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 document, 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. Turska was further evaluated by playing some matches against another Classical Checkers computer program.

Keywords: Classical Checkers, Game Playing, Adversarial Search, Alpha-Beta Search

1 Introduction

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.

Games are interesting research subjects for several reasons. First, games have a closed domain with well-defined rules, in contrast to real-world problems, which makes them easier to represent and analyse. Second, 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. Finally, games can be used to test new ideas in problem solving.

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.

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. In this document we describe a computer program created to play the game of Classical Checkers, Turska. Checkers is a kind of board game played between two players that involves diagonal moves of uniform pieces and mandatory leap captures. Furthermore, it is a two-player deterministic zero-sum non-cooperative sequential game with perfect information.
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. 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.

Briefly, in Checkers, pieces may only move diagonally and to empty squares and captures are mandatory. There are two kinds of pieces: men and kings. 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. In Classical Checkers, 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. Kings, on the other hand, may move any number of squares along unblocked diagonals. The full list of rules for the game of Classical Checkers can be consulted elsewhere [5].

We start by presenting the background in section 2. Then, in section 3 we estimate and analyse the complexity of Classical Checkers. To the best of our knowledge, this is the first time the complexity of Classical Checkers is estimated and analysed. In section 4 we describe the main components of Turska. We also analyse the results of some of the experiments made with the program to understand how the program could be improved. Finally, in section 5 we conclude this work by discussing the results obtained, drawing some conclusions, and discussing future work directions.

This document is an abridged version of [11].

2 Background

Humans play games by considering different combinations of possible moves for themselves and possible replies by their opponents. Each legal combination of moves and replies is a possible line of play.

Computers try to mimic humans by systematically analysing the different lines of play and building a game tree with that information. 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.

In this section we introduce a method that allows computers to play games, minimax, and an improvement to it, alpha-beta, as well as some techniques to enhance alpha-beta. We finish by taking a brief look at Chinook, a state-of-the-art English Checkers computer program.

2.1 Minimax

In what follows, we are only concerned with two-player zero-sum non-cooperative sequential games with perfect information. In these games, the first player is called Max and the second, Max’s opponent, is called Min. Max’s goal is to maximize its own utility, whilst Min’s goal is to minimize Max’s utility. The utility of a position is a numeric value that represents the best possible game outcome from that position for a certain player. For example, in Checkers there are three possible outcomes for a game: win, loss, or draw, with utilities +1, −1, and 0, respectively. To obtain the utility of a position for Min we only have to negate its value for Max, due to the zero-sum property.

To determine which line of play is the best we can use minimax [26,25]. 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 value of the game tree.
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 tree.

After finding the value \( x \) of one child of some Max node \( n \), Max should only be concerned with moves that have a value greater than \( x \), as he is trying to maximize his gains. Thus, \( x \) is a lower bound on the actual minimax value of \( n \). Moreover, the lower bound is an upper bound for Min. If Min has a reply with a value inferior to \( x \), there is no need to further search the current branch, as it will not affect the value of the root, and thus it can be pruned. Conversely, a Min node can impose an upper bound on its Max children.

The alpha-beta (\( \alpha-\beta \)) [14] algorithm introduces two bounds: \( \alpha \), the lower bound, and \( \beta \), the upper bound. The algorithm is similar to minimax, but while it traverses the game tree it keeps track of its two parameters and prunes away subtrees with values that fall outside the search window, i.e. the interval between \( \alpha \) and \( \beta \).

Like minimax, alpha-beta must also traverse the game tree down to the terminal nodes. In most games, this is not possible due to the massive size of the game tree. Thus, the search must stop at some point. 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. A commonly used criterion to decide when to stop the search and apply the evaluation is the search depth.

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. The minimal tree is the minimal part of a game tree necessary to determine the minimax value of the root. The number of nodes of the minimal tree is in the order of the square root of the number of nodes of the game tree [14].

There are many techniques to enhance alpha-beta in order to bring it closer to the theoretical limit. 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. Iterative deepening (ID) [7,23] and the history heuristic [19] are examples of two well-known techniques that are used to improve move ordering. 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., [4,13]). Aspiration search (AS) [3,9] and NegaScout (NS) [17,18] are examples of two algorithms that take advantage of artificially reduced search windows.

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. If we detect these repeated positions, we can eliminate the redundant effort that is wasted by re-searching them. A transposition table (TT) [10] serves as a cache of recently explored positions. 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. The enhanced transposition cutoffs (ETC) [16] technique is often used along with transposition tables to maximize the use of the information stored in the transposition tables.

2.4 Case Study: Chinook

The Chinook\(^1\) project was created in 1989, at the University of Alberta, Canada. 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 [22], and in 1997 retired from competitive play. In 2007, English Checkers was solved by the Chinook team [20]. The game-theoretic value is draw.

Game-playing programs are usually composed by two main components: search and knowledge. Chinook’s search component consists of a parallel NegaScout with some enhancements, such as aspiration search, iterative deepening, the history heuristic, transposition tables, enhanced transposition cutoffs, and several application-dependent search extensions and reductions. The knowledge component is composed by an evaluation function, an opening book, and endgame databases.

3 Complexity of Classical Checkers

The complexity of a game is usually described using two different measures: the state-space complexity and the game-tree complexity [1]. In this section 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.

Games for which the game-theoretic value, i.e. the outcome when all players play optimally, are 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. Both state-space and game-tree complexities are important factors in determining the solvability of a game [1,24].

3.1 State-Space Complexity

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.

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 [21]. The number of positions having \( n \) or fewer pieces on the board can be calculated with:

\[
\sum_{w=0}^{\text{Min}(n,12)} \sum_{b=0}^{\text{Min}(n-w,12)} \sum_{W=0}^{\text{Min}(n-w-W,12)} \sum_{B=0}^{\text{Min}(w,4)} \sum_{f=0}^{\text{Min}(n-w-W-b-W,12)} \text{NP}(w,b,W,B,f) - 1,
\]

\[
\text{NP}(w,b,W,B,f) = \binom{4}{f} \binom{24}{w-f} \binom{28-(w-f)}{b} \binom{32-w-b}{W} \binom{32-w-b-W}{B}
\]

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

\(^1\)http://webdocs.cs.ualberta.ca/~chinook/
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 total number of positions is approximately equal to $5 \times 10^{20}$.

The calculation above is misleading, as it considers positions that cannot occur in a game. For example, given a reasonable playing strategy, no position with 24 pieces on the board in which some are kings can be reached. We can obtain a tighter bound on the state-space complexity of Classical Checkers by taking some assumptions into account, like for example, only considering 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 [11]. The new estimation for the state-space complexity of Classical Checkers is then $4 \times 10^{19}$.

### 3.2 Game-Tree Complexity

The game-tree complexity of game 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.

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.

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 [8].

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 obtained indicate that the average branching factor is relatively stable, around 6.4. For the average game length we obtained values between 53 and 66. Conferencing with a Checkers expert we concluded that the best estimate for the average game length would be of about 60 ply [8], 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 $6.4^{60} = 2 \times 10^{48}$.

### 3.3 Comparison with Other Games

In table 1 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. [21]. The high
Table 1. The complexity of Classical Checkers compared to that of other games.

<table>
<thead>
<tr>
<th>Game</th>
<th>Length</th>
<th>Branching Factor</th>
<th>Complexity State-Space</th>
<th>Complexity Game-Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>English Checkers [15,1,21]</td>
<td>70</td>
<td>2.8</td>
<td>$10^{15}$</td>
<td>$10^{21}$</td>
</tr>
<tr>
<td><strong>Classical Checkers</strong></td>
<td><strong>60</strong></td>
<td><strong>6.4</strong></td>
<td>$10^{19}$</td>
<td>$10^{48}$</td>
</tr>
<tr>
<td>International Checkers [1]</td>
<td>90</td>
<td>4</td>
<td>$10^{30}$</td>
<td>$10^{54}$</td>
</tr>
<tr>
<td>Othello [1]</td>
<td>58</td>
<td>10</td>
<td>$10^{28}$</td>
<td>$10^{58}$</td>
</tr>
<tr>
<td>Chess [1,6]</td>
<td>80</td>
<td>35</td>
<td>$10^{46}$</td>
<td>$10^{123}$</td>
</tr>
<tr>
<td>Amazons (10 × 10) [12]</td>
<td>84</td>
<td>374, 299</td>
<td>$10^{10}$</td>
<td>$10^{212}$</td>
</tr>
<tr>
<td>Go (19 × 19) [1]</td>
<td>150</td>
<td>250</td>
<td>$10^{172}$</td>
<td>$10^{359}$</td>
</tr>
</tbody>
</table>

similarities between English Checkers and Classical Checkers may be an indicator that the latter could also be solved with current technology.

Looking at table 1 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.

4 Turska

We have implemented and experimented several search algorithms and techniques in our Classical Checkers computer program, Turska. In this section we will analyse them. We will also analyse several replacement schemes for transposition tables. We will finish by describing Turska’s evaluation function and by analysing the matches played between Turska and another Classical Checkers computer program, Windamas.

Each experiment was repeated several times, each time with a different base search depth. Our test set consists of 140 positions and is divided into three subsets, each one related to a specific game phase: opening, middle game, and endgame. This way we can better analyse how each enhancement affects Turska, as some enhancements have been identified to be more useful in certain phases of the game.

4.1 Transposition Tables, NegaScout, and Iterative Deepening

In this experiment, we compared four algorithms: alpha-beta (named Alpha-Beta without TT in the figures), alpha-beta using a transposition table (Alpha-Beta), NegaScout using a transposition table (NegaScout), and NegaScout using iterative deepening and a transposition table (NegaScout + ID). Figure 1(a) shows the results of the experiment for the middle game test set. A logarithmic scale is used. The comparison measure used is the total node count, which is the number of nodes visited for all positions in the test set.

As we can see from figure 1(a), the use of transpositions tables is essential in any game-playing program. The total number of nodes visited reduced drastically, about two orders of magnitude. The enhancement is more noticeable with deeper searches, as the number of transpositions increases. Although the gains of using NegaScout are significant, they are more noticeable with deeper searches, as the parts of the game tree that are pruned are expectedly bigger. 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 combined with iterative deepening and transposition tables.

By itself, iterative deepening is not very useful, but when used along with a transposition table, it 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.

The results for the other game phases were similar for all three enhancements.

4.2 Aspiration Search and Enhanced Transposition Cutoffs

In this experiment, we took the NegaScout version that uses iterative deepening and a transposition table from the previous experiment and added aspiration search to two of the versions and enhanced transposition cutoffs to another (NS + ETC). The search window was centred around the score obtained from the previous iteration with an expected error of 100 in one version (NS + AS 100), and an expected error of 50 in the other (NS + AS 50). Figure 1(b) shows the results of the experiment for the middle game test set.

Some studies have shown that the use of aspiration search can lead to search reductions of about 20% (see, e.g., [9]). 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. 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.

The results of aspiration search for the opening phase of the game were worse, whilst for the endgame there were minor gains for sufficiently deep searches. As expected, enhanced transposition cutoffs accounted for significant gains in all game phases.
4.3 Replacement Schemes for Transposition Tables

When a collision occurs, i.e. a position is about to be stored in the transposition table and there is no free entry available, an existing entry has to be replaced. If the relevant information stays in the transposition table, the search may be more efficient. In the previous experiments, we only took the depth of the subtree rooted at the position into consideration. In this experiment we compared three replacement schemes: Deep, the one used up until now; Big, for which what matters is the size of the subtree, and not the depth of the subtree; and Number of Hits, for which the depth or size of the subtree is ignored and what matters is the number of times a position was encountered. The latter is a new replacement scheme, whereas the other two were already experimented before by other authors [2].

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.

Figure 2 shows the results of the experiment for the middle game test set. 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 scheme is slightly better than the Deep scheme. Nevertheless, the Deep scheme is preferred, as it is simpler and does not have the increased memory needs of the Big scheme.

4.4 Evaluation Function

Turska’s evaluation function has currently 8 heuristic functions. 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, based solely on the number of pieces on the board: opening, 19 to 24 pieces; middle game, 9 to 18 pieces; and endgame 0 to 8 pieces. The value of each heuristic may differ for each phase of the game. 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. Succinctly, these are the Turska’s heuristics:

- Material Balance: weighted difference of the number of pieces each side has.
- Piece Mobility: the degree to which the pieces are free or restricted in their movements.
- Back Rank: how protected the back rank is.
- Balance: how equally the pieces are distributed over the board.
- Bridge Exploit: one man controlling two opposing men forming a bridge.
- River Control: encourage players to control the river.
- Runaway: the ability of a man to advance unimpeded and be promoted to king.
- Triangle: encourage the triangle formation.
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 \textit{Simple}), consisting only of material balance, and a more complex version (named \textit{Complex}), with the heuristics just described.

For shallower searches, \textit{Complex} was clearly superior to \textit{Simple}, while for deeper searches, both versions scored similarly. \textit{Complex} has more knowledge and thus is able to distinguish positions better. On the other hand, \textit{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, \textit{Simple} and \textit{Complex} should score about the same.

4.5 Results Against Other Programs

We evaluated our program by playing some matches against another Classical Checkers computer program, Windamas\textsuperscript{2} (version 7.00). Windamas was created by Jean-Bernard Alemanni in 2001 and is now freeware.

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 to continue searching while the opponent plays has also been disabled.

The search algorithm used in Turska was the best from section 4, i.e. NegaScout with iterative deepening, a transposition table, enhanced transposition cutoffs, and the more complex evaluation function. The programs were configured to use a transposition table with a similar size. Windamas played with the default level, excellency (level 5). In Turska this corresponds to playing with a search depth of 18 ply.

The programs played four matches. Windamas won all the matches. In the first two matches Turska did some bad moves. The games were already lost in the opening phase of the game. In the other two matches it played better. In the third match 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 forth 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.

5 Conclusions and Future Work

We have developed a Classical Checkers computer program, which we named Turska. In addition, we have estimated the complexity of Classical Checkers. Its state-space complexity is $10^{19}$ and its game-tree complexity is $10^{48}$. These results indicate that Classical Checkers is harder than English Checkers, and that it can probably also be solved with current technology.

We have also experimented and analysed several search enhancements. Our results seem to be in conformity with most results available in the literature, except for the case of aspiration search. Our efforts to improve how the information stored in the transposition table was managed were not very successful. Although there were some minor improvements, they were not significant enough to justify the increased memory needs.

\textsuperscript{2}http://alemanni.pagesperso-orange.fr/index.html
The matches against Windamas showed that in some cases Turska is not able to
distinguish between positions very well. Consequently, Turska’s evaluation function
needs further work. Additional future work suggestions include the parallelisation of
the program, in order to augment Turska’s brute force search power, and the creation
of an opening book and endgame databases, which would improve the playing strength
of the program considerably. The latter are crucial to solve the game.

References

Maastricht University, 1994.
2. Dennis M. Breuker. Memory versus Search in Games. PhD thesis, Maastricht University,
1998.
4. Murray S. Campbell and Thomas A. Marsland. A comparison of minimax tree search
6. Shirish Chinchalkar. An upper bound for the number of reachable positions. Interna-
13. Hermann Kaindl, Reza Shams, and Helmut Horacek. Minimax search algorithms with
and without aspiration windows. IEEE Transactions on Pattern Analysis and Machine
Intelligence, 6(4):293–326, 1975.
1993.
dam, 1996.
of Waterloo, 1986.
Müller, Robert M. Lake, Paul Lu, and Steve Sutphen. Checkers is solved. Science,
23. David J. Slate and Lawrence R. Atkin. Chess 4.5 – the northwestern university chess
24. Hendrik J. van den Herik, Jos W. H. M. Uiterwijk, and Jack van Rijswijck. Games
Princeton University Press, 1944.
26. Ernst F. F. Zermelo. Über eine anwendung der mengenlehre auf die theorie des
schachspiels. In Proceedings of the Fifth International Congress of Mathematicians,
pages 501–504, 1913.