Data Structures and algorithms

The 8 queens’ puzzle

Ross Jones

The 8 queens puzzle is where 8 queens have to be placed on an 8x8 chess board so that none can take each other.

I implemented an algorithm which solves the task by using a stack to go across all columns (apart from the player’s placed queen’s column) and place a queen in each. It checks to see if a row is already taken by another queen before placing a queen. If a queen cannot be placed in the column, due to the space search reaching the end of the board, then the column is popped off the stack so that the previous column’s queen can be moved. Once the last queen is placed then a solution has been found, so it is outputted to the console. The last column is then reused during the next loop to move its queen. This algorithm runs on the rule: once the first queen is moved off of the board then all solutions have been found. The diagonal directions a queen can take are initially checked by recursive functions to see if a queen is already occupying the current diagonal.

I chose to use a stack because it’s a LIFO (last in first out) data structure, which fits well with trying to backtrack across the columns. The stack will only ever be 7 items long at max, so it doesn’t take up much data. I could have implemented this with a tree, but that would also need a stack to search it anyway; so I decided to shorten it.

Also, I used a dictionary to reference to a queen’s location. The keys were set to the columns and the values to the row. This turned out to work very smoothly in sync with the stack, due to columns already being saved. The algorithm works by working on 1 column at a time, so it can use the ColInUse variable to reference to the dictionary to find the queen’s row. If a queen hasn’t been placed already, then their row value in the dictionary is 9 (default/null). This tells the algorithm that there isn’t a queen in this column, so it has to find the next available row to start searching for a space. If a queen is already on the column then it is removed from the board and the row is found in the dictionary. This row is then removed from the dictionary and is used to find a new space for this column’s queen. This dictionary implements dynamic programming as it cuts out having to look for the column’s queen each time the column is checked.

The way the algorithm checks if a row is taken or not is by saving a row to an array once it is taken. This array is then checked by the algorithm when a new queen is being placed to see if the current row is already taken. If the current row is taken then the algorithm moves onto the next row, otherwise the row is added to the array and the queen is placed. This also implements dynamic programming due to cutting out the need to search entire rows with another sub-algorithm. The array is searched with a linear search because the array will never be bigger than 8 items long (there are only 8 rows on the board). This search is faster than a binary search in this instance, due to having to sort the array for a binary search.

The way OOP is implemented is by adding a class called QueenSorter, which is filled with methods which would be used multiple times throughout the program. This saves space and time in the algorithm.

For searching diagonally, I chose to implement recursive functions. This is due to the function’s ability to return a value compared to a simple for loop. A for loop would also be very difficult to implement in this instance, since the diagonals can start anywhere on the board. To keep the function simple, I implemented 4 recursive methods for each diagonal direction (SE,SW,NW,NE). Each method call inspects a single square on the board and then the next method call inspects the next square in the diagonal respectively. If a queen is found in one of the squares then the method will return false, which flows down the rest of the method calls until it reaches the initial method call. If the method detects that it is off the board, then it returns true; which flows down the calls as well. This is respective to each direction. The algorithm calls each direction respectively when checking if a space is free for a queen. If all return true, then the space is free.