INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Classical Checkers

João Carlos Correia Guerra

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Novembro 2011

“Once the game is over, the king and the pawn go back in the same box.”

Italian Proverb

Abstract

For the last 60 years, games have been an important research subject in the field of Artificial Intelligence. For some time the focus was mainly on creating strong computer Chess programs. With the rise of master-level game-playing programs the trend changed. Now, the focus is on determining which games can effectively be solved and on solving them. The recent breakthrough with Monte-Carlo Tree Search (MCTS) techniques in complex games, such as Go and Amazons, also led the research away from alpha-beta. However, MCTS has not yet beaten alpha-beta in more simple games, such as Checkers and Chess.

In this dissertation, we describe a computer program created to play the game of Classical Checkers, Turska, and, for the first time, we estimate and analyse the complexity of Classical Checkers. Additionally, some experiments were made with the program. The results are analysed in order to better understand how each one of the studied search algorithms and techniques behave in the domain under study, and how they can be improved. Given that the enhancements that take advantage of transpositions were the ones that accounted for more savings, we tried to improve them. We did so by trying to improve the information stored in the transposition table and how it was managed.

The program was evaluated by playing some matches against another Classical Checkers computer program. In short, the results indicate that Turska's evaluation function needs further work. This is re-stated in the future work, along with other research suggestions.

Keywords

Classical Checkers
Game Playing
Adversarial Search
Alpha-Beta Search

Resumo

Nos últimos 60 anos, os jogos, na sua generalidade, têm sido um importante tema de pesquisa na área da Inteligência Artificial. Durante muito tempo o objectivo era criar programas de computador que jogassem bem Xadrez. Com o surgimento de programas que jogam a um nível bastante elevado a tendência mudou. O foco está agora em determinar que jogos podem ser resolvidos e em resolvê-los. Os recentes avanços com o método de Monte-Carlo em procura (MCTS) em jogos mais complexos, como Go e o Jogo das Amazonas, colocaram a procura alfa-beta em segundo plano. No entanto, o MCTS ainda não bateu a procura alfa-beta em jogos mais simples, como o Jogo das Damas e o Xadrez.

Nesta dissertação descrevemos um programa de computador, denominado Turska, criado para jogar a variante clássica do Jogo das Damas. O programa foi também utilizado para, através de algumas experiências, se compreender melhor como cada um dos algoritmos e técnicas de procura estudados se comportam no domínio em estudo, e como podem ser melhorados. Tendo em conta que as técnicas que tiram partido de transposições foram as responsáveis por mais e maiores ganhos, tentámos melhorá-las. Pela primeira vez, a complexidade do jogo é estimada e analisada.

Para avaliar o nosso programa jogámos alguns jogos contra outro programa. Os resultados indicam que a função de avaliação do Turska precisa de ser mais trabalhada. Esta observação é reafirmada nas sugestões de trabalho futuro, juntamente com outras sugestões de investigação.

Palavras-Chave

Jogo das Damas
Jogos
Procura Adversarial
Procura Alfa-Beta

Preface

For the past year or so this thesis has been a constant part of my life. Now that it is finished, I wish to thank several people for helping and supporting me throughout the process.

First of all, I would like to thank my supervisor, professor Inês Lynce, for guiding me through the whole process and for commenting on the progress of the thesis every week. I would also like to thank professors Rui Prada and Fausto Almeida for their many comments and suggestions, which have greatly enhanced the quality of this dissertation. Furthermore, I would like to express my gratitude towards Veríssimo Dias for being available to discuss and share some of his Checkers knowledge, and for providing feedback on Turska.

Additionally, I want to thank my fellow friends and colleagues, who were also doing their master's theses, especially Miguel Miranda, for being a reliable beta tester and for the several discussions on research in general. Moreover, I thank my friend Céline Costa for helping me find a particularly nasty bug. I also want to thank Liliana Fernandes and Tiago Ferreira for reading drafts of this dissertation. Their comments and suggestions helped improve the readability of some parts of this document. In addition, I would also like to thank Catarina Correia Rocha for always being there when I needed a good friend to talk to, and for helping me take my mind off work when I needed to.

Last, but not least, I would like to thank my family for their continuous support during the whole studies.

Acknowledgements

The research reported in this dissertation is supported, in part, by Instituto de Engenharia de Sistemas e Computadores, Investigação e Desenvolvimento em Lisboa (INESC-ID), with financial support from the Fundação para a Ciência e a Tecnologia (FCT).

The research has been carried out under the auspices of the SAT Group at INESC-ID (Software Algorithms and Tools for Constraint Solving).

João Guerra

Lisbon, November 2011

Contents

List of Figures

List of Tables

List of Abbreviations

α-β	Alpha-Beta
AS	Aspiration Search
ETC	Enhanced Transposition Cutoffs
ID	Iterative Deepening
MT	Memory-Enhanced Test
MTD	Memory-Enhanced Test Driver Framework
NS	NegaScout
PNS	Proof-Number Search
PV	Principal Variation
PVS	Principal Variation Search
TNC	Total Node Count
TT	Transposition Table

1 Introduction

Games have been a leisure for human beings almost as long as civilisation exists. Not only are they fun to play, but they can also be challenging and competitive. Games have also been known to stimulate the human brain and help in the development of mathematical reasoning. Because of this, certain games, such as Checkers or Chess, have been taught in schools in some countries. In the past centuries, some games, most notably Chess, have gained a certain level of status and even a place in culture. Thus, there is a stimulus for people to excel in the games they like. Chess is the most extreme example, with lots of competitions and championships happening regularly all around the globe. The reputation for owning a Chess title is very high. Although in general most games continue to be played as a leisure, for some people it has become their profession, the best example being Poker. Given the increased interest and competitiveness, some games are being studied thoroughly. This led to an increased literature about some games. Additionally, over the past years the use of computers as a tool to study games has been increasing.

In the early 1950s, advances in the fields of computer science and artificial intelligence raised a special interest in games. The hypothesis of creating artificial players for games had just become real. A large number of game-playing programs has emerged ever since. A small number of games has been solved, that is, considering that the players play the best they can, the outcome of the game has been determined. Although in some games the artificial players have challenged and even conquered the human supremacy, in others they are no match for humans. The first computer program to officially conquer the human supremacy was Chinook. Chinook became the first computer program to win a championship title from a human back in 1994[1]. The game in question was English Checkers.

Games are interesting research subjects for several reasons:

- Games have a closed domain with well-defined rules, in contrast to real-world problems, which makes them easier to represent and analyse.
- Although most games are easy to learn, they are difficult to master. Creating a strong game-playing program can sometimes provide insights into how people reason, as the way game experts think is usually studied to devise knowledge that computers can understand.
- Games can be used to test new ideas in problem solving.

[1]Chinook did not actually won the title from the human. A new championship was created to address competitions between humans and computers, as the title of World Champion is usually disputed solely between human beings. Instead, Chinook was declared the World Man-Machine Champion.

Usually, advances in game research are only applicable to the game in study and to similar games. Only sometimes can the ideas, algorithms, and techniques be translated to other domains.

1.1 Problem Statement and Contributions

In game-playing research, and search research in general, the objective is to create a computer program that can autonomously solve a given problem. In our case the problem is to play the game of Classical Checkers. The main problem addressed in this thesis can be stated as:

Problem Statement 1 *How can we develop a computer program to play Classical Checkers?*

To answer this question we will study alpha-beta, a game-playing method that has been shown to perform well in Checkers. Then, we will use Chinook as a case study to better understand how alpha-beta and enhancements are used in practice, as well as other components, such as opening books and endgame databases. Following that, we will estimate the complexity of Classical Checkers and compare it to that of other games, particularly of other Checkers variants. To the best of our knowledge, this is the first attempt at estimating the complexity of Classical Checkers.

After building our Classical Checkers computer program, Turska, two other questions will arise:

Problem Statement 2 *How do the studied search algorithms and techniques behave in the domain under study, and how can they be improved?*

Problem Statement 3 *What game-specific knowledge is required to improve our program?*

To answer the first question we will compare several of the studied search algorithms and techniques. Additionally, we will experiment several replacement schemes for transposition tables, including a new one, and investigate some of the suggestions mentioned by Breuker for additional information in the transposition table [8]. The research will always be performed in the domain of Classical Checkers. To answer the second question we will gather some specific knowledge about the game, either by studying knowledge available in other Checkers variants or by obtaining feedback from a Checkers expert, Veríssimo Dias. We will evaluate the knowledge of our program by running some experiments and by playing some matches against another program.

1.2 Document Outline

In chapter 2 we present the game in which we focus, Classical Checkers, along with some formal background behind games: some of their features and categorisations. Then, we introduce the field of adversarial search, specialising in alpha-beta search. The chapter includes an explanation of bounds, the minimal tree, node types, search enhancements, and problems related with imperfect decisions in games. The chapter ends with the description of some relevant issues related with solving games and game complexity. The notions introduced in this chapter are used throughout the document.

In chapter 3 we take a look at Chinook, a state-of-the-art English Checkers computer program, and at the proof procedure used to weakly solve the game of English Checkers. Other components that are used in game-playing programs, such as opening books and endgame databases, are introduced as well. The game of English Checkers is also presented, along with an overview of the history of computer Checkers.

In chapter 4 we estimate and analyse the state-space and game-tree complexities of Classical Checkers, which are then compared to the ones of other games, particularly of other Checkers variants.

In chapter 5 and 6 we describe Turska's search and knowledge components, similarly to how it was done with Chinook, in chapter 3. In chapter 5 we analyse the results of the experiments made with several search algorithms and techniques. The purpose is to determine how the enhancements behave for different search depths and game phases, as well as the influence of the enhancements on the search performance. In chapter 6 we describe the program's evaluation function and quiescence search, and compare some versions of them. We conclude the chapter by analysing the matches played between Turska and Windamas.

We conclude this dissertation with chapter 7. We discuss the results obtained, draw some conclusions, and discuss future work directions.

In appendix A we explain the rules of Checkers, in brief, and introduce the standard board notations for both Classical and English Checkers. In appendix B we describe AND/OR trees and the basic Proof-Number Search algorithm. The reading of appendix A is required to better understand chapter 6, and the reading of appendix B is advised before proceeding with the second part of chapter 3.

2 Games and Adversarial Search

Since the birth of Artificial Intelligence games have been one of the main topics of research in the field. We start by presenting the game in which we will focus, Classical Checkers, along with its properties. Then, we introduce a method that allows computers to play games, minimax, and an improvement to it, alpha-beta. Next, we discuss some techniques to enhance alpha-beta and finish by describing some relevant issues related with solving games and game complexity.

2.1 Classical Checkers

Checkers[1] is a kind of board game played between two players that involves diagonal moves of uniform pieces and mandatory leap captures. Checkers is believed to originate from Alquerque[2]. The rules for the game of Checkers can be consulted in appendix A.

Classical Checkers, also known as Portuguese or Spanish Checkers, is the most played Checkers variant in Portugal, Spain, and in some countries in Northern Africa and Southern America. This variant is considered to be the oldest and thus is considered as the original Checkers game[3]. Some computer programs exist for this variant[4]. Profound, by Álvaro Cardoso, and Windamas, by Jean-Bernard Alemanni, are recognized as being the strongest [18].

2.1.1 Game Properties

In game theory, games can have several properties and categorisations:

- **Cooperative** (or **coalitional**) vs **non-cooperative**. The most basic entity in any game model is the player. A **player**, or **agent**, may be interpreted as an individual or as a group of individuals making a decision. Once the set of players is defined, we may distinguish between two types of models: cooperative and non-cooperative. In cooperative game models, players can form coalitions and are supposed to be able to discuss the situations and agree on a rational joint plan of action, an agreement that should be assumed to be enforceable. On the other hand, in non-cooperative game models, it is assumed that each player acts independently, without collaboration or communication with any of the others.

[1]Depending on the form of the English language used, the term used to refer to the game may differ. *Checkers* is the term used in American English, whilst *Draughts* is the one used in British English. In this document we will use the American term. In the literature, the terms *Checkers* and *Draughts* are often incorrectly used to refer to the games of English Checkers and International Checkers, respectively.

[2]As investigated by Arie van der Stoep, `http://www.draughtshistory.nl/origin.htm`.

[3]As investigated by Arie van der Stoep, `http://www.draughtshistory.nl/Evolution.htm`.

[4]A small survey can be found at `http://alemanni.pagesperso-orange.fr/apage_spa_e.html`.

- **Sequential** (or **dynamic**) vs **simultaneous** (or **static**). In sequential games, each player chooses his actions before the others can choose theirs, i.e. players take turns playing. An important property of sequential games is that the latter players have some information about the actions taken by the former players. Simultaneous games are the counterpart of sequential games in the sense that there is no sequentiality, and each player chooses his actions without any knowledge of the other players actions.

- **Deterministic** vs **stochastic**. A game is deterministic if there is no randomness involved in the game, e.g. the roll of a die. If a game includes a random element, then it is stochastic.

- **Perfect information** vs **imperfect information**. A game is said to have perfect information if each player, when making any decision, is perfectly informed of all the events that have previously occurred. Checkers is an example of such game as each player can see all the pieces on the board at any time. A game has imperfect information if it allows players to have only partial or no information about the actions taken previously when making decisions.

- **Zero-sum** vs **non-zero-sum**. Informally, in a zero-sum game, the gain or loss of a player, at any moment in the game, is balanced by the losses and gains of the other players, at that same moment. This means that if the gains and losses of all players are summed, the result will be equal to zero[5]. In non-zero-sum games, the result of the sum will be different from zero.

Game theory and many of the previous concepts were first introduced and developed by von Neumann and Morgenstern [61]. Since then, game theory has been a field under constant development.

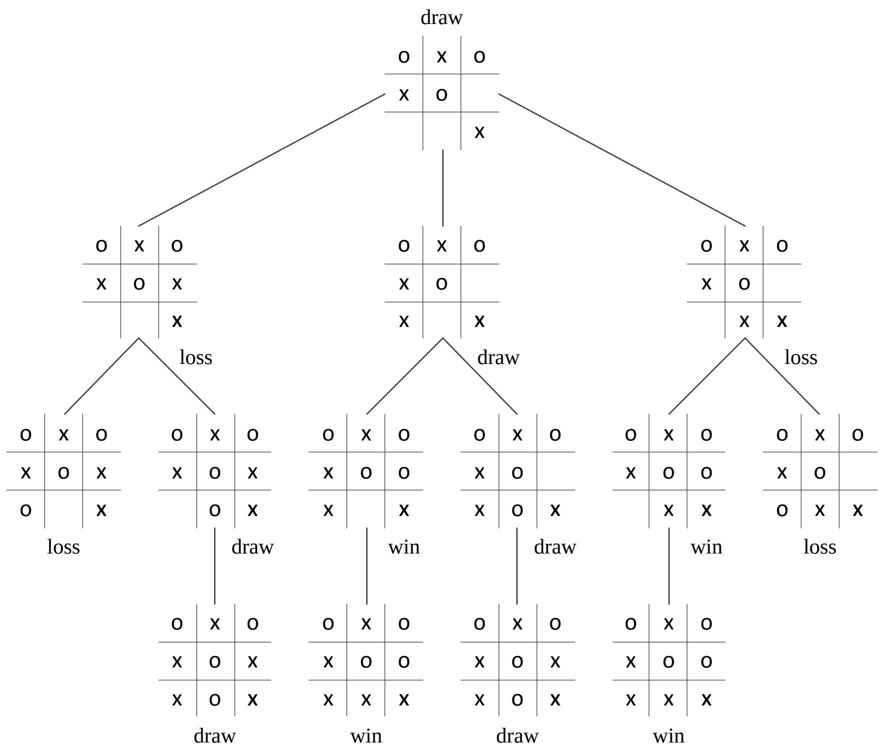
Checkers is a two-player deterministic zero-sum non-cooperative sequential game with perfect information. Games with these properties are called **abstract strategy games** and are studied within **combinatorial game theory** [16, 4], which is a branch of game theory.

2.2 Adversarial Search

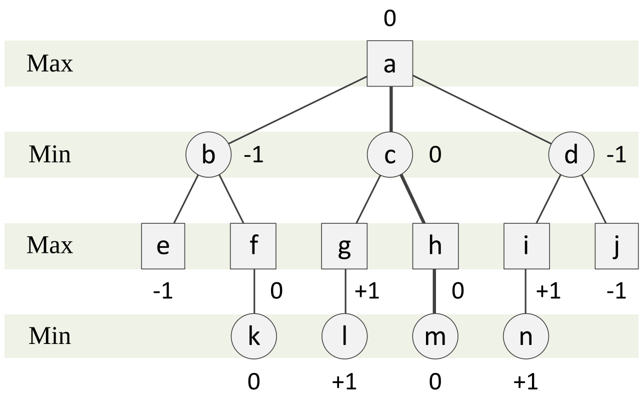
Humans play games by considering different combinations of possible moves for themselves and possible replies, or counter-moves, by their opponents. Each legal combination of moves and replies is a possible line of play. The most probable line of play is called the **principal variation** (**PV**).

Computers try to mimic humans by systematically analysing the different lines of play and building a **game tree** with that information. As an example, let us look at Tic-Tac-Toe. Figure 2.1(a) shows the possible lines of play from a certain Tic-Tac-Toe game position. The root of the tree is the current position, the nodes are legal positions, and the branches are legal moves. Leaf nodes are called **terminal nodes** and represent positions where the game ends. Considering the possible lines of play to find the best one is usually referred to as **searching the tree** and a move by one player is called a **ply**.

[5]Assuming that the gain of a player is represented by a positive number and the loss by a negative one.



(a)



(b)

Figure 2.1: The possible lines of play from a certain Tic-Tac-Toe game position (a) and its minimax tree (b).

2.2.1 Minimax

In two-player zero-sum non-cooperative sequential games with perfect information, the first player is called **Max** and the second, Max's opponent, is called **Min**. A **minimax tree** is a game tree where each node is annotated with its minimax value. The **minimax value** of a position is the utility of that position for Max. The **utility**, or **payoff**, of a position is a numeric value that represents the best possible game outcome from that position for a certain player. For example, in Tic-Tac-Toe, as in Checkers, there are three possible outcomes for a game: win, loss, or draw, with utilities $+1$, -1, and 0, respectively. There are games with a wider variety of possible outcomes. Figure 2.1(b) shows the minimax tree for our

Tic-Tac-Toe example. Nodes where it is Max's turn to play are depicted with squares, nodes where it is Min's turn to play are depicted with circles.

The minimax value of a position can be computed with the following formula:

$$\textsc{Minimax}(\textit{position}) = \begin{cases} \textsc{Utility}(\textit{position}) & \text{if } \textit{position} \text{ is terminal,} \\ \text{Maximum}_{p \in \textsc{Successors}(\textit{position})} \textsc{Minimax}(p) & \text{if } \textit{position} \text{ is a Max position,} \\ \text{Minimum}_{p \in \textsc{Successors}(\textit{position})} \textsc{Minimax}(p) & \text{if } \textit{position} \text{ is a Min position.} \end{cases}$$

Minimax uses backward induction to compute the value of a position. It starts by computing the values of the terminal positions and then backs up these values using maximum and minimum operations. When all positions have been analysed, we have the minimax value of the tree (n.b. the minimax values of the tree and of the root are the exact same thing). Figure 2.1(b) shows an example of a minimax tree with value 0. The principal variation is shown with a thicker line. Note that the minimax value of a tree is a function of its leaves and not of the root. The principal variation is the path from the root to the leaf that gives the root its value. There may be more than one principal variation as it is possible that more than one leaf propagates its value back to the root.

Minimax assumes that both players play **optimally**, i.e. every player always tries to maximize its utility. If one of the players does not play optimally, then the other should perform as good or even better. Although there may be other strategies that perform better than minimax against suboptimal, or fallible, opponents, we will consider that the opponent always plays optimally. Notice that Max's goal is to maximize its own utility, whilst Min's goal is to minimize Max's utility. To obtain the utility of a position for Min we only have to negate its value for Max, due to the zero-sum property. In fact, the names of the players come from this relation between their utilities. It is also this opposition between the players utilities that makes the situation adversarial.

The name *minimax* comes from the fact that each player maximizes its own utility while minimizing the opponent's utility at the same time. Although minimax was first introduced by Zermelo [63], only years later was the minimax theorem correctly described [61].

If we take advantage of the zero-sum property of games, we can reformulate minimax slightly differently. This new formulation is called **negamax** [28]:

$$\textsc{Negamax}(\textit{position}) = \begin{cases} \textsc{Utility}(\textit{position}) & \text{if } \textit{position} \text{ is terminal,} \\ \text{Maximum}_{p \in \textsc{Successors}(\textit{position})} - \textsc{Negamax}(p) & \text{otherwise.} \end{cases}$$

We should make clear that when using the minimax formulation Utility returns the utility of a position for Max, whilst when using the negamax formulation it returns the utility of the position relative to the side playing; otherwise the formulations are not equivalent.

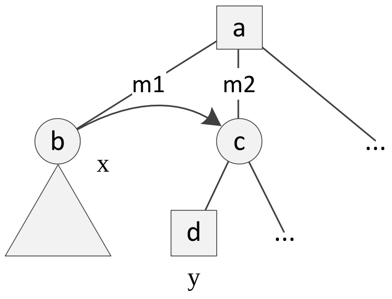


Figure 2.2: Value propagation in game trees.

Before continuing we should point out that we have been using the term *game* as it meant *abstract strategy game*. As in this document we are only concerned with abstract strategy games, more particularly the game of Classical Checkers, the algorithms and techniques we are going to present are introduced in this context. Nevertheless, some of the algorithms can be extended to other kinds of games. For the subset of games that can be represented with game trees, adversarial search is often referred to as **game-tree search**.

2.2.2 Alpha-Beta

To find the value of a game tree, minimax examines all of its nodes. In most cases, this is not only infeasible due to the massive number of nodes that may exist in a game tree, but also non-optimal because there are many nodes that do not affect the value of the root.

Alpha-beta (α-β) [28][6] is an improvement to the minimax algorithm that can possibly avoid the examination of the entire game tree. The idea is not to analyse parts of the tree that do not affect the value of the root. This technique is called **pruning**. Consider the example in figure 2.2. After finding the utility x of the first move, *m1*, Max should only be concerned with moves that have a utility greater than x, as he is trying to maximize his utility. Now consider the situation where Max makes a second move, *m2*, and the first reply of Min returns an utility y, such that $y \leq x$. Since Min is trying to minimize Max's utility, the eventual utility of *m2* will be at most y. Hence, Max will never play *m2* since it already has a better option, *m1*. In other words, node a, a Max node, has imposed a lower bound on its child c, a Min node. Conversely, a Min node can impose an upper bound on its Max children. Because of this, the alpha-beta algorithm introduces two bounds:

- The lower bound, α, equals the highest minimax value found so far, i.e. the minimum value that Max is guaranteed to have.
- The upper bound, β, equals the lowest minimax value found so far, i.e. the maximum value that Min is guaranteed to have.

The alpha-beta algorithm is shown in figure 2.3. $-\infty$ denotes a value that is lower than or equal to any possible utility for a terminal node, and $+\infty$ a value that is greater than or equal to any possible

[6]The early history of α-β is somewhat obscure. Knuth and Moore provide an overview of this history [28].

```
ALPHA-BETA(node, $\alpha$, $\beta$)
    if node is terminal then
        return UTILITY(node)
    if node is MAX then
        utility = $-\infty$
        for each node $n \in$ CHILDREN(node) do
            utility = MAX(utility, ALPHA-BETA($n$, $\alpha$, $\beta$))
            if utility $\geq \beta$ then // $\beta$ cutoff
                break
            $\alpha$ = MAX($\alpha$, utility)
    else // node is MIN
        utility = $+\infty$
        for each node $n \in$ CHILDREN(node) do
            utility = MIN(utility, ALPHA-BETA($n$, $\alpha$, $\beta$))
            if utility $\leq \alpha$ then // $\alpha$ cutoff
                break
            $\beta$ = MIN($\beta$, utility)
    return utility
```

Figure 2.3: The alpha-beta algorithm.

utility for a terminal node. The interval between α and β is called the **search window**. If the value of a node falls outside the search window, the corresponding subtree is pruned.

After searching a node using the alpha-beta algorithm, one of three things can happen [20]:

1. *utility* $\leq \alpha$. The result is below the search window (**failing low**).

2. *utility* $\geq \beta$. The result is above the search window (**failing high**).

3. $\alpha <$ *utility* $< \beta$. The result is within the search window (**success**).

If the value of a node falls outside the search window, it represents a bound on its minimax value, otherwise the value of the node has been found. Success can be assured by calling alpha-beta with the **full window**, i.e. $[-\infty, +\infty]$. The version of alpha-beta shown in figure 2.3 is called **fail-soft alpha-beta** [20] because it allows the search to return values that lie outside the search window (bounds) when the search fails, either low or high. Other versions of alpha-beta return α or β in the same situations. In this document, we will only consider the fail-soft versions of the algorithms presented. The reason for choosing these versions will be given later.

Figure 2.4 shows a game tree annotated with its final α and β values. The alpha-beta algorithm updates the values of α and β as it traverses the tree (from left to right) and prunes away branches with values known to be worse than the current values of α or β. The initial values of α and β are $-\infty$ and $+\infty$, respectively. From this example, it is possible to see that the effectiveness of the algorithm is highly dependent on the order in which nodes are examined, e.g. if node h was examined before node g, there would have been no cutoff in the subtree rooted at h. Later we will see some techniques to improve the order in which nodes are examined, and thus increase the chance of cutoffs. Such techniques are called **move ordering** techniques.

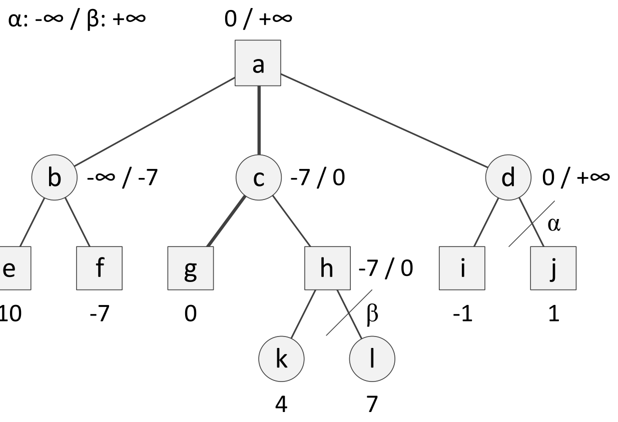


Figure 2.4: A game tree annotated with the final values of α and β. In each cutoff, the type is indicated. The principal variation is shown with a thicker line.

2.2.3 The Minimal Tree

The **minimal tree**, or **critical tree**, is the minimal part of a game tree necessary to determine the minimax value of the root. Thus, a game tree may have several minimal trees. The minimax value of the root depends only on the nodes of the minimal tree, the remaining nodes do not affect its value. For a game tree with fixed depth d and uniform branching factor w, the number of leaf nodes is w^d, and the number of leaf nodes of its minimal tree is $w^{\lfloor d/2 \rfloor} + w^{\lceil d/2 \rceil} - 1$ [28]. Although the minimal tree is a significant improvement over the minimax tree, its size is still exponential with respect to its depth.

In an alpha-beta minimax tree, we can identify three types of nodes [28, 32]:

- **PV**, or **type-1**. The root of the tree is a PV node. At a PV node all the children have to be explored. Furthermore, at least one of the children gives the parent its value. Pick one of such children. That child is a PV node, all the others are CUT nodes.
- **CUT**, or **type-2**. At a CUT node there is a cutoff. The child that causes the cutoff is an ALL node. If there are several such children, pick one arbitrarily. In a perfectly ordered game tree only one child of a CUT node needs to be explored. A **perfectly ordered game tree** is a game tree in which the best move is searched first and a cutoff occurs immediately after, at CUT nodes.
- **ALL**, or **type-3**. At an ALL node all the children have to be explored. Every child of an ALL node is a CUT node.

Before searching a node we do not know its type. We can only know the type of a node after it has been explored. Thus, before searching a node we refer to it as **expected-PV**, **expected-CUT**, or **expected-ALL** node. If there is no cutoff at an expected-CUT node, then the node becomes an ALL node. If at least one child of an expected-ALL node turns out not to be a CUT node, then the expected-ALL node becomes a CUT node. If all the expected-CUT nodes on a path from the root to a leaf become ALL nodes, a new principal variation emerges (all the nodes on the path have become PV

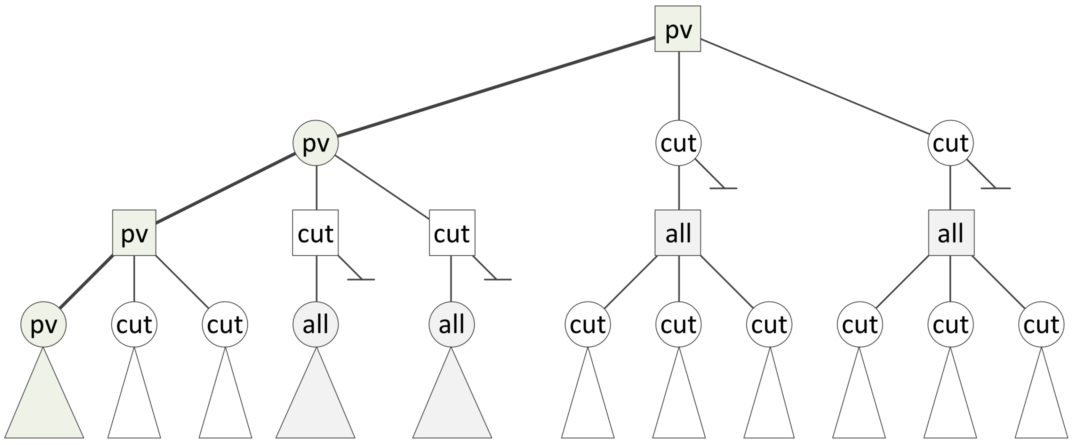


Figure 2.5: The structure of the minimal tree. The principal variation is shown with a thicker line.

nodes). Figure 2.5 shows the structure of the minimal tree. Every child of a PV or ALL node is a part of the minimal tree, but only one of the children of a CUT node is.

2.2.4 Imperfect Decisions

For the game of Tic-Tac-Toe, the game tree is small enough so that it can be entirely explored within a reasonable amount of time. In most games, this is not possible due to the massive size of the game tree. Thus, the search must stop at some point. When the game tree is too large to be generated completely, a **search tree** is generated instead. Instead of stopping only at terminal positions, the search can now stop at any position. Using this new approach, we may not be able to identify the utility of a position, and so we resort to an evaluation function. An **evaluation function** returns a heuristic assessment of a position, i.e. an estimate of the real utility of a position. For terminal positions the evaluation function should return the true utility, for non-terminal positions the value returned should be strongly correlated with the actual chances of winning. The range of values returned by an evaluation function is usually significantly wider than just win, loss, or draw. This allows for a better discrimination between moves. The value returned by an evaluation function is called **score**. Evaluation functions are necessarily application dependent. In Checkers, they typically include features such as material balance, piece mobility, and positional control, amongst others.

The new version of the (heuristic) alpha-beta algorithm is shown in figure 2.6. This version uses the negamax formulation. As with utility functions, if an algorithm uses the negamax formulation, the evaluation function must return the score relative to the side playing; otherwise the algorithm will be incorrect. Notice also that for the algorithm to be correct the values of α and β have to be swapped and negated at each recursion. This is a consequence of using the negamax formulation.

Heuristic Search Problems

One of the problems with heuristic search is how to decide when to stop the search and apply the evaluation function. Usually the search depth is used as the primary criterion. Another commonly used

```
ALPHA-BETA($node$, $\alpha$, $\beta$, $depth$)
  if $node$ is terminal or $depth == 0$ then
    return EVALUATE($node$) // or perform a quiescence search
  $score = -\infty$
  for each node $n \in$ CHILDREN($node$) and $score < \beta$ do
    $score$ = MAX($score$, −ALPHA-BETA($n$, $-\beta$, $-\alpha$, $depth - 1$))
    $\alpha$ = MAX($\alpha$, $score$)
  return $score$
```

Figure 2.6: The negamax version of the (heuristic) alpha-beta algorithm.

criterion is time control – stopping the search when the time run outs. A major depth-related problem is the **horizon effect** [5]. The horizon effect arises when some negative event is inevitable but postponable. Because only a part of the game tree has been analysed, it will appear that the event can be avoided when in fact it cannot. As the event is now over the horizon (beyond the depth of search), it is no longer a problem. Thus, the score reported for the move will not reflect its eventual outcome.

One way of mitigating the horizon effect is to increase the search depth. Another one is to perform a **quiescence search** [54] when the maximum search depth is achieved, instead of immediately evaluating the position. A quiescence search extends the search at a leaf position until a stable position is reached. A **stable**, or **quiet**, position is one that has a score that is unlikely to exhibit wild swings in the near future. For example, in Checkers, a position in which the opponent is threatening to capture several pieces is unstable. It would not make sense to stop the search in such a position, as the score returned by the evaluation function would not take the loss of the pieces into account, as the capture is over the horizon.

In general, the quality of play improves with the depth of search, at least up to some point (see, e.g., [26]). However, there are some games where the opposite is true. This phenomenon is called **search pathology** [33] and is not observed in most common games, such as Checkers, Chess, and Othello.

2.3 Search Enhancements

The performance of alpha-beta depends largely on the order in which positions are examined. In the worst case alpha-beta examines the entire game tree, while in the best case only the minimal tree is explored. We will discuss some techniques to enhance alpha-beta in order to bring it closer to the theoretical limit.

2.3.1 Transposition Tables

In games, different sequences of moves may lead to the same position. Such sequences are called **transpositions**. Thus, game trees are not really trees but instead graphs, as some positions appear in various places throughout the tree (e.g. in figure 2.1 nodes k and m represent the same position, as well as nodes l and n). If we detect these repeated positions, we can eliminate the redundant effort that is wasted by re-searching them. A **transposition table (TT)** [22] serves as a cache of recently explored positions.

```
ALPHA-BETA($node$, $\alpha$, $\beta$, $depth$)
*   $old\text{-}\alpha = \alpha$
*   $tt$ = RETRIEVE($node$)
*   if $tt.depth \geq depth$ then
*       if $tt.flag$ == LOWER-BOUND then $\alpha$ = MAX($\alpha$, $tt.score$)
*       else if $tt.flag$ == UPPER-BOUND then $\beta$ = MIN($\beta$, $tt.score$)
*       else /* $tt.flag$ == EXACT-VALUE */ return $tt.score$
*       if $\alpha \geq \beta$ then return $tt.score$

    if $node$ is terminal or $depth == 0$ then
        return EVALUATE($node$)

*   if $tt.move$ then // try the move from the transposition table first
*       $score = -$ALPHA-BETA($tt.move$, $-\beta$, $-\alpha$, $depth - 1$)
*       $move = tt.move$
*       $\alpha$ = MAX($\alpha$, $score$)
    else
        $score = -\infty$
        $move$ = NULL

    for each node $n \in$ CHILDREN($node$) $\setminus$ $tt.move$ and $score < \beta$ do
        $temp = -$ALPHA-BETA($n$, $-\beta$, $-\alpha$, $depth - 1$)
        if $temp > score$
            $score = temp$
            $move = n$
            $\alpha$ = MAX($\alpha$, $score$)

*   if $score \leq old\text{-}\alpha$ then $flag$ = UPPER-BOUND
*   else if $score \geq \beta$ then $flag$ = LOWER-BOUND
*   else /* $old\text{-}\alpha < score < \beta$ */ $flag$ = EXACT-VALUE
*   STORE($node$, $flag$, $score$, $depth$, $move$)

    return $score$
```

Figure 2.7: The alpha-beta algorithm using a transposition table.

For each position, the table stores information such as the position score, the search depth at which it was encountered, and the best move. Before searching a position, the table is consulted to see if it has been already explored. If so, the information stored may be sufficient to stop further searching at this position (cause a cutoff). If the information is insufficient to cause a cutoff, then the search window may be narrowed and the best move from a previous search can be considered first. Since the move was considered the best before, there is a good probability that it is still the best.

Figure 2.7 shows the alpha-beta algorithm for use with a transposition table. The transposition table code is marked with an asterisk (*). Details concerning the retrieval, storage, and replacement of information are omitted for clarity. Breuker has published an extensive study on transposition tables and their use [8].

Zobrist Hashing

Transposition tables are usually implemented as hash tables. A commonly used hash function in game-playing programs is the one proposed by Zobrist [64]. In Checkers there are 4 different pieces (2 kinds $\times$ 2 colours) and 32 squares. For every combination of piece and square a pseudo-random number is generated. This gives us a total of 128 pseudo-random numbers. Additionally, there is an extra pseudo-random number that indicates if it is the second player's turn to play. To compute the hash value of a position, we do an exclusive-or (XOR) operation with the pseudo-random numbers associated with that position.

This method allows us to compute the hash value of a position incrementally. For example, in Checkers, given the hash value of a position, we can obtain the hash value of one of its successors easily. For that, we only need to do an XOR operation with the pseudo-random numbers associated with the corresponding move, i.e. the square from which the piece is moving and the square to which the piece is moving, plus the squares of the captured pieces, if there are any. To undo a move we proceed similarly.

If we implement a transposition table as a hash table, we have to tackle with two types of errors [64]. A **type-1** error occurs when two different positions have the same hash value. Type-1 errors happen because the number of hash values is usually much lower than the number of positions in the game. This can introduce search errors, as the information stored in the transposition table can be used for the wrong positions. We can completely avoid this type of errors by storing the whole position in the transposition table. However, in many games this takes up too much memory, and thus is not used in practice. In practice, type-1 errors are mitigated by using a hash key to distinguish between positions with the same hash value. Because a transposition table's size is limited, we cannot store all the positions encountered during the search. A **type-2** error, or **collision**, occurs when a position is about to be stored in the transposition table and the corresponding entry is already occupied. Consequently, a choice has to be made about which position to keep in the transposition table. Such choices are based on a **replacement scheme**. Replacement schemes are discussed later in chapter 5.

Enhanced Transposition Cutoffs

Often, when the information stored in the transposition table does not cause an immediate cutoff, the exploration of the best move does. However, the subtree under the move is still searched. If there was another move whose stored information could cause an immediate cutoff, then we would save search effort, as no subtree would be explored. Thus, before searching any move at a position, we check if the information stored for any of the position's successors can cause a cutoff. If so, unnecessary search effort has been avoided. This technique is called **enhanced transposition cutoffs** (**ETC**) [36]. Due to the increased computational effort, ETC is usually not performed at all nodes, but just at the nodes that are for example more than 2 ply away from the horizon.

Figure 2.8 shows an example of the kind of cutoffs possible with ETC. Consider node c, whose information stored in the transposition table does not cause an immediate cutoff. The transposition table suggests that move m should be tried first at c. If the exploration of m causes a cutoff, the other moves will not be explored. Now consider that one of the other children of c, node f, transposes into another part of the game tree, node d, which has already been searched. If d's stored information is enough to cause a cutoff at c, node e does not need to be explored. Before searching any move at a

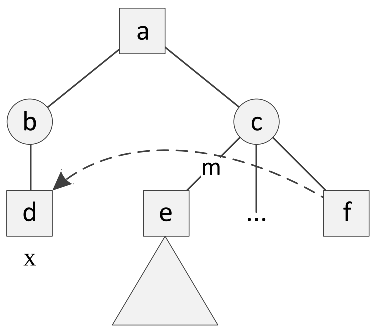


Figure 2.8: An enhanced transposition cutoff.

node, ETC accesses the information stored in the transposition table for each one of the node's children looking for transpositions.

2.3.2 Killer and History Heuristics

Often, during the search most moves tried in a position are quickly refuted, usually by the same move. The **killer heuristic** [1] involves remembering, at each search depth, the moves that are causing more cutoffs (the **killers**). When the same depth is reached again in the game tree, the killer moves are retrieved and used, if valid. By trying the killer moves before the other moves, we increase the chance of cutoffs. The move searched just before a cutoff is usually referred to as the **refutation**, because it refutes its predecessor.

The **history heuristic** [41] is a generalisation of the killer heuristic. Instead of only remembering the best killer moves at each depth, the history heuristic maintains a record for every move seen so far. Each move has an associated history score that indicates how often it has been good (caused a cutoff or had the best score). Moves are then ordered by their history score and examined by this order.

Both the killer and history heuristics are application-independent techniques to perform move ordering. Transposition tables store the exact context under which a move was considered the best. The history heuristic records which moves are most often best, but has no knowledge about the context in which they were considered the best.

2.3.3 Iterative Deepening

Iterative deepening (**ID**) [17, 55] is a technique used in many game-playing programs. The idea behind iterative deepening is that the best move for a $(d-1)$-depth search is likely to also be the best move for a d-depth search. Hence, a series of searches is performed, each one with increasing depth. At the end of each search, the moves are ordered based on their scores and this ordering is used in the next search. The rationale behind iterative deepening is that a shallow search is a good approximation of a deeper search. At each iteration, the previous iteration's solution is refined. In contrast, without iterative deepening the search can go off in the wrong direction and waste lots of effort before stumbling on the best line of play.

Move ordering can be further improved if iterative deepening is used along with transposition tables and heuristics such as the killer and history heuristics, as the previous searches fill the tables with useful information. Move ordering is more important at the root of the game tree. Considering the best moves first narrows the search window, increasing the chance of cutoffs.

Another advantage of using iterative deepening is for time control. If the search is being constrained by time, when time runs out, the best move (up to the current search depth) is known. If there is time left, a deeper search can be started.

2.3.4 Search Window Reductions

The most obvious way to improve the effectiveness of alpha-beta is to improve the order in which positions are examined. If the most relevant positions are examined first, the number of cutoffs done by alpha-beta may increase. Another way of improving alpha-beta's effectiveness is to reduce its search window. If the search window is smaller, the search will be more efficient as the likelihood of pruning parts of the game tree increases. If the search window is reduced artificially, there is a risk that alpha-beta may not be able to find the score. Hence, a re-search with the correct window may be necessary. In practice, the savings of smaller search windows outweigh the overhead of additional re-searches (see, e.g., [13, 27]).

Aspiration Search

Aspiration search (**AS**) [10, 21] uses a search window centred around an expected value, $[EV-\delta, EV+\delta]$, where EV represents the expected value (the expected result of the search), and δ a reasonable range of uncertainty, or expected error. Such windows are referred to as **aspiration windows**. An aspiration search can result in one of three cases:

1. $score \leq EV - \delta$. The result is below the aspiration window. Another search, with window $[-\infty, score]$, is required to find the actual score.
2. $score \geq EV + \delta$. The result is above the aspiration window. Another search, with window $[score, +\infty]$, is required to find the actual score.
3. $EV - \delta < score < EV + \delta$. The result is within the aspiration window and it was determined with less effort than with a search using a full window.

Aspiration search is usually combined with iterative deepening. The result of the $(d-1)$-depth search can be used as the expected value for the d-depth search. Note that δ is application dependent. Aspiration search is a gamble. If the result is within the aspiration window, then the enhancement wins, otherwise an additional search is needed. Aspiration search is usually only used at the root of the game tree.

Null-Window Search

The **null-window search** [34, 20], also known as **minimal-window search** or **zero-window search**, takes the aspiration search idea to an extreme: it uses a search window of width one[7], i.e. $\beta - \alpha = 1$,

[7]Assuming that the smallest possible difference between any two values returned by the evaluation function is 1.

```
NegaScout($node$, $\alpha$, $\beta$, $depth$)
  if $node$ is terminal or $depth == 0$ then
    return Evaluate($node$)
  $first\text{-}move$ = FirstChild($node$)
  $score = -$NegaScout($first\text{-}move$, $-\beta$, $-\alpha$, $depth - 1$)
  $\alpha$ = Max($\alpha$, $score$)
  for each node $n \in$ Children($node$) $\setminus$ $first\text{-}move$ and $score < \beta$ do
    $temp = -$NegaScout($n$, $-\alpha - 1$, $-\alpha$, $depth - 1$)
    if $temp > \alpha$ and $temp < \beta$ and $\underline{depth > 2}$ then // re-search
      $temp = -$NegaScout($n$, $-\beta$, $-temp$, $depth - 1$)
    $score$ = Max($score$, $temp$)
    $\alpha$ = Max($\alpha$, $score$)
  return $score$
```

Figure 2.9: The NegaScout algorithm.

called **null window**, **minimal window**, or **zero window**. A null-window search always returns a bound on the score. The algorithm assumes that the first move considered at each position is the best, and consequently, the remaining are inferior, until proven otherwise. The algorithm uses a full or aspiration window to score the first move, then a null window is used for the remainder of the search. As with aspiration search, a re-search may be necessary to find the actual score. In the best-case scenario, the first move considered is really the best and all the others will fail, either low or high. The biggest advantage of using a null-window search over a full-window search is the possibility of generating cutoffs at ALL nodes.

NegaScout and Principal Variation Search

The **NegaScout** (**NS**) algorithm [37, 38] is an example of a null-window search algorithm. It is an enhanced version of the fail-soft alpha-beta algorithm that uses null windows. NegaScout uses a wide search window for the first move, hoping it will lead to a PV node (it assumes that the first move considered at each node is the best), and a null window for the others, as it expects them not to be part of the principal variation (it may be useful to remember the structure of the minimal tree, figure 2.5). If one of the null-window searches fails high, then the corresponding node may become a new PV candidate and may have to be re-searched with a wider search window. The NegaScout algorithm is shown in figure 2.9. The searching of the first move has been unrolled from the loop for readability purposes. Notice that a fail-soft search of the last two levels of a game tree always returns an exact score, and thus no re-search is needed (underlined code in the figure). This is only true for fixed-depth game trees. If we ignore this extra enhancement, then **Principal Variation Search** (**PVS**) [31] and NegaScout are identical.

As with alpha-beta, there is also a version of NegaScout that is not fail-soft. The fail-soft enhancement lets the search return values that lie outside the search window. These can be used to tighten the search window for re-searches and to make the transposition table's pruning more effective. Thus, a fail-soft version is usually preferred over a non fail-soft.

Memory-Enhanced Test Drivers

The **Memory-Enhanced Test Driver Framework** (**MTD**) [36] takes the null window idea to the extreme. MTD algorithms search all positions with a null window. They are composed by two components: the **Memory-Enhanced Test** (**MT**) and the **driver**. The MT is essentially a null-window alpha-beta search that uses a transposition table. It is based on Pearl's Test procedure [35]. The driver is a function that guides the search, in this case, the MT.

MTD(f) is an MTD algorithm that was shown to perform better, on average, than NegaScout in Checkers, Chess, and Othello [36]. MTD(f) performs a series of searches until the result converges. In order to work, MTD(f) is given the expected value, as with aspiration search. For a thorough description of the MTD framework, as well as comparisons and equivalences to other algorithms refer to Plaat's work [36].

2.3.5 Selective Search

The alpha-beta algorithm and variants traverse the game tree in a depth-first manner. This means that they completely explore each branch of the tree before turning their attention to the next. This brings two problems, besides the already mentioned horizon effect. First, the algorithm is not able to follow the most promising lines of play completely. Often, it is only possible to determine the eventual outcome of a line of play past the search depth's limit. Quiescence search and **selective deepening** techniques try to counterbalance this problem. The second problem is wasted search effort. If the evaluation function is reasonably accurate and it scores a move as bad, then there is no point in continuing to search it deeper. This justifies the use of forward pruning techniques. There is another class of algorithms that traverses game trees in a best-first manner. Generally, they perform worse than alpha-beta and are not used in practice [36, 25].

The pruning method used in alpha-beta is called **backward pruning**, since the cutoffs are caused by the values returned by the algorithm. This pruning method is sound, as it does not affect the search result. **Forward pruning** techniques prune away branches if it is believed that the nodes in the branches are unlikely to affect the search result. Hence, these techniques can eliminate important moves from consideration. The rationale behind such techniques is that the time saved by pruning non-promising moves is used to search others deeper, and thus increase the overall quality of play. **Singular Extensions** [3] is an example of a selective deepening technique. This technique tries to find a single move at a position that seems to be better than the alternatives. It does such through a shallow search. If there is such a move, it is searched deeper. Another technique is the **null-move reduction** [19]. A **null move** is a move in which the side to play passes and so the opponent gets to play two times in a row. By doing a null move we can get a lower bound on a position, because all the other moves are presumably better (for some games, like Checkers, this is not true, as passing can be beneficial). If this lower bound causes a cutoff, then it was found in a cheap way, otherwise little search effort was wasted.

Some forward pruning techniques such as **ProbCut** [11], **Multi-ProbCut** [12], and **Multi-Cut**

[7, 62] use the result of a shallow search to predict if a deeper search would be relevant or not. Other techniques such as **Razoring** [6, 23] and **Futility Prunning** [41, 23] use the static evaluation (the score returned by the evaluation function) of nodes near the frontier to identify bad lines of play that should not be considered further. The **frontier** is defined as the set of all leaf nodes at a given point in the search.

2.3.6 Search Instability

Search instability is a phenomenon that appears when enhancements such as transposition tables and forward pruning techniques are used. For example, consider that the transposition table has information stored from a previous n-ply search. Now consider that you are performing a m-ply search and that $m < n$. The results from the n-ply search might cause cutoffs on your current m-ply search. If the positions that caused the cutoffs are replaced in the transposition table, the next time you encounter them you may obtain a different score. This can lead to strange behaviours like obtaining a fail low on the re-search of a fail high. In general, there is no solution to completely solve this problem.

2.4 Solving Games

Games for which the **game-theoretic value**, i.e. the outcome when all players play optimally, is known are said to be **solved**. This value indicates whether a game is won, lost, or drawn from the perspective of the player that makes the first move. For two-player zero-sum non-cooperative sequential games with perfect information, there are at least three distinct definitions for solving a game [2]:

- **Ultra-weakly solved**. The game-theoretic value of the initial position(s) has been determined.
- **Weakly solved**. The game is ultra-weakly solved and a strategy has been determined to achieve the game-theoretic value from the initial position(s). This means that if the opponent makes a mistake, then the other player does not necessarily know a strategy to achieve the game-theoretic value.
- **Strongly solved**. A strategy has been determined to achieve the game-theoretic value from any legal position.

Note that for the weak and strong definitions reasonable resources have to be assumed, otherwise it would be possible to determine the game-theoretic value of any game given a large enough amount of time and space. Usually, when trying to solve a game the interest lies mostly in determining a winning strategy and not so much on determining the game-theoretic value. Note that in this document we do not take into consideration games that can be solved and played solely by mathematics. Such games are described elsewhere (see, e.g., [4]). For a survey of solved games and methods for solving games refer to the work by van den Herik et al. [59].

2.4.1 Game Complexity

The complexity of a game is usually described using two different measures: the state-space complexity and the game-tree complexity [2]. The **state-space complexity** of a game is defined as the number of different legal game positions reachable from the initial position(s) of the game. It provides a bound on the complexity of games that can be solved through complete enumeration. The **game-tree complexity** of a game is defined as the number of leaf nodes in the solution search tree of the initial position(s) of the game. The **solution search tree** of a position is the smallest full-width game tree that is sufficient to determine the game-theoretic value of the position. The game-tree complexity is an estimate of the size of the game tree that must be examined to solve the game. It can also be seen as a good indication of a game's decision complexity, i.e. the difficulty of making the correct decisions. Both state-space and game-tree complexities are important factors in determining the solvability of a game [2, 59].

Often, it is difficult to calculate the game-tree complexity of a game. However, an approximation can be calculated by counting the number of leaf nodes b^p of the game tree with as depth the average game length p in ply, and as branching factor the average branching factor b of the game.

3 Case Study: Chinook

The **Chinook**[1] project was created in 1989, at the University of Alberta, Canada. Its goals were, first, to build a world-championship-calibre English Checkers computer program, and second, one day try to solve the game. The program entered competitive play in 1989, in both human tournaments and computer olympiads. In 1994, Chinook was declared the World Man-Machine Champion in English Checkers, and in 1997 retired from competitive play. Despite some work had been done throughout these years, only in 2001 major efforts were put into solving the game. In 2007, English Checkers was solved by the Chinook team. The game-theoretic value is draw.

In this chapter we will study Chinook and the proof procedure used to weakly solve the game of English Checkers. This chapter is based in literature about Chinook [50, 53, 43, 44, 47, 49, 45]. Although there is a significant amount of literature about the program, some of its technicalities have never been published [46].

3.1 English Checkers

English Checkers, also known as British or American Checkers, or simply Checkers, is probably the most widely known Checkers variant in the world. This variant is also the most played in the British Commonwealth and in the United States of America. The first computer Checkers programs were developed for this variant. The first program was written by Christopher Strachey in 1952 [56], but it was not until Samuel developed his program [39, 40], in the 1950s and 1960s, that the game received special attention, as it was incorrectly reported by the media that Samuel's program had beaten a Checkers master and solved the game, in 1962. As a result of this misunderstanding, efforts on developing world-class Checkers programs stopped for some time. Efforts to develop stronger programs were retaken some years later, most notably in the late 1970s by a team from Duke University with the PAASLOW program [58]. In 2007 the chapter was somehow closed with the Chinook team solving the game of English Checkers.

We can get a loose upper bound on the state-space complexity of English Checkers by counting the number of possible combinations of men and kings on the board. This gives us about 5×10^{20} different positions [52] (consult section 4.1 for details). A tighter bound on the state-space complexity of the game is 10^{18} [52]. Having an average branching factor of about 2.8 and an estimated average game length of 70 ply, the game-tree complexity of English Checkers can be approximated by $2.8^{70} = 2 \times 10^{31}$ [2].

[1]`http://webdocs.cs.ualberta.ca/~chinook/`

3.2 The World Man-Machine Checkers Champion

Game-playing programs are usually composed by two main components: search and knowledge. Chinook's search component consists of a parallel alpha-beta algorithm with some enhancements. The knowledge component is composed by an evaluation function, an opening book, and endgame databases.

3.2.1 Search Algorithm

Chinook uses a parallel version of NegaScout [30] with iterative deepening (two ply at a time) and aspiration search (with a base δ of 35) at the root. When a fail low happens at the root, instead of re-searching with the window $[-\infty, bound]$, the iterative-deepening search is restarted. Furthermore, it uses a two-level transposition table. In the first entry the position with the greatest search depth is stored, in the second entry the most recent position is stored. Chinook also uses enhanced transposition cutoffs. Moves are ordered using only the transposition table and the history heuristic, no application-depend knowledge is used. The program does not have a quiescence search, the search continues until a stable position is reached. This is accomplished by using search extensions.

3.2.2 Search Extensions

Chinook uses a myriad of application-dependent search extensions and reductions. In lines of play where one player has less pieces than the other and has no positional compensation, the search depth may be reduced by one third or one half. Positions with pending captures are always extended until a position without captures is reached. In positions in which the opponent is threatening to capture pieces the search may be extended, depending on how bad is the threat. These are some of the search extensions and reductions used in the program. A more complete description is available in the literature about Chinook.

To reduce the error of misevaluating a position due to hidden tactics, Chinook was extended with **tactical tables**, which store tactical patterns. When a position is evaluated, the tactical tables are consulted. If there is a match (i.e. the position contains a pattern), the program extends the search to resolve the tactics.

3.2.3 Evaluation Function

Chinook uses **lazy evaluation**. When evaluating a position, only part of the evaluation function is executed at first. If the intermediate score falls outside the search window by a certain margin, then the rest of the evaluation function does not need to be executed, as the eventual score would not affect the search result; otherwise the evaluation function is executed completely. This allows us to save computational effort in positions that are not relevant.

Chinook's evaluation function has more than 20 heuristics. The knowledge in the evaluation function was devised from several sources: experience from building other game-playing programs, input from a

Checkers expert, experience playing with the program, and post-game analysis of Chinook's matches. The evaluation function was manually tuned along a period of several years.

For a position p, the score returned by the evaluation function equals the sum of applying all the heuristic functions to p:

$$evaluate(p) = \left[\sum_{i=1}^{n} heuristic_i(p) \times weight_i\right] \times fudge$$

where n represents the number of heuristic functions, $heuristic_i(p)$ the result of applying the heuristic function i to position p, $weight_i$ the weight of heuristic function i, and *fudge* the fudge heuristic. The weight of each heuristic may differ for each phase of the game. Chinook divides the game into four phases (opening, early middle game, late middle game, and endgame), based solely on the number of pieces on the board. Depending on the game phase, the sum of the heuristics is multiplied by a constant whose value is shown in table 3.1. This was named the fudge heuristic, and its goal is to make the program favour positions with more pieces on the board over positions with fewer.

Game Phase	Number of Pieces	Constant Value
1	20–24	1.34
2	16–19	1.17
3	12–15	1.00
4	0–11	0.83

Table 3.1: The fudge heuristic.

Each heuristic deals with a particular piece of Checkers knowledge. A heuristic may recognize a pattern in the dispersion of the pieces that may indicate trouble, recognize a specific situation that is known to be good, etc. For example, the mobility heuristic analyses the degree to which the pieces are free or restricted in their movements. Based on that, the heuristic estimates whether the position is better for one of the players or if it is equal for both, and then returns a numerical value that reflects that. Information about some of Chinook's heuristics is available in the literature about the program (for weights see [51]).

3.2.4 Book Knowledge

During the opening phase of the game, the search plus evaluation function may not be sufficient to find the best moves. Chinook's authors wanted the program to be innovative, deviating from standard opening moves, and at the same time to play well. Although in the beginning its opening book was small (contained a little over 6 000 positions), later on it was merged with the opening book of another program. This resulted in an opening book with more than 60 000 positions, each of which was verified by Chinook. The program was instructed to search for new moves in the openings (cooks) while verifying the book. In its opening book there is also an **anti-book**, a library of moves that should not be made, as they lead to positions that are known to be bad or lost. Chinook follows its opening book blindly for the first moves of the game. To still permit Chinook to play innovatively, the program has an option to use mostly the anti-book, instead of the entire book.

In addition to the opening book just described, the program has also another book [42]. This second book consists of pre-processed grandmaster games (of Marion Tinsley) and is only used to bias the search when the program is out of the book. A bonus is added to a move in the search if it is in this book. For moves leading to winning lines of play, the bias is stronger than for those leading to draws. A slight negative bias is used to lead the search away from moves that lead to losses.

3.2.5 Endgame Databases

The endgame databases contain all positions with 8 or fewer pieces. For each position the game-theoretic value (win, loss, or draw) is stored. The best move for each position is not stored, as it would lead to a significant increase of the database size. Once a lost position is reached the program may not offer the maximum resistance. This is due to not storing the best move. The endgame databases were constructed using **retrograde analysis** [57].

To compute the n-piece database we need the $1, 2, \ldots, (n-1)$-piece databases to be completed. There are also some dependencies between positions in the same database. For example, to compute the 4-piece database where all pieces are men, we need to have already computed all the other 4-piece positions, as in some positions it may be possible to promote some of the men. More information about these dependencies can be found elsewhere [29].

The retrograde analysis algorithm works as follow. Initially, the game-theoretic value for all the n-piece positions is set to *unknown*. Then, for each position consider that Black is to play (n.b. Black is Max in English Checkers) and apply the following rules (in order of precedence):

1. Set the position's value to *win* if there is a move that leads to a White loss.
2. The value of the position remains *unknown* if there is a move to a White position whose value is also *unknown*.
3. Set the position's value to *draw* if there is a move that leads to a White draw and Black is unable to win.
4. Set the value of the position to *loss* if all the moves lead to White winning.

The algorithm iterates until no more n-piece positions can be solved. At that point, all the remaining *unknown* positions must be draws. Starting at the end of the game, retrograde analysis tries to solve positions closer to the beginning of the game. In principle, this technique could be used to solve the entire game.

3.2.6 Draw Differentiation

Not all draws are equal. In some draws there is a chance of inducing the opponent into an error, as the correct line of play is difficult to identify. Such draws are referred to as **strong draws**. Draws where the correct moves can be trivially found by the opponent are referred to as **weak draws**. Chinook can differentiate between these two types of draws. When the program finds a drawing position in the endgame databases, it performs a search with the databases disabled. By turning off the databases, the program can use the result of the search to score the position. The resulting score is then scaled properly so that errors do not affect the choice, e.g. prevent moves with high drawing scores to be preferred over moves with serious winning chances. This means that drawing positions can now have a score different from zero. If Chinook has a choice between playing a strong drawing move or a non-drawing move that yields a small advantage, it prefers the strong draw.

3.3 Solving Checkers

The Chinook team created a proof procedure to solve the game of English Checkers. The goal of this procedure was to determine the game-theoretic value of the game, i.e. whether the game is won, lost, or drawn from the perspective of the player that makes the first move. The proof procedure had the following components:

- Endgame databases. Chinook's endgame databases were extended to included all positions with 10 or fewer pieces [48].
- Proof tree manager. This component manages the search. It maintains the tree of the proof in progress and decides which positions need to be considered next to advance the proof.
- Proof solvers. Given a position by the proof tree manager, a proof solver uses Chinook and a variant of Proof-Number Search to determine the game-theoretic value of the position.

Figure 3.1 illustrates the proof procedure. The number of pieces on the board are plotted (vertically) versus the logarithm of the number of positions (the widest point is with 23 pieces, with a little over 10^{20} positions). Only the portion of the state-space that is inside the inner oval is relevant to the proof. A position may be irrelevant because it is either unreachable or not required for the proof. The small open circles indicate positions for which the game-theoretic value has been determined. The dotted line shows the boundary between the top of the proof tree, as seen by the proof tree manager, and the parts that are computed by the proof solvers. This image was taken from [49].

Several openings had to be considered to weakly solve English Checkers. Obviously, not all possible openings were considered. We will not go into the details on how the minimal number of openings to obtain the game-theoretic value was discovered. Next, we will explain how the proof procedure works for one opening. Before proceeding, the reading of appendix B, which presents Proof-Number Search, is recommended. For further details about the entire proof procedure consult the relevant literature [47, 49, 45].

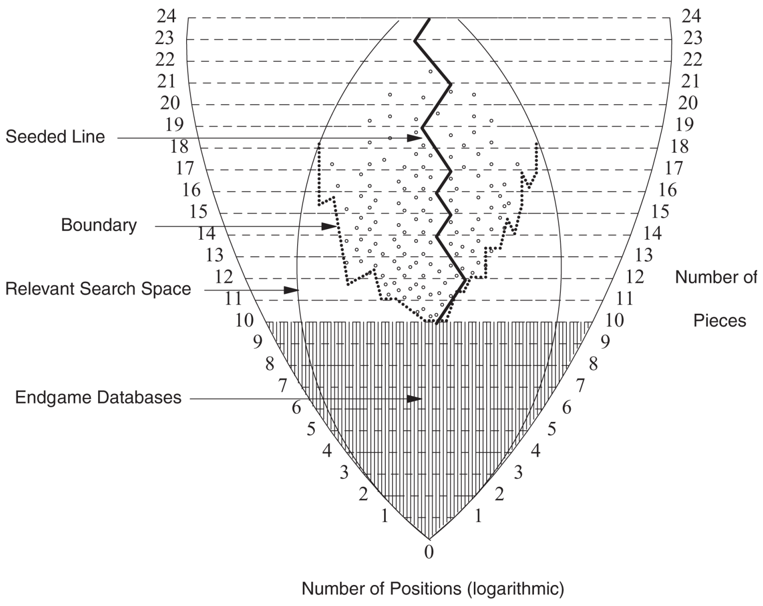


Figure 3.1: The proof procedure.

3.3.1 Proof Tree Manager

The proof tree manager starts off with an empty proof tree. Then, it is given an initial line of play. The proof manager starts by solving the last position of the given line of play, then it backs up one ply and solves the new tree. This procedure is repeated until the initial position is reached and the game-theoretic value is determined. Although not necessary, seeding the proof procedure with an initial line of play is done to increase its performance.

The proof is done iteratively. At the beginning of each iteration a heuristic threshold is set. For unsolved positions a heuristic value is computed. If the value is outside the current threshold, then the position is not considered in the current iteration. Only the positions with values within the threshold are explored. Positions that have been solved need no further work, as their game-theoretic value is already known. The exploration of positions with heuristic values outside the threshold is postponed until they are shown to be necessary for the proof. This ensures that only the possibly relevant positions will be considered. Once the proof tree is complete for a given threshold, the threshold is increased and a new iteration begins. The proof tree manager uses the heuristic values to choose which positions to consider next.

In addition to the usual values given to a position (win, loss, draw, and unknown) two new values are added: likely win and likely loss. If a position's heuristic value exceeds the threshold, the position is considered a likely win. If the heuristic value falls below the negative threshold, the position is considered a likely loss. The iterative process continues until the threshold exceeds the largest possible heuristic value. At this point all the likely wins and likely losses have been solved, as well as the proof tree.

3.3.2 Proof Solver

The proof solver uses Chinook and PNS. Chinook is used to determine the heuristic value of a position. If the heuristic value of a position is within the current threshold, the proof solver attempts to formally solve the position with PNS. If Chinook indicates that the position is a possible win, the proof solver first tries to prove or disprove the position as a loss. This is done because answering the opposite question is usually easier. Given that the expected game-theoretic value of the whole proof was a draw (which in fact is), the disproof of a win or of a loss may be sufficient for the proof tree manager. If this question is answered within the maximum time allocated, another search is started to answer the original question. Solving a position implies finding a path to the endgame databases or to an already solved position.

4 Complexity of Classical Checkers

In this chapter we will estimate and analyse the complexity of Classical Checkers. This should give us some insight into the difficulty of the game, comparatively to others. The analysis is divided into two parts: state-space complexity and game-tree complexity.

4.1 State-Space Complexity

Similarly to what has been done for other games, in particular English Checkers, we can get a loose upper bound on the state-space complexity of Classical Checkers by counting the number of possible combinations of men and kings on the board [52]. The number of positions having n or fewer pieces on the board can be calculated with:

$$\sum_{w=0}^{\textsc{Min}(n,12)} \sum_{b=0}^{\textsc{Min}(n-w,12)} \sum_{W=0}^{\textsc{Min}(n-b,12)-w} \sum_{B=0}^{\textsc{Min}(n-w-W,12)-b} \sum_{f=0}^{\textsc{Min}(w,4)} \mathrm{NP}(w,b,W,B,f) - 1,$$

$$\mathrm{NP}(w,b,W,B,f) = \binom{4}{f}\binom{24}{w-f}\binom{28-(w-f)}{b}\binom{32-w-b}{W}\binom{32-w-b-W}{B}$$

where n is the number of pieces, w the number of white men, b the number of black men, W the number of white kings, B the number of black kings, and f the number of white men on the first row. Note that a man can only occupy 28 of the total 32 squares (if it is on the kings row it has been promoted to king), and that each side can have at maximum 12 pieces on the board. The subtraction of the result by 1 handles the case of zero pieces on the board. The number of positions for $n = 1$ to 24 is shown in table 4.1. The total number of positions is approximately equal to 5×10^{20}.

The calculation above is misleading, as it considers positions that cannot occur in a game. Although there are many legal positions with 24 pieces on the board, only a small fraction can be reached in a game. For example, given a reasonable playing strategy, no position with 24 pieces in which some are kings can be reached. As in general there are many unreachable positions within the set of legal positions, some authors have imposed constraints on the positions considered in order to obtain tighter bounds on the state-space complexities of the games considered.

We can obtain a tighter bound on the state-space complexity of Classical Checkers by taking the following assumptions into account. First, we should only consider positions with less than 9 kings on the board, as positions with more than 8 kings are very unlikely to occur naturally in a game. Second, positions in which a side has a big advantage over the other (more than 4 pieces) should be discarded,

# Pieces	# Positions
1	120
2	6 972
3	261 224
4	7 092 774
5	148 688 232
6	2 503 611 964
7	34 779 531 480
8	406 309 208 481
9	4 048 627 642 976
10	34 778 882 769 216
11	259 669 578 902 016
12	1 695 618 078 654 976
13	9 726 900 031 328 256
14	49 134 911 067 979 776
15	218 511 510 918 189 056
16	852 888 183 557 922 816
17	2 905 162 728 973 680 640
18	8 568 043 414 939 516 928
19	21 661 954 506 100 113 408
20	46 352 957 062 510 379 008
21	82 459 728 874 435 248 128
22	118 435 747 136 817 856 512
23	129 406 908 049 181 900 800
24	90 072 726 844 888 186 880
Total	500 995 484 682 338 672 639

Table 4.1: The state-space complexity of Classical Checkers.

as they will almost surely lead to a win for the strong side. Satisfying these two constraints, the new estimation for the state-space complexity of Classical Checkers is 4×10^{19}, which is a significant reduction from the previous estimation, 5×10^{20}. Note, however, that this estimation is generous, as it still includes some unlikely positions.

4.2 Game-Tree Complexity

To calculate the game-tree complexity of Classical Checkers we have to first estimate its average game length and average branching factor. To do this, we collected a vast amount of official and non-official games between Checkers experts (3331 games in total). The games were provided by a Checkers expert [18].

Several estimations were done. The first estimations were obtained from the games that finished with n or fewer pieces on the board. We only selected games that finished with n or fewer pieces because many games were not completely annotated, and often the players would agree on terminating the game in early stages, which could lead to bad estimations. For the second estimations our Classical Checkers computer program, Turska, would play a pre-defined amount of moves past the annotated games end. The results are shown in table 4.2. The average game length and average branching factor are shown for the two estimation methods and the two values of n considered. The number of games considered for each

Turska	n	# Games	Length	Branching Factor
No	6	995	56.45	6.24
	8	2057	53.40	6.24
Yes	6	2882	66.63	6.58
	8	3230	66.13	6.63

Table 4.2: The average game length and average branching factor estimations for Classical Checkers.

calculation is also shown. We can see that the average branching factor is relatively stable, around 6.4, whilst the average game length is not. Conferencing with a Checkers expert we concluded that the best estimate for the average game length would be of about 60 ply [18], which lies within our estimations. Having an average branching factor of about 6.4 and an estimated average game length of 60 ply, the game-tree complexity of Classical Checkers can be approximated by $6.4^{60} = 2 \times 10^{48}$.

The average branching factor was calculated by summing the branching factor for every position considered and then dividing the result by the total game length. Another method considered was to calculate the average branching factor for each game individually, and then do the average between them all. This last method resulted in higher branching factors.

4.3 Comparison with Other Games

In table 4.3 we compare Classical Checkers with other games. As expected, Classical Checkers is harder than English Checkers: the game-tree complexity of Classical Checkers is higher than the one of English Checkers. Although the average game length is smaller, the branching factor is higher. The branching factor of both games is very similar at the beginning of the game, but as soon as there are kings the branching factor of Classical Checkers increases significantly compared to the one of English Checkers, due to having flying kings. It is also due to the flying kings that the average game length is shorter. We feel that the state-space complexities of both games should be more similar. The small, but still significant, difference between the games state-space complexities is mainly due to the assumptions taken when calculating the respective complexities. For a description on how the state-space complexity of English Checkers was calculated refer to the work by Schaeffer et al. [52]. The high similarities between English Checkers and Classical Checkers may be an indicator that the latter could also be solved with current technology.

			Complexity	
Game	Length	Branching Factor	State-Space	Game-Tree
English Checkers [30, 2, 52]	70	2.8	10^{18}	10^{31}
Classical Checkers	**60**	**6.4**	$\mathbf{10^{19}}$	$\mathbf{10^{48}}$
International Checkers [2]	90	4	10^{30}	10^{54}
Othello [2]	58	10	10^{28}	10^{58}
Chess [2, 15]	80	35	10^{46}	10^{123}
Amazons (10×10) [24]	84	374, 299	10^{40}	10^{212}
Go (19×19) [2]	150	250	10^{172}	10^{359}

Table 4.3: The complexity of Classical Checkers compared to that of other games.

Looking at table 4.3 we note that the average branching factor of Classical Checkers is higher than the one of International Checkers. Given that the rules of both games are very similar, and that the board of the International variant is bigger (10×10 instead of 8×8), we believe that perhaps the existing estimation for the average branching factor of International Checkers is too conservative.

For a comparison between more games, not including Classical Checkers, refer to the work by van den Herik et al. [59].

5 Search in Turska

During the development of our Classical Checkers computer program, Turska, several search algorithms and techniques were implemented and experimented with. The program was developed iteratively and incrementally, starting from the most basic techniques. In this chapter we will analyse each one incrementally. We will also analyse several replacement schemes for transposition tables, and how storing additional information in the transposition table affects the search. Although several search algorithms and techniques were tested, the evaluation function and quiescence search remained the same throughout the experiments. The discussion of these two is deferred to chapter 6.

Similarly to other works in the area, each experiment was repeated several times, each time with a different base search depth. This way we can get an idea on how the performance gain of each enhancement grows. It also helps identifying the usefulness of the enhancements, e.g. if an enhancement is only useful from 30-ply searches onwards, then it is not useful in practice, as the normal search depth is much lower.

The main comparison measure that is used is the **total node count** (**TNC**), which is the number of nodes visited for all positions in the test set. The performance gain is always shown as a percentage. All other measures are shown in thousands, unless stated otherwise. In the experiments, we do not show the execution time because it is directly correlated with the total node count in our program. In any case, the execution time is not a very reliable metric because it depends on external factors, such as the machine clock rate and cache.

5.1 The Test Set

Our test set consists of 140 positions. There are common and difficult positions, positions from tournament games, and problem positions, some of which are unlikely to occur naturally in a game. The test set tries not to be biased towards specific game features.

The test set is divided into three subsets, each one related to a specific game phase. The opening test set contains 20 positions, the middle game test set 80, and the endgame test set 40. This way we can better analyse how each enhancement affects our program, as some enhancements have been identified to be more useful in certain phases of the game.

Another way to conduct the experiments would be to make Turska play against itself a number of times. If we had used this approach, the program would have to have a random component so it would choose a random move in positions with more than one best move; otherwise all the games would be the same. The random component would bring a problem as the positions that would be tested could be very

		α-β	α-β + TT				
Game Phase	Depth	TNC	TNC	# Hits	# Cutoffs	# Best	Gain
Opening	8	616	352	9	8	0.5	42.7
	12	25 147	8 003	269	242	14	68.1
	14	157 099	36 760	1 412	1 265	79	76.6
	18	5 473 529	777 894	32 629	29 751	1 624	85.7
Middle Game	8	2 729	1 794	63	57	3	34.2
	12	210 132	59 416	3 751	3 278	295	71.7
	14	2 233 382	312 856	25 426	21 654	2 425	85.9
	18	387 726 342	8 727 082	857 444	736 417	79 511	97.7
Endgame	8	7 165	2 068	226	201	16	71.1
	12	1 075 723	55 182	11 008	9 305	1 216	94.8
	14	14 970 164	215 582	49 479	40 995	6 091	98.5
	18	3 799 543 707	3 462 567	786 873	663 526	88 901	99.9

Table 5.1: Alpha-beta vs alpha-beta using a transposition table.

different for each experiment, and thus it would be easier to have discrepancies in the results. Breuker discusses several test methods and sets, for Chess [8].

5.2 Transposition Tables

In this experiment, a plain version of alpha-beta was compared with a version of alpha-beta using a transposition table. We used a four-level transposition table, like some of the strongest computer Chess programs of today do, e.g. Stockfish[1]. Our transposition table has 2^{19} buckets, but given that each bucket has four entries, it can store information for a maximum of 2^{21} (2 million) positions. According to some studies this number represents a good trade-off between memory and search [36, 8]. Further increasing the transposition table's size would account for minor benefits.

The transposition table used is what is called a traditional transposition table. For each position in the table the following information is stored (this is what is stored in each entry):

- *key*: used to handle hash collisions, i.e. to distinguish amongst different positions with the same hash value (that go into the same bucket).
- *flag*: indicates whether *score* is an exact value, a lower bound, or an upper bound.
- *depth*: equals the depth of the subtree rooted at the position.
- *score*: the score of the position obtained from the search. It may be used to narrow the search window.
- *move*: the move that was considered the best (caused a cutoff or had the best score). If no immediate cutoff can be obtained from the transposition table, this move is the one that is considered first.

When a collision occurs, i.e. a position is about to be stored in the transposition table and the corresponding bucket is full, the entry with the lowest *depth* in the bucket is replaced. If the position

[1]`http://www.stockfishchess.com/`

		α-β	NS		
Game Phase	Depth	TNC	TNC	# Re-searches	Gain
Opening	8	352	336	1	4.6
	12	8 003	6 871	5	14.1
	14	36 760	29 153	10	20.6
	18	777 894	496 996	50	36.1
Middle Game	8	1 794	1 605	5	10.5
	12	59 416	43 167	30	27.3
	14	312 856	202 173	66	35.3
	18	8 727 082	4 184 390	309	52.0
Endgame	8	2 068	1 769	2	14.4
	12	55 182	37 225	10	32.5
	14	215 582	141 484	19	34.3
	18	3 462 567	1 840 040	57	46.8

Table 5.2: Alpha-beta vs NegaScout.

is already stored in the transposition table, the information from the previous search is always replaced. Since the information is being replaced, it means that it was not very useful as it did not cause a cutoff. Thus, there is no point in not replacing it, as the new information might be more useful the next time. This will be the same for the other replacement schemes we will experiment later.

The results of the experiment are shown in table 5.1. The total node count is shown for both algorithms. For the enhancement is also shown the number of hits – how many times there was information stored in the transposition table for the position about to be searched; the number of cutoffs – how often the information caused a cutoff; and the number of times the move stored continued to be the best. The performance gain is shown in percentage. The other measures are shown in thousands.

As we can see from table 5.1, the use of transpositions tables is essential in any game-playing program. The total number of nodes visited reduced drastically, from a maximum of billions to thousands of millions. As expected, the gain is superior in the endgame because there are more transpositions. The enhancement is more noticeable with deeper searches, as the number of transpositions increases. We can observe that the information stored in the transposition table was very effective. Even when the information did not cause a cutoff, the move stored was often the best.

5.3 NegaScout

In this experiment, alpha-beta was compared with NegaScout. Both algorithms used a transposition table, like the one described in the previous experiment. Table 5.2 shows the results obtained. Although the gains are significant, they are more noticeable with deeper searches, as the parts of the game tree that are pruned are expectedly bigger. The use of null-windows allows the search to prune away subtrees that are inferior without knowing their exact values.

NegaScout performs well in Classical Checkers because at each position usually only 2 or 3 of the possible moves are relevant. NegaScout assumes that the first move considered at each position is the best. If these moves are searched first, the chance of cutoffs increases. Hence, NegaScout works better if

		NS	NS + ID		
Game Phase	Depth	TNC	TNC	Last Iteration	Gain
Opening	8	336	116	73.7	65.3
	12	6 871	2 164	72.7	68.5
	14	29 153	6 246	65.3	78.5
	18	496 996	70 015	69.8	85.9
Middle Game	8	1 605	666	74.1	58.4
	12	43 167	10 045	72.6	76.7
	14	202 173	37 707	73.3	81.3
	18	4 184 390	364 807	66.9	91.2
Endgame	8	1 769	957	81.2	45.9
	12	37 225	13 288	70.3	64.3
	14	141 484	40 551	67.2	71.3
	18	1 840 040	222 387	58.1	87.9

Table 5.3: NegaScout vs NegaScout with iterative deepening.

combined with iterative deepening and transposition tables.

5.4 Iterative Deepening

In this experiment, two versions of NegaScout, as described in the previous experiment, were compared. One of the versions used iterative deepening (two ply at a time, starting with 4 ply) and move sorting at the root. The results of the experiment are shown in table 5.3. Besides the total node count, for the enhancement is also shown the percentage of nodes visited in the last iteration (relative to the total number of nodes visited).

Move ordering is more important at the root of the game tree. If not combined with transpositions tables, iterative deepening is only useful to order moves at the root of the tree. When used along with a transposition table, it also fills the table with useful information for the next iteration. As good move ordering is performed, less positions are stored in the table and consequently the chance that the most relevant positions for the next iteration are still in the table is higher. Iterative deepening and transposition tables were the enhancements that accounted for more savings.

Although the number of nodes of the search tree is expected to grow at each iteration, table 5.3 shows that with increasing depth the number of nodes visited in the last iteration decreased. This means that the best move was probably found in the previous iterations, and that on the last iteration the search was well directed.

5.5 Aspiration Search

In this experiment, three versions of NegaScout were compared. All versions used iterative deepening and transposition tables. Furthermore, two of the versions used aspiration search at the root. The search window was centred around the score obtained from the previous iteration with a δ of 100 in one version, and a δ of 50 in the other. Table 5.4 shows the results obtained. For the versions using aspiration search

		NS	NS + AS 100			NS + AS 50		
Game Phase	Depth	TNC	TNC	# Fails	Gain	TNC	# Fails	Gain
Opening	8	116	158	5	−35.9	160	10	−37.3
	12	2 164	3 251	7	−50.2	3 231	16	−49.3
	14	6 246	10 691	8	−71.1	10 648	18	−70.4
	18	70 015	109 838	9	−56.8	109 213	25	−55.9
Middle Game	8	666	827	27	−24.0	829	51	−24.4
	12	10 045	11 941	48	−18.8	11 863	93	−18.0
	14	37 707	39 953	60	−5.9	39 099	115	−3.6
	18	364 807	390 780	68	−7.1	396 742	140	−8.7
Endgame	8	957	1 028	16	−7.4	1018	28	−6.4
	12	13 288	13 562	31	−2.0	13 564	49	−2.0
	14	40 551	38 064	39	6.1	38 323	60	5.4
	18	222 387	191 663	52	13.8	206 176	77	7.2

Table 5.4: NegaScout vs NegaScout with aspiration search.

the number of fails (low plus high) is shown (in actual units, not in thousands).

Some studies have shown that the use of aspiration search can lead to search reductions of about 20% (see, e.g., [21]). However, in our experiments aspiration search turned out to be a bad bet, as in general it led to more nodes being searched, due to lots of re-searches at the root (although it is not shown, most were fail highs). This means that the previous iteration's solution was refuted by the current iteration, and thus the aspiration window was incorrect, as a re-search was necessary to find the correct value. This may indicate that Turska's evaluation function needs more work.

5.6 Enhanced Transposition Cutoffs

In this experiment, two versions of NegaScout were compared. Both versions used iterative deepening and transposition tables. One of the versions also used enhanced transposition cutoffs. Due to the increased computational effort, the technique is only performed at nodes that are more than 2 ply away from the horizon. The results of the experiment are shown in table 5.5. As with transpositions tables, the enhancement is more noticeable in the endgame.

		NS	NS + ETC	
Game Phase	Depth	TNC	TNC	Gain
Opening	8	116	111	4.0
	12	2 164	1 967	9.0
	14	6 246	5 571	10.8
	18	70 015	58 990	15.7
Middle Game	8	666	634	4.7
	12	10 045	8 766	12.7
	14	37 707	31 401	16.7
	18	364 807	282 006	22.6
Endgame	8	957	829	13.3
	12	13 288	9 704	26.9
	14	40 551	29 100	28.2
	18	222 387	126 544	43.0

Table 5.5: NegaScout vs NegaScout with enhanced transposition cutoffs.

5.7 Replacement Schemes for Transposition Tables

In the previous experiments, whenever there was a collision in the transposition table, the entry with the lowest *depth* in the bucket was replaced. Breuker analysed several replacement schemes for transposition tables [8]. Here, we repeat some of his experiments, but adapting the replacement schemes for our four-level transposition table. We also experimented a new replacement scheme, *Number of Hits*. These are the replacement schemes that were experimented:

- *Deep.* This was the scheme used up until now. When there is a collision, the entry with the shallowest subtree is replaced. The rationale behind this scheme is that a subtree that is searched to a greater depth usually has more nodes than a subtree that is searched to a smaller depth. Thus, more search effort was spent when searching the larger subtree. If the larger subtree is eventually retrieved from the transposition table, it saves more work, as the search space is further reduced.
- *Big All* and *Big 1.* The depth of the search tree may be a bad indicator of the amount of search effort spent and that may be potentially saved, due to pruning. Thus, it may be preferable to store the positions with the biggest subtrees (the ones with more nodes) instead of the ones with the deepest subtrees. Consequently, more memory is needed to store the same number of positions in the transposition table, as the number of nodes of each subtree must be stored in each entry. When there is a collision, the entry that is replaced is the one with the smallest subtree, excluding the one with the deepest subtree. Big All counts a position from the transposition table as n nodes, where n is the number of nodes in the subtree of that position, whereas Big 1 counts the position as a single node.
- *Number of Hits.* In this scheme the depth or size of the subtrees is ignored, what matters is the number of times a position was encountered. Thus, when there is a collision, the entry that is replaced is the one that was used less times, excluding the one with the deepest subtree.

Preliminary results showed that using NegaScout with iterative deepening and a transposition table would lead to insignificant differences in the total node count. For this reason, the tests were run using only NegaScout with a transposition table, so that the differences would be more noticeable. Table 5.6 shows the results obtained. The gains shown are relative to the *Deep* scheme.

Our results indicate that there is hardly any difference between the *Big All* and *Big 1* schemes. Furthermore, the *Number of Hits* scheme revealed to be a bad bet, even though we were expecting it to account for minor improvements in the endgame, due to the increased number of transpositions. This is because the scheme does not take into account the amount of work done to investigate a position. For deeper searches, the *Big* schemes are slightly better than the *Deep* scheme.

The results show that given a transposition table with a reasonable size and a bucket system, the replacement scheme hardly affects the search results, thus the simplest replacement scheme, *Deep*, is preferred. Our results appear to be in conformity with the ones obtained by Breuker [8].

		Deep	Big All		Big 1		Number of Hits	
Game Phase	Depth	TNC	TNC	Gain	TNC	Gain	TNC	Gain
Opening	8	336	336	0.00	336	0.00	336	0.00
	12	6 871	6 871	0.00	6 871	0.00	6 871	0.00
	14	29 153	29 138	0.04	29 138	0.04	29 156	−0.01
	18	496 996	492 047	0.99	492 047	0.99	506 445	−1.90
Middle Game	8	1 605	1 605	0.00	1 605	0.00	1 605	0.00
	12	43 167	43 167	0.00	43 167	0.00	43 167	0.00
	14	202 173	202 760	−0.29	202 760	−0.29	202 296	−0.06
	18	4 184 390	4 084 870	2.37	4 084 870	2.37	4 329 131	−3.45
Endgame	8	1 769	1 769	0.00	1 769	0.00	1 769	0.00
	12	37 225	37 211	0.03	37 211	0.03	37 214	0.03
	14	141 484	153 746	−8.66	153 746	−8.66	145 194	−2.62
	18	1 840 040	1 735 133	5.70	1 735 133	5.70	2 044 726	−11.12

Table 5.6: Comparison of several replacement schemes for transposition tables.

5.8 Information in Transposition Tables

In this experiment, we investigated how storing additional information in the transposition table affects the search. We tried some of the suggestions mentioned by Breuker [8]:

- *PV.* We add a flag to each entry that indicates whether the node has ever been PV or expected-PV. Such nodes are never removed from the transposition table, independently of the replacement scheme in use.
- *Two Bounds.* Instead of storing only the score of the position and whether it is an exact value, a lower bound, or an upper bound, we store the two bounds, α and β, with separate *depth*s for each.

As in the previous experiment, the tests were also run using NegaScout with a transposition table. The results of the experiment are shown in table 5.7. Here, the number of cutoffs and of re-searches is shown in actual units, and not in thousands as before.

The information about whether a node is PV or not does not seem to be important, as the fluctuation of gain is almost negligible. The results of the *Two Bounds* transposition table version experiment were different from what we were expecting. We were expecting that the number of cutoffs would increase along with the reduction of the total node count. Instead, what happened was that for shallower searches the number of cutoffs decreased, as well as the total node count. For deeper searches, the number of cutoffs increased significantly, but so did the total node count. This was due to re-searches. For deeper searches, the number of re-searches increased much more than for shallower searches. The *Two Bounds* version appears to perform especially worse in the endgame, where normally enhancements that take advantage of transpositions perform better. As more information is stored in the transposition table, the probability that deeper subtrees are transposed to shallower searches is higher. This led to more cutoffs and re-searches in general.

Game Phase	Depth	NS TNC	NS # Cutoffs	NS # Re-searches	PV TNC	PV Gain
Opening	8	336	10 568	1 231	336	0.00
	12	6 871	246 028	5 520	6 871	0.00
	14	29 153	1 133 928	10 290	29 153	0.00
	18	496 996	20 896 101	50 097	497 006	0.00
Middle Game	8	1 605	71 728	5 532	1 605	0.00
	12	43 167	2 811 969	30 219	43 167	0.00
	14	202 173	15 893 660	66 257	202 172	0.00
	18	4 184 390	403 192 819	309 308	4 161 820	0.54
Endgame	8	1 769	207 607	2 768	1 769	0.00
	12	37 225	6 747 845	10 123	37 225	0.00
	14	141 484	29 865 764	19 136	141 114	0.26
	18	1 840 040	402 949 891	57 908	1 917 693	−4.22

Game Phase	Depth	Two Bounds TNC	Two Bounds # Cutoffs	Two Bounds # Re-searches	Two Bounds Gain
Opening	8	334	10 436	1 232	0.58
	12	6 844	244 147	5 524	0.39
	14	29 103	1 134 306	10 320	0.16
	18	547 252	25 109 333	51 074	−10.11
Middle Game	8	1 599	71 490	5 519	0.38
	12	42 957	2 796 029	30 011	0.48
	14	202 713	16 074 865	66 517	−0.26
	18	4 314 335	443 468 564	309 503	−3.10
Endgame	8	1 747	203 840	2 747	1.25
	12	38 494	7 220 719	10 429	−3.40
	14	139 781	29 881 364	18 662	1.20
	18	2 520 848	602 221 109	62 017	−36.99

Table 5.7: Storing additional information in the transposition table.

5.9 Other Implementation Details

In this section we will describe some of the details regarding the implementation of Turska. This includes the generation of moves, which was omitted before for clarity purposes, and the detection of draw by repetition, amongst others.

Terminal Position Evaluation

The evaluation function is called when a heuristic terminal position (as stated by the quiescence search) is reached. In the case a real terminal position is reached, i.e. the player to move has no pieces left or cannot move any of his pieces (stalemate), the position is not evaluated by the evaluation function. Instead, the search returns a value lower than any value the evaluation function can return.

Draw by Repetition

Another way to end the game is by draw by repetition. When a position is reached at least three times, any one of the players can claim a draw. To detect draws by repetition we have a dedicated, small hash table that uses linear probing. The hash function used is the same as in the transposition table, but now only the high-order bits of the hash values are used, as this hash table is smaller than the transposition table. This way we avoid computing two hash values. To mitigate type-1 errors we use hash keys (see section 2.3.1), which, in this case, are also the ones used in the transposition table. Another way of detecting repetitions would be to use the transposition table. However, the transposition table can easily overload and lose entries that could be relevant.

Move Generation

In games with high branching factors it may be a waste of time to generate all the moves at each position, as the exploration of the first moves can be enough to cause a cutoff. Thus, moves are often generated incrementally. In Turska we do not do this. Implementing an incremental move generator would account for some extra overhead and complexity, which in a game with a small branching factor like Checkers are not justifiable enough.

In Checkers captures are mandatory. As in general there is no fast way to check whether there are captures or not at a certain position, due to flying kings, we first try to generate the captures. Because of the quantity-quality rule in Classical Checkers, we first generate the kings captures, as the probability of capturing more pieces with a king is higher than that of with a man. If there are no captures, then we can generate the non-capturing moves.

Move List

In the first versions of Turska a move was represented by an object that in turn represented a position. To iterate over all the moves at a certain position, we only needed to pass each object to the corresponding recursion of the search function. Although the move generation was simple, generating all the moves at a position was computational expensive, because many objects were created. Later on we decided to change the representation of moves, as a move can be represented solely by an integer, whose active bits correspond to the squares associated with the move, i.e. the square from which the piece is moving and the square to which the piece is moving, plus the squares of the captured pieces, if there are any. Although iterating over all the moves is now more complex, because a move has to be "done" and then "undone", it has a smaller memory footprint. Preliminary results showed that using this scheme the move generation was significantly faster.

5.10 Summary

We have implemented and experimented several search algorithms and techniques in Turska. In this chapter we analysed the results obtained by enabling each one incrementally. Our results seem to be in conformity with most results available in the literature, except for the case of aspiration search. We also tried some previously suggested replacement schemes for transposition tables, as well as a new one, trying to get some search performance gain. Our attempts failed miserably. Even when there was some performance gain, it was minor and not significant enough to justify the increased memory needs. A similar situation occurred in the experiments with additional information on the transposition table.

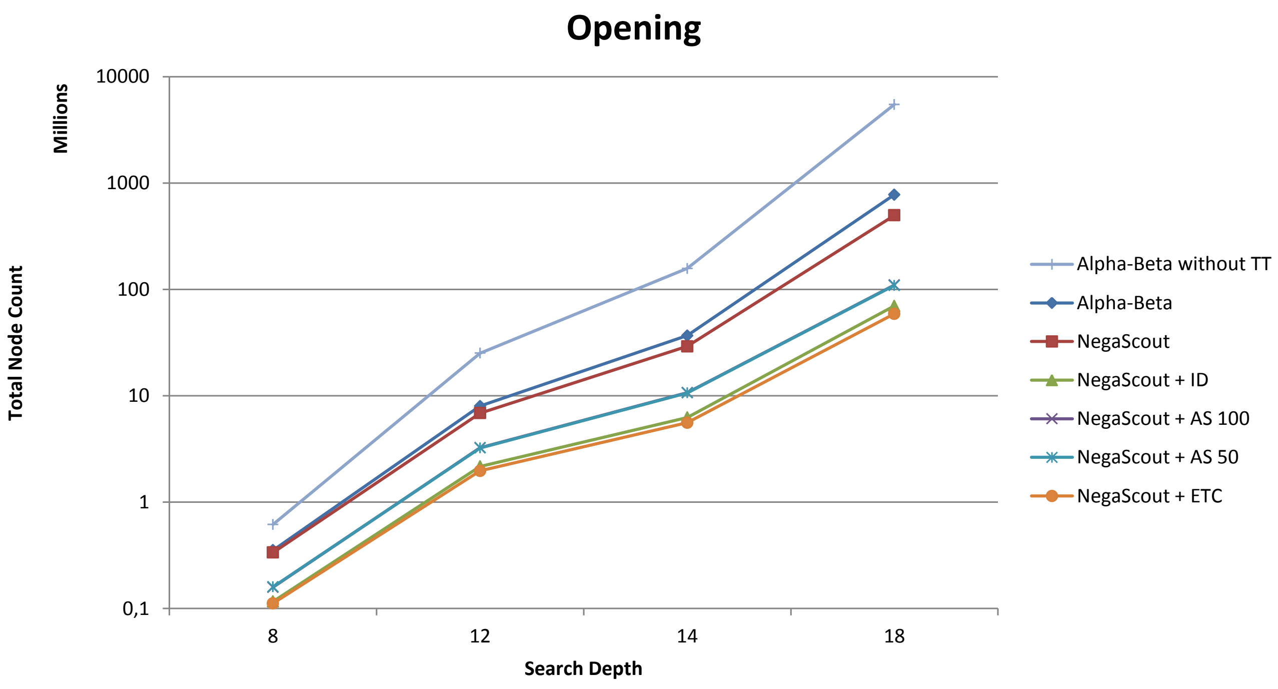


Figure 5.1: Comparison of the total node counts for each search experiment in the opening.

Nevertheless, we showed that the search space can be reduced drastically. If we compare the most naïve version with the most enhanced version, we obtain search tree size reductions within two to four orders of magnitude.

Figures 5.1, 5.2, and 5.3 show the total node counts for the search experiments, in millions. A logarithmic scale is used. In each figure are shown the results for a different test set: the opening test set, the middle game test set, and the endgame test set. Seven algorithms are compared: plain alpha-beta (named *Alpha-Beta without TT* in the figures); alpha-beta with a transposition table (*Alpha-Beta*); NegaScout with a transposition table (*NegaScout*); NegaScout with iterative deepening and a transposition table (*NegaScout + ID*); two versions of NegaScout with aspiration search, iterative deepening, and a transposition table, each one with a different δ (*NegaScout + AS 100* and *NegaScout + AS 50*); and a version of NegaScout with iterative deepening, a transposition table, and enhanced transposition cutoffs (*NegaScout + ETC*). In the figures the distinction between *NegaScout + AS 100* and *NegaScout + AS 50* is not clear, because their lines overlap. In figures 5.2 and 5.3 the distinction between *NegaScout + ID* and the *NegaScout + AS* algorithms is also not very clear. The results of the experiments are consistent. In every game phase, *NegaScout + ETC* outperforms the remaining algorithms. Notice also that the gap between *Alpha-Beta without TT* and *NegaScout + ETC* increases by about one order of magnitude between each game phase.

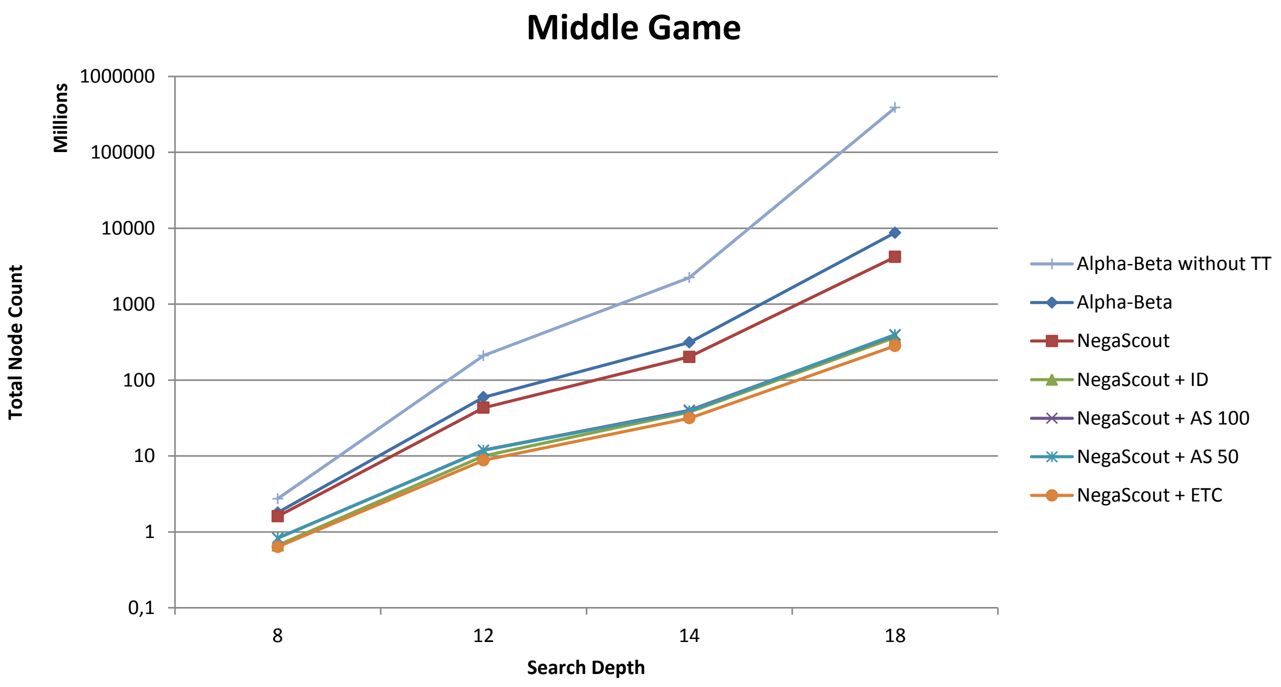


Figure 5.2: Comparison of the total node counts for each search experiment in the middle game.

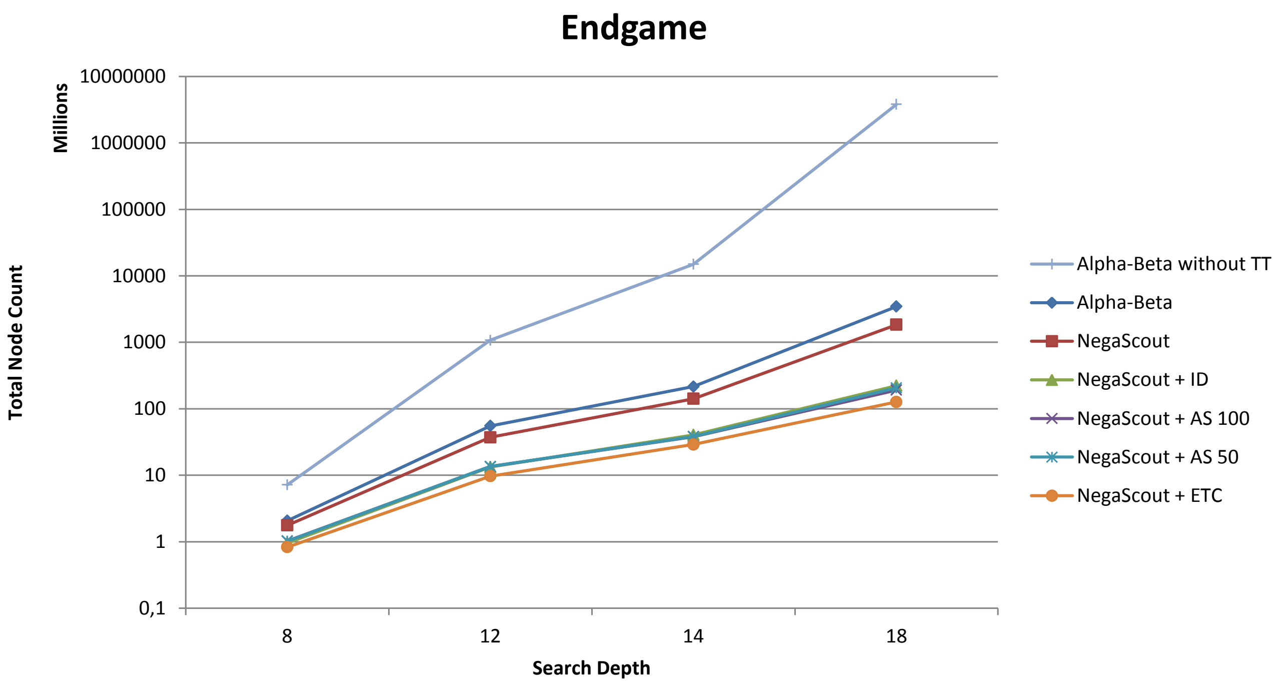


Figure 5.3: Comparison of the total node counts for each search experiment in the endgame.

6 Knowledge in Turska

Usually, the search does not reach the terminal positions due to the extensive size of the game tree. Consequently, a quiescence search is often used after the main search to extend some lines of play. The quiescence search tries to make game-playing programs apply the evaluation function only to stable positions, and thus minimize problems such as the horizon effect. Furthermore, if a game-playing program is to be competitive, the evaluation function must provide good estimations of the real utilities for the positions evaluated.

In this chapter we will describe Turska's evaluation function and quiescence search. We will also make some experiments with both of them and analyse the results obtained. Finally, we will evaluate our program by playing some matches against another Classical Checkers computer program, Windamas.

6.1 Evaluation Function

Turska's evaluation function has currently 8 heuristic functions. They will be described next, but without going into implementation details. The knowledge in the evaluation function was devised from a few sources: feedback from a Checkers expert and knowledge available in the literature of some Checkers variants [39, 14, 50, 45][1]. Some heuristics, such as material balance, piece mobility, and positional control, are also common in other games, such as Arimaa and Chess.

The program divides the game into three phases (opening, middle game, and endgame), based solely on the number of pieces on the board. The value of each heuristic may differ for each phase of the game. Table 6.1 shows how Turska divides the game into the three phases, and table 6.2 shows the values of each heuristic for each one of those game phases. In total, there are 45 parameters that can be tuned. The evaluation function was tuned manually. Due to time restrictions, limited testing and tuning was performed.

Game Phase	Number of Pieces
Opening	19–24
Middle Game	9–18
Endgame	0–8

Table 6.1: Game phases in Turska.

It may be interesting to compare Turska's heuristics with the ones used by Chinook [50, 45]. Some ideas present in Chinook can be adapted to Classical Checkers, whilst other cannot. In fact, we took

[1] `http://www.jimloy.com/checkers/checkers.htm`, `http://alemanni.pagesperso-orange.fr/page3_e.html`

Heuristic		Game Phase		
		Opening	Middle Game	Endgame
Material Balance	Man Value	100	100	100
	King Value	250	250	300
Piece Mobility	Mobile Man	1	2	3
	Mobile King	2	3	4
	Frozen Man	−5	−5	−5
	Frozen King	−12	−12	−15
	Dead Man	−20	−20	−20
	Dead King	−50	−50	−60
	Back Rank	5	5	3
	Balance (Bonus)	4	6	3
	Balance (Penalty)	−4	−6	−3
	Bridge Exploit	0	−10	−15
	River Control	0	25	50
	Runaway	50	50	60
	Triangle	10	0	0

Table 6.2: Turska's heuristics and their values.

some ideas from Chinook, the most important being the mobility classification.

Material Balance

The material balance is the most important knowledge component in the evaluation function, as it dominates all others. Usually, in a game-playing program there is a main reference value to which the values of the heuristics are relative to. In Chess it is the value of a pawn, in Checkers it is the value of a man. For example, the value of a king is equal to 2.5 times the value of a man in the opening and middle game, and equal to 3 times the value of a man in the endgame. The value of a king is greater in the endgame than in the other game phases because kings can move more freely in the endgame.

Piece Mobility

The degree to which the pieces are free or restricted in their movements is very important. Usually, a position where a player has more options is preferable over one where it has fewer options. But the number of options is not the most important factor per se, the quality of the options is. We classify the mobility of a piece on a certain direction as either *mobile*, *frozen*, or *dead*. If a piece can move freely in one direction, i.e. there is at least one move in that direction for which the piece is not immediately captured by the opponent, then it is regarded as *mobile* in that direction. If a piece can move in one direction but the moves always lead to the piece being captured, then it is regarded as *frozen* in that direction. A piece is regarded as *dead* if it cannot move at all.

In previous versions we considered that a piece was *frozen* in one direction only if the moves led to piece exchanges that were not favourable to the player. If a move led to an exchange of pieces favourable to the player, then the piece would be regarded as *mobile* in that direction. This meant that for every move tried, the complete line of play would have to be analysed, at least until the captures ended. Although this classification made Turska play a little better, it also made the program play much slower.

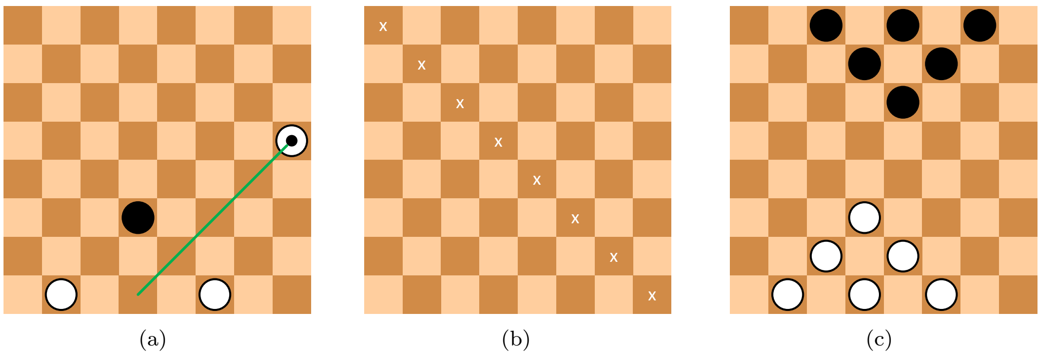


Figure 6.1: Identifying bridge exploits (a). The river (b) and the triangle formations (c).

Thus, the simpler algorithm to classify the mobility of pieces is used.

Back Rank

Since kings are very important in the game, a player should protect its back rank in order to prevent his opponent to promote men. Some defensive formations are stronger and thus are rated better. Weaker formations are rated worse. The bonus equals the number of pieces in the back rank times the value of the heuristic (see table 6.2). If a player has two pieces in the back rank and they form a strong bridge, the bonus is incremented. If they do not form any bridge, the bonus is decremented. If a player has three pieces in the back rank and none is in the double corner, the bonus is also decremented. If there are opposing kings, the bonus is decremented by two.

Balance

If a player only occupies a side of the board and the opponent has pieces in both sides, the opponent may be in advantage, as he probably has more options and unimpeded paths to promote men. Thus, it is important that a player's pieces are well distributed over the board. We say that the pieces of a player are well distributed over the board if the difference between the number of pieces in each side of the board is not greater than one. If a player's pieces are well distributed over the board, and the other player's pieces are not, the latter receives a penalty.

Bridge Exploit

A bridge is a defensive formation on the back rank of a player in which two of its own men are separated by an empty square. In figure 6.1(a) the white men on squares 2 and 4 form a (strong) bridge. If it were not for the white king on square 17, the black man on square 11 would be controlling the two white men. If any of the white men moved, it would be captured and Black would promote a man. Thus, the black man would be controlling the two white men. In the example shown, the black man is not controlling the white men because of the white king, as it can sit between one of the white men and the black man, and render Black's positional advantage useless. Identifying this kind of positions is important, as one of the players is in great disadvantage.

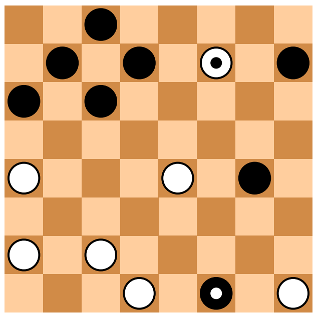


Figure 6.2: A game position.

River Control

The river ("rio" in Portuguese) is the diagonal that includes squares 1 and 32 (figure 6.1(b)). It is the most important diagonal in Classical Checkers. Controlling the river is important, especially in the endgame. In some positions in which one player is in disadvantage, controlling the river can sometimes lead to a draw, instead of a loss.

Runaway

A man is regarded as *runaway* if it can be promoted to king in one move. Identifying *runaway* men is important to evaluate positions more accurately.

Triangle

The triangle is a strong defensive formation in the opening phase of the game. The formation is shown in figure 6.1(c) for both players.

6.1.1 Static Evaluation Example

We will now give an example of the assessment of a position by the evaluation function, in order to better understand how it works. Figure 6.2 shows the game position that will be used as example. There are 15 pieces on the board, thus Turska identifies the game phase as the middle game. Hence, we will use the values from the 3rd column of table 6.2. The scores shown next are all given for White.

1. Material Balance

   White has 6 men and 1 king ($6 \times 100 + 1 \times 250 = 850$). Black has 7 men and 1 king (-950).

   Initial score: $850 + -950 = -100$ *points.*

2. Piece Mobility

   The white man on square 7 is *left mobile* (2), whereas the ones on squares 1 and 14 are *left frozen* (-10). The white men on squares 8 and 14 are *right mobile* (4). On the other hand, the white men on squares 3, 7, and 16 are *right frozen* (-15). Both kings are *mobile* in every direction they can move ($12 + -6 = 6$). The black men on squares 28 and 31 are *dead* (40). The black men on squares

	# Wins		
Depth	Simple	Complex	# Draws
8	10	89	1
12	14	74	12
14	25	64	11
18	40	47	13

Table 6.3: Comparison of evaluation functions.

13 and 24 are *left mobile* (-4) (n.b. Black's left is White's backwards right), whereas the ones on squares 23 and 27 are *left frozen* (10). While the black man on square 13 is *right mobile* (-2), the ones on squares 23 and 25 are *right frozen* (10).

Cumulative score: $-100 + 2 + -10 + 4 + -15 + 6 + 40 - 4 + 10 + -2 + 10 = -59$ *points.*

3. Balance

   White has 4 pieces on the left side of the board and 3 on the right side, thus it receives a bonus (6). Black's pieces are not well distributed over the board. Because White's pieces are well distributed over the board and Black's pieces are not, Black receives a penalty (6).

   Cumulative score: $-59 + 6 + 6 = -47$ *points.*

4. Back Rank, Bridge Exploit, River Control, Runaway, and Triangle

   Although the white men on squares 1 and 3 form a (weak) bridge, the bonus is nullified because Black has a king ($(2-2) \times 5 = 0$). The same goes for Black. There are no bridge exploits, no player has full control of the river, no men can be promoted in one move, and no player has men forming the triangle.

   Final score: -47 *points.*

6.1.2 Experimental Evaluation

We run an experiment to identify if the heuristics we created are useful. We played 100 matches between a version of our program with a simple evaluation function (named *Simple*), consisting only of material balance, and a more complex version (named *Complex*), with the heuristics just described. The values of the heuristics were as shown in table 6.2. We played 50 matches, then swapped the order of play and played the other 50 matches.

The search algorithm used was the best from the previous chapter, i.e. NegaScout with iterative deepening, a transposition table, and enhanced transposition cutoffs. The transposition table was semi-reset between moves, i.e. the moves from the previous searches were retained in the table, but the rest of the information was discarded. The program also included a random component to allow it to choose a random move in positions with more than one best move. The move is chosen from amongst those with the best score. The component was initialised with pre-defined seeds.

The results of the experiment are shown in table 6.3. For shallower searches, *Complex* is clearly superior to *Simple*, while for deeper searches, both versions score similarly. *Complex* has more knowledge

	Simple			Complex			
Depth	# Wins	Time	Eval. Depth	# Wins	Time	Eval. Depth	# Draws
8	33	17	0.68	52	548	14.02	15
12	47	216	0.65	42	6 092	14.46	11
14	41	827	0.64	44	22 275	13.91	15
18	43	7 316	0.65	41	222 309	13.26	16

Table 6.4: Comparison of quiescence searches.

and thus is able to distinguish positions better. On the other hand, *Simple*'s ability to distinguish positions increases with the depth of search. Theoretically, by increasing the search effort, less knowledge is required to achieve the same level of performance. Thus, with a sufficiently deep search, *Simple* and *Complex* should score about the same.

6.2 Quiescence Search

Turska's quiescence search is simple: the search is always extended until the side to play has no captures pending. We also tried a more complex quiescence search: instead of only extending the search until the side to play has no captures pending, we also extended the search after a capture, by one ply. Often, after a capture the other player may not perform a capture, but some of its moves may lead to other captures by the first player. This way we can follow some lines of play more extensively.

We run an experiment to see which version rated better. We played 100 matches, like in the previous experiment. Again, to the version with the simpler quiescence search we called *Simple*, and to the other we called *Complex*. *Complex*'s base search depth was always 2 ply less than *Simple*'s base search depth. The search algorithm used was the best from the previous experiment.

The results of the experiment are shown in table 6.4. Besides the number of wins and draws, it is shown the total execution time, in seconds, and the average evaluation depth, i.e. the average depth at which nodes were evaluated inside the quiescence search. For example, an evaluation depth of 14 in a 16-ply base search means that the nodes were evaluated 30 ply away from the initial position.

Although *Complex* searched deeper on average, and took about 30 times more time to play, the two versions scored similarly. Thus, the "enhanced" quiescence search is actually worse than the simpler one. Our idea for a better quiescence search turned out to be bad. The search does follow the lines of play more extensively, but too extensively.

6.3 Results Against Other Programs

We evaluated our program by playing some matches against another Classical Checkers computer program, Windamas[2] (version 7.00). Windamas was created by Jean-Bernard Alemanni in 2001 and is now freeware (n.b. its source code is not available). Alemanni has also created other computer Checkers programs.

[2] `http://alemanni.pagesperso-orange.fr/index.html`

		Turska		Windamas		
Match #	Opening	Colour	Time	Colour	Time	Result
1	free	white	31	black	37	0 – 1
2	free	black	50	white	45	0 – 1
3	11−14 22−19	white	24	black	43	0 – 1
4	11−14 22−19	black	15	white	46	0 – 1

Table 6.5: Turska vs Windamas.

Windamas has 9 levels, ranging from novice (level 0) to international grandmaster (level 8). There is not much information about what changes for each level, besides the time it takes to play. The program has also an opening book, which was disabled because Turska does not have an opening book. The feature that allows Windamas to continue searching while the opponent plays has also been disabled.

The search algorithm used in Turska is similar to the one used in sections 6.1.2 and 6.2, i.e. NegaScout with iterative deepening, a transposition table, enhanced transposition cutoffs, the more complex evaluation function, and the simpler quiescence search. The program was changed to only perform ETC at nodes that are more than 4 ply away from the horizon. Furthermore, the transposition table was semi-reset between moves, i.e. the moves from the previous searches were retained in the table, but the rest of the information was discarded. Turska's transposition table occupies 24 megabytes. We do not know how many positions can Windamas transposition table hold, thus we selected the option that was more similar to Turska's configuration. Hence, Windamas was configured to play with a transposition table that occupies 25 megabytes. Windamas played with the default level, excellency (level 5). It takes an average of 5 seconds per move (only considering the moves in which it makes some computation). In Turska this corresponds to playing with a search depth of 18 ply. Contrary to the two previous experiments, the program did not include a random component to select moves.

In official Classical Checkers tournaments, the first 2 moves are randomly chosen for each match. This means that each player is forced to play the initial move it was assigned. After that the game proceeds normally, meaning that each player can choose freely which moves to play. After the match finishes the players swap colours and then play another match.

The programs played four matches, two with free opening, and another two with the 11−14 22−19 opening. The results of the matches are shown in table 6.5. For each match it is shown the colour of each program, and the time, in seconds, that each program took to play. Windamas won all the matches.

In the first match Turska played bad, doing three bad moves almost in a row near the end of the opening phase of the game. In the second match Turska did a bad move, which made it be in material disadvantage for most part of the game. It realised quickly that the game would be a win for Windamas and did not offer much resistance. In the third match Turska played better than in the previous two. The game was balanced up until the middle game with both sides having 8 pieces. Then, the program did not play the correct moves and later fell into a trap. In the fourth match Turska also made it well into the middle game. In the late middle game, it was not able to distinguish the good move apart from the bad ones and played one of the bad moves. The game could have been easily a draw.

Although Turska can search relatively deep within a reasonable amount of time, it is still not able to distinguish between positions very well. It happens that in some cases the program does not distinguish between the real best move and the other (bad) moves. Thus, it often plays one move that is not the best. These results indicate that the current heuristics need to be further tuned, and that the program may also need more knowledge.

7 Concluding Remarks

We have developed a Classical Checkers computer program, which we named Turska. For that, we first studied the algorithms and techniques that can be used to build game-playing programs for a certain class of games: two-player zero-sum non-cooperative sequential games with perfect information. Then, we studied a state-of-the-art English Checkers computer program, Chinook, to better understand how the studied topics were used in practice. This allowed us to see which algorithms and techniques performed better in the game of Checkers. Next, we described the components of our program: the search algorithm and the evaluation function. With this we managed to answer the first problem stated back in chapter 1:

Problem Statement 1 *How can we develop a computer program to play Classical Checkers?*

In this chapter the problems stated in chapter 1 are re-addressed and evaluated. Then, we conclude by mentioning some directions for future research.

7.1 Conclusions

In chapter 4, we estimated the game-tree complexity of Classical Checkers to be approximately 10^{48}. Comparing to the one of English Checkers, 10^{31}, we concluded that Classical Checkers is harder, as the game-tree complexity is higher. As both variants are very similar, and given that the state-space complexities are very similar as well, Classical Checkers can probably also be solved with current technology.

Problem Statement 2 *How do the studied search algorithms and techniques behave in the domain under study, and how can they be improved?*

We experimented and analysed several search enhancements in chapter 5, most related with transpositions and search windows. For sufficiently deep searches, all enhancements, except aspiration search, accounted for significant reductions of the search tree. In total, the size of the search tree was reduced by two to four orders of magnitude. The reductions increased with the depth of search and the game phase. In general, the later the stage of game, the more noticeable were the enhancements. The enhancements that take advantage of transpositions, such as transposition tables and enhanced transposition cutoffs, were the ones that accounted for more savings, especially when combined with iterative deepening. Our results seem to be in conformity with most results available in the literature, except for the case of aspiration search. The bad results obtained with aspiration search indicate that Turska's evaluation function needs more work. This was later confirmed in chapter 6.

To answer the second part of the question above we focused on transposition tables. Given that the enhancements that take advantage of transpositions were the ones that accounted for more savings, we tried to improve the information stored in the transposition table and how it was managed. We did so by trying to ensure that the most relevant information stayed in the table (using different replacement schemes), and by storing additional and possibly relevant information in the table. Although there were minor improvements with some of the replacement schemes, they were not significant enough to justify the increased memory needs. The results of the experiments with additional information were similar: in general, the additional information did not improve the search. Thus, the additional information does not appear to be very relevant.

Problem Statement 3 *What game-specific knowledge is required to improve our program?*

In chapter 6 we made some experiments with Turska's evaluation function and quiescence search. The results indicate that the evaluation function needs further work. The matches against Windamas showed that in some cases Turska is not able to distinguish between positions very well. Thus, it often plays one move that is not the best. In Checkers it is especially important to not make mistakes, particularly in the opening phase of the game. Although sometimes it may not be immediately clear, a bad move done in the beginning of the game directly leads to a no-win, and often to a loss.

7.2 Future Work

Some enhancements can be made to the current research work. We suggest the following:

- *Evaluation Function*
  The evaluation function needs to be further tuned. It may be interesting to investigate machine-learning methods to do so. Limited success was already achieved in English Checkers with such methods [51]. Moreover, the evaluation function may also need more knowledge. Hence, working with a Checkers expert may continue to be relevant.

- *Parallelism*
  The brute force power of Turska's search component can be further improved with parallelisation. For that, parallel search algorithms have to be studied (see, e.g., [9] for a survey). This includes algorithms that use tree-splitting, work-stealing, and other parallelisation techniques.

- *Selective Search*
  Our program did not use many selective search techniques besides quiescence search. Thus, it may be interesting to investigate how other techniques, such as forward pruning and search extensions, would perform (see, e.g., section 2.3.5 for a survey).

- *Databases*
  Although not very interesting from a scientific viewpoint, an opening book and endgame databases would improve Turska's playing strength considerably. The latter are crucial to solve the game.

A The Rules of Checkers in Brief

Checkers is played by two players, sitting on opposite sides of a checkerboard – a rectangular square-tiled board in which the squares are of alternating dark and light colour. One of the players has white, or light, pieces and the other black, or dark, ones. The former player is referred to as **White** and the latter as **Black**. In Classical Checkers White moves first, whereas in English Checkers Black does. Players play alternately and may only move their pieces. The playable surface consists only of the dark squares. The goal of the game is to capture all the opponent's pieces or leave the opponent without any legal move. Figures A.1(a) and A.1(d) show the starting positions for a game of Classical and English Checkers, respectively. In Classical Checkers the double corner is to the left of each player, whereas in English Checkers it is to the right of each player. In figures A.1(b) and A.1(e) are shown the standard board notation for each one of these two variants.

Pieces may only move diagonally and to empty (vacant) squares and captures are mandatory. To capture an opposing piece, the piece that is performing the capture has to jump over the other and land on an empty square on that diagonal. A capture may only be done if the diagonally adjacent square behind the piece that is being captured is empty. Multiple pieces may be captured in one move. This is done by performing several successive jumps, each jump corresponding to a capture. It is illegal to jump over the same piece more than once. Only after a capture is finished, may the captured pieces be removed from the board. In Classical Checkers, if there are several captures possible, the player must choose the one that captures the maximum number of pieces (quantity rule). If several of these captures capture the same number of pieces, the player must choose the one that captures more kings (quality rule). If after applying these two rules there are still a few possible captures, then the player may choose whichever he wants. In English Checkers the rules are simpler: when there are several captures possible, the choice is always up to the player. A player loses the game when he has no pieces or cannot move any of his pieces.

There are two kinds of pieces in Checkers: **men** and **kings**. A man can be promoted to, or crowned, king. To differentiate between men and kings, an additional piece is placed on the top of each piece that is promoted (the placed piece must be of the same colour). At the beginning of the game all pieces are men, and for a man to be promoted to king it has to reach the kings row, which is the farthest row opposite to the player controlling the piece.

Men may only move one square forwards and they capture by advancing two squares in the same direction, jumping over the opposing piece in the intermediate square. Although in most variants, such as the Classical and English, men may only capture forwards, in some variants they may also capture

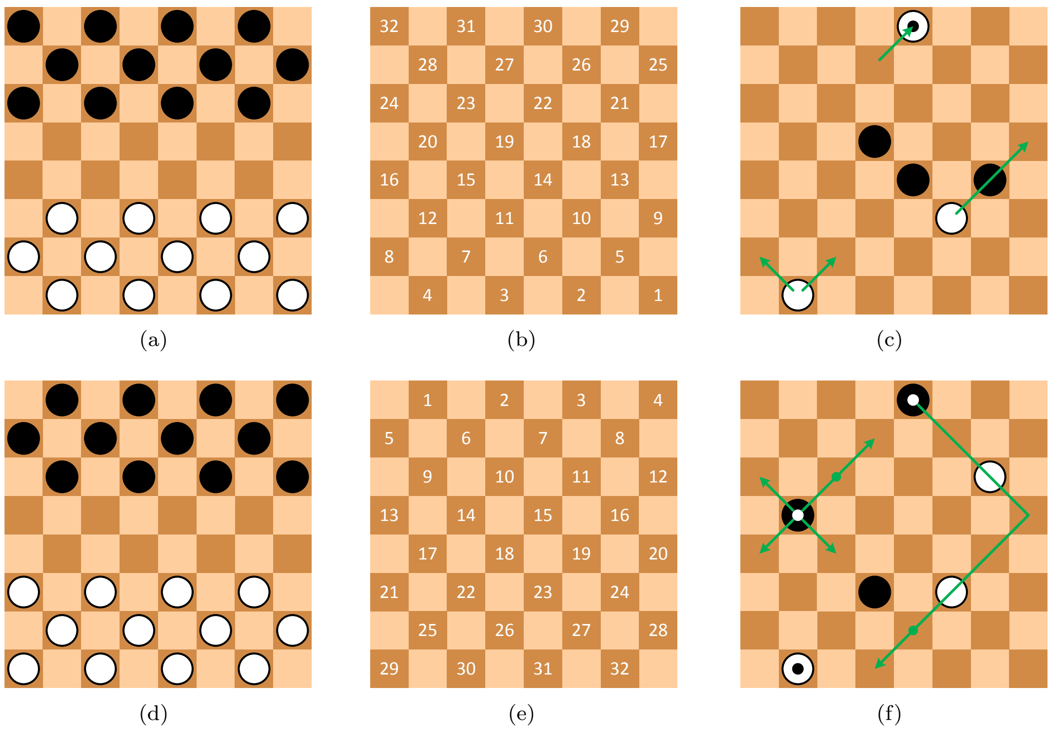


Figure A.1: Several situations in Classical and English Checkers.

backwards.

There are two types of kings: kings and **flying kings**. Both types of kings may move and capture in any direction, i.e. both forwards and backwards. While kings may move only one square, flying kings may move any number of squares along unblocked diagonals. Unlike men and kings, flying kings do not need to be adjacent to an opposing piece to capture it, they need only to be on the same diagonal and the path to the piece must be free. When performing a capture, flying kings may land in any empty square following the captured piece, whereas men and kings have to land on the adjacent square. While in Classical Checkers kings are flying kings, in English Checkers they are not.

In figure A.1(c) several situations using white men in Classical Checkers are shown:

- The white man on square 4 has two possible moves: from square 4 to square 7 (4−7) and from square 4 to square 8 (4−8).
- The white man on square 10 has a mandatory capture: from square 10 to square 17 (10×17), capturing the black man on square 13 in between.
- The white man on square 27 moved to square 30 (27−30) and was promoted to king.

In figure A.1(f) several situations using black (flying) kings in Classical Checkers are shown:

- 20−15, 20−16, 20−23, 20−24, and 20−27 – a king with five possible moves.
- 30×17×3 and 30×17×6 – a capture with two possible landings.

B Proof-Number Search

The Proof-Number Search (PNS or pn-search) algorithm was introduced by Allis in 1994 [2]. Since then, PNS has been used to solve several games, such as English Checkers and others [49, 59]. Several variants of the algorithm exist [60]. We will describe the basic PNS algorithm, but first we will introduce AND/OR trees, as they will be necessary to better understand PNS.

B.1 AND/OR Trees

An **AND/OR tree** is a (rooted) tree with some extra properties. In an AND/OR tree there are two types of nodes: **AND nodes** and **OR nodes**. Each node can have one of three values: *true*, *false*, or *unknown*. A node whose value is either *true* or *false* is a **terminal node**.

The value of an AND node A is determined by applying the following rules (in order of precedence):

1. If A has at least one child with value *false*, then the value of A is *false*.
2. If A has at least one child with value *unknown*, then the value of A is *unknown*.
3. Otherwise, the value of A is *true*.

The value of an OR node O is determined by applying the following rules (in order of precedence):

1. If O has at least one child with value *true*, then the value of O is *true*.
2. If O has at least one child with value *unknown*, then the value of O is *unknown*.
3. Otherwise, the value of O is *false*.

An AND/OR tree is **solved** if the value of its root has been established as either *true* or *false*. A solved tree with value *true* is said to be **proved**, whereas a solved tree with value *false* is said to be **disproved**.

B.2 Proof-Number Search

Proof-Number Search (**PNS** or **pn-search**) is a best-first search algorithm especially suited for finding the game-theoretical value of games. A game tree can be seen as an AND/OR tree in which Max nodes are OR nodes and Min nodes are AND nodes. If the game-theoretical value is a win, the game tree is

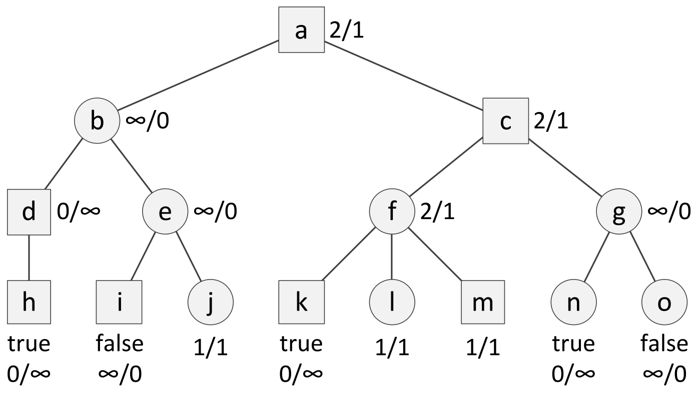


Figure B.1: An AND/OR tree annotated with its proof and disproof numbers.

proved, otherwise (loss or draw) it is disproved. Unlike other best-first search algorithms, PNS does not need application-dependent knowledge to determine the most-promising node, i.e. which node to explore next. In PNS, the most-promising node is referred to as the most-proving node. The **most-proving node** of an AND/OR tree T is a leaf node, which by obtaining the value *true* reduces T's proof number by 1, and by obtaining the value *false* reduces T's disproof number by 1.

The **proof number** of an AND/OR tree T is the minimum number of leaf nodes that have to be proved to prove the tree. Conversely, the **disproof number** of T is the minimum number of leaf nodes that have to be disproved to disprove the tree. Figure B.1 shows and AND/OR tree where each node has an associated proof and disproof number, which are shown by this order on the right or below each node. In terminal nodes, the value is shown (true or false). AND nodes are depicted with circles and OR nodes with squares. Non-terminal leaf nodes (j, l, and m) have proof and disproof number 1, because only the node itself needs to obtain the value *true* (*false*) to prove (disprove) the subtree rooted at itself. Terminal nodes with value *true* (*false*) have proof (disproof) number 0, since their values have already been proved (disproved). Terminal nodes with value *false* (*true*) have proof (disproof) number ∞, since nothing can undo the fact that the nodes have been disproved (proved).

The PNS algorithm is shown in figure B.2. The function EVALUATE takes a node as argument and assigns to it the value *true*, *false*, or *unknown*. The attribute *pn* of a node represents its proof number and the attribute *dn* its disproof number. PROOF-NUMBER-SEARCH encodes the main loop of the search. First, the root of the AND/OR tree is evaluated and developed, i.e. its children are generated and evaluated. Then, at each iteration a node is selected, developed, and its proof and disproof numbers are propagated through its ancestors. The algorithm terminates when the tree is solved, i.e. when either the proof or disproof number of the root becomes 0. To prove an AND node, all of its children have to be proved. Thus, its proof number equals the sum of its children proof numbers. Conversely, the disproof number of an OR node equals the sum of the disproof numbers of its children. To disproof an AND node, only one of its children has to be disproved, thus its disproof number equals the disproof number of the child with the smallest disproof number. To prove an OR node, only one of its children has to be proved. Function SET-NUMBERS computes the proof and disproof numbers of a node as described above.

```
PROOF-NUMBER-SEARCH($root$)
    EVALUATE($root$)
    SET-NUMBERS($root$)

    while $root.pn \neq 0$ and $root.dn \neq 0$ do
        $node$ = SELECT-MOST-PROVING-NODE($root$)
        DEVELOP-NODE($node$)
        PROPAGATE-NUMBERS($node$)

    if $root.pn == 0$ then
        $root.value$ = TRUE
    else if $root.dn == 0$ then
        $root.value$ = FALSE

SET-NUMBERS($node$)
    if $node$ is developed then
        if $node$ is AND then
            $node.pn = \sum_{n \in \text{CHILDREN}(node)} n.pn$
            $node.dn = \text{MIN}_{n \in \text{CHILDREN}(node)} n.dn$
        else // $node$ is OR
            $node.pn = \text{MIN}_{n \in \text{CHILDREN}(node)} n.pn$
            $node.dn = \sum_{n \in \text{CHILDREN}(node)} n.dn$
    else
        if $node.value$ == TRUE then
            $node.pn = 0$
            $node.dn = \infty$
        else if $node.value$ == FALSE then
            $node.pn = \infty$
            $node.dn = 0$
        else // $node.value$ == UNKNOWN
            $node.pn = 1$
            $node.dn = 1$

SELECT-MOST-PROVING-NODE($node$)
    while $node$ is developed do
        $i = 1$
        if $node$ is AND then
            while $node.child[i].dn \neq node.dn$ do
                $i = i + 1$
        else // $node$ is OR
            while $node.child[i].pn \neq node.pn$ do
                $i = i + 1$
        $node = node.child[i]$

    return $node$

DEVELOP-NODE($node$)
    for each node $n \in$ CHILDREN($node$) do
        EVALUATE($n$)
        SET-NUMBERS($n$)

PROPAGATE-NUMBERS($node$)
    while $node \neq$ NULL do
        SET-NUMBERS($node$)
        $node = node.parent$
```

Figure B.2: The Proof-Number Search algorithm.

To obtain the most-proving node we start at the root and use the following procedure. If the current node is an AND node, we traverse the child with the lowest disproof number, otherwise we traverse the child with the lowest proof number. When a leaf node is reached we have found a most-proving node. Function SELECT-MOST-PROVING-NODE selects the most-proving node as just described.

Bibliography

[1] Selim G. Akl and Monroe M. Newborn. The principal continuation and the killer heuristic. In *ACM Annual Conference*, pages 466–473, 1977.

[2] Louis V. Allis. *Searching for Solutions in Games and Artificial Intelligence.* PhD thesis, Maastricht University, 1994.

[3] Thomas S. Anantharaman. *A Statistical Study of Selective Min-Max Search in Computer Chess.* PhD thesis, Carnegie Mellon University, 1990.

[4] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways for Your Mathematical Plays.* Academic Press, 1982.

[5] Hans J. Berliner. *Chess as Problem Solving: The Development of a Tactics Analyser.* PhD thesis, Carnegie Mellon University, 1974.

[6] Jonh A. Birmingham and Peter Kent. Tree-searching and tree-pruning techniques. In *Advances in Computer Chess*, volume 1, pages 89–107. Edinburgh University Press, 1977.

[7] Yngvi Björnsson. *Selective Depth-First Game-Tree Search.* PhD thesis, University of Alberta, 2002.

[8] Dennis M. Breuker. *Memory versus Search in Games.* PhD thesis, Maastricht University, 1998.

[9] Mark G. Brockington. A taxonomy of parallel game-tree search algorithms. *International Computer Chess Association Journal*, 19(3):162–174, 1996.

[10] Alexander L. Brudno. Bounds and valuations for abridging the search of estimates. *Problems of Cybernetics*, 10:225–241, 1963.

[11] Michael Buro. *Techniken für die Bewertung von Spielsituationen anhand von Beispielen.* PhD thesis, University of Paderborn, 1994.

[12] Michael Buro. Experiments with multi-probcut and a new high-quality evaluation function for othello. In *Games in AI Research*, pages 77–96. Maastricht University, 2000.

[13] Murray S. Campbell and Thomas. A. Marsland. A comparison of minimax tree search algorithms. *Artificial Intelligence*, 20(4):347–367, 1983.

[14] Sena Carneiro. *Iniciação ao Jogo das Damas.* Editorial Presença, 1978.

[15] Shirish Chinchalkar. An upper bound for the number of reachable positions. *International Computer Chess Association Journal*, 19(3):181–183, 1996.

[16] John H. Conway. *On Numbers and Games.* Academic Press, 1976.

[17] Adrianus D. de Groot. *Thought and Choice in Chess.* Mouton Publishers, 1965.

[18] Veríssimo N. Dias. Personal Communication, 2011.

[19] Christian Donninger. Null move and deep search: selective-search heuristics for obtuse chess programs. *International Computer Chess Association Journal*, 16(3):137–143, 1993.

[20] John P. Fishburn. *Analysis of Speedup in Distributed Algorithms.* PhD thesis, University of Wisconsin-Madison, 1981.

[21] James J. Gillogly. *Performance Analysis of the Technology Chess Program.* PhD thesis, Carnegie Mellon University, 1978.

[22] Richard D. Greenblatt, Donald E. Eastlake, III, and Stephen D. Crocker. The greenblatt chess program. In *AFIPS Fall Joint Computer Conference*, pages 801–810, 1967.

[23] Ernst A. Heinz. Extended futility pruning. *International Computer Chess Association Journal*, 21(2):75–83, 1998.

[24] Patrick P. L. M. Hensgens. A knowledge-based approach of the game of amazons. Master's thesis, Maastricht University, 2001.

[25] Andreas Junghanns. Are there practical alternatives to alpha-beta in computer chess? *International Computer Chess Association Journal*, 21(1):14–32, 1998.

[26] Andreas Junghanns and Jonathan H. Schaeffer. Search versus knowledge in game-playing programs revisited. In *International Joint Conference on Artificial Intelligence*, pages 692–697, 1997.

[27] Hermann Kaindl, Reza Shams, and Helmut Horacek. Minimax search algorithms with and without aspiration windows. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(7):1225–1235, 1991.

[28] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[29] Robert M. Lake, Jonathan H. Schaeffer, and Paul Lu. Solving large retrograde analysis problems using a network of workstations. In *Advances in Computer Chess*, volume 7, pages 135–162. Maastricht University, 1994.

[30] Paul Lu. Parallel search of narrow game trees. Master's thesis, University of Alberta, 1993.

[31] Thomas. A. Marsland and Murray S. Campbell. Parallel search of strongly ordered game trees. *Computing Surveys*, 14(4):533–551, 1982.

[32] Thomas A. Marsland and Fred Popowich. Parallel game-tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7(4):442–452, 1985.

[33] Dana S. Nau. *Quality of Decision versus Depth of Search on Game Trees.* PhD thesis, Duke University, 1979.

[34] Judea Pearl. Asymptotic properties of minimax trees and game-searching procedures. *Artificial Intelligence*, 14(2):113–138, 1980.

[35] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, 1984.

[36] Aske Plaat. *Research Re: Search & Re-Search.* PhD thesis, Erasmus University Rotterdam, 1996.

[37] Alexander Reinefeld. An improvement to the scout tree search algorithm. *International Computer Chess Association Journal*, 6(4):4–14, 1983.

[38] Alexander Reinefeld. *Spielbaum-Suchverfahren.* Springer-Verlag, 1989.

[39] Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[40] Arthur L. Samuel. Some studies in machine learning using the game of checkers. ii – recent progress. *IBM Journal of Research and Development*, 11(6):601–617, 1967.

[41] Jonathan H. Schaeffer. *Experiments in Search and Knowledge.* PhD thesis, University of Waterloo, 1986.

[42] Jonathan H. Schaeffer. The games computers (and people) play. In *Advances in Computers*, volume 50, pages 189–266. Academic Press, 2000.

[43] Jonathan H. Schaeffer. Search ideas in chinook. In *Games in AI Research*, pages 19–30. Maastricht University, 2000.

[44] Jonathan H. Schaeffer. Technology transfer from one high-performance search engine to another. *International Computer Chess Association Journal*, 24(3):131–146, 2001.

[45] Jonathan H. Schaeffer. *One Jump Ahead: Computer Perfection at Checkers.* Springer, 2009.

[46] Jonathan H. Schaeffer. Personal Communication, 2010.

[47] Jonathan H. Schaeffer, Yngvi Björnsson, Neil Burch, Akihiro Kishimoto, Martin Müller, Robert M. Lake, Paul Lu, and Steve Sutphen. Solving checkers. In *International Joint Conference on Artificial Intelligence*, pages 292–297, 2005.

[48] Jonathan H. Schaeffer, Yngvi Björnsson, Neil Burch, Robert M. Lake, Paul Lu, and Steve Sutphen. Building the checkers 10-piece endgame databases. In *Advances in Computer Games*, volume 10, pages 193–210. Kluwer Academic Publishers, 2003.

[49] Jonathan H. Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert M. Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007.

[50] Jonathan H. Schaeffer, Joseph C. Culberson, Norman C. Treloar, Brent Knight, Paul Lu, and Duane Szafron. Reviving the game of checkers. In *Heuristic Programming in Artificial Intelligence: The Second Computer Olympiad*, pages 119–136. Ellis Horwood, 1991.

[51] Jonathan H. Schaeffer, Markian Hlynka, and Vili Jussila. Temporal difference learning applied to a high-performance game-playing program. In *International Joint Conference on Artificial Intelligence*, pages 529–534, 2001.

[52] Jonathan H. Schaeffer and Robert M. Lake. Solving the game of checkers. In *Games of No Chance*, pages 119–133. Cambridge University Press, 1996.

[53] Jonathan H. Schaeffer, Robert M. Lake, Paul Lu, and Martin Bryant. Chinook: the world man-machine checkers champion. *AI Magazine*, 17(1):21–29, 1996.

[54] Claude E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41(314):256–275, 1950.

[55] David J. Slate and Lawrence R. Atkin. Chess 4.5 – the northwestern university chess program. In *Chess Skill in Man and Machine*, pages 82–118. Springer-Verlag, 1977.

[56] Christopher S. Strachey. Logical or non-mathematical programmes. In *Proceedings of the ACM National Meeting (Toronto)*, pages 46–49, 1952.

[57] Thomas Ströhlein. Untersuchungen über kombinatorische spiele. Master's thesis, Technical University of Munich, 1970.

[58] Thomas R. Truscott. The duke checkers program. *Journal of Recreational Mathematics*, 12(4):241–247, 1979–80.

[59] Hendrik J. van den Herik, Jos W. H. M. Uiterwijk, and Jack van Rijswijck. Games solved: now and in the future. *Artificial Intelligence*, 134(1–2):277–311, 2002.

[60] Hendrik J. van den Herik and Mark H. M. Winands. Proof-number search and its variants. In *Oppositional Concepts in Computational Intelligence*, pages 91–118. Springer, 2008.

[61] John von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior.* Princeton University Press, 1944.

[62] Mark H. M. Winands. *Informed Search in Complex Games.* PhD thesis, Maastricht University, 2004.

[63] Ernst F. F. Zermelo. Über eine anwendung der mengenlehre auf die theorie des schachspiels. In *Proocedings of the Fifth International Congress of Mathematicians*, pages 501–504, 1913.

[64] Albert L. Zobrist. A new hashing method with application for game playing. Technical Report 88, University of Wisconsin-Madison, 1970.