Arbitrary Constraint Satisfaction Problem-Solving

Ben Williams '25, October 2023

Map Coloring and CSP Basics

In our ConstraintSatisfactionProblem's (CSP) initialization, we take the variables, domains, and constraints as parameters to store in the object. Other than also initializing a `total_search_calls = 0`, there is nothing else in the base CSP setup.

The MapColoringProblem (MCP) extends the ConstraintSatisfactionProblem class. However, in the MCP's initialization, we take a `map_file` and `num_colors` as parameters. The `map_file` must have the province and neighbors in the following format:

```
province_name; neighbor_name_1, neighbor_name_2, ... , neighbor_name_n
```

We use a helper function in order to parse the file. Each province represents a variable, and the neighbors help us build constraints. We use `num_colors` in order to help define the constraints. Every province/variable's domain consists of all possible colors, and the constraints consist of all possible non-duplicate pairs between neighbors.

For instance, for provinces 1 and 2 that neighbor eachother, and colors 0, 1, and 2. This is the constraint between 1 and 2 (and (2, 1) respectively):

```
States                   Constraints
(1, 2) : {(0, 1), (0, 2), (1, 0), (1, 2), (2, 0), (2, 1)}
```

Brute Force : Solving the Map Coloring Problem

Using the Australia map as an example, we can illustrate the least efficient method for solving a CSP, but at least show that the variables, domains, and constraints were defined properly.

In our brute force solver, we try every possible assignment looping through each of the variables' possible domain values. We do this by recursively iterating on each of the variables.

So, for the Australia map coloring problem, we start with the following assignment:

```
[0, 0, 0, 0, 0, 0, 0]
```

We eventually reach a valid solution. The search, however, is extremely slow:

```
[0, 1, 2, 0, 1, 0, 0]
627 total brute force search calls
```

Basic Backtracking Solver

The backtracking solver does not require any parameters to call initially. It has optional parameters for the `assignment`, `domains`, `inference`, `select_variable`, and `order_domain`. The `assignment` and `domains` are used in the recursive calls during the backtracking, but the other three are completely optional function parameters that allow you to pass in functions to run inference, variable selecting, and domain selecting that we will go over later.

We initialize the assignment to a list whose length is the same as the number of variables, and initialize each entry to None. We initialize `domains` to be a copy of the `self.domains`. Since the `domains` might be edited by the inference algorithm, we do not want to lose the original domains.

Next, we set `variable` to be a variable with a currently unassigned value, and loop through every value in its domain. We first check if setting each value violates any constraints within the current assignment, and if not, assign the varible to that value, and recurse on the backtracking function - passing in our current assignment and domains as the `assignment` and `domains` parameters.

If we have tried every possible value in the `variable`'s domain, then we return None, which forces the algorithm to backtrack. If we do find a successful assignment, we just return the assignment.

Either we will always fail to assign a value and find that there is no solution, or we will eventually have assigned values to all the variables. If all the values are assigned (and therefore we are not able to select our `variable`), we return the current assignment as our base case.

For the Australia map coloring, the backtracking algorithm (even with none of the heuristics enabled) is already a massive improvement from the brute force algorithm. It finds the same solution very quickly:

```
[0, 1, 2, 0, 1, 0, 0]
8 Total backtracking search calls
```

Circuit Board Problem

Initialization

The CircuitBoardProblem is a subclass of the normal ConstraintSatisfactionProblem described before. In its initialization, we take two integers `board_width`, `board_height` and a list of pairs/tuples `components` as parameters. With this, we can define the `variables`, `domains`, and `constraints` necessary for the CSP.

Each variable will represent a component, where its domain is every possible position we can place the component on the circuit board without any of it going off the edge. Components are in the form `(width, height)`.

We use a helper function `get_component_pair_constraints` to get the constraints for any two component pairs. In this helper function, we return a list of all the possible locations (treated as a 1D array of length `board_width * board_height`) where we can place each component without either of them going off the edge or overlapping each other.

Discussion

For the following list of components in a 10*3 board:

```
[(3, 2), (5, 2), (2, 3), (7, 1)]
```

The backtracking algorithm finds a solution quickly:

```
AAABBBBBCC
AAABBBBBCC
DDDDDDD.CC
5 total search calls
```

Describe the domain of a variable corresponding to a component of width w and height h, on a circuit board of width n and height m

The domain of this component would be found as follows (in pseudocode based off of `find_component_domains`:

```
domain = []
curr_row = 0
for location in {0, 1, ...,  n * m - 1}:
    if floor(location / n) > curr_row:
        curr_row += 1
    if location - (curr_row * n) + w <= n:
        if h + curr_row <= m:
            domain.append(location)
return domain
```

We could also use set notation to define the domain:

```
Domain = { l < n*m | (l + w - (floor(l / n) * m)) <= n
            and floor(l / n) + h <= m }
```

Consider components A and B above, on the 10x3 board. Write the constraint that enforces the fact that the two components may not overlap. Write out legal pairs of locations explicitly

The legal pairs are as follows:

```
{(17, 0), (5, 10), (17, 12), (0, 5), (10, 3), (11, 5),
(2, 5), (0, 14), (11, 14), (16, 1), (10, 15), (7, 1),
(1, 15), (16, 10), (6, 11), (7, 10), (5, 0), (17, 2),
(12, 15), (17, 11), (11, 4), (10, 5), (0, 4),
(16, 0), (1, 5), (10, 14), (0, 13), (6, 1), (7, 0), (1, 14),
(15, 10), (7, 12), (6, 10), (12, 5), (17, 1), (17, 10),
(10, 4), (0, 3), (1, 4), (10, 13), (15, 0), (0, 15),
```

```
(11, 15), (7, 2), (6, 0), (2, 15), (7, 11), (16, 11)}
```

Where the first value in each pair is somewhere where we can place A, and the second is somewhere we can place B. Note that in the illustration, the top-left corner represents location 0, and the bottom right represents location 63. We are considering the locations of the top-left part of each component.

In the illustrated example, we have A placed at location 0, and B at location 4. This is a valid pair, and can be found at the end of the fourth row of pairs in the above list.

Describe how your code converts constraints, etc, to integer values for use by the generic CSP solver

In general, to convert an x, y coordinate to the integer values used to describe board locations, we multiply the y value by the `board_width` and add the x value to that. This was used to find the domains shown above.

To generate the constraints, we use a helper function `get_component_pair_constraints` which takes two variables/components as parameters. We loop through the domain of the first component and "place" the component there (call this p1) by having a set of all the board locations it would take up. Then, we loop through and "place" the second component at the locations of its domain (call this p2). If any of these locations taken up by the second component overlap with those taken up by the first, then (p1, p2) is not added to the constraints set. Otherwise, if there is no overlap, (p1, p2) is added to the constraints set.

Heuristics

As mentioned before, the backtracking solver has optional parameters for `inference`, `select_variable`, and `order_domain`. We can pass in functions for these parameters to try and improve the performance of the backtracking algorithm. We will briefly describe the following heuristics, and then how much they help lessen the search.

Inference - MAC

MAC is a modified version of AC3. It takes a `variable`, its assigned `value`, the `assignment`, and the `domains` as parameters. It loops through all the **unassigned** neighbors of the variable, and finds values in their domains that are not valid after we assign the `value` to `variable`. We keep a list of lists of these values to remove from the neighbor's domains. We return this list of lists to be removed, condensing the domains of all the neighbor variables. If any of the neighbor variables is left with an empty domain, we report that the inference has failed, and that the `value` assigned cannot work.

Variable Selection - Minimum Remaining Values

We want to select a variable that has the fewest choices left possible. So we loop through all the domains and return the variable that has the smallest domain. This pairs best with inference, as the domains change over time.

Domain Ordering - Least Constraining Values

Within a domain, we want to first pick the value that causes the least conflicts and is most likely to succeed. In the backtracker, we are looping through all possible values in the domain. So, we sort the domain from the least constraining value to the most constraining values, and return the sorted list.

Results

We use a "medium" test to compare how basic backtracking works, and then progressively add in heuristics to help decrease the number of search calls. This medium test consists of a 15x5 board, and the following components:

```
[(5, 5), (3, 3), (2, 2), (1, 4), (4, 1),
(2, 2), (1, 4), (4, 1), (6, 1), (4, 2)]
```

Here are the results:

```
Testing backtracking
AAAAABBBCCFF.DG
AAAAABBBCCFF.DG
AAAAABBBEEEE.DG
AAAAAJJJJHHHHDG
AAAAAJJJJIIIIII
7602 total search calls
----------------

Testing backtracking with inference
AAAAABBBCCFF.GD
AAAAABBBCCFF.GD
AAAAABBB.JJJJGD
AAAAAHHHHJJJJGD
AAAAAEEEEIIIIII
3416 total search calls
----------------

Testing backtracking with inference and min-remaining-values
AAAAADGBBBJJJJ.
AAAAADGBBBJJJJ.
AAAAADGBBBEEEE.
AAAAADGHHHHCCFF
AAAAAIIIIIICCFF
```

```
23 total search calls
----------------

Testing backtracking with inference, min-remaining-values,
    and least-constraining-value
IIIIIIFFCCAAAAA
DGEEEEFFCCAAAAA
DGBBB.HHHHAAAAA
DGBBB.JJJJAAAAA
DGBBB.JJJJAAAAA
17 total search calls
```

We can see that the heuristics can greatly reduce the number of total search calls. Note that in this case, there re multiple different solutions, and each board looks a bit different.

We now show an example of how we can use the heuristics to help solve a board that only has one valid solution. This "hard" test consists of a 20x6 board and has the same components as the smaller 10x3 example **times 4**, and with an additional 2x2 component that must go in the center.

This board cannot be solved in a reasonable amount of time (hundreds of thousands of search calls +) unless we use all of our heuristics. It allows us to demonstrate just how helpful combining all the heuristics are:

```
Testing backtracking with inference, min-remaining-values,
    and least-constraining-value
OONNNNNMMMIIIJJJJJKK
OONNNNNMMMIIIJJJJJKK
OOPPPPPPPPQQLLLLLLLKK
GGHHHHHHHQQDDDDDDDCC
GGFFFFFEEEAAABBBBBCC
GGFFFFFEEEAAABBBBBCC
254 total search calls
```

It is able to find the solution incredibly quickly.

Local Search and the Circuit Board - Min Conflicts

The Min-Conflicts local search is very simple in principle. We begin by selecting a random but valid assignment. Then, we select a random conflicted variable, change its value to whatever violates the least constraints, and repeat until we end up with a solution or until we reach a pre-determined maximum number of iterations. We have various helper functions `violates_least_constraints` and `get_conflicted_variables` to help with this.

This would frequently solve the map-solving problem or the small 10x3 CBP fairly quickly. However, sometimes, the algorithm would get stuck with say two variables that conflicted with each other, and repeatedly select the same state as it had the least conflicts possible. One could think of this as a local minimum towards the solution, or a plateau. Without being pushed out of the hole or forced to walk a new path on the plateau, the algorithm would be permanently stuck.

But how do we define an arbitrary plateau search that works for any inheritor of the ConstraintSatisfactionProblem? Ideally, we would walk side-to-side to other states with the same "score", but this "score" and much less the method of finding an equal-score state are not possible to define easily.

There are a few possible solutions to this:

Just restart

The first and weakest idea was just to restart the Min Conflicts walk entirely after a certain number of iterations (say 1/10 of the max). Unfortunately, the number of local minima for our "medium" 15x5 board example is far greater than the number of solutions, so we still repeatedly get stuck.

Russel and Norvig suggest this approach in some cases, citing Leta et al. (1993).

A Recently-Visited List - Tabu Search

Using a small recently-visited list and not allowing the algorithm to revisit these states, or as Russel and Norvig call it, a **tabu search**, is another idea to try to solve the problem.

But if we don't allow the algorithm to revisit these states, which state do we choose instead? My first idea was to choose the next-best (or next-next-best, however long until we are outside the recently-visited list or until we have exhausted all possibilities.) least conflicting value, but we still end up stuck at local minima.

My next idea was to pick one variable at random (even if there are already 0 conflicts), and change its value to a random value in its domain, essentially making a random change anywhere in the assignment. This does allow it to break out of the local minima, but is often not much different than starting out again at ground 0.

For our medium example, the random-change method does *sometimes* work when we set the max-iterations to 100,000:

```
Num-iterations 34674
JJJJGD.BBBAAAAA
JJJJGD.BBBAAAAA
HHHHGD.BBBAAAAA
FFCCGDEEEEAAAAA
FFCCIIIIIIAAAAA
```

Though many of the processes here are faster than the backtracking algorithm, the number of iterations renders it inferior as it still takes up more time overall. In the following case, where we got quite lucky with the local search, we still spent nearly 6x as much time in the local-search:

```
Testing backtracking
(solution not illustrated, but found)
7602 total search calls
Time elapsed: 0:00:00.522585
----------------

Testing local search
Num-iterations 11992
(solution not illustrated, but found)
Time elapsed: 0:00:02.803531
```

The process of finding equivalent "score" states that would allow us to move along the plateau would take another search to find… ruining the point of a local search.

N-Queens Problem

Definition and Implementation

The N-Queens problem describes how we can place N queens on an NxN chess board without any of them threatening each other. This can be thought of as a CSP, where each variable represents a queen, the domains the possible locations we can put it, and the constraints where queens do not threaten each other.

We define the domains by a column per queen. Though one can imagine the domain of each queen being the entire NxN board, we can already subtly implement one of the rules by just limiting the domains, reducing our domain size by a factor of N.

We define the constraints queen-by-queen. We loop through all possible locations on the column the queen can be placed, and calculate which locations the queen would threaten if it were placed there. Then, we loop through all the board locations - if it is not threatened by the queen, then add the (location, other location), whose corresponding queen is based on the column of the other location, to the constraints.

There is also an `illustrate_solution` function similar to the CBP.

Results and Local Search Discussion

While the circuit board may not have as many densely distributed solutions, which is needed for the local search to run well, the N-Queens does not have this issue. In fact, the N-Queens seems to perform well regardless of the number of queens, and actually succeeds more frequently when there are more queens.

We of course begin our testing by using our normal backtracking with all the inference, variable, and value heuristics tacked on.

```
Testing backtracking with heuristics on 4 queens:
. Q . .
. . . Q
Q . . .
. . Q .
9 Total search calls
----------------
Testing backtracking with heuristics on 8 queens:
. . Q . . . . .
. . . . . Q . .
. . . Q . . . .
. Q . . . . . .
. . . . . . . Q
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
129 Total search calls
----------------
Testing backtracking with heuristics on 16 queens:
. Q . . . . . . . . . . . . . .
. . . . Q . . . . . . . . . . .
. . . . . . . . . . Q . . . . .
. . . . . . . . Q . . . . . . .
. . . . . . . . . . . . . Q . .
. . . . . Q . . . . . . . . . .
. . . . . . . . . . . . Q . . .
. . . . . . Q . . . . . . . . .
. . . . . . . . . . . . . . . Q
. . . . . . . . . . . Q . . . .
. . . . . . . . . . . . . . Q .
. . . Q . . . . . . . . . . . .
. . . . . . . Q . . . . . . . .
. . . . . . . . . Q . . . . . .
. . Q . . . . . . . . . . . . .
Q . . . . . . . . . . . . . . .
15345 Total search calls
----------------
Testing backtracking with heuristics on 24 queens:
. Q . . . . . . . . . . . . . . . . . . . . . .
. . . Q . . . . . . . . . . . . . . . . . . . .
. . . . . Q . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . Q . . . . . . . . . .
. . . . . . . . . . Q . . . . . . . . . . . . .
```

```
. . . . . . . . . . . . . . Q . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . Q . .
. . . . . . . . . . . Q . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . Q .
. . . . . . . . Q . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . Q . . . . .
. . . . . . . . . . . . . . . . . . . . . . . Q
. . . . . . . . . . . . . . . . . . . Q . . . .
. . . . . . . . . Q . . . . . . . . . . . . . .
. . . . . . . Q . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . Q . . .
. . . . . . . . . . . . . . . . Q . . . . . . .
. . . . . . . . . . . . Q . . . . . . . . . . .
. . . . . . . . . . . . . . . Q . . . . . . . .
. . . . . . . . . . . . . . . . . Q . . . . . .
. . . . . . Q . . . . . . . . . . . . . . . . .
. . . . Q . . . . . . . . . . . . . . . . . . .
. . Q . . . . . . . . . . . . . . . . . . . . .
Q . . . . . . . . . . . . . . . . . . . . . . .
484194 Total search calls
```

While the results for the backtracking are nice and all, and show the scope of the problem when searching *smart*, the results for the local search are far more interesting. For each of the following statistics, we repeated the local search with `max_iters = 10000`:

```
For 4 Queens:
Failed 100% of the time
----------------
For 8 Queens:
Failure frequency: 67%, average iterations for success: 85.33333333333333
----------------
For 16 Queens:
Failure frequency: 16%, average iterations for success: 108.29761904761905
----------------
For 24 Queens:
Failure frequency: 4%, average iterations for success: 123.3125
```

Discussion of results

W e run into a common compromise here that we do with many randomized algorithms. This algorithm has a chance for failure, but the work performed is so small that we can just repeat the algorithm until we find success, especially for large problems.

The issue for the local search is again that it relies on being **densely distributed throughout the state space**. That is the case for the N-Queens problem, but not for all problems. That makes the local search the ideal way to solve large N-Queen problems.

Past 24 queens, it takes way too long for the backtracking algorithm to compute the N-Queens solution. Just for fun, here is the 64 queen solution found by the local search. It takes longer for the pre-computation of constraints than for the actual solving (does not fit right on pdf, check .md):

```
Testing min-conflicts on 64 queens:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Q . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . Q . . . . . . . . . . . . . . . . . .
. . . . . . . . . . Q . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . Q . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Q . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . Q . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Q . . . . . . .
. . . . . . . . . . . . Q . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . Q . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Q . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Q . . . . . . . .
. . . . . . . . . . . Q . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . Q . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Q . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . Q . . . . . . . . . . . . . . . . . . . .
. . . Q . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . Q . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Q .
. . . . . . . . . . . . . . . . . . . . Q . . . . . . . . . . . . . . . . . . . . . . . . .
Q . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . Q . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . Q . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Q . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Q . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . Q . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Q . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Q . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . Q . . . . . . . . . . . . . . . . . . . . . . . . . .
```

```
. . . . . . . . . . . . . . . . . Q . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . Q . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . Q . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Q . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Q
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . Q . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . Q . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . Q . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Q . . . .
. Q . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . Q . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Q . .
. . . . . . . . . . . . . . . . . . . . . Q . . . . . . . . . . . . . . . . . . . . . . . .
. . Q . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . Q . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Q . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . Q . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . Q . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
143 Total iterations
----------------
```

The algorithm failed on the 64-Queens problem only once out of a hundred times, with an average of 118 iterations.