Fast and Furious Game Playing: Monte Carlo Drift

Conception 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 ones turn, a player can visualize ones 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, MPI and CAF 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 with the help of 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 a simple implementation of the MCTS algorithm. All the functions related to it are included in the mcts namespace. It take as a parameter the abstract class *TheGame*, the position (*Bitboard*) to start the search with and an object (*MctsArg*) to set the parameter to the MCTS algorithm such as the time to search on, the depth of the tree, the number of simulation to run and the number of nodes to create in the tree. The following Class Diagram1 represent the links between the differents kinds of objects.

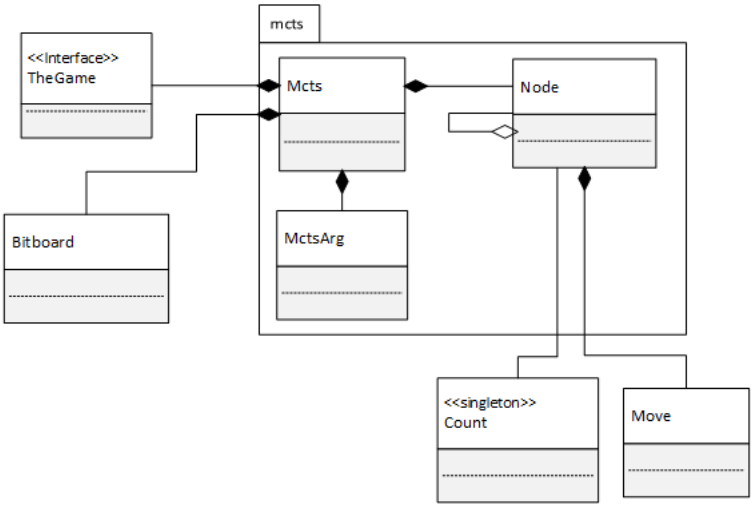

Figure 1: *Class Diagram of the mcts namespace*

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

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

The following figure2 represent the order of the call for the main function while providing 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 tree exploration.

UpdateNode : make the 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 feedback the result to its parents.

forceSetUCT : In the event of a winning move, the uct value is set to 10 in order to make sure that each and every further explorations go through that node.

feedbackWinningMove : feedback the winning move 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 children of the parent 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

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 first hand the ones the programs works on; on the other hand, the parameters and utilities.

3.1 Boards and nodes

3.1.1 Bitboards

Boards are stored as bitboard. As the game is played on a 8×8 board, it is convinient to use a 64 bit integer to store the positions of the pieces. That way, each and every of a kind of pieces are stored on the same x64 integer, saving space as opposed to a matrix 8×8 retaining all the informations. Players own rabits, cats, dogs, horses, camel and elephant; thus using 6 integer for each player do the job. Adding an additionnal bitboard to store the position of every pieces of each players helps to increase the speed of the algorithm by reducing the number of test required to be done during the playout phases. It also allows quick tests and modifications such as bittwingling given the nature of the data.

3.1.2 Nodes

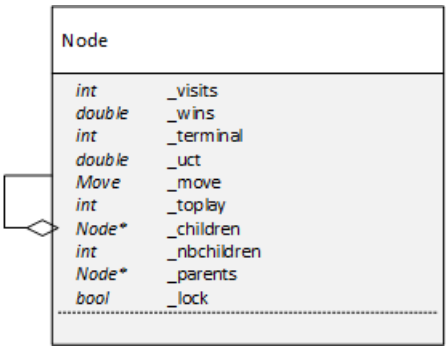

Figure 3: *Details of the data cointained in a node.*

Nodes contains statistics about 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 grouping them on a countinuous memory segment. Therefore the time to gain access to them is decreased.

3.1.3 Prunning

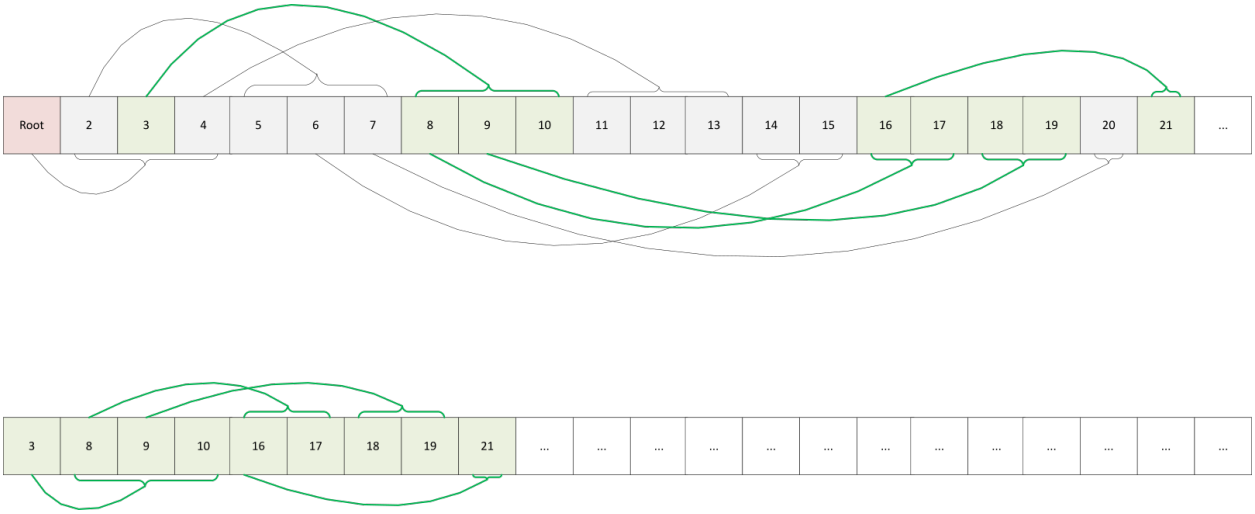

Figure 4: *Prunning of the tree (before and after).*

In order to prune the tree, we use the following method : create a copy of the current tree (*_tree*) into a buffer (*_buff*). The root of the buffer tree will be a copy the chosen node. Then the children are saved going dpwn through the branches. The advantage of this method is that you only copy the nodes you want to keep. However the memory used of the buffer need to be the same as the one 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 use 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 aviable memory in order to not overload the RAM and to let some left for other operations such as simulation. It also allow us to make sure that the swap won't be used to store the tree as it heavily impact on the speed of its exploration.

3.2 parameters and Utilities

3.2.1 Parameters

Instead of directly passing all the parameters to the mcts object at its intentiation, we decided to create an object that would provide them given 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 files are updated. The main idea applied here is to separate the data from the algorithm with the same principle as the MVC design pattern.

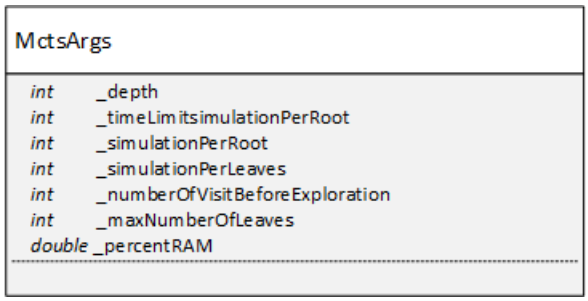

Figure 5: *Details of the parameters of the algorithm.*

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 numbers are stored in binary on computers, it is more interesting to get the value of their log in base 2 and to divide it by $ln(2)$.

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

$$ln(x) = log2(x) \times \frac{1}{ln(2)} \tag{3}$$

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

Depending on the main operating system (Windows or Linux), the calculus of $log2(x)$ will differ. On linux, a quadratic approximation is made, for more details, refer to annexes7.3. On windows, *_BitScanReverse64(&y, x)* is slightly faster.

3.2.3 Random numbers : Mercene Twister

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

4 Parallelisation

4.1 Strategy

The parallelization strategy we have chosen for this project is *Root-Root parallelization*. *Root parallelization* consists in giving the tree to develop to ev thread, 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. This method has the great benefit of minimizing the communication between the actors (the threeads or the machines). That is because they only communicate at the beginning and at the end of the algorithm, without needing any further synchronization. The *Root Parallelization* is depicted in figure 6.

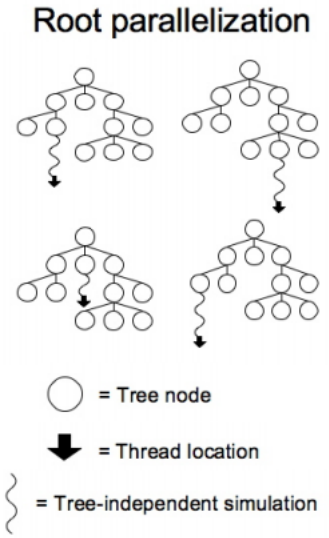

Figure 6: Overview of *Root Parallelization* [**?**]

Root-Root parallelization consists in applying this strategy on two levels : on threads and on machines. For this 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 same structure will be repeated locally on each machine, with one master thread collecting the results of the other threads.

4.2 On computer

Given the size of the tree to be explored (at least 400 000 000 nodes), a Root paralellisation strategy on computers could not be kept. A 11 000 000 nodes tree require about 2 Go of RAM7. If this should be timed by 40 and the number of core, there would be not enough space.

```
There is       39 percent of memory in use.
There are    3981 total MB of physical memory.
There are    2411 free  MB of physical memory.
max num of leaves : 10940207

There is       93 percent of memory in use.
There are    3981 total MB of physical memory.
There are     255 free  MB of physical memory.
```

Figure 7: *Memory used before and after the creation of nodes.*

With this in mind, a Tree paralellisation strategy has to be implemented on the Computer. It can be done by multithreading the *While // loop simulations* descriped in the base algorithm implementation2.

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.

In order to prevent any conflict between threads, a small critical section has to control the tree expansion. This can be done by using the following statement before the call of *UpdateNode(node)*:
#pragma omp critical

4.3 On cluster

The *Message Passing Interface Standard* (*MPI*) is a message passing library standard. We might use it for machine parallelization if the tests show it gives better results than *CAF* (see part **??**). 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 8.

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. Synchronal sending request 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.

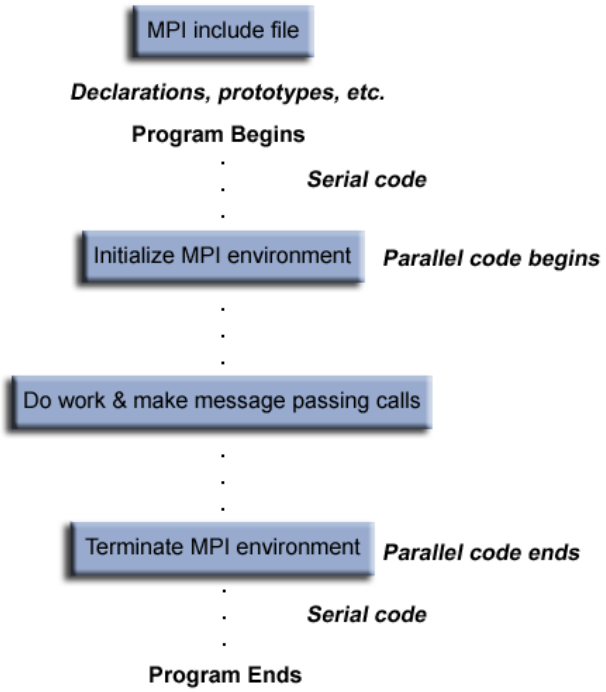

Figure 8: General MPI Program Structure. [?]

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 9. This will be especially useful for the algorithm, as it will require to regroup the results of every process (see part **??**).

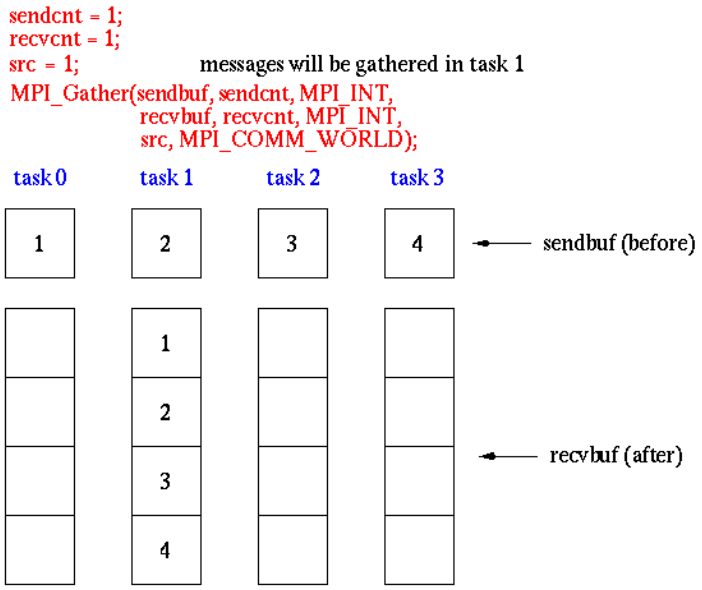

Figure 9: Example of a Collective Communication Routine. [?]

5 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 funcionnalities 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

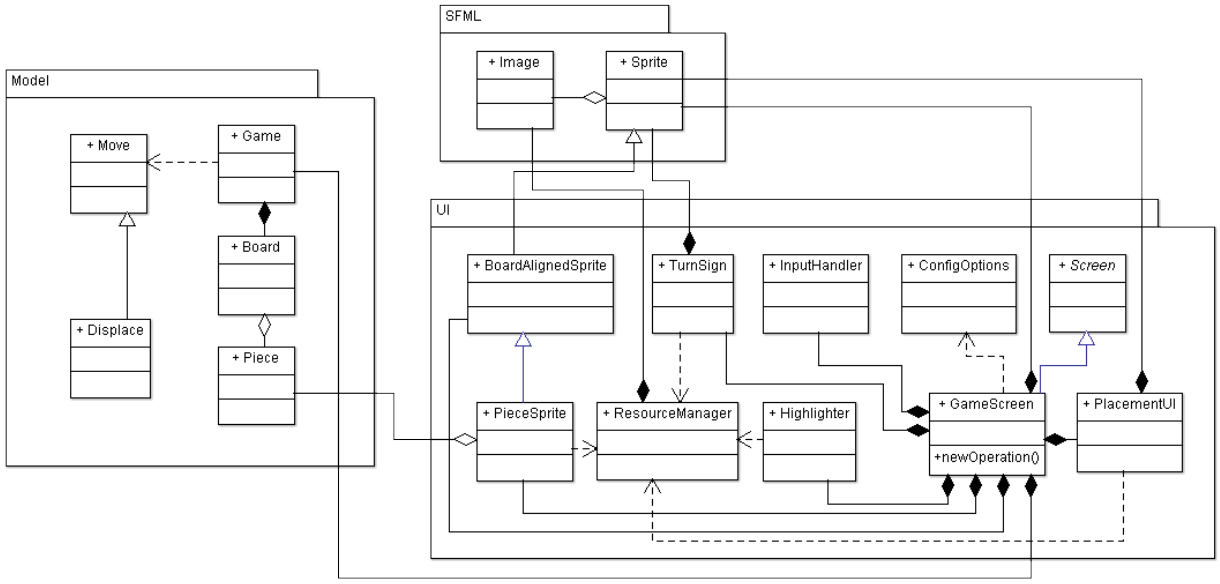

Figure 10: A UML diagram for the UI.

A UML diagram for the UI is presented in figure 10. In this diagram, you can see three modules:

- The model, rogrouping 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

5.2 The model

The model handles the rules of the game. A game is modelized by the class *Game*. This class conains a *Boar*tself 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.

6 Conclusion

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

7 Annexes

7.1 MCTS Class Diagram

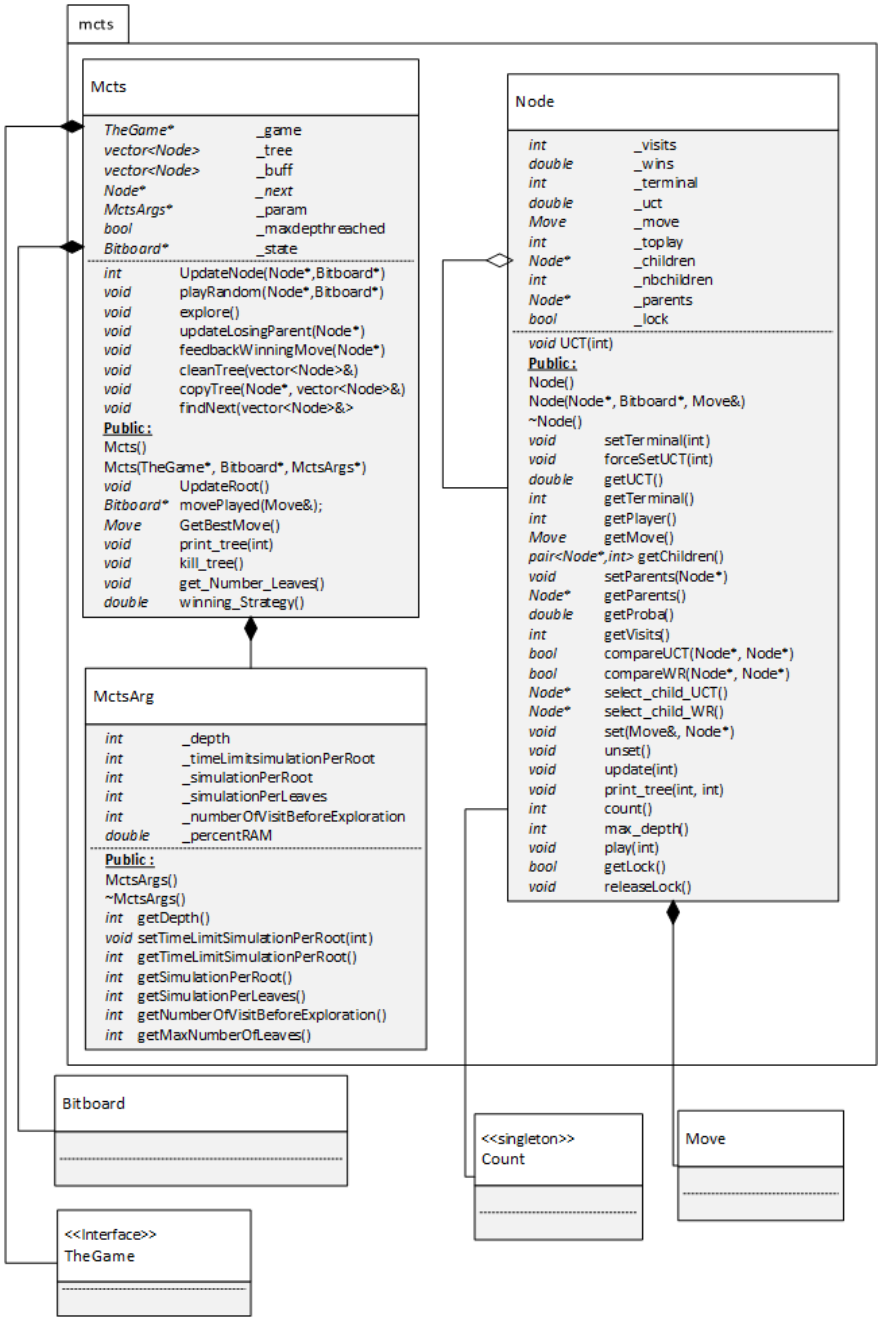

Figure 11: *Details of the MCTS namespace.*

7.2 Data Structure Class Diagram

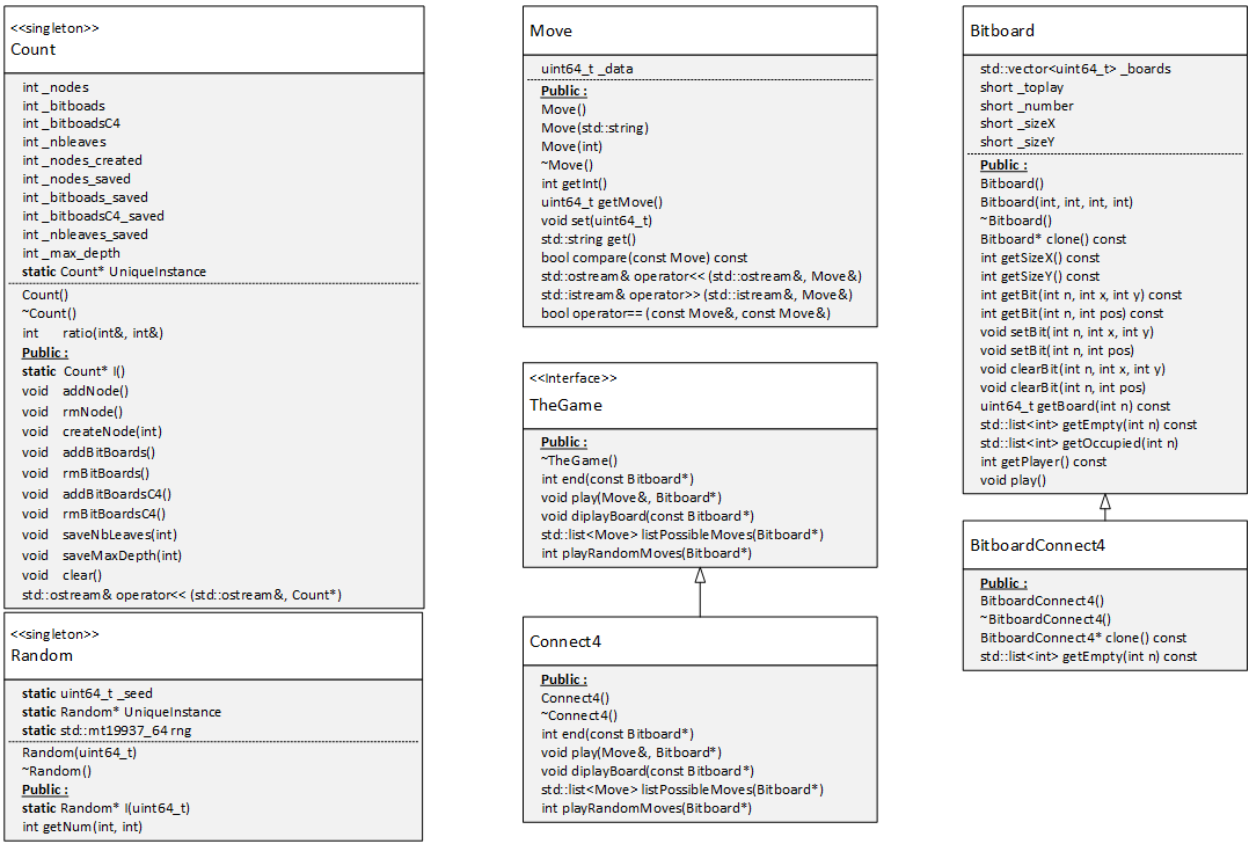

Figure 12: *Details of the DataStructure class diagram.*

7.3 Linux $log2(x)$ Implementation

```
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 }
```