ISP Mandatory Assignment 2

BDDs – N-Queens

Overview

This mandatory assignment is to implement an application that supports a user to solve an *n*-queens problem.

You should form groups of two to three students. You will be provided with a Java code support but implementations in other languages are also acceptable.

The provided software uses functions that are not accessible in Java versions older than Java 1.5 (also called 5.0).

General problem Description

In this task, you will implement an *interactive configurator* for specifying a valid solution to the ***n*-queens problem**. This is the problem of putting n chess queens on an $n \times n$ chessboard such that none of them is able to capture any other using the standard chess queen's moves. The color of the queens is irrelevant in this puzzle, and any queen is assumed to be able to attack any other. Thus, a solution requires that no two queens share the same row, column, or diagonal. For more information about the problem/puzzle, you can check Wikipedia.

Interactive configurators are applications that help users in selecting *valid options*, usually when buying a product. On average, it is hard for users to detect which options are valid given all the rules, and if the configurator doesn't give suggestions on what to choose, the user might end up in a dead-end during interaction – he selected some options so far, but there are no more available options for remaining parameters. For the *n*-queens problem, a user might put a queen in a field that seems ok, but there is actually no way to successfully finish putting all the remaining queens (without attacking each other).

Your Task

Your task is to implement an interactive *n*-queen configurator. That means, a user iteratively puts a queen on the board and the configurator calculates the remaining free chess fields that the user can select for the next queen. Depending on the selections of the user, the configurator should forbid some cells, or put the queens on the cell if it must be there. The following screenshots are presenting a use case.

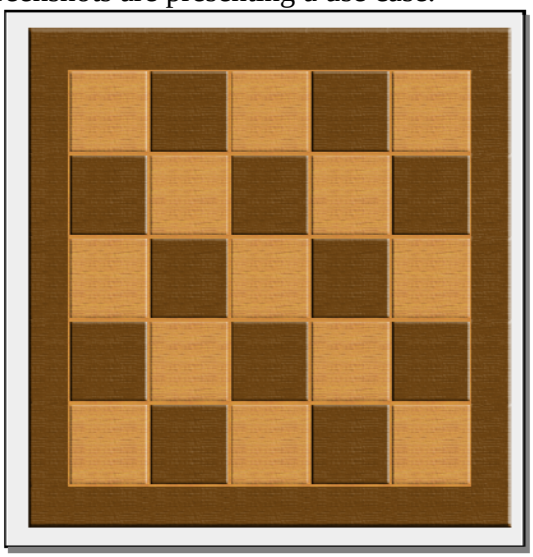

Figure 1 - board

The above screenshot is the empty chess board. You should be able to see the board if you compile and run the java code that is provided.

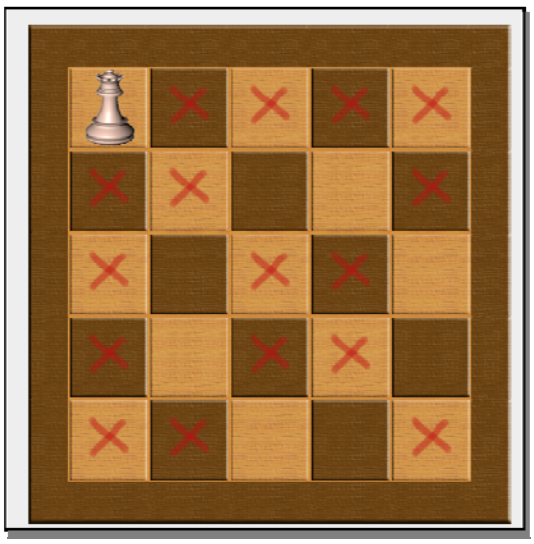

Figure 2 - Inserting the first queen

The screenshot of figure 2, is the result of inserting the first queen at the top-left corner. You can see that apart from the rows/columns/diagonal restrictions, there are also some "strange" crosses. That's because there will not be a solution if there is a queen at the top-left corner and at one of the crosses. This should be detected by the configurator, since it is hard for the user to figure out.

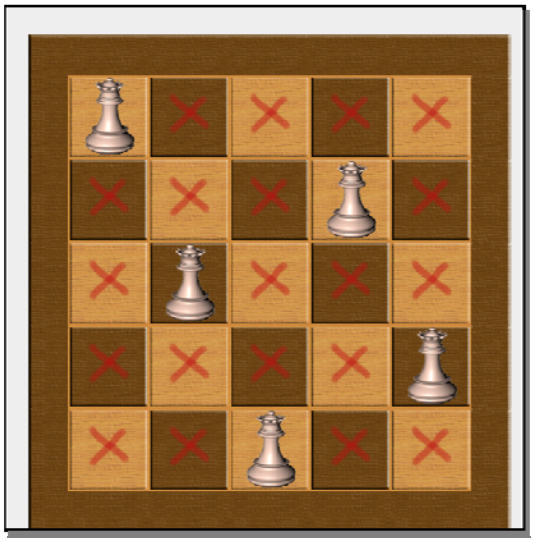

Figure 3 - Inserting the second queen

If the user inserts a second queen at bottom-middle (at this specific size of board and with this specific order), the configurator puts remaining three queens automatically since these are the only three remaining valid positions.

You should implement the interactive configurator using **Binary Decision Diagrams (BDDs).** All the rules about *n*-queens can be written as a *logical function* over some variables. You then *compile* the rules into a BDD. Then, all the valid options can be extracted from a BDD by *calculating valid domains* for every variable. The interactive configurator simply takes a user assignment, translates it into a logical restriction, restricts the BDD and then from the restricted BDD reads remaining available options.

What is provided

1. BDD Package

A package that helps you to create and perform operations on BDDs is provided. It is in the `javabdd-1.0b2.jar`. It gives you everything you need for BDD manipulation. The API of the package is also provided in the folder `apidocs`. Furthermore `BDDExamples.java` contains examples of how to use the package, the objects and some basic methods.

2. GUI
A java project is provided to you that consist of 3 classes.

`QueensGUI`
This is the class that visualizes the board. You don't have to change anything in this class. The only important method is the `mouseClicked`. In this method the column and row that the user clicked are taken and parsed through to the logic by `insertQueen`. We expect that the logic makes some changes to the board and finally we repaint it.

`ShowBoard`
This class starts the application. It initializes the GUI and the `QueensLogic`, it sets up the `Jframe` and finally shows it.

`QueensLogic`
This is the class that you have to implement. It will be explained at the "implementation suggestion" section. It is the one that will create the BDD and use it in such a way to calculate free board cells for remaining queens.

Compilation and Execution

To execute the initial code outside an integrated development environment (such as Ecplise), only using the command line, you need to specify a class path to `javabdd-1.0b2.jar`.

To execute on Windows, try the following:

```
java -cp "javabdd-1.0b2.jar;." ShowBoard
```

Note that a folder separator on Linux is ":" instead of ";", so for example to compile the code on Linux, you should use:

```
javac -cp "javabdd-1.0b2.jar:." *.java
```

Implementation suggestion

In order to allow you to choose your own programming style and way of defining things, we just describe what is needed, instead of creating java interface classes or abstract methods.

Step by step you have to do the following:

1. Build n-queen BDD
    a. Input: size of board n
    b. Output: BDD representing all the rules of the n-queens problem. It is recommended to use *n***n* BDD variables, one for each cell.
2. Add the rules of the n-queen problem to the BDD of the board (i.e. restrict the board BDD with the rules of n-queen).
    a. Input: n-queens BDD
    b. Output: the new BDD, restricted with the rules.
3. Get the assignment from the board.
    a. Input: the current board
    b. Output: n-queens BDD restricted with the current board positions
4. Get the Valid Domains
    a. Input: n-queens BDD
    b. Output: vector of possible Boolean values for each BDD variable.
5. Update the board according to the ValidDomains
    a. Input: vector of domains
    b. void: updates the positions of the board.
6. Call the appropriate methods from the `InsertQueen` method of the `QueensLogic` class.
    a. This function exists and is called from the UI when the user clicks. It takes the x, y coordinates of the queen that has to be inserted.
    b. Here, you can use the above functions to reduce the n-queens BDD with the user's choice, calculate valid domains and update the board.

You may declare some variables global and use them directly instead of having the input and output as described. You are also welcome to come up with your completely new/alternative implementation.

Practical Advices

The time to construct the initial BDD depends on the *node-number* and *cache* given to the BDD-package's initialization function. For fast compilation of a BDD representing an 8x8 board, we suggest using about 2 000 000 *nodes* and 200 000 *cache*. Depending on your machine capabilities, you can reduce the number of nodes, and keep the cache to be about 10% of the number of the nodes.

If you are thinking to start with a small board, I would recommend 5x5. You might encounter some weird behavior on 4x4 board as rules should forbid some initial selections, even before the user has started.

Consider your logical rules carefully. They should prevent two queens to be at the same row, at the same diagonal, at the same column. They should also enforce that in each column at least one queen *must* be put.

Create some extra print-out methods to help you check the validity of your code. You will have to fix a variable ordering in a BDD. We suggest you to create a function that for each cell variable x_{ij}, calculates its position in the BDD variable ordering (for the 8-queens example, each x_{ij} is mapped to a variable between x_0, x_1,…,x_{63}).

Hand-in

You should hand in the following:
• Compilable and runnable source code with useful/clarifying comments in Java or C#.
• A report of 1-3 pages explaining your solution, especially the key points of the implementation.