ANALYSIS AND IMPLEMENTATION OF THE GAME OF OCTI

Cédric P.E. Roijakkers

Master Thesis CS-05-12

thesis submitted in partial fulfilment
of the requirements for the degree of
Master of Science of Knowledge Engineering
in the Faculty of General Sciences
of the Universiteit Maastricht

Thesis committee:

prof. dr. H.J. van den Herik
dr. ir. J.W.H.M. Uiterwijk
prof. dr. E.O. Postma
dr. H.H.L.M. Donkers
dr. ir. P.H.M. Spronck

Universiteit Maastricht
Institute for Knowledge and Agent Technology
Department of Computer Science
Maastricht, The Netherlands
September 2005

Preface

This report is my master thesis, which describes research performed at the IKAT (Institute for Knowledge and Agent Technology) of the Universiteit Maastricht. The subject of the research is the implementation of AI techniques for the fairly new game of OCTI, a two-person zero-sum board game with perfect information.

I wish to thank several people for helping me to write this thesis and to perform the actual research. First of all, my supervisors: prof. dr. H.J. van den Herik for reading my thesis and commenting on its readability, and my daily advisor dr. ir. J.W.H.M. Uiterwijk, without whom this thesis would never have included so many interesting details. Both also helped arouse my interest for computer game playing in their course "Search techniques" (Zoektechnieken), in the fourth year of my study. The other committee members are also recognised for their time and effort in reading this thesis. Moreover, I would like to express my gratitude to Jan Willemson (University of Tartu, Estonia) who was so kind to read my thesis and comment on the contents. Many of his remarks led to clear improvements of the text. Furthermore I wish to thank all my fellow students and my family for their never ending support.

Cédric Roijakkers
Voeren, September 2005

Abstract

Board games and computer game-playing have been a subject for research in computer science and artificial intelligence for quite some years now. Beating humans at their own games has been a dream for many computer scientists since the early 1950s. A dream which eventually came true in 1994, when CHINOOK became the first man-machine world champion in Checkers.

This thesis describes the analysis and implementation of a fairly new two-player zero-sum board game called OCTI, and more precisely the variant "OCTI: New Edition". To begin with, both the state-space complexity and game-tree complexity of the game are computed. Using these values, suitable techniques to make a computer play the game adequately are selected, using both search algorithms and limited game knowledge. These algorithms are later refined with heuristics, forward pruning and transposition tables. Finally, some tests are run to select the best algorithm and fine-tune the knowledge-based evaluation function.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A game is a series of interesting choices. — Sid Meier

This chapter is a general introduction to games, Artificial Intelligence and computer game-playing. In section 1.1 we will first define the domain of computers and games, followed by an introduction to computer game-playing in section 1.2. In section 1.3 we will define the problem statement and research questions and finally in section 1.4 an outline of this thesis is given.

1.1 Games and computers

Games have been around as long as humanity has. Some games are meant purely for entertainment, while other games challenge the intellect of humans. Board games usually offer a pure form of abstraction, allowing two (or more) players to compete against each other without having to worry about tedious logistics.

Games come in all sorts of shapes and sizes, but some have some properties in common with others, which allows us to categorise them. For example we can distinguish them by the number of players, the kind of information they provide to the players, whether they are convergent or divergent, and so forth.

The idea to have computers play games as intelligently as possible has been launched as early as the 1950s by Shannon (1950) and Turing (1953). Both described a chess-playing program which, ultimately, led to Deep Blue defeating the reigning World Champion (Schaeffer and Plaat, 1997). In over half a century, many advances have been made, and computers are getting better and better in lots of games, but humans are still in control of several games, like Backgammon and Go. There are four reasons that make games interesting research subjects.

1. The knowledge in games is mostly exact and well-defined, which makes

them easier to translate to computer programs, compared to real-life problems (van den Herik, 1983).

2. Although the rules of most intelligent games are easy to learn, playing the game is quite hard (Minsky, 1968).

3. Games can be used to test new ideas in problem solving, which can later be used in other fields of study like mathematics and economics (Nilsson, 1971).

4. Creating a machine that plays an intelligent game can sometimes give a new insight into the way people reason. Famous representatives of this statement are de Groot (1946), and Newell and Simon (1972).

Beating humans at their own game is the summum for AI researchers specialising in games. But building a World Champion computer program can sometimes be a long and difficult process, often going one step at a time. In the beginning, each advance can be only applicable to the very narrow field of the game itself, but sometimes it can be translated to other domains. Research in games is therefore not only important to the game-playing community, but sometimes also to the rest of the AI world.

1.2 Computer game-playing

Abstract games are played in two different ways by computers: using knowledge or using brute-force search. The choice of method depends on the game, and its complexity.

If a purely knowledge-based technique is chosen, the game is played using knowledge only, and a search algorithm is not used. This is only possible if sufficient information of the initial state and all the subsequent states for playing the game is available. It should be possible to store the game knowledge, and of course storing and discovering new game knowledge has to be feasible too, an example of a game which can be played (and even solved) using knowledge only is the game of Nim (Bouton, 1901).

If a purely search-based technique is chosen, the game tree of the game is constructed and explored, and a search algorithm is used to select the best move. In a game tree, a node represents a position on the board and an edge represents a move. A leaf represents a terminal position: either win, loss or draw. If possible in time and space restrictions, the entire game tree of the game is constructed and saved. This way, the exact outcome of any move is known and the game is considered solved. If this is not possible, a search tree is generated. This tree is only a part of the game tree, with the root being the current position to evaluate. In a search tree, the leaf nodes also include non-terminal positions. An evaluation function is used to estimate the value of a certain position and this evaluation is used in conjunction with the MINIMAX algorithm (von Neumann and Morgenstern, 1944) to search the tree.

1.3 Problem statement and research questions

In computer game-playing, the goal is to make a computer play a certain game as good as possible. So the problem statement for this thesis is the following:

> *How can a computer program be written that plays the game of OCTI as efficiently as possible?*

In order to answer this problem statement, several other questions arise. We will deal with three research questions.

1. *What is the complexity of OCTI?*

To answer this research question, the complexity of OCTI needs to be computed.

2. *What known techniques can be used to play OCTI?*

Once the complexity of the game is known, techniques to play the game will be selected. If the game is not very complex, this will usually be a full enumeration. More complex games will traditionally use a search algorithm with some limited game-specific knowledge. Very complex games normally are played using knowledge-rich methods. When the techniques used to play the game are known, the next step is the acquisition of knowledge.

3. *What game-specific knowledge is required to make a computer play OCTI?*

If we exclude complete enumeration, then clearly all games have to be played with some form of game-specific knowledge. Depending on the chosen approach (search-based or knowledge-based), this amount of knowledge can vary.

Once these three questions are answered, a program to play OCTI can be built.

1.4 Outline of this thesis

The outline of the thesis is as follows:

- Chapter 1 contains a general introduction to games and computers, explains how computers play games, and defines the problem statement and research questions.
- Chapter 2 gives an introduction to the abstract board game OCTI and its strategies.

- Chapter 3 discusses a complete complexity analysis of OCTI, including state-space complexity, game-tree complexity and a comparison of OCTI and other abstract board games.
- Chapter 4 states the techniques used to make the program play OCTI: tree construction, search algorithms, move ordering, windowing techniques and transposition tables.
- Chapter 5 states the general principles of OCTI, revisits the strategies and explains the implementation of the evaluation function for the game.
- Chapter 6 lists tests and their results of the various search algorithms.
- Chapter 7 lists tests and their results of the evaluation function tweaking.
- Chapter 8 concludes the thesis with an evaluation of the problem statement, answers to the research questions, final conclusions and recommendations for future research.

Chapter 2

The game of OCTI

Good positions don't win games, good moves do. — Gerald Abrahams

This chapter is an introduction to the main research subject: the game of OCTI. First, a general introduction to OCTI and its variants is given in section 2.1, followed by the rules of "OCTI: New Edition" in section 2.2. In section 2.3 the OCTI notation is defined by means of an example game, and finally in section 2.4 some strategies for OCTI are given.

2.1 Introduction to OCTI

OCTI is the name of a series of board games developed by Donald Green. There are currently two versions: "OCTI: New Edition" (formerly named "OCTI for Kids") which is played on a 6×7 board, and "OCTI-Extreme" (also known as "OCTI-X") which is played on a 9×9 board (Green, 2000b; Sutton, 2002; Gendelman and Meshulam, 2004). The basic idea in both versions of the game is to capture one or all of the opponent's home bases (called "OCTI squares") or capture all of his pieces (called "pods"). In order to achieve this, a player must first equip his pods with small pegs called "prongs" which will enable the pod to move in the chosen direction. This way, a pod can be equipped with up to 8 prongs which will allow it to move in every direction on the board.

2.2 Rules of "OCTI: New Edition"

As seen in the previous section, several versions of the game of OCTI exist. This thesis deals with the 6×7-variant, called "OCTI: New Edition". For the sake of clarity, throughout the text we will be using the term "OCTI" whenever "OCTI: New Edition" is meant.

The game starts with 4 pods per player (4 yellow and 4 red). The pods are

placed on the starting squares, called OCTI squares. Furthermore, each player has a supply of 12 prongs which can be inserted in any direction on any of his own pods. Figure 2.1 shows the starting position.

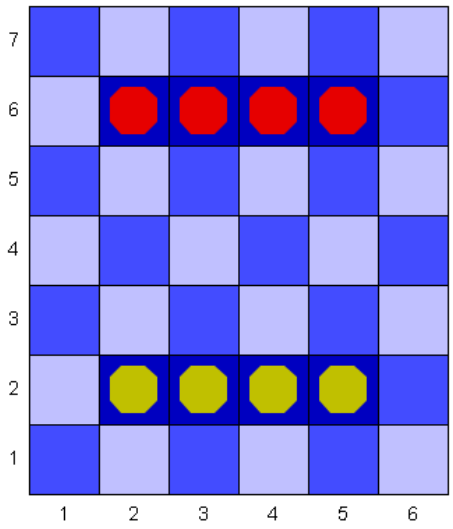

Figure 2.1: Starting position in OCTI.

The player with the yellow pods must make the first move. A move can be any of three things: (1) adding a prong to a pod, (2) moving a pod to an adjacent square, or (3) jumping over a pod in an adjacent square. In the starting position, moving and jumping is not yet allowed since a pod needs a prong to be able to move in the direction of the prong. In OCTI, there is a fundamental difference between jumping and capturing. In this thesis, we will define "jumping" as the process of jumping over a pod without capturing it, and "capturing" as both jumping the pod and removing it from the board.

When a player chooses to add a prong to one of his pods, the turn ends and the next player must make a move. A pod can hold 8 prongs, named A through H. Prong A faces upwards from player yellow's point of view. The other prongs are named clockwise following the alphabet. At first, each player starts with 12 prongs, but prongs can be stolen from the opponent by capturing the opponent's pods. Once a prong is placed on a pod it cannot be removed, and pods cannot be rotated during the game.

Moving a pod to an adjacent square is only possible when the pod has a prong in that direction. The target square must also be empty. In "OCTI-Extreme" it is also possible to stack pods of the same player on top of each other, but this is not allowed in "OCTI: New Edition". Once a player has moved one of his pods, the turn ends and the next player is to move.

Jumping is the most complicated move possible in the game. Jumping pods is only possible when three requirements are met: the pod must have a prong in the direction of the jump, the square that is being jumped must hold a pod (either friendly or enemy) and the square where the pod will land must be

empty. Furthermore, multiple jumping is allowed. When after a jump the pod lands on a square where another jump is possible, the player can jump again. Jumping, however, is never mandatory. When a pod is being jumped over it can be captured, but this is not required either. This way, jumping can be used to drastically improve the mobility of pods, or to capture both friendly and enemy pods. When a pod is captured, its prongs are removed and given to the player which has captured the pod. The player can then use these prongs to add to his own pods later in the game. Captured pods are removed from the board and can never be added again. Jumping over the same pod in one series of jumps is not allowed however, which brings the maximum number of jumps in one series to 7.

The game ends when a player steps with one of his pods on one of the OCTI squares of the opponents. This can be either via a jump move, or via a normal move. Stepping on one of the OCTI squares immediately ends the game. Another option to end the game is to make it impossible for your opponent to move, either by capturing all of his pods, leaving him with only empty pods and no more prongs to add or block all remaining pods so that they cannot move or jump again. This will also end the game immediately. The player to step on an enemy OCTI square or to block the opponent is declared the winner.

Even though no documentation can be found in the literature about draws in OCTI, they are possible. When both players position their pods in such a fashion that they can not add any more prongs, nor move their pods on the board the position could be declared a draw. The same could be said about endless repetitions of both player making the same sequence of moves over and over again. However, in the official rules on the OCTI website, nor in previous reports of tournaments, draws are not mentioned.

The rules to all the OCTI variants can be found in Green (2000b).

2.3 OCTI notation and example game

The squares in OCTI are numbered 1 through 6 horizontally and 1 through 7 vertically, all seen from player yellow's point of view. Since a player can make three kinds of moves in the game, the notation also has three different forms.

When a player adds a prong to a pod, first the location of the pod is given, followed by the letters of the prongs it already contains, a plus sign and finally the letter of the prong being added. Adding prong A to an empty pod in square 2,2 is written as follows: `22+a`. If this pod already had prongs A, C and F and prong B is being added, the line reads: `22acf+b`.

When a pod is being moved, first the coordinates of the starting position are given, followed by the prongs equipped on the pod. Next a dash is written and finally the coordinates of the target position are written. Moving a pod from 2,2 to 2,3 is written as follows: `22a-23`. If the pod has multiple prongs, the notation might be: `22abh-23`.

When a pod makes a jump move, first the starting coordinates of the jump are written along with the equipped prongs on the pod. A dash follows with the coordinates of the square the pod is landing on. If the player chooses to capture the pod jumped, an x is inserted. Should the player choose to make another jump, another dash is added along with the coordinates of the new target square, and optionally an x when the pod jumped is captured. This sequence is repeated as often as necessary. A pod moving from 2,3 via 4,1 and 4,3 to 6,1 while capturing the last pod jumped is written as follows: `23ad-41-43-61x`. Capturing all pods along the way would be written as: `23ad-41x-43x-61x`.

A game of OCTI can be saved into an SGF file (Saved Game File) using special codes and headers. This game can be read and written by the program to allow games to be saved. The header file for "OCTI: New Edition" is as follows:

```
(
;GM[19] RU[kids]
```

Some extra codes can be added, e.g., the time and date of the game and the names of both players. But the only required codes are `GM[19]` to identify the game of OCTI and `RU[kids]` to identify the "OCTI: New Edition" rule set.

The next lines in the file are all move lines, written as a semicolon and a capital B or W to identify the player (in OCTI, B stands for the yellow player and W for the red player). Followed by the move in OCTI notation enclosed by square brackets. A final closing bracket ends the file. The following file shows a short example:

```
(
;GM[19] RU[kids]

;B[32+a]
;W[36+e]
;B[32a-33]
;W[36e-35]
;B[33a-34]
;W[35e-33x]
;B[42+g]
;W[33e-32]
)
```

This corresponds with the game shown in figure 2.2.

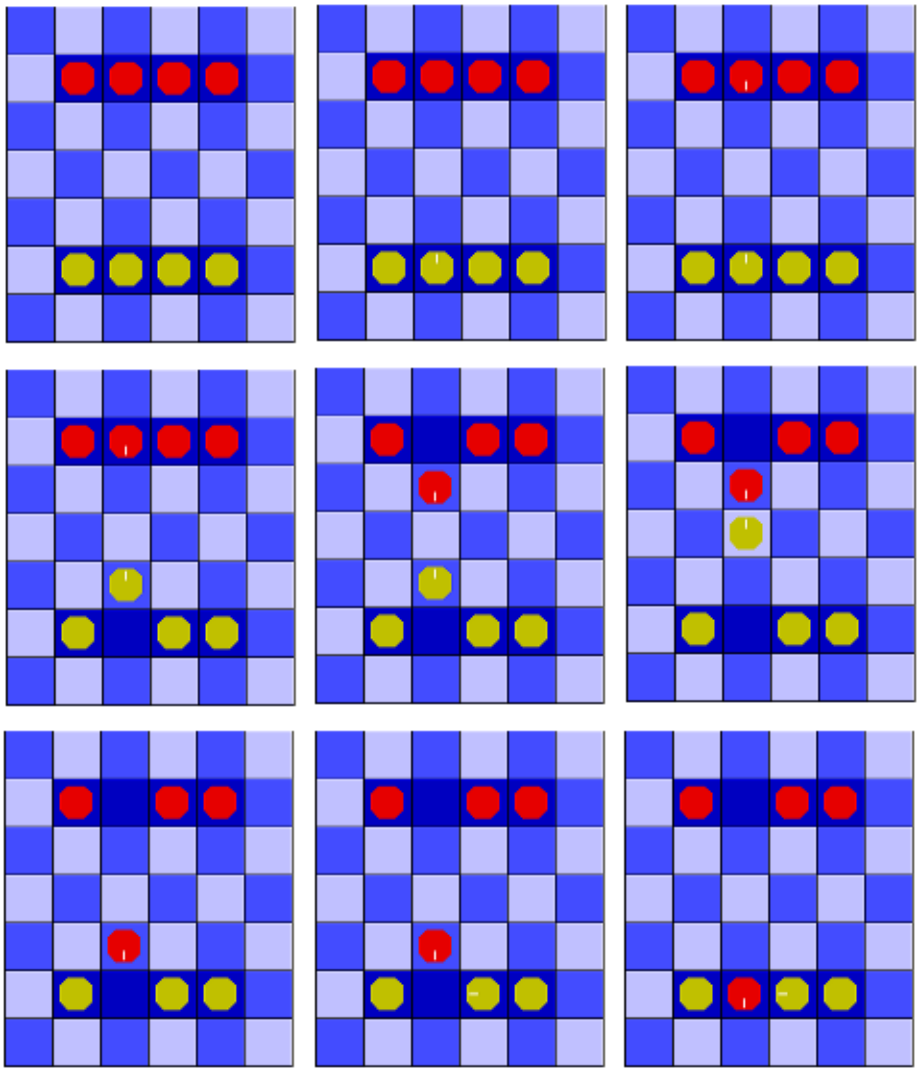

Figure 2.2: OCTI example game.

2.4 Strategies in OCTI

To play a game of OCTI, a player has to keep four things in mind while deciding which move to play. These are the basic strategies of OCTI, taken from Green (2000a):

1. *Trade-off between building and moving*

This is a thing a player has to consider continuously: choose to either add a prong to a pod or move a pod to a new location. When a player chooses to

move a pod, this means he[1] loses a turn to add a prong. So it is better only to move a pod when it really offers an advantage. Wandering around the board makes the opponent's pieces more developed while yours remain the same.

2. Move your pods in teams

It is better to keep two pods close to each other. When pods are moving in formation, they can cover more ground together. Especially due to the jumping rule, and the fact that you are not required to capture jumped pods, two pods can be a serious threat to the opponent, even if he is still a few squares away.

3. Do not let the opponent capture pods and/or prongs

This rule is quite straightforward. Every player has only 4 pods and 12 prongs. Amongst skilful players, a game can take quite some time before finishing and the supply of prongs might run out. Therefore, try to keep your prongs in your possession, in some cases this could even mean capturing your own pods to not let them fall in enemy hands.

4. Make pods move forward and backward

In the short version of the game, where a player is trying to capture only one OCTI square, it is very important to make your pods be able to move forward and backward. When a player launches an attack on an enemy OCTI square, he can capture the pod guarding the base, and since his pod has a prong equipped in the opposing direction, step back onto the OCTI square the next turn, effectively winning the game.

[1] Throughout this thesis, we will be using "he" or "his" whenever "he or her" or "his or her" is meant.

Chapter 3

Complexity analysis of OCTI

Games are a compromise between intimacy and keeping intimacy away. — Eric Berne

A good step to get to know a game, and to learn the best way to implement it is by starting with a complete complexity analysis. This analysis consists of two different values: the state-space complexity and the game-tree complexity. In section 3.1 we will first compute the state-space complexity, followed by the game-tree complexity in section 3.2. Finally, in section 3.3 we will compare OCTI's complexity to the complexity of other games.

3.1 State-space complexity

The state-space complexity is the total number of different board configurations possible in a game. Exactly calculating this number is often very difficult; therefore upper bounds are most commonly used, including some illegal positions that can never be reached. For OCTI, we assume that the number of illegal positions is much smaller than the number of legal positions. The actual count of legal positions will probably be fairly comparable to this upper bound including illegal positions. Therefore, we will compute the state-space complexity including these illegal positions.

In OCTI there are 6×7, thus in total 42 squares which all pods can occupy. The maximum number of pods on the board is 8 (4 pods per player), and the minimal number is 1 (1 pod, all others are captured). A game can never have more than 8 pods, which is the starting position and no pods can be added to the game; or less than 1 pod which is a terminal position. Let R be the number of red pods on the board, and Y the number of yellow pods on the board. So the total number of positions with 2 or more pods is derived in the following formula:

$$\sum_{R=1}^{4}\sum_{Y=1}^{4}\binom{42}{R}\binom{42-R}{Y}$$

The amount of possible pod positions can be found in table 3.1.

pods	possibilities R/Y	pod positions
2	1	1,722
3	2	68,880
4	3	1,567,020
5	4	25,520,040
6	3	325,238,732
7	2	3,399,269,328
8	1	29,979,666,990
Total:		**33,731,332,712**

Table 3.1: Number of possible pod positions.

The terminal positions, with only 1 pod on the board are not included in the count. However, this number is very simple to compute: there are 42 squares which all can be occupied by one red or one yellow pod, which add another 84 possibilities to the count. This brings the total number of pod positions to 33,731,332,796.

This only includes the total number of ways to place the pods on the board. However, pods can be loaded with up to 8 prongs per pod, with a maximum of 24 prongs, 12 for each player. Players can also steal prongs from the opponent when capturing a pod, giving its prongs to the capturing player.

There are 8 slots on each pod to insert prongs, thus 256 different prong combinations per pod. However, the total number of prongs on the board is limited to 24, which needs to be taken into account when computing the total number of prong configurations. In the following formula, N is the total number of pods (red and yellow) on the board and I the total number of prongs on all pods. The factor $max\,(0, 24 - I + 1)$ makes sure that illegal positions are not added to the count.

$$\sum_{I=0}^{8N}\binom{8N}{I} \times max\,(0, 24 - I + 1)$$

The total amount of possible positions can be found in table 3.2.

As said before, this count includes some unreachable positions. Examples are pods that have moved in a direction for which they have no prong, or more than one pod standing on an enemy base. However, for the game of OCTI we assume that the actual number of legal positions will not be much lower than this number, so we define the state-space complexity of OCTI as $O(10^{28})$.

pods	pod positions	prong positions	total positions
1	84	256	21,504
2	1,722	1,114,112	1,918,500,864
3	68,880	218,103,808	$1.50229903 \times 10^{13}$
4	1,567,020	38,656,145,007	$6.05749523 \times 10^{16}$
5	25,520,040	$5.57744796 \times 10^{12}$	$1.42336695 \times 10^{20}$
6	325,238,732	$5.43832534 \times 10^{14}$	$1.76875404 \times 10^{23}$
7	3,399,269,328	$3.18846469 \times 10^{16}$	$1.08384502 \times 10^{26}$
8	29,979,666,990	$1.14296287 \times 10^{18}$	$3.42656462 \times 10^{28}$
Total:	**33,731,332,796**	$\mathbf{1.17539697 \times 10^{18}}$	$\mathbf{3.96477064 \times 10^{28}}$

Table 3.2: Total number of possible positions.

3.2 Game-tree complexity

The game-tree complexity is a number indicating the amount of different games that can be played. To calculate the size of the game tree, two values are needed: the average branching factor, and the average game length. Both values here are based on game play on the Internet, found at `http://www.octi-online.com/`.

Sutton (2002) has analysed games of OCTI played over the Internet, and has concluded that the average branching factor of OCTI is 31. When the game starts, each player has 32 possible moves (4 pods with each 8 prongs to add). The branching factor does not vary much in the beginning of the game. In the middle game, due to the jumping rule, the branching factor can be as high as 101. In the endgame, the branching factor is lower, sometimes even as low as 2. To compute the game-tree complexity, we will use the value of 31 obtained by Sutton (2002).

The length of a 6×7 OCTI game is rather short, due to the relative small size of the board and the fact that OCTI is more of a tactical game. Game lengths rarely surpass 25 moves. This would bring the total game tree size to 31^{25}, or 1.92×10^{37}.

3.3 OCTI compared to other games

As we have seen in sections 3.1 and 3.2, the state-space complexity of OCTI is $O(10^{28})$ and the game-tree complexity is $O(10^{37})$. In other words, OCTI has a relatively small state-space, but a relatively large game-tree. This makes OCTI best approached by brute-force methods and some knowledge of the game, e.g., using intelligent search. OCTI's complexity compared to other games' complexity can be found in table 3.3, based on van den Herik, Uiterwijk, and van Rijswijck (2002).

Based on this list of complexities, OCTI can be compared to the game of Checkers. Checkers, as opposed to OCTI, is a rather old game and has been the subject of AI research for quite some time. This resulted in the creation of Chinook,

Game	State-space	Game-tree
Nine Men's Morris	10^{10}	10^{50}
Pentominoes	10^{12}	10^{18}
Awari	10^{12}	10^{32}
Kalah (6,4)	10^{13}	10^{18}
Connect-Four	10^{14}	10^{21}
Domineering (8×8)	10^{15}	10^{27}
Dakon-6	10^{15}	10^{33}
Checkers	10^{21}	10^{31}
OCTI (6×7)	$\mathbf{10^{28}}$	$\mathbf{10^{37}}$
Othello	10^{28}	10^{58}
Qubic	10^{30}	10^{34}
Draughts	10^{30}	10^{54}
Chess	10^{46}	10^{123}
Chinese Chess	10^{48}	10^{150}
Hex (11×11)	10^{57}	10^{98}
Shogi	10^{71}	10^{226}
Renju (15×15)	10^{105}	10^{70}
Go-Moku (15×15)	10^{105}	10^{70}
Go (19×19)	10^{172}	10^{360}

Table 3.3: Complexity of OCTI compared to other games.

which eventually became the first Man-Machine World Champion in any game in 1994 (Schaeffer, 1997). Since OCTI is only a few orders of magnitude more complex than Checkers, it can be assumed that, with the necessary research, OCTI can one day beat a human opponent, too.

If we compare OCTI to Checkers once more, it can be assumed that, with the necessary time and research, strongly solving OCTI is possible. Quite recently, a breakthrough has been made in Checkers on solving a common and well-known opening called the "White Doctor" (Schaeffer *et al.*, 2005a; Schaeffer, 2005b). It is expected that, if sufficient computing resources are available, Checkers will be completely solved in a few months.

Chapter 4

Search

Without error there can be no brilliancy. — Emanuel Lasker

The concept of game trees has been previously defined in section 1.2. Later, in section 3.3 we have stated that the complexity of OCTI does not permit a full enumeration of the whole game. This means that when the game tree for OCTI will be constructed, we will only be able to search through it to a given depth. Therefore, search methods will have to be developed which will guide the search as efficiently and consistently as possible. The goal is to look as deeply as possible for a certain position and time.

In this chapter, we will first discuss the methods used to construct the search tree in section 4.1. Next, in section 4.2 we will introduce algorithms for searching through a tree. We will discuss the $\alpha\beta$-algorithm and its extensions used in our program in sections 4.3, 4.4, 4.5 and 4.6. Furthermore, in section 4.7 we will also list some other methods which were not implemented, and discuss the overall framework of our program in section 4.8.

4.1 Tree construction

Constructing the search tree is done by means of move generation in games. At a given position, all the moves are generated that will lead to another legal position. Also, we will have to check if a given node is a terminal position. Both matters will be discussed in subsections 4.1.1 and 4.1.2.

4.1.1 Move generation

Generating legal moves in OCTI is a quite straightforward process. As stated in section 2.1, there are three types of moves in the game of OCTI: adding a prong, moving a pod and jumping and/or capturing pods. Depending on the given position, not all three types of moves are always possible. For example,

when no pod on the board has a prong equipped, jumping and moving will not be possible.

The move generator automatically detects the type of position and invokes the subroutines that generate the actual moves. Thus, when no more prongs can be added, the prong-move generator will not be called. When jump-moves are involved, the move generator does a quick depth-first search to generate all possible jumps, with and without captures. Since these moves rarely surpass 3-4 jumps per move, doing this extra search will not slow the move generator down significantly.

4.1.2 Detection of terminal nodes

Detecting terminal nodes is once again quite straightforward in OCTI. As stated in section 2.1, the game is over when a pod steps on an opponent's OCTI square, or when the opponent cannot move any more. First, the algorithm checks for enemy pods on a home base, and next a quick scan of the board is done to evaluate if all pods are blocked for moving. If any of these scans is successful, the game is deemed over and the player that made the previous move is declared the winner.

4.2 Searching

Searching through game trees is usually done using *depth-first search*. In this type of search, the first branch to an immediate successor of the current node is recursively expanded until a leaf node is reached. The algorithm then backtracks and checks the remaining branches.

Depth-first searching usually works like a charm for games, but there is a caveat. A depth limit has to be provided before the search starts. Since we do not know how many seconds the search is going to take, a technique called *iterative deepening* is used (Russell and Norvig, 1995). With ID, the search starts to a given ply depth x, finishes, and then starts over the search to depth $x+1$. This process is then repeated until the allowed search time runs out. This might seem like a lot of wasted time on re-searching nodes, but the overhead is surprisingly small.

Because of ID, the search can be stopped at any time, and a valid best move at that time can be returned. The longer the search has been running, the deeper the algorithm has searched.

4.3 Alpha Beta

While the algorithm is searching the game tree, the value of the best terminal node found so far might change. Certain moves in the tree have such a value that they cannot affect the expected value of the tree any more. The $\alpha\beta$-algorithm takes advantage of this situation and prunes the search tree by not searching useless branches. The algorithm is credited to John McCarty in 1956, although he did not publish it. Brudno (1963) was the first to write it down. Its correctness was later proved by Knuth and Moore (1975).

We will use $\alpha\beta$ for searching in OCTI, since it is still the most well-known and widely-used search algorithm in games. Most chess-playing programs use it. The algorithm uses lower (α) and upper (β) bounds to guide its search through the tree. These bounds are used to cut off certain branches of the tree. For a more detailed explanation of the algorithm see Russell and Norvig (1995).

4.4 Windowing techniques

A way to optimise the $\alpha\beta$-search is to fine-tune the $\alpha\beta$-window. As we know, $\alpha\beta$ uses a window to keep track of the lowest and the highest value encountered. Branches with a value outside the window are not searched and are cut off. Usually it is faster to search with a smaller window since this yields the most cut-offs. However, sometimes this also cuts off a good branch and re-searching is necessary. Changing the $\alpha\beta$-window from the default $[-\infty, \infty]$ to a smaller window is called a windowing technique. In the following subsections we will discuss three windowing techniques: aspiration search in subsection 4.4.1, principal-variation search in subsection 4.4.2 and null-move search in subsection 4.4.3.

4.4.1 Aspiration search

Aspiration search is not actually a new form of search, but merely a different way of starting each $\alpha\beta$ iteration. Aspiration search uses aspiration windows: when an iteration of $\alpha\beta$ is finished, the returned minimax value m is used to calculate the new α and β values. Quite often, a static factor f is used, such that $\alpha = m - f$ and $\beta = m + f$. This smaller window makes a quicker search possible. However, if the minimax value is outside the window, a re-search with the default $[-\infty, \infty]$ window is necessary. More information about the actual algorithm can be found in Shams, Kaindl, and Horacek (1991).

4.4.2 Principal-variation search

With principal-variation search (PVS), the window is kept as small as possible, with its lower bound set to α, and its upper bound to $\alpha + 1$. The basic idea

behind this method is that it is cheaper to prove a subtree inferior, than to determine its exact value. It has been shown that this method does well for bushy trees like in chess. In section 3.2 we have seen that the branching factor of OCTI (31) is in the same range as chess (35), therefore it might be a good idea to try PVS in OCTI too. Provided we have a good move-ordering mechanism, PVS reduces the size of the search tree. For a more detailed description of the algorithm, see Marsland (1986).

4.4.3 Null-move search

Making the null move means changing who is about to play, without moving any piece on the board. It is equal to passing, which is not allowed in OCTI. The reason to use the null move is to enable us to establish threats which is important for move ordering.

The null move was first described by Beal (1989) and later extended by Donninger (1993). The idea is to perform a depth-limited search with the null move before doing the actual search. The search depth of the null move search is normally the search depth of the normal search, but reduced by a factor R, which is usually set to 2. The null move is not always searched however. In the following situations no null move is used.

- The previous move was a null move.
- The search is at the root of the tree.
- The current player is at a considerable disadvantage, the chances of a cut-off being extremely low.

We have assumed that making any move is always better than passing. In OCTI however, some situations can occur (though be it rarely) that passing would be a better choice. But since passing is not allowed, using the null move in this situation could return an illegal move. This situation is called *zugzwang* (Uiterwijk and van den Herik, 2000). To avoid this problem, the null move is also not used when the side to move can not add a prong any more. Adding a prong always fortifies a pod, and therefore eliminates zugzwang.

4.5 Move ordering

Since the $\alpha\beta$-algorithm is a depth-first search algorithm, ordering the search tree such that the branches which will yield the best score are evaluated first will greatly speed up the searching. However, if we knew which branches would result in the best score at any time, searching would no longer be necessary. To order the search tree as good as possible, we have to create an ordering function which tries plausible moves first which yields big cut-offs. In order to select plausible moves, heuristics are used.

There are many different types of heuristics. For example game-specific versus game-independent heuristics, or static versus dynamic heuristics. In this thesis, we will focus on game-specific and game-independent heuristics. Game-specific heuristics use game characteristics to order the moves. Game-independent heuristics are based on other properties of search algorithms.

In this section we will describe two game-independent heuristics: the killer-move heuristic in subsection 4.5.1 and the history heuristic in subsection 4.5.2. In subsection 4.5.3 we will briefly discuss the game-specific heuristics.

4.5.1 Killer-move heuristic

The killer move heuristic is a fairly simple heuristic based on the fact that a move which was best for a certain position will probably also be pretty good for a similar position. It was first proposed by Huberman (1968). Since $\alpha\beta$ is a depth-first search algorithm, implementing it is pretty straightforward. When the algorithm has found a certain move to be the best move for a certain position, this move is saved to be tested first in the next position (at the same search depth). To begin with, this "killer move" is checked for validity in the current position. If the move is possible, it is added first to the move list. If it is not a valid move, move generation proceeds as usual. The cost to implement killer moves is quite cheap. For k moves at n plies, its memory requirement is only $k \times n$, no problem for any modern computer. For more information about killer moves, see Akl and Newborn (1977). An example of a killer-move list is given in table 4.1.

depth	**killer**
1	
2	42d+b
3	52+a
4	42bd-64x
5	64bd+a
6	64abd-65

Table 4.1: Killer-move list example.

4.5.2 History heuristic

In a certain way, the history heuristic is pretty similar to the killer-move heuristic. It was first defined by Schaeffer (1983). The history heuristic saves a history table for each move in the game. In OCTI, two tables are required per player: one for moving pods over the board, and one for adding prongs to pods.

The first history table records the moves that pods can make on the board. This table consists of $42 \times 42 = 1764$ entries. These represent 42 *from-squares* and 42 *to-squares*. Quite obviously this table includes some impossible moves, e.g., from one edge of the board to another, but this does not cause a problem

for the later usage so it is ignored. Each time a move that involves moving a pod from one square to another is found to be the best move for a certain position, its counter in the history table is incremented. When later a new node is searched, the possible moves are ordered by descending order of their scores. The second history table works exactly like the first one, except this one stores moves that involve adding prongs. This table has a size of $42 \times 8 = 336$. When at a certain position in the search tree a move is found to be the best, the algorithm automatically selects the corresponding table and increments the score of that move. When a list of moves to be ordered is presented, each table is used when necessary.

History tables are kept throughout the entire game. They are initialized once at the beginning of the game and only reset when the game is finished and a new game is started. Once a player has made a move, each score in the table is divided by two when a new search process begins. Once again, the memory requirement of this heuristic is fairly small. The total number of scores kept is $(1764 + 336) \times 2 = 4200$, with the average size of a move in memory of 6 bytes, this is less than 25 kilobytes.

4.5.3 Game-specific ordering

As we have stated in section 2.4, there are several general strategies which have to be taken into consideration when playing a game of OCTI. Two of these can be used in move ordering: (1) *trade-off between building and moving* and (2) *do not let the opponent capture pods and/or prongs*.

Hurting your opponent is usually smarter than fortifying your own pieces in a war-based game like OCTI. Capturing all the opponent's pieces will result in an automatic win. Therefore, moves that allow jumping/capturing of enemy pods are placed first in the list of possible moves. Next are the moves which add a prong to one of your own pods. Building your pieces is a better idea than moving a piece. The remaining moves are then added at the end of the list.

Move ordering, however, is not the only place where game strategies are implemented. The evaluation function, discussed in chapter 5, is the foremost place for this type of game logic. Move ordering should be aware of the game strategies, but a good evaluation function should also take them into account.

4.6 Transposition table

While searching a game tree, it can sometimes happen that the exact same position is searched more than once. For example, the same position is reached via a different route in the tree. These positions are called *transpositions*. If the results of a search on a position can be stored somewhere, re-searching the same position later can be avoided or drastically shortened by using those stored results.

Transpositions can happen all the time in OCTI, even though they are rare in the beginning of the game. Since in OCTI the pieces on the board can change due to the addition of prongs, the same position will return less often. But since we are using iterative deepening, using the transposition table will almost certainly have a positive effect during the entire game.

In the following subsections we will be discussing the details of our implementation of the transposition table mechanism. In subsection 4.6.1 we will first describe the hashing algorithm, followed by the use of the table in subsection 4.6.2, and lastly the probability of errors in subsection 4.6.3.

4.6.1 Hashing

In an ideal world, saving all the positions encountered in the search tree and their results in memory would be the best option. However, this is not possible due to memory restriction in our current computers. A transposition table is therefore implemented as a hash table using some hashing method.

In OCTI there are two different pieces (red and yellow pods), which each can hold 256 different combinations of prongs. Furthermore, there are 42 squares on the board which can all be occupied. For any possible combination a random number is generated: a grand total of 21,504 ($2 \times 256 \times 42$) random numbers. The hash-value of a position is computed by doing an XOR operation on the numbers associated with the piece-square combination of that position. In addition, if the current player to move is red (the second player) an extra random number is added to the hash. This method is called Zobrist-hashing (Zobrist, 1970).

If the transposition table consists of 2^n entries, the first n bits of the hash value of a certain position are used as an index value for the table element. The remaining bits (the hash key) are used to distinguish among different positions mapping on the same hash index. For OCTI we use a 64-bit hash value (computed with 64-bit random values) and a transposition table which can hold 2^{20} or 1,048,576 entries.

4.6.2 Use of the table

Each element in the transposition table has following components:

hash The hash value of the current position;

move The best move to make in the current position;

score The $\alpha\beta$ value of this position;

flag Defines the type of score: exact, upper bound or lower bound;

depth The depth at which this position was searched.

The complete hash is stored in the transposition table to check for so-called type-2 errors (Breuker, 1998). More on these errors is given in subsection 4.6.3. This is done because sometimes the index value is the same, even though the hash value is different. There is also a possibility that the retrieved move is not valid for the current position. To avoid playing an illegal move or crashing the entire program, the retrieved move is first checked for validity. The transposition table can be used in three ways:

1. The depth to be searched is less than or equal to the depth stored in the transposition table and the score of the position is exact. The position does not have to be searched further and the stored score is returned to the algorithm. This is the main reason transposition tables are used.

2. The depth to be searched is less than or equal to the depth stored in the transposition table and the score of the position is not exact. The score of the position can be used to narrow the $\alpha\beta$-window, the α value is adjusted if the score is a lower bound or the β value is adjusted if the score is an upper bound.

3. The depth to be searched is greater than the depth stored in the transposition table. The move stored in the table is used first while searching the possible moves. This type of move ordering can sometimes lead to faster searches.

For a more detailed description of the algorithm, see Marsland (1986) and Breuker (1998).

4.6.3 Probability of errors

Using a transposition table as a hash table introduces two types of errors. The first type is a type-1 error. A type-1 error occurs when two different positions have the same hash value. This mistake will not be recognised and can lead to wrong evaluations in the search tree. This type of error is absolutely not wanted.

Let N be the number of distinguishable positions, and M be the number of different positions to be stored. The probability that this error will not happen is given by the following equation, taken from Breuker (1998):

$$P\,(no\ errors) \approx e^{-\frac{M^2}{2N}}.$$

As an example we consider our program, which searches 30,000 nodes per second. If it plays OCTI using a total of three minutes of thinking time, the number of nodes searched is 5.4×10^6. Breuker (1998) states that for 30% of the nodes an attempt is made to store them in the transposition table. For now, we will assume this percentage is also valid for OCTI. In this example, this is 1.62×10^6

nodes. If the hash value consists of 64 bits, the probability of at least one type-1 error is:

$$1 - e^{-\frac{(1.62 \times 10^6)^2}{2 \times 2^{64}}} \approx 7.11 \times 10^{-8}.$$

Because the transposition table is small, we cannot store all the positions occurred in the search. It happens that a position is to be stored in an entry, which is already occupied by another position. This is called a type-2 error or a collision. A choice has to be made which of the two involved positions should be stored in the transposition table. There are several replacement schemes (Breuker, Uiterwijk, and van den Herik, 1994), which deal with the collision problem. We use a simple scheme that overwrites the element in the table if the depth of the position stored in the table is smaller than or equal to the depth of the current search. Notice that for every new search process the transposition table is cleared.

4.7 Other methods

In the literature, many other enhancements to $\alpha\beta$ exist. Even though they are not implemented in our program, we will shortly describe two examples here.

Futility pruning (Heinz, 1998) is a well-known technique. The idea is not to spend more time looking at a given node, because it is considered bad. The problem, however, is that it is not known when a node is bad. In other, more older games, like Chess, this knowledge is well-developed, but in newer games like OCTI it is guesswork due to the lack of grandmasters.

The *countermove heuristic* (Uiterwijk, 1993) is another move-ordering technique. The idea is that many moves also have a "natural" countermove, which is the best move to respond, irrespective of the actual board position. Since the results of using the countermove heuristic are often similar to those of the history heuristic, we have chosen not to implement this method.

4.8 Overall framework

The overall way to combine all known techniques in the program is as follows. First, the transposition table is checked if it contains some information about the current position. If it does, and the search depth in the table is deep enough, the tree is pruned using the data in the table. If it does not, the move stored in the table is first used in the move ordering. Next, if available, the killer move is added second in the list of possible moves. And lastly, the remaining moves are generated for the current position which are then ordered using the history tables. When zugzwang is not an issue, a depth-reduced null-move search is done before the actual search with the principal-variation search algorithm.

Chapter 5

Evaluation function

Winning is enjoyable, but losing does not detract from the pleasure of playing. — Nakayama Noriyuki

As stated in the previous chapters, the evaluation function is (when combined with the search algorithm) the heart of a computer game-playing program. In order to conduct an intelligent search, the evaluation function should give a good estimate of the current game value at any time. In section 5.1 we will first examine some general principles of OCTI. We will list useful strategies in section 5.2, describe the actual implementation of the evaluation function in section 5.3 and briefly discuss its fine-tuning in section 5.4.

5.1 General principles of OCTI

The goal of OCTI is either to occupy one of the enemy bases, or to disallow the opponent to move by capturing all of his pods for example. Winning a game can thus be achieved via two distinct methods. Quite obviously, the evaluation function has to take this duality into account.

The primary objective in the game is to occupy as quickly as possible one of the opponent's bases. When playing OCTI, this is the preferred and easiest strategy to win the game. Therefore, positions which have friendly pods closer to the enemy bases should yield a better score.

The secondary objective in the game is to capture as many of the opposing pods as possible, without endangering your own pods. In the evaluation function, a position with less friendly pods should have a worse score than a position with more pods.

5.2 Strategies revisited

Aside from the obvious objectives in the game, several strategies can be used as a rule of thumb to yield better play. The four main strategies which were previously discussed in section 2.4 should therefore be incorporated into the evaluation function. Below we provide a quick reminder of the main strategies.

- Trade-off between building and moving
- Move your pods in teams
- Do not let the opponent capture pods and/or prongs
- Make pods move forward and backward

To make the evaluation function more efficient, a few conditions have to be added to these strategies. More details on the actual implementation of the function are given in the next section.

5.3 Implementing the evaluation function

The basic idea of the evaluation function is to attribute a certain number of points to each pod on the board. These points are calculated using two major factors: material difference, described in subsection 5.3.1 and the position of the pods on the board, described in subsection 5.3.2.

5.3.1 Material difference

Some preliminary testing has been done to discover good values for pods and prongs. These have been set at 1000 points for a basic pod and 100 points for a basic prong. Several bonuses can be added, 500 points for a friendly pod in the vicinity of the current pod or 200 points for a prong pointing in the direction of the opponent's OCTI squares. An additional bonus of 400 points can be added when a pod has prongs in opposing directions, as this can be important in the endgame.

The first basic strategy states that it is better to build your pieces than to move your pieces when not necessary. In the evaluation function, additional points are added to a pod when a prong is added. When a player is not in danger of losing any pieces, adding a prong is often the best choice. Not making this move will result in a loss of 100 points. If a player has no more available prongs, he will not be able to gain more points by fortifying his pieces and will have to try other methods.

The second strategy claims that is is better to move your pods in teams, as this makes higher mobility possible and allows for quick attacks on the opponent's

pieces. In the evaluation function, pods have a basic value of 1000 plus their prongs, but when a friendly pod is located in an adjacent square and a prong allows a jump, an additional 500 bonus points is added to this current pod score. If the other friendly pod can also jump over this pod, it too will receive 500 bonus points. This allows for a steady bonus when pods are kept together. In tactics, this both allows for fast moving due to jumps, but also for blocking the opponent's jumps over your pods.

The third strategy says that it is vital not to let your opponent capture any pods and/or prongs. Capturing prongs is only possible when capturing pods. When a player captures a pod, the current player loses the points attributed to the captured pod, thus his score will be significantly lower. The search algorithm will make sure not to lose a pod, since this is a very important basis for points in the evaluation function. Losing pods makes it harder to win, so the punishment in the evaluation function should be severe.

The fourth strategy implies that it is advisable to have your pods move forward and backward when near the opponent's OCTI squares, as this allows capturing a pod on a home base and stepping on it (thus winning) the next move. In the evaluation function this is implemented as an additional bonus of 400 points once the pod has past the middle of the board (rows 5, 6 and 7 for the yellow player; and rows 1, 2, and 3 for the red player). Thus the pod must have both prongs A and E, B and F, C and G, and/or D and H.

5.3.2 Pod position

Another important factor when attributing points to a certain position is the place of the pods on the board. When the pods are closer to the opponent's OCTI squares, the score should be higher as winning is easier. To accomplish this, the distance of the square where the friendly pod is located to the enemy bases is calculated. The vertical dimension of the board is used as a yardstick, such that the smaller the distance is, the higher the bonus factor will be. Some preliminary testing revealed that 5000 is a good value to use as a bonus score. If d is the distance from a square to the opponent's home bases, the static factor $Score_{Dist}$ is calculated as follows:

$$Score_{Dist} = (7 - d) \times 5000.$$

This factor is calculated for each pod the player has, so if a player loses a pod to the opponent, he will also lose the bonus due to the positioning of his pods on the board.

5.4 Fine-tuning the evaluation function

The values used in the current version of the evaluation function have only been obtained through some preliminary testing. In the early stages of program development, these values have proved to be the most effective after some trial-and-error testing. However, these values are probably far from optimal and the evaluation function still needs some fine-tuning. In chapter 7 however, some extensive testing will be done with the evaluation function, to make it as optimal as possible.

Chapter 6

Search algorithm testing

One bad move nullifies forty good ones. — Horowitz

This chapter describes some numerical tests about the techniques described in chapter 4. All results in this chapter are taken from a series of 100 games of OCTI. Each algorithm played 50 times as yellow, and 50 times as red. A small random factor was also added to the evaluation function to avoid playing the same game 100 times. For each turn, the search depth of the algorithm was fixed, and iterative deepening was enabled.

First, the various windowing techniques were tested in section 6.1. The best algorithm was then extended with move ordering techniques in section 6.2 and later the transposition tables were added in section 6.3. Finally, in section 6.4 a short conclusion is given about all search algorithm tests.

The windowing techniques were tested first, since they do not change the tree traversal but only allow for more and better cut-offs. The transposition tables were tested last since they should always have a positive effect whenever iterative deepening is used. In all tests, the results will be displayed as the average number of nodes searched per search depth, the time in seconds it took to reach that depth and the percentage of nodes gained when using two different search algorithms.

6.1 Windowing techniques

Three windowing techniques were implemented into the game: aspiration search, principal variation search, and the null move. First the two algorithms will be tested against plain $\alpha\beta$ in subsections 6.1.1 and 6.1.2, then the best algorithm will be tested against itself with the null move enabled in subsection 6.1.3. These tests were run at a fixed search depth of 5 plies, as it almost takes a day to complete a test run of 100 games at this depth.

6.1.1 Aspiration search

In the first test, plain $\alpha\beta$ played against aspiration search. For aspiration search, the window was set from $\alpha - 1500$ to $\beta + 1500$ when starting a new search.

	Alpha Beta		**AS**		
depth	# sec	# nodes	# sec	# nodes	% gain
1	0.0	35	0.0	34	2.8
2	0.008	540	0.011	647	−19.8
3	0.130	8,212	0.126	8,147	0.8
4	1.760	105,709	2.011	116,786	−10.4
5	21.314	1,194,108	19.657	1,134,986	4.9

Table 6.1: Aspiration search test results.

As table 6.1 shows, aspiration search performs much like normal $\alpha\beta$, with only a small enhancement at depth 5.

6.1.2 Principal-variation search

In this test, $\alpha\beta$ was compared to principal variation search.

	Alpha Beta		**PVS**		
depth	# sec	# nodes	# sec	# nodes	% gain
1	0.0	34	0.0	39	−14.7
2	0.008	521	0.010	676	−29.7
3	0.123	7,747	0.119	6,729	13.4
4	1.717	100,714	1.275	69,247	31.2
5	19.893	1,144,571	10.929	566,722	50.4

Table 6.2: Principal-variation search test results.

As shown in table 6.2, principal variation search performs a lot better than aspiration search. At deeper search depths, the performance gain ranges from 30% to 50%.

6.1.3 Null-move search

In the last test of this series, two different versions of principal variation search played against each other. One used the null move heuristic, and the other did not. For the null move, R (the reduced-depth parameter) was set to 2.

	Null move disabled		Null move enabled		
depth	# sec	# nodes	# sec	# nodes	% gain
1	0.0	40	0.0	40	0.0
2	0.009	633	0.011	745	−17.6
3	0.126	7,123	0.047	2,818	60.4
4	1.269	65,311	0.398	22,515	65.9
5	11.869	624,345	2.237	121,311	80.5

Table 6.3: Null-move search test results.

As table 6.3 clearly shows, when the null move is used (from search depth 3 onward) it yields a significant reduction in nodes searched, ranging from 60% to 80%.

6.2 Move ordering

Two different game-independent move-ordering techniques were implemented for OCTI: the killer-move heuristic, and the history heuristic (the use of the transposition move is tested in subsection 6.3.3). The best algorithm this far, PVS with the null move enabled, will play against its counterpart with the killer move enabled in subsection 6.2.1, and later with both ordering techniques enabled in subsection 6.2.2. Furthermore, in subsection 6.2.2 a few variants of the history heuristic are also tested. In all these tests, most settings remain unchanged, but the search depth was locked at 6 ply since move ordering enabled a considerable speed-up in search time.

6.2.1 Killer-move heuristic

In the killer-move test, PVS with the null move enabled was equipped with the killer-move heuristic. One killer was recorded per search depth.

	Killer disabled		Killer enabled		
depth	# sec	# nodes	# sec	# nodes	% gain
1	0.0	41	0.0	41	0.0
2	0.011	675	0.008	472	30.0
3	0.057	3,026	0.028	1,435	52.5
4	0.411	20,296	0.181	7,711	62.0
5	2.401	114,343	0.790	33,345	70.8
6	15.789	770,456	5.045	207,272	73.0

Table 6.4: Killer-move heuristic test results.

As shown in table 6.4, using the killer heuristic as move ordering on top of PVS and null move yields another speed-up of up to 73% at depth 6.

6.2.2 History heuristic

To test the impact of the history heuristic, six tests were run. In test (a), the best algorithm this far, PVS with the killer move enabled, played against its counterpart with the history heuristic enabled. For the first test, the values in the history tables were incremented with 1 each time a best move was found. In test (b) the best algorithm from test (a) played against another history variant, where the tables were incremented with d, the depth at which the move was found to be best. In the next two tests, (c) and (d), the same set-up was used, but the incrementation values were set to d^2 and 2^d.

The best algorithm after these four tests will once again be tested against its counterpart that uses game-specific ordering. In test (e) we will order the moves in such a way that jumps and captures are tried before all other moves. In test (f) the moves will be ordered in yet another way: first jumps and captures, next moves involving adding prongs to the pods, and lastly the remaining moves.

	History disabled		**History (+1) enabled**		
depth	# sec	# nodes	# sec	# nodes	% gain
1	0.0	41	0.0	39	4.8
2	0.009	466	0.008	242	48.0
3	0.032	1,412	0.028	778	44.9
4	0.191	7,233	0.199	3,964	45.1
5	0.866	33,748	0.970	19,206	43.0
6	5.412	203,366	5.889	112,868	44.5

Table 6.5: History heuristic test (a) results.

As table 6.5 shows, using the history heuristic yields an additional reduction in search nodes of about 45%.

In test (b), two algorithms with history-heuristic move ordering were tested against each other. One player used the previously used +1-scheme, while the other player increments the counters with d, the search depth at which the best move was found.

	History +1		**History +d**		
depth	# sec	# nodes	# sec	# nodes	% gain
1	0.0	39	0.0	39	0.0
2	0.006	234	0.006	242	−3.4
3	0.006	779	0.025	754	3.2
4	0.138	3,549	0.142	3,597	−1.3
5	0.780	19,279	0.801	17,314	10.1
6	4.657	101,459	5.144	110,091	−8.5

Table 6.6: History heuristic test (b) results.

The results in table 6.6 show that both schemes are only marginally different. We will therefore continue to use 1 as incrementation factor in the following tests.

In test (c), the counters were incremented with the square of the search depth d^2 at which the best move was found.

	History +1		**History +$\mathbf{d^2}$**		
depth	# sec	# nodes	# sec	# nodes	% gain
1	0.0	39	0.0	38	2.5
2	0.006	234	0.006	238	−1.7
3	0.022	742	0.024	755	−1.7
4	0.134	3,585	0.151	3,628	−1.1
5	0.699	18,449	0.765	18,367	0.4
6	4.091	97,439	4.855	106,539	−9.3

Table 6.7: History heuristic test (c) results.

Once again, as table 6.7 shows, both schemes are almost identical to each other in node reduction. In the next test, 1 will be the reference incrementation factor.

In test (d), the fourth and last test of this series, the factor 2^d was used to increment the history counters.

	History +1		**History +$\mathbf{2^d}$**		
depth	# sec	# nodes	# sec	# nodes	% gain
1	0.0	39	0.0	39	0.0
2	0.007	244	0.006	242	0.8
3	0.026	766	0.024	764	0.2
4	0.155	3,600	0.148	3,705	−2.9
5	0.804	18,411	0.771	18,484	−0.3
6	4.995	104,292	5.095	107,443	−3.0

Table 6.8: History heuristic test (d) results.

Table 6.8 shows that once more there is almost no difference between both schemes. For OCTI, there is no difference in node pruning when different versions of the history heuristic are being used. Winands (2004) reports similar results for Lines Of Action, however the results are most likely game-dependent. For simplicity, we will use the basic +1-scheme in the rest of the tests.

So far, all the tests of the history table were done by re-ordering all new generated moves with the table. In the following three tests, basic game-specific knowledge is combined with the history tables. A quick reminder: jumping and/or capturing is usually considered the best action, followed by adding a prong and thus fortifying your pieces, and lastly moving your pods on the board.

In test (e), first all jump- and/or capture-moves were generated, which were then ordered using the history tables. Then, normal prong-moves and pod-moves were generated together, which were then ordered using the history tables and were added to the list of moves.

	Normal history		**Ordered history**		
depth	# sec	# nodes	# sec	# nodes	% gain
1	0.0	39	0.0	40	−2.5
2	0.006	246	0.008	367	−49.1
3	0.024	761	0.030	1,140	−49.8
4	0.135	3,627	0.177	5,449	−50.2
5	0.701	17,066	0.763	23,381	−37.0
6	4.692	103,807	4.377	124,499	−19.9

Table 6.9: History heuristic test (e) results.

Using this version of history ordering with game-specific knowledge actually performs worse than standard history ordering, as shown in table 6.9. In the next test, the standard will therefore be normal history ordering.

In the sixth and last test, (f), first the jump/capture-moves were independently generated and ordered using the history tables. Next, the prong-adding moves were also generated and ordered and added to the list. Lastly, the remaining pod moves were generated and ordered and were also appended to the list.

	Normal history		**Ordered history**		
depth	# sec	# nodes	# sec	# nodes	% gain
1	0.0	38	0.0	39	−2.6
2	0.005	238	0.009	372	−56.3
3	0.024	761	0.031	1,150	−51.1
4	0.147	3,854	0.194	5,798	−50.4
5	0.729	18,641	0.839	25,554	−37.0
6	4.202	104,157	5.237	154,466	−48.3

Table 6.10: History heuristic test (f) results.

As table 6.10 shows, using this game-knowledge supported version of history ordering once again performs worse than standard history move ordering. The normal history heuristic will therefore remain our standard for the upcoming tests.

6.3 Transposition table

To test the effect of transposition tables on OCTI, each of its components was enabled alone and tested against the reference player at this time: PVS with null move, killer move and history heuristic (+1). In subsection 6.3.1 the transposition table is only used when an exact hit has been found. In subsection 6.3.2 only the upper/lower bound hits in the table are used. In subsection 6.3.3 the table is solely used as a move ordering mechanism, and finally in subsection 6.3.4 all components are enabled and the table is fully used.

6.3.1 Only using exact table hits

In this test, the transposition table pruning was only enabled when the minimax value stored in the table was an exact value, thus obtained from the evaluation function. The number of tthits is therefore defined as the number of times that exact values in the tables were used to cut off the search.

	TT disabled		**TT enabled (exact)**			
depth	# sec	# nodes	# sec	# nodes	#tthits	% gain
1	0.0	39	0.0	38	0	2.5
2	0.011	241	0.009	249	0	−3.3
3	0.047	766	0.044	784	1	−2.3
4	0.217	3,597	0.316	3,664	2	−1.8
5	0.977	18,858	1.599	18,075	15	4.1
6	4.859	106,084	10.505	108,655	46	−2.4

Table 6.11: TT (exact) test results.

As table 6.11 shows, only pruning the tree when an exact hit is found in the transposition table yields almost no reduction in nodes searched. The gain ranges from −3.3% to 4.1%, which makes almost no difference. Furthermore, the added overhead of adding, replacing and retrieving positions makes the algorithm slower.

6.3.2 Only using upper and lower bound pruning

In this test, only upper and lower bound hits in the table were used to speed up the search process. These values are obtained from previous α and β values and can be used to adjust the search window. In this case, the tthits count is the number of upper and lower bound hits that were used to prune the tree.

	TT disabled		**TT enabled (ubound/lbound)**			
depth	# sec	# nodes	# sec	# nodes	#tthits	% gain
1	0.0	39	0.0	38	0	2.5
2	0.013	236	0.008	241	1	−2.1
3	0.044	751	0.039	736	11	1.9
4	0.234	3,727	0.271	3,344	109	10.2
5	0.975	19,990	1.221	14,693	645	26.4
6	5.152	111,117	6.057	73,672	3,482	33.6

Table 6.12: TT (upper/lower bound) test results.

As shown in table 6.12, using upper and lower bound reduces the number of nodes with about 33% at search depth 6.

6.3.3 Only using move ordering

In this test, the transposition table was only used as a move ordering mechanism. If the current position has a value and a move stored in the table, that move is tried first when searching the position, even if the value stored in the table could have been used to cut off the tree. The tthits value stands for each time the transposition table is used to re-order the moves.

	TT disabled		**TT enabled (ordering)**			
depth	# sec	# nodes	# sec	# nodes	#tthits	% gain
1	0.0	37	0.0	39	0	−5.4
2	0.013	237	0.007	228	1	3.7
3	0.043	751	0.042	740	14	1.4
4	0.232	3,632	0.310	3,561	124	1.9
5	0.901	17,569	1.552	17,178	656	2.2
6	5.229	106,712	8.461	87,362	4,996	18.1

Table 6.13: TT (ordering) test results.

As the results in table 6.13 show, only using the move ordering feature of the transposition table yields a reduction in nodes searched of about 18% at search depth 6.

6.3.4 Full use of the table

In the last test of this series, the transposition table was fully enabled, combining all three features tested in the previous tests. This time, tthits is the number of times some information stored in the table was used in the search. This can either be an exact hit, an upper/lower bound hit or a re-ordering hit.

	TT disabled		**TT enabled (full)**			
depth	# sec	# nodes	# sec	# nodes	#tthits	% gain
1	0.0	38	0.0	38	0	0.0
2	0.013	240	0.008	241	1	−0.4
3	0.046	770	0.038	731	14	5.0
4	0.227	3,691	0.272	3,371	121	8.6
5	0.968	19,065	1.310	14,583	689	23.5
6	5.247	105,389	6.325	73,939	3,653	29.8

Table 6.14: TT (full) test results.

As table 6.14 shows, using the full transposition table lowers the number of nodes searched with about 30% at depth 6.

6.4 Chapter conclusions

As seen throughout this chapter, most search enhancements yield a positive result in search nodes reduction. So far, only aspiration search has a negative influence. Therefore, the recommended search algorithm will use the following enhancements:

- Principal-variation search;
- Null-move search with factor $R = 2$;
- The killer-move heuristic with 1 killer per depth;
- The history-heuristic with factor +1;
- The transposition table fully used.

Chapter 7

Evaluation function testing

Some part of a mistake is always correct. — Savielly Tartakover

The evaluation function has two major components: the material difference and the position of the pods on the board. These two scores are added up to form the actual evaluation of the current position.

This chapter describes several tests, each consisting of 100 games of OCTI with the players switching sides halfway through. In each of these tests, a factor ranging from 0.0 to 2.0 in 10 steps was multiplied with one of the components of the evaluation function. After 100 games, the factors which gave the best results (highest win percentage) were once again tested in 10 smaller steps to find the best value. This value is then set as the reference when testing the next component.

In section 7.1 we will first test the material difference component, followed by the pod position component in section 7.2. Finally, in section 7.3 we will give some short chapter conclusions.

7.1 Material difference testing

To find the best material difference factor, 10 tests were run, each consisting of 100 games of OCTI. In each of the tests, one player had its material difference component of the evaluation function multiplied with a factor ranging from 0.0 to 2.0 in 10 steps (thus 0.0, 0.2, 0.4, 0.6, 0.8, 1.2, 1.4, 1.6, 1.8 and 2.0). The other player played with the reference factor 1.0. After each test, the win percentage of the player with the varying factor was recorded. Figure 7.1 plots these win percentages against the factor used in the test.

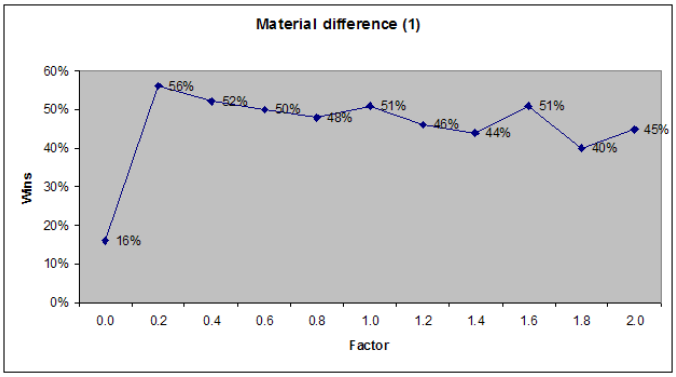

Figure 7.1: Test results: Material difference (1).

As figure 7.1 shows, the optimum lies around factor 0.2, so 10 more tests were run with factors ranging from 0.0 to 0.4. The factor was increased by 0.033 each time. The results can be found in figure 7.2.

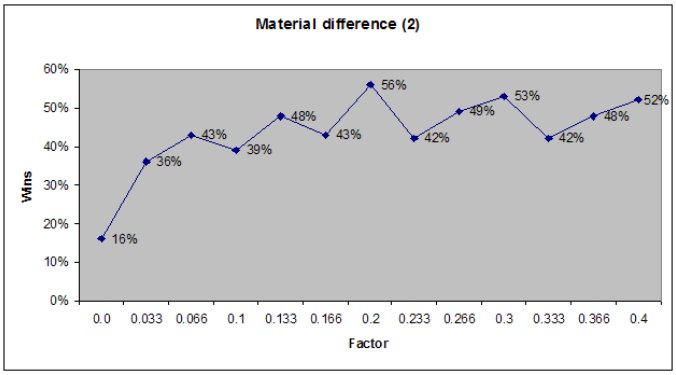

Figure 7.2: Test results: Material difference (2).

As both tests show, factor 0.2 yields the most wins (56%), thus this factor is the optimum to use in the final evaluation function. This factor will now become our reference for both players in the following test.

7.2 Pod position testing

These tests are similar to the first series of tests described in the previous section. The factors were once again ranging from 0.0 to 2.0 in 10 steps. The previously obtained factor 0.2 for the material difference is used in both players in this test. Figure 7.3 plots the win percentage against the factor used.

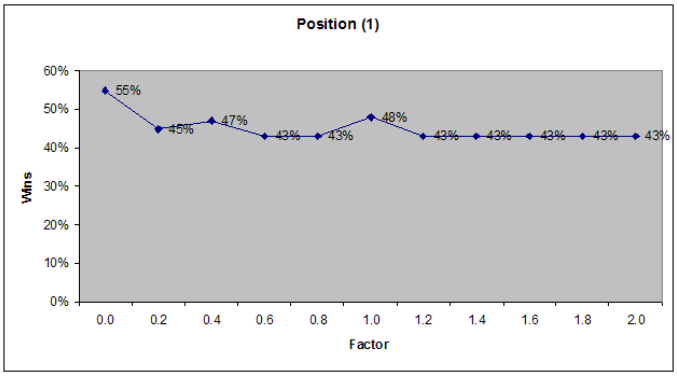

Figure 7.3: Test results: Position (1).

As shown in figure 7.3, the optimum lies around factor 0.0. For the second series, once more 10 tests with a factor around 0.0 were run, of which the results are shown in figure 7.4. The games with a factor below 0 are not shown in the graph, since they resulted in a 100% loss each time. With a factor below 0, the evaluation function actually encouraged defeat, the player tried to capture his own pods as soon as possible.

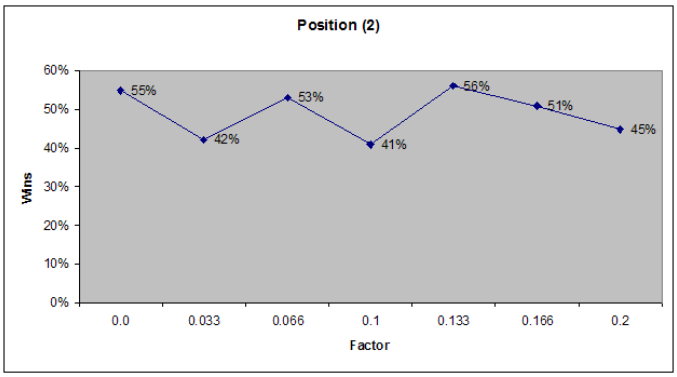

Figure 7.4: Test results: Position (2).

As can be concluded from both tests, factor 0.133 is the optimum for the second component of the evaluation with 56% wins.

7.3 Chapter conclusions

After testing the evaluation function, factor 0.2 for the material difference component and factor 0.133 for the pod position component were found to be the best choice. These will be incorporated in the final evaluation function of the program. This also shows that the pod position component is actually the more important component in the evaluation, even though its factor is lower, the total score when compared to the material difference factor is still higher.

Chapter 8

Conclusions

Once the game is over, the king and the pawn go back in the same box. — Italian proverb

This chapter contains the final conclusions on our research. Section 8.1 revisits the problem statement and research questions, and section 8.2 lists possibilities for future research.

8.1 Problem statement and research questions revisited

In section 1.3 we have defined the following research questions:

What is the complexity of OCTI?

In chapter 3, we have seen that the state-space complexity of OCTI is $O(10^{28})$, and the game-tree complexity is $O(10^{37})$. These numbers however are only an approximation, as the state-space complexity includes some unreachable positions, and the game-tree complexity is based on games between humans and is subject to change once more computer programs that can play OCTI are created. Both the state-space complexity and the game-tree complexity of OCTI are comparable to that of the game of Checkers. Completely solving OCTI is, like Checkers, probably possible with the necessary time and research.

What known techniques can be used to play OCTI?

As described in chapter 4, many known techniques can be used for OCTI. Compared to standard $\alpha\beta$-search, PVS, the killer move heuristic, the history heuristic, null-move pruning and transposition tables all cause a more or less substantial reduction in nodes searched.

What game-specific knowledge is required to make a computer play OCTI?

The evaluation function described in chapter 5 implements different strategies that are deemed useful in OCTI, like moving your pods in teams and not letting your pods get captured.

Now that the research questions have been answered, we can also formulate an answer to the problem statement:

How can a computer program be written that plays the game of OCTI as efficiently as possible?

To write a computer program to play OCTI, known search algorithms like $\alpha\beta$-search and its enhancements: PVS, killer move heuristic, history heuristic, null move pruning and transposition tables can be used. Basic game knowledge in the form of a static evaluation function based on game strategies also have to be used.

8.2 Future research possibilities

As always, some enhancements can be made to the current research. The complexity analysis of OCTI is only an estimate. When the game gains popularity in the AI world, a better count can be calculated.

The search algorithms and enhancements can be more fine-tuned. The null move is used in the game, with a pretty basic way to define zugzwangs. Maybe a better definition can be found. The factor R is currently set at 2 in the whole game, an adaptive R depending on the search depth might provide better results. Storing two killer moves in the search might provide a reduction in nodes. The same argument can be made for storing two transpositions for each table key. After a search, dividing the counts in the history tables by another value than 2 might be beneficial. And maybe other techniques that are not described here can be applied to OCTI. Furthermore, it might be useful to try to construct an opening book for OCTI. Also, an endgame database could be another possible addition. Even so, it could be possible to create a more efficient or faster implementation of OCTI.

The evaluator is still a crude estimate of the value of a board position. A more elaborate (and perhaps faster) evaluator might possibly be created. Also techniques like pattern matching and machine learning might be applied to OCTI, since they are under investigation in other games, including Checkers.

Finally, the program could be tested against other opponents to measure its actual playing strength. These opponents can be either humans or computers. Due to the absence of human grandmasters and other OCTI programs this was not possible in the scope of this research.

Bibliography

Akl, S. G. and Newborn, M. M. (1977). The Principal Continuation and the Killer Heuristic. *1977 ACM Annual Conference Proceedings*, pp. 466–473.

Beal, D. F. (1989). Experiments with the null move. *Advances in Computer Chess*, Vol. 5, pp. 65–79.

Bouton, C. L. (1901). Nim, a Game with a Complete Mathematical Theory. *Annals of Mathematics*, Vol. 2, No. 3, pp. 33–39.

Breuker, D. M., Uiterwijk, J. W. H. M., and Herik, H. J. van den (1994). Replacement Schemes for Transposition Tables. *ICCA Journal*, Vol. 17, No. 7, pp. 183–193.

Breuker, D. M. (1998). *Memory versus Search in Games*. Ph.D. thesis, Universiteit Maastricht, Maastricht, The Netherlands.

Brudno, A. L. (1963). Bounds and Valuations for Abridging the Search of Estimates. *Problems of Cybernetics*, Vol. 10, pp. 225–241.

Donninger, C. (1993). Null Move and Deep Search: Selective-Search Heuristics for Obtuse Chess Programs. *ICCA Journal*, Vol. 16, No. 3, pp. 137–143.

Gendelman, D. and Meshulam, R. (2004). TESTME2 wins OCTI 6×7 tournament. *ICGA Journal*, Vol. 27, No. 3, pp. 181–183.

Green, D. (2000a), OCTI rules and strategy. `http://www.octi.net/`.

Green, D. (2000b), Rules to OCTI. `http://www.octi.net/`.

Groot, A. D. de (1946). *Het Denken van den Schaker, een Experimenteel-psychologische Studie*. Ph.D. thesis, University of Amsterdam. In Dutch.

Heinz, E. A. (1998). Extended Futility Pruning. *ICCA Journal*, Vol. 21, No. 2, pp. 75–83.

Herik, H. J. van den (1983). *Computerschaak, Schaakwereld en Kunstmatige Intelligentie*. Ph.D. thesis, Delft University of Technology. In Dutch.

Herik, H. J. van den, Uiterwijk, J. W. H. M., and Rijswijck, J. van (2002). Games solved: Now and in the future. *Artificial Intelligence*, Vol. 134, pp. 277–311.

Huberman, B. J. (1968). *A Program to Play Chess End Games*. Ph.D. thesis, Stanford University, Computer Science Department, USA. Technical Report no. CS-106.

Knuth, D. E. and Moore, R. W. (1975). An Analysis of Alpha-beta Pruning. *Artificial Intelligence*, Vol. 6, pp. 293–326.

Marsland, T. A. (1986). A Review of Game-Tree Pruning. *ICCA Journal*, Vol. 9, No. 1, pp. 3–19.

Minsky, M. (1968). *Semantic Information Processing*. M.I.T. Press, Cambridge, MA, USA.

Neumann, J. von and Morgenstern, O. (1944). *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, NJ, USA.

Newell, A. and Simon, H. A. (1972). *Human Problem Solving*. Prentice-Hall Inc., Englewood Cliffs, NY, USA.

Nilsson, N. J. (1971). *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill Book Company, New York, NY, USA.

Russell, S. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice-Hall Inc, Englewood Cliffs, NJ, USA.

Schaeffer, J. and Plaat, A. (1997). Kasparov versus Deep Blue: The Rematch. *ICCA Journal*, Vol. 20, No. 2, pp. 95–101.

Schaeffer, J. (1983). The History Heuristic. *ICCA Journal*, Vol. 6, No. 3, pp. 16–19.

Schaeffer, J. (1997). *One Jump Ahead*. Springer-Verlag Inc., New York, NY, USA.

Schaeffer, J., Björnsson, Y., Burch, N., Kishimoto, A., Müller, M., Lake, R., Lu, P., and Sutphen, S. (2005a). Solving Checkers. *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pp. 292–297.

Schaeffer, J. (2005b). Solving Checkers: First result. *ICGA Journal*, Vol. 28, No. 1, pp. 32–36.

Shams, R., Kaindl, H., and Horacek, H. (1991). Using aspiration windows for minimax algorithms. *Proceedings IJCAI-91 (Sydney, Australia)*, pp. 192–197.

Shannon, C. E. (1950). Programming a Computer for Playing Chess. *Philosophical Magazine*, Vol. 41, No. 7, pp. 256–275.

Sutton, C. (2002). Computers and OCTI: Report from the 2001 Tournament. *ICGA Journal*, Vol. 25, No. 2, pp. 105–112.

Turing, A. M. (1953). Digital Computers Applied to Games. *Faster than Thought*, pp. 286–297.

Uiterwijk, J. W. H. M. and Herik, H. J. van den (2000). The Advantage of the Initiative. *Information Sciences*, Vol. 122, No. 1, pp. 43–58.

Uiterwijk, J. W. H. M. (1993). The Countermove Heuristic. *ICCA Journal*, Vol. 15, No. 1, pp. 8–15.

Winands, M. H. M. (2004). *Informed Search in Complex Games*. Ph.D. thesis, Universiteit Maastricht, Maastricht, The Netherlands.

Zobrist, A. L. (1970). *A New Hashing Method with Application for Game Playing*. Technical Report 88, Computer Science Department, The University of Wisconsin, Madison, WI, USA. Reprinted in (1990) *ICCA Journal*, Vol. 13, No. 2, pp. 69–73.