Solving Dots-And-Boxes

Joseph K Barker and **Richard E Korf**
{jbarker,korf}@cs.ucla.edu

Abstract

Dots-And-Boxes is a well-known and widely-played combinatorial game. While the rules of play are very simple, the state space for even very small games is extremely large, and finding the outcome under optimal play is correspondingly hard. In this paper we introduce a Dots-And-Boxes solver which is significantly faster than the current state-of-the-art: over an order-of-magnitude faster on several large problems. Our approach uses Alpha-Beta search and applies a number of techniques—both problem-specific and general—that reduce the search space to a manageable size. Using these techniques, we have determined for the first time that Dots-And-Boxes on a board of 4×5 boxes is a tie given optimal play; this is the largest game solved to date.

Introduction

Dots-And-Boxes is a combinatorial game popular among children and adults around the world. It is easily played with pen and paper and has extremely simple rules. Despite its apparent simplicity, a vast number of unique games can be played on even a very small board.

In a game of Dots-And-Boxes, the players draw a rectangular grid of dots and take turns drawing lines between pairs of horizontally- or vertically-adjacent dots, forming boxes. A game's size is defined in terms of the number of boxes, so a 3×3 game has nine boxes. A player captures a box by completing its fourth line and initialing it, and then must draw another line. After all lines on the grid have been filled in, the player who has captured the most boxes wins. A player is *not* required to complete a box if they are able to do so.

As a two-player, perfect-information game, it is natural to ask what the game outcome is if both opponents play optimally. This is called *solving* the game. Due to the very large size of Dots-And-Boxes state spaces, only games as large as 4×4 boxes have previously been solved (Wilson 2010).

This paper presents a solver for Dots-And-Boxes that can determine the value of the largest-solvable games over an order of magnitude faster than the previous state-of-the-art solver. We use Alpha-Beta minimax search and a number of domain-specific enhancements. Many of these techniques are drawn from existing literature but are generally unusable by the previous state-of-the-art solver. In addition, there has been little discussion on use of these techniques in the context of computational search. We present the first thorough discussion of these techniques and their effectiveness.

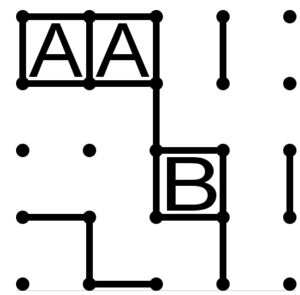

Figure 1: A 4×4 game of Dots-And-Boxes in progress. A has captured two boxes to B's one and has the lead.

We also discuss the use of some generic search techniques in Dots-And-Boxes. These techniques are commonly used in heuristic search, but there are non-obvious adaptations of them to the domain that greatly improve their effectiveness.

In addition to outperforming the state-of-the-art solver on several benchmark problems, we have solved the game on a board of 4×5 boxes. This is the largest game solved to date, and is a tie given optimal play by both opponents.

Problem Overview

Depending on how we are exploring Dots-And-Boxes, there are two ways we can describe a game "state". While playing a game, it is irrelevant who has captured which boxes; all that matters is the edge configuration and *how many* boxes have been captured by each player. This representation we refer to as a *scored* state. However, we note that, no matter the score, an optimal strategy for a player is to maximize the number of remaining boxes they can capture. Thus, the optimal strategy at any point depends only on the configuration of filled-in edges and not the score. A state that encodes only which edges are filled in we refer to as an *unscored* state.

While the rules are simple, the state space of games on even small boards is very large. An $m \times n$ game has $p = m(n+1)+(m+1)n$ edges and 2^p possible unscored states, as any combination of filled-in lines is a legal state. Even worse, a legal game can be played by filling in edges in any order. A naïve, depth-first exploration would thus generate $p!$ states, mostly duplicates reached through different orderings of the same moves. Without detecting and pruning duplicate states, the problem quickly becomes unsolvable. For example, the largest problem previously solved—the 4×4 game—has 40 edges and thus has a state space of 2^{40} and a naïve search space of $40!$. Symmetries of the board (more fully discussed later) can be used to reduce the state space somewhat, although only by a constant factor.

Dots-And-Boxes is *impartial*, which means that the set of available moves depends only on the board configuration and not who the current player is. This is as opposed to a partial game like Chess, where each player can only play pieces of a certain color. Most impartial games use the *normal play* convention, where the last player to move wins; these games can be efficiently solved by use of the Sprague Grundy Theorem (Berlekamp, Conway, and Guy 2003). Since players win at Dots-And-Boxes by having the highest score, however, this theorem is not applicable; this makes Dots-And-Boxes somewhat unusual as an impartial game.

There are two senses in which one can solve Dots-And-Boxes. One can either ask whether a player has a winning strategy (i.e., can guarantee they capture more than half the available boxes), or one can ask *by what margin* a player wins in an optimal strategy (i.e., how many more boxes than their opponent they can capture). For example, 3×3 Dots-And-Boxes is a win for the second player by three boxes: no matter the first player's strategy, the second player can ensure that they capture at least three more boxes. In this paper, we address the second question.

Due to the regularity of the Dots-And-Boxes problem-space graph, and the fact that we are not finding a binary win/loss value, we found the commonly-used Proof-Number Search (PNS) (Allis, van der Meulen, and van den Herik 1994) to not be effective in this domain. Instead, we use the standard Alpha-Beta minimax search algorithm. We do a complete Alpha-Beta search of the search graph, establishing the exact win margin of the game. We solve the win margin primarily because this is the approach of the previous state-of-the-art solver; it also provides more information without significantly impacting runtime (discussed further under *experiments*).

Previous Work

A number of books (Berlekamp 2000; Guy 1991; Berlekamp, Conway, and Guy 2003) discuss strategies for playing Dots-And-Boxes, but not solving it. In addition, there are a number of strong Dots-And-Boxes agents (Grossman 2010; Roberts 2010) that can play competitively, but not solve larger instances. The existing state-of-the-art solver was written by David Wilson (2010) and has solved the 4×4 game as well as a set of previously unsolved, partially-filled-in 5×5 games from (Berlekamp 2000). Wilson has provided us with his source code and so we use his actual implementation for comparison.

Wilson's solver uses retrograde analysis (Ströhlein 1970; Thompson 1986). For every unique game state it finds the number of remaining boxes capturable through optimal play by working backwards from the final state. The optimal strategy for a given state is determined by looking at the already-computed values of its successors and picking the optimal move. The algorithm starts at the final game state, in which all edges are filled in. It then determines the values for every state with all but one edge filled in, considering the move leading to their single successor (the final game state); this last move will always be a capture, leaving the predecessors with a value of one or two. The algorithm proceeds in this way, finding the value of states with n edges using the values of states with $n + 1$ edges, until it reaches the starting state with no edges filled in, at which point it knows the value of the entire game. Since it is working backwards, the solver cannot know which already-captured boxes belong to whom, and is thus solving unscored states.

Duplicate states have the same number of edges and thus occur at the same search depth; as this approach generates each state at a given depth exactly once, it guarantees that exactly the 2^p unique states (ignoring symmetries) are generated instead of the $p!$ states of a naïve search.

However, as retrograde analysis works backwards from the end state it cannot know *a priori* which states are part of an optimal strategy and must do a complete exploration of the problem space; 2^p is less than $p!$ but still a very large number. In addition, this approach has very large storage requirements: at a minimum, all nodes at a given depth must be generated (and stored) before nodes at the preceding depth can be considered. Wilson's solver uses disk storage, which mitigates this problem somewhat, but even this approach quickly reaches practical limitations. The 4×5 problem, with 49 edges and 5.6×10^{14} states, has $\binom{49}{25}$ nodes in its widest layer; even after accounting for symmetries, this problem is 1,024 times larger than the 4×4 problem. The 4×4 game takes 11,000 seconds with Wilson's solver; the 4×5 problem would take 8 terabytes of disk space, 9 gigabytes of RAM and—assuming runtime scales linearly with the size of the state space—130 days to solve.

Alpha-Beta minimax search is a very well-known game-solving algorithm. It performs a depth-first search of the search space while maintaining local lower (alpha) and upper (beta) bounds on the values a subtree can have that could affect the minimax value of the root. Any subtree whose value is proven to fall outside this range can be eliminated without completely exploring it. In contrast to retrograde analysis, then, Alpha-Beta can prune irrelevant states from search and avoid exploring the entire search space. However, as a depth-first algorithm, it cannot easily detect if a newly-generated state has been previously seen and may do redundant work to determine the new state's value.

Techniques Applied

Chains

In most games, states exist whose optimal moves are easy to determine. For trivial examples, consider games like Tic-Tac-Toe or Connect Four where one wins by having a certain number of pieces in a line. If the current player can make a move that completes a winning line, or prevents the opponent from completing a winning line on their next move, then the optimal move *must* be to fill in that position.

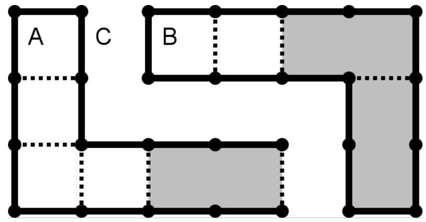

Figure 2: Examples of chains

In Dots-And-Boxes, similar situations arise in states with *chains*, which are sequences of one or more capturable boxes. Examples of two chains are shown in figure 2. If only one end of a chain is initially capturable (i.e., is a box with three edges filled in), we call it *half-open* (labeled A in figure 2). If both ends are initially capturable, it is a *closed* chain (labeled B in figure 2). Most of the moves on a state with chains can be provably discarded as non-optimal, significantly reducing the branching factor.

In a state with a half-open chain, only two move sequences can possibly be part of an optimal strategy: capture every available box (and then make another move), or capture all but two boxes and then fill in the end of the chain—leaving two capturable boxes for the opponent. The remaining configuration of two boxes capturable with a single line is called a *hard-hearted handout*. For the half-open chain labeled A in figure 2, the moves required to leave a hard-hearted handout (colored gray) are shown as dotted lines.

The possibly-optimal moves in states with a closed chain are similar: capture every available box (and then make an additional move), or capture all but *four* boxes and fill in the edge that separates them into two hard-hearted handouts. The chain labeled B in figure 2 is closed; dotted lines show the moves required to leave two hard-hearted handouts.

In states with more than one chain, we can completely fill in all but one of the chains and follow the appropriate strategy for the last-remaining chain. In these cases, a half-open chain should be left for last, if possible, as this requires sacrificing only two boxes when leaving a hard-hearted handout.

For an intuition of these rules, refer to figure 2. One option for the current player is to capture all available boxes in chains A and B; she must then make one additional move in region C which will leave all six remaining boxes to be captured by the opponent. This results in a final score of 12-6 in favor of the first player. Alternatively, the first player could leave the hard-hearted handout in A for the opponent but capture all remaining boxes. The opponent's best response would be to capture the two boxes in the hard-hearted handout and then make a move that would leave the boxes of region C capturable by the *current* player. This strategy would result in a score of 16-2 in favor of the first player.

These rules are most thoroughly described (with a proof sketch of their validity) in (Berlekamp 2000).

Our solver handles chains with a preprocessing step. When expanding a node with chains, we first capture all of the boxes that are provably part of an optimal strategy using the preceding rules. If this results in the option of leaving a hard-hearted handout, our solver only considers the two possibly-optimal options: capture the handout (and then make another move) or leave the handout for the opponent. Note that in the second case the opponent's optimal strategy will also be to capture the handout; this means that both options in fact result in our solver considering the same state, but with a different player to move. This strategy effectively collapses consecutive moves into a single compound move and reduces the overall branching factor of the problem space, at the cost of more expensive node expansions.

Transposition Tables

Transposition tables are a well-known technique for reducing duplicate work in depth-first searches. A transposition table is a cache of explored states that associates with each stored entry its backed-up minimax value. If a newly-generated state has been previously explored its stored value can be retrieved, avoiding the duplicate work of determining its value a subsequent time.

In most games, the identity of the current player must be stored in each transposition-table entry, as the optimal strategy (and hence the value of the board) depends on which player's turn it is. This means that the same board configuration can potentially be stored twice; once for each player to move. Since Dots-And-Boxes is impartial, each state has the same optimal strategy regardless of the current player and we do not need to encode the current player in our entries. This results in a space reduction. It also makes individual entries in the table more powerful than in other domains, as they can match more states in a search. In particular, it is possible to prune a node as a duplicate even if that state has never before been explored with the current player to move.

A more subtle detail arises from our choice to solve the margin of victory, rather than a binary win/loss value. If we solve the win/loss value, we will never compute the exact margin of victory of any particular node and cannot store this value in the transposition table. Instead, we can only store whether the state was a win for the current player *given their score when it was explored*; thus, entries must store the current player's score and whether the state is a win or a loss given that score. This restricts the power of the transposition table. Consider a stored entry that labels a board a win for the current player given a particular score. If we explore that state with a lower score for the current player we cannot prove it a win, since the lower current score results in a lower final score in optimal play and the entry does not encode by how much the current player can win.

A solver that computes the win margin, however, determines the number of remaining incomplete boxes capturable in optimal play; this information is useful regardless of the current score. This means that a stored transposition-table entry can be used for any state being explored, regardless of the current-player's score. This makes transposition table entries in a margin-of-victory solver more powerful than the equivalent entries in a win/loss solver. In addition, the transposition table can store more entries in memory, as entries need not store the score so far.

These facts help explain the counterintuitive fact that finding the margin of victory can be done in comparable time to computing a simple win/loss value, even given that we are solving a strictly harder problem.

Finally, we note a non-standard technique we use in encoding our transposition-table entries. In general, a table entry can store a bound on the minimax value rather than the exact value itself. If a stored value is exact, the current node can be pruned without searching; otherwise, we can use the stored value to tighten the alpha or beta bounds when searching beneath the current state.

Surveying the literature, we find that by far the most common technique for storing minimax values in a transposition table is to store two fields: a minimax value and a flag indicating whether that value is an upper bound, a lower bound, or exact. An alternative approach is to store *both* an explicit lower and upper bound, with equal bounds implying an exact value. The latter technique captures strictly more information; however it requires a bit more space and is only valuable in cases where a search can generate both upper and lower bounds on the minimax value of the same node. This happens relatively rarely in most domains, making this technique not worth the additional space requirements (Breuker 1998); this opinion is supported by the very infrequent discussion of this technique in the literature.

In Dots-And-Boxes, however, we found the technique of storing two bounds to be noticeably more effective than storing a single bound and a flag, providing a uniform decrease in search time despite the greater space requirements; this is somewhat surprising. While we have not verified this, we speculate that the reason for its effectiveness comes from the impartiality of Dots-And-Boxes. Due to chains, it is common for alpha-beta to encounter two descendants of a node with identical boards but different players to move. In these cases, it is plausible for the minimax value to fall below the alpha bound in one case (producing an upper bound) and above the beta bound in the other (producing a lower bound).

Symmetries

There are a number of trivial symmetries in Dots-And-Boxes that reduce the problem space. The mirror image of a state is also a legal game whose optimal strategy mirrors that of the current state. All Dots-And-Boxes instances have horizontal and vertical symmetry, and square boards have diagonal symmetry. We store canonical representations of states in the transposition table so that all states that are identical under symmetries map to the same entry. These symmetries reduce the size of the search space by a factor of 4 on most boards and a factor of 8 on square boards.

We make use of an additional, non-obvious symmetry to further reduce the size of the search space. We observe that, for purposes of strategy, the two edges that make up any corner of a Dots-And-Boxes board are identical; that is, given a board with a pair of unfilled corner edges, filling in either edge results in states with identical minimax values.

To understand this property, consider an alternate representation of Dots-And-Boxes called Strings-And-Coins, which is played on graphs. Boxes in Dots-And-Boxes become nodes (“coins”) in Strings-And-Coins; lines separating boxes become edges connecting nodes (“strings”) to each other (or to a special “ground” node, for nodes at the edge of the board). Players take turn cutting strings; if all four strings attaching a coin are detached, the player pockets the coin and takes another turn. Figure 3 gives an example of equivalent states in Dots-And-Boxes and Strings-And-Coins.

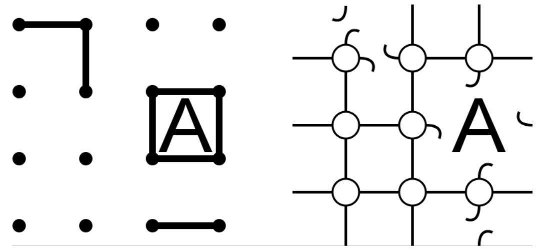

Figure 3: Two equivalent game states, in Dots-And-Boxes (left) and Strings-And-Coins (right) notation. Player A has captured one box (or, equivalently, one coin).

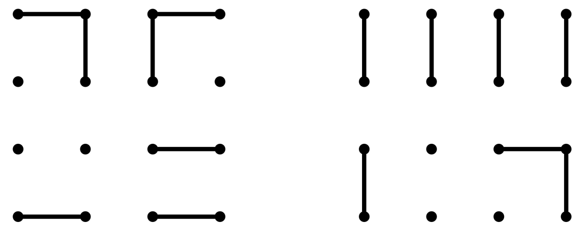

Figure 4: Two equivalent Dots-And-Boxes states. They differ only by which of pairs of corner edges have been filled.

Consider the two strings connecting a corner coin to the “ground”: if we cut either string the resulting graphs are isomorphic. As such, the optimal strategy of play on either one must be the same, and the states are duplicates. An example of corner-edge symmetry is given in figure 4.

For our purposes, this technique allows us to reduce the branching factor of nodes that have a pair of unfilled corner edges; for these states we need only consider filling in one of the two corner edges, as filling in the other results in a duplicate state. A simple way to do this is to consider filling in the topmost edge of a corner pair first and only consider the bottommost edge in states where the topmost edge is already filled. When using this technique, however, note that using reflections to generate canonical representations for the transposition table becomes complicated. For example, a diagonal reflection of a state with only a top corner edge filled results in a state with only a bottom corner edge filled. Such a state could never be reached in search and would thus not be a valid canonical representation.

Move Ordering

The order that Alpha-Beta explores children of a node strongly influences the amount of work required to determine its value. An effective *move-ordering heuristic* sorts moves in decreasing order of value for the current player, under the intuition that making stronger moves first will tighten search bounds for later moves, creating more search cutoffs.

An obvious heuristic would be to consider capturing moves first; however, all such moves are part of chains and are dealt with by the rules of the previous section. Thus, our move ordering only considers non-capturing moves. Of those, the heuristic considers moves that fill in the third edge of a box last, as they leave a capturable box for the opponent. The remaining moves are explored by considering edges in an order radiating outwards from the center of the board.

This is a very effective heuristic, despite its extreme simplicity. On the 4×4 solution, for example, this approach reduced the runtime by a factor of 17 over a simple left-to-right, top-to-bottom move order.

Verifying Correctness

We tested correctness using two significant values: the margin of victory given optimal play and the optimal opening move. For each problem considered in *Experiments*, we generated these values using Wilson's solver, as well as our solver with each search feature described. The result was the same in all cases, giving us a high degree of confidence that the proofs generated by our solver are correct.

Experiments

We conducted a number of experiments on Dots-And-Boxes to quantify the contribution of our search enhancements. We recorded the run time against 30 benchmark tests and then removed each enhancement individually, generating the same data to see its relative contribution. We include the runtime of Wilson's solver for comparison. These experiments were performed on a 3.33 GHz Intel Xeon CPU with 4 GB of RAM allocated for the transposition table.

Our results present only timing information and not the numbers of nodes expanded. This is because the concept of "expanding a node" differs in several of these techniques; evaluating the children of a node in Retrograde Analysis is very different than in Alpha-Beta (even ignoring the cost of disk I/O). This is true even within our various Alpha-Beta implementations. For example, the constant factor required to analyze chains at each state greatly increases the time per node, but results in dramatically fewer node expansions.

We performed our experiments on 18 problems from (Berlekamp 2000); these are the rows labeled 2B through 19B in table 1 (B for Berlekamp). In addition, we considered empty boards of various dimensions, shown in rows labeled 3×3 through 4×4 in table 1. These boards are provided in order of increasing number of edges.

The columns of table 1 show summarize our results. Column 2 gives the runtime or our complete Alpha-Beta solver. Column 3 gives the runtime of Wilson's retrograde analysis solver. Column 4 shows the speedup of our solver, given as a ratio of column 2 divided by column 3. The remaining columns show the result of disabling features in our alpha-beta solver, given as a ratio of the runtime with the feature removed divided by the runtime with all features enabled (column 2). We disabled chain analysis (column 5), causing the solver to consider all moves on boards with chains (not just provably-optimal moves). We tested the contribution of our center-biased move-ordering heuristic by comparing it to a left-to-right, top-to-bottom move ordering (column 6). We ignored corner-edge symmetry (column 7), requiring the search to consider both moves that fill in a pair of corner edges (even though these result in duplicate states). We modified the transposition table to store only a single bound (along with a flag) as opposed to an upper and a lower bound (column 8). We considered a partial transposition table, where stored entries only match the current state if they share the same player to move (column 9). Finally, we solved the simple win/loss value of the game rather than the win margin (column 10).

While the contribution of each technique is not uniform on all problems, there are some clear trends. Of all the techniques considered, filling in chains most consistently produces an improvement on our benchmarks—producing over an order-of-magnitude improvement on most of them. Chains arise very often in any game of Dots-And-Boxes, so it is not surprising that this technique provides such a big improvement, despite the overhead required.

Three of the remaining techniques—accounting for corner symmetries, storing two bounds in the transposition table, and using an impartial transposition table—provide relatively uniform improvements across the benchmarks. Corner symmetries speed up search by a factor of two to four, while the two transposition table techniques each contribute up to a factor of two speedup in general. Each of these techniques provides strictly more information to the search than their alternative at essentially no cost, so it is not surprising that they consistently improve performance. However, the margin by which they do so is worth noting.

The most interesting anomaly comes from our move-ordering heuristic. On larger games played on empty boards, our center-biased move-ordering heuristic is extremely effective, reducing the runtime by a factor of 17 on the 4×4 game; on other benchmarks, however, the improvements are much more modest. On those games, the board is not initially empty and a significant number of the opening moves create capturable boxes for the opponent and would be considered last by the move ordering; this would have the effect of essentially ignoring the center-outwards ordering of moves, and may explain its marginal contribution.

By far the most surprising result comes from solving the win margin of the game, rather than the win/loss value. On most of the benchmarks (apart from the anomalous twelfth Berlekamp problem), computing the win margin has only a small effect on performance: in general performance is somewhat degraded on Berlekamp's problems and somewhat improved on the empty boards. While these results are mixed, the two solution approaches are comparable in speed, making it a reasonable choice to solve the more informative win-margin problem. Given that solving the win margin is strictly more informative than the win/loss value, it is very unexpected that it can be found in comparable time, and even more surprising that the win margin is easier in some cases.

Finally, we have solved the 4×5 Dots-And-Boxes game, determining it to be a tie given optimal play. This is the first time this game has been solved and it is the largest Dots-And-Boxes game solved so far. The search took ten days to complete on a 3.33 Xeon with a 24GB transposition table. Our solver completed over ten times faster than our conser-

Problem	$\alpha\beta$, All Features	Retrograde Analysis	$\alpha\beta$ Speedup	No Chains	Left-Right Move Order	No Corner Symmetry	One-Bound TTable	Partial TTable	Win/Loss Search
3x3	0.05s	1.62s	x32.48	x2.60	x2.00	x3.58	x1.80	x1.40	x0.40
1x8	0.08s	2.24s	28.03	3.25	5.50	4.13	1.63	1.25	1.25
2x5	0.32s	4.03s	12.59	2.75	3.62	3.65	1.94	1.12	1.34
1x9	0.20s	13.03s	65.13	6.35	8.65	2.34	2.25	0.85	0.65
1x10	4.92s	107.56s	21.87	3.12	5.17	2.81	2.86	0.71	1.10
3x4	1.46s	39.39s	27.00	3.68	10.02	3.86	2.06	1.17	1.75
2x6	12.00s	72.02s	6.00	3.05	2.36	3.90	1.87	1.10	1.52
1x11	5.60s	980.34s	175.09	10.04	13.72	2.44	2.36	0.88	0.73
1x12	268.88s	7995.39s	29.74	7.44	12.39	3.26	4.98	4.91	5.85
2x7	143.88s	2450.13s	17.03	5.97	3.54	3.54	2.26	1.22	3.46
3x5	310.18s	4988.18s	16.08	10.62	4.03	4.59	1.96	1.50	1.14
4x4	274.44s	10977.38s	40.00	6.33	17.19	3.88	2.10	1.70	4.48
2B	20.40s	296.45s	14.53	15.17	1.33	2.71	1.75	1.38	0.88
3B	2.86s	9.85s	3.44	16.11	1.17	1.67	1.48	1.47	1.07
4B	1.79s	35.42s	19.79	13.93	1.32	2.08	1.37	1.46	0.82
5B	0.31s	7.40s	23.87	51.87	1.22	2.00	1.96	1.22	2.00
6B	0.83s	5.11s	6.17	37.16	1.09	2.09	1.98	1.36	0.58
7B	21.36s	276.31s	12.94	5.11	1.45	2.22	1.83	1.35	0.30
8B	11.96s	67.88s	5.68	16.31	0.94	1.76	1.61	1.43	0.63
9B	41.83s	2231.62s	53.35	13.25	1.20	2.30	1.65	1.49	0.68
10B	1.59s	37.98s	23.89	23.84	1.44	2.13	1.76	1.32	0.95
11B	1.07s	9.02s	8.43	23.64	1.47	2.71	1.44	1.43	0.64
12B	7.94s	88.07s	11.09	29.37	1.91	2.90	2.50	1.52	34.38
13B	0.65s	8.38s	12.90	19.72	0.97	0.98	1.42	1.49	0.80
14B	3.02s	19.76s	6.54	11.72	1.18	1.68	1.38	1.45	0.85
15B	3.50s	30.84s	8.81	17.66	1.12	1.30	1.47	1.53	0.95
16B	5.77s	155.26s	26.91	11.77	1.06	0.99	1.52	1.44	0.86
17B	8.04s	227.62s	28.31	13.58	0.95	1.28	1.62	1.44	0.54
18B	86.87s	6168.57s	71.01	37.63	2.33	2.97	2.61	1.49	0.34
19B	14.72s	85.10s	5.78	24.34	1.22	3.22	2.38	1.51	1.54

Table 1: Timing results for several empty boards and 18 problems from (Berlekamp 2000).

vative estimate of 130 days using Wilson's solver.

Discussion

There are several reasons why Dots-And-Boxes is a game worth studying. First and foremost, it is an extremely popular and widely known game, and is familiar to a wide variety of people from around the world. In addition, it is an exceedingly simple game; the difficulty in solving the game comes primarily from the size of the problem space and not from any inherent complexity in the rules themselves. Finally, the game is an unusual example of an impartial game that cannot be addressed by the Sprague-Grundy theorem; most of the games addressed in the literature are partial.

Despite this, there has been remarkably little coverage of Dots-And-Boxes in the literature; what material exists does not discuss applying computational search. As a consequence, the value of each of the existing techniques is unknown. Our paper is the first to synthesize these techniques and present them in the context of heuristic search, providing a thorough discussion of their utility.

The combination of these techniques, along with some non-obvious modifications generic techniques, have allowed us to solve the previously unsolved 4×5 game. More importantly, however, our solver establishes a formal benchmark against which future research on this problem can be judged.

References

Allis, L. V.; van der Meulen, M.; and van den Herik, H. J. 1994. Proof-number search. *Artificial Intelligence* 66(1):91 – 124.

Berlekamp, E. R.; Conway, J. H.; and Guy, R. K. 2003. *Wnning Ways For Your Mathematical Plays*, volume 3. A K Peters.

Berlekamp, E. 2000. *The Dots-and-Boxes Game*. A K Peters.

Breuker, D. M. 1998. *Memory versus Search in Games*. Doctoral thesis, Maastricht University.

Grossman, J. P. 2010. Dabble. `http://www.mathstat.dal.ca/~jpg/dabble/`.

Guy, R. K., ed. 1991. *Combinatorial Games*. American Mathematical Society.

Roberts, P. 2010. Prsboxes. `http://www.dianneandpaul.net/PRsBoxes/`.

Ströhlein, T. 1970. *Untersuchungen ber kombinatorische*

Spiele. Ph.D. Dissertation, Technischen Hochschule München.

Thompson, K. 1986. Retrograde analysis of certain endgames. *International Computer Chess Association Journal* 9(3):131–139.

Wilson, D. 2010. Dots-and-boxes analysis index. `http://homepages.cae.wisc.edu/~dwilson/boxes/`.