ECM3401

Specification and Design Document

Designing Atlas, A Chess Engine

Ben Keen

January 2010

Contents

1 Specification **3**
1.1 Introduction . . . 3
1.2 Program Description . . . 3
1.3 Specification . . . 3

2 Design **4**
2.1 Introduction . . . 4
2.2 Basic Design . . . 4

3 Class Listing and Further Design **5**
3.1 Representing the board in memory - "Board" . . . 5
3.2 Generating and making moves - "Generator" . . . 6
3.3 Searching the game tree - "Search" . . . 6
3.4 Improving the evaluation - "Quiescence" . . . 6
3.5 Evaluating board positions - "Evaluate" . . . 7
3.6 Obtaining opening moves - "Book" . . . 7
3.7 Defining the output - "CommsProtocol" . . . 7
3.8 Controlling the program - "Game" . . . 7
3.9 Program Flow - how Atlas will play chess. . . . 8
3.10 Program extensions and redundancy planning . . . 8

4 Testing and Evaluation **8**
4.1 Move Generation . . . 9
4.2 Search and Evaluation . . . 9
4.3 Competitive testing using other chess engines . . . 9

5 Time Plan **10**

References **11**

1 Specification

This report details a more advanced specification for the Atlas engine than given in the Project Statement and Plan, and follows up with a design for implementing this system. The program is defined in a more formal fashion, then the design outlined and detailed. A plan for testing the program is given, and then a breakdown of the tasks to be performed in the remaining project time is described.

1.1 Introduction

Initially, when creating the Project Statement and Plan, it was thought that this Project would involve the creation of both a front and back end for a chess computer i.e. a Graphical User Interface (GUI) *and* an engine. During the Literature Review, however, a fantastic insight was gained into the complexity of creating a chess computer from scratch. From this, it was decided that it is beyond the scope of the Project timeline to create both a front and back end for a program of this complexity. As the majority of the interesting work, both from an academic and programming standpoint, lies within the creation of the engine, all effort will now be focussed on this part of a chess computer.

1.2 Program Description

Atlas will be a computer-based chess-playing engine designed to adhere to certain chess-computer standards. In order to play against Atlas, a player will be able to use one of two methods. The first method, which will be created initially, will be a simple text-based move entry system. The player will be able to enter moves in a standard form, as well as other arguments to see the board, the current position evaluation (i.e. which side is winning) and other such items. The player will also be able to see the "thought process" (i.e. the search tree) that Atlas creates as it searches to find moves.

The second method will allow a player to play against Atlas by interacting with a GUI. More specifically, an open source GUI called Arena (http://www.playwitharena.com/) will be used. Support for this style of play will follow at a later date in the project, but should increase the user experience of Atlas immensely.

Starting with a simple system such as the first method above will allow play against the Atlas engine from a very early stage. It will aid debugging as well as user interaction testing and with such a basic system of entry it will be easy to add any other features that may be desired on the fly.

1.3 Specification

The Atlas engine is aiming to adhere to these specified goals, described in the design and the Literature Review:

- It will be able to play an entire, fully-legal game of chess against a human or computer opponent.
- It will utilise BitBoards and rotated BitBoards to represent the board and generate moves.
- It will be capable of move tree searching using the Alpha-Beta algorithm to a depth of 5 ply (i.e. 2 and a half moves).
- It will evaluate a position based on the material balance of the chess board at this point, and as an extension will look at basic strategic balance.
- It will utilise very basic Quiescence Searching.
- It will use an book of opening book created specifically by the programmer.
- It will have a bespoke text-based interface and also be able to interact with a GUI using the Universal Chess Interface (see design section).
- A player will be able to enter moves in all standard chess notations to play a move.
- It will support the use of Forsyth-Edwards Notation (see www.chessgames.com/fenhelp.html) to define a partially played game .
- It should be able to search and play a move within a reasonable amount of time (approximately 15 seconds).
- It should attain an ELO (see www.fide.com for info.) of approximately 1500 - that of a good amateur player.
- *Stretch Goal* - It will support the use of Transition Tables.
- *Stretch Goal* - It will implement the Killer Heuristic.
- *Stretch Goal* - It will implement the History Heuristic.
- *Big Stretch Goal* - It will implement Parallel Alpha-Beta searching to work on multiple processors simultaneously.

From this specification it is quite possible to generate a design that will fit the timeframe that exists for the Project. Please note that Stretch Goals will only be achieved, or indeed attempted, if there is an abundance of time during the final stages of the creation of the program.

2 Design

2.1 Introduction

Before defining the design of the program, it is important to give a brief introduction to a few of the standards which will be adhered to within Atlas.

Universal Chess Interface The Universal Chess Interface, or UCI, is a protocol that allows chess engines to communicate with a graphical user interface. They do this by passing specific text commands to the GUI to represent the current status of the engine (e.g. bestmove [move]).

As the bulk of creating a chess computer is the engine programming and many UCI-compliant GUIs are readily available, most modern chess engines will support either this, or the similar Xboard protocol. This allows the engine programmer to concentrate on the challenging work of the project without having to create a bespoke GUI.

Long Algebraic Notation Long Algebraic Notation (LAN) is the standard way of representing chess moves within a UCI-compliant chess engine. It is an expansion of the basic Algebraic Chess Notation, which gives moves in a form such as "Nf3" (k**N**ight moves to **f3**).

LAN details both the start and end square of a piece, such as "Ng1f3" (k**N**ight moves from **g1** to **f3**). LAN also has the advantage of rendering game logs somewhat easier to understand for chess amateurs.

2.2 Basic Design

In order to design the program, it was important to understand the process that a chess computer goes through to make a move. This process is fairly standard across chess engines, and is usually like such:

- Define a way to represent the board in memory and generate moves from this.
- Utilise a Search function to examine the possible tree of moves.
- Examine the outcomes of this search with an Evaluation function, and find the highest value move.
- Play the best move found using the Search function by interacting with the board.

From this, we can see that there are certain standard aspects which are vital to the operation of the chess computer engine. In the next paragraphs, the way in which Atlas will implement these important sections, and other main sections, will be detailed.

Board Representation The board will be represented in memory using a series of BitBoards, as described in detail in the Literature Review. The decision was made to utilise BitBoards as they provide a great increase in efficiency for a chess engine once they are correctly set up. The Rotated BitBoards also described within the Literature Review will be used as well, in order to maximise the gains in efficiency.

There exists a possibility that the level of understanding required to utilise BitBoards will be too high, and if this is the case a much simpler Offset representation will be used.

Move Generation The move generator within the program will initially create a list of pseudolegal moves (i.e. moves that are fundamentally legal, but may breach rules of check) before checking these moves using a specific function. This is primarily being done to allow more transparency within the debugging process for this section of the code. Were a purely legal move generator in place, any problems in the basic move generation could possibly be obfuscated by the component which tests for check.

Search Function The program will be using a relatively simple search function to begin with: a Negamax implementation of the Alpha-Beta search algorithm. This algorithm is a fair amount more simplistic than some modern algorithms, but the more complex algorithms often require a much more in-depth knowledge of search functions than can be gained in the duration of this project. If that turns out not to be the case, however, and there is extra time remaining - a more complex search function such as the MTD(f) algorithm will be examined.

Quiescence Search Quiescence Searching is rather essential to the engine if it is to be capable of playing an entire game of chess through to completion. This is due to the Horizon Effect, described in more detail within Marsland [1991] and the Literature Review. The Quiescence Search within Atlas will be a very basic one, and will define a position as "quiet" if there are no possible captures to which the position leads. Due to this implementation's simplistic nature, it would be possible to extend it to include more complex notions of quiescence as detailed in Marsland [1991]. This extension will only be prioritised, however, if Atlas is seen to have an issue resolving the Horizon Effect.

Evaluation Function Evaluation, as mentioned in the Literature Review, is the most complex part of the program from a chess point of view. Thus, Atlas will be using a very basic evaluation function to begin with which simply uses the Material Balance. As an extension to the complexity of the evaluation function, it would be possible that some more ambiguous concepts such as pawn structure and mobility could be used to also define the evaluation of the board. It is important to note, however, that the most significant aspect of the evaluation function will *always* be the Material Balance.

The Opening Book In the opening phase of a chess game, chess computers can suffer from a particular weakness based on their deterministic nature. That weakness is the notion that a chess computer, if chosen to play first, would always play the same move as it has chosen that to be the “best” move for the situation. In essence, while this is correct, to behave more organically and unpredictably, Opening Books were created.

An Opening Book is just a list of moves, one of which the chess engine will pseudo-randomly select and use in the opening of a game in order to alleviate predictability. This also allows chess programmers to customise their opening to situations if required. As it negates the need to find a move using the search function in the opening, it also therefore saves on processor time and speed up play at the beginning of the game. A great deal more information about this topic can be found within Buro [1999].

Within Atlas, a very basic Opening Book will be created based on openings played by high-level human chess players.

Communication Protocol and handling user input As mentioned above, Atlas will support the Universal Chess Interface (UCI). As standard, however, it will utilise a simple text-based command line interface which will allow the program to be run without the necessity for a GUI to be installed. User input will be handled by matching input against a series of possible inputs, and this will control the program.

3 Class Listing and Further Design

In this section follows a class-by-class listing of the *main* classes and functions within the program. Note that it is practically impossible at this stage to list *every* class or function that will exist in the final program. It is, however, possible to identify some aspects of the program which could be prioritised over others and as such, functions that are deemed of secondary importance are marked with “Extension Function”.

Figure 1 below is a diagram of the structure of the program and details on how the classes fit together. The classes will each be described in detail within this section.

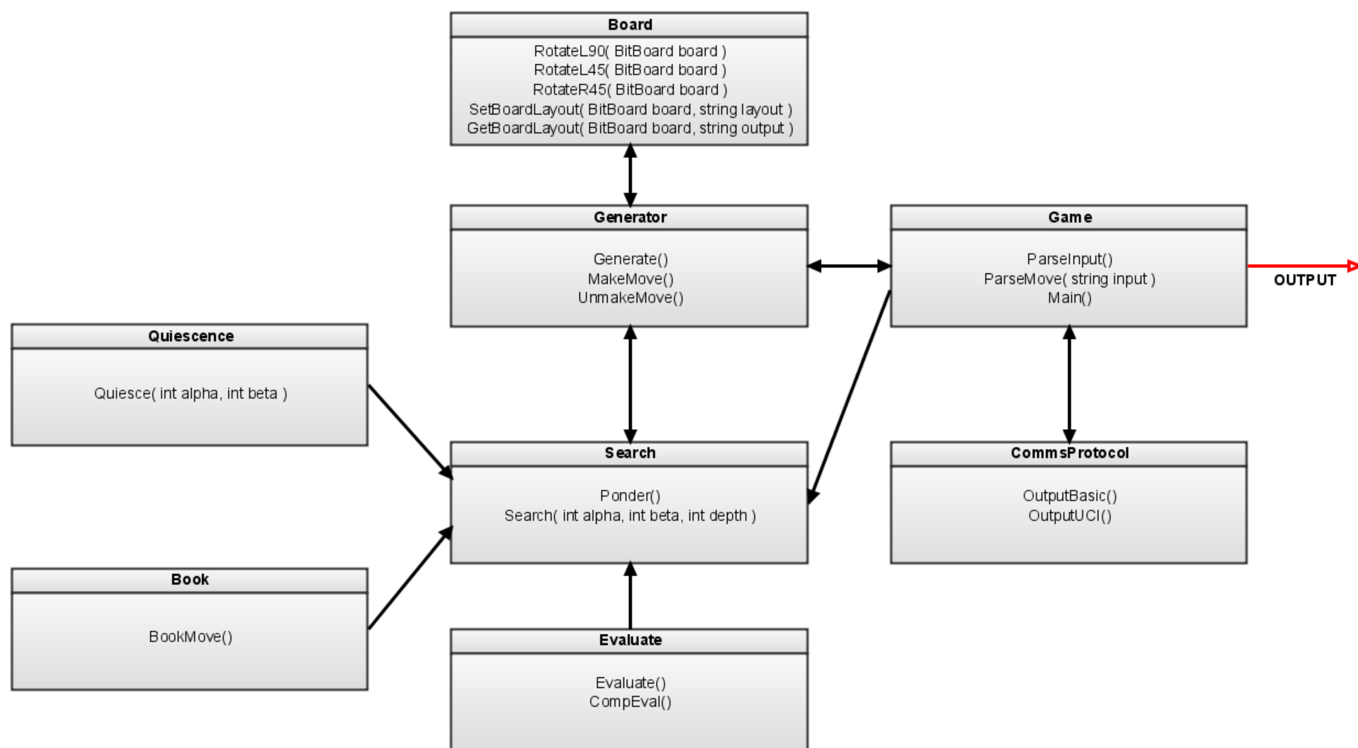

Figure 1: Class diagram representing the basic structure of the Atlas engine.

3.1 Representing the board in memory - “Board”

This class will be responsible for the management and manipulation of the BitBoards which represent the chess board internally. The Board class is the “backbone” of the chess engine, in that all other parts of the program rely on it, and as such it will be created first. **RotateL90(BitBoard board)** This function will “rotate” the supplied BitBoard (board) by 90 degrees to the left, and then returns this rotated BitBoard. This will be used to more easily generate moves along the Files of the board, as opposed to the ranks.

RotateL45(BitBoard board) This function "rotates" the supplied BitBoard (board) by 45 degrees to the left, and then returns the rotate BitBoard. This will be used in the move generation of "Sliding" pieces, such as the Queen.

RotateR45(BitBoard board) The opposite of RotateL45, this function "rotates" a given BitBoard (board) to the right by 45 degrees, and then returns the rotated BitBoard.

SetBoardLayout(BitBoard board, string layout) This function sets the specified BitBoard (board) to the layout (layout) given (in Forsyth-Edwards Notation) in order that the board may be set to any starting position desired. This allows the engine to support partially played games, or provide a rudimentary game-state saving system.

It sets the layout by parsing the FEN string (layout) and setting the board to this layout. It then determines other factors about the game situation such as whether each side has castling rights and which side is next to move.

GetBoardLayout(BitBoard board, string output) Converse of the SetBoardLayout function, this will convert the current board layout into Forsyth-Edwards Notation. This forms the other half of the aforementioned rudimentary saving system, allowing the player to keep track of the current game state.

3.2 Generating and making moves - "Generator"

The Generator class is responsible for the generation and storage (within arrays) of possible moves for the chess engine to make. This specific implementation will be a "pseudo-legal" move generator. This means that it will check if a move is allowed for simple reasons such as square occupation, but that it will not test whether or not the King would be left in check after the move. Handling this will be done elsewhere in code to allow the Generator class to remain more basic.

This class is probably the most important part of the program, despite its relative simplicity. It is extremely important that this class is free of bugs, as any errors will propagate throughout the program and either weaken the engine's performance, or simply leave it unable to function correctly. The testing section below details a way in which the bugs will be minimised within this class.

Generate() This function is responsible for the generation of all possible moves from a specific position. It will utilise multiple BitBoards from within the Board class to generate all pseudo-legal moves and then store them in various arrays.

GenerateCaptures() Similarly to the Generate() function, this will generate all possible captures from a given board position. This is used by the Quiescence Search for determining quiescent positions.

TestCheck(Move move) This function will test if the supplied move is fully legal by analysing whether or would lead to the player being put in check or not.

MakeMove() MakeMove() is a chess computer standard function that will be responsible for actually "playing" a given move when it has been decided upon. It will update the board layout and any other information as necessary in order to play the move.

UnmakeMove() This, simply the opposite of MakeMove() will revert the last made move in order to retrieve the previous board position and data structure. This will be used within the Search class during Ponder() for when the timer runs out, among other places.

3.3 Searching the game tree - "Search"

The Search class manages and performs all of the move tree searching for the Atlas engine. It utilises a Negamax implementation of Alpha-Beta search, as detailed in the basic design. The basic mode of operation will be that Ponder() is called, and then from within there Search(...) will be called.

Ponder() This function will be called from within the Game class' Main function and is responsible for timing the search process. Ponder() will call Search(...) from within over and over, stopping when a pre-defined time limit is reached. It will also stop if the maximum search depth is reached within the given time.

Search(int alpha, int beta, int depth) This is the move tree searching algorithm for the Atlas engine. The variable depth represents the maximum search depth to which the algorithm should be run. Much greater detail about the operation of the Alpha-Beta search algorithm can be found within the Literature Review.

3.4 Improving the evaluation - "Quiescence"

The Quiescence class is solely responsible for performing the Quiescence Search for Atlas, and functions within the boundaries described in the basic design section of this report.

The reason behind having the Quiescence Search in a separate class is to distinguish the two searches, both of which use an alpha and beta variable, from one another. Separating them allows the Quiescence Search to be improved upon more easily if time allows it.

Quiesce(int alpha, int beta) This function is responsible for performing the Quiescence Search. As defined above, it will assess positions as quiet if they do not lead to captures. Example pseudocode for this can be seen within Moreland [2001].

3.5 Evaluating board positions - "Evaluate"

The Evaluate class is responsible for determining the value of a specific board position for use in the Search class. It does this as described above, by using Material Balance. Separating it from the Search class allows easy extension of either class without bloating a single file too greatly.

Evaluate() This function is the only one which will exist to begin with, and is responsible for determining the material balance of the board. It will return a score which will be positive if it advantageous for white, and negative if black is in a better position.

Extension Function **- CompEval()** The more complex type of evaluation, the strategic evaluation, will be handled from within this function if it is written. This will be called from within Evaluate() and will return a value to indicate the strategic importance of the given position. The two measures of positional evaluation will then be combined within Evaluate().

3.6 Obtaining opening moves - "Book"

In order to play moves from an opening book as described in the basic design, there must be a class named Book which is responsible for the reading and parsing of the opening book file. It will be used during the opening phase of the game only, and will be called from within Search.Ponder().

BookMove() BookMove() is the function which manages the entire process of playing a move from the opening book. The book class will be passed a file name, and BookMove() will then select a move. It will do this by reading in the contents of the file, pseudo-randomly selecting one of the moves in the file, and then passing information about this move to the rest of the program.

3.7 Defining the output - "CommsProtocol"

As mentioned previously in the report, the engine will be able to either operate using a basic text-based interface, or by using a GUI. In order to decide which mode it is running in, the CommsProtocol class exists. This will be responsible, at the beginning of a game, for determining which mode the player wishes to use.

OutputBasic() This function will put the game into the basic text-based mode. This will be reflected within the main function in Game, and the game will proceed in this mode until changed or the game ends.

OutputUCI() This function will set the game to run in UCI mode, which is the mode for communicating with an external GUI. This mode will most probably require the opening of an external GUI at the same time in order to communicate, but the specific implementation of the GUI will not be known until creation.

3.8 Controlling the program - "Game"

The Game class is the main class for the program and contains the functions for dealing with player input as well as the Main function. This is the class that is most likely to contain extra functions when actually created as it is very hard to predict the exact method by which the entire game will be managed. The Game class controls the player's turn, and when communicating directly with the CommsProtocol, will output the required text to work with either the basic text interface or the GUI.

ParseInput() ParseInput is the function which is responsible for testing what the player wants to do by reading in the input from the command line and then testing it against a multitude of options.

ParseMove(string input) This function will be called from ParseInput() and exists so that the player is able to enter moves in standard chess formats (Short or Long Algebraic Notation) in order to play them.

Main() The Main function within the Game class is responsible for running the program, as well as keeping track of a multitude of variables. One of the most important, engine-specific items that it will track is the side which is next to move. It will also be the location from which Search.Ponder() is called.

3.9 Program Flow - how Atlas will play chess.

Now that the classes have all been detailed, it is important to get an idea of how they will work together within the program. In order to give the best possible description of how the design for Atlas will achieve the specification, however, it would be pertinent to look at how Atlas processes the move sequence given at the start of the basic design section.

1. The executable is run, and the player is presented with a command line interface. The player is then able to enter a number of different commands to control the program.
2. In this example, the player enters the command to have Atlas make a move. For simplicity's sake, we shall assume that it is not a move taken directly from an opening book.
3. Upon receiving this instruction, Generator.Generate() is called and the list of possible moves are created using the board layout within Board.
4. Having generated the moves, Search.Ponder() is called, and the search process begins (Reminder: Ponder will call Search(...) repeatedly).
5. When appropriate, Evaluate.Evaluate() will be called from within Search, and the score of the evaluated move will be returned to Search.
6. Once the time limit is reached for Ponder(), the current best move as determined by Search and Evaluate will be returned to the Generator class, which will make that move with MakeMove().
7. This will update the board layout within Board, and cause the game turn to switch to the player from within Main().
8. The player then makes their move, and the cycle will repeat.

3.10 Program extensions and redundancy planning

The principle of modularity is the core design tenet for this program. As this is a project which could potentially have an extremely large scope for work within the engine programming, it is essential that it is created in a way which allows additional features to be added on top of current ones when desired or necessary. As demonstrated in the class diagram, and the time plan which follows, the Atlas engine has been constructed in such a way (i.e. Quiescence separate to Search). This should allow a program to be up and running early, whilst also mitigating against the possibility of any problems with the core classes.

The program is able to be extended in various ways if time allows this, and where applicable this has been indicated in this report (cf. Quiescence Search). The other way that the program could be improved is through the addition of extra features or enhancements. During the Literature Review, a listing of enhancements to the search and evaluate processes was created. From this, certain extensions have been selected as noteworthy or simple to implement and these are detailed below. Whilst these are by no means the enhancements that would definitely be used, they are likely candidates at this stage.

The Killer Heuristic Originally devised by Greenblatt et al. [1967] and detailed more in the Literature Review, the Killer Heuristic increases the efficiency of the search function. Implementation would be a relatively simple matter of storing necessary moves.

The History Heuristic Harder to implement than the Killer Heuristic, but a similar premise, the History Heuristic stores a list of historically successful moves in order to increase search efficiency. Whilst this would increase the speed of play of the engine, the level of difficulty of implementation might be rather high for an extension. More detailed information can be found within Schaeffer [1989].

Transposition Tables Described in great detail by Marsland [1991], Transposition Tables are a relatively common feature within modern chess computers that allow storage of large numbers of previous moves in a table for later access. Whilst these would provide a relatively large impact on the engine efficiency, the implementation time may prove too long.

Parallel Search The only topic within this section not covered in the Literature Review is that of Parallel Searching - the idea of using multithreaded programs to utilise multiple-processor computers. The theory behind this is far too complex to detail here, but fantastic descriptions can be found within Rasmussen [2004] and Marsland and Campbell [1982]. It would be ideal if it were possible to implement parallel search, as it is a relatively recent and interesting topic. There are major concerns, however, that the theory would simply be too in-depth or that the implementation might not possible within the time frame.

4 Testing and Evaluation

Testing of the system will be split into three phases which are based on the main aspects of the program. These tests have been standardised throughout years of chess computer creation, and therefore a number of the test harnesses used will be implementations of standard algorithms or processes. These are described within the relevant sections where applicable.

4.1 Move Generation

The move generation of the program will be tested using a common debugging function called *Perft* (*Perf*ormance *t*est). This function was first used by Hyatt [2003] within his chess program *Crafty*. The function works on a very simple basis - it will step through the move generation tree and count the number of leaf nodes present. The function serves two purposes - both as a debugging tool and as an accurate test harness for new implementations of move generation algorithms. A standard implementation of the Perft algorithm is shown below.

```
1  typedef unsigned long long u64;
2
3  u64 Perft(int depth)
4  {
5      MOVE move_list[256];
6      int n_moves, i;
7      u64 nodes = 0;
8
9      if (depth == 0) return 1;
10
11     n_moves = GenerateMoves(move_list);
12     for (i = 0; i < n_moves; i++) {
13         MakeMove(move_list[i]);
14         nodes += Perft(depth - 1);
15         UndoMove(move_list[i]);
16     }
17     return nodes;
18 }
```

Algorithm 1: C-style pseudocode for a standard implementation of the Perft algorithm

When the Perft function is run, it will generate a number based on the current board position and number of plies to examine. It is then possible to compare this value with predetermined values that are readily available online from previous chess computer programmers.

Ostensibly, it is possible to compare the amount of time taken to generate the Perft results and use this as a measure of how fast the program's move generator is by comparison. This approach, however, can lead to misconceptions, as each implementation of the Perft algorithm will vary slightly from program to program.

4.2 Search and Evaluation

The Search and Evaluation functions of the program will be tested using a specific test suite called *Test-Positions*. This test, first designed and proposed by Kopec and Bratko [1982], has been the standard testing procedure for these elements of a chess program since the publication of that paper.

There exist many test suites which can be used, which have been contributed by several eminent chess computer programmers. Examples of these are:

- The Bratko-Kopec test suite. [Kopec and Bratko, 1982]
- The Kaufman test suite. [Edwards, 1993]
- The Nolot Suite - as introduced by Mark-Francois Baudot. [Baudot, 1994]
- Silent But Deadly - a test suite by Dann Corbit. [Corbit, 2008]

There are several other test suites which could be used to increase the "skill" of the program, but these will suffice to test the Search and Evaluation functions if time does not permit further testing. Due to the more complex, chess-based nature of these test suites, it is unlikely that Atlas will be able to solve a number of these problems, but it will prove useful to find the limitations of the software anyway.

4.3 Competitive testing using other chess engines

When the program is at a stage that it is capable of adequately playing a game of chess, then it will be very interesting to compare its performance to that of other chess engines. The results from this can then be analysed in order to improve the performance of the program.

In order to pit two chess engines against one another, a piece of commercial software called Fritz 12 will be used. Fritz is a very high-level chess engine, and this program allows users to play against it, and other, engines. Another excellent feature of the program is that it allows the insertion of *any* chess engine as long as it conforms to the UCI standard. As Atlas will be created to conform to this standard, it is able to be inserted into Fritz. Fritz is then able to simulate games between any two engines, and give a detailed breakdown of the outcome. This data will prove very valuable for testing purposes in the later stages of the project. It is also worth noting that the use of Fritz 12 provides a GUI for any chess engine running through the UCI. This is described above in the design stage of this document.

5 Time Plan

Tasks are split into three types based on their difficulty and importance to the overall program. Basic tasks are likely to take as long as indicated, advanced tasks are liable to take longer than shown, and extension tasks are generally unknown or very hard to predict a duration for. As such, this plan shows a best-case scenario for the project timeline, with the most likely outcome that the red tasks will go unperformed if the orange tasks take longer than anticipated.

The most important tasks from a functional point of view have generally been given a higher priority, in order to quickly get the program to a working state. Also, the program would still function within the specification if only the basic tasks were completed, and this adds a layer of redundancy to the program.

The durations of the tasks below are a best guess, as it is very difficult to predict the time it will take to perform some of these. It should be noted, however, that the larger tasks from within the design (i.e. Search class) have been assigned larger durations within the Gantt chart.

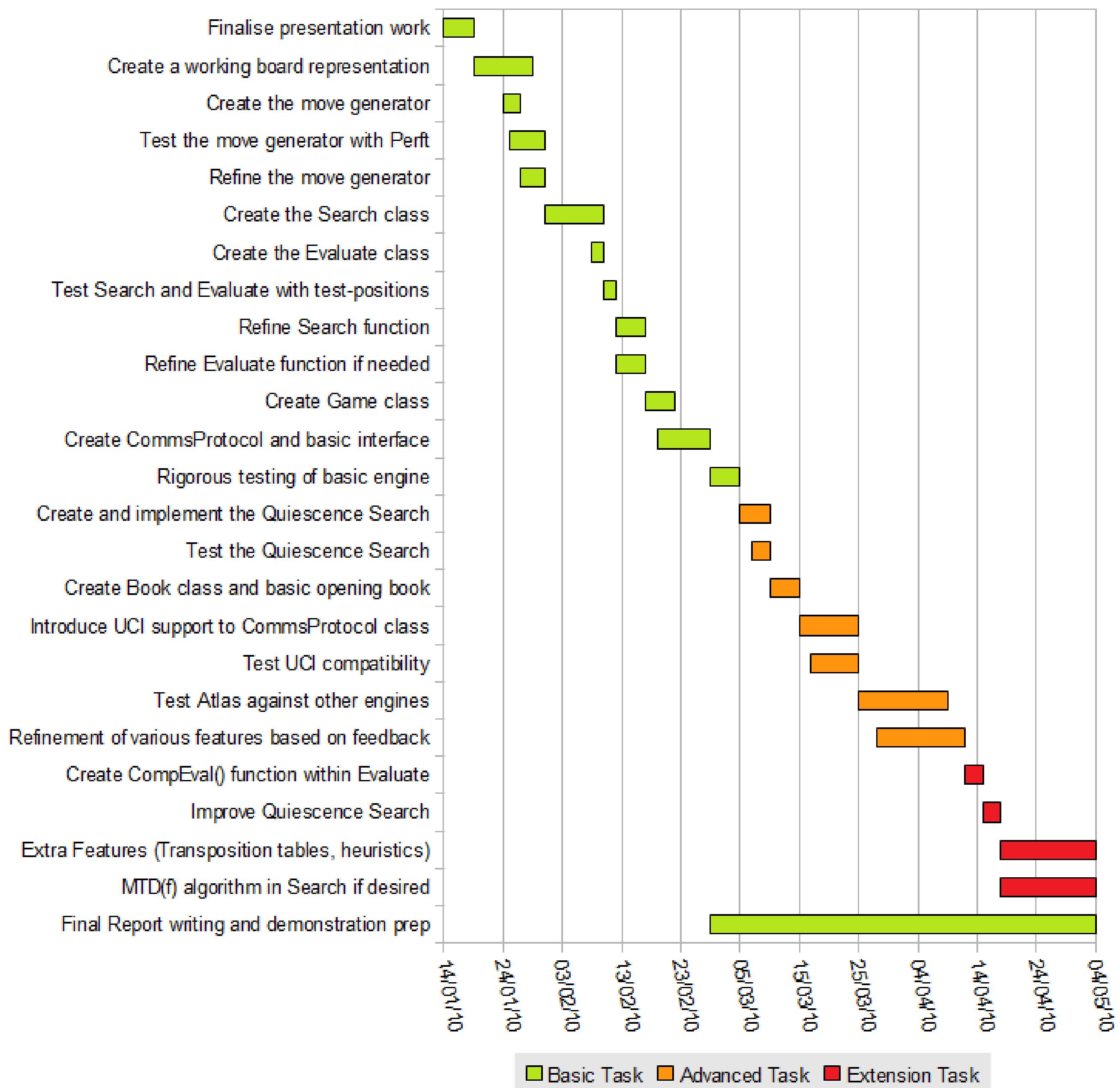

Figure 2: Gantt chart detailing time plan for remaining Project work.

References

M-F. Baudot. 11 tactical positions computers can't solve. http://groups.google.com/group/rec.games.chess/browse_frm/thread/79a6da1235df64e7?pli=1, 1994.

M. Buro. Toward Opening Book Learning. *ICCA Journal*, 22:98–102, 1999.

D. Corbit. Silent But Deadly. http://cap.connx.com/sbd.epd, 2008.

S. J. Edwards. Kaufman test suite in SAN/FEN . http://groups.google.de/group/rec.games.chess/browse_thread/thread/43e7aea0212adf5c#|Kaufman, 1993.

R.D. Greenblatt, D.E. Eastlake, and S.D. Crocker. The Greenblatt Chess Program. In *Procs. Fall J. Computer Conf.*, volume 31, pages 801–810. Thompson Books, 1967.

R. M. Hyatt. Crafty Command Documentation. http://www.cis.uab.edu/hyatt/craftydoc.html, 2003.

D. Kopec and I. Bratko. A Comparison of Human and Computer Performance in Chess. In *in Advances in Computer Chess III, edited by M.R.B. Clarke, Pergamon*, pages 57–72, 1982.

T. A. Marsland. Computer Chess And Search, 1991.

T.A. Marsland and M. Campbell. Parallel Search of Strongly Ordered Game Trees. *ACM Computing Surveys*, 14:533–551, 1982.

B. Moreland. Quiescence Search. http://web.archive.org/web/20040427014440/brucemo.com/compchess/programming/quiescent.htm, 2001.

D. Rasmussen. Parallel Chess Searching and Bitboards. Master's thesis, Danmarks Tekniske Universitet, 2004.

J. Schaeffer. The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11:1203–1212, 1989.