Constraint Satisfaction on N-Queens

Eckel, TJHSST AI1, Fall 2019

# Background & Explanation

A constraint satisfaction problem is a problem that can be stated as a series of variables and a series of relationships between them (equations or constraints) that cause certain choices for one variable to block particular choices of another. Think of a Sudoku puzzle – placing a 1 in a space prevents any other space in that row, column, or block from also being a 1.

One way to solve a constraint satisfaction problem is through **simple backtracking**. This is the recursive process of taking a state, choosing an undetermined variable, then trying each available value for that variable with a recursive call. If no value is possible for a given undetermined variable, return something that indicates failure, and if each value of a variable returns failure, return failure in turn. If success is achieved, return the final state up the recursive stack. The pseudocode looks like this:

def csp\_backtracking(state):

if goal\_test(state): return state

var = get\_next\_unassigned\_var(state)

for val in get\_sorted\_values(state, var):

create new\_state by assigning val to var

result = csp\_backtracking(new\_state)

if result is not None:

return result

return None

The functions get\_next\_unassigned\_var and get\_sorted\_values are where all the cleverness in this process lives. Choosing the right heuristics to determine the best variable to try next, and then choosing the right heuristics to determine the best value of that variable to try next, can produce orders of magnitude of difference in runtimes (much like the differences between search algorithms in our first two units).

A second way to solve a constraint satisfaction problem is by creating an initial state, finding how many conflicts each current choice has, and iteratively moving the most conflicted choice to its least conflicted option until you achieve a successful solution. This is called **incremental repair**.

# Part 1: N-Queens with Simple Backtracking in Row-Major Order

The N-Queens problem is famous. Given an n x n chessboard, place n queens so that none of them can attack any others. We can state this as a CSP problem by making each row a variable, and the location of the queen in that row the value. The most trivial implementation of CSP, then, would have get\_next\_unassigned\_var just pick the first row where a queen hasn’t been placed, and get\_sorted\_values would simply produce, in ascending order, a list of all available spaces in that row.

Implement this basic form of the N-Queens CSP problem, using the pseudocode above and writing your own methods for get\_next\_unassigned\_var and get\_sorted\_values following this basic guidance. It should be able to solve the standard 8x8 version almost instantly, and should solve board sizes in the low 20s in a manageable number of seconds.

Using an algorithm that processes each row from 0 to the end in order and chooses the first available value for each row from 0 to the end in order as well, the first solutions you should arrive at for 8x8, 9x9, and 10x10 are:

[0, 4, 7, 5, 2, 6, 1, 3]  
[0, 2, 5, 7, 1, 3, 8, 6, 4]  
[0, 2, 5, 7, 9, 4, 8, 1, 3, 6]

# Part 2: Improve Your Backtracking Algorithm

As it turns out, this choice of algorithm – row major order – is very nearly the worst possible way to solve NQueens. Go back to those get\_next\_unassigned\_var and get\_sorted\_values functions and find something that makes the problem solve more quickly.

The most important improvements here will be in the choices of next variable and next value. Should you pick spaces that add the most new constraints to the board? The least? Closer to the center? Closer to the edges? Randomly? Something more subtle? You can also improve how you track your information. Should you add some form of meta-data to the state that gets passed along with the recursive call?

As you work with this problem and get larger and larger answers, use this code to test if they are valid:

**def** test\_solution(state):  
 **for** var **in** range(len(state)):  
 left = state[var]  
 middle = state[var]  
 right = state[var]  
 **for** compare **in** range(var + 1, len(state)):  
 left -= 1  
 right += 1  
 **if** state[compare] == middle:  
 print(var, **"middle"**, compare)  
 **return False  
 if** left >= 0 **and** state[compare] == left:  
 print(var, **"left"**, compare)  
 **return False  
 if** right < len(state) **and** state[compare] == right:  
 print(var, **"right"**, compare)  
 **return False  
 return True**

This will also allow me to verify that your solutions are correct, so please copy/paste it into your code.

Your goal here is to make an implementation of simple backtracking that will solve an NQueens board of size greater than 100 in less than 5 seconds.

An important note: you get to pick the board size. Your algorithm doesn’t have to work with *every* board greater than 100, just at least one of them. This is important because some algorithms get weirdly stuck on certain numbers. Last year, one student could solve every board except the ones that were 4 more than a multiple of 6. I have an algorithm that is incredibly fast except on the specific numbers 106 and 108, for some reason. This kind of thing is normal and perfectly fine.

To get this signature, show me a run of your code that does all of the following:

* Generate a solution from scratch for a board of size greater than 100.
* Verify the solution with a call to the test\_solution function above.
* Run time is less than 5 seconds (show me this using time.perf\_counter()).
* I will also want to look at your code to verify the call to the test\_solution function. Make sure your code is readable enough that I can do so easily.

Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Part 3: N-Queens with Incremental Repair

Next, we want to try the incremental repair algorithm. Generate an initial board state – randomly, perhaps, or using some kind of greedy algorithm – and then fix it incrementally until the board is solved.

It’s fairly likely that as you work through this you’ll run into problems with infinite loops, code choosing the same variable and then moving it to the same place over and over. You might want to look into the choice function in the random library to help add some randomness to your algorithm so it’s less likely to go in circles. I should also point out that smaller boards are more likely to go in circles. If you find yourself getting caught in infinite loops on boards with size under 30, try some larger ones and the algorithm may work better.

Once again, your goal here is to make an implementation of incremental repair that will solve an NQueens board of size greater than 100 in less than 5 seconds.

To get this signature, show me a run of your code that does all of the following:

* Generate an initial, flawed state of size greater than 100.
* Output the number of conflicts in the initial state.
* Incrementally repair the initial state. (If possible, I’d love to see a tracker for the number of conflicts that slowly decreases to zero.)
* Once found, verify the solution with a call to the test\_solution function in part 2.
* Run time is less than 5 seconds (show me this using time.perf\_counter()).
* I will also want to look at your code to verify the call to the test\_solution function. Make sure your code is readable enough that I can do so easily.

Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Specification

This assignment is **complete** if:

* You have received both signatures above.

For **resubmission**:

* Complete the specification correctly.

# Specification for Outstanding Work: NQueens Optimization

Outstanding work credit is available for optimizing this assignment beautifully. You will show me this on your own computer. You can use any algorithm you like – simple backtracking, incremental repair, or something else entirely.

This extension **receives outstanding work credit** if your code does all of the following:

* Solve every NQueens board from size 8 to size 200 in less than 30 seconds. Show me this by getting an initial time.perf\_counter() value, then making a while loop that continues to solve puzzles as long as the current time minus the start time is less than 30 seconds.
* Store every solution in a list as they are found.
* After the 30 seconds have passed, loop over the solution list and verify all of them with my code above (or present me with a very convincing argument that this is unnecessary).