Artificial Intelligence CS 534, Spring 2019

Project Part II Report

By Prakash Baskaran and Archit Kumar

Objective

To design, implement, and apply a Constraint Satisfaction Problem (CSP) Solver to an application in the area of “Computer Games and Puzzles”.

Introduction

A constraint satisfaction problem consists of mainly three domains, the variable domain, the value domain, and the constraints domain. A problem is solved when each variable has a value that satisfies all the constraints on the variable. Several heuristics are used along with constraint propagation and the search method to eliminate large portions of the search space by identifying variable/value combinations that violate the constraints.

Our specific problem belongs to the “Computer Games and Puzzles” domain. We sought to implement techniques learnt in the course CS 534, on solving the ‘n-queens’ problem. This problem first appeared in the form of 8-queens puzzle back in 1848. Franz Nauck extended the puzzle to the n-queens problem.

Desired Goal

The general goal for this problem is to place ‘n’ queens on a ‘n x n’ chessboard in such a manner that no two queens attack each other, considering that a queen has the freedom of moving either horizontally, vertically, or along a diagonal on the chessboard.
In order to have a reference for setting our goal, we looked up the time taken by a tool developed by Google, whose statistics are posted on the following link: -
(https://developers.google.com/optimization/cp/queens).

“Google” says that their algorithm did not seek to find solutions fast but to find solutions using a simple approach. They sought to count the total possible number of solutions for a given number of queens while measuring the time taken to come up with the solution by the algorithm. With this in mind, we set our desired goal as trying to solve the “n-queens problem” for up to 30 queens. Also, instead of counting all possible solutions for the problem, we decided to only look for one possible solution. This way we can reduce the time required to arrive at a solution.

Constraint Satisfaction Problem Description

The ‘N-queens’ problem can be formulated as a constraint satisfaction problem in the following manner. We have the following sets:

Variable Set:

Every queen is treated as a variable.

$V = \{0^{th}, 1^{st}, 2^{nd}, \ldots\ldots\ldots, n^{th} \text{ queen}\}$

Domain Set:

Every position on the chess board is part of the domain of the queen. Note that to optimize our code, the queens are placed one at a time, with every row/column containing one queen piece. Hence, assuming that we are placing queens one column at a time, we have the following domain for the n^{th} queen:

$D_n = \{(0,n), (1,n), (2,n), (3,n), \ldots\ldots, (n,n)\}$

Constraint Set:

Lastly, considering that a queen piece can move horizontally, vertically, and diagonally, for the constraint set we have three constraints for queen placement which includes constraint on row, column constraint and diagonal constraint. To avoid constraint conflict no two queens can be placed in the same row, column or diagonal.

Hence, for a pair of queens, say ‘p’ and ‘q’, where the chessboard positions are defined in the form of (i, j) coordinates, we have constraints for queen ‘p’ (C_p) as : -

- $i_p \neq i_q$, Row constraint
- $j_p \neq j_q$, Column constraint
- $abs(i_p - i_q) \neq abs(j_p - j_q)$, Diagonal constraint

Methodology

- **Search Method:**

  Backtracking search, in case of a failure while assigning a variable a value from its domain, we shall backtrack to the most recent previous assignment and try another valid value.

- **Heuristics:**

1. Minimum Remaining Values (MRV): To decide variable ordering, based on the remaining number of valid values in the domains of unassigned variables.
2. Degree Heuristic: In case of the 'n-queens' problem the "degree heuristic" does not help in solving ties in variable ordering because every queen is involved in an equal number of constraints with respect to every other variable in the variable set. Hence, we proceed to pick a random variable for assignment in the case of a tie.
3. Least Constraining Value (L.C.V): To aid in value ordering this heuristic will select a value to assign to a variable such that it constrains other unassigned variables minimally.
4. The Arc Consistency Algorithm(AC-3) has been implemented before the execution of the program as pre-processing to handle constraints and avoid conflicts.
5. Maintaining Arc Consistency(M.A.C): Additionally, Arc-consistency has been called during each assignment when the program runs, in the form of the M.A.C algorithm. When assignment to a variable X_i happens then AC-3 is called which starts with a queue of all arcs that are formed between the variable X_i and its unassigned neighbors and AC-3 then proceeds in the usual manner. This way we ensure a recursive propagation of constraints.

- Our program finds **only one solution** from the set of all possible solutions. However, we this solution and exploit the symmetric nature of the problem to find other possible solutions.
- The solution obtained is converted into a matrix and then we **rotate the matrix thrice** in counter-clockwise direction to get 3 more solutions.
- We also **flip the matrix vertically, horizontally and diagonall**y (transpose) to get 3 more solutions. In summary, we **extract a maximum of 7 possible solutions** from a given solution.
- We could also take a combination of rotation and flipping to obtain more solutions but chose not to do it.

Part I adaptation to Part II

1. No degree heuristic : In the n-queens problem we no longer make use of the degree heuristic. This is because in the case of a variable(variable being a queen) assignment tie, we are unable to choose a queen/variable that is involved in the least number of constraints with other unassigned variables/queens. Every queen occurs in an equal number of constraints with other variables/queens. Hence, we made changes to our code such that the algorithm picks any random queen/variable for assignment during a tie.
2. MAC implementation: Earlier our program implemented the AC3 algorithm only as a measure of preprocessing. However, this time we implemented AC3 during program runtime, additionally, in the form of the Maintaining Arc Consistency (M.A.C) algorithm to reduce compute time.
3. Constraint modification : While part I of the project sought to solve a general CSP, such as assigning processors to processes, our goal this time was different, assigning positions to queens on a chess board. To accommodate this change, we modified existing constraints and made them specific to our application problem by introducing row, column and diagonal constraints for a given queen/variable.

Pseudo Code

Function BACKTRACKING*(assignments, unassigned)*

if *unassigned* is {empty} then return *assignments*

var = SELECT_UNASSIGNED_VARIABLE*(unassigned)*

for each *value* in ORDER_DOMAIN_VALUES(*var, assignments, unassigned*) do

    if *value* is consistent with *assignments* given *constraints,* then

        add{*var*=*value*} to *assignments*

        assignments = AC3-consistency*(assignments, unassigned)*

        result = BACKTRACKING*(assignments, unassigned)*

    if *result* ≠ failure then return *result*

    remove {*var* = *value*} from *assignments*

return *failure*

Function SELECT_UNASSIGNED_VARIABLE(unassigned)

mrv_list = {}

for each *var* in *unassigned* do

    add {permissible_values*(var)}* to *mrv_list*

var = argmin(*mrv_list)*

return *var*

Function ORDER_DOMAIN_VALUES (*var, assignments, unassigned*)

lcv_list = {}

for each *val* in permissible_values(*var*) do

add {*var*=*val*} to *assignments*

assignments = AC3-consistency(*assignments, unassigned*)

n_c = number_of_constraints(*assignments, var*)

add {n_c} to lcv_list

remove {*var*=*val*} from *assignments*

lcv_list = arg(sort(*lcv_list*, key=ascending))

return *lcv_list*

Function AC3_CONSISTENCY(assignments, unassigned)

queue = create_arc_queue(*unassigned, assignments*)

while *queue* is not empty do

(*Xi, Xj*) = remove-first(*queue*)

updated_assignment, removed = REMOVE-INCONSISTENT-VALUES(*Xi, Xj*)

If *removed* == TRUE then

for each *Xk* in neighbors(*Xi*) do

add *(Xk, Xi*) to queue

return *updated_assignment*

Function REMOVE-INCONSISTENT-VALUES (Xi, Xj)

removed = FALSE

for each x in domain(Xi) do

if no value y in domain(Xj) allows (x, y) to satisfy the constraint (Xi, Xj)

then delete x from domain(Xi); *removed* = TRUE

return *removed*

Compiling and Running the Program

The main code (written in python2) for our chosen problem, the ‘n queens’ problem, is in submitted file “csp_solver_nqueens.py”. Also note that a function file “plot_nqueens.py” is imported by the main program file and is used to create a plot for the arrangement of queen pieces onto the chessboard. The plot depicts the chess board empty spaces in ‘purple’ color and the queen piece containing squares in ‘yellow’ color. The “plots” folder contains plots for varying number of queens.

To run the program, perform the following steps:

1. Open a new command terminal
2. Navigate to the project directory
3. Run the following command:
    a. python csp_solver_nqueens.py <n_queens>
    b. For example, python csp_solver_nqueens.py 15
4. By default, n_queens = 8,
    a. python csp_solver_nqueens.py

Program Output:

Number of Queens not specified!
Taking N=8 by default.

Current Assignment
{Empty State}
Current Assignment
0 (0, 0)

Current Assignment
0 (0, 0)
1 (7, 1)

Current Assignment
0 (0, 0)
1 (7, 1)

2 (4, 2)

Current Assignment

0 (0, 0)

1 (7, 1)

2 (3, 2)

Current Assignment

0 (0, 0)

1 (6, 1)

Current Assignment

0 (0, 0)

1 (6, 1)

2 (4, 2)

Current Assignment

0 (0, 0)

1 (6, 1)

2 (4, 2)

3 (7, 3)

Current Assignment

0 (0, 0)

1 (6, 1)

2 (4, 2)

3 (7, 3)

4 (1, 4)

Current Assignment

0 (0, 0)

1 (6, 1)

2 (4, 2)

3 (7, 3)

4 (1, 4)

5 (3, 5)

Current Assignment

0 (0, 0)

1 (6, 1)

2 (4, 2)

3 (7, 3)

4 (1, 4)

5 (3, 5)

6 (5, 6)

Current Assignment

0 (0, 0)

1 (6, 1)

2 (4, 2)

3 (7, 3)

4 (1, 4)

5 (3, 5)

6 (5, 6)

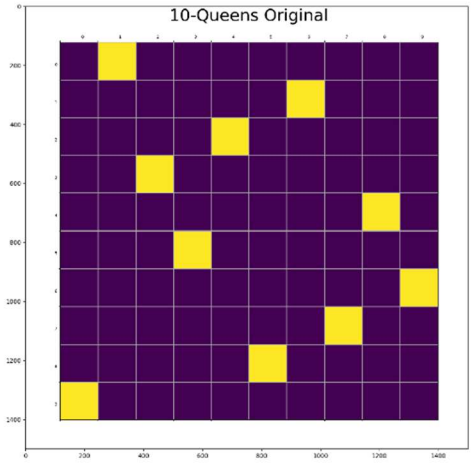
7 (2, 7)

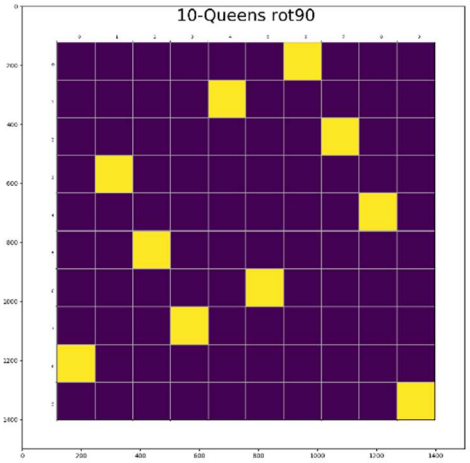
CSP Assignment Success!

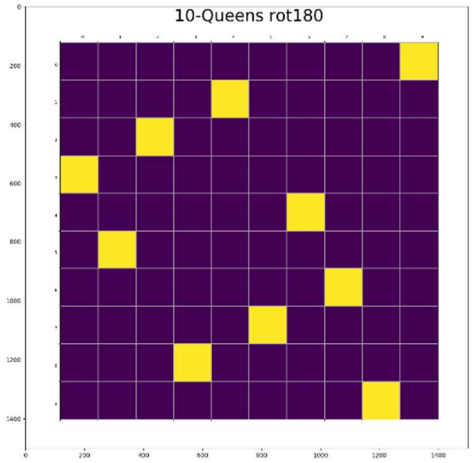
Time Taken (sec): 0.0110189914703

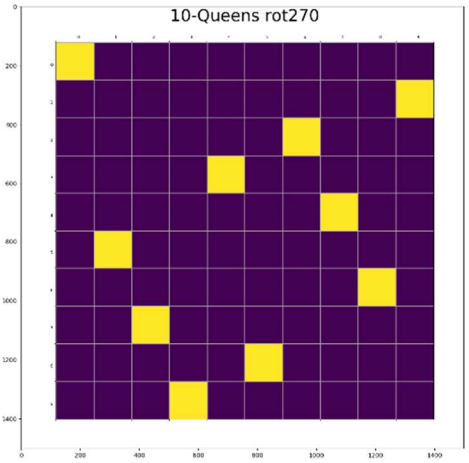
Outputs

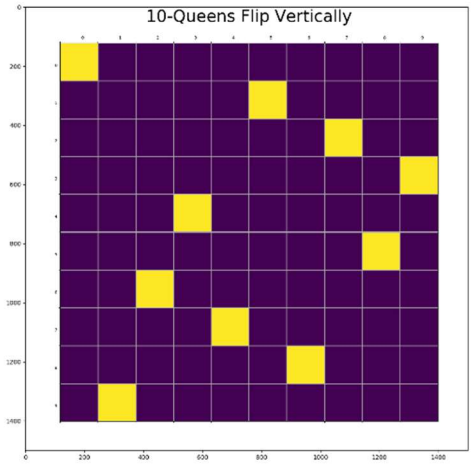
N = 10

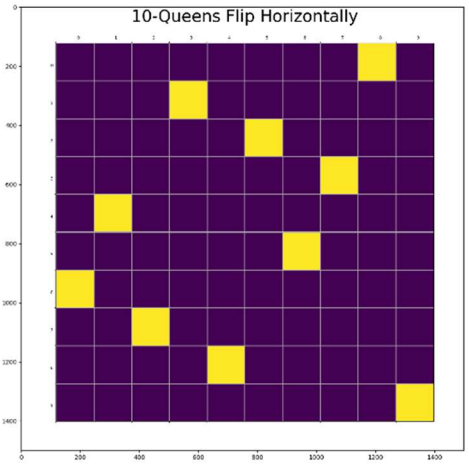

10-Queens Original


10-Queens rot90

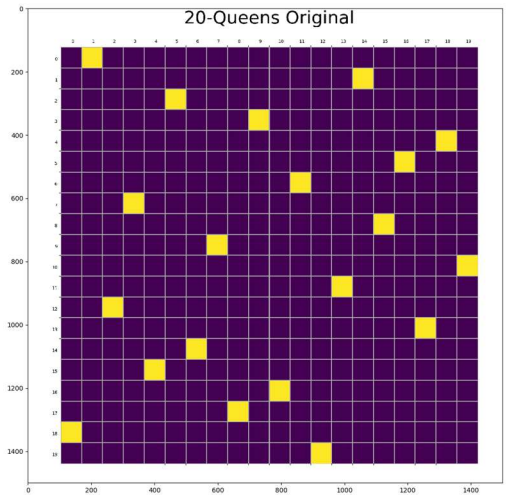

10-Queens rot180

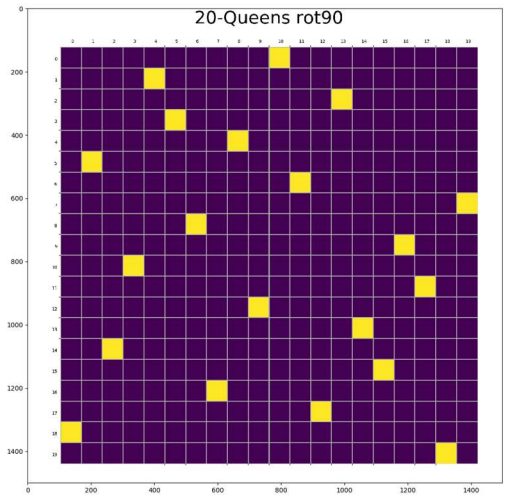

10-Queens rot270

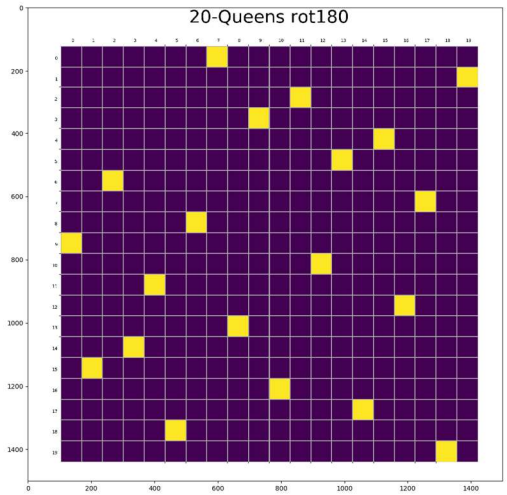

10-Queens Flip Vertically

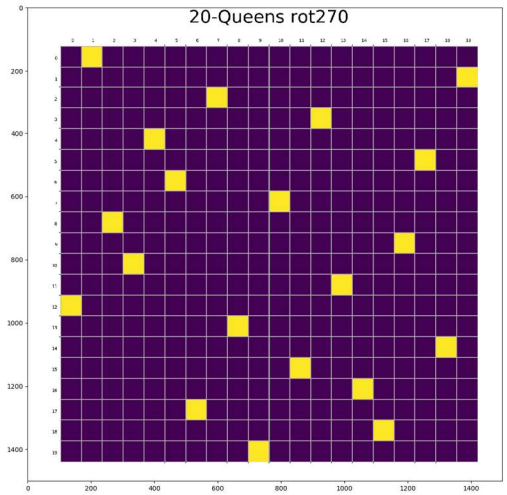

10-Queens Flip Horizontally

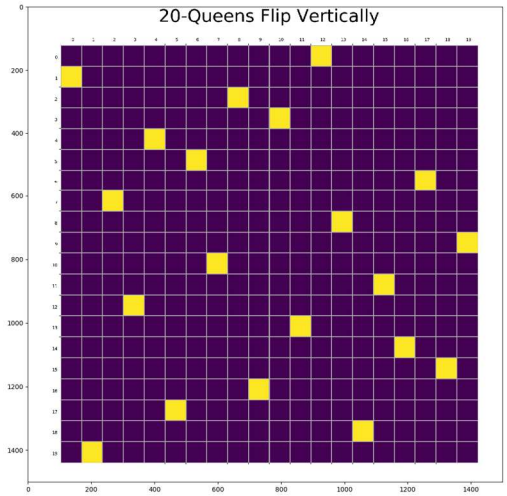
N = 20

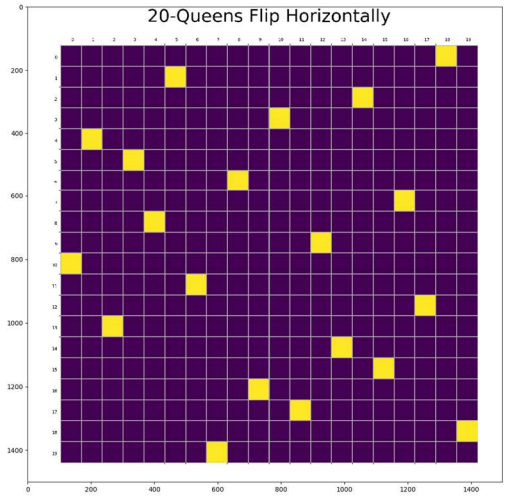

20-Queens Original


20-Queens rot90

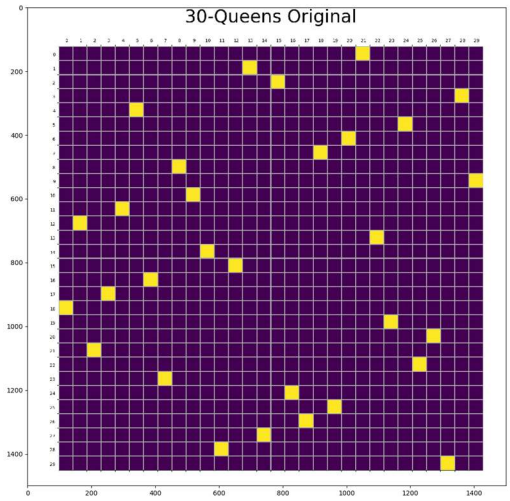

20-Queens rot180

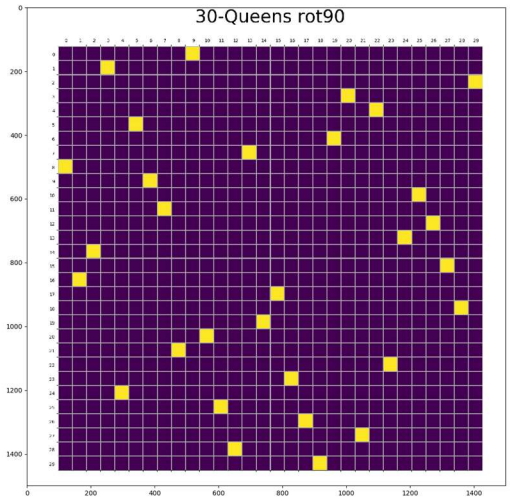

20-Queens rot270

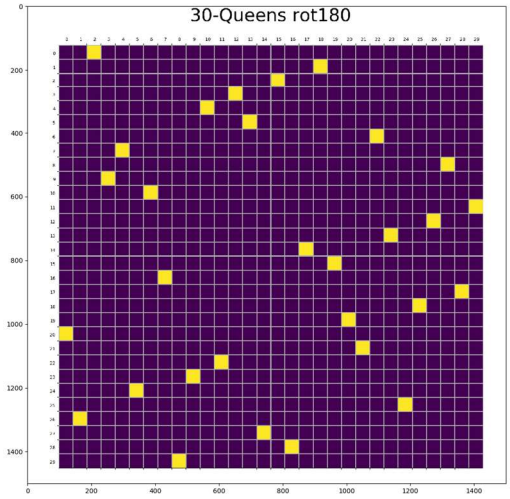

20-Queens Flip Vertically

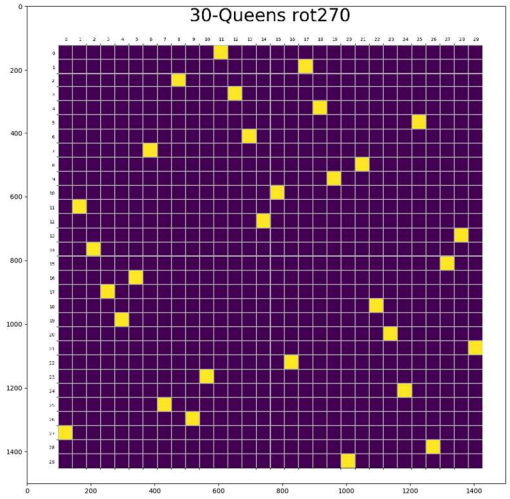

20-Queens Flip Horizontally

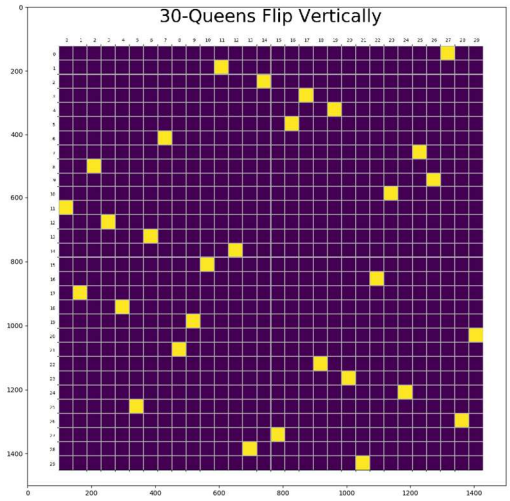
N = 30

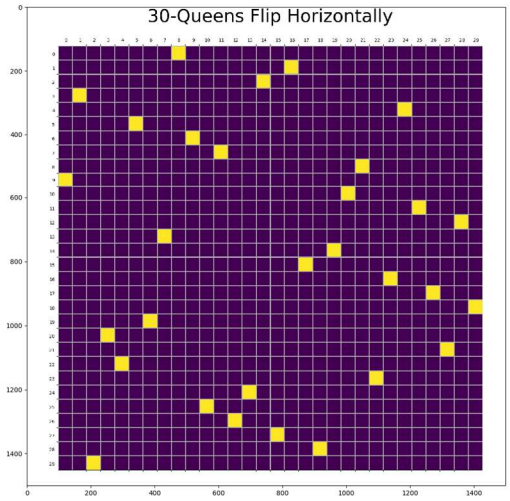

30-Queens Original


30-Queens rot90

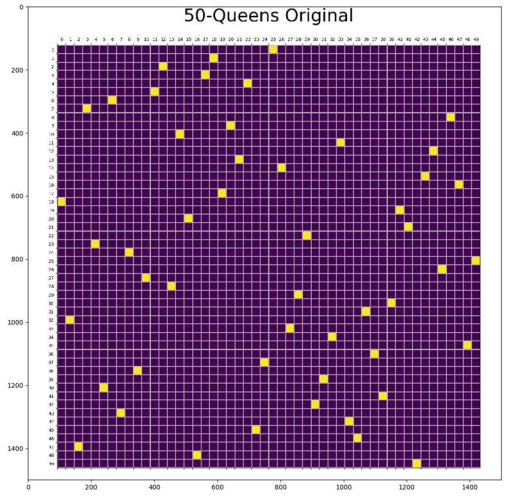

30-Queens rot180

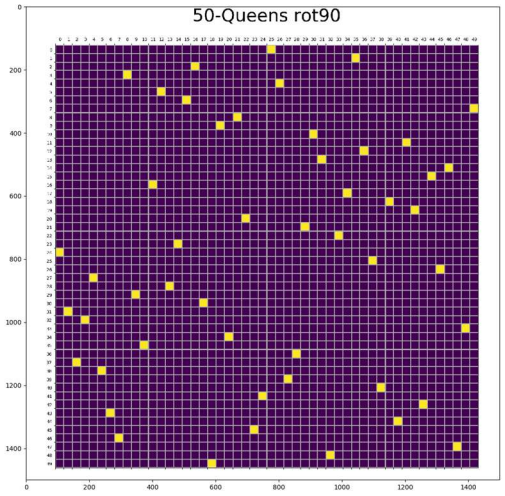

30-Queens rot270

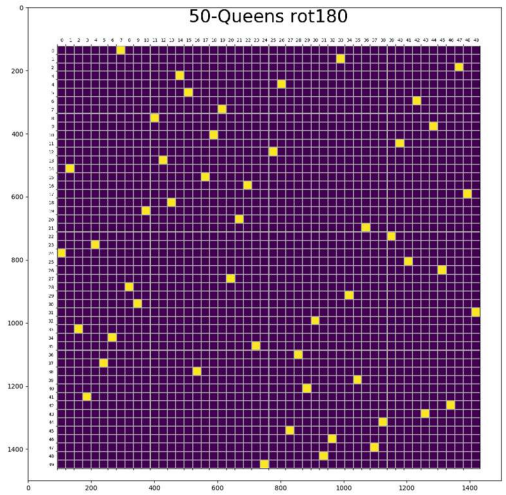

30-Queens Flip Vertically

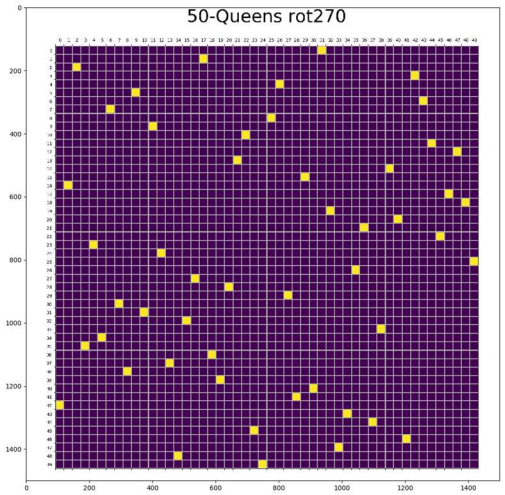

30-Queens Flip Horizontally

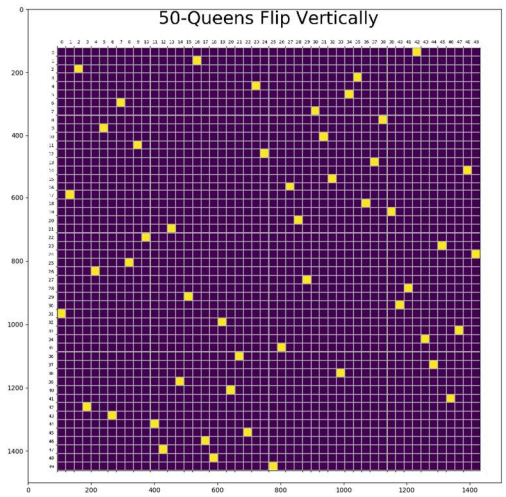
N = 50

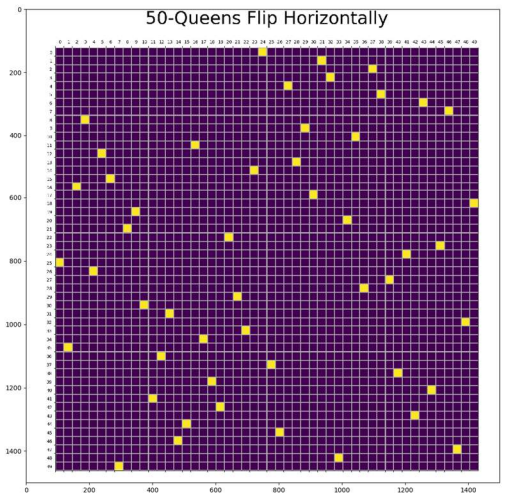

50-Queens Original


50-Queens rot90


50-Queens rot180

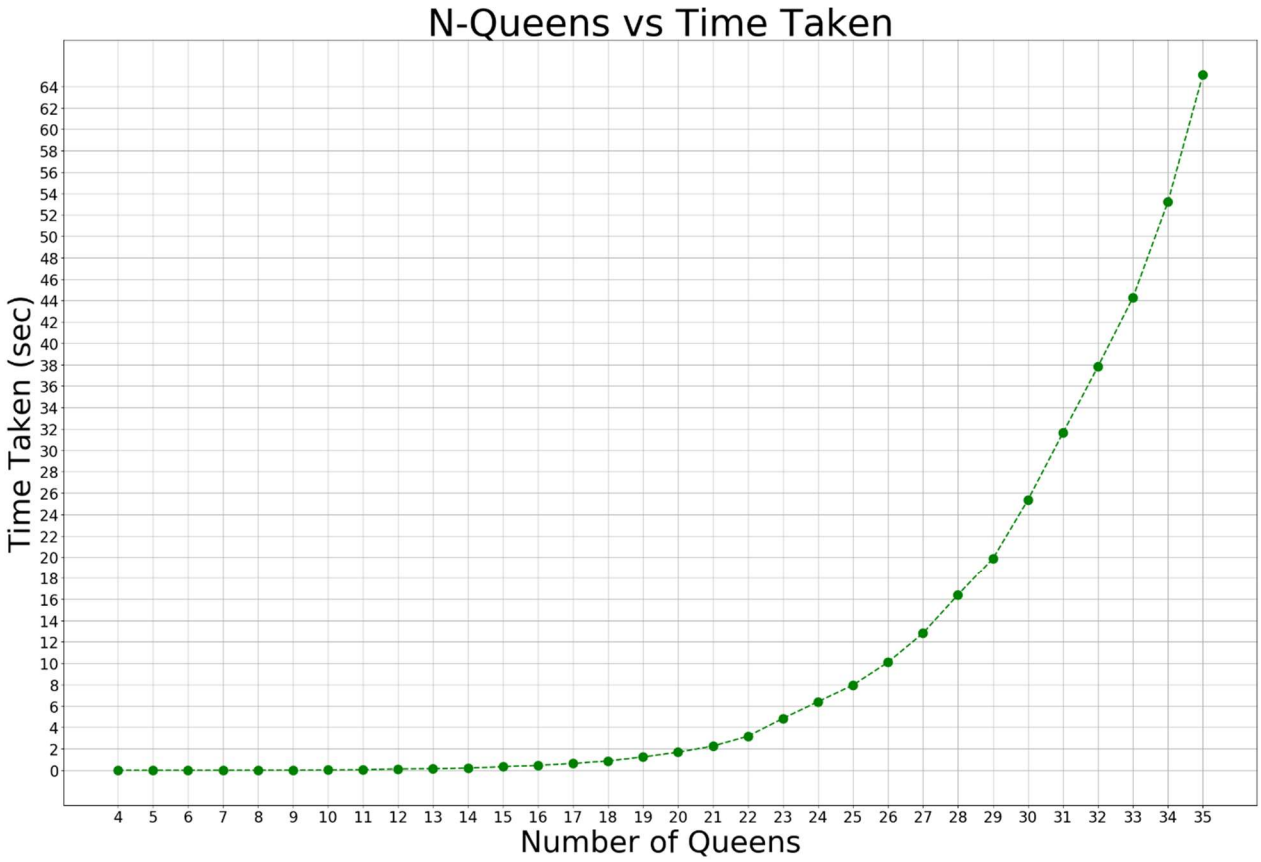

50-Queens rot270


50-Queens Flip Vertically


50-Queens Flip Horizontally

Time Complexity

S. No	No. of queens	Time taken (secs)
1	5	0.001
2	10	0.034
3	15	0.365
4	20	1.830
5	25	6.529
6	30	24.78
7	35	65.01
8	40	154.20
9	45	318.83
10	50	636.02



- We found through our results that we were able to come up with one possible solution using our algorithm, in under 11 mins for 50 queens.
- We found that looking only for one possible arrangement of queens on a given board rather than counting all possible number of arrangements, significantly reduced compute time.
- From the graph above, we observe that the time taken to solve the problem is exponentially proportional to the number of queens.

Program Strengths and Weaknesses

Strengths:

- Always produces a solution for N>= 4
- Can solve a large ‘n-queens’ problem in a significantly lower amount of time.
- Uses AC-3 and MAC algorithms to reduce variable domains and help reduce computation time.
- Uses the symmetric nature of the problem to come up with more than one solution.

Weakness:

- Does not tell us the total number of possible solutions.