N-Queens Problem

CISC352

Sean Tippett
10108181

Henrietta Virág
10142490

Below, functions have been added in the order in which the **minConflicts()** algorithm needs them to do their own tasks.

updateConflictHypothetical() takes the following 5 parameters: masterList, row, newColumn, oldColumn and csp. The function makes a hypothetical change to the contents of masterList based on the row, newColumn and oldColumn parameters. It is used for testing to see whether or not newColumn would be a better place than oldlColumn for the placement of the next queen, indicated by row. It increments the column count, left and right diagonal counts for this newColumn. It then decrements the column count, left and right diagonal counts for the oldColumn to account for the hypothetical use of newColumn. The function returns masterList.

The **bestColumn()** function takes a queen, the masterList and the problem size (csp). It calls **emptyColumnChecker()**, **randomColumnChecker()** and **seanSaysWeAreCheckingAllSquares()**. If **emptyColumnChecker()** returns a column that will have 0 conflicts after the queen is placed here **bestColumn()** uses it as the next best location for the queen. Otherwise it returns -1, in which case **randomColumnChecker()** is called to find a column that would have a single conflict after the queen has been placed there. If **randomColumnChecker()** returns a column that guarantees a single conflict, then **bestColumn()** will use this as the next best column. If it returns -1, the randomly generated list didn't contain a column that would result in a single conflict after the placement. Now **seanSaysWeAreCheckingAllSquares()** is called to find a location for the current queen. If **seanSaysWeAreCheckingAllSquares()** is called, its result will be returned by **bestColumn()**.

emptyColumnChecker() takes a queen, the masterList, the problem size and an originalColumn. It iterates through the list of empty columns and makes a hypothetical change to the input matrix by placing the queen in this column. It then calls **conflicts()** to determine the number of conflicts that were created with this column choice. If there are no conflicts it returns this column as the next best choice to place a queen. If putting the queen in this column results in additional conflicts, reverse the hypothetical change by using the originalColumn parameter that was given to the function. It returns -1 in this case to indicate that it didn't find a 0 conflict column. When **emptyColumnChecker()** can't find a 0 conflict column it's not useful anymore. Now, **bestColumn()** calls **randomColumnChecker()** with a queen, the masterList, the problem size (csp) and an originalColumn as its parameters. It generates a list of size 100. If csp is less than 100, the list size will be csp itself. The function fills the list with random numbers in the range from 1 to csp, inclusive. Each of these numbers will represent a column in the matrix. It then takes each column of this randomly generated list to observe the input matrix at that column. It makes a hypothetical change by putting the next queen in each of these randomly chosen columns until it finds one in the input matrix that results in a single conflict. If the result of calling the **conflicts()** function is 1 conflict, we use this column to put the next queen there. Considering the matrix setup up until this point, the function will easily find a queen with a single conflict. Otherwise, **randomColumnChecker()** will reverse the hypothetical change and returns -1. **bestColumn()** receives this negative integer and goes on to call **seanSaysWeAreCheckingAllSquares()** that takes a queen, the masterList, the problem size (csp) and an originalColumn. It searches all columns for a minimum conflict position. For each column, it makes a hypothetical change by placing the queen here for testing. It then calls **conflicts()** to see if this placement results in a single conflict. This is vely likely to occur so it returns this column because this is the best decision that it can make. If placement of the queen in this column doesn't result in a single conflict, **seanSaysWeAreCheckingAllSquares()** will undo this hypothetical change and it will return the originalColumn, which will then also be returned by the **bestColumn()** function.

The **initialize()** function takes two parameters, masterList and csp, which is the problem size that determines the dimensions of the chessboard. It achieves a suboptimal solution by setting up the board such that it is as close to the actual solution as possible based on the 4 constrains of the problem. It fills the initialMatrix, colCounts, rightDiagonalCounts, leftDiagonalCounts lists with zeros and the emptyColumns list with ones to initialize the masterList. It then shuffles the list of empty columns; so instead of having the number of columns in ascending order (i.e.: [1, 2, 3, 4] for csp = 4), the list is in random order (i.e.: [3, 2, 4, 1]). The function iterates through the initial matrix row by row and adds a queen to a carefully chosen column in each row. The column to be used for the next best placement of a queen is provided by the **bestColumn()** function. After the placement of the queen, this best possible spot is guaranteed to result in the minimum number of conflicts in that column. Once a queen is placed in a column, this column is removed from the list of empty columns (emptyColumns). That way the function won't choose this column again in the future for another queen. The function returns the initialized masterList as its result.

leftDiagonalConflicts() takes a queen, the masterList and the problem size (csp). The queen parameter is used to indicate the current row number. The list of left diagonal conflicts in masterList contains 2n-1 number of entries, one for each left diagonal in the matrix. To get the left diagonal conflicts, for each (row, col) that has a queen in this location, this queen belongs to a left diagonal. The formula ***((csp - 1) - (row - (col - 1)))*** will give the index in the left diagonal list where this queen belongs. If the entry at this index in the list is 1, there is no conflict in this left diagonal because we only have a single queen there. If the entry is more than 1, we have a conflict for this left diagonal. This function returns an integer, which is the number of conflicts that it finds for the current queen in its left diagonal.

rightDiagonalConflicts() takes a queen and the masterList. The queen parameter is used to indicate the current row number. The list of right diagonal conflicts in masterList also contains 2n-1 number of entries, one for each right diagonal in the matrix. To get the right diagonal conflicts, for each (row, col) that has a queen in this location, this queen belongs to a right diagonal. The formula ***(row + (col + 1))*** will give the index in the right diagonal list where this queen belongs. If an entry at the found index in the list is 1, there is no conflict in this right diagonal because we only have a single queen there. If the entry is more than 1, then we have a conflict for the right diagonal of this queen. The function returns an integer, which is the number of conflicts that it finds for the current queen in its right diagonal.

The **conflicts()** function takes a queen, the masterList and the problem size (csp). It operates on the initialized matrix and returns the total number of conflicts for the current queen. It counts the number of queens that are currently in the column. It then calculates the number of left and right diagonal conflicts for this queen using the functions **leftDiagonalConflicts()** and **rightDiagonalConflicts()**, respectively. It returns the total number of conflicts for this queen by adding up the above 3 results.

constraints() takes the masterList as its only parameter. It returns false if any of the 4 constrains have been violated for any of the queens and true if none of the constraints have been violated for any of the queens on the board. The function uses colCounts that contains n entries, the number of queens for each column of the matrix. If there is more than 1 queen per column **constraints()** returns false because this constraint has been violated. It takes the right diagonal counts list from the masterList and checks to see if there's any entry that is larger than 1. If yes, a conflict exists and **constraints()** returns false. It does the exact same thing for the right diagonal counts. **constraints()** returns true if and only if all entries of both rightDiagonalCounts and leftDiagonalCounts are either 0 or 1 and every entry of colCounts is 1.

legalMove() takes a queenToRepair, the queensMoved list, a potentialColumn, the masterList and the problem size (csp). Before it moves a queen, it needs to make sure that after placement this queen won't conflict with another queen that has been moved before.

minConflicts() takes the problem size (csp) and it is the main algorithm that aims to solve the problem by repairing a chessboard with a suboptimal solution. If the problem size is less than 1000, this number will be the maximum number of steps the algorithm will take to find a solution. This function operates on the initial matrix that is a suboptimal solution provided by the **initialize()** function. The function first checks whether the initial board happens to be the solution by calling **constraints()**. If there are no constraints violated, there is no need to repair and the masterList is returned. It then randomly chooses queens with conflicts to repair. The function calls **bestColumn()** to make a hypothetical change. In this change, the column for the current queenToRepair is provided by one of the functions of **bestColumn()**. For optimization purposes, when the problem size exceeds 500, **minConflicts()** calls **legalMove()** and keeps two lists, one for moved queens and one for those queens that have not been moved yet. This way we make sure that after placing the current queen in the best possible column with the minimum number of conflicts, the queen won't conflict with another queen that has been moved previously.

algoHandler() takes the problem size as its parameter and sets up the ranges for the small, medium and large input sizes. It allows a certain number of attempts to solve the problem for each of the problem sizes. By allowing multiple attempts, the need for backtracking is prevented, especially as the problem size increases.