Solving Pentomino Puzzles Using Backtracking

members: Liz Demin

Contents:

I: Original Goal

"For my project, I want to create a program that will solve various pentomino puzzles by using recursive backtracking to fill a given board… It will start by putting one pentomino in a certain position which will create a "new" board. Then it will attempt to place a different pentomino on the "new" board, and so on recursively. The program will attempt to find all (or none, if none exist) solutions for the given board."

II: Developed Algorithm

Description of algorithm (from poster):

The algorithm, done by the recursive function solve(), consists of 6 nested for loops.

The first for loop iterates through all pieces. The second for loop iterates through all permutations for the given piece.

The 3rd and 4th for loops iterate through the board's y and x coordinates, respectively.

Next, a secondary function placePiece() is called and passed the current board y and x coordinates, and the current piece, permutation, and board. placePiece() uses two for loops that iterate through the current piece's y and x coordinates and overlay the board 2D array with the piece's 2D array. If the overlay is successful, the piece is placed, and if not, the function returns.

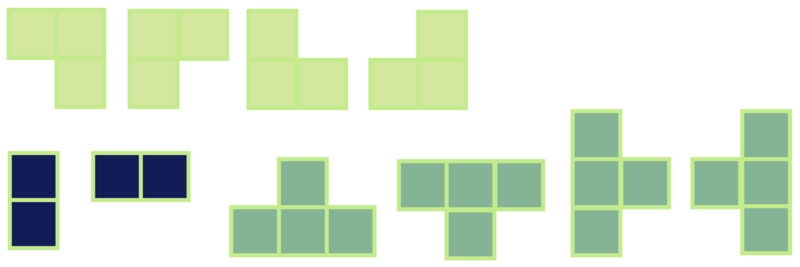
With these iterations, all possibilities are tried, and when all 12 pentominoes can be placed on the board successfully, a solution is recorded.

Pseudocode (from poster):

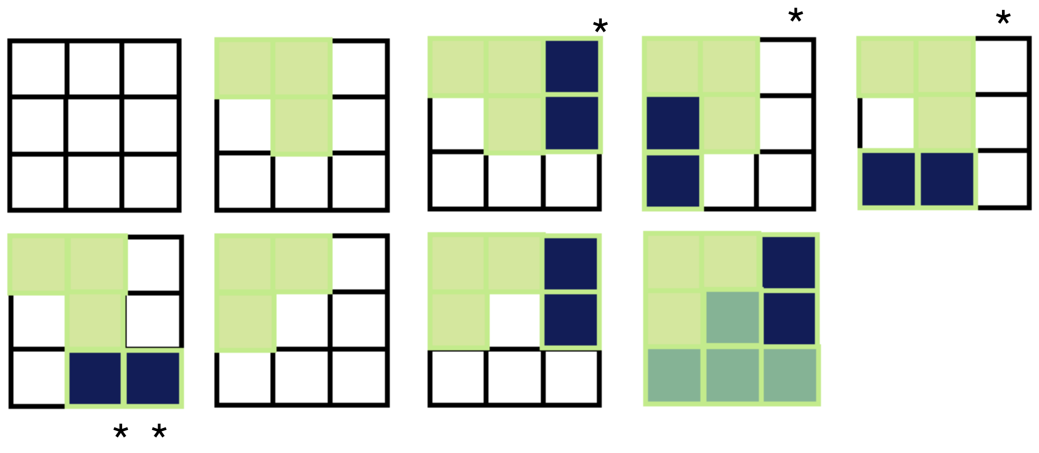
```
solve(board, pieces):
    for each piece in pieces:
        for each permutation for the piece:
               for each y coordinate of the board:
                    for each x coordinate of the board:
                         placepiece(boardy, boardx,
                                   piecenum, permutation, board)
                         if piece has been placed:
                              remove placed piece from pieces
                              if pieces is empty:
                                   record solution
                              else solve(updated board,
                                         updated pieces)

placepiece(boardy, boardx, piecenum, permutation, board):
     for each y coordinate of the piece:
          for each x coordinate of the piece:
               if the current square of the piece is filled:
                    y = board y + piece y
                    x = board x + piece x
                    if the piece goes out of bounds:
                         return false
                    if board[x][y] is not empty:
                         return false
                    board[x][y] = piecenum
     return true
```

Example of small-scale execution of the algorithm (from poster):

3 pieces and their permutations:



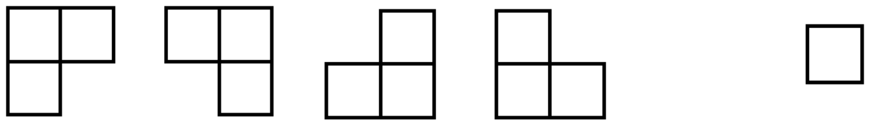
piece placement (* represents backtracking):



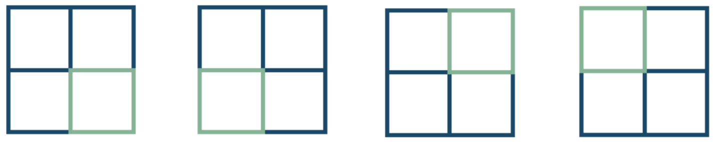
III: Development Process

The first thing I focused on was correctly coding my algorithm so it would work with a small board and a small amount of pieces. This was the longest stage of the project process and involved many sequential stages of fixing errors. I developed my algorithm by hand first and traced it out completely on a small board to ensure that it was correct. Then I attempted to code the algorithm with two different 2x2 pieces and a 2x2 board.

First piece (with 4 permutations) and second piece (with 1 permutation):



With this setup, I performed my debugging until my output matched the 4 solutions:



After that, I attempted to use my algorithm on bigger boards with more pieces until all bugs were fixed and I was satisfied with the algorithm implementation.

Throughout the debugging process, I quickly realized that I was too idealistic in my original goal of filling a board with all 12 pentominoes in all possible ways. The maximum depth of the recursion wasn't the problem, but the runtime was. The runtime increased too quickly and I realized that, without optimization, my program would not be able to find even one solution to a board that large (in a reasonable amount of time) simply because it would take too long. Because of this, my final product is a modification from my original goal, which is explained in section IV.

IV: Results

Since I realized that my program would not be able to find solutions to the traditional pentomino problems, I wanted to find a way to showcase my backtracking algorithm on pieces and boards that could actually be solved in a reasonable amount of time. I decided to try solving smaller boards using subsets of the 12 pentominoes so the exhaustive backtracking would be manageable.

I created a second program, PentominoPuzzleSolverLite, that comes hardcoded with 3 different boards of increasing size with an increasing amount of pieces. Board 1 is 3x5 and is filled by 3 pieces, board 2 is 4x5 and is filled by 4 pieces, and board 3 is 5x6 and filled by 6 pieces. The program can successfully find all solutions for board 1 and board 2 (3 and 4 solutions, respectively) in under a minute, but finding 4 solutions for board 3 takes over 10 minutes.

After the solutions for the given board are found, they are printed along with some useful outputs: the number of solutions, the solutions themselves, the time in milliseconds it took to complete the calls, and the total number of recursive calls. The experimental results are discussed in detail in section VI.

The original program, PentominoPuzzleSolver, is functional as well. The user can enter board dimensions and the program will attempt to find solutions. All 12 pentominoes are hardcoded into the program. In the final version, I have the algorithm stop as soon as it finds one solution (in hopes that it would successfully find a first solution), but as of writing this report, I have not run it for long enough.

My overall goal for this project was to write a recursive backtracking algorithm that would exhaustively find all possible variations of placing pieces on a board. Even with PentominoPuzzleSolverLite, I accomplished my goal. With future optimizations, I can make it so the traditional pentomino puzzles can be solved more efficiently.

V: Description of Project Components

PentominoPuzzleSolverLite.java — components as follows:

class variables:

- int[][] board — the board to fill
- boardYdim and boardXdim
- pieceYdim and pieceXdim, which are both 5 because all Pentominoes are 5x5
- ArrayList<int[][]> solutions — to hold all found solutions
- HashMap<Integer, ArrayList<int[][]>> pieces — maps the piece number to the list of permutations for that piece
- int numCalls — a counter that keeps track of the number of times the recursive function is called

main():

- prints introduction
- prompts user to pick board 1, 2, or 3
- sets up the board and pieces accordingly by calling board1setup(), board2setup(), or board3setup()
- calls the solve function while keeping track of the start and stop time
- prints the solutions and useful information about the execution

board1setup(), board2setup(), board3setup():

- these three functions set up the board and pieces depending on which board number is picked by the user
- all return a list of the usable piece numbers
- note: an explanation of the piece numbering convention can be found in README.txt

solve(int[][] board, int[] usable pieces):

- this recursive function performs the algorithm as described in section II
- it is exhaustive and tries all possible combinations of placing each piece

- iterates through pieces, then permutations, the y and x coordinates of the board
- it has a call to the function placePiece()
- does not return anything

placePiece(int boardy, int boardx, int currentPiece, int[][] currentPermutation, int[][] currentBoard):

- works as part of the recursive function as described in section 2
- iterates through the y and x coordinates of the given piece and looks at the given coordinate of the given board
- returns false if piece cannot be placed and true if piece is placed

doesSolutionExist(int[][] sol):

- iterates through the list of existing solutions to determine if the given solution already exists in the list
- iterates through the y and x coordinates and compares them, looking for a difference
- returns true if it finds a match, returns false if not

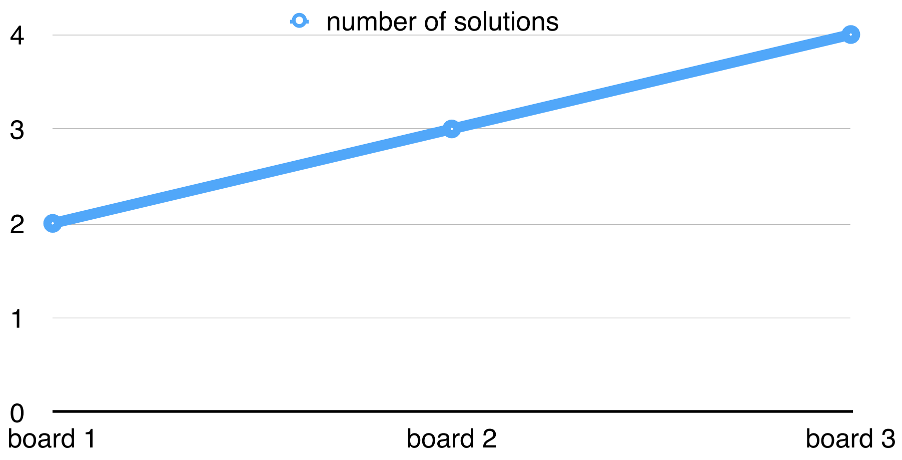
print2DArray(int[][] myArray):

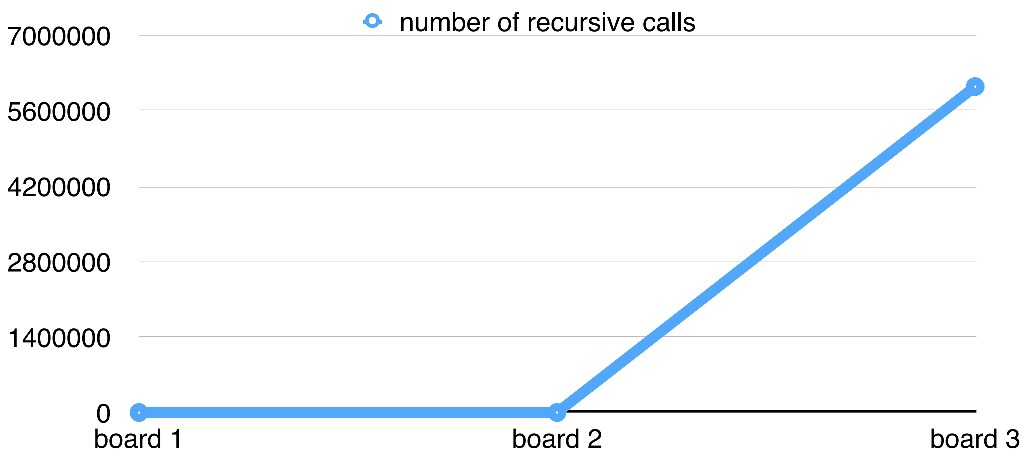
- prints the given 2D array with basic formatting for readability

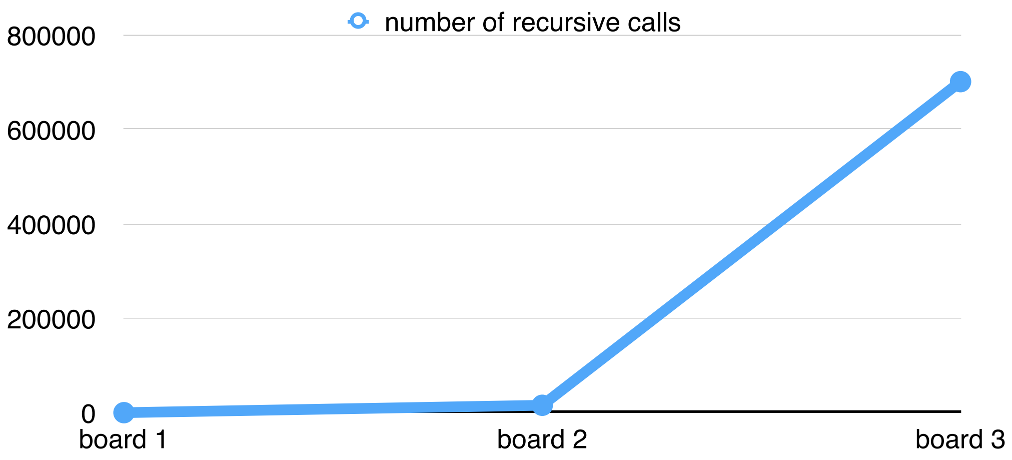
PentominoPuzzleSolver.java — the original program as described in the section above with all of the same components as PentominoPuzzleSolverLite.java.

VI: Experimental Results and Analysis

	board 1	board 2	board 3
number of pieces	3	4	6
board dimensions	3x5	4x5	5x6
number of solutions	3	4	4
number of recursive calls	57	1240	6048009
execution time in ms	593	16237	701711


number of solutions
4
3
2
1
0
board 1
board 2
board 3


number of recursive calls
7000000
5600000
4200000
2800000
1400000
0
board 1
board 2
board 3


number of recursive calls
800000
600000
400000
200000
0
board 1
board 2
board 3

The complexity of the algorithm depends on the number of pieces, size of the pieces, number of permutations for each piece, and the size of the board. The summation of the area of the pieces must equal the area of the board in this implementation. So, as the number and size of pieces increases, the size of the board must also increase, and the runtime increases as well. The number of recursive calls grows more quickly than the execution time. The number of solutions shows a positive relationship as well.

The solve() function has 4 for loops and the placePiece() function has 2 for loops. The doesSolutionExist() function has 3 for loops. There are also a few other for loops to copy arrays inside the solve() function. I suspect that the complexity of my algorithm is something like O(n^x+n^z) where x and z are some constants. The complexity is much worse than I feared; as mentioned previously, the usability of the program would increase by a lot with some optimization.

If this program is was used to solve actual pentomino puzzles, the area of the board and the number of pieces and their permutations would be constant (equal to 60, since each pentomino is 5 squares big and there are 12 possible pentominoes). The only thing that would change would be the dimensions of the board. In that case, I think the runtime would be approximately the same for all boards.

The complexity of this algorithm would be much more interesting to analyze on pieces with increasing size and number of permutations, like I did in PentominoPuzzleSolverLite. As is, the algorithm can be used with any board dimensions and any pieces as long as the pieces are coded in. Pentomino puzzles are well known, but there are other board cover problems that can be solved with backtracking.

VII: Possible Optimizations

1. Make the solve function return() as soon as a solution is found

   Having it return at this point would eliminate some extra iterations through y and x (since they would be unnecessary anyway).

2. Have the board be processed for “holes”

   It would be useful if the program could iterate through the board and find “holes” between the pieces that have already been placed. By holes, I mean empty squares that can’t be filled with another piece because they are too small or located near an edge. That way, the program can backtrack immediately instead of continuing to place pieces until it gets to the deepest level of recursion.

3. Multithreading

   The efficiency would increase if several “deep” paths could be run at the same time. That way, instead of finding solutions sequentially through iteration, multiple paths of recursion could look for solutions simultaneously.

4. Eliminate some permutations

   The algorithm generates a lot of solutions that are mirror images or flips of each other because the permutations of each piece are mirror images or flips. Getting rid of some of the permutations would get rid of those “duplicate” solutions.

5. Find an alternate data structure to represent the matrices

The program is inefficient because it uses a large amount of for loops to cycle through the 2D arrays.

6. Have the program place and remove pieces instead of copying the board and the pieces every time

Since Java passes by reference, a new board must be created before calling placePiece() so the previous board does not get ruined by a piece that cannot be placed. After it is called and a piece is placed, the list of available pieces must be copied to reflect that one piece has been "removed". Having a function to remove pieces would eliminate the need for a few for loops.

VIII: Future Plans

I am happy that I managed to implement a recursive backtracking algorithm properly. However, since my original goal was to make a pentomino puzzle solver, I would like to optimize my program so that it can actually find solutions for bigger boards with more pieces. I also want to store pieces in a text file instead of hardcoded as matrices — this would allow the user to input any pieces and use them to fill any board. Alternatively, the user could enter board dimensions and pick which pentominoes should be used to fill it. Ultimately, I would get rid of everything that's hard-coded into the program and allow the user to enter all parameters.

There are some pentomino puzzles that include holes in their board. I would like my program to be able to solve boards like this as well. It is also possible to create shapes with pentominoes (other than rectangles) so I would like to modify my program to be able to input and solve shaped puzzles as well as rectangular ones.

The project that I am passing in is certainly not bullet-proofed. I did not have time to add in all error handling for incorrect inputs, so I want to go back and ensure that the program will only attempt to solve when all inputs are correct.

In the end, I would like to make this a graphical program. Solving pentomino manually is a very visual process so I think it would be beneficial to have a better visual interface for my program. I would want a better way of representing the board and perhaps even animation of the pieces being placed on the board.

Overall, this project was fun to code and taught me a lot about debugging. After a great deal of optimization, I think this program will be functional and easy to use as a puzzle solver. I plan on implementing all of the comments above and perhaps making it web-friendly so other people can see and use it. While writing this program, I learned that even a relatively simple algorithm can be hard to implement and optimize.