Cracker Barrel Peg Game - Final Report:

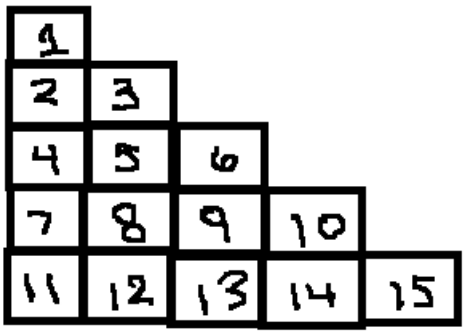
Tyler Laudenslager – CSC 447 Artificial intelligence 1 – Fall 2021

Task Definition: The real-world problem the system is designed to solve is the Cracker Barrel peg game. The measurement of success for this system is if I can produce a list of moves that if followed directly will lead to having only one peg left on the board.

However, the system I built takes an arbitrary number to denote the size of the board that you would like to obtain the move list for. In order words the system I built is not limited to just the small Cracker Barrel peg board size with a length of five. It is built to solve for any arbitrary triangle board size. (Note: I encountered a major difficulty in getting the program to work for a triangle of side length of 6+. This is because for some reason the memory explodes.)

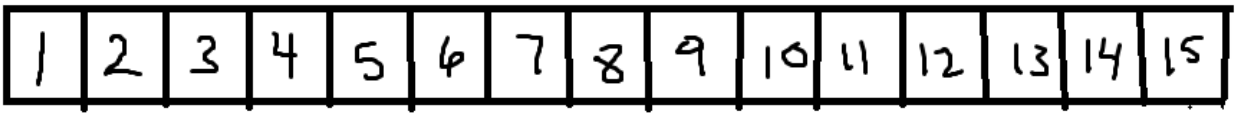
Infrastructure: I think this is the most interesting part of the entire project though. I developed a way to represent a triangle data structure in a one-dimensional array using a two-dimensional abstraction. The following paragraphs will explain what I designed.

The cracker barrel peg game is a triangle board. I decided to represent the triangle as leaning to the left such that the top of the triangle has one item, and the bottom has the number of items defined by the size of the board. Accordingly, if we have the normal peg game of size five the triangle would result in the following representation.



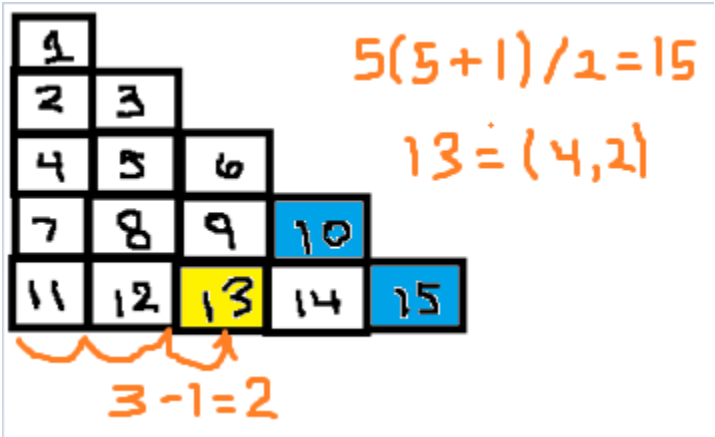
When I originally developed this program, I used a data structure such that I defined multiple arrays to hold the corresponding elements. This was not an ideal way to represent this structure because for the size of the triangle I would have to make at least size number of arrays to hold the structure. However, I figured out a way to have the convenience of using this two-dimensional data structure representation but store the data in a one-dimensional data structure underneath.

The one-dimensional data structure is represented as you would imagine.



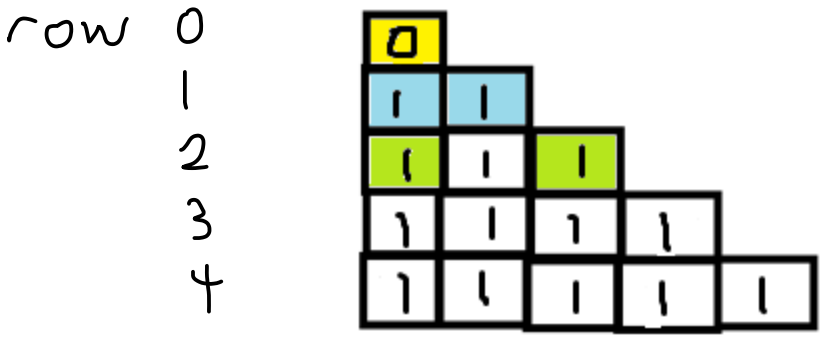
I wrote the initial algorithm for finding and applying moves to each board using the two-dimensional data structure, however due to the amount of memory I was using I needed to try to use the least amount of memory possible. So, naturally I created a number mapping in which I could use the algorithm I developed for the two-dimensional data structure but store the data in a one-dimensional structure instead.

The benefit of creating this number mapping is that I can use existing code I have already developed while at the same time saving a lot of memory. The number mapping algorithm works in this way. I noticed first that the size of the side of the triangle corresponded to the number of items in the triangle such that a triangle of size 5 would have 15 elements. This is because 5+4+3+2+1 = 15. Then I noticed that the number 10 was on the 4th row so naturally 4+3+2+1 = 10. I eventually realized this was referred to as triangular numbers and the algorithm to generate these numbers was proven in discrete math class, but I had no idea it works perfectly for generating triangles. I found more information here at this site https://www.mathsisfun.com/algebra/triangular-numbers.html. This means that for every row I can identify the right number where each row stops. This allowed me to turn a position on the one-dimensional array into a pseudo position on the two-dimensional structure. For example, suppose we were trying to find the two-dimensional position for item 13 in the structure above. I did this by identifying which two right numbers on the triangle I was between in which I would use the minimum of those numbers to subtract from the position I wanted to transform as well as subtracting one. This made me sure I had the right position to account for starting each row at zero. This allowed me to obtain the column of the position which would be 2 because position 13 is between 10 and 15 such that when I subtract 13 by 10 as well as subtract by 1 to account for the row position being at zero, I obtain the value of 2. The picture below explains the method more clearly.



Now for finding the row position. I used a range between (1...5) for board size 5. Once, I found the number I used the number that corresponds to the outcome of the function x(x+1)/2. Such that 5(5+1)/2 = 15. Then the corresponding row is number would be 4 because we must account for the fact that the first row is row 0. Therefore, position number 13 in the one-dimensional array corresponds to position (4,2) in the two- dimensional pseudo structure. Now I can save memory while also taking advantage of the algorithm I already was using to figure out the possible moves and perform an action on each board.

The algorithm for finding all possible moves was interesting I felt because it was a lot more involved than I thought it would be initially. The problem I faced was making sure the valid moves I returned stayed inside the valid positions in the structure I was using. This is a common issue when dealing with two dimensional structures as an abstraction or not. The first idea I came up with after designing how I would be able to find all possible moves from a specific board state is that I simply need to keep track of all the open positions on the board and start from there. In order words if the one position on the two-dimensional structure is does not have a peg I should simply go two rows down and check every other peg to see if there is a valid move available. This in contrast to searching the entire board for every single possible combination each time. I ended up going with this method because I thought it was reasonable that you would only be concerned with the open positions on the board and the relative pegs around that position than to be concerned with the entire board.



I ended up realizing that for any row less than 3 you cannot make any moves in the up, left, or right direction. However, once you get to row 3 you have one possible move up and one move right for position 4 on the structure. For any row greater than 2 (with the top of the triangle being row 0) I realized that I cannot make any move to the left when the column position is less than 2. This is because there are simply not enough positions to make a valid move. I realized that I would have trouble going to the right as well. This is because at row 2 we can make a move left from the end of the row and make a move from the beginning of the row if the conditions on the board are correct. However, the number of possible moves increases proportional to the current number row we are at. Thus, the constraint is simply if the row is greater or equal to two and the column is less than or equal to the row – 2. For example, if there is a zero-position located at 8 on the board above we know that the row position is greater than 2 as well as the column which is less than or equal to the row – 2. Such that if 8 is

denoted (3,1) than 3 > 2 and 3-2 >= 1. Therefore, we can use this constraint to tell us if we are in the boundaries of the data structure.

Once I figured out the constraints on the board as explained above, I would check the relative values around the open position and if the middle value had a peg, I would add the move to the move list and check the next move available and see if I should add that move as well.

Approach: One of the biggest challenges in building the system was creating the infrastructure. I never had experience representing a triangular data structure in memory before and I was not sure what the best way to do it was at first. This is because I ended up creating the first implementation such that when I tried to expand the board from size 5 to size 6 my computer OS killed the process. I believe this problem occurs simply because I am allocating to much data on the heap, and I do not know the Rust programming language proficiently enough to have any idea how I could solve it. I ended up having to rewrite the data structure to try to use less memory. However, even with having less overhead because I was using a one-dimensional array instead of multiple arrays to hold the data. I still could not get a triangle of size 6 to print out the move list or tell me that there is no solution. This was slightly discouraging.

The approach I choose to use was a simple searching algorithm in which I would find all the moves that were legal to be made from each board and do all the moves until I came upon a terminal case in which I would then print the move list I kept track of along the way. This works excellent for a triangle of size 5 I obtain the move list almost instantly with the optimized Rust program. The problem with this approach however is that we need to store every single board that can be generated in memory until we find the solution move list. This is because I am using a breadth first search checking every option until there is a move list solution, or we eliminate all possible moves and do not reach a terminal state. I chose this because I cannot have an inaccurate solution because I would like to win the game.

There is one implementation choice specific to the problem. This is that I do not have to keep a reference to the board states I have already encountered because each time an action is taken a peg becomes removed.

Literature Review:

I ended up doing the literature review after I implemented my version of the problem in Rust simply because some programming code on the internet is terrible and can give you bad ideas. I did not want to incorporate ideas that involved hardcoded values such as in this example here written in python. https://www.jakepusateri.com/blog/peg-game-python/. Now this python implementation is not terrible, but it does not scale well. Meaning that it only works for a triangle of size 5 instead of being implemented for any size triangle. This work provides additional insight though that I did not consider when making my system. The program above also identifies how many solutions exist for each starting position indicating that starting with a peg removed from the middle is very hard with only 1550 solutions identified. The position that has a lot of solutions identified would be to have the peg be removed from either position 2 or

3 in my system. I would say in some respects my system would be contradictory to the solution presented on the website link above. The solution shown above does not algorithmically define the moves available for each board state and does not dive deeper into the constraints needed to assess which moves are valid for a larger board state. There is one aspect of this program that I did not think about and that is using recursion to make a move and using a form of backtracking to test all possible avenues.

The next one I found is here because it's the next on the list https://codereview.stackexchange.com/questions/88629/solution-to-the-cracker-barrel-peg-game-triangle-game. This implementation also has hardcoded values for the program implementation with each slot value on the peg board associated with the middle value plus the end value. I would say this is also a contradictory implementation to the system I wanted to produce because it does not scale for larger board sizes.

The one interesting project I found is located here https://github.com/Techman/puzzle-pegs. This is simply because of the way the person decided to try to find the possible moves available using a searching pattern. However, this program is still hardcoded for a board size of 5. Which is still contradictory to the purpose of the system I created which is implemented to be able to solve the peg game using any arbitrary size of board.

The next relevant link is found here https://www.daniweb.com/programming/software-development/threads/403489/triangle-peg-solitaire-solver. This is complementary to the system I was developing however the author of the program did not seem to get it to work. The reason this is complementary to my system I was developing is the person wanted to create a system that would solve for an arbitrary size board not just the default size 5 board.