Technical Document

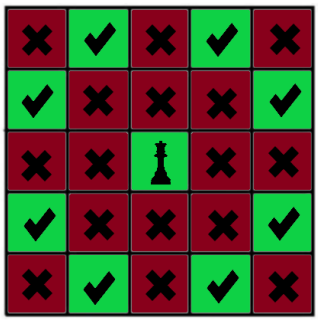
Solving the N-Queen Problem Using a Min-Conflicts Algorithm

March 2, 2018

Author: Garett MacGowan

Problem

The N-Queen problem, inspired by the game of chess, is a constraint satisfaction problem in which the constraints are defined by "n" queens, where $\{n \in \mathbb{N} \mid n = 1 \text{ or } n > 3\}$. For n queens, the size of the chess board for which the queens will be placed is n × n squares. Each n queen is a classic chess queen which can attack any other chess piece positioned vertically, horizontally, or diagonally in relation to its location on the board. For the problem to be solved, no queen should be placed on the board such that it can attack another queen without an intermediate move and there should be n queens located on the n × n chess board.



The illustration to the left depicts a queen and its "safe spots" highlighted in green, and "conflicting spots" highlighted in red. For the center queen to not violate problem constraints, no other queen may be placed in a red square. Successive queens placed on the board may only be placed within the green squares. Each time a queen is placed, the problem becomes more constrained.

Considerations

In a naïve approach, one might try and randomly place queens on the chess board until a constraint satisfying configuration is found. This is obviously a bad solution because the probability of generating a constraint satisfying configuration is extremely low.

One method for generating a constraint satisfying configuration is a backtracking algorithm which works by placing queens one row at a time in positions which satisfy the constraints for all previously placed queens. Once it becomes obvious that the constraints cannot be satisfied with the selected queen positions, the algorithm backtracks to a state from which it thinks it went wrong in placing the queens. This method is not as poor as the naïve approach, but it is still quite slow as n becomes large.

Solving with Min-Conflicts Approach

A minimum conflicts approach to solving the N-Queen problem is a good way to find a solution more quickly than backtracking and naïve approaches. In a minimum conflicts approach, an initial board configuration phase is used to create an arrangement of queens that is close to correct. That is, the configuration phase attempts to create a randomly generated board that has minimum conflicting queens. To accomplish this, a Greedy approach is used. The approach works as follows: Queens are placed one row at a time, with only one queen being placed per row. Each queen placed contributes values to a heuristic lookup list which is used to

determine the conflict cost of each position in the rows below it. Each time a new row is considered, the costs of placing a queen at each column in that row are determined using the heuristic lookup list. A column which is not already occupied by a queen is randomly selected and its cost is determined. If the column has competitively low cost, the queen is placed there. The process continues until all rows are occupied by a queen.

The next step of the minimum conflicts approach is to repair the solution generated by the Greedy board configuration. This step utilizes local search, and is thus not guaranteed to generate a constraint satisfying board configuration. In the repair phase, a list of conflicting queens is determined, and conflicting queens are moved in the order of highest to lowest conflict. The queen selected to be moved considers the cost of all positions in its row, and moves itself to the lowest cost position. This process continues until either a solution is found, or it is determined that a solution will likely not be found due to local optima.

Attempted Solution

In the section below, I have described the functions which my solution uses to solve the n-queen problem with a min-conflicts approach. All functions described below are written in Python version 3.6 with standard libraries.

main():
Called by
Program start
Calls
readProblemFile(), createNChessBoard(), solveNQueen(), writeSolutions()
Data Structures Present
problemList, solutionList, currentBoard, occupiedPositiveDiagonals, occupiedNegativeDiagonals, result
Description

This function handles receiving a list of problem sizes from readProblemFile() which it then uses to generate initial board configurations via createNChessBoard() and solutions using solveNQueen(). Every solution generated is appended to the solutionList and is sent to writeSolutions() to be written to a file.

readProblemFile():
Called by
main()
Calls
None
Data Structures Present
problemList
Description

This function reads lines from a file called “nqueens.txt” located in the working directory of the script. The file should contain lines with one number on each line. Each number is

appended to the problemList and represents the n × n board size which the problem will generate and solve.

writeSolutions():
Called by
main()
Calls
None
Data Structures Present
solutionList
Description

This function handles writing the n-queen solutions generated by the algorithm to a file named “nqueens_out.txt”. Each line in the file is the solution to the problem size defined by the input file “nqueens.txt”. The solution format is such that the first element in each list represents the column position of the queen in the first row of the board. As such, the second element represents the column position of the queen in the second row. This scheme continues for each element in the solutionList.

CreateNChessBoard():
Called by
main()
Calls
largeGreedHelper(), smallGreedHelper(), bisect.bisect_left()
Data Structures Present
board, occupiedPositiveDiagonals, occupiedNegativeDiagonals, unchosen
Description

This function handles generating the initial board configuration. It works by first generating a list of unchosen column positions. Next, it looks through each row, determining the column for which a queen should be placed using the largeGreedHelper() and smallGreedHelper() functions. Once a queen is placed in a column, the position is removed from the list of unchosen columns. Additionally, the positiveDiag and negativeDiag values derived from the row and column of the selected queen are added to the occupiedPositiveDiagonals and occupiedNegativeDiagonals lists in a sorted order using the bisect.bisect_left() function, which uses binary search to find an insertion point. This is all to assist in heuristic generation. Once queens have been placed in each row, the filled board is returned to main() along with the occupiedPositiveDiagonals and occupiedNegativeDiagonals lists.

Note: rows <= 66% n are calculated by largeGreedHelper() and rows > 66% n are calculated by smallGreedHelper(). This is due to the high cost of list deletion for large n. Clustering becomes a problem for largeGreedHelper() around 66% n and is thus switched out for smallGreedHelper() when this occurs.

smallGreedHelper():
Called by
smallGreedHelper(), createNChessBoard()

Calls
smallGreedHelper(), costCheck()
Data Structures Present
occupiedPositiveDiagonals, occupiedNegativeDiagonals, unchosen, tempUnvisited
Description

This function handles selecting queen column positions. The function randomly selects potential positions from a list of unchosen columns. It then calculates the cost of placing a queen at that location via the costCheck() function. If the cost of placing the queen at the column is acceptable, the function returns the column and cost of that position. Otherwise, the function keeps track of a list of unvisited columns and randomly searches the list for acceptable positions, updating the list of unvisited columns as it progresses. If no acceptable position is found, the function recursively calls itself with the next best cost target which it saw during the previous search cycle.

Note: This method is most useful for small n values as it is costly to maintain the list of unvisited columns. On the other hand, selecting spawning locations for queens in the most random way is a major contributing factor that allows the algorithm to find suitable locations quickly. Early clustering negatively impacts the Greedy approach's performance as it nears completion. This is a major consideration in the largeGreedHelper() function.

largeGreedHelper():
Called by
largeGreedHelper(), createNChessBoard()
Calls
largeGreedHelper(), costCheck()
Data Structures Present
occupiedPositiveDiagonals, occupiedNegativeDiagonals, unchosen
Description

This function serves the same purpose of the smallGreedHelper() function but it exists to speed up the board configuration phase for large problem sizes. It works to eliminate the high cost associated with maintaining the tempUnvisited data structure used in smallGreedHelper(), while also keeping clustering in check. To do this, a column is randomly selected from the list of unchosen columns. If the cost of placing a queen at the selected column is acceptable, then the function returns the column and cost of that position. If the cost of placing a queen at the selected location is not acceptable, the function searches through the remainder of the possibilities in a hybrid random/linear way. It randomly begins searching from either the right or the left side of the randomly selected column. It also randomly selects a direction to search (ascending or descending). If a suitable column is found, the function returns the column and cost of that position. If a suitable column is not found after searching through all options, the next best position is determined by a recursive call to the function with the new cost target set to the most competitive cost seen.

SolveNQueen():
Called by

main()
Calls
checkConflicts(), costCheck(), searchList(), bisect.bisect_left(), removeFromVerticals(), repairConflicts(), validateConflicts(), sorted()
Data Structures Present
currentBoard, occupiedVerticals, conflictingQueens, columnDecider, occupiedPositiveDiagonals, occupiedNegativeDiagonals,
Description

This function works to repair the board configuration generated by the configuration phase. Since the repair is using a local search strategy, a maxSteps number is defined to prevent the program from hanging on local optima. Since the board configuration phase guarantees that only one queen would be placed per column, the occupiedVerticals list is initiated as such. The list of conflictingQueens is used to determine which queens to move and is sorted in ascending order based on the quantity of conflicts. The queen with the highest amount of conflicts is moved first. If hanging due to the max conflict queen not resolving to a lower conflict state is detected, a panic mode is initiated which enforces conflicting queens to be randomly selected.

For each step in maxSteps, a conflicting queen is chosen and the cost of moving the queen to each column in its row is determined by a summation of the costCheck() function and the amount of queens already placed on the column. If a zero conflict location is spotted, the queen is moved into that position and the heuristic lookup lists occupiedPositiveDiagonals, occupiedNegativeDiagonals, and occupiedVerticals are updated to reflect the change. Additionally, the queen that is now in a non-conflicting position is removed from the list of conflictingQueens and the next conflicting queen is considered. If a location has less than or equal conflicts to the queen which we are trying to move, the column along with its amount of conflicts are appended to the columnDecider list. The columnDecider list is used to determine a location to move the queen to if no column is found to have zero conflicts. columnDecider is sorted in ascending order and one of the cheapest columns is selected to be the position to move the queen to. After the position is selected, occupiedPositiveDiagonals, occupiedNegativeDiagonals, occupiedVerticals, and conflictingQueens are updated accordingly. Since the position which the queen has been moved to has the possibility of introducing a conflicting queen which is not already in the list of conflictingQueens, the conflictingQueens list is repaired with the repairConflicts() function. Finally, the costs of all conflicting queens must be validated with validateConflicts() because introducing a conflict on one queen means introducing a conflict with at least one other queen. If the length of the conflictingQueens list ever reaches 0, the algorithm knows it has found a solution and the board is returned.

checkConflicts():
Called by
solveNQueen()
Calls
costCheck()
Data Structures Present
currentBoard, occupiedPositiveDiagonals, occupiedNegativeDiagonals, occupiedVerticals

Description

This function is used to generate the list of conflicting queens present on any given board. It works by calculating the sum of the result of the costCheck() function and the amount of queens present in a given column. It builds the conflictingQueens list in the form [(column, conflictCount, row)]

costCheck():

Called by

smallGreedHelper(), largeGreedHelper, checkConflicts(), solveNQueen(),

Calls

searchList()

Data Structures Present

occupiedPositiveDiagonals, occupiedNegativeDiagonals

Description

This function determines the diagonal lookup values given a column and a row and then passes them to searchList() in order to determine how many diagonal conflicts are present in any given board position. The function returns the quantity of diagonal conflicts at the specified column and row in the board.

searchList():

Called by

costCheck()

Calls

bisect.bisect_left()

Data Structures Present

space

Description

This function determines the quantity of a target value present in a sorted list. It works by finding the left-most index of the target in the space it is searching using the binary search based bisect_left() function. It then steps through the list adding 1 to a counter each time it comes across the target value. Once the target value is no longer found, the function returns the resulting count.

removeFromVerticals():

Called by

solveNQueen()

Calls

None

Data Structures Present

occupiedVerticals, columnOccupantIndices

Description

This function works to remove a queen from the occupiedVerticals list by searching through the list of queens at a specific column until it finds the queen at the specified row and deletes it. The function returns the updated occupiedVerticals list.

repairConflicts():
Called by
solveNQueen()
Calls
conflictHelper()
Data Structures Present
conflictingQueens, occupiedPositiveDiagonals, occupiedNegativeDiagonals, occupiedVerticals, negativeDiagPossibles, positiveDiagPossibles
Description

This function works to repair the conflictingQueens list after a queen has been moved into a conflicting position. Its purpose is to add conflicting queens to the list of conflicting queens which are missing. It works by first evaluating missing conflicting queens vertically, then diagonally from a reference point on the board. The function assumes the reference point is the position of a queen that has just been moved. The vertical search simply scans the list of queens occupying the column, and checks if the queens are present in the conflictingQueens list. If they are not present, they are added to conflictingQueens in the required format. When performing the diagonal checks, a conflictHelper() function is used to generate a list of queens which should be present conflictingQueens. Each queen defined by the conflictHelper() is then checked against conflictingQueens and missing queens are added.

conflictHelper():
Called by
repairConflicts()
Calls
bisect.bisect_left()
Data Structures Present
space
Description

This function builds and returns a list of conflicting queens assuming space is either occupiedPositiveDiagonals or occupiedNegativeDiagonals. It works by finding the left-most index of the target in the space it is searching using the binary search based bisect_left() function. It then steps through the space, adding each row and column to the conflict list. Once the target value is no longer found, the function returns the resulting conflictList.

validateConflicts():
Called by
solveNQueen()
Calls
costCheck()
Data Structures Present
conflictingQueens, occupiedPositiveDiagonals, occupiedNegativeDiagonals, occupiedVerticals
Description

This function is used to validate the costs of each conflicting queen in conflictingQueens. It works by calculating the amount of conflicts present for each queen and checking the value

against what is stored in the conflictingQueens structure. If the conflict count recorded in the structure does not match what was calculated, the conflictCount is updated. Once all conflicting queens have been validated, they are returned.

Acknowledgements

Due to time restrictions, I was unable to increase the efficiency of my algorithm to a satisfying state. I hope to be able to solve n = 1,000,000 in five minutes or better. Specifically, the solveNQueen step could use some major design improvements regarding majorly redundant checking. I have learned many things over the course of this development process and hope to further improve upon my solution.