University of Alberta

Library Release Form

Name of Author: Nathan Bullock

Title of Thesis: Domineering: Solving Large Combinatorial Search Spaces

Degree: Master of Science

Year this Degree Granted: 2002

Nathan Bullock
108 10820 78 Ave
Edmonton, Alberta
Canada T6E 1P8

Date: ________________

When we walk with gratitude, we do not walk with arrogance and conceit and egotism, but rather with a spirit of thanksgiving that is becoming to us and will bless our lives. We should all be thankful to the Almighty for His wonderful blessings upon us.

Gordon B. Hinckley

University of Alberta

Domineering: Solving Large Combinatorial Search Spaces

by

Nathan Bullock

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2002

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Domineering: Solving Large Combinatorial Search Spaces** submitted by Nathan Bullock in partial fulfillment of the requirements for the degree of **Master of Science**.

Dr. Martin Müller
Co-Supervisor

Dr. Jonathan Schaeffer
Co-Supervisor

Dr. Bruce Sutherland

Dr. T. Anthony Marsland

Date: ____________________

Abstract

Domineering, also known as crosscram, is a perfect-information, two-player game. It is played on a board which consists of a subset of a square lattice. The game is played by two players who take turns placing 2×1 tiles upon the board. One player is only allowed to place a tile in a horizontal orientation and the other is only allowed to place a tile in a vertical orientation. Tiles are not allowed to overlap. The game ends when one player, the loser, is unable to place any more tiles onto the board.

We have created a search program which is able to prove who wins on many different sizes of boards. Some of which no one else has ever been able to solve. The main improvement we have come up with is an evaluation function which is able to statically determine a winner at a shallower point in the search, allowing us to eliminate large portions of the search space. This, along with a few other improvements, has enabled us to solve board positions with just a fraction of the number of nodes which previous solvers have needed.

Acknowledgements

A special thanks to my parents, for their love and guidance. To my eternal companion, Kelli, for her willingness in helping me to achieve my goals. To our children, for their laughter and unique insight into the world around them.

I would like to thank both of my supervisors. Martin Müller for all of his ideas, comments, and revisions. Jonathan Schaeffer for giving me the motivation to keep going when I was ready to quit. I would also like to thank all those who have made thinking so much fun: Ryan Hayward, Lorna Stewart, Stephen Wismath, Rex Forsyth, Kim Francis, and many others. There are so many others who I owe my thanks to. To all of you. Thanks!

Most importantly, for His help, from Whom all truth emanates, I am humbly grateful.

Table of Contents

List of Figures

Chapter 1

Introduction

Domineering, also known as crosscram, is a perfect-information, two-player game. It is played on a board which consists of a subset of a square lattice. The 8×8 board is a popular size used by human players. The game is played by two players who take turns placing 2×1 tiles upon the board. One player is only allowed to place a tile in a horizontal orientation and the other is only allowed to place a tile in a vertical orientation. Tiles are not allowed to overlap. The game ends when one player, the loser, is unable to place any more tiles onto the board.

Domineering was introduced by Göran Andersson around 1973. Since then it has been investigated from both a mathematical and an artificial intelligence point of view.

Mathematicians have examined domineering using an elegant theory called combinatorial game theory [1]. This approach has allowed them to determine exact combinatorial game values for many of the smaller game boards which, because of the power of combinatorial game theory, allows them to determine the combinatorial game value for any set of these boards added together. For example if the value of the 2×2 board is known, then the value of two 2×2 game boards played together can be determined. These results have appeared in a number of books on combinatorial game theory [4, 2].

Artificial intelligence researchers have also taken an active role in looking at the game of domineering and, through the use of state-of-the-art search engines, have been able to determine the game theoretic value of many different sizes of domineering boards [3]. In domineering there exist four possible values for each position:

- A vertical win, denoted by V, meaning that regardless of who goes first vertical will always win.
- A horizontal win, denoted by H, meaning that regardless of who goes first horizontal will always win.
- A first player win, denoted by $1st$, meaning that regardless of who goes first the first player will always win.
- A second player win, denoted by $2nd$, meaning that regardless of who goes first the second player will always win.

Interest in domineering among artificial intelligence researchers has increased as a result of domineering tournaments organized by Professor Elwyn Berlekamp, one of the founders of combinatorial game theory. The first tournament was held at MSRI in Berkeley in July of 1994 [10] with a prize of $500 US and the second was held as part of the IJCAI'97 conference in Nagoya, Japan with a prize of $1000 US.

1.1 Playing Domineering

To refer to a move, we will specify each move by the row and column number of its upper (for vertical) or left square (for horizontal) coordinates. We label the rows top to bottom from 1 to the number of rows on the board, and columns left to right from 1 to the number of columns on the board. Figure 1.1 shows an example of a game on a 4×5 board with vertical moving first.

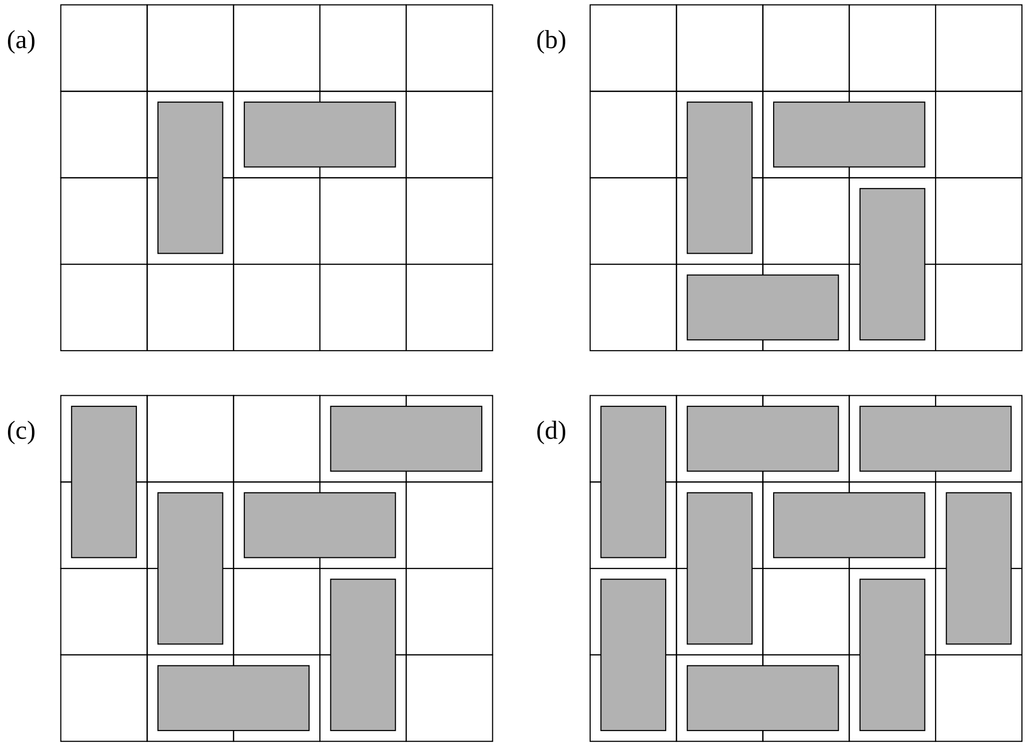

Figure 1.1: Game on a 4×5 board with vertical moving first. (a) First two moves are (2,2) and (2,3). (b) Next two moves are (3,4) and (4,2). (c) Next two moves are (1,1) and (1,4). (d) Last moves are (3,1), (1,2), and (2,5). Horizontal can't move; vertical wins.

1.2 DOMI

The first published program to solve domineering positions was the program DOMI authored by Breuker et al. [3]. This became our catalyst for writing an improved domineering solver.

The major strengths of DOMI are its use of transposition tables and its move ordering heuristic. Both of which were instrumental in enabling it to solve domineering board positions as efficiently as it did.

Some of the key results which DOMI calculated were the $m \times n$ boards where $2 \leq m \leq 8$ and $m \leq n \leq 9$ (see Figure 1.2). Note the exponential growth of the size of the proof trees for the increasing board sizes.

Since the publishing of these results, DOMI has been improved to the point

Board Size	Result	Nodes	Board Size	Result	Nodes
2×2	1	1	4×7	V	1984
2×3	1	2	4×8	H	12,024
2×4	H	13	4×9	V	45,314
2×5	V	15	5×5	2	604
2×6	1	14	5×6	H	1,500
2×7	1	17	5×7	H	13,584
2×8	H	67	5×8	H	30,348
2×9	V	126	5×9	H	177,324
3×3	1	1	6×6	1	17,232
3×4	H	10	6×7	V	302,259
3×5	H	19	6×8	H	3,362,436
3×6	H	40	6×9	V	18,421,911
3×7	H	77	7×7	1	408,260
3×8	H	74	7×8	H	12,339,876
3×9	H	99	7×9	H	320,589,295
4×4	1	40	8×8	1	441,990,070
4×5	V	87	8×9	V	70,918,073,509
4×6	1	1327			

Figure 1.2: Game-theoretic values for various sizes of domineering boards and the number of nodes DOMI needed to calculate the values [3].

that it can now solve 8×9 domineering in approximately 3 billion nodes in size and 9×9 domineering with 25 billion nodes [9].

1.3 Extending Results To Larger Boards

Using search techniques for solving domineering boards obviously limits the size of the largest board that can be solved. Lachmann et al. came up with a number of different rules which enable the extending of the results on smaller boards to determine the game theoretic values of larger boards [6]. We will denote the value of an $m \times n$ board as $[m \times n]$.

The first observation is that on $2 \times n$ boards, vertical, if vertical has the first move, is able to split the board into two different sizes of boards, each of which still have 2 rows. Then by looking at the game theoretic values for each of these smaller boards we can determine some information about the larger board. For example we know $[2 \times 5] = V$, therefore if vertical has the first move on a 2×11 board they can play across the sixth column splitting the board into two 2×5 boards, of which we know they can win in both. Note that this doesn't say anything about $2 \times n$ boards when horizontal has the first turn.

From this observation they came up with the rule shown in Figure 1.3 which can be used to determine the game theoretic value of a $2 \times (g + h + 1)$ domineering board.

$[2 \times h]$ \ $[2 \times g]$	2nd	V
2nd	1st or V	1st or V
V	1st or V	1st or V

Figure 1.3: Matrix which allows us to determine $[2 \times (g + h + 1)]$ when we know $[2 \times g]$ and $[2 \times h]$.

The second observation is much more general. On any board horizontal can avoid placing a tile across two adjacent columns, effectively playing an $m \times (g+h)$ game as the sum of an $m \times g$ and an $m \times h$ game. This observation is much more general since it works for any size of board. From this observation they came up with the rule shown in Figure 1.4 which can be used to determine the game theoretic value of an $m \times (g + h)$ domineering board. For example since $[5 \times 2] = H$ and $[5 \times 5] = 2$ we can determine that $[5 \times (5 + 2)] = H$.

$[m \times h]$ \ $[m \times g]$	1st	2nd	H
1st		1st or H	1st or H
2nd	1st or H	2nd or H	H
H	1st or H	H	H

Figure 1.4: Matrix which allows us to determine $[m \times (g + h)]$ when we know $[m \times g]$ and $[m \times h]$.

One more observation they made was that since neither player can have an advantage on a square board, all $n \times n$ boards must have a value of either 1 or 2.

Through the use of Breuker et al.'s results, a theoretical result for the 2×31 board, and the previous observations, Lachmann et al. were able to determine the winner for all boards with 2, 3, 5, and 7 rows. Also they were able to determine the winners for all boards with 4, 9, and 11 rows, except for a finite number of smaller boards. Through the use of our domineering solver, *Obsequi*, described in this thesis, we have been able to determine the game theoretic values for a few previously unsolved game boards, notably 4×19, 4×21, 6×14, 8×10, and 10×10, which has filled in a number of the holes in Lachmann et al.'s original results. See Figure 6.5 for an updated chart of those boards for which we can calculate the game theoretic value.

1.4 Motivation

Game and puzzle programs search large combinatorial search spaces. Research is constantly looking for ways to make these search programs more efficient. Possible methods to do this are large databases of endgame positions, more efficient search algorithms, specialized local searches, or game-specific static evaluations.

Domineering is a nice game to use for exploring heuristic search. It has simple rules, nice mathematical properties, a challenging search space, and is amenable to combinatorial game theory analysis. As a measure of the progress we have made, 2 years ago 9×9 domineering had not yet been solved, today we are not only able to solve 9×9 domineering in a matter of hours but we can solve 10×10 domineering, which is at least a thousand times more difficult.

1.5 Contributions of Thesis

In this thesis we will show that one effective way to reduce the size of the proof trees, for different sizes of domineering boards, is to come up with a more advanced evaluation function. This improved evaluation function will be able to determine the winner of a game at a shallower point in the search tree, allowing us to prune a potentially large number of positions from the search.

This substantial pruning of positions from the search tree has enabled us to solve the standard domineering board, 8×8, in a matter of seconds where previously it had taken hours. As well it has enabled us to solve larger domineering boards than have ever before been solved, most notably the 10×10 board.

In this thesis we will also take a look at search enhancements, each of which incrementally helps to improve our solver. These include an improved move ordering scheme and an improved replacement scheme for our solver's transposition table.

One more noteworthy contribution is the source code to *Obsequi*, our domineering solver. We would like to allow other researchers to be able to study the game of domineering without having to spend weeks or months writing a program which is able to run with the speed and efficiency necessary to solve the game on larger boards. The code is publicly available at `www.cs.ualberta.ca/~games/domineering`.

Chapter 2

Alpha-Beta Search of Minimax Game Trees

The search space of a two player game can be looked at as a minimax tree where one player (Max) is always trying to play the move which has the maximum value associated with it and the other (Min) is trying to play the move which has the minimum value associated with it (or in other words minimizing Max's score). See Figure 2.1 for an example of a minimax tree.

Each parent node takes the maximum or minimum of its children. For example, in Figure 2.1, the player, Max, at the top node of the tree has a choice of obtaining a 4 or a 5, and obviously chooses the 5. The minimax algorithm propagates (or backs up) values from the leaf nodes of the tree to the root, alternating between maximizing and minimizing.

Note that *interior* nodes, or those nodes which have children, do not have an assigned value but gain their values from their children, while *leaf* nodes, those nodes with no children, must have a specific value associated with them.

2.1 Alpha-Beta Algorithm

The entire minimax search tree does not need to be built in order to determine the value at the root of the tree. Some of the nodes in the tree can be proven to be irrelevant. For example, assume we have a minimax tree with a root

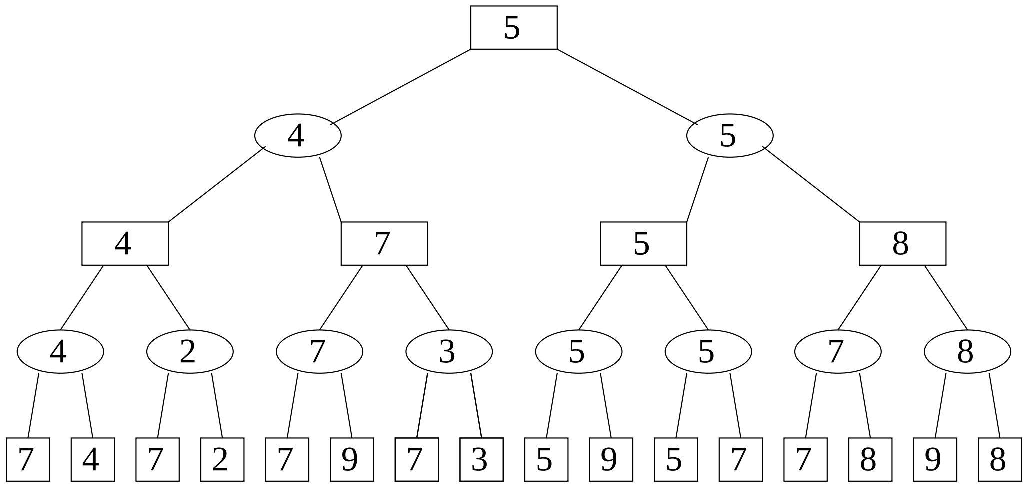

Figure 2.1: Minimax Tree. The boxes represent the maximizing player, Max, while the ellipses represent the minimizing player, Min. The lines represent the possible move choices that could be made.

(Max) node which has two children, A and B, and have already determined A has a value of 5 and are currently trying to determine the value of B. If we determine any of B's children have a value less than or equal to 5 then we can prune the rest of B's children from the tree since, because B is a Min node, it is now guaranteed to backup a value less than or equal to 5 which will not affect the value of the root of the tree.

The alpha-beta search algorithm is a depth first traversal of a minimax tree which maintains, for each subtree, lower (alpha) and upper (beta) bounds on the range of values which can affect the value of the root of the tree. When the value of a subtree is determined to be outside of this range, the algorithm is then able to prune the subtree without further search, since it has been proven to not affect the value of the root of the tree. For the basic alpha-beta algorithm see Figure 2.2.

We can use the alpha-beta algorithm on the same search space that we looked at in Figure 2.1. In doing this we see that we are able to still determine the correct value of the root of the tree while finding cutoffs which allow us

```
int
AlphaBeta(Position p, int alpha, int beta)
{
  Children child[];    /* Array for children of position. */
  int best = MIN_INT;  /* Smallest possible value. */
  int result, number, i;

  /* Generate children of current position. */
  number = generate_children(p, child);

  /* No children therefore leaf node: return value of leaf. */
  if(number == 0) return evaluate(p);

  /* Iterate through all children. */
  for(i = 0; i < number; i++) {

    /* Recurse using nega-max variant of alpha-beta. */
    result = -AlphaBeta(child[i], -beta, -alpha);

    /* Keep best result so far. */
    best = MAX(result, best);

    /* Prune if out of range. */
    if(best >= beta) return best;

    /* Possibly need to increase lower bound. */
    alpha = MAX(alpha, best);
  }

  return best;
}
```

Figure 2.2: General Alpha-Beta Algorithm. Nega-max is the idea that if we swap alpha and beta and negate them as well as the return value, we can treat the Min nodes identically to the Max nodes.

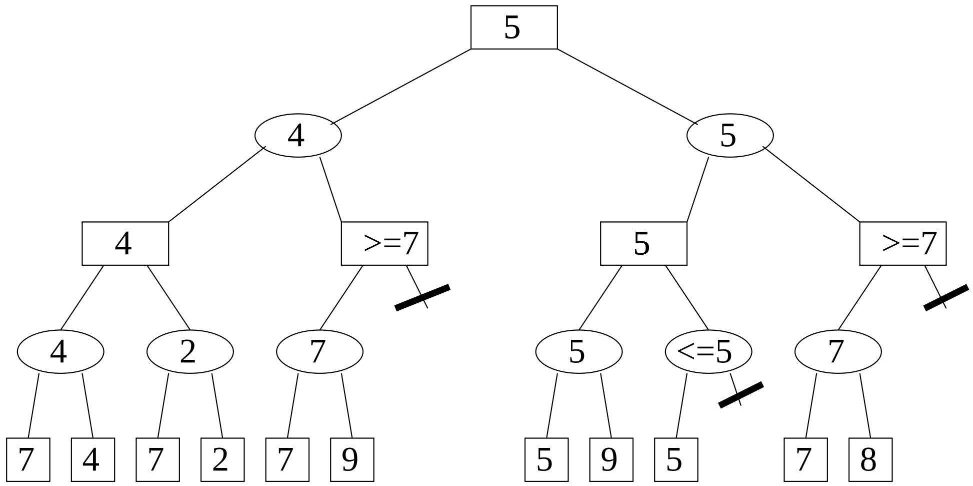

Figure 2.3: Alpha-Beta Pruning. Dark lines indicate pruning of the tree assuming we are traversing the tree from left to right.

to reduce the number of nodes which we have to look at in the tree (see Figure 2.3).

2.2 Move Ordering

One important thing to note about the alpha-beta pruning algorithm is that the number of nodes which are pruned greatly depends on the ordering of the nodes at each level of the tree. In fact in the worst possible case alpha-beta pruning will not find any cutoffs and will have to search just as many nodes as a naive minimax algorithm. At the same time if we are able to find the best possible ordering of the nodes, this being that Max always generates the child with the largest value first while Min always generates the child with the minimum value first, we will be able to find the largest possible number of cutoffs (see Figure 2.4). To be more precise, assume we have a search space with a branching factor of b and a depth of d. The simple minimax algorithm will generate a tree with approximately b^d nodes, while the alpha-beta algorithm will, in the best case, generate a tree with approximately $b^{\frac{d}{2}}$ nodes. (Note that

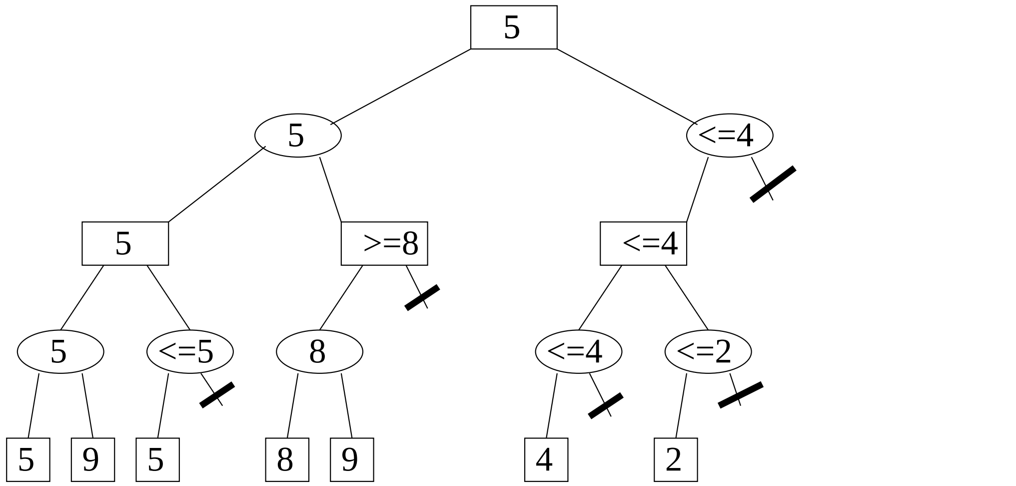

Figure 2.4: Alpha-Beta Pruning with Optimal Move Ordering. Dark lines indicate pruning of the tree assuming we are traversing the tree from left to right.

through the use of transposition tables the number of positions searched can be reduced further.)

Since we don't know the value of an interior node until the subtree of this node has been examined, a heuristic needs be employed to predict the best ordering of the children of a node. It is easy to see that it is very desirable to have a heuristic which is able to accurately order the children of a node in relation to what their values really are.

2.3 Enhancements

There are many other methods that can be used in creating further reductions in the number of nodes which need to be examined in a game tree. Some of these are iterative deepening, transposition tables, refutation tables, minimal window, aspiration search, and the killer heuristic. (Jonathan Schaeffer gives an explanation of these different techniques and compares them in [8].)

Each of these methods work by either improving the ordering of the interior nodes, reducing the size of the alpha-beta bounds which increases the chance of pruning, or re-using information in the event that we see an identical subtree later in the search.

Chapter 3

Determining Who Wins

The number of nodes in a typical search tree grows exponentially with the depth of the tree. Given this fact it is easy to see that it is desirable to determine the winner in a game at the shallowest point possible, since this will save us from searching a large portion of the tree.

In *Obsequi* we determine a lower bound on the number of moves that a player, α, could make, given the current board position and using a certain strategy. Then we determine an upper bound on the number of moves that α's opponent, β, could make, given the strategy that α used to get their lower bound. If α's lower bound is greater than (or equal to, depending on who just moved) β's upper bound then we can conclude that α wins. Similarly we can also do this for β and determine if they win. In this chapter we let α denote the player for whom we are trying to determine a lower bound, or in other words the player for whom we are trying to determine if they win given a specific board position. We refer to α's opposition as β.

3.1 Definitions

We define an **unoccupied** square as one which is not currently covered by a tile. Conversely an **occupied** square is one which is covered by a tile or which is not part of the board.

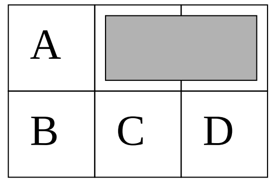

Figure 3.1: An example of a 2×3 domineering board. The shaded 1×2 rectangle is a tile which was placed by the horizontal player.

An **available** square is a square which a player can place one of their tiles across. A square is **unavailable** for a player if its borders are occupied in such a way that there is no way for that player to place a tile on that square. In Figure 3.1 the square marked by an A is unavailable for the horizontal player while the squares B, C, and D are all available.

We can also look at squares from the perspective of where a player's opponent can play. A **protected** square is one which the opponent is unable to play on; from the opponent's perspective this square would be considered unavailable. An **unprotected** square is one which the opponent could place a tile upon; the opponent would consider this square available. In Figure 3.1 the square marked by an A is protected for the vertical player, while the squares B, C, and D are all unprotected for the vertical player.

3.2 Board Properties

In order to determine a lower bound on the number of moves a player can make, we define three types of board properties which we use to calculate this bound. These three properties are: safe areas, vulnerable areas, and protective areas. Note that these properties are determined seperately for the horizontal and vertical player.

A **vulnerable area** for a player, α, is a pair of adjacent squares where α can place a tile (see Figure 3.2(a)).

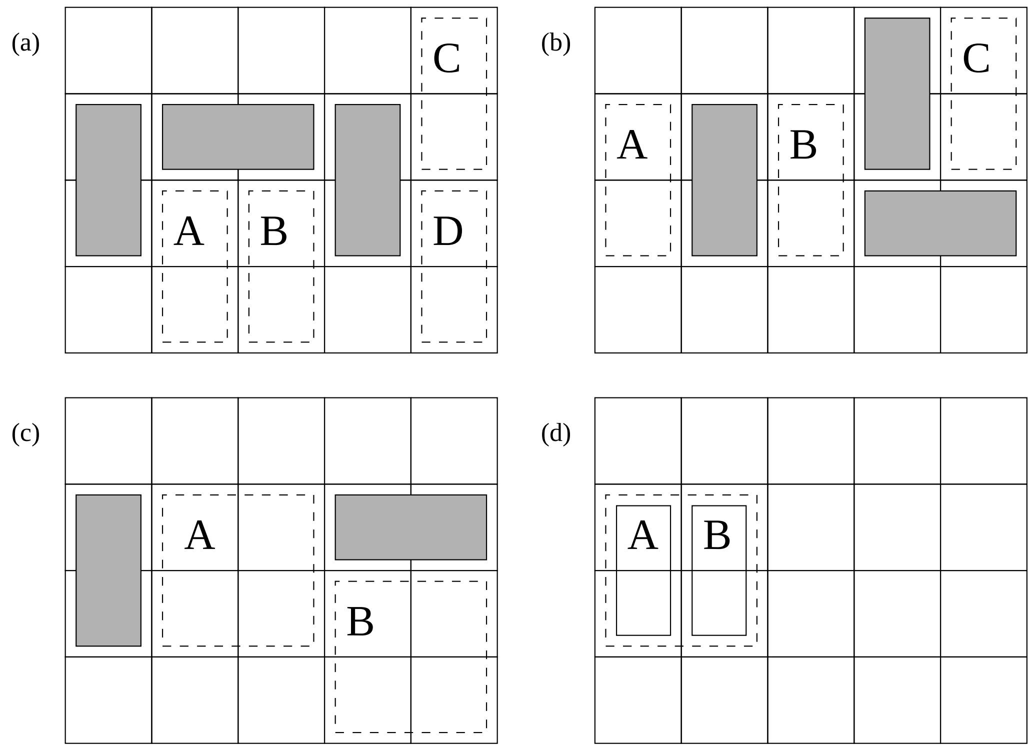

Figure 3.2: Board properties from the perspective of the vertical player. (a) A, B, C, and D are vulnerable areas. (b) A, B, and C are safe areas. (c) A and B are both protective areas. (d) The square outlined with a dotted line is a protective area, A is where a safe area would be created if vertical played a tile at position B.

A **safe area** is a vulnerable area where both squares are protected squares for α, or in other words unavailable to β (see Figure 3.2(b))[1]. It is easy to see that it is impossible for β to place a tile which would overlap with one of α's safe areas. The concept of a safe area (or safe move) was also used in [3].

A **protective area** for a player, α, is a 2×2 unoccupied region of the board where one of the sides is bordered by occupied squares (or the edge of the board) in such a way that α is able to place a tile, completely inside this area, in such a way that the other 2 squares, not covered by α's tile, form a safe area for α afterwards (see Figures 3.2(c) and 3.2(d)).

[1]Note that the terms safe and vulnerable may be a poor choice of words since a safe area is a subset of a vulnerable area. Although, we still feel that, the terms accurately describe the properties of the areas.

Important: Two of α's areas are considered adjacent if a single one of β's tiles could overlap both areas.

3.3 Board Cover

In order to use these board properties to get bounds on the number of moves a player has remaining, we need to determine how many of these properties exist on a given board. The rules for covering the board with these various properties are:

1. No two areas can overlap. Or in other words no square can be contained within two different areas.

2. No two protective areas can be adjacent. For example the covering of the board in Figure 3.2(c) would be invalid for the vertical player since areas A and B are adjacent.

To improve the lower bound on the number of moves a player has left, given a certain board position, we distinguish two types of vulnerable areas. Those which are not adjacent to any board property are type 1 vulnerable areas, those which are adjacent to other board properties are type 2 vulnerable areas. This naming scheme is to make it clear that β, with a single tile, could overlap up to two of α's type 2 vulnerable areas, but only one of α's type 1 vulnerable areas. For example, A and B in Figure 3.2(a) are type 2 vulnerable moves, while C and D are type 1 vulnerable moves.

An example of a complete covering of a board is shown in Figure 3.3.

3.4 Game Playing Strategy

Given a certain board position, we want to determine a strategy for α which allows us to determine a lower bound on the number of moves α can make.

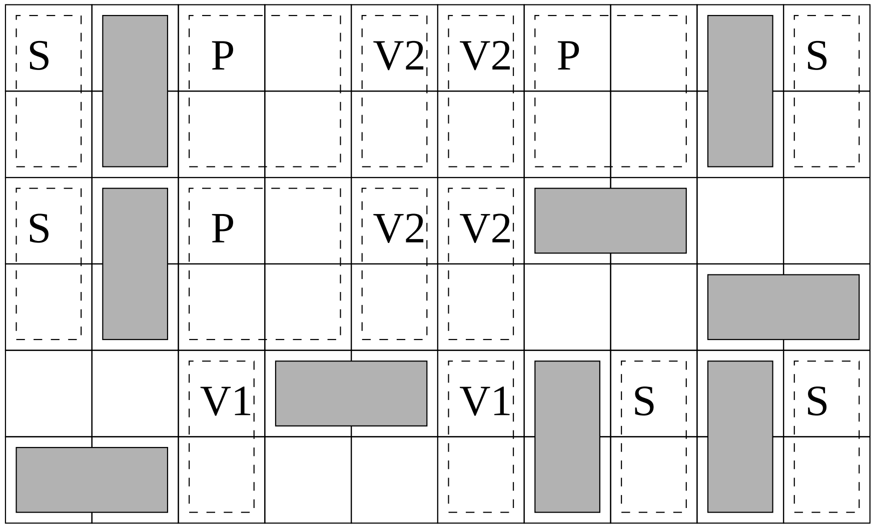

Figure 3.3: A possible covering of a 6×10 board for the vertical player. P denotes a protective area, $V2$ a type 2 vulnerable area, $V1$ a type 1 vulnerable area, and S a safe area.

The strategy is fairly simple. First we assume that it is currently β's turn. We will show later that this assumption does not really affect the size of the proof tree. Then we reply to each move β makes in the following way:

- If β places a tile over at least one of α's areas, then α responds by playing in the same type of area as β's tile overlapped. More explicitly:

  1. If β places a tile over one of α's protective areas, and this was not α's last protective area, α will respond by playing in another protective area. Note that this applies even if β's tile overlapped both a protective and a type 2 vulnerable area.
  2. Else, if β places a tile over one or two of α's type 2 vulnerable areas, and these were not α's last type 2 vulnerable areas, α responds by playing in another of these areas.
  3. Else, if β places a tile over one of α's type 1 vulnerable areas, and this was not α's last type 1 vulnerable area, α responds by playing

in another of these areas.

4. Else, if there are no areas remaining of the same type as that which β placed their tile upon, α responds by playing in any available area type.

- If β's tile does not overlap any of α's area types, then α can play a tile in either a safe area, a vulnerable area, or a protective area.

By following this strategy α is guaranteed to get some fraction of each type of area which they have marked in their board cover. Furthermore, by looking at the properties of the board and the number of moves α was able to play, we can determine an upper bound on the number of moves β can make. Finally, by comparing these upper and lower bounds, and taking into account who's turn it is, we can determine if α will win by following this game playing strategy.

We have no illusions that this strategy is an optimal one, but from the results we will show in Section 6.1 it appears to be very affective for computing bounds. What makes our strategy nice is that it is simple, makes sense intuitively, and is formulated in a way which makes it easy to prove bounds on the number of moves each player can still make. The main goal of this approach is to recognize easy wins and losses early in our search.

3.5 Lower Bound on Number of Moves Remaining

Now that we have defined a set of board properties, rules for creating a board cover, and a game strategy, we can compute a reasonable lower bound on the number of moves a player has left. We will refer to the number of protective areas that exist in α's covering of the board as $prot(\alpha)$, the number of type

2 vulnerable areas as $vuln2(\alpha)$, the number of type 1 vulnerable areas as $vuln1(\alpha)$, and the number of safe areas as $safe(\alpha)$.

Theorem 3.5.1 *If it is currently β's turn, and α plays with the strategy given in Section 3.4, then α has at least $moves(\alpha)$ left where:*

$$moves(\alpha) = 2 \cdot \left\lfloor \frac{prot(\alpha)}{2} \right\rfloor + \left\lfloor \frac{vuln2(\alpha)}{3} \right\rfloor + \left\lfloor \frac{vuln1(\alpha)}{2} \right\rfloor + safe(\alpha) + f(\alpha)$$

Where

$$f(\alpha) = \begin{cases} 1 & \textit{if } vuln2(\alpha) \bmod 3 \neq 0 \textit{ and } vuln1(\alpha) \bmod 2 \neq 0 \\ 0 & \textit{otherwise.} \end{cases}$$

A few special cases can be removed in the case where $prot(\alpha) \bmod 2 = 1$, by canceling one of the protective areas in α's cover of the board and converting it into two type 2 vulnerable moves. This guarantees that $prot(\alpha) \bmod 2 = 0$, simplifying the above equation and the proof.

Proof: We know from our general game strategy, given in Section 3.4, that α attempts to reply to any of β's moves by playing in the same type of area which β's tile overlapped. From this we can guarantee α will be able to place their tiles in some fraction of each type of area which has been included in the board cover.

1. Due to the nature of a safe area, namely the opponent cannot place a tile which will overlap it. α will be able to play a tile in every one of the safe areas which are marked on the board regardless of how β plays. Thus α is guaranteed $safe(\alpha)$ moves.

2. Since β can only block one of α's protective areas with each of their tiles and then the very next turn α will place a tile within one of these

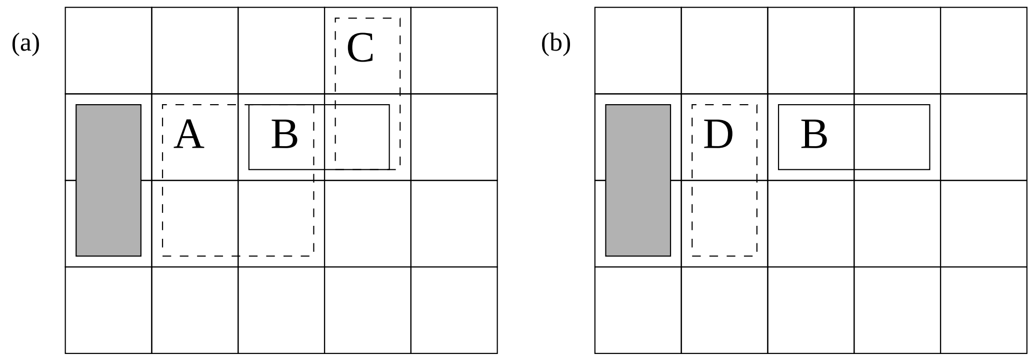

Figure 3.4: Crossing a protective and type 2 vulnerable area. (a) Opponent places a tile, B, across both a protective area, A, and type 2 vulnerable area, C. (b) Player cancels both A and C and creates a new type 2 vulnerable area, D.

protective areas, α will be able to occupy at least $\left\lfloor \frac{prot(\alpha)}{2} \right\rfloor$ of those areas. Also we know that each protective area α places a tile in α also creates another safe area for themselves. Therefore each protective area which α is able to occupy gives them 2 moves. Hence, α's protective areas guarantee α at least $2 \cdot \left\lfloor \frac{prot(\alpha)}{2} \right\rfloor$ moves.

It is possible for β to overlap both a protective area and a type 2 vulnerable area with one tile (see Figure 3.4(a)). However in doing this β covers only one square in the protective area. Therefore, from the remaining portion of α's protective area, another type 2 vulnerable area can be created to make up for the one which was lost (see Figure 3.4(b)).

3. Since β can place a tile over at most two of α's type 2 vulnerable areas with each move and α can respond by placing a tile in another type 2 vulnerable area, α will be able to occupy at least $\left\lfloor \frac{vuln2(\alpha)}{3} \right\rfloor$ of their type 2 vulnerable areas.

4. Similarly, since β can place a tile over at most one of α's type 1 vulnerable areas with each move and α can respond by placing a tile within another type 1 vulnerable area, α will be able to occupy at least $\left\lfloor \frac{vuln1(\alpha)}{2} \right\rfloor$ of

their type 1 vulnerable areas.

There is one special case that we should take note of. If $vuln2(\alpha) \bmod 3 \neq 0$ and $vuln1(\alpha) \bmod 2 \neq 0$ then there will be a point when β will have to place a tile across either one or two remaining type 2 vulnerable areas or they will have to place a tile over a remaining type 1 vulnerable area. In either case α will not be able to respond by playing on that same type of area, but will have to respond by placing a tile on the other type of vulnerable area. Since the number of this other type of area is not evenly divisible by either 3 or 2 respectively, α will be guaranteed at least one more move of this type. $\square$

A lower bound on the number of moves that α has remaining if it is currently α's turn to move can also be easily calculated. We can simply place a tile on the board, according to some heuristic, and then calculate the lower bound after that move.

3.5.1 Example of Lower Bound

Consider the board covering given in Figure 3.3, let α be the vertical player, and assume that it is horizontal's turn to play. We can use Theorem 3.5.1 to determine a lower bound on the number of moves that vertical can make.

In Figure 3.3, $prot(\alpha) = 3$, $vuln2(\alpha) = 4$, $vuln1(\alpha) = 2$, and $safe(\alpha) = 5$. Since $prot(\alpha) \bmod 2 = 1$ we convert one protective area into two type 2 vulnerable areas, leading to $prot(\alpha) = 2$, $vuln2(\alpha) = 6$, $vuln1(\alpha) = 2$, and $safe(\alpha) = 5$. Now we can use our formula:

$$\begin{aligned} moves(\alpha) &= 2 \cdot \left\lfloor \frac{prot(\alpha)}{2} \right\rfloor + \left\lfloor \frac{vuln2(\alpha)}{3} \right\rfloor + \left\lfloor \frac{vuln1(\alpha)}{2} \right\rfloor + safe(\alpha) + f(\alpha) \\ &= 2 \cdot \left\lfloor \frac{(2)}{2} \right\rfloor + \left\lfloor \frac{(6)}{3} \right\rfloor + \left\lfloor \frac{(2)}{2} \right\rfloor + (5) + (0) \\ &= 10 \end{aligned}$$

No matter how β plays, α can play at least 10 more tiles.

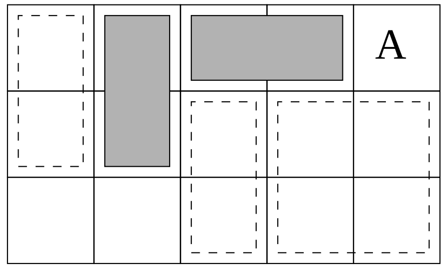

Figure 3.5: The square, A, is not covered by Vertical's board covering and is not available to Horizontal. (The dotted rectangles denote Vertical's board covering.)

3.6 Upper Bound on Number of Moves Remaining

We now would like to place an upper bound on the number of moves that β could play, given a specific board position and assuming that α will play according to the strategy given in Section 3.4.

In the previous section we determined that α is guaranteed, regardless of how β plays, to be able to play at least $moves(\alpha)$ more tiles onto the board. Therefore, this enables us to look at what properties the game board will have after α has placed all of these tiles, realizing of course that α must follow the strategy given in Section 3.4 and that we don't know how β will play.

Since α has placed at least $moves(\alpha)$ tiles, β has at least $2 \cdot moves(\alpha)$ fewer unoccupied squares available to play their tiles on. Let $squares(\beta)$ denote the number of squares which are still unoccupied after α has placed their tiles upon the board.

Further there can be squares on the board which are not covered by α's board cover, yet are unavailable to β. Let $unavail(\beta)$ denote the number of squares which fit this description (see Figure 3.5). These squares will never be played on since β is unable to and α's strategy doesn't take them into consideration.

Finally there are a number of squares which are included in α's covering of the board but which, due to the properties of these squares and of what we know of how α will play their tiles, α will not actually cover with a tile and β will be unable to play on. Let $unplayable(\beta)$ denote the number of these types of squares. We describe these in more detail below.

3.6.1 Unplayable Squares

To determine the number of squares which are included in α's board cover but which neither α will cover with a tile nor which β will be able to play on, we define two more types of board properties.

An **option area** for α is an unprotected, unoccupied square which is appended to an already existing safe area, in such a way as to create a 1×3 rectangle (see Figure 3.6(a)). To include an option area in a board cover, it cannot be adjacent to any other board properties contained in the board cover (see Figure 3.6(b)).

There are three types of option areas, type 1, 2, and 3, corresponding to the number of squares which they make unavailable for β if the option is played (see Figure 3.6(c)). We note that a type 3 option is a subset of a type 2 option since it makes at least 2 squares unavailable to β. Similarly a type 2 option is a subset of a type 1 option. We refer to the number of each of these different areas as $op1(\alpha)$, $op2(\alpha)$, and $op3(\alpha)$ respectively.

A **vulnerable area with a protected square** for α is a type 1 or type 2 vulnerable area in which one of the squares is unavailable to β. The importance of these vulnerable areas with protected squares is that for each of these types of areas which α does not place a tile within $unplayable(\beta)$ can be increased by one. We refer to the number of each of these different areas as $vuln2\_p(\alpha)$ and $vuln1\_p(\alpha)$. **Important**: The squares which these areas cover are the only ones which will be counted twice in our board cover, each of these areas

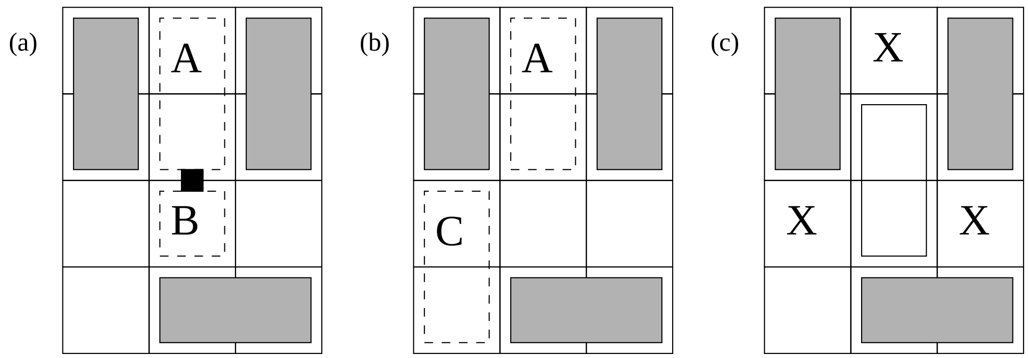

Figure 3.6: Option Area for Vertical. (a) B marks a type 3 option area for the safe area A. (b) If C is in the board cover, B couldn't be added. (c) All of the squares denoted by an X would be unavailable for the opponent if Vertical took the option.

will be counted as both a type 2 vulnerable area and a type 2 vulnerable area with a protected square, or as both a type 1 vulnerable area and a type 1 vulnerable area with a protected square.

To account for these new properties we refine the strategy given in Section 3.4 a little bit. First, if α has the choice of playing in a type 1 vulnerable area which doesn't contain a protected square and one which does, they will choose to play in the one which doesn't. Same thing for the type 2 vulnerable areas. Since we are still playing type 1 or type 2 vulnerable moves when we are supposed to, this does not affect the proof of Theorem 3.5.1. Second, any move which β makes which overlaps one of α's option areas, α will respond by taking one of its options which is of the same value as what β covered. This also doesn't affect the proof of Theorem 3.5.1 since β's move wouldn't overlap any of α's other areas.

Theorem 3.6.1 *If it is currently β's turn, and α plays with the strategy given in Section 3.4, then there is at least unplayable(β) squares which are covered in α's covering of the board, which will not be used by any of α's moves(α) tiles and which will not be available to β where:*

$$
\begin{aligned}
unplayable(\beta) \;=\; & \left(vuln2\_p(\alpha) - \left(\left\lfloor \tfrac{vuln2(\alpha)}{3} \right\rfloor - \left\lfloor \tfrac{vuln2(\alpha) - vuln2\_p(\alpha)}{3} \right\rfloor\right)\right) \\
& + \left(vuln1\_p(\alpha) - \left(\left\lfloor \tfrac{vuln1(\alpha)}{2} \right\rfloor - \left\lfloor \tfrac{vuln1(\alpha) - vuln1\_p(\alpha)}{2} \right\rfloor\right)\right) \\
& + 3 \cdot \left\lfloor \tfrac{op3(\alpha)}{2} \right\rfloor + 2 \cdot \left\lfloor \tfrac{op2(\alpha)}{2} \right\rfloor + \left\lfloor \tfrac{op1(\alpha)}{2} \right\rfloor + f(\alpha) + g(\alpha)
\end{aligned}
$$

and

$$
f(\alpha) = \begin{cases} -1 & \textit{if } vuln2(\alpha) \bmod 3 \neq 0 \textit{ and } vuln1(\alpha) \bmod 2 \neq 0 \\ & \quad \textit{and one of } vuln2\_p(\alpha) > 0 \textit{ or } vuln1\_p(\alpha) > 0. \\ 0 & \textit{otherwise.} \end{cases}
$$

and

$$
g(\alpha) = \begin{cases} 0 & \textit{if } vuln2(\alpha) \bmod 3 \neq 0 \textit{ and } vuln1(\alpha) \bmod 2 \neq 0. \\ 0 & \textit{else if } vuln2(\alpha) \bmod 3 = 0 \textit{ and } vuln1(\alpha) \bmod 2 = 0. \\ 3 & \textit{else if } op3(\alpha) \bmod 2 = 1. \\ 2 & \textit{else if } op2(\alpha) \bmod 2 = 1. \\ 1 & \textit{else if } op1(\alpha) \bmod 2 = 1. \\ 0 & \textit{otherwise.} \end{cases}
$$

To remove a few special cases from the proof:

- *If* $prot(\alpha) \bmod 2 = 1$*, cancel one of the protective areas in* α*'s cover of the board and convert it into two type 2 vulnerable moves. Hence,* $prot(\alpha) \bmod 2 = 0$.

- *If neither of the first two cases in* $g(\alpha)$ *apply make it so that all of* $op3(\alpha)$*,* $op2(\alpha)$*, and* $op1(\alpha)$ *have an even value, except one, by converting option areas to smaller types as needed. For example if you had one of each of the different types of option areas, and since a type 2 is a subset of a type 1, you could label the type 2 option area as a type 1 leaving one type 3 option area and two type 1 option areas.*

- *If one of the first two cases in* $g(\alpha)$ *apply make it so that all of* $op3(\alpha)$*,* $op2(\alpha)$*, and* $op1(\alpha)$ *have an even value by converting option areas to* *smaller types as needed. For example if you had one of each of the different types of option areas, and since a type 3 is a subset of a type 2, you could label the type 3 option area as a type 2 and completely remove the type 1 option area leaving two type 2 option areas.*

Proof:

1. By the strategy given in Section 3.4, and our slight refinements given above. α will be able to place a tile in at least $\left\lfloor \frac{vuln2(\alpha) - vuln2\_p(\alpha)}{3} \right\rfloor$ of their type 2 vulnerable areas which do not have a protected square before they could possibly run out of these types of areas. Therefore they will not have to play in $\left(vuln2\_p(\alpha) - \left(\left\lfloor \frac{vuln2(\alpha)}{3} \right\rfloor - \left\lfloor \frac{vuln2(\alpha) - vuln2\_p(\alpha)}{3} \right\rfloor\right)\right)$ of their type 2 vulnerable areas which contain a protected square, each of which leaves one square which β can't use.

2. An identical argument can be used to determine the number of type 1 vulnerable areas which contain a protected square that α will not have to place a tile within.

3. For the option areas, each time β blocks one of α's option areas, α will be able to immediately respond by placing a tile within one of their other option areas of an equal value to what β just blocked. Therefore, we can see that we will be able to use at least $\left\lfloor \frac{op(\alpha)}{2} \right\rfloor$ of each option type. We can also see that for each type 3 option we use it makes three squares unavailable for β, two for type 2 options, and one for type 1 options.

Finally there are two special cases which need to be taken care of. The first one is just an extension of the special case in the previous proof, where $vuln2(\alpha) \bmod 3 \neq 0$ and $vuln1(\alpha) \bmod 2 \neq 0$. In Theorem 3.5.1 we were able to show that this special case enabled α to place one more tile on one of the two types of vulnerable moves. This means that unless both of $vuln2\_p(\alpha)$ and

$vuln1\_p(\alpha)$ equal zero this extra move that α plays could possibly occupy one more of their vulnerable moves which contain a protected square, meaning that we over-calculated the number of squares in α's cover which will be unavailable to β by one.

The second special case is when α has only one of $vuln2(\alpha) \bmod 3 \neq 0$ or $vuln1(\alpha) \bmod 2 \neq 0$ and an uneven number of one of their option types. This will mean that at some point in the game β will have to play across one of these two types of areas, and α will be able to respond in the other. This will give α either one more option or one more vulnerable move. If they receive the extra option then this removes up to 3 squares from β (the value of the option area). If they receive the extra vulnerable move this removes two squares from β (the area of the extra tile α can play) and β will need two more squares than they did before since α will now be able to play $moves(\alpha) + 1$ tiles. Since no options are worth more than 4 squares it is always in β's interest to count the value of the option and forget about giving α an extra move.$\square$

3.6.2 Available Squares for Opponent

We have now determined the number of unoccupied squares β will have remaining after α has played their $moves(\alpha)$ tiles, the number of these squares which were not contained in α's board cover which are not available to β, and the number of squares within α's cover which will not be available for β. From all of this we can conclude that β can play a maximum of $\left\lfloor \frac{avail(\beta)}{2} \right\rfloor$ moves, where

$$avail(\beta) = squares(\beta) - unavail(\beta) - unplayable(\beta).$$

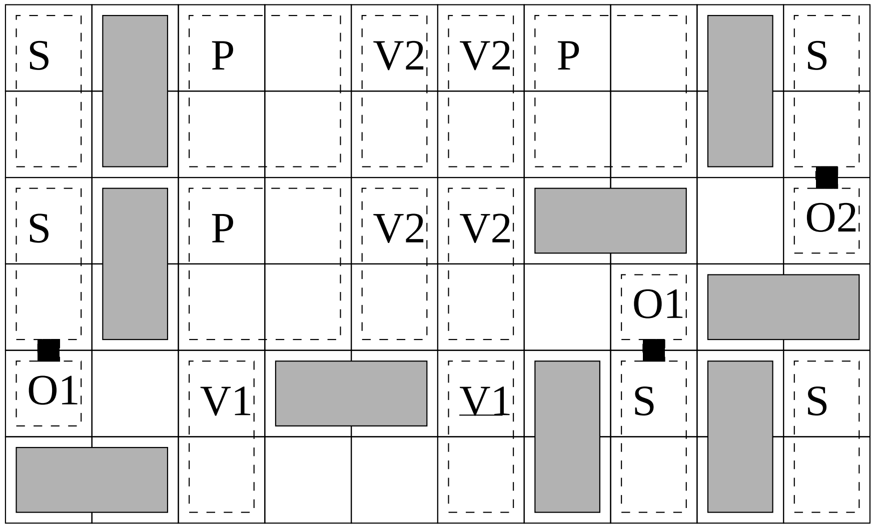

Figure 3.7: A possible covering of a 6×10 board for the vertical player. The squares denoted with $O1$ and $O2$ are option areas and the underlined $V1$ denotes a type 1 vulnerable area with a protected square.

3.6.3 Example of Upper Bound

Consider the example given in Section 3.5.1 and the augmented board covering given in Figure 3.7. From these and Theorem 3.6.1 we determine an upper bound on the number of moves that horizontal can make.

In Section 3.5.1 we already showed that α, the vertical player, can still play at least 10 tiles, with $prot(\alpha) = 2$, $vuln2(\alpha) = 6$, $vuln1(\alpha) = 2$, and $safe(\alpha) = 5$. In Figure 3.7 we can also see $squares(\beta) = 22$, $unavail(\beta) = 0$, $op1(\alpha) = 2$, $op2(\alpha) = 1$, $op3(\alpha) = 0$, $vuln2\_p(\alpha) = 0$, and $vuln1\_p(\alpha) = 1$.

Now we can use our formula:

$$
\begin{aligned}
unplayable(\beta) &= \left(vuln2\_p(\alpha) - \left(\left\lfloor \frac{vuln2(\alpha)}{3} \right\rfloor - \left\lfloor \frac{vuln2(\alpha) - vuln2\_p(\alpha)}{3} \right\rfloor\right)\right) \\
&\quad + \left(vuln1\_p(\alpha) - \left(\left\lfloor \frac{vuln1(\alpha)}{2} \right\rfloor - \left\lfloor \frac{vuln1(\alpha) - vuln1\_p(\alpha)}{2} \right\rfloor\right)\right) \\
&\quad +3 \cdot \left\lfloor \frac{op3(\alpha)}{2} \right\rfloor + 2 \cdot \left\lfloor \frac{op2(\alpha)}{2} \right\rfloor + \left\lfloor \frac{op1(\alpha)}{2} \right\rfloor + f(\alpha) + g(\alpha) \\
&= \left((0) - \left(\left\lfloor \frac{(6)}{3} \right\rfloor - \left\lfloor \frac{(6)-(0)}{3} \right\rfloor\right)\right) \\
&\quad + \left((1) - \left(\left\lfloor \frac{(2)}{2} \right\rfloor - \left\lfloor \frac{(2)-(1)}{2} \right\rfloor\right)\right) \\
&\quad +3 \cdot \left\lfloor \frac{(0)}{2} \right\rfloor + 2 \cdot \left\lfloor \frac{(0)}{2} \right\rfloor + \left\lfloor \frac{(2)}{2} \right\rfloor + (0) + (0) \\
&= 1
\end{aligned}
$$

To determine the upper bound on the number of squares β can place a tile on we use the formula from Section 3.6.2:

$$
\begin{aligned}
avail(\beta) &= squares(\beta) - unavail(\beta) - unplayable(\beta) \\
&= (22) - (0) - (1) \\
&= 21
\end{aligned}
$$

Therefore we can determine that β can play at most $\left\lfloor \frac{21}{2} \right\rfloor = 10$ tiles.

3.7 The Winner Is ...

In a given board position α can still make at least $moves(\alpha)$ moves, no matter what α's opponent does, provided that α plays according to our given strategy. We also know that α's opponent can place at most $\left\lfloor \frac{avail(\beta)}{2} \right\rfloor$ tiles. If it is currently β's turn and $moves(\alpha) \geq opp\_moves(\alpha)$, then α can win. Similarly, if it is currently α's turn and $moves(\alpha) > opp\_moves(\alpha)$, then α can win.

3.7.1 The Example

From our examples in Section 3.5.1 and Section 3.6.3 we know α can play at least 10 more tiles and β can play at most 10 tiles. Therefore, since it is currently β's turn we can deduce that α has a win in this position.

Chapter 4

Search Enhancements

All search programs, when optimized for a specific search task, will use a number of different enhancements to the normal alpha-beta search algorithm. These enhancements could include iterative deepening, transposition tables, move ordering heuristics, specialized local searches, and many other general as well as problem-specific ideas.

In our program a couple of the enhancements which we felt were worth analyzing were the use of transposition tables in our solver (or more specifically the replacement scheme used with the transposition tables) and the move ordering scheme. Both of these ideas were already looked at by Breuker et al. in their program DOMI [3]. We have extended their findings with some improvements of our own. We have also made one other enhancement which they did not investigate, by proving that one type of move is always inferior to some other move on the board we can prune these moves from the search.

4.1 Transposition Tables

Most if not all search programs for game-playing or other search-related problems build trees during the traversal of the search space of the problem. A common situation is that multiple paths reach the same state in the search space. Unless we are able to determine that we have seen a state before and

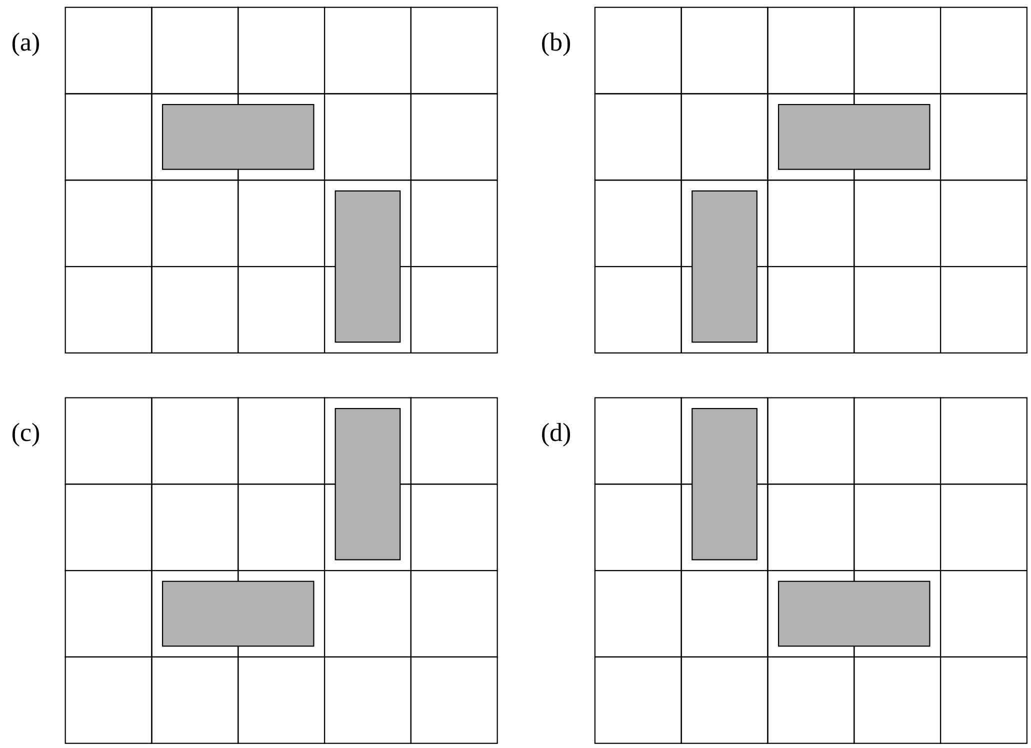

Figure 4.1: Board Symmetries. Each of these boards are equivalent to each other through a vertical and/or horizontal reflection. We don't attempt to find symmetries through rotation since it affects the meaning of vertical and horizontal.

know its value, we will end up duplicating previously done searches below that state.

In game-playing and solving programs, transposition tables [7, 5] are commonly used to store a game state and the value associated with that state. These tables are designed for fast access to the entries in the table. If a state from the search is found in the table, then its value can be retrieved and further search at the node might be unnecessary. We will commonly refer to the game state as the game position.

Transposition tables can be used to exploit the symmetries which exist in a given search space. For example in domineering if a position is identical to the horizontal reflection of a different position, and in both cases it is the same player's turn to move, then the positions can be considered isomorphic.

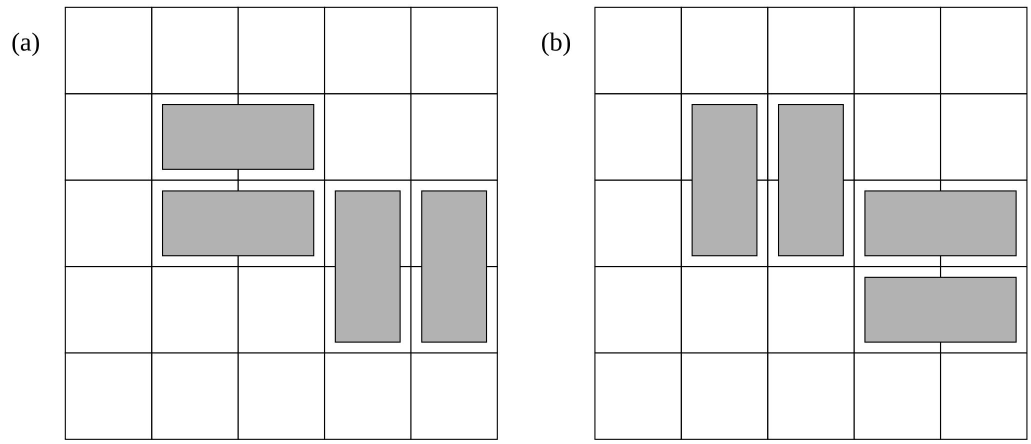

Figure 4.2: Board equivalence. It is easy to see that these two boards are equivalent to each other. Only empty squares are important in determining the equivalence of domineering positions.

In domineering a position can be mapped to its horizontal and/or vertical reflection (see Figure 4.1). All isomorphic positions can be looked up in the transposition table. Alternatively, a position could be mapped to a canonical position and then looked up in the transposition table. It is interesting to point out that in domineering it is unimportant to distinguish between which tiles have been placed on the board but only on which squares are currently unoccupied (see Figure 4.2).

4.1.1 Implementation

Our transposition table implementation is very similar to that used in the program DOMI [3]. The transposition table is just a basic hash table. A random integer is associated with each square on the board. Then for a given position we exclusive-or (xor) all of the integers together which are associated with occupied squares. The resulting value is used as the hash value for that specific position (modulo the hash table size). This general method of determining a hash value is commonly called a Zobrist hash function [11]. Note that since we are using a hash table, collisions between different board positions can occur.

During a search, as we determine the values of different board positions we can store these values into the transposition table by using the hash value associated with that position as the index. This also allows us to check in the hash table, before searching a position's subtree, to determine if we have already determined the value of that position. Besides storing the value of the position in the hash table, in order to differentiate between positions which hash to the same location in the transposition table, we also need to store a key which uniquely identifies that position (in our case we store the entire position) and which player moves next. We also store other information associated with this position in the hash table, such as the size of the subtree we searched to determine the positions value.

4.1.2 Replacement Schemes

For a large search space there are not enough entries in the transposition table to contain information about each distinct position which is examined. Therefore, when two different positions have the same hash value, we need to have some method to decide which position will stay in the transposition table and which is removed.

Breuker et al. looked at a number of different replacement schemes and ran tests to determine which of them worked best in the game of domineering [3]. The replacement schemes they considered were:

- New. This scheme places the new position in the transposition table, and discards the older position (i.e. the one currently in the table).
- Deep. This scheme places the position with the deepest subtree in the transposition table. In other words, if a position in the table has a search depth of 5 and the new position has been searched to a depth of 7, then replace.

- Big. This scheme places the position with the biggest subtree (the subtree with the largest number of nodes) in the transposition table. For example, if position A was searched to a depth of 5 and took 10,000 nodes, and position B was searched to a depth of 7 but took only 200 nodes, put A in the transposition table.

- TwoBig and TwoDeep. These are both two-level transposition tables. Each hash value has two entries associated with it. The first entry uses either a deep or a big replacement scheme. If the new position is not stored in the first entry it is always stored in the second entry. The result is a combination of the Big or Deep scheme and the New scheme. The second entry captures recency and improves the temporal locality in the search.

Breuker et al. concluded that TwoBig was the best of all of these replacement schemes.

In our implementation of a domineering solver we came up with two other replacement schemes worth considering. Both are just slight variations on the Big scheme. We call these two new schemes FindFirst and FindWorst. Both of these schemes use a single-level transposition table which stores the size (number of nodes) of the subtree with the position.

Our two replacement schemes work as follows:

- FindFirst. From the symmetries in the game of domineering each position has four possible entries in the transposition table which it maps to (i.e. by reflection). In this scheme we check each of these four entries, replacing the first entry which has a smaller subtree than this new position.

- FindWorst. In this scheme we determine which of the four entries which a position maps to has the position with the smallest subtree stored in

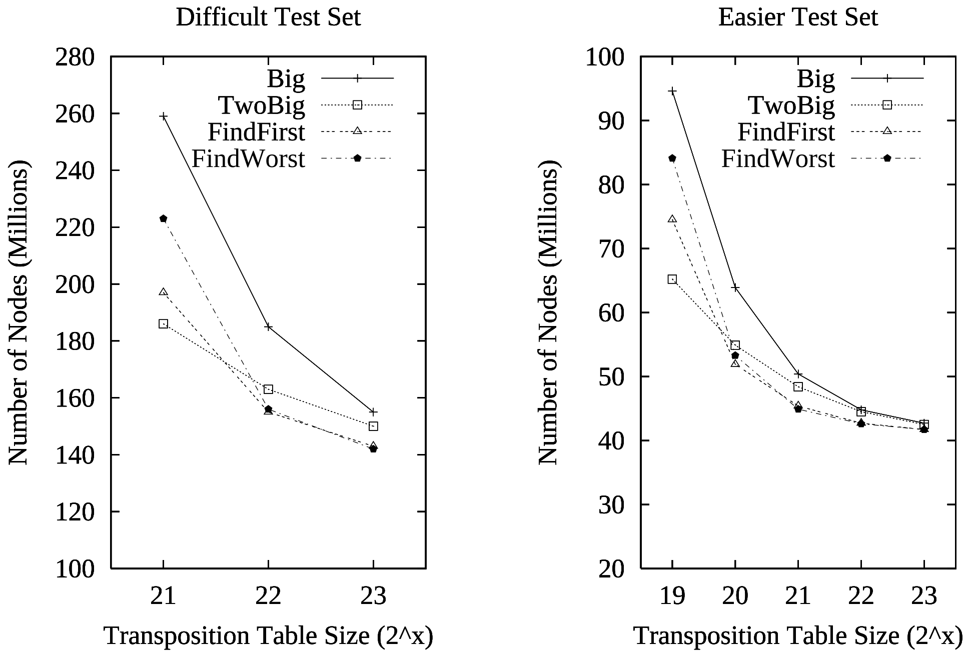

Figure 4.3: Transposition Table Experiments. The x axis of each graph measures the different sizes of transposition tables for each of our experiments. The y axis shows the average number of nodes required to solve the different cases in each data set. Note the very close similarity between the two test sets, except that they differ from each other by a factor of about four.

it. Then if the new position's subtree is larger than this smallest subtree, we replace that position with the new one.

4.1.3 Experimental Results

To measure the effectiveness of these four different replacement schemes we looked at positions from our 10×10 proof set. From these positions we picked 8 positions which needed about 200 million nodes in order to determine a winner and 10 positions which needed about 50 million nodes in order to solve. We will call these two test sets Difficult and Easier. Then we processed each of these two test sets with different combinations of hash-table size and replacement scheme, taking the average number of nodes which were required to solve the position sets under each set of conditions. The results are shown in Figure 4.3.

4.1.4 Analysis of Replacement Schemes

The FindWorst results closely mirrored the FindFirst results. However since FindWorst is computationally more expensive, FindFirst is the more practical of the two. Also TwoBig seemed to be a superior scheme in comparison to the simpler Big scheme, confirming the results reported by Breuker et al. [3].

In comparing the two schemes FindFirst and TwoBig, the choice of a replacement scheme really depends on the amount of memory available and the difficulty of the problem. Our results would seem to point out that there is a crossover point, approximately when the number of nodes is 50 times larger than the size of the transposition table, below which the FindFirst replacement scheme begins to outperform the TwoBig replacement scheme. FindFirst is computationally less expensive than the TwoBig scheme, and on average is able to process 2-3% more nodes in the same amount of time.

To emphasize the difference between the two replacement schemes we would like to point out one section in Figure 4.3. With a transposition table size of four million entries (2^{22}), on the difficult test set the difference between FindFirst and TwoBig was on average eight million nodes or about five percent. Then if we include the cost of the two replacement schemes into the figures the difference jumps to around 8 percent. This we consider fairly significant. Also if we assume that the two replacement schemes equal out as the number of nodes approaches 50 times the size of the transposition tables, FindFirst will be the superior scheme for problems sizes of between 0 to 200 million nodes (when we have a four million entry transposition table).

From this we conclude that either one of these schemes can be the superior scheme to use but that it depends on the size of the search space and the amount of memory available. To solve 10×10 domineering took many CPU months of computing power and 8 percent translates into a savings of around

2 weeks of computing!

4.2 Move Order Heuristic for Domineering

As we already noted in Section 2.2, a good move ordering heuristic is very important to the alpha-beta search algorithm.

In our domineering solver we used the same move ordering scheme as that of Breuker et al. [3]. Each move is assigned a score which reflects (1) how many guaranteed moves a player has compared to their opponent and (2) the total number of moves a player has compared to their opponent. The higher the score, the more likely the move is good. The formula for this move ordering value is

$$value = real(\text{Max}) - real(\text{Min}) + safe(\text{Max}) - safe(\text{Min}),$$

where $real(x)$ denotes the maximum number of tiles that the player x could place on the board if their opponent did not place any more tiles upon the board, and $safe(x)$ denotes the number of tiles that player x could place on the board regardless of how many tiles their opponent placed on the board (formal definitions are given in Section 1.2). The children of a node are considered in decreasing order of their move score.

For the most part, this heuristic worked very well as a first-order move ordering scheme. However, for any given node many of its children recieve the same score. Therefore we felt that a second-order move order scheme would improve the move ordering.

4.2.1 Exploiting Symmetries in Domineering

In domineering we know that if a position is identical to the horizontal and/or vertical reflection of another position, and in both cases it is the same player's turn, then we can consider these positions as being identical. This means that once we have searched the subtree of one of these positions and determined the value of that position, there is no need to search the subtree of the other position, since both positions must have the same value.

Therefore we should be able to exploit this property of domineering by guiding the search towards positions which are more similar with respect to reflection. This will increase the chances of isomorphic positions in the search.

In order to do this we came up with a secondary move ordering scheme. Each move is assigned a score which denotes how early in the search a move which is a reflection of itself was played. The earlier a reflective move was played the more likely the move will guide us to an isomorphic position which already exists in the transposition table. The formula for the value of the move (x,y) is

$$value(x, y) = min \begin{cases} depth(m - x + v, & y) \\ depth(m - x + v, & n - y + h) \\ depth(x, & n - y + h) \end{cases}$$

where $depth(x, y)$ is the depth at which the move (x, y) was played by the current player or infinite if that move has not been played, m is the number of rows on the board, n is the number of columns on the board, v is 2 if the current player is vertical and 1 otherwise, and h is 2 if the current player is horizontal and 1 otherwise. The children of a node will be considered in increasing order of this secondary move score.

For example (see Figure 4.4) assume that our program is trying to solve 6×4 domineering with horizontal going first (note that this is a win for horizontal). Our program first chooses a move for horizontal, then it must try all possible positions for the vertical player to prove that vertical can't win (Figure 4.4 (a) and (b) each show one of vertical's possible moves). On horizontal's second move there are a number of positions which are valued identically by our primary move ordering function. Now if we were just generating our moves in

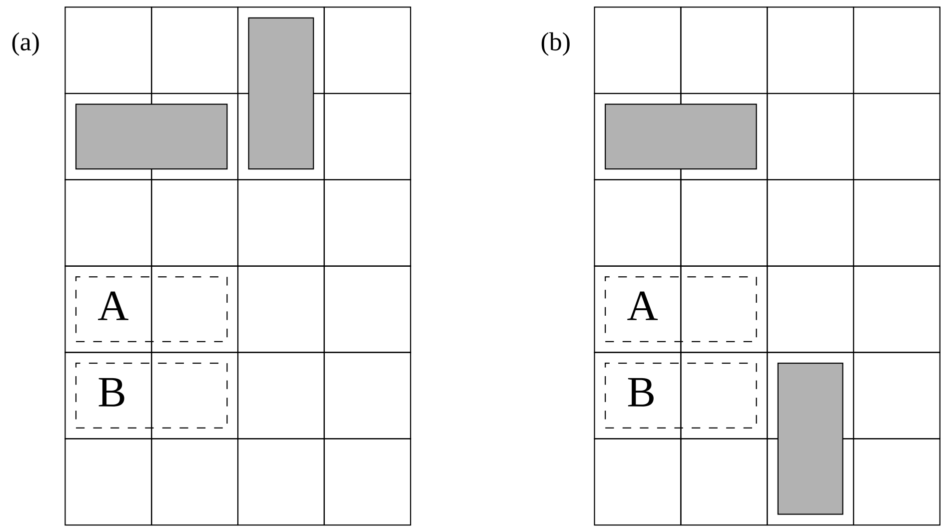

Figure 4.4: Exploiting the Symmetries of Domineering. These two boards represent two different positions, two ply deep, in the proof of 6×4 domineering with horizontal going first. Note that if we play in position A in both of them we have to continue to search both positions. If we play B they become transpositions of each other.

a top-to-bottom, left-to-right fashion we would end up playing A first in both cases, and we would have to search both subtrees to find out if both were wins for horizontal. But if we used our secondary ordering function we would find that in both cases we would play B first, since B would recieve a value of 1 and A would recieve a value of ∞ from our secondary move ordering scheme, making both positions identical and giving us the opportunity to prune one of the cases from our search tree.

4.2.2 Analysis of Exploitation

We want to compare the primary move ordering scheme, which we described in Section 4.2, against our enhanced version which uses both the primary and secondary move ordering schemes. We ran a number of tests comparing the number of nodes which it took to solve different board sizes using each of these two move ordering methods. We found that adding a symmetry bias to the scheme substantially decreased the growth of the number of nodes at each

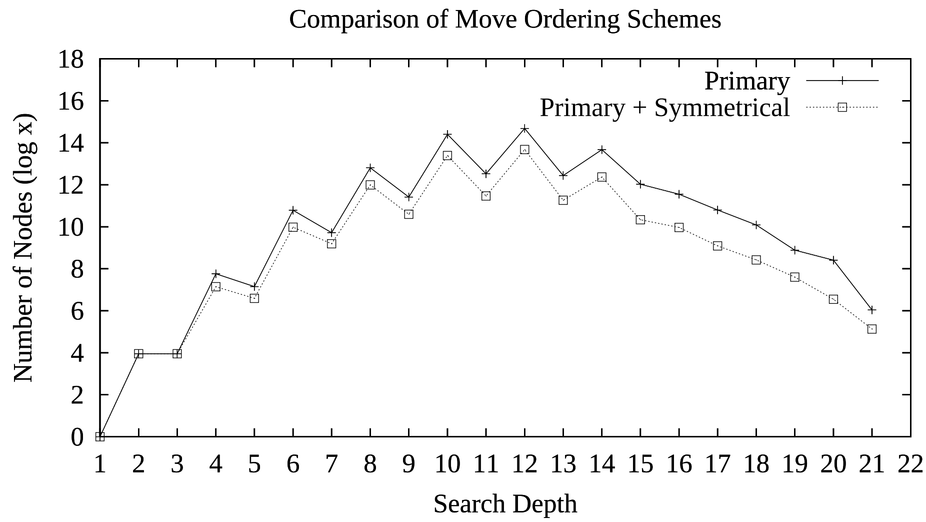

Figure 4.5: Comparison of primary move ordering scheme vs the primary scheme enhanced with our symmetrical move ordering scheme. Graph shows the number of nodes examined at each ply in the 8×8 proof tree for each move ordering implementation. Note the y axis uses a logarithmic scale.

Depth	Primary	Primary+ Symmetry	Depth	Primary	Primary+ Symmetry
1	1	1	12	2,381,100	871,659
2	52	52	13	251,905	77,648
3	52	52	14	863,380	235,807
4	2,346	1,263	15	166,431	30,700
5	1,271	728	16	103,767	21,289
6	48,217	21,622	17	49,112	8,833
7	16,612	9,823	18	24,086	4,538
8	365,997	161,329	19	7,248	2,003
9	90,709	39,980	20	4,488	698
10	1,802,293	652,707	21	420	169
11	274,926	95,345			

Figure 4.6: This gives a tabular view of the data displayed in Figure 4.5.

subsequent level of depth in the search. For example in solving 8×8 domineering, using only the primary move ordering scheme, our program had to search 6 million nodes to determine the winner, but by using the combination of the two move ordering schemes it took only slightly over 2 million nodes. As expected, both versions had to search to the same depth to determine a winner, but the number of nodes at each depth was significantly reduced. (See Figure 4.5 and Figure 4.6). Note that the odd/even saw-tooth affect in the graph is due to the assymetry of alternating Max and Min nodes.

4.3 Ignoring Safe Moves

One further enhancement that *Obsequi* can take advantage of is the idea that as long as there exists a vulnerable area for the current player there is no need to place a tile in a safe area.

Theorem 4.3.1 *Given a domineering board which contains at least one safe area and one vulnerable area for α and it is α's turn to play. Then there exists a vulnerable area which is as good or better a move for α than to play in any of the safe areas.*

Proof: Assume the optimal move sequence for α and β until the end of the game is $m_1(\alpha), m_1(\beta), m_2(\alpha), m_2(\beta), \ldots$ and that $m_1(\alpha)$ was placed in a safe area. There are two cases we need to look at.

First. At least one of α's moves was placed in a vulnerable area. Let $m_x(\alpha)$ be the first vulnerable move α played. Since $m_x(\alpha)$ is the first vulnerable move, $m_1(\alpha), m_2(\alpha), \ldots, m_{x-1}(\alpha)$ are all safe moves. α could get the same results by playing $m_x(\alpha)$ first, then $m_1(\alpha), m_2(\alpha), \ldots, m_{x-1}(\alpha)$, since $m_1(\alpha), m_2(\alpha), \ldots, m_{x-1}(\alpha)$ are all safe moves and therefore β could not interfere with them.

Board Size	Safe Moves	No Safe Moves
7×8	1,030,221	949,209
7×9	7,472,487	6,052,516
8×8	2,272,909	2,023,301

Figure 4.7: Proof tree sizes for various boards with safe moves either being generated or not.

Second. None of α's moves were placed in a vulnerable area. Let $m_1(\alpha)$ be replaced with an existing vulnerable move. Since this vulnerable move can disrupt at most one of α's safe areas, and it can only decrease the amount of space β has to place their moves, it can not negatively affect α. $\square$

This enhancement is a very simple idea to help reduce the branching factor in the game of domineering. Figure 4.7 shows a number of tests which show that this enhancement does have a small affect on the size of the proof trees for various sizes of boards. Node expansion also becomes more efficient since there are fewer child positions to examine and evaluate. We will refer to these types of relationships, where one move is guaranteed to always be at least as good or better than another, as a dominance relationship.

4.4 Conclusion

In this chapter we have introduced three new search enhancements: a new replacement scheme, FindFirst, a secondary move ordering function which exploits the symmetries which exist in domineering, and a dominance relation between vulnerable and safe moves. The first two of these enhancements are modifications of previously published ideas, yet in some cases they give substantial improvements to the size of the proof trees. The third enhancement is a new idea for domineering, which possibly has great promise for further improvements. Possibly other dominance relations can be discovered which will further reduce the branching factor of domineering.

All of these enhancements are important for building a better domineering search program, and yet even together they do not have nearly as much impact on the size of the search trees as our improved evaluation function.

We still feel that there are large areas for improvement in search enhancements for domineering. One of those areas would be an improved transposition replacement scheme which tries to merge the improvements gained from Find-First with the sustained effectiveness of the TwoBig scheme. Another area for further research would be to try to prove further dominance relationships. Possibly there exists patterns, where one position in the pattern dominates another position, allowing us to reduce the branching factor of domineering further.

Chapter 5

Solving 10×10 Domineering

One of the goals we wanted to reach in doing research on domineering was to solve larger boards than had ever been solved before. The pinnacle being the 10×10 board - the smallest interesting-sized unresolved problem.

Previous to our research, 8×8 domineering had been solved by various people, as well as many other smaller board sizes had also been solved by a number of people. 9×9 domineering had been solved by Breuker et al. with their program DOMI [3]. We wanted to try the next step: 10×10 domineering.

5.1 Estimation of Difficulty

We can get an estimate of the total size of the search space for 10×10 domineering by estimating the total number of different board positions which are reachable. To estimate this we looked at the first 5 ply of the search space (see Figure 5.1). We note that on average the number of moves available decreases by about 3 with each ply of the search. This would mean that to search to the end of the game a depth of at least 30 would need to be reached. Also it can be noted that for the first five ply the effective branching factor ranges from 25 to 77. Therefore a conservative estimate of the effective branching factor at each ply would be a factor of 25. This data suggests that the size of the search space is approximately $25^{30} = 8.6 \cdot 10^{41}$.

Depth	Branching Factor	Nodes at Current Depth	Effective Branching Factor
0	90	1	25
1	86	25	77
2	83	1944	41
3	79	81609	43
4	77	3520539	48
5	74	169944142	-

Figure 5.1: Growth of 10×10 Search Tree.

Another method to determine the difficulty of the problem is to look at the number of nodes which it took to solve smaller boards and then just extend those numbers to the larger board. For example in the published results by Breuker et al. [3] 7×7 domineering took about $4 \cdot 10^5$ nodes, 8×8 took $4 \cdot 10^8$ nodes, and 8×9 took $7 \cdot 10^{10}$ nodes. At this rate of growth we estimated that 9×9 could easily need more than $4 \cdot 10^{11}$ nodes and 10×10 domineering may take upwards of $5 \cdot 10^{14}$ nodes to solve. This is a number which is probably beyond our current computational resources.

The good news is that if we use this same method of estimation with the number of nodes which *Obsequi* needed to solve 8×8 domineering, $2 \cdot 10^6$, and 9×9 domineering, $2.5 \cdot 10^9$ nodes. We get a much smaller estimate of around $3 \cdot 10^{12}$ nodes for *Obsequi* to solve 10×10 domineering.

5.2 Splitting up the Work

Examining 3 trillion nodes is obviously much better than 500 trillion, but with only one processor we estimate that it would have taken upwards of 150 days, probably more since the transposition table wouldn't have been big enough to be effective. Therefore the work needed to be split up.

Our initial assumption was that 10×10 domineering was a first player win. Therefore we ran our solver with the condition that every node at the eighth

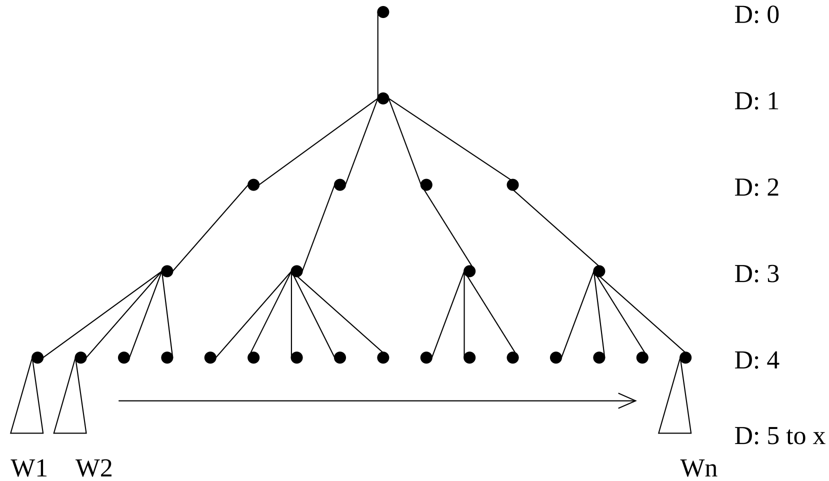

Figure 5.2: Minimal Alpha-Beta Tree of Depth 4 (assuming first player win). Since we are assuming a first player win, the first player only needs to make one move at each of his turns, while all of the second player's possible moves need to be examined. The small triangles at the bottom of the tree represent the independent searches that can be divided among many different processors to validate that the values of the nodes at the fourth ply of the tree are all wins for the first player.

ply of the search was a first player win, effectively giving us a possible start of the proof DAG (directed acyclic graph), with transpositions removed, of the first 8 ply of 10×10 domineering. (See Figure 5.2). We then had the program write all the leaf nodes of this DAG to a large file. The 650,531 nodes which were generated were then split into a number of different work files, and each file was assigned to a different processor. Each of these processors then worked on verifying that in fact all of these nodes were first player wins.

Obviously, if any losses were found we would then need to re-run our solver with the added knowledge of which cases were second player wins, from which a new set of nodes would be generated. The process would iterate from there until we were able to find a set of leaf nodes which were all first player wins. At this point we would then have a correct proof for 10×10 domineering.

5.3 10×10 A First Player Win

The computers which we used to solve this problem varied from Pentium 3 600 MHz machines to Pentium 3 900MHz machines, with transposition tables which varied in size from between 1 million entries to 8 million entries, depending on the amount of memory available on the given machine. The number of nodes which they could examine per second ranged from 140,000 to 210,000.

3,541,685,253,370 (3.5 trillion) nodes later the results were in: 10×10 domineering is indeed a first player win. If vertical is the first player then the winning move is position (1,2). *Obsequi*'s original move ordering had been good enough that there was no need for a second iteration of the solving process: all 8-ply positions in the DAG were proven to be wins.

5.4 Correctness

Computer programs are inherently problem ridden; a subtraction sign instead of an addition sign, an = instead of an ==, etc. Therefore we would like to give a couple of reasons why we feel that the results can be trusted.

First, we have examined hundreds of random positions in the search trees. These positions have been examined to make sure that the right set of moves were generated, to make sure that the evaluation function returned the correct value for the given position, and to make sure the correct board positions were generated when we applied these moves to the current board position. Second, we ran *Obsequi* on all of the boards which have been previously solved by other researchers, and verified that *Obsequi* returns the same values for each of these boards. Third, in solving 10×10 domineering, we tried to make our method of breaking the problem up into smaller sub-problems as simple as possible, so as not to introduce new complexities which may have errors attached to them.

One final point we would like to address is the fact that *Obsequi* did not make any mistakes for the first player in any of their first four moves. This may seem surprising, but for example in 8×8 domineering, *Obsequi* didn't make any mistakes until the sixth ply of the search, and then only made 2 mistakes out of 9862 moves. Therefore given the fact that the search space for 10×10 is considerably bigger, it is completely believable that *Obsequi* was able to extend its perfect accuracy to the eighth ply of the search.

For all of the reasons stated above we are very confident in our results. But as always independent confirmation of our 10×10 results would definitely be welcome and lend even greater credibility to the results.

Chapter 6

Conclusion

The improvements we have made in the evaluation function, move ordering, and the transposition table replacement scheme have made an enormous impact on the size of domineering positions we are able to solve.

One measure of how far we have progressed is the length of time it takes to solve a specific board position. In "Solving 8×8 Domineering" by Breuker et al. [3] it was mentioned that it took them 600 hours and almost 71 billion nodes to determine that 8×9 domineering was a win for the vertical player. *Obsequi* is able to tell us the same result in less than twenty minutes and needs to examine fewer than 260 million nodes. This is a huge improvement in a time span of just 2 years, and represents a difference which is far larger than just the improvements in computer hardware. It is illustrative to compare DOMI's results, Figure 1.2, to those of *Obsequi*, Figure 6.1.

A second measure of the progress which has been made is the number of new board positions which *Obsequi* can solve (See Figure 6.2).

The most impressive example though is our proof tree for 10×10 domineering. If we assume that the entire search space is on the magnitude of 25^{30}. We know that alpha-beta in the best case could reduce this to $25^{15} = 9.3 \cdot 10^{20}$. Therefore, since our proof for this board only examined $3.5 \cdot 10^{12}$ positions, our enhancements reduced the number of nodes which needed to be examined

Board Size	Result	Nodes	Board Size	Result	Nodes
2×2	1	1	4×7	V	802
2×3	1	2	4×8	H	2,570
2×4	H	7	4×9	V	13,570
2×5	V	6	5×5	2	259
2×6	1	8	5×6	H	324
2×7	1	2	5×7	H	2,210
2×8	H	26	5×8	H	2,467
2×9	V	65	5×9	H	11,669
3×3	1	1	6×6	1	908
3×4	H	5	6×7	V	24,227
3×5	H	14	6×8	H	204,813
3×6	H	16	6×9	V	1,374,535
3×7	H	43	7×7	1	31,440
3×8	H	33	7×8	H	949,209
3×9	H	100	7×9	H	6,052,516
4×4	1	23	8×8	1	2,023,301
4×5	V	42	8×9	V	259,064,428
4×6	1	583			

Figure 6.1: Game-theoretic values for various sizes of domineering boards and the number of nodes *Obsequi* needed to calculate the values. (We used a transposition table with 2^{23} entries and the FindFirst replacement scheme.) Compare to Figure 1.2.

Board Size	Result	Nodes
4×19	H	314,148,901
4×21	H	3,390,074,758
6×14	H	1,864,870,370
8×10	H	4,125,516,739
9×9	1	1,657,032,906
10×10	1	3,541,685,253,370

Figure 6.2: New game-theoretic values which have been determined by *Obsequi* for various interesting board positions, as well as the number of nodes needed to calculate the values. Note that 9×9 has been previously solved by DOMI [9]. To solve the above positions we used a transposition table with 2^{23} entries and the TwoBig replacement scheme, except for 4×19 where we used the FindFirst replacement scheme, and 10×10, which was solved as described in Chapter 5.

by a factor of 10^8.

6.1 Evaluation Function

The most significant improvement in our solver was our evaluation function. This enabled *Obsequi* to prune lines of search from its proof trees far faster than previous programs. This means that when a losing move is made our enhancements are able to determine this much sooner and therefore prune the subtree of that position.

In the 8×8 proof tree this enables our program to examine 40 times fewer nodes than it would without these enhancements. For example, with all of *Obsequi*'s enhancements turned on it takes 2,023,301 nodes to prove $[8 \times 8] = 1$. If we turn off all the enhancements to the evaluation function it takes 84,034,856 nodes. See Figure 6.3 and Figure 6.4 for a more detailed look at how individual enhancements affect the size of the proof tree for 8×8 domineering.

6.2 Extending To All Rectangular Boards

Even with all the progress which has been made in solving the game of domineering we are still only able to solve a fairly small number of domineering boards. Thanks to the results of Lachmann et al. [6] we are able to extend the results obtained through search techniques to much larger boards.

The new results which we have computed, in conjuction with the rules discovered by Lachmann et al. [6], have enabled us to determine the values of many more board positions. Some of the more interesting values which we have obtained, which were not previously known, are:

- $[4 \times 19] = [4 \times 21] = H$. With these two results we now are able to determine who wins on all $4 \times x$ boards. $[4 \times x] = H$ for all values of x,

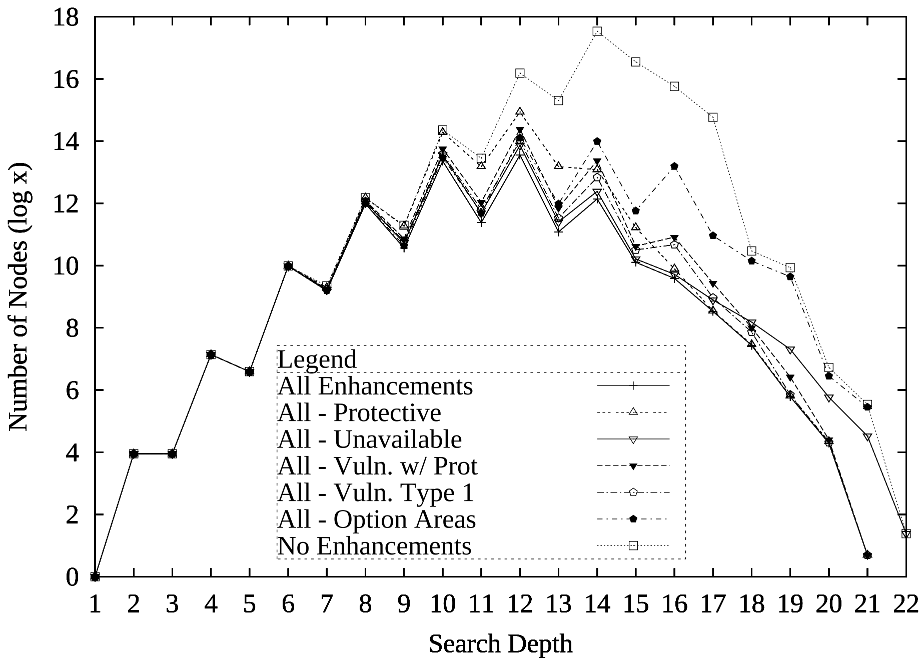

Figure 6.3: Comparison of different evaluation functions (number of nodes at each ply of the search). The x axis is the depth or ply of the search, the y axis is the log of the number of nodes which were examined at that depth of the search. "All - A" denotes that we used all of the enhancements except A.

Enhancements	Size of Proof Tree (nodes)
All Enhancements	2,023,301
All - protective areas	6,610,775
All - unavailable squares	2,566,004
All - vulnerable areas w/ protected squares	4,045,384
All - vulnerable type 1 areas	2,972,216
All - option areas	4,525,704
No Enhancements to Evaluation Function	84,034,856

Figure 6.4: Comparison of the size of the proof trees for 8×8 domineering given a certain evaluation function. "All - A" denotes that we used all of the enhancements except A.

where $x \geq 14$.

- $[6 \times 14] = H$. With this new result we can now determine $[6 \times x] = H$ for all even values of x, where $x \geq 20$.
- $[8 \times 10] = H$. This is the first $8 \times x$ board which has been determined to have a value of H.
- $[10 \times 10] = 1$. This is the largest square board ever solved.

See Figure 6.5 for an updated table of who wins on rectangular boards.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	>27
1	2	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
2	V	1	1	H	V	1	1	H	V	1	1	H	2	1	1	H	H	1	1	H	H	H	1	H	H	H	1	H
3	V	1	1	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
4	V	V	V	1	V	1	V	H	V	H	V	H	2	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
5	V	H	V	H	2	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
6	V	1	V	1	V	1	V	H	V	1	1	H	V	H		H		1h	1h	H		H	1h	H	1h	H	1h	
7	V	1	V	H	V	H	1	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
8	V	V	V	V	V	V	V	1	V	H	V		V			2h		1h		H				1h				
9	V	H	V	H	V	H	V	H	1	H	1h	H	1h	H	1h	H	1h	H	1h	H	1h	H	H	H	H	H	H	H
10	V	1	V	V	V	1	V	V	V	1			V							2h		1h				1h		
11	V	1	V	H	V	1	V	H	1v		12	H		1h	1h	H		1h	1h	H		H	1h	H		H	1h	
12	V	V	V	V	V	V	V		V		V	12	V											2h				
13	V	2	V	2	V	H	V	H	1v	H		H	12	H		H		H		H		H		H		H		
14	V	1	V	V	V	V	V		V		1v		V	12														
15	V	1	V	V	V		V		1v		1v				12													
16	V	V	V	V	V	V	V	2v	V		V		V			12												
17	V	V	V	V	V		V		1v								12											
18	V	1	V	V	V	1v	V	1v	V		1v		V					12										
19	V	1	V	V	V	1v	V		1v		1v								12									
20	V	V	V	V	V	V	V	V	V	2v	V		V							12								
21	V	V	V	V	V		V		1v												12							
22	V	V	V	V	V	V	V		V	1v	V		V									12						
23	V	1	V	V	V	1v	V		V		1v												12					
24	V	V	V	V	V	V	V	1v	V		V	2v	V											12				
25	V	V	V	V	V	1v	V		V																12			
26	V	V	V	V	V	V	V		V	1v	V		V													12		
27	V	1	V	V	V	1v	V		V		1v																12	
>27	V	V	V	V	V		V		V																			12

Figure 6.5: Updated chart of what we know about who wins Domineering on rectangular boards. New results which we have obtained from *Obsequi* are shaded. The y axis is the number of rows, x axis is the number of columns. A value such as 1h means the position is either a first player or horizontal win (further work needs to be done to determine the exact value).

Chapter 7

Future Work

There are a number of other methods which we feel have some promise in improving the search on domineering boards. Some of these are:

- A further improved evaluation function. We feel that although a significant amount of progress has been made on the evaluation function there is still room for continued improvement. For example some vulnerable moves, when they are played, create a square which is unavailable to both players. This property could possibly be taken into account. Any single new improvement will likely make a small impact on the depth of the search, but due to the large branching factor can have a significant impact on the size of the proof tree.

- A better board packing algorithm. The algorithm we used in packing different areas onto the board is basically a greedy algorithm. We know that it does not pack the properties onto the board optimally, and as the board sizes which we try to solve grow its deviation from an optimal packing will probably increase.

- Almost all search programs could benefit from a method which would increase the number of transpositions which the search tree contained (a heuristic which guides the search towards previously seen positions).

Due to its high number of transpositions Domineering could be a good test bed for this type of research.

- Finding other moves, besides safe moves, which can be proven to always be inferior to some other move. Possibly a number of patterns could be found where it can be proven that if a move matches pattern A and another move matches pattern B, A will always be at least as good as B. These patterns would help to reduce the branching factor of the search and continue to decrease the cost of node expansion. We looked at a number of these types of ideas but so far have been able to either find counter examples for the patterns or were unable to prove that the pattern was correct.

7.1 Insights into the Game of Domineering

There were also a number of ideas that we looked at, which didn't work.

One of these was building a database of combinatorial game values for small board positions, once the board is decomposed into these smaller positions we would be able to determine the value of the larger position. This problem would appear to have a large amount of potential but what we found was that our improved evaluation function was determining the value of the board before the board started to break up into non-trivial partitions. Therefore this idea was abandoned.

One more comment we have for future researchers of domineering is that we feel that solving 11×11 domineering is definitely attainable. Although to do this another large jump is going to have to be made. As processors get faster and transposition tables get larger, along with a possible 40 or 50 times reduction in the number of nodes in a given Domineering proof tree, a solution can probably be found for the 11×11 board in a reasonable amount of time.

Chapter 8

Appendix

This section contains a number of charts and graphs which show the growth of the size of the proof trees for increasing board sizes. As we can see from these graphs the size of the proof trees increase exponentially.

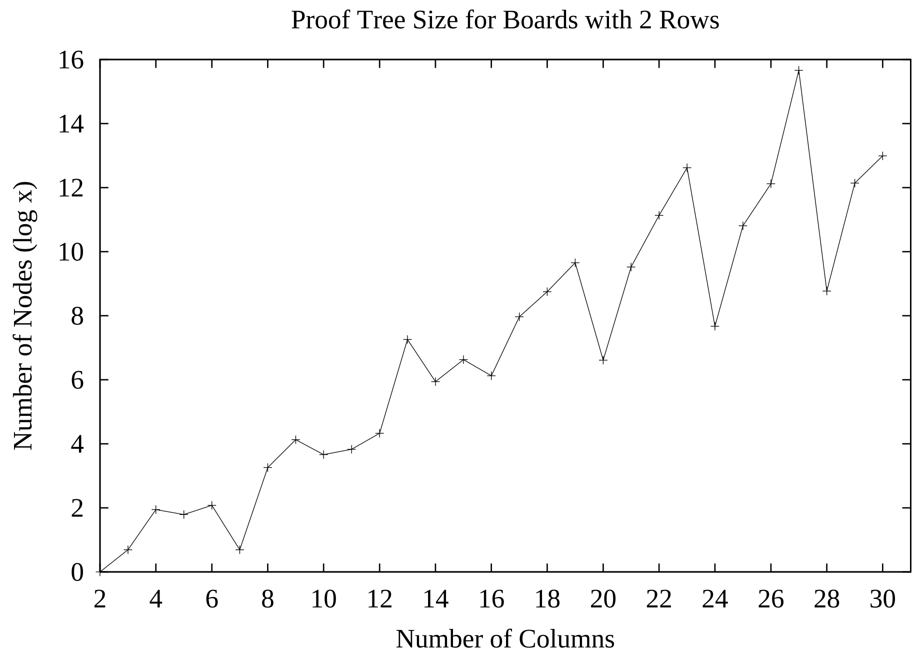

Figure 8.1: Growth of proof tree size for $2 \times n$ boards. Note the y axis uses a logarithmic scale.

Board Size	Result	Nodes	Board Size	Result	Nodes
2×2	1	1	2×17	H	2,880
2×3	1	2	2×18	1	6,317
2×4	H	7	2×19	1	15,597
2×5	V	6	2×20	H	743
2×6	1	8	2×21	H	13,628
2×7	1	2	2×22	H	68,319
2×8	H	26	2×23	1	302,707
2×9	V	65	2×24	H	2,148
2×10	1	39	2×25	H	49,373
2×11	1	46	2×26	H	183,975
2×12	H	76	2×27	1	6,342,676
2×13	2	1,417	2×28	H	6,447
2×14	1	381	2×29	H	187,416
2×15	1	756	2×30	H	438,905
2×16	H	457			

Figure 8.2: Proof tree size for various $2 \times n$ boards.

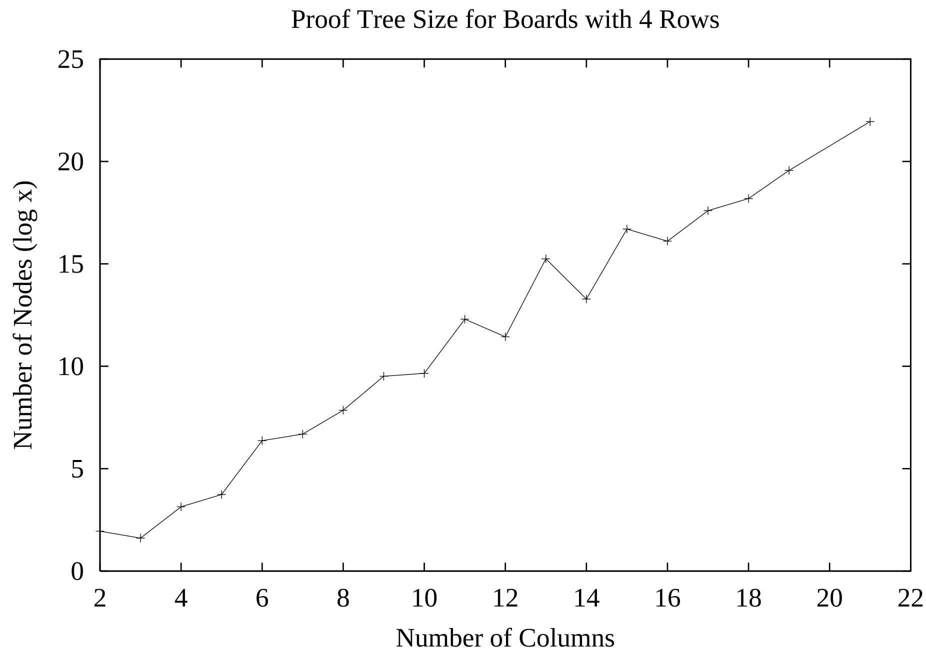

Figure 8.3: Growth of proof tree size for $4 \times n$ boards. Note the y axis uses a logarithmic scale.

Board Size	Result	Nodes	Board Size	Result	Nodes
4×2	V	7	4×12	H	92,876
4×3	V	5	4×13	2	4,177,688
4×4	1	23	4×14	H	587,431
4×5	V	42	4×15	H	17,854,350
4×6	1	583	4×16	H	9,916,799
4×7	V	802	4×17	H	43,950,177
4×8	H	2,570	4×18	H	79,398,047
4×9	V	13,549	4×19	H	314,148,901
4×10	H	15,570	4×20	H	N/A
4×11	V	218,713	4×21	H	3,390,074,758

Figure 8.4: Proof tree size for various $4 \times n$ boards. N/A denotes that this value was never computed.

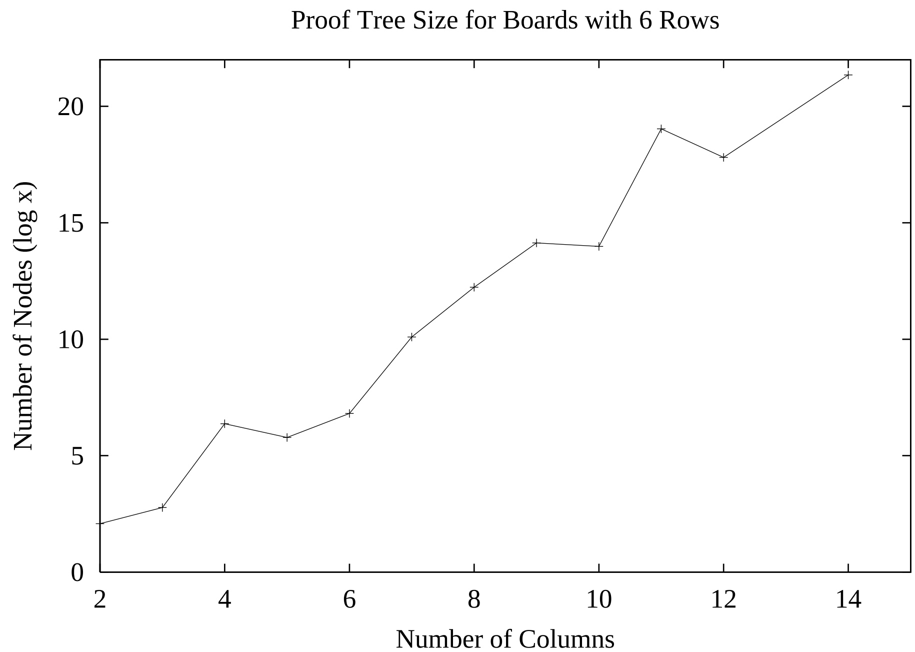

Figure 8.5: Growth of proof tree size for $6 \times n$ boards. Note the y axis uses a logarithmic scale.

Board Size	Result	Nodes	Board Size	Result	Nodes
6×2	1	8	6×9	V	1,374,535
6×3	V	16	6×10	1	1,184,184
6×4	1	583	6×11	1	184,246,797
6×5	V	324	6×12	H	54,126,298
6×6	1	908	6×13	V	N/A
6×7	V	24,227	6×14	H	1,864,870,370
6×8	H	204,813			

Figure 8.6: Proof tree size for various $6 \times n$ boards. N/A denotes that this value was never computed.

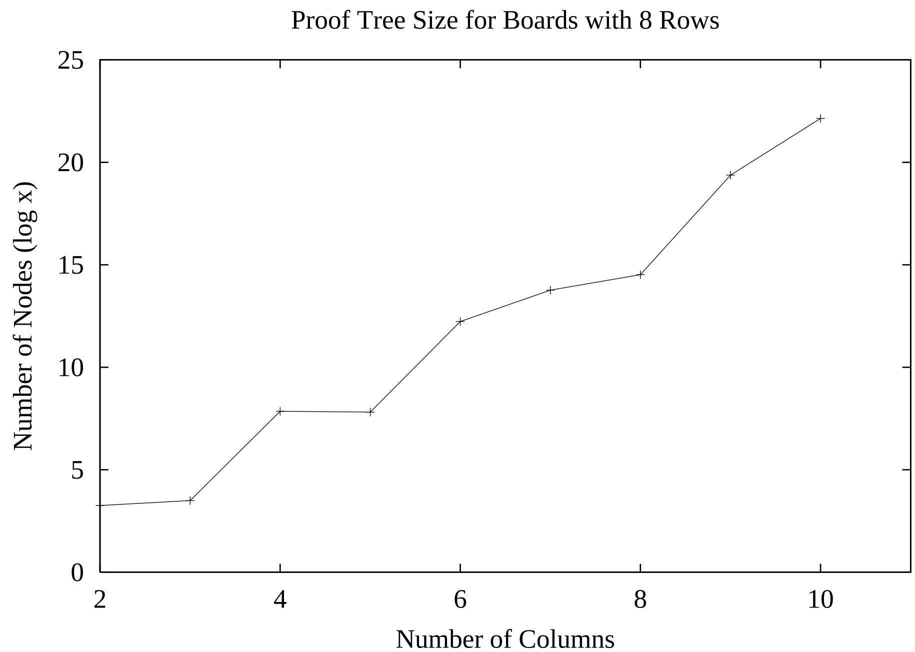

Figure 8.7: Growth of proof tree size for $8 \times n$ boards. Note the y axis uses a logarithmic scale.

Board Size	Result	Nodes	Board Size	Result	Nodes
8×2	V	26	8×7	V	947,776
8×3	V	33	8×8	1	2,023,301
8×4	V	2,570	8×9	V	259,064,428
8×5	V	2,467	8×10	H	4,125,516,739
8×6	V	204,837			

Figure 8.8: Proof tree size for various $8 \times n$ boards.

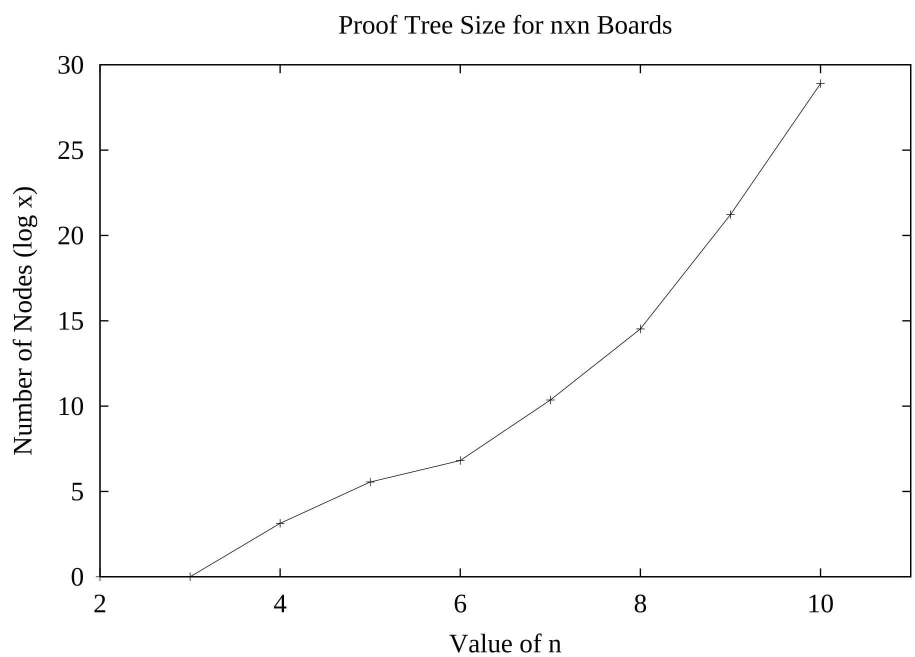

Figure 8.9: Growth of proof tree size for $n \times n$ boards. Note the y axis uses a logarithmic scale.

Board Size	Result	Nodes	Board Size	Result	Nodes
2×2	1	1	7×7	1	31,440
3×3	1	1	8×8	1	2,023,301
4×4	1	23	9×9	1	1,657,032,906
5×5	2	259	10×10	1	3,541,685,253,370
6×6	1	908			

Figure 8.10: Proof tree size for various $n \times n$ boards.

Bibliography

[1] E. R. Berlekamp. Blockbusting and Domineering. *J. Combin. Theory, Series A*, 49:67–116, 1988.

[2] E. R. Berlekamp, J. H. Conway, and R. K. Guy. *Winning Ways for Your Mathematical Plays*. Academic Press, New York, 1982.

[3] D. M. Breuker, J. W. H. M. Uiterwijk, and H. J. van den Herik. Solving 8×8 Domineering. *Theoretical Computer Science*, 230:195–206, 2000.

[4] J. H. Conway. *On Numbers and Games*. Academic Press, London, 1986.

[5] R. D. Greenblatt, D. E. Eastlake, and S. D. Crocker. The Greenblatt Chess Program. In *Fall Joint Computing Conference Proceedings*, volume 31, pages 801–810, San Francisco, 1967.

[6] M. Lachmann, C. Moore, and I. Rapaport. Who Wins Domineering on Rectangular Boards. *MSRI Workshop on Combinatorial Games*, 2000.

[7] Aske Plaat. *Research Re:search and Re-search*. PhD thesis, Tinbergen Institute and Department of Computer Science, Erasmus University Rotterdam, June 1996.

[8] Jonathan Schaeffer. The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11):1203–1212, 1989.

[9] J. W. H. M. Uiterwijk. Personal communication, 2001.

[10] J. West. Championship-level play of Domineering. In R.J. Nowakowski, editor, *Games of No Chance*, volume 29, pages 85–91. Cambridge University Press, Cambridge, MSRI Publications, 1996.

[11] A. L. Zobrist. A new hashing method with application for game playing. Technical Report 88, Computer Science Department, University of Wisconsin, Madison, 1970. reprinted in ICCA J. (2) (1990) 69-73.