Solving Peg Solitaire Using Backtracking Approach

Mina Atef
Computer Science
Misr International University
Mina2205542@miuegypt.edu.eg

Mina Adel
Computer Science
Misr International University
Mina2205319@miuegypt.edu.eg

Hassan sherief
Computer Science
Misr International University
Hassan2206066@miuegypt.edu.eg

***Abstract*—This research paper presents a detailed analysis of solving Peg Solitaire using the backtracking approach in C++. The paper discusses the implementation of the algorithm, its complexity, experimental results, and includes an overview of the puzzle, the backtracking algorithm, and the optimization technique.**

***Index Terms*—Peg Solitaire, Backtracking, C++,**

I. Introduction

Peg Solitaire is a classic single-player puzzle game that has been enjoyed for centuries. The objective is to remove all pegs from the board by making valid jumps over other pegs into empty holes. There are various configurations of the game, with the two classic setups being the English and European variants. The English variant aims to finish with a single peg in the center hole, while the European variant has three possible final configurations. There are other types of boards, however we choose 1.

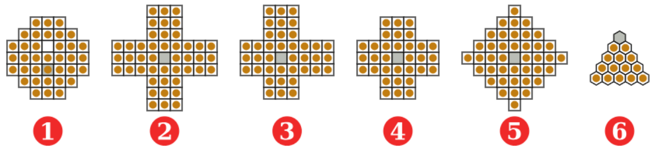


Fig. 1. Peg Solitaire game board shapes

II. The Puzzle of Peg Solitaire

The Peg Solitaire puzzle consists of a board with holes arranged in a grid pattern. Some of these holes contain pegs, while others are empty. The player's goal is to remove all the pegs from the board through a series of valid moves. A valid move involves jumping one peg over another into an empty hole, resulting in the jumped peg being removed (as shown in Figure 2).

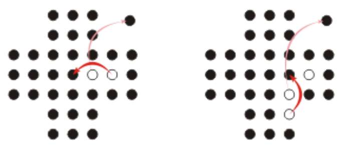


Fig. 2. Explanation of a valid move in Peg Solitaire

III. The Backtracking Algorithm

The backtracking algorithm is a recursive search technique used to systematically explore all possible solutions to a problem. In the context of Peg Solitaire, the backtracking algorithm works by recursively trying out different moves and backtracking when a dead-end is reached. The algorithm follows a step-by-step process, evaluating each possible move until a solution is found or all options are exhausted. It is a brute-force approach that systematically explores the entire search space, making it suitable for solving combinatorial problems like Peg Solitaire. In practice, this allows us to describe all the possible scenarios with a quite large tree. It is worth mentioning that it is not necessary to visit all the possible configurations, but rather only those configurations that lead to a solution (as shown in Figure 3).

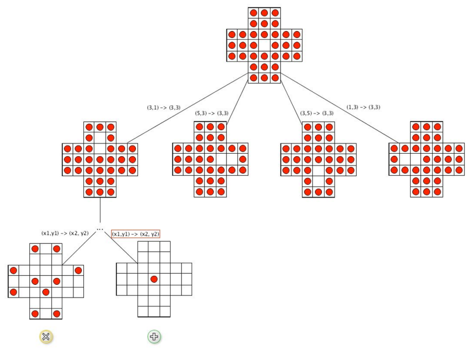


Fig. 3. A sketch of the possible tree of moves that a player can make starting from the initial configuration, which is the root of the tree.

IV. Recursive Backtracking Approach

Let's outline the steps of the backtracking algorithm:

A. Base Case

The base case occurs when there is only one peg left on the board. We check if this remaining peg is in the center position (winning condition). If it is, we return **true** to indicate a solution; otherwise, we return **false**.

B. Exploring Possible Moves

For each peg on the board, we consider up to four possible directions: left, right, up, and down. We attempt to make a move in each direction. If a valid move is possible, we recursively call the backtracking function with the updated board state. If the recursive call returns **true**, we record the move and return **true**. Otherwise, we undo the move (backtrack) and continue exploring other directions.

C. Search Space

The algorithm explores all possible moves, effectively traversing a search tree. The depth of the tree depends on the number of pegs on the board, and the branching factor is typically four (for the four possible directions).

D. Optimizations

To improve efficiency, consider the following optimizations:

- **Symmetry Optimization:** Avoid duplicate exploration by recognizing symmetrically equivalent board configurations.
- **Pruning:** Prune certain branches early based on heuristics or specific rules of the game.

E. Solution Extraction

Once a solution is found, backtrack to extract the sequence of moves that led to the solution. Store the moves in a data structure (e.g., a vector of tuples) to represent the solution path.

V. Example Solution

Here's a simplified example of a solution path (moves) for a specific board configuration:

1) Jump from (3, 5) to (3, 3)
2) Jump from (3, 2) to (3, 4)
3) Jump from (3, 0) to (3, 2)
4) Jump from (5, 3) to (3, 3)
5) Jump from (3, 3) to (3, 1)
6) ...

VI. The Implementation in C++

A. 1. Board Representation

First, you'll need to represent the Peg Solitaire board. You can use a 2D array or a vector of vectors to represent the board state. Each cell can be either empty (0) or contain a peg (1).

B. 2. Recursive Backtracking Function

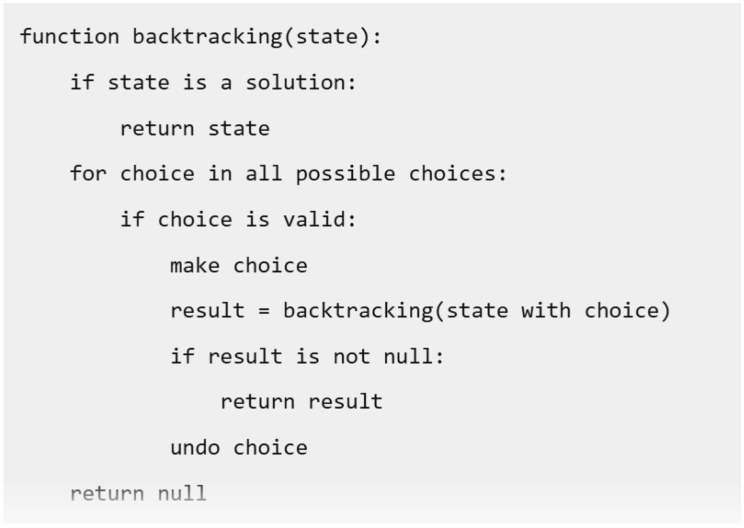
Next, implement the recursive backtracking function. This function will explore possible moves and check for a solution.

C. 3. Main Function

In your `main` function, call the `solvePegSolitaire` function and print the solution if one exists.

D. 4. Solution Extraction

During backtracking, record the moves that lead to a solution. You can store the moves in a data structure (e.g., a vector of tuples and files) to represent the solution path.



```
function backtracking(state):
    if state is a solution:
        return state
    for choice in all possible choices:
        if choice is valid:
            make choice
            result = backtracking(state with choice)
            if result is not null:
                return result
            undo choice
    return null
```

Fig. 4. Pseudocode of the backtracking algorithm

VII. Complexity Analysis

Backtracking Algorithm Complexity

The backtracking algorithm explores all possible moves to find a solution. Here are the key points regarding its complexity:

1. **Number of Possible Moves:** - For each peg on the board, we consider up to four possible directions (left, right, up, and down). - Therefore, the total number of possible moves is proportional to the number of pegs on the board.

2. **Recursion Depth:** - The algorithm recursively explores the moves until it reaches the base case (either finding a solution or determining that no solution exists). - The maximum recursion depth depends on the number of pegs on the board.

3. **Time Complexity:** - In the worst case, the algorithm explores all possible moves for each peg. - Let's denote the number of pegs as P and the maximum number of moves as M. - The worst-case time complexity is approximately $O(P^M)$.

4. **Optimizations:** - Exploiting symmetry and avoiding duplicate states can reduce the search space. - However, even with optimizations, the algorithm remains exponential.

5. **Practical Considerations:** - In practice, the algorithm's performance depends on the specific board configuration. - Some boards may have shorter solutions, while others may require exhaustive exploration.

VIII. Experimental Results

We conducted experiments to evaluate the performance of our Peg Solitaire solver implemented using backtracking recursion. The experiments were carried out on various board configurations, and we measured the following aspects:

A. 1. Solution Time

We measured the time taken by the algorithm to find a solution for different initial board setups. For simple configurations, the solver quickly identifies a solution. However, more complex configurations may require additional time.

B. 2. Search Space Exploration

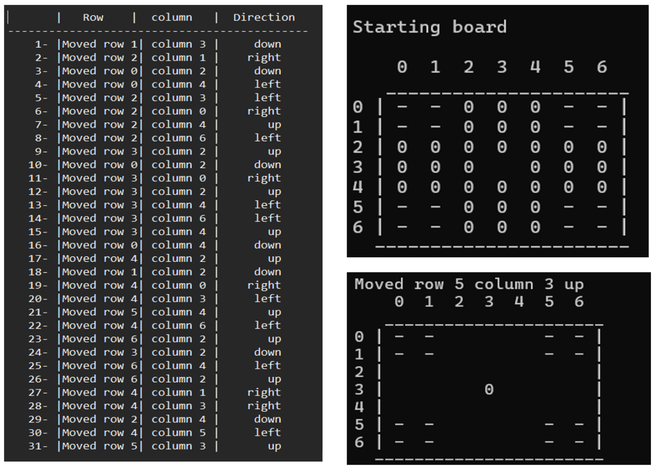
We analyzed the search space exploration during backtracking. The algorithm systematically explores possible moves, pruning branches that lead to dead ends. Symmetry optimization significantly reduces the number of redundant states explored.

C. 3. Solution Path Length

We recorded the length of the solution path (number of moves) for each solved puzzle. The path length varies based on the initial configuration and the specific moves chosen.

D. 4. Optimization Impact

We compared the solver's performance with and without symmetry optimization. Symmetry optimization significantly improves efficiency, especially for larger boards.



```
|        |   Row     |  column    |  Direction
------------------------------------------------
     1-  |Moved row 1| column 3 |        down
     2-  |Moved row 2| column 1 |       right
     3-  |Moved row 0| column 2 |        down
     4-  |Moved row 0| column 4 |        left
     5-  |Moved row 2| column 3 |        left
     6-  |Moved row 2| column 0 |       right
     7-  |Moved row 2| column 4 |          up
     8-  |Moved row 2| column 6 |        left
     9-  |Moved row 3| column 2 |          up
    10-  |Moved row 0| column 2 |        down
    11-  |Moved row 3| column 0 |       right
    12-  |Moved row 3| column 2 |          up
    13-  |Moved row 3| column 4 |        left
    14-  |Moved row 3| column 6 |        left
    15-  |Moved row 3| column 4 |          up
    16-  |Moved row 0| column 4 |        down
    17-  |Moved row 4| column 2 |          up
    18-  |Moved row 1| column 2 |        down
    19-  |Moved row 4| column 0 |       right
    20-  |Moved row 4| column 3 |        left
    21-  |Moved row 5| column 4 |          up
    22-  |Moved row 4| column 6 |        left
    23-  |Moved row 6| column 2 |          up
    24-  |Moved row 3| column 2 |        down
    25-  |Moved row 6| column 4 |        left
    26-  |Moved row 6| column 2 |          up
    27-  |Moved row 4| column 1 |       right
    28-  |Moved row 4| column 3 |       right
    29-  |Moved row 2| column 4 |        down
    30-  |Moved row 4| column 5 |        left
    31-  |Moved row 5| column 3 |          up
```

```
Starting board

    0  1  2  3  4  5  6
   ____________________
0 | -  -  0  0  0  -  - |
1 | -  -  0  0  0  -  - |
2 | 0  0  0  0  0  0  0 |
3 | 0  0  0     0  0  0 |
4 | 0  0  0  0  0  0  0 |
5 | -  -  0  0  0  -  - |
6 | -  -  0  0  0  -  - |
   ____________________
```

```
Moved row 5 column 3 up
    0  1  2  3  4  5  6
   ____________________
0 | -  -              -  - |
1 | -  -              -  - |
2 |                        |
3 |           0            |
4 |                        |
5 | -  -              -  - |
6 | -  -              -  - |
   ____________________
```

Fig. 5. results

IX. Conclusion

The experimental results demonstrate that our solver efficiently finds solutions for various board configurations. While simple puzzles are quickly solved, more complex setups may require additional time. Overall, the backtracking approach provides a systematic and effective strategy for solving Peg Solitaire puzzles

Peg Solitaire,Backtracking Approach, C++.

***Index Terms*—Peg Solitaire, Backtracking, C++.**

References

[1] Beasley, D. (2003). The history of peg solitaire. Retrieved from http://www2.stetson.edu/~efriedma/papers/pegsolitaire/pegsolitaire.html

[2] Knuth, D. E. (2000). Dancing links. *Millennial Perspectives in Computer Science*, 187.

[3] Berger, R. (1992). The game of Hex: A new approach to a classic game. *The Journal of Recreational Mathematics*, 24(3), 134-137.

[4] Harder, D. W. (n.d.). Peg solitaire. *Algorithms and Data Structures*. Retrieved from https://ece.uwaterloo.ca/~dwharder/aads/Algorithms/Backtracking/Peg_solitaire/

[5] Júlio Reis - Wikimedia. Retrieved from https://commons.wikimedia.org/wiki/File:Peg_Solitaire_game_board_shapes.svg

[6] Beasley, D. (2003). The history of peg solitaire. Retrieved from http://www2.stetson.edu/~efriedma/papers/pegsolitaire/pegsolitaire.html

[7] Knuth, D. E. (2000). Dancing links. *Millennial Perspectives in Computer Science*, 187.

[8] Berger, R. (1992). The game of Hex: A new approach to a classic game. *The Journal of Recreational Mathematics*, 24(3), 134-137.

[9] Harder, D. W. (n.d.). Peg solitaire. *Algorithms and Data Structures*. Retrieved from https://ece.uwaterloo.ca/~dwharder/aads/Algorithms/Backtracking/Peg_solitaire/

[10] Júlio Reis - Wikimedia. Retrieved from https://commons.wikimedia.org/wiki/File:Peg_Solitaire_game_board_shapes.svg