A Chinese Checkers-playing program

Master Thesis by
Paula Ulfhake F94

Supervisor: Jan Eric Larsson
Department of Information Technology
Lund University

1999-2000

Tables

Tables of Content

Table of Figures

Table of Tables

Introduction

A program playing chinese checkers has been designed. The computer controls one player with a human opponent as the other player in a two-player game. The goal was to create a program that would play good chinese checkers with a reasonable amount of wait time while the computer makes its move. This report describes the program and the algorithms it is based upon.

Computer programs playing chess are quite common and well understood. This, however, does not apply to chinese checkers and literature is very hard to find. An Internet search yielded only web sites where you could buy or play the game. A search for chinese checkers in article databases at the University Library (UB2) gave only two results, both more than twenty years old. Judging from the abstracts they were not relevant to this work. Therefore, the project had to start from the beginning by investigations of different search and evaluation algorithms from chess theory.

On different turns, it is possible to choose from a different number of moves. This means that a constant search depth will result in a different number of searched nodes, depending on the situation on the board. To take advantage of this, an adaptive search algorithm is used. This algorithm adjusts the search depth in an attempt to keep the number of searched nodes within a predetermined range.

One problem in chinese checkers is trailing marbles. One or two marbles left behind by the computer will need many moves to reach their final positions at the end of the game. The required search depth to catch this is excessive. Therefore, this must be taken care of in the evaluation function. The special evaluation function written to avoid this introduces a reward for moving one of the two last marbles. This gives a dynamic aspect to the evaluation.

Another feature of the final version of the evaluation function is that it favors marble-positions close to the horizontal center of the board. The reason is that there probably are more paths to the goal in the middle of the board than at the edges. Consequently, the computer plays a more efficient game.

When the program was run on a 300 MHz Celeron PC, the adaptive algorithm took about 10 seconds when the maximum number of searched nodes was set to 300 000. The resulting search depth is then 4 ply or more. This is sufficient to beat most human players.

The program is designed using Microsoft Visual C++ 5.0 under Windows 98. It consists of seven program and header file pairs, all together more than 5000 lines of code including that generated by the compiler. Designing the program took approximately twelve 40-hour weeks, over a period of eight months.

1: Description of the game

The Rules

Chinese Checkers is played on a board shaped like a six-pointed star, with one player's home at each star point. Unlike chess, which is played on a flat board, chinese checkers is played with round marbles on a board with holes, where the marbles rest. The holes are connected by lines forming a hexagonal pattern (Figure 1). The players are only allowed to move their marbles along these lines [1].

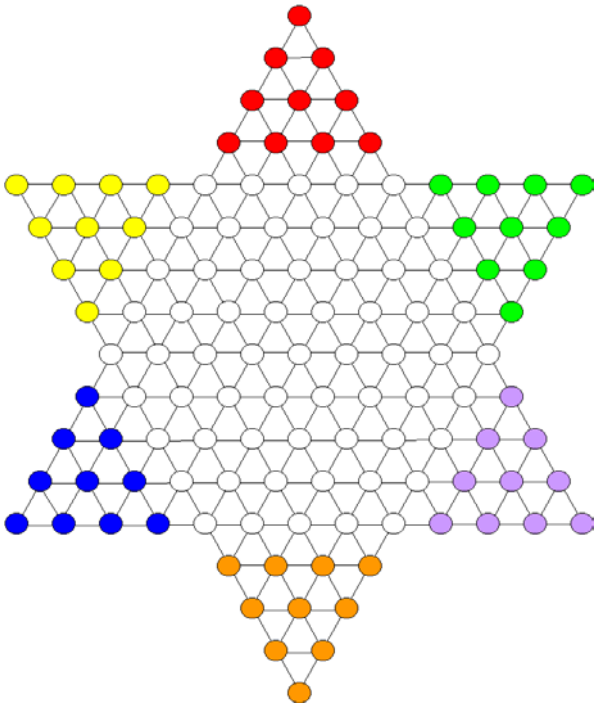

Figure 1. The board

Two to six players can play the game, each having ten same-colored marbles. At the start of the game, the player's marbles are in the ten holes of the star point that has the same color as his marbles. The goal is to move all marbles of your color from your starting point to the star point on the opposite side of the board, here called the goal. A marble can move by rolling to a hole next to it or by jumping over one marble, of any color, to a free hole, along the lines connecting the holes in a hexagonal pattern. (Figure 2) The player can make several jumps in a row, but only one roll. The player can not both roll the marble and jump with it at the same turn. When the player has moved one of his marbles, the turn passes on to the next player. Unlike chess, you never remove any game pieces from the board.

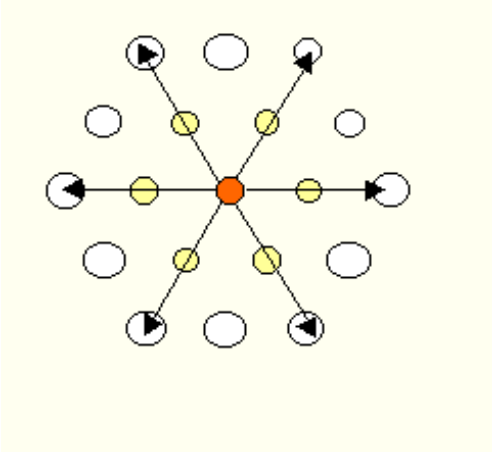

Figure 2. The central marble can either roll to one of the six lightly colored holes next to it, or if these holes are filled with other marbles, use them to jump along the arrows to one of the six free holes the arrows point to.

Since it is allowed to make several jumps in a row, it is strategically important to make it possible to so. By building structures for the marbles to jump on, it is possible to quickly move a marble to the opposite side of the board. The ability to recognize the opportunity to make a long jump is critical for playing a good game.

It is also important not to leave any marbles behind when the others move over the board. Those marbles will need more turns to cross the board than if they had been moved with the others, since their opportunities to make long jumps are fewer.

The players should always be placed symmetrically around the board. Two players use colors opposite to each other; three players use colors forming a triangle; four players form two pairs of opposite colors; five players is not recommended; six players use all colors.

Only the two-player game has been studied, one human against the computer. The computer plays with the red marbles starting at the top of the board and the human opponent plays with the opposite orange marbles.

Comparison with chess

Chess programs are very common now and there is a lot of literature about them. This has been exploited in this project by modifying methods from chess programs to suit chinese checkers.

Both chess and chinese checkers are turn-based board games. When one player has made his move the other player has some time to think before making his move. Although the game pieces and their moves are different the basic principle of choosing the best move is still the same for both games. The value of each possible move is calculated and the move that gives the best result is chosen. Chess literature describes different algorithms to find the best move to make. Some of these will be presented further on in the text.

2: User interface

The first thing the user sees when he starts the program is the program window with the gameboard and a menu bar, see figure 3. How to use the program and the different menu choices are described below.

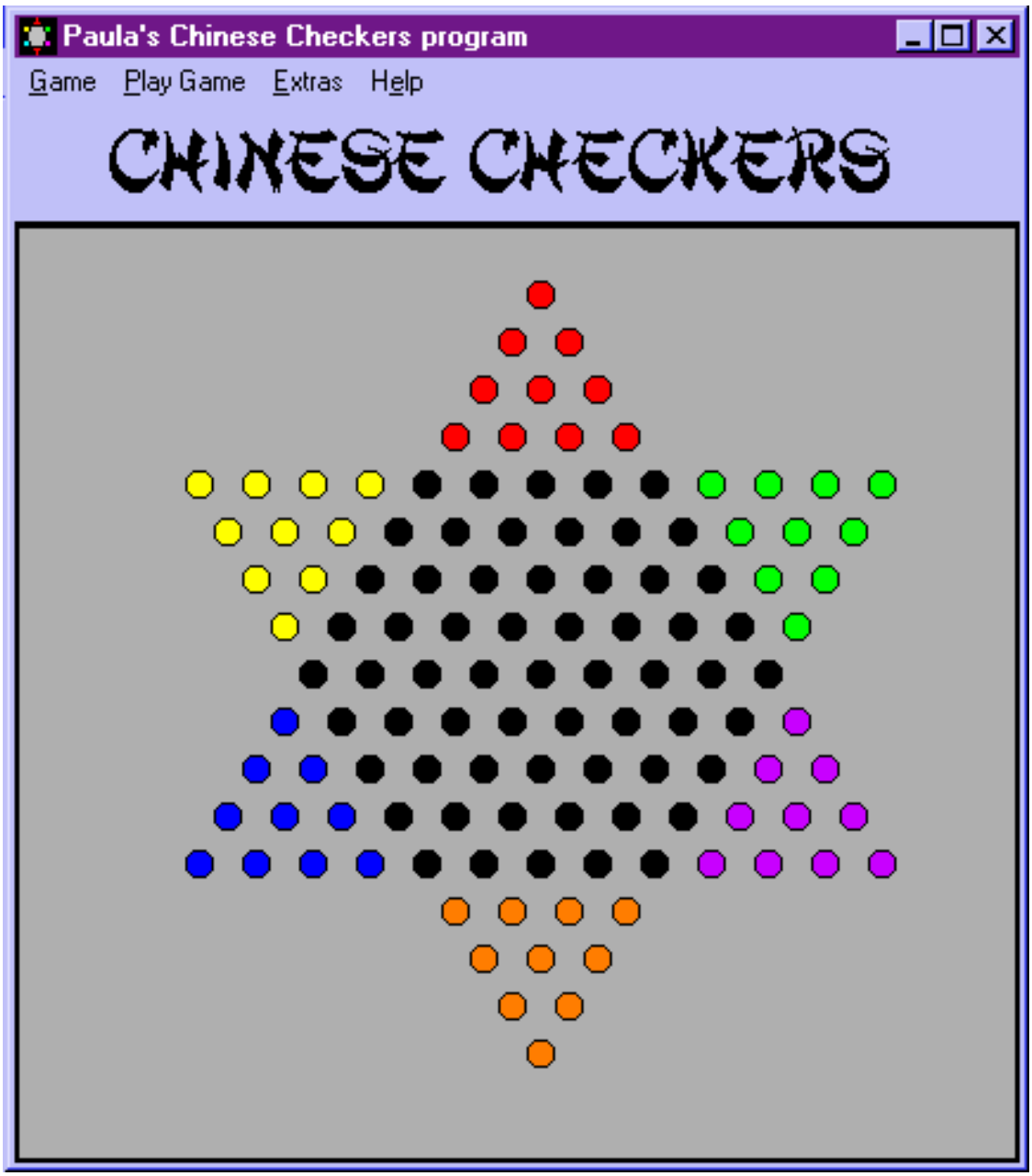

Figure 3. The program window

Starting and playing the game

To start a new game, choose "Play Game" from the menu bar. The choices in the Options dialog window that pops up are explained later in this chapter. When you're ready to start playing press OK.

Whose turn it is to start is randomly decided. If it is the computers turn, it will start automatically. When it is your turn, move the mouse pointer to the marble you want to move and press down the left mouse button. With the button pressed down move the mouse pointer to the hole you want to move the marble to and then release the mouse button. It the move is permitted the marble will move to the hole and the computer will then make its move. If the move is not permitted nothing will happen and you have a new chance to make a move. The latest move that has been made is marked with a white line from the hole the marble started in to the hole in which it landed. This makes it easier to see what move the computer made.

You and the computer will take turns making a move, with the computer automatically making its move as soon as you have made a legal move, until either you or the computer wins. When either of you moves his last marble into the goal, a message box pops up telling who won. You can then quit playing or start a new game by choosing "Play Game".

Since only legal moves are made it is impossible to cheat when playing a regular game. When not playing a game it is possible to move the marbles without following the rules. This is useful for trying different positions. To find the computers response to the situation on the board choose "Extras" "Evaluate Testboard". The pop-up screen gives information about the computers search for the right move to make.

The menus

On the menu bar, you have the following options: "Game", "Play Game", "Extras", and "Help".

If you click on "Game", you open a menu with the choices "Options" and "Quit". If you choose "Options" the Options dialog is displayed allowing you to change the algorithm the computer is using etc. If you choose "Quit" the program exits.

As described above choosing "Play Game" starts the game. You can also use this to start a new game even if you have not finished the game you are playing.

The Extras menu contains "Show special Board" and "Evaluate Testboard". "Show special Board" opens to reveal the choices "Reset Board (Show Startgame Testboard)", "Show midgame Testboard", and "Show endgame Testboard". With the "Reset Board" option, the board is restored to the starting position. "Show midgame Testboard" and "Show endgame Testboard" display special configurations of the marbles on the board, to simulate a position in the middle of a game and a position at the end of the game (figure 4). These positions have been used to compare the results of different algorithms. When the "Evaluate Testboard" option is chosen the computer makes a move based on the configuration of marbles on the board and displays the statistics for that move in the Results dialog (see further down).

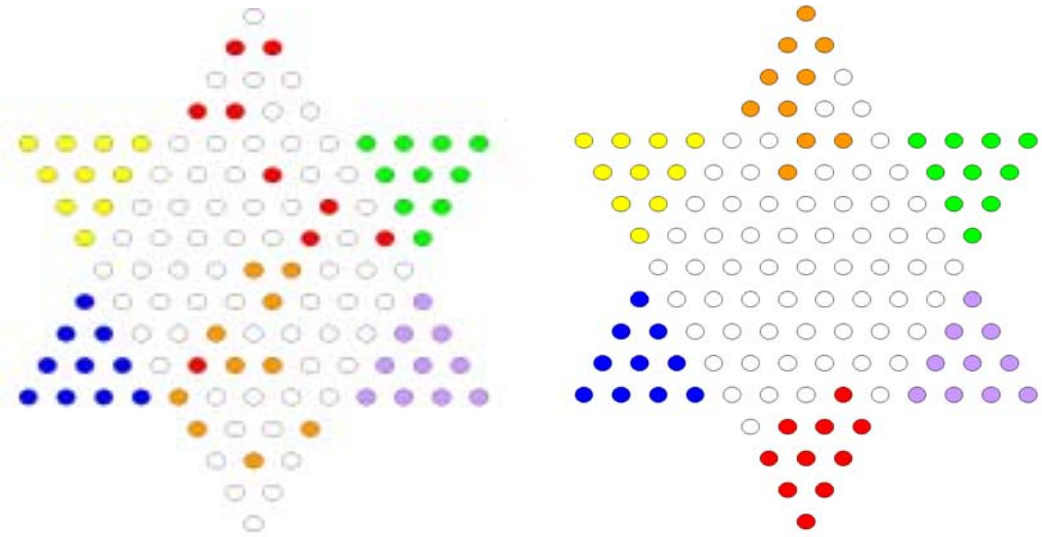

Figure 4. Left: the midgame testboard. Right: the endgame testboard

The "Help" menu has the options "Help" and "About...". "About ..." displays the About screen with information about the program. Help is currently unavailable.

The Options dialog

In the Options dialog (figure 5) you can change how the computer is playing.

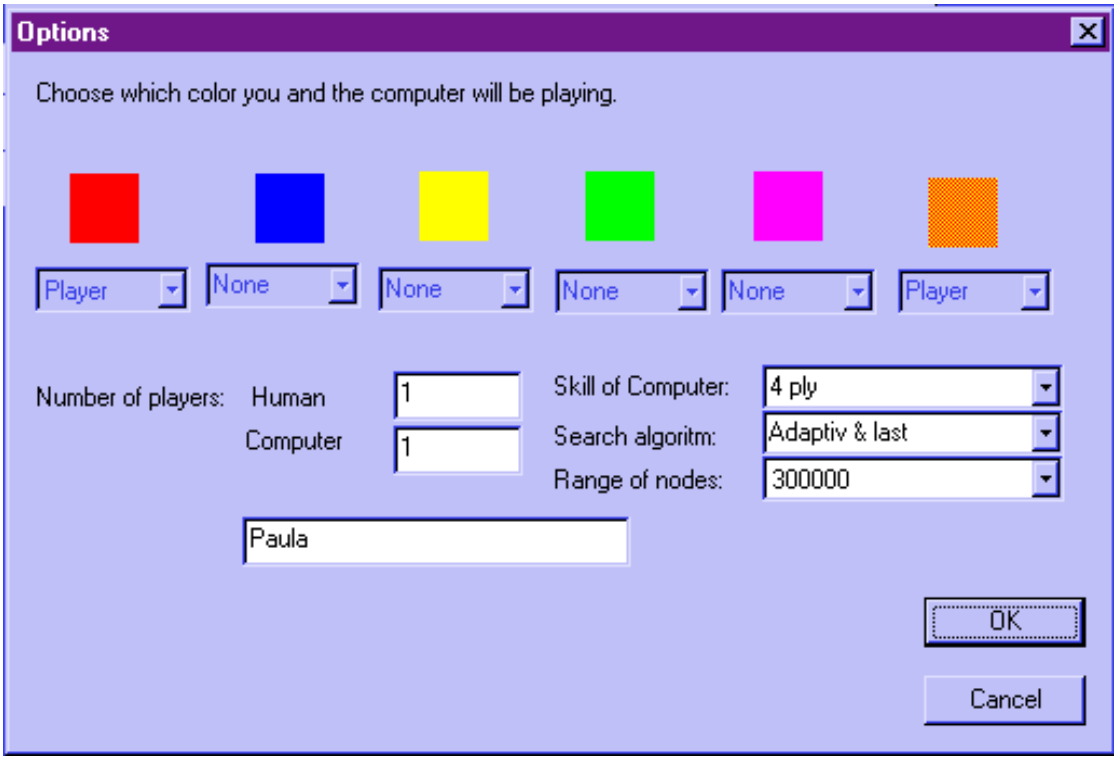

Figure 5. The Options dialog

You can choose which algorithm it will use by changing the "Search algorithm" combo-box. The meaning of the different choices is explained in detail in chapter 6, but they are also listed in table 1 for quick reference.

Table 1. Quick overview of the choices in the "Search algorithm"-combobox.

Name	Description
Minimax	The earliest algorithm. It explores all possible moves for a certain number of turns forward.
Alphabeta	The earliest algorithm using the alpha-beta search algorithm to limit the number of possible moves looked at.
Alphabeta 2	Similar to *Alphabeta* but with the moves searched in a different order for the different players.
Alphabeta 2_I	Same as *Alphabeta 2* but with the moves searched in opposite order.
Alphabeta 3	Similar to *Alphabeta* but with the moves to the furthest holes checked first.
Alphabeta 3_I	Same as *Alphabeta 3* but with the moves searched in opposite order.
Alphabeta 4	Similar to *Alphabeta* but with the longest moves searched first (maximum difference between start hole and finish hole).
Alphabeta 5	Similar to *Alphabeta 4* but with the moves searched in a different order to take advantage of curved board model.
Alphabeta 6	Same as *Alphabeta 4* but with an incremental evaluation function.
Alphabeta 6_2	Same as *Alphabeta 6*, but stops the search when the search-tree reaches a certain size.
Minimax 2	Same as *Minimax*, but with incremental evaluation function
Begin & End Game	Searches only own marbles' moves at start and end of game, uses *Alphabeta 6* in mid-game.
Adaptive algorithm	Same as *Begin & End Game*, but searches a different number of turns depending on the number of possible moves, i.e. if the tree grows too large it restarts the search to search fewer turns or if the tree is too small it restarts the search to search one more turn.
Adaptive algorithm2	Uses an adaptive search-depth as in *Adaptive Algorithm*, but using *AlfaBeta6_2* only, i.e. searches both players' moves in all parts of the game.
Adaptive & last marble	Same as *Adaptive algorithm* but also checks for trailing marbles.

"Skill of the computer" is a combo-box where you can choose how many ply (turns) the computer is allowed to look ahead. You can choose any integer between 1 and 7. The reason you can not choose a number higher than 7 is that even with a very fast computer it would take too long time. This is because the wait time grows exponentially as a function of the number of turns.

The adaptive algorithms all use the "Range of nodes" combo-box instead of the "Skill of the computer" combo-box, to choose how large search-tree is allowed. The size of the search tree is influenced not only by how many turns the computer looks forward, but also by how many different moves it is possible to choose from at each turn. Limiting the size of the search tree

is therefore a better measure of how long you have to wait for the computer to make its move, than just deciding the number of turns to look ahead.

The Results dialog

The Results dialog shows the result of the evaluation of the board (figure 6). The number of nodes visited in the search tree, how many milliseconds the search took and the number of ply (turns) the computer looked forward is shown. Most of the information in the tables in this report comes from this dialog. The Results dialog is shown when "Evaluate Testboard" is chosen in the "Extras" menu.

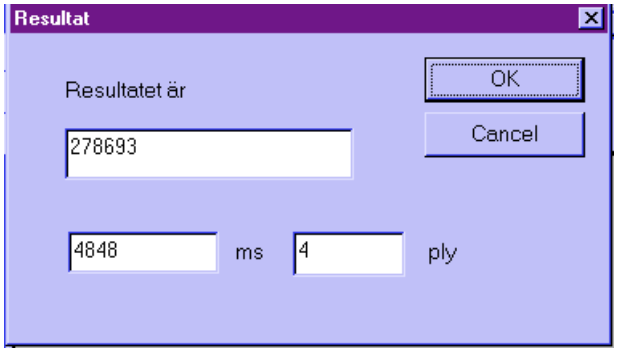

Figure 6. The Results dialog

3: General game theory

Principles of evaluation

In order to be able to decide which move to make one has to somehow evaluate the current situation and what the result will be when a move is made. A human player does this by looking at the gameboard and visually estimating the value of the situation and the possible opportunities or threats. The computer does this a little differently. Instead of looking at the whole game board at once to recognize a promising move, it instead calculates an evaluation number for each situation and uses these numbers to search the possible moves to find the move with the best evaluation number. To be able to find the winning moves it is important that the evaluation function gives an accurate value of the possible moves relative each other. It should also be a value that is common to both players so that one player wishes to increase this number and the other wants to decrease it.

The search-tree

Every move a player can choose from makes different moves possible for the opponent. This can be viewed as a tree where the original position on the board is the trunk of the tree and the possible moves are the branches. Every branch divides into several new branches that are the possible responses to that move. This dividing can continue forever. Both chess and chinese checkers have this structure. A search tree is built from all these different possibilities of move and response.

In the most basic search tree all possible moves from a position are included in the tree, and all possible moves following that move, and so on. To keep to tree finite and of manageable size, you have to limit the number of moves you look at. The first and most important way you do this is by limiting the number of consecutive turns you look at. The next step is 'pruning the tree' by cutting away branches that it is not necessary to look at, or have a low probability of producing the best move.

Description of common search-algorithms in chess-programs

The search algorithm for the most basic search-tree is called the minimax-algorithm in chess literature [2][3], because you try to maximize the evaluation number at the computer's move and minimize it at the players move. It always finds the best move but can be very time-consuming, because it does not employ any pruning.

Another option is the alphabeta tree-pruning algorithm [2][3]. It is very simple, but still very powerful. Figure 7 is used as an example to illustrate this principle, and a pseudo-code description can be found in [4]. Suppose you have evaluated the first branch from the first node, and the evaluation number is three. When you evaluate the second branch from this node, you find that its first leaf has the value 4. Since you are maximizing the evaluation number at this level of the tree, you can be sure that the backed up value will be 4 or higher. This is clearly more than three and therefore the second branch will not be chosen since the evaluation number is minimized here. This means that you can disregard the rest of the leaves in this branch, which saves time. In the second branch from the top level the value two or less is backed up. Since this is less than three at a level where the value is maximized, no other leaves need to be looked at in this branch. At levels where the value is minimized, the method is called β-pruning and at levels where it is maximized, it is called α-pruning. The algorithm is called alphabeta tree-pruning since α-pruning and β-pruning is alternated. This algorithm always finds a move with the same value as the minimax-algorithm, but faster.

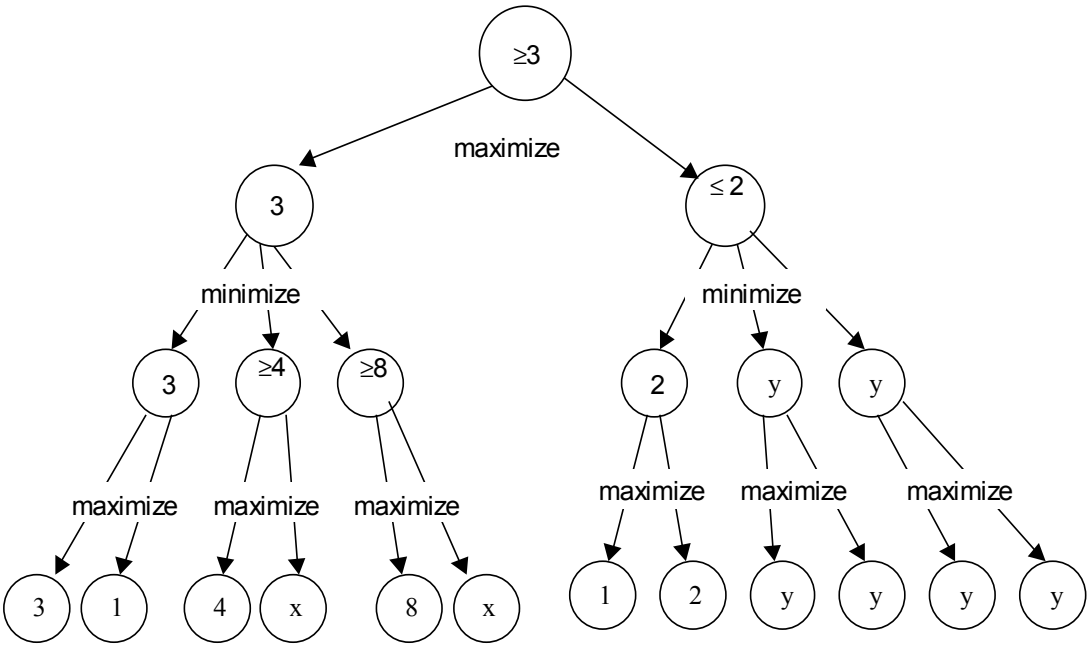

Figure 7. Principles of the alphabeta algorithm

4: Implementation aspects

Implementation of the board

Looking at the gameboard, one can see three axis of symmetry at 60º angle to each other. To make a compact implementation of the board a parallelogram was superimposed over the gameboard so that its horizontal and the diagonal lines coincide with lines on the gameboard, see Figure 8. This parallelogram is used to create a 17x17 matrix representing the board. A very important characteristic of this representation is its regularity; i.e. all lines have the same length (17). This makes the program simpler and a faster. To further gain speed the 17x17 matrix is stored as a 289-element vector.

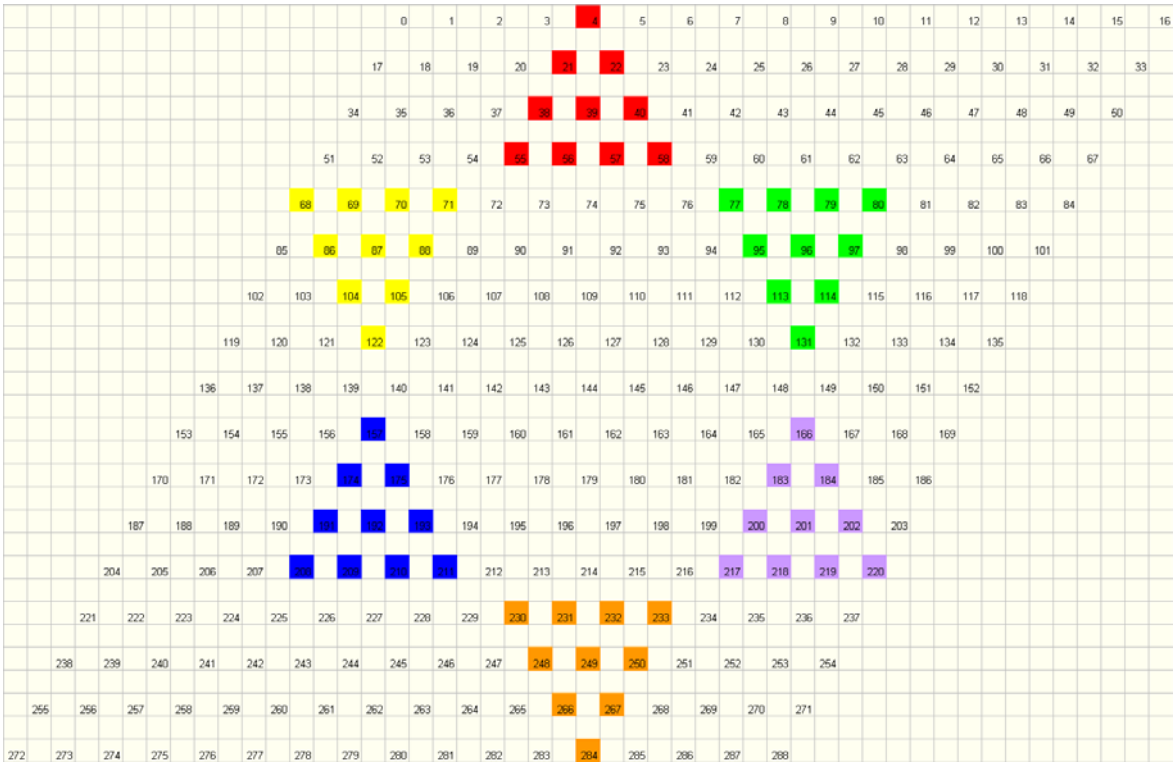

Figure 8. The implementation of the board

The elements in the matrix are numbered from the upper left corner of figure 8; thus, the computer's marbles start at low numbers and the player's at high numbers.

Each element in the vector or matrix can have one of eight values: outside (the board), empty, or a color (red, blue, green, orange, yellow or purple).

Finding a legal move

To make sure only legal moves are made the program checks all possible moves by the marble that is to be moved. To find them the program makes a series of jumps from the starting point. First it checks if the marble can jump over one marble, then if it can make one more jump from that position, until all possible jumps have been tried. Any visited hole is marked, so that it will not be visited again, starting an infinite recursion. Finally, the program checks to see if the marble can roll to a hole next to it. The marking of visited holes serves two purposes. It also contains the end result of the investigation, since the holes marked as visited are the ones it is possible to jump to. The information is stored in a vector of the same

length (289) as the one representing the board. This vector contains booleans. If it is possible to move to a hole on the board then that element in the vector is true. If the move the player chooses is among the possible moves, it is executed and the turn goes to the computer. Figure 9 shows an example of different legal moves for one red marble.

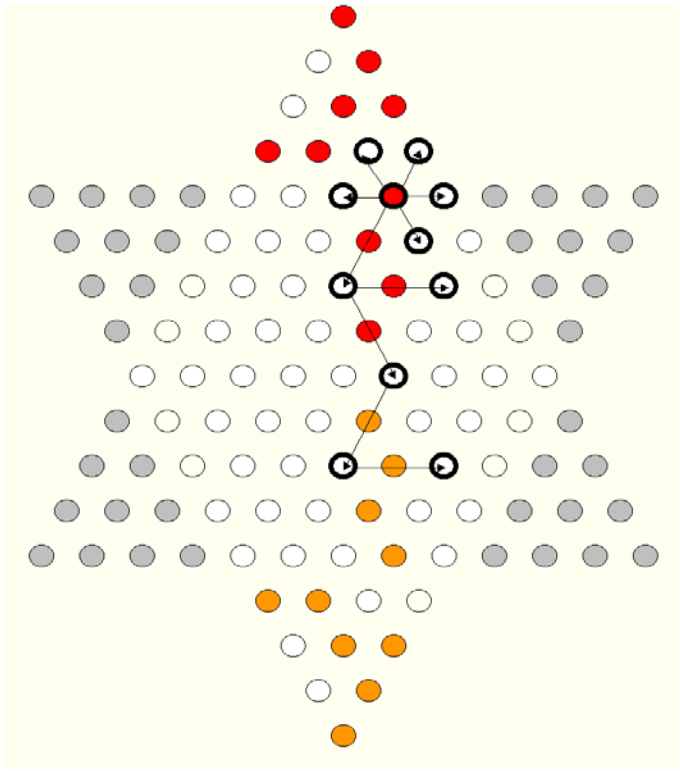

Figure 9. All possible moves by red ringed marble.

The computer obviously has to be able to compare all different moves from all the computer's marbles at the same time. The algorithm outlined above is used for all of the computer's marbles in turn to form a matrix of possible moves. Each column in this matrix represents all possible moves for one marble. All moves in the matrix are then tried one by one to find the best move.

End of game detection

After either the computer or the player has made a move, the board is checked to see if the game has been won. This is done by checking if all holes in the red and the orange star point are filled with marbles of the opponent's color. The same check also is made in the evaluation function at each node in the search tree, so that the algorithms can stop searching down the branch if the game is won or lost.

5: Evaluation

A simple evaluation function

The object of the game is to move all marbles to the opposite side of the board from where they started. The further down the board the marbles are, the higher the value of the situation. Of course, this also applies to the opponent's marbles; the closer to the starting point they are the better the situation for the player.

The above statement was used in the first evaluation function tried. This was done by adding together the row number of both players' marbles' positions. Since the computer starts with its marbles on low numbers, it will try to maximize this number, and the player will try to minimize it.

To make sure to computer knows the goal, a large number is added if all marbles have moved into the goal. In the same way if the player has reached his goal, a large number is subtracted. The number added or subtracted is different depending on how far down the search tree it is. A winning move in an earlier turn should obviously be valued higher than one further on.

Evaluation using a curved board model

The original evaluation function made the marbles move towards the other end of the board, but it did not do anything about their horizontal placement. The marble positions in the horizontal middle of the board should be valued higher by the evaluation function than the positions at the horizontal edges. The reason is that there are more paths to make long jumps in the middle of the board.

The evaluation function used earlier can be visualized as a flat board with a constant upward slope, which is to be climbed by the computer's marbles. If the computer moved a marble one row towards its opponent, the evaluation number grew with a fixed amount independent of the horizontal movement. Using this slope analogy, the new evaluation function makes the board curved in the horizontal plane. Instead of flat slope, it can then be visualized as a ridge. (Figure 10)

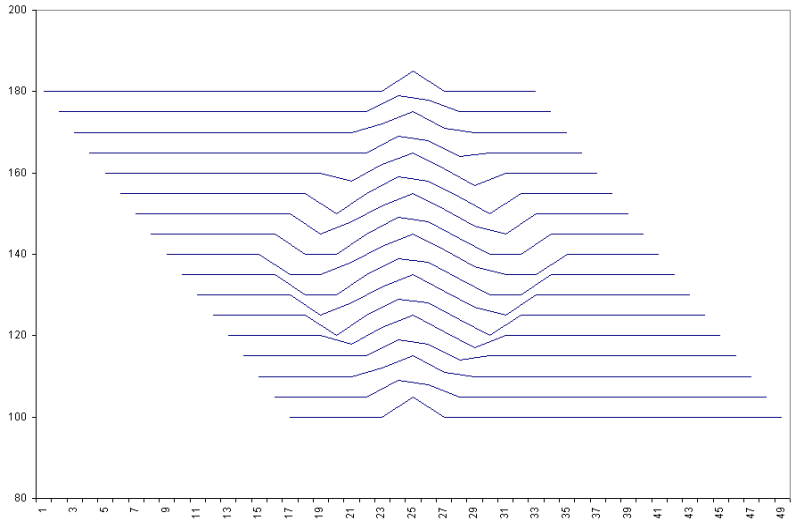

Figure 10. Curved model of board for computer. The apparent upward slope at the edges is in fact outside the board and does not influence the evaluation function.

The curvature of the board is different for the player, since he is trying to minimize the value of the position. Instead of an upward slope and a ridge, it can be viewed as a downward pipe (figure 11).

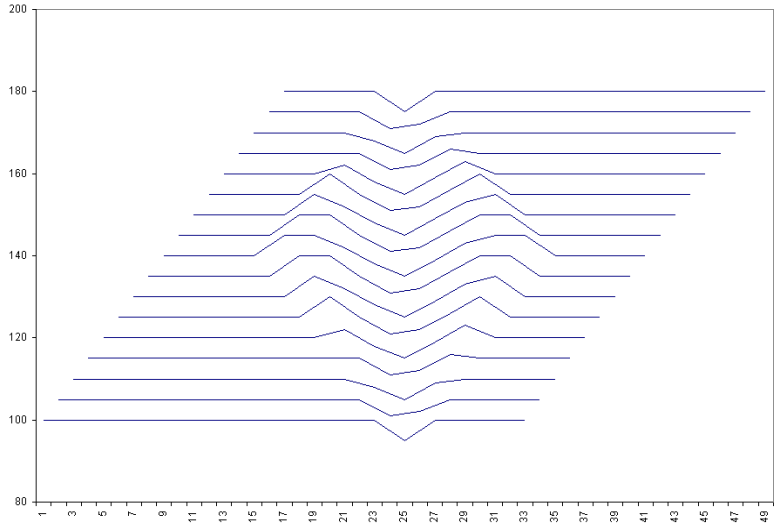

Figure 11. Curved model of board for player. The apparent downward slope at the edges is in fact outside the board and does not influence the evaluation function.

To create a fast implementation of the curved model the evaluation function looks up the value of each marble's position in the vectors shown graphically above and then adds those numbers together.

Incremental evaluation

Using a curved board made the computer play better. To make it also faster an incremental evaluation function was tried. Starting by computing the ordinary evaluation function once, the computer further down in the search-tree only computes the difference in evaluation value between the starting point and the end point of the move. This value is then added to the evaluation number of the situation one level up in the tree. Since the search algorithm is the same and the evaluation number of each situation will be the same, the same move is made and only the elapsed time is different. This evaluation function is introduced in *Alphabeta 6*, which is based on *Alphabeta 4* using this evaluation function as the only change. Table 2 shows how much faster *Alphabeta 6* is.

Table 2. Comparison between elapsed time for *Alphabeta 4* and *Alphabeta 6* with the midgame testboard

Ply	Number of nodes	*Alphabeta 4*: time (ms)	*Alphabeta 6*: time (ms)
1	57	15	1
2	279	19	12
3	3726	196	110
4	24416	1733	1156
5	399452	24098	15051
6	4689510	291014	178304

Left-behind marbles

Using a simple evaluation function, the computer often leaves one or two marbles behind. At the end of the game those marbles will take several additional turns to reach the goal, because their possibilities to make long jumps will be limited. This can not be captured by the search algorithm because the limited search-depth causes a horizon effect. Any event occurring outside the search tree, i.e. beyond the horizon, is not accounted for. One way to take care of this is to increase the search depth. This, however, is not realistic because of the exponential growth of search tree as a function of its depth. Instead, the algorithm *Adaptive & last marble* uses a new evaluation function that makes moving one of the last two marbles more profitable.

This new evaluation function is a modification of the incremental evaluation function. At the nodes in the search tree where one of the two last marbles has moved, the evaluation function calculates an additional term. This term involves the positions of the last four marbles and the number of rows the marble moved forward. These numbers are used according to the following equation:

```
Term = min (number of rows the last marble is behind the second
last marble, number of rows the last marble moved forward) + min
(number of rows the last marble is behind third last marble,
number of rows the last marble moved forward) + min (number of
rows the last marble is behind forth last marble, number of rows
the last marble moved forward) + min (number of rows the second
last marble is behind the third last marble, number of rows the
second last marble moved forward) + min (number of rows the second
last marble is behind the forth last marble, number of rows the
second last marble moved forward)
```

where the *min*-function has the value of the smallest of its input parameters.

This term is then added to the result of the evaluation function. Despite its complicated first-sight appearance, the additional term has a general structure that easily lends itself to be extended to include any number of trailing marbles. It is also simple enough not to significantly slow down the program. With this extension to the evaluation function, the computer becomes very hard to beat for a human player.

6: Search algorithms

The minimax-algorithm

The first search algorithm written for this program was an implementation of the minimax algorithm, where the search tree is completely traversed. The search starts with the marble resting in the hole with the highest number. Using the matrix containing all possible moves the move to the hole with the highest number is tried first, followed by all moves by that marble from highest to lowest hole number. All moves by all marbles are tried in turn, and the same order is used for the opponent's moves.

If the evaluation number is larger than 10000 (the computer has won) or smaller than -1000 (the player has won) the search down that branch is not continued. This is to make sure the computer is capable of identifying when it has won the game. The search in other branches, however, is continued in order to be able to find ways to win the game in fewer turns. This is used not only in the minimax algorithm, but in all algorithms in the program.

The alphabeta algorithm with different search orders

The benefit of the alphabeta-algorithm greatly depends on the order in which the moves are searched. One extreme would be to order the tree such that every move has to be checked. Nothing is then gained compared to the minimax algorithm. If the search order is carefully thought out, however, the gain in speed is orders of magnitude compared to minimax, for a chinese checkers search tree of typical depth.

The first implementation of the alphabeta-algorithm is called *Alphabeta*. The computer recursively searches all the possible moves for one marble at a time, starting with the marble resting in the hole with the highest number. The possible moves are tried with the move ending in the hole with the highest number tried first, in the same way as in the minimax algorithm. Figure 12 gives an example of this search order. Since it searches from higher to lower numbers for both players, this can not be ideal.

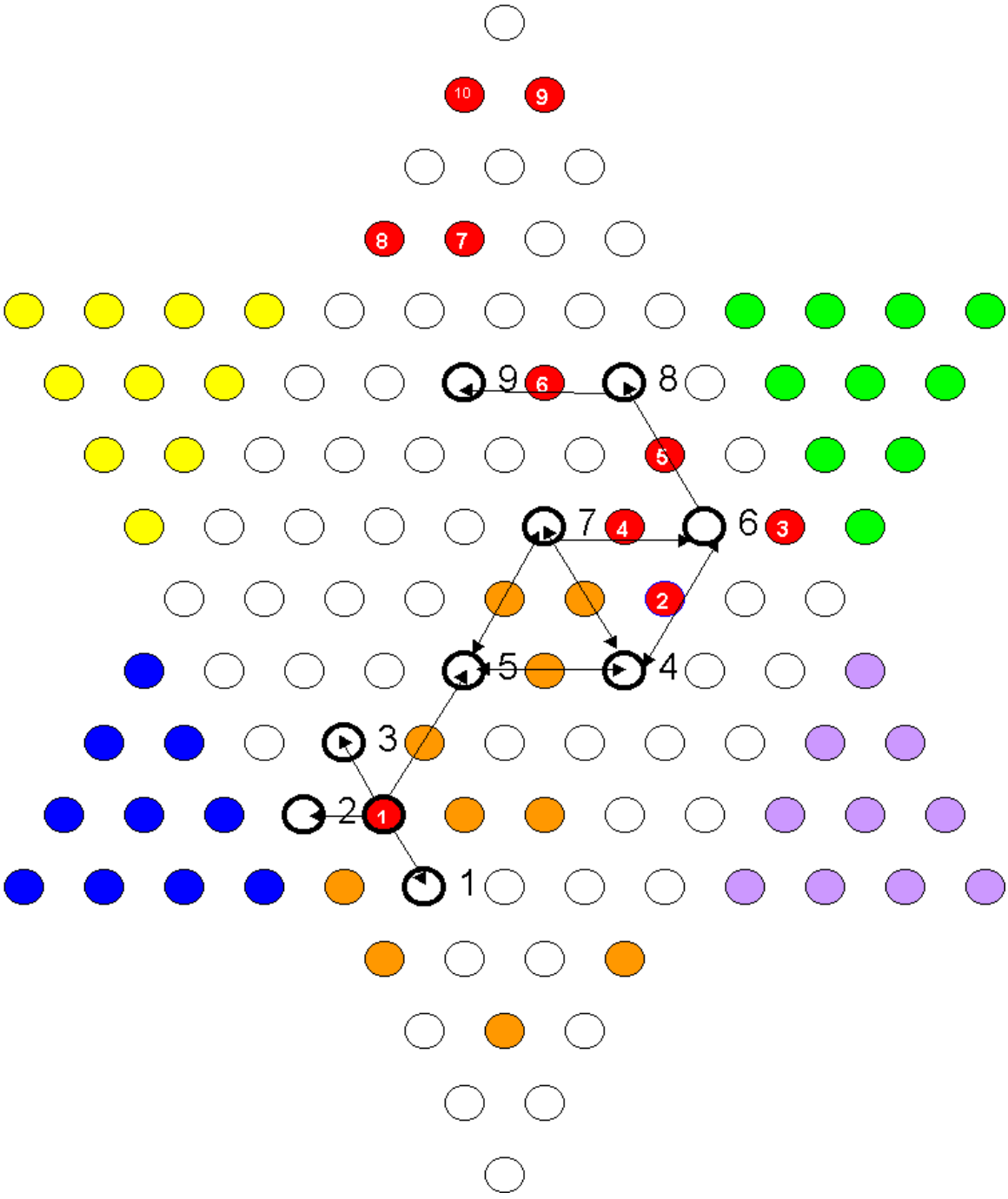

Figure 12. The red marbles are numbered in the order they are searched in the *alphabeta* and *minimax* algorithms, and all possible moves for the first searched red marble are numbered in the order they are searched.

Searching from different directions for different players, still in number order, was then tried. *Alphabeta 2* searches all moves for each marble from high numbers to low numbers for the computer and from low numbers to high numbers for the player. *Alphabeta 2_I* reverses the order the holes are searched for both the player and the computer.

Alphabeta 3 compares all moves for all marbles to a certain hole, for all holes in number order. The order is different for the player and for the computer. *Alphabeta 3_I* is the reverse of *Alphabeta 3*.

Comparing all these algorithms gave a mixed result, as no direction could be shown best in all situations.

Making several jumps in a row results in a large difference between the number of the hole the marble starts from and the hole it stops in. *Alphabeta 4* orders the moves so that the largest jumps towards the goal are searched first. This gives a much better result than the earlier algorithms.

After changing the evaluation-function (see previous chapter) the search-tree in *Alphabeta 4* was modified to search the most central holes first, making *Alphabeta 5*, but this made little or no difference on the number of searched moves.

The results of the above experiments are shown in figures 13 and 14. As you can see, the search tree grows exponentially with the number of ply.

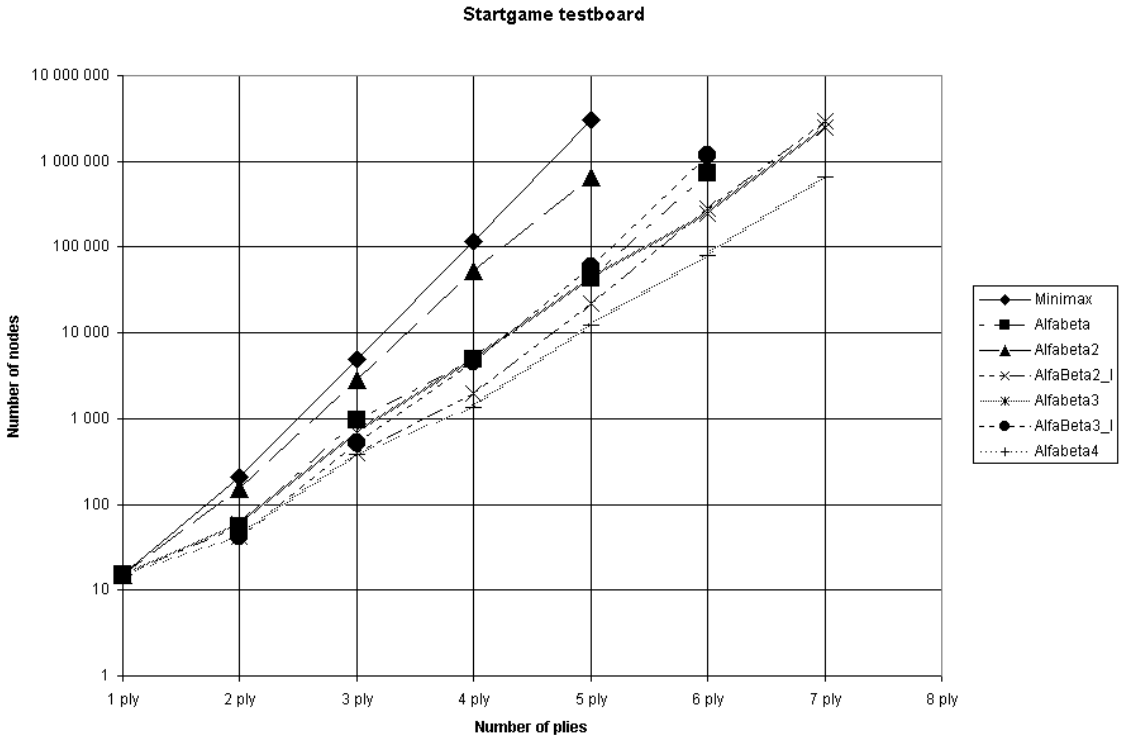

Figure 13. Comparison between the number of nodes searched by different algorithms at different number of ply with the startgame testboard.

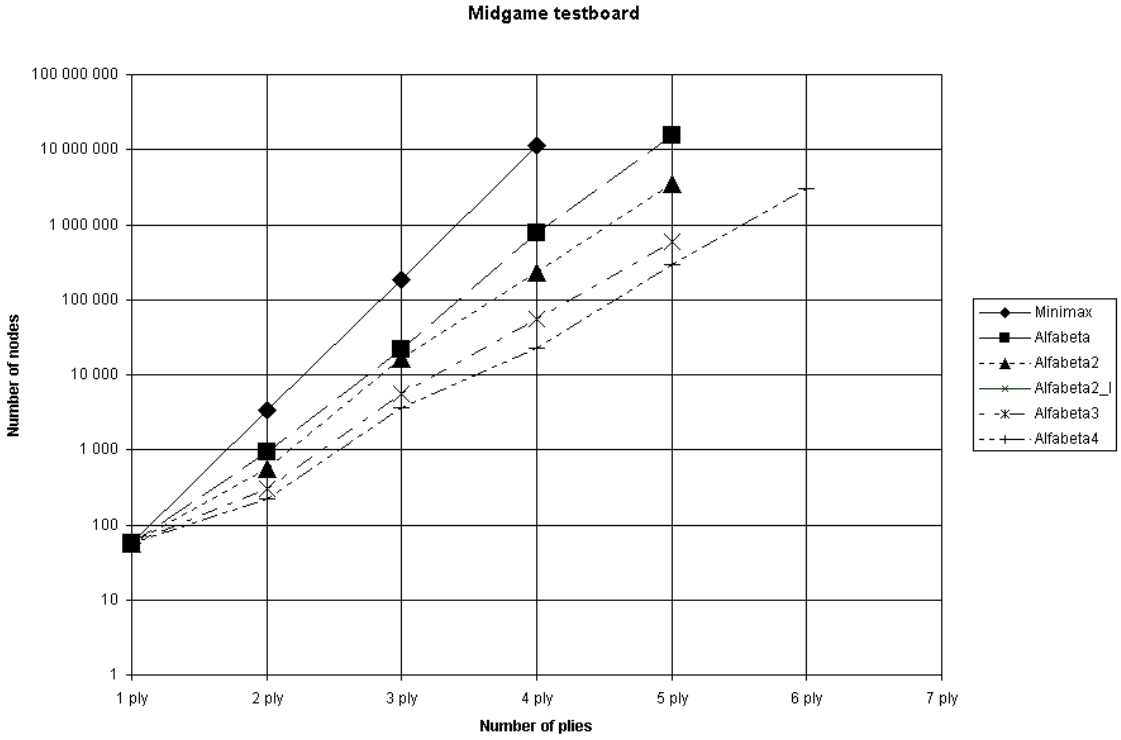

Figure 14. Comparison between the number of nodes searched by different algorithms at different plies with the midgame testboard.

Beginning and endplay

In the beginning of the game, it is possible for the players to move their marbles independent of each other. Since the marbles do not interact, it is possible to save some time and computation by only looking at ones own marbles the first few turns. This is the reasoning behind the *Begin & End Game* - algorithm, which uses a full-length search of only the computer's moves through the search tree. Since only the computer's marbles are looked at, the evaluation number is always maximized. Looking only at one player's marbles two turns ahead is comparable to looking at 1st players turn, 2nd players turn, and 1st players turn. This means one has to compare the efficiency of the *Begin & End Game*-algorithm at two ply with *Alphabeta 6* at three ply. This is done in table 3 for some different number of plies, for the first move of the game.

The algorithm results in the same move being made after looking forward the same number of turns for the computer, supporting the theory that it is possible to look at just one set of marbles when they are far from each other. The players are far apart when the closest marbles of different color is more than two rows in the matrix apart, two or fewer ply down in the search tree. If the marbles come into contact further down the tree, it does not influence the algorithm. If they come into contact sooner, the algorithm is interrupted and resumed with the *Alphabeta 6* algorithm.

Table 3 compares the *Alphabeta 6* and the *Begin & End Game* algorithms.

Table 3: Comparison between the *Alfabeta 6* and *Begin & End Game* algorithms

Sequence of moves	Ply	Number of nodes	Time (ms)
Begin & End Game			
Computer	1	15	0
Computer, computer	2	347	2
Computer, computer, computer	3	9 219	140
Comp, comp, comp, comp	4	278 693	3 954
Alphabeta 6			
Computer	1	15	0
Computer, player, computer	3	597	19
Comp, play, comp, play, comp	5	18 086	951
Comp, play, comp, play, comp, play, comp	7	1 920 951	93 177

Adaptive search-depth algorithm

On different turns, it is possible to choose from a different number of moves. To make the best possible use of the computing time it is therefore advantageous to look forward a different number of ply depending on the size of the search-tree. This is done by interrupting the search if the number of checked nodes is larger than a previously decided number. The search is then repeated with a one ply shallower search-tree. If the number of nodes in the search-tree is less than 10 percent of the maximum allowed number of nodes, the search is repeated with one ply deeper search-tree, unless the number of ply already has been lowered.

This means that the computer can take advantage of the fact that searching a certain number of nodes takes the same time independent of the number of ply in the search tree. In a more complex situation where the tree grows fast, the search can be done in a reasonable time by searching fewer plies; in a situation with few choices, more ply can be evaluated. This also means that the player has an easy way of limiting the time he has to wait for the computer to make a move.

This type of adaptive search-depth algorithm is used in *Adaptive algorithm*, *Adaptive algorithm 2* and *Adaptive & last marble*. *Adaptive algorithm* uses the *Begin & End Game*-algorithm in the beginning of the game and the *Alphabeta 6*-algorithm when the players marbles are in contact. *Adaptive algorithm 2* uses only *Alphabeta 6*. The reason for making two algorithms was to be able to compare the number of ply used in different situations, and to make it easier to check for errors. The special properties of *Adaptive & last marble* has been described in chapter 5.

7. Future developments

The next step in increasing the search depth and/or speed would be to apply razoring i.e. cutting of branches from the search tree based on assumptions rather than certainty. A result is that the branch containing the best move might be cut off by mistake. The assumptions must therefore be carefully formulated. The more branches that are cut off, the higher the gain in speed, but also the risk of neglecting the best moves.

Another possible development is to expand the program to include more than two players. These can be controlled by either the computer or by humans. This will significantly increase the size of the search tree.

The user interface can be further improved by adding the possibility to change colour, undo moves and save unfinished games. The appearance can be improved by making the board and marbles look three-dimensional. Sounds can also be added.

Conclusions

It is difficult to find references describing implementation of chinese checkers playing programs. The goal was therefore to find suitable algorithms and to create a program that would play good chinese checkers with a reasonable amount of wait time while the computer makes its move. This goal has been met since the program can beat most human players with a waiting time of a few seconds per move on a standard PC.

To meet this goal several different evaluation functions and search algorithms have been tried. The final version of the program is quite advanced involving both adaptive search depth and a fast, incremental evaluation function.

The evaluation function encourages play in the horizontal middle of the board, since in general there are more paths toward the goal there. To avoid trailing marbles causing extra moves in the endplay, the evaluation function rewards moving one of the last marbles. To gain speed the evaluation function uses the result of the previous move and adds an increment to it, instead of calculating the evaluation function from scratch. This is called incremental evaluation.

The search algorithm is based on the alpha-beta algorithm. In this algorithm the ordering of the moves in the search tree is critical to speed. In the final version the moves were ordered according to vertical distance. The complexity of the game is much larger in the middle of the game compared to the beginning or the end. To handle this an adaptive search depth was introduced. Furthermore, a different search algorithm disregarding the opponents marbles was used in the beginning and end of the game.

References:

[1] Game Rule Cabinet http://www.iserv.net/~central/GAMES/checkers/CHINCHEC.htm
The rules of chinese checkers as copied from those accompanying the game when it is bought in a shop.

[2] Birmingham, John and Kent, Peter *Tree-Searching and Tree-Pruning Techniques, COMPUTER CHESS COMPENDIUM*

[3] Eppstein, David, Dept. Information & Computer Science, UC Irvine, http://www.ics.uci.edu/~eppstein/180a/w99.html
Lecture notes for ICS 180, Winter 1997: Strategy and board game programming

[4] Marsland, T.A, *A review of game-tree pruning,* ICCA journal, Vol.9, No.1, March 1986