Solving N-Queen's Puzzle through Constraint Satisfaction Models in Kotlin*

Corey J. Reichel
The University of Dayton
Dayton, Ohio USA
reichelc1@udayton.edu

ABSTRACT

Puzzles such as eight spaces or queens games are useful for testing the efficiency of a given algorithm in which there is an exhuastible domain. These puzzles are good models to develop algorithms that have applications to various domains in computer science [1]. With these types of puzzles minimizing search space dramatically improves performance and memory usage as the size of the problem increases. We demonstrate valid solutions to Queen's game on varying size boards with different types of algorithms using lazy evaluation and depth first search algorithms in Kotlin. Starting from simple brute-force algorithms that merely guess and check, to more refined algorithms such as depth-first and look-ahead that attempt to trim the search space by eliminate unnecessary computations.

KEYWORDS

Arc Consistency, Constraint Satisfaction Problem, Queens Puzzles, Forward Checking,

ACM Reference Format:
Corey J. Reichel. 2019. Solving N-Queen's Puzzle through Constraint Satisfaction Models in Kotlin. In *Proceedings of CPS 452: Emerging Programming Languages (CPS 452),* Saverio Perugini (Ed.). ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Due to their simplicity, brute force algorithms such as trying every possible outcome and checking are easy to understand and implement. However, as problem spaces become large and solution constraints become complex and numerous, brute force algorithms quickly become untenable. We examine N-Queens puzzle through the Constraint Satisfaction model as a solution to the performance inefficiencies incurred by brute force methods, while maintaining relative algorithm simplicity.

N-Queens puzzle was selected as the problem these algorithms operate over because there are currently no known formulas that relate the size of the board to fundamental solutions or to non-fundamental solutions (where non fundamental solutions are solutions that can be obtained from rotating, reflecting, etc. fundamental solutions). Additionally, N-Queens is non-deterministic and "at least as hard as the hardest problem in NP" [3].

*Produces the permission block, and copyright information

Table 1: Sample Solution Sizes for N-Queens

N	Fundamental	Total
4	1	2
5	2	10
6	1	4
7	6	40
8	12	92
9	46	352

2 N-QUEENS PUZZLE

In 8 Queens puzzles, we construct an 8x8 chess board and attempt to place 8 queens. These queens must be placed so that no queen is in a row and column with another queen. Additionally queens cannot occupy the same diagonals as another queen. Generalizing this, for any *N* board we attempt to place N-queens on that board. It is known for N = 2,3 there are no solutions as a queen will share a row, col, or diagonal with another queen.

A brute force solution that checks every possible arrangement of queens has 4,426,165,368 (64 choose 8) possible solution to enumerate and then check for validity. This can simply be improved by eliminating all those in which there is a queen in the same row and column as another, bringing it down to 40320 (8!) solutions. However in the General case, N*N choose N is abysmal and N! is also unsatisfactory. Thus there is a need for algorithms that reduce the search space.

3 CONSTRAINT SATISFACTION PROBLEM

Constraint Satisfaction Problems (CSP) attempt to model problems through the use of constraints and states. The constraints for N-Queens puzzles would be that no queen can share a row, column and diagonal with another. Constraints can be either abstracted as above, or explicitly enumerated. These states need not be completed answers, but are generally subject to the constraints of the problem (eg., a valid state could be one where 3 out of 8 queens have been placed, but these queens do not interfere with each other yet).

State progress to another state by taking an action (placing a queen on the board). Depending on implementation actions may: lead to a state that is not evaluated for validity as the final solution is instead evaluated (place all queens, then check a completed board for validity), lead to a state that is then evaluated for validity (place a single queen then check the potentially incomplete board), lead to an immediately valid state but not necessarily lead to a valid solution (place a single queen only in a non-conflicting space).

3.1 Back-Tracking

All of the algorithms we present today were implemented through back-tracking variants of CSP. In back-tracking models a depth first approach is used to discover an answer or, as in our case, the set of all possible answers. Once we discover that our current state is either invalid or will never contain completed valid solutions, we return one move back, remove the previously selected move from the valid list and continue with a new value until the entire tree has been exhausted

3.2 Forward Checking

In Forward checking, we will only take actions that lead to immediately valid solutions. For an N-queens problem, each column of the board is an array list that contains all remaning non-conflicting squares (ie. rows) for that column. At the start each column will have numbers 1,2...N as no queen has been placed. Once a queen has been placed we remove all conflicting values from each column. When we place the second queen we select from a column that has the least number of options, placing a queen at one of the remaining rows within that column, again updating accordingly. If there is ever a column that does not have a queen and does not have any remaining row values, then there is no possible solution for the current arrangement of queens and therefore does not require further searching. Whenever an Nth queen has been place, a solution has been found. Boards do not need to be checked for validity because actions only return valid states. Actions do not need to be evaluated for validity as the remaining rows in each col will only ever be valid actions because all invalid actions are removed after a queen is placed.

There are two benefits to forward checking: not checking the validity of a given action, and trimming the search space. Although an action and state is not explicitly checked for validity, removing invalid actions from the set of all actions might be considered validation. However, often times it is far simpler and efficient to check the ramifications of an action and update accordingly, than it is to check if the action itself is valid. As more queens get placed, the list of rows within each column gets smaller, therefore each queen placed will effectively shorten the computation required for the next queen to be placed. Removing rows from a column removes entire branches of computation that would ultimately result in no solutions be discovered, thus each iteration effectively trims the search space.

3.3 Arc Consistency

The last algorithm we implement is Arc Consistency. In Arc Consistency we again trim the search space after taking an action. However after the initial cull, the remaining valid spaces are again evaluated for future validity based off of implications of the current board and the constraints.

Applying this to N-Queens, we first start by taking an action that yields an immediately valid state by selecting a column with the fewest remaining valid spaces for a queen. After updating the valid spaces matrix based on the space selected we check the remaining

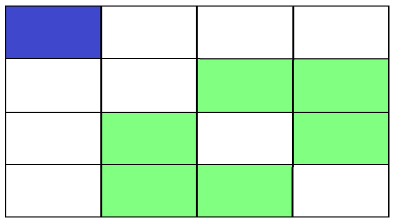

Figure 1: Board after placing one queen. Blue denotes spaces with a queen, green denotes still valid spaces for a queen and white denotes invalid positions for a queen.

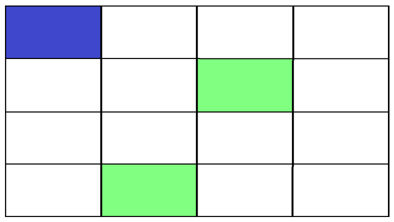

Figure 2: Board after all arcs have been accounted for and no more implications can be drawn. Note: This is implementation dependant and not the only possible conclusion to be drawn. However the conclusion will demonstrate that there are no solutions with a top left corner queen.

arcs, or constraints. If there is an implication that an action is only temporarily valid but ultimately leads to no solutions then remove it from the action space. we continue to check arcs until no we have checked each arc and no changes have occurred to the action space (or until a row/col is empty). Note that once a change has occurred to the action space all previously checked actions that are still in the action space must once again have their arcs evaluated.

For example, in a 4×4 square placing a queen in the top-left corner of the board will remove many potential actions from the board (figure 1). However if we look at the second column, the only options are 3 and 4, both of which indicate that column 3 row 4 is an incompatible space for a queen to also be in. Because column 2 row 3 and 4 are the only option in column 2 and we must have a queen in every column, column 3 row 4 can never be yield a solution. Once we remove that we also notice that because of column 3 row 2, column 2 row 3 is also invalid and so on and so forth (figure 2). We stop checking for implications after every arc for every remaining valid position on the board has been checked and has caused no further changes. Then we once again place a queen as we did before, this time with a much smaller action space.

Arc consistency is faster at trimming the search space at earlier stages than forward checking as it is an augmentation to forward checking. However, in many implementations, checking the arcs itself will incur a hefty computation price as well as require a non-trivial amount of memory usage when checking arcs. The more arcs there are the higher these penalties will become while also increasing the degree to which the tree is trimmed at earlier stages. In general however, arc consistency will out perform forward checking.

4 CONCLUSION

Forward checking and Arc consistency are amongst the most effective paradigms for search problems [2]. We conclude that through the usage of forward checking and arc consistency enabled through the CSP model, we are able to effectively decrease the search space of non-deterministic problems such as N-Queens.

REFERENCES

[1] M. A. Ayub, K. A. Kalpoma, H. T. Proma, S. M. Kabir, and R. I. H. Chowdhury. 2017. Exhaustive study of essential constraint satisfaction problem techniques based on N-Queens problem. In *2017 20th International Conference of Computer and Information Technology (ICCIT)*. 1–6. https://doi.org/10.1109/ICCITECHN.2017.8281850

[2] C. Bessiere and J. . Regin. 1994. An arc-consistency algorithm optimal in the number of constraint checks. In *Proceedings Sixth International Conference on Tools with Artificial Intelligence. TAI 94*. 397–403. https://doi.org/10.1109/TAI.1994.346465

[3] S. Güldal, V. Baugh, and S. Allehaibi. 2016. N-Queens solving algorithm by sets and backtracking. In *SoutheastCon 2016*. 1–8. https://doi.org/10.1109/SECON.2016.7506688