Fast and Furious Game Playing: Monte Carlo Drift

Design report

Prateek Bhatnagar, Gabriel Prevosto,
Benoît Viguier

Supervisors: Nikolaos Parlavantzas, Christian Raymond

02/02/2015

Contents

1 Introduction

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* algorithm.

We have chosen the game Arimaa because it is a two-players strategy board game not solved[1].

A human plays a game by thinking of all possible moves as per one's imagination and then opts for the best amongst them. Before taking his/her turn, a player can visualize the options and predict how an opponent will counteract them. The algorithm will do the same by building a search tree containing the different posibilities. The Minimax algorithm does it by exploring all possibilities, which is heavy. The MCTS algorithm is lighter and converges to the Minimax algorithm, therefore it has been chosen for this project.

This algorithm will be parallelized in order to optimize it in a set of multi-core machines, allowing it to go further into the search tree, thus improving its efficiency.

In this report, the data structures that will be used for the development of the project are discussed. A parallelization strategy will be finalised from the various strategies that have been discussed till now by referring the test results. Various parallelisation frameworks viz. OpenMP and MPI will be tested and the most optimal framework for the project will be chosen for implementation of the project. A study for the GUI of the project is done.The design of the software that provides the GUI is done using UML diagrams including the detailed explanation about the various interfaces that will be used.

[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.

2 Base algorithm

The base algorithm is an implementation of the MCTS algorithm. All the functions related to it are included in the mcts namespace. As a parameter, it takes the abstract class *TheGame*, the position (*Bitboard*) to start the search with and an object (*MctsArg*) to set parameters to the MCTS algorithm: for example, the time to search, the depth of the tree; the number of simulations to run; and the number of nodes to create in the tree. The following Class Diagram1 represents the links between the different kinds of objects.

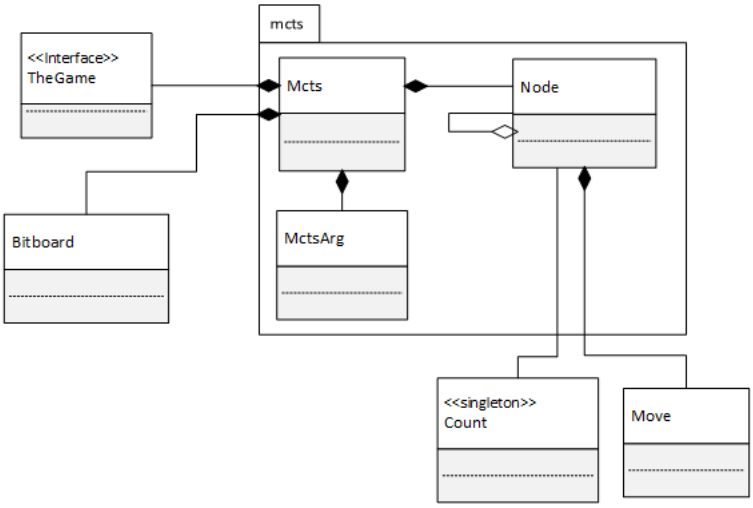


Figure 1: *Class Diagram of the mcts namespace*

The class Node represents the tree, it is stored as an array in the mcts object. A detailed implementation is provided in the Data Structure part3.

The singleton *Count* is used for the statistics (such as the number of leaves, nodes created, and the number of run). Whilst providing statistics it also makes sure that no memory leaks are happening by counting creations and destructions of the main objects.

The figure2 below represents the order of calls for the main function and provides some details about the implementation.

```
Move GetBestMove() {
       explore() {
          While {                                          // loop simulations
              UpdateNode(node);                            // expand the _root
              While {                                      // explore the tree
                     node = select_child_UCT();            // select the node with UCT
                     UpdateNode(node);                     // expand the node
              }
              If node non terminal
                     result = playRandom(node);            // run random simulations
                            update(result);                // feedback the results
              elseif winning move
                     node->forceSetUCT(10);
                     feedbackWinningMove(node);
                            updateLosingParent(node);
                     update(result);
              else
                     update(result);
          }
        }
        _root->select_child_WR()->getMove();
        // chose the child with highest the winrate and return its move
      }
```

Figure 2: *Overview of the implementation of the function GetBestMove()*

explore: start the exploration of the tree.

UpdateNode: make sure the node has children and update its terminal value if required.

select_child_UCT: go through all the children of a node and return the one with the highest result at the UCT function (refer to the *pre-analysis report, part 3.4.4*).

playRandom: start to run the random simulations and return the winner/tie.

update: update the node statistics and feed the result back to its parents.

forceSetUCT: In the event of a winning move, the uct value is set to 42 in order to make sure that each and every further exploration goes through that node, (all the values calculated by the UCT function are below 42).

feedbackWinningMove: feed the winning move back to its parent: one does not want to go to that position because it would be a winning strategy for the opponent.

updateLosingParent: in the event of all the siblings being a losing move, update its UCT value to winning strategy in order to propagate the results.

select_child_WR: return the child with the highest win rate.

3 Data structure

Reading only 2 moves ahead on Arimaa requires at least $20000^2 = 4 \times 10^8$ positions to be explored. If the size of a node is 100 octets, in order to store such a tree, we would require $4 \times 10^{10} octets \approx 37,3 Gio$. Given the size of the data our software will be working on, it requires an efficient way of storing them. There are 2 kinds of data, on the one hand the ones the program works on, and on the other hand, the parameters and utilities.

3.1 Representation of the playouts

3.1.1 Nodes

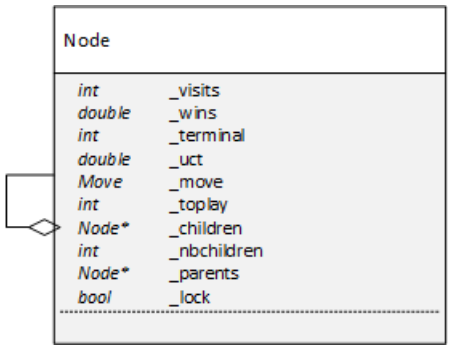


Figure 3: *Details of the data cointained in a node.*

Nodes contains statistics on the previous results; a pointer to their parent; a pointer to the first of their children; and the number of children they own. They are stored in an array (*_tree*) set at the begining of the program, thus they are grouped on a countinuous memory segment. Therefore the time to gain access to them is decreased. We also properly aligned the data structures inside the nodes using *g++ -Wpadded*, in order to minimize its size.

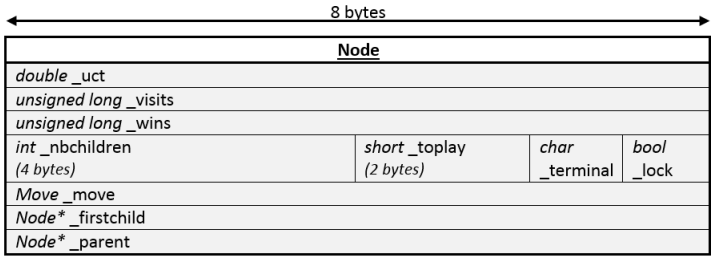


Figure 4: *Optimized node structure.*

3.1.2 Pruning

Once a move has been played, only the sub-tree of its node is kept. The others have to be discarded in order to retrieve some memory. The process of removing these branches is called pruning.

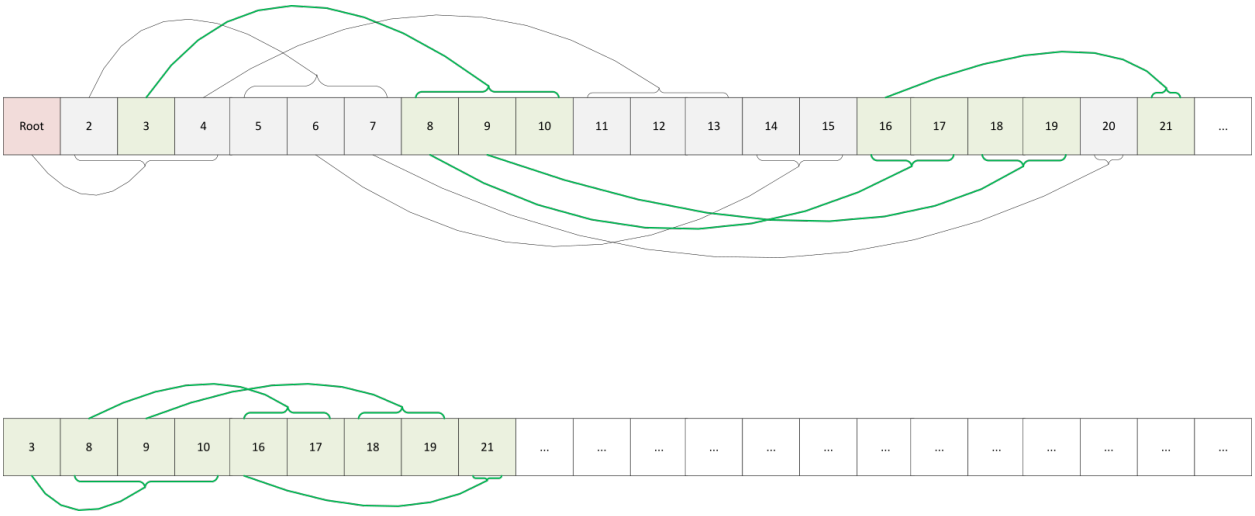


Figure 5: *Prunning of the tree (before and after).*

In order to prune the tree, we use the following method: transform a copy of the current tree (*_tree*) into a buffer (*_buff*). The root of the buffer tree will be a copy of the node. Then the children are saved proceeding down through the branches. The advantage of this method is that you only copy the nodes you want to keep. However, the memory used by the buffer needs to be the same as that of the tree before the prunning. Thus, the maximum memory that can be used by the tree (*_tree*) is half the memory used by the program. In order to dermine the number of leaves to be created, the program checks how much memory there is left on the computer, and uses a fixed percentage of it.

$$N = \frac{R \times 90\%}{2} \tag{1}$$

N = number of leaves.
R = RAM left to use.

We chose to limit the memory used by the tree to 90% of the available memory in order not to overload the RAM and to leave some for other operations such as simulations. It also allows us to make sure that the swap will not be used to store the tree because it impacts heavily on the speed of its exploration capacities.

3.1.3 Bitboards

Boards are stored as bitboards. As the game is played on a 8×8 board, it is convenient to use a 64 bit integer to store the position of the pieces. That way, each and every kind of piece is stored on the same x64 integer, saving space as opposed to a matrix 8×8 retaining all the information. Players own rabbits, cats, dogs, horses, a camel and an elephant; thus using 6 integers for each player is the least . Adding a bitboard to store the position of every piece of each player helps to increase the speed of the algorithm by reducing the number of tests required, according to the playout phases. It also allows quick tests and modifications such as bit twiddling.

3.1.4 Game abstraction

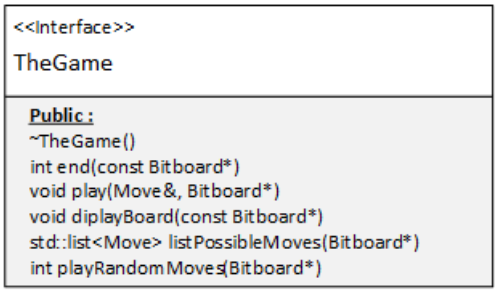


Figure 6: *Details of methods cointained in the interface TheGame.*

In order to know the moves that can be played and the winning conditions, the mcts algorithm has to have access to a class describing the game. This abstract class is implemented and passed as a parameter at the instantiation of the mcts object. It allows genericity in the algorithm and permits its use on any board games such as Connect4 or Arimaa. This interface has 4 main methods :

end: check if the board provided is a final state.

play: play a given move on a given board.

listPossibleMoves: list all the possible moves on any given a board: this is used when a node is to be expanded.

playRandomMoves: play random moves until a final state is reached and a winner is returned. This is the random simulation part.

3.2 Parameters and utilities

3.2.1 Parameters

Instead of directly passing all the parameters to the mcts object at its instantiation, we decided to create an object that would provide them using appropriate inline getters. The point of this is to allow quick modification on the parameters without having to rewrite some parts of the mcts to make sure that each file is updated. The main idea applied here is to separate the data from the algorithm using the same principle as the parameter object pattern.

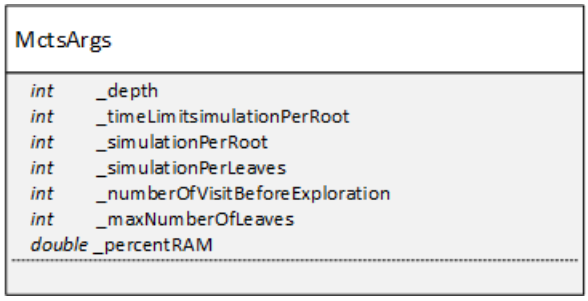


Figure 7: *Details of the parameters of the algorithm.*

3.2.2 Fast log

The MCTS algorithm does not require an exact value of the $ln(x)$ function in order to calculate the UCT value of a node. As binary numbers are stored on computers, it is more interesting to get their log in base 2 and to multiply it by $ln(2)$.

$$log2(x) = \frac{ln(x)}{ln(2)} \tag{2}$$

$$ln(x) = log2(x) \times ln(2) \tag{3}$$

$$ln(x) = log2(x) \times 0.69314718f \tag{4}$$

Depending on the main operating system (Windows or Linux), the calculus of $log2(x)$ will differ. On linux, a quadratic approximation is made. For further details, refer to annex 7.3. On windows, *_BitScanReverse64(&y, x)* is slightly faster.

3.2.3 Random numbers: Mersenne Twister

Given the number of playouts to be simulated, the MCTS algorithm requires a fast random number generator. The Mersenne Twister which is implemented in the STL is faster than the basic rand() function. Therefore we decided to use it.

4 Parallelization

4.1 On clusters

The parallelization strategy we have chosen to use on clusters is *Root parallelization*. *Root parallelization* consists in giving the tree that we want to develop to every actor (here each machine is an actor), letting them develop it randomly without any communication with the environment during a certain amount of time, and then merging the results of each tree. It is depicted in figure 8. This method has the great benefit of minimizing the communication between the machines, as they only communicate at the beginning and at the end of the algorithm, without needing any further synchronization.

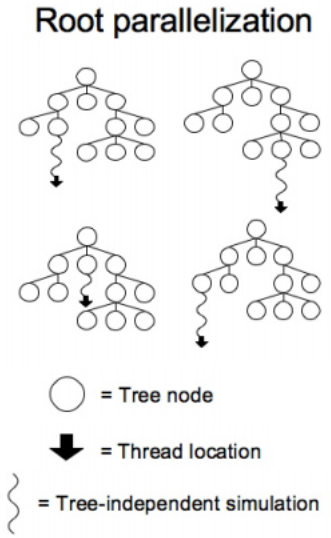


Figure 8: Overview of *Root Parallelization* [2]

In order to apply this strategy, we have chosen a master-slave approach, with one master machine collecting the results of every other machine once they are done with their processing. The algorithm will proceed as follows :

- The master will asynchronously send the state of the game to the slaves
- The slaves will get the state of the game sent by the master with blocking receive requests
- Every machine will run its simulations by itself, building its own tree
- At regular intervals, each slave will asynchronously send messages with the current state of its tree
- The master will then receive these messages with asynchronous receive requests, and update its own tree with their results

- At the end of the given time, the master will process the best move with the help of its tree

The *Message Passing Interface Standard* (*MPI*) is a message passing library standard. We plan to use it for machine parallelization if the tests give satisfying results. In order to make the different machines communicate, it opens ssh connexions between them. The general structure of a MPI program can be seen in figure 9.

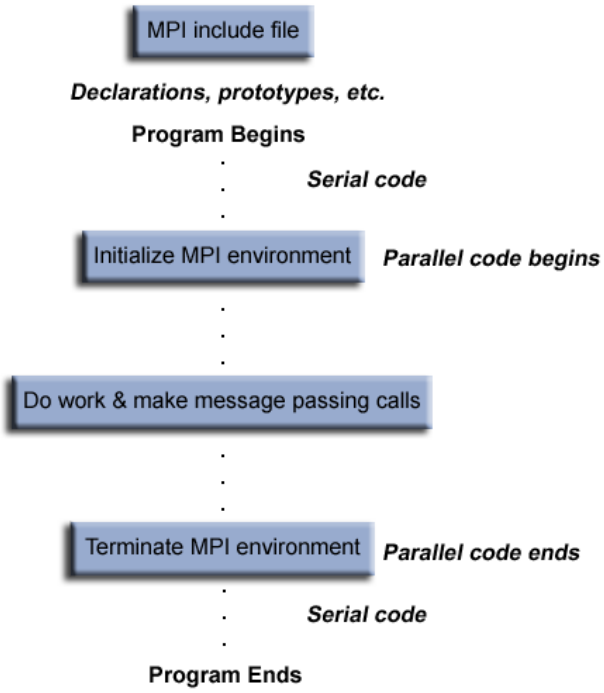


Figure 9: General MPI Program Structure. [1]

MPI uses objects called communicators and groups to define which collection of processes may communicate with each other. Within a communicator, every process has its own unique, integer identifier assigned by the system when the process initializes. These ranks are used by the programmer to specify the source and destination of messages.

MPI allows for synchronized, blocking and non blocking routines. Synchronized sending requests only return after the message has been received, while blocking ones return after it is safe to modify the data in the sending buffer, and non blocking ones return immediately after sending the order, but offer no guarantee that it has already been executed.

Another type of routines handled by MPI is Collective Communication Routines. They allow to make a similar operation on every process of a given communicator. For example, it is possible to spread data from a single process to each other process, or on the contrary to regroup data from every process into a single one (as represented in figure 10), where all the data is gathered in task 1). This will be especially useful for the algorithm, as it will require to regroup the results of every process.

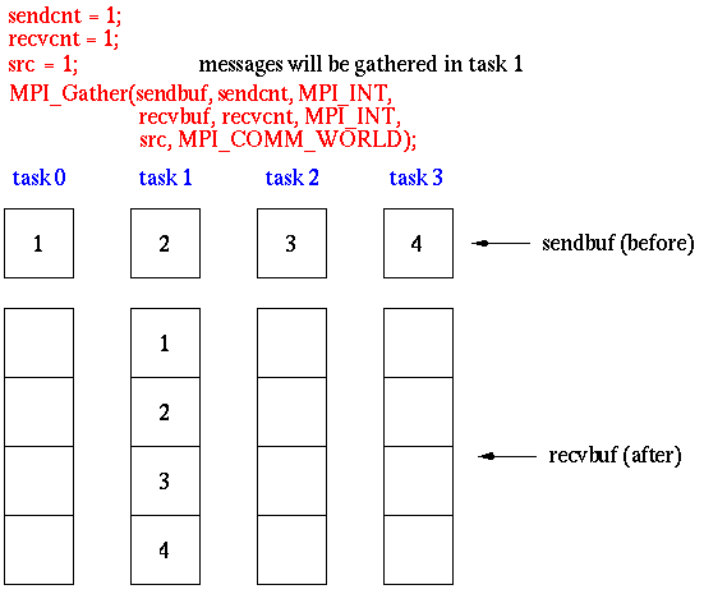


Figure 10: Example of a Collective Communication Routine ([1]). Here we can see task 1 gathering the data from every other task.

4.2 On computers

Given the size of the tree to be explored (at least 4×10^8 nodes, see 3), a root parallelisation strategy within a machine can not be applied. A 11-million-nodes tree requires about 2 Gio of RAM. Should it be timed by 40 ($11 \times 10^6 \times 40 \approx 4 \times 10^8$), it would requires 80 Gio of RAM. Multiple threads, each working on seperate trees, would requires much more memory than there is available.

With this in mind, a tree parallelisation strategy has to be implemented on the computer. In this method, multiple threads share the same tree 11, the exploration is done the same way as it is with one thread. The main problem is that multiple threads can access the same node and corrupt data given by another thread. To avoid such problems, we will be using *virtual loss* and *local mutexes*.

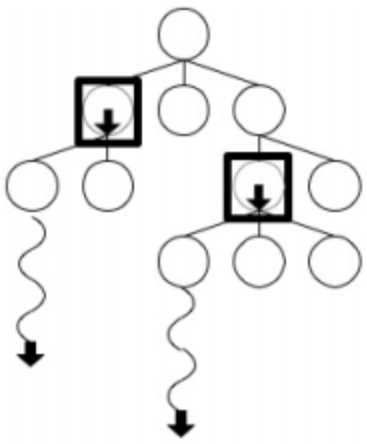
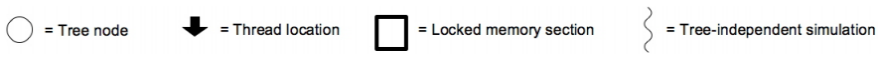


Figure 11: *Tree parallelization*



Should a thread go through a node, it automaticaly increases the value of its visit counter. This will virtually decrease the winning rate of the node and therefore decrease its attractiveness.

Each node also has a boolean to serve as a lock in order to prevent concurency. Its access is done in a critical section (atomic capture). Should a thread access a locked node, it will stop exploring and start its simulations.

This method of parallelization can be done by multithreading the *While // loop simulations* described in the base algorithm implementation 12. Using OpenMP, this is easily implented with a single #pragma statement:
#pragma omp parallel shared(i,timeend)
where *i* (the number of simulation run), and *timeend* (the time until simulations are to be run), are defined as shared variables. That way, we make sure that the time limit and total number of simulations are not overtaken.

In order to prevent any conflict between threads, a small critical section has to control the lock of the nodes. Placing the following statement before *node->getLock()* achieves this:
#pragma omp critical

```
explore() {
  #pragma omp parallel                          // start multithreading
  |   While {                                   // loop simulations
  |       UpdateNode(node);                     // expand the _root
  |       While {                               // explore the tree
  |           node = select_child_UCT();        // select the node with UCT
  |
  |           #pragma omp critical               // start critical region
  |           |   lock = node->getLock();       // test and acquire lock
  |
  |           if( lock ) break;                 // get out and start simulations
  |
  |           UpdateNode(node);                 // expand the node (if not locked)
  |           node->releaseLock();              // release lock
  |       }
  |       result = playRandom(node)
  |       […]
  |       update(result);
  |   }
```

Figure 12: *Overview of the implementation of the tree parallelization using OpenMP*

5 The Graphical User Interface

5.1 Overview of the UI

A user interface (*UI*) will help visualizing the game. It will be coded in *C++* and will use the graphic library *SFML 1.6*. Its functionalities will include the following:

- 1 Versus 1, player versus AI and AI versus AI
- Mouse and keyboard controls
- Configurable controls and resolution
- Assisted unit placement
- Game saving and loading
- Game replay
- Fullscreen mode
- Graphical mods

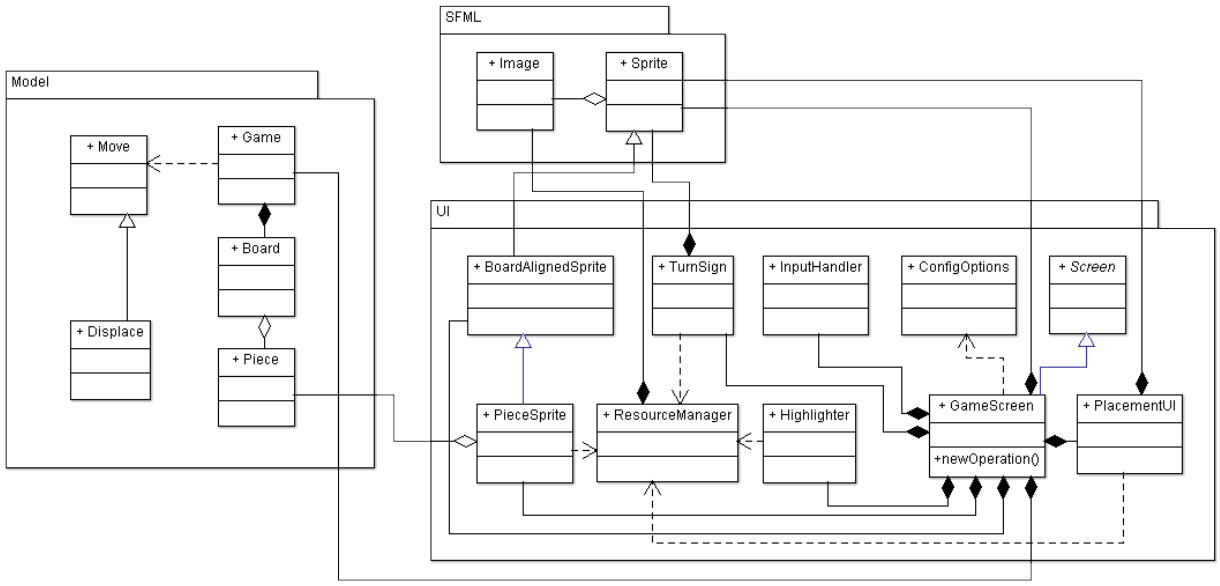


Figure 13: A UML diagram for the UI.

A UML diagram for the UI is presented in figure 13. In this diagram, you can see three modules:

- The model, regrouping the rules of Arimaa
- The UI itself, handling the inputs and outputs, and interfacing with the Model

- The SFML, a library that loads and displays the assets (such as text and sprites[2])

5.2 The model

The model handles the rules of the game. A game is modelized by the class *Game*. This class contains a *Board*, itself containing instances of *Piece*. While the *Board* allows any and all modifications to the pieces' placement, the *Game* only allows actions that follow the rules. This is done through the use of the *Move* class, representing a move in the game. Any instance of the *Move* class can be verified by the *Game*, and executed if found valid. The class *Displace* inherits from *Move*, and modelizes pushes and pulls.

5.3 The UI

The core of the UI lies in the class *Screen*. It modelizes a screen that can be displayed and updated at each frame. The *GameScreen* is the main screen of this UI: it displays the game as it is played. Among the other important classes are *ResourceManager*, that uses the *Flyweight* design pattern to manage the assets; then there is *InputHandler*, that associates keyboard inputs with strings representing the different actions, and also *ConfigOptions*, that loads the options contained in a .ini file and relays them to the application. Most of the other classes represent the different graphical elements (pieces, the cursor...) and will not be explained in detail here.

[2]A computer graphic which may be moved on-screen and otherwise manipulated as a single entity.

6 Conclusion

The focus of this report is related to designing of the software application. The data structures that are used for the development of the MCTS algorithm application are finalised such as bit board and nodes. An attempt has been made to derive the association between the data structures, interfaces and MCTS using the class diagram.

The parallelization strategy that has been chosen is root parallelisation with MPI framework for clusters. The tree parallelization strategy has been chosen for implementation on the machine(computer) with OpenMP framework. The testing of both the strategies has been done successfully.

A graphical user interface is designed and is demonstrated with the help of UML diagram and is divided into three parts as the user interface, the model and the SFML.

The development of this report will serve as a base support for the concrete development of the project.

7 Annexes

7.1 MCTS Class Diagram

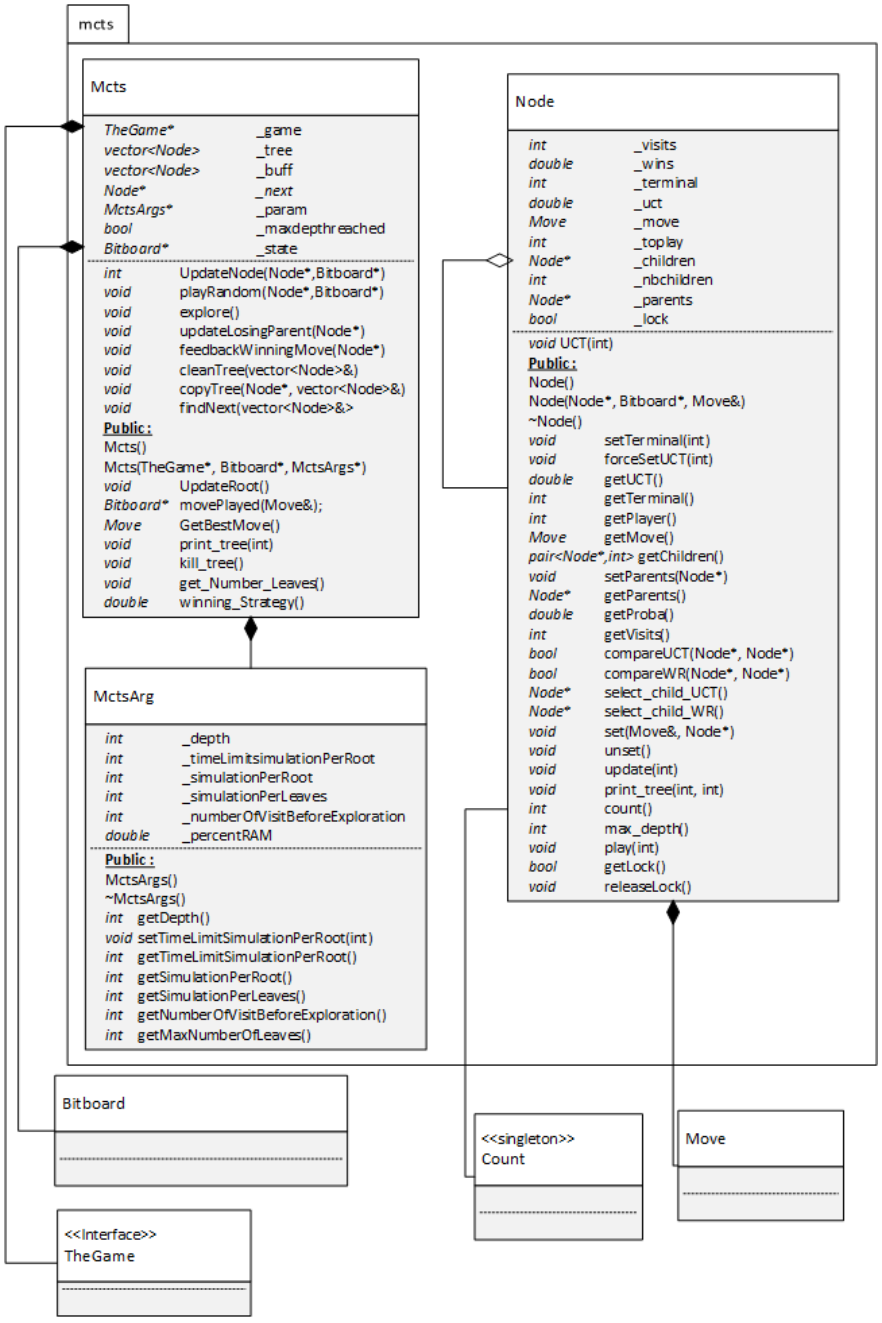


Figure 14: *Details of the mcts namespace.*

7.2 Data Structure Class Diagram

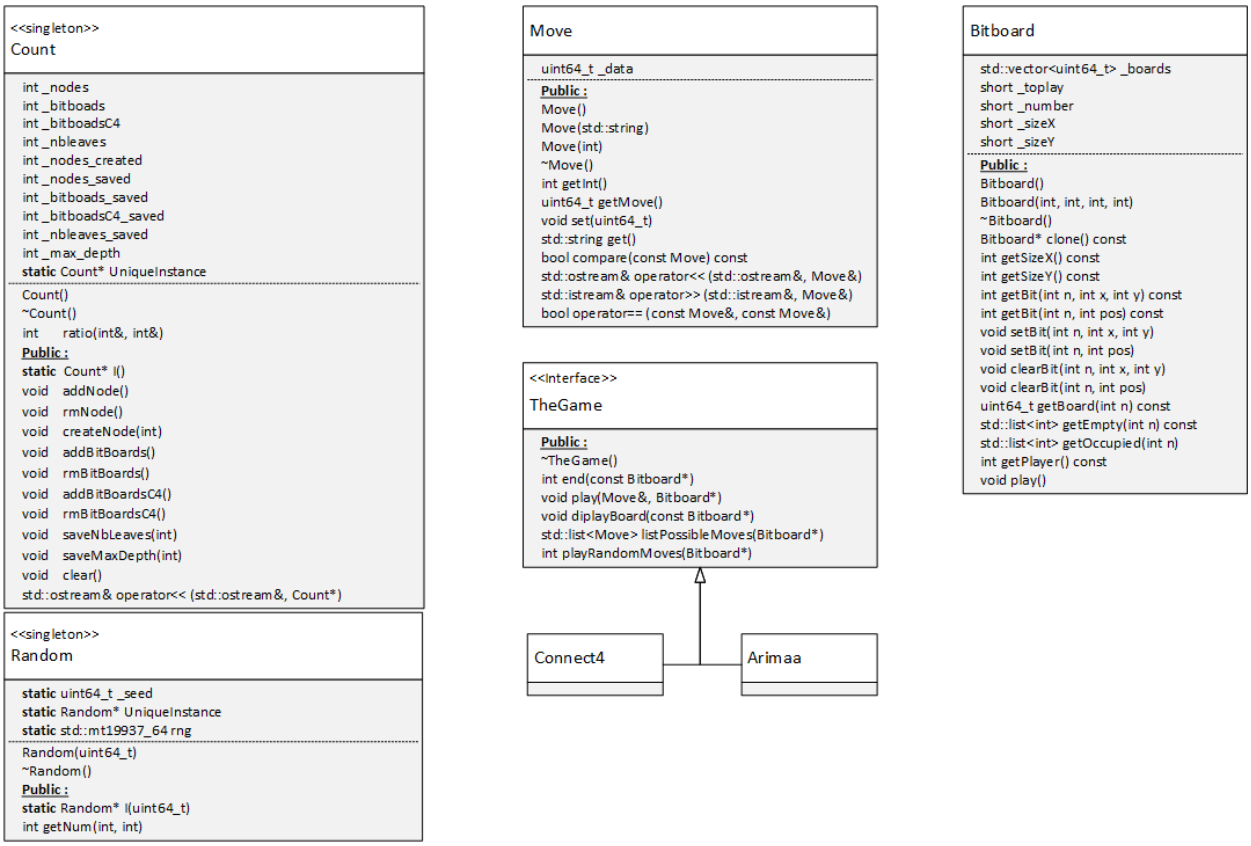


Figure 15: *Details of the DataStructure class diagram.*

7.3 Linux $log2(x)$ Implementation [3]

```
1  static inline float log2(float val)
2  {
3    int* const  exp_ptr = reinterpret_cast <int *> (&val);
4    int         x = *exp_ptr;
5    const int   log_2 = ((x >> 23) & 255) - 128;
6    x &= ~(255 << 23);
7    x += 127 << 23;
8    *exp_ptr = x;
9    val = ((-1.0f / 3) * val + 2) * val - 2.0f / 3;
10   /*
11   The line computes 1+log2(m), m ranging from 1 to 2.
12   The proposed formula is a 3rd degree polynomial keeping first derivate
         continuity.
13   Higher degree could be used for more accuracy.
14   */
15   return (val + log_2);
16 }
```

References

[1] Lawrence Livermore Blaise Barney. Message passing interface (mpi). `https://computing.llnl.gov/tutorials/mpi/`.

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

[3] Laurent de Soras. Fast log2(float x) implementation c++. `http://stackoverflow.com/questions/9411823/fast-log2float-x-implementation-c#answer-10702739`.