complex evaluation than on the ratio one, the former having a greater variance.

3.8 Other classical improvements

3.8.1 Robust-max

The concept of Robust-max was proposed for the first time by Coulom in [34]. While the implementation at that time was slightly different, the concept stayed similar. It consists, at the top level, when the MCTS process has been stopped and one has to decide a move to play, not to select the one with the highest average evaluation, but the one with the higher count of visits. While the move with the best average evaluation is quite often also the one with the highest visits count (since it has been exploited lots of times), it may happen that both these moves differ. Mostly, this happens when a new move appears superior to the one considered until now, but was not visited as much because of time constraints. In this respect, playing the move with the highest count of visits is a mark of safe play, since it consists in choosing the move in which we have the most trust (which should also be good since it has been exploited by the MCTS process very often).

Since both these implementations make sense in their own way, we implemented Robust-max in CAMPYA and compared the performance of the Robust-max using program to those of the simple program. Results are given in Table 3.7.

Setting	Classic evaluation	Complex evaluation	Ratio evaluation
40000 samples	-24	-16	-23
5 minutes	-42	-18	-28

Table 3.7: ELO gain from the implementation of Robust-max

Although not very clear because the loss in ELO is not very important, these results tend to suggest that the use of Robust-max in Campya is not worth it if not just a bad idea. Since Robust-max is widely used for the game of Go, this was surprising. We imputed at first these results to our implementation of MCTS Amazons, and more specifically to the grouping of nodes in the tree. To confirm this, we also implemented Robust-max first in a version of our program without nodes grouping, and second using a more symbiotic approach with node grouping, consisting in selecting first the Amazons movement with the biggest number of visits compared to other movements, and from there to select the best shoot according to Robust-max. Unfortunately, even without grouping nodes, versions of our program using Robust-max always performed as good or worse than versions not using it (between 0 and 30 ELO of loss depending on the evaluation), and similar results were observed for our hybrid implementation.

3.8.2 Discounted UCB

One of the main differences between the simple N-armed bandit problem and the choice of a move in a game-tree inside the Monte-Carlo framework is that, while the probability distribution of the rewards are stationary for the N-armed bandit problem, they are not in a game-tree. The latter will evolve with the discoveries of new good moves as well as of their potential refutations. This makes later rewards in the MCTS process more significant than earlier ones, since they carry more precise informations. For this reason, we can bias the evaluation of the nodes in the tree to take into account that fact.

Kocsis *et al.* proposed in [49] the discounted UCB algorithm. It consists in computing the average value $V(P)$ of a node P using the following formula, where the $R(P, i)$ is the reward gotten by node P at the i^{th} iteration, N is the number of iterations of the MCTS process at the time the computation is made, and $\gamma < 1$:

$$V(P) = \frac{\sum_{i=1}^{N} \gamma^{N-i} * R(P, i)}{\sum_{i=1}^{N} \gamma^{N-i}} \tag{3.8}$$

To test the efficiency of discounted UCB, we implemented it into our program Campya. Since this implementation is relatively painless (it only requires to modify the update Formulas 3.2 and 3.3), we only measured the gain in performance of this feature in limited time. In this experiment, we set $\gamma = 0.9999$, which means that the visit count of every node will be limited to 10000 (but this will correspond to more than the latest 10000 visits, since each visit except the last one has a weight inferior to 1). Results are summarized in Table 3.8.

Setting	Classic evaluation	Complex evaluation	Ratio evaluation
5 minutes	+7	+27	+31

Table 3.8: ELO gain from the implementation of discounted UCB

The gain in performance is existing, but not convincingly strong. Also, we can see that the gain is higher for more sophisticated evaluation functions. Since the results of those carry more meaning, this is not surprising. We shall conclude from this that discounted UCB is a nice feature to add to an Amazons MCTS engine due to its positive results coupled to its simplicity, but this inclusion should be made accordingly with the evaluation used. One also has to be careful, since this implementation could conflict with other features such as Robust-max (Section 3.8.1).

Chapter 4

Solving and playing Amazons endgames

This chapter is an updated an abridged version of

1. J. Kloetzer, H. Iida, and B. Bouzy: "Comparative Study of Solvers in Amazons Endgames", 2008 IEEE Symposium on Computational Intelligence and Games, Perth, Australia.

2. J. Kloetzer, H. Iida, and B. Bouzy: "Playing Amazons Endgames," *ICGA Journal*, 32(3):140–148, 2009.

For the purpose of building a strong Amazons program, playing well in the opening and in the middle game are not sufficient. Playing optimally in the endgame is also very important if one is to consider beating top human players. However, among techniques varying from precise search engines such as DFPN to sample-based such as MCTS, it is not clear which is optimal to tackle the problem of the endgame in the game of the Amazons. MCTS has been shown effective for solving Go problems [85] but this method is also nowadays the standard for the game itself.

In this chapter, we will study traditional techniques, DFPN [62] and minimax with Alpha-Beta [47] (later abbreviated as only Alpha-Beta), as well as newer ones such as MCTS (as described in Chapter 3), WPNS [80], and Temperature Discovery Search [58] (abbreviated as TDS), with the goal of finding the one leading to the best playing of both single subgames situations and combinations of subgames for the game of the Amazons. The main goal still being to build a strong game playing engine for tournament conditions by adding to it a powerful endgame playing engine, we focus here on realistic endgames situations not specifically balanced for one player and playing in limited time. We shall also see if the new Amazons standard of MCTS can keep up with this difficult task as well as with game-playing.

We will introduce in Section 4.1 basics about Amazons subgames and how to play them well or solve them. Section 4.3 will present our experiments to tackle the task of playing well in a single subgame, while the experiments of Section 4.4 will deal with the task of playing well combinations of subgames. The conclusion follows in Section 4.5.

4.1 Amazons subgames

4.1.1 Amazons endgame is a combination of subgames

The game of the Amazons usually reaches a point where the board is split into distinct regions, each one containing Amazons which can no longer interact with other Amazons in other regions: we will call these regions subgames. For example, the position of Figure 4.1 has four subgames: two one-player subgames on the left side, one subgame on the bottom right, and the last one in the center and around. To play well in such a position, it is necessary both to be able to play optimally (to get the best score) in each subgame and to be able to choose the best subgame to move into.

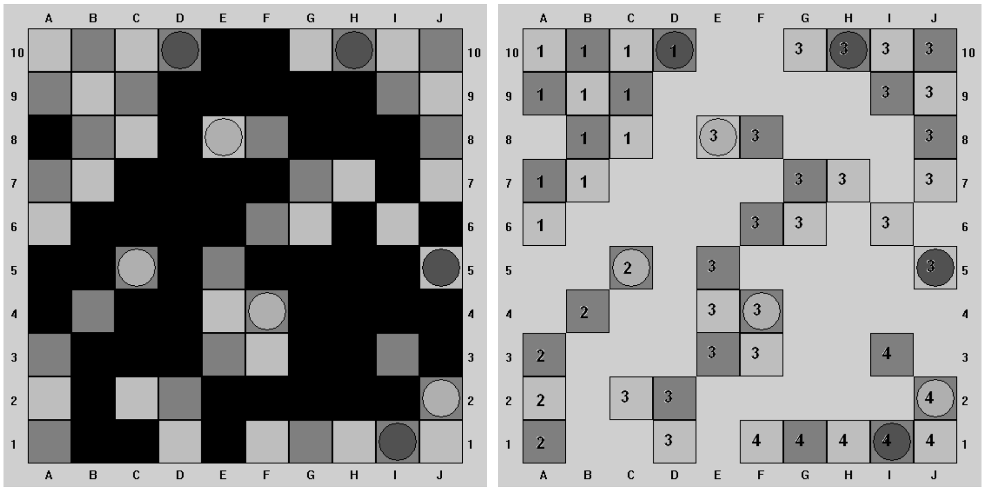


Figure 4.1: Left: An Amazons endgame position; Right: the same position split into subgames

Since those have no interaction whatsoever with each other, it is possible to apply combinatorial game theory [31] to know the value of the main game and which subgame should be played first. Using combinatorial game theory, we can compute for each subgame of a position its true value in terms of *number* (in the combinatorial sense) which, summed together, will give us the exact value (again in terms of *number*) of the main position. From that point, it is possible to choose the move which maximizes the result for the first player as the best move to play.

4.1.2 Solving Amazons subgames using AND/OR Tree-Search

Solving an Amazons subgame falls into one of two categories, depending if the subgame contains one or more Amazons for only one player or for both. With only one player, the goal is to fill the territory of the subgame with the maximum possible number of moves. The difficulty then appears in the case of defective territories, in which the configuration of the Amazon(s) and the arrows does not allow the player to fill every square of it. Defective territories have been presented in Section 2.3.2. Now, if both players have Amazons in the subgame, and if this subgame is considered in isolation from other parts of the game, the goal is to get the maximum moves out of it compared to the number of moves that the opponent will get. Quite often, the situation ends with two or more one-player subgames, and the player with the biggest territory will then win.

AND/OR Tree-Search

AND/OR trees are a specific kind of trees in which leaf nodes are two-valued, 0-false or 1-true, and internal nodes are either OR nodes or AND nodes. Following the same rules as in traditional logic, an OR node is true (has value 1) if at least one of its children is true, and an AND node is false (has value 0) if at least one of its children is false. An AND/OR tree can be used to represent a usual game-tree. In this case, leaf nodes with the value 1 are usually nodes corresponding to a win, while those with value 0 correspond to a loss. As for internal nodes, AND nodes and OR nodes are alternated in the same fashion as minimax with the root node being an OR node. Draws, if they exist, must be handled either as true nodes in which case we will search for strategies giving at least a draw, or as false nodes in which case we will search for strategies giving only a win.

Several algorithms exist to solve AND/OR game-trees, that is to find the value of the root [64]. Among them, the most famous is without a doubt Proof-Number Search. We will present it in the next section as well as two other of its versions, DFPN and WPNS.

On a terminology note, proving a node consists in showing that its value is true (1) while disproving it consists in showing that its value is false (0).

Proof-Number Search

Proof-Number Search (abbreviated as PNS) was developed in 1994 [8] by Allis *et al.*. The main idea of PNS is to make use of Conspiracy Numbers [57], in the present case proof numbers (pn) and disproof numbers (dn), to search first nodes whose value is easier to prove or disprove. The search in PNS is conducted until the value of the root - true or false - is found by iteratively expanding a most proving node, *i.e.* one of the easiest node to prove or disprove.

The proof-number (resp. disproof number) of a node N is defined as the minimum number of leaf nodes which value have to be determined to be able to prove (resp. disprove) N. The following rules are used to defines these numbers:

- For a terminal leaf node, the proof number (resp. disproof number) is equal to 0 (resp. ∞) if the node is a win for the first player, and to ∞ (resp. 0) if the node is a loss.
- For a non-terminal leaf node, both proof and disproof numbers are set to 1.
- For an internal node, the proof and disproof numbers pn and dn are given by the Formulas 4.1 and 4.2.

$$\text{For an OR node: } pn(N) = \min_{c=child(N)} pn(c) \qquad dn(N) = \sum_{c=child(N)} dn(N) \tag{4.1}$$

$$\text{For an AND node: } pn(N) = \sum_{c=child(N)} pn(c) \qquad dn(N) = \min_{c=child(N)} dn(N) \tag{4.2}$$

Simply put, for an OR node, we need to prove only one of its child to prove the node, so its proof number should be equal to the minimum proof number of the child nodes. On the other hand, to disproove an OR node it is necessary to disproove all of this child, thus the sum. By replacing disproof numbers by proof numbers and vice-versa, the same can be said of AND nodes.

Depth-First Proof Number Search

DFPN is a varianr of PNS developed in 2002 [62] using the same theory and algorithm. The first motivation of Nagai when he proposed DFPN, and even PDS (Proof-Disproof Search) before that [61], was to combine the best of both worlds: Proof numbers and their derivatives leading to a quick search, and best first having fewer memory requirements [64]. It notably adds two threshold numbers for each node during the exploration of the tree to facilitate the search of a most proving node. DFPN is known to have solved some of the longest Shogi problem created (like "Micro Cosmos" [62]), thus accrediting its performances.

Weak Proof Number Search

WPNS is another variant of PNS recently developed [80] whose performances have been shown slightly better than those of PNS. It has the main advantage of better handling the presence of Directed Acyclic Graphs (DAG) in the explored tree. Due to the presence of a sum in Formula 4.1, potential transposition nodes are counted twice in the calculus of proof numbers for AND nodes, leading to over-estimations of these values. For this reason, in the original algorithm we just replace Formula 4.2 used to calculate proof numbers for an AND node with this one:

$$pn(N) = \max_{c=childNode(N)} pn(c) + |N.unsolvedChilds| - 1 \tag{4.3}$$

This formula simply replaces the value of the children's proof number by 1 except for the biggest one. Also, the formula calculating disproof numbers for OR nodes is replaced in the same way.

This formula does not anymore represent exactly the number of nodes that have to be proved in order to prove one node, but all in all only ordering of the move is important, not the correct values of the pn and dn for each node.

The two variations of DFPN and WPNS being independent of each other, it is obviously possible to adapt both these methods to create a Depth-First version of WPNS.

Using AND/OR solving techniques for Amazons subgames

The goal of two-player subgames being multi-valued and not two-valued, traditional and/or search methods such as PNS cannot be used as-is. Instead, we must modify their search process as described in [8]: given a result to prove (e.g. white wins by 5 points), we tag leaf nodes as true if their value is equal or superior to this result, and false otherwise. The search for the best result is then an iterative process: we try first to prove the minimum possible result (in Amazons, a loss for the first player by a number of points equal to the number of empty squares in the subgame), then increase it until we can disprove the result to try. The value of the subgame is then the latest result that the search was able to prove. This obviously requires us to be able to order and enumerate the different possible results, which is fortunately the case for all score based games similar to the game of the Amazons.

Specifically for the game of the Amazons, several improvements can be made to speed up the search. This consists mostly in tagging true or false internal nodes of the search tree, either because the result is already proved or because it will be impossible to prove it. Considering a position where we want to prove that a player P can win by N points, points, we propose the following rules:

- After player P passes, the node can be tagged as false.
- If player P has been able to play N moves after a pass of his opponent, the node can be tagged as true.
- If player P is to move while his opponent did not yet pass and there are strictly less than N empty - or reachable - squares on the board, the node can be tagged as false.

4.1.3 Playing Amazons endgames

Playing well in Amazons endgames supposes to play well for two tasks: the first one, as presented above, is to play well into a single subgame, be it one- or two-player. For this task, it is possible either to use a traditional solver (adapted with the procedure presented in Section 4.1.2) like Proof-Number Search (PNS) [8] or some of its variants, or to use a more traditional Amazons engine, be it a traditional Alpha-Beta engine [51] or a more recent MCTS engine [45, 54].

Next to playing well in one subgame remains the problem of what move to select if we play in a position composed of several subgames. Two methods can be used here: the first one is to use traditional game-engines like presented before to handle the task of choosing the right subgame while at the same time choosing a move. The second one is to choose a subgame first and then use any method to select a move inside the chosen subgame. While most algorithms perform a search over the full board, it is possible to decompose the task with the help of a heuristic such as Temperature-Discovery search.

We will now describe several of the algorithms that can be used in such a situation.

The Alpha-Beta method

Despite not being specifically designed for solving, game playing engines based on iterative deepening, minimax, an evaluation function and the Alpha-Beta method [47] can also be used for that task. The latter's main purpose is to cut parts of huge game-trees and it is nowadays universally used in Tree-Search based game playing engines.

The Monte-Carlo method

As for the Alpha-Beta method, despite not being designed for solving, the Monte-Carlo method can be used to find solutions to game problems. Its main weakness is obviously that it will return an approximation of an evaluation, while when solving problems we deal with exact solutions, resulting in an inability to "catch" the mathematical precision beneath that task. But, despite this drawback, this method has already shown some good results: Zhang *et al.* show in [85] that MCTS with UCT is very efficient to produce Go capturing problem solvers. In this last assertion, solving a problem has to be understood as being able to play well in said problem from the beginning to the end. The reader can refer to Chapter 3 for more details on MCTS.

The Hotstrat heuristic

Hotstrat is a strategy that was proposed by Conway in [31]. Facing a sum of subgames, it consists in playing a move in the subgame with the highest temperature. The latter is an indication of how important it is to move first in a subgame with respect to the others. While the temperature of a subgame is pretty demanding in terms of computational power, Müller *et al.* proposed in [58] a heuristic used to approximate it called Temperature Discovery Search (TDS). A program using it then chooses a move in a combination of subgames in two steps: first it computes the temperature (or an approximation) of each subgames, then performs a local search inside the subgame to find the best move. This last search can be performed with a traditional Alpha-Beta engine. This way, it is possible to run a local and deeper search for the best move in a single subgame rather than considering the whole position. For reasons of simplicity, we will call later TDS the combination of Hotstrat, Temperature Discovery Search and a local search using minimax with Alpha-Beta.

Proof-Number Search

While playing is not the first goal of PNS, it is perfectly possible to use it (or any of its variations, e.g. DFPN) as a playing engine. It suffices to play from a position the move returned as the move leading to a win by a PNS solver, or none of them if the position is disproved. In the case of multi-valued games, the same way as described in Section 4.1.2, a simple heuristic is to play the move leading to a win for the best result that the search was able to prove.

4.2 Building a problem database

Unlike more popular games such as Chess, Checkers, Shogi or Go and despite some previous work on Amazons endgames [59], there is no easily accessible database of problems for the game of Amazons. Moreover, such collections of problems are usually written by human experts of the game and provide interesting problems with a good quality. Such problems usually have a unique solution (or few of them). That aspect is important in the field of game programming because it allows the programmer to check easily the righteousness of a program's or algorithm's answer.

For the purpose of this study, we had to build such a problem database. But we need to define first what kind of problems we will deal with.

4.2.1 Amazons problems

The main part of the game which can provide us with interesting problems for the game of Amazons is the endgame. The main difference with other traditional games is that the goal in Amazons endgames is to fill (if only one side is involved) or to gain (if both sides are involved) territory: solving a problem consists in finding out the two values of the position, for each player moving first and, in a game-oriented perspective, one of the best moves for each player.

To build this problem database (which is, in fact, a positions database), we focused on some distinct aspects:

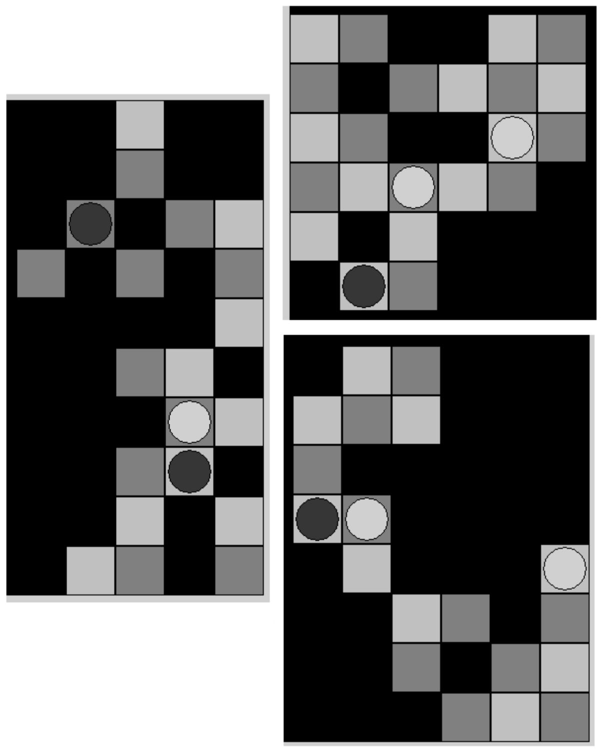


Figure 4.2: Examples of Amazons endgames positions

- To help analyzing the strengths or weaknesses of every assessed method, problems should be of various sizes and contents.
- Difficulty of problems (mostly in term of size) should be capped. Not because too difficult problems would be uninteresting, but because we are dealing with limited time and with more practical than theoretical situations, and also because we need to solve the problems before utilizing them in experiments.

4.2.2 Creating the problems

Since creating a problem database by hand was out of the question, there were basically two main possible choices to create the problems:

- Use a routine to create positions automatically
- Extract positions from Amazons game scores

Since there is no easy way to create a ”position-making” routine and that we wanted to cope with practical situations, we chose the second solution. We used the following procedure:

- Fix a set of features for the problems we want to extract (number of Amazons, number of empty squares, possibility of movement of each player...)
- Read game scores until a territory containing the correct set of features is created (see Figure 4.3)

- Extract the position corresponding to the territory with the position of the Amazons, and go back to reading

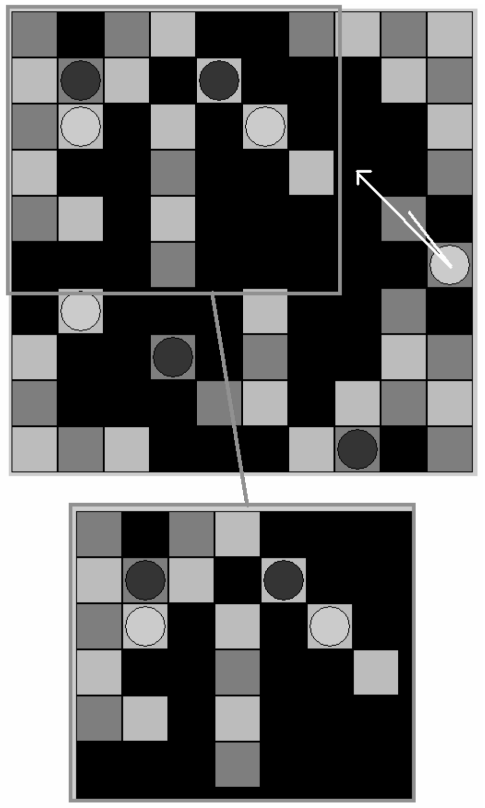


Figure 4.3: In search for positions containing 2 White and 2 Black Amazons. After the last move (showed in White), the part on the top left is isolated from the rest of the board and contains the correct number of Amazons for each player: it is extracted and saved in the database.

Each position extracted can give birth to two problems, considering which player plays first. This simple procedure, provided that we have a various set of game scores and a correct set of features, should allow us to extract various non-artificial positions and to attain our goal of creating a good problems database.

Following this procedure, we created a database of around 1300 realistic positions. These positions are of size (number of empty squares) 2 to 50, and each of them gave birth to 2 problems (with each side going first). We had to reject around half of the problems either because we did not had enough of them to represent their size or because we were unable to solve them in reasonable time. This left us with approximately 1300 remaining problems of size 2 to 21 (not to be confused with the original 1300 positions). The distribution of these problems can be seen in Figure 4.4.

4.3 Playing right in one subgame

The following sections will describe our experiments to tackle the problem of playing well into a single subgame.

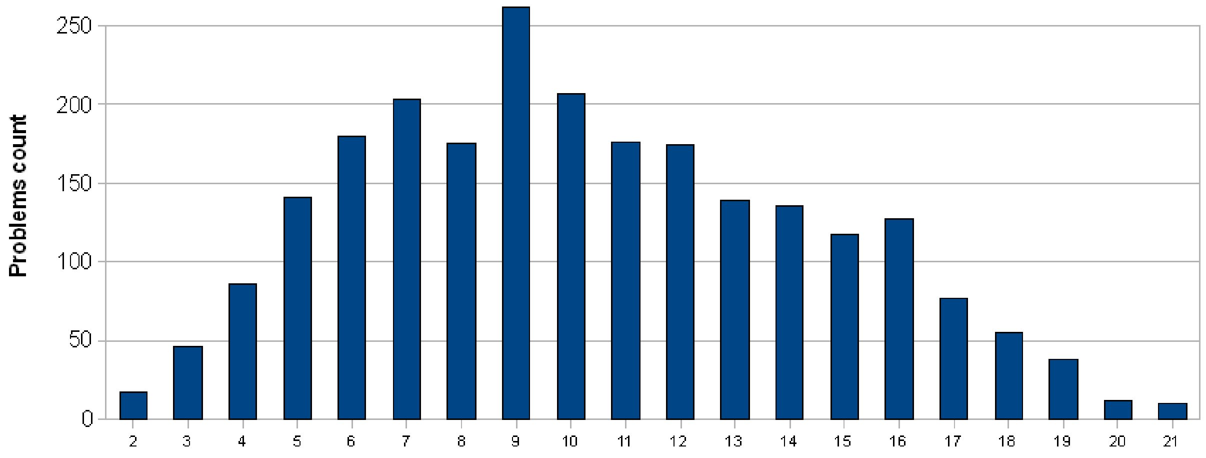


Figure 4.4: Problems count by size

4.3.1 Experiment settings

For this experiment, we compared 2 traditional solvers and 2 game playing engines. The first two are a traditional DFPN solver and a depth-first version of a WPNS solver, both described in Section 4.1.3. Both are using the Amazons specific improvements presented in Section 4.1.2. The other two are a traditional Alpha-Beta game playing engine (using Iterative Deepening, a Transposition Table, move ordering mostly based on the moves evaluations, and forward pruning such as described in [10]), and an MCTS engine based on our program Campya (which main features are described in Appendix B). Both use the same evaluation function based on the accessibility as an approximation of the territory (see Section 2.2.2).

All four algorithms were evaluated on the database of problems presented in Section 4.2. Each of the algorithms was given the same quantity of time: 5, 10 or 20 seconds depending on the setup, to find a move for each problem.

4.3.2 Assessing the different methods involved

Each of the algorithms was assessed the same way: for a given problem, we first get a move out of each of them (the best move chosen by a game playing engine, or the move proving the best result obtained in the iterative process for the solvers). Then, if that move is not found in our database, we evaluate the position resulting from playing it with a more powerful solver - a traditional PNS solver given unlimited time - and compare it to the value of the original problem. The value can only be equal - in which case one of the best moves was chosen - or less - in which case we can set the difference between both results as the error made by the algorithm assessed.

Using the procedure presented in Section 4.1.2 to solve a problem we get two informations: the best move selected by the solver and the last result that the solver was able to prove in its iterative process. This is, in a way, the evaluation of the problem by the solver. This evaluation can be exact or just a minimum. We could assess both our solvers - DFPN and WPNS - using this evaluation; however, we decided against it and chose to evaluate the same way solvers and game-playing engines as presented above. The first reason is to keep the comparison clear by comparing similar informations, and the second is because there are situations in which a solver can easily prove that a move can win by a certain score but not prove that this move is optimal for the best score (see Figure 4.5 for an example). In this case, assessing a solver by its returned evaluation would be unfair to it. Since our first goal is to play well in the endgame, it made sense to only consider the move returned by a solver to assess it.

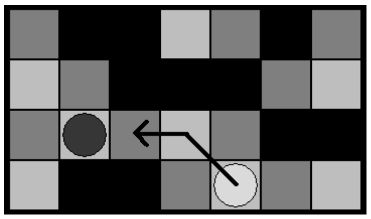


Figure 4.5: Proving that this move leads to a win in this position is less difficult than to prove that it indeed leads to a win by 5 points

4.3.3 Results and discussion

Table 4.1: Percentage of problems correctly answered by each algorithm and (average error) for those wrongly answered

	5 seconds	10 seconds	20 seconds
DFPN	92.09 (3.49)	93.65 (3.26)	95.12 (2.97)
WPNS	95.92 (2.39)	97.18 (2.34)	98.28 (2.37)
Alpha-Beta	97.31 (1.25)	97.64 (1.27)	97.81 (1.25)
MCTS	94.74 (1.66)	94.87 (1.66)	95.41 (1.64)

Table 4.1 gives the percentage of problems correctly answered by each algorithm, given different quantities of time. Unsurprisingly, most of the problems being of reasonable size, they are easily tackled by all of the algorithms.

Compared to the standard which is Alpha/Beta search, the performances of MCTS are somehow disappointing: it makes more errors and those are usually more important in terms of the score difference to the optimal value. It seems that MCTS scales slightly better, but 20 seconds for a move is already close to what a program will get in the endgame in tournament conditions.

However, considering that the iterative procedure used to solve a multi-valued problem is quite time-consuming, the performances of both PNS-based solver are very good. Traditional Proof-Number Search (here represented by DFPN) shows some weaknesses but shines with the improvements given by WPNS. In practice, we observed that most of the iterations take little if not no time at all and that both solvers usually use most of their time on the most difficult iteration. Also, the approximation made by playing the move given by the last iteration solved seems to function pretty well as long as the solver gets enough time. Otherwise, it may make some pretty bad mistakes, as shown by the average error (in brackets in Table 4.1). Finally, the improvements to the solving process presented in Section 4.1.2 were necessary: the performances of these solvers dropped by more than 10% if for instance the last of the three was missing.

Finally, since the game of the Amazons permits the apparition of transpositions, and so of Directed Acyclic Graphs into the game-tree, we had expected WPNS to perform better than DFPN on this testbed, and were not disappointed: although the results of WPNS are not as good as those of Alpha-Beta, the improvement in terms of performances over DFPN is quite clear. But, when comparing directly the results of DFPN and WPNS, it appears that DFPN was able to get strictly better results than WPNS on 17 problems, while WPNS got strictly better results on 57 problems: This shows us that WPNS is not strictly better than DFPN, and that the method should be tested more deeply before saying that it is a clear improvement over PNS.

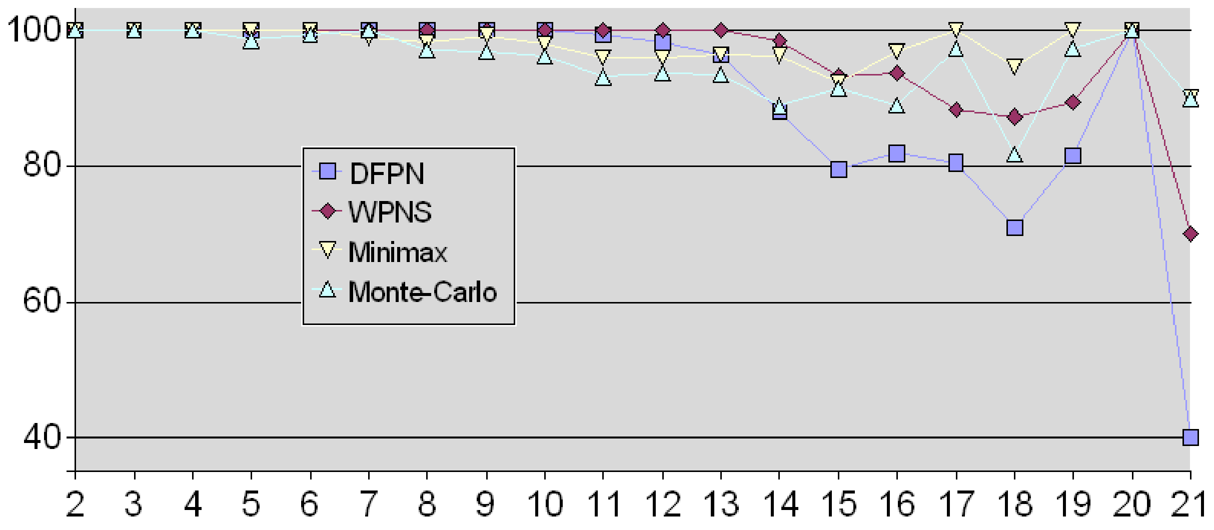


Figure 4.6: Percentage of problems correctly answered (20 seconds time)

Now, looking at the more detailed results of Figure 4.6, we can also see that the performances of the PNS-based solvers tend to slowly decrease after a certain size (10-11) while still being better up to around size 15. On the other hand, the performances of the game playing engines, even if weaker for even smaller problems and always irregular, tend to stay stable whatever the size and are better for size 16 and onward. Also, the simple fact that we had to reject around half of our problems of size 20 and more because we were not able to solve them in reasonable time (problems not included in the present results, mentioned in Section 4.2) confirms the lack of performances of PNS based algorithms to handle bigger size problems.

There are unfortunately few problems of size 18 or more in our database, so the results for these sizes are far from being statistically representatives. The fact that all problems of size 20 are perfectly solved by every algorithm for example is a fortuitous consequence from the simple fact that our 13 problems of size 20 were even easier to solve than some of the smaller ones. However, the decrease in performances of DFPN and WPNS for problems or size 12 and more is quite clear, so we will for now extrapolate for bigger sizes.

These results suggest that the inclusion of a solver could be a welcome improvement in any Amazons program to correctly play in the endgame once the play reaches difficult small positions. In this case, some knowledge could also be added from the game playing engines themselves, like what is done in DFPN+ in [62].

These results also tend to confirm those of another recent study [85], which showed good results for the Monte-Carlo method to play Go problems. However, their results for the Alpha-Beta method, outperformed both by DFPN and badly by Monte-Carlo, do not concur with the present ones. The most plausible explanation to this behavior would be the presence, in the case of our testbed, of a good evaluation function in the Alpha-Beta engine. Although it is possible to create a good evaluation function for the game of Amazons, even if it is a simple one, this task is much more complicated for the game of Go, even for a task as simple as endgames. So even if both algorithms (Iterative Deepening with Alpha-Beta and MCTS with UCT) are powerful enough by themselves, it clearly seems that using an evaluation function is a necessary need for the game of Amazons to build a strong game playing engine.

Finally, with slightly worse performances overall for all algorithms, the same conclusions can be drawn for time settings of 5 and 10 seconds. For this reason, we chose to present only the results for 20 seconds.

4.4 Playing the big picture

Since playing the real Amazons game involves playing a combination of subgames in the endgame, we also experimented with those. The settings and results of such experiments are presented in the next sections.

4.4.1 Experiment settings

To create a set of positions to use for this experiment, we used combinations of positons taken from the database presented in Section 4.2. For that, we randomly associated together positions of any size to fit into a classical 10×10 Amazons board, including those of large size not mentionned in the results of Section 4.3 because we were unable to solve them in limited time and possible one-player subgames. We created this way 300 Amazons endgames positions, split into sets of a hundred each consisting respectively of 2, 3 and 4 subgames. The number of Amazons in each positions is not fixed and does not go over 4 Amazons for each player to respect the rules of the game.

We used three game playing engines in this experiment: an Alpha/Beta engine, an MCTS engine (both presented in Section 4.3.1), and a minimax game playing engine based on TDS and the Hotstrat strategy as presented in Section 4.1.3. For the latter, Alpha/Beta is used after the computation of the temperatures to select a move into the hottest subgame.

We compared directly the performances of each algorithm by making them play against each other. Each pair of engines played in total four games for each position of our endgames set: each endgame was played with both sides playing first, each engine then playing first two times. A time limit was set to 10 seconds per move for each match.

4.4.2 Results and discussion

The results of this experiment in terms of numbers of games won by the first player are given in Table 4.2 for each of the programs AB (Alpha/Beta), MC (Monte-Carlo Tree-Search) and TDS. It should be noted that the positions created for this experiment are not specifically fair and do not give the same fighting chances to both players. For this reason, we also included AB-AB matches to have a reference value to compare the other results to.

Table 4.2: Percentage of problems won for the first player of each pair tested

1st player	2nd player	2 subgames	3 subgames	4 subgames	All problems
AB	AB	41%	40%	51%	44%
AB	MCTS	40.5%	37.5%	45%	41%
MCTS	AB	42.5%	42%	51.5%	45%
MCTS	TDS	44%	42.5%	56%	47.5%
TDS	MCTS	40%	36%	45.5%	41.5%
AB	TDS	42.5%	41.5%	52%	45.5%
TDS	AB	39.5%	38%	48.5%	42%

From these results, we can easily infer that MCTS performs slightly better than Alpha/Beta, and that both perform better than TDS.

Unexpectedly, TDS could not get good performances against both other playing engines while it is specifically designed for this task. We associate these results to the time used in the pre-computations necessary for TDS to work correctly. The results presented in [58] were much more in favor of TDS, but the test-bed was different: the problems used then were much more balanced and perhaps more difficult to handle with simple Alpha/Beta search, while at the same time allowing to TDS the possibility to shine.

On the other hand, the Alpha-Beta and the MCTS based engines seem to have comparable performances on most of the problems, with a slight advantage to the MCTS engine, advantage which appears bigger with more subgames. This result can be explained by the usual better performances of MCTS in more general situations and its better ability to "catch the big picture", and that even if it lacks the precision of Alpha-Beta or other solvers such as DFPN for playing the subgames themselves right.

Since most of the positions of the test-bed are unfair for one of the players (as can be seen by the AB/AB comparison), we cannot base our results on comparing only the percentages of victory of each engine: we also have to compare the results of each of them on each single problem. For that, given a single problem and two engines, we compared the score obtained by both engines with each of them playing first against the other and noted which got the best score. A summary of these results is given in Table 4.3.

This comparison tends to validate both the performances presented above for all number of subgames and, when looking at the average difference in score between each engine, the ability of Alpha-Beta based engines (both traditional minimax search and TDS) to be more precise in their search, leading to overall better average scores.

Analyzing these results through other angles, we also discovered a relation with the size of the problems for the AB-MC comparison: Alpha/Beta performs better than MCTS on bigger problems. For problems of size up to 25, MCTS performs better than Alpha/Beta on 60 problems versus 46 problems with Alpha/Beta getting better results (and 158 drawn problems), while for problems of size 30 and more, these results go to 65 for MCTS versus 76 for Alpha/Beta (and 198 drawn problems). While lots of the problems are still draws,

Table 4.3: For each pair A-B of engines: with A and B alternatively playing first, number of problems with A getting a better score than B / A and B getting the same score / B getting a better score than A. On the second line: average score difference for each of these problems.

Pair	2 subgames	3 subgames	4 subgames	All problems
AB-MC	51 / 108 / 41	44 / 103 / 53	68 / 56 / 76	163 / 267 / 170
	3,82 / - / 2,02	2,57 / - / 1,83	2,68 / - / 2,79	3,01 / - / 2,31
MC-TDS	46 / 107 / 47	60 / 107 / 33	106 / 48 / 46	212 / 262 / 126
	2,76 / - / 4,13	2,83 / - / 2,73	3,21 / - / 2,8	3 / - / 3,28
AB-TDS	49 / 114 / 37	66 / 105 / 29	109 / 54 / 37	224 / 273 / 103
	2,86 / - / 2,14	2,55 / - / 1,79	2,94 / - / 1,95	2,81 / - / 1,97

this contrasts with the results observed before showing that MCTS performs better when the number of subgames is larger. Still, this is not contradictory, since it suggests that Alpha/Beta performs better against MCTS with 2 long subgames than with 4 shorter ones, and vice-versa for MCTS.

4.5 Conclusion

With the main overall goal of improving the playing level of Amazons programs, we have focused here on the task of playing well endgames situations. Since this problem is related to combinatorial game theory, we have compared in this chapter the performances of several algorithms handling two tasks: playing well single subgames or combinations of subgames. In both aspects, Alpha/Beta search with an evaluation function appears as a good standard to play correctly in Amazons endgames and does not need specific improvements to play combinations of subgames such as TDS. Nevertheless, it also appears that PNS-based solvers could be a welcome inclusion in any Amazons program to play perfectly in smaller subgames. Some progress can also be made to improve these solvers, as shown by the very good results of WPNS in this study. Finally, even if it lacks the precision of the former engines, the new MCTS engines seem better suited to play combinations of subgames,a task for which the order of play is important.

On the future of this topic, we hope to improve the precision of our MCTS engine so that it could surpass Alpha-Beta in all ways, being able to play as good in a single endgame while at the same time having better performances in playing right a whole game consisting of several subgames. On possibility could be to modify MCTS to handle the task of choosing a subgame, leaving to a more precise engine such as PNS the task to find the best move in the area in a short amount of time. Also, building a database of endgame positions could be a welcome inclusion for all of the tested algorithms to shorten the search and improve their precision.

Chapter 5

Learning of an Opening-Book for a Monte-Carlo based program

5.1 Introduction

Creating an Opening-Book for a game playing program has always been a natural follow-up of said program achieving good performances. Game programs are traditionally good in the middle game and have specific algorithms or strengths to deal with endgame positions. However, all this will be rendered useless if the program makes blunders in the opening stage of the game and attain a position that it will have difficulties to recover from.

Game programmers usually either create Opening-Books by hand (using expert knowledge) or automatically using tree expansion algorithms and an evaluation function. Automatic creation have been at some point limited by constraints of memory, but technological advancement now permits computers to store a big Opening-Book and access it quickly without the problems computers faced in the past. Now, on the utility of such Opening-Books (whatever the creation method) one should remember that a good Opening-Book - a book containing good moves according to some authority like human experts - does not always make a program more efficient: it can actually also make it weaker. If a programs uses an Opening-Book whose main variation leads to a position that the program cannot handle well, the addition of said Opening-Book will be definitely bad for the program. An Opening-Book has to be adapted to the program it is attached to.

Since our program CAMPYA has already achieved a good playing level (refer to Chapter 3 and Appendix B), building an Opening-Book for it is a natural next step. But, it should be noted that, as an MCTS-based programs, it has a playing style which is very different from traditional minimax programs: we can thus expect that Opening-Books created for more traditional programs will be less efficient for it, perhaps even leading to deterioration in terms of performances.

This section will present our work towards creating an Opening-Book for MCTS programs. We will use the game of the Amazons as a testbed. After this introduction, Section 5.2 will focus on Opening-Books, what they are and what algorithms are used to create them. In Section 5.3, we will present how to adapt an MCTS algorithm named UCT to create Opening-Books for CAMPYA, followed by experiments and results in Section 5.4. Finally, the conclusion will follow in Section 5.5.

5.2 Creating Opening-Books

An Opening-Book for a computer program is basically a game-tree with annotated nodes (an evaluation or any other kind of knowledge), and the root being the initial position of the game. It should contain moves that are expected to be played, either because they are strong or for any other justifiable reason (preferred moves of some specific opponents, moves hard to refute in limited time...) that would make the program play stronger. Without this main idea in mind, an Opening-Book is rendered useless, since at the first moment an opponent plays a move which is not in book it is of no use anymore. We say in this case that the opponent played out of book. An Opening-Book can be used in two main ways: the first one is to just pick the best move when the program finds itself into a position which is in the book, and the other one is to use the information in the book to support a search with more expert knowledge.

Opening-Books are usually built in one out of two ways, or a mix of both. Lincke [52] distinguishes between passive and active book creation methods: if available, one can create an Opening-Book by hand using expert knowledge. This is usually done by human experts of the game, and is the active method. If this knowledge is not available or if one just wishes so, it is possible to create an Opening-Book automatically using a specific algorithm. Karpetian *et al.* argue that a mix approach is best, since any automatic construction method could give birth to overlooked defects. One of such algorithm for automatic construction will be presented in Section 5.2.1, while Sections 5.2.2 and 5.2.3 will deal more specifically with the game of the Amazons and the problem of the evaluation of Opening-Books.

5.2.1 A simple best-first Opening-Book creation algorithm

The following algorithm was presented first by Buro in [21], and discussed and refined in [52]. In this algorithm, the Opening-Book is expanded iteratively using an evaluation function, usually provided by the program using the Opening-Book. It consists in a simple best-first algorithm: as noted by Buro, if we have a method to select moves we can use it to create a process of automatic expansion of a game-tree. However, as Buro notes, an Opening-Book needs deviations. For this he proposes to add to every node in the tree a next-best node, making so that each node of the tree has at least a leaf node as a brother node. The pseudocode of his algorithm is given in Figure 5.1

The function used in this algorithm to evaluate moves in not specifically a simple positional evaluation. It can be the result of a deeper search, or any more sophisticated approach as long as it returns an evaluation for the various moves tried.

One of this algorithm's main weaknesses is its lack of exploration, as discussed in [52]. All in all, an Opening-Book will be useful only if you play against the right opponents, that is against opponents who play moves which are in the book. In that sense, a good Opening-Book should not only contain good moves or only moves that we want our programs to play, but also moves that we expect the opponent to play. However, it is in the nature of best-first search to find itself in situations where it will not expand nodes expected to be played by the opponent. If a node deep in the tree is found with a heuristic value superior to the value of a child node of the root, no matter how small the difference may be, a depth-first search algorithm will try to expand the former. This can lead to disastrous results in terms of balance of the Opening-Book. Playing with such a book, a

```
tree = {root}
while ( process continues )
  currentNode = root
  while ( has_children( currentNode ) )
    currentNode = get_best_child( currentNode )
  end while
  firstNode = get_best_child_with_evaluation( currentNode )
  tree->add( firstNode )
  secondNode = get_next_best_child_with_evaluation( currentNode )
  tree->add( secondNode )
  currentNode->update_min_max_value()
  currentNode = currentNode->parent
  if ( all_children_are_expanded( currentNode ) )
    tree->add( get_next_best_child_with_evaluation( currentNode ) )
  endif
  while ( currentNode != root )
    currentNode->update_min_max_value()
    currentNode = currentNode->parent
  endwhile
endwhile
```

Figure 5.1: Pseudocode for Opening-Book creation

smart opponent could choose to play one of these child nodes of the root which value is almost optimum, and would be trading a small difference in terms of evaluation (that is, if the latter is correct) for the big gain of playing an out of book position, rendering the Opening-Book useless. To avoid such situations, if one wishes to create an Opening-Book automatically (that is, not by hand) some exploration component should be present in the algorithm to deal with this phenomenon.

As they noticed that the very concept of best-first violates the goal of maximizing the number of expected moves in book, Lincke *et al.* proposed in [52] the drop-out expansion. Their basic idea is to penalize nodes which are deep in the tree to favor exploration of shallower nodes. They propose to create an Opening-Book in such a way that getting out of book early should be more strongly penalized. This way, if the opponent plays such variations, the drawback of not having them inside the Opening-Book will be compensated by the fact that the variation played by the opponent is bad. This idea is again, of course, tied to the assumption that the evaluation function used is correct. But, since all automatic Opening-Book creation methods use at some point or another an evaluation function, we can for now consider that assumption as valid.

Lincke *et al.* notice that the main benefits of the drop-out expansion are the following:

- "Slightly worse" moves are not ignored forever during the book creation process
- We dispose with the drop-out expansion of a parameter to control the shape of the book (which can be a good point for the book evaluation, since it provides the programmer with a tool to control the shape of his book)
- The drop-out expansion gives an insurance against the opponent's moves

5.2.2 Creating an Opening-Book for the game of the Amazons

Karapetyan *et al.* study in [43] the performances of the algorithm presented in Section 5.2.1 for the game of the Amazons. Their test program is a traditional program with minimax-based search and an evaluation function, Invader.

Their results are quite mitigated: although the application of the algorithm described in Section 5.2.1 was successful to build an Opening-Book, the results were not clear, mostly since it is difficult to evaluate the efficiency of such an Opening-Book. Nevertheless, some of the points they raised specific to the game of the Amazons are to be noted:

- Creating an efficient Opening-Book for the game of the Amazons is a difficult task because of the high branching factor of the game (no less than 2000 moves from the initial position). Also, since long branches are not very practical for the game of the Amazons, the book should be made wide and shallow.

- Creating an Opening-Book for the game of the Amazons using only the best-first approach described in Section 5.2.1 gives unbalanced books which are not very practical. Since the drop-out expansion does not seem to give good results, Karapetyan *et al.* propose two concepts calleddepth-penalty and width-penalty to create more balanced trees.

- The creation method should be automatic. There is no expert knowledge available for the game, making this property necessary. This also agrees with what Buro claims in [21], in that programs nowadays should be able to update their books without human intervention. However, the authors also point that it should be easy to correct the Opening-Book by hand. Although this argument is valid, the authors also admit that the playing level of their program is much stronger than theirs, so whether correcting the book automatically generated by a stronger program by hand is the right thing to do or not is not clear.

- A correction to the evaluation is necessary. Most programs of Amazons share the property that the evaluation at odd and even depth vary a lot: this is known as the odd-even effect, and while it is present in lots of games, it is particularly acute for the game of Amazons. Thus, if one is to use an evaluation function, nodes evaluations should be compared at the same parity level: all odd, or all even. In the case of minimax-searches, making a 2-deep search takes a short amount of time but a 3-deep search may require too much time. To deal with this case, the authors propose an approximation consisting in doing first a 2-deep search, and then if the seach ends at an odd level, to expand and evaluate the leaf node of the principal variation to get a feeling of what the real minimax value should be at the odd level. This is just an approximation but, according to the authors, is much more effective to use than just to ignore the odd-even effect.

It should also be noted that the game of the Amazons brings another benefit to the creation of an Opening-Book. Since the game allows no draw, there is no chance that the creation process of such a book would be stuck into one variation contending both players. For this reason, it is possible to automatically create an Amazons Opening-Book without worrying about that aspect (which could very well appear by using best-first).

5.2.3 Evaluating Opening-Books

Evaluating an Opening-Book is a hard task. Indeed, it will help a program to play good moves in the first stage of a game only if the opponent cooperates, that is, if he plays moves which are in-book. As soon as a game is out of book, the latter is rendered useless. So, an Opening-Book's value is always correlated to the field of opponents facing the program using it. A book fitted to play against other computer programs could very well be useless against more adaptive opponents using different playing styles, that is human players.

For this reason, evaluating Opening-Books should always include some human check to testify that the book is fit to the environment it will be used in. Automating the evaluation by testing it against a various field of opponents is a natural first step to judge an Opening-Book and get a first feeling of it but, unless the field is really that various, should not be sufficient to validate the efficiency of such a book.

This step could be necessary however in the case where the creation method of the Opening-Book and the playing algorithm of the program using it show discrepancies. This is especially true for Opening-Books created by hand. As Buro notes in [21], a program should understand its winning chances after being out of book. But it could happen that, while a sequence of moves is very good in the hand of a human player, a computer program playing the same sequence will not be able to exploit it at its fullest, thus potentially leading to a disaster. In that case, evaluating through real games the performances of a program using the Opening-Book becomes a necessary step before any use of said book can be done in the program.

5.3 Using UCT to learn an Opening-Book

Details about the UCT algorithm can be found in Chapter 3. We will in this section discuss the possible implementation of meta-UCT, an algorithm using a UCT program to play pseudo-random games. Next to this, we will discuss the use of the UCT algorithm to the creation of Opening-Books.

5.3.1 Meta-UCT

Since the idea of Monte-Carlo Tree-Search is to play stochastic games and average their results to compute node evaluations, one could want to improve the quality of these random games to improve in turn the quality of the Tree-Search above it. While this has been tried a lot for game playing programs, it appears that including knowledge can worsen the performances of the program more than improve it. Two reasons for that: first, the knowledge included may bias the results of the Tree-Search in a wrong way and second, including too much knowledge may have a bad effect on the performances of the engine - that is, the number of stochastic samples per time unit could decrease too much compared to what is gained by adding the knowledge.

Extremely, one could use a full program to replace the random policy used to play samples games in MCTS, be it UCT or anything else. In the former case, we would obtain a Meta program, that is an MCTS program using another MCTS program to play pseudo-random games. Some have already mentioned using a UCT program instead of such random games, for example to solve puzzles such as Morpion solitaire [23] or Kakuro [24]. The pseudocode of such a meta-UCT program is given in Figure 5.2.

```
function getBestMoveMetaUct(Position, Endingcondition)
  initialize(tree, Position)
  while ( Endingcondition not satisfied )
    endingnode = tree.root
    Pos = copy(Position)
    while ( endingnode is in the tree )
      endingnode = chooseChildOf(endingnode)
      play(Pos, move leading to endingnode)
    end while
    tree.add(endingnode)
    while ( Pos is not ended position ) \\ definition to be determined
      play(Pos, getBestMoveUct(Pos)) \\ game in the Meta program
    end while
    V = metaEvaluation(Pos) \\ evaluation to be determined
    while( endingnode =/= tree.root )
      update(endingnode, V)
      endingnode = endingnode.fathernode
    end while
    update(root, V)
  end while
  return(move m with highest value)
end function
```

Figure 5.2: Pseudocode for Meta-MCTS

5.3.2 Why use UCT?

Classical Tree-Search algorithms based on simple minimax with an evaluation function have, as seen before, a very limiting aspect: the lack of exploration in the process of the Opening-Book creation. The UCT algorithm, on the other hand, has been designed to not forget moves by using a bias factor, and is widely recognized as being relatively robust against adversity, because of this high exploration rate.

Since the algorithm is now accepted as a search engine as a concurrent of the traditional minimax method, there is a real possibility that it would perform well even for a task such as Opening-Book creation. Also, if one wants to create an Opening-Book for a Monte-Carlo Tree-Search based program, creating the book with an algorithm similar to the one used in the playing engine should surely give better results than one using a completely different method.

One could argue that, since the UCB formula relies on a constant which has to be tuned to lead to better performances of play, the algorithm is not well suited to the task: indeed, making an Opening-Book takes much more computation time than just getting a move from a position of the game, thus making the optimization of the constant a process too long to consider. Furthermore, as we have seen in Section 5.2.3, there is no easy way to evaluate an Opening-Book, necessary step to optimize the constant. However, since the value of the book is determined mostly by what we expect from it, we claim that there is no real need to optimize this constant for such a task. If the book obtained through an experiment does not fit what is expected from it, we can perfectly change the constant to allow more or less exploration and thus get a book satisfying our expectations (a task for which the actual processing powers of computers obviously helps).

Finally, as we noticed in Section 5.3.1, using too heavy random games can hinder an MCTS program, both because of a possible bias and because it may slow it down. However, we are here searching for a high level of knowledge, so if bias there is then the program is faulty. As for the processing power needed, this is not a real problem since the task of building an Opening-Book is not real time anyway.

5.4 Experiments and results

We will first discuss how to evaluate Opening-Books in Section 5.4.1. Then, the next sections 5.4.2 to 5.4.5 will be devoted to experiments made using UCT or some variations of it. Performances obtained in tournament conditions will follow in Section 5.4.6.

5.4.1 Opening-Book evaluation

For the reasons discussed in Section 5.2.3, we evaluated the various Opening-Books created during the following experiments in two ways. First, we integrated them in our program Campya and evaluated the performances of this new version of the program by playing against a field of various opponents (though all based on our program Campya; see Section 3.4.1 for a description of our experimental methodology). Since the program tested uses fast time settings, even a little bit of time gained using a book could lead to much better performances, so we should be careful looking at these results. But still, this will give us a first glance at evaluating Opening-Books. This performance based experiment is always realized using two versions of our program. The first one plays with a fixed number of random games per move and the second one with a fixed time per game. The first version's results should show if the program played better moves, while the second's should show if indeed the inclusion of the Opening-Book to gain time leads to better play in tournament conditions.

Second, if the performances of a book seem good, we will take a closer look at the book created and analyze it more carefully to see if, from a human perspective, the book could be useful if used in a field with more various opponents: how deep it is, are there various variations for the moves, are the most popular moves in the book etc.

5.4.2 Experiment 1: using best-first

This first experiment was made to validate the point that a book created with a method leading to a different playing style than the one used by the program using it would not lead to good results.

We created three Opening-Books with the algorithm described in Section 5.2.1 of respectively 500, 1000 and 1500 nodes each. We evaluated nodes as proposed in [43] by making a two-depth minimax search with a possible adjustment to avoid a possible odd-even effect as presented in Section 5.2.2, and a penalty based on the depth of the nodes to be evaluated. The evaluation function used for this experiment was the one used in the program Campya. We then introduced these three Opening-Books into our program Campya, and measured the difference in rating with the original version with both unlimited time (40000 samples per move) and limited time (5 minutes per game). The results are summarized in Table 5.1.

Creation	Number of nodes	Unlimited time	Limited time
Best-first	500	-186	-110
Best-first	1000	-227	-167
Best-first	1500	-184	-138

Table 5.1: Difference in ELO with the program version not using the Opening-Book

Even if, from a human perspective, the book obtained with this method seem to contain reasonable moves, the performances-based experiment definitely says otherwise: all versions of the program using any of these books have their performances drop much below the performances of the standard program without the book (between 100 and 230 ELO). This result validates out hypothesis that a book created with a method different than the one that the program uses to play the game will not lead to good results.

These results could be biased by the fact that the node evaluation, the result of a possibly corrected two-depth search, is not similar to the evaluation from our program. For this reason, we also experimented with an evaluation returned by our program after a fixed number of iterations of the MCTS process. However, the Opening-Books created with such settings were no better than the one using the two-depth search as an evaluation. We conclude from this experiment that best-first search is the main responsible for the non adaptation of these Opening-Books to our MCTS based program. There is no evidence however that the evaluation function is faulty in any way, so we will continue to use it.

5.4.3 Experiment 2: building a tree using plain UCT

One natural step to make an Opening-Book with UCT was to just let our program expand a tree from the initial position for a very long period of time, and take the tree at the end of the computation as our Opening-Book. This was our first approach of the problem with UCT and is a very simple solution. We ran several experiments with different time settings (30 minutes, 2 hours and 5 hours - more lead us to memory problems), and pruned at the end of the computations all nodes which did not satisfy a minimum number of visits (50 or 100) to keep only reliable evaluations estimations.

Unfortunately, the trees created this way have a major issue: however high we set the exploration value in the UCB formula, the number of simulations (around 5 millions for a 30 minutes experiment) is so high that at some point the algorithm just exploits its results and do not explore anymore at the root node. This in turn causes the algorithm to concentrate most of its computations to a single move (4053038 visits out of 4544998, so 89% of the visits for one of our 30 minutes experiments), causing the tree to be completely unbalanced. Also, the performances of the program using such books just did not change (47% to 52% win against a version of the program not using a book). We can conclude from this that, since a huge number of simulations does not help in UCT, we should probably use a smaller number of heavier playout.

5.4.4 Experiment 3: using Meta-MCTS

The main idea behind using meta-MCTS to build an Opening-Book is that the knowledge contained in it is quite high-level and should be accurate. For that, an evaluation based on moves played by a good-level program should be much better than just random moves, even if the process takes more time.

Several questions were raised to use meta-MCTS:

- Concerning the samples evaluation
  - For how long should we play sample games? Until the end of the games like in the traditional MCTS approach, or for a few number of plies like what is used in Monte-Carlo Amazons programs ?
  - How should we evaluate terminal positions? With 0/1 (for loss/win), or using an evaluation function?
- Concerning nodes expansion/pruning: since the evaluation process takes a long time, the computation will only be done for a few thousands of samples. This in turn means that, since there are more than 2000 moves in the initial game position, we should prune most of them. Moreover, a good number of them will not be played by players with a good skill,so we just do not want them in the book.
- Since we have time for long computations, it is possible to pre-initialize the values of the nodes of the game tree in UCT. This is usually done using some expert knowledge or patterns in game playing engines, and can perfectly be adapted to our case; but there remains the following questions: what kind of knowledge to use and how ?

For this first series of experiments, we decided to prune nodes using a simple evaluation of the nodes at depth 2. This evaluation is also used as a pre-evaluation for each node in the UCB formula: for a node with no visit, which usually make the computation of their father's UCB evaluation impossible, the score given by the formula is set to the value of this evaluation. This use of the evaluation is similar to what the developers of the Go program Mogo called First Play Urgency, as presented in [82, 40].

Six different Opening-Books were created using this setup, with various settings, which are summarized in Table 5.2. We then introduced these Opening-Books in our program Campya, and measured the difference in rating with versions of the program not using them, both with unlimited (40000 samples per move) and limited time (5 minutes per game), with the same setup as the experiment in Section 5.4.2. The results of this comparison are given in Table 5.3.

Even with different settings, it was not clear which settings gave the best results both in terms of performance and of human evaluations. However, certain patterns emerged:

- With the possible exception of book #5, all books created were beneficial to the program using them, with a 50 ELO of improvement for the best versions .
- As for an Amazons game playing engine itself, it is better to play a few moves and then evaluate rather than play a full game. At least, the book created with this setting gave better quality moves (measured by the performances with unlimited time). Also, from a human perspective, the moves contained in such books look more reasonable.

Book index	Number of nodes	Exploration factor in UCB	Evaluation of the sample games	Time for a sample game
1	1000	5	evaluation after 20 moves	60 seconds
2	1000	8	evaluation after 20 moves	60 seconds
3	1000	5	evaluation after 20 moves	120 seconds
4	1000	8	evaluation after 20 moves	120 seconds
5	1000	5	win/loss after a full game	60 seconds
6	1000	8	win/loss after a full game	60 seconds

Table 5.2: Settings for the Opening-Books created with Meta-MCTS

Creation	Book index	Unlimited time	Limited time
Meta-MCTS	1	+2	+49
Meta-MCTS	2	+24	+28
Meta-MCTS	3	+15	+33
Meta-MCTS	4	-8	+35
Meta-MCTS	5	-30	+45
Meta-MCTS	6	+12	+38

Table 5.3: Difference in ELO with the program version not using the Opening-Book

The main variations of Opening-Books 1, 2 and 5 can be found in Appendix C.

5.4.5 Experiment 4: mixing Meta-MCTS with the traditional approach: Meta-UCT

In the end, experiments using meta-UCT seemed to lead to honorable results both in terms of performances and of adaptation of the Opening-Book created to the program using it. However, the previous experiment is based on a very simple form of static pruning and cannot really generalize to anything.

To solve this problem, and directly inspired from the classical Opening-Book creation algorithm presented in [21], we propose the algorithm of Figure 5.3, that we will call Meta-UCT. The idea is to use the same principle as meta-MCTS. But in the expansion phase, instead of expanding all nodes satisfying a certain condition (that is, pruning all the others), to expand the best one and the next best one (for comparison purposes in UCT) with their respective evaluations. Also, we use here direct evaluations instead of sample games, as described in Section 3.5.2.

For the purpose of testings, we used our program Campya as an evaluation tool by doing an MCTS run of 200000 iterations. The selection of the best leaf nodes and deviations was made through a simple minimax search at depth 2, with the adjustment presented in Section 5.2.2 to avoid possible odd-even effects. The reason why we used such a search and not a Monte-Carlo search is that we needed a value both quickly computed and precise. While Monte-Carlo can quickly order move, its evaluations are not that precise with short computations.

Finally, to take into account the fact that we use here a more precise evaluation, the UCB formula was tweaked to

$$\frac{\sum_{t=1..T} R(t,i) + K * E(i)}{\sum_{t=1..T} C(t,i) + K} + A * \sqrt{\frac{ln(T)}{\sum_{t=1..T} C(t,i)}}$$

where K was a fixed constant (that we varied in the experiments) and $E(i)$ is the evaluation given to the node i by the minimax search at the time it was selected to be added to the tree. The settings are summarized in Table 5.4. Results using a similar procedure as described in Section 5.4.4 are given in Table 5.5.

Book index	Number of nodes	Exploration factor in UCB	Constant K
7	1000	2	5
8	1000	5	5
9	1000	2	2
10	1000	5	2

Table 5.4: Settings for the Opening-Books created with Meta-UCT

The various Opening-Books created using these settings had a much better quality compared to the ones of the previous experiments in terms of performances, and without

```
function createOpeningBook(Position, Endingcondition)
  initialize(tree, Position)
  set(tree.root, PERMANENT)
  while ( Endingcondition not satisfied )
    endingnode = tree.root
    Pos = copy(Position)
    while ( hasChildsInTheTree(endingnode) ) \\ Select the best leaf node
        with UCB
      endingnode = chooseChildOf(endingnode)
      play(Pos, move leading to endingnode)
    end while
    if ( endingnode is PERMANENT)
      firstNode = getBestChild(endingnode)
      tree.add(firstNode) \\ Add its best child to the tree
      V = evaluate(firstnode) \\ Evaluate the child and set it to permanent
      update(firstNode, V)
      set(firstnode, PERMANENT)
      tree.add(getNextBestChild(endingnode, tree)) \\ Add a non permanent
          sibling node
    else
      V = evaluate(endingnode) \\ Evaluate the node itself if not permanent
      set(endingnode, PERMANENT)
    end if
    update(endingnode, V)
    endingnode = endingnode.fathernode
    if ( allChildArePermanent(endingnode) )
      tree.add(getNextBestChild(endingnode, tree))
    endif
    while( endingnode =/= tree.root )
      update(endingnode, V)
      endingnode = endingnode.fathernode
    end while
    update(root, V)
  end while
end function
```

Figure 5.3: Pseudocode for Meta-UCT for Opening-Book creation

Creation	Book index	Unlimited time	Limited time
Meta-UCT	7	+7	+24
Meta-UCT	8	+96	+45
Meta-UCT	9	+90	+65
Meta-UCT	10	+8	+133

Table 5.5: Difference in ELO with the program version not using the Opening-Book

the need of any static pruning. In addition, from a human perspective, they seem to have a much nicer shape and much more reasonable moves and variations.

However, it should be noted that books created with this technique and an exploration factor too low (in the present case, almost four times less as in the playing engine with a similar evaluation function) are very narrow. At the root node, only three child nodes are expanded, making very easy to get in out-of-book positions. Book #10 surely gets it good performances in limited time from the fact that it is used in a field of similar programs playing similar moves, but this will not be verified in tournament conditions. The best compromise in the present case to set the exploration factor of UCB to a higher value (5, so still around two times less as in Campya itself) and the K constant quite high too (5 in our case).

The main variations of Opening-Books 8 and 10 can be found in Appendix C.

5.4.6 Results in tournament

We also had the opportunity to test the most promising Opening-Book created in real tournament conditions. This tournament was part of the Computer Olympiad, organized in 2009 in Pamplona (Spain). Four programs entered the tournament, including Campya. Four games were played against each opponent with 30 minutes of time for the whole game for each side.

Regarding more specifically the Opening-Book, the results were as follow:

- Against 8 Queens Problem, a traditional minimax algorithm using an evaluation function quite different from the one used in Campya (from the word of the author), the games were in-book for a very short period of time: 3 moves when Campya played first, and 2 when playing second. At least, we had an answer for every second move of 8 Queens Problem.

- Against Invader, which also uses Monte-Carlo Tree-Search but with a completely different evaluation function, the games were in-book for 2 moves when playing second, and respectively 3 and 5 moves when playing first. Since both programs have very different playing styles, this 5 moves deep variation was a good surprise, especially since Campya won this game. Furthermore, one of the games with only 2 moves in book was a variation that our program did not care to explore (it was only 2 moves deep), and Campya won that game too, which is a good sign.

- Against Bit, a classical minimax program with an evaluation function from which we have few informations, the respective games stayed in our book for 3 and 8 moves when playing first, and 3 moves when playing second (that is, the 3rd move, played by Bit, was inside our book, but we had no answer to it). Considering that once again the playing style of Bit is quite different from that of our program, this is another really good surprise.

As for the quality of the moves themselves, even if we had good news especially with both won games against Invader, all participants agreed that, more than the first few moves of the game, the outcome will mostly be determined during the middle game; and so that the quality of the moves in-book cannot be clearly drawn from the outcome of the games. However, the time saved during the opening phase will only be beneficial for a program using an Opening-Book to help during the middle game, so we were quite pleased with the results in real tournament conditions.

5.5 Conclusion

We presented in this section different ways of building Opening-Books for programs based on the Monte-Carlo Tree-Search framework. Results show that using in an MCTS program an Opening-Book created using the traditional best-first algorithm leads to bad performances, probably because of the difference in playing style between minimax based programs and Monte-Carlo based programs. We then proposed a new algorithm for automatic Opening-Book learning, based on the MCTS algorithm with UCT.

Future work include testing the proposed methods for other games (more specifically, the game of Go would be a good testbed) and compare the results to traditional Opening-Book learning techniques. Moreover, finding a better way to evaluate Opening-Books would be nice, but we have few hopes in the matter.

Chapter 6

Conclusion and future works

In this dissertation, we have studied how Monte-Carlo based methods could be adapted to create a strong playing program for the game of the Amazons, as well as several various enhancements to increase the general playing level of such a program. In this section, we will summarize our results and answer to the research questions mentioned in Section 1.3.2. In Section 6.1, we will present our general answer to the problem of adapting Monte-Carlo techniques for the game of the Amazons as well as some more specific details on how this can be achieved. In the next Section 6.2 we will summarize our study of the playing of Amazons endgames and how Monte-Carlo techniques fit in it, while in Section 6.3 we will present our results on how to create an Opening-Book for Monte-Carlo based programs, more specifically for the game of the Amazons. Finally, after a summary in Section 6.4, we will present in Section 6.5 possible future works in the domain of Monte-Carlo Amazons.

6.1 Adapting Monte-Carlo for the game of the Amazons

Research questions: Is it possible to build a strong Amazons playing program using Monte-Carlo Tree-Search (MCTS) techniques ? Could techniques used for the game of Go or other games be applied for an MCTS program playing Amazons? Could traditional improvements for the game of the Amazons be used in a Monte-Carlo based program?

As we have seen in Chapter 3, while it is possible to directly apply traditional Monte-Carlo techniques with Tree-Search for the game of the Amazons, the resulting programs are very weak, lacking both strategical and tactical sense. Our proposed idea of grouping nodes does help, but not to the point of making an MCTS program able to compete even against intermediate human players.

To reach our goal, a major improvement was necessary. We proposed for that to implement an evaluation function as it is done in traditional minimax-based Amazons programs and use it to evaluate random games, thus breaking with the traditional evaluation paradigm of Monte-Carlo. We showed that, while this removes most of the stochastic aspect of Monte-Carlo which usually makes its strength, it also allows us to gain much more informations from the random games played and thus drastically improves the performance of our program. In a way, this is the Amazons way of gaining more informations while in other games, pseudo-random games are usually played [82]. However, we also showed that while reduced, the stochastic aspect of Monte-Carlo cannot be completely removed. Obviously, major changes to the implementation in the future could prove us otherwise.

We also presented possible ways of integrating the AMAF (or RAVE) heuristic to help drive the search in an MCTS framework. While our proposed implementation differs from what is usually done for example for the game of Go [38], the results are very promising whatever the evaluation function on which the program is based. We also studied the possible improvements provided by the Discounted UCB technique as well as Robust-max.

In the end, with the correct improvements we were able to create an Amazons game playing program competitive enough to defeat traditional minimax-search based programs (results in competitions are presented in Appendix B). To reach this goal, we used techniques taken from the minimax world (an evaluation function, grouping nodes..) and proposed ways to adapt them to create a strong MCTS program. We also showed how to adapt several now standard techniques for Monte-Carlo based programs (mainly AMAF). But it should be noted that almost none of these could be applied as-is, and that while the game of the Amazons have elements from both the world of territory games like Go in which MCTS succeeds and from piece-moving games like Chess in which traditional search techniques succeeds, it also has its own small world in which all these techniques need to be adapted in order to create a strong program.

6.2 Getting the best of endgames

Research questions: Is MCTS able to handle precise playing like one finds in endgame situations? Are specific improvements needed?

To answer this question, we proposed in Chapter 4 a comparative study of Monte-Carlo with other traditional search techniques used to solve or play endgames. Since Amazons endgames combine the tasks of playing well both in a single subgame and confronted with a sum of subgames, it made sense to compare Monte-Carlo to some top techniques for both these tasks.

It appears that, while our Monte-Carlo based program obtained strictly inferior results to several top of the line solvers for the single subgame solving task, its results were not much worse than what can be obtained with Alpha-Beta of DFPN even without specific improvements. We also showed in this study how it is possible to use a traditional AND/OR tree solving algorithm such as PNS to play well in a endgame in short time, provided that the endgame position is not too big: up to size of around 15, our implementation of the WPNS technique could play perfectly almost all of the positions in our problems database with less than 5 seconds to decide for a move.

We also showed that, despite having slightly inferior results in this first task, our Monte-Carlo program held its own in the task consisting of playing well in combinations of subgames, surpassing Alpha-Beta in this case. Since playing well in a single subgame is a sub-task of this problem, this means that the playing strength of Monte-Carlo is good enough to compensate for this lack of success in single-subgame solving, clearly showing the good strategical sense of a Monte-Carlo based implementation.

6.3 Creating Opening-Books for an MCTS Amazons program

Research questions: Is it possible to directly use standard Opening-Books of more traditional programs for Monte-Carlo based programs? If not, is it possible to adapt them to a different playing style? Is it possible to design more specific Opening-Books or methods to create them for Monte-Carlo based programs?

Since MCTS programs usually have a different playing style compared to traditional minimax programs, we expected that using Opening-Books designed for the latter family of programs would not improve the performance of MCTS programs. Our experiment presented in Chapter 5 confirmed that Opening-Books created using traditional methods based on the minimax paradigm are not adapted to Monte-Carlo Amazons programs. Even more than that, these books deteriorate their performance.

To solve this problem, we proposed several ways inspired from the MCTS framework to automatically create Opening-Books adapted to MCTS programs for the game of the Amazons. Our first idea is based on a concept usually called Meta-MCTS and shows promising results, but is difficult to adapt directly to other games without enough trials and errors. Our second proposed idea, named Meta-UCT, also shows promising results without needing that much experiment. From the success that MCTS gets for other games such as the game of Go, we believe that our technique can also be applied for various games to create Opening-Books adapted to Monte-Carlo based programs.

6.4 Summary

We presented in this dissertation a study of Monte-Carlo techniques for game playing and how they can be adapted to create a strong Amazons playing program. We also showed how MCTS could handle tasks complementary to game playing, namely creating Opening-Books and playing well endgames. Finally, we also presented several improvements that can be made to such a program to increase its level. While we did not succeed in creating a program able to defeat all potential human or computer opposition, we showed how it was possible to build an MCTS based Amazons program which could handle its own against traditional top of the line minimax-based programs as well as against strong human players. Considering the youth of such Monte-Carlo techniques compared to what has been done in the field of minimax search, we believe that Monte-Carlo Amazons program could very well become the new reference in the world of Computer Amazons.

6.5 Future works

While our program Campya already attained a good playing level, we believe some more work can be done to improve its efficiency:

- While the use of an evaluation function was a major boost in terms of performances compared to traditional MCTS implementations, the computation time needed to compute this function is still a bottleneck. This is mainly due to its main component consisting in a search for shortest path from Amazons to empty squares. Unlike other classical values such as material balance of liberty count, this value cannot be easily incremented or decremented after each move and have to be recomputed each time. Finding either a new and quicker evaluation function based on different features or another way to compute the actual one would undoubtedly be a boost in terms of performances.

- As mentioned in Chapter 4, the creation of an endgames database as well as the inclusion of a precise solver for Amazons endgames would be a welcome inclusion to handle the tactical part of playing endgames correctly.

- As it is done for other games such as Go or Lines of Action, it should be possible to create pseudo-random games including some kind of knowledge to help the evaluation process of MCTS. The sheer number of possible moves and the difficulty to easily compute informations from a board state has until now thwarted our tries to implement knowledge into the random games to bias them, but we lose no hope in the matter.

Some other possible works are more tied to the MCTS engine and less specifically to the game of the Amazons:

- While the UCT algorithm returns great results considering the amount of work that one has to invest for it, we believe that some improvements could be made to it both in its exploration and exploitation parts. The game of the Amazons having a huge branching factor, it is unfortunately not rare that an MCTS program misses a good move because other good moves exist which are not as good. On the other hand, and once again because of this branching factor, exploring deep nodes is often a fruitless task since the search will never gather enough information to make it worthwhile, thus more exploitation should be needed. Ultimately, we would like an algorithm exploring more, while being able to exploit more quickly good moves, for example good answers to previous moves. Something along the line of what is done in Realization Probability Search [79] could be a possibility.

Bibliography

[1] The arimaa challenge. `http://arimaa.com/arimaa/challenge/`.

[2] Computer go mailing list. `http://computer-go.org/pipermail/computer-go/`.

[3] Private communication.

[4] Computer beats pro at u.s. go congress, 2008. `http://www.usgo.org/index.php?\%23_id=4602`.

[5] B. Abramson. Expected-outcome: a general model of static evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(2):182–193, 1990.

[6] S.G. Akl and M.M. Newborn. The principal continuation and the killer heuristic. In *Proceedings of the ACM 1977 annual conference*, pages 466–473. ACM New York, NY, USA, 1977.

[7] L.V. Allis and U.R. Limburg. *Searching for solutions in games and artificial intelligence*. PhD thesis, Maastricht: Rrijksuniversiteit Limburg, 1995.

[8] L.V. Allis, M. van der Meulen, and H.J. van den Herik. Proof-number search. *Artificial Intelligence*, 66(1):91–124, 1994.

[9] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47(2-3):235–256, 2002.

[10] H. Avetisyan and R.J. Lorentz. Selective Search in an Amazons Program. *Lecture Notes in Computer Science*, pages 123–141, 2003.

[11] D. Billings, A. Davidson, J. Schaeffer, and D. Szafron. The challenge of poker. *Artificial Intelligence*, 134(1-2):201–240, 2002.

[12] B. Bouzy. Associating domain-dependent knowledge and Monte Carlo approaches within a Go program. *Information Sciences*, 175(4):247–257, 2005.

[13] B. Bouzy. Associating Shallow and Selective Global Tree Search with Monte Carlo for 9 x 9 Go. *Lecture Notes in Computer Science*, 3846:67, 2006.

[14] B. Bouzy. Move-Pruning Techniques for Monte-Carlo Go. *Lecture Notes in Computer Science*, 4250:104–119, 2006.

[15] B. Bouzy and T. Cazenave. Computer Go: an AI oriented survey. *Artificial Intelligence*, 132(1):39–103, 2001.

[16] B. Bouzy and G. Chaslot. Bayesian generation and integration of K-nearest-neighbor patterns for 19x19 go. In *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games*, pages 176–181. Citeseer, 2005.

[17] B. Bouzy and B. Helmstetter. Monte-Carlo Go Developments. In *Advances in Computer Games: Many Games, Many Challenges: Proceedings of the ICGA/IFIP SG16 10th Advances in Computer Games Conference (ACG 10), November 24-27, 2003, Graz, Styria, Austria*, pages 159–174. Kluwer Academic Publishers, 2004.

[18] B. Brugmann. Monte carlo go. *Manuscript available by Internet anonymous file transfer from bsdserver. ucsf. edu, file Go/comp/mcgo. tex. Z*, 1993.

[19] M. Buro. The Othello match of the year: Takeshi Murakami vs. Logistello. *ICCA Journal*, 20(3):189–193, 1997.

[20] M. Buro. Simple Amazons Endgames and Their Connection to Hamilton Circuits in Cubic Subgrid Graphs. *Lecture Notes in Computer Science*, pages 250–261, 2001.

[21] M. Buro. Toward opening book learning. *Advances In Computation: Theory And Practice*, pages 81–89, 2001.

[22] M. Campbell, A.J. Hoane, and F. Hsu. Deep blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.

[23] T. Cazenave. Reflexive monte-carlo search. In *Computer Games Workshop*, pages 165–173. Citeseer, 2007.

[24] T. Cazenave. Monte-Carlo Kakuro. In *Proceedings of 12th Advances in Computer Games Conference (ACG12)*, 2009.

[25] T. Cazenave and N. Jouandeau. On the parallelization of UCT. In *Computer Games Workshop*, pages 93–101. Citeseer, 2007.

[26] T. Cazenave and N. Jouandeau. A parallel Monte-Carlo tree search algorithm. *Computers and Games*, 5131:72–80, 2008.

[27] G. Chaslot, C. Fiter, J.B. Hoock, A. Rimmel, and O. Teytaud. Adding expert knowledge and exploration in Monte-Carlo Tree Search. In *Proceedings of 12th Advances in Computer Games Conference (ACG12)*, 2009.

[28] G.M. Chaslot, M.H. Winands, and H.J. Herik. Parallel monte-carlo tree search. In *Proceedings of the 6th international conference on Computers and Games*, pages 60–71. Springer-Verlag, 2008.

[29] G.M.J. Chaslot, M.H.M. Winands, H. Herik, J. Uiterwijk, and B. Bouzy. Progressive Strategies for Monte-Carlo Tree-Search. *New Mathematics and natural Computation*, 4(3):343–357, 2008.

[30] B.E. Childs, J.H. Brodeur, and L. Kocsis. Transpositions and Move Groups in Monte Carlo Tree Search. In *Proceedings of the IEEE 2008 Symposium on Computational Intelligence and Games, Perth*, 2008.

[31] J.H. Conway. *On Numbers And Games*. Academic Press, 1976.

[32] R. Coulom. Bayeselo, 2005. `http://remi.coulom.free.fr/Bayesian-Elo`.

[33] R. Coulom. Computing elo ratings of move patterns in the game of go. In *Proceedings of the Computer Games Workshop*, pages 113–124, 2007.

[34] R. Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Lecture Notes in Computer Science*, 4630:72–83, 2007.

[35] R. Coulom and K. Chen. Crazystone wins 9x9 Go Tournament. *ICGA Journal*, 29(3):96–97, 2006.

[36] H. Finnsson and Y. Björnsson. Simulation-based approach to general game playing. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI*, pages 259–264, 2008.

[37] T. Furtak, M. Kiyomi, T. Uno, and M. Buro. Generalized Amazons is PSPACE-Complete. In *International Joint Conference on Artificial Intelligence*, volume 19, pages 132–137. Lawrence Erlbaum Associates Ltd, 2005.

[38] S. Gelly and D. Silver. Combining online and offline knowledge in UCT. In *Proceedings of the 24th international conference on Machine learning*, pages 273–280. ACM Press New York, NY, USA, 2007.

[39] S. Gelly and Y. Wang. Exploration exploitation in go: UCT for Monte-Carlo go. In *Twentieth Annual Conference on Neural Information Processing Systems (NIPS 2006)*. Citeseer, 2006.

[40] S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of UCT with patterns in Monte-Carlo go . Technical Report 6062, INRIA, November 2006.

[41] J. Hashimoto, T. Hashimoto, and H. Iida. Context Killer Heuristic and its Application to Computer Shogi. In *Computer Games Workshop, Amsterdam, The Netherlands*, pages 61–68, 2007.

[42] T. Hashimoto, Y. Kajihara, N. Sasaki, H. Iida, and J. Yoshimura. An evaluation function for amazons. *Advances in Computer Games*, 9:191–202, 2001.

[43] A. Karapetyan and R.J. Lorentz. Generating an Opening Book for Amazons. *Lecture Notes in Computer Science*, 3846:161–174, 2006.

[44] J. Kloetzer. Amazons in Pamplona: Invader Confirms its Power. *ICGA Journal*, 2010. to be published.

[45] J. Kloetzer, H. Iida, and B. Bouzy. The Monte-Carlo Approach in Amazons. In *Computer Games Workshop, Amsterdam, The Netherlands*, pages 113–124, 2007.

[46] J. Kloetzer, H. Iida, and B. Bouzy. A Comparative Study of Solvers in Amazons Endgames. In *Proceedings of the IEEE 2008 Symposium on Computational Intelligence and Games, Perth*, 2008.

[47] D. Knuth and R. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975.

[48] L. Kocsis and C. Szepesvari. Bandit Based Monte-Carlo Planning. *Lecture Notes in Computer Science*, 4212:282–293, 2006.

[49] L. Kocsis and C. Szepesvári. Discounted-ucb. In *2nd PASCAL Challenges Workshop*, 2006.

[50] C.S. Lee, M.H. Wang, G. Chaslot, J.B. Hoock, A. Rimmel, O. Teytaud, S.R. Tsai, S.C. Hsu, and T.P. Hong. The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):73–89, 2009.

[51] J. Lieberum. An evaluation function for the game of amazons. *Theoretical Computer Science*, 349(2):230–244, 2005.

[52] T.R. Lincke. Strategies for the Automatic Construction of Opening Books. *Lecture Notes in Computer Science*, pages 74–86, 2001.

[53] R. Lorentz. First-time Entry Amazong wins Amazons Tournament. *ICGA Journal*, 25(3):182–184, 2002.

[54] R. Lorentz. Amazons discover Monte-Carlo. In *Computers and Games, Beijing, China, September/October 2008*, pages 13–24, 2008.

[55] R. Lorentz. Invader wins Amazons Event. *ICGA Journal*, 2009. to be published.

[56] H. Matsubara, H. Iida, and R. Grimbergen. Natural developments in game research: From Chess to Shogi to Go. *ICCA Journal*, 19(2):103–112, 1996.

[57] D. A. McAllester. Conspiracy numbers for min-max search. *Artificial Intelligence*, 35:287–310, 1988.

[58] M. Müller, M. Enzenberger, and J. Schaeffer. Temperature discovery search. In *Nineteenth National Conference on Artificial Intelligence (AAAI 2004)*, pages 658–663, 2004.

[59] M. Müller and T. Tegos. Experiments in computer amazons. *More Games of No Chance*, pages 243–260, 2002.

[60] P.A.C.R. Munos and P.A. Coquelin. Bandit Algorithms for Tree Search. *Arxiv preprint cs/0703062*, 2007.

[61] A. Nagai. A new AND/OR tree search algorithm using proof number and disproof number. In *Proceedings of Complex Games Lab Workshop*, pages 40–45, 1998.

[62] A. Nagai. *Df-pn Algorithm for Searching AND/OR Trees and Its Applications.* PhD thesis, Department of Information Science, University of Tokyo, 2002.

[63] J.T. Saito, M.H. Winands, J. Uiterwijk, and H.J. van den Herik. Grouping nodes for Monte-Carlo tree search. In *Proceedings of the Computer Games Workshop*, pages 125–132, 2007.

[64] M. Sakuta and H. Iida. And/Or Tree-Search Algorithms in Shogi Mating Search. *ICGA Journal*, 24(4):218–229, 2001.

[65] J. Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11):1203–1212, 1989.

[66] J. Schaeffer, Y. Björnsson, N. Burch, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen. Solving checkers. In *International Joint Conference on Artificial Intelligence*, volume 19, pages 292–297. Citeseer, 2005.

[67] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007.

[68] J. Schaeffer, J.C. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53(2-3):273–289, 1992.

[69] J. Schaeffer and H.J. Van den Herik. Games, computers, and artificial intelligence. *Artificial Intelligence*, 134(1-2):1–7, 2002.

[70] C.E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41(4):256–275, 1950.

[71] B. Sheppard. World-championship-caliber Scrabble. *Artificial Intelligence*, 134(1-2):241–275, 2002.

[72] D. Silver and G. Tesauro. Monte-Carlo simulation balancing. In *Proceedings of the 26th Annual International Conference on Machine Learning*. ACM New York, NY, USA, 2009.

[73] H.A. Simon and A. Newell. Heuristic problem solving: The next advance in operations research. *Operations research*, 6(1):1–10, 1958.

[74] D.J. Slate and L.R. Atkin. Chess 4.5the Northwestern University chess program. *Chess skill in Man and Machine*, pages 82–118, 1977.

[75] O. Syed and A. Syed. Arimaa - a new game designed to be difficult for computers. *ICGA Journal*, 26:138–139, 2003.

[76] I. Szita, G. Chaslot, and P. Spronck. Monte carlo tree search in settlers of catan. In *Proceedings of 12th Advances in Computer Games Conference (ACG12)*, 2009.

[77] G. Tesauro. Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, 134(1-2):181–199, 2002.

[78] F. Teytaud and O. Teytaud. Creating an Upper-Confidence-Tree program for Havannah. In *Proceedings of 12th Advances in Computer Games Conference (ACG12)*, 2009.

[79] Y. Tsuruoka, D. Yokoyama, and T. Chikayama. Game-tree search algorithm based on realization probability. *ICGA Journal*, 25(3):132–144, 2002.

[80] T. Ueda, T. Hashimoto, J. Hashimoto, and H. Iida. Weak Proof-Number Search. In *Computers and Games, Beijing, China, September/October 2008*, pages 157–168, 2008.

[81] T. Ugajin and Y. Kotani. The improvement of playout using transition probability of Monte-Carlo Shogi (in Japanese). In *Proceedings of the 14th Game Programming Workshop in Japan 2009*, pages 107–110, 2009.

[82] Y. Wang and S. Gelly. Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, pages 175–182, 2007.

[83] M.H.M. Winands. 8QP wins Amazons Tournament. *ICGA Journal*, 30(3):163, 2007.

[84] M.H.M. Winands and Y. Björnsson. Evaluation Function Based Monte-Carlo LOA. In *Proceedings of 12th Advances in Computer Games Conference (ACG12)*, 2009.

[85] P. Zhang and K. S. Chen. Monte-Carlo Go Tactic Search. In *Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)*, pages 662–670, 2007.

Appendix A

The ELO rating system

A.1 What is the ELO rating system ?

The ELO rating system is the name given to a method used to rank players of certain games or sports created by Arpad Elo, a Hungarian born American professor (for reasons of readability, we will stick to writing ELO when speaking of the rating system). This rating can also be used to predict results of confrontations between two players or teams rated with it. While it was first created to be used for the game of Chess, for which it is still widely used, it is now also used for other games, whether they are deterministic or not, and even by some sports associations.

Using the ELO system, each player in the given environment is given an ELO rating (we will simply speak of rating from now on) which is a simple numerical value. Each player's rating evolves with the strength of the player and its performances against other rated people, and the difference between the rating of two players provides an indication of the expected result of a match between the two players.

A player's rating is only a useful information if compared to the rating of other players, which means that any scale could do for this system. Elo designed his system so that a difference of 200 points between the ratings of two players means that the expected result of a match between these two players is 0.75. With a win awarded 1 point and a loss 0, this means that we would expect the best of the two players to win 75% of his matches against the lower ranked one. Draws are not directly included in this score, but are considered as half a win and half a loss. The possible score of the match (for games such as Go, Othello or Amazons) does not matter. Later, the United States Chess Confederation (USCF) aimed for average Chess club players to get a rating of around 1500.

Nowadays, how much the ELO rating system can be criticized, it is often modified to correct several of its main issues and recognized as an international standard to provide players with ratings. It is even possible to use ELO ratings for other applications such as evaluating rules and/or patterns inside game playing engines, as it is described in [33].

A.2 Mathematical details

Given two players W and B rated respectively R_W and R_B, their expected score playing against each other is computed as being respectively $S_W = \frac{1}{1+10^{\frac{R_B-R_W}{400}}}$ and $S_B =$

$\frac{1}{1+10^{\frac{R_W-R_B}{400}}}$. We shall note here that $R_W + R_B = 1$.

It is also possible to write $S_W = \frac{Q_W}{Q_W+Q_B}$ and $S_B = \frac{Q_B}{Q_W+Q_B}$ with $Q_i = 10^{\frac{R_i}{400}}$. From this, we can see that each difference of 400 points in rating means that the expected score of the higher ranked player is magnified by 10 time compared to the expected score of his opponent.

Now, if a player participates to some tournaments or just simple matches against other rated players, his result will be very often different from his expected score. If his results are superior to the expected score, his rating is probably too low and should be increased, while if his results are inferior, his rating is probably too high. Given the same player W whose rating is R_W, the latter is updated after each of his match using the formula $R'_W = R_W + K * (S - S_W)$, where S is the final score of the match and S_W his expected score before the match.

K is called the K value and is a critical term in this formula. The use of a high K-value means that a player's rating will evolve faster, and perhaps never converge to a correct value, whereas using a low K-value means that a player's progress could be very slow. For this reason, beginner players are usually given a higher K-value, while more established players are given a lower one. As an example, the USCF gives a K-value of 25 to beginner Chess players, which goes to 15 once the player has played 30 matches. Over a rating fixed as 2400 (which corresponds to Master or Grandmaster level in Chess), this K-value is dropped to 10.

A.3 Possible issues of the ELO rating system

The main mathematical issue of the model is its dependency to the K-value. While the system works pretty well now with some arbitrary values given to it, no mathematician is really satisfied with the use of a fixed K-value. Ideally, one should have a higher K-value when progressing quickly, and a lower one when progressing slowly. But while the current system for example proposed by the USCF tries to replicate this phenomenon, it does not take into account the possibility that a player whose rating did not evolve much for some time suddenly evolves more quickly. Also, most disagree with the initial value given by Elo of 10 for the K-value to use, showing statistical evidence that was not available at the time.

Some more practical issues exist too. For example, the simple fact that one player's rating does not evolve if he does not play. This means that a good player could "protect" his rating by playing fewer matches and/or easier matches. In Swiss rounds tournaments, this also has the possible side effect of making players drop from a tournament because they lost their first matches and fear losing even more in the remaining rounds when playing against lower rated players, against whom they have very few to gain. For this reason, Chess grandmaster John Nunn proposed a new model to select players opponents at high level, using both the ELO ratings of the players and their activity.

Other potential issues include inflation (a situation in which the player's ratings would be over what they should really be, and where the situation does not change because all what matters is how each of those players compare to the others) and deflation (using the pure ELO system, each match is a transaction of points from one player to the other, making that the pool of points never changes even when more players enter it).

A.4 Expected score from ELO difference

The following Table A.1 gives the expected probability of winning for a player given the difference between his ELO rating and the rating of his opponent. We will assume here that the expected score is the one for the strongest player, the expected score for the weakest only being its complement to 1.

ELO difference	Expected score
0	0.50
20	0.53
40	0.56
60	0.59
80	0.61
100	0.64
120	0.67
140	0.69
160	0.72
180	0.74
200	0.76
300	0.85
400	0.91
...	...

Table A.1: Expected probability of winning given a difference in ELO rating

Appendix B

The program Campya

B.1 History of the program

The program Campya was developed by the author at the Japanese Advanced Institute of Science of Technologies (JAIST) from the year 2006 to the year 2009. This program is a remake of the program Gammazon, which was created by the author at the Paris 5 University (René Descartes) during the year 2006. This change of name was motivated by the fact that we noticed that another Amazons playing program named Gammazon had already been created. The development version of Campya, however, is always named Gammazon.

Both Gammazon and Campya used Monte-Carlo from the start of their development, quickly followed by Monte-Carlo Tree-Search. The author also developed an Alpha-Beta version of Campya for the purpose of testing.

Campya started to use an evaluation function in the beginning of Spring 2007, which was quickly followed by a disastrous appearance at the 10^{th} Computer Olympiad in Amsterdam. The program's evaluation function was later improved to include some more advanced concepts. The MCTS engine was also improved several time during the 3 years development period. Both these allowed the author to enter the 11^{th} and 12^{th} Computer Olympiad respectively in Beijing and Pamplona where Campya made a much better impression. Technical improvements were stopped after this occasion, but testing and tuning of several parts of the program (mostly the evaluation function and the use of AMAF) continued.

B.2 Technical specifications

We will in this section present specifications of the program Campya. However, since the main focus of this work was to develop a Monte-Carlo Tree-Search engine, we will not provide any detail of the other engines developed during the same period, the Alpha-Beta engine and the DFPN/WPNS engines. We will focus on the MCTS engine and, for reasons of simplicity, we often call the MCTS engine by the name of the program.

B.2.1 Code

Campya is written in C++. This choice was motivated by both questions of speed and portability (we wanted to make it run both on Windows and Linux). The GUI was created using the wxWidgets package, once again for reasons of portability.

The main core of Campya, the Amazons playing engine as well as the several search engines created, represents around 16000 lines of code. Among the several interfaces we created, the internal Gammazon interface (a simple console based interface) and its extensions (to deal with solving problems or creating Opening-Books) represents around 5000 lines of code while the point and click interface written using wxWidgets represents 2000 lines of code.

B.2.2 Basic engine

Apart from the Amazons playing core, the main part of the program consists in its Monte-Carlo Tree-Search engine. Campya uses a simple MCTS engine as described in Section 3.2.

The Selection process of Campya functions as described in Section 3.2. It uses UCB to select the best leaf node to expand at every step of the process when all the children of a given node are in the tree, and select a random moves among those not explored yet in the case where all of the children of a given node are not yet in the search tree. The UCB formula used in Campya is described in Section 3.2.3. The exploration factor of this formula is not fixed, since it highly depends of the evaluation function used. Some results about it are presented in Section B.2.5.

The expansion process of Campya has not been modified from its first version, although we have tested several other possible implementations. We follow here the idea proposed by Coulom in [34], that is to add to the game-tree at each iteration the first node not found in the game-tree during the Selection process.

The Evaluation process of Campya is what distinguishes it the most from other traditional MCTS programs. As described in Section 3.5, it consists in first playing a given number of random moves from the position we want to evaluate, and then calling an evaluation function onto the position attained after playing the random moves. Details on the evaluation function are given in Section B.2.3. The number of moves to play which gave us the best results is not clear: we have good results in Campya both when playing 3 or 5 random moves from the position to evaluate, and with playing random moves until the process attain a depth equal to the depth of the root node of the game-tree plus 6 or 7.

B.2.3 Evaluation function

The evaluation function used in Campya is directly inspired from the work of Lieberum presented in [51]. It consists of three components: Queen Distance, King Distance, and Mobility. These components are binded together to create a single evaluation function.

Queen Distance

We define the Queen Distance of an Amazon to a square $Q(A, S)$ as the number of Amazons moves (Queen moves in Chess) needed to place the Amazon A on the square S. This number is set to 0 for the square the Amazon stands on, and to ∞ for squares unreachable for this Amazon. From this function, we can define the Queen Distance of a player to a square $Q(P, S)$ as the minimum number of moves this player would need to place one of his Amazons on the square S: $Q(P, S) = \min_{A \in P} Q(A, S)$.

The Queen Distance component of our evaluation is then computed by comparing the value of $Q(P, S)$ for both players and for all squares of the board. We can define the function *Qcomp* for a square S as:

$$Qcomp(S) = \begin{cases} 1 & if \quad Q(White, S) > Q(Black, S) \\ -1 & if \quad Q(Black, S) > Q(White, S) \\ 0 & if \quad Q(White, S) = Q(Black, S) = \infty \\ \tau_Q & if \quad Q(White, S) = Q(Black, S) \neq \infty \end{cases}$$

From this function, we compute $Q(Position) = \sum_{S \in Position} Qcomp(S)$. While neutral squares usually benefit the first player to move, we curiously found that we got our best results with $\tau_Q = 0$.

King Distance

The King Distance is computed exactly in the same way as the Queen Distance, but using King-like moves (from the game of Chess) instead of Queen moves. It rewards a good balance of the Amazons on the board. We can then define:

- For an Amazon A and a square S, $K(A, S)$ is the number of King moves needed to place the Amazon A on the square S (a King move is a move of one square, in any direction); this value is set to 0 and ∞ in the same conditions as the Queen Distance.

- For a player P, $K(P, S) = \min_{A \in P} K(A, S)$

- For a square S, $Kcomp(S) = \begin{cases} 1 & if \quad K(White, S) > K(Black, S) \\ -1 & if \quad K(Black, S) > K(White, S) \\ 0 & if \quad K(White, S) = K(Black, S) = \infty \\ \tau_K & if \quad K(White, S) = K(Black, S) \neq \infty \end{cases}$

- $K(Position) = \sum_{S \in Position} Kcomp(S)$

The same way as for the Queen Distance, our better implementation uses $\tau_K = 0$.

Mobility

We will need first to define the number of liberties of a square. We set $L(S)$ as the number of empty squares among the 8 squares directly neighboring S (or 5 or 3 if S is on a side of the board). We will also speak of the number of liberties of an Amazon for the number of liberties of the square this Amazon stands on. From there, and the same way as presented in [51], we will define a first measure of the mobility of an Amazon A (owned by player P whose opponent is $opp(P)$), using the Queen and King Distance, as $m(A) = \sum_{S | Q(A,S)=1 \& Q(opp(P),S) \neq \infty} 2^{-K(A,S)} * L(S)$. Simply put, it is the sum of the liberties of all squares that the Amazon can reach in one move, with a weight decreasing with the distance to the square. Squares not reachable by the opponent, that is square part of the player's territory, are not counted in this measure.

From there, we need to come up with a value of the mobility being on a similar scale as the Queen and King Distances. Lieberum proposes in [51] that an Amazon entrapped at the beginning of the game, for example with a mobility of less than 5, should be given a penalty of 10 points. Using this as a basis, we came up with the following measure for the penalty to give to an Amazon A depending on its mobility: $Pm(A) = \frac{30}{5+m(A)}$. With this formula, an enclosed Amazon is given a penalty of 6 points if completely enclosed, and 3 points with a mobility of 5. While the value of 10 points penalty perhaps applies for minimax based search, it definitely does not work as is for CAMPYA, thus we lowered it.

Finally, we can compute the overall Mobility of the board as the sum of the mobility penalties for all the Amazons: $M(Position) = \sum_{A \in Black} Pm(A) - \sum_{A \in White} Pm(A)$.

Final evaluation function

While the Queen Distance is a value which is important all throughout an Amazons game and gives a good approximation of the territory near the end game, the King Distance favoring a good Amazon repartition on the board and the Mobility are both important at the beginning of the game but much less near the end. For this reason, we need to come up with a measure of the advancement of the game.

As noted in [51], simply with counting the number of empty squares on the board we cannot make a difference between situations where territories are almost already enclosed and others where all the Amazons still fight all over the board, both having sometimes the same number of empty squares. For this reason, the author proposes the following measure of the advancement of the game:

$$\omega(Position) = \sum_{S | Q(White,S) \neq \infty \& Q(Black,S) \neq \infty} 2^{-|Q(White,S) - Q(Black,S)|}$$

Simply put, this formula count the number of empty squares on which both player still fight (according to the Queen Distance) and gives a weight to the others decreasing more and more for squares deep in one player's territory.

Using this value, we can finally put our evaluation function:

$$Evaluation(Pos) = \Psi(\sigma_1(\omega) * Q(Pos) + \sigma_2(\omega) * K(Pos) + \sigma_3(\omega) * M(Pos))$$

The tuning of the different functions Ψ, σ_1, σ_2 and σ_3 has been done by hand. We will here give the values of these various functions for the three evaluation functions described in Section 3.5.1:

- For the Classic evaluation function:
  - $\Psi(I) = I$
  - $\sigma_1(\omega) = 1$
  - $\sigma_2(\omega) = 0$
  - $\sigma_3(\omega) = 0$

  This evaluation function only uses the Queen Distance.

- For the Complex evaluation function:
  - $\Psi(I) = I$
  - $\sigma_1(\omega) = 1$

- $\sigma_2(\omega) = \begin{cases} 1 & if \quad \omega > 40 \\ \frac{\omega}{40} & if \quad \omega \leq 40 \end{cases}$
- $\sigma_3(\omega) = \begin{cases} 0.8 & if \quad \omega > 40 \\ 0.8 * \frac{\omega}{40} & if \quad \omega \leq 40 \end{cases}$

- For the Ratio evaluation function:
  - $\Psi(I) = \frac{1-e^{-20*I}}{1+e^{-20*I}}$
  - $\sigma_1(\omega) = 1$
  - $\sigma_2(\omega) = \begin{cases} 1 & if \quad \omega > 40 \\ \frac{\omega}{40} & if \quad \omega \leq 40 \end{cases}$
  - $\sigma_3(\omega) = \begin{cases} 0.8 & if \quad \omega > 40 \\ 0.8 * \frac{\omega}{40} & if \quad \omega \leq 40 \end{cases}$

  This evaluation function only uses the Queen Distance.

B.2.4 Hardware

Campya generally runs on two different kinds of hardwares:

- For testing, we run it on a set of computers equipped with either Pentium IV 3GHz or Athlon 2.2 or 2.6GHz, and at least 1Gb of RAM (2 for some of the machines); these machines are equipped with Windows XP SP2.
- In tournaments conditions, we run Campya on a laptop machine equipped with an Intel Core 2 Duo 1.66 GHz and 2Gb of RAM. This machine then usually runs on the latest version of Ubuntu Linux.

B.2.5 Additional tests

We provide in this section additional tests done during the development of Campya.

Tuning of the exploration constant in UCB

Table B.1 shows the ELO ratings obtained by our program with various values of the exploration constant C in the UCB formula. Since this setting does not affect the speed of the engine, we tested it only with unlimited time.

Processing power

We provide in Table B.2 the ELO ratings observed for Campya with various number of evaluations per move.

It is interesting to note the hole in the ratings around 5000 samples, indicating that a program running with fewer samples (here 2500) performs better. We attribute that to the fact that this value is close to the branching factor of the game, and that in turn the search at depth 2 is too much biased (some kind of horizon effect because all nodes have to be visited at least once before UCB can make any decision). We also note with some regret that, while the results observed here seem close to what is observed for 9x9 Go [2]

Setting	Exploration constant	Classic evaluation	Complex evaluation	Exploration constant	Ratio evaluation
40000 samples	5	1478	1618	0.5	1658
40000 samples	8	1555	1710	0.8	1846
40000 samples	9	1560	1729	0.9	1885
40000 samples	10	1500	1779	1.0	1832
40000 samples	12	1483	1748	1.2	1731

Table B.1: ELO ratings for each of the three evaluations for various values of the UCB exploration constant

Setting	Classic evaluation	Complex evaluation	Ratio evaluation
2500 samples	1139	1404	1351
5000 samples	986	1219	1136
10000 samples	1137	1443	1365
20000 samples	1365	1568	1624
40000 samples	1500	1737	1745
80000 samples	1641	1873	1937
160000 samples	1698	1959	2035

Table B.2: ELO ratings for each of the three evaluations for various numbers of random games

(100 ELO gain for each doubling in power), there seem to be some kind of diminishing return. Hopefully, the discovery of better search techniques or the inclusion of knowledge in the search could push back this effect.

B.3 Tournament history

We will present in this section the various tournaments in which Campya performed and provide the reader with records of the games played.

B.3.1 12th Computer Olympiad, Amsterdam (Netherlands), 2007

Even if it is true that the Amazons tournament have never been a huge part of the Computer Olympiad, the 12th Computer Olympiad in Amsterdam hosted one of the smallest Amazons tournament ever, with 2 entries: Campya, and 8 Queen Problems (8QP) from Johan De Köning. Campya still being a young program at that time, the result of the tournament became clear after the first 2 or 3 games. Both programs played 8 games. A full report can be found in [83]

Game 1: 8QP **(White) vs** Campya **(Black); Result: White+10**

1.	D1D7G7	A7D4D1	2.	G1E3C3	D10D8A5	3.	A4B5H5	J7J6B6
4.	B5C4D3	D4F4F7	5.	E3F2E3	J6C6J6	6.	J4H6H9	F4G4E2
7.	C4B4H10	G10F9B9	8.	H6I7H7	F9E8J8	9.	D7D6F8	E8E6I6
10.	F2G2D5	C6A4B3	11.	D6C7E5	E6C8B7	12.	G2H3H4	G4G1G4
13.	I7F10I7	C8E10E9	14.	F10F9F10	E10C8E10	15.	F9E8B5	A4A3A4
16.	B4F4B4	G1H2F2	17.	C7B8C9	C8F5E4	18.	B8C7C8	F5F6C6
19.	H3I3H3	D8D9D6	20.	C7A9C7	D9C10A10	21.	F4G5G6	H2I1G3
22.	I3I2I3	I1J1G1	23.	I2J2I2	J1I1J1	24.	G5F5G5	I1H2H1
25.	F5E6E7	F6F5F6	26.	E8D9D10	A3B2D2	27.	D9D7D9	F5F4F5
28.	A9A6A9	F4F3F4	29.	J2J5J2	F3G2F3			

Game 2: Campya **(White) vs** 8QP **(Black); Result: Black+Resign**

1.	D1D7G7	G10E8C8	2.	J4J6B6	E8E3D4	3.	J6I6F3	J7J5F1
4.	G1H2E5	E3I7D2	5.	D7D8C7	D10E10H7	6.	D8E9A9	E10F9F10
7.	E9F8E8	F9G8B3	8.	I6J6E6	A7C9F9			

Game 3: 8QP **(White) vs** Campya **(Black); Result: White+17**

1.	D1D9G9	G10D7E8	2.	J4J6B6	J7G4I6	3.	G1F2F10	D10H6F6
4.	J6H8I7	H6J4H4	5.	F2G2J5	G4C4B4	6.	A4A3I3	D7D3D8
7.	D9C8A8	A7C7D7	8.	G2D5I5	J4J1A1	9.	H8H7F5	C7B8B7
10.	D5H1D5	J1G4G2	11.	H7G7G6	C4C5G1	12.	H1H2D6	B8D10A7
13.	G7H6E9	D10E10H7	14.	H6F8F9	E10C10C9	15.	A3C3E5	G4I2H1
16.	F8H6E3	C5C4C7	17.	C3C2D1	D3B3C3	18.	C8B9B10	B3B2B3
19.	H2H3F3	C4D3E4	20.	C2D2E2	B2C2B2	21.	H3H2H3	I2J3I2
22.	H6H5G4	C10D9E10	23.	D2C1D2	C2B1C2	24.	H2I1J2	J3I4J4
25.	B9C8B9	D9D10C10	26.	H5H6H5	D10D9D10	27.	C8B8C8	D3A6A3
28.	I1F4G5	B1A2B1						

Game 4: Campya **(White) vs** 8QP **(Black); Result: Black+30**

1.	D1D7G7	A7F2A7	2.	A4B5E2	G10E8H8	3.	J4F4E3	E8E6F7
4.	F4I7J6	E6C4B4	5.	D7D8F10	F2G2F1	6.	I7I8I4	G2H2G2
7.	G1J1G1	H2J2I2	8.	J1I1C7	J7H7H2	9.	B5F5H5	J2H4J2
10.	F5D5G5	H4E4D4	11.	D5C6F9	H7I7H7	12.	D8D6I6	E4F4F5
13.	C6E4E8	D10D7E6	14.	E4C2C3	I7J8I7	15.	C2B1A1	J8H10E7
16.	D6D5A8	H10I9H9	17.	D5B7B5	D7C8B8	18.	B7C6B7	C4B3D5
19.	I8J9I8	C8D7D6	20.	B1E4B1	I9J8H10	21.	J9I9J9	F4G3F4
22.	E4C2B2	G3F3E4	23.	C2D2C2	F3F2E1	24.	C6C4C5	B3A2B3
25.	I9I10I9							

Game 5: 8QP **(White) vs** Campya **(Black); Result: White+23**

1.	D1D8B6	G10G3G8	2.	J4E9E10	A7D7E7	3.	D8D9C10	D10C9A7
4.	A4C2C6	J7G7F7	5.	D9C8D9	D7D3D7	6.	E9I9I3	C9B8B9
7.	G1D4D6	G3F2F4	8.	D4E4C4	G7H8I8	9.	E4F5H7	D3H3H5
10.	F5G4G7	H8J6H4	11.	C2E2E3	F2E1B4	12.	E2D2D1	E1E2E1
13.	D2D3D2	E2G2G3	14.	G4F3G4	B8B7B8	15.	D3D5A5	H3J5H3
16.	I9J8I7	J6E6H6	17.	J8J6I6	G2F1G2	18.	D5D4D3	E6E5F5
19.	J6I5J6	B7C7E9	20.	C8D8C8	J5I4J4	21.	F3F2E2	F1G1H1
22.	D4D5D4	C7B7A6	23.	I5J5I5	E5E6E4	24.	F2F1F3	I4J3I4
25.	D5C5E5	G1F2G1	26.	C5B5D5	B7A8A9	27.	D8C7B7	

Game 6: Campya **(White) vs** 8QP **(Black); Result: Black+18**

1.	G1G7D7	J7G4B4	2.	A4A6E10	A7E3E1	3.	D1F3H3	D10B8F8
4.	G7E5G5	G4I4J5	5.	J4G7F7	G10G9H8	6.	G7I5J4	E3D3I8
7.	I5I6F9	I4I2G2	8.	F3F4F5	I2I4G4	9.	F4I1I3	I4H5I5
10.	A6C8C2	D3C3G3	11.	C8A6D3	C3D4G1	12.	E5E3F4	D4C3C7
13.	I1I2H2	B8B7A7	14.	E3E9E2	B7D9D8	15.	E9E4A8	D9B7B5
16.	E4E9B9	H5H6H7	17.	I2J3I2	H6E6G6	18.	I6J6J10	E6B6A5
19.	E9E5D6	B6E3B6	20.	J6J9I9	G9F10I10	21.	J3H5J7	B7D9B7
22.	H5H6I6	F10G10G7	23.	E5E4C6	C3C4C3	24.	E4E5F6	D9E8C10
25.	E5D4E5	E8E6D5	26.	D4C5D4	C4A2C4	27.	H6H4H5	

Game 7: 8QP **(White) vs** Campya **(Black); Result: White+25**

1.	D1D9H5	G10G4G9	2.	J4H6B6	A7D7A7	3.	A4C4C10	G4G3E1
4.	G1F2E3	J7I8H9	5.	H6I7H7	D10J4E4	6.	D9E8E5	J4F8E9
7.	C4C6D6	D7C8I2	8.	C6B7C7	F8F3F9	9.	F2G2E2	C8D7E7
10.	B7C6D5	D7C8E10	11.	C6B7C6	F3F6J10	12.	I7J8F4	F6J6H6
13.	J8I9H8	I8J9I8	14.	G2H3G4	J6H4J4	15.	E8G6G5	H4I4I3
16.	G6E8D7	G3G2H2	17.	H3H4H3	G2G3G2	18.	H4I5J5	J9I10G10
19.	I9J9I9	C8C9D10	20.	B7B8C8	C9B9A9	21.	B8A8B8	B9D9B9
22.	I5H4I5	G3F3G3	23.	J9J8J9	I4J3I4	24.	A8B7A8	D9C9B10
25.	E8D9D8	I10H10I10						

Game 8: Campya **(White) vs** 8QP **(Black); Result: Black+11**

1.	G1G7D7	J7E2E1	2.	D1D5D2	D10F8G8	3.	D5E5C5	G10I8C2
4.	A4A6I6	A7B6A5	5.	J4H4B4	I8J8H8	6.	G7J7G7	E2D3I8
7.	A6B7F3	J8G5J8	8.	J7J1G4	F8E9E10	9.	H4H5H4	G5F4F7
10.	J1G1G3	F4H6E6	11.	E5C3C4	H6J4H2	12.	H5G5C9	E9D8F6
13.	G1F1E2	D8C8C7	14.	F1I4J5	J4I3H3	15.	G5E3E4	D3D4D6
16.	B7A8B8	C8B9A9	17.	A8B7D9	B9C8B9	18.	I4I5E5	I3I4I1
19.	C3B3C3	B6B5C6	20.	I5J4I3	I4H5I5	21.	B3A4A3	H5G6J3
22.	E3H6E3	B5A6B5	23.	B7A7B7	A6B6A6	24.	J4I4H5	G6H7F5
25.	H6I7G5	H7H6H7	26.	A7A8A7	D4D5D3	27.	I7J7I7	

B.3.2 13th Computer Olympiad, Beijing (China), 2008

The Amazons tournament taking place in the 13th Computer Olympiad in Beijing saw the come back of 8QP and Campya, and the entry of two other programs: InvaderMC was the new version of the program INVADER from Richard Lorentz (we will call it INVADER for simplicity), featuring Monte-Carlo Tree-Search as CAMPYA does, and BIT was a traditional minimax program developed by students of the Beijing Institute of Technology. Every program played the three others 4 times, for a 12 round tournament. A full report can be found in [55]. CAMPYA ended up third with 3 wins against BIT and 1 win against INVADER (future winner), but with a much better play than the previous year in Amsterdam.

Game 1: 8QP **(White) vs** CAMPYA **(Black); Result: White+13**

1.	D1D7G7	J7G4B4	2.	J4J6B6	D10F8D8	3.	A4B3F3	F8F4C4
4.	B3D3D2	F4E5C5	5.	D7B7A6	A7B8A8	6.	D3D4E4	G4G2E2
7.	B7D9B7	G10I8I5	8.	G1I1E1	G2H2J2	9.	D9C9C6	B8C8I2
10.	I1G1G5	C8D9J3	11.	J6H8E8	H2G2H2	12.	G1E3C3	E5D6F4
13.	E3F2F1	D6F6F10	14.	D4E5D5	F6E6F5	15.	E5F6D4	E6F7H5
16.	H8H9F9	F7E6G8	17.	F6J6F6	E6E7C7	18.	H9H10F8	G2G3G1
19.	J6J5H3	E7D7A10	20.	H10H8I7	D7C8B8	21.	H8I9H10	D9C10D9
22.	I9H9H7	C10D10E9	23.	J5J7J10	D10C10B9	24.	H9I9H8	C10D10B10
25.	I9H9I9	C8E6C8	26.	J7J6J7	E6F7E6	27.	J6H6G6	G3G2G4
28.	H6J6H6	I8J9I8	29.	J6J4H4	D10C10E10	30.	C9D10C9	F7D7F7
31.	J4I3I4	D7D6E5	32.	I3J4I3	D6D7E7	33.	H9G10H9	D7D6D7
34.	G10G9G10	J9I10J9	35.	J4J5J4	G2H1G2	36.	J5I6J5	H1I1J1
37.	I6J6I6	I1H1I1	38.	F2E3F2	pass			

Game 2: CAMPYA **(White) vs** 8QP **(Black); Result: Black+5**

1.	G1G9D9	D10I5I3	2.	A4D7G4	J7H7J5	3.	J4H4B10	H7D3H3
4.	D1C2C7	A7F2G3	5.	H4H5C5	I5H6H8	6.	C2C3E5	D3D5D3
7.	D7C8E8	D5B7B8	8.	C8D7B5	F2D2C1	9.	D7C6A6	G10H9D5
10.	C6F6E6	D2F2F5	11.	F6C9C8	F2D2G5	12.	H5I5I8	B7A8C10
13.	C3D4A1	H9H10H9	14.	C9E7I7	A8C6D6	15.	G9G10G6	H6I6J6
16.	G10G9F10	C6D7D8	17.	E7G7H7	I6H5J3	18.	G7F8E7	D2E3E4
19.	D4C3C2	E3F2D2	20.	C3D4A4	F2F4E3	21.	F8D10H6	D7C6D7
22.	D10C9A9	C6A8B9	23.	G9G10G9	H5I4H5	24.	D4B4B3	A8C6B6
25.	I5I6I5	H10J8H10	26.	I6J7I6				

Game 3: Campya **(White) vs** Invader **(Black); Result: White+5**

1.	D1D9G9	G10B5B4	2.	G1D4D3	B5E5B5	3.	A4D1I6	D10H6C1
4.	D1F3H5	J7D7J7	5.	F3F7F5	E5G3I3	6.	J4H4H2	A7C7C9
7.	D9C8H8	G3G2G8	8.	F7I7E7	G2G3G4	9.	H4H3F1	G3F2J6
10.	D4C3E1	D7G10I8	11.	I7F7F10	G10E8F8	12.	F7I7F7	E8D9E8
13.	C3C4C2	C7A7B8	14.	C8A6G6	H6J4H6	15.	A6C8D8	F2D4D6
16.	I7J8J10	A7B7G2	17.	H3I4J3	J4I5G3	18.	I4H4I4	B7B6C6
19.	C8A10A5	D9C10B9	20.	A10A8A10	B6A7B7	21.	C4C5B6	C10E10B10
22.	C5D5E4	E10C8E6	23.	D5C5E5	D4C3D4	24.	J8H10I10	C8E10C10
25.	C5C4A2	C3D2C3	26.	H10G10F9	D2E3D2	27.	C4C5D5	E3F3D1
28.	H4H3H4	F3E3G1	29.	C5C4C5	E3E2F3	30.	G10H10G10	E2F2E2
31.	H10J8H10	E10C8C7	32.	J8I7G7	A7A6A7	33.	I7J8J9	

Game 4: Invader **(White) vs** Campya **(Black); Result: White+2**

1.	D1D9E10	G10G3I5	2.	J4F4F8	D10C9G5	3.	F4F6F3	A7C5C7
4.	A4B4B8	J7G7G6	5.	D9H9H7	G7H8G9	6.	F6A6D9	C5C3C6
7.	G1H2C2	C9C8I2	8.	B4E7D7	G3H3E6	9.	A6C4I4	C8D8F10
10.	H9I8H9	H3G3D6	11.	H2F2E1	H8I9F6	12.	I8I7H8	I9J8I8
13.	I7J7I7	G3H2H1	14.	F2D4H4	H2F4E3	15.	J7G4J7	F4H2H3
16.	C4A6F1	C3C5A7	17.	D4C4B5	C5A3D3	18.	C4C3B4	D8C8B7
19.	E7D8B10	A3B3A4	20.	A6B6D4	H2G3G2	21.	B6C5C4	B3B2A3
22.	C5E5F5	G3H2F4	23.	C3D2C1	J8H10J8	24.	D2C3B3	H10G10E8
25.	D8E9F9	C8D8C9	26.	E5E4E5	H2J4J1	27.	G4G3I1	J4J5J6
28.	G3G4I6	J5J4G1	29.	G4G3H2	G10J10H10	30.	G3G4G3	D8C8D8
31.	E9D10C10	C8B9C8	32.	D10E9D10	B2A2B2	33.	C3D2D1	A2A1A2
34.	D2E2F2	J4I3J2						

Game 5: Campya **(White) vs** BIT **(Black); Result: Black+3**

1.	G1G9D9	J7G4I4	2.	A4C6B6	D10H6H8	3.	D1F3F4	A7A3E3
4.	J4I5I7	G10D7B7	5.	F3G2H3	A3C3C5	6.	C6A4B4	G4G3F3
7.	A4B3G8	C3C2C4	8.	G9F8C8	G3H2J4	9.	I5G5J5	D7D8F6
10.	B3A3C1	D8E8A4	11.	G5F5D3	E8D7D5	12.	F5I5I6	D7E8F7
13.	I5E5I5	E8D8D6	14.	E5F5H7	D8E8E6	15.	F5G5G7	C2B2B3
16.	F8E9F8	H6G6F5	17.	G5J2G5	E8F9G10	18.	G2D2C3	H2I2E2
19.	D2E1D2	G6H5G6	20.	E1G3H4	I2G2F2	21.	A3A2A3	G2H2G2
22.	A2B1A2	H2I1H2	23.	B1C2B1	I1H1D1	24.	J2I1I2	H5G4H5
25.	E9E10E9	H1G1H1	26.	I1J1I1	G1F1E1	27.	J1J2J1	F1G1F1
28.	J2I3J2	B2A1B2	29.	E10B10F10	F9E8B5	30.	B10C9E7	E8D7D8
31.	C9A9A6	D7C6C7	32.	A9B10E10				

Game 6: BIT **(White) vs** Campya **(Black); Result: Black+14**

1.	D1D7G7	A7D4B4	2.	G1E3E9	J7G4G6	3.	J4H4C9	D10D8G5
4.	A4C6C3	G4H3F3	5.	C6E6H9	G10G8I6	6.	D7C8C7	D8D5D9
7.	E6C6F9	G8H8E8	8.	E3D3C4	D5B5B9	9.	C8E6E5	D4E3D4
10.	E6G8H7	B5C5G9	11.	D3D2I2	C5A5A9	12.	H4J6J9	E3E2E1
13.	J6I7I9	A5D5F7	14.	I7J6J8	H3J5I5	15.	D2F4I4	J5J4H2
16.	F4H4J2	J4I3G3	17.	H4H3H4	D5C5F8	18.	H3F1H3	H8I7H8
19.	J6J4J3	I7J6J5	20.	F1F2F1	E2E3E2	21.	C6B5B6	C5C6C5
22.	B5A6B5	E3C1E3	23.	A6C8G4	C6D7C6	24.	C8A6A3	D7C8D7
25.	A6A4D1	C1B2B3	26.	A4A7B7	C8B8A8			

Game 7: 8QP **(White) vs** Campya **(Black); Result: White+2**

1.	D1D8B6	G10G3B3	2.	J4J5E10	D10F8B4	3.	G1F2F7	J7E2B5
4.	A4A2A6	A7C7C2	5.	A2A1H8	F8G7B2	6.	A1C1I1	G7I7F4
7.	C1D2D1	E2E3C3	8.	F2F3B7	C7C9G9	9.	F3H5E5	G3G4I4
10.	H5H3F3	I7I6J6	11.	J5F5I8	I6G8G7	12.	F5H7H5	G4G5I7
13.	H7I6C6	G5I3F6	14.	D8E8F8	C9D8D9	15.	H3C8C9	D8C7A9
16.	E8G10H9	G8F9D7	17.	G10F10H10	C7B8B10	18.	F10E9C7	I3H3E6
19.	I6H7E4	H3H4G4	20.	H7I6G8	B8B9B8	21.	C8E8C8	B9C10D10
22.	D2G2D2	H4I5J5	23.	E8E7E8	E3D3D6	24.	G2I2E2	I5H4H2
25.	I2H3I3	D3E3G1	26.	I6H6I5	E3F2G2	27.	H6G5F5	F9F10F9
28.	H3J1H3	F2C5E3						

Game 8: Campya **(White) vs** 8QP **(Black); Result: Black+0**

1.	G1G8D8	D10H6H9	2.	A4D7G4	H6D2D6	3.	J4H4J6	J7F7F8
4.	D7C8C3	A7B7B3	5.	D1E2E8	F7H7H8	6.	G8G6H6	H7J5F5
7.	G6I8I4	J5J2F2	8.	H4I3I2	J2J5H7	9.	I8I7I5	J5J1B1
10.	E2D3E3	J1H1H4	11.	I3H2G2	D2A2A9	12.	D3B5C6	A2E2C4
13.	B5D5D2	E2F3J3	14.	C8C9G9	F3E4F4	15.	H2G1I3	E4D3F1
16.	G1H2I1	D3D4C5	17.	I7J8H10	B7B8C8	18.	D5E6E4	G10E10D9
19.	E6F7B7	D4E5E7	20.	H2G3H2	E5D4D5	21.	F7G7E5	E10G8E6
22.	G3F3D1	G8E10G8	23.	F3E2D3	E10D10A10	24.	C9B9C9	B8A8B8
25.	B9C10B10	A8A5A8	26.	E2F3H3				

Game 9: Campya **(White) vs** Invader **(Black); Result: Black+1**

1.	G1G8D8	D10I5B5	2.	D1D7G4	J7F7G7	3.	A4D4F2	F7F3F7
4.	J4I3G5	F3D3H3	5.	D7C8B8	A7C7I1	6.	D4C3C6	G10G9B9
7.	G8H9H5	C7E5H2	8.	H9G8I6	I5J4I4	9.	G8I8F8	G9D9C9
10.	I8H8J6	E5D6D7	11.	C8A6A2	D6B4A5	12.	A6A7E3	B4D4B6
13.	A7C7E5	D4B4D6	14.	C7A7C7	D3I8H9	15.	A7A10I10	D9E10B10
16.	H8G9D9	J4J3J2	17.	G9F9E9	I8G8G9	18.	I3J4J5	J3H1C1
19.	C3D3F1	G8H7H6	20.	D3D5G2	H1J3I3	21.	D5C4C3	B4B3B1
22.	J4I5J4	B3C2F5	23.	F9G8I8	C2D3D5	24.	G8F9F10	H7H8G8
25.	I5G3F4	D3C2E2	26.	C4B3A4	C2D3E4	27.	B3C2D2	D3C4B3
28.	C2D3D4	C4C5C4	29.	D3C2B2	E10D10C10	30.	G3H4I5	J3H1G1
31.	A10A9A10	H1J3H1	32.	A9A7A9	H8I7J7	33.	A7B7C8	J3I2J1
34.	C2D1E1	I7H8J10	35.	B7A7A8	C5B4A3	36.	A7A6A7	B4C5B4
37.	D1C2D1	H8I7J8	38.	F9E8F9	I7H7G6	39.	A6B7A6	D10E10D10
40.	C2D3C2	I2J3I2	41.	H4G3F3	H7H8I7	42.	E8E7F6	H8I9J9
43.	E7E6E8	I9H10G10	44.	G3H4G3	H10I9H10	45.	E6E7E6	I9H8I9
46.	pass							

Game 10: Invader **(White) vs** Campya **(Black); Result: White+2**

1.	D1D9E10	G10G3I5	2.	J4F4F8	D10C9G5	3.	F4F6F3	A7C5C7
4.	A4B4B8	J7G7G6	5.	D9H9H7	G7H8G9	6.	F6A6D9	C5C3C6
7.	G1H2C2	C9C8I2	8.	B4E7D7	G3H3E6	9.	A6C4I4	C8D8F10
10.	H9I8H9	H3G3D6	11.	H2F2E1	H8I9F6	12.	I8I7H8	I9J8I8
13.	I7J7I7	G3H2H1	14.	F2D4H4	H2F4E3	15.	J7G4J7	F4H2H3
16.	C4A6F1	C3C5A7	17.	D4C4B5	C5A3D3	18.	C4C3B4	D8C8B7
19.	E7D8B10	A3B3A4	20.	A6B6D4	H2G3G2	21.	B6C5C4	B3B2A3
22.	C5E5F5	G3H2F4	23.	C3D2C1	J8H10J8	24.	D2C3B3	H10G10E8
25.	D8E9F9	C8D8C9	26.	E5E4E5	H2J4J1	27.	G4G3I1	J4J5J6
28.	G3G4I6	J5J4G1	29.	G4G3H2	G10J10H10	30.	G3G4G3	D8C8D8
31.	E9D10C10	C8B9C8	32.	D10E9D10	B2A2B2	33.	C3D2D1	A2A1A2
34.	D2E2F2	J4I3J2						

Game 11: Campya **(White) vs** BIT **(Black); Result: White+0**

1.	D1D8G8	A7D4B4	2.	A4C6F9	J7G4D7	3.	G1F2F5	D10H6E9
4.	J4J6F10	G4G3C7	5.	J6I7I8	G10G9E7	6.	C6C4C6	H6H5E2
7.	I7H8H9	G9F8E8	8.	F2H2H4	G3C3I3	9.	D8B8B6	C3B3G3
10.	C4C3E3	F8H10J8	11.	H2H3J5	D4D5D2	12.	C3C2E4	B3B2G7
13.	B8A7A3	D5C4A6	14.	C2B3D1	H5H7I7	15.	B3C3C2	C4D4C4
16.	H8I9H8	B2B3B2	17.	H3I4H5	H7I6G6	18.	A7A10A8	D4E5D4
19.	I4H3I4	E5F4F2	20.	H3I2G4	F4F3H1	21.	I2J2G2	I6I5H6
22.	J2J4J1	H10I10H10	23.	I9J10I9	I10J9I10	24.	A10B10A10	B3A2B3
25.	C3D3C3	A2A1A2	26.	B10B9A9	A1B1A1	27.	B9B7E10	B1C1B1
28.	B7D9D8	F3F4E5	29.	D9B7A7	I5I6I5	30.	J4J2J4	I6J7J6
31.	J2H2H3	J7I6H7	32.	B7B8B7	F4G5F4	33.	H2I2J3	G5F6G5
34.	B8B9B10	F6E6D6	35.	B9C10D10	E6D5A5	36.	I2H2J2	D5B5A4
37.	H2I2I1	B5C5B5	38.	C10D9C10	C5D5C5	39.	I2H2I2	D5E6D5
40.	D9C9D9	E6F7E6	41.	C9C8B9	F7F8F6	42.	H2G1H2	F8G9G10
43.	C8B8C9	I6J7I6	44.	G1E1G1	G9F8G9	45.	E1F1E1	F8F7F8
46.	B8C8B8	pass						

Game 12: BIT **(White) vs** Campya **(Black); Result: Black+23**

1.	D1D9E9	G10G3I5	2.	J4F4I7	D10B8F8	3.	G1B6C7	B8E8E6
4.	F4G4I6	J7I8C2	5.	A4E4H7	I8G8E10	6.	E4D4D7	G8G5F6
7.	D9C9E7	A7A4A10	8.	G4H3F5	A4B4A5	9.	D4C4F4	G3E3C3
10.	C4E2B5	B4C4C5	11.	B6B7D5	G5I3I4	12.	H3G3G8	E3F3G4
13.	G3H3G3	F3F2H2	14.	H3H5F7	I3J2H4	15.	E2D1D4	F2E2E1
16.	D1D3D1	E2E3E2	17.	D3E4H1	E3F3E3	18.	E4D3E4	E8C8D9
19.	D3D2D3	J2J10J8	20.	D2C1A3	C4A2B2	21.	C9D8B10	J10I9F9
22.	H5G6H6	I9H8G7	23.	D8C9D10	C8A8D8	24.	C1B1A1	A8A7B8
25.	B1C1D2	A7A6A9	26.	B7B6B7	A6A8C10	27.	C9B9C9	A8A6A8
28.	B9C8B9	A2B3A2	29.	C1B1C1	B3B4C4	30.	G6G5G6	H8H10F10
31.	G5H5G5	F3F2G1	32.	B6C6B6	B4A4B3	33.	C6D6C6	A6A7A6
34.	D6E5D6	H10I10I9	35.	pass				

B.3.3 14th Computer Olympiad, Pamplona (Spain), 2009

The Amazons tournament hosted at the 14th Computer Olympiad saw the come-back of the same four programs as the previous year in Beijing. Campya made there a much better impression, scoring 2 wins and 2 loss against each other program, most of the matches being pretty close. With 6 wins, it also barely missed the silver medal got by 8QP with 7 wins. As the previous year, Invader came back with the gold medal with a total of 9 wins (but only two against Campya, its weakest score). A full report can be found in [44].

Game 1: Campya **(White) vs** 8QP **(Black); Result: Black+3**

1.	D1D9G9	G10B5B3	2.	J4H6B6	A7D7E8	3.	A4C4C6	D7H3C8
4.	G1F2C5	B5A5D2	5.	C4B4A4	A5A9H2	6.	B4D4G4	H3D3C3
7.	D9B9B8	J7I7H8	8.	F2F1F8	I7I5E1	9.	F1H3J5	D10I10C4
10.	H6H7A7	A9A8A10	11.	B9E9B9	I5I3I7	12.	H7G7B7	D3E2D3
13.	D4E3E6	I3H4H7	14.	E9F10B10	I10H9G10	15.	G7G5G8	H4I3I2
16.	H3H5J7	I3H4I4	17.	E3F3H3	H4G3F2	18.	F10D8D4	G3H4J2
19.	D8C7F10	H4G3G2	20.	H5F7F4	G3I5H5	21.	G5H6G5	A8A9A8
22.	F7F6C9	H9I10I8	23.	H6I6H6	I5G3I5	24.	F6G6E4	G3H4G3
25.	C7D8D7	I10H9H10	26.	D8D10C10	H4I3H4	27.	G6G7F7	I3J4I3
28.	I6J6I6	J4J3J4	29.	D10E10E9	H9J9J8	30.	E10D10D9	J9I10H9
31.	G7G6G7	I10J9J10	32.	G6F5D5	J9I10J9	33.	F5G6F5	I10I9I10
34.	G6F6G6	E2D1E2	35.	F3E3F3				

Game 2: BIT **(White) vs** Campya **(Black); Result: White+12**

1.	G1G8I6	D10D3B5	2.	A4E4E3	J7G7E5	3.	E4B7B8	G7F6I3
4.	J4H4H10	G10E8C8	5.	D1C2E2	F6H6D6	6.	B7C7I7	H6G5C9
7.	C7D7G4	D3C3C6	8.	D7B7E7	G5I5G5	9.	B7B6D4	I5J4J3
10.	B6B7A6	J4J5G2	11.	H4H3H6	J5I5G3	12.	H3I4H4	E8F9D7
13.	C2D3F5	C3C2A4	14.	D3C3E1	F9H9H7	15.	C3B3D3	C2B2A1
16.	B3C4A2	H9G9D9	17.	C4B3D5	A7A10A7	18.	B7A8C10	I5J5J8
19.	B3C3A3	G9F8F10	20.	A8A9B9	F8E9A5	21.	G8F9F8	B2C2D2
22.	F9H9F9	E9E8H5	23.	H9G8F7	C2B3C4			

Game 3: Campya **(White) vs** Invader **(Black); Result: White+4**

1.	D1D8B6	G10G3I5	2.	J4G4F3	A7E7E8	3.	A4D4D6	E7E2E1
4.	G1H2H10	E2C4C3	5.	G4G7F6	D10C9C8	6.	G7I7D2	J7H9D5
7.	I7H8J6	G3H3D7	8.	D4D3H7	H3H4E4	9.	H2H3I4	H4G3G5
10.	D8C7C5	C4B3C2	11.	D3C4B4	G3H2E2	12.	C7A9A4	C9B8B9
13.	A9B10D8	H9G9E9	14.	H3G3G1	H2H3H2	15.	C4A6C4	G9H9J7
16.	B10E10G8	H9G9G10	17.	E10D10D9	H3H4H3	18.	A6B7A8	G9F8F10
19.	H8I9G9	F8G7E7	20.	I9H8J8	G7H6H5	21.	H8I7H8	B8A7C9
22.	G3F4G3	A7B8C7	23.	F4F5G4	H6G6H6	24.	F5E6F5	G6F7F9
25.	I7I6J5	H4J2H4	26.	E6E5E6	B8A7A5	27.	E5D4D3	B3B2A1
28.	B7A6B7	A7B8A9	29.	D4F2F1	J2J4J2	30.	F2G2H1	B2A3B3

Game 4: 8QP **(White) vs** Campya **(Black); Result: Black+0**

1.	D1D7G7	A7D4B2	2.	J4J6I6	G10E8H8	3.	D7D9C10	D10B8I1
4.	G1G6I8	B8G3I5	5.	J6J4J6	G3I3I4	6.	D9F9J9	J7H7H1
7.	G6G5G6	H7G8A2	8.	J4J2D2	E8D7I2	9.	A4C4C8	I3G3I3
10.	C4C7D8	D7C6B6	11.	C7E5E1	C6D6D5	12.	E5E4C2	G3F3E3
13.	E4D3A6	D4C4H4	14.	D3E2F1	D6C7E5	15.	G5G4G3	F3F5F2
16.	G4G5G4	F5F3F6	17.	E2D3E2	C7A9E9	18.	D3B3B5	C4B4C4
19.	G5F4E4	A9A7D10	20.	F9F7B7	A7A9D9	21.	F7E7C7	G8G9F8
22.	E7F7H9	B4D6A3	23.	F4J8F4	G9H10I9	24.	F7G8G10	D6E7B4
25.	G8F9C6	H10G9F10	26.	F9G8E6	E7D7F9	27.	G8F7E7	G9G8H7
28.	B3C3D3	D7E8D7	29.	J2J4J5	A9B9C9	30.	J4J3J4	G8G9G8
31.	J3J2J3	F3G2F3	32.	J2J1J2	G9H10G9	33.	J8G5F5	

Game 5: CAMPYA **(White) vs** BIT **(Black); Result: White+0**

1.	D1D9G9	A7E3H6	2.	G1G8F9	D10B8B4	3.	J4I4I6	G10I8H9
4.	G8H8H7	J7J3H3	5.	A4C2J2	J3I3J3	6.	I4E4H4	B8C8C3
7.	C2E2H2	I3I5D5	8.	E4E7G5	I5H5F3	9.	E7D6G6	I8I9I7
10.	E2D2D4	H5G4F4	11.	H8J8H8	I9I8J7	12.	J8I9J9	E3E2E1
13.	D6D7F5	E2B5E2	14.	D2C2A4	G4G1F1	15.	D9E9E3	C8D8C7
16.	E9B9B6	D8E7B10	17.	B9C8A8	B5C4C6	18.	D7E8D8	C4A6D3
19.	E8F7E8	A6B7B9	20.	C8B8C8	E7F8D6	21.	I9H10I9	B7A6A7
22.	C2B3C2	A6B5C4	23.	H10D10E9	B5A6B7	24.	F7E7F7	F8G7E5
25.	E7F6E7	A6A5A6	26.	D10F10D10	A5B5A5	27.	F10G10J10	B5C5B5
28.	G10F10I10	G7F8G7	29.	B8A9A10	F8G8F8	30.	B3B1D1	G1F2G3
31.	F6E6F6	F2G2F2	32.	B1C1D2	G2G1G2	33.	E6D7E6	G1H1G1
34.	F10G10H10	H1I2H1	35.	A9B8A9	I8J8I8	36.	C1B1C1	I2I3I4
37.	G10E10D9	I3I2J1	38.	E10F10E10	I2I3I1	39.	B1B2B3	I3J4J5
40.	F10G10F10	J4I5J6	41.	B8C9B8	I5H5G4	42.	C9C10C9	H5I5H5
43.	B2B1A1	I5J4I5	44.	B1B2B1	J4I3J4	45.	B2A2B2	I3I2I3
46.	A2A3A2	pass						

Game 6: INVADER **(White) vs** CAMPYA **(Black); Result: White+0**

1.	D1D8B6	G10G3G8	2.	G1E3I7	A7D7E8	3.	A4C6I6	D7D3E4
4.	C6C5C10	D10C9E9	5.	D8C8D8	J7H9H6	6.	C5D6B8	G3H3J5
7.	J4G4J4	H9H7E7	8.	D6F6H8	H7F7H7	9.	E3D2J2	D3C3C2
10.	C8E10F9	C9C8E6	11.	D2D5H5	F7F8H10	12.	E10G10G9	F8G7G5
13.	F6D4F6	C8C6D6	14.	D5B5B3	C6C8E10	15.	G10J7J6	C8C6A8
16.	B5A6C8	C6B5A5	17.	D4C4C6	C3E3C3	18.	C4E2I2	E3D2D1
19.	E2E3E2	B5C5B5	20.	E3D4A4	D2E3G3	21.	D4D2D5	E3F2F4
22.	D2C1A1	F2E3G1	23.	C1A3C1	C5D4B4	24.	G4F3H1	E3F2G2
25.	F3G4F3	H3H4H3	26.	G4F5E5	G7G6F7	27.	F5G4F5	

Game 7: CAMPYA **(White) vs** 8QP **(Black); Result: White+1**

1.	D1D8B6	G10G2H1	2.	G1E3G1	A7A5I5	3.	J4H4D4	D10E9H6
4.	A4B4B5	J7E7F8	5.	E3E6H9	E7F6F7	6.	E6F5J9	G2I4G6
7.	F5H5D5	A5A3G3	8.	H5J7G7	F6E6G4	9.	B4B3B2	E9F10J6
10.	H4I3H3	E6E2D3	11.	J7H7J5	I4H5H4	12.	B3C2A4	A3C3C8
13.	D8C7C4	C3B4C3	14.	C7D8D6	F10E10G8	15.	H7I7F10	E10E9E7
16.	D8C9D10	E9D9C10	17.	C9D8E9	H5J7J8	18.	I7I8I6	J7I7J7
19.	I8H8I8	E2D2H2	20.	C2D1B3	D2E2E1	21.	D1C1F4	E2E3D2
22.	D8C9D8	B4A5A9	23.	C9A7C9	D9E8C6	24.	C1D1F3	E3E5E2
25.	A7D7A7	E8G10G9	26.	H8I9H10	I7H8I7	27.	I3I4I1	E5F5H5
28.	D1A1D1	A5B4A3	29.	A1C1C2	G10E8D9	30.	C1A1A2	F5E5E6
31.	D7B7D7	B4A5A6	32.	I9I10I9				

Game 8: BIT **(White) vs** CAMPYA **(Black); Result: White+4**

1.	D1D9G9	A7D4F2	2.	J4H6J6	G10C6G2	3.	A4C4C3	D4D3F1
4.	G1I3I10	J7H7F7	5.	I3E3B6	D10C9G5	6.	D9C8G4	C6F3F4
7.	H6I5G3	F3D5B7	8.	E3D2C2	D5B5E8	9.	I5H6D10	D3E4D3
10.	C4B4A4	B5C4C5	11.	B4B5B3	E4G6C6	12.	H6I6H5	G6H6J4
13.	I6I5J5	C4B4E4	14.	B5C4A6	B4A3C1	15.	C4D4B4	H6D6F6
16.	D4E5D5	H7H6I6	17.	E5E7E5	H6F8H6	18.	C8F5I8	F8G8G6
19.	F5B9B10	G8F8E9	20.	E7D8B8	D6E7D7	21.	D8D9C8	F8F9E10
22.	D9D8D9	E7D6E7	23.	D8C7D8	A3B2A1			

Game 9: CAMPYA **(White) vs** INVADER **(Black); Result: Black+0**

1.	D1D9G9	G10C6E8	2.	J4H6J6	J7G4B4	3.	G1G3C7	A7E3B3
4.	A4B5B7	C6D5C4	5.	B5B6E6	D5C5C6	6.	B6A5A10	E3D2A2
7.	G3H3H5	C5F5C5	8.	H6H7F7	F5F6D8	9.	H7G7G5	F6H6F6
10.	G7H7C2	D10C9A7	11.	A5A3C1	D2D4B2	12.	H3G3H3	H6I7H8
13.	H7I8H7	G4J4E9	14.	D9C8E10	D4D3D7	15.	C8B9C8	D3E4G2
16.	G3F3E3	J4H4F4	17.	F3G3F3	H4I3I5	18.	G3I1D1	I3H2G3
19.	I1I2I3	H2I1F1	20.	I2J3H1	I1J2I2	21.	I8H9I8	I7I6J5
22.	H9H10J8	I6H6F8	23.	J3J4J3	H6G6F5	24.	H10H9G8	C9B10D10
25.	B9C9B9	B10A9B10	26.	C9B8A8				

Game 10: 8QP **(White) vs** Campya **(Black); Result: White+6**

1.	D1D8B6	G10G3G8	2.	J4E9E10	A7E7B4	3.	E9I9J8	D10C9C7
4.	A4A8F3	E7C5D5	5.	G1D4C4	J7I7G5	6.	D8D7D6	C5A5A7
7.	D4D1A4	G3E1E6	8.	D7H7D7	E1D2C2	9.	H7H4F2	C9C8B9
10.	H4I4D4	C8E8C8	11.	I9G9D9	I7I5G7	12.	D1C1H1	E8F8D10
13.	A8B7A6	I5H5H8	14.	B7C6B5	H5H4H5	15.	G9F10D8	F8G9F9
16.	F10I10I5	G9I9G9	17.	I10G10I8	H4H3H4	18.	I4J5I4	I9H9H10
19.	J5J6J1	H3F5F8	20.	J6I6F6	F5H7H6	21.	I6J5I6	H7F5I2
22.	J5J3H3	F5E5E1	23.	J3J4G1	E5H2G2	24.	J4J7J2	D2D3D2
25.	J7H7E4	D3A3B2	26.	C1D1A1	A3E3E2	27.	H7F5F4	H2J4J7
28.	D1B1A2	J4H2G3	29.	C6A8A10	H2J4J3	30.	F5H7I7	J4H2I1

Game 11: Campya **(White) vs** BIT **(Black); Result: White+5**

1.	D1D9G9	A7E3H6	2.	J4F4F9	D10B8G8	3.	G1H2H5	J7J3G3
4.	H2I3I8	G10J10F6	5.	A4B4F8	B8C8C10	6.	B4C3D2	J3I4J4
7.	F4E4D4	E3D3F3	8.	E4H7E4	I4H3H2	9.	H7H8H10	J10J8I7
10.	D9D8D9	D3C2D3	11.	C3B3B8	C2B2C2	12.	H8G7I9	C8C7C9
13.	G7H7J5	B2A3B4	14.	D8D7D8	A3A1I1	15.	B3A2B2	C7C6C8
16.	H7F5C5	J8J6I5	17.	A2A6A2	C6D5A8	18.	A6C6E6	D5D6F4
19.	F5H7E7	J6J7I6	20.	I3I2I4	A1H1G2	21.	H7F5G5	H3I3H3
22.	C6A6C6	D6C7A7	23.	A6B6B7	C7E5C7	24.	D7D5D6	J7J9J6
25.	F5H7F5	J9I10H9	26.	I2J2I2	H1E1A1	27.	D5C4D5	I10J10I10
28.	J2J3J1	I3J2I3	29.	H7H8H7				

Game 12: Invader **(White) vs** Campya **(Black); Result: Black+0**

1.	G1G9F10	D10D3B5	2.	A4E4E3	J7G4I4	3.	J4G7B7	G10H9H6
4.	G7C7C3	A7C9G5	5.	D1E2I2	G4G3D6	6.	G9G8B3	H9H8H9
7.	E4H4E4	C9C8E6	8.	G8G9C9	D3D2D5	9.	G9G7I7	G3H3F1
10.	G7G8G7	H3G3I3	11.	E2D1F3	D2B2D2	12.	C7A5A3	C8C7A9
13.	A5A8D8	C7B8A7	14.	A8C10A8	B8C8C4	15.	D1C2C1	C8D7E8
16.	C10E10C8	D7C7C5	17.	H4H2F2	C7E7H10	18.	G8H7J5	H8I8G8
19.	E10E9F8	E7F7H5	20.	E9D9G9	F7F6G6	21.	H7I6I5	G3G1G3
22.	I6J7J10	G1H1G2	23.	J7J9J6	H1I1G1	24.	H2H3F5	I1H2H1
25.	J9J8J9	F6E7F7	26.	H3H4H3	E7F6D4	27.	C2D3B1	B2C2A2
28.	D9C10A10	F6E7C7	29.	H4F4E5	H2I1H2	30.	J8I9I10	I1J1J4
31.	C10E10B10	J1J2J3	32.	D3E2D1	E7D7E7	33.	E10F9G10	D7C6D7
34.	F9E10F9	C6B6C6	35.	E10D10E10	B6A6A4	36.	D10E9D9	A6A5B4
37.	F4H4F4	I8H8I8	38.	E9D10E9	H8H7I6	39.	D10C10D10	C2B2C2
40.	C10B9B8	B2A1B2	41.	I9J8I9	H7H8H7	42.	H4G4H4	A5A6A5
43.	B9C10B9	A6B6A6	44.	J8J7J8	J2J1J2	45.	E2E1E2	J1I1J1
46.	pass							

Appendix C

Opening-Books

This section will present the best variations of several of the Opening-Books created for the study presented in Chapter 5. We only display here variations for which the number of visits was superior to a given threshold. Of course, the full Opening-Books contain more nodes, but are not included here completely for reasons of space. For each one of them, the number displayed before the parenthesis is the average evaluation of the sequence (in the minimax sense, positive for the first player), and inside the parenthesis, the number of times it was visited during the process.

C.1 Opening-Books created using Meta-MCTS

The following three Opening-Books were created using Meta-MCTS as described in Section 5.4.4. The three of them were created with 1000 nodes. Only variations with more than 10 visits are displayed.

C.1.1 Evaluation based on full games

The following Opening-Book is the one with index 5 in Section 5.4.4. It used full games played in the evaluation phase with 60 seconds per side and an evaluation based on win/loss. The exploration factor was set to 5.

Root 0.17(1000)
D1D7I7 -0.675676(37)
D1D7I7 G10G3D6 -0.833333(36) D1D8B6 0.148515(101)
D1D8B6 G10G3I5 0.2(100)
D1D8B6 G10G3I5 J4G4J1 1(10)
D1D8B6 G10G3I5 J4G4J4 2.05882(17)
D1D8B6 G10G3I5 J4F4F10 1.66667(12)
D1D8B6 G10G3I5 J4F4F10 A7D7B5 1.36364(11)
D1D8B6 G10G3I5 J4F4F10 A7D7B5 D8G8J5 2(10)
D1D8B8 -0.16129(62)
D1D8B8 G10G3G8 -0.0819672(61)
D1D8B8 G10G3G8 G1E3B6 1.66667(15)
D1D8A8 -0.434783(46)
D1D8A8 G10G3G8 -0.333333(45)
D1D8A8 G10G3G8 G1E3B6 1(25)

```
D1D8A8 G10G3G8 G1E3B6 A7D7A7 -1.36364(11)
D1D8G8 -0.37037(54)
D1D8G8 A7D4F2 -0.283019(53)
D1D8G8 A7D4F2 J4H6I6 2(10)
D1D8J8 0.630252(238)
D1D8J8 G10G3I5 0.611814(237)
D1D8J8 G10G3I5 J4G4I2 0.294118(17)
D1D8J8 G10G3I5 J4G4I2 A7D4D1 -1.36364(11)
D1D8J8 G10G3I5 J4G4B9 1.6(50)
D1D8J8 G10G3I5 J4G4B9 A7D4A1 1.36364(11)
D1D8J8 G10G3I5 J4G4B9 A7D4D1 2(10)
D1D8J8 G10G3I5 J4G4B9 A7D4D7 0.555556(18)
D1D8J8 G10G3I5 J4G4G10 1.60714(56)
D1D8J8 G10G3I5 J4G4G10 A7D4D1 -0.238095(21)
D1D8J8 G10G3I5 J4G4G10 A7D4B4 1.84211(19)
D1D8J8 G10G3I5 J4G4G10 A7D4B4 A4B3E3 3.18182(11)
D1D8J8 G10G3I5 J4G4I6 0(14)
D1D8J8 G10G3I5 J4G4I6 A7D4B4 -0.384615(13)
D1D8J8 G10G3I5 J4F4G4 1.42857(42)
D1D8J8 G10G3I5 J4F4G4 A7C5G5 2.36842(19)
D1D8J8 G10G3I5 J4F4G4 A7C5G5 G1H2H7 1(10)
D1D8J8 G10G3I5 J4F4G4 A7D4D7 0.454545(22)
D1D8J8 G10G3I5 J4F4G4 A7D4D7 A4C4C3 1.66667(15)
D1D8E9 -0.142857(70)
D1D8E9 G10G3G8 -0.0724638(69)
D1D8E9 G10G3G8 J4H6B6 1.36364(11)
D1D8E9 G10G3G8 J4G7I7 1(10)
D1D8F10 0.263158(114)
D1D8F10 G10G3G8 0.221239(113)
D1D8F10 G10G3G8 J4H6F4 0.833333(12)
D1D8F10 G10G3G8 J4H6J6 1.66667(24)
D1D8F10 G10G3G8 J4H6J6 A7D4A1 0(16)
D1D8F10 G10G3G8 J4G7I7 1.31579(19)
D1D8F10 G10G3G8 J4G7I7 A7C5B4 1.11111(18)
D1D9H5 -0.526316(38)
D1D9H5 G10G3I5 -0.405405(37)
D1D9G9 0.601852(216)
D1D9G9 J7G4B4 0.627907(215)
D1D9G9 J7G4B4 J4H6G5 0.263158(19)
D1D9G9 J7G4B4 J4H6G5 G10I8G6 0(18)
D1D9G9 J7G4B4 J4H6F4 0.263158(19)
D1D9G9 J7G4B4 J4H6F4 G10I8E8 0(18)
D1D9G9 J7G4B4 J4H6H3 -0.454545(11)
D1D9G9 J7G4B4 J4H6H3 G10I8E8 -1(10)
D1D9G9 J7G4B4 J4H6E6 1.17647(34)
D1D9G9 J7G4B4 J4H6E6 G10I8D8 1.06061(33)
D1D9G9 J7G4B4 J4H6E6 G10I8D8 G1G3H3 2.27273(11)
```

D1D9G9 J7G4B4 J4H6E6 G10I8D8 G1G3H3 A7C5C8 2(10)
D1D9G9 J7G4B4 J4H6B6 1.36364(44)
D1D9G9 J7G4B4 J4H6B6 G10I8D8 1.27907(43)
D1D9G9 J7G4B4 J4H6F8 -0.454545(11)
D1D9G9 J7G4B4 J4H6F8 G10I8G6 -1(10)
D1D9G9 J7G4B4 J4G7B7 1.60377(53)
D1D9G9 J7G4B4 J4G7B7 G10I8D8 1.53846(52)
D1D9G9 J7G4B4 J4G7B7 G10I8D8 G1F2B6 2.03704(27)
D1D9G9 J7G4B4 J4G7B7 G10I8D8 G1F2B6 D10F8E8 1.92308(26)
D1D9G9 J7G4B4 J4G7B7 G10I8D8 G1G3D3 2(20)
D1D9J9 -1.5(20)
D1D9J9 G10G3G9 -1.31579(19)

C.1.2 Evaluation based on short games - low exploration

The following Opening-Book is the one with index 1 in Section 5.4.4. It used 20 moves long games played in the evaluation phase with 60 seconds per side and an evaluation function to evaluate the final position. The exploration factor was set to 5.

Root 0.630396(1000)
D1D7G7 -1.71479(13)
D1D7G7 A7E3E6 -2.51864(12)
D1D7I7 1.04104(296)
D1D7I7 G10G3D6 0.976386(294)
D1D7I7 G10G3D6 G1F2B6 2.19409(90)
D1D7I7 G10G3D6 G1F2B6 D10F8E7 2.62053(36)
D1D7I7 G10G3D6 G1F2B6 D10F8E7 D7C8G4 2.86239(20)
D1D7I7 G10G3D6 G1F2B6 D10F8E7 D7C8G4 G3C3A3 2.01851(19)
D1D7I7 G10G3D6 G1F2B6 D10F8E7 D7C8G4 G3C3A3 J4G7F7 3.81313(14)
D1D7I7 G10G3D6 G1F2B6 D10F8E7 J4G7G9 4.91956(12)
D1D7I7 G10G3D6 G1F2B6 D10F8E7 J4G7G9 J7E2C4 5.32715(11)
D1D7I7 G10G3D6 G1F2B6 D10F8H6 3.26987(18)
D1D7I7 G10G3D6 G1F2B6 D10F8H6 J4I3I6 9.24826(11)
D1D7I7 G10G3D6 G1F2B6 D10F8I5 -1.5992(27)
D1D7I7 G10G3D6 G1F2B6 D10F8I5 J4H4H8 -1.10333(13)
D1D7I7 G10G3D6 J4G7C3 0.602241(19)
D1D7I7 G10G3D6 J4G7C3 D10F8H8 -1.57211(16)
D1D7I7 G10G3D6 J4G7G5 0.56959(22)
D1D7I7 G10G3D6 J4G7G5 D10F8C8 -2.62475(16)
D1D7I7 G10G3D6 J4G7G4 1.73403(39)
D1D7I7 G10G3D6 J4G7G4 D10F8F6 1.65823(19)
D1D7I7 G10G3D6 J4G7G4 J7H9C4 -1.1374(13)
D1D7I7 G10G3D6 J4G7I5 1.61962(37)
D1D7I7 G10G3D6 J4G7I5 D10F8D8 -1.15236(11)
D1D7I7 G10G3D6 J4G7I5 D10F8H8 1.403(13)
D1D7I7 G10G3D6 J4G7G10 0.337919(13)
D1D7I7 G10G3D6 J4G7G10 D10F8C8 -0.940225(11)
D1D7I7 G10G3D6 J4G7J10 0.984593(20)

D1D7I7 G10G3D6 J4G7J10 D10F8F1 -1.91673(10)
D1D7I7 G10G3D6 J4F8F3 0.4723(23)
D1D8A8 0.36762(84)
D1D8A8 G10G3G8 0.421123(83)
D1D8A8 G10G3G8 G1E3E10 0.760466(13)
D1D8A8 G10G3G8 J4H6B6 0.900118(14)
D1D8A8 G10G3G8 J4H6J6 0.223098(10)
D1D8A8 G10G3G8 J4G7G4 1.40407(17)
D1D8G8 0.510722(102)
D1D8G8 A7D4F2 0.380394(101)
D1D8G8 A7D4F2 G1G4G7 2.57559(23)
D1D8G8 A7D4F2 G1G4G7 G10C6F3 1.8546(22)
D1D8G8 A7D4F2 G1G4G7 G10C6F3 J4J6F10 3.14487(15)
D1D8G8 A7D4F2 G1G4G7 G10C6F3 J4J6F10 J7I7J8 0.866426(11)
D1D8G8 A7D4F2 J4H6F6 2.72843(20)
D1D8G8 A7D4F2 J4H6F6 G10B5G5 2.99765(13)
D1D8J8 0.576105(127)
D1D8J8 G10G3A3 -0.00520071(122)
D1D8J8 G10G3A3 J4G7C3 -0.279525(16)
D1D8J8 G10G3A3 J4G7C3 J7H7H1 -1.42855(14)
D1D8J8 G10G3A3 J4G7I5 0.751285(22)
D1D8J8 G10G3A3 J4G7I5 J7H7H8 0.95161(21)
D1D8J8 G10G3A3 J4G7B7 -0.130292(13)
D1D8J8 G10G3A3 J4G7B7 A7C5D4 0.108073(12)
D1D8J8 G10G3A3 J4G7E9 0.870409(29)
D1D8J8 G10G3A3 J4G7E9 J7H7H1 3.22082(19)
D1D8J8 G10G3A3 J4G7E9 J7H7H1 A4C4G4 5.65223(13)
D1D8J8 G10G3A3 J4G7J10 0.914004(25)
D1D8J8 G10G3A3 J4G7J10 A7C5D4 -0.191371(23)
D1D8F10 -0.600461(32)
D1D8F10 G10G3G8 -0.842264(31)
D1D9H5 0.79651(181)
D1D9H5 G10G3I5 0.783889(180)
D1D9H5 G10G3I5 J4F4D2 1.47368(47)
D1D9H5 G10G3I5 J4F4D2 A7D4A7 -4.83129(11)
D1D9H5 G10G3I5 J4F4G4 2.08241(65)
D1D9H5 G10G3I5 J4F4G4 A7C5C1 -1.46198(28)
D1D9H5 G10G3I5 J4F4G4 A7D4A7 1.98742(11)
D1D9H5 G10G3I5 J4F4B8 0.798063(17)
D1D9H5 G10G3I5 J4F4F10 0.904339(20)
D1D9H5 G10G3I5 J4F4F10 A7C5G5 -2.38023(15)
D1D9J9 1.01166(145)
D1D9J9 G10G3G9 0.987372(144)
D1D9J9 G10G3G9 J4H6H10 2.81476(40)
D1D9J9 G10G3G9 J4H6H10 D10F8F1 1.78076(34)
D1D9J9 G10G3G9 J4H6H10 D10F8F1 A4B4G4 3.84356(14)
D1D9J9 G10G3G9 J4G7E5 1.47648(18)

D1D9J9 G10G3G9 J4G7E5 A7C5B4 0.929026(16)
D1D9J9 G10G3G9 J4G7I5 1.38041(24)
D1D9J9 G10G3G9 J4G7I5 A7C5B4 -0.785751(20)
D1D9J9 G10G3G9 J4G7J10 1.71237(23)
D1D9J9 G10G3G9 J4G7J10 A7C5B4 0.696765(20)

C.1.3 Evaluation based on short games - high exploration

The following Opening-Book is the one with index 2 in Section 5.4.4. It used 20 moves long games played in the evaluation phase with 60 seconds per side and an evaluation function to evaluate the final position. The exploration factor was set to 8.

Root 0.643294(1000)
D1D7G7 -2.76791(21)
D1D7G7 A7E3E6 -2.6637(20)
D1D7I7 -1.06907(38)
D1D7I7 G10G3D6 0.815533(21)
D1D7I7 G10G3I5 -3.49977(16)
D1D8B6 -3.67632(17)
D1D8B6 G10G3I5 -4.24206(16)
D1D8B8 -1.85877(26)
D1D8B8 G10G3G8 -1.5666(25)
D1D8A8 1.02428(164)
D1D8A8 G10G3D6 0.644191(42)
D1D8A8 G10G3D6 J4G7G4 4.35407(13)
D1D8A8 G10G3D6 J4G7I7 1.90203(10)
D1D8A8 G10G3G8 0.764956(90)
D1D8A8 G10G3G8 J4H6F4 2.05346(10)
D1D8A8 G10G3G8 J4H6B6 2.704(12)
D1D8A8 G10G3G8 J4H6J8 2.30737(10)
D1D8A8 G10G3G8 J4G7J4 2.51425(12)
D1D8A8 G10G3I5 1.86673(31)
D1D8A8 G10G3I5 J4G4F3 3.54324(14)
D1D8G8 1.48941(507)
D1D8G8 A7D4F2 3.18264(91)
D1D8G8 A7D4F2 J4H6F4 5.42426(12)
D1D8G8 A7D4F2 J4H6H5 5.86061(13)
D1D8G8 A7D4F2 J4H6C6 5.64007(11)
D1D8G8 J7G4G2 5.4141(15)
D1D8G8 J7G4B4 1.85712(174)
D1D8G8 J7G4B4 J4H6E6 -0.629278(11)
D1D8G8 J7G4B4 J4H6E6 G10I8I5 -0.674464(10)
D1D8G8 J7G4B4 J4H6B6 4.51449(89)
D1D8G8 J7G4B4 J4H6B6 G10I8I5 4.18518(86)
D1D8G8 J7G4B4 J4H6B6 G10I8I5 G1G3H3 3.40502(10)
D1D8G8 J7G4B4 J4H6B6 G10I8I5 H6H7B7 6.39509(48)
D1D8G8 J7G4B4 J4H6B6 G10I8I5 H6H7B7 A7A9H2 6.31759(47)
D1D8G8 J7G4B4 J4H6B6 G10I8I5 H6H7B7 A7A9H2 G1G3B8 6.70031(43)

D1D8G8 J7G4B4 J4H6B6 G10I8I5 H6H7B7 A7A9H2 G1G3B8 D10F8F2 6.28713(40)
D1D8G8 J7G4B4 J4H6E9 1.45674(26)
D1D8G8 J7G4B4 J4H6E9 G10I8I5 1.00022(25)
D1D8G8 J7G4C8 0.164541(225)
D1D8G8 J7G4C8 J4H6F4 0.340833(26)
D1D8G8 J7G4C8 J4H6F4 G10I8B1 -0.253862(25)
D1D8G8 J7G4C8 J4H6E6 -0.386989(26)
D1D8G8 J7G4C8 J4H6E6 G10I8I1 -0.794899(24)
D1D8G8 J7G4C8 J4H6B6 1.53499(61)
D1D8G8 J7G4C8 J4H6B6 G10I8I5 1.55998(60)
D1D8G8 J7G4C8 J4H6B6 G10I8I5 G1G2E4 5.3315(15)
D1D8G8 J7G4C8 J4H6B6 G10I8I5 G1G2E4 A7D7A7 3.57757(13)
D1D8G8 J7G4C8 J4H6E9 0.567338(30)
D1D8G8 J7G4C8 J4H6E9 G10I8I5 0.846391(29)
D1D8G8 J7G4C8 J4H6H9 1.16555(49)
D1D8G8 J7G4C8 J4H6H9 A7C5A5 0.91376(48)
D1D8J8 -1.02012(38)
D1D8J8 G10G3I5 -1.56733(37)
D1D8E9 -3.68299(11)
D1D8E9 G10G3G8 -4.84455(10)
D1D8F10 -2.03961(22)
D1D8F10 G10G3G8 -2.07727(21)
D1D9H5 0.591674(153)
D1D9H5 G10G3G9 -1.12113(89)
D1D9H5 G10G3G9 J4G7E5 0.2437(12)
D1D9H5 G10G3G9 J4G7J4 -0.267223(10)
D1D9H5 G10G3G9 J4G7J10 1.92593(22)
D1D9H5 G10G3I5 2.90935(63)
D1D9H5 G10G3I5 J4F4D2 7.40865(16)
D1D9H5 G10G3I5 J4F4F3 5.21165(10)
D1D9H5 G10G3I5 J4F4G4 3.59469(10)

C.2 Opening-Books created using Meta-UCT

The following two Opening-Books were created using Meta-UCT as described in Section 5.4.4. They were created with 10000 nodes. We only display here variations which got more than 40 visits.

C.2.1 Opening-Book created with low exploration

The following Opening-Book is the one with index 10 in Section 5.4.5. The pre-evaluation used to select best moves was done using a 2-depth minimax search. Positions were evaluated with Campya given 200000 playouts, and the pre-evaluation computed using minimax was given a weight factor (the K constant in the equation of Section 5.4.5) of 5. The exploration factor was set to 2.

Root 3.75315(10000)
D1D7G7 3.40149(6016)

D1D7G7 A7C5I5 2.80693(579)
D1D7G7 A7C5I5 G1E3E9 2.41695(41)
D1D7G7 A7C5I5 G1D4G4 3.27229(465)
D1D7G7 A7D4B2 3.42927(67)
D1D7G7 A7D4A1 3.46573(45)
D1D7G7 A7D4D1 3.05803(135)
D1D7G7 A7D4B4 2.84525(447)
D1D7G7 A7D4B4 G1E3D3 3.08947(148)
D1D7G7 A7D4B4 G1E3C3 2.22061(45)
D1D7G7 A7D4B4 G1E3H3 3.02836(120)
D1D7G7 A7D4B4 J4H6I6 4.62822(70)
D1D7G7 A7D4A7 3.36408(53)
D1D7G7 A7D4D6 3.51331(52)
D1D7G7 A7E3B3 3.20129(175)
D1D7G7 A7E3B3 J4H6F4 3.77324(140)
D1D7G7 A7E3A3 1.95899(1454)
D1D7G7 A7E3A3 J4H6J6 1.97686(1435)
D1D7G7 A7E3A3 J4H6J6 D10B8F4 -0.333919(68)
D1D7G7 A7E3A3 J4H6J6 D10B8F4 H6I6B6 0.392601(50)
D1D7G7 A7E3A3 J4H6J6 D10B8C8 -0.568211(135)
D1D7G7 A7E3A3 J4H6J6 D10B8C8 H6I6B6 -0.63624(134)
D1D7G7 A7E3A3 J4H6J6 D10B8E8 0.177051(54)
D1D7G7 A7E3E6 2.96785(351)
D1D7G7 A7E3E6 J4H6H10 2.96735(349)
D1D7G7 A7E3E7 3.94078(46)
D1D7G7 A7E3E8 2.93339(357)
D1D7G7 A7E3E8 J4H6H9 2.59146(62)
D1D7G7 A7E3E8 J4J6F10 4.01725(144)
D1D7G7 A7E3H6 3.124(228)
D1D7G7 A7E3H6 J4J6F10 3.27378(220)
D1D7G7 A7E3J8 3.82775(52)
D1D7G7 A7F2G2 3.4527(65)
D1D7G7 A7F2F6 3.19761(74)
D1D7G7 A7F2I5 3.26237(107)
D1D7G7 A7F2I5 J4F4D2 4.52262(44)
D1D7G7 J7G4B4 4.06692(42)
D1D7G7 J7G4G6 3.62512(57)
D1D7G7 J7G4H5 3.46484(73)
D1D7G7 J7G4I6 3.13193(145)
D1D7G7 J7F3D5 3.68233(104)
D1D7G7 J7E2G4 4.1733(47)
D1D7G7 J7E2G4 G1E3B6 4.53169(40)
D1D9G9 4.28889(3978)
D1D9G9 A7D4B4 4.02058(84)
D1D9G9 A7D4B4 G1E3C3 4.49033(52)
D1D9G9 A7D4D8 3.64171(189)
D1D9G9 A7D4D8 G1E3C3 3.75469(183)

D1D9G9 A7E3E1 4.13299(83)
D1D9G9 A7E3E1 J4H6C6 4.80301(43)
D1D9G9 A7E3F2 4.51765(45)
D1D9G9 A7E3B3 3.46992(426)
D1D9G9 A7E3B3 A4D4D7 3.6056(416)
D1D9G9 A7E3A3 3.58082(205)
D1D9G9 A7E3A3 J4H6J6 3.57838(203)
D1D9G9 A7E3G3 3.43195(435)
D1D9G9 A7E3G3 J4H6F4 4.90332(120)
D1D9G9 A7E3G3 J4H6B6 3.50324(115)
D1D9G9 A7E3H3 4.01632(86)
D1D9G9 A7E3H3 J4G7J4 4.91029(49)
D1D9G9 A7E3I3 3.47311(392)
D1D9G9 A7E3I3 J4H6C6 3.03139(103)
D1D9G9 A7E3I3 J4H6B6 4.01109(215)
D1D9G9 A7E3I3 J4H6J6 2.77222(58)
D1D9G9 A7E3J3 3.74353(167)
D1D9G9 A7E3J3 J4F4F9 5.48524(40)
D1D9G9 A7E3B6 4.00172(72)
D1D9G9 A7E3A7 3.66629(191)
D1D9G9 A7E3A7 J4H6C6 3.97703(166)
D1D9G9 A7E3E6 3.84618(111)
D1D9G9 A7E3E7 3.8068(120)
D1D9G9 A7E3E7 J4H6C6 4.61525(58)
D1D9G9 A7E3G5 3.95496(115)
D1D9G9 A7E3G5 J4H6J6 4.36894(94)
D1D9G9 A7E3H6 5.48083(56)
D1D9G9 A7E3H6 J4F4F9 5.54754(54)
D1D9G9 G10E8I4 5.20103(40)
D1D9G9 G10E8C8 4.61737(67)
D1D9G9 G10E8C8 J4H6B6 5.39317(45)
D1D9G9 G10C6H1 5.39679(40)
D1D9G9 G10B5I5 4.08007(172)
D1D9G9 G10B5I5 A4C4C5 5.62033(72)

C.2.2 Opening-Book created with high exploration

The following Opening-Book is the one with index 8 in Section 5.4.5. The pre-evaluation used to select best moves was done using a 2-depth minimax search. Positions were evaluated with Campya given 200000 playouts, and the pre-evaluation computed using minimax was given a weight factor (the K constant in the equation of Section 5.4.5) of 5. The exploration factor was set to 5.

This Opening-Book is the one currently used in Campya.

Root 4.5609(10000)
D1D7G7 4.53408(1291)
D1D7G7 A7D4B2 1.21808(42)
D1D7G7 A7E3B3 1.78504(42)

D1D7G7 J7F3C6 2.1922(46)
D1D7I7 3.86563(149)
D1D8B6 4.63037(2111)
D1D8B6 G10G3I1 4.71075(50)
D1D8B6 G10G3D6 4.4782(62)
D1D8B6 G10G3G6 4.50491(70)
D1D8B6 G10G3G6 J4G7B7 5.05343(54)
D1D8B6 G10G3G7 4.16313(115)
D1D8B6 G10G3G8 3.9272(114)
D1D8B6 G10G3G8 J4G7B7 4.20265(46)
D1D8B6 G10G3I5 3.08165(954)
D1D8B6 G10G3I5 J4G4F3 3.59011(220)
D1D8B6 G10G3I5 J4G4H3 3.01636(56)
D1D8B6 G10G3I5 J4G4J1 3.27943(99)
D1D8B6 G10G3I5 J4G4J4 3.16469(85)
D1D8B6 G10G3I5 J4G4G10 3.285(113)
D1D8B6 G10G3I5 J4G4I6 3.2625(105)
D1D8B6 G10G3I5 J4F4C7 3.11034(78)
D1D8B6 G10G2C6 4.49092(62)
D1D8B6 G10G2G7 4.44436(98)
D1D8B6 G10G2G7 J4H6H3 5.38861(46)
D1D8B6 G10G2H3 J4H6G7 5.38861(46)
D1D8I8 4.5784(1507)
D1D8I8 G10G3I5 1.6128(59)
D1D8I8 G10G3I5 J4G4G7 1.65936(58)
D1D8J8 4.31453(502)
D1D8E9 4.05217(229)
D1D8F10 3.89657(177)
D1D9G6 4.00714(197)
D1D9G9 4.71189(3785)
D1D9G9 A7D4E3 4.27139(41)
D1D9G9 A7D4F2 3.3546(92)
D1D9G9 A7D4C4 4.01671(41)
D1D9G9 A7D4B4 3.7365(67)
D1D9G9 A7D4D8 3.9352(54)
D1D9G9 A7E3C1 4.02655(52)
D1D9G9 A7E3C3 3.33803(83)
D1D9G9 A7E3B3 2.64331(269)
D1D9G9 A7E3B3 J4H6B6 3.22756(61)
D1D9G9 A7E3B3 J4G7I7 3.29997(68)
D1D9G9 A7E3A3 3.47561(98)
D1D9G9 A7E3A3 A4D4D7 3.99361(49)
D1D9G9 A7E3G3 3.60968(66)
D1D9G9 A7E3H3 3.96299(57)
D1D9G9 A7E3I3 4.28729(51)
D1D9G9 A7E3J3 3.36005(104)
D1D9G9 A7E3C5 4.03874(41)

D1D9G9 A7E3A7 3.70794(73)
D1D9G9 A7E3E6 4.27514(43)
D1D9G9 A7E3E7 3.64346(70)
D1D9G9 A7E3G5 3.34216(108)
D1D9G9 A7E3H6 5.02691(58)
D1D9G9 A7E3H6 J4F4F9 5.08242(56)
D1D9G9 A7E3J8 3.78332(57)
D1D9G9 G10C6C2 3.47849(77)
D1D9G9 G10C6C8 3.29135(87)
D1D9G9 G10B5I5 4.17817(86)
D1D9G9 G10B5J5 4.22506(42)
D1D9G9 G10B5E8 4.91055(45)
D1D9G9 J7G4G5 3.96131(41)

Summary

In the field of Artificial Intelligence (AI), game programming has since the early days of Computer Chess been considered as both a good challenge and a good test-bed to try new AI techniques or models. One of these models especially, called minimax, has become a de facto standard for lots of board games and is still used nowadays in most Chess, Checkers or Othello programs. A successfull implementation of such a program is highly dependent of the quality of its evaluation function, way above quality improvements of the algorithm itself.

Abramson proposed in 1993 a new framework for game-programming based on Monte-Carlo methods. The main concept of minimizing and maximizing results which gave its name to minimax is still present in a Monte-Carlo program but the evaluation, based on random samplings, is fundamentally different. Monte-Carlo programs evaluate moves based on their average expected result using random simulations. They are in their basic form very easy to program and need very few knowledge specific to a game except its rules. Due to their inherent stochastic aspect, Monte-Carlo methods have been shown efficient for stochastic games such as Backgammon or Poker. Thanks to the addition of new improvements like Tree-Search, they have also been shown very efficient to create strong Go programs, one game which since long resisted to the minimax framework. Go evaluation functions are notably difficult to create. But however strong Monte-Carlo with Tree-Search (MCTS) is for the game of Go, it remains to be seen if the method can be applied as efficiently to other classes of game. Since most put forward the fact that Go is a territory game as a reason for the success of MCTS, we decided to stufy the potential of MCTS to a game which would be an intermediate between Go and other classes of game, the game of the Amazons.

After the introduction in Chapter 1, we describe in Chapter 2 the Game of the Amazons and the various reasons why we chose it as a test-bed. While its complexity it not as big as games such as Shogi or Go, the game of the Amazons poses inherent challenges, the main one being a large game-tree. Due to the rules of the game, the number of moves available to one player in the beginning of the game is usually between one thousand and two thousand, making a full width search difficult. Amazons endgames have also been proven to be NP-complete.

In Chapter 3, we present the main part of our work, that is the adaptation of Monte-Carlo techniques to the game of the Amazons. We wish in this section to answer to the following questions: *Is it possible to build a strong Amazons playing program using Monte-Carlo Tree-Search (MCTS) techniques ? Could techniques used for the game of Go or other games be applied for an MCTS program playing Amazons? Could traditional improvements for the game of the Amazons be used in a Monte-Carlo based program?* However strong Monte-Carlo techniques may be for another territory game, the game of Go, we show that these methods cannot be used as-is and require deep adaptations. We also show however that with the correct improvements, in the present case a change in the Monte-Carlo based evaluation by integrating an evaluation function, we can obtain a strong MCTS Amazons program. Improvements for MCTS or classical Amazons programs

have unfortunately to be also deeply modified to improve the playing level of an MCTS program. Some of them work as is like move splitting, while others not at all but provide great boost with the correct implementation like AMAF/RAVE. The explanation as to why is, unfortunately, not always clear.

MCTS programs being usually stronger strategically but weaker tactically, we experimented with Amazons endgames which combinatio problem has been shown to be a difficult task. For this reason we focus in Chapter 4 on the following questions: *Is MCTS able to handle precise playing like one finds in endgame situations? Are specific improvements needed?* To our surprise, even compared with top of the line solvers such as DFPN and Alpha-Beta, MCTS holds its own. And even if it whows weakness in the tactical sense, simple MCTS is able to get slightly better results than minimax-based methods in multiple-subgames playing.

Next, we propose in Chapter 5 our answers to the following questions: *Is it possible to directly use standard Opening-Books (OB) of more traditional programs for Monte-Carlo based programs? If not, is it possible to adapt them to a different playing style? Is it possible to design more specific Opening-Books or methods to create them for Monte-Carlo based programs?* After showing that traditional OB creation methods are not suited for MCTS programs, we propose to use MCTS techniques themselves to create such OB and why they are suited for the task. With the correct changes, mostly revolving around the use of Meta-MCTS, we show that such methods can indeed be used to automatically build OB for an MCTS game playing program. To get to this result, we had to go as far as removing the Monte-Carlo evaluation from MCTS, making the resulting algorithm an hybrid between classical minimax methods and Monte-Carlo methods.

Finally, the conclusion in Chapter 6 summarized our main results as well as the answers to the various research questions enounced so far. Given the results obtained by our MCTS Amazons program CAMPYA, we conclude that MCTS methods have the potential to be used to create strong game playing programs for other games, although that comes with an important price in the adaptation of the playing engine and the improvements to fit to these games. We also propose in this section several potential future research, both around MCTS methods themselves and the game of the Amazons.

Résumé

In the field of Artificial Intelligence (AI), game programming has since the early days of Computer Chess been considered as both a good challenge and a good test-bed to try new AI techniques or models. One of these models especially, called minimax, has become a de facto standard for lots of board games and is still used nowadays in most Chess, Checkers or Othello programs. A successfull implementation of such a program is highly dependent of the quality of its evaluation function, way above quality improvements of the algorithm itself.

Abramson proposed in 1993 a new framework for game-programming based on Monte-Carlo methods. The main concept of minimizing and maximizing results which gave its name to minimax is still present in a Monte-Carlo program but the evaluation, based on random samplings, is fundamentally different. Monte-Carlo programs evaluate moves based on their average expected result using random simulations. They are in their basic form very easy to program and need very few knowledge specific to a game except its rules. Due to their inherent stochastic aspect, Monte-Carlo methods have been shown efficient for stochastic games such as Backgammon or Poker. Thanks to the addition of new improvements like Tree-Search, they have also been shown very efficient to create strong Go programs, one game which since long resisted to the minimax framework. Go evaluation functions are notably difficult to create. But however strong Monte-Carlo with Tree-Search (MCTS) is for the game of Go, it remains to be seen if the method can be applied as efficiently to other classes of game. Since most put forward the fact that Go is a territory game as a reason for the success of MCTS, we decided to stufy the potential of MCTS to a game which would be an intermediate between Go and other classes of game, the game of the Amazons.

After the introduction in Chapter 1, we describe in Chapter 2 the Game of the Amazons and the various reasons why we chose it as a test-bed. While its complexity it not as big as games such as Shogi or Go, the game of the Amazons poses inherent challenges, the main one being a large game-tree. Due to the rules of the game, the number of moves available to one player in the beginning of the game is usually between one thousand and two thousand, making a full width search difficult. Amazons endgames have also been proven to be NP-complete.

In Chapter 3, we present the main part of our work, that is the adaptation of Monte-Carlo techniques to the game of the Amazons. We wish in this section to answer to the following questions: *Is it possible to build a strong Amazons playing program using Monte-Carlo Tree-Search (MCTS) techniques ? Could techniques used for the game of Go or other games be applied for an MCTS program playing Amazons? Could traditional improvements for the game of the Amazons be used in a Monte-Carlo based program?* However strong Monte-Carlo techniques may be for another territory game, the game of Go, we show that these methods cannot be used as-is and require deep adaptations. We also show however that with the correct improvements, in the present case a change in the Monte-Carlo based evaluation by integrating an evaluation function, we can obtain a strong MCTS Amazons program. Improvements for MCTS or classical Amazons programs

have unfortunately to be also deeply modified to improve the playing level of an MCTS program. Some of them work as is like move splitting, while others not at all but provide great boost with the correct implementation like AMAF/RAVE. The explanation as to why is, unfortunately, not always clear.

MCTS programs being usually stronger strategically but weaker tactically, we experimented with Amazons endgames which combinatio problem has been shown to be a difficult task. For this reason we focus in Chapter 4 on the following questions: *Is MCTS able to handle precise playing like one finds in endgame situations? Are specific improvements needed?* To our surprise, even compared with top of the line solvers such as DFPN and Alpha-Beta, MCTS holds its own. And even if it whows weakness in the tactical sense, simple MCTS is able to get slightly better results than minimax-based methods in multiple-subgames playing.

Next, we propose in Chapter 5 our answers to the following questions: *Is it possible to directly use standard Opening-Books (OB) of more traditional programs for Monte-Carlo based programs? If not, is it possible to adapt them to a different playing style? Is it possible to design more specific Opening-Books or methods to create them for Monte-Carlo based programs?* After showing that traditional OB creation methods are not suited for MCTS programs, we propose to use MCTS techniques themselves to create such OB and why they are suited for the task. With the correct changes, mostly revolving around the use of Meta-MCTS, we show that such methods can indeed be used to automatically build OB for an MCTS game playing program. To get to this result, we had to go as far as removing the Monte-Carlo evaluation from MCTS, making the resulting algorithm an hybrid between classical minimax methods and Monte-Carlo methods.

Finally, the conclusion in Chapter 6 summarized our main results as well as the answers to the various research questions enounced so far. Given the results obtained by our MCTS Amazons program Campya, we conclude that MCTS methods have the potential to be used to create strong game playing programs for other games, although that comes with an important price in the adaptation of the playing engine and the improvements to fit to these games. We also propose in this section several potential future research, both around MCTS methods themselves and the game of the Amazons.

Curriculum Vitae

Julien Kloetzer was born in Fontenay sous Bois in France, on the 5th of November 1982, and lived there in the neighborhood of Paris for most of his childhood. In 2002, he successfully entered the Ecole Nationale Superieure des Telecommunications - a typically French kind of engineering high school - in Paris after a nationwide exam and got his diploma in 2005, the equivalent of a Masters Degree in engineering. There, he acquired important skills in programming, artificial intelligence as well as in foreign languages and oral presentations. With the hope to begin working on a Ph.D. thesis, he also entered the Pierre and Marie Curie university in Paris where he obtained in 2006 a Masters degree in research specialized in artificial intelligence.

In Fall 2006, after 24 years of life in France, he left for Japan and entered the Japan Advanced Institute of Science and Technologies (or JAIST), a Post-graduate university located in Kanazawa in the province of Ishikawa. There, he entered officially the JAIST doctoral course in 2007 to work on his game research project at the Computer and Game Research Unit, lead by Professor Hiroyuki Iida. He focused his work on the game of the Amazons, the end result being this published Thesis. He also participated to several international conferences and workshops focused on game programming, and developed the Amazons program Campya which, after three participations in the Computer Olympiad between 2007 and 2009, now plays at a top level. He also took the opportunity of being in a foreign country to both act as a teacher of the French language and as an ambassador of his country by giving inter-cultural talks or lessons in schools and other festivals or institutions. He finally defended successfully his Ph.D. Thesis in February 2010. Starting from April 2010, he shall be employed as a post-doctoral researcher in the same institution for another year at least.