And the Winner is...

Solving Triomineering using Combinatorial Game Theory

Daniel Brüggemann, Mantas Gagelas, Maarten Weber, Bas Willemse

January 20, 2016

Abstract

Games without randomness and with perfect information normally can only be solved for a specific game environment. Small changes like changing the board size or modifying the pieces used by players usually introduce new challenges for the implementation of the game solver. A precise understanding of the specific game domain and its properties is needed, and extending the domain to new areas might help to improve this understanding and reveal new techniques for a more efficient solver. We look at the combinatorial game Domineering, where one player places vertical and the other places horizontal dominoes on a board, and the player who first cannot place a new domino loses. We extend its rules by using not dominoes, but 3-tile-pieces (Triomineering), and build a solver for this variant of the game. The solver uses a combination of search heuristics and strategies from the field of Combinatorial Game Theory, especially using the representation of a board state as a combinatorial game value. We explain how to calculate these game values of subgames and how these can be used to make the solver more efficient, meaning how to reduce the number of game tree branches that need to be searched by the solver. The developed results might allow a better insight into the general domain of Domineering and how to solve it efficiently.

Contents

1 Introduction

Humans created games to entertain the mind and challenge their problem solving skills for thousands of years. However, only with the up rise of computer technology in the last decades, it became possible to create artificial players who came close to and, in some cases, beat the best human competitors. With vastly increasing computational power and increasing interest in AI due to the rise of computer games, it became possible to not only find good moves for a game, but even completely solving them [5].

Solvable games without randomness and with perfect information normally can only be solved for a specific game environment. Small changes like increasing the board size by one or modifying the pieces used by players usually introduce new challenges which might be computationally intractable with current technology.

Following this observation, the combinatorial game called Domineering is regarded, that has been solved up to a board size of 10-by-10 tiles [4], and use triominos instead of dominos to create a new game with unknown outcome for all board sizes. The aim is to solve Triomineering up to a board size as large as possible and compare the solver's results with the results from Domineering, with the expectation to gain further insight into the game's domain.

First, the rules of the original game and the extension to this version are explained. This is then followed up by introducing the use of heuristic game trees to search through the moves of a game. Afterwards, a look is taken at the properties of combinatorial games and incorporating its strategies into the search trees that are used to create an enhanced solver for Triomineering. Finally, the results are summarized and an outlook for further research areas in this game's domain is presented.

2 Domineering and Triomineering

Domineering belongs to the class of combinatorial games. It is a sequential 2-player game, and contains no randomness. Each player has perfect infor-

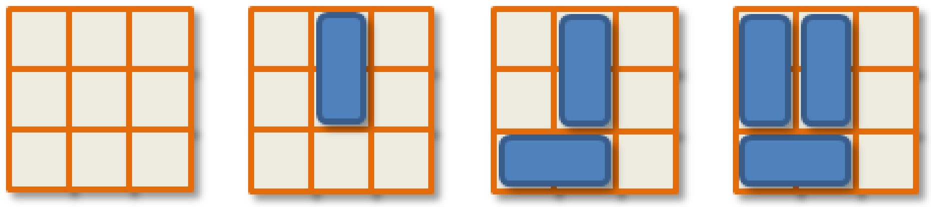

Fig. 1: Domineering on 3-by-3 board. *Vertical* blocks three possible moves of *horizontal* with his first move, gaining an advantage. *Horizontal* loses on turn 4. First player to move wins on this board.

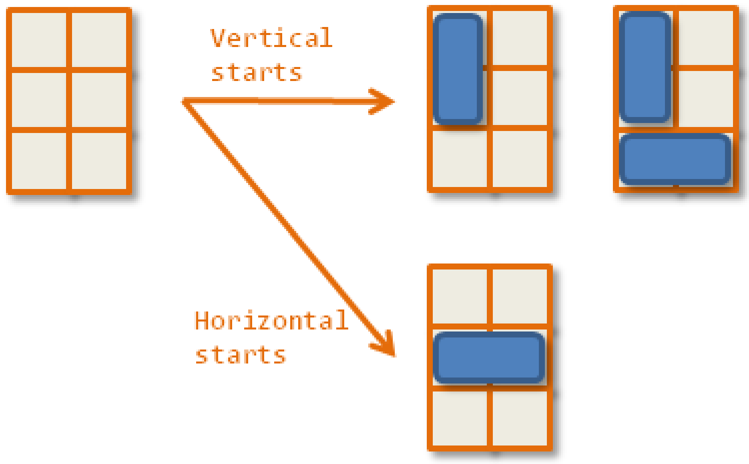

Fig. 2: Domineering on 3-by-2 board. It does not matter which player starts, *horizontal* will always win.

mation about the game state, meaning there is no hidden information. The board is rectangular, made up of equally spaced tiles and is of arbitrary size.

Domineering is a partisan game, meaning that each player has different moves available. The player usually called the "left player" or the "vertical player" can only place vertical dominos on the board, the "right player" or "horizontal player" can only place horizontal dominos. The first player not able to place a domino on the board loses the game, making the other player the winner. The vertical player always starts. As each domino reduces the number of free tiles on the board and the number of tiles is finite, this is a finite game guaranteed to end. Figures 1 and 2 show two examples of simple Domineering games played on different board shapes.

These examples also indicate the four different outcome classes of Domineering; a board can be a first-player-win, a second-player-win, a vertical-player-win or a horizontal-player-win. A draw is not possible - at some point, one player will be out of moves and loses.

Domineering has already been solved up to a size of 10-by-10 [4]. Instead of trying to solve an even bigger board size, in this project, a change to the game rules is introduced; both players now use triominos, composed of three tiles, which is why this game version is called /textitTriomineering. Even though this seems like a minor difference, it changes the outcome of certain board sizes. Figure 3 shows an example where the results of Domineering and Triomineering are not equal. A result for Triomineering can therefore not be derived with certainty from the Domineering results. This creates a new game domain which could yield interesting new results and strategies for the corresponding game solver. For the remainder of this paper, all game examples will use the game version Triomineering.

3 Methodology

Looking at Figures 1, 2 and 3, one intuitively recognizes the best move for each player, namely the one that takes the most moves away from the opponent. On bigger boards however, a strategy to determine the optimal move has to be developed.

The simplest strategy is to traverse through every possible combination of moves the two players can make in Triomineering. These moves are usually represented as nodes of a tree; the root node represents the start of the game, and each leaf node represents the end with one of two outcomes (*vertical* or *horizontal* wins). With each layer of the tree, the players alternate in placing a triomino on the board.

In Figure 4, the game tree for a 4-by-4 board is shown. In all layers, there are actually more move options, but these are removed to make a simpler tree as they create the same game situations as already shown - this is further elaborated in Section 3.1.

The game tree shows that a win for any of the two players is possible. If

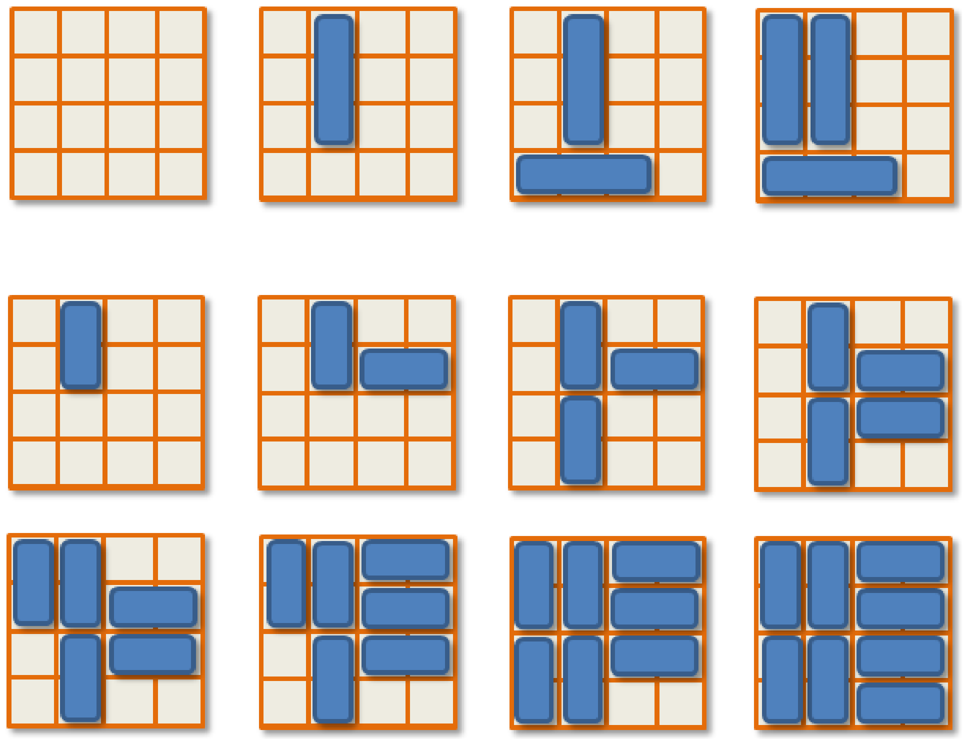

Fig. 3: Comparison between Triomineering and Domineering on the same board. While Triomineering is won on turn 4 by the first player to move, Domineering is won on turn 9 by the second player.

vertical decides to make the left move in the first layer, *horizontal* will win this game, because in the second layer it will always decide to make the right move so that *vertical* loses. However, *vertical* has the opportunity to choose the right move option in layer 1, thereby cutting off any possibility for *horizontal* to win. The conclusion is that because *vertical* starts, it wins on the 4-by-4 board. Therefore, in evaluating any game state, it is necessary to distinguish between moves that can be made if the opponent doesn't cut them off, and moves that can not be prevented by the opponent.

To decide in a more mathematical fashion which move is the optimal one for a player, a simple heuristic formula to generate a move's value is used:

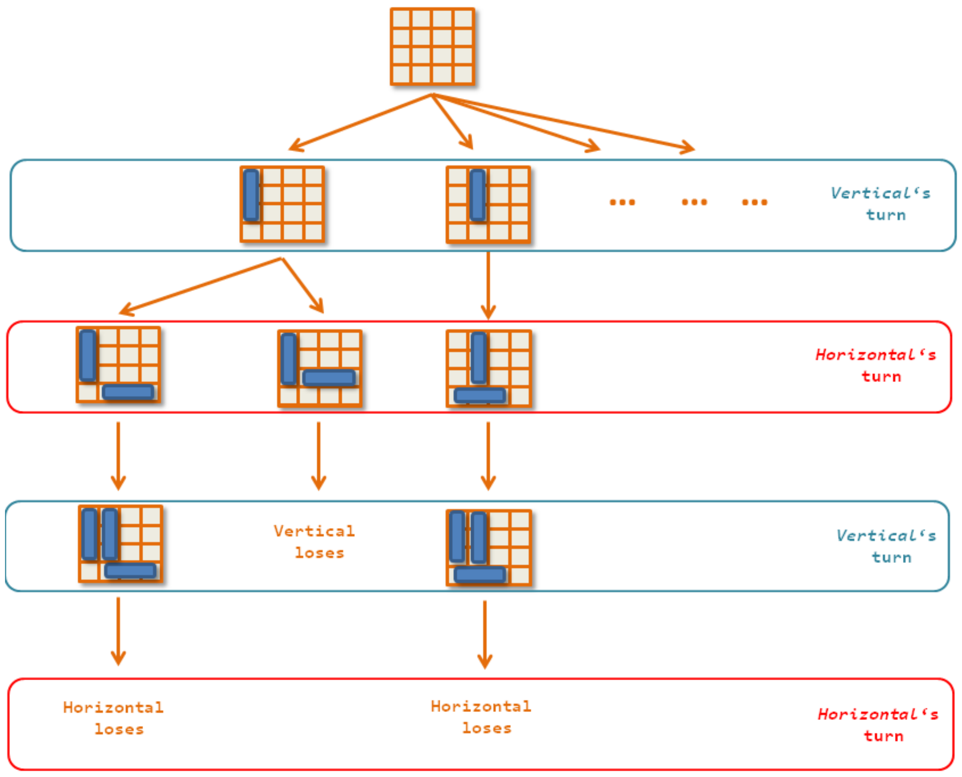

Fig. 4: Simplified game tree for a Triomineering game.

$$v(move) = \delta(m_p(move) - m_o(move)) + \delta(s_p(move) - s_o(move))$$

Fig. 5: Evaluation function used in Negamax

v(move) is the evaluation value that the move m receives. *m(move)* is the delta of the maximum number of moves possible before and after execution of move m for the respective players if the opponent would not move at all. *s(move)* is the number of safe moves for a player, meaning the moves that the opponent can not prevent by placing a tile on its own.

The optimal move is the move with the highest value. By doing this, the need to go through the whole game tree is removed, and it is possible to cut off branches; this process is called *pruning* and is of great value when searching game trees with large numbers of nodes. There are more techniques to prune a bigger percentage of the game tree. Note however that these techniques are generalized for all game trees and are not specific to Triomineering. At a certain board size and thus game tree size, general approaches are not sufficient anymore and we have to look at the specific properties of the game we are trying to solve.

3.1 Combinatorial Game Theory

Combinatorial Game Theory, or CGT, was being used in different fields for several decades, but only recently theorists started to incorporate its concepts and techniques into search algorithms to make the search for optimal game moves more efficient [2].

It is essential to understand the domain and the properties of Triomineering to make its solving process more efficient - the more efficient our solver becomes, the bigger is the board size we can solve with it in a reasonable amount of calculation time. Many properties of Domineering still hold in Triomineering.

The most important one of these properties is the independence of subgames. Looking at Figure 6, the two placed tiles divide the board into two spaces consisting of six and four tiles. Placing a tile in one of these can never influence the other space. Thus, for our game tree, we define these spaces as subgames and solve each of them individually. This vastly reduces the number of nodes that have to be searched, because instead of adding the moves of the subgames as options in the entire game tree, we look at them separately.

Splitting the whole game into subgames if possible is useful, but it also makes it necessary to change the layer system described before and always look at both player's move options in each layer. This might seem incorrect at first sight - how is a player able to move multiple times in a row? It becomes clear if we keep in mind that this is a subgame and the other player might take his turn to make a move in *another* subgame, therefore allowing the other player to move multiple times in *this* subgame.

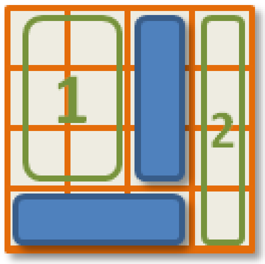

Fig. 6: Dividing the board into two subgames, denoted by 1 and 2.

CGT also introduces its own value representation. For each game state represented by a node in the game tree, we can calculate a CGT value that is similar to the heuristic value we created in section 3, but is more refined. It introduces different game value types like Numbers, Star, Up, Down, Switch etc. whose elaboration would go beyond the scope of this paper. Still, an important contribution to be gained from these value types is the distinction between *Hot Games* and *Cold Games*, describing the subgames at a certain game state.

A game is described as cold if it does not matter which player moves first because the layout of the board defines a certain advantage for one player independent of the moves available to the other one. Figure 7 shows a cold game where *vertical* is in favor - even if *horizontal* moves first, *vertical* has a "save" move. If *vertical* makes a move, it can block *horizontal*'s move, which is why it has an advantage here. Players do not want to play in cold games, because "save" moves can be done last, after all hot games are played.

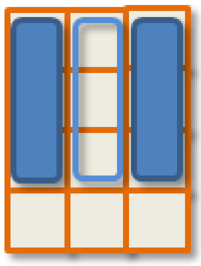

Fig. 7: Example of a cold game with advantage for *vertical*

Hot games are games where the player who moves *first* gains an advantage. Figure 8 depicts a hot game where a move from either of the players "steals" the opponent's move. Clearly, each player wants to play in a hot game first before moving to the cold games. [6, pp. 159]

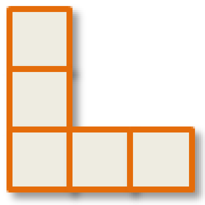

Fig. 8: Example of a hot game - both players want to move first in this game.

When all game states have a CGT game value assigned to them, the optimal next move for a player is the one in the hottest game (the game with the highest so-called *temperature*). Moves are chosen by their temperature, creating an order for the moves available in all subgames. Thereby, we are able to use subgames to efficiently prune our search tree without risking making a nonoptimal move.

3.2 CG Suite

To be able to quickly do calculations, and check if the solver that was created makes moves correctly when working with Triomineering, a library called CG Suite [1] was used.

3.3 Negamax

When applying CG Suite on the task of solving Triomineering it was noted that as the board size increased the time it took to solve the game increased at a rapid rate. To be able to solve larger boards it was clear that using purely CG Suite would not suffice and as such Negamax was implemented to aid CG Suite. This approach is based on the research done by Uiterwijk[3].

Negamax is a slightly altered version of Minimax. Minimax is an AI technique that, simplified, tries to minimize the opponent's maximum move. It is a tree-based search algorithm, that checks possible moves and tries to find the best one. It originates in zero-sum game theory, and is popular for its efficiency, ease to implement and heuristics.

Negamax functionally works the same as Minimax, but is easier code-wise. Instead of having the layers switch between maximizing and minimizing, the minimizing players' evaluation scores are negated, and all layers are maximized.

The pseudo code for the Negamax algorithm can be found in Algorithm 1.

In addition to using Negamax, Alpha-Beta pruning was implemented. Alpha-Beta pruning allows for pruning of part of the tree mid-search, by keeping track of both players' best option, and pruning parts that are guaranteed to be worse than this option[7].

When traversing through the tree, Alpha-Beta pruning will cut off any value for which the maximizing player knows he can get a higher score, and any value for which the minimizing player knows he can get a lower score. When applied to a Negamax-tree, it will return the same tree, with the same values, but with less explored nodes.

```
negamax(node, depth, alpha, beta, color)
if depth = 0 or node is a terminal node then
    return color $\times$ the heuristic value of node
end
bestValue := -$\infty$ ;
childNodes := GenerateMoves(node) childNode := OrderMoves(node) for
each child of node do
    val := -negamax(child, depth - 1, -$\beta$, -$\alpha$, -color)
    $\alpha$ := max($\alpha$, val)
    if $\beta \leqslant \alpha$ then
        break;
    end
end
Initial call for Player A's root node:
rootNegamaxValue := negamax( rootNode, depth, $-\infty, +\infty$, 1)
```

Algorithm 1: Pseudocode Negamax with Alpha-Beta Pruning(Breuker Dennis, 1998)

3.4 Transposition Tables

To reduce the time spent searching, and prune moves away, transposition tables were implemented. Transposition tables work as follows: whenever a move is made, the board state is converted to a unique hash or ID. Then, a list of previous hashes is checked to see whether this board state has come up before. If it has, then the value of this board is the same as that one (and therefore the board is the exact same), and the information that has already been gained on the other board can be used. If the hash is not in the list yet, then the new hash is added to the list, and the solver continues running as normally.

To further improve this, it is certain that mirrored board states (mirrored in horizontal and/or vertical directions) are the exact same. Because of this property, whenever a move is made, it is possible to check four different hashes in the list of known boards, and add these to the list if they're unique and new. This further reduces the number of moves that have to be investigated.

4 Experiments

Two experiments were designed to test the Solver. Firstly a test was made to determine the effect of changing the piece size. Secondly, to speed up the negamax certain additional techniques were used thus a test was made to establish how effective they were. The specifications of the machine that was used for the experiments are given in section 9.

4.1 Experiment One: Domineering versus Triomineering

The solver that was created for Triomineering can give results, determining which player wins on which board, and how many nodes need to be explored by the solver in order to get to this result. The goal of this experiment is to firstly give results for the Triomineering game, and then compare these to the results of domineering to see if there are similarities or patterns that can be recognized.

4.2 Experiment Two: Additions to the Solver

This experiment shows the improvements that were made to the standard Negamax solver. These improvements are: adding move ordering to the solver; adding transposition tables to save the previously visited board states in an effective manner; adding symmetric versions of the transposition tables; adding all of the previously mentioned improvements

4.2.1 Move Ordering

Move ordering is applied when going through the Negamax-tree. It ensures that the best moves are explored first by looking at all the child-nodes and ordering these based on the evaluation score explained in figure 3.

4.2.2 Transposition Tables

Transposition tables, as explained in section 3.4, allow pruning of moves.

4.2.3 Symmetric Transposition Tables

The symmetric property of the game can be used to add four different transposition tables per board state (as explained in section 3.4).

4.2.4 All improvements combined

This part adds all of the improvements together to see how big the difference is compared to the normal version.

5 Results

The tables below show the winners for different board sizes. For every board size, both players were tested as a starting player. The F means that the first player to move won in both instances, the S means that the second player to move won in both instances, the V means that regardless of starting player, the vertical player won, and the H means that regardless of starting player, the horizontal player won. The result obtained for the non-square boards can be translated to the flipped version of that board, i.e. 3×6 gives a horizontal player win, while 6×3 would give a vertical player win. Horizontal and vertical swap in this case, while the first and second player wins stay the same (3×5 and 5×3 have the same results).

Board Size	Result	Nodes	Board Size	Result	Nodes
3×3	F	4	6×6	F	738
3×4	F	4	6×7	V	1,659
3×5	F	6	6×8	V	2,161
3×6	H	38	6×9	F	25,253
3×7	F	62	6×10	V	106,629
3×8	V	86	7×7	F	14,753
3×9	F	36	7×8	V	34,313
3×10	F	43	7×9	H	23,050
4×4	F	4	7×10	H	338,348
4×5	F	4	8×8	S	682,867
4×6	H	24	8×9	H	30,664
4×7	H	69	8×10	H	375,244
4×8	H	150	9×9	F	948,169
4×9	H	445	9×10	F	5,411,465
4×10	H	733			
5×5	F	4			
5×6	H	27			
5×7	H	44			
5×8	H	166			
5×9	H	355			
5×10	H	721			

Table 1: Experiment One: Triomineering

Board Size	Result	Nodes	Board Size	Result	Nodes
2x2	F	1	4x7	V	1,984
2x3	F	2	4x8	H	12,024
2x4	H	13	4x9	V	45,314
2x5	V	15	5x5	S	604
2x6	F	14	5x6	H	1,500
2x7	F	17	5x7	H	13,584
2x8	H	67	5x8	H	30,348
2x9	V	126	5x9	H	177,324
3x3	F	1	6x6	F	17,232
3x4	H	10	6x7	V	302,259
3x5	H	19	6x8	H	3,362,436
3x6	H	40	6x9	V	18,421,911
3x7	H	77	7x7	F	408,260
3x8	H	74	7x8	H	12,339,876
3x9	H	99	7x9	H	320,589,295
4x4	F	40	8x8	F	441,990,070
4x5	V	87	8x9	V	70,918,073,509
4x6	F	1,327			

Table 2: Experiment One: Domineering[3]

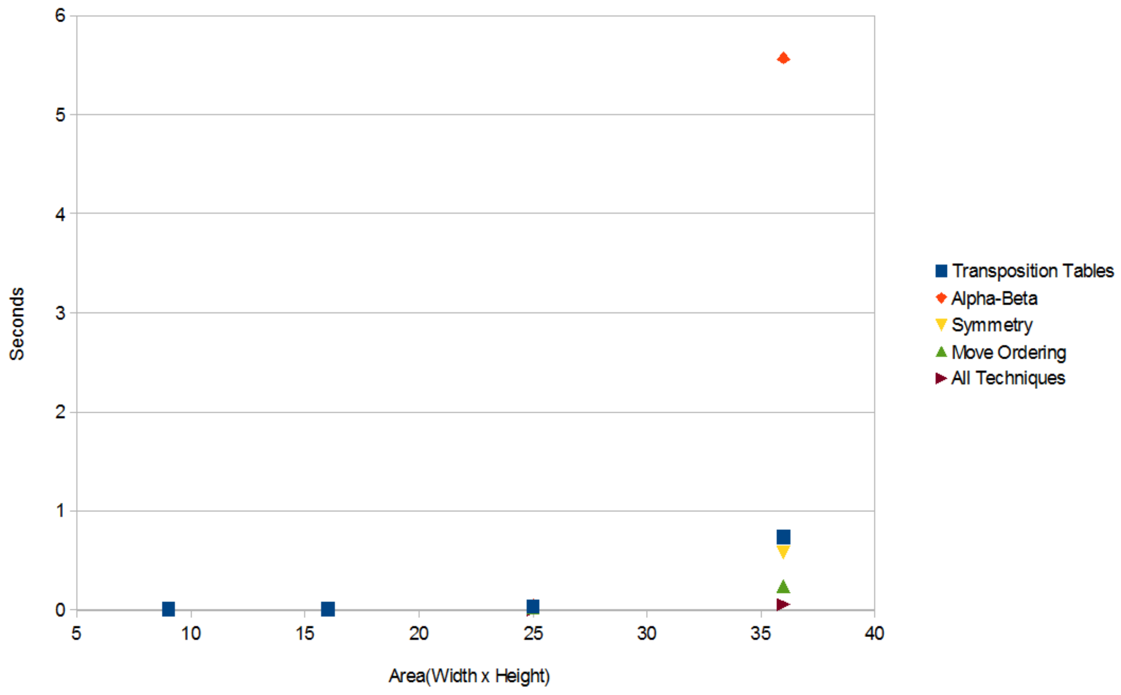

Fig. 9: Experiment 2: Running Times of Different Configurations

6 Discussion

6.1 Experiment One: Domineering versus Triomineering

When comparing the two tables to each other, a lot of similarities can be seen: the most likely players to win are either the first player to play, or the horizontal player in most cases. Then, on square boards, in every instance except for one (8×8 and 5×5 for Triomineering and Domineering respectively), the first player wins. On those specific sizes, the second player wins. It seems like the result from Domineering translates over here (for Domineering, the board width is 2.5 pieces wide, while for Triomineering, the board width is 2.667 pieces wide). Aside from that, for most sizes, it seems

like the results from Domineering, when up-scaled, translate to the results from Triomineering.
An interesting observation is that the nodes explored for Domineering scale up very explosively on the larger sizes, which is something that was not measured as extremely in Triomineering. This can be explained by the number of moves to be explored is a lot lower in Triomineering (an 8×8 Domineering board translates to a 12×12 Triomineering board). Then, there are a few board sizes in the Triomineering table where there is a low number of nodes explored (8×9 only has 30,664 explored nodes, while both 8×8 and 8×10 need a lot more nodes). This is likely because of specific shapes being formed, or because of symmetry in the transposition tables.

6.2 Experiment Two: Additions to the solver

When observing figure 9 it is clear which techniques helped the most and which suffered the most as the board size increased. In appendix B a table can be found that specifies the time taken for each run. Without any added techniques the alpha-beta solver took 5.569 seconds to solve a 6 by 6 board. With the addition of the transposition table this time was cut down to 0.737 seconds. With the addition of symmetry this fast time could be decreased even further to 0.576 which is around a 20% speed up. Symmetry has the impact it has due to all played pieces being "identical" and thus many board states are actually identical but are just flipped.
The speed up provided by the transposition tables seems impressive but pales in comparison to the speed up provided by the move ordering technique which reached a time of 0.239 seconds on a 6 by 6 board. As the size of the game board increases the branching factor has a clear impact on the speed of solving the game and thus being able to do the best move first will be highly useful.
The final observation that should be stated is that none of these techniques could solve an 7 by 7 board on their own in less then 5 minutes but when combined they were able to solve a 9 by 9 board in 61 seconds.

7 Conclusion

This paper introduced the game domain of Triomineering and proposed a combination of techniques to solve it, using search tree heuristics on one side and CGT properties on the other. CGT values are more specific to the game domain and thereby a powerful tool to create a more efficient game solver for Triomineering.

The results of this paper contribute solver results to a game variation not explored thoroughly yet. Extending the game domain of Domineering is a valuable way of gaining new insight into and a different view on known techniques and may help to find novel techniques to make solvers on this domain more efficient.

8 Future Work

While combinatorial game values calculated by the Combinatorial Game Suite were used to determine the correctness of the solver's results and were partly used in the solver's search strategy, actually incorporating these values for sub games in the search to increase the solver's speed still needs to be implemented. The challenge here is to combine the different solver prunings with a way of adding the game values of different sub games that is still proven to create correct results.

Different techniques were elaborated on. There are still more optimizations possible, the most notable ones being endgame databases and replacement schemes. Endgame databases can be used to compute the value of small sub games beforehand that are repeated many times in the game tree to avoid recalculating them every time they occur. In contrast to transposition tables, which are built up from scratch every time a search is started, the endgame values could be stored in an external file or database and be loaded into the cache by the solver upon start of the program. Replacement schemes are used to handle collisions, which can occur if two different boards have the same hash values; different strategies have to be examined for that to find the most efficient one.

References

[1] Combinatorial game suite. `http://sourceforge.net/projects/cgsuite/`. A. Siegel.

[2] M. Barton. Incorporating combinatorial game theory into an alpha-beta solver for the game of domineering. Master's thesis, Maastricht University, 2014.

[3] H.J. van den Herik D.M. Breuker, J.W.H.M. Uiterwijk. Solving 8 x 8 domineering. (230).

[4] G. B. Hinckley and N. Bullock. *Domineering: Solving Large Combinatorial Search Spaces*. University of Alberta, 2002.

[5] H.J. Van den Herik J.W.H.M. Uiterwijk and L.V. Allis. A knowledge-based approach to connect-four. the game is over: White to move wins! Technical report, 1989.

[6] R. Nowakowski M. Albert and D. Wolfe. *Lessons in play: an introduction to combinatorial game theory*. CRC Press, 2007.

[7] P. Norvig S. J. Russell. *Artificial Intelligence: A Modern Approach (3rd ed.)*. Pearson Education, 2010.

9 Appendix A. System Specs

All experiments were done on the following system:
Operating System: Ubuntu 14.04 64-bit
Ram: 7.6GB
Processor: Intel Core i7-3630qm CPU@2.4GHZ*8
Program: NetBeans 8.1

10 Appendix B. Experiment B Table

Area	Alpha-Beta	Transposition Tables	Symmetry	Mover Ordering	All Techniques
9	0.004	0.004	0.002	0.004	0.004
16	0.006	0.004	0.005	0.005	0.003
25	0.055	0.032	0.023	0.012	0.005
36	5.569	0.737	0.576	0.239	0.053
49	*	*	*	*	0.465
64	*	*	*	*	14.395
81	*	*	*	*	61.005

* implies run took longer then 5 minutes

Table 3: Experiment Two: Time Taken Per Area(Seconds)