Peg Solitaire: A New Algorithm for an Old Game

A Technical Report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science

University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree

Bachelor of Science, School of Engineering

Michael Starego

Spring, 2022

On my honor as a University Student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

Advisor

Daniel Graham, Department of Computer Science

Peg Solitaire: A New Algorithm for an Old Game

CS 4991 Capstone Report, 2022

Michael Starego
Computer Science
The University of Virginia
School of Engineering and Applied Science
Charlottesville, Virginia USA
mds8dp@virginia.edu

ABSTRACT
I propose a graph search algorithm that can be used to efficiently find solutions to exceptionally large peg solitaire puzzles. This algorithm uses a variation of bidirectional breadth-first search to find a path between a given start and end state while keeping a very small number of nodes in the fringe and being agnostic to the geometry of the underlying puzzle. I developed and tested this algorithm in C++ on a 2016 MacBook Pro with a 2.7 GHz Quad-Core Intel Core i7 processor. This program consistently found solutions in a matter of minutes for puzzles as large as the 15-row triangular board, which has 2^{120} potential board positions and requires 118 moves to solve. Future uses of this algorithm include application to other search problems in the peg solitaire realm, such as finding the shortest solution for a given board geometry.

1. INTRODUCTION

Peg solitaire is a centuries-old board game that involves a single player and a board of holes containing pegs or marbles. The game starts with one hole empty and every other filled. A move consists of jumping one peg over another to land in an empty hole, at which point the jumped peg is removed from the board. The game is complete when there is only one peg remaining.

Variations of peg solitaire, such as Chinese Checkers, Hi-Q, and the Cracker Barrel Peg Game can be found throughout popular culture. With these variations come different board geometries. Figure 1 shows the 5-row triangular board (also known as the Cracker Barrel Peg Game) as well as the popular French and English geometries. The experiments detailed in this paper were performed on the triangular board geometry due to its ability to scale to larger and larger sizes by simply adding more rows.

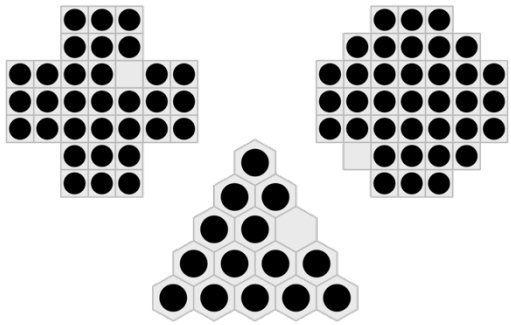

Figure 1: (Left to right) The English, triangular, and French board geometries.

2. BACKGROUND

Those who have studied peg solitaire have devised slightly different formulations of the problem at hand. For example, some define a move as one peg jumping into one hole, while others consider this action to be a "jump" and a move can involve multiple consecutive jumps made with the same peg. While the latter definition is useful when considering the shortest-length solution (under the former definition all solutions have the same length), this paper is not concerned with that property, so the former definition of the term "move" is used.

In addition, it is necessary to define the inputs and outputs for this formulation of the problem: a solution consists of a sequence of board states in which the

transition between each state is a valid move. The start state must have a hole at position *s*, and the end state must have a peg at position *e*. The integers *s* and *e* are provided as input and the solution is given as output. It is not necessarily true that there is a solution between every *s* and *e*, or that there exists a valid *e* for every *s*. The reasons for this have been explored at length in prior literature, and thus are excluded from the scope of this paper.

3. RELATED WORKS

Much of the research that has been done on the peg solitaire puzzle is attributed to mathematician George Bell, who has published dozens of papers on the topic. Bell has developed multiple heuristic search techniques for efficiently traversing the peg solitaire state space and has shown by induction how a solution can be constructed for a triangular or a hexagonal board of any size [1, 2].

Bell's research regarding large boards has been focused on finding the shortest-length solution using either inductive reasoning or graph search techniques. The largest boards that he has found short solutions for are the 12-row triangular board (78 holes) and multiple rectangular boards up to the 12x12 board (144 holes) [3]. Some recent improvements have been made to Bell's work on heuristic searches, such as the work done by Barker and Krof in 2012 that used bidirectional as opposed to unidirectional BFS to improve the efficiency of Bell's algorithm [4].

While this body of existing work is fairly comprehensive, the algorithm I propose has the novel ability to find solutions efficiently on large boards without the use of any heuristic function and with very limited assumptions about the nature of the underlying puzzle. This implies that there is much room for improvement in the algorithm's ability to find solutions (including shortest-length ones) by applying the heuristics developed by Bell and others. Furthermore, the algorithm's high generalizability suggests that it may have applications outside of the peg solitaire realm.

4. ALGORITHM DESIGN

The goal of this algorithm is to find a path from the start state of a peg solitaire puzzle, where position *s* is empty and the rest of the board is filled, to an end state where position *e* is filled and the rest of the board is empty, given a board size and geometry. The algorithm makes three assumptions about the mechanics of the underlying puzzle: First, that a board of size *n* can be represented by an *n*-bit unsigned integer. This requires a one-to-one mapping from each position on the board to a bit in the integer, where a "1" represents a peg and a "0" represents a hole. Second, that each move decreases the total number of pegs on the board by exactly one. Thus, every solution must be $n - 2$ board states in length. Third, that the logic for moving backward through the puzzle (i.e. undoing moves) is the same as moving forward, except with the concept of a peg and a hole reversed. This principle will be described formally in the following section. All other information about the mechanics of the puzzle will be encoded in a function named *getChildren* that takes in a board and outputs its children, which are the set of boards that can be reached by executing exactly one move.

4.1 Naïve Bidirectional Search

Consider the following naïve approach to bidirectional peg solitaire search. Take starting board *a* (where position *s* is empty) and perform breadth-first search (BFS) to construct search tree *A*, stopping when the depth of *A* is roughly *n*/2. Then take end board *b* (where position *e* is filled) and perform BFS moving in the backward direction to construct search tree *B*, stopping when a board *m* is found that also exists in tree *A*. A solution path from *a* to *b* can be constructed by combining the path from *a* to *m* in tree *A* with the reverse of the path from *b* to *m* in tree *B*.

At first glance, there is an issue with this approach: the problem formulation only provides the ability to move forward through the puzzle (via the *getChildren* function) and does not explicitly provide a way to move backward. Fortunately, our third assumption implies a solution to this. Consider the complement of a board *b*, denoted as *b'*, which has a peg where *b* has a hole and a hole where *b* has a peg. Formally, the third

assumption states that for every board *b* and each of its children *c*, the board *b'* is a child of *c'*. Intuitively, this assumption holds for traditional peg solitaire: in the forward direction a move involves replacing a sequence of peg-peg-hole with hole-hole-peg, while in the backward direction a sequence of hole-hole-peg is replaced with peg-peg-hole.

This property implies that the original algorithm can be reformulated to only require searching in the forward direction. Instead of growing tree *B* backward from board *b*, it can be grown forward from its complement *b'*. In order for the two searches to "meet in the middle," tree *A* must contain some board *m* whose complement *m'* appears in tree *B*. Thus, tree *B* will yield a path from *b'* to *m'*, which, when the complement is taken of each board along that path, produces a backward path from *b* to *m*.

4.2 Searching on Large Boards

This naïve approach is effective for small boards but not large ones. The size of the search space for large boards is too great to feasibly grow complete search trees using BFS. We will instead have to prune these trees to maintain a reasonable size. This can be done by simply placing a ceiling on the number of nodes that each tree level can contain. We will refer to this value as the capacity of the search tree.

This alone is not enough to make this search technique effective for large puzzles. Even if the capacity is extremely high – say, millions of nodes – this number is dwarfed in comparison to the total number of nodes on each level. Consider the 15-row triangular board, where $n = 120$, at the level where all boards contain 60 pegs. The number of possible configurations of 60 pegs in 120 holes is equal to $9.7 * 10^{34}$. While all of these configurations may not be reachable in this puzzle, many of them are, and it is clear that the average computer cannot store any significant fraction of this many integers. Thus, an additional mechanism is necessary if we are to reasonably expect a solution to be found.

Consider a mechanism that will force the boards in tree *A* to be as different as possible from the boards in tree *B* using survivorship bias. When each subsequent tree level is computed, the vast majority of new nodes must be discarded. The only nodes that survive this process on a given level are the ones that are the most different from the boards in the other tree at the same level. Define the difference between two boards as the number of positions in which they differ – that is, in a given position one board has a peg and the other has a hole. It follows that the two trees must be grown simultaneously: they will take turns computing levels and the boards from the last computed level on the opposite side will be used to select the survivors of the current one. This process will create a feedback loop that will cause the boards in each tree to maintain similarity to each other while maximizing their difference to the boards in the other tree. Ultimately, this will make it likely for these two trees to generate complementary boards.

The process of computing a level of nodes in a given tree can be formalized as follows. Assume that we are operating on tree *A*. First, take each board in the previous level of *A* and generate its children. Combine these new nodes into a set S_A. Next, construct a board *x* to represent the average board among the last computed level of the opposite tree, S_B, using the following method. For each board position, count the number of boards in S_B that have a peg in that position. Construct *x* by placing a peg in the positions with the highest frequency value until *x* has the same number of pegs as each board in S_B does. Finally, compute the difference value between each board in S_A and board *x*. Select the boards with the highest difference values to comprise the new level, such that it is filled to capacity if possible. Discard the rest of the boards.

Unlike the naïve approach, this strategy is incomplete, meaning that it is not guaranteed to generate a solution on each run. However, the experimental results show that it is able to effectively find solutions for large puzzles while expanding an extraordinarily small fraction of the total nodes in the search space.

5. RESULTS

The algorithm was implemented for triangular board geometries in C++ and run on a 2016 MacBook Pro with a four-core 2.7Ghz Intel i7 processor and 16GB RAM. It takes five parameters: r, the number of rows in the board, s, the start hole position, e, the end peg position, and c, the capacity of the search trees. The program outputs a visual representation of a solution found (if any), the total number of complement pairs found, and its runtime in seconds.

Tables 1-3 present results from running the program for triangular puzzles while varying the size of the board and its capacity. Each row in the table represents a single run of program. For each test, the start position is 12 and the end position is 24. Note that the positions are indexed from top to bottom, left to right. Capacity represents the maximum number of boards that the program maintains on each level of the search tree. Runtime is reported in seconds. Complement pairs found describes the number of pairs of boards from the leaves of the two search trees that are complements of each other. Note that it is very likely that many distinct solution paths can be constructed from each complement pair found.

Table 1: 12-Row Board (n = 78)

Capacity	Runtime (s)	Complement Pairs Found
5,000	1.56	0
10,000	2.66	14
25,000	7.63	88
50,000	15.33	170
100,000	36.49	331

Table 2: 14-Row (n = 105)

Capacity	Runtime (s)	Complement Pairs Found
10,000	6.42	0
50,000	28.56	9
100,000	57.86	16
250,000	142.81	16
500,000	307.96	327

Table 3: 15-Row (n = 120)

Capacity	Runtime (s)	Complement Pairs Found
10,000	7.60	0
50,000	35.21	0
100,000	84.82	42
250,000	195.21	4
500,000	356.15	16

It is noteworthy that when the capacity is held constant, the runtime of the program increases linearly as the size of the board increases. This result garners a positive outlook on the program's ability to perform on even larger boards, although it is clear that the likelihood of finding complement pairs decreases for a given capacity as the board size increases. Overall, it is clear that the algorithm is effective at finding solutions to these large triangular boards given sufficiently high capacity.

6. CONCLUSION

This algorithm was able to efficiently produce solutions for large peg solitaire puzzles up to the 15-row triangular board. It uses bidirectional BFS with a powerful survivorship bias mechanism that incentivizes the two search trees to meet in the middle. It is able to find solutions while expanding an infinitesimally small number of nodes compared to the total size of the search space. The algorithm makes very few assumptions about the underlying mechanics of the puzzle, which implies a high degree of generalizability.

7. FUTURE WORK

In future experiments, this strategy can be adapted to solve other problems in the peg solitaire realm, such as finding shortest-length solutions for large boards. Also, more work can be done to model the likelihood of the algorithm finding a solution with given parameters. Finally, heuristics can be used to increase the chance of viable nodes surviving, which will improve the effectiveness and efficiency of the algorithm.

REFERENCES

[1] George I Bell. 2007. Solving triangular peg solitaire. arXiv preprint math/0703865 (2007).

[2] George I Bell. 2009. Notes on solving and playing peg solitaire on a computer. arXiv preprint arXiv:0903.3696 (2009).

[3] George I. Bell. Retrieved February 24, 2022 from http://recmath.org/pegsolitaire/

[4] Joseph K. Barker and Richard E. Korf. 2012. Solving peg solitaire with bidirectional BFIDA. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence* (*AAAI'12*). AAAI Press, 420–426.