Configuration Project
n-Queens Problem

Group 9

Anders Wind Steffensen

Mikael Lindemann Jepsen

&

Morten Albertsen

April 14th 2016

1 Approach

Before discussing the approach it is worth noting that in the game board, which is a 2D array of integers. In this array a value of 1 denotes a queen, -1 denotes a position where there cannot be a queen, and 0 denotes a position where the configuration still allows either value.

In the BDD however we only work on binary variables. In here `True` denotes a queen, while `False` represents a position that cannot have a queen. While a variable is unassigned (or not restricted enough) it can take on both values, representing the case where either a queen or a cross can be placed.
We decided to build up the BDD tree to begin with, to allow for easy insertion of queens.

1.1 BDD construction

The approach to building the *binary decision diagram* (BDD) goes in several stages.
A TRUE-BDD is used as a basis.

Row-rules:
By the rules of the n-queens problem, there must be exactly one queen in each row.

We implemented this by iterating over each row and for each row, an initially false BDD is built, `rowBDD`. Now, we iterate over the columns, producing a BDD, `ithVar`, corresponding to a queen being placed in that cell. When having placed a queen, we require that the remainding spots in the row are false, thereby making sure additional queens cannot be placed within that row. The `ithVar` BDD is now put into a disjunction with the `rowBDD`. With `rowBDD` being initially false, atleast one of the `ithVar` BDDs must be true, thereby requiring atleast one queen in the row. In the end all of the rowBDDs are put together as conjunctions requiring that there is atleast one queen in each row.

The same procedure is repeated for columns, producing a BDD that makes sure a queen is placed in each column. The BDDs of the row rules and the column rules are put into conjunction with eachother, requiring that it is true that there is both a queen for each row *and* a queen for each column.

The remaining rules of the *n-queens* problem is the diagonal rules, which states that a queen cannot be placed in the same diagonal, be that upwards or downwards, as another queen.

The construction of these BDDs are outlined in the following paragraphs.

Before diving in, it is worth noting that in contrast to the above rules, there is no requirement that a queen must be placed in every diagonal on the gameboard. Therefore we are going to use implications instead of conjunctions. The reasoning is, that placing a queen in the cell position implies that no queen must be placed in the diagonal from the cell.

We implemented the diagonal rules by iterating over each cell in the gameboard. `ithVar` represents the BDD in which the cell is set to true. Then, by travelling diagonally upwards to the boundary of the board from the cell, we reverse direction and traverse the diagonal and constraining every other cell from being true, and combine these into a conjunction, forming the upwards `diagonal` BDD. Hereafter we make `ithVar` imply `diagonal`.

A similar procedure is carried out for the downwards diagonal.

Finally, the diagonal-, row- and column rules are combined using conjunctions, producing the final ruleset needed for the *n-queens* problem.

When the game is initialized and the BDD has been built, the program checks whether the current configuration is satisfiable. If not, this means that there is no solution for the given configuration, and it therefore assigns crosses to all positions in the game board. This is important for game boards of size 2 and 3, where no solutions exist.

1.2 Placing a queen

When placing a queen, we check whether the position, where the queen is supposed to be placed, is either already occupied or if a queen cannot be placed there. If neither, we continue by placing a queen into that spot on the gameboard.

Continuing, the BDD is restricted, such that the variable in the BDD, that corresponds to the spot on the gameboard, is set to true.

The gameboard is then updated, such that it reflects the new restriction.

`updateGameBoard(...)` iterates over the variables in the BDD, and updates (if needed) the corresponding spot on the game board. This is done by restricting a given variable to true, and checking whether this assignment results in an unsatisfiable configuration. If this is the case, the variable must be assigned to false in the BDD and -1 on the game board. Similarly the variable is restricted to false, and the configuration is checked for unsatisfiability. Again, if this is the case, the variable must be assigned to true in the BDD and 1 on the game board.

In the case where neither results in an unsatisfiable configuration, we do not assign the value in the board. This is because this variable can be either value, based on the choice of the user.

By doing this, we achieve the placement of additional queens and crosses on the game board.

`UpdateGameBoard(...)` is also called when the game board is initialized to restrict positions where it is not possible to find a satisfiable solution.

2 Conclusion

We believe we've solved the problem. We have tested the implementation, and to our beliefs it works.
For game boards of sizes below 11, the BDD is created in less than one second. Size 11 takes a few garbage collections, but eventually makes it. We think this has to do with the recommended values for node table and cache.

We have checked that for game boards of size 2 and 3 there are no solutions, and that some positions are initially unavailable for certain sizes of game boards.

Furthermore the game board of size one, the only cell is initially assigned a queen.