Solving Peg Solitaire Using Advanced Backtracking

Luke Althoff
AlthoffLJ19@gcc.edu

Katherine Bennett
BennettKE20@gcc.edu

Keegan Woodburn
WoodburnKB20@gcc.edu

I. Introduction

Single-player games have been a facet of society for as long as people have had free time to spare. Following in the wake of these games is a wide field of material within game theory. Throughout the past millennium, thousands of games, algorithms, and mathematical models have been formulated. One of the more popular of these games is peg solitaire. There are different variants of the game such as the European style, the diamond variant, and the triangular variant. While each variation introduces its own unique complexity, triangular peg-solitaire lends itself to be a particularly interesting and difficult form of the game, largely due to its shape.

II. Terminology

Triangular peg-solitaire is based around an equilateral triangle with sides of length n, which is split into $\frac{n\cdot(n+1)}{2}$ holes, also called cells [Fig 1a]. These boards are commonly denoted by T_n, with n being side length and the n^{th} triangular number. The two most used methods of notating these boards are alphanumeric indexing [Fig 1b] and skew indexing [Fig 1c]. Alphanumeric indexing assigns a tuple (x, y) to a cell, where x is a letter, y is a number, and (x, y) has not previously been assigned to a cell. Skew indexing assigns a tuple (i, j) to a cell, where i and j are in the set of whole numbers and (i, j) has not previously been assigned to a cell.

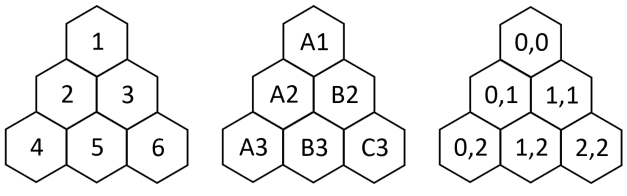


Fig. 1: An example of the same board T_3 (a) with alphanumeric indexing (b) and skew indexing (c).

Each cell in a board is given a binary value indicating whether or not the respective cell contains a peg. At the beginning of the game the board is described as the initial board $T_n^{initial}$, where only one cell is left empty [Fig 2a]. This empty cell is called the starting vacancy. The goal board T_n^{goal} is a board where only one cell, called the final cell, contains a peg [Fig 2b] [3]. While the location of the starting vacancy and final cell varies from game to game, most rule sets require the final cell in the goal state to be the same cell that was the starting vacancy in the initial state.

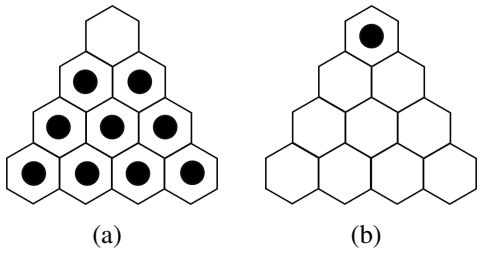
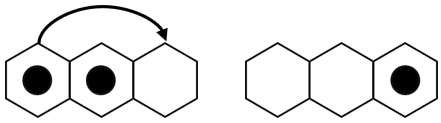


Fig. 2: A typical initial board (a) and goal board (b).

To get from the initial state to the goal state, a series of jumps must be made. A jump is made when a peg in cell c_0 moves over another peg in cell c_1, and into an empty cell c_2, where c_0, c_1, and c_2 are arbitrary collinear cells. This action results in the removal of the peg in c_1 [Fig 3]. A jump is represented by a tuple with the starting and end coordinate of the peg that is moving. For example, when a jump is made from A1 to A3, we denote this as (A1, A3). One or more jumps made by one peg is called a move, which is denoted by each coordinate the peg reaches in the order it reaches it at.

While a jump can be understood as an individual action, it is important to understand the context in which this jump occurs. The concept of a position helps contextualize jumps within a given board. The positions of an arbitrary board T_n is the set p_n of all cell triplets in T_n that are collinear. For example, given



(a) (b)

Fig. 3: A jump over c_1 from c_0 to c_2 (a), which results in the removal of the peg in c_1 (b).

a board T_4, the triplets (A1, A2, A3) and (C5, D5, E5) are both in the set of positions produced by T_4 [Fig. 4].

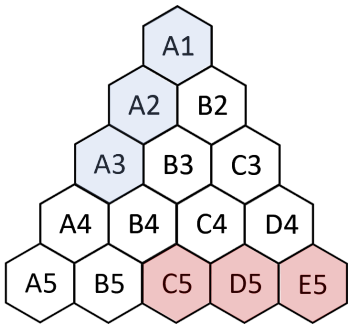


Fig. 4: The position (A1, A2, A3) is highlighted in blue, and the position (C5, D5, E5) is highlighted in red.

The cardinality of positions p_n produced by an arbitrary board T_n is given by the following recursion:

$$|p_n| = \begin{cases} 0 & n < 3 \\ 3 & n = 3 \\ |p_{n-1}| + 3 \cdot (n-2) & n > 3 \end{cases}$$

From these positions, we derive a set of possible equivalence classes. George Bell describes a similar process of partitioning the set of positions but uses the cell values relative to each other in a position in order to obtain a more concise partition [2]. However, for the sake of clarity and the purposes of this paper, we define the class P_n to be the n^{th} element in the set $\{(0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,1,1)\}$. Other positions not listed here such as (1, 0, 0) and (1, 1, 0) would be in the classes with the values (0, 0, 1) and (0, 1, 1), respectively. This is because the direction of a position is arbitrary and can be read either way. This means that every position p in the set of positions can be mapped to some class P_n according to the $p's$ cell values. For example, let the cells in the position (A1, A2, A3) have the values (0, 0, 1) respectively. This would mean that the position (A1, A2, A3) would be in the class P_2. Additionally, note that if one of the cell's values were to change, the position may also change to a different class.

III. Solving a Board

A. Backtracking

A straightforward approach to solving a given board T_n is to use backtracking. This method is the most common search-based algorithm used when solving peg-solitaire [3]. The algorithm works by first getting the jumpable positions produced by T_n which are the positions in class P_4. Then, a jump is chosen at random, applied to the current board, and a recursive call is made on this altered board. This same process happens until the goal state is met, or every possible jump has been exhausted.

Algorithm 1 Backtracking

Input: A board T_n and the goal board T_n^{goal}
Output: An array of jumps J to get from T_n to T_n^{goal}.

```
if T_n = T_n^goal then
    return J
end if
jumpable_positions ← T_n's jumpable positions
for jump ∈ jumpable_positions do
    take jump, creating a new board T_n'
    J_{i+1} ← jump, where i is the length of J
    if Backtrack(T_n') ≠ false then
        return J
    end if
    undo jump j
    remove J_i
end for
return false
```

Though backtracking works to solve a given board, it has an average branching factor of 2.2, giving it an exponential rate of growth. Throughout the game, Matos explains that this branching factor increases for the first half of the game, then begins to decrease after that [2]. He explains that it initially increases because as the game progresses, each cell is more likely to be in a jumpable position. In the second half of the game, the number of available jumps for each cell begins to decrease at the same rate, since as each jump is made, the board becomes sparser which reduces the number of jumpable positions. Kiyomi and Matsui also discuss this concept in the context of an upper bound of the number of jumpable positions in their formalization of an integer programming solution to peg-solitaire [6]. While the branching factor changes over time, its average of 2.2 produces an exponential runtime.

IV. Improving the Algorithm

While the cause of the algorithm's poor performance is due to its large branching factor, there are a number of ways to improve run-time.

A. A Dynamic Programming Approach

An extension of the backtracking algorithm we use is dynamic programming. Dynamic programming is a useful method of improving the performance of recursive algorithms by storing specific data about subproblems so that this cached data can be referred to later.

Algorithm 2 Dynamically Programmed Backtracking

Input: A board T_n, the goal board T_n^{goal}, and a set of memorized states X
Output: An array of jumps J to get from T_n to T_n^{goal}.

```
if T_n = T_n^goal then
    return J
end if
if T_n ∈ X then
    return false
end if
jumpable_positions ← T_n's jumpable positions
for jump ∈ jumpable_positions do
    take jump, creating a new board T_n'
    J_{i+1} ← jump, where i is the length of J
    if Backtrack(T_n') ≠ false then
        return J
    end if
    undo jump j
    remove J_i
end for
add T_n to X
return false
```

This process requires three things to be true of the subproblems: they must be smaller, repeatable, and independent. We can show that these are true for the original backtracking algorithm. Each jump j moves the board T_n to a new board state $T_n^{'}$. This action reduces the number of cumulative jumps possible, making each $T_n^{'}$ an effective sub-problem of T_n. Additionally, multiple paths throughout the search tree achieve the same state, since some given board states can be found in multiple different ways. This means that there are numerous repeated sub-states throughout the search tree. Finally, the paths to two different sub-problems do not overlap, meaning the sub-problems are independent. Because of these truths, we can use dynamic programming to enhance our backtracking algorithm further.

Memoizing the states with no solution allows us to effectively prune bad paths by comparing and rejecting new states according to the previously memoized states. We can enhance this method by leveraging the symmetry of the triangular board and comparing states to mirrors and rotations of previously memoized states as well.

The main disadvantage of this method is that memoizing states will use a significant amount of memory. Considering that we incorporate rotations such that each saved board gives two additional unique boards to check, if there is no solution, we would be storing approximately one-third of the search tree.

B. Pagoda Functions

A common approach for further constraining the algorithm and reducing runtime is through pagoda functions, a strategy discussed by multiple authors [2]-[6]. These functions provide another method of finding unsolvable board states.

The pagoda function of a given board T_n is equal to the sum of the pagoda values of all T_n's occupied cells. After any given jump changes the board, the pagoda function of the new board will never increase [2]-[5]. If the pagoda function of the current board is less than the possible pagoda functions of any goal state, that board is unsolvable [2] [4]. There are two steps to incorporating pagoda functions in our algorithm: (1) implement an algorithm for finding pagoda values for variable board sizes, and (2) check the pagoda function of board states after each jump. Step 1 must occur before the start of the main backtracking solver method.

To use a pagoda function, each cell of a board c must firstly be assigned a pagoda value $P(c)$. A set of pagoda values is valid if, for any three collinear cells c_0, c_1, and c_2, the pagoda values of those cells, $P(c_1), P(c_2)$, and $P(c_3)$, are such that $P(c_0) + P(c_1) \geq P(c_2)$. Commonly, these pagoda values are limited to the domain $\{-1, 0, 1\}$ [Fig. 5].

There are infinitely many pagoda value sets, but only some are useful [2]. To find an infeasible board, for example, the pagoda value set must allow for a board's pagoda function to drop below the possible pagoda functions of goal states.

Pagoda values for the triangular peg solitaire board can be found using the backtracking search algorithm.

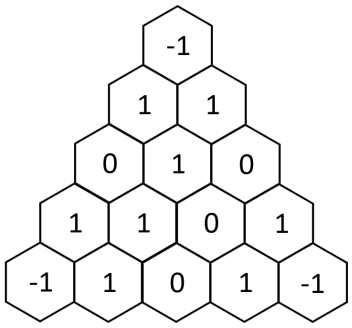


Fig. 5: A pagoda function for T_5. Note that only values within $\{-1, 0, 1\}$ are used.

For the purposes of this report, there are three constraints for this algorithm: (1) two of any three collinear cells must have pagoda values whose sum is greater than the pagoda value of the third, (2) the domain of the pagoda value of a cell is limited to $\{-1, 0, 1\}$, and (3) there must be at least two cells with the value -1. The third constraint ensures that the pagoda values produced are useful. Because of the bounds on the pagoda value domain, the minimum pagoda function for a goal board with a single peg is -1. This constraint allows the pagoda function to feasibly drop below this value. Algorithm 3 finds pagoda values using this method.

After valid pagoda values are found, we can add a pagoda function check into our main backtracking solver algorithm such that a branch is pruned if its pagoda function is below -1.

Algorithm 3 Map Pagoda Values

Input: A dictionary L_{pm} mapping cells to pagoda values, and a set L_{um} of cells that have yet to be mapped with a pagoda value

Output: A board mapping all cells to pagoda values, if it exists

```
current ← an unmapped cell from L_um
remove current from L_um
for pagoda_value ∈ {−1, 0, 1} do
    valid = true
    L_pm(current) ← pagoda_value
    for position p in T_n's set of positions do
        if cell c1, c2, c3 ∈ L_pm, where c1, c2, c2 ∈ p
        and (L_pm(c1) + L_pm(c2) >= L_pm(C3)
         or L_pm(c3)+L_pm(c2) >= L_pm(C1)) then
            valid = false
        end if
    end for
    if (L_um = ∅ and there are fewer than two −1
    pagoda values) then
        valid = false
        break
    end if
    if (valid) then
        return MapPagodaValues(L_pm, L_um)
    end if
end for
remove current from L_pm
add current to L_um
return false
```

V. Results

A. Normal Backtracking

When implementing the backtracking function, the branching factor was quickly evident, with an exponential growth in run-time as the board size n was incrementally increased (as discussed in Section III: A above).

We successfully obtained data for board sizes 5, 6, and 8; however, for a board sizes of 7 and 9, our backtracking algorithm ran for multiple hours with no solution, implying a remarkably long runtime. Data collected can be found in Fig. 6. in the next section.

B. Backtracking with Additional Algorithms

Implementing backtracking with the dynamic programming enhancements, including rotation and mirror functions to leverage symmetry resulted in much faster times when running trials. Implementing backtracking with both dynamic programming and pagoda function enhancements was still faster than backtracking by itself, but was slower than the dynamic programming enhancement on its own. A side-by-side comparison of trial times for basic backtracking, backtracking with dynamic programming, and backtracking with dynamic programming and pagoda together are shown in Fig. 6. In general, T_7 took longer than T_9 because T_7 has no solution, so the algorithm runs through the entire search tree.

The reduction in efficiency with the addition of the pagoda function is likely due to two reasons: (1) the runtime of checking the pagoda function and (2) the unique layout characteristics of a triangular peg solitaire board. In our implementation, checking the pagoda function involves summing the pagoda values of each occupied cell. If n is the side length of the triangular board and there are $n(n+1)/2$ occupied cells (the worst case), the pagoda function is an $O(n^2)$

N-value	Runtimes (s)			Solvable
	B	DP	DP+PG	
5	0.124	0.026	0.029	*True*
6	27.240	0.172	1.693	*True*
7	>5400	2748.993	>5400	*False*
8	754.346	0.814	1.731	*True*
9	>5400	1362.659	2443.473	*True*

Fig. 6: A table showing side-by-side trial times in seconds for backtracking (B), dynamic programming (DP), and dynamic programming with pagoda functions (DP+PG) for different board sizes (N).

operation. Performing this operation after each jump likely increased runtime significantly. The triangular peg solitaire board is also unique in that cells can be at the center of three positions rather than just two. For other boards where pagoda functions have been used, such as the cross-shaped and diamond-shaped boards [2] [6], cells are arranged in a rectangular coordinate system. In this system, each each cell is at the center of at most two positions. A cell's involvement in three positions significantly tightens constraints on pagoda value sets, limiting their usefulness. The added runtime of our pagoda function implementation together with this limited usefulness caused the pagoda function to reduce the effectiveness of our backtracking peg solver rather than enhance it. This was exemplified when we ran backtracking with pagoda functions alone for T_6. Basic backtracking solved this board in about 27.24 seconds, while the pagoda function enhancement took 44.89 seconds.

VI. Conclusion

Overall, the results show that backtracking with dynamic programming was an effective method of reducing runtime, while pagoda functions were ineffective and even detrimental to runtime. The pagoda functions' inefficacy is again most likely because of the $O(n^2)$ runtime for each pagoda function check and the fact that a cell can be at the center of up to three unique positions rather than two. Dynamic programming, however, was successful in reducing runtime, with its main disadvantage being memory complexity.

With topmost cell as the initial vacancy position, solutions were found for all board sizes tested except when $n = 7$. Because this n-value is unsolvable, the algorithm would run through the entire search tree and exhibit exceptionally long program runtimes.

We obtained consistent results for low board sizes, particularly using Dynamic Problem Solving methods, but board sizes over $n = 9$ had much larger runtimes. There are alternative methods beyond those discussed in this report for solving triangular boards such as purging [4], which may be worth exploring in further research. Further optimizing the pagoda functions would also be worth exploring further.

References

[1] A. Matos, 'Depth-first search solves Peg Solitaire', 11 2002.

[2] C. Jefferson, A. Miguel, I. Miguel, and S. A. Tarim, 'Modelling and solving English Peg Solitaire', Computers & Operations Research, vol. 33, no. 10, pp. 2935–2959, 2006.

[3] E. Berlekamp, J. Conway and R. Guy, Purging pegs properly, in Winning Ways for Your Mathematical Plays, 2nd ed., Vol. 4, Chap. 23, A. K. Peters, 2004, pp. 803–841.

[4] G. I. Bell, 'Solving Triangular Peg Solitaire'. arXiv, 2007.

[5] J. K. Barker and R. E. Korf, 'Solving peg solitaire with bidirectional BFIDA', Proceedings of the National Conference on Artificial Intelligence, vol. 1, pp. 420–426, 01 2012

[6] Masashi Kiyomi and Tomomi Matsui. Integer programming based algorithms for peg solitaire problems. In Computers and Games, page 229240. Springer-Verlag, 2001.