Fast and furious game playing : MonteCarlo drift
Pre-study and analysis report

Prateek Bhatnagar, Baptiste Bignon,
Mikaïl Demirdelen, Gabriel Prevosto,
Dan Seeruttun--Marie, Benoît Viguier

Supervisors : Nikolaos Parlavantzas, Christian Raymond

10/23/2014

Abstract

This project is about developing an artificial intelligence for a board game using the *Monte Carlo Tree Search Algorithm* and is coined as *Fast & Furious Game Playing, Monte Carlo Drift*. The board game chosen for this project is *Arimaa*.

Arimaa was conceived and developed in 2003 by Umar Syed, a computer science engineer. It was intentionally made difficult for computers to play, while following simple rules. Umar Syed offered a prize of 10,000 USD to the first program that can beat a human player in a game of 6 or more matches.

A revolution in scheduling algorithms originated in the *Monte Carlo Tree Search* algorithm (*MCTS*). *MCTS* is a heuristic search algorithm used for making decisions. It concentrates on analyzing the optimum moves by expansion of a search tree of random samplings called the search space. In this project, the techniques of the *MCTS* algorithm are utilized to find what move to make.

The program will be implemented using parallelisation techniques, so as to run it on different systems simultaneously. Eventually, the program will be executed on Grid'5000, a cluster of multi-core machines.

Contents

1 Introduction

In 1997, Deep Blue, a supercomputer built by IBM, won a six games match against Garry Kasparov, the current world chess champion. Humans got beaten in Chess, but remain undefeated in other games. Consequently, researchers are looking for improvements in Artificial Intelligence.

The project is called *Fast & Furious Game Playing, Monte Carlo Drift*. Its purpose is to create an Artificial Intelligence able to compete against humans using the *Monte Carlo Tree Search*.

The focus of this project is on two player strategy board games, while avoiding games already solved[1]. That is why this project is about the game *Arimaa*.

To evaluate the best move, the algorithm develops a tree by creating nodes for all possible moves. Statistics are computed by evaluating these nodes and are results of a deeper exploration of a nodes. The algorithm ~~would~~ then ~~be~~ able to choose the best move according to it.

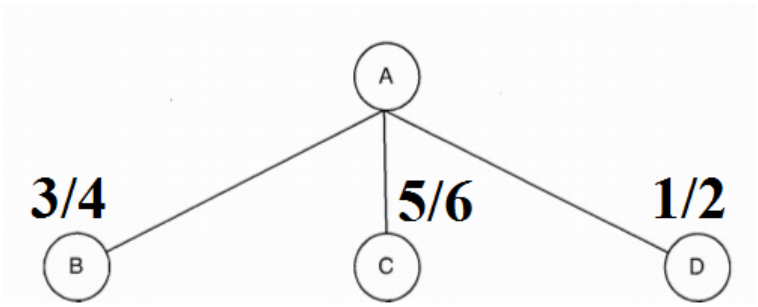


Figure 1: An exploring tree :
The probabilities related to the nodes are the numbers, so the best node is the node B according to the statistics. See more in part use 3.1.

The *Monte Carlo Tree Search* has been used in the past for *Draughts*, or *Chess*. It explores numerous random possibilities, and takes good decisions to win the game. The algorithm would be parallelized in order to use it in a multi-core machine, allowing it to go further into the search tree, thus improving its efficiency.

Different parallelization methods will be studied in order to choose the most suitable, for this project. The exploration of the tree will depend on the parallelization method. The initial phase of the project would be the analysis of the latest papers concerning the technologies that might be of use. The consecutive phases will be about making a choice among

[1] A game solved is a game where good algorithms are able to find the perfect move in each situation to win, or to draw. For instance, *Tic Tac Toe* or *Draughts* are solved games.

these technologies, to decide on the specifications.

Finally, in the last phase, a solution will be implemented, and executed on Grid'5000, a cluster of multi-core machines. The interesting part about this project is the creation of an artificial intelligence as optimized as possible.

2 The Game

2.1 Presentation of Arimaa

Arimaa is played on a board composed of 64 tiles, similar to a chess board. Like Chess, there are 6 types of pieces, instead they are totally different from those in Chess. From weakest to strongest, they are : rabbits (8 per player), cats, dogs, horses (2 of each per player), camels and elephants (one of each per player).

Figure 2: The different piece types in Arimaa.

Each player, starting with the gold player, places all of ones pieces on the two back rows of ones respective side. Then, the gold player plays the first turn. On ones turn, each player disposes of four moves. One can use these moves on a single piece, or on as many pieces as they desire.

All pieces can move on an adjacent square (but not diagonally), except for the rabbit which cannot move backwards. A piece can instead use two moves to push or pull a weaker adjacent enemy piece, as shown in figure 3.

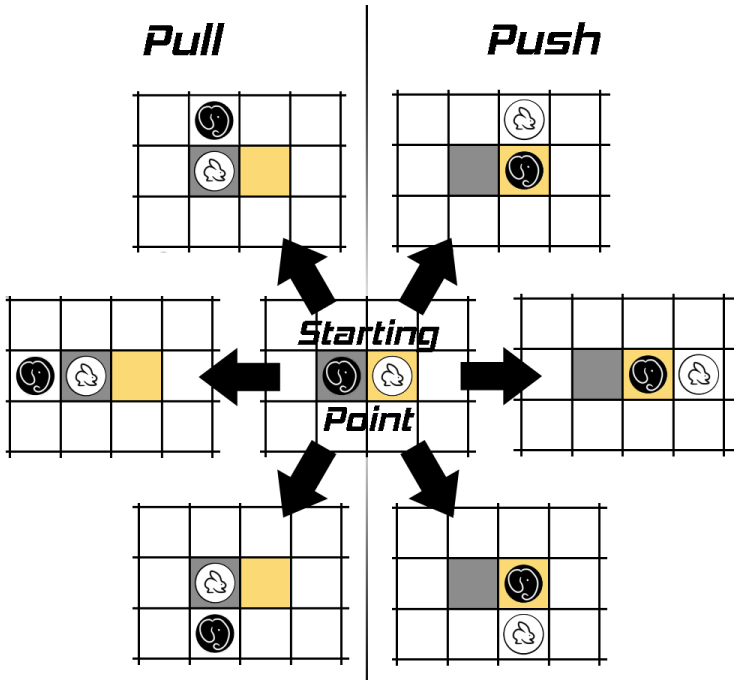


Figure 3: The different ways you can push or pull a weaker enemy piece.

A piece placed adjacent to a stronger enemy piece is frozen. When a piece is frozen, it

cannot move. As shown in figure 4, a piece cannot be frozen while there is an ally piece adjacent to it.

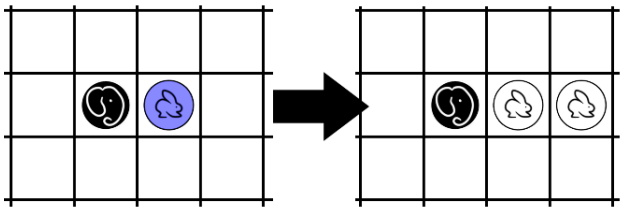


Figure 4: Example of the freezing mechanic. When the two rabbits are together, the one next to the elephant is no longer frozen.

There are four traps on the board. As shown in figure 5, any piece sitting on a trap with no ally piece next to it dies.

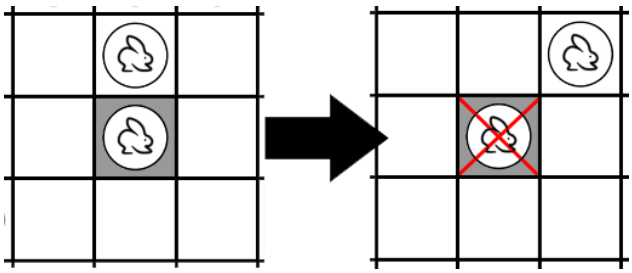


Figure 5: Example of the traps mechanic. As soon as there is no other ally pieces adjacent to the rabbit standing on the trap, it dies.

There are four ways to win the game:

- Victory by reaching the goal: A player wins the game if one of the rabbits reaches the other end of the board.
- Victory by elimination: A player wins the game if one eliminates all the rabbits belonging to his opponent.
- Victory by elimination: A player wins the game if ones opponent can't make a move on his turn.
- Victory by repetition: If the same position happens three times in a row, the player that makes it happen the third time, loses the game.

3 Algorithms

3.1 Search tree

In order to find the best possible move starting from a position, algorithms build search trees. For example Tic-Tac-Toe: [3]

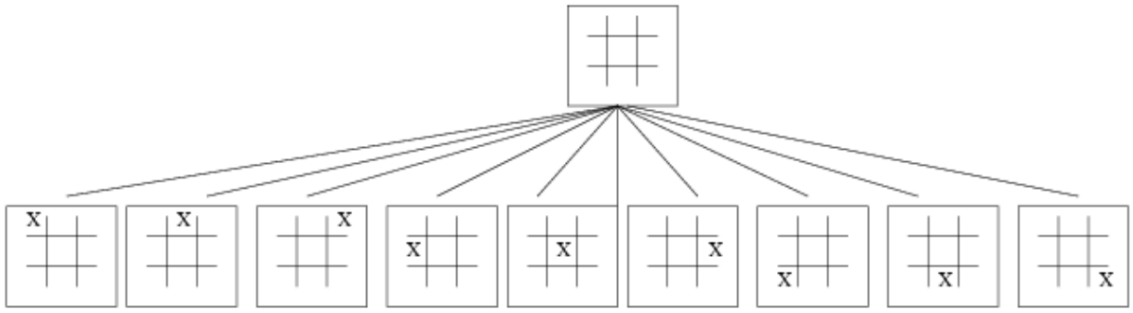


Figure 6: All possible moves starting from an empty board.

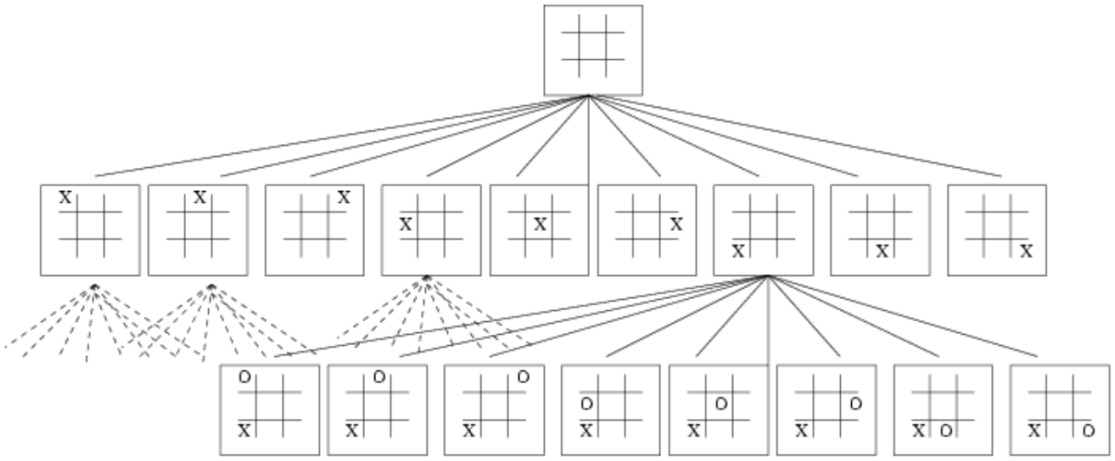


Figure 7: List of all possible moves for the other player.

This expansion is continued until a winning position for the required player is found. Such a tree is called a search tree.

However depending on the number of moves possible for the chosen game, the tree might be too large to be explored completely. Therefore in order to simplify the search, the algorithm will try to evaluate the odds of winning in each position that is explored. Those values will be stored in each node and used by the program to find a good move to play from the present position.

3.2 The Minimax algorithm

The Minimax algorithm is a way of finding an optimal move in a two player game. In the search tree for a two player game, there are two kinds of nodes, nodes representing ones moves and nodes representing the opponent's moves.[1]

Figure 8: Nodes representing ones moves are generally drawn as squares, these are also called *MAX* nodes.

Figure 9: Nodes representing the opponent's moves are generally drawn as circles, these are also called *MIN* nodes.

The goal of a *MAX/MIN* node is to maximize/minimize the value of the subtree rooted at that node. To do this, a *MAX/MIN* node chooses the child with the greatest/smallest value, and that becomes the value of the *MAX/MIN* node.

Note that it's typical for two player games to have different branching factors at each node. The move one makes could have an impact on what moves are possible for the opponent. In this example, one is ignoring what the game is in order to focus on the algorithm.

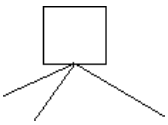


Figure 10: At the start of the problem, Minimax checks the single present node.

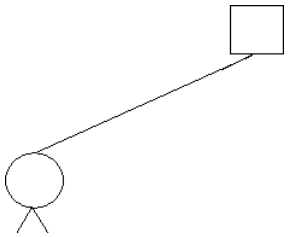


Figure 11: It begins like a depth first search, generating the first child.

So far we've really seen no evaluation values. The way Minimax works is to go down a specified number of full moves (where one *full move*' is actually a move by each player), then calculate the evaluation values for states at that depth. For this example, we're going to go down one full move, which is one more level.

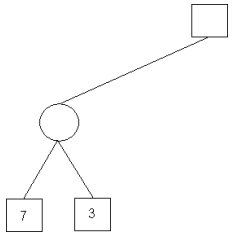


Figure 12: we generate the values for those nodes.

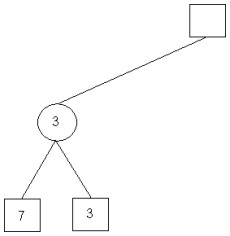


Figure 13: It chooses the minimum of the two child node values, which is 3.

The max node at the top still has two other children nodes that we need to generate and search.

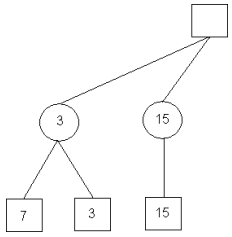


Figure 14: Since there is only one child, the min node must take it's value.

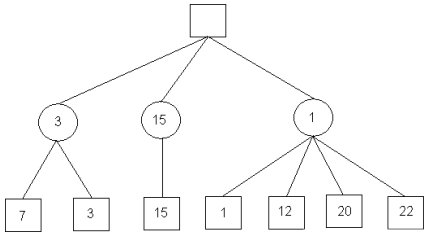


Figure 15: The third min node chooses the minimum of it's child node values, 1.

Finally we have all of the values of the children of the max node at the top level, so it chooses the maximum of them, 15, and we get the final solution.

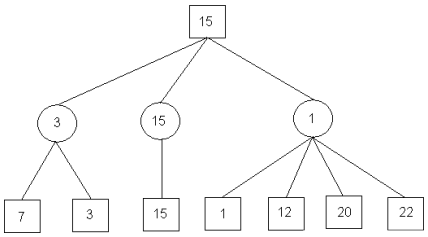


Figure 16: Final tree.

What this tells us is that we should take the move that leads to the middle min node, since it'll lead to the best possible state for us one full move down the road.

3.3 The $\alpha\beta$ pruning

The $\alpha\beta$ method is a heuristic that decrease the number of leaf that will be explored by the Minimax algorithm. That way, the size of the tree will be smaller, the algorithm will be able to dive further and the time spend on more interesting subtree is greater.
If the leaf's position is less interesting than its parents, the algorithm won't explore anyfurther.

3.4 Monte Carlo Tree Search Algorithm

3.4.1 Introduction

Monte Carlo Tree Search (MCTS) is an algorithm used for making optimal decisions in Artificial Intelligence (AI) problems such as solving games or decision making in project managment. It is based on making a big number of random simulations in order to get trustfull datas. To make such simulations, the program play the moves randomly for each players. Once it reach a conclusion (win or loss), the program compute the statistics to get the odds of winning.

3.4.2 How does it works ?

The Algorithm create a tree with all possible solution with a small depth. Then it start to run random simulations starting from the leaves in order to test the odds of the outcome. Once we got enough the results (usually we are using time based simulations) we feed back the results and make the decision depending on the odds of each subsequent leaves.

3.4.3 Example

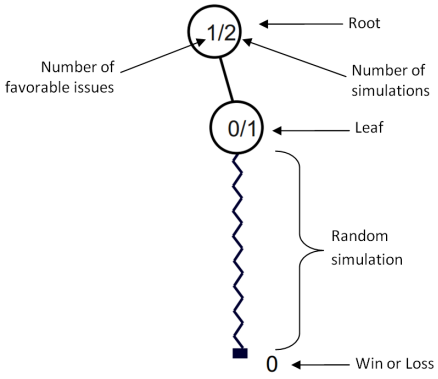


Figure 17: Legend of the following figures.

Figure 18: Run a first simulation from the root, get a favorable issue (will be considered as a *win*).

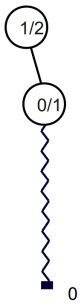


Figure 19: Create a first leaf at depth 1 and run the simulation, get an unfavorable issue (considered as a *loss*).

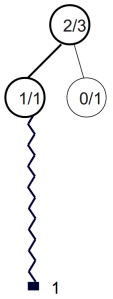


Figure 20: Create a second leaf at depth 1 and run the simulation (*win*).

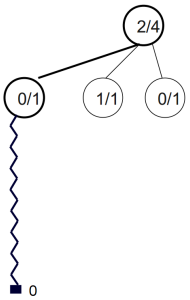


Figure 21: Create a third leaf at depth 1 and run the simulation (*loss*).

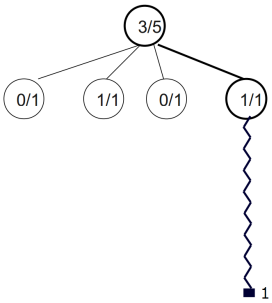


Figure 22: Create a fourth leaf at depth 1 and run the simulation (*win*).

Right now the odds of winning are 3/5. Now that we tested all the possible outcomes at depth 1, we will expend the tree on the favorable leaves (here the second and fourth).

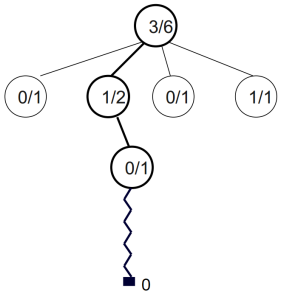


Figure 23: Create a leaf at depth 2 with parent the 2nd leaf at depth 1 and run the simulation (*loss*), update the odds value of the node and making it less interesting than the fourth node. Therefore the algorithm will now work on the fourth node.

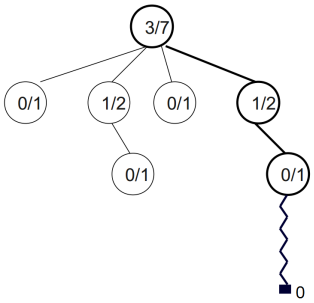


Figure 24: Create a leaf at depth 2 with parent the fourth leaf at depth 1 and run the simulation (*loss*), update the odds value of the node and making it as interesting as the second node. The algorithm will now work on the second node.

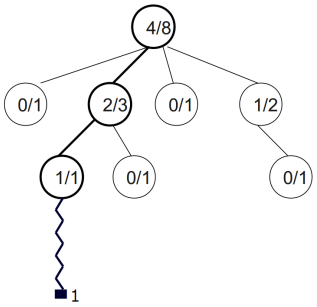


Figure 25: Create a second leaf at depth 2 with parent the second leaf at depth 1 and run simulation (*win*), update the odds value and continue to develop this leaf.

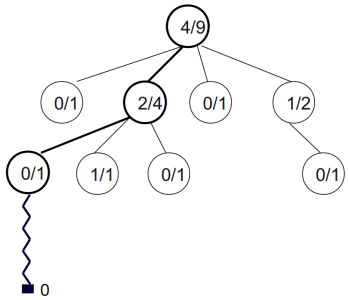
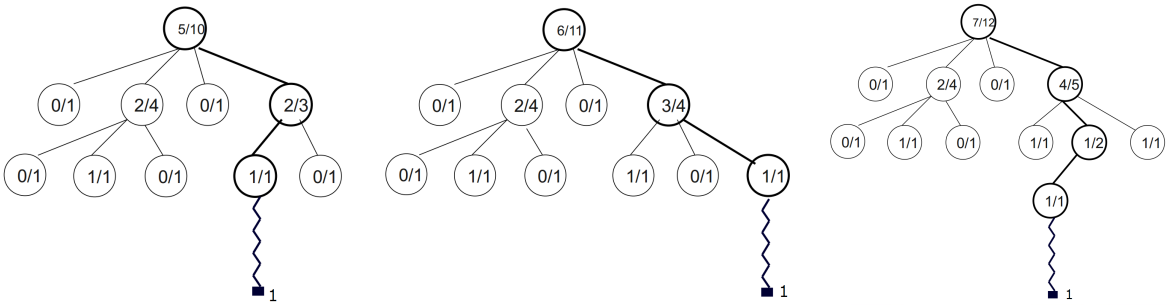


Figure 26: Create a third leaf at depth 2 with parent the second leaf at depth 1 and run simulation (*loss*), update the odds value and switch to the fourth leaf.

Continue the Algorithm until a decent about of simulation are run and/or the time limit is .



Make a decision : here we chose the fourth leaf.

3.4.4 How to select the leaves to develop ?

In the previous exemple, we chose to not expend leaves without winrates. But depending on the results of the simulations, wins can vary greatly. Therefore we will run more simulations on each leaf before chosing the ones to develop. For practical purpose we will select the leaves to expend that has the highest value of the cost function UCT (Upper Confidence Bound 1 applied to Trees).

$$f = \frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}}$$

UCT function[2]

- w_i : number of wins after the ith node
- n_i : number of simulations after the ith node
- c : exploration parameter – theoretically equal to $\sqrt{2}$ but in practice chosen empirically
- t : total number of simulations in a given tree node, equal to the sum of all n_i

The more a leaf is developed, the less it's cost is worth it. This way we can be sure that a leaf with low winrate isn't completely forgotten.

3.4.5 Why using the Monte Carlo Tree Search ?

Compared to other algorithm like Minimax, this one is generic, once you set the rules of the games, given enough time, it will solve it. The advantage of MCTS with it's basic form is that you don't need to implement functions to improve the researches. Based on its random simulations, it will determine by itself which are the good options and which aren't.
The more you run simulations, the more accurate the results will be.

3.4.6 How much power do we need ?

The more the game has possible moves, the more power it require to solve. In order to get plausible decisions, it needs to go deeper in the tree and to search enough leaves. If the time or number of simulations is not sufficient, the algorithm might miss some important branches and fail to give plausible results. Therefore in order to get decent results, using high-end computer is mandatory, it allows us to get access to multi-threading technology in order to parallelize the simulations.

4 Strategies and state of the art

4.1 Strategy of root parallelization

Now, let's see how we can parallelize the Monte-Carlo Tree Search to optimize our algorithm. A classic MCTS is an algorithm which sequentially creates random development of the game. In order to speed up the results, develop more nodes of the tree search or even have more realistic statistics, we will parallelize our tree. It means that we will distribute parts of the tree development to multiple threads, among multiple computers. Therefore, each thread on each computer will have less executions and our algorithm will be more efficient.

Currently, there are three principal strategies about how to parallelize the tree. They are called: Leaf Parallelization, Root Parallelization and Tree Parallelization[6, 8].

4.2 Leaf Parallelization

The Leaf Parallelization is the easiest way to parallelize the tree. In this method, only one thread traverses the tree and adds one or more nodes to the tree when the leaf node is reached. Then, all the threads will independently play the game. Once they all have finished, they back-propagate their results to the leaf and then, only one thread change the tree's global results. The Leaf Parallelization method is depicted in figure 27.

The advantage of this method is that its implementation seems very simple. The threads don't need to be synchonized. However, there are two major problems.

- We don't know the time it will take to a thread to finish the game. Therefore, it will take, in average, more time to do n games with n threads, than one game with one thread, since this method wait for the last one.
- There is no communication between the threads. If a majority of the threads, the faster ones, have led to a loss, it will be very likely than all of them lead to a loss. And so, we will develop the last one for nothing. ...

4.3 Root Parallelization

The second method is the Root Parallelization. It consists in giving each thread the same tree during the same amount of time. They will independently and randomly develop their tree and, at the end of the time, they will merge all of the results. This method can also be called "Single-run parallelization" or "Slow-Tree Parallelization", for instance, and is depicted in figure 27.

Its advantage is also one of its drawbacks. Indeed, the threads don't communicate between each other. On one hand, it means there is some redundancy is the development of the tree. On the other hand, the lack of communication increases drastically the speed of the program. Actually, the strength of this strategy lies in the little communication.

4.4 Tree Parallelization

Finally, the third method is called the Tree Parallelization. In this method, multiples thread share the same tree and, they can randomly choose a leaf and develop it. The main problem of this method is that multiple threads can access the same node and corrupt data given by another thread. To prevent this corruption there is two main methods proposed by Guillaume Chaslot, Mark Winands, and H. Jaap van den Herik [6]. The first one is to put mutexes, or mutual exclusions (mutex), on the tree and the second is to implement a virtual loss. The first method is depicted in figure 27. The other one is explained below.

The mutexes can be either global or local. The global mutexes block access to the whole tree when a thread is accessing it. Yet, the lock causes a major loss of time when the average of the thread's execution is high which is often the case. The local mutexes block only the node that the thread is using in order not to block the entire tree. This method is better but still implies an important number of locking and unlocking. However, we can use fast-access mutexes and spinlocks to increase the speed of the program.

Nevertheless, according to Markus Enzenberger and Martin Müller [9], the data which could be corrupted by the lack of mutexes are negligible compared to the speed decrease of the program, especially when the number of threads exceeds 2. Therefore, we can assume that, to be the most efficient, we can simply suppress mutexes.

The second method, the virtual loss, consists in decreasing the value of the node the first thread access. When the second thread searches a node to develop, he'll take this node only if it's considerably better than the others. This strategy allows nodes with high percentages of winning to be visited by multiple nodes and avoid redundancy on the other ones.

4.5 Comparison

Currently, the best strategy to adopt is the Root parallelization on those specific test conditions. According to some tests [6, 10], the advantages of the Root parallelization outcomes its drawbacks. Indeed, for 16 threads, a program using Root Parallelization will win 56% of the time against 36.5% for the Leaf parallelization, and 49% for the Tree parallelization with the use of a virtual loss. Moreover, the Root is always twice faster than the Tree with virtual loss. We can explain that by the fact that numerous trees are massive, and so, you will lose much time doing communication and synchronization. You don't have any of those problems with the Root strategy. Furthermore, this method is also very simple to compute.

4.6 Hybrid Algorithms

Now, we can wonder if there is some research which has already be done on hybrids technologies of parallelization. Some parallelization strategies are a combination of multiples methods with some additional content, but they are very complex to compute and are efficient in specific case.

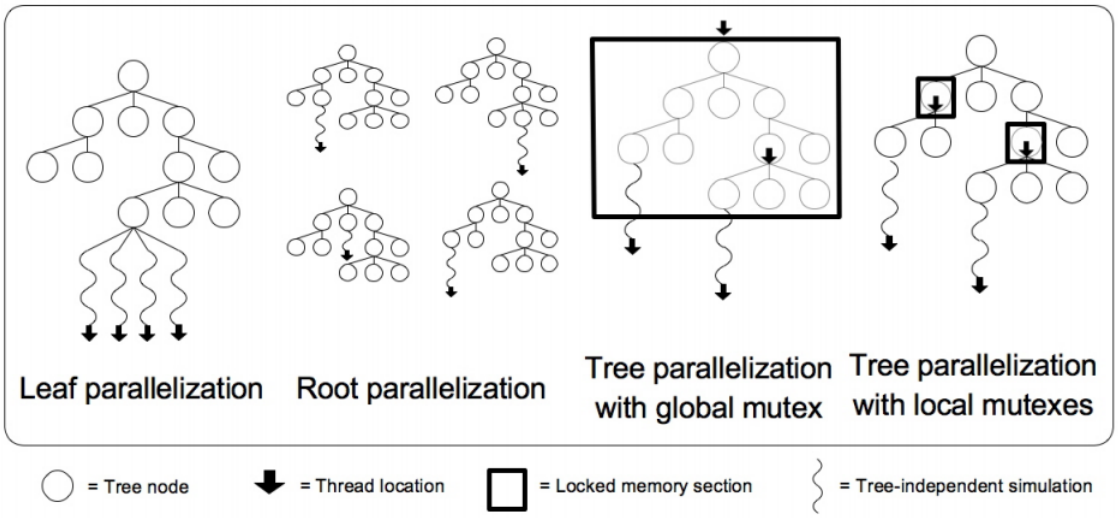


Figure 27: Comparison of all trees

4.6.1 UCT-Treesplit

The UCT-Treesplit algorithm[11] retakes the base of Root algorithm and add to it work clusters on which compute nodes can process. It is depicted in figure 28. It's made for High-Performance Computers (or HPC) which have cluster parallelization and shared memory parallelization. Moreover, this method demands a lot of communication and so, the network latency is a very important factor of slow-down.

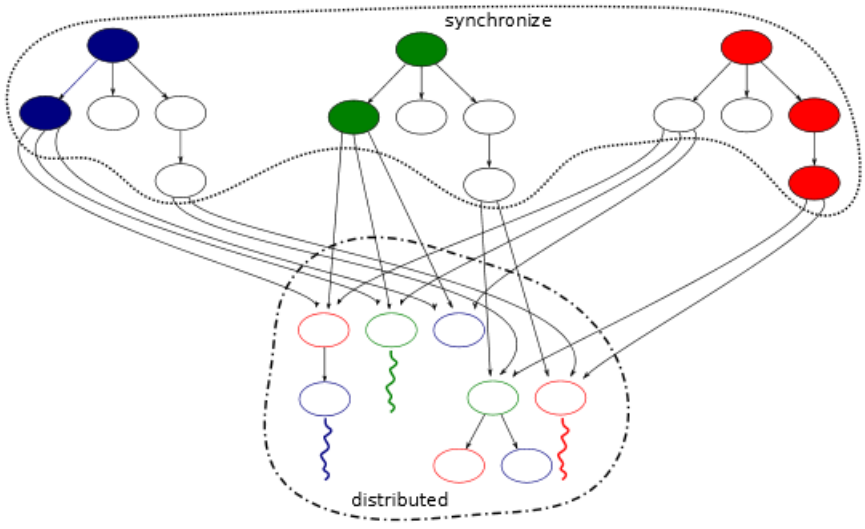


Figure 28: UCT-Treesplit scheme

4.6.2 Block Parallelization

Another hybrid algorithm that is efficient is the Block Algorithm[7]. It is depicted in figure 29. It's working by giving some instructions to GPUs and some other instructions to CPUs. The Block Parallelization is a blending of Leaf Algorithm and Root Algorithm. As GPUs can run hundred of threads, it is used to develop a specific node with Leaf strategy whereas CPU is used to develop the trees using the Root strategy.The benefit of this method is that more since each of these components are used in the way they have been made for.

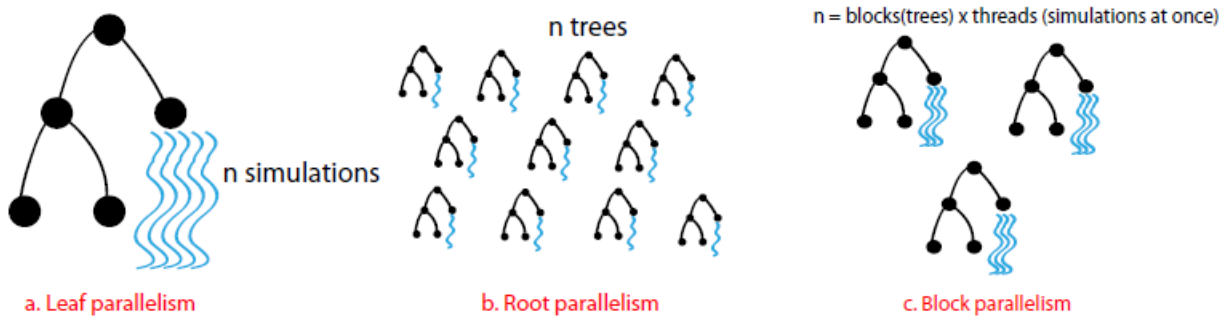


Figure 29: Scheme of the Block Parallelization comparing to Leaf and Root

4.7 Conclusion

The strategy we will use to parralelize our algorithm depends on the hardware we will use. Since we will use both cluster parallelization and shared memory parallelization, we can choose a simple Root parallelization between the different computers, and them, use another algorithm, specialized in shared memory parallelization, like Tree Parallelization. If we prefer to use GPU instead of CPU or even both of them, we can use the Block Parallelization. Also, if our network is fast and our computers fast we can use UCT-Treesplit to parallelize our algorithm. We just have to choose what's best for our configuration.s

4.8 State of the art of MCTS

4.8.1 History of computers vs Humans

Until 2002, methods based on decompositions and positions evaluations were used in order to solve such games. From 2002 to 2005 the Monte Carlo algorithm was used in order to find the best moves. Since 2006, it's implementation in a tree (MCTS) has been developped, rocketing the results in term of Artificial Intelligence on Go. On june 5th, 2013, Zen a Go programm defeated Takuto Ooomote (9 Dan) with a 3 stone handicap.[4]

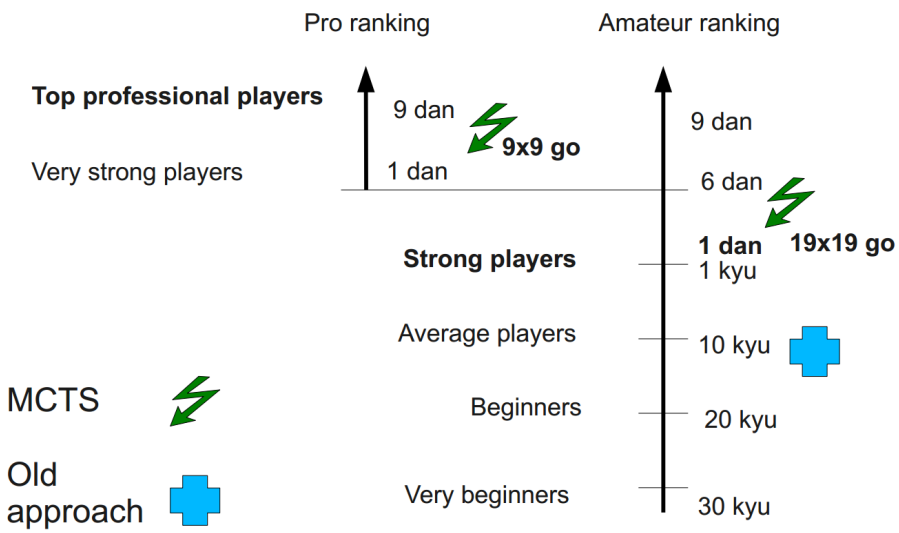


Figure 30: Comparison between Go algorithms and human skill (*2011*)[5].

The ranking system of the game of Go is the following : kyu are for students ranks, Dan are masters ranks. Beginners start at 30 kyu and advance downward the kyu grades. Once one ranks over 1 kyu, he will receive the 1st Dan (as the black belt in Judo) and will move upward through the Dan ranks until the 9th.
However MCTS wasn't the first method applied in order to solve Arimaa, the $\alpha\beta$ method was used first. At the moment the top programms (Bomb by David Fotland : 2002 to 2008, Clueless by Jeff Bacher 2009) are ranked about 1800 elo[2]. For comparison, strongest humans players are rated around 2450 elo.[8]

[2] *The Elo rating system is a method for calculating the relative skill levels of players in competitor-versus-competitor games such as chess. It use also used for Arimaa. Beginers rank around 1200 elo, experts around 2000 elo and International Masters over 2400 elo.*

5 Solutions and schedule

5.1 Candidates software and technologies

5.1.1 Language

At the beginning, the first choice for the project was C++ for coding the software. This is for simple reasons, C++ allows to create in most of the cases better software. Moreover it has several complete graphical libraries, like Qt or SFML, so it's easy to create the graphical interface that we need for the project.

5.1.2 Software

The choice of the sofware solution is fixed so that it can be used to develop the project. The coding part will be realized on Microsoft visual studio 2013, and the version manager will be Git. These two software are often used in the industry so it's better to be familiar with them. For the bibliography, Zotero will be used, it allows the creation of .bib, the bibliographic format that will be used to write the reports.

5.1.3 Grid'5000

Grid'5000 is a cluster of computers shared between many sites in France, it links a lot of computer of different centers of research. In addition of good CPU, some of this machines have a NVIDIA Tesla GPU that could be used to parallelize the MCTS. Of Course, use of this cluster will require some specific parallelization that will be presented in the next part.

5.1.4 Parallelization

Multi cores parrelization

OpenMp is a very simple and efficient API[3] that supports parallelization. It consists some pre-compilation instructions, with only a few lines of code. It allows to obtain a parallel version of the algorithm. As it can be seen in the figure 31, it uses a large part of shared memory which allows a quick and efficient communication between the different threads. But it's not designed for using a cluster of computers, because the memory access will not be efficiant for computers which are far away. But there is a method to avoid this problem that is addressed later.

[3] Application Programming Interface

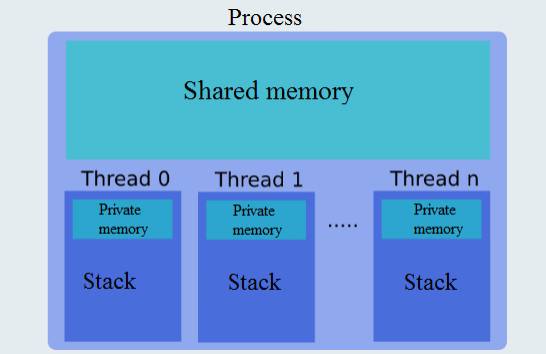


Figure 31: OpenMp : memory management

Computer parallelization

MPI is a more complex method of parallelization than openMp but it is more complete, in addition to use several cores on a single machine, MPI can also be used to designe the software that uses a cluster of machines. As it can be seen in the figure 32, it doesn't use shared memory, all the data of the threads are duplicated at their creation. It uses signals to permit the communication between the threads. One of its advantages is that it makes possible to use multiple computers, as the data are duplicated there isn't the problem of time to access the memory. But attention is to be paid for the cost of communication between the threads, if too many signals are used, there will be loss of time that was gained using the parallelization. It's adapted for tree parallelization, because the only things to be done is to duplicate the data at the beginning and return the result at the end of the assign time.

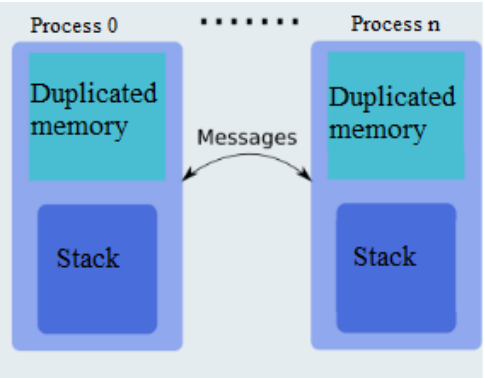


Figure 32: MPI : memory management

Hybrid parallelization

As it is seen that openMp and MPI have their respective pros and cons, but problems can be reduced by using both. That means MPI can be used to dispatch the work onto the

different computers and successively openMp can be used to divide it on different threads of each machine. This can permit to use together multiple parallelization strategy like tree between the machine and leaf or root on each machine. Also, it reduces the manual treatment that is to be defined in the code to manage the threads. It also means that as the interaction between the threads is reduced, so maximum of the parallel code is kept.

Boost Library

Boost is a classical library of C++ which permits the management of threads, amongst other things.Bbasically, it can only be used for one computer but it also contains a socket handling which should easily permit a parallelization between several machines. The library is simple to use and complete. Also, it holds tools for graph modelisation that can improve a lot of efficiency of the algorithms.

GPU implementation

During the research,it was observed that efficiancy of MCTS algorithms improved by using GPU.As Kamil Rocki said in his thesis "Large Scale Monte Carlo Tree Search on GPU" [7] : one GPU's performance is equivalent to 50 CPU threads. But this implementation has some defaults, in fact that Gpu possess very less cache memory, so if the data model is too big the parallelization will be inefficient. Also the trees that it creates will be less deep than those of a CPU, but when a CPU can develop two branches, a GPU will be developing hundreds of branches simultaneously. Another thing that is to be kept in mind is, a GPU can switch to another thread immediately without any cost of context switching.

Grid5000 has NVIDIA GPU, so one of the possibility is to use hybrid as it was described previously by using MPI (or boost library) and openAcc (an equivalent of openMp which allows to use GPU) or CUBA (a framework for GPU parallelization develop by NVIDIA). In this way great trees will be developed and have a better solution compared with using only CPU, even if the trees will be less deep.

5.2 Planning ~~management~~

To manage the work, MS Project will be used to schedule all the tasks like who does what, and how long it will be. This part is about the functioning of the schedule

At the start of the project, we added on MS Project all the deadlines that are demanding by the bill of the project. This deadlines are the different reports (analysis, specifications, ..), the oral presentation and the rendering of the code of the project. After this, each week we decide the tasks we are going to work on for next week, then we add it on the planning and associated each one with the person who will do it.

For the moment we don't plan the coding part because until we have chosen all the technologies we will use, there are many tasks that we can't estimate the length. The only one we have done is to create a model of the game and its graphical interface to allow us to play the game. This will permit us to better understand the game and make easier the implementation of the artificial intelligence.

The second part plan is the UML modeling. During the preparation of the next report, we will dedicate one week to it, with the specifications that we will define it will allow us to plan almost completely the coding of our project.

Another important point is the sharing of our roles. For the moment we have define 3 specifics roles :

- Dan is the secretary. It means he has to organise the meetings and write their reports.
- Gabriel is the master application, he has to the final decision about each point of the coding part.
- Baptiste is in charge of the planning, he will set the deadlines of every tasks, and check if the deadlines are respected.

Moreover each people is archivist, all of us have to search the informations that we need to realize our software. Those who don't have a special role for the moment will be responsible of one of the next reports.

6 Conclusion

Finally, our project is about creating an Artificial Intelligence able to compete with others computers.

Arimaa is a two players game. It has been designed to be difficult to foresee for computers, but easy to play for humans. In order to realize our project, we will need some concepts and technologies. We will use the MonteCarlo Tree Research algorithm, to take the best decisions in our game. We will decide what move to play according to this algorithm figures. Because of the numerous moves possible, we will need to choose to develop some of the better ones, that will depend on the chosen parallelization tree. Any variation could totally change statistics. We already analyse the state of the art of Arimaa and the Monte-Carlo Tree Research. Consequently, we will base our work on these thesis, to best predict decisions, without doing what has already been done. We already know how to play this game. So it would be easier to create strategies for our Artificial Intelligence.

The point of this project is as well to test our program upon Grid5000, a network of multi-core machines. Then we will be able to compete with other computers, comparing algorithms and power of calculus.

References

[1] http://cs.ucla.edu/ rosen/161/notes/minimax.html.

[2] http://en.wikipedia.org/wiki/monte_carlo_tree_search#exploration_and_exploitation.

[3] https://www.cs.tcd.ie/glenn.strong/3d5/minimax-notes.pdf.

[4] http://www.computer-go.info/h-c/.

[5] Bruno Bouzy. http://www.slideshare.net/bigmc/montecarlo-tree-search-for-the-game-of-go, 2011.

[6] Guillaume Chaslot, Mark Winands, and H. Jaap van den Herik. *Paralel Monte Carlo Tree Search*. PhD thesis.

[7] Kamil Rocki and Suda Reiji. *Parallel Monte Carlo Tree Search on GPU*. PhD thesis.

[8] Tomas Kozelek. Method of mcts and the game arimaa. Master's thesis, 2009.

[9] Markus Enzenberger and Martin Müller. *A Lock-free Multithreaded Monte-Carlo Tree Search Algorithm*. PhD thesis.

[10] Méhat Jean and Cazenave Tristan. Tree parallelization of ary on a cluster.

[11] Tobias Graf, Ulf Lorenz, Marco Platzner, and Lars Schaefers. *Parallel Monte-Carlo Tree Search for HPC Systems*. PhD thesis.