IAI Project 2 – The *n*-queens problem

Overview

Your task

In groups of 3-4 you are going to implement an interactive configurator that supports a user in solving an *n*-queens problem (rules further below). Preferably, the game should be implemented in Java and otherwise in C#; if you have other language-preferences, you should talk to the TAs to have it approved. It is required that you use BDDs to solve the task.

Do note that a BDD-package is provided for you and that this is a jar-file. Thus, if you do not write in Java, you need to implement a BDD data structure and methods for manipulating BDDs yourself, or alternatively find another package for it.

The *n*-queens problem

The *n*-queens problem is the problem of putting *n* chess queens on an *n* × *n* chessboard such that none of them is able to attack any other using the standard chess queen's moves. The colour 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. And since *n* queens should be put on the board, there *need* to be a queen in each row and in each column. For more information about the problem, you can check Wikipedia.

Interactive configurators

As mentioned, your task is to implement an interactive configurator. These 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. Therefore, if the configurator does not give suggestions on what to choose, the user might end up in a dead-end during interaction where no more options are available for the 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 other queens).

The application

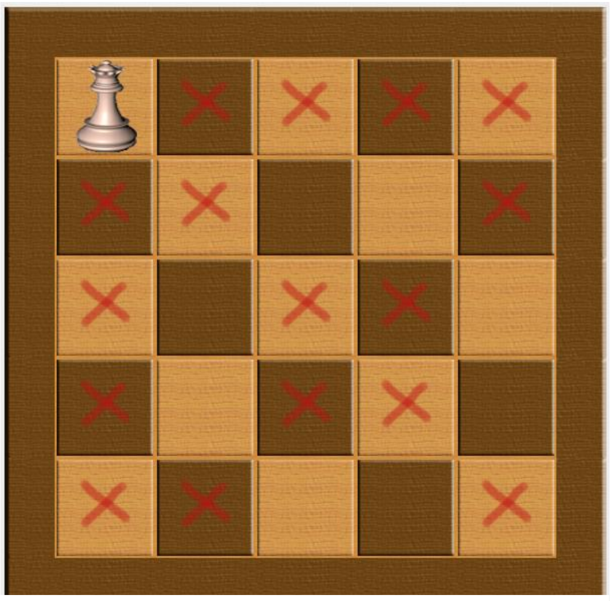
Figure 1- Inserting a queen on an empty 5 x 5 board

In the application, the user iteratively puts a queen on the board while the configurator calculates the remaining free chess fields that the user can select for the next queen. Depending on the user's selections, the configurator should forbid some fields (visualized by a red cross on the specific fields), or put queens on some fields if they need to be there. For example, Figure 1 shows the expected result of inserting the first queen at the top-left corner of the empty 5 x 5-board.

Apart from the "illegal moves" due to the rows-, columns- and diagonal restrictions, also other fields are marked as illegal. That is because there does not exist a solution where a queen is placed at the top-left corner while another is placed on one of the crossed-out fields

If the user now inserts a second queen at the bottom-middle, the configurator places the remaining three queens automatically since these are the only three remaining valid positions.

What is provided

Code

You are provided with a BDD package and java code support as explained below.

1. BDD Package

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

Read the class BDDExamples.java carefully, so that you at least have a basic idea how to make and manipulate BDDs using the package before starting with the *n*-queen problem itself.

2. Java code support

Four classes and an interface is provided. You do not have to change anything in the provided files. In fact, we recommend that you *do not change anything* in these classes. The classes/interface are:

- **QueensGUI** A GUI class that shows the board of the puzzle and listens for input (mouse clicks) from the user.
- **Queens** This is the class with the main-function. It reads the provided parameters, starts the application and shows the puzzle. It (hopefully!) gives reasonable feedback if you do not provide it with the required parameters; these are described a little further below.
- **IQueensLogic** This is the interface that you need to implement. It has three methods that should be implemented, namely *initializeBoard(int size)*, *getBoard()* and *insertQueen(int column, int row)*. The latter is the one you will spend most of your time implementing. See the documentation in the java-file to see the expected behaviour/output of these methods.
- **PrimitiveLogic** A simple implementation of the required interface. When inserting a queen it only puts the queen where requested and does not keep track of which other positions are made illegal by this move.

Read the documentation of the class **IQueensLogic** so you understand what methods you have to implement. If you implement in Java, you do not need to read/understand the code of **QueensGUI** or **Queens** (but of course, feel free to do so). Otherwise, i.e. if you implement in C#, on top of finding or making classes to manipulate BDDs and documenting this, you have to implement classes similar to **IQueensLogic** and **QueensGUI** and document how to use them.

When drawing the board, the GUI uses a 2-dimensional integer-array (returned by *getBoard()* in the interface), where the values are one of the following:

- 1 : there is or *has to be* a queen on the field in question.
- -1 : there cannot be a queen on the field in question
- 0 : neither of the above, i.e. given the current placement of the queens, it is still optional to put a queen on the specified field.

The columns on the board are numbered from left to right with the numbers from 0 to *n*-1, while the rows are numbered from *top to bottom* with the numbers from 0 to *n*-1.

Compilation and Execution

Setup

Download the provided files. Setup a project folder as you would usually do. In what follows, let's assume you call it "QueensProject". In order to show the images correctly, it is important, that "QueensProject" has a subdirectory called "imgs" which contains the images (just as it is in the folder you downloaded).

This time, if you want to use the provided code, you also need to include the jar-file. If you are using an IDE, then please refer to the IDE's documentation for including jars.

To execute the initial code outside an IDE, only using the command line, besides the parameters for the main-function you need to specify a class path to javabdd1.0b2.jar (see example further below).

Parameters

The main method accepts 2 parameters from the command-line. These are as follows:

1. The puzzle logic specified by the name of the class implementing **IQueensLogic**.
2. The size of a quadratic game board. This should be a number greater or equal to 5. This parameter is optional and defaults to 8 if not specified.

The smallest legal size of the board is set to 5. This is because the rules forbid some initial selections on a 4 x 4, even before the user has started.

Test that things are set up properly by running **Queens** with the parameter: 'PrimitiveLogic 6'. Either do this from your IDE or by opening a command prompt, go to "QueensProject"'s directory and try the following

```
java –cp "javabdd-1.0b2.jar;." Queens 6
```

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 hints and suggestions

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 this 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.

Solution approach (Spoiler alert!)

One approach to solving the exercise is as follows below. Though, you are also welcome to come up with your own alternative implementation, as long as you implement the interface (or corresponding methods if you write in another language).

1. Implement the first two functions.
   Use a 2-dimensional integer array to represent the board, initialize this in *initializeBoard*, and return it in *getBoard* (just like in the **PrimitiveLogic** class).

2. Create the initial rules.
   Use *n***n* variables and make a method for retrieving the correct variable or variable number for a given position (column and row). Make a BDD that represents the rules of the puzzle using these variables.

3. Make a method for assigning values to variables
   Make a method for restricting the BDD representing the rules when a queen is inserted a specific place (the corresponding value should then be true).

4. Get the valid values for each variable:

Given the rules and positions of the queens (variable assignments), find out if the variable has to be true (i.e. the corresponding position has to contain a queen), has to be false (i.e. it is not possible to place a queen at the corresponding position), or neither (i.e. it is possible, but not necessary, to place a queen at the corresponding position).

5. Update the board

   Make a method for updating the board according to the found valid domains of each of the variables.

6. Implement *insertQueen*

   Let insertQueen call the appropriate methods (i.e. make a variable assignment and update the board).

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 8 x 8 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.

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. However, don't necessarily implement all the rules at once; you could for example first try if your implementation works if queens can only attack each other horizontally.

Use print-out methods to help you check the validity of your code.

Hand-in

You should hand in the following:

- Compilable and runnable source code with useful/clarifying comments in Java or C#. Only provide classes that you have written yourself.
- A report of 1-3 pages explaining your solution, especially the key points of the implementation.

Name your project and class(es) such that it is easy for the TAs to figure out which group (number) they belong to.