Helsinki Metropolia University of Applied Sciences
Degree Program in Information Technology

Claudio M. Camacho

Adaptive Behavior in Artificial Intelligence

Bachelor's Thesis. October 31, 2009.
Supervisor: Jaakko Pitkänen, Principal Lecturer
Language Advisor: Taru Sotavalta, Senior Lecturer

Helsinki Metropolia University of Applied Sciences Abstract

Author	Claudio Camacho
Title	Adaptive behavior in artificial intelligence
Number of Pages	55
Date	October 31, 2009
Degree Programme	Information Technology
Degree	Bachelor of Engineering
Supervisor	Jaakko Pitkänen, Principal Lecturer

The goal of this project was to study the characteristics and consequences of using dynamic behavior in artificial intelligence (AI). The main purpose was to create a computer chess program with two different artificial intelligence engines and using them as a tool for performing different experiments.

The project was carried out by developing a simple chess program with two different AI engines, one containing classical AI techniques without the use of dynamism, and another AI engine with adaptive behavior and dynamic evaluation techniques.

The results showed a remarkable improvement on the adaptive AI engine over the classic AI engine. The results also demonstrated the challenge of implementing proper adaptive behavior for the system. Nevertheless, it was proved that it was feasible to program certain dynamism, as an extension to the classic AI engine, thus enabling the intelligent agent to provide adaptive and human-like behavior when playing a game.

To summarize, the results proved the possibility to create a simplified model of a computational intelligent system with certain basic dynamic behavior capabilities. Nonetheless, it is recommended to further develop the current implementation in order to enable the intelligent agent to widen its dynamic behavior functionality and to implement basic learning capabilities.

Keywords	artificial intelligence, adaptive behavior, computer chess

Contents

Abstract 3

1 Introduction 4

2 Literature Review 5
- 2.1 Approaches to Artificial Intelligence 5
- 2.2 Computer Chess and Human Cognition 5

3 Computer Chess Techniques 7
- 3.1 Board Representations 7
- 3.2 Evaluation 8
- 3.3 Tree Traversing and Minimax 9
- 3.4 Learning Methods 12

4 Implementation of Chess0 14
- 4.1 Application Design and Purpose 14
- 4.2 Common Techniques 16
  - 4.2.1 Negamax 16
  - 4.2.2 Alpha-Beta Pruning 18
  - 4.2.3 The Heuristic Function 20
- 4.3 Improvements to the Dynamic AI Engine 24
  - 4.3.1 Randomization 24
  - 4.3.2 Dynamic Heuristics 26
  - 4.3.3 Quiescence Search and the Horizon Effect 31

5 Results 36
- 5.1 Randomization Tests 36
- 5.2 Dynamic Heuristics Tests 39
- 5.3 Quiescence Search Tests 44

6 Discussion 49
- 6.1 Achievements 49
- 6.2 Application Fields 51

7 Conclusions 52

References 53

1 Introduction

Artificial intelligence (AI) is, according to the Compact Oxford English Dictionary of Current English, the capability of computer systems to perform tasks which require the involvement of human intelligence [1]. Furthermore, Computational Intelligence (CI) is defined as the subset of artificial intelligence, which covers the studies of adaptive mechanisms for providing intelligent behaviorism in dynamic environments [2, 3-4].

The purpose of this project is to study the basic principles behind computational intelligence, experimenting with the concept of dynamic behavior in artificial intelligence. In addition to this, this projects aims at implementing such dynamism in a manner that the intelligent agent would be capable of simulating basic human-like behavior, as proposed by Alan Turing in his Turing Test. According to the Turing Test, a machine can be tested in order to discover if it is able to imitate human intelligence, thus disguising among actual human beings. [3, 77-79]

The goal of this project is to develop a chess engine, as a computational intelligent system, whose intelligence might follow an adaptive pattern, imitating the human intelligence when playing a chess match. Moreover, this chess engine should include both a classical artificial intelligence engine and a renovated adaptive engine, thus providing the necessary tools for analyzing the improvements and benefits of using adaptive techniques.

The scope of this project is limited to a basic study and a simple implementation based on computational intelligence, thus narrowing the range of study to the properties of interest. First, there is limited availability of proper equipment (for testing purposes) and, second, there is certain incapability of knowing if the Turing Test would be passed or not by the implementation of this project.

2 Literature Review

2.1 Approaches to Artificial Intelligence

The first problem that arises when discussing Artificial Intelligence (AI) is how to define intelligence. Most AI-related research processes rely on the study of the human intelligence, at first, whereas, at the second stage, they aim at defining how a machine or computer system could perform a given task demonstrating human-like behavior. [4, 3-6]

Currently, there is no unified paradigm that would establish the path of research in the field of AI. Instead, different approaches are being investigated in order to tackle the problem of creating the most suitable representation model for the human reality. Hence, there is a constant debate between different research groups, where each group defends its approach to be the correct one. The most important researches are based on Symbolic AI, Computational Intelligence (including all Connectionism paradigms), and Artificial Life, among others. [3, 11-18]

Computational Intelligence (CI) requires that the intelligent system develops itself and learns from a dynamic environment [2, 3-4]. On the other hand, Symbolic AI deals with the abstract duality of symbol-association [3, 71-95]. Therefore, and having in mind the current trends in computing, CI is one of the most researched approaches inside AI, and it includes different branches such as artificial neural networks, evolutionary computation, artificial immune systems and fuzzy systems [2, 3-13]

2.2 Computer Chess and Human Cognition

Computer chess is an example of an AI challenge which benefits from the principles of CI. The thinking style during a chess match requires a comprehensive amount of computation, thus becoming suitable for being implemented as a CI system. Nonetheless, there are certain aspects, such as the representation of the board and the pieces, that are considered more a cognition problem than a computational problem. Therefore, a computer chess program involves a CI system with both cognition and computation capabilities. [4, 20-24]

Accordingly, a computer chess program has to be capable of representing the reality (the chess board and the pieces, in this case) taking two key factors. into account. First, the internal representation must be as close to reality as possible; otherwise the system

will not work. Second, the internal representation must be as optimal as possible, hence enabling the computational model to process a wider range of information in a narrower time frame. However, the second factor is purely conditional since a chess program may still represent the reality perfectly without any optimization. [5, 1-37]

As regards cognition, a computer chess program, according to the Turing Test, must prove a certain level of understanding in the structure of the board and how the pieces are placed in order to produce a natural flow of the game [5, 87-100]. In consequence, a computer chess, and any computer system, that is tackling the Turing Test must account for a set of learning methods, both short-term and long-term, thus leading to performance improvements in the dynamism of its behavior [6, 1-3].

Besides the performance improvements, dynamic learning affects the overall behavior of the system, resulting in a more flexible behavior. For instance, hardwired behavior denotes a fixed pattern of behavior through the life cycle of the task performed by the system. However, dynamic behaviors are based on initial principles that are not completely hardwired, but they are modified by a given set of functions and algorithms, during the life cycle of the task. [6, 3-6]

The computer chess cognition principles and how learning methods are put into practice condition how human beings perceive the intelligence of that system. Most computer chess programs denote a common style of playing, applying brute force-like algorithms in order to find the best move at any point in the match. Nevertheless, humans do not always play the best move, as they analyze the situation and generate a response based on other factors besides the actual chess board. Hence, a computer chess program is most likely to be detected as a mere machine, by experienced human chess players. [5, 62-87]

In conclusion, a computer chess program must, in order to depict human-like behaviorism and tackle the Turing Test problem, adapt itself through each of the moves in a game, thus altering its learning principles along the different situations presented across different games. As a result, the computer chess program may acquire, with time, a certain degree of dynamism that could lead to highly flexible behaviors found in human responses when interacting with a complex environment. [7, 53-55]

3 Computer Chess Techniques

3.1 Board Representations

The first step towards reality cognition and recognition in computer chess is the representation of the board and the pieces. Admittedly, a piece of software must be told, in exact terms, what a piece is, what a square is and what the board is. As the current technology sets the requirements of how information is stored and processed, the representation of the board must be, generally speaking, a numerical approximation, since current technology is built on the top of simple arithmetic calculations. [5, 1-2]

A simple representation of a chess board may be based on a two-dimensional matrix of eight rows and eight columns, where each entry into the matrix contains the representation of one piece or an empty square. This is a straightforward approach, from the human-cognition point of view. However, there are alternative solutions, such as creating a unidimensional array of 64 components. The differences between different representations affect the manner in which the information is internally processed by the system. Therefore, the main goal of studying different representations is to produce a proper representation with the maximum possible efficiency. [8]

Besides the classical way of representing a board in a vector of pieces and squares or a two-dimensional matrix, there is a relatively new technique which is based on the use of 64-bit computer extensions to perform computations in the minimum possible amount of processor instructions. This technique relays on the so-called *bitboards* (or bitmaps). In chess, there is the coincidence that there are exactly 64 bits, which is the current computer trend in processor designs. Therefore, it is possible to codify a whole board of chess in a single number of 64 bits. The gains are due to the fact that a board can be processed in a single processor instruction with current computers, whereas the classical matrix representation takes at least 64 processor instructions to be processed. [8]

Bitboards cannot represent every piece, but just contain a 0 or a 1 on a square. However, current computer chess programs store different bitboards (e.g. a bitboard for the white bishops, another one for the black pawns) and then perform operations by using simple boolean algebra against different bitboards. Figure 1 demonstrates the usage of two different bitboards which can be used together to perform a chess operation in a single processor instruction. [8]

	A	B	C	D	E	F	G	H
8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0

a) white king position

	A	B	C	D	E	F	G	H
8	0	1	1	1	1	1	1	0
7	1	1	1	1	1	1	1	1
6	1	1	1	1	1	1	1	1
5	1	0	0	1	1	1	0	0
4	0	0	1	1	1	1	1	0
3	0	0	1	0	0	0	0	1
2	0	0	0	1	0	0	0	0
1	0	0	0	0	1	0	0	0

b) black pieces attack

Figure 1: Bitboard superposition technique

The bitboard *a* in figure 1 shows the position of the white king, represented by a 1, and all the 0s are the other 63 squares, where the white king is not residing. On the other hand, the bitboard *b* in figure 1 depicts the set of squares to which black pieces are currently attacking. In order to see if the white king is in check or not, a computer chess program using these bitboards may simply superpose them using boolean multiplication (logic operation AND). If the result of this operation is different than zero, then the white king is in check (one or more black pieces are attacking the square where the white king is), otherwise the white king is not in check (AND operation results in zero because no black attack superposes with the white king position).

Finally, comparing the classical matrix approach with the new bitboard technique, the matrix-based computer chess would have to check all the squares one by one, against the square of the king, and create an accumulative partial result, giving the answer after at least 64 processed squares. Nevertheless, using bitboards, the very same operation is performed at once, being executed in one processor operation (on 64-bit processors) or in two (on 32-bit processors). [8]

3.2 Evaluation

One of the fundamental blocks for building a computer chess program is an evaluation function, sometimes referred to as the heuristic function. This function has the role of generating an absolute estimate for a given position in a chess board (static evaluation), as well as providing some orientating data for the AI program, thus guiding the search of a move. [5, 7-8]

An artificial intelligence application must be guided in the process of searching what to do next. This is one of the basic principles around AI, since the intelligent agent is supposed to achieve an objective and to be guided through its life cycle, until it achieves its given task or goal. This is the existence purpose of the AI agent. Accordingly, a chess program must be taught how to achieve the victory during the course of a game. Therefore, the heuristic evaluation function determines how suitable the position in the board is towards the goal achievement. In other words, the chess program must evaluate each position during the game and understand what is a *good* move and a *bad* move. [5, 7-8]

There are different ways of evaluating a chess board. The evaluation function is a fine-tuned mathematical expression that uses a weighted mean for estimating how proper the position is. The heuristic function may vary from one chess program to another, and the formula is typically designed by the author of the chess program. [5, 7-34]

The classical approach for generating an evaluation function has always been to create a weighted mean using separate values for the material on the board (how many pieces are left and how valuable they are) and the position of the pieces (how well situated they are on the board). The material is the most valued factor, since a material loss immediately leads to a position where it is most likely to be lost by the player lacking material. Normally, the ratio between the material evaluation and the positional evaluation is of the order of 3000:1, meaning that the material evaluation is 3000 times more important than the positional evaluation. [5, 7-34]

The classical approach for evaluating a position is static, meaning that the evaluator shall not know about what has happened before that position nor what is it going to happen after that position. Nevertheless, modern trends tend to incorporate additions to the evaluator, by which the heuristic function is informed about previous states of the board, thus providing the chess program with a more dynamic overview of the game. [9, 5-24]

3.3 Tree Traversing and Minimax

In most AI applications where there is a known set of possibilities (*moves* in the case of chess), programmers tend to implement a function for generating every possible combination of choices and then traversing the tree in search for the best possible combination. The word *best* refers to the most suitable move for the AI agent, in order to achieve its given

task or goal in the minimum possible amount of time. This goal is typically the victory, when referring to a game. [10]

For instance, the AI engine of a tic-tac-toe game has to compute a total of 362880 games. This number is relatively small, with respect to the current computation trends and capabilities. Therefore, a tic-tac-toe AI program may generate the whole tree of possibilities, play each branch, and finally decide what the best move is at any time of the game. In this case, there is no need for a heuristic evaluation function. This is due to the fact that every possible game can be played, and hence the machine responds with a 100% accuracy. [11]

In chess, however, the number of possible combinations (different games) is assumed to approach infinity. Therefore, in order to know which is the best move to perform at a current position (meaning that best is towards the victory of the moving player), a computer chess program should play a vast number of possible matches in order to discover the winning move at any time. The current computation technology is limited, and hence applications in which such a vast number of calculations is involved are not capable of providing a perfect result. Instead, these applications use fuzzy techniques which intend to discover which part of the analysis can be avoided, thus reducing the volume of the data to be processed dramatically. [5, 43-48]

An expensive part of a computer chess is the one in charge of generating the moves. They are generated in the form of a non-binary tree, where moves of a same player on the same turn are the siblings of a branch in the tree. The tree has the depth, which is measured in *plies* (half-moves). A computer chess program is not capable of playing 10^{500} different matches, but, instead, has to evaluate a board several moves ahead of the current board position. That is, the computer chess program generates a tree of a moderate depth and modifies the current board applying all the moves on each branch, then it evaluates that situation. Finally, the value of each branch of the tree (each possible game) is backed up, thus selecting the branch with a higher probability of success. [5, 7-43]

The method is typically achieved with a combination of a move tree (data structure) and a minimax-like algorithm. The minimax method is a simple recursive algorithm to decide the best move using the philosophy that the maximum benefit is the minimum loss. Therefore, a minimax algorithm traverses the move tree and finds the minimum possible loss for the moving player, taking into account that the opponent will maximize

its possibilities. The evaluation of each node is, in turn, performed by a heuristic function. [5, 38-43]

Figure 2 demonstrates the usage of the minimax algorithm. First, at depth 1, the computer selects the maximum possible score. Then, at depth 2 (that is, all possible replies to the moves at depth 1), the computer selects the minimum scores, in order to minimize the losses (since it is the opponent who is playing now). Afterwards, at depth 3, the computer applies the same logic as in depth 1, thus recursively finding the minimum loss out of all the generated possible moves. [12]

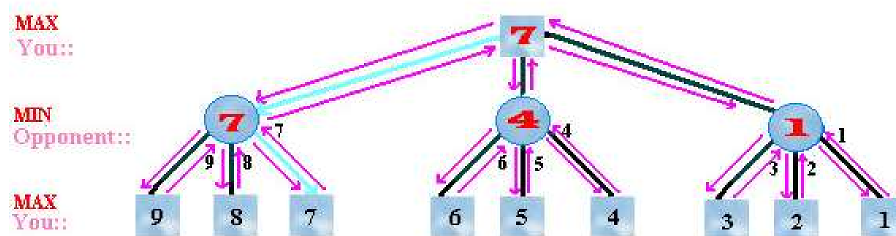

Figure 2: Move tree with branch scoring [12]

Eventually, the number of possibilities in a move tree may become large. Therefore, the volume of data in the move tree must be substantially reduced, hence enabling the computer system to process the information within a finite amount of time. For this purpose, there are two common methods: decreasing the depth of the tree (smaller in height) or pruning non-interesting leaves (smaller in width). [12]

Reducing the depth of the tree is generally a non-effective method, since the computer chess program has less capability of seeing what will happen in future game situations. Nonetheless, it is possible to shrink the width of the tree, which consists of eliminating those branches in which it is impossible to achieve a winning result. The heuristic function plays a critical role here, since it is used by the tree-pruning algorithm to determine if the branch is interesting or not, towards the target of the computer chess program. [12]

The pruning algorithm applies the heuristic function at every depth in the tree. As soon as an evaluation on any of the moves (at any depth) turns to be worse than a previously evaluated move, the computer chess program may cut off that branch of the tree. This assumption is based on the fact that the opponent is supposed to play the best possible

reply at every moment. Figure 3 shows another example of a move tree, which has now been pruned using a heuristic pruning. Evidently, the volume of the data to be analyzed in the tree from figure 3 is reduced by applying the cut-off technique to the original move tree. [11]

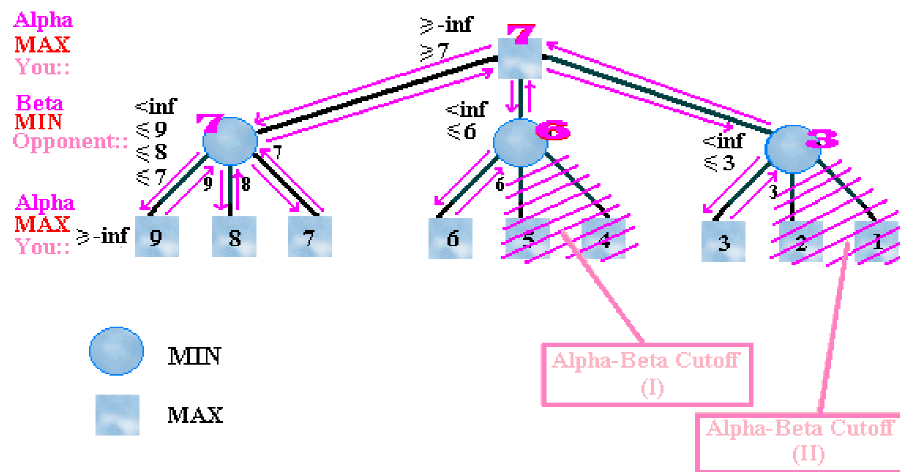

Figure 3: Pruned move tree with branch scoring [12]

The idea of pruning is general, and its implementation may be performed in different ways. Nevertheless, the main idea is to reduce the size of the data to be processed by shrinking the width of the data tree. Modern techniques do not simply rely on a heuristic function to perform the cut-off, but on several heuristic functions and other situational factors that may show evidence of a non-interesting branch, thus leading to a cut-off on that part of the tree. [13, 7-18]

3.4 Learning Methods

Generally, computer chess is studied as a concrete application of Artificial Intelligence. Hence, there is no concrete technique for achieving learning, but computer chess programs utilize common learning methods described in AI and CI. A computer chess program may be implemented using any general learning technique, such as neural networks, Bayesian networks. [14, 1-15]

Despite the general learning methods researched alongside AI and CI, there are certain simple configurations that allow a computer chess program to learn. These configurations

may introduce both short-term learning and long-term learning. Short-term learning typically refers to volatile learning, where the machine learns while it is functioning and then it clears its memory up on termination. On the other hand, long-term learning can be thought as of permanent data storage, where the machine learns while functioning, but it stores the learned data on static storage up on termination. [5, 60-61]

Short-term learning is a typical feature in current computer chess applications. It is usually referred to as *transposition tables*, since it is based on the idea that any board seen during a game may be stored in the memory. Then, if the same board arises once again further on in the same game, the machine will be able to recall it from its memory. The only benefit of the transposition tables is merely the performance gain, since the machine recalls an already-analyzed position, thus not having to analyze again this position, and generating an immediate reply to that situation. Furthermore, these transposition tables may be permanently stored in a disk file, thus becoming long-term learning, eventually. [5, 60-61]

4 Implementation of Chess0

4.1 Application Design and Purpose

Chess0 is a computer chess program, which aims at providing a simple chess engine with a fully understandable source code and offering the basic tools for analyzing the behavior of different AI implementations. Chess0 is an example of simple AI for playing two-player turn-based games, and it includes two different AI approaches in the same application: a classical AI engine and an adaptive AI engine.

The classical AI engine is composed of a heuristic evaluation function, a minimax-like algorithm and an alpha-beta pruning technique for speeding up the decision tree search operations. In addition, the adaptive AI engine supports new features such as randomization, dynamic heuristics and quiescence search, which dramatically improves the behavior of the application compared to the classical behavior.

Figure 4 depicts the general organization of the application. The user communicates either with a command-line application manager or with a graphical chess board. As it can be understood from figure 4, the application manager further relies on a game manager, which, in turn, keeps track of the current game, using a board representation.

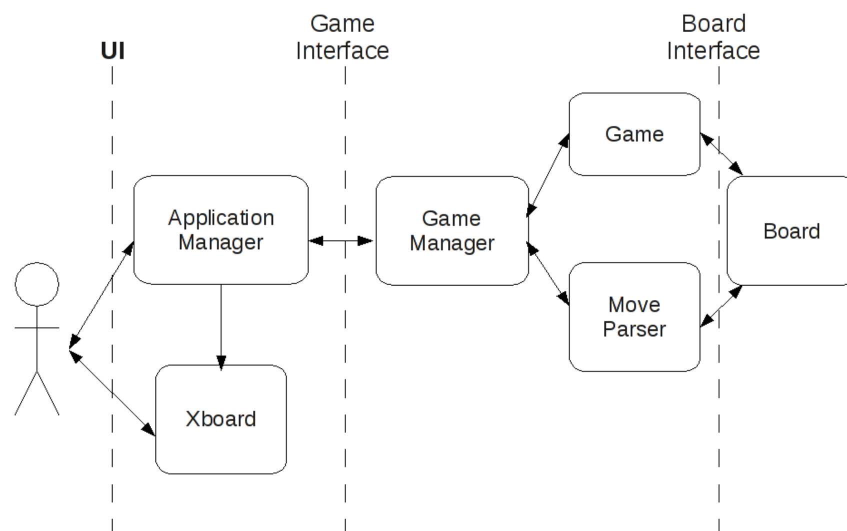

Figure 4: Chess0 architectural design

Moreover, Chess0 can be understood as implementing several interfaces between the application components. According to figure 4, the user interface is provided either by the

command-line application manager or by the graphical interface. Then, there is a game interface between the game manager and the game module, which assures that several games may be abstracted by the game manager and further presented to the application manager as a whole. Finally, the board interface is the border between the game and the board objects, in which the board represents the reality for this concrete application (such as squares or pieces).

Figure 5, on the other hand, sketches the relation between the different AI engines in Chess0 and how they are managed by the application. The impact of having two AI engines running at the same time allows the user to switch, on the fly, from one engine to another. The ultimate goal of this feature is to provide an easy tool for testing and comparison purposes.

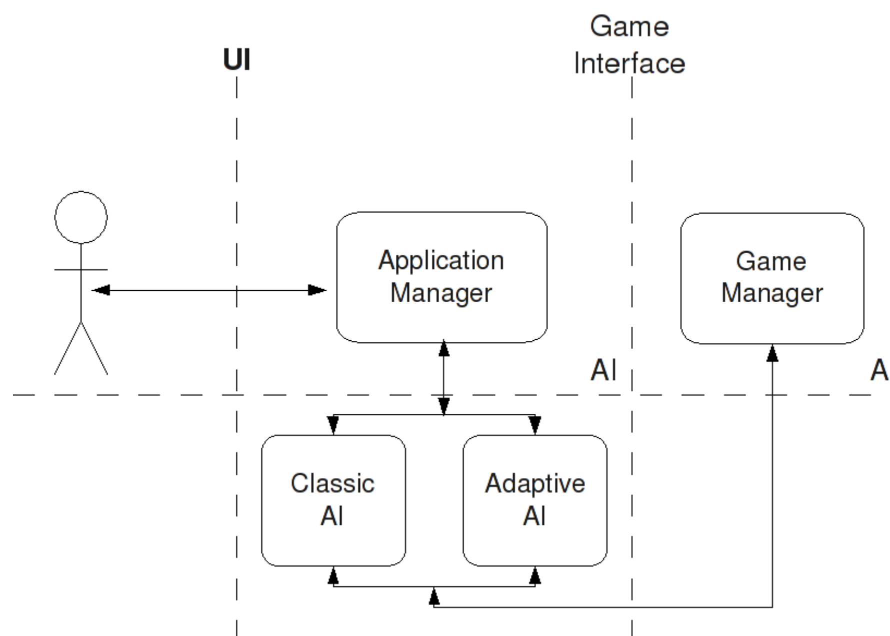

Figure 5: Chess0 multiple AI-engine interface

For instance, as it is demonstrated in figure 5, a user may request the application manager to switch from an AI engine to another. Therefore, the behavior of the chess engine may be changed while running the application, thus allowing the user to test a move and its behavior for a given situation using one type of AI and then switching to the other type of AI and testing the same move.

To summarize, the main feature of Chess0 is that it provides two different AI engines, while they as interchangeable on the fly (while running the application). Both engines share some parts of the code, which are the common techniques, whereas the adaptive AI engine includes new features that aim at improving the overall behavior of the application.

4.2 Common Techniques

4.2.1 Negamax

Chess0 aims at implementing a simple chess engine with two built-in computer chess AI engines. One using the classical approach, that is a simple heuristic function, a minimax-like algorithm and alpha-beta pruning for the decision tree. The other AI engine, named adaptive AI engine, implements a set of dynamic features besides the classical ones. However, they share the source code for the common techniques.

First, the program studies each reply by analyzing the decision tree using a minimax-like algorithm named *Negamax*. Negamax is a simplification over the classical Minimax algorithm where the two necessary functions (the one for maximizing and the one for minimizing) are synthesized into one unique function which is negated and inverted every time it is called. Listing 1 unveils the generally accepted pseudo-code for the Minimax algorithm. [16, 249-252]

```
function minimize(tree_depth):
    if tree_depth is 0: return evaluation;
    best_score = -infinity;
    for all valid moves as cursor do:
        apply_move(cursor);
        myscore = maximize(tree_depth - 1);
        undo_move(cursor);
        if myscore is greater than best_score:
            best_score = myscore;
    return best_score;

function maximize(tree_depth):
    if tree_depth is 0: return evaluation;
    for all valid moves as cursor do:
        apply_move(cursor);
        myscore = minimize(tree_depth - 1);
        undo_move(cursor);
        if myscore is greater than best_score:
            best_score = myscore;
    return best_score;

function minimax(tree_depth):
    if tree_depth is 0: return evaluation;
    for all valid moves as cursor do:
        apply_move(cursor);
        myscore = minimize(tree_depth - 1);
        undo_move(cursor);
        if myscore is greater than best_score:
            best_score = myscore;
    return best_score;
```

Listing 1: Pseudo-code for the Minimax algorithm found in previous versions of Chess0

According to the information presented in the code listing 1, the Minimax algorithm uses three functions that are practically similar. First, the *minimax* function is used as a root function, for all the valid moves on the current board situation. Then, the machine tries to minimize the loss (thus maximizing the win) by calling the *minimize* function. Further, the *minimize* function calls the *maximize* function, and thus the program begins to search recursively, as minimize and maximize call each other until the limit depth of the tree has been reached (this is the base case for the recursive algorithm in this case). [13, 5-9]

The code presented in listing 1 denotes similarities across functions, which may be reduced

into a more compact format. This is the purpose of the Negamax algorithm [13, 5-9]. The compact version of the Minimax may be achieved using several similar algorithms. However, Chess0 uses Negamax, which is a straightforward reduction of the Minimax code. Listing 2, provides the final version of the minimax-like algorithm in its compacted form.

```
function negamax(tree_depth):
    if tree_depth is 0: return evaluation;
    best_score = -infinity;
    for all valid moves as cursor do:
        apply_move(cursor);
        myscore = -negamax(tree_depth - 1);
        undo_move(cursor);
        if myscore is greater than best_score:
            best_score = myscore;
    return best_score;
```

Listing 2: Pseudo-code for the Negamax algorithm found in Chess0

As can be seen from listing 2, the Negamax algorithm is a simplified rewrite of the Minimax algorithm, where all the functionality is compacted into a single function. The *negamax* function calls recursively itself, and it negates the returned value at each depth in the tree. This means that, at depth $N-1$, the score is negated, as well as at depth $N-3$, etcetera. Hence, the score is negated as the depth changes by two in the decision move tree. This negation explains the opposite nature of the functions *maximize* and *minimize* in listing 1. In fact, these two functions are similar, except that they seek exactly the opposite case: one looks for the best score for the moving player, and the other one looks for the best score for the opponent. [17]

4.2.2 Alpha-Beta Pruning

Despite the limited amount of search iterations found in the Minimax/Negamax algorithm, due to the limited depth of search (height of the decision tree), the number of evaluated moves is large even in relatively small decision trees. For instance, the complete list of moves for a non-started game is of 20 possible moves for the white side, which is the starting and moving player at that situation. Then, in the first move for the black side, there are 20 replies to each of the 20 possible moves of the white side. This means that, already at the second ply (that is, the first move for the black side), there are 400 possible

moves. At the third ply there are over 1,000 possible positions, and after the fourth ply there are more than a million positions.

On an average, on a decision tree for a computer chess program, there are b^d possible positions, where b represents the number of possible branches for a given position, and d denotes the depth limit of the decision tree. This implies that for a chess position at ply 4, the minimax-like algorithm must examine about one million positions in order to produce a response. Furthermore, if a game is sufficiently advanced and there are many possible branches, the number of nodes to examine may grow dramatically, thus slowing the computational resources. [5, 43-48]

During the 1960s, people in the AI field discovered an algorithm which was able to cutoff *non-interesting* branches of the tree, reducing the amount of data to process in a drastic way. This algorithm was named *alpha-beta pruning*, since it uses two variables (alpha and beta) to control where the decision tree may be pruned, hence cutting off the subsequent branches. By using this method, the number of nodes is reduced to $2 * \sqrt{b^d}$, which is a critical improvement over the b^d number of moves to be calculated with a plain minimax-like algorithm. [5, 43-48]

In order to achieve the improvement, the alpha-beta pruning method maintains the variables alpha and beta updated at all moments during the search. Alpha has an initial value of $+\inf$ and beta has an initial value of $-\inf$. Alpha denotes the *best* possible score that the computer can achieve, whereas beta represents the scores of the moves which prevent the opponent from achieving more than the best score found until now. In other words, the alpha-beta pruning method drives the minimax-like search on the decision tree in a way that the search must find only moves that increase beta (a better move for the computer) or reduce alpha (a worse move for the opponent). [5, 46]

When the program evaluates a move whose score is better than the best move found so far, it must save that move as the best move found so far and update the value of beta. Similarly, when the program evaluates a move whose score limits its opponent, it reduces the value of alpha. The final result is then generated after the whole search, and it must lie between alpha and beta. As alpha and beta become closer, the program measures the final result in a more accurate way. Furthermore, when alpha crosses beta, meaning that alpha is less than or equal to beta, the algorithm stops the search on that branch, thus cutting off the part of the tree under that node. This cut-off is the improvement over the simple

minimax-like algorithm, and it happens when a better move may not be found anymore in a decision tree, thus avoiding unnecessary and expensive calculations. [5, 43-48]

Figure 6 exemplifies the use of alpha-beta pruning on a decision tree. It is important to note that a minimax-like algorithm must be applied to the decision tree, thus involving a heuristic function in order to evaluate each node. These evaluations will provide the base for the alpha-beta pruning method to decide which branches shall be trimmed off the tree and which branches shall remain as *valid ones*. [18, 10-13]

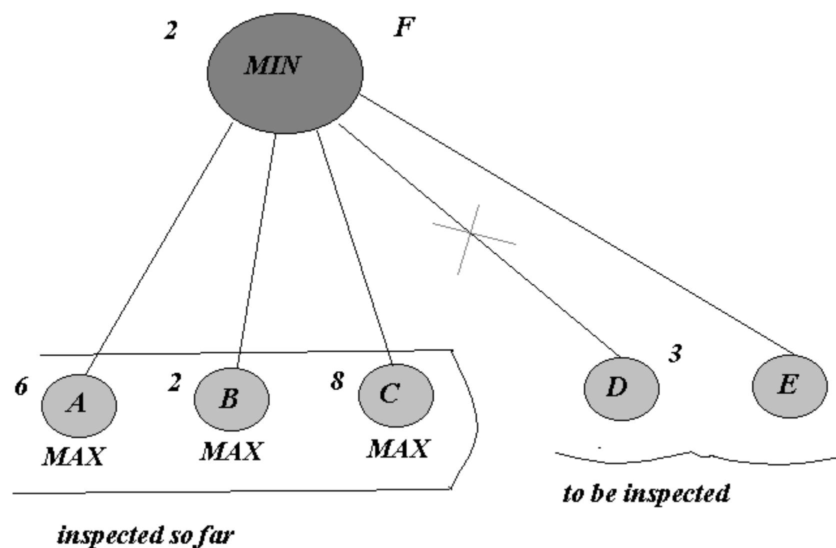

Figure 6: Example of decision tree pruning based on Minimax values [18, 11-12]

Figure 6 demonstrates the alpha-beta workflow as the branch under the node *D* is trimmed, since it cannot provide an improved result over the last recorded results. The evident improvement is, therefore, the reduction of the data volume to be processed, thus speeding up the search on the decision tree. [18, 1-13]

4.2.3 The Heuristic Function

Chess0 has been designed using two different AI engines, in order to provide a proper tool for comparing the achievements of introducing adaptive techniques to the AI behavior. Nevertheless, AI engines designed for game playing must provide a heuristic function, unless the game has a properly limited number of possible combinations. [5, 7-43]

The heuristic function represents both the rules of the game that is being played and the quality of how the game is being played, according to the given purpose to the AI program. In chess, the evaluation function is based on two critical factors: material and position. The material evaluation concerns the number of pieces and their value, whereas the positional evaluation refers to how properly the pieces are situated along the board. Evidently, the heuristic evaluation radically changes from one application to another, while the common goal remains: to define what is *good* and what is *bad*, thus providing a mechanism to the AI program to decide towards its goal. [5, 7-43]

In chess, each piece has a value, according to its mobility on the board and its capabilities of attacking the enemy. For instance, the queen is typically given the highest value, the rook is given half of the queen's value, the knights and bishops are less worthy than the rook, and finally the pawns are the measurement unit, which denotes the lowest score for them. However, this value assignment is a heuristic task, since the exact values are not absolute, and they are assigned differently depending on the player or even on the situation on the board. [5, 7-43]

Concretely in chess, material is the most valuable aspect. Positional evaluations are not as valuable, hence resulting in a weighted mean between the material and positional evaluation, where the material is typically 1000 to 3000 times more valuable than the position. Equation (1) shows a probable example of a simple and general heuristic evaluation function: [5, 7-8]

$$score = \frac{(3000 * material) + position}{3001} \tag{1}$$

$$material = white_material - black_material \tag{2}$$

To be more precise, the heuristic function returns an absolute value between $-\inf$ and $+\inf$, where a negative value means a *good* score for the black, and a positive value means a *good* score for the white. By extension, a return value of 0 represents total equity between the two players. Furthermore, the material evaluation for one player, as

it is presented in equation (2), is calculated by summing the values of each piece of that color on the board. Hence, the player with more pieces on the board will have a higher probability of winning the game, according to the heuristic function. [5, 38-48]

Normally, the material is balanced during a game, meaning that no big differences exist between the players' material. Therefore, the accuracy of measuring heuristically is stressed on the positional evaluation. This is usually achieved by rating different squares on the board with different values, thus biasing the computer program to place its pieces on higher-rated squares rather than on lower-rated squares. Moreover, there exist several key points that must be taken into account when evaluating a position, and they are all evaluated separately and added up together for generating the final positional evaluation. [5, 7-48]

Some important positional aspects that are typically included in chess engines are the center control (central squares are higher-rated than close-to-the-border squares), piece mobility (how many possible moves a piece can perform from a given position), development (according to the chess theory and the rules of strategy at the beginning of a chess game), the pawn structure and the king safety (how easy it is to attack the opponent's king and how easy to defend one's own king). Equation (3) depicts an example of positional evaluation, taking these several factors into account. [5, 7-43]

$$pos = \frac{(5 * control) + (2 * mobil.) + devel. + (2 * pawns) + (3 * king)}{13} \tag{3}$$

The weights presented in equation (3) may vary, since this is a rough estimate that the programmer must take into account when designing the chess engine. Modifying the weights in equation (3) would imply a change in the behavior of the chess engine, producing different results with different formulas. As a consequence, an AI program may be biased in its behavior by simply modifying the weights of its evaluations. Figure 7, demonstrates an exemplar representation of square-rating. [5, 28-31]

8	1	1	1	1	1	1	1	1
7	1	2	2	2	2	2	2	1
6	1	2	4	4	4	4	2	1
5	1	2	4	8	8	4	2	1
4	1	2	4	8	8	4	2	1
3	1	2	4	4	4	4	2	1
2	1	2	2	2	2	2	2	1
1	1	1	1	1	1	1	1	1
	A	B	C	D	E	F	G	H

8	1	2	3	4	4	3	2	1
7	1	5	6	7	7	6	5	2
6	1	6	8	9	9	8	6	3
5	1	7	4	10	10	4	2	4
4	1	7	4	10	10	4	2	4
3	1	6	8	9	9	8	6	3
2	1	5	6	7	7	6	5	2
1	1	2	3	4	4	3	2	1
	A	B	C	D	E	F	G	H

Figure 7: a) Central squares bias; b) Encouraged biasing [5, 18-19]

In the board *a)*, the biasing is completely normal. The squares on the edge are less valuable, and the rating of any square increases as the square approaches the central four squares. Similarly, a chess engine may be biased with a more encouraging rating, for instance by the increase of the difference in value between one square and another. In figure 7, the board *b)* shows how this is achieved, by increasing the square value at any step towards the center, resulting in a more encouraging result for the AI engine at the time of the positional evaluation. [5, 18-19]

Chess0 uses a set of formulas similar to the ones presented in this chapter. These formulas are implemented in the classical AI engine, which uses a fixed evaluation, meaning that the same set of formulas is applied to every board situation, independently of which events have occurred on the board or which events are likely to happen in the future. That is, the heuristic evaluation function in the classical AI engine in Chess0 is completely static for a given board situation.

Nevertheless, there are several problems that arise with the use of a completely static heuristic evaluation function. To begin with, the development evaluation dramatically varies from one phase of the game to another. The development evaluation should be taken into account only at the beginning of a game. Furthermore, the king's safety evaluation must be more valuable as the game approaches its end, since the objective of the game is to capture the opponent's king while preventing the opponent from capturing one's own king. [5, 7-43]

For that purpose, many chess engines nowadays implement a slightly adaptive set of formulas, which vary as the board situation is modified through the game [19, 1-4].

Chess0 implements this feature in the adaptive AI engine, which is a dynamic version of the classic engine. The main difference is that the evaluation function in the adaptive engine uses adaptiveness as its basis, in order to achieve more accurate measurements and improved results at any point of a chess game.

4.3 Improvements to the Dynamic AI Engine

4.3.1 Randomization

According to the description of the functionality in the minimax-like algorithm, a computer chess program will analyze a part of the decision tree and try to find the minimum possible loss, considering that decision the best possible move. This implementation will result in a common behavior, where the machine always picks what its best decision is. By extension, this signifies a fixed behavior for similar situations. [5, 38-43]

One of the main purposes in the AI field is to simulate and approach human-like behavior. Chess players are provided with feelings, as they are human, and therefore they may generate different responses to the same board, under different spatial or temporal situations. Additionally, a human presents a characteristic emphasis on discovering, where new things are sometimes tried out, in search of new possibilities. This behavior follows a close-to-random regression. [20, 562-571]

A simple implementation of such behavior in AI applications is the use of randomization. Randomization enables the application to select a random response from a set of possible responses, instead of picking always the properest decision. However, this randomization method must be controlled until certain extent. The fact of not having a controlled randomization process could cause the AI application to behave completely randomly in most situations. [21, 16-24]

In Chess0, the randomization process is implemented using a so-called randomization threshold. The randomization threshold is the regulative variable, which does not allow the randomization process to escape out of the controlling boundaries. In addition, this implementation is possible due to the nature of the decisions, since, in computer chess, they are basically a move response and its absolute score (win or loss after applying that move). The pseudo-code in listing 3 describes the implementation of the randomization method in Chess0.

```
find all possible moves
sort the moves from best to worst
while moves left do
    margin = best_possible_score − RAND_THRESHOLD
    if (current_move_score < margin)
        dismiss current_move
    else
        selectable_moves += current_move
done
selection = pick a random move from selectable_moves
return selection
```

Listing 3: Pseudo-code for the randomization process found in Chess0

According to the pseudo-code in listing 3, the chess engine behaves normally as it would do in a non-randomized minimax-like algorithm. However, instead of just keeping the best move (as was described in the negamax algorithm in combination with the alpha-beta pruning method), the chess engine remembers all the evaluated root nodes, associating them with their score. Afterwards, the engine discards the moves that are below the minimum allowed score, that is the best possible score minus the randomization threshold, thus performing a fully controlled randomized selection of the move.

It is important to note, nonetheless, that the randomization threshold may not affect the response in certain situations. One straightforward situation in which randomization may not apply is when all non-best scores are below the randomization threshold, thus leading to the selection of the best move for that response. As a result of the randomization, the chess engine typically behaves more imprecisely. This implies two important characteristics that the chess engine acquires: [21, 16-24]

1. The machine behaves more close to human-like behavior, due to the imprecisions in its decisions [21, 16-24].
2. The machine is capable of discovering new game lines that could not analyze further if there was no randomization, since it would always select the same response for the same given board situation [21, 16-24].

Consequently, the implementation of the randomization process in the adaptive AI engine enables Chess0 to play in a more human-like manner, as well as trying out different variants that might be interesting and were not considered when using a fixed-behavior classic AI engine.

4.3.2 Dynamic Heuristics

The heuristic evaluation is not completely precise, since it is typically implemented as a static evaluation function, meaning that the same heuristic thinking is applied to any board position at any point of the game. The static heuristic evaluation function is normally implemented in such a way that it is not concerned the previous states of the board nor possible future states. Therefore, the AI engine is not capable of having a proper understanding of the game flow. [22, 26-28]

In Chess0, in the classic AI engine, the heuristic evaluation function is straightforward. It simply evaluates the amount of material and the current position, without taking into account the changes on the board. On the other hand, the adaptive AI engine utilizes a reduced set of features that provides certain degree of dynamism to the decision-taking process. Some of the dynamism aspects taken into account in the adaptive AI engine in Chess0 are:

- game phase recognition: Chess0 applies different decision-taking techniques, depending on the game phase
- move ordering: Chess0 recognizes which moves are more important than others, analyzing the first, thus imitating human-like behavior
- dynamic quanta: Chess0 understands the distribution of the pieces on the board, not only statically, but also according to the chain of moves performed by the opponent and its plan to secure its king

As an initial approach, the classic AI engine is acceptable, as it provides a generic understanding of a given position. However, as the static heuristic evaluation function does not understand about the game flow, the AI engine would have to search using large depth limits, in order to discover an accurate response. This would require much computation, since the computation time increases exponentially as the decision tree depth increases. However,

using dynamic adaptations to the heuristic function, the AI engine may become aware of the overall situation on the board, and in fact of the overall game flow, thus generating more proper response in less time. [22, 26-28]

First, the adaptive AI engine implemented in Chess0 is aware of the game parts, meaning that it may recognize when a game is in the opening, in the middle-game or in the end, thus applying different heuristics. This is generally achieved by counting the number of pieces left on the board and their position with respect to their original starting position. Depending on the game phase, Chess0 is capable of using more or less processing time, due to the nature of the chess game.

For instance, in the opening, the chess theory states that it is most important to develop the minor pieces (the pawns, knights and bishops, as well as castling), whereas in the middle-game it is most important to control critical squares that may lead to an attack against the opponent's king [5, 8-34]. The adaptive AI engine in Chess0 uses less computation in the opening and a positionally stronger heuristic function, whereas it uses more computation in the middle-game and end-game, caring about the attack to the opponent's king. This phase recognition and variable-depth adjusting is exemplified in the code in listing 4 (this is an original part of the source code extracted from Chess0).

```
void Adaptive::configAutoDepth()
{
    // set the depth to its base value, before modifying it
    depth = odepth;

    // in the opening, use less depth
    if (board->getPhase() == PHASE_TYPE_OPENING)
    {
        depth -= AI_DEPTH_DEVIATION;
    }
    // in the end, increase depth as pieces decrease
    else if (board->getPhase() == PHASE_TYPE_END)
    {
        depth += (4 * AI_DEPTH_DEVIATION) - board->getNPieces();
    }
}
```

Listing 4: Chess0 C++ code for adjusting a variable-depth search

As can be perceived from listing 4, Chess0 uses a constant defined as *AI_DEPTH_DEVIATION*, which is a natural number representing the variance that can be applied to the search depth limit. For instance, consider a search depth set to 4 and an AI depth deviation set to 2. Then, in order to know the actual search depth for an end-game, equation (4) demonstrates that the search depth is then increased from its original value of 4 to an actual search depth value dictated by equation (5).

$$depth = initial_depth - ((4 * depth_deviation) - number_of_pieces_left) \quad (4)$$

$$depth = depth + (8 - number_of_pieces_left) \quad (5)$$

According to listing 4, and as exemplified in equations (4) and (5), the adaptive AI engine of Chess0 is capable of self-regulating the amount of computation, depending on its necessities. This feature only allows Chess0 to restrict the computation resources, which are, however, present in human-like thinking, since a human may not process unnecessary information depending on the phase of the game [23, 143-164].

A second feature present in Chess0 or more concretely in its adaptive AI engine, is the capability of ordering moves and recognize their significance, depending on the nature of the move itself, and its score. The technique used in Chess0 is named *Most Valuable Victim / Least Valuable Attacker* (MVV/LVA), which is an algorithm for ordering the moves taking into account the fact that less-valuable pieces attacking higher-value pieces may return a higher score. For example, if in the list of moves there is such a move that a pawn may capture a queen, then that move will be most likely to return a high score, since the pawn is the least valuable piece on the board, and the queen is the most valuable piece on the board. [24, 1-10]

The MVV/LVA method enables two important properties in a computer chess program. First, it boosts the performance of the search, since probable higher-score moves are evaluated before probable lower-score moves, thus speeding up the alpha-beta pruning, as the non-interesting parts of the tree are cutoff early in the search process. Second, the computer chess program is biased to behave closer to the human-like approach of

thinking, since human players first analyze the moves that seem of higher importance. [24, 1-10]

```
void Adaptive::initSearch()
{
    root_moves = board->getAllValidMoves(color);

    // 1) set the order for each move, before sorting them
    for (unsigned int i = 0; i < root_moves.size(); i++)
    {
        // if a move is not a capture, set the lowest priority
        if (!root_moves[i].isCapture())
            root_moves[i].setOrder(9999);
        else
        {
            int vo = 0; int vd = 0;
            Piece *po = board->getPieceAt(root_moves[i].getOrigin());
            Piece *pd = board->getPieceAt(root_moves[i].getDest());
            vo = po->getValue() * po->getColor();

            // now set the ordering number for future sorting
            root_moves[i].setOrder(1000 - vd + vo);
        }
    }

    // 2) sort the moves using MVV/LVA
    sort(root_moves.begin(), root_moves.end(), move_compare);

    // 3) sort moves according to their score in the previous search
    for (unsigned int i = 0; i < best_moves.size(); i++)
        for (unsigned int j = 0; j < root_moves.size(); j++)
            if (best_moves[i].equals(root_moves[j]))
            {
                Move tmp = root_moves[j];
                root_moves.erase(root_moves.begin() + j);
                root_moves.insert(root_moves.begin(), tmp);
            }
}
```

Listing 5: Chess0 C++ code for sorting the moves using MVV/LVA

As can be seen from listing 5, the implementation of the MVV/LVA ordering is simple, where the moves are first obtained, then rated according to their score and their nature,

and finally sorted from the most interesting to the least interesting. Finally, the alpha-beta pruning benefits from this ordering by cutting off the decision tree in less time. Astonishingly, this method, although simple, is a precise approach to human-like thinking strategies, thus providing the chess AI engine with a proper rational thinking flow. [24, 1-10]

Another feature present in the AI engine of Chess0 is the capability of using dynamic quanta for evaluating a board position. In principle, Chess0 uses, both in the classic AI engine and in the adaptive AI engine, a score-based square distribution on the board, meaning that each square on the board has a value or score (a *quantum*). This quantum is a number from $-\inf$ to $+\inf$, which provides an estimate of how valuable that square is for a player, where negative values are a positive result for the black player.

The classic AI engine in Chess0 uses such a feature. However, these quanta are not modified during the course of the game, and the values of the squares are the same throughout the game, where the central squares are the most valuable ones, according to the general rules of chess theory [5, 18-19]. On the other hand, the adaptive AI engine in Chess0 uses dynamic quanta, where the scores for each square are updated after each move on the board, thus adapting and conditioning the response generation to the current situation. Figure 8 illustrates a comparison between the static quanta and the dynamic quanta implementation in Chess0.

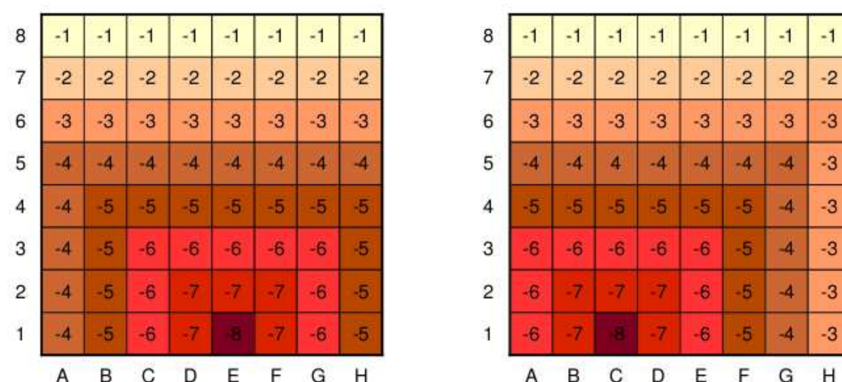

Figure 8: Colored diagram of different quanta within a same game in Chess0

Both boards in figure 8 are representations of the quanta according to the attack against the opponent's king, where the player to use these quanta is the black side, due to the negative sign of the squares values. On the first board, the opponent's king is on the square $e1$, since the maximum score is on that square. However, on the second board,

the opponent's king seems to be shifted onto square $c1$, probably because there was a long-castle before this situation.

It is important to note the tonality of the squares. The coloring scheme depicts how the scoring is re-arranged around the opponent's king, in the case of the attack quanta. This method is implemented in the adaptive AI engine of Chess0 using Euclidean distances. That means that the highest score is always assigned to the square where the opponent's king size is residing, and afterwards, the surrounding squares are given a lower score as they are placed farther from the highest-score square. In general, implementing this method requires from the application to recalculate the quanta for each player after each move, according to the current board situation. This feature adds dynamism to the AI engine, since it constantly verifies the current situation and then modifies its decision values.

4.3.3 Quiescence Search and the Horizon Effect

Finally, one of the most important features of the adaptive AI engine in Chess0, is the implementation of the quiescence search. The quiescence search is a technique for obtaining proper evaluations in decision tree searches, where the search function is not accurate enough. Furthermore, the quiescence search solves the AI problem of the *horizon effect*, which is one of the main goals of current AI researches from a general point of view. [25, 5-15]

The horizon effect is a limitation to any AI application, by which the application is constrained in its perception of the reality, normally due to a computational conditioning. This means that the AI application is not capable of representing or understanding the reality beyond its perception, and hence its response to the environment is limited as well, according to the horizon effect. In computer chess it is easy to understand the nature of the horizon effect, as it is conditioned by the search depth limit on the decision tree. The move tree is illustrated in figure 9. [25, 7-15]

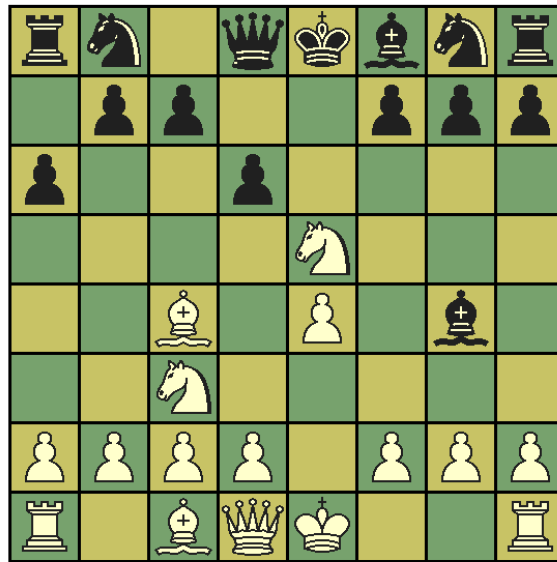

Figure 9: Demonstration of a typical non-quiescent position

In figure 9, there are two possibilities for the black side, who is moving from that position. First, the black side can capture the white's queen, to which the white player would reply by capturing the pawn with its white bishop and the check mate is not avoidable after two moves. On the other hand, the black side can prevent this check mate by not capturing the white's queen.

Accordingly, if a computer chess program is commanded to evaluate that position and to decide for the black side, the response would completely depend on the search depth that is given to its algorithm. In the example presented in figure 9, a computer chess program must search at a depth of 4 at least, in order to avoid the check mate situation, which would result in the maximum lost. However, if the computer chess is programmed to search only until a depth of 3 plies or less, it will not discover the check mate, thus believing that capturing the white's queen will be the most valuable move response. [25, 5-15]

Exemplified by figure 9, the horizon effect for a computer chess program is typically created from the search depth limit, which disallows the AI engine to see further in a game, when analyzing the decision tree up to some given depth limit. Anything that happens outside of that scope will not be of importance to the AI engine. Therefore, in order to avoid unexpected results in the heuristic search due to dramatic changes that are not studied out of the scope, this horizon effect must be palliated. The most common approach to avoid the horizon effect is to provide the AI agent with the capability of

sensing when a situation is stable or not. This is due to the fact that stable situations are not likely to present dramatic changes in the near future, whereas unstable situations will most probably present a dramatic change in the environment in the near future. [26]

The minimax-like algorithm searches until a given depth and, when it reaches the terminal nodes of the decision tree, it ends up calling a quiescence search algorithm, which estimates if the position is stable or not. If a position is stable (quiet position), it can be evaluated with a simple heuristic function, since it will not present dramatic changes in the near future. However, if a position is not stable (not quiet), the simple heuristic function will not provide an accurate result. Hence the tree must be traversed deeper, breaking the search depth limit and crossing over the line drawn by the horizon effect. A simple implementation of the quiescence search extension to the minimax-like algorithm is presented in code listing 6. [25, 5-15]

```
function negamax(tree_depth):
    if tree_depth is 0: return quiescence();
    best_score = -infinity;
    for all valid moves as cursor do:
        apply_move(cursor);
        myscore = -negamax(tree_depth - 1);
        undo_move(cursor);
        if myscore is greater than best_score:
            best_score = myscore;
    return best_score;

function quiescence():
    best_score = alpha;
    for all valid moves as cursor do:
        apply_move(cursor);
        if cursor is (capture & check & promotion):
            myscore = -quiescence();
        else
            myscore = evaluation;
        undo_move(cursor);
        if myscore is greater than best_score:
            best_score = myscore;
    return best_score;
```

Listing 6: Chess0 negamax pseudo-code, combined with a quiescence search

As is presented in the pseudo-code in listing 6, the quiescence search is a function similar to the negamax function, however it has no depth search limit. The quiescence search basically searches for a quiet node in the decision tree, meaning that the move under analysis must not be a capture, a check nor a promotion. If the move is one of those critical moves, then the quiescence will continue iterating by calling itself. However, if the quiescence search finds a node whose move is not one of the critical moves, then it will return the normal evaluation, without recursing over itself anymore. This behavior is further exemplified in figure 10. [26]

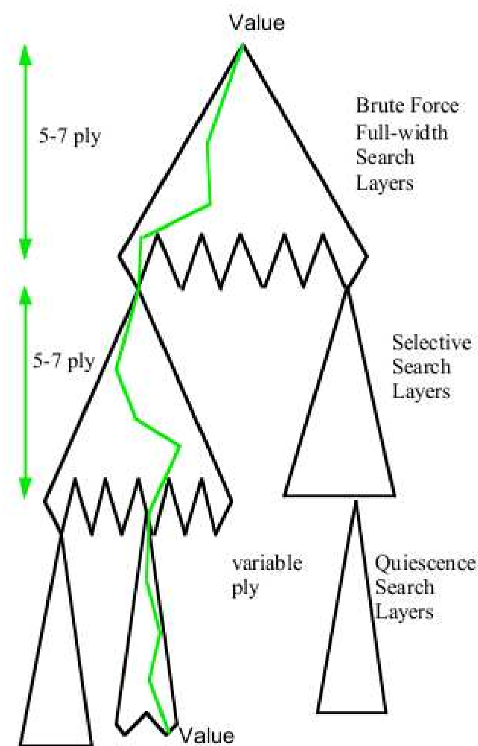

Figure 10: Quiescence search impact on the decision tree traversing [25]

The example in figure 10 presents a decision tree in which the search depth limit has been set to seven plies. Nevertheless, some of the terminal nodes in the decision tree must present critical moves, such as captures, checks or promotions), since the decision tree is expanded in one more ply at some of its terminal nodes. This is the effect of calling a quiescence search function inside the minimax-like algorithm. The quiescence search will definitely ensure that the heuristic evaluation takes place only at stable nodes. [25, 5-15]

Last, it is important to note that the quiescence must be defined to the AI agent, in order to make it understand what a stable situation is and what an unstable situation is. Once these two concepts are defined to the AI agent, it will be provided with a simulation of sense, by which it will decide whether to evaluate the current situation or not, or rather go further in its analysis. For computer chess, this definition is given by the type of move under analysis. However, for any other application, the quiescence search must be researched, and adapted accordingly to the specific AI application. [27, 321-322]

5 Results

5.1 Randomization Tests

The first experiment demonstrates the randomization mechanism used by the adaptive AI engine. The classic AI engine is fixed in behavior, and thus it always selects the move with a higher score. Practically, the classic AI engine will play the same game indefinitely when playing against itself. On the other hand, the adaptive AI engine will introduce random alterations, thus varying the path of the game. Figure 11 shows the position from which the classic and the adaptive engines are asked to provide a response.

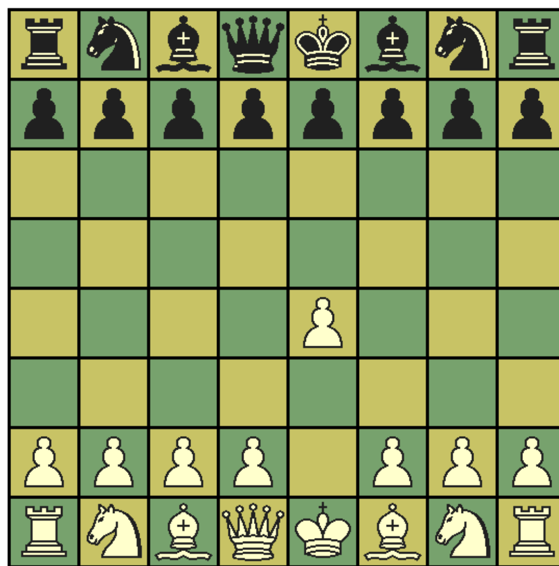

Figure 11: Initial position: black to move against the move 1.$e4$

Parting from figure 11, there the black side has 20 different replies to the move. After Chess0 performs a search on its decision tree, it selects the move with higher score, when using the classic AI engine. However, when using the adaptive AI engine, Chess0 may pick other moves close to the higher-score move, but not necessarily the move with the highest score in the search. These data is studied in table 1, with a sample of 100 different games.

Table 1: Adaptive AI randomization for generating move responses

Classic AI Response	**Adaptive AI Response**
1.. e4 100%	1.. e5 20% 1.. e6 40% 1.. d5 40%

The examples presented in table 1 are a result of asking both Chess0 AI engines for a response to the same move. The classic AI engine provided always the same response to the same initial move, whereas the adaptive AI engine presents different move response with a different ratio. These data were gathered using a randomization threshold of 0.50. Admittedly, the wider the randomization threshold, the larger number of different moves would be provided by the adaptive AI engine.

The randomization threshold allows the chess engine to play using a wider variety of moves, thus being able to explore different branches in the decision tree. Figure 12 plots a simple graph showing the relation between the randomization threshold and the wideness of the variety in move responses.

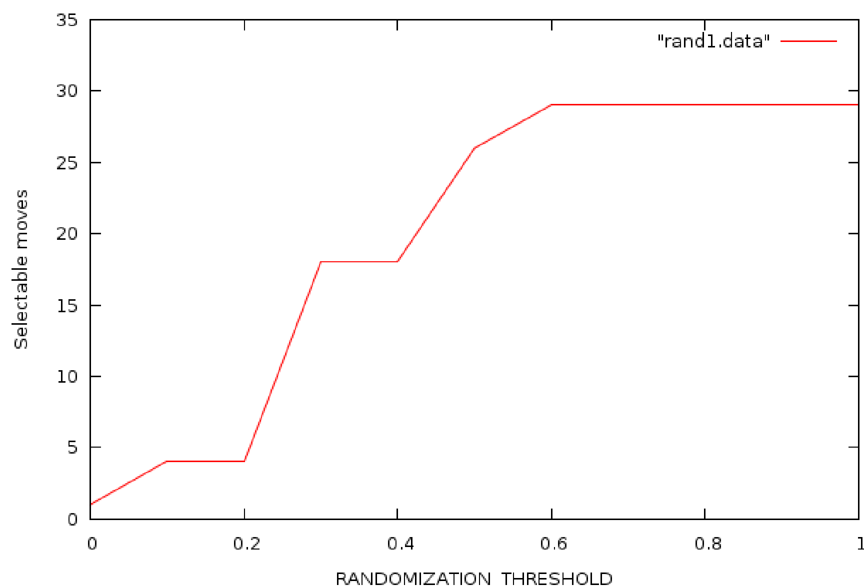

Figure 12: Change in the variety of moves respect to the randomization threshold

According to the data exposed in figure 12, the randomization threshold plays an important role in the decision taken by the chess engine. These data are taken from a response given by the adaptive AI engine in Chess0 when presented with a certain situation in the board. In that board situation, the black moves, having many possible move responses, due to the distribution of the pieces on the board.

Reflected in the graph in figure 12, the number of selectable moves are the moves that are considered close to good, when compared to the move with the highest score possible. This decision behavior is hence dictated by the randomization threshold. As the randomization

threshold increases, the range of selectable moves widens as well, thus providing a larger number of choices to the chess engine.

Astonishingly, figure 12 depicts an horizontal asymptotic bound for the number of selectable moves, whose scores lie 0.60 from the best score. On one line, it is possible that the amount of total valid moves has been reached already. On the other line, it is possible that the chess engine needs a much higher randomization threshold in order to allow worse moves to be selectable as well.

From the experimental point of view, it is also important to analyze the probability of victory and defeat when applying a randomization threshold to the adaptive AI engine. Figure 13 synthesizes a collection of data gathered from running dozens of games using different randomization threshold, where the x-axis represents the value of the randomization threshold applied on each game, and the y-axis represents the percentage of victories after playing a total of 10 games.

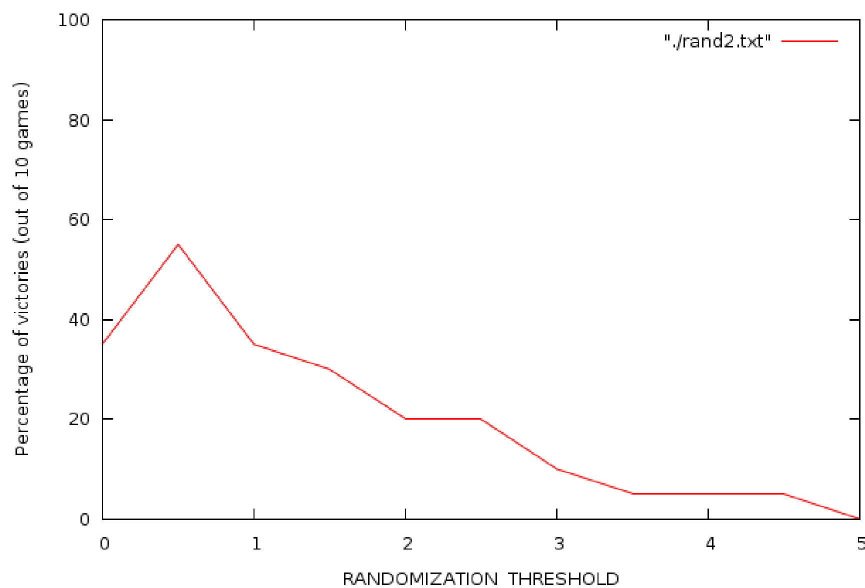

Figure 13: Efficiency of the randomization threshold, in terms of game wins

Interestingly, the data presented in figure 13 suggests that the Chess0 adaptive AI engine plays in a more efficient manner when the randomization threshold lies between 0.00 and 1.00, optimally 0.50. According to the data in figure 13, from 0.00 to 0.50, approximately, the efficiency of the program increases. Nevertheless, after 0.50, as the randomization threshold increases, the efficiency of the program decreases. Thus, the interesting part of

the analysis is the range of values for the randomization threshold lying between 0.00 and 1.00. Please notice the difference in scales between figures 12 and 13, which is necessary in the latter, in order to depict the overall behavior after several games.

One explanation for the behavior depicted in figure 13 is that the heuristic evaluation of Chess0 is not perfect, due to its heuristic nature. Therefore, when the randomization threshold is zero, meaning no tolerance at all, the program may avoid analyzing any branch that is not evaluated with the highest score. Nonetheless, the program might avoid analyzing branches that further lead to more efficient results, since the evaluation has a certain degree of inaccuracy. On the other hand, when the randomization threshold is close to 0.50, the program will allow itself to analyze different branches, thus being able to discover more efficient decisions.

Finally, and as shown in figure 13, when the randomization threshold is too large, meaning greater than 0.50, the decisions taken by the program are of a low score, thus leading to non-efficient results. Furthermore, much randomization decreases the efficiency of the program in a quasi-linear manner, according to the graph in figure 13. Hence, it can be concluded from this experiment that the randomization threshold is somewhat dependent on the heuristic function of the program, where 0.50 is the approximate optimal value for the concrete implementation in Chess0.

5.2 Dynamic Heuristics Tests

The technique of using dynamic quanta when assigning a certain score to each square on the board has an impact on the overall result of the chess engine decisions. In Chess0, the classic AI engine does not use dynamic quanta, and hence the evaluation is fixed, weighing the central squares more than the rest of the squares. However, the adaptive AI engine modifies the quanta on the squares, depending on the position of the enemy's king and thus giving more preference to aggressive positions, towards attacking the opponent's king. Figure 14 provides an example of the position.

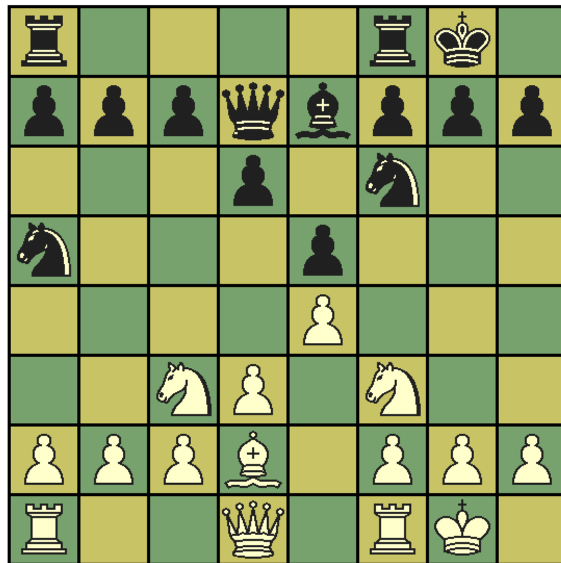

Figure 14: Middle-game position, white moves

In the position given by figure 14, the white moves in the middle of the game. According to the implementation of the classic AI engine, the machine should evaluate the central squares as the most valuable, whereas the adaptive AI engine should consider more profitable the moves closer to the opponent's king, due to the different quanta for different AI engines. Table 2 shows a list of the move evaluations of both types of the engine.

Table 2: Different evaluations of moves for a given position

Move	**Proximity**	**Classic Evaluation**	**Adaptive Evaluation**
Rb1	1	0.60	0.58
Rc1	1	0.62	0.56
Qb1	1	0.54	0.52
Qc1	1	0.58	0.54
Qe1	1	0.63	0.60
Qe2	2	0.67	0.66
Re1	1	0.63	0.60
Kh1	1	0.53	0.34
a3	2	0.50	0.36
a4	2	0.52	0.45
b3	3	0.51	0.38
b4	3	0.53	0.38
Bc1	1	0.48	0.56
Be1	1	0.49	0.58
Be3	3	0.61	0.66
Bf4	4	-2.52	-2.59
Bg5	**5**	**0.58**	**0.75**
Bh6	**6**	**-2.26**	**-2.32**
g3	3	0.51	0.63
g4	4	-0.73	-0.79
h3	3	0.50	0.61
h4	4	0.52	0.66
Nb1	1	0.42	0.48
Ne2	2	0.58	0.51
Na4	2	-2.88	-2.94
Nb5	3	-2.67	-2.91
Nd5	**5**	**0.42**	**0.51**
d4	4	0.41	0.49
Ne1	1	0.46	0.47
Nd4	4	-3.13	-3.25
Nh4	4	0.31	0.62
Nxe5	**5**	**-2.40**	**-2.58**
Ng5	**5**	**0.49**	**0.69**

The data in table 2 represents a sample of decisions from the classic AI engine and the adaptive AI engine when taking a decision given the same situation. The moves with higher proximity to the opponent's king are marked in bold. Generally, the adaptive AI engine evaluates moves with higher proximity as more valuable moves. However, this fact does not necessarily happen at every moment. For instance, moves involving captures or decisions with higher preference than just position may overcome the effect of the dynamic quanta.

Another feature that Chess0 implements is the *MVV/LVA* move ordering technique. As explained in chapter 4, this technique is an aid for sorting the moves between different search levels in the decision tree. Moving the "most-likely-to-be-best" moves to the beginning of the tree, the alpha-beta algorithm improves its speed, since it is most likely that the cutoff will occur at the very beginning of the tree. Therefore, if such a cutoff takes place early in the decision process, the volume of data to be processed will be reduced dramatically. The position is illustrated in figure 15, which presents a classical chess problem named *smothered mate*. [24, 1-10]

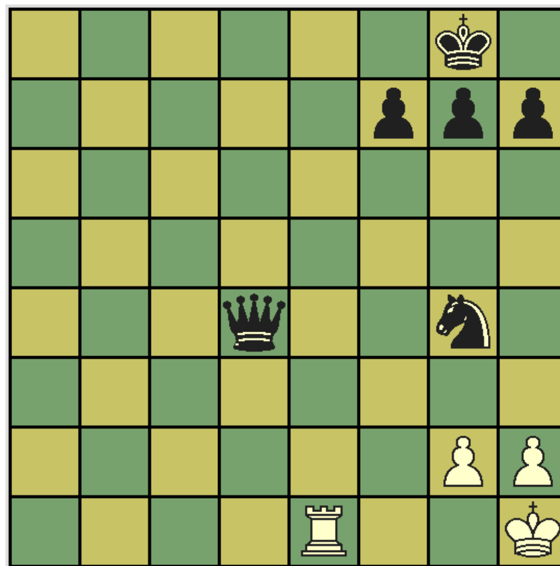

Figure 15: Smothered mate, a classical simple chess problem [28]

In order to test the improvement of the *MVV/LVA* algorithm, Chess0 provides a command named "solve", which is given a certain position and the chess engine tries to find which move will lead to the victory for the moving side in the minimum number of moves. In the smothered mate, presented in figure 15, the black moves to mate the white, and the check-mate comes after four moves, and it is unavoidable after the black plays *Nf2*. This

experiment was run using the classic AI engine, and then using the adaptive AI engine. Figures 16 and 17 depict the running speed of the algorithm compared to the search depth used when solving the problem.

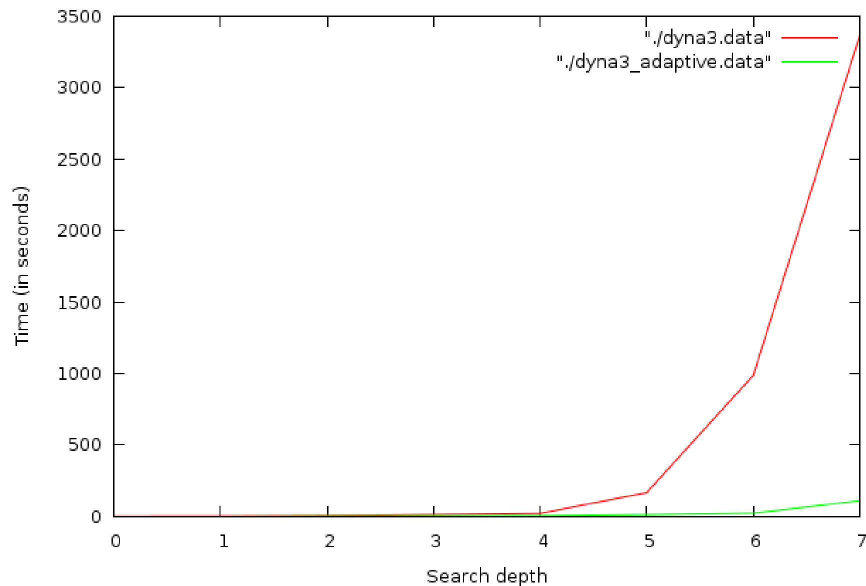

Figure 16: Search time in seconds, classic pruning versus MVV/LVA

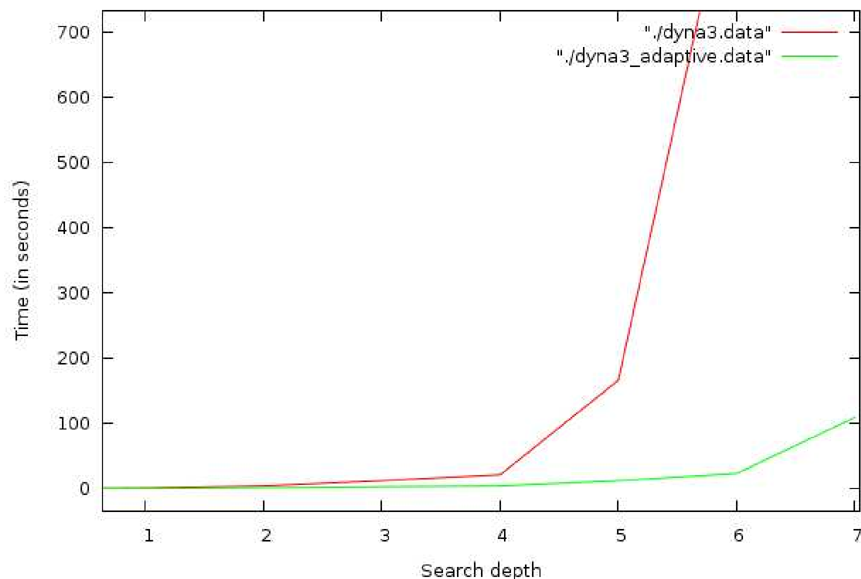

Figure 17: Search time in seconds, classic pruning versus MVV/LVA (zoomed)

In figures 16 and 17, the red line represents the classic AI engine search process, using standard alpha-beta pruning without any move ordering technique. On the other hand,

the green line represents the alpha-beta pruning algorithm aided by the *MVV/LVA* move ordering technique. In figure 16, it is noticeable that the search in the classic AI engine takes more than 55 minutes, whereas the search in the adaptive AI engine does not exceed 3 minutes. This is evidently a vast improvement.

In practice, the classic AI engine searches for a decision on the tree using different depths. However it looks for the same moves in the same order at any search depth. On the contrary, the adaptive AI engine analyzes every move at depth 1, and then sorts the moves by their probability of being the best move. On search depth 2, the adaptive AI engine already performs a cut-off at the beginning of the tree, thus reducing dramatically the amount of data to be processed. Further, this cut-off at an early stage continues at every search depth, thus making the adaptive AI engine to analyze uniquely the first branch of every tree, and avoiding the rest of the calculations.

Figure 17 is a magnification of the results presented in figure 16. The exponential growth in computation resources is observed from search depth 4, where the function starts to increase the Y-values rapidly. The results in figure 16 and 17 conclude that the *MVV/LVA* move ordering algorithm is an actual improvement, making the engine up to 30 times faster, depending on the search depth at every decision.

5.3 Quiescence Search Tests

The quiescence search is one of the most interesting features implemented in Chess0, since it provides not only proper efficiency, but actual human-like behavior. In order to test the quiescence search capabilities of Chess0, there are three experiments which tell the functionality capabilities, the functionality impact and its overall behavior improvement over the classic AI engine. The first experiment consists of querying both AI engines (classic and adaptive) over a same position, using different search depths, in order to obtain an idea of their decision making process. Figure 18 presents an interesting position.

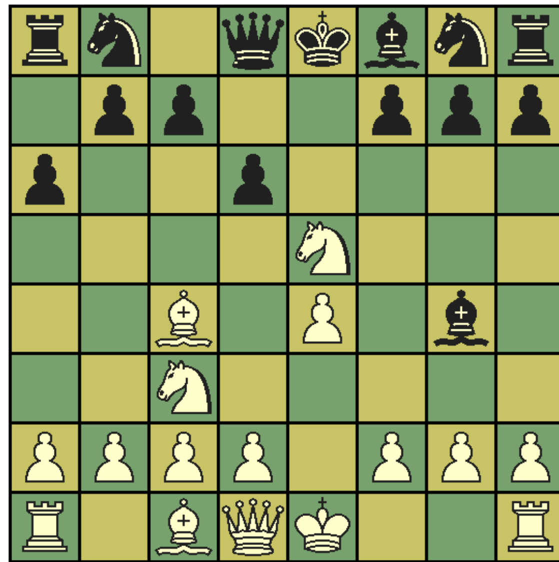

Figure 18: Demonstration of a typical non-quiescent position

In the position on figure 18, the black moves to capture the white's queen or avoid check mate. If black captures the queen of its opponent, the check-mate is unavoidable in four plies, giving the victory to the white. Therefore, a normal chess engine without quiescence search, should not be capable of discovering the check-mate if the search depth is set to less than four plies. The purpose of the quiescence search is to be able to search deeper in the decision tree, if the situation is not stable, such as the one in figure 18, where there are several captures and checks to the black king. Table 3 includes a collection of different responses for both AI engines at different search depths.

Table 3: Check-mate trap response, classic AI versus adaptive AI

Depth	**Classic AI Response**	**Adaptive AI Response**
1	1.. Bxd1	1.. Ke7
2	1.. Bxd1	1.. dxe5
3	1.. Bxd1	1.. dxe5
4	1.. Ke7	1.. dxe5
5	1.. dxe5	1.. dxe5
6	1.. dxe5	1.. dxe5

According to the data presented in table 3, the adaptive AI engine avoids the check-mate already at search depth of one, by moving the king. However, the classic AI engine needs a search depth of four plies, in order to discover the check-mate and perform the movement of the king that was detected by the adaptive AI engine already at search depth

1. By extension, the quiescence search discovers a better move already at ply 2, by taking the piece in $e5$. This move is discovered by the classic AI engine at search depth 5. Hence, this experiments demonstrates the efficiency of the quiescence search, even when using low search depth values.

As an extension to the normal search algorithm, the quiescence search requires more computation time, since it performs more calculations than a plain minimax-like algorithm. However, these calculations only take place when there are dramatic moves, which are those capable of altering the order of the position in a short period of time. Consequently, when the number of dramatic moves is considerably large, the quiescence search requires much computation time. The graph in figure 19 represents the computation time required by the quiescence search according to the number of dramatic moves in the decision tree.

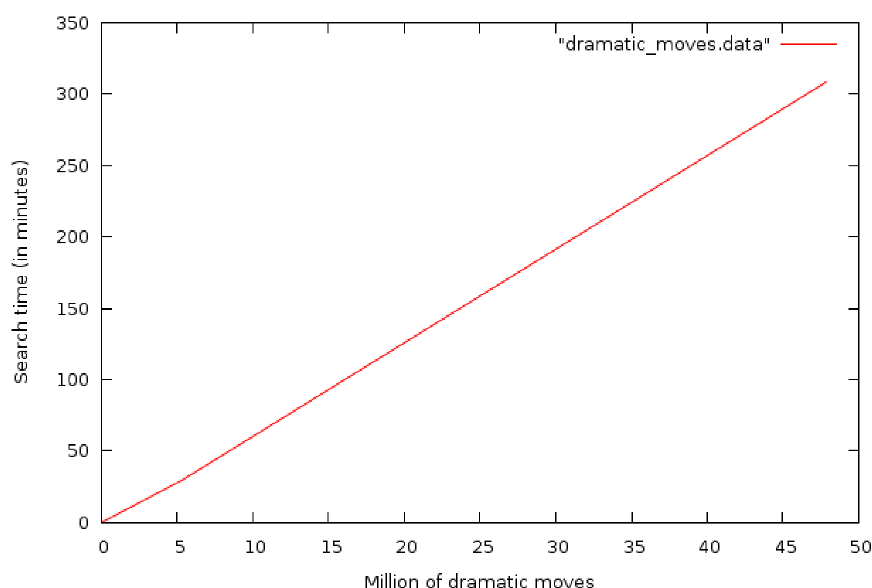

Figure 19: Search time per number of dramatic moves

It is noticeable from the data in figure 19 that the amount of time required to compute a quiescence search is roughly proportional to the number of dramatic moves found in the decision tree. There is, however, a slight inflection point at six million moves, which may be explained by the change in the depth search, thus introducing certain exponential growth at that point. The growth ratio is approximately of 100,000 moves per minute, meaning that Chess0 utilizes one minute to search through 100,000 dramatic moves using the current implementation of the quiescence search algorithm.

Finally, an important experiment is to confront the adaptive AI engine against the classic AI engine, thus demonstrating the overall behavior efficiency over the classic techniques. This experiment was run using different search depths for both engines. One part of the experiment was run with a quiescence search time limit, and the other one without any time limit. This is due to the fact that Chess0 implements a timeout feature in the quiescence search, in order to stop the search at some point, even when the depth limit has not been reached. Figure 20 depicts three experiments, where the time-constrained quiescence search is in green and red (green for a 30-seconds limit and 60-seconds limit) and the unlimited quiescence search is in green.

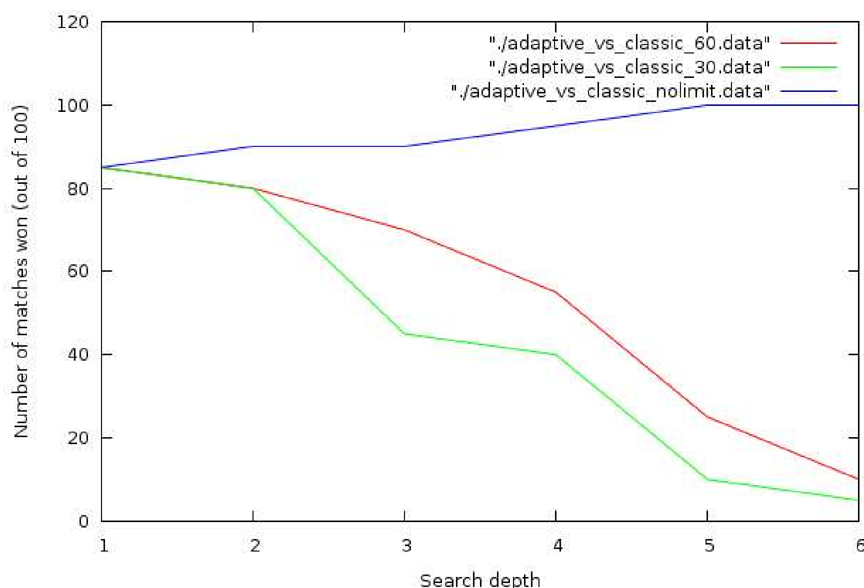

Figure 20: Number of victories (adaptive AI versus classic AI)

At first, it is noticeable that the time limit feature is an impediment for the perfection of the quiescence search. According to figure 20, the adaptive AI engine, running with a 30-seconds time limit, starts winning with a ratio of 85% against the classic engine at search depth 1. However, as the search depth is increased, the classic engine will start to take over the adaptive AI engine, since the classic AI engine does not have a time limit. As expected, when the time limit is increased to 60 seconds, the adaptive engine starts to play better. This is due to the fact that the quiescence search returns the best move found so far when the time limit is exceeded, and not the absolutely best move.

Furthermore, figure 20 depicts the improvement in the behavior of the adaptive AI engine when it is not limited in time. The adaptive AI engine always scores more than 90% of the victories after a search depth of two plies. The quiescence search with no time restriction is substantially slower than the one with a time limit, as explained in figure 19, due to the large number of dramatic moves along the decision tree.

6 Discussion

6.1 Achievements

The results of the experiments present some interesting arguments for discussing the comparison between the classic AI engine and the adaptive AI engine implemented in Chess0. As a recall for the purpose of the project, the aim was to create an improved version of a classic AI engine for a computer chess application. The purpose of the experiments was to compare the differences, both in behavior and performance, between the classic implementation and the improved version.

The three main differences between the classic AI engine and the adaptive AI engine are the randomization extension, the use of dynamic heuristics and the implementation of the quiescence search. Each improvement aims at providing dynamism to the behavior found in the classic AI implementation, thus turning it into a modern chess engine, with proper capabilities for simulating human-like behavior.

First, the randomization extension works, in principle, as expected, according to the results presented in chapter 5. The adaptive AI engine is, by using this randomization, capable of producing alternative branches and discovering new paths along a chess game. Moreover, the randomization threshold, which is a setting for fine-tuning the behavior of the adaptive AI engine, proved to be a working tool, as explained in figure 13.

Second, Chess0 received an important improvement by using dynamic heuristics. The most important components of such dynamism are the use of dynamic quanta for the heuristic evaluation (dynamic heuristics) and the technique of move ordering. As a result of the dynamic heuristics, Chess0 is capable of using human-like thinking, as it becomes aware of the position on the board and it reformats its decisions based on how the board is evolving during a game. These capabilities are working properly, as reported in the dynamic heuristics experiments. Nevertheless, in relation with what is depicted by the experiments, dynamic quanta only adds a slight bias to the decision algorithm.

In practice, the move ordering technique mainly provides performance improvements, speeding up the decision tree traversing. This is necessary since computation in the quiescence search is highly expensive. However, the main purpose of the move ordering technique was to provide human-like thinking to the chess engine, which was achieved, although not noticeable for the end-user.

Last, the quiescence search implementation of Chess0 is another feature that is working properly. This is proved in the last set of experiments, especially in the first experiment, where the adaptive AI engine is capable of discovering unstable situations already at search depth 1 in the decision tree. In addition to this, the quiescence search proved to be a more efficient solution, compared to the classic AI implementation, since it achieves more wins using the same search settings.

Generally, the adaptive AI engine implemented in Chess0 seems to work efficiently, and it is definitely an improvement over the classic AI engine. This is the conclusion of the experiments, at a glance. Nonetheless, there are several points that are arguable about the implementation of the improvements in the adaptive AI engine:

1. The scope of this project was limited in time and, therefore, the improvements over the classic AI engine leave room for further refinement, such as improving the accuracy of the heuristic evaluation.
2. The number of experiments does not suffice to fully determine if the adaptive AI engine is a complete improved replacement for the classic AI engine, although it proves to behave more properly in most situations.
3. Due to the low performance of the application, the number of the experiment runs has been kept low, typically to 10 runs per experiment. In fact, the search depth tested in the experiment does not exceed seven, which may not be conclusive for all cases, since most commercial AI engines use search depths of up to 15.

In addition to all the features included in the adaptive AI engine, machine learning is an extension that could have been designed and implemented into Chess0's adaptive AI engine. However, the time and resource constraints for this project set the development pace at more concrete areas for dynamism, such as quiescence search and dynamic heuristics. As a matter of fact, the sole component for achieving basic machine learning would have required the amount of time equivalent to develop the whole application of Chess0 as it is currently implemented.

6.2 Application Fields

Nowadays there are AI applications in any field, and not just in IT-related projects. More concretely, computational AI and dynamic behavior are required in, for example, general simulation, medicine researches, biotechnology. Furthermore, these technologies expand more rapidly due to the improvements and discoveries in computer science, where parallel computing allows AI systems to model human brains with large amounts of computation resources. [29, 2-86]

Chess0 includes several features in its adaptive AI engine that could be used in other fields. Chess0 is an example of adaptive AI demonstrating the use of dynamic heuristics and quiescence search. However, these features are well applicable to other fields. Dynamic heuristics is a vague emulation of the behavior occurring in neural networks. Neural networks have an initial configured behavior, and they modify these behaviorism rules as they face new situations. Similarly, the dynamic heuristics found in Chess0 adapts its way of thinking (and generating a response) as the situations on the board change.

On the other hand, the quiescence search is a technique that has not been put into practice in general AI fields, as it has been used in games and finding paths for a known set of variants in applications containing decision trees [26]. Technology experts discuss the possibility to integrate computational AI (with its intrinsic dynamic behaviorism capabilities) for simulating real-life situations where a decision must be taken and there are several risks to be evaluated. Computational intelligence may be a key for discovering which decisions lead to a higher margin of probability for a risk to take place, thus optimizing decision taking. [30, 357-373]

7 Conclusions

The goal of this project was to develop a chess engine consisting of two different AI components. First, a classic AI engine using basic techniques for decision tree traversing, and then an adaptive AI engine putting dynamic techniques into practice, thus providing the tools for comparing the improvements of the adaptive AI engine (with dynamic behavior) over the classic AI engine (with static behavior).

The outcome of the project was the creation of an adaptive AI engine with for Chess0. The results proved that it is possible to create a simplified model of a computational intelligent system with some basic dynamic behavior capabilities. However, the results also demonstrated the challenge of establishing a proper adaptive-behaviorism system, which may vary its response functionality while interacting with its environment. The results proved that it is feasible to program a dynamism model inside the system, as an extension to the classic AI engine, thus enabling the intelligent agent to provide human-like behavior when playing a game.

This project was limited both in time and resources, thus concentrating on a simple study about dynamic AI systems and their behavior, such as in computational intelligence. The constraints ranged from equipment lack to the availability of applying real-life tests to the application such as the Turing Test.

Finally, it is recommended to further develop the current implementation, probably by integrating parallelization support and a distributed data management system, hence enabling the intelligent agent to widen its dynamic behavior functionality and implementing basic learning capabilities. It would be a concrete point to be improved towards the Turing Test preparation and future computational extensions to the system.

References

1 Simpson, John and Weiner, Edmund. Compact Oxford English Dictionary of Current English. Oxford: Oxford University Press; 2005.

2 Engelbrecht, Andries P. Computational Intelligence, An Introduction, Second Edition. University of Pretoria, South Africa: Wiley; 2007.

3 Franklin, Stan. Artificial Minds. Cambridge, Massachusetts: The MIT Press; 1995.

4 Moursund, David. Brief Introduction to Educational Implications of Artificial Intelligence. Oregon, United States of America: University of Oregon; 2006.

5 Levy, David. The Chess Computer Handbook. London: B. T. Batsford Ltd.; 1984.

6 Nilsson, Nils J. Introduction to Machine Learning. Stanford, California: University of Stanford; 1996.

7 Morris, Rober. Deep Blue versus Kasparov: The Significance for Artificial Intelligence. United States of America: AAAI Workshop; 1997.

8 Hyatt, Robert. Chess program board representations [online]. University of Alabama at Birmingham; 2004.
URL: http://www.cis.uab.edu/hyatt/boardrep.html

9 Steinberg, Louis. Introduction to Artificial Intelligence: Lecture 5. Rutgers University: Dept. of Computer Science, Hill Center. New Jersey; May 13, 2002.

10 Willestofte Berg, Casper and Petersen, Hans Gregers. A simplex approach for the tuning of a chess evaluation function. Technical University of Denmark; January 23, 2006.

11 St. Denis, Paul and Grim, Patrick. Fractal Images of Formal Systems. The Journal of Philosophical Logic 1997, 26: 181-222.

12 Chan, Pui Yee; Choi, Hiu Yin and Xiao, Zhifeng. Data Structures and Algorithms: Topic #11: Game trees. Alpha-beta search [online]. School of Computer Science. McGill University; 1997.
URL: http://www.cs.mcgill.ca/˜cs251/OldCourses/1997/topic11/

13 Marsland, T.A. Computer Chess Methods. University of Alberta: Computing Science Department. Edmonton, Canada; December 15, 1990.

14 J. Nilsson, Nils. Introduction to Machine Learning Notes. Stanford University: Robotics Laboratory, Department of Computer Science. Stanford; December 4, 1996.

15 Silver, David; Sutton, Richard S. and Mller, Martin. Sample-Based Learning and Search with Permanent and Transient Memories. Unversity of Alberta: Department of Computing Science. Edmonton, Alberta; 2007.

16 DeLoura, Mark. Game Programming Gems. Charles River Media: United States of America; 2000.

17 Berent, Adam. Computer Chess Information and Resources: Move Searching and Alpha Beta [online]. ChessBin; February 11, 2009.
URL: http://www.chessbin.com/post/Move-Searching-Alpha-Beta.aspx

18 Tolun, Mehmet R. Artificial Intelligence: Game Playing. ankaya University: Department of Computer Engineering. Anakara, Turkey: February 3, 2007.

19 Jerz, John L. A Proposed Heuristic for a Computer Chess Program. Fairfax, Virginia: October 2, 2009. (needs correction)

20 Turing, Alan M. and Copeland, B. Jack. The Essential Turing: The ideas that gave birth to the computer age. New York: Oxford University Press; 2004.

21 Dresher, Melvin. The mathematics of games of strategy: theory and applications. Canada: The Rand Corporation; 1981.

22 Saariluoma, Pertti. Foundational analysis: presuppositions in experimental psychology. London: Routledge; 1997.

23 Saariluoma, Pertti. Chess and content-oriented psychology of thinking. University of Helsinki, Finland; 2001.

24 Boul, Marc and Zilic, Zeljko . An FPGA Move Generator for the Game of Chess. Montreal, Canada; 2002.

25 Marsland, T.A. and Bjrnsson, Y. Variable Depth Search. Alberta, Canada: University of Alberta; 1999.

26 Frayn, Colin. Computer Chess Programming Theroy: Quiescence Search [online]. Beowulf Computer Chess Engine; August 1, 2005.
URL: http://www.frayn.net/beowulf/theory.html#quiescence

27 Kent, Allen and Williams, James G. Encyclopedia of computer science and technology (volume 27, suppl. 12). CRC Press: United States of America; October 29, 1992.

28 Surratt, David. Smothered Mate [online]. Chessville; 2009.
URL: http://www.chessville.com/instruction/Smothered_Mate.htm

29 Reusch, Bernd. Computational Intelligence: Theory and Applications. Dortmund, Germany: Springer; 2001.

30 Chen, Zhengxin . Computational Intelligence for Decision Support. United States of America: CRC Press LLC; 2000.