Bootch: A World-Class Metasquares Player

Jeremiah Moon
Advisor: Stanley Eisenstat

May, 2002

Abstract

This paper examines the board game Metasquares, and the creation of Bootch, an automated Metasquares player. Existing automated Metasquares players are reviewed. Challenges encountered while attempting to improve Bootch, as well as their solutions, are discussed. Match results between Bootch and Anysquare, the strongest automated Metasquares player until now, are reported. Match results between Bootch and Mike Roth, the strongest human player, are also reported.

1 Introduction

Humans have a long and impressive history of inventing and playing board games. To name a few prominent examples, we have Go, invented c. 2300 B.C.; Chess, invented c. 500 A.D.; and Checkers, invented c. 1200 A.D. In addition, new board games are being invented constantly, adding themselves to this already-sizeable list. Most of them are not very successful, and are quickly forgotten. Every once in awhile, however, a new game finds success: it is embraced by an active community of players interested in playing and exploring various aspects of the game. Metasquares is one such game.
Scott Kim, a professional puzzle- and game-designer, invented Metasquares in 1996. It was featured on AOL Games from 1996-1997, during which time about 100,000 people played it. This popularity can most likely be attributed to the game’s sheer simplicity, and therefore accessibility. Despite its simplicity, however, Metasquares is strategically deep and computationally interesting. Today, Metasquares fans find each other at http://www.metasquares.com, where they use the universal game client to play each other.

2 Metasquares Game Play

Metasquares is played on a board similar to a Go board, except that it is 8×8 and not 19×19. Two players take turns placing his/her respective token (e.g. player 1 has black tokens, player 2 has white tokens), and points are earned by forming square(s) with one’s own tokens. The number of points earned equals the area of the smallest square with edges parallel to the board sides that encloses the formed square. For example, in the following diagram, the player with the black tokens has formed the square b3-c7-g6-f2:
The smallest square with edges parallel to the board sides that encloses this square is $b2-b7-g7-g2$. Here, “side length” is defined as the number of points on a line, so the enclosing square is $6 \times 6$. Therefore, the pictured square earns 36 points. In the following diagram, the player with the black tokens would form two squares by moving to $d4$ ($b5-c7-e6-d4$ and $d4-e6-g5-f3$), and would earn $2 \cdot 16 = 32$ points. However, the player with the white tokens has already moved there, successfully blocking off these two squares. The goal is to get 150 points, at least 15 points ahead of the opponent, to win.

3 Previous Work

Because Metasquares is not as well-known a game as others like chess or checkers, very little effort has been put into writing programs that play it. Specifically, three programs deserve mentioning.

The simplest program is the one included with the universal game client (available http://www.metasquares.com/msquarer.exe). This program per-
forms an exhaustive search of the game tree using a simple evaluation function, and searches to a maximum depth of 2 plies. While this provides a very quick response time, it is not a strong program, and can be handily beaten by strong human players.

A slightly more interesting program is Metagenius, written by an unknown author (available http://www.metasquares.com/metagenius.exe). This program allows the user to specify the search depth, allowing arbitrarily strong game play. No information on its evaluation function or its search algorithm is available. Unfortunately, the response time becomes prohibitively great before the program’s play becomes sufficiently strong. Whatever search algorithm this program is using, it is just not fast enough.

The most successful Metasquares program is Anysquare, written by Maze-yar Makoui (available http://www.anysquare.de). This program does not use a search algorithm, but instead uses a neural network. This results in an instantaneous response time. Further, its strength of play is on par with the best human players, making it a very successful program all-around.

4 Bootch Basics

4.1 Board Representation

There are 64 cells on the standard $8 \times 8$ Metasquares game board. Each of these cells can be empty, occupied by player 1, or occupied by player 2. Therefore, the simple scheme of using 2 bits per cell allows us to represent a game state using 128 bits, or 16 bytes. The lower bound on the number
of bits required to represent a game state is $\log_2(3^{64}) \approx 101$ bits, or roughly 12.7 bytes. The small space advantage offered by the theoretical lower bound becomes negligible, however, when byte-alignment is taken into account. Further, the extra computation necessary to pack and unpack the optimal board representation does not seem justified in this context. Therefore, Bootch uses the simple scheme to represent a game state with 16 bytes.

4.2 Square Detection

To keep track of how many points each player has, it is sufficient to detect, after each move, how many points that last move earned for that player. We can perform this in at most $63 \cdot 3$ comparisons. Suppose that player 1’s last move is at $A$. Then for each of the other 63 cells $B$, we check to see if it is occupied by player 1’s piece. If it is, then the vector $\overrightarrow{AB}$ uniquely determines cells $C$ and $D$ such that cells $ABCD$ form a square, and go around the square in that clockwise order. So, we just check to see if $C$ and $D$ are occupied by player 1’s tokens, and if they are, we count the square.

This method counts every newly-formed square once and only once. Each newly-formed square can be uniquely represented as a clockwise path that starts with the last move, i.e. $ABCD$. Since we are checking each of the other 63 cells $B$, we are guaranteed to find every new square at least once. Further, since each square has a unique clockwise path originating at the last move, we are guaranteed to find every new square at most once.

This method runs in time linear with respect to the number of cells on the board. If each player kept track of all the cells he or she is occupying, then
the running time could be reduced to be linear with respect to the number of pieces that player currently has on the board. In practice, however, it seems unlikely that this more sophisticated implementation would actually yield a significant improvement in performance. First, this method requires additional housekeeping to maintain a list of the cells we’re occupying. Also, the memory accesses required to retrieve our currently-occupied cells, then to look up $C$ and $D$ on the board, would most likely not be contiguous. Finally, what little performance advantage that remains after these overheads are taken into account would diminish as more and more pieces are added to the board. Therefore, Bootch uses the straightforward method that is linear with respect to the number of cells on the board.

4.3 Search and Evaluation

Bootch uses the basic $\alpha - \beta$ search algorithm, with enhancements to be discussed in the next section. Also, the basic static evaluator is simply the difference between my score and the opponent’s score. In the next section we also discuss enhancements to this static evaluator.

5 Bootch Enhancements

5.1 Hash Table

In a normal $\alpha - \beta$ search, a great deal of redundant computation is performed. For example, suppose we’re performing a search of depth 5 plies. In this search, we will come across the case where I first move $A$, then my
opponent moves $B$, then I move $C$. After computing this subtree, we will have to recompute it all over again for the case where I first move $C$, then my opponent moves $B$, then I move $A$. This works recursively down the tree, so a tremendous effort is wasted recomputing subtrees. We can eliminate this redundancy with hashing: anytime the value of a subtree is computed, we hash the game state represented by that subtree and insert its game evaluation into the hash table.

5.1.1 Table Type

Originally, we used chaining, dynamically allocating and freeing memory as each entry was added and cleared. Profiling showed this to be highly inefficient, as the majority of time was being spent on memory allocation. Our solution was to allocate memory for the entire table at program initialization, and then use a header table to point to our location in table. In this scheme, the hash function is used to index into the header table, which has a pointer to a location in our allocated memory. This scheme has the benefit that no time is spent on memory allocation during the execution of the program, and the drawback that the tablesize is fixed at compile-time. This drawback is not a serious problem, however, if we allocate enough space such that we will not exhaust our allocated memory for almost all cases.

The above scheme is preferable to open-addressing for a couple reasons. First, in the above scheme, we can choose a relatively large modulus for the hash function at minimal cost. Since the modulus corresponds only to the size of the header table, choosing a large modulus means allocating space
for that many more pointers and not that many more hash table entries. In our case, a hash table entry uses roughly 5 times as much memory as a pointer, making this significant. This allows us to reduce the probability of collision using minimal memory. Open-addressing requires more memory in another sense: the hash table must not become too full to maintain good performance. In our scheme, performance is not affected by the number of allocated entries used.

Originally, we used Knuth’s multiplicative method for our hash function. This provided a low rate of collision, and was satisfactory overall. However, experimentation with other various hash functions showed Robert Jenkin’s 96-bit mix function to be much better. This is mostly because because it offers a similar rate of collision, but is much faster to compute. While this function requires 96 bits of input, our game state is 128 bits. Therefore, we generate the first 32 bits of the input by taking the exclusive-or of the first 32 bits of our game state with the second 32 bits of our game state. Then the second 32 bits of input is taken from the third 32 bits of our game state, and the third 32 bits of input is taken from the last 32 bits of our game state.

5.1.2 Transformations and the Canonical Form

We can extend the idea of hashing a board position to the following: instead of inserting the game state representation directly, we can take advantage of the symmetries of the board to insert the canonical game state. For any given state, there is 8-way symmetry defined by each possible combination of {vertical mirror, horizontal mirror, matrix transposition}. These 8 candi-
dates are compared to each other with a string comparison, and the minimum is defined to be the canonical game state. This way, we are reducing the hash table size by a potential eight-fold.

After implementing this feature, experimentation revealed that reducing each position to the canonical form actually hurt performance. This is because in typical game play, symmetrical positions almost never arise. In other words, there almost always did not exist two different states reachable from the current state that both reduce to a common canonical form. Thus, we actually lost performance from the time taken to compute the canonical form. Therefore, Bootch does not look for symmetry in its use of hashing.

5.1.3 Implementation Details

Bootch does not hash the game state when the subtree has depth less than 2. Experimentation showed that storing these shallow subtrees actually yields worse performance. This is probably because they are not retrieved often enough to justify the cost of inserting them into the hash table. Not hashing the game state for subtrees of depth less than 2 had the added benefit of greatly reducing the hash table size.

5.2 Move Ordering

In order for $\alpha - \beta$ search to outperform the straightforward min-max search, we must search good moves first: if you consistently seach the worst move first, a $\beta$-cutoff is never achieved, making $\alpha - \beta$ perform no better than min-max. Originally in our $\alpha - \beta$ search, we processed the available cells in a
fixed order from left to right and bottom to top. In the average case, this puts the best move somewhere in the middle of our search, which leaves a lot of room for improvement.

Our solution was to first perform a very shallow search to determine what the most likely best move is, and to expand that subtree first. Thereafter, the remaining moves are searched in fixed order as before. In essence, this is creating an ordered list (of length 1) of the probable best moves and expanding those subtrees in that order. Increasing the length of this list beyond 1 offers significantly decreasing margin of utility, and does not seem to justify the cost of its computation. Through experimentation, we found that the following configuration works well: for each subtree of depth greater than 2 plies, first perform $\alpha - \beta$ of depth 2 plies to determine the probable best move. This significantly reduced our search time, as well as the hash table size.

5.3 Static Evaluator

Originally, we just used the difference in scores as our static evaluator. This was certainly adequate, but did not capture two other crucial elements of the game: termination distance and square valuation.

5.3.1 Termination Distance

According to our naive static evaluator, a lead of 140-120 is equally valuable as a lead of 40-20. But in reality, a lead of 140-120 is very much more valuable than a lead of 40-20. This is because in the former case, the opponent is
forced into an exclusively-defensive mode to prevent you from reaching 150 points, giving you the initiative. To take this into account, we first compute the score difference, then multiply that by some function of our distance to termination. Experimentation showed the following function to work well: 

\[(1 + \frac{p_1+p_2}{900})\], where \(p_1\) is player 1’s score and \(p_2\) is player 2’s score. In this case, a lead of 40-20 would be valued at \(20 \cdot (1 + \frac{40+20}{900}) \approx 22.33\), while a lead of 140-120 would be valued at \(20 \cdot (1 + \frac{140+120}{900}) \approx 25.78\).

### 5.3.2 Square Valuation

Suppose that my two best moves, \(A\) and \(B\), both get me 16 points, and that my opponent cannot make any squares in the foreseeable future. The naive static evaluator gives \(A\) and \(B\) equal value, and will pick whichever it encountered first. Now suppose that \(A\) is a corner square, say \(a_1\), and \(B\) is a more central square, say \(d_3\). It is widely-known that corner squares are positionally poor, because the number of squares in which they can potentially participate is relatively small. Therefore, \(B\) would most likely be the correct move in this case, despite the fact that there is no difference in scores in the foreseeable future.

Bootch accounts for this by using a static table containing the total number of points of all the squares that a given cell can participate in. For example, \(a_1\) can potentially participate in 7 squares that total 203 points, and \(d_2\) can potentially participate in 25 squares that total 488 points. \(a_1\) and \(d_2\) represent the minimum and the maximum totals, respectively, and all the other cells fall somewhere in between. We want to add some function
of this total to the product obtained in the previous section.

Originally we just divided the total by some constant, and this worked reasonably well. However, experimentation showed that it is better to divide the square of the total by a constant. Because the square valuation is intended to be a tie-breaker between seemingly equivalent moves, we want this constant to be relatively large. Empirically, a value of 60,000 seems to work well.

5.4 Opening Book

With all the enhancements mentioned so far, Bootch still did not know what to do in the opening. Because it did not see any possible squares in its search depth, it would make the same fixed moves in its search order each time for the first couple moves. This is clearly unacceptable when Bootch’s opponent is busy setting up forks! We solved this with a simple opening book.

This opening book is 4 plies deep, and attempts to set up a large fork. That is, it does not attempt to block the opponent’s fork, but rather sets up a fork of its own. It randomly chooses among 8 equivalent fork set ups. If both forks are allowed to go through, then we are tied, and that is fine. If the opponent attempts to cut off our fork, then we are out of book, and the \( \alpha - \beta \) search will cut off the opponent’s fork, and that is also fine.

Note that we are not trying to establish an opening advantage with an extensive book. Our goal is more modest: we just want to start off without a clear disadvantage. This is primarily because the time required to prepare an extensive opening book simply was not available. Also, even if the time had
been available, preparing an extensive book is unlikely to have been much of a computer science learning experience.

6 Bootch Results

An average response time of at most 10 seconds seems reasonable, and with all of the above enhancements, Bootch is able to perform an exhaustive search at a depth of at least 5 plies in this time frame. Quite often Bootch searches at a depth of 6 plies in this time frame, and towards the end of the game even a depth of 7 plies is performed. All of this was performed on a Pentium 4 processor operating at 1.7 GHz. This response time, while slower than the instantaneous Anysquare, is quite acceptable. We allocate $2^{18} = 262,144$ elements for the hash table and for the header table. This totals to $2^{18} \cdot (24 + 4) = 7,340,032$ bytes of total memory usage, which is also reasonable. Bootch competed against the best automated Metasquares player and the best human Metasquares players.

6.1 Anysquare

Bootch played a 20-game match against Anysquare, the neural network program. This posed a little bit of a problem, since both programs are deterministic: Bootch would play the same opening book each time, and they would end up playing the same identical game over and over. Therefore, we temporarily disabled the opening book, and instead had Bootch make its first two moves at random. This put Bootch at a significant disadvantage:
Anysquare almost always started off with a successful fork, and a 16 point advantage. Despite this, however, Bootch did quite well: alternating colors every game, it won 14 out of 20 games. The results are summarized in the table below:

<table>
<thead>
<tr>
<th>Game</th>
<th>Player 1</th>
<th>Score (Player 1 - Player 2)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bootch</td>
<td>170 - 133</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Anysquare</td>
<td>122 - 154</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Bootch</td>
<td>131 - 158</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>Anysquare</td>
<td>129 - 163</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Bootch</td>
<td>181 - 157</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Anysquare</td>
<td>140 - 159</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>Bootch</td>
<td>121 - 185</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>Anysquare</td>
<td>166 - 92</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>Bootch</td>
<td>156 - 138</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>Anysquare</td>
<td>143 - 158</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>Bootch</td>
<td>180 - 97</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>Anysquare</td>
<td>120 - 151</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>Bootch</td>
<td>201 - 121</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>Anysquare</td>
<td>159 - 111</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>Bootch</td>
<td>158 - 127</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>Anysquare</td>
<td>210 - 133</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>Bootch</td>
<td>176 - 118</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>Anysquare</td>
<td>154 - 66</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>Bootch</td>
<td>178 - 150</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>Anysquare</td>
<td>117 - 185</td>
<td>1</td>
</tr>
</tbody>
</table>

### 6.2 Humans

Bootch played 4 games against Mike Roth, the highest-rated human Metasquares player. Bootch did employ its basic opening book for this match. The games were untimed, with the human taking somewhat more time than Bootch. The results are summarized in the table below:
<table>
<thead>
<tr>
<th>Game</th>
<th>Player 1</th>
<th>Score (Player 1 - Player 2)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bootch</td>
<td>182 - 135</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Mike</td>
<td>174 - 97</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Bootch</td>
<td>130 - 156</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>Mike</td>
<td>155 - 88</td>
<td>0</td>
</tr>
</tbody>
</table>

After the match, Mike commented that, in his opinion, Bootch would probably beat all but the ten best humans. When asked how Bootch would do against the ten best humans, he replied that the games would probably go 50/50, with the player moving first winning each game.

7 Conclusion

This paper presented my experience writing a Metasquares player from scratch. From the basic $\alpha - \beta$ search algorithm and the simple score difference static evaluator, Bootch emerged as the result of incremental evolutionary changes. At this point, it would not be unfair to say that Bootch is probably the strongest Metasquares-playing program available, and that it is at least on par with the best human players.

While this fact does somewhat reduce the motivation to improve Bootch, there are a couple areas in which Bootch is definitely lacking. The first is the opening book. As previously stated, the current opening book attempts merely to start off without a disadvantage. A more extensive book would allow Bootch to do more than this, and seek an advantage from the opening. Second, and less definite, is improving the static evaluator. Currently, Bootch considers 3 dimensions: score difference, termination distance, and square valuation. It is not clear what other dimensions would be useful to
the static evaluator, but it seems likely that there would be many more. Finally, Bootch is deterministic, and in a long match against a single human opponent, this can lead to repetitive play. Introducing a random factor into its move selection, perhaps selecting randomly among the best two moves, may create for interesting gameplay.