**Lab 6 – Backtracking**

Work with your pair-programming partner.

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’ll start by solving the 8-queens problem and then move on to the general version, but first an overview.

***Problem 1: Solving the 8-queens problem***

***Part 1.1:***

One strategy to solving the *n*-queens problem is to guess at a solution. However, there are possible ways to put *n* queens on an *n* × *n* chessboard. For eight queens, this is 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 if we realize that no two queens can be in the same column or the same row, which also implies that there must be one queen in each column and row. This observation greatly reduces the number of possible placements: *n*! in general or 8! = 40,320, for eight queens. The *n*! comes from the fact that there are *n* possible placements for a queen in the first column, leaving only *n*-1 possible queen placements in the second column (excluding the row where the queen has been placed in the first column), *n*-2 possible placements in the third column, *etc*. Thus, the total would be. That’s still a lot of possibilities to check, but we can reduce this a bit further by carefully choosing our programming strategy.

Our strategy uses a technique called *backtracking*. We can tentatively place a queen in the first row of the first column and then look to place a queen in the second column. We can try the first row in the second column but find that the first queen can attack it horizontally; next, we try the second row, but discover that the first queen can attacked it along the diagonal; finally, we try the third row 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 if the first safe position; and in the fifth column, we can choose the fourth row. The board below demonstrates this process for eight queens, with a**Q** marking 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.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* |
| *1* | **Q** | **X** | **X** | **X** | **X** | **X** |  |  |
| *2* |  | **X** | **X** | **Q** | **X** | **X** |  |  |
| *3* |  | **Q** | **X** |  | **X** | **X** |  |  |
| *4* |  |  | **X** |  | **Q** | **X** |  |  |
| *5* |  |  | **Q** |  |  | **X** |  |  |
| *6* |  |  |  |  |  | **X** |  |  |
| *7* |  |  |  |  |  | **X** |  |  |
| *8* |  |  |  |  |  | **X** |  |  |

Now when we try to place a queen in the sixth column, we find that all of the squares come under attack. We know that something has to 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 backing up 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 problem by hand to figure out a solution for the simpler 6-queens problem (limit the board to 6 x 6) (***yes, draw the solution on a sheet of paper using a pencil so that you may erase frequently … alternatively, you may use the board or a spreadsheet like Excel***). Show your solution to the instructor or TA before you proceed.

***Part 1.2:***

Take some time to study the two supplied classes *TwoDimGrid* and *EightQueens*. Note that these are very similar to the *Blob* and *Maze* problems from class. *EightQueens* is a graphical program that uses Java Swing components but that all the graphics work is already in place; you won’t need to write or modify any code that directly accesses the graphics.

After you feel comfortable enough please complete following two methods in the *Queens.java* file; the other methods are all written for you.

* private boolean isUnderAttack(int column, int row): This methods checks that there are no queens already placed on the same row in previous columns. It also checks that neither diagonal in previous columns already contains a queen. You MUST use *one* loop here like below, where *offset* is the number of columns before the current *column*:

for (int offset = 1; offset <= column; offset++){

…

}

The body of the loop can check for a queen in the same row and in either of the two diagonals. If the code finds a conflicting queen, 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’ll get an *ArrayIndexOutOfBoundsException*, so you’ll need to check for those special cases.

* public boolean placeQueens(int currentColumn): The following pseudo-code describes the algorithm for placing queens in columns, given that the previous columns contain queens that cannot attack one another. It is called initially with a value of zero. This algorithm only captures the logic for the solution and may be missing some important details; thus, do not simply translate it to java as is.

Also note: class *EightQueens* contains the methods setQueenand removeQueen to help you place and remove a queen from the board easily. Please use them, since they take care of handling the graphics.

**algorithm** placeQueens(*currentColumn* : int) : Boolean

**if** you’ve successfully placed queens in all columns on the board **then**

**return** true

**end if**

**else** [*there are more columns to process; that is, currentColumn is still less than (or equal to?) n*]

**loop** **until** you find the first row in *currentColumn* where a queen can be placed safely [*i.e., is not under attack*]

**call** *setQueen* to place a queen there

**call** *placeQueens* recursively to attempt to place a queen in the next column [*even if that column is off the grid*]

**if** attempt succeeds **then**

return true

**else** [attempt has failed]

remove the queen placed earlier

**end if**

**end loop**

**return** false [*no feasible queen placement was found for* currentColumn]

When your program is working properly, change the start place in main method's call to q.placeQueens(0) to a different starting column and see what happens.

***Problem 2: Solving the N-queens problem***

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