Solving the Triangle Peg Game

Arnold C. Baldoza
arnold.baldoza@gmail.com

Abstract. A C++ program using a depth-first search algorithm was developed to solve the 15-Peg Triangle Board Game. This game has 438,984 unique solutions out of the possible 7,335,390 outcomes. Optimizations to the baseline implementation employed a mechanism to abandon a potential solution once a board configuration had been determined to be unsolvable (i.e., does not result into a valid ending configuration) and a bitboard representation of the board. These optimizations produced significant reductions in processing time.

Keywords: triangle peg game, artificial intelligence, depth-first search

1 Introduction

When visiting a Cracker Barrel restaurant, one can find a 15-peg triangle board game on each table. The game consists of a small wooden triangle board with 15 holes and with 14 pegs. In this paper, we identify each hole by incrementally numbering them from zero to 14, starting from the apex of the triangle. Figure 1 provides an image of the 15-peg triangle board and the label corresponding to each peg location.

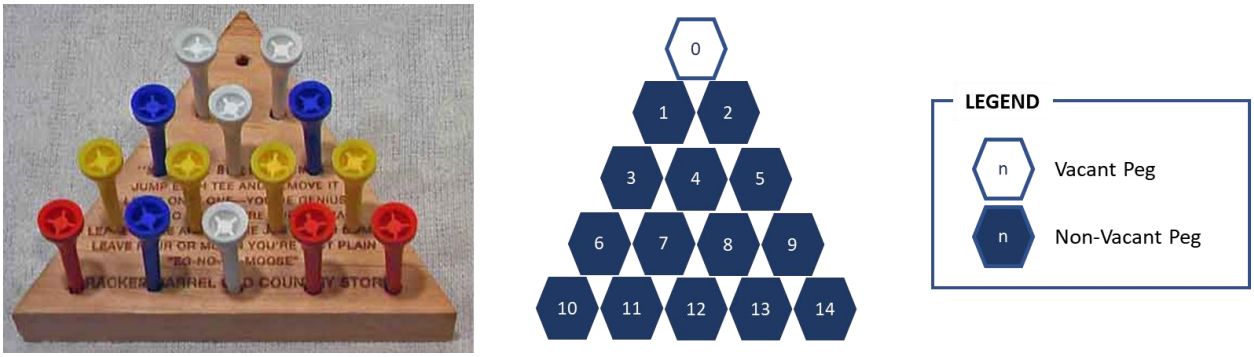


Figure 1. The 15-Peg Triangle Board Game[1] and the corresponding board notation.

The game starts with a peg in every hole, except one, which is referred to as the **starting vacancy**. A player **moves** or **jumps** one peg over another into a vacancy on the board, removing

[1] Joe Nord, "Triangle Peg Board Game – Solutions to Amaze Your Friends," https://www.joenord.com/puzzles/peggame/index.html (last accessed November 28, 2020).

the peg that was jumped over. To denote a jump, we will identify the starting and ending positions separated by a dash. For example, Figure 2 illustrates jump 3 – 0 from an initial position with a starting vacancy of 0. From the initial position, Peg 3 can jump Peg 1 as there is a vacant peg location adjacent to Peg 2 (i.e., Peg 0). When executing the 3 – 0 jump, Peg 3 is moved to Peg 0 and Peg 1 is removed.

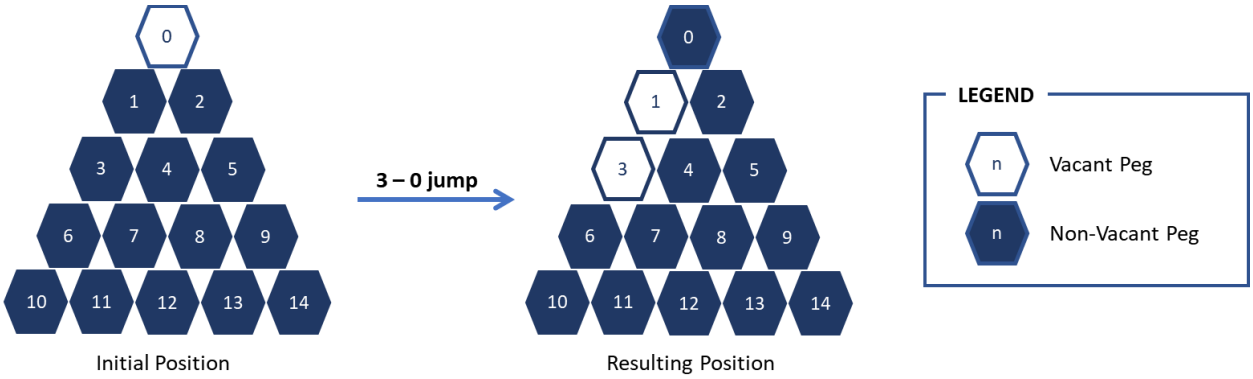


Figure 2. Illustrating a 3 – 0 jump.

Figure 3 provides an alternative depiction of the 3 – 0 jump, starting from an initial position with a starting vacancy of 0. This alternative depiction may be useful in illustrating jumps than the 3 – 0 notation.

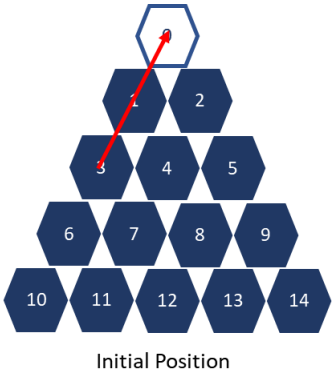


Figure 3. Alternative Depiction of the 3 - 0 jump.

To denote multiple jumps of the same peg, instead of listing each jump separately, we will shorten the notation by specifying the starting peg location, the intermediary peg locations, and the final peg location. For example, instead of specifying the multiple jumps depicted in Figure 4 as a series of moves (i.e., 3 – 0, 0 – 5, 5 – 12), we will simply notate it as a single move annotated as 3 – 0 – 5 – 12.

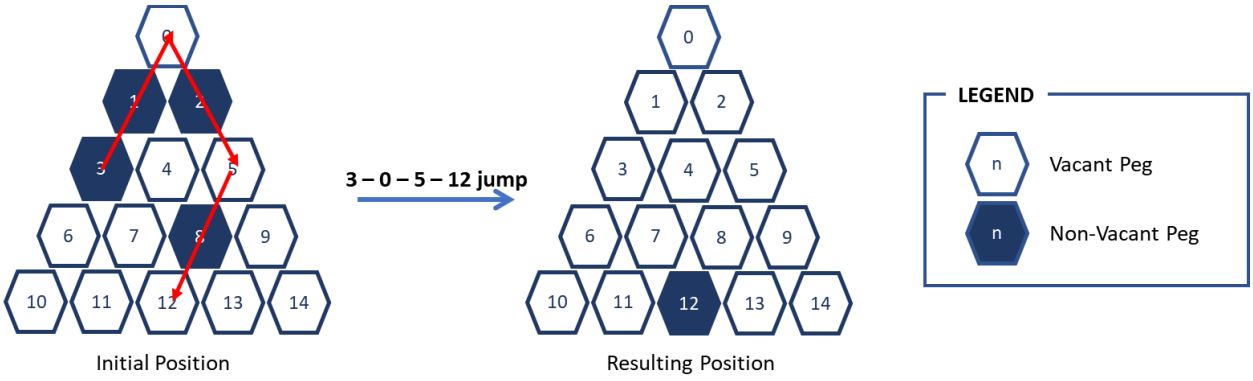


Figure 4. A Multiple Jump Move.

In general, the objective of the game is to finish in a position that has only one peg in any peg position. Variations of the game allow for different ending configurations[2], but for the purposes of this paper, we will consider the game to end when only one peg remains, regardless on the position of the peg.

2 Methods

We coded a C++ program that finds all possible winning solutions given a specified starting position. This program employs the following Depth-First-Search algorithm:

[2] Bell, George I., "Solving Triangular Peg Solitaire," *Journal of Integer Sequences*, Vol. 11 (2008), Article 08.4.8, https://cs.uwaterloo.ca/journals/JIS/VOL11/Bell/bell2.pdf (last accessed November 28, 2020); Proctor, Grover B. Jr, "Cracker Barrel Conundrum," https://grover.news/2016/03/15/cracker-barrel-conundrum/ (last accessed November 28, 2020).

```
Depth-First-Search (board) {
      If (board is in winning configuration (namely, has only one peg left) {
            // Solution has been found
            Show-The-Solution()
      } Else {
            If (Board has no available moves) {
                  Board is unsolvable
            } Else {
                  For each available move {
                        nextBoard = Board.Perform Move
                        Keep-Track-Of-Solution()
                        Depth-First-Search(nextBoard)
                  }
            }
      }
}
```

The initial implementation of the program employed an array representation of the triangular peg board. Namely,

```
#define NUMBER_OF_PEGS 15

enum PEGSTATUS { Empty = 0, Full = 1 };

class Board
{
public:
      PEGSTATUS Pegs[NUMBER_OF_PEGS];

      void Initialize(int emptyPeg);
      bool isEqual(Board p);
      int RemainingPegs(void);
      void ShowBoard(void);
};
```

This representation is relatively slow. Optimizations to the data structures used in the program can be found in the Discussions section.

3 Results

There are 438,984 unique solutions out of the possible 7,335,390 outcomes. Table 1 provides the solution space of the game for each of the 14 different starting configuration.

Table 1. Solution Space of the Triangle Peg Game.

Starting Vacancy	Number of Games with Solutions	Number of Games with No Solutions	Total Number of Games	% of Games
0	29,760	538,870	568,630	5.2%
1	14,880	279,663	294,543	5.1%
2	14,880	279,663	294,543	5.1%
3	85,258	1,064,310	1,149,568	7.4%
4	1,550	136,296	137,846	1.1%
5	85,258	1,064,310	1,149,568	7.4%
6	14,880	279,663	294,543	5.1%
7	1,550	136,296	137,846	1.1%
8	1,550	136,296	137,846	1.1%
9	14,880	279,663	294,543	5.1%
10	29,760	538,870	568,630	5.2%
11	14,880	279,663	294,543	5.1%
12	85,258	1,064,310	1,149,568	7.4%
13	14,880	279,663	294,543	5.1%
14	29,760	538,870	568,630	5.2%
	438,984	**6,896,406**	**7,335,390**	**6.0%**

As confirmed by data, we found that the puzzle has four equivalent starting positions. These equivalent classes reflect either a rotation of the board or a mirroring of the board along its vertical access. The equivalent starting positions are:

Table 2. Equivalent Starting Vacancy Classes.

Class	Equivalent Starting Vacancy
A	0, 10, 14
B	1, 2, 6, 9, 11, 13
C	3, 5, 12
D	4, 7, 8

As one can note in Figure 5, Classes A, C, and D are 120-degree rotations of a board with a particular starting vacancy. For example, the board with starting vacancy of 0 can be rotated 120-degree counter-clockwise to produce an equivalent board with a starting vacancy of 10.

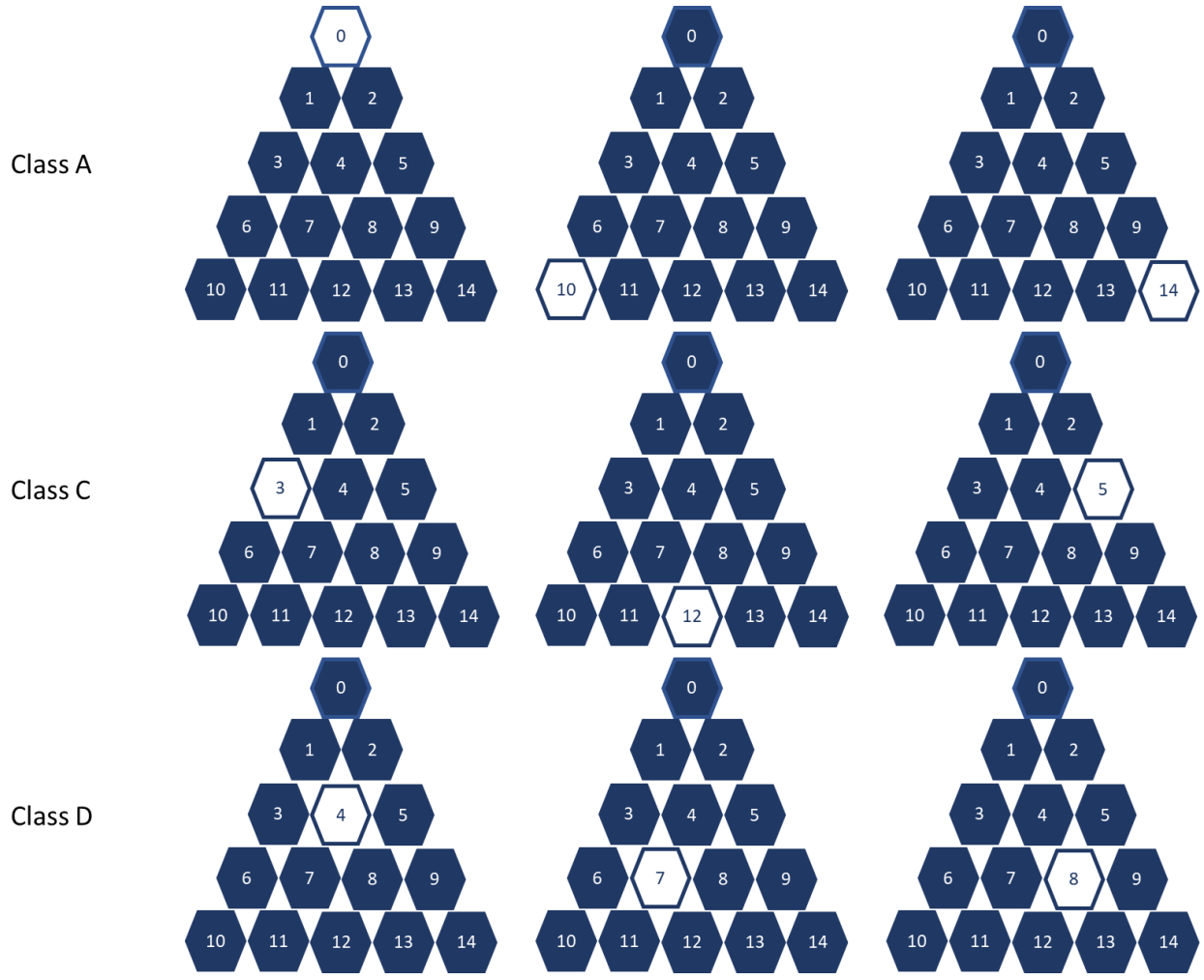


Figure 5. Class A, C, and D Starting Vacancy Equivalents.

.

Similarly, Class B starting position equivalents are rotations or reflections or both of a board with a particular starting vacancy around the vertical axis. For example, the board with starting vacancy of 1 can be rotated 120-degree counter-clockwise to produce an equivalent board with a starting vacancy of 11 and then can be reflected along its vertical axis to produce an equivalent board with a starting vacancy of 13.

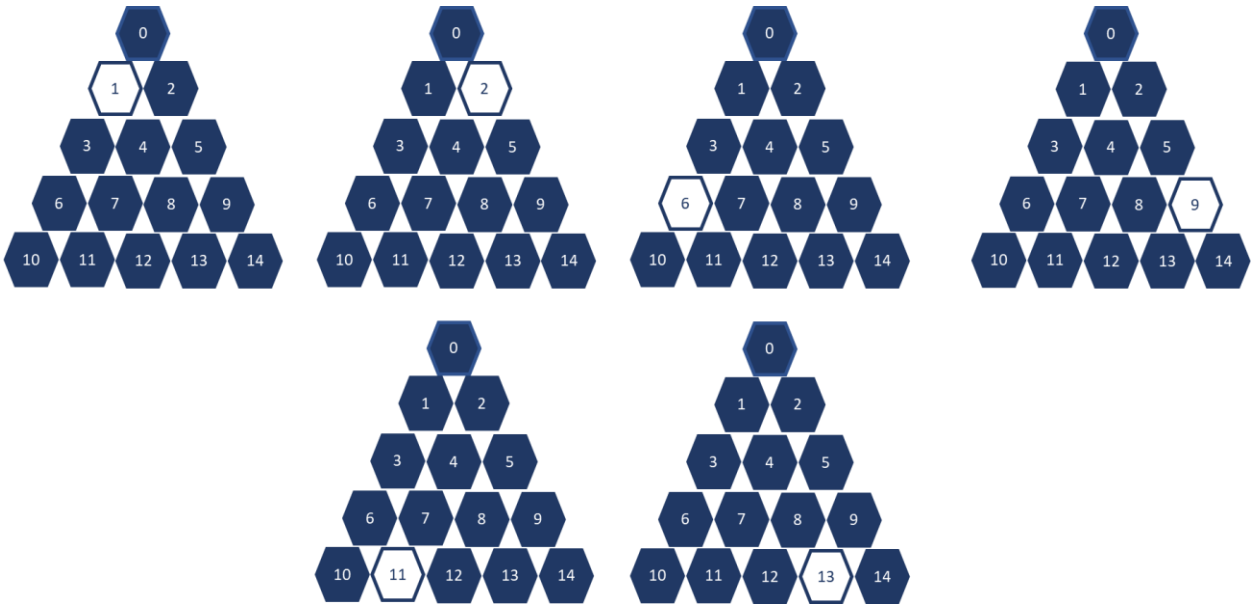


Figure 6. Class B Starting Vacancy Equivalents.

We use Starting Vacancy Equivalents to reduce the number of starting vacancies explored when performing timing runs during optimization. Instead of timing all 14 starting vacancy configurations, we only timed four runs, each run associated with a different equivalent class of starting configurations. Sample solutions for each Starting Position Class can be found in Appendix A.

We also captured timing statistics for our implementation. These timing statistics allow us to quantify the changes to the data structures used in the program as we introduced optimizations to reduce processing times. The specifications of the computer used in capturing these timing statistics are:

- Intel® Core™ i7-8700 CPU @ 3.20 GHz 3.19 GHz
- 6 Physical Cores
- 12 Logical Processors
- 32.0 GB RAM

This initial implementation resulted in the runtimes provided in Table 3. Because we did not have control of the processes that were concurrently executing in the background, we ran each starting position class five times to get an average runtime.

Table 3. Runtimes for Baseline Implementation.

Starting Position Class	Number of Games with Solutions	Number of Games with No Solutions	Total Number of Games	Run 1 (sec)	Run 2 (sec)	Run 3 (sec)	Run 4 (sec)	Run 5 (sec)	Average Duration (sec)
A	29,760	538,870	568,630	4.234	4.191	4.184	4.259	4.258	4.225
B	14,880	279,663	294,543	2.190	2.163	2.220	2.171	2.104	2.170
C	85,258	1,064,310	1,149,568	8.488	8.354	8.442	8.486	8.328	8.420
D	1,550	136,296	137,846	1.018	1.028	1.013	1.036	0.970	1.013

4 Analysis

Reviewing our code, we identified certain board-related functions (i.e., `Board::isEqual()` and `PegBoard::CopyBoard()`) accounted for a significant portion of processing time. An array representation of the triangular peg board does not lend itself to efficient algorithms that would determine if two boards were equal and that would copy one board into another. A bitboard representation of the board was used in the first optimization effort.

A basic depth-first search algorithm would terminate the search for a solution if (a) a sequence of moves resulted in a valid ending configuration or (b) there are no valid moves to perform. We improve the performance of the search algorithm by adding another termination criterion – if the board configuration had been previously identified as not being transformable through a sequence of valid moves to a valid ending configuration. This mechanism was included under the second optimization effort.

Consistent with our concept of Starting Vacancy Equivalents, we can increase the efficiency of this additional termination criterion by not only tracking board configurations that

are unsolvable, but by also tracking the rotations and reflections about the vertical axis of these board configurations. This optimization has not yet been implemented.

Moves were generated by looking at all possible moves and determining which ones were valid in the current board configuration. Move generation was invoked during every recursive call to the depth-first search algorithm as long as the current board configuration was not in a valid ending configuration. There may exist a more efficient method of generating all valid moves, but this is left as future work.

5 Discussion

First Optimization: Bitboard Representation

We identified that certain board-related functions (i.e., `Board::isEqual()` and `PegBoard::CopyBoard()`) accounted for a significant portion of processing time. The initial implementation of `Board::isEqual()` is O(n/2) as it, on the average, was required to look at half of the board before it could determine that the two boards were not equal.

```
bool Board::isEqual(Board p) {
	bool r = true;
	int i = 0;
	while ((i < NUMBER_OF_PEGS) && (r)) {
		r = (Pegs[i] == p.Pegs[i]);
		i++;
	}
	return r;
}
```

Similarly, `PegBoard::CopyBoard()`) was O(NUMBER_OF_PEGS) as it copied each peg on the board.

```
void PegBoard::CopyBoard(PegBoard src) {
      for (int i = 0; i < NumberOfPegs; i++)
            SetPeg(i, src.GetPeg(i));
      pathTo = src.pathTo;
      boardSolvable = src.boardSolvable;
}
```

To reduce the processing time associated with these functions, we changed the implementation of the board. Instead of representing the board as an array (i.e., `PEGSTATUS Pegs[NUMBER_OF_PEGS]`), we represented the board as a single `UNSIGNED SHORT INT`. This representation uses a bit to represent each peg – if the bit is zero, the peg location is vacant; if the bit is one, the peg location contains a peg (i.e., not vacant).

As a result of this new representation, `Board::isEqual()` becomes O(1) as it is a simple comparison between two `UNSIGNED SHORT INT`s.

```
bool Board::isEqual(Board p) {
      return (Pegs == p.Pegs);
}
```

Similarly, `PegBoard::CopyBoard()` is O(1) as it is transformed into a simple assignment.

```
void PegBoard::CopyBoard(PegBoard src) {
       Board = src.Board;
       pathTo = src.pathTo;
       boardSolvable = src.boardSolvable;
}
```

An additional heuristic was implemented in an attempt to reduce processing times. With the bitwise representation of the board, determining if a peg at a specified position (pos) contains a peg or not can be performed via the following operation:

```
Pegs & (1 << pos)
```

where `Pegs` is the `UNSIGNED SHORT INT` representing the board and `pos` is the position being queried. If the result is equal to zero, the peg at position pos is zero else it is one. To speed up the program, we implemented an array that contained the result of `(1 << pos)` for every position on the board. Specifically, we implemented:

```
#define NUMBER_OF_PEGS 15

const unsigned short FULL_BOARD = 0b0111111111111111;
const unsigned short int EMPTY_PEG = 0b0000000000000000;

const unsigned short FULL_PEG[NUMBER_OF_PEGS] = {
0b0000000000000001,
0b0000000000000010,
0b0000000000000100,
0b0000000000001000,
0b0000000000010000,
0b0000000000100000,
0b0000000001000000,
0b0000000010000000,
0b0000000100000000,
0b0000001000000000,
0b0000010000000000,
0b0000100000000000,
0b0001000000000000,
0b0010000000000000,
0b0100000000000000 };
```

As a result, the contents of the board at position pos can be queried via the formula

```
Pegs & (FULL_PEG[pos])
```

Runtimes associated with the bit-wise representation optimization are in Table 4.

Table 4. Runtimes for Implementation with the Bit-Wise Representation Optimization.

Starting Position Class	Run 1 (sec)	Run 2 (sec)	Run 3 (sec)	Run 4 (sec)	Run 5 (sec)	Average Duration (sec)	% Improvement over Baseline
A	4.046	3.976	3.959	3.981	3.956	3.984	5.72%
B	2.040	2.011	2.059	2.036	2.040	2.037	6.10%
C	7.914	7.886	8.015	7.906	7.929	7.930	5.82%
D	0.958	0.960	0.978	0.962	0.096	0.791	21.93%

Second Optimization: Tracking Unsolvable Configurations

To speed up the implementation, the first optimization tracked unsolvable configurations (i.e., configurations that did not result into a valid ending configuration). A board is considered unsolvable if (a) the board is not in a valid ending configuration and there are no valid moves left to perform, or (b) the board is not in a valid ending configuration and all moves from the current configuration does not lead to a valid ending configuration. With this in mind our depth-first search algorithm was modified as follows:

```
Depth-First-Search-With-LookUp (board) {
      If (board is in winning configuration) {
            // Solution has been found
            Show-The-Solution()
      } Else {
            If (Board has no available moves) {
                  Board is unsolvable
            } Else {
                  If Board is on the Unsolvable List {
                        Don't True to Solve it
                  } Else {
                        If No Available Moves
                              Add to Unsolvable List
                        Else {
                              For each available move {
                                    nextBoard = Board.Perform Move
                                    Keep-Track-Of-Solution()
                                    Depth-First-Search(nextBoard)
                                    Keep-Track-If-Any-Child-Is-Solvable
                              }
                              If No Child Was Solvable {
                                    Add to Unsolvable List
                              }
                        }
                  }
            }
      }
}
```

Employing this optimization, significantly less board configurations were visited as the algorithm did not further explore configurations that had been previously identified as unsolvable. Specifically, 40.9% of the games were halted when an unsolvable configuration was identified:

Table 5. Percentage of Games Halted by Unsolvable Configuration.

Starting Position Class	Number of Games with Solutions	Number of Games with No Solutions	Number of Unsolvable Boards Identified	Number of Games Halted By LookUp	% of Halted Games versus Total Number of Games
A	29,760	538,870	99	236,401	41.6%
B	14,880	279,663	463	128,932	43.8%
C	85,258	1,064,310	114	461,653	40.2%
D	1,550	136,296	557	52,845	38.3%

Consequently, the runtimes associated with each starting position class were faster than the non-optimized version of the program. Table 6 documents the runtimes for implementing the Look-Up table, but not the bit-wise representation.

Table 6. Runtimes for Optimization with Tracking Mechanism Only.

Starting Position Class	Run 1 (sec)	Run 2 (sec)	Run 3 (sec)	Run 4 (sec)	Run 5 (sec)	Average Duration (sec)	% Improvement Over Baseline
A	3.105	2.939	2.897	2.562	2.709	2.842	32.73%
B	1.637	1.656	1.656	1.443	1.496	1.578	27.29%
C	6.062	6.105	5.972	5.387	5.364	5.778	31.37%
D	0.675	0.667	0.584	0.585	0.586	0.619	38.85%

Table 7 lists the runtimes for the implementation of both the Look-Up table and the bitwise representation.

Table 7. Runtimes for Bit-Wise Representation and Tracking Mechanism Implementation.

Starting Position Class	Run 1 (sec)	Run 2 (sec)	Run 3 (sec)	Run 4 (sec)	Run 5 (sec)	Average Duration (sec)	% Improvement Over Baseline
A	1.716	1.665	1.673	1.684	1.691	1.686	60.10%
B	0.828	0.825	0.826	0.823	0.822	0.825	61.98%
C	3.552	3.597	3.577	6.559	3.571	4.171	50.46%
D	0.289	0.284	0.285	0.296	0.290	0.289	71.49%

Future Work

Other optimizations may be identified to reduce the processing required to produce all possible solutions given an initial configuration. We identified some future optimizations in the Analysis section. They are:

- Tracking the rotations and reflections of unsolvable board configurations; and
- More efficiently generating available moves.

In addition to these, future work may include:

- In its current implementation, the program resets the list of unsolvable board configurations after every run. Consequently, this list is not shared with subsequent runs. A global list may be implemented to capture ALL unsolvable positions. This optimization may speed up the algorithms as the list of unsolvable board configurations grows with future iterations.
- We currently employ a iterative function that counts the number of pegs on the board. However, based on the rules of the game, every jump reduces the number of remaining pegs by one. This fact can be used to employ a variable to keep track of the remaining pegs. This variable would be initialized to `NUMBER_OF_PEGS` and would be reduced by one after every move.
- Currently, the algorithm displays the solution once it is found. It does not keep track of each solution found. If needed, one can employ a list of solutions and push to this list to be able to access the list of solutions later.
- Solving Variations of the Game. The current program considers the game solved when only one peg remains, regardless of the position of the peg. The program may be tailored to solve for other (winning) ending configurations.

Code, Data, and Materials Availability

The computer software code can be found at https://github.com/arnoldbaldoza and can be used freely in accordance to the GNU .

References

1. Bell, George I., "Solving Triangular Peg Solitaire," *Journal of Integer Sequences*, Vol. 11 (2008), Article 08.4.8, https://cs.uwaterloo.ca/journals/JIS/VOL11/Bell/bell2.pdf (last accessed November 28, 2020);
2. Proctor, Grover B. Jr, "Cracker Barrel Conundrum," https://grover.news/2016/03/15/cracker-barrel-conundrum/ (last accessed November 28, 2020).

Arnold Baldoza is a free-lance programmer. He received his BS degree in computer science from Ateneo de Manila University in 1990, a BS degree in computer engineering from the University of Arizona in 1996, a MS degree in computer engineering from Syracuse University in 1999, and a MA degree in homeland security and defense from the Naval Postgraduate School. His current research interests include game theory, neural networks, and reinforcement learning.

Caption List

Fig. 1 The 15-Peg Triangle Board Game and corresponding board notation.

Fig. 2 Illustrating a 3 – 0 jump.

Fig. 3 Alternative Depiction of the 3 – 0 jump.

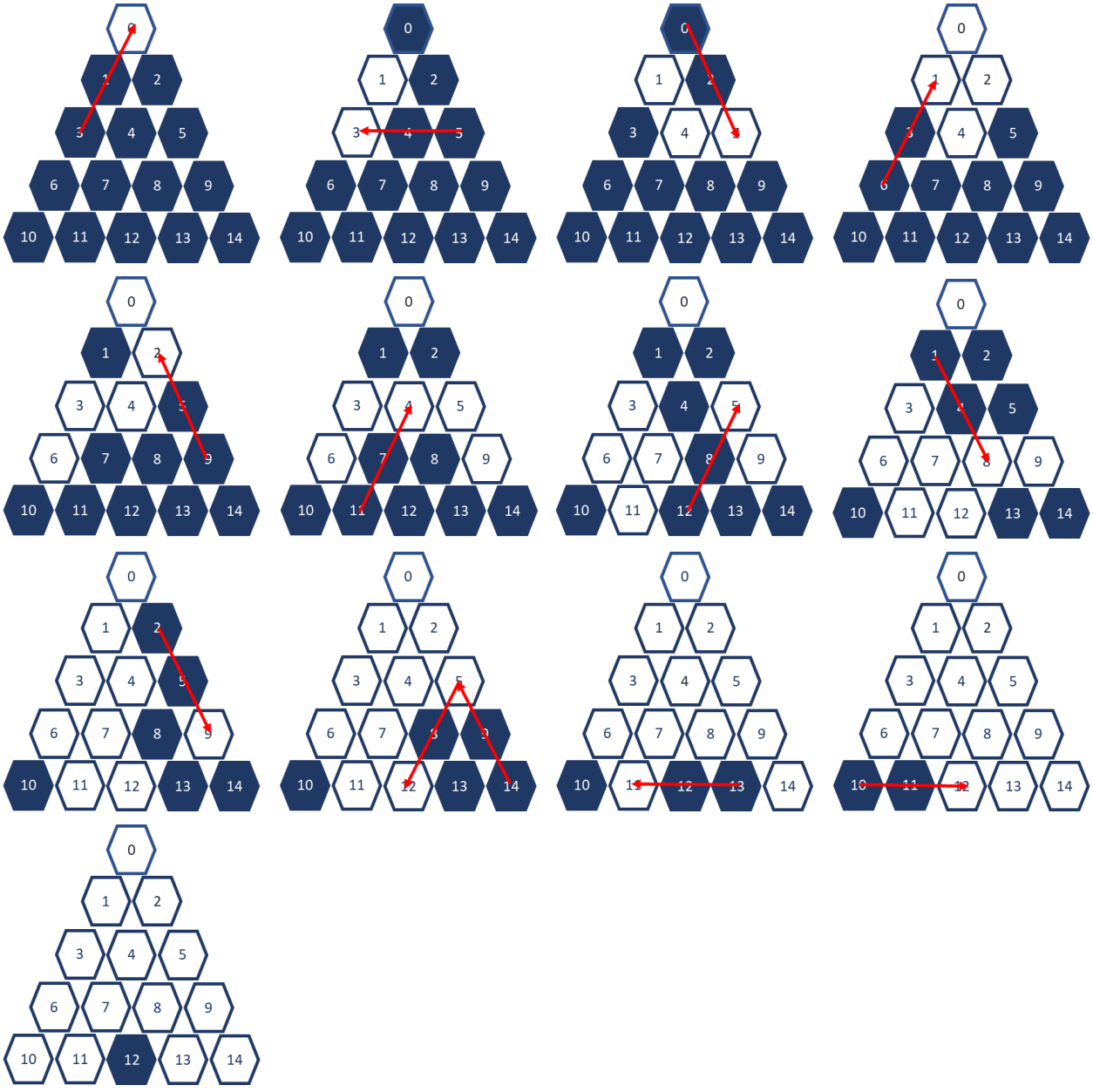
Fig. 4 A Multiple Jump Move.

Table 1 Solution Space of the Triangle Peg Game.

Appendix A

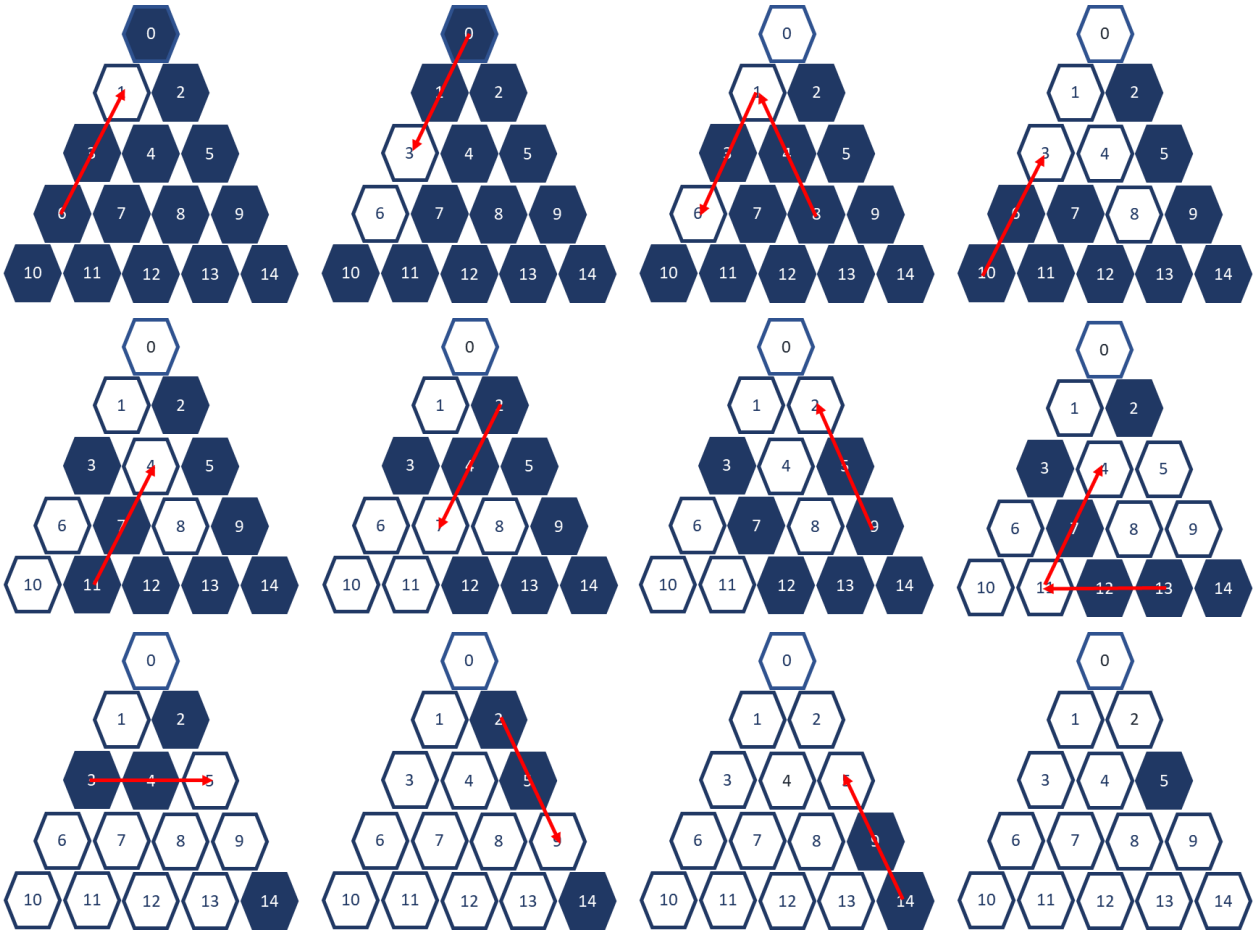
A.1 Sample Solution for Class A (Starting Vacancy of 0)

3 – 0; 5 – 3; 0 – 5; 6 – 1; 9 – 2; 11 – 4; 12 – 5; 1 – 8; 2 – 9; 14 – 5 – 12; 13 – 11; 10 – 12



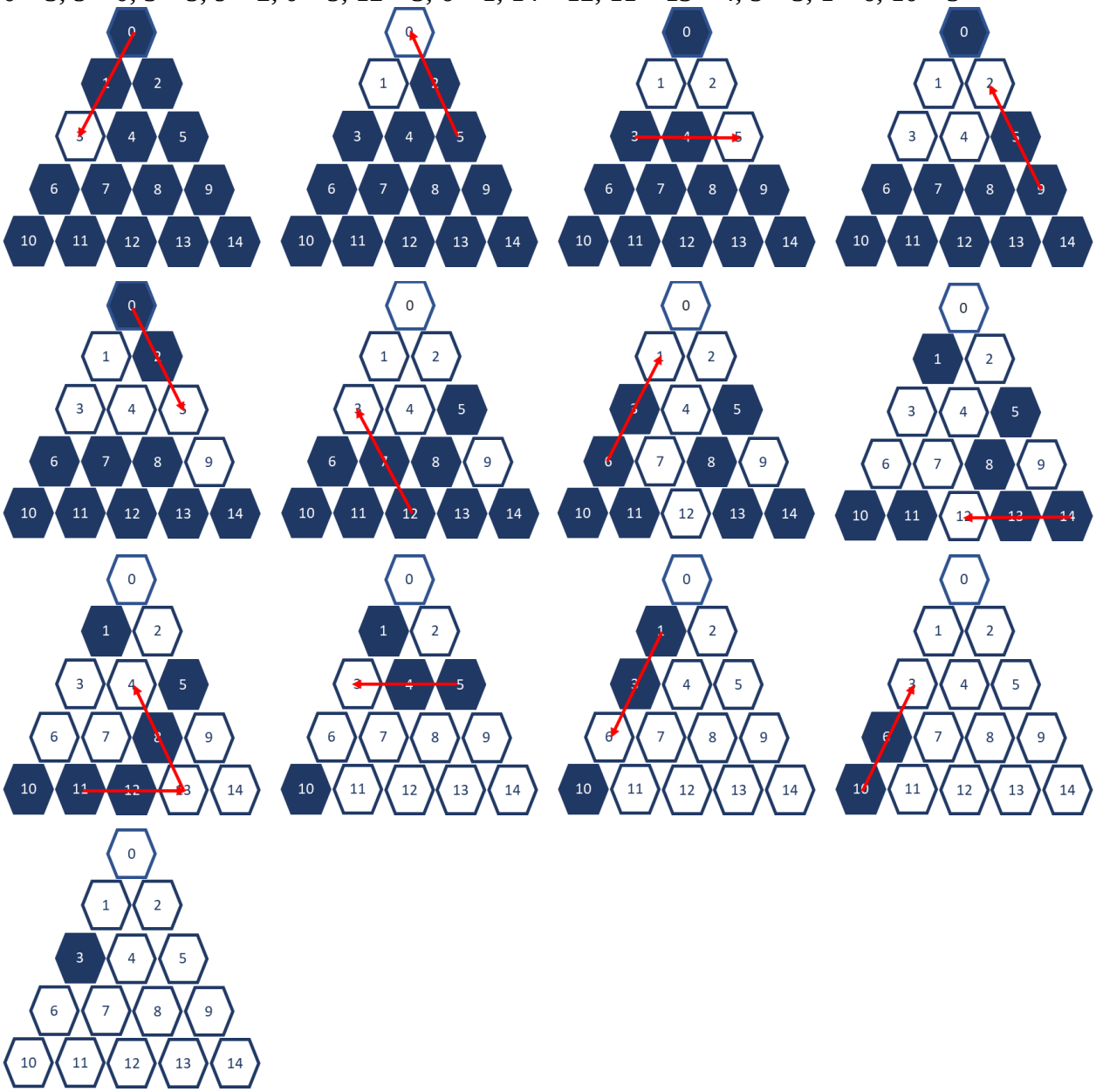
A.2 Sample Solution for Class B (Starting Vacancy of 1)

6 – 1; 0 – 3; 8 – 1 – 6; 10 – 3; 11 – 4; 2 – 7; 9 – 2; 13 – 11 – 4; 3 – 5; 2 – 9; 14 – 5



A.3 Sample Solution for Class C (Starting Vacancy of 3)[3]

0 – 3; 5 – 0; 3 – 5; 9 – 2; 0 – 5; 12 – 3; 6 – 1; 14 – 12; 11 – 13 – 4; 5 – 3; 1 – 6; 10 – 3



[3] The sample solution with a starting vacancy of 3 is also an example of a solution to a **complement problem**, or **reversal**, where the starting vacancy and the ending vacancy are the same position.

A.4 Sample Solution for Class D (Starting Vacancy of 4)

11 – 4; 9 – 7; 1 – 8; 2 – 9; 6 – 1; 0 – 3; 13 – 11; 3 – 12; 11 – 13; 14 – 5 – 12; 13 – 11; 10 – 12