Solving Peg Solitaire with Bidirectional BFIDA*

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

Abstract

We present a novel approach to bidirectional breadth-first IDA* (BFIDA*) and demonstrate its effectiveness in the domain of peg solitaire, a simple puzzle. Our approach improves upon unidirectional BFIDA* by usually avoiding the last iteration of search entirely, greatly speeding up search. In addition, we provide a number of improvements specific to peg solitaire. We have improved duplicate-detection in the context of BFIDA*. We have strengthened the heuristic used in the previous state-of-the-art solver. Finally, we use bidirectional search frontiers to provide a stronger technique for pruning unsolvable states. The combination of these approaches allows us to improve over the previous state-of-the-art, often by a two-orders-of-magnitude reduction in search time.

Introduction

Peg Solitaire is a simple one-player puzzle. It is played on a board which has a number of holes, some of which are occupied by pegs. Any peg can jump over an adjacent peg to land in an empty hole, and the jumped-over peg is removed from the board. The objective of the game is to make a sequence of jumps that leave the board in a specified goal configuration. While not required, the initial position generally has a single empty hole and the goal board has a single peg remaining, often left in the initially-empty hole.

While peg solitaire can be played on a number of different board types, we restrict ourselves to games played on a rectangular grid. On these, pegs can jump vertically or horizontally, but not diagonally. Figure 1 shows solvable opening states on the three primary boards studied in this paper.

The most basic question we can ask is whether there exists *any* sequence of jumps that transforms the initial state into the goal state. In this paper, however, we address the optimization problem, where we treat consecutive jumps by the same peg as a single move and try to find the fewest number of moves required to generate the goal state. To disambiguate, we will use the word "move" to apply exclusively to one or more single jumps by the same peg.

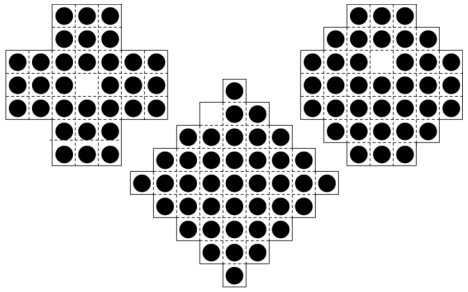

Figure 1: Examples of solvable initial states on the English (top left), French (top right), and Diamond(5) boards

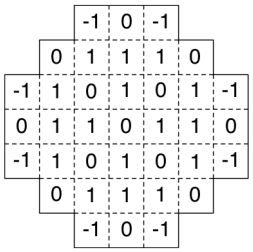

Figure 2: Pagoda values for the French board

Background

Peg Solitaire Constraints

The problem of playing peg solitaire has been addressed in a number of books. In particular (Beasley 1985) describes techniques used to prove certain board states as unsolvable. We use three of these in our work: position classes, pagoda functions (or resource counts), and peg types.

A complete discussion of *position classes* is beyond the scope of this paper. For our purposes, it suffices to say that every board configuration can be easily classified as belonging to a particular position class. If the initial and goal states are of different classes, an instance can be discarded as unsolvable without performing any search.

A *pagoda function* can be used to prune certain states as unsolvable. To define one, we assign each hole on the board a number subject to the following constraints: for any three horizontally- or vertically-adjacent holes with values x, y, and z we have that $x \leq y + z$ and $z \leq y + x$. The pagoda

value of a board state is given by the sum of the values of every occupied hole. Since a single jump removes pegs from two adjacent holes and places a peg in a third consecutive hole, a board's pagoda value can only stay the same or decrease after a jump is made. Thus, if we reach a state whose pagoda value is less than that of the goal, we know the state is a dead end and we cannot reach the goal. Figure 2 gives values for an effective pagoda function on the French board.

We classify all pegs on a board into four distinct *types*. A peg's type is defined by the parity of the row and column that it appears in, so a 2×2 square would contain one peg of each type. Since a jump moves a peg either two holes horizontally or vertically, the type of a peg never changes during a game. Thus, the number of pegs of a given type never increases and in games where the goal state has one peg, the count of that peg type can never reach zero. Also note that pegs can only jump other pegs of certain types. In particular, pegs types are divided into two classes: pegs that have both row and column of the same parity, and pegs that do not. Pegs in one class can only jump pegs in the opposite class.

BFIDA*

Breadth-First Iterative-Deepening A* (BFIDA*) is a well-known search algorithm (Zhou and Hansen 2006) based on Breadth-First Heuristic Search (BFHS). The algorithm performs a series of successively larger breadth-first searches. A cutoff is maintained on each node's f-cost, the sum of its cost so far (its g-cost) and a heuristic estimate of the remaining cost to the goal (its h-cost). Any node whose f-cost exceeds the current cutoff is pruned. Repeated iterations of breadth-first search are done with increasingly large f cutoffs until a solution is found of minimal cost. In general, each iteration of search will do significantly more work than the preceding, so the majority of time is spent on the final iteration. To avoid ambiguity, when we refer to an "iteration" in the context of BFIDA* this will refer exclusively to an entire breadth-first search with a given cost cutoff.

There are two properties of the peg solitaire search space that make BFIDA* an appropriate choice of algorithm. First, the search space on bigger boards can be quite large, such that solving instances in main memory is not practical with more traditional algorithms such as A*. As a breadth-first algorithm, however, BFIDA* is very amenable to disk-based techniques such as delayed-duplicate detection (Korf 2008), and can thus address larger problem instances.

In addition, peg solitaire has a very large number of duplicate states that occur through different move sequences. These duplicates will tend to occur at the same level of the search space and can thus be easily pruned in a breadth-first search by examining only the most-recently generated level. A depth-first approach such as DFIDA* (Korf 1985), by contrast, would not be able to detect these duplicates and so would perform considerable redundant work.

BFIDA* has one limitation relevant to our work, however, which is that it does not strongly benefit from tie breaking (Zhou and Hansen 2006). In a depth-first algorithm like DFIDA*, the last iteration of search can be extremely small, since the algorithm can use intelligent node ordering to go directly to the goal node. Since BFIDA* is breadth-first, however, it cannot generate the goal until every node at the immediately preceding level has been generated. All optimal solutions occur at the same level, so generating all remaining solutions is just a matter of expanding the remainder of the penultimate level, which will be small relative to the size of the overall iteration. Thus, finding *all* optimal solutions is not significantly harder than finding the *first* solution.

We note one important exception, which can occur in domains with weaker heuristics like peg solitaire. In these domains, the heuristic may not prune many nodes near the goal state, allowing BFIDA* to find solutions on an iteration before the last. Since the cost of these solutions exceeds the current cutoff, search cannot terminate immediately. However, BFIDA* *can* terminate as soon as it starts an iteration with a cutoff equal to that of the lowest-cost solution so far; this effectively prunes the last iteration of search. Robert Holte notes a similar property of DFIDA* in (Holte 2010).

Previous Work

Most of the previous work on computational search in peg solitaire has been done by George Bell, using both brute-force and heuristic techniques. (Bell 2007) discusses application of heuristic-search techniques (specifically BFIDA*) to peg solitaire. It focuses primarily on a variant in which diagonal moves are allowed, but contains discussion of search on orthogonal peg solitaire as well. The main result in orthogonal solitaire is a heuristic that improves over brute-force search on the English Board. On the two other boards considered, the French and Diamond(5), this heuristic produces little improvement over a simple brute-force search.

(Bell 2012) provides a comprehensive catalog of solutions for several popular board shapes, including those studied in our paper. The results on these boards were generated using bidirectional brute-force search. In orthogonal peg solitaire, the brute-force search algorithm used to generate these solutions can generally be considered the state of the art. The one exception is on the English board, where a heuristic approach provides modest improvements.

We have found one reference to previous work on bidirectional BFHS, used in the context of symbolic search (Richards 2004). This work is not formally published and we have been unable to acquire a copy, so our understanding comes from papers referencing it (Jensen et al. 2006). In this approach, simultaneous BFHS searches are done forward from the start state and backward from the goal state one level at a time, using the same cutoff for each. If the backward and forward levels intersect, an optimal solution is found with cost equal to the current search cutoff.

Expanding Levels By Jumps, Not Moves

The algorithm described in (Bell 2007) is a straightforward implementation of BFIDA* with a custom heuristic for Peg Solitaire. An important consideration is that moves are compound actions; that is to say, any move by a single peg can be extended to a longer move from the same initial state by appending another jump by the same peg.

As BFIDA* uses a breadth-first search, this raises the question of what constitutes a "level" in breadth-first search.

Bell considers a level as consisting of all nodes that can be reached by one move from some node on the preceding level. That is, he generates the next level of search by looking at every node on the current level, finding all *moves* that can be made on that board, and placing the nodes that result from applying those moves on the succeeding level.

This approach follows the traditional model of BFIDA*, where all nodes on a level have the same g value. In peg solitaire, however, it presents some problems. Node expansion is complicated: generating all successors to a node involves a secondary search of legal jump sequences, adding complexity. In addition, doing a secondary search of moves on different states may result in duplicate intermediate states. Detecting and pruning these duplicates is non-trivial.

A more fundamental problem with this approach is that, since moves of different lengths remove different numbers of pegs, not all nodes on the same level of search have the same number of pegs. This means that duplicate states appear at different levels of search. We must then either store multiple levels of the problem space for duplicate detection, or pay the cost of additional, duplicate work.

An alternative approach—the one used in our solver—is to consider each level as being all nodes that are reachable by performing a single *jump* from some node on the preceding level. Under this strategy, generation of children becomes trivial and quick: the successors to a node on the next level are simply all nodes that can be reached by a single jump on the board. This, combined with the fact that all nodes at the same depth now have the same number of pegs, means that we are able to easily detect all duplicates in a search.

The downside to this approach is that it involves additional bookkeeping complexity. Since not all nodes on the same level have been generated by the same number of moves, we need to store with each node its associated g cost. In addition, we need to store with each node the identity of the last peg moved; this allows us to determine whether a jump made from that node is a continuation of the last move or starts an entirely new move. These two additional fields impose a small additional space overhead. Finally, care must be taken when deciding when to prune a node, since it may be possible to extend the move that generated a current node and thereby reduce its h-value. We cannot prune a node until the move that generated it is complete.

At the cost of some additional bookkeeping space, however, we have now guaranteed that all duplicate nodes will occur at the same level of search. This means that duplicate detection requires storing only the current level of search, and we do not need to waste additional space or time looking for duplicates in previous search levels.

Improved Heuristics

We have developed an improved heuristic for peg solitaire. Importantly for our algorithm, it is a consistent heuristic. Our heuristic uses three different components based on properties of the board: the number of occupied corner holes, the number of pegs of certain types on the board, and the number of occupied *Merson regions* (described later). We call to these components $h_c(n)$, $h_t(n)$, and $h_m(n)$, respectively.

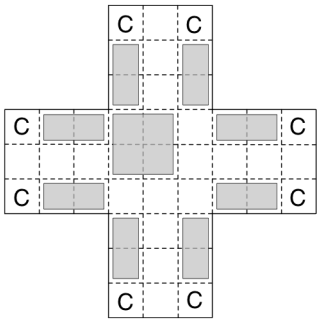

Figure 3: Corners and Merson regions on the Wiegleb board.

Our first component, $h_c(n)$, is based on corners, which are holes that cannot be jumped over by a peg. Figure 3 labels corners using a "C" on a large board known as the Wiegleb. $h_c(n)$ is simply the count of corners that are occupied by a peg in the current state but not the goal. Each such corner must be vacated to find a solution. Since pegs in those holes cannot be captured, they can only be removed from their corner hole with a move starting at that hole. Thus, $h_c(n)$ is a lower bound on the optimal solution cost. Note that $h_c(n)$ counts the number of moves required to remove pegs from certain holes, not to remove them from the board entirely.

For every peg type, there is some number of pegs that must be removed from the current board to reach the goal. In addition, there is a maximum number of pegs of that type capturable in a single move. For example, consider the type of peg that can occupy corners on the English board: a maximum of four such pegs can be removed by a single move. Thus, there is a minimum number of moves required to reduce the count of each peg type to its count in the goal state. Our second heuristic component, $h_t(n)$, is based on this property. In isolation, $h_t(n)$ could be defined as the maximum of the number of moves required to remove excess pegs of each type. However, moves that vacate corner holes will capture other pegs. To ensure that $h_t(n)$ is independent of $h_c(n)$, we calculate $h_t(n)$ based only on peg types that cannot be captured by moves originating in a corner.

The third component we call $h_m(n)$, and relies on *Merson regions*, contiguous regions of the board with the property that, if all holes are occupied, pegs in the region can only be removed by moves that originate within that region. Figure 3 shows nine Merson regions on the Wiegleb board as gray rectangles. $h_m(n)$ simply defines Merson regions on the board (excluding corners) and gives the count of regions filled in the current state but not the goal. Our implementation simply defines a static set of non-overlapping Merson regions, so it would be possible to improve upon this technique using, for example, dynamically-selected regions.

$h_t(n)$ and $h_m(n)$ cannot be combined, as a move vacating a Merson region might take pegs of any sort. A move affecting $h_c(n)$, meanwhile, will not affect $h_t(n)$ or $h_m(n)$. Thus, our complete heuristic function is defined as $h_c(n) + max(h_t(n), h_m(n))$.

This heuristic improves on Bell's in two ways. Bell defines one heuristic that accounts for occupied corners and

the minimum number of moves required to remove pegs of certain types. However, he appears to not have extended his heuristic to account for boards where different corners are occupied by pegs of different types. This means that on boards like the French and Diamond(5), where two different peg types can occupy corners, his heuristic would undercount the occupied corners and moves required to remove excess pegs. In addition, he defined a separate heuristic based purely on Merson regions but uses these two heuristics independently, while our heuristic takes the max of these two approaches. This is primarily helpful on the larger Wiegleb board, where Merson regions make a noticeable difference.

An important property of our heuristic is that a move can reduce its valuation of a state by at most one. A move can affect our heuristic in three different ways: it can vacate a single corner, it can reduce by one the number of moves needed to remove excess pegs of some type, or it can vacate a single Merson region. Each of these possibilities reduces the heuristic value by one. The first possibility cannot happen in combination with the second and third and we take the max of the second and third values. Thus, the heuristic value can decrease by at most one after any move and our heuristic function is monotone and consistent, not just admissible.

Bi-Directional BFIDA*

Our search algorithm is a bidirectional form of BFIDA*. It performs two independent BFIDA* searches, one forward from the start and one backward from the goal (in the case of multiple goal states, the backward search is seeded with each possible goal rather than a single root node). Each search has an independent cutoff. After a complete breadth-first iteration of BFIDA* is done in one direction, the cutoff for that direction is increased. To keep work in each direction balanced, we choose whether to perform a forward or backward iteration next based on which direction had the fewest number of nodes expanded on the previous iteration (this is very similar to Pohl's cardinality principle (Pohl 1971)).

As we do an iteration of BFIDA* with a given cutoff, we maintain a frontier of all nodes which have been expanded but had a child pruned for exceeding that cutoff. Coming from the opposite direction, we cannot reach the goal (or start) state without passing through this frontier. The reason we keep all nodes expanded—rather than those generated but pruned—is that we want to guarantee that we have the optimal path to all nodes on our frontier. We can guarantee this property since we are using a consistent heuristic. Note that any node pruned for violating pagoda or peg type constraints need not be stored on the frontier, as such a node cannot possibly be part of a solution.

The frontier is stored on a level-by-level basis: each level of the frontier contains only nodes with the same number of pegs. During search, we check each node selected for expansion against the opposing frontier. If the frontiers intersect, we have a candidate solution and we keep track of the lowest-cost solution found. Finding intersections between the frontiers can be efficiently done, even on disk. To eliminate duplicate states on disk using delayed-duplicate detection we efficiently sort each breadth-first layer on disk and then do a linear scan to detect duplicates. Frontiers are similarly kept as sorted files. Thus, when expanding a layer we are traversing a sorted file. To find intersections with the opposing frontier we simply traverse the appropriate file in parallel, adding just the cost of one additional file read.

In addition, we note that we need not explore the children of any node that has intersected the opposing frontier. Since our heuristic is consistent and we have expanded the same node from both directions, we must have the optimal path from the start to the goal through this node. Thus, we cannot improve upon this solution by exploring further children beneath it. Note that this does *not* guarantee that we have found the optimal overall solution, though.

The fact that frontier intersections provide us with candidate solutions leads to one of the main benefits of our approach, and the most important contribution of our paper. If we are searching for a single optimal solution we can stop searching once we prove that the best solution found so far is in fact optimal. Any time we find a frontier intersection we have a candidate solution and, since this is the intersection of two searches coming from opposite directions, its cost will likely exceed the current f-cutoff. In all of our experiments, we found the optimal solution on an iteration with a smaller-than-optimal cost cutoff. As soon as we *start* an iteration whose cutoff is the cost of the best solution found so far, we can stop as that solution has been proved optimal.

In unidirectional BFIDA*, as discussed earlier, an optimal solution is generally only found near the very end of an iteration with the optimal cost cutoff. The overall effect of our technique, then, is to eliminate the last iteration of BFIDA* entirely, at the cost of some number of iterations of backward search. In general, BFIDA* is effective because each iteration of search is much larger than the last, so removing the last iteration should significantly reduce the length of time spent searching. Our results in *Experiments* demonstrate that this is indeed the case.

Propagation Of Pagoda And Peg-Type Constraints

We can leverage the fact that our bidirectional search frontiers do not pass each other to improve our ability to prune unsolvable states. As mentioned earlier, pagoda values and counts of pegs of a certain type are properties of a game that can never increase during play. Thus, we can prove certain states to be unsolvable if, for example, their pagoda value becomes less than that of the goal.

Note that, in forward search, any path from a node to the goal must pass through the backward search frontier. Thus, the pagoda value of the current node must be equal to at least the minimum pagoda value of all nodes on the backward search frontier, not just the goal node. The same is true for peg-type constraints. In backward search, we keep track of the minimum pagoda value and peg type counts of all nodes added to the final frontier and we use this value to prune unsolvable nodes during forward search. We do the converse for nodes added to the frontier in forward search.

Board	Start hole	End peg	Bell (2012)	Bell (2007)	BFIDA*		BD-BFIDA*		BD-BFIDA*, no constraint propagation	
				Time	Time	Nodes	Time	Nodes	Time	Nodes
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)
English	(3, 3)	(3, 3)	91.4	46.6	† 3.0	4.07	***2.9***	3.90	4.0	5.55
	(0, 3)	(3, 3)	85.8	49.8	† ***3.5***	4.79	5.2	6.90	7.8	10.79
	(0, 3)	(0, 3)	189.0	49.3	27.6	32.83	***3.2***	4.41	4.5	6.44
	(0, 3)	(6, 3)	196.6	51.9	27.7	32.83	***3.2***	4.41	4.5	6.44
	(3, 3)	(0, 3)	171.8	9.2	7.0	9.57	***0.3***	0.50	0.4	0.59
	(2, 3)	(2, 3)	156.7	4.9	12.5	16.28	***1.3***	1.84	1.9	2.70
	(0, 2)	(3, 2)	128.3	6.8	5.0	6.78	***0.3***	0.38	***0.3***	0.38
	(0, 3)	(3, 3)	133.9	8.2	6.7	9.10	***0.5***	0.73	***0.5***	0.73
	(1, 3)	(4, 3)	376.2	8.1	5.4	7.21	***1.4***	1.65	***1.4***	1.66
	(1, 3)	(1, 3)	355.5	452.4	† ***39.6***	45.90	52.2	62.68	55.0	68.13
	(2, 3)	(5, 3)	132.3	80.2	† ***5.0***	6.78	6.4	8.78	6.3	8.93
	(0, 2)	(3, 5)	139.3	107.7	† ***6.7***	9.10	12.0	16.46	13.0	18.41
	(0, 2)	(0, 2)	316.0	17.5	13.3	18.06	***1.9***	2.73	***1.9***	2.74
	(2, 3)	(2, 0)	368.2	29.6	19.8	26.13	1.6	2.21	***1.5***	2.24
	(0, 2)	(6, 2)	314.5	17.8	13.3	18.06	***1.9***	2.73	***1.9***	2.75
	(1, 3)	(4, 0)	319.4	31.7	20.9	27.46	***1.8***	2.57	***1.8***	2.72
	(2, 2)	(2, 2)	201.0	5.0	4.0	5.54	***1.4***	1.98	***1.4***	1.98
	(1, 2)	(3, 3)	188.0	2.6	2.3	3.24	1.4	1.69	***1.3***	1.69
	(1, 2)	(1, 2)	218.1	43.5	16.1	21.05	1.1	1.54	***1.0***	1.54
	(2, 2)	(2, 5)	326.2	128.7	49.3	59.95	3.9	5.56	***3.7***	5.56
	(1, 2)	(4, 5)	226.8	43.2	16.1	21.05	1.1	1.54	***1.0***	1.54
Total:			01:17:13	00:19:53	00:05:05		00:01:45		00:01:55	
French	(0, 2)	(0, 4)	9,804	-	174	191.08	16	16.80	***15***	17.46
	(0, 2)	(3, 1)	5,515	-	92	100.95	***7***	8.11	***7***	8.11
	(0, 2)	(3, 4)	3,825	-	85	92.11	***5***	5.42	***5***	5.43
	(0, 2)	(6, 4)	9,908	-	173	191.08	***15***	16.86	16	17.49
	(2, 3)	(4, 0)	7,530	-	1,959	1,744.32	***363***	354.75	370	386.90
	(2, 3)	(1, 3)	4,033	-	562	521.45	107	109.43	***105***	109.62
	(2, 3)	(4, 3)	2,410	-	542	495.58	80	82.30	***79***	83.00
	(1, 3)	(2, 0)	5,666	-	384	396.32	***22***	25.32	23	27.13
	(1, 3)	(5, 3)	3,125	-	102	106.49	***5***	5.84	***5***	5.84
	(1, 3)	(2, 3)	2,031	-	96	100.43	***5***	5.40	***5***	5.41
Total:			14:57:27		01:09:29		00:10:25		00:10:30	
Diamond(5)	(1, 3)	(1, 5)	8,855 (703)	- -	3,013	2,445.92	***63***	61.57	484	469.85
	(4, 6)	(1, 5)	8,585 (1,215)	- -	6,202	4,864.44	***272***	250.68	1,064	1,009.64
	(1, 3)	(4, 2)	8,585 (1,272)	- -	414	356.47	***74***	72.23	164	151.10
	(4, 6)	(4, 2)	1,282 (1,268)	- -	472	395.24	73	70.59	***72***	70.59
Total:			07:35:07 (01:14:16)		02:48:21		00:08:02		00:29:44	

Table 1: Timing values and numbers of nodes expanded for all solvable instances on the English, Diamond(5), and French boards. Times of individual instances are given in seconds and node counts are given in millions.

Experiments

We performed our experiments on three different boards: the English, French, and Diamond(5) boards (see figure 1). The rows of table 1 summarize our results on the solvable instances of these boards reported in (Bell 2012). The first column gives the shape of the board, the second column gives the location of the empty hole in the start state, and the third column gives the location of the final peg in the goal state. Peg locations are given as row/column coordinates on the smallest box that encloses the board, starting at (0, 0) in the top-left corner. For example, (3, 3) is the center-most hole on the English and French boards. All experiments were run on a 3.33 GHz Intel Xeon with 48 GB of RAM. We find the same minimum path length as Bell in all cases.

Bell has graciously provided us with his solver, which we have used to regenerate his results on our machine. Recall that these results represent the state of the art on minimal-length peg solitaire solutions. Column 4 shows the results of the brute-force bidirectional search algorithm used in (Bell 2012), which was previously the strongest solver on the French and Diamond(5) boards. Note that this column has two numbers for each instance on the Diamond(5) board. The first gives the timing numbers of the default algorithm. In addition, Bell has manually found a tighter constraint on pagoda values on this board. This tighter constraint is of the form found by our propagation technique. The number in parentheses is for the solver with this constraint enabled.

Column 5 shows the results of the BFIDA* algorithm of (Bell 2007) on the English board, where it is most effective. We have not generated results for the other boards, but according to Bell they do not significantly improve on brute-force search. His solver only performs a single iteration of BFIDA* at a time, so the numbers given are for the final iteration with the optimal cost cutoff. A complete run would require slightly more time for the initial iterations.

Columns 6 and 7 give the time and numbers of nodes expanded with our implementation of unidirectional BFIDA*. As mentioned earlier, with a sufficiently weak heuristic, unidirectional BFIDA* can find an optimal solution on an iteration with a cutoff less than the optimal solution cost. And, indeed, this occurs in five instances. These instances are marked with a dagger. Ignoring these cases, our BFIDA* solver compares favorably to Bell's and generates comparable or slightly faster results in most cases.

Our heuristic significantly improves upon those given in (Bell 2007) on the French and Diamond(5) boards. While the previous heuristics barely improve upon brute-force search, we have managed to easily solve all of these instances using simple unidirectional BFIDA*. In the case of the French board, our unidirectional heuristic search is often over an order-of-magnitude faster than the previously-optimal solver. On the Diamond(5) board, our solver does not benefit from Bell's manually-derived tightened constraints, but still comfortably beats his default solver.

Columns 8 and 9 give the time and nodes expanded with our bidirectional BFIDA* algorithm. These are our strongest results. In all cases, our solver was able to find an optimal solution on an iteration before the last. Our solver is significantly faster than the previous state-of-the-art, generally beating it by one or two orders of magnitude. In the most extreme example, the third problem of the French board, our solver is able to reduce the solution time from over an hour down to five seconds of runtime. While some of this improvement is due to the improved heuristic, our bidirectional solver is itself able to significantly improve over our unidirectional approach, often by an order of magnitude.

As noted previously, our unidirectional solver occasionally finds optimal solutions before the last iteration. Significantly, unidirectional search was only able to outperform our bidirectional approach in these cases, due to the overhead in the latter of additional iterations of backward search.

Columns 10 and 11 give the time and nodes expanded with bidirectional BFIDA*, but without propagation of pagoda and peg-type constraints. That is, we only place constraints on nodes using the pagoda and peg-type values of the start and goal states, not values calculated from the search frontiers. In most problem instances this technique does not affect search much, producing only modest reductions in node counts at the cost of slightly more expensive node expansions. On three of the Diamond(5) instances, however, this technique was able to find the tighter constraints that Bell enforces manually. This dramatically speeds up runtime. Since the technique usually has little negative impact and has the capacity to dramatically improve performance, we believe that it should be used in general.

Finally, we solved all instances of the Wiegleb board, pictured in figure 3. Bell's solver took three months to solve all 35 solvable instances on this board. Our solver, meanwhile, took just a week and a half. We have benefited strongly from disk-based techniques on this problem: on larger instances our solver uses up to a terabyte of disk storage.

Conclusions And Future Work

The primary contribution of our paper is a novel approach to performing bidirectional BFIDA*. This approach, which generalizes beyond the domain of peg solitaire to other domains with reversible operators and heuristics, has the useful property that it can find single optimal solutions without performing an iteration of search with a cutoff equal to the optimal solution cost. In our experiments, this technique generally provides significant performance improvements over a more conventional unidirectional search.

In addition, we have provided improvements to the previous state-of-the-art techniques for solving peg solitaire. We have devised a new approach for searching by levels which allows us to catch all duplicate states. We have improved over existing heuristics. Finally, we make use of the bidirectionality of our search to to automatically derive tighter constraints on search. The combination of these techniques allows us to solve problem instances significantly faster than the state-of-the-art, often by up to two orders of magnitude.

An improved implementation of DDD for disk-based search works by partitioning each layer into disjoint hash tables small enough to fit into memory (Korf 2008), avoiding the overhead of sorting. We have implemented this approach and tentative results indicate significant speedups (on the order of 25% on some larger problems). However, we were not able to complete these experiments in time for publication.

References

Beasley, J. 1985. *The Ins And Outs Of Peg Solitaire*. Oxford University Press.

Bell, G. I. 2007. Diagonal peg solitaire. *Integers: Electronic Journal Of Combinatorial Number Theory* 7(G1):20.

Bell, G. I. 2012. George bell's peg solitaire page. `http://home.comcast.net/~gibell/pegsolitaire`.

Holte, R. C. 2010. Common misconceptions concerning heuristic search. In *Proceedings of the Third Annual Symposium on Combinatorial Search*, 46–51.

Jensen, R. M.; Hansen, E. A.; Richards, S.; and Zhou, R. 2006. Memory-efficient symbolic heuristic search. In *International Conference on Automated Planning and Scheduling*, 304–313.

Korf, R. E. 1985. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence* 27:97–109.

Korf, R. E. 2008. Linear-time disk-based implicit graph search. *J. ACM* 55:26:1–26:40.

Pohl, I. 1971. Bi-directional search. *Machine Intelligence* 6:127–140.

Richards, S. K. 2004. Symbolic bidirectional breadth-first heuristic search. Master's thesis, Mississippi State University.

Zhou, R., and Hansen, E. A. 2006. Breadth-first heuristic search. *Artificial Intelligence* 170(45):385 – 408.