Lab 6 – Backtracking

Objectives:

In this lab, you will solve the *n*-queens chess problem using recursion. A chessboard contains 64 squares that form eight rows and eight columns. The most powerful piece in chess is the queen because it can attack any other piece in its row, in its column, or along both diagonals. The 8-queens problem asks you to place eight queens on the chessboard so that no queen can attack any other queen. The *n*-queens problem is a generalization of this problem: placing *n* queens on an *n* × *n* chessboard. We will start by solving the 8-queens problem and then move on to the general version, but first an overview.

Problem 1 (of 2): Solving the 8-Queens Problem

Part 1.1: Understanding the solution

One strategy to solving the $\boldsymbol{n}$-queens problem is to guess at a solution. However, there are $\binom{\boldsymbol{n^2}}{\boldsymbol{n}} = \frac{\boldsymbol{n^2!}}{\boldsymbol{n!(n^2-n)!}}$ possible ways to places $\boldsymbol{n}$ queens on an $\boldsymbol{n\ x\ n}$ chessboard. For eight queens, this is $\binom{\boldsymbol{64}}{\boldsymbol{8}} = \boldsymbol{4,426,165,368}$ ways to arrange eight queens on a chessboard of 64 squares, so it will take a *long* while for you to check them all. Try it if you like, but not during today's lab please!

We can make the problem a bit simpler by realizing that no two queens can be in the same column or the same row. This observation greatly reduces the number of possible placements to $\boldsymbol{n}$ factorial – $\boldsymbol{n}!$ – in general, or $\boldsymbol{8}! = \boldsymbol{40,320}$ for eight queens. The $\boldsymbol{n}!$ comes from the fact that there are $\boldsymbol{n}$ possible placements for a queen in the first column, leaving only $\boldsymbol{n-1}$ possible queen placements in the second column (excluding the row where the queen has been placed in the first column), $\boldsymbol{n-2}$ possible placements in the third column, *etc*. Thus, the total would be $\boldsymbol{n} \times (\boldsymbol{n-1}) \times (\boldsymbol{n-2}) \times \ldots \times \boldsymbol{1} = \boldsymbol{n}!$. This is still a lot of possibilities to check, but we can reduce this a bit further by carefully choosing our problem-solving strategy.

Our strategy uses a technique called *backtracking*. We can tentatively place a queen in the first row of the first column (row=0, col=0) and then look to place a queen in the second column. We can try the first row in the second column (row=0, col=1) but find that the first queen can attack it horizontally; next, we try the second row (row=1, col=1), but discover that the first queen can attacked it along the diagonal; finally, we try the third row (row=2, col=1) and find it to be safe. Similarly, we can place a queen in the third column, where the fifth row is the first safe position; then in the fourth column, where the second row is the first safe position; and in the fifth column, we can choose the fourth row. This can be seen in the board on the right where a **Q** marks a placed queen and an **X** marks an illegal queen placement (i.e., one that would result in an attack from another queen) given the placement of queens in previous columns.

	0	1	2	3	4	5	6	7
0	Q	X	X	X	X	X		
1		X	X	Q	X	X		
2		Q	X		X	X		
3			X		Q	X		
4			Q			X		
5						X		
6						X		
7						X		

Attempting to place a queen in the sixth column fails because all of the squares (or rows) in that column are under attack. We know that something must change with our earlier placements, so we try moving the queen in the fifth row. When we realize that no placement of the fifth queen leads to success, we backtrack again and try a new placement for the queen in the fourth row. We continue trying placements and backtracking until we either successfully place all eight queens or run out of rows.

It is possible to write a solution using this strategy with loops and a bunch of housekeeping arrays, but the code is quite cumbersome. Instead, we will use recursion to arrive at a simple elegant solution.

Start by tracing the solution to the simpler 6-queens problem (limit the board to 6 x 6) by hand using pen and paper (or whiteboard). Like before, if you cannot do this by hand then you SHOULD NOT try to program it. Show your solution to the instructor or TA before you proceed.

Part 1.2: Implementing the solution

Related class examples that you'll find handy: Solution to ***Maze.zip***.

To visualize the logic discussed here, please visit the ***Lab06_Backtracking*** visualization *https://faculty.csbsju.edu/irahal/SW/DataStructuresVisualizations/Algorithms.html*. Study this visualization carefully, starting with the 4-queens, followed by the 6-queens. Slow down the animation speed and use the *Step Back*/*Forward* buttons (after you pause). Observe how recursive method ********placeQueens******** is called to place a new queen in the current column for which it checks all rows in the current column until it finds a spot not under attack. The latter check is accomplished by calling method ********isUnderAttack********.

Once you feel comfortable with all the details of the solution presented in the visualization, take plenty of time to study the two supplied classes ***TwoDimGrid*** and ***EightQueens*** and discuss them with your partner. Class ***TwoDimGrid*** is very similar to the one we used in the ***Maze*** problem from class. ***EightQueens*** is a graphical program that uses Java Swing components, with all the needed graphics fully functional; you will not need to write or modify any code that directly accesses the graphics.

Once you are ready, proceed to complete the following two methods in the ***EightQueens.java*** file; the other methods are all written for you and should not be modified in this part.

`private boolean isUnderAttack(int row,int col)`

This method checks that there are no queens already placed on the same row or diagonal in previous columns. You MUST solve this using a single loop like below, where `offset` is the number of columns before the current column: `for (int offset=1; offset<=col; offset++){ ... }`
The body of the loop can check for a queen in the same row and in either of the two diagonals. If a conflicting queen is found, break the loop, and return `true`; otherwise, return `false`.
Note that the diagonal row could be negative or past the last row; if you try to check for a queen in those rows, you will likely get an `ArrayIndexOutOfBoundsException`, so you will need to check for those special cases. You may have noticed that the logic described here is (slightly) different from the visualization. We will revisit this point later in this lab.
Make sure to call `getBoardAt(row,col)` to access current value at location `board[row][col]`.

`public boolean placeQueens(int currentColumn)`

The logic for this method is similar to method `findMazePath` in the maze backtracking problem from class but with fewer base cases since we

(1) don't need to worry about going off the board (grid of cells), and
(2) don't use cell colors to make decisions (but we still color any cell containing a queen).

It is initially called with a parameter value of zero and repeats until every column contains a queen that is not under attack by previously placed queens. To place a queen in `currentColumn`, the method assumes all previous columns contain queens that cannot attack one another and *loops* over all rows in `currentColumn` to find the first row for a queen placement that isn't under attack (meaning that `isUnderAttack` for that cell is FALSE)–a bit similar to how `findMazePath` searches for a LEGAL-colored cell to move to.

(1) If such a placement is possible, a queen is temporarily placed there after which the method attempts to ***recursively*** find a suitable queen placement in the next column. The method returns true if it succeeds in its attempt–similar to how `findMazePath` colors the current cell green and attempts to move to one of the neighbors.
(2) Otherwise, the queen is removed to allow us to check the remaining rows in `currentColumn` for a different possible queen placement. False is eventually returned if no possible placements are found in

the `currentColumn` indicating the need to backtrack to the previous column–similar to a dead-end scenario in `findMazePath`.

Note class ***EightQueens*** contains complete methods `setQueen` and `removeQueen` to help you place and remove a queen from the board easily. Please use them since they take care of handling the graphics along with other details.

When your program works properly, change the start place in `main` method's call to `q.placeQueens(0)` to a different starting column and see what happens. When you have finished implementing and testing your work, demo it to the lab instructor or the TA.

Part 1.3: Improving method `isUnderAttack`

Let us revisit method `isUnderAttack`. If you observe the visualization closely, you will notice that when checking if a newly placed queen is under attack, the location of the new queen is compared directly to the locations of the queens already on the board. In other words, the visualization does not seem to loop over the rows and diagonals to find a conflict like we did. Rather, it checks if the new queen conflicts with the first placed queen, then the second, etc. until it either finds a conflict or checks all placed queens.

To do this, we must store the location of every queen placed so far on the board. As a matter of fact, class ***EightQueens*** already has an array declared as an instance variable for this particular purpose: `private int queenLocations[]`. Every array location indicates a column, and the location value indicates the row placement for that queen (meaning that the row where a queen has been placed in column 0 is stored in array location 0, and queen in column 1 is stored in array location 1, etc.). The array values specify the row location for a queen in a particular column (for example, `queenLocations[1]=3` indicates that a queen has been placed in row 3, column 1).

Initially, array `queenLocations` values are initialized to -1 indicating that no queen has been placed yet. Method `setQueen(int row,int col)` updates `queenLocations[col]=row` and method `removeQueen(int row,int col)` resets `queenLocations[col]=-1`. In other words, we actually have complete information on queen placements so far readily available!

Now it is time to create an updated version for method `isUnderAttack` called `isUnderAttack`**`V2`** with the same parameters and return type. Do that and modify your `placeQueens` to call method `isUnderAttack`**`V2`** instead of `isUnderAttack`.

Method `isUnderAttack`**`V2`** should loop over array `queenLocations` to find if an earlier queen conflicts with the new queen at location `row` and `col`. As before, we may ignore checking the current `col` since we know we are only placing one queen per column. How do we know if a previous queen has been placed in the same `row`? What about the same diagonal? Checking diagonals is a bit tricky but note the following hint: take any two board cells in the same diagonal (consider the upper-left to lower-right diagonal first), do you notice a pattern when you compare the difference in the cells' row coordinates and column coordinates? What about cells in the lower-left to upper-right diagonal?

When you have finished implementing and testing your work, demo it to the lab instructor or the TA.

Problem 2 (of 2): Generalizing Solution to the N-Queens Problem

Part 2.1: Solving the N-Queens Problem

Create another copy of your complete ***EightQueens.java*** program from ***Problem 1*** and name it ***NQueens.java***. Make all the necessary changes to the program so that the user enters a grid size $\boldsymbol{n}$, and the program will then solve the problem for an $\boldsymbol{n\ x\ n}$ board instead of an $\boldsymbol{8\ x\ 8}$ board.

Part 2.2: Counting the Total Queen Placements on the Board

To compare performance between different algorithms solving the ***NQueens*** problem, it would make sense to compare the *total* number of queen placements made to arrive at the solution (including ones that eventually didn't work). This should require only three changes to your program

(1) Introduce a `public static int placementCounter=0` in your ***NQueens*** class
(2) Find the best place in your program to increment `placementCounter` indicating that a new temporary queen placement was made
(3) Display the value of `placementCounter` at the end of the `main` method

When you have completed your ***NQueens*** class, test it for a variety of board sizes. You should get 171 queen placements for the 6-Queens problem and 876 for the 8-Queens. Show your work to the lab instructor or the TA when you are ready.