Constraint Satisfaction on N-Queens

Eckel, TJHSST AI1, Fall 2023

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 Sudoku – placing a 1 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 unit).

In addition, aside from a recursive backtracking search, 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**. You'll get one GREEN credit for implementing each of the algorithms, for a total of two.

The N-Queens Problem

The N-Queens problem is famous. Given an *n* x *n* chessboard, place *n* queens so that none of them can attack any others (in other words, no two queens appear on the same row, column, or diagonal line.)

There's an intuitive way to state this as a CSP problem that doesn't actually work very well; you might think we should make each space a variable with a value of either "queen" or "not queen". This will work, but it's slow.

Instead, we can encode an important feature about the problem – that there is only one queen in each row – into our representation, tremendously limiting the search space before we even begin. We do this by making each **row** a variable (please note: **not** each **space**, each **row**), and value of that variable is the location of the queen **in that row**.

So, for example, this single list of ints would represent an **entire solved 2d board**: `[0, 4, 7, 5, 2, 6, 1, 3]`

This says that the queen in row 0 is in column 0, the queen in row 1 is in column 4, the queen in row 2 is in column 7, etc.

Warm Up: N-Queens with Simple Backtracking in Row-Major Order

With the model on the previous page established, 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 that aren't attacking any previously placed queens.

I **strongly** recommend that you do **not** use a value of -1 to represent a row without a queen present. Many students default to this, but it has a frequent error. It's very easy to write code that checks diagonal attacks that makes perfect sense intuitively, but that thinks that a queen in column 0 is attacking an unplaced queen ("at -1") in the next row down. Instead, use None or use, like, -10000000, some huge negative number.

Then, `get_next_unassigned_var` is just looking for the first None, or the first -10000000, in your list. The `get_sorted_values` function is the only one with any of the problem's real logic in it – that function should only return a list of spaces where a queen could legally be placed, based on the previously placed queens. It is possible, of course, that this returns an empty list – note that the algorithm pseudocode, in this case, will just skip the for loop and backtrack immediately.

Go ahead and 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]
```

Then, **check in with me in person or send me a message on Mattermost** verifying those three outputs on your code.

Once you're sure you have this part right, move on to the next page.

GREEN Credit #1: Improve Your Backtracking Algorithm

As it turns out, this choice of algorithm – row major order – is very nearly the worst possible way to solve N-Queens. 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 a couple of N-Queens boards of size greater than 30 in less than 5 seconds.

An important note: you get to pick the board sizes. Your algorithm doesn't have to work with *every* board greater than 30, just at least two 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 on this assignment. N-Queens is a surprisingly weird problem.

Before moving on, make sure your code can do all of the following:

- Generate a solution from scratch for at least two different board sizes greater than 30.
- Verify each solution with a call to the `test_solution` function above.
- Run time is less than 5 seconds for generating each solution; verify using `time.perf_counter()`.

Get Your Code Ready to Turn In

When you turn in your code, I want to be able to see that simple backtracking is working. I'll run your code (no command line arguments, you pick the board sizes) and verify the output, then I'll open your code and look at the bottom. Please end your file with lines of code that clearly do the following, in this order:

1. Store a `time.perf_counter()` call
2. Generate a solved state of size > 30 using simple backtracking; print it (as a list of integers like on page 1)
3. Verify it using the `test_solution` function in part 2
4. Repeat steps 2 and 3 on a different size > 30
5. Use a final `time.perf_counter()` call to print the time this all took; the final number shouldn't be more than 20 seconds

Specification

Submit a single Python script to the link on the course website.

This assignment is **complete** if:

- The "Name" field on the Dropbox submission form contains your **class period**, then your **last name**, then your **first name**, in that order.
- Your code does everything specified in the "Get Your Code Ready to Turn In" section, in that order.
- Your code generates the solutions from scratch; ie, you don't have some kind of pre-stored or excessively specific algorithm that works for only the sizes you chose. If you're not sure, please ask!

Then, continue on to implementing Incremental Repair on the next page.

GREEN Credit #2: N-Queens with Incremental Repair

Next, we want to try the incremental repair algorithm. Generate an initial board state and then fix it incrementally until the board is solved. I don't recommend generating the original state randomly; try to avoid collisions where possible, so you're starting in a good place, just without any backtracking. Then, fix the remaining problems.

The basic idea here is to take the current flawed state, find the variable causing the most problems, and change it to the value that causes the fewest problems. Repeat until there are no more problems. Some subtleties here:

- This only tends to be a good strategy in situations where the solution space is pretty dense, ie a situation where there are many ways to solve the problem. A sudoku with one solution is a bad application, but N-Queens has lots of solutions at each size (proportionally more as the size increases, in fact!) Thus, this is a pretty solid idea here.
- 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. I strongly recommend looking 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 a couple of N-Queens board of size greater than 30 in less than 5 seconds each.

Before moving on, make sure your code can do all of the following:

- Generate an initial, flawed state of size greater than 30, and output the number of conflicts in the initial state.
- Incrementally repair the initial state, showing the number of conflicts after each pass decreasing to zero. One possible specific approach here - find the queen on the board that is currently attacking the largest number of other queens. If there's a tie, randomly pick from among those *with the highest number*. Find the space on that row that attacks the fewest number of other queens. If there's a tie, randomly pick from among those *with the lowest number*.
- Once found, verify the solution with a call to the `test_solution` function in part 2.
- Run time is less than 5 seconds; verify using `time.perf_counter()`.
- Repeat all of the above with a different size greater than 30.

As you run your algorithm, **print the state and the number of conflicts after each modification**. You'll know you have this right when **the number of conflicts never goes up – it only remains the same or decreases at each step.**

Get Your Code Ready to Turn In

When you turn in your code, I want to be able to see that incremental repair is working. I'll run your code (no command line arguments, you pick the board sizes) and verify the output, then I'll open your code and look at the bottom. Please end your file with lines of code that clearly do the following, in this order:

1. Store a `time.perf_counter()` call
2. Generate a flawed state of size > 30; print the state (as a list of integers) and the number of conflicts
3. Incrementally repair that flawed state, **printing the state and number of conflicts after each modification** (so, you should have **many** lines of output)
4. Once the number of conflicts reaches 0, verify this solution again using the `test_solution` function in part 2
5. Repeat steps 2-4 on a different size > 30
6. Use a final `time.perf_counter()` call to print the time this all took; the final number shouldn't be more than 20 seconds

Specification

Submit a single Python script to the link on the course website.

This assignment is **complete** if:

- You follow the instructions on the submission form to format your submission properly.
- Your code does everything specified in the “Get Your Code Ready to Turn In” section, in that order.
- Your code generates the solutions from scratch; ie, you don’t have some kind of pre-stored or excessively specific algorithm that works for only the sizes you chose. If you’re not sure, please ask!