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

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

Encadrants : Nikolaos Parlavantzas, Christian Raymond

10/15/2014

Abstract

This project is about inducing artificial intelligence for board games 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*.

Arima was conceived and developed by Umar Syed, a computer science engineer in 2003. It was intentionnaly made difficult for computers to play, whie keeping simple rules.The first program that can beat a human player in a game of 6 or more matches is entitled to win a prize of 10000 USD, given by Umar Syed.

A revolution in scheduling algorithms originated in the *Monte Carlo Tree search Algorithm* (*MCTS*). In this project, the techniques of the *MCTS* algorithm are utilized to find what move to make. *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.

The program will be implemented using parallelisation techniques to be able to run on different systems simultaneously .Eventually, the program will be executed on grid 5000, a cluster of multi core machines.

Contents

1 Presentation of our project

1.1 Generalities

Our project is called Fast and furious game playing, MonteCarlo drift. Our purpose is to create an Artificial Intelligence able to compete against humans using the MonteCarlo Tree Research.
We will only focus on two players games. Furthermore, we want to avoid games already resolved. We will choose something not studied entirely. We want to work on some new application. That is why we are interested by Arimaa.

For our game, we will need a program and statistics to make a good Artificial Intelligence. Each move should be calculated using a reliable method. MonteCarlo Tree Research is an algorithm able to take these optimal decisions. It has been used in the past for draughts, or chess. By exploring numerous possibilities, it will become possible to know what move is the better one. We will parallelize this algorithm in order to use it in a multi-core machine, to improve ~~his~~ efficiency. That MCTS algorithm is better than the classic Min-Max algorithm, that is why we will use it.

We will analyse parallelization methods, we will present it, and then we will choose the one adapted to our project. Thanks to the results of these latest methods, we will be able to choose a state resulting of the current move. Then we will explore the tree and with the same methods as before, we will figure out what the opponent will most probably do. The way we will be exploring the tree will only depend on the parallelization method. The first part of our project will be the analysis of latest thesis of technologies we will use, in order to choose the best one, and using it on the right environment, to improve ~~his~~ efficiency. In the next part, we will choose technologies we will need to achieve our goals, we will create a UML diagram to settle down our program.

Finally, in the last part, we will implement this program, and its documentation and test his executing on Grid5000, a cluster of multi-core machines. What is interesting in this project is we will create an Artificial Intelligence using technologies and methods fully optimized. Then we will create a program that can lead to true improvements for current algorithms applied to this game.

1.2 Presentation of Arimaa

Arimaa is played on a board composed of 64 tiles, like a chess board. Like in chess, there are 6 types of pieces, but they are not those chess players are used to. 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).

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

Figure 1: The different piece types in Arimaa.

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 2.

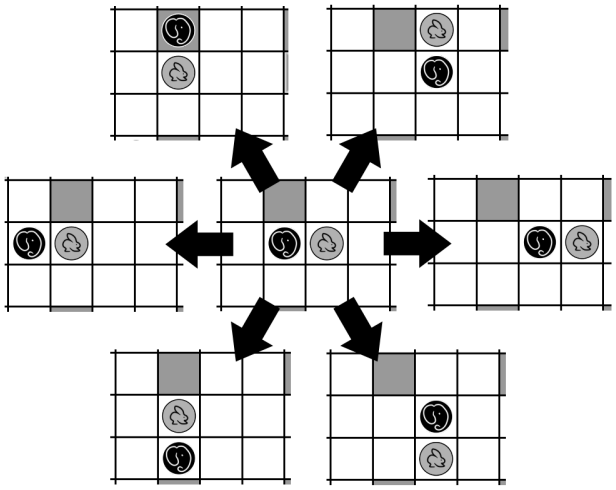


Figure 2: The different ways you can push or pull a weaker enemy piece.

A piece sitting next to a stronger enemy piece is frozen. When a piece is frozen, it cannot move. As shown in figure 3, a piece cannot be frozen while there is an ally piece beside it.

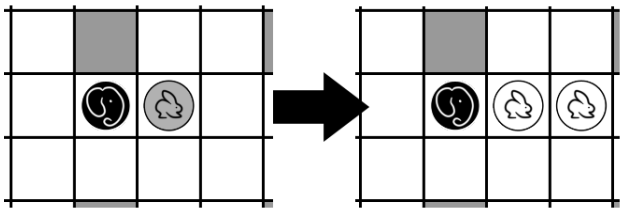


Figure 3: Example of the freezing mechanic.

There are four traps on the board. As shown in figure 4, any piece sitting on a trap with no ally piece next to it dies.

There are three ways to win the game :

Victory by reaching the goal You win the game if one of your rabbits reaches the other end of the board.

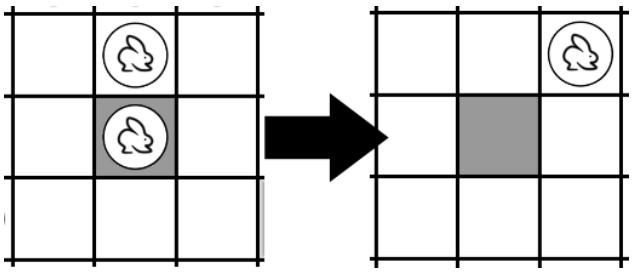


Figure 4: Example of the traps mechanic.

Victory by elimination You win the game if you eliminate all the rabbits belonging to the opponent.

Victory by elimination You win the game if the 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.

2 Algorithms

2.1 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 *your* moves and nodes representing your *opponent*'s moves.

Figure 5: Nodes representing your moves are generally drawn as squares, these are also called MAX nodes.

Figure 6: Nodes representing your opponent's moves are generally drawn as circles, these are also called MIN nodes.

The goal at 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 I make could make restrictions on what moves are possible for the other player, or possibly remove restrictions. Note also that in this example, we're ignoring what the game or the probem space are in order to focus on the algorithm.

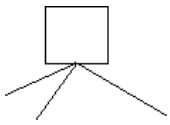


Figure 7: When we start the problem, all minimax sees is the start node.

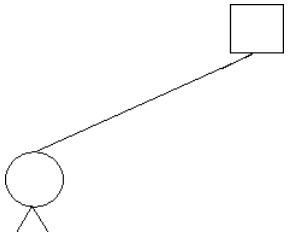


Figure 8: It begins like a depth first search, generating the first child.

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 you and a move by your opponent), then calculate evaluation values for states at that depth. For this example, we're going to go down one full move, which is one more level.

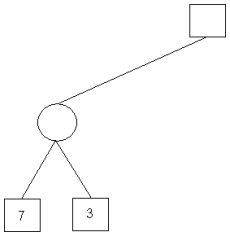


Figure 9: we generate the values for those nodes.

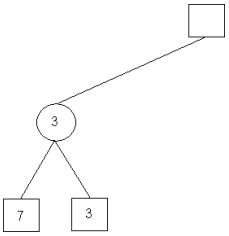


Figure 10: It chooses the minimum of the two child node values, which is 3.

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

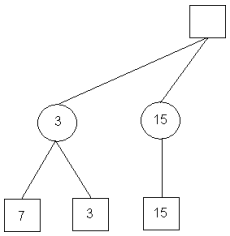


Figure 11: Since there is only one child, the min node must take it's value.

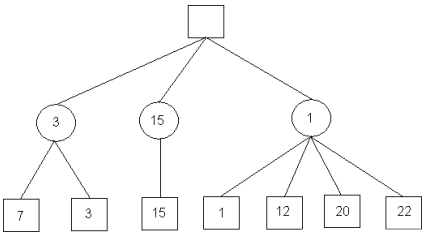


Figure 12: The third min node chooses the minimum of it's child node values, 1.

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.

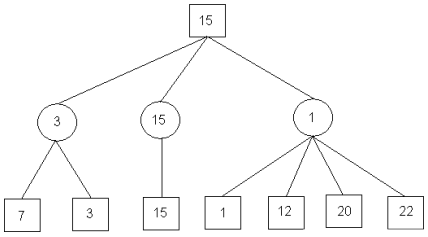


Figure 13: Final tree.

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.

2.2 The $\alpha\beta$ pruning

The $\alpha\beta$ method is a search algorithm 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.

2.3 Monte Carlo Tree Search Algorithm

2.3.1 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.

2.3.2 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.

2.3.3 Example

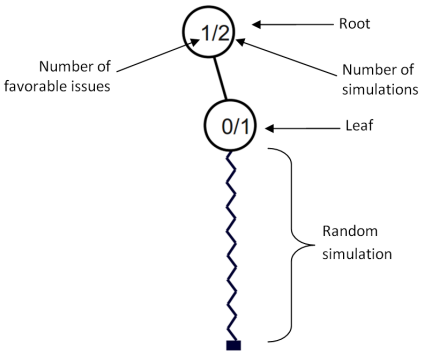


Figure 14: Legend of the following figures.

Figure 15: Run a first simulation from the root, get a favorable issue (will be considered as a *win*).

Figure 16: Create a first leaf at depth 1 and run the simulation, get an unfavorable issue (considered as a *loss*).

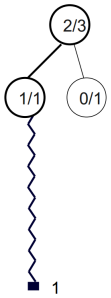


Figure 17: Create a second leaf at depth 1 and run the simulation (*win*).

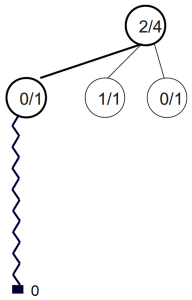


Figure 18: Create a third leaf at depth 1 and run the simulation (*loss*).

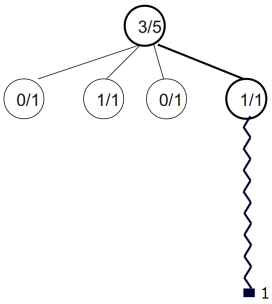


Figure 19: Create a fourth leaf at depth 1 and run the simulation (*win*).

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).

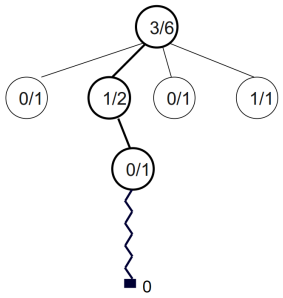


Figure 20: Create a leaf at depth 2 with parent the 2nd leaf at depth 1 and run the simulation (*loss*), update the odds value of the node and making it less interesting than the fourth node. Therefore the algorithm will now work on the fourth node.

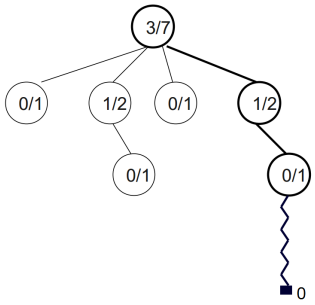


Figure 21: Create a leaf at depth 2 with parent the fourth leaf at depth 1 and run the simulation (*loss*), update the odds value of the node and making it as interesting as the second node. The algorithm will now work on the second node.

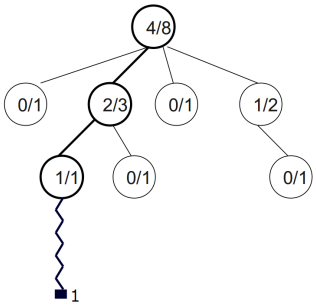


Figure 22: Create a second leaf at depth 2 with parent the second leaf at depth 1 and run simulation (*win*), update the odds value and continue to develop this leaf.

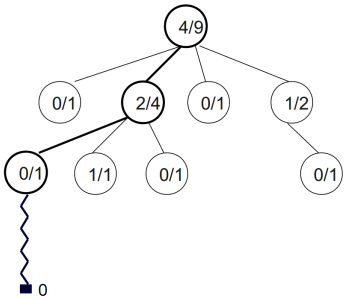
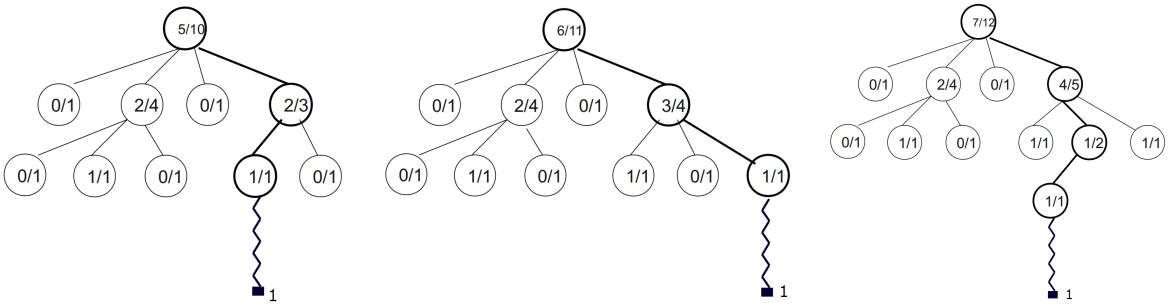


Figure 23: Create a third leaf at depth 2 with parent the second leaf at depth 1 and run simulation (*loss*), update the odds value and switch to the fourth leaf.

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.

2.3.4 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

- 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.

2.3.5 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.

2.3.6 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.

3 Strategies and state of the art

3.1 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.

3.2 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 advantage of this method is that its implementation seems very simple. There are no problems of mutual exclusions, or mutexes, between the threads. However, there are two major problems. The first one is that 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. The second problem is that there's 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.

3.3 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.

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. Since there is no communication, multiple threads can develop the same sub-tree. On the other hand, the lack of communication increases drastically the speed of the program. Each thread executes a different tree and as they wait for the end of the time, the program does not lose time in communication between threads. Actually, the strength of this strategy lies in the little communication.

3.4 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[REFERENCE ?]. The first one is to put mutexes on the tree and the second is to implement a virtual loss.

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 [1], 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.

3.5 Comparison

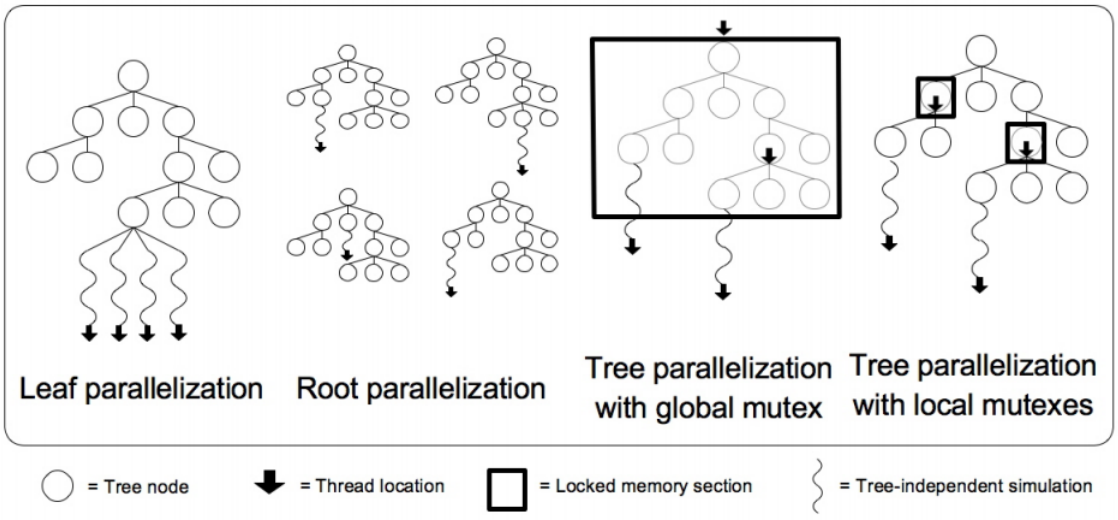


Figure 24: Comparison of all trees

Currently, the best strategy to adopt is the Root parallelization. According to some tests [2], the advantages of the Root parallelization ~~outcomes~~ its drawbacks. Indeed, for 16 threads, a program using Root Parallelization will win 56

3.6 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. There is, for instance, the UCT-Treesplit algorithm which retake the base of Root algorithm and add to it work clusters on which compute nodes can process. 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.

3.7 Conclusion

In our case, we will also have those two types of parallelization to compute so we can use this type of algorithm. We can also choose a simple Root parallelization between the different computers, and them, use another algorithm, specialized in shared memory parallelization, within them.

3.8 State of the art of MCTS

3.8.1 History of computers vs Humans

In 1997, Deep Blue a computer built by IBM won a six games match against the current chess world champion Garry Kasparov. Humans got beaten on Chess, but remain undefeated on Go, therefore the research has switched to that game. 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.

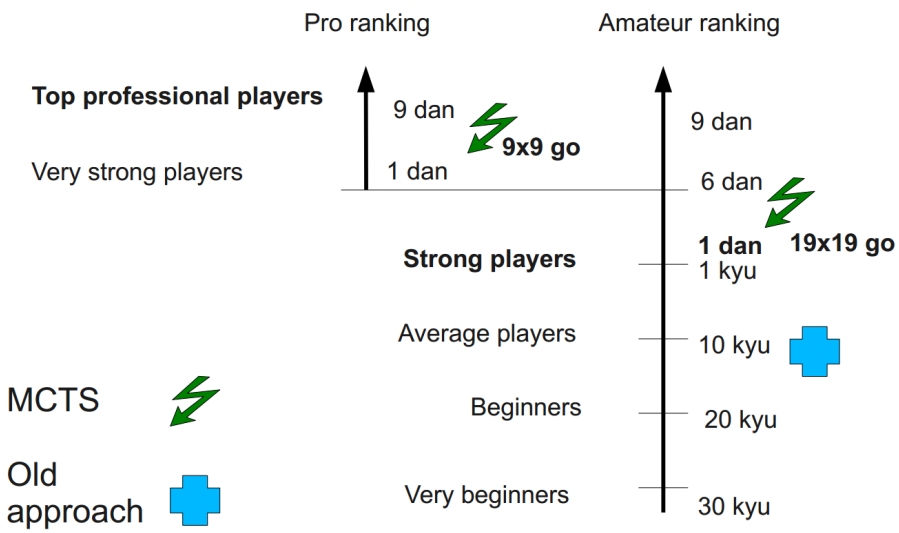


Figure 25: Comparison between Go algorithms and human skill (*2011*).

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. For comparison, strongest humans players are rated around 2450 elo. (source : Method of MCTS and the Game Arimaa, Tomas Kozelek, 2009, Master's Thesis).

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 200 elo and International Masters over 2400 elo.

4 Solutions and schedule

4.1 Candidates software and technologies

4.1.1 Language

Quickly, at the beginning of the project, we choose to use C++ to code our software. This is for simple reasons, C++ is a fast language so it's more indicate than Java to create an efficient algorithm. 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.

4.1.2 Software

We fixed the choice of the software solutions we will use to develop the project. The coding will be realized on Microsoft visual studio 2013, and the version manager will be git. This two software are very used in the industry so we have to be familiar with them. For the bibliography we will use Zotero, it permit the creation of .bib the bibliographic format use by LaTeX that we will used to write our reports.

4.1.3 Grid5000

Grid5000 is a cluster of computers of the IRISA, it links a lot of computer of different centers of research. Some of ~~this~~ machines have, in addition ~~of~~ a good CPU, a NVIDIA Tesla GPU that could be use to parallelize the MCTS. Of Course use this cluster will require some specific parallelization that we will present in the next part.

4.1.4 Parallelization

OpenMp

OpenMp is a very simple and efficient parallelization method. It consists in some pre-compilation instructions, with only a few lines it permits to obtains a parallel version of our algorithm. But it isn't exempt of default, in fact, as you can see on the next illustration, it use a large part of shared memory, it allows a quick and efficient communication between the different threads but it's not efficient for using a cluster of computers, because the memory access will be too long if this memory is used by two computers which are far away. But like we will see after, there is a method to avoid this problem.

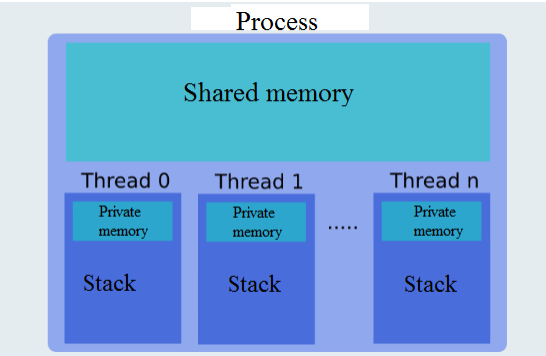


Figure 26: OpenMp : memory management

MPI

MPI is a more complex method of parallelization than openMp but it's more powerful. As you can see on the illustration, it doesn't use shared memory, all the data of the threads are duplicate 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 their isn't the problem of time to access the memory. But we have to pay attention to the cost of communication between the threads, if we use too much the signals we will lose all the time we can gain with the parallelization. It's very adapted for a tree parallelization, because with it we have just to duplicate the data at the beginning and return the result at the end of the assign time.

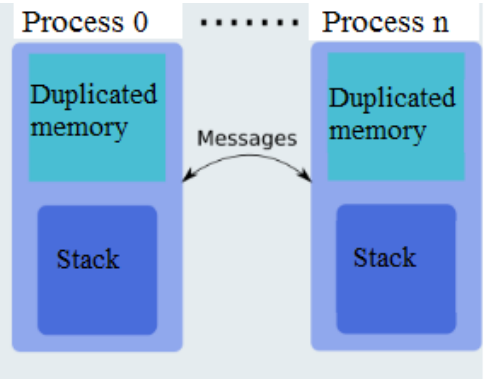


Figure 27: MPI : memory management

Hybrid parallelization

As we see openMp and MPI have each one their advantage and inconvenience, but we can reduce the problems by using both. That means that we can use MPI to dispatch the work onto the different computers and after use openMp to divide it on the different threads of each machine. This can permit to use together multiples parallelization strategy like tree between the machine and leaf or root on each machine. Also it reduce the manual treatment

that we have to define in the code to manage the threads. It means too that we reduce the threads interaction, so we kept a maximum of the parallel code.

Boost Library

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

GPU implementation

During our research we heard about using the GPU to improve the MCTS algorithms efficiency, as Kamil Rocki said in his thesis "Large Scale Monte Carlo Tree Search on GPU" one GPU's performance is equivalent to 50 CPU threads. But this implementation has some defaults, in fact the Gpu possess very few 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 2 branches a GPU will develop hundreds branches simultaneously. Another thing we have to know is that a GPU can switch to another thread immediately without any cost of context switching. Grid5000 has NVIDIA GPU so one of our possibility is to use hybrid as I describe 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 we should be able to develop greats trees and have a better solution compare with using only CPU, even if the trees will be less deep.

4.2 Planning management

One of the purposes of this project is to learn how manage and plan our work. For this we have to use MS Project to indicate all the tasks we have to do, who does each one, and how long it will be. So how did we chose to do this ?

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 differents reports (analysis, specifications, ..), the orals and the rendering of the code of the project. After this, each week we decide all tasks we will do for next week, then we add them on the planning and associated each one with the personn who will do it.

For the moment we don't plan the coding part because until we will have chose all the technologies we will use, there are many tasks that we can't estimate the length. The only one we do 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 role distribution, 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, it's him who sets the deadlines of every tasks, and who check that everything is doing in time.

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.

5 Conclusion

Finally, our project is about creating an Artificial Intelligence able to beat an human playing the game *Arimaa*.

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 Search Algorithm* to take the best decisions in our game. We will decide what move to play according to its results. Because of the numerous moves possible, we will need to develop the better ones only, using parallelisation on different systems. We already analyzed the state of the art of *Arimaa* and the *MonteCarlo Tree Search*. Consequently, we will base our work on these theses, to make it possible to use the best technologies with the best environment without doing again what has already been done. We already know how to play this game, making it easier to create strategies for our Artificial Intelligence. We handle a task's schedule to control our work time ofr every aspect of the project, the point being mostly to learn how to schedule a project, foresee surprising events, and answer the demand.

The point of this project is also to test our program with Grid5000, a powerful network of multi-core machines. Then, we will be alble to use the full potential of our program in its maximal capacities, against a human.

References

[1] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. The boost graph library - 1.56.0.

[2] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. The boost graph library - 1.56.0.