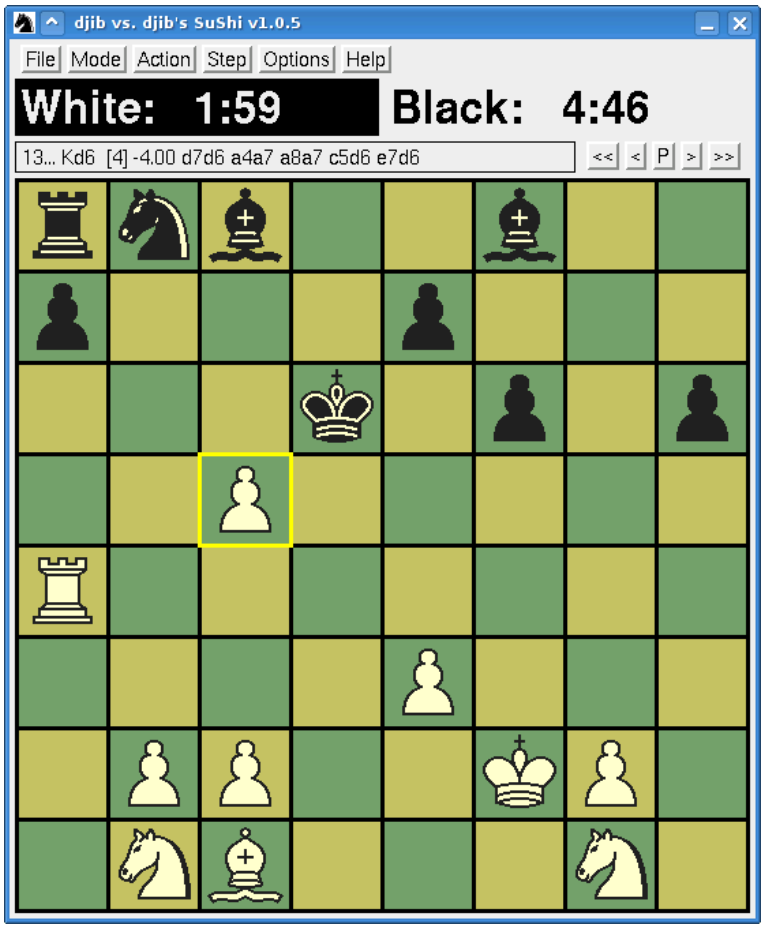
DJIB

Programming a *Suicide Chess* playing program

Diploma in Computer Science

Churchill College

September 25, 2019



Proforma

Name:	djib
College:	Churchill College
Project Title:	Programming a *Suicide Chess* playing program
Examination:	Diploma in Computer Science, July 2006
Word Count:	11574 words[1]
Project Originator:	djib
Supervisor:	William Tunstall-Pedoe

Original Aims of the Project

The aim of the project was to program a computer to play *Suicide Chess*. The program was to be designed using common chess programming algorithms such as *alpha-beta pruning*. I was planning to test its level using *suicide chess* problems and playing against *Sjeng*, a well known suicide chess program.
I then expected to improve my program by adding extensions such as move ordering, a transposition table, intelligent handling of time, ...
I was not intending to design my own user interface, but instead use *XBoard*, a famous protocol compatible with many interfaces.

Work Completed

I programmed every feature I suggested in my project proposal.
The computer player uses *Alpha-Beta Pruning* with *Iterative Deepening* and *Principal Variation First Search*.
I have fully implemented the *XBoard* protocol.
To test my program I have built-in a collection of 23 endgame problems. I also played it and won against other well known *Suicide Chess* programs.
I implemented these extensions:

- an opening book library of 25 variants,
- two static evaluation functions, one for midgame and one for endgame, using *Piece Values*, *Square Weights* and *Mobility*,
- *Adaptive Depth*, *Quiescence Search* and *Move Ordering* in the *alpha-beta pruning*.

Special Difficulties

None

[1] Word count given by the LaTeX suite *Kile*: includes titles, code, pseudo-code and footnotes. *Original Aims of the Project*: 93 words. *Work Completed*: 98 words.

Declaration

I, djib of Churchill, being a candidate for the Diploma in Computer Science, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

1 Introduction **1**

2 Preparation **5**
2.1 *Suicide Chess* Theory 5
2.2 Language 7
2.2.1 *Java* 7
2.2.2 *Eclipse* and *Subversion* 7
2.3 Chess Programming 7
2.4 Board representation 8
2.4.1 Array representation 8
2.4.2 *Bitboard* representation 8
2.5 Searching Algorithms 10
2.5.1 Evaluation 10
2.5.2 Searching and the *MinMax* algorithm 10
2.5.3 Alpha-Beta pruning 11
2.6 *XBoard*, an interface for Chess programs 12
2.6.1 Problems with *XBoard* 13
2.6.2 Playing on the FICS 15

3 Implementation **17**
3.1 Organisation 17
3.2 Type safety 18
3.3 Class *Board* 18
3.3.1 Datafields 18
3.3.2 Functions 20
3.3.3 Constructors 22
3.3.4 Evaluation function 23
3.4 Rules and Move Generation 26
3.4.1 Pieces except Pawns 26
3.4.2 Pawn 27
3.4.3 Storing legal moves 27
3.5 *XBoard* protocol 28
3.6 Searching 29

3.6.1 MinMax . . . 29
3.6.2 Alpha-Beta pruning . . . 30
3.6.3 Thinking output . . . 31
3.6.4 *Iterative Deepening* and *Principal Variation First* . . . 32
3.7 Extensions . . . 33
3.7.1 Opening Book . . . 33
3.7.2 Quiescence search . . . 33
3.7.3 Adaptive Depth search . . . 35
3.7.4 Move ordering . . . 36

4 Evaluation **37**
4.1 Efficiency . . . 37
4.2 Evaluation function . . . 38
4.2.1 Evaluation function in the endgame . . . 38
4.2.2 When does the endgame start? . . . 38
4.2.3 Evaluation function in the midgame . . . 39
4.2.4 A closer look at the *Cheeseparing Version* . . . 40
4.2.5 Versions with mobility . . . 41
4.3 Search Speed . . . 41
4.3.1 MinMax vs Alpha-Beta . . . 41
4.3.2 *Alpha-Beta* efficiency and move ordering . . . 42
4.4 Problem solving (Search extension efficiency) . . . 42
4.5 Matches against *Sjeng* and *KKF* . . . 45
4.5.1 Preliminary comments . . . 45
4.5.2 Results . . . 46

5 Conclusion **49**

Bibliography **51**

A Relevance of game statistics **53**

B A game between *Sjeng* and my program **55**

C Configuration file **58**

D Typical program output **60**

E Implemented *XBoard* commands **62**

F *Suicide Chess* problems (study positions) **65**

G Project Proposal **67**

List of Figures

1.1 Justification of the Stalemate rule 2

2.1 Pieces in Losing Chess 6
2.2 Representing a board with a 8 array 8
2.3 *Bitboard* representing the position of white pawns in the beginning of the game. 9
2.4 The *MinMax* algorithm 11
2.5 The *Alpha-Beta pruning* algorithm: dotted nodes will not be examined. 12
2.6 The *XBoard* protocol Version 2 (excluding features that I did not implement). Most commands are presented in Appendix E. For further explanations read [16, XBoard Chess Engine Communication Protocol] 14

3.1 Two opening move sequences leading to the same result . . . 34

4.1 Method timings 37
4.2 Result of matches of different version of my program. The result are written as: *games won by the program in the first program - games won by the second program - draw* 40
4.3 Comparison of the number of nodes examined by the *MinMax* algorithm (MM) and the *Alpha-Beta pruning* algorithm (AB) in typical *Suicide Chess* situations 42
4.4 Some test positions 43
4.5 Time (in centiseconds) to generate a move. Comparison of *Alpha-Beta pruning* (AB), *Move Ordering* (MO), and *Principal Variation First Search* (PV) 44
4.6 Efficiency in problem solving. The problems are available in *FEN* notation in Appendix F 45
4.7 Result of matches between different versions of my program and *Sjeng* or *KKF* 46
4.8 A 200 games 10 seconds per game match between my *Suicide Chess* program *djib's SuShi, Kamikaze Version* and *Sjeng* . . 47

A.1 Relevance of game statistics 54

Acknowledgements

Thank you to my project supervisor, William Tunstall-Pedoe, for guiding me in the right tracks.
Thank you to Dr. John Fawcett, my director of studies for his help and availability during all the year.
Thanks a lot to Rémi and Johann for showing interest in my project and letting me explain my problems to them. It helped me a lot.
Thanks to Yann for introducing me to the power of *Eclipse* and *Subversion*.
Thanks to Simon for the long discussions about our respective projects.
Thanks to James for lending me his book.
Thanks to all my fellow *Diploma* and *Part II General* students for this incredible year.

Chapter 1

Introduction

In 2002, I spent 4 weeks on a sailing boat. One of the main distraction on board was playing chess. A friend introduced me to the concept of *Suicide Chess*[1], a chess variant. Neither of us knew much about *Suicide Chess* theory, but we found that games were quicker and more dynamic than *Chess*.

In this report I shall consider that the reader is familiar with *Chess*. They should at least know how pieces move, how pawns promote to other pieces and what an "en passant" capture is.

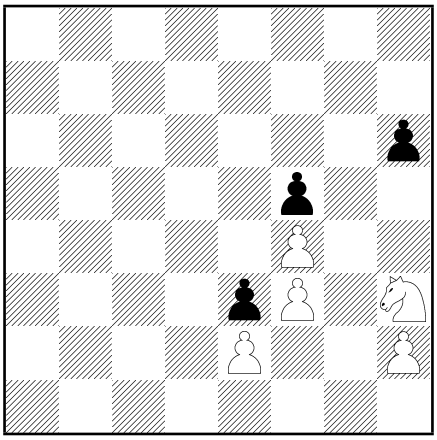
The rules of *Suicide Chess* are the following[1]:

- The opening setup is as in normal Chess. All pieces move as in normal Chess (but see below for the King).
- Capturing is compulsory. When a player can capture, but has different choices to capture a piece, he may choose which piece to capture.
- There is no check or checkmate. The king plays no special role in the game, and can be taken as any other piece.
- Pawns may also promote to Kings.
- Castling is not allowed.
- Stalemate is a win for the stalemated player. (International rules)

In my *project proposal*, I wrote:

> In the case of stalemate, the winner is the player who has the smallest number of pieces. If the number is equal, the game is drawn.

[1] *Suicide Chess* is also called *Losing Chess*, *Giveaway Chess*, *Killer Chess* or *Take-all Chess*. It is called *Vinciperdi* in Italian and *Qui perd gagne* in French[1]



White plays and wins.
Solution:

1.	♘h3–g1!	h6–h5
2.	h2–h3	h5–h4

White cannot move anymore. With the *FICS* rules, this would be a victory for *Black* that has less material than *White*. However, with the *International Rules*, the ones I chose, that position is a win for *White*.

Figure 1.1: Justification of the Stalemate rule

Those are the *Free Internet Chess Server (FICS)* rules, but they are less logical[15] than the ones above: it would be just like in Chess, looking at the material advantage when a player is in checkmate to decide who is winning. To convince yourself, just look at the example Figure 1.1.

Programming a *Suicide Chess* player was for me an opportunity not only to discover more about that game I enjoyed so much but also to apply some interesting software engineering techniques. Moreover I thought that programming such a game would be less complex than Chess, and thus more adapted for a diploma project, but as a matter of fact, Stanislav Goldovski, a pioneer in *Suicide Chess* theory proved me wrong:

> "Losing chess does involve theory and strategy. Moreover, it seems to be a very complicated and profound one. Once basic tactical blunders can be avoided, the strategic depth becomes apparent - mobility, space, development, weak squares, king safety, pawn structure. **Indeed, it is even possible that the game may rival regular chess in profundity**. For most players Losing Chess is still in its "romantic" period, when tactical battles usually rage, but many have already discovered its deep strategic

side, things we could only dimly see before".

There already exists several *Suicide Chess* programs on the Internet. Some are no longer maintained or even available on the Internet, some others are not free. I managed to play against the following three programs:

- *Losing Chess* on `http://play.chessvariants.org/erf/LosingCh.html` which is terribly poor,
- *KKFChess*[5], a rather good program against which I sometimes managed to win,
- *Sjeng*[17], a reference:

  > "In suicide and losers chess Sjeng beats all but the very best humans, and will beat non-specialised programs easily".

  In *Sjeng*'s *known bugs* list, the author wrote "Loses sometimes" which summarises how good the program is. This program is open-source though there is now a version called *Deep Sjeng* that costs 40 euros and that is said to be even better.

A longer list is available on [4, ICGA: Losing Chess Information].

Chapter 2

Preparation

When I started this project, I knew nothing about artificial intelligence and how to write game programs. My project supervisor guided me by giving me the right keywords: *static evaluation function*, *minmax*, *alpha-beta pruning*, *iterative deepening*, ..., and during the first few weeks of my project I researched those terms.
I also read the book [12, Computer Gamesmanship].

2.1 *Suicide Chess* Theory

I found a few interesting articles ([8], [13] and [1]) about *Suicide Chess* strategy but the content was often quite limited. It is not until much later that, searching for webpages in French, I came across Fabrice Liardet's fantastic [14, Pion.ch], the only page that deals with every aspect of the game, from the opening strategy to the endgame. This website also goes into much more detail than every other website.

Unfortunately, though a few people programmed *Suicide Chess* computers, I didn't find any webpage about how to actually implement the basic concepts of *Suicide Chess*. For example, in *Chess*, it is of common knowledge that the Queen has a value of 9 pawns, a Rook 5 pawns, a Bishop 3 and a half, and a Knight 3. I based my experiments on Stanislav Goldovski's table[7] (cf Figure 2.1). I will refer to that table later in the choice of values for my evaluation function. I will also explain how I read some of *Sjeng* source code to try and design a better evaluation function than the one I used initially.

[1] zugzwang: German for "compelled to move". It is a situation in which playing is a disadvantage and leads to losing. If one could "pass" that would not happen.

Piece	Advantages	Drawbacks
King	The most important piece. Its safe moves are often needed to avoid zugzwang[1].	Too slow for other tasks.
Queen	Very useful for middle-game tactics, for attacking the king, weak squares, etc.	Dangerous in open positions, in the endgame.
Rook	Best endgame piece, quick and powerful.	Can easily turn into a 'loose cannon'.
Bishop	Good for endgame. Also for draws (due to opposite colour bishops).	See for Rook. Much worse even.
Knight	Very good for destroying pawn formations, for 'forking' weak squares.	Too immobile, very bad in the endgame.
Pawns	Very useful for restriction of the opponent's pieces, and also destroying pawn formations.	Slow, immobile. Often dangerous when it comes to promotion.

Figure 2.1: Pieces in Losing Chess

2.2 Language

2.2.1 *Java*

To implement my program I decided to use *Java* for the following reasons:

- *Java* is the language taught in the course *Foundations of programming*,
- *Java* is cross platform,
- I like the type-safety of the language and the high level datastructures.

I had to test that *Java* could interact with *XBoard*, but since *Java* programs can read and write from and to the *standard input/output* there was no problem.

I also discovered later in my project that *Java* can generate documentation files automatically (using `javadoc`), provided that the programmer respects certain formatting rules. This is a feature that I really appreciated because it forced me to write coherent comments for every function I was programming.

2.2.2 *Eclipse* and *Subversion*

To program, I used *Eclipse*. *Eclipse* is an amazing *IDE*, and along with *Subclipse*, a plug-in that allows interfacing *Subversion*, a version control system, it provides the best environment for developing in *Java*.

I used a *Subversion* server running on my own computer to do backups of each change in the source, and I did manual backups on an external drive every week to avoid disaster. Being able to revert to any previous version is a feature that I really appreciated.

2.3 Chess Programming

I found many pages on the Internet about Chess programming, but the website that has been the most useful is by far [10, GameDev.net]. Not only did it explain every important component, but it also proposed an order for implementing those components:

> "In order to play chess, a computer needs a certain number of software components. At the very least, these include:
>
> * Some way to represent a chess board in memory, so that it knows what the state of the game is.
> * Rules to determine how to generate legal moves, so that it can play without cheating (and verify that its human opponent is not trying to pull a fast one on it!)

-5	-4	-3	-2	-1	-3	-4	-5
-6	-6	-6	-6	-6	-6	-6	-6
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
6	6	6	6	6	6	6	6
5	4	3	2	1	3	4	5

Figure 2.2: Representing a board with a 8 array

* A technique to choose the move to make amongst all legal possibilities, so that it can choose a move instead of being forced to pick one at random.
* A way to compare moves and positions, so that it makes intelligent choices.
* Some sort of user interface".[10]

That is the order that I chose for implementing my program.

2.4 Board representation

2.4.1 Array representation

Probably the most intuitive way of representing a board is a 2 dimensional array of size 8×8, each element in the array being a byte coding for the piece on the board (see Figure 2.2).

Instead of that representation, it is also possible to store for each piece type an array with the square(s) it is on. This representation has a major drawback: is not convenient for knowing what piece is on a square.

Even though those representations are quite intuitive, they are also quite inefficient, and in games like chess, played on a 64 square board, *bitboards* are more often used nowadays.

2.4.2 *Bitboard* representation

Instead of an *array representation*, I have decided to use a *bitboard representation* because it is more challenging and also more efficient.

The *bitboard* representation was developed by the *Kaissa* team in the Soviet Union in the late 1960s. It is becoming increasingly popular due to

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0

Figure 2.3: *Bitboard* representing the position of white pawns in the beginning of the game.

the fact that 64, the number of squares on the board, is the size of one word (or two) on some computers nowadays, and it will be on most computers in the near future. My computer is a 32 bit architecture and I would expect my program to be quite a lot faster on a 64 bit architecture[2].

> "On 32-bit microprocessors, many of the operations we want to do on bitmaps are not efficient because they have to be done piece-meal using 32-bit registers. For AND, OR, XOR and similar operations, this doesn't hurt much, but for shifting, and, more importantly, finding the first one-bit that is set, current machines don't offer any features to make these operations efficient. [...] It is highly likely that future chess programs will migrate to this representation or be left behind in performance. [...] For now (2004) the bitmap and offset approaches seem to be nearly equal, with the one exception that the evaluation using bitmaps is always going to be significantly faster"[9].

A *bitboard* is a 64bit number where each bit represent one square on the board, and the bit value represents the presence or the absence of a piece on that square.
Typically, a chess board would be represented using twelve 64bit numbers representing the positions of *white kings*[3], *white queens*, *white bishops*, *white knights*, *white rooks* and *white pawns* (cf. Figure 2.3), and 6 other numbers for black.

The *bitboard* representation has the advantage of being very versatile compared to array representation. For example if you want to know all

[2] *Sun* released a 64 bit version of the *Java* virtual machine
[3] There might be more than one king in *Suicide Chess* since pawns can promote to king

squares with a white piece on them, just `OR` the six white *bitboard*s. If you want to know all free squares, `OR` the 12 numbers and then invert the result.

Efficiency is easily understandable: finding the *white queen* position is a matter of loading the *white queen bitboard* and then finding the first one-bit that is set. On an *array* representation, this would require loading up to 64 numbers in memory.

More about the different board representation and the efficiency of *bitboard*s can be found on [9, Dr. Robert Hyatt's webpage]. In particular, *bitboards* can be used to easily generate legal moves, using shifting and rotated bitboard [6], but I didn't use that method as it is very advanced.

2.5 Searching Algorithms

2.5.1 Evaluation

Suicide Chess is defined as a **game of perfect information**, because both players are aware of the entire state of the game at all times: just by looking at the board, you can see which pieces are alive and where they are located. From a position, one can estimate and tell which player has the highest chances of winning. This process is called **evaluation** and is a function of the position. Typically a positive *evaluation value* represents an advantage for *White*, and a negative is an advantage for *Black*.

A few *Chess* programs in the past have used only a static evaluation function[4], but an *evaluation function* is often not good enough to tell, just by looking at a position, which move is the one that will give the best outcome for the current player. Even humans, that are very good at intuitively spotting a few possible good moves, have to analyse ahead to see if those moves will turn out to be good indeed, or bad. This is called *searching*.

2.5.2 Searching and the *MinMax* algorithm

From any arbitrary position in *Suicide Chess*, it is possible to construct a **search tree**, where the root is the current position, its children are the positions after a legal move from one player, its grandchildren are positions after a legal move from one player and then a legal move from the other player, etc., alternating players, until one side exhausts all possible legal moves.

Given any position, is theoretically possible to look at the outcome (the leaves of the *search tree*) of every move and choose the move that guarantees the best outcome. Unfortunately the search tree is so big in *Suicide Chess* that it is impossible to analyse all branches down to the leaves. Instead, we will search the tree to a fixed depth (called ply), and then estimate how good the resulting position is using the *evaluation function*.

[4]The *TECH* program is an example[12].

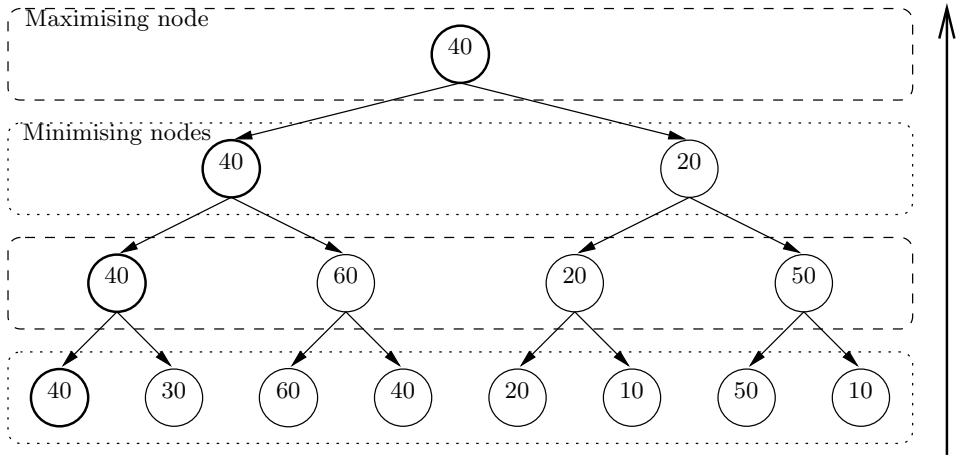


Figure 2.4: The *MinMax* algorithm

MinMax[5] is the most common algorithm in two player games of perfect information, where each player is supposed to play rationally[6]. It relies on the fact that each player will always pick the best move they can, ie. the one that has the best *evaluation value*. In other words, *White* will try to maximise the *evaluation value* and *Black* will try to minimise it.

The algorithm operates as follows: each node at the chosen *search depth* in the tree is evaluated using the *evaluation function*. The parents are then attributed the maximum (if it is *White*'s turn at that depth) or the minimum (if it is *Black*'s) of their children's values. Working this way up the *search tree*, the move that will end up with the best value for the current player will be chosen as the best move (cf. Figure 2.4).

2.5.3 Alpha-Beta pruning

Alpha-Beta pruning is an improvement of *MinMax search*. It is quite similar to *MinMax* but is avoids useless calculation. In other words it stops evaluating a branch in the *search tree* if this branch has been proved to be worse than a previously examined branch.

In order to do so, we need two values, often called α and β that hold the lowest score that *White* (the maximising player) can achieve and the highest score that *Black* (the minimising player) can achieve respectively. Usually α is initialised to $-\infty$ and β is initialised to $+\infty$ (closer bounding values can be used, but do not guarantee a result).

[5]The *MinMax* algorithm is sometimes also called *MiniMax*. I have chosen to use *MinMax* in this report

[6]From *Sjeng* documentation, I read that it does not use *MinMax* but *Proofnumber Search* when playing *Suicide Chess*.

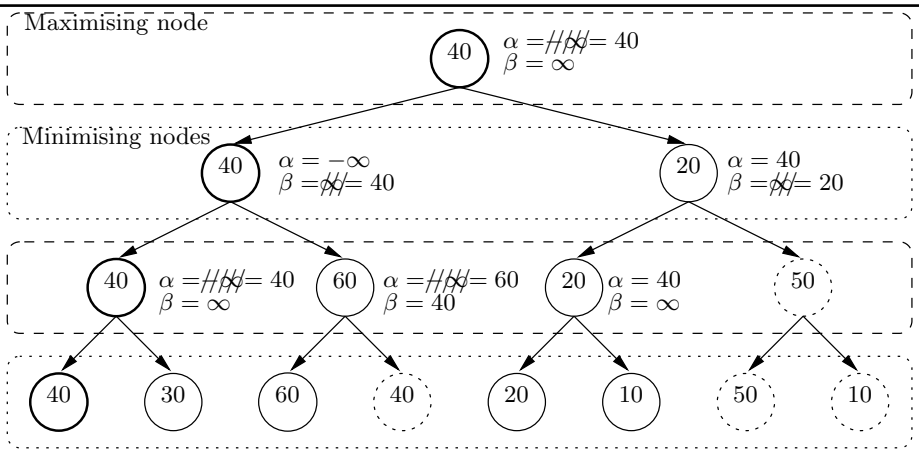


Figure 2.5: The *Alpha-Beta pruning* algorithm: dotted nodes will not be examined.

When in a branch $\beta \leq \alpha$, that branch can be pruned because it will not improve either side's current best score (cf Figure 2.5).

2.6 *XBoard*, an interface for Chess programs

Though I like graphics, I didn't want to design a user interface because interfaces take a long time to develop. I wanted to focus my available time on the algorithmic aspects of the project.

XBoard is a standard graphical interface for playing *Chess*. It was first developed to interface *GNUChess* but then most programs started using it. There is another protocol, the *Universal Chess Interface*, but I decided not to use it before writing the project proposal because it in inelegant (the entire list of moves must be transmitted each time) and does not allow console play which I thought was going to be an easy way of debugging my program[18].

Though *XBoard* is not multi-platform, it is available for *Linux*, *Mac OSX* and *Windows*.

From the *XBoard* documentation I also read that *XBoard* could be used to play matches between two computers, which is a very attractive feature. In the end it turned out that this feature is quite limited (for example it is not possible to give different time or ply depth to the two computers), but it was useful anyway.

XBoard has a long list of signals that can be exchanged between the chess program and the board [16]. This protocol is retro-compatible, and works with any version of *GNUChess* which makes it quite complex. Recently, Tim Mann, the author of *XBoard* has decided to simplify the protocol and standardise it:

> "I've had to make the protocol description more precise, I've added some features that GNU Chess does not support, and I've specified the standard semantics of a few features to be slightly different from what GNU Chess 4 does".

I decided to make my program only compatible with version 2 (the standardised version) of this protocol. I have fully implemented the protocol, but not all its features. This is possible thanks to the `feature` command that is used to describe to *XBoard* which of *XBoard*'s features have been implemented. See Figure 2.6.

2.6.1 Problems with *XBoard*

I appreciated having *XBoard* but I had some problems and disagreements with *XBoard*'s design:

- *XBoard* checks that moves are valid. This means that I could not play until my program had fully implemented all the rules. I don't understand why *XBoard* does that: one might think an interface is to do what it is told to do!
- *XBoard* can do matches between two computers, which I appreciated, but it is not possible to attribute different search time or search depth to each program and that really is a feature that I would have liked to have.
- I also find that the first version of the *XBoard* protocol is quite messy and has too many commands. This is due to its retrocompatibility with all versions of *GNUChess*. Apart from a full listing of all commands[16], I didn't find any website that explained exactly in what order and under what circumstance the signals would be sent to my program. I implemented only the commands available in the second version of the protocol, but my first steps with the *XBoard* protocol were quite tough: for example my program would quit unexpectedly, due to *XBoard* sending a `SIGINT` signal used to interrupt searching in the first versions of *GNUChess*!
- *XBoard* has a debug flag that displays debug information. Unfortunately all the information (*XBoard* specific, *GUI* interaction, dialogue with first and second computer) is displayed in the same console and that makes the whole thing hard to read.

These are the reasons why I designed a simple *ASCII* interface. If my program doesn't receive a `xboard` signal, it automatically switches to that interface. That gave me more freedom, and allowed better and easier testing. In Appendix D you can see the *ASCII* output of my program.

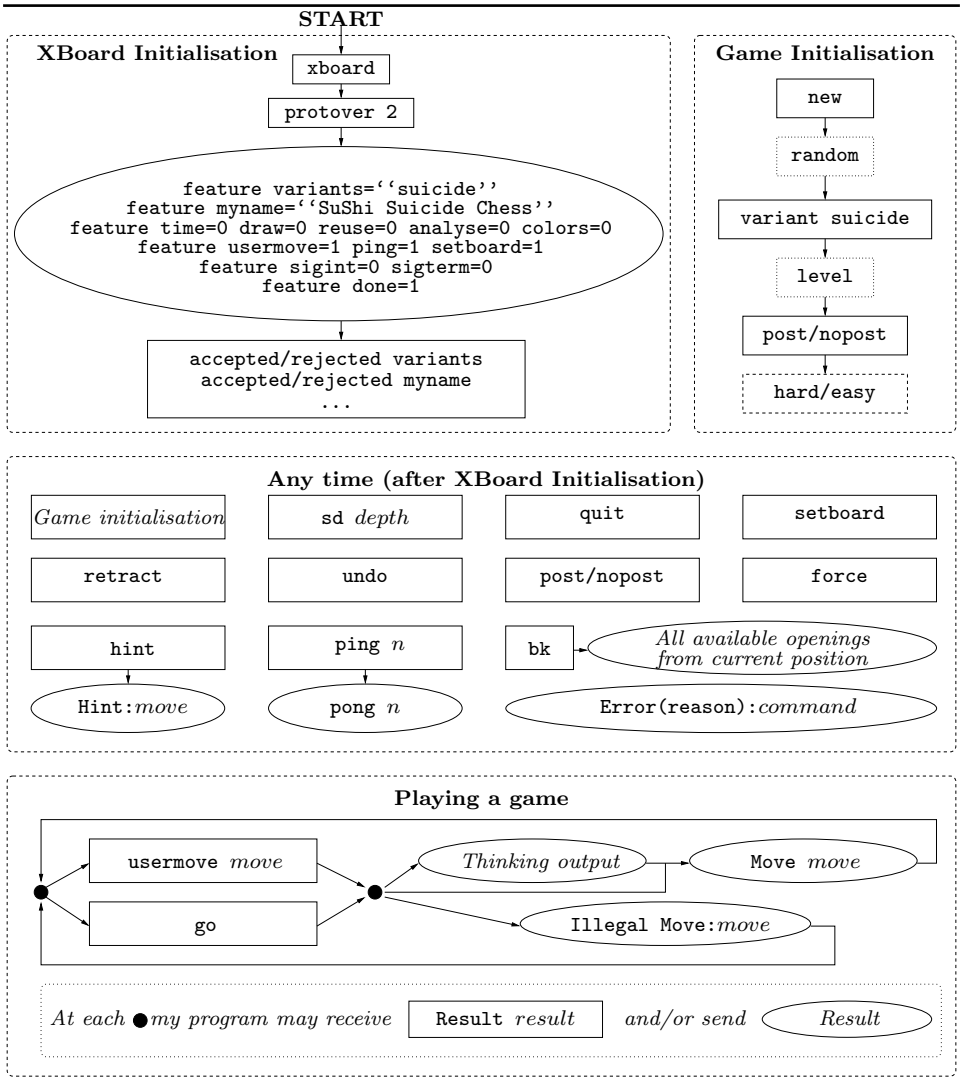


Figure 2.6: The *XBoard* protocol Version 2 (excluding features that I did not implement). Most commands are presented in Appendix E. For further explanations read [16, XBoard Chess Engine Communication Protocol]

2.6.2 Playing on the FICS

XBoard allows easy interfacing of the *Free Internet Chess Server*. Using *Zippy*, a module in *XBoard*, it is possible to play on the *FICS* with a chess program.

I wanted to use that feature, and I even managed to play a few games with a guest account, but guest accounts do not provide ranking. I applied twice (in February and in June) for a computer account and never got an answer.

Chapter 3

Implementation

3.1 Organisation

Here is the order in which I developed my *Suicide Chess* program:

- Design an efficient implementation of a board, with the possibility to add pieces, remove pieces, and move pieces.
- Program a move generation function (that implies an implementation of the rules of the game). At this stage it was possible to play two player games using the *ASCII* interface I developed.
- Implement a computer player that would simply pick random moves in the list of all legal moves.
- Interface my program with *XBoard* (which I first intended to do earlier, but I found out that I needed my program to be able to play).
- Implement the *MinMax* algorithm, which required designing an evaluation function.
- Add a collection of problems that the computer could either load independently, or play on all of them and then display statistics.
- Implement *Alpha-Beta pruning*.
- Improve my static evaluation function.
- Add extensions to my program: an *Opening Book*, *Move Ordering*, *Iterative deepening* with *Principal Variation First* search, *Quiescence Search*, *Adaptive Depth*, ...[1]

[1] Note that those extensions are not the ones I originally planned. I chose those because I found them more interesting for *Suicide Chess*.

3.2 Type safety

First of all, in order to ensure type safety in my program, I created three high level types (classes) :

Piece Defines constants for representing each piece. A piece can be defined (constructed) using its *piece number* or using its *piece type* and colour.

Square Defines a square on the board and provides an abstraction for the *bitboard squares* (2.4.2). It is defined as a file and a rank. It can be constructed from another *Square*, from a *String* or from a *bitboard square number*. A `NotAValidSquare` exception is thrown when trying to create illegal squares (like 'f9').

Move Defines a representation for moves. It makes use of *Piece* and *Square*. Every instance of a *Move* is necessarily a valid move[2], and an exception `NotAValidMove` is thrown otherwise. Eventually a *Move* holds special flags saying if the move is a capture, is an *en passant* capture, sets an *Square* for *en passant* captures or is a promotion (in which case it also stores the promotion piece).

3.3 Class *Board*

The class *Board* I defined has three objectives:

1. Storing the board state (including for example the current player),
2. Storing the board value (static evaluation)[3] and the number of moves that have been made (for draw detection),
3. Applying *Moves* and update the two points above.

3.3.1 Datafields

I stored the bitboards as an array

```
1 protected long bitBoards[];
```

This array has dimensionality 14, holding two "extra" *bitboards*: *White Pieces* and *Black Pieces*. That will avoid many boolean operations, because especially in move generation, we often need to know if a square is occupied by a piece of the opposite colour (a capture) or a piece of the same colour (blocking sliding pieces).

[2]Valid move does not necessary mean legal. Moving a pawn from `e2` to `e5` is possible, but not legal.

[3]More about the static evaluation later.

The array is indexed using the *Piece number* constants defined in the class Piece. For example `bitBoard[Piece.WHITE_PAWN];` would return the *White Pawn bitboard* and `bitBoard[Piece.BLACK];` would return the *Black Pieces bitboard*.
I also used a `private int[] numberOfPieces;` to store the number of each pieces. Though it is redundant with information contained in the *bitboards*, it allows more efficient evaluation of a board position.
The board also holds a special *Square* datafield for *en passant* captures and two move clocks to be able to tell when there is a draw.

I needed a way to **access the *i*th bit of a number**, so I created the following masks[4]:

Bit masks

```
protected static long mapSquaresToBits[];

static {
    mapSquaresToBits = new long[NB_OF_SQUARES];
    for(int i=0; i<NB_OF_SQUARES; i++) {
        mapSquaresToBits[i] = (1L << i);
    }
}
```

Eventually, I needed a way to **get the bit number corresponding to a Square object**:

Mapping from Square to bit number

```
/**
 * Converts a Square to a number representing a bit
 */
public static int squareToBitBoardSquare(Square square) {
    return square.getFileNb() -1
            + (square.getRank()-1)*NB_OF_FILES;
}
```

We can see that for example 'a1' will be mapped to 0, the less significant bit of a *bitboard*.

[4]Note that the masks are defined as a static variable. There is no need to have one instance of those for each instance of a *Board*: it would be wasting memory, but also time.

3.3.2 Functions

From all those definitions, the implementation of most of the *Board class* functions is often very condensed, like testing if a square is empty (using the bit masks defined above)

Class Board: isEmpty function

```
/**
 * Returns a boolean telling if a Piece is on a Square.
 */
public boolean isEmpty(Square square, Piece piece) {
    //find the mask for the corresponding bit
    long mask =
        mapSquaresToBits[squareToBitBoardSquare(square)];

    //AND the mask and the right piece bitboard
    if ((bitBoards[piece.getPieceNumber()] & mask) == 0) {
        return true;
    } else {
        return false;
    }
}
```

or adding a piece on the board (using the boolean operation `OR`)

Class Board: addPiece function

```
protected void addPiece(Square square, Piece piece) {
    //add Piece to corresponding bitboard.
    bitBoards[piece.getPieceNumber()]
        |= mapSquaresToBits[squareToBitBoardSquare(square)];

    //update the bitboard of all pieces of that colour
    bitBoards[piece.getColor()]
        |= mapSquaresToBits[squareToBitBoardSquare(square)];

    //update the number of pieces
    numberOfPieces[piece.getPieceNumber()] += 1;
    numberOfPieces[piece.getColor()] += 1;
}
```

Note that `addPiece` is protected. It is not `public` because for security reasons, only an instance of a *Move* applied to the function `doMove` can be used to add and remove pieces. It is not `private` because I used a inherited version of that class to do testing and debugging.

The main function in the *Board* class is `doMove`

doMove(Move) source code

```
public void doMove(Move move) throws NoPieceOnSquare, NotAValidSquare {
    this.halfmoveClock++;

    if (move.isCaptureMove()) {
        if (move.isEnPassant()) {
            removePiece(move.getEnPassantSquare(), move.getCapturedPiece());
        } else {
            removePiece(move.toSquare(), move.getCapturedPiece());
        }
        //reinitialise halfmoveClock since there has been a capture
        this.halfmoveClock=0;
    }

    removePiece(move.fromSquare(), move.getMovingPiece());
    if (move.isPromotionMove()) {
        addPiece(move.toSquare(), move.getPromotionPiece());
    } else {
        addPiece(move.toSquare(), move.getMovingPiece());
    }

    if(move.getMovingPiece().getPieceNumber()==Piece.PAWN) {
        this.halfmoveClock = 0; //a pawn has been moved
    }

    this.enPassant=false;
    if (move.enablesEnPassant()) {
        this.enPassant=true;
        this.enPassantSquare=move.getEnPassantSquare();
    }

    if(this.currentPlayer==Piece.WHITE) {
        this.currentPlayer=Piece.BLACK;
    } else {
        this.fullmoveNumber++; //black has just played
        this.currentPlayer=Piece.WHITE;
    }

    evaluateNewBoardValue(move);
}
```

3.3.3 Constructors

A *Board* can be constructed in three different way.

- With no argument, which creates a *Board* in the standard chess set-up,
- with another *Board* as an argument, which does a copy of that board,
- using a FEN[5] *String*.

The third constructor is probably worth exploration: taking a string of the form[6]:

```
FEN notation
rank8/rank7/.../rank1 currentPlayer - enPassantSquare
      halfMoveClock fullMoveNumber
```

I needed to parse it into a valid *Board* instance. For example,

```
new Board("rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w - - 0 1")
```

and `new Board()` must give the same result: the standard board set-up.

To do this parsing, I used *Java*'s method

```
split function in class String
String[] split(String regex)
```

that splits a *String* in an *String Array* using the separator defined by the regular expression `regex`. That function is available since *Java 1.4*.
The regular expression I used is `"/|\\s"` which splits the *String* on every space or stroke character. I then parse every element in the array (using `String.equals()`, `Integer.parseInt()` and `Character.isDigit()`) throwing an exception I defined (`UnableToParseFENStringException`) if any step failed.

[5] Forsyth-Edwards Notation: more details on
`http://en.wikipedia.org/wiki/Forsyth-Edwards_Notation`

[6] In *Chess*, the `-` represent the Castling status, but Castling is not allowed in *Suicide Chess*

3.3.4 Evaluation function

Since the evaluation function is very closely linked to a *Board*, I decided to implement it within the *Board* class.
A positive value for the `boardValue` means that the position is evaluated in favour of White.

First Version: number of pieces

In the early days of my program, the evaluation function was really simple, but probably corresponding pretty well to what a beginner would do.

First evaluation function
```
1 boardValue = numberOfPieces[Piece.BLACK] - numberOfPieces[Piece.WHITE];
```

Since the aim of the game is to get rid of all our pieces before the opponent (ignoring stalemate), it makes sense to just take the difference of the number of pieces on each side as a reference.

Second Version: piece value

In later versions of the program I decided to add a different value to each piece. Those value were first stored in the *Piece* class but then **I decided to create an external configuration file (a textfile) to be able to change those values and experiment without having to recompile the program**.

The problem, as I presented it in the *Preparation* is that the pieces tend to have different values depending on what phase of the game is being played. I had thus the idea of **using two evaluation functions**, one for the endgame and one for the midgame. To detect what phase of the game was being played, I decided that I would look at the number of pieces the opponent has as well as the number of pawns (more on that point in the *Evaluation* chapter). Those two values are stored in the configuration file as well.

The board value can be **updated in constant time** using this method: whenever a piece is captured, subtract its value.

Third Version: mobility

I read that mobility was a very important criteria in *Suicide Chess*: for example it is always good to have more than one choice when you have to capture.

I used two types of mobility, the **primary mobility** (how many moves can I play?) and the **secondary mobility** (how many moves could I play if I didn't have to capture the opponent?). Because of the capture rule, without the secondary mobility, the value of the mobility was varying too

much to be used as a reliable indicator of the strength of a position. I think it is more logical to take into account the mobility of other pieces even if only a few moves are allowed.

Just like before I separated each of those values in two, one for the midgame and one for the endgame, storing those four values in the configuration file.

Third evaluation function

```
1  //midgame
2  if (Rules.getLegalMovesCapture().size()!=0) {
3      boardValue+= Rules.getLegalMovesCapture().size()
4                   *ConfigFile.getPrimaryMobilityValueMidgame()
5              +Rules.getLegalMovesNonCapture().size()
6                   *ConfigFile.getScondaryMobilityValueMidgame();
7  } else {
8      boardValue+= Rules.getLegalMovesNonCapture().size()
9                   *ConfigFile.getPrimaryMobilityValueMidgame();
10 }
```

The main problem with mobility is that updating the board value takes a lot more time than with the previous evaluation function: when a piece moves, it changes the mobility of other pieces as well. Having to process the mobility of each piece every time is not efficient, as opposed to a constant time complexity in the previous evaluation function, and unfortunately did not lead to better play as I will see in the *evaluation* section.

Fourth Version: square value

To keep on improving my evaluation function I decided to analyse *Sjeng*'s source code: *Sjeng*'s evaluation function is very complex and relies a lot on pawn positioning (doubled pawns, passed pawns, ...). I didn't have time to implement all that, but *Sjeng* has something else I didn't implement: *square weights*. Just like controlling the centre is important in *Chess*, it is also quite important in *Suicide Chess*.

What *Sjeng* does is multiply the value of each piece by the weight of the square it is on. I adapted my program to add this feature. Once again the *square weights* can be modified from the configuration file.

Unlike *mobility*, that change does not lead to an strong overhead in calculating the *board value*. There is a slight change in that the evaluation function is updated every time a piece moves and not every time there is a capture, but updating the value of the board is a matter of subtracting the weight of the square the pieces comes from, and adding the weight of the square the pieces goes to. Of course, an extra subtraction is needed for captures, and promotions are also to be taken separately, but **unlike *mobility* those operations are done in constant time**.

Problem library

To be able to **test if my program was improving** after each change in the endgame evaluation function, I created a database of 23 problems (study positions) found on the Internet (mainly the easy and medium problems from GOLDOVSKI's Losing Chess page[8]). That technique is called *Regression Testing*.
Those problems are stored in an external textfile, in *FEN* standard notation. I chose the convention that *White* would be the winner, having to adapt the original problem from time to time. You can find the list of problems in the Appendix F.

My program is able to either load any problem from the default problem file (or any other file formatted in the same manner) and play from that position, or to play all problems in turn.

Since I do not have the solutions of the problems, as they are not available on the Internet and I haven't solved them all myself, I decided that those problems would be more "testing positions" than problems. By that I mean that the computer would play both players, and report every position for which *White* didn't win in the end. It is quite different from problem solving, **but it demonstrates that the computer can take advantage of a winning position**.

3.4 Rules and Move Generation

3.4.1 Pieces except Pawns

For each piece except the pawns, I used precalculated moves: for every possible position of a *Queen*, a *Rook*, a *Bishop*, a *King* and a *Knight*[7] on the board (64 positions), I have an array of all the *Squares* they can go to. This takes a lot of memory (about 5000 integer values), and is surely less efficient than if I had used *rotated bitboards*, but it is still much more efficient than having to process all the legal squares each time.
I didn't write that database of moves myself, because it does not involve any programming skills. Instead I used regular expressions to adapt François-Dominic LARAMÉE[10]'s code. That was possible because the pieces move in the same way in Chess and *Suicide Chess*.

The database of moves is stored as a `private static int[][][][] movesAllowed;`:

1. The first dimension is indexed using the **Piece number**. That first dimension was not in the original database, it is my own idea to avoid duplicated code.

   Example
   ```
   movesAllowed[Piece.QUEEN]
   ```

2. The second dimension is the **Square number** (bit number). For example, the moves allowed when the Queen is in 'a1' are given by:

   Example
   ```
   movesAllowed[Piece.QUEEN][0]
   ```

3. The third dimension is **the ray**, ie. one "sliding direction": one piece is not allowed to jump over other pieces[8], and thus should stop before any piece of the same colour or on any other piece of the opposite colour. For example there are only 3 rays in `movesAllowed[Piece.QUEEN][0]`, the first one being the file `a`, the second one being the diagonal `a1-h8` and the third one being the rank `1`.

4. The fourth dimension are the **possible destination Squares in one ray**. As the index increase, they move further away from the piece.

[7] A queen in fact is just the same as a *Rook* and a *Bishop*, but this would lead to the *Queen* being a special case.
[8] The *Knight* is a special case, having only one move in each ray.

To know all the moves from one square for any piece (except the pawn), I have the following code :

Move generation (pseudo code)

```
given a pieceNumber and a squareNumber;
for each ray
    for each destinationSquare in that ray
        if there is a piece of the same colour on the destination square
            break; //don't slide further in the ray

        create a new Move(fromSquare, destinationSquare);

        if Move is a capture
            add to list of legal capture moves;
            break; //don't go further in the ray
        else
            add to list of legal non capture moves;
    //endfor destinationSquare
//endfor ray
```

3.4.2 Pawn

I implemented the move generation for the pawn by a series of `if` statements.

- If the pawn is on the rank before last, enable promotion for this pawn's move (every move will be converted to 5 moves: promotion to queen, king, rook, bishop and knight).
- If the pawn is on any file but the first, see if the pawn can capture left (pieces or *en passant* Square if the corresponding flag is set in the current *Board* class).
- If the pawn is on any file but the last, see if the pawn can capture right (ibid.).
- If no capture is possible allow moving one square ahead (if no piece is blocking),
- and if the pawn hasn't moved yet (if it is on its starting rank) allow moving two squares ahead (if no piece is blocking). This move sets a special "en passant" flag.

3.4.3 Storing legal moves

I decided to store legal moves in a *List* and not an *Array* as the number or legal moves can vary quite a lot. *Java* has two structures, the *Vector* and the *ArrayList*. I decided to use the second as I read that *ArrayList* is faster and more efficient (the overhead in *Vector* is due to the fact that *Vector* is synchronised, but I don't need that in my program).

I also decided to use two *ArrayList*s rather than one: one to store the list of non capture moves and one for the list of capture moves. This representation is efficient because of the capture rule in *Suicide Chess*: I can instantly access the list of compulsory captures if any. Having only one list would require scanning through all the legal moves which is highly inefficient.

Legal Moves

```
1     private static ArrayList<Move> legalMovesNonCapture;
2     private static ArrayList<Move> legalMovesCapture;
```

3.5 *XBoard* protocol

I could not implement the *XBoard* protocol earlier because my program wasn't able to play. At this point I had it play picking up a random move in the list of all legal moves.

I created a class called `XBoardProtocol` to implement the protocol. Given a command, this class returns a command code. In the core of my program I then have a `switch-case` with the corresponding action for each command. (See Appendix E for a detailed list of implemented commands.)

The only difficulty in that part of the implementation was to understand what signals *XBoard* would send to my program, and in what order (cf. Figure 2.6 in the *Preparation* chapter).

3.6 Searching

3.6.1 MinMax

"Make it work before you make it work fast." – Jon Bentley

The most natural way to implement *MinMax* is to use recursion and that is the solution I chose.

MinMax pseudo code

```
1
2  private bestMovesList;
3
4  private MinMax(Board bitboard, int currentDepth) {
5      if (maximumDepth has been reached)
6          return the boardValue;
7
8      generate the listOfLegalMoves;
9      if (the listOfLegalMoves is empty)
10         if (player is black)
11             return 'blackWins' value;
12         else
13             return 'whiteWins' value;
14
15     if (player is black) //trying to minimise
16         bestScoreSoFar = maximum possible score; //whiteWins + 1
17
18         for (each move in listOfLegalMoves)
19             currentScore = MinMax(bitboard.doMove(move),currentDepth+1);
20             if (currentScore <= bestScoreSoFar)
21                 if (currentScore < bestScoreSoFar)
22                     bestScoreSoFar = currentScore;
23                     if (currentDepth is 0)
24                         clear bestMovesList;
25                 if (currentDepth is 0)
26                     add current move to bestMovesList;
27
28     else //currentPlayer is White, trying to maximise
29         bestScoreSoFar = minimum possible score; //blackWins - 1
30         bestMovesList = empty;
31
32         for (each move in listOfLegalMoves)
33             currentScore = MinMax(bitboard.doMove(move),currentDepth+1);
34             if (currentScore >= bestScoreSoFar)
35                 if (currentScore > bestScoreSoFar)
36                     bestScoreSoFar = currentScore;
37                     if (currentDepth is 0)
38                         clear bestMovesList;
39                 if (currentDepth is 0)
40                     add current move to bestMovesList;
41
42     return bestScoreSoFar;
43 }
```

This function is private and is accessed using a global function. Given the list of best moves (`bestMovesList`) processed by the *MinMax* function, that global function returns one move from that list randomly. I chose to select a random move because I didn't want my computer to play the same game every time it gets to a situation when more than one move have the same value.
This implementation is not conventional. In a classic implementation, the *MinMax* would return only one move, but since I wanted to focus my efforts on non game-specific techniques, my static evaluation function is fairly crude and, as a consequence, many different positions evaluate to exactly the same value.
Though it is not conventional, it is not less efficient, and that is why I adopted that method.

3.6.2 Alpha-Beta pruning

To program *Alpha-beta pruning* I needed to be able to return more than one value: I wanted to know not only the value of alpha (or beta, depending on the current depth), but also the *real* value of a branch.
Knowing just the value of *alpha* or *beta* is not enough: if a branch does not improve the value of *alpha* (or *beta* depending on the branch), it can either mean that this branch has a move with the same value than a move previously analysed – in which case this branch should be considered as a good variation as well – or that the branch has not even reached alpha – in which case it can be safely ignored.

I created an inner class in my class `ComputerPlayer` to be able to return both values. I called it `ReturnWrapper`. In the following pseudo code, I will represent those two values as a couple `{a,b}`.

Alpha-beta pseudo code

```
1  private bestMovesList;
2
3  private AlphaBeta(Board bitboard, int currentDepth, int alpha, int beta) {
4      if (maximumDepth has been reached)
5          return {boardValue, boardValue};
6
7      generate the listOfLegalMoves;
8      if (the listOfLegalMoves is empty)
9          if (player is black)
10             return {'blackWins' value, 'blackWins' value};
11         else
12             return {'whiteWins' value, 'whiteWins' value};
13
14     if (player is black) //trying to minimise
15         bestScoreSoFar = maximum possible score; //whiteWins + 1
16
17         for (each move in listOfLegalMoves)
18             {currentScore,currentAlphaBeta}
```

```
19                = AlphaBeta(boardCopy,currentDepth+1,minimum score,beta);
20
21            //calculating new value of beta
22            if (currentAlphaBeta < beta)
23                beta = currentAlphaBeta;
24
25            //calculating branch value
26            if (currentScore <= bestScoreSoFar) {
27                if (currentScore < bestScoreSoFar) {
28                    bestScoreSoFar=currentScore;
29                    if (currentDepth is 0) {
30                        clear bestMovesList;
31                add current move to bestMovesList;
32
33            if(beta<alpha)
34                return (alpha,bestScoreSoFar); //pruning
35            return (beta,bestScoreSoFar);
36
37    else //if (player is white) //trying to maximise
38        bestScoreSoFar = minimum possible score; //blackWins - 1
39
40        for (each move in listOfLegalMoves)
41            {currentScore,currentAlphaBeta}
42                = AlphaBeta(boardCopy,currentDepth+1,minimum score,beta);
43
44            //calculating new value of alpha
45            if (currentAlphaBeta > alpha)
46                alpha = currentAlphaBeta;
47
48            //calculating branch value
49            if (currentScore >= bestScoreSoFar) {
50                if (currentScore > bestScoreSoFar) {
51                    bestScoreSoFar=currentScore;
52                    if (currentDepth is 0) {
53                        clear bestMovesList;
54                add current move to bestMovesList;
55
56            if(beta<alpha)
57                return (beta,bestScoreSoFar); //pruning
58            return (alpha,bestScoreSoFar);
```

3.6.3 Thinking output

After each call of *Alpha-Beta pruning* function, my program displays the best variation found as well as the time taken and the number of nodes examined, in the format required by *XBoard*: `depth evaluation time nodes variation`.
My program also displays that information each time it finds a better variation in the *Alpha-Beta pruning*. A typical output from my program is available in the Appendix D.

To be able to print the best variation, I needed to store the sequence

of moves that lead to some score. Since *Alpha-Beta pruning* is bottom-up, starting from the leaves and going up to the root, I could simply trace the best moves from the node up to the root in order to have the best variation in a branch.
I implemented this by adding a *String* to my `ReturnWrapper`. The current move just has to be concatenated at the beginning of the `String` for the variation to be displayed from the root down to the node.

3.6.4 *Iterative Deepening* and *Principal Variation First*

Iterative Deepening is programmed by calling the *Alpha-Beta pruning* function with increasing maximum depth. I chose to start from 2.

The main interest of *Iterative Deepening* is to be able to do *Principal variation first search*[9], ie. analysing the best variation found during the previous searches before analysing any other variations. This is justified by the fact that *Alpha-Beta pruning* is much more efficient if the best moves are analysed first.
In every step of *Iterative Deepening*, before starting an analysis to a greater depth, my program splits the principal variation *String* into a *Move Array*. I added a boolean in the arguments of the *alpha-beta pruning function* to tell if the program is currently analysing the principal variation (in which case the next move in the principal variation must be analysed first) or not (in which case any move can be analysed first).
The following source code replaces the beginning of the for loop in the *Alpha-beta pruning* pseudo code above

Principal Variation First pseudo code

```
1  if (inPrincipalVariation)
2      doMove principalVariation[currentDepth+1];
3      call alpha-beta with inPrincipalVariation = true;
4
5  for (each legal move)
6      if((inPrincipalVariation) and
7              (move equals principalVariation[currentDepth]))
8          break; //move already analysed
9      doMove move;
10     call alpha-beta with inPrincipalVariation = false;
```

[9] *Iterative Deepening* is also justified by transposition tables[11], but I didn't implement those

3.7 Extensions

3.7.1 Opening Book

In *Suicide Chess*, the opening phase is really important. **Out of the 20 possibles moves for white, 7 have been proved to lead to a defeat** as you can see on Cătălin FRÂNCU's [3, Suicide Chess Book Browser]. That is the main reason why I decided to add an *Opening Book* to my program. I created a class called `OpeningBook`.

All known openings are stored in a file, with one opening per line. Each move is written in *Algebraic Notation* and is separated from the others by a space. A typical line would be:

A line from the opening book

```
1 e2e3 e7e6 d1f3 f8a3 b2a3 d8g5 f3b7 g8g2 b7b8 g2h1
```

This representation is the same one *Sjeng* uses, and in fact, I used the same file as *Sjeng*.

When loading an opening book file, I parsed the file into a `private static ArrayList<String[]> book;` so that `book.get(i)[j];` returns the *j*th move of the *i*th variant (when the indexing values are defined).
I also used a boolean array that has the same size as the `ArrayList` above and is used to tell whether a variation can be played from the current position. After each move, the boolean corresponding to each variation that ran out of move, or each variation that does not correspond to that move that has been played is changed to `false`.

This representation does not allow transposition of opening books: moves have to be played exactly in the order written in the opening book. For example, if we have the two beginnings shown in Figure 3.1, that end up in the same position, only the first one would be played by my program as a known opening. It is important to notice that **transpositions are not as common as they are in Chess since capturing is compulsory**.

3.7.2 Quiescence search

In the *MinMax* search or *Alpha-Beta* search, the search depth is fixed and that can cause problems. Typically, in *Suicide Chess*, if one node of the search tree has only a few possible moves (typically a few captures), then those moves should be examined as they will change the material balance and could be dramatic for one player. This process can be repeated until the branching factor is more than some chosen value (a *quiet* position).
This extension has very little cost (since the branching factor is very small) but leads to a better evaluation since **it avoids the *horizon effect* where imminent material loss is hidden by the limited depth**.

1.	b2–b3	g7–g6
2.	c2–c4	

or

1.	c2–c4	g7–g6
2.	b2–b3	

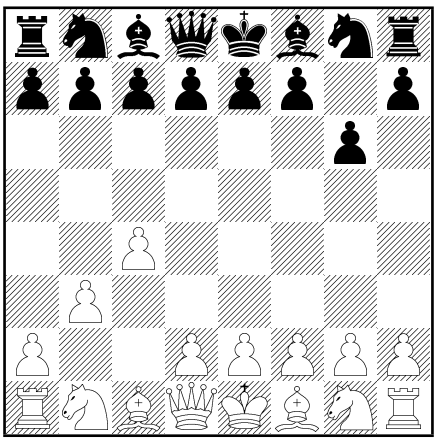


Figure 3.1: Two opening move sequences leading to the same result

I implemented *Quiescence Search* so that it can be enabled or disabled easily, as well as configured (before compilation). In particular, it is possible to decide the maximum branching factor (capture moves only) and the maximum depth (to avoid spending too much time looking at very complex situations).

Quiescence search configuration

```
1  /**
2   * Enable quiescence search.
3   */
4  public static final boolean QUIESCENCE_SEARCH = false;
5  /**
6   * If more than that many possibilities (branching factor),
7   * don't analyse further.
8   */
9  public static final int QUIESCENCE_LIMIT = 4;
10 /**
11  * Maximum number of Plies the computer will ever go to
12  */
13 public static final int MAX_QUIESCENCE_DEPTH = 8;
```

The only change I had to do in *Alpha-Beta pruning* was adding the following test: when the `currentDepth` equals the `currentMaxDepth`[10] (and

[10]The initial value of `currentMaxDepth` it the current depth in *Iterative Deepening*

is less than `MAX_QUIESCENCE_DEPTH`), I then check if this node is quiet, and otherwise call the *alpha-beta* function again with `currentMaxDepth+1`.

3.7.3 Adaptive Depth search

This is an idea that I had when implementing *Quiescence Search*: in *Suicide Chess*, there are many forced moves, with only one or two possibilities (because of the capturing rule). Compared to moves with a branching factor of 20 or more, those moves are going to be examined very quickly. I though that it then makes sense to search to one ply further in those branches.
I later discovered that there is a similar technique in Chess called "singular extensions" where certain forced moves are not counted in the ply count.

Also, if the branching factor from the root of the search tree is 1, then *adaptive search* would immediately return that move without analysing any further. This allow for instant response on compulsory moves and avoids wasting precious seconds in blitz games.

I decided to enable *Quiescence Search* and *Adaptive Depth Search* **only during the last step of *Iterative Deepening***, otherwise, every iteration in the *Iterative Deepening* process was very time consuming.

3.7.4 Move ordering

I implemented move ordering using Java's `Collections.sort(List list, Comparator c)`. The sorting algorithm is a modified *Merge Sort*, which is an efficient algorithm (order $n\log(n)$).

The `List` to be sorted is the `ArrayList<Move>`, but I had to implement a `Comparator`:

MoveCompare Class declaration

```
1 class MoveCompare implements Comparator<Move>
```

I implemented the constructor so that the order in which the comparator will order moves depends on which player is to play. `compare(MoveA, MoveB)` must return a positive number if *MoveA* is "bigger" than *MoveB*, and this leads to the function `compare(MoveA, MoveB)` being somehow unintuitive:

- `sort()` sorts elements from the "smallest" to the "biggest",
- if it is *White*'s turn, then the move with the highest score should be analysed first,
- so **moves with a high score need to be considered "smaller" than moves with a low score**.

Those points lead to the following implementation:

Implementation of compare

```
1  public int compare(Move one, Move another) {
2      Board oneBoardCopy = new Board(bitboard);
3      Board anotherBoardCopy = new Board(bitboard);
4
5      oneBoardCopy.doMove(one);
6      anotherBoardCopy.doMove(another);
7
8      //sortOrder is defined in the constructor
9      //sortOrder = +1 is white is to play
10     return sortOrder*
11         (anotherBoardCopy.getBoardValue()-oneBoardCopy.getBoardValue());
12 }
```

Chapter 4

Evaluation

4.1 Efficiency

JMP [2] is a profiler for Java that can be used to trace objects usage and method timings. I used it on a 4 ply search from a starting position and got the following statistics (Note that *JMP* slows the program considerably. The above values do not directly correspond to the real-time simulation, however the proportions are correct.)

Looking at the time spent in each method (Figure 4.1) we can see the the *Alpha-Beta* method is by far the most time consuming, which is logical due to its exponential complexity. The *Move generation* and *Evaluation function* come just after, which is also logical.
The only two unexpected methods here are `isEmpty` and `getPiece`, that I would have expected to be more efficient. On the other hand, as I wrote in the *Preparation*, I only have a 32 bit architecture and those two functions use 64 bit boolean operations.

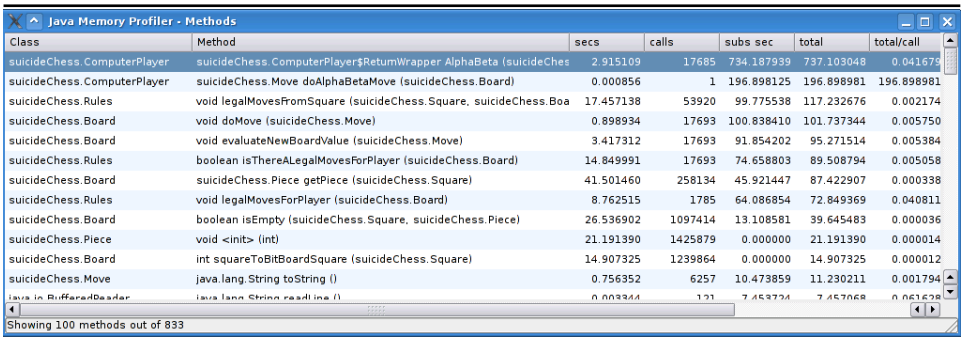


Figure 4.1: Method timings

4.2 Evaluation function

My evaluation function has many parameters. I will explain how I chose each one of them.

In order to decide which one of two different programs or versions is better I made them play sets of games against another. Modelling the outcome of each single game using a *Bernoulli Distribution*, I will justify in Appendix A how I drew conclusions from the results of those matches.

4.2.1 Evaluation function in the endgame

I read on [14, Pion.ch] that in the endgame: `R>K=Q=B>N>P` (where N is the Knight). I translated that to

```
PAWN    QUEEN   KING    BISHOP  KNIGHT  ROOK
-90     -40    -40      -40     -70     -10
```

in my configuration file. The rook has the lowest negative number because I want the program to try and get rid of other pieces first.

With those values, 11 problems in my list of 23 problems were solved when searching to depth 4, 15 when searching to depth 6.
I tried other values, for the *Queen-King-Bishop* value and the *Knight* value, but I didn't find any configuration respecting the order on [14, Pion.ch] that improved those statistics so I decided to keep the values above.

Also, the very first evaluation function that I programmed, which only took into account the number of pieces was terrible at solving problems: 9 were solved searching to depth 4 and only 11 when searching to depth 6 so I decided not to use it either.
To establish that it wasn't good enough, I played 50 games to ply 6 between that version and the version above. Results were concluding (see Appendix A) since my first evaluation function lost 30 games and won 19.

Eventually, although my program has the ability to use *square weights* and *mobility* in the endgame, I opted against using these features in the endgame since they did not improve the results significantly.

4.2.2 When does the endgame start?

On [14, Pion.ch] again, Fabrice Liardet wrote that the endgame starts when there are only a few pawns left on the board.

In the beginning, to test whether a position was an endgame position, I only took into consideration the number of pawns, but this is too weak. My program would sometimes lose all its pieces and be left with only 4 or 5 pawns: those positions had to be recognised as endgame positions. I thus decided to added a second parameter: the number of pieces. (See Appendix B for an example of how this second parameter is important.)

I chose that 3 pawn or less on either side or 6 pieces or less on either side is an endgame position. 4 pawns seemed too early to me, and with 2 pawns only I found that my program often switched to the endgame evaluation function too late.
To justify this choice, I used "self-play", playing matches of my program against itself[1] changing only the number of pawns and pieces before switching to endgame mode.

- The version that switched to endgame when there are less than 2 pawns or 4 pieces on one side lost 40-60 (and no draw) against the version that switches when there are less than 3 pawns or 6 pieces.
- The version that switched to endgame when there are less that 4 pawns or 8 pieces on one side lost 46-53 (and one draw) against the version that switches when there are less than 3 pawns or 6 pieces.

One could argue that the result of the second match is quite balanced, but common sense had me decide that a position with 8 pieces on both sides of the board should not be considered as an endgame position.

4.2.3 Evaluation function in the midgame

Choosing the right evaluation function in the midgame was much harder than the one in the endgame because I didn't have any reference talking about relative strength of each piece.
Moreover since I am not a very good *Suicide Chess* player, it is hard for me to tell whether a choice made by one evaluation function will end up being good in the endgame or bad.
Eventually I didn't have test positions to do automatic testing like in the endgame.

To try and overcome that problem, I did matches between different versions of the evaluation function. The results of those matches are shown in Figure 4.2. The versions are as follows: (apart from the first one, every function uses the evaluation function described earlier for the endgame)

1. No evaluation function: the program chooses (random choice) any valid move.
2. The program uses the number of pieces on each side. I shall refer later to this version as the *Kamikaze Version* because it tries to get rid of all pieces from the very beginning.
3. This version is the same as the *Kamikaze Version*, but with positive piece values in the midgame. The exact values will be presented later.

[1] I chose to use the *Cheeseparing Version* of the program as it is the one with two very distinct strategies, cf. later.

Test match: 2 against 1	50-0-0
Test match: 2 against 2	24-25-1
3 against 2	11-39-1
4 against 2	21-28-1

Figure 4.2: Result of matches of different version of my program. The result are written as: *games won by the program in the first program - games won by the second program - draw*

4. That version is the same as the previous one but having different square weights in the midgame. The square weights are available in the full configuration file in Appendix C. Those square weight values come from *Sjeng*'s source code. I shall refer to this version later as the *Cheeseparing Version* because it tries not to giveaway pieces in the beginning.

All matches were done with *Alpha-Beta Pruning* to 6 ply, *Principal Variation First Search* and the *Opening Book*, but no other extension.

The two test matches proves that:

- my evaluation function plays much better than a monkey who just knows the rules,
- when doing matches between programs of the same level, I get balanced results.

From the two other matches I can conclude that the *Kamikaze* version is the best of all. In all further tests I used either that *Kamikaze* version or the *Cheeseparing* Version. I wanted to keep at least two because they have very different strategies in the midgame which makes it interesting for further tests.

4.2.4 A closer look at the *Cheeseparing Version*

The values I used for this version are highly experimental. I analysed games played by my program against the *Kamikaze Version* or against *Sjeng*, and modified the values in response to bad moves my program made. The values I chose are:

```
PAWN    QUEEN   KING    BISHOP  KNIGHT  ROOK
10      90      70      -10     30      50
```

For example, if I ended up with a negative value for the bishop since with any positive value, my program would often loose because of one bishop.

Also I didn't put less than -10 otherwise my program was trying too hard to get rid of his bishop and missed other more important moves. Similarly, pawns cannot have a higher value otherwise my program would often have too many pawns in the endgame and they have a very bad mobility.

I am sure that I could have found better values and I would have appreciated the help of a professional player in choosing those values because I do not know enough theory to justify them. Unfortunately I don't know any good *Suicide Chess* player. I tried to contact Fabrice LIARDET with the address available on his website[14], but I received no answer.

4.2.5 Versions with mobility

I decided to give up the version with mobility because as I said in the implementation, calculating the mobility is much slower than just having piece values and square weights. As an example, from a starting position, searching to depth 6, a version without mobility finds a move in 30seconds whereas it takes 4 minutes for the version with!
The inefficiency comes from the fact that calculating the mobility involves running the move generation routine on every leaf node (to be able to get the evaluation value) when without it, it isn't necessary.

[10, GameDev.net] explains that for *Chess* mobility is rarely used as is: it would more likely penalise Knights near the edge and give a strong weight to Rooks on open files than simply count the number of moves available on each side. Probably similar arguments could be found for *Suicide Chess*, and it would definitely make the evaluation faster, but I didn't find any.

4.3 Search Speed

4.3.1 MinMax vs Alpha-Beta

The table Figure 4.3 summarises the number of nodes that are analysed from a starting position, a typical quiet midgame position, typical explosive[2] middle position, and an endgame position (Figure 4.4). The *Alpha-Beta pruning* has no search extension. The value next to the number of nodes is the ratio (rounded to the nearest integer) of nodes explored to that depth compared to a search of one ply less. This can be seen as an average branching factor at that depth.

This table proves that *alpha-beta pruning* is indeed a very good improvement, especially when taking into account the fact that the result from the *alpha-beta pruning* and the *minmax search* are the same. Except in situations where the branching factor is very low, *Alpha-Beta pruning* almost always divides this factor by two.

[2]By *explosive* I mean that there are forced captures on both sides.

	Plies	Opening		Quiet		Explosive		Endgame	
MM	2	421		462		9		106	
AB	2	421		462		9		106	
MM	3	8,488	20	8,249	18	65	7	2,163	11
AB	3	4,659	11	6,181	13	65	7	169	**1**
MM	4	161,787	19	107,576	13	224	3	15,474	7
AB	4	55,696	12	40,676	7	208	3	1,479	**9**
MM	5	2,894,459	18	1,736,722	16	816	4	157,336	10
AB	5	382,243	7	565,231	14	467	2	6,346	4
MM	6	49,158,621	17	22,312,916	13	2,334	3	1,095,856	7
AB	6	5,117,764	13	2,448,076	4	1,237	3	21,717	3
MM	7	N/A		N/A		8,572	4	11,287,991	10
AB	7	N/A		N/A		1,958	2	151,187	7
MM	8	N/A		N/A		41,732	5	N/A	
AB	8	N/A		N/A		6,501	3	N/A	

Figure 4.3: Comparison of the number of nodes examined by the *MinMax* algorithm (MM) and the *Alpha-Beta pruning* algorithm (AB) in typical *Suicide Chess* situations

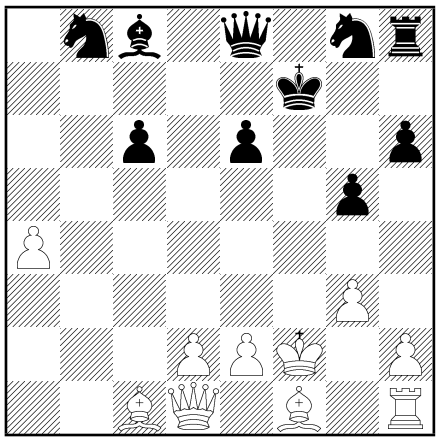
4.3.2 *Alpha-Beta* efficiency and move ordering

Now that we have seen that *Alpha-Beta* is indeed more efficient, let's look at the time taken by my program to analyse all those nodes. For this section I used the *Kamikaze version*.
In the table Figure 4.5 the times are give in centiseconds. That is what *XBoard* asks for in the thinking output formatting[16], and those values come straight from my program's output.

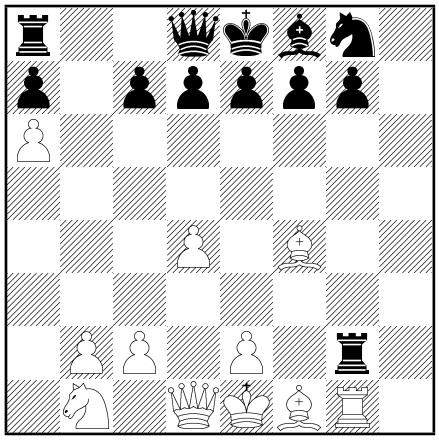
We can see that *principal variation first search* gives much better results than *Alpha-Beta* pruning. The time "lost" in doing *iterative deepening* is negligible compared to the advantage of examining less nodes. We can see that in explosive situations and in the endgame it is particularly efficient, which comes straight from the fact that only a few sequences of moves turn out to be good.
On the other hand, *Move Ordering* is incredibly inefficient (in time) and does not reduce the number or nodes examined much. This is due to the fact that in the *Kamikaze Version*, the *evaluation value* does not change for as long as there are no captures.

4.4 Problem solving (Search extension efficiency)

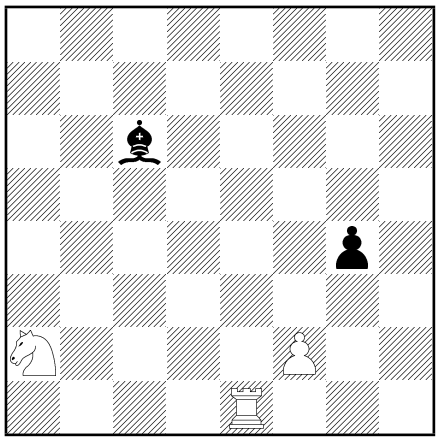
As I wrote before, I implemented a collection of 23 problems.
I ran various versions of my program on those problem to see which version was the best. The results are in the table Figure 4.6. (*SuShi* is the name I



(a) A typical quiet midgame position



(b) A typical explosive midgame position



(c) A typical endgame position

Figure 4.4: Some test positions

Plies		Opening		Quiet		Explosive		Endgame	
AB	2	421	0	462	3	9	0	106	0
MO	2	421	14	462	4	9	0	106	0
PV	2	421	1	462	3	9		106	0
AB	3	4,659	11	6,181	9	65	0	169	0
MO	3	4,659	23	6,181	15	65	0	169	4
PV	3	5,080	10	6,643	9	74	0	231	0
AB	4	55,696	39	40,676	44	208	2	1,479	5
MO	4	55,696	128	40,670	119	203	3	965	6
PV	4	59,711	39	47,036	42	123	0	786	3
AB	5	382,243	465	565,231	439	467	2	6,346	10
MO	5	382,243	1943	565,179	1263	461	3	5,430	31
PV	5	436,417	439	622,965	440	234	0	3,574	6
AB	6	5,117,764	4150	2,448,076	2893	1,237	5	21,717	24
MO	6	5,117,764	15318	2,444,289	9338	1,161	6	18,134	79
PV	6	5,021,235	3586	3,063,028	3084	682	2	15,339	16
AB	7	N/A		N/A		1,958	4	151,187	209
MO	7	N/A		N/A		1,702	6	137,521	891
PV	7	N/A		N/A		1,885	3	95,068	104
AB	8	N/A		N/A		6,501	13	N/A	
MO	8	N/A		N/A		5,145	24	N/A	
PV	8	N/A		N/A		5,337	10	N/A	

Figure 4.5: Time (in centiseconds) to generate a move. Comparison of *Alpha-Beta pruning* (AB), *Move Ordering* (MO), and *Principal Variation First Search* (PV)

Version	Unsolved	
Sjeng	2	
	4 Plies	6 Plies
SuShi	12	8
SuShi with *Quiescence* search	12	6
SuShi with + *Adaptive Depth* search	10	5
SuShi with *Quiescence* and *Adaptive Depth* search	8	6

Figure 4.6: Efficiency in problem solving. The problems are available in *FEN* notation in Appendix F

gave to my program).
Adaptive Depth search extends the search each time it reaches a node with 2 or less branches. *Quiescence Search* extends the final node when it has a branching factor or 4 or less but never goes to more than an 8 plies search.

This table proves that *Quiescence Search* and *Adaptive Depth* are indeed great improvements. When both are combined, the program becomes as good as a version that searches two plies further.

4.5 Matches against *Sjeng* and *KKF*

4.5.1 Preliminary comments

I tried to play matches with *Sjeng* against *KKFChess* but *Sjeng* kept crashing after the same three moves from *KKFChess*.

As I wrote before, *XBoard* cannot give different time control to the two programs in matches. **I thus had to have the same settings for both programs in a match**.
Since *KKFChess* implements neither *time control* nor *depth limit*, I could not experiment as much as I did with *Sjeng*, but I wanted to have at least one other program in my results.
Sjeng is amazingly good at finding killer moves. Most of the time it plays every move very quickly, but then suddenly stops, thinks for a few seconds, and this generally means that it has found a winning combination. To avoid those pauses, I tried to limit *Sjeng*'s search depth using *XBoard*'s commands, but *Sjeng* doesn't seem to have implemented them (the variation *Sjeng* displayed were longer than the required depth). Fortunately, *Sjeng* implements time control: when I did matches in less that 30 seconds per game, *Sjeng* didn't do those pauses anymore and my program managed to beat it (Figure 4.8).

Match	Time control	Result
vs *KKFChess*		
Kamikaze Version, 6 ply	5 min	8-42-0
Cheeseparing Version, 6 ply + *Quiescence Search* and *Adaptive Depth*	5 min	15-34-1
Kamikaze Version, 6 ply + *Quiescence Search* and *Adaptive Depth*	5 min	23-26-1
vs *Sjeng*		
Kamikaze Version, 6 ply	5 min	0-50-0
Kamikaze Version, 4 ply	1 min	0-50-0
Kamikaze Version, 4 ply	30 seconds	**55-45-0**
Kamikaze Version, 4 ply	10 seconds	**107-92-1**

Figure 4.7: Result of matches between different versions of my program and *Sjeng* or *KKF*

4.5.2 Results

The results of the matches are available in the table Figure 4.7.

As you can see from the table, my program, with search extensions is just a bit poorer than *KKFChess*. You can also see how the search extensions add a lot of strength to my program.

My program simply cannot rival against *Sjeng* when doing long games, but it gets very good when playing blitz games.
This victory against an established widely-regarded program in a long match shows that I have created a highly effective implementation of a *Suicide Chess* program by focusing on search techniques.

As little of my time was spent researching and improving the static evaluation function and as I am not a strong player myself, I believe the program can be made considerably stronger by adding more game-specific knowledge: perhaps with the assistance of a strong *Suicide Chess* player. However, this was not a priority for this project.

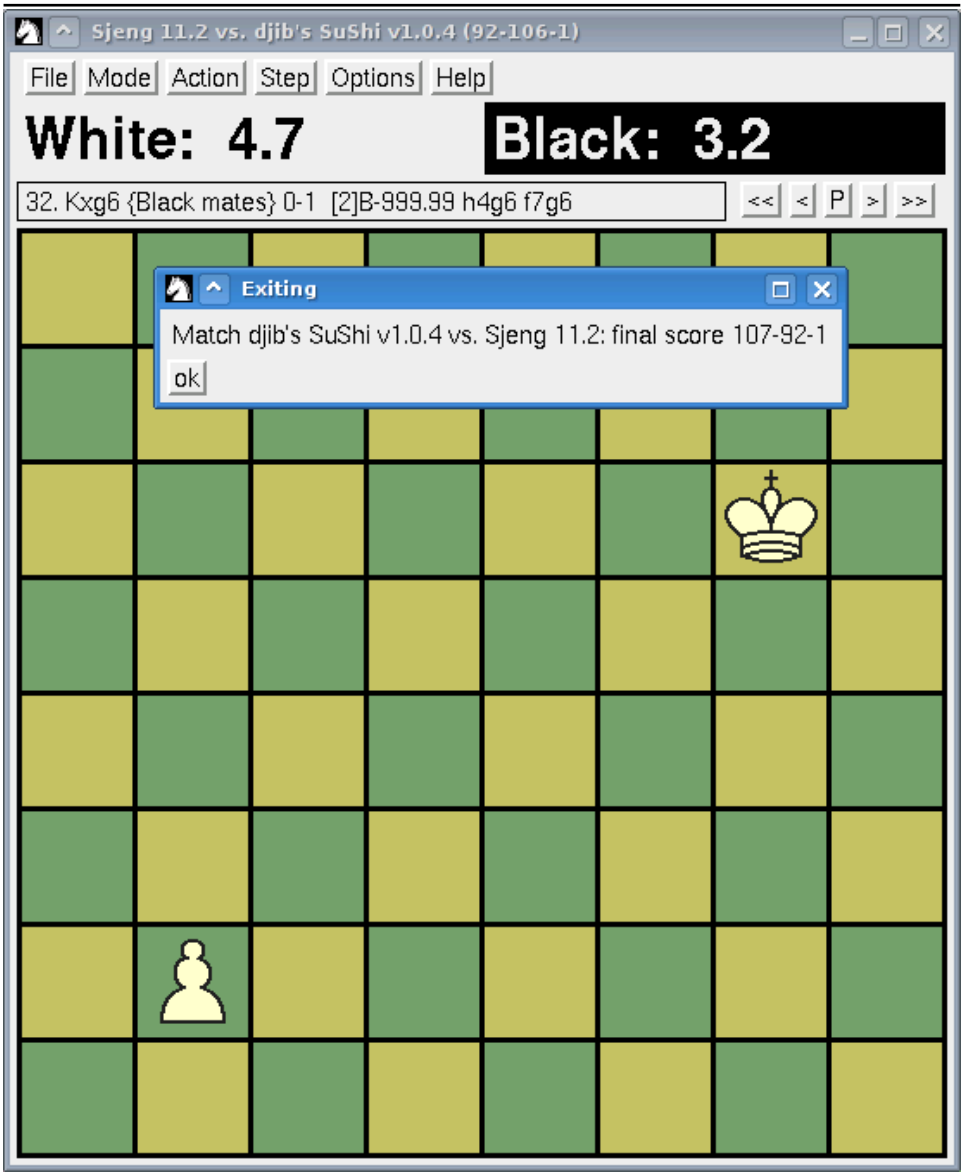


Figure 4.8: A 200 games 10 seconds per game match between my *Suicide Chess* program *djib's SuShi, Kamikaze Version* and *Sjeng*

Chapter 5

Conclusion

I am very satisfied with my project.

My program has won a 200 blitz games match against *Sjeng*, one of the best *Suicide Chess* programs. This proves that my implementation (including my choice of datastructures and search techniques) was very efficient. It is particularly the case considering that *Sjeng* is a compiled program developed in *C++* whereas my implementation was written in *Java*.
I am planning to further improve the efficiency of my program by adding *transposition tables*[11], implementing dynamic handling of time, and maybe rewriting the move generation to use *rotated bitboards*[6].

During my project I emphasised software engineering techniques rather that *Suicide Chess* specific heuristics. I improved my program's strength using many Artificial Intelligence techniques, mainly *Search Extensions* (*Adaptive Depth Search*, *Quiescence Search*), but also using other techniques such as an *Opening Book*. With the help of a professional player, I could greatly improve my program's strength by adding more sophistication to the evaluation function.

I am pleased that I managed to program a computer player that is now much better than I am. I also appreciate that most Artificial Intelligence techniques I used are not specific to *Suicide Chess* and can be adapted to most games of perfect information where the two players play in turns.

My program runs on most common *Operating Systems*, with a user-friendly interface, and its level can be adapted using a configuration file. I am planning to combine all those features to get my program running on my website using an interface I will design.

Finally, I am pleased to see I managed to organise and implement such a project. I learnt many Software Engineering techniques and I will feel more confident next time I have to start a project from scratch. This experience has been a strong complement to the diploma course and I very much appreciate it.

Bibliography

[1] Chessvariants.org: Loosing chess.
http://www.chessvariants.org/diffobjective.dir/giveaway.html.

[2] Java memory profiler.
http://www.khelekore.org/jmp/.

[3] Cătălin Frâncu.
Suicide chess book browser.
http://catalin.francu.com/nilatac/book.php.

[4] ICGA.
ICGA: Losing Chess Information.
http://www.cs.unimaas.nl/icga/games/losingchess/.

[5] Andrew Frankk.
KKFChess.
http://freespace.virgin.net/andrew.fankk/.

[6] Colin Frayn.
Computer chess programming theory.
http://www.frayn.net/beowulf/theory.html.

[7] Stanislav Goldovski.
Losing chess strategy.
http://www.matf.bg.ac.yu/~andrew/suicide/StanGold/theory.htm.

[8] Stanislav Goldovski.
Stanislav Goldovski's Losing Chess page.
http://www.matf.bg.ac.yu/~andrew/suicide/StanGold/Index.htm.

[9] Robert Hyatt.
Chess program board representations.
http://www.cis.uab.edu/hyatt/boardrep.html.

[10] François-Dominic Laramée.
Gamedev.net – chess programming.
http://gamedev.net/reference/programming/features/chess1/.

[11] François-Dominic Laramée.
Gamedev.net – chess programming: Transposition tables.
http://www.gamedev.net/reference/programming/features/chess2/page4.asp.

[12] David Levy.
Computer Gamesmanship – The complete guide to creating an structuring games programs.
Century Publishing, 1983.

[13] Richard Lewis.
An introduction to suicide strategy.
http://www.matf.bg.ac.yu/~andrew/suicide/2004/en002.htm.

[14] Fabrice Liardet.
La page de "qui perd gagne".
http://www.pion.ch/.

[15] Fabrice Liardet.
Règles internationales du "qui perd gagne".
http://www.pion.ch/Losing/rules.html.

[16] Tim Mann.
Xboard chess engine communication protocol.
http://www.tim-mann.org/xboard/engine-intf.html.

[17] Gian-Carlo Pascutto.
Sjeng: a chess-and-variants playing program.
http://sjeng.org/indexold.html.

[18] Aaron Tay.
WinBoard vs UCI.
http://www.aarontay.per.sg/Winboard/uciwinboard.html.

Appendix A

Relevance of game statistics

The aim of Appendix A is to give an insight to the game statistics used to argue that one program is better than the other.

Given two programs **A** and **B**, I assume that the games form a *Bernoulli Distribution*:

$$P(\mathbf{A} \text{ wins}) = \hat{p} \qquad P(\mathbf{B} \text{ wins}) = \hat{q} = 1 - \hat{p} \tag{A.1}$$

Given a probablility α, I suppose that '**A** better than **B**' means:

$$P(\mathbf{A} \text{ wins against } \mathbf{B}) > \alpha \tag{A.2}$$

I want to know: **given n games, what is the probability c of having A better than B given observation A wins p matches.**

$$c = P\left(\mathbf{A} \text{ is better than } \mathbf{B} | P(\mathbf{A} \text{ wins}) = \hat{p} = \frac{p}{n}\right) \tag{A.3}$$

$$= \sum_{k=\lfloor \alpha n \rfloor + 1}^{n} \binom{n}{k} \left(\frac{p}{n}\right)^k \left(\frac{1-p}{n}\right)^{n-k} \tag{A.4}$$

In other words c is the sum of all probabilities of outcomes where **A** won more than $\alpha \cdot n$ games.

Choosing a value for α and given p based on the result of n matches, we can calculate c.

I programmed it in *Python*:

Game statistics

```
1 def factorial(n):
2         if n <= 1: return 1
3         return n*factorial(n-1)
4
5 def combinations(k,n):
6         return factorial(n)/(factorial(k)*factorial(n-k))
7
8
```

α	p	n	c	Explanation
Test cases				
0.5	100	100	1.00%	If a program has won all its games (p) then this program is surely better.
0.99	50	100	0.00%	If a program has to win all its games to be better but it only won half its games then it has no chance of being better (it is important to note that this does **not** mean that the other one is better).
0.5	50	100	46.02%	If I consider a program to be better than another when it wins **strictly** more than half the time, then if a program won just half the games, the probability that it is indeed better is just less than 1/2.
0.5	51	100	54.00%	In the same case, if one program wins just a litte bit more than half the games, then there is more than 1/2 a chance that the program is indeed better.
50 games				
0.5	27	50	66.57%	
0.5	30	50	90.22%	
0.5	33	50	98.59%	
0.5	35	50	99.76%	
100 games				
0.5	52	100	61.84%	
0.5	55	100	81.73%	
0.5	58	100	93.50%	
0.5	60	100	97.29%	

Figure A.1: Relevance of game statistics

```
9  def relevant(alpha, p, n):
10         """Returns the probability of one program being indeed better.
11         * p number of games won by the program,
12         * alpha is the critical percentage of wins
13           above which I consider a program to be better,
14         * n is the number of games played."""
15
16         sum = 0
17         for k in range(int(alpha*n)+1,n+1):
18                 sum += combinations(k,n) * \
19                 (float(p)/n)**k * (1-float(p)/n)**(n-k)
20         return sum
```

Results are held in Figure A.1.

Appendix B

A game between *Sjeng* and my program

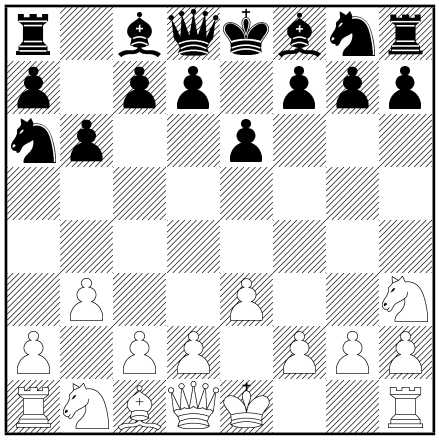
Here is a game played between my program (*Kamikaze version* with *Quiescence Search*) and *Sjeng*. The time control is 10s per player.

1.	e2–e3	

The most common opening move.

1.	...	e7–e6
2.	b2–b3	b7–b6
3.	♗f1–a6	♘b8×a6
4.	♘g1–h3	

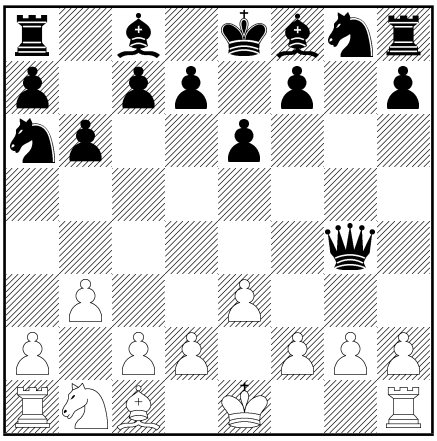
Last move from my program's opening book.



4.	...	g7–g5
5.	♘h3×g5	♕d8×g5
6.	♕d1–g4	♕g5×g4?

A strange move from *Sjeng*. 6. ..., ♕e3 would have been better. The

queen is now in a bad position.



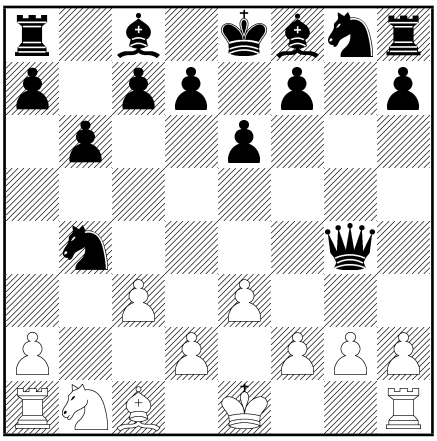
7. b3–b4!

A very good answer from my program. Instead of trying to get the queen to capture all its pieces, it also forces either the knight or the bishop to capture the pawn without having to capture back.

7. ... ♘a6×b4

8. c2–c3?

A weak move from my program. The reply 8. ..., ♕g2; 9. b4, ♕g1 would have improved the situation for *Sjeng*.



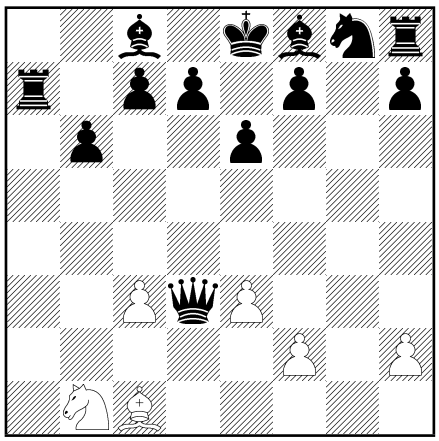
8. ... ♘b4×a2?

Sjeng didn't see the variation above.

9. ♖a1×a2 ♕g4×g2
10. ♖a2×a7 ♖a8×a7
11. ♖h1–g1 ♕g2×g1
12. ♔e1–d1 ♕g1×d1

13. d2–d3 ♕d1×d3

My program now switches to an endgame evaluation because it has 6 pieces. Note that there are more than 3 pawns on the board.

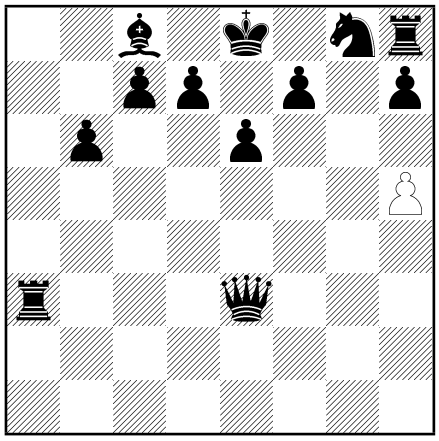


14. ♘b1–a3 ♗f8×a3

Instead I would have tried to get rid of the queen 14. ..., ♕e3; 15. e3, ♖a3; 16. ♗a3, ♗a3, leaving white with three pawns but the situation is desperate for *Sjeng* anyway.

15. ♗c1×a3 ♖a7×a3
16. f2–f4 ♔a1×c3
17. h2–h3 ♕d3×e3
18. h3–h4 ♔e2×f4
19. h4–h5

Sjeng resigns (the queen, the knight and the bishop attack `h6` so there is no escape).



Appendix C

Configuration file

A typical configuration file

```
This is the configuration file for Sushi chess.

Separate each value with tabs.
If you want to use the program defaults settings,
just delete the corresponding three letter acronym.


Pieces value during midgame (pvm)
        PAWN    QUEEN   KING    BISHOP  KNIGHT  ROOK
pvm     10      90      70      -10     30      50

Pieces value during endgame (pve)
        PAWN    QUEEN   KING    BISHOP  KNIGHT  ROOK
pve     -90     -40     -40     -40     -70     -10

Squares weight during midgame (swm)
swm
-20     -10     -10     -10     -10     -10     -10     -20
-13     0       3       5       5       3       0       -10
-10     2       15      15      15      15      2       -10
-10     7       15      25      25      15      7       -10
-10     7       15      25      25      15      7       -10
-10     2       15      15      15      15      2       -10
-13     0       3       5       5       3       0       -10
-20     -10     -10     -10     -10     -10     -10     -20

Squares weight during endgame (swe)
swe
10      10      10      10      10      10      10      10
10      10      10      10      10      10      10      10
10      10      10      10      10      10      10      10
10      10      10      10      10      10      10      10
10      10      10      10      10      10      10      10
10      10      10      10      10      10      10      10
10      10      10      10      10      10      10      10
10      10      10      10      10      10      10      10

```

```
38  Primary and secondary mobility value during midgame (mvm)
39  The primary mobility value is the number of legal moves.
40  The secondary mobility value is all possible but non legal moves.
41  mvm     0       0
42
43  Primary and secondary mobility value during endgame (mve)
44  mve     0       0
45
46  Number of pawn and number or pieces on the opposite team before
47  switching to endgame strategy (npp)
48  npp     3       6
```

Appendix D

Typical program output

A typical output

```
  +---+---+---+---+---+---+---+---+
8 |'r'|'n'|   |   |'k'|   |'n'|'r'|
  +---+---+---+---+---+---+---+---+
7 |   |   |'p'|   |   |'p'|   |'p'|
  +---+---+---+---+---+---+---+---+
6 |   |   |   |   |'p'|   |   |   |
  +---+---+---+---+---+---+---+---+
5 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
4 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
3 |   |   |   |   |   |   | P |   |
  +---+---+---+---+---+---+---+---+
2 |   |   | P | N | P | P |   | P |
  +---+---+---+---+---+---+---+---+
1 |   |   | Q |   | K | B |   |'b'|
  +---+---+---+---+---+---+---+---+
    a   b   c   d   e   f   g   h

-> White:
2       1000    0       32      c1b1 h1g2
2       5000    0       34      c1a1 a8a1
2       8000    0       44      f1g2 h1g2
2       10000   0       64      e2e4 h1e4
2       10000   0       74      e2e4 h1e4
3       3000    0       78      e2e4 h1e4 d2e4
3       5000    0       101     c1a1 a8a1 e1d1
3       8000    2       155     f1g2 h1g2 c1b1
3       8000    3       220     f1g2 h1g2 c1b1
4       17000   3       298     f1g2 h1g2 e2e4 g2e4
4       17000   9       922     f1g2 h1g2 e2e4 g2e4
5       17000   9       1076    f1g2 h1g2 h2h3 g2h3 c1b1
5       17000   15      2965    f1g2 h1g2 h2h3 g2h3 c1b1
6       26000   18      3861    f1g2 h1g2 h2h3 g2h3 g3g4 h3g4
6       26000   37      16097   f1g2 h1g2 h2h3 g2h3 g3g4 h3g4
Found 1 good moves.
move f1g2
```

```
38    +---+---+---+---+---+---+---+---+
39  8 |'r'|'n'|   |   |'k'|   |'n'|'r'|
40    +---+---+---+---+---+---+---+---+
41  7 |   |   |'p'|   |   |'p'|   |'p'|
42    +---+---+---+---+---+---+---+---+
43  6 |   |   |   |   |'p'|   |   |   |
44    +---+---+---+---+---+---+---+---+
45  5 |   |   |   |   |   |   |   |   |
46    +---+---+---+---+---+---+---+---+
47  4 |   |   |   |   |   |   |   |   |
48    +---+---+---+---+---+---+---+---+
49  3 |   |   |   |   |   |   | P |   |
50    +---+---+---+---+---+---+---+---+
51  2 |   |   | P | N | P | P | B | P |
52    +---+---+---+---+---+---+---+---+
53  1 |   |   | Q |   | K |   |   |'b'|
54    +---+---+---+---+---+---+---+---+
55      a   b   c   d   e   f   g   h
56
57  -> Black:
```

Appendix E

Implemented *XBoard* commands

Implemented XBoard commands

```
1  /**
2   * Stops ASCII play
3   */
4  public static final int XBOARD = 0;
5  /**
6   * Protocol version >= 2: changes the way my program and xboard interact
7   */
8  public static final int PROTOVER = 1;
9  /**
10  * Received a 'protover' < 2
11  */
12 public static final int NOPROTOVER = 2;
13 /**
14  * Starts a new game
15  */
16 public static final int NEW = 3;
17 /**
18  * Quits the program
19  */
20 public static final int QUIT = 4;
21 /**
22  * Received a move
23  */
24 public static final int MOVE = 5;
25 /**
26  * Received an invalid move reply from XBoard
27  */
28 public static final int INVALID = 6;
29 /**
30  * XBoard tells the computer to play the current colour
31  */
32 public static final int GO = 7;
33 /**
34  * XBoard user asks for a Hint
```

```
35   */
36  public static final int HINT = 8;
37  /**
38   * XBoard asks for computer to be put in force mode (check moves but don't play)
39   */
40  public static final int FORCE = 9;
41  /**
42   * XBoard did not accept sending moves with 'usermove' command
43   * This is a parameter used when 'protover' >=2.
44   */
45  public static final int ACCEPTED_USERMOVE = 10;
46  /**
47   * XBoard did not accept variant 'suicide chess'
48   */
49  public static final int NOT_ACCEPTED_SUICIDE = 11;
50  /**
51   * XBoard sent a ping signal
52   */
53  public static final int PING = 12;
54  /**
55   * XBoard sent an Undo signal
56   */
57  public static final int UNDO = 13;
58  /**
59   * Xboard sent a remove signal (treated at two undoes)
60   */
61  public static final int REMOVE = 14;
62  /**
63   * Xboard asks to display thinking output
64   */
65  public static final int POST = 15;
66  /**
67   * Xboard asks NOT to display thinking output
68   */
69  public static final int NOPOST = 16;
70  /**
71   * XBoard changes the board position
72   */
73  public static final int SETBOARD = 17;
74  /**
75   * XBoard sends a signal to change ply depth
76   */
77  public static final int SETPLY = 18;
78  /**
79   * XBoard sends an accepted signal
80   */
81  public static final int ACCEPTED = 19;
82  /**
83   * XBoard sends an 'variant suicide' signal
84   */
85  public static final int VARIANT_SUICIDE = 20;
86  /**
87   * XBoard sends a book request
88   */
```

```
89  public static final int BOOK = 21;
90  /**
91   * Unknown command
92   */
93  public static final int UNKNOWN = -1;
```

Appendix F

Suicide Chess problems (study positions)

My collection of 23 problems

```
# adapted from http://www.matf.bg.ac.yu/~andrew/suicide/2004/en002.htm
k7/8/6R1/8/8/8/8/8 w - - 0 1
8/8/8/8/8/6R1/p7/8 b - - 0 1
1B2R3/1Q6/1N6/7N/3K4/6R1/p7/8 b - - 0 1
# from pion.ch
8/8/7p/5p2/5P2/4pP1N/4P2P/8 w - - 0 1
# from http://www.matf.bg.ac.yu/~andrew/suicide/StanGold/problems.htm
# easy
8/P7/8/k7/P7/8/8/8 w - - 0 1
8/8/8/4b3/8/1R5R/4P3/8 w - - 0 1
8/8/p7/8/8/8/7P/8 w - - 0 1
8/5p2/7p/8/7P/8/5P2/8 w - - 0 1
8/P7/8/8/8/5k2/8/7b w - - 0 1
2q5/3k4/8/8/8/5K2/8/8 w - - 0 1
8/8/8/6b1/1NR5/8/8/8 w - - 0 1
8/P7/8/8/1nn5/8/8/8 w - - 0 1
# medium
r7/p4P2/P7/2P4p/3PP3/6b1/6B1/8 w - - 0 1
8/8/2b5/8/6p1/8/N4P2/4R3 w - - 0 1
5B2/3p1k1p/5N2/8/8/8/8/8 w - - 0 1
8/8/7N/4n3/3k4/8/4P2K/8 w - - 0 1
8/8/8/8/8/3p4/2N5/6N1 w - - 0 1
6nk/8/8/8/8/7N/8/8 w - - 0 1
8/8/8/1r1P1r2/8/8/1P3P2/8 w - - 0 1
8/8/8/8/3p2N1/8/3P4/1R6 w - - 0 1
# special case, white draws (easy)
B2n4/pP6/1b6/8/8/8/8/8 w - - 0 1
# special case, medium draws (medium)
8/p1pPn3/p7/8/8/8/8/8 w - - 0 1
8/8/4BN2/8/8/8/8/r7 w - - 0 1
```

Appendix G

Project Proposal

Diploma in Computer Science Project Proposal

Programming a *Suicide Chess* playing program

djib, Churchill College

Originator: djib

September 25, 2019

Special Resources Required

The use of my own IBM PC (1.3GHz Pentium Centrino, 256Mb RAM and 25Gb Disk).

Project Supervisor: W. Tunstall-Pedoe

Director of Studies: Dr J. Fawcett

Project Overseers: L. Paulson & C. Hadley

Introduction

Suicide Chess is a chess variant in which the objective is to lose all of your pieces. The rules of the game are the same as those of chess except for the following additional rules:

1. Capturing is compulsory.
2. When a player can capture, but has different choices to capture a piece, they may choose among them.
3. The king plays no special role in the game, i.e,
   (a) It may be captured like any other piece.
   (b) There is no check or checkmate.
   (c) Castling is not allowed.
   (d) Pawns may also promote to King.
4. In the case of stalemate, the winner is the player who has the smallest number of pieces. If the number is equal, the game is drawn.

The aim of this project is to program a *Suicide Chess* computer player.

Work that has to be done

The project breaks down into the following main sections:

1. Reading about the standard techniques and algorithms that are used to program a chess-playing computer and find ways to adapt them to suicide chess. Learning about *suicide-chess* heuristics.
2. Understanding the *XBoard*[1] protocol[2] since *XBoard* will be used as the interface for the *suicide chess* program.
3. Implementing a version of the game that checks if the move in a two player game are valid.
4. Adding a *static evaluation function* as well as a *search function* to get the computer to play.
5. Testing the program by playing against some well known *suicide chess* programs like *sjeng*[3][4] and by giving it puzzles to solve. Those statistics could be used as criteria to evaluate my project.
6. Eventually considering refinements to the search methods.

Starting Point

The *suicide chess* program will be programmed from scratch.

[1] http://www.tim-mann.org/xboard.html

[2] There are two famous protocols used by chess interfaces. *XBoard* protocol and *Universal Chess Interface* protocol. This project will be using the *XBoard* protocol because it's not possible to play with *UCI* engines in console mode for testing (or at least not convenient because it is a stateless protocol so it would require entering the whole movelist each turn!). Moreover *XBoard* protocol is supported by more *GUIs* than the *UCI* protocol.

[3] Named number one losers/giveaway player on the *Internet Chess Club*

[4] *sjeng* is compatible with the *XBoard* protocol

Resources

I plan to use my own IBM PC. I have an external USB disk that I will be able to use to do backups twice a week. I also plan to use my *PWF* account to do backups once every fortnight.

Work Plan

Planned starting date is 05/12/2005.

December 5th - December 11th

I will start by searching for any previous work on the subject and look for Suicide Chess theory. I will also read on chess-playing techniques and algorithms : minimax, alpha-beta pruning, static evaluation functions (material, mobility, centre control), transposition tables (hashing, Zobrist), quiescence search, singular extensions, etc.

December 11th - December 18th

During this period of time I shall consider how the techniques studied in the previous week can be adapted to *suicide chess*. I also plan to spend time to understand *XBoard* protocol and try simple examples (sequences of moves) to see how the interface is being used. I will also try simple examples to see how to play against other *suicide chess* programs (like *sjeng*)[5].

January 3rd - January 30th

During January, I plan to start implementing the program. This first version of the program should be able to check if the moves in a two player game are valid, but will not be able to play. This involves coding a move generation function (generates a list of legal moves for a given position), storing the position of the board (and thus deciding what datastructure to use to hold the position of the bord), ...

January 31st - March 3rd

In February I will program a static evaluation function (gives a score for the current position) as well as a basic search function (searches using minimax/alpha-beta pruning for the best move to play. Use iterative deepening to make best usage of time allocated to search). At this stage the it should be possible to play against the computer.

February 26th - March 3rd

During this week, while keeping on programming, I will write the 'progress report'.

March 6th - March 31st

In March I want to create a function that tests the level of the computer player by using some test suite of positions and evaluating the solution time, the percentage of good moves, ... I also want to automate playing against other *suicide chess* programs to be able to evaluate the level of the computer player by looking at the performance against other programs.

[5] I plan to program in *Java* but if interfacing with other programs is impossible using that language, I will then choose *C++*

Such a feature will allow fast testing of the performances of the computer player after major modifications in the source code.

I shall also play a tournament of several hundred games with another program. Create a large test suite and run program with it giving it different amounts of time per position to find the best move (e.g. 5 seconds, 30 seconds, 3 minutes, 30 minutes). I will create a table of scores.

April 17th - May 11th

- Improving the program using move ordering. For example listing moves which place one of the pieces on a square where it can be captured are probably worth exploring earlier than others. Similarly capturing low value pieces should be listed before capturing high value pieces.
- If I have time I would also try to implement a hash table and improve the search function using search extensions such as intelligent handling of time (ie: spend more time on moves that seems to be important for the rest of the game), thinking during your opponent's time, ...

I plan to test program after each major change.

June

This is when I plan to write the report.